A SCALABLE AND TUNABLE ADAPTIVE RESOLUTION PARALLEL I/O FRAMEWORK

by

Sidharth Kumar

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computing

School of Computing

The University of Utah

May 2016

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Sidharth Kumar**

has been approved by the following supervisory committee members:

Valerio Pascucci	, Chair	**2/8/2016** Date Approved
Mary Hall	, Member	**12/15/2015** Date Approved
Martin Berzins	, Member	**12/15/2015** Date Approved
Feifei Li	, Member	**12/10/2015** Date Approved
Venkatram Vishwanath	, Member	**12/15/15** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

The increase in computational power of supercomputers is enabling complex scientific phenomena to be simulated at ever-increasing resolution and fidelity. With these simulations routinely producing large volumes of data, performing efficient I/O at this scale has become a very difficult task. Large-scale parallel writes are challenging due to the complex interdependencies between I/O middleware and hardware. Analytic-appropriate reads are traditionally hindered by bottlenecks in I/O access. Moreover, the two components of I/O, data generation from simulations *(writes)* and data exploration for analysis and visualization *(reads)*, have substantially different data access requirements. Parallel writes, performed on supercomputers, often deploy aggregation strategies to permit large-sized contiguous access. Analysis and visualization tasks, usually performed on computationally modest resources, require fast access to localized subsets or multiresolution representations of the data. This dissertation tackles the problem of parallel I/O while bridging the gap between large-scale writes and analytics-appropriate reads.

The focus of this work is to develop an end-to-end adaptive-resolution data movement framework that provides efficient I/O, while supporting the full spectrum of modern HPC hardware. This is achieved by developing technology for highly scalable and tunable parallel I/O, applicable to both traditional parallel data formats and multiresolution data formats, which are directly appropriate for analysis and visualization. To demonstrate the efficacy of the approach, a novel library (PIDX) is developed that is highly tunable and capable of adaptive-resolution parallel I/O to a multiresolution data format. Adaptive resolution storage and I/O, which allows subsets of a simulation to be accessed at varying spatial resolutions, can yield significant improvements to both the storage performance and I/O time. The library provides a set of parameters that controls the storage format and the nature of data aggregation across the network; further, a machine learning-based model is constructed that tunes these parameters for the maximum throughput. This work is empirically demonstrated by showing parallel I/O scaling up to 768K cores within a framework flexible enough to handle adaptive resolution I/O.

To my parents Ashwini and Susan,
my sisters Avantika and Polan,
and the Asha Devi clan.

CONTENTS

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation could not have been accomplished without the help of many whom I would like the thank. I would first like to thank my family, whose endless support made this work possible. I would like to thank my advisor and mentor, Valerio Pascucci, for his continued guidance and encouragement. I hope this dissertation is not the end, but the beginning of a long collaboration. I would also like to thank the other members of my committee, Venkat, Martin, Mary, and Feifei, for their feedback on this work. I would like to especially thank Venkat along with Phil Carns, both from Argonne National Laboratory, who were a great pillar of support during the earlier phase of my PhD.

I would also like to thank the many labmates/collaborators from the Data Analysis group along with my many friends in Salt Lake City: Shashank, Tom, Vimal, Jim, Hoa, Anand, Cameron, Duong, Shusen, Protonu (and many more). Finally, I would like to thank Argonne Leadership Computing Facility (ALCF), National Energy Research Scientific Computing Center (NERSC), and KAUST Supercomputing Laboratory (KSL) for providing me access to their supercomputig resources.

PART I

INTRODUCTION AND BACKGROUND

CHAPTER 1

MOTIVATION AND CONTRIBUTION

In this supercomputing era, with the continued surge of computing resources, scientists are simulating more and more complex phenomena. These simulations often generate enormously large datasets; for example, a typical Uintah [1] or an S3D [2] simulation dumps several hundreds of gigabytes of data every time-step. Dealing with large datasets presents several challenges, ranging from efficient parallel data writes to effective data exploration through analytics and visualization. Parallel I/O for writes from simulations is a challenge in part because of complex interdependencies between I/O middleware and hardware. Moreover, the two different tasks– data generation (writes) and data exploration (reads)- often have substantially different requirements. For instance, visualization and analysis mostly performed on computationally modest resources require fast access to localized subsets or multiresolution representations of the data. On the other hand, parallel writes, performed on supercomputers, often deploy aggregation strategies to permit large-sized contiguous access. Clearly the data access patterns for the two tasks are contradictory in nature, leading to a scenario in which optimization in one leads to a compromise in the other. The existing I/O solution for large-sized datasets faces these challenges; they both struggle to demonstrate scalability during parallel writes while also being inefficient during reads for data exploration, which leads to poor utilization and ineffective use of the available computation resources.

This dissertation addresses this problem by developing an end-to-end adaptive-resolution data movement framework that provides efficient I/O, while supporting the full spectrum of modern HPC hardware. This is achieved by developing technology for highly scalable and tunable parallel I/O, applicable both to traditional parallel data formats and multiresolution data formats directly appropriate for data analytics. To demonstrate the efficacy of the approach, a novel library (PIDX) is developed that is highly scalable and tunable and is capable of adaptive-resolution parallel I/O. With PIDX, the output of parallel writes can be efficiently used for analysis and visualization tasks.

It is often the case that the data layout optimized for parallel computing does not match with the layout optimized for network and storage access. In a situation like this, performing naive I/O leads to underutilization of the compute resources. The research done as part of this dissertation leads to the development of an efficient, flexible, and high-performing I/O software layer that intercepts and transforms unstructured application I/O into well-structured I/O suited to the underlying network and storage system. The I/O transformations also expose tunable parameters that can control the nature of flow of data across the network, all the way to the storage system. Using the customizable I/O transformations for a multiresolution, analysis-friendly data format made it possible for the first time to establish a fast and interactive read component (optimized for latency) in the otherwise write-heavy (optimized for bandwidth) I/O software stack.

Another big challenge in the field of parallel I/O is performance tuning. Realizing high I/O performance for a broad range of applications on all HPC platforms is difficult, in part because of complex interdependencies between I/O middleware and hardware. The parallel file system and I/O middleware layers offer optimization parameters that in theory can result in optimal I/O performance. Unfortunately, it is not easy to derive a set of optimized parameters, as it largely depends on the application, HPC platform, problem size, and concurrency. In order to optimally use HPC resources, an auto-tuning system can hide the complexity of the I/O software stack by automatically identifying parameters that accelerate I/O performance. Earlier work in auto-tuning I/O research has proposed analytical models, heuristic models, and trial-and-error approaches. The models can then be used to obtain optimal parameters. Unfortunately, all these methods have known limitations [3] and do not generalize well to a wide variety of settings. Modeling techniques based on machine learning overcome these system limitations and build a knowledge-based model that is independent of the specific hardware, underlying file system, or custom library used. Based on the flexibility and independence of a variety of constraints, machine learning techniques have achieved tremendous success in extracting complex relationships from the training data only. To this end, the PIDX I/O library was used as a use-case to build a machine learning model. The first step was to perform a detailed characterization study of the collective I/O algorithm, including both network aggregation and file system I/O. The goal was to understand how performance is affected by combinations of fixed input parameters (system and data characteristics) and tunable parameters, following which the behavior of the I/O library was modeled using machine learning techniques. The model has the ability to predict performance and identify optimal tuning parameters for a given

scenario. With approaches such as this, the efficiency of the performance model increases over time when more training data become available.

With simulations increasing in size and complexity, it is difficult for I/O and storage systems to keep pace with the increasing amount of data that scientists need to store. Increasing I/O time and storage costs lead to a curtailment in frequency of analysis and checkpoint dumps and also slow down postprocessing analysis tasks. One way to effectively tackle this problem of increasing I/O time and storage cost is to perform adaptive/sparse data dumps from simulations. Sparse I/O is performed by transforming the typical dense multidimensional array output of a simulation into a light-weight sparse representation that can then be written instead of storing the data in their entirety. Existing HPC I/O software such as pnetCDF [4], parallel HDF [5], and ADIOS [6] have supports only for dense, multidimensional arrays. This dissertation presents a novel storage and I/O methodology for performing sparse data dumps. In particular, two use-cases of sparse data dumps suited for uniform resolution simulations are presented: 1) region-of-interest (ROI) and 2) reduced resolution. ROI data dumps correspond to an I/O and storage methodology that allows regions/subsets of any simulation to be written at varying resolutions. With reduced-resolution dumps, a down-sampled version of the entire domain is written out. The two approaches lead to a significant improvement in both the I/O time as well as the storage footprint of the simulation.

1.1 End-to-end data movement pipeline

An end-to-end data movement framework is comprised of two important steps, data generation *(writes)* from the simulation and analysis and visualization *(reads)* for data exploration. For the first step, a parallel I/O library, such as Parallel netCDF [4], PIDX [7], or Parallel HDF5 [5], is used to *write* data from simulations running on compute nodes to the storage device of a HPC system (see path A of Figure 1.1). For the second process, data are *read* for analysis and visualization either by an analysis/visualization cluster (see path B of Figure 1.1) or by any commodity hardware (see path C of Figure 1.1). Parallel writes are typically optimized for bandwidth as opposed to reads, which are optimized more for latency. Simulation data can also be directly forwarded to an analysis/visualization cluster, skipping the storage altogether. This process is performed by I/O forwarding libraries such as Glean [8], which intercepts calls of I/O libraries to move data directly to the analysis/visualization cluster. Once the data are at the cluster, pertinent analysis and visualization tasks can be performed followed by writes back to the storage (see path D of Figure 1.1). In-situ analysis and visualization techniques [9] that involve minimal data

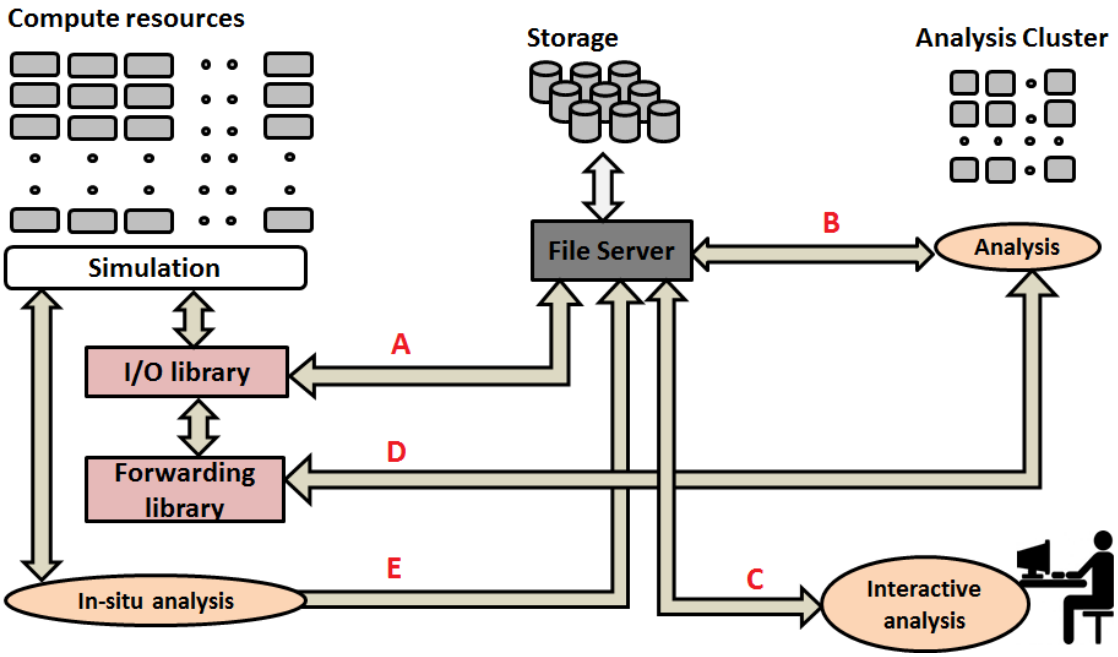


Figure 1.1: End-to-end data movement infrastructure for a simulation. Path A is for checkpoints, restarts, and analysis dumps; Paths B and C are for postprocessing analysis tasks; Path D shows I/O forwarding; and Path E shows in-situ analysis workflow.

movement are also becoming very prominent. With this methodology, instead of storing the simulation data in their entirety, only some condensed representation of the data (example, merge tree representation) is stored to the disk (see path E of Figure 1.1). A detailed description of the two aspects (writes and reads) of the framework follows.

1.1.1 Checkpoints and analysis dumps (writes)

HPC applications survive failures by saving their state in files called checkpoints on stable storage, usually a globally accessible parallel file system. When a fault occurs, the application rolls back to a previously saved checkpoint and restarts its execution. Checkpoints are usually very expensive and hence are performed at very low frequency. For example, a Uintah simulation typically performs a checkpoint dump at every thousandth time-step. While computation speed is increasing as described by Moores law, I/O subsystem speeds are not improving nearly as fast, so the gap between computation and I/O speeds continues to widen. Hence, it is very important to squeeze every bit of performance from the I/O system. Existing I/O solutions struggle to demonstrate scalability, especially as the number of cores in supercomputers touches the million mark. Besides checkpoint dumps, simulations also perform intermediary analytics and visualization dumps. These dumps are usually smaller in size and are performed at a frequency higher than checkpoint dumps. For

example, the Uintah simulation performs an analysis dump at every hundredth time-step. These dumps could ideally be sparse data dumps, where regions of simulations that are scientifically more important could be stored at higher resolution while the remaining regions could be stored at a smaller resolution. Currently used I/O solutions dump data in a very robust general purpose format that does not provide much leverage for analytics and visualization; they also do not provide the capability to perform sparse (adaptive resolution) data dumps.

1.1.2 Postsimulation work-flow (reads)

Massively parallel scientific simulations often generate large datasets that can range in size from terabytes to petabytes. Efficient, selective reading of the data is required for several reasons. For instance, in comprehensive, postprocess analytics, reading the full resolution data is often performed, but it is possibly only for a subset of the data or variables of interest under study. Moreover, depending on the algorithm used, even analysis that is conceptually performed at full resolution may be achieved without accessing all the data. In fact, fast processing of coarse approximations or smart use of complementary meta-data can allow skipping large regions that fail to yield any useful output. Another major reason for needing efficient, selective data reads is for user-directed interactive analysis and exploration. Unlike restarts and bulk postprocessing, interactive tasks often perform using lower-resolution data, possibly with reduced spatial extent. For example, coarse-resolution data can be used to compute an approximation of comprehensive analysis results. Furthermore, spatial extent can be restricted since only data within a visible region actually needs to be loaded. Commonly used parallel I/O libraries such as PnetCDF [4] and parallel HDF [5] have to some extent demonstrated scalable write performances. However, the data written by these libraries are very general purpose in nature. Such libraries do not have any concept of level-of-detail or resolution, making it inefficient to access the data. These general purpose data formats usually need to be postprocessed to some other analytics and visualization friendly format that is both computationally expensive as well as wasteful of precious storage space.

1.2 I/O transformations

The I/O stack is the software layer sitting between the application and the storage hardware. It provides data model support and I/O transformation for obtaining high performance on today's I/O systems. The I/O software stack is shown in Figure 1.2 (a). The data model maps application abstractions onto storage abstractions and provides data

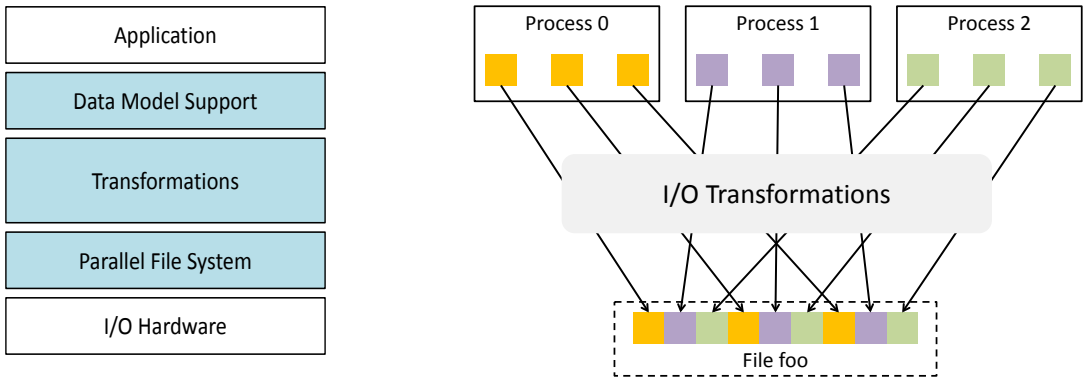


Figure 1.2: The I/O software stack and I/O transformations. (a) All layers of the I/O software stack. (b) I/O transformations: steps taken to efficiently map data from the application process to locations in the file.

portability. I/O transformations are the most integral component of the stack; they refer to a broad set of techniques that facilitate effective translation of data distributed across cores, residing in the simulation layout (row-order or column order) into a stream of bytes stored in files on the parallel file system (see Figure 1.2 (b)). After a simulation finishes performing computation and decides to move data to storage, it has to go through the external network that connects the compute nodes to the storage. I/O transformations ensure that both data movement across the network as well as data access to the file system are performed to extract maximum performance out of the network and the storage system. This dissertation presents the following four novel I/O transformations to facilitate high performance scalable I/O:

1. Data aggregation for subfiling: Subfiling allows storing of data to a tunable number of files, and data aggregation mitigates inefficiencies caused by small-sized access to the file system. This work presents a transformation incorporating the benefits of both data aggregation and subfiling. With this transformation, a subset of processes (aggregators) combines data from all processes into large contiguous buffers to write to a hierarchy of files, which reduces both the number of accesses to the parallel file system (aggregation) as well avoids lock contention of files (subfiling). In this transformation, the numbers of both files and aggregators are tunable parameters, granting a great degree of flexibility while adapting to the needs of a diverse variety of file systems. The efficacy of this transformation is demonstrated on a multiresolution data format.

2. Data restructuring: Data layout optimized for parallel computing does not always match with the layout that is optimized for I/O and storage. The restructuring transformation increases locality by aligning the application level layout to the storage level layout, thus extracting maximum performance. The efficacy of this transformation is demonstrated using a multiresolution data format. In accordance with the storage layout, the phase alters the distribution of data among processes while keeping it in its multidimensional application format. This transformation is performed using efficient, large, nearest-neighbor messages.

3. Domain partitioning: Domain partitioning alters the two-phase I/O algorithm in a way that reduces all global communication and synchronization during data aggregation. Communication is limited to local subgroups of processors by partitioning the global domain and writing the partitions as multiple independent datasets. The subgroups are isolated such that each file is written completely in parallel, resulting in a framework that is scalable up to the largest systems available today. An important strength of this approach is its simplicity, which makes it a great candidate for a practical implementation, especially for exascale I/O systems where synchronization will become a huge bottleneck.

4. Chunking: This is a step in which n $\times$ n $\times$ n blocks of spatially co-located elements are "chunked" together and then compressed. These compressed chunks become the new atomic elements taking the place of individual samples. This step is critical to obtain good compression while using block-based compression techniques.

The above four transformations have been implemented within the framework of the *Parallel IDX (PIDX)* I/O library [10, 7, 11, 12, 13]. PIDX, instead of writing traditional data formats (like netCDF [4]), translates and writes simulation data in a multiresolution format (see Figure 1.3) that is inherently more suitable for visualization and analysis. The data format dumped by PIDX is called the IDX format. The IDX format provides efficient, cache oblivious, and progressive access to large-scale scientific data by storing the data in a hierarchical Z (HZ) order [14], which increases the locality of data access for common queries, making it possible for scientists to interactively analyze and visualize data of the order of several terabytes [15]. With PIDX, the output of parallel writes can be efficiently used for analysis and visualization tasks. The I/O library hence plays a crucial role in meeting the demands of both large-scale simulations writes and analytics-appropriate reads. The other two commonly used I/O styles for writing simulation data are shared file I/O

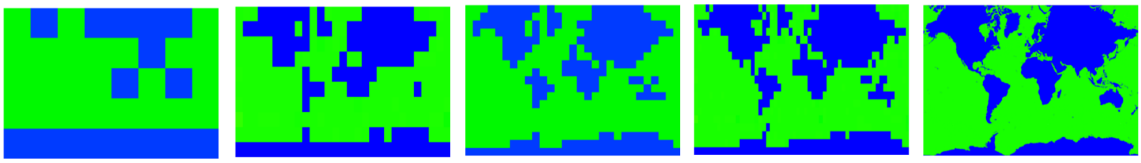


Figure 1.3: Coarse-to-fine resolution (HZ) levels of a two-dimensional IDX dataset.

and file-per-process I/O. With shared file I/O, all processes write to one single shared file, and in file-per-process I/O, each process writes out its data to independent files. Both these techniques fail to perform on write side, read side, or both. On some supercomputers, the file-per-process approach yields reasonable write performance up to medium scale, but fails badly when used for visualization purposes mainly because, on a relatively modest computational resource, a considerable amount of time goes into performing file I/O operations (open, seek, and close) for the large number of files, rendering analysis and visualization nonscalable. Other commonly used parallel shared-file I/O libraries are PnetCDF [4] and parallel HDF [5], both of which are implemented atop message passing interface (MPI) and MPI-IO [16] functionality and use collective I/O [17] for optimizations. They have been demonstrated to show scalable write performances only up to moderate scales, after which performance typically plateaus-out. These libraries, however, are very general purpose with no concept of level-of-detail or resolution and therefore provide little to no leverage in visualization and analysis. PIDX finds a middle ground between the two extreme I/O approaches, demonstrating both write and read performance. The PIDX I/O library is open-source and can be downloaded from the url: https://github.com/sci-visus/PIDX.

1.3 Modeling I/O performance

The performance of most parallel I/O libraries is influenced by characteristics of the target machine, characteristics of the data being accessed, and the value of tunable algorithmic parameters. HPC systems vary widely in terms of network topology, memory, and processors. They may also use different file systems and storage hardware that behave differently in response to I/O tuning parameters. For instance, the Intrepid [18] and Mira [19] supercomputers of the Argonne National Laboratory use a GPFS [20] filesystem, as opposed to the Lustre filesystem [21] used by supercomputers Hopper [22] and Edison [23] at the National Energy Research Scientific Computing Center (NERSC). Unfortunately, the right combination of tunable parameters is highly dependent on the application, HPC platform, and problem size/concurrency. Scientific application developers do not have the time or expertise to take on the substantial burden of identifying good parameters for each

problem configuration and so resort to using system defaults, a choice that frequently results in poor performance.

This dissertation tackles the problem of parameter tuning by performance characterization and modeling. The PIDX I/O library is used as a case-study, for it presents a range of parameters that can be used to tune performance on a diverse set of HPC machines. The first step is a detailed characterization study of the PIDX collective I/O algorithm, including both network aggregation and file system I/O. The goal is to understand how performance is affected by combinations of fixed input parameters (system and data characteristics) and tunable parameters. Then the behavior of the I/O library is modeled using machine learning techniques. Modeling techniques based on machine learning overcome these system limitations and build a knowledge-based model that is independent of the specific hardware, underlying file system, or custom library used.

The models are first validated on (training) datasets from low core count. Then they are used for throughput prediction on test datasets from the high core count regime. The goal is to show that such a model would be useful for approximately predicting the behavior of a system in higher core count scenarios where brute-force sensitivity studies would be both costly and resource intensive. The samples from the high core count regime are obtained by augmenting the regression model with a sampling technique. Consequently, the model is used in an adaptive framework, where its performance improves over multiple simulation time-steps, thus auto-tuning the system parameters to achieve higher performance over time.

1.4 Adaptive I/O

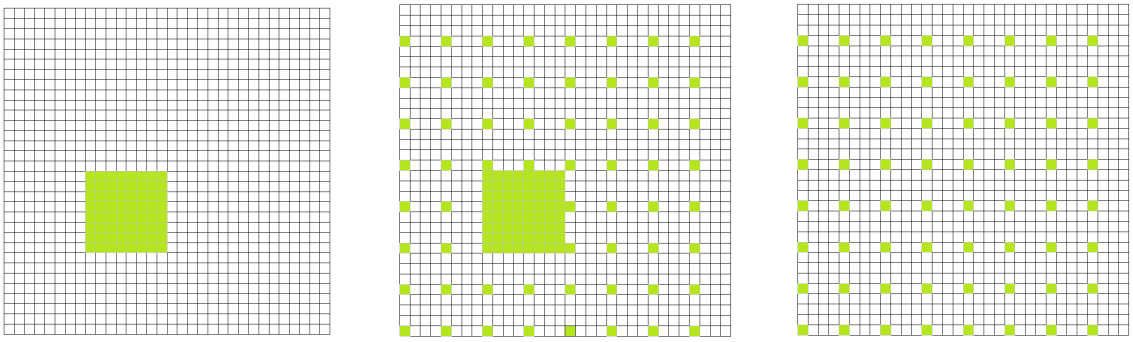
The increasing storage footprint for large-scale simulation is a big problem, which results in large I/O times for both data generation and for analysis and visualization. There is a compelling need for an adaptive I/O strategy that can reduce both the storage footprint as well as data access time. With adaptive storage strategy, data within different regions of a simulation could be adaptively saved at varying resolutions. This approach becomes even more effective as many simulations are heavily padded to avoid boundary artifacts, and often the phenomena of interest, e.g., ignition, extinction, etc., are confined to a comparatively small part of the domain. Therefore, writing distinct analysis or visualization snapshots that either store only the regions-of-interest or grade the saved resolution according to some importance measure could significantly reduce the overall data size without impacting the results. The primary technical challenge of adaptive storage strategy is to attain a high I/O throughput given the data might be distributed unevenly and potentially consist of many

small isolated regions, which can lead to fragmented accesses to both memory and disk. This problem is addressed by *layout-aware data aggregation*, an I/O transformation suited specifically for sparse data dumps. Following are the two use-cases of sparse data dumps presented specifically for uniform resolution simulations:

1. Region-of-interest: With this method of I/O, a subset of the domain is written at the native resolution while either discarding or down-sampling the remaining region of the simulation (see Figure 1.4 (a) and Figure 1.4 (b)). A challenge for these kinds of writes is to identify the regions-of-interest before they are written to disk. To this end, some in-situ analysis can be performed that helps in identifying the regions-of-interest. Both automated and sophisticated techniques such as merge or contour trees or techniques such as range thresholding can be used to identify the regions-of-interest.

2. Reduced resolution: With this method of I/O, a lowered resolution version of the data is written to the disk (see Figure 1.4 (c)). Data are down-sampled first and then written to the disk. The resolution can be controlled by the user. Generating simulation results at reduced resolution allows speed-up of both parallel writes as well as exploratory visualization.

There is also currently a marked trend of simulations moving towards adaptive resolution techniques, e.g., Adaptive Mesh Refinement (AMR) [1], to better manage multiple scales and couple detailed dynamics with large-scale behaviors. Most current high-end I/O frameworks [4, 5] are optimized for uniform grids, and, in fact, adaptive resolution grids are often simply represented and written as a collection of uniform grids at different resolutions. Such representations lead to fragmented, inefficient I/O. Furthermore, for convenience, many approaches unnecessarily replicate data on multiple levels, increasing the data footprint and decreasing I/O performance. In addition to addressing adaptive resolution dumps for uniform resolution simulations, this dissertation also provides algorithms for writing and storing adaptive resolution data for adaptive mesh refined (AMR) simulations. In the proposed approach, all AMR grids are merged into a single sparsely populated grid with data being stored adaptively. This approach to storing data has several advantages. It leads to faster data access, avoids any unnecessary data replication, and also provides both spatial and hierarchical locality. These advantages make it more suitable for analysis and visualization tasks, thus improving the overall productivity of the application developer. Both sparse data I/O and AMR I/O are implemented within the framework of PIDX.



(a) (b) (c)

Figure 1.4: Example of spatial layouts of sparse data. (a) and (b) Examples of spatial layouts of an ROI grid. (c) An example of spatial layout of a reduced-resolution grid.

1.5 Dissertation contributions

The research done as part of this dissertation has led to development of an end-to-end adaptive-resolution data movement framework that achieves fast I/O for adaptive simulations dumps/restarts, analytics, and visualization, without trading write (large-scale writes) efficiency for read (analytics-appropriate reads) efficiency, while supporting the spectrum of modern HPC hardware. To summarize, this dissertation has contributed to the following three areas of research :

1. **Scalable I/O transformations** (PART II: Chapters 4, 5, and 6)

   This research develops four I/O transformations (restructuring [11], aggregation [7], domain partitioning, and chunking) to facilitate high performance scalable I/O. Restructuring and aggregation, respectively, optimize memory and storage access, two critical factors in achieving scalable I/O. Domain partitioning helps mitigate synchronization overheads, whereas chunking enables block-based compression. All these transformations facilitate scalable movement of data from the compute resources to the storage and can be used to improve the performance of existing and emerging I/O libraries. In this research, these transformations are used to develop an I/O library PIDX [10, 7, 11, 12, 13] that writes data directly in visualization appropriate data formats, thus bridging the gap between analysis appropriate reads and large-scale writes.

2. **I/O performance tuning** (PART III: Chapter 7)

   Performance tuning is achieved by performance characterization and modeling [12]. In the first step, a detailed characterization study of the PIDX collective I/O algorithm is performed, following which performance models are built using regression analysis

on datasets collected during the characterization study. The regression models are then used to predict I/O performance and identify optimal tuning parameters.

3. **Adaptive-resolution parallel I/O** (PART IV: Chapters 8 and 9)
   This research [24] develops algorithms to facilitate adaptive-resolution I/O. More specifically, algorithms have been developed for sparse I/O catering to the needs of uniform-resolution simulations, and I/O for adaptive-resolution simulations (example AMR).

1.6 Dissertation structure

The remaining chapters in this dissertation are outlined below:

- **Chapter 2:** covers the relevant background work in the field of parallel I/O.
- **Chapter 3:** discusses the multiresolution cache-oblivious IDX data format [14]. This chapter explores the features that makes the format more suitable for analysis and visualization. The PIDX library developed as part of this research writes data in the IDX format.
- **Chapter 4:** discusses data restructuring [11] and aggregation [7], key transformations for scaling parallel I/O. Data aggregation is used to facilitate efficient disk access and restructuring is used to facilitate efficient memory access. This chapter also introduces the PIDX [10, 7, 11, 12, 13] I/O library.
- **Chapter 5:** discusses the domain partitioning transformation used to reduce data movement and synchronization costs. This chapter also discusses fixed-bit block-based compression, which enables data access at both varying numerical precision and spatial resolution.
- **Chapter 6:** introduces transformations designed for parallel reads (used for restarts). This chapter also discusses two other kinds of reads, view-dependent reads and parallel lowered-resolution reads, both of which can be used for real-time data exploration.
- **Chapter 7:** presents I/O performance tuning [12] using performance characterization and modeling.
- **Chapter 8:** introduces algorithms [24] for sparse I/O for uniform resolution simulations. In particular, two use-cases (region-of-interest I/O and reduced-resolution I/O) of sparse I/O are discussed.

- **Chapter 9:** introduces algorithms [24] designed specifically for I/O of adaptive resolution simulations (exmple AMR).
- **Chapter 10:** discusses potential topics for future work and concludes the dissertation.

CHAPTER 2

RELATED WORK

This chapter outlines the previous work done in the area of I/O systems for large-sized datasets. The first section of this chapter discusses the current state of supercomputing resources, with an emphasis on the I/O-related hardware. The second section of the chapter discusses the state-of-art practice in I/O systems.

2.1 Supercomputer infrastructure

Supercomputers are identified by their immense computational capability stemming from their thousands and millions of parallel processing units. As an example, leadership class machines such as the IBM Blue Gene/Q supercomputer at the Argonne National Laboratory [25] and the Cray XT system at the Oak Ridge National Laboratory [26], respectively, consist of 768K and 225K processing units and have peak performances of 10 and 1.75 petaflops. Supercomputers often differ from each other with respect to architecture, interconnect network, operating system environments, file system, and many other parameters. A common concept is that of processing units (cores) and nodes. Processors are grouped into nodes, which then are linked together with the interconnect network. Compute nodes perform the simulation and I/O nodes provide I/O functionality. Compute nodes generally do not have access to the file system, so data are routed through I/O nodes while interfacing with file system software.

2.1.1 Parallel file system

The file system is the direct interface to the storage system of the supercomputer. Being able to use and access the file system optimally is key to I/O performance. File systems often have several tunable parameters, such as stripe count and size, that need to be set optimally. In the next subsection, we explore the two most commonly used file-systems, Lustre [21] and GPFS [20], in more detail. Both Lustre and GPFS file systems are scalable and are part of multiple computer clusters with thousands of client nodes and several petabytes (PB) of storage, which makes these file systems a popular choice for supercomputing data centers, including those in industries such as meteorology, oil, and gas exploration.

Lustre [21] is an open-source, high-performance parallel file system that is currently used in over 60% of the top 100 supercomputers in the world. It is designed for scalability and is capable of handling extremely large volumes of data and files with high availibility and coordination of both data and metadata. For example, the Spider supercomputer at Oak Ridge National Laboratories has 10.7 petabytes of disk space and moves data at 240 GB/second. The architecture is based on storage of distributed objects. Lustre delegates block storage management to its back-end servers and eliminates significant scaling and performance issues associated with the consistent management of distributed block storage metadata.

GPFS (General Purpose File System [20]) is a parallel file system for high-performance computing and data-intensive applications produced by IBM. It is based on a shared storage model. It automatically distributes and manages files while providing a single consistent view of the file system to every node in the cluster. The filesystem enables efficient data sharing between nodes within and across clusters using standard file system APIs and standard POSIX semantics.

2.1.2 Network topology

Network topology describes how compute nodes in a supercomputer are connected and plays an important role in data movement. Compute nodes are designed to optimize internode communications for simulation. Data movement for I/O typically is designed around the network topology. The Mira supercomputer at Argonne National Laboratories is an IBM Blue Gene/Q system [27] that uses a 5D torus network for I/O and internode communication. The Edison machine employs the "Dragonfly" topology [23] for the interconnection network. This topology is a group of interconnected local routers connected to other similar router groups by high-speed global links. The groups are arranged such that data transfer from one group to another requires only one route through a global link. Processor counts in supercomputers continue to grow, and as a result, it becomes more essential to exploit the structure and topology of the interconnect network.

2.2 State-of-the art practices in I/O systems

There are three considerations in deciding on a file format for data storage. The first is I/O speed. Given a supercomputer configuration, the first consideration is how costly it is to transfer data from compute nodes to the storage medium. The more closely a data format matches the data in memory, the more efficient the I/O will be. If the format does not match up well, then the data must be reformatted in memory, and the performance of

this operation is subject to the network infrastructure. The second consideration is what the data will be used for. If the data are to be used for restarts, one format may be suitable, but if the data are to be visualized, an entirely different format may be the best. The third consideration is size. The smallest size possible for uncompressed data is the size of the data itself. However, this size may grow with metadata, replicated data, and unused data. Metadata is required in some form in all formats. It may simply describe the dimensions of the data, or more complex information such as data hierarchy. Depending on the format, metadata size may be negligible, or it may rival the size of the data itself. Some file formats have replicated data. For example, a pyramidal scheme of a rectilinear grid may store pixel values at every level of the pyramid. Similarly, unused data, such as filler pixels in a data blocking scheme, can increase the storage footprint.

The number of actual files is also a consideration. In one approach, commonly called file-per-process I/O or $N - N$ output, each processor writes to its own file. N is the number of processors. This approach limits the choice of file format. That is, many formats require global knowledge of the data, such as in the case of storing data hierarchically. If each processor writes its own file, then global knowledge of the data is not available and sophisticated formats cannot be used. Further, I/O nodes must write to a large number of files, creating a bottleneck. Finally, the number of files produced may also put a burden on downstream visualization and analysis software. Nevertheless, $N - N$ strategies are popular due to their simplicity. $N - 1$ approaches route all data to a single file. These approaches are more complex due to the required internode communication, but there is much more flexibility to optimize. A straightforward optimization is that of larger block writes to disk. That is, large chunks of data can be written at a time, making writes more efficient. Optimization in file structure is also possible. If the data are to be used for visualization, a hierarchical stream-optimized format such as IDX [14] may be used. Subfiling approaches ($N-M$) provide ultimate flexibility, where the number of files is tuned for maximum performance.

2.2.1 Formats and libraries

The most popular file formats in large-scale simulation are of the $N-1$ variety, of which PnetCDF and HDF5 are the most common. Formats generally have an accompanying I/O library, easing use of a particular format. This section discusses both formats and libraries.

MPI-IO is a standard, portable interface for parallel file I/O that was defined as part of the MPI-2 (Message Passing Interface) Standard in 1997. It can be used either directly by applications programmers or by writers of high-level libraries as an interface for portable,

high-performance I/O in parallel programs. MPI-IO is an interface that sits above a parallel file system and below an application or high-level I/O library. Here it is often referred to as *middleware* for parallel I/O. MPI-IO is intended as an interface for multiple processes of a parallel program that is writing/reading parts of a single common file. MPI-IO can be used for both independent I/O, where all processes perform write operations independent of each other, as well as in collective I/O mode, where processes coordinate among each other and a few processes end up writing all the data.

HDF5 [5], short for "Hierarchial Data Format, version 5," is designed at three levels: data model, file format, and I/O library. The data model consists of abstract classes such as files, groups, datasets, and datatypes, which are instantiated in the form of a file format. The I/O library provides applications with an object-oriented programming interface that is powerful, flexible, and high performing. The data model allows storage of diverse data types, expressed in a customizable, hierarchical organization. The I/O library extracts performance by leveraging MPI-IO collective I/O operations for data aggregation. Owing to the very customizable nature of the format, HDF5 allows users to optimize their data type, making performance tuning partially the responsibility of the user.

PnetCDF [4] is another popular high-level library with similar functionality to HDF5 but in an file format that is compatible with serial NetCDF from Unidata. PnetCDF is a single shared file approach ($N - 1$) and is optimized for dense, regular datasets. It is inefficient for hierarchical or region-of-interest (ROI) data in both performance and storage, and so is used only in rectilinear simulation environments.

PLFS (parallel log-structured file system) is an I/O library that remaps an applications preferred data layout into one that is optimized for the underlying file system [28]. Testing on Panasas ActiveScale Storage System and IBMs General Parallel File System at Los Alamos National Lab and on Lustre at Pittsburgh Supercomputer Center, PLFS, has shown that this layer of indirection and reorganization can reduce the checkpoint time by up to several orders of magnitude for several important benchmarks and real applications.

PIDX I/O library [11, 24] enables concurrent writes from multiple cores into the IDX format, a cache-oblivious multiresolution data format inherently suitable for fast analytics and visualization. PIDX is an $N - M$ approach, contributing to better performance. The number of files to generate can be adjusted based on the file system. This approach extracts more performance out of parallel file systems and is customizable to specific file systems. Further, PIDX utilizes a customized aggregation phase, leveraging concurrency and leading to more optimized file access patterns. PIDX is naturally suited to multiresolution AMR datasets, region-of-interest (ROI) storage of rectilinear grids, and visualization and analy-

sis. IDX does not need to store metadata associated with AMR levels or adaptive ROI; hierarchical and spatial layout characteristics are implicit. Note that the PIDX I/O library is developed as a result of the research performed during this dissertation.

ADIOS [6] is another popular library used to manage parallel I/O for scientific applications. One of the key features of ADIOS is that it decouples the description of the data along with transforms to be applied to that data from the application itself. ADIOS supports a variety of back-end formats and plug-ins that can be selected at run time.

GLEAN [8, 29], developed at Argonne National Laboratoy, provides a topology-aware mechanism for improved data movement, compression, subfiling, and staging for I/O acceleration. It also provides interfaces for co-analysis and in-situ analysis, requiring little or no modification to the existing application code base.

Adios and Glean are high-level I/O libraries that can internally use low-level libraries such as parallel netCDF and parallel HDF. Most of these I/O libraries have been optimized for parallel I/O writes from simulations. The data generated from these formats continue to suffer in their ability to support effective visualization.

2.2.2 Performance characterization and modeling

This section discusses previous work related to I/O capability, performance, and scalability of leading HPC systems. A body of work has focused on the characterization study of Jaguar, a Cray XT3/XT4 machine, at Oak Ridge National Laboratory. For example, Yu et al. [30] studied the scalability for each level of the storage hierarchy (up to 8,192 processes). Fahey et al. [31] characterized I/O performance for a file with constant sizes. Both leveraged insights from their studies to tune and optimize I/O performance for scientific applications but also presented challenges towards scaling I/O for larger core counts. More recently, Xie et al. [32] characterized bottlenecks of the multistage I/O pipeline of Jaguar on the Lustre filesystem. Their study used IOR for benchmarking and talked about the straggler phenomenon where I/O bandwidth becomes limited due to a few slow storage targets (stragglers). Other existing I/O characterization works for Cray machines include [33] and [34].

Similar studies by Lang et al. [35] on the I/O characterization for IBM Blue Gene/P system have highlighted I/O challenges faced by IBM Blue Gene/P system Intrepid at the Argonne National Laboratory. The authors reports the capacity of each I/O stage, studying individual components and building up to systemwide application I/O simulations. Another work [8] on the Blue Gene/P supercomputer presented topology-aware strategies for choosing aggregators for two-phase I/O. Focusing mainly on parallel file-systems, a

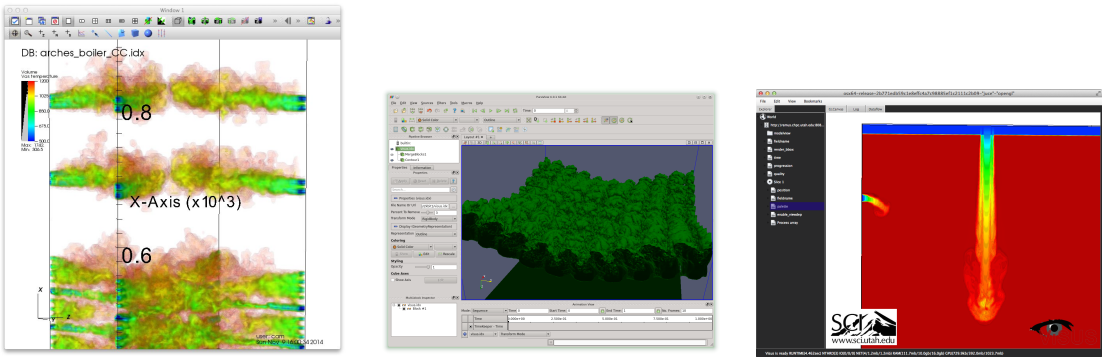
detailed study comparing performances of GPFS, Lustre, and PVFS was done by Oberg et al. [36].

Performance modeling of large-scale parallel systems has been addressed in Lublin et al. [37] where the authors modeled workloads to characterize supercomputers and claimed this approach to be better than trace-based modeling. Following [3], most of the existing work can be categorized as analytical models, heuristic models, and trial-and-error methods. Analytical models, proposed in [38, 39], are usually difficult to construct owing to the complexity of modern day multicore processors. An alternative line of work [40] developed a heuristic performance model based on offline benchmarking. Heuristic models, a popular approach [41], are usually too closely tied to the underlying system and hence do not generalize beyond the target hardware and application. Current machines are too complex, and such heuristic performance modeling is tedious, difficult, and error-prone in such scenarios. Recently, numerous works have focused on using machine learning for auto-tuning and performance studies. Machine learning models were used [3] to estimate performance degradation on multicore processors. A neural network-based optimization framework that can dynamically select and control the number of aggregators was proposed in [42]. A genetic algorithm-based auto-tuning framework was proposed [43] for the parallel I/O library HDF5. Shan et al. [44] use an IOR synthetic benchmark to parameterize I/O workload behavior and predict its performance on different HPC systems. In the fields of modeling storage, Randy et al. [45] discuss analytic performance models for disk arrays. Other relevant research related to performance modeling of I/O workloads includes [46], [47], [48], and [49].

2.2.3 Postprocessing workflows

How data are stored has a direct impact on how it is visualized and analyzed. The vast majority of simulations undergo significant analysis after completion. Figure 2.1 shows examples of visualizations. Three popular visualization packages are briefly discussed in the next section.

VisIt and ParaView ([50], [51]) are popular distributed parallel visualization and analysis applications. They are typically executed in parallel, coordinating visualization and analysis tasks for massive simulation data. The data are typically loaded at full resolution, requiring large amounts of system memory. Both packages utilize a plugin-based architecture, so many formats are supported by both. They are open source and platform independent and have been deployed on various supercomputers as well as on desktop operating systems such as Windows, Mac OS X, and Linux.



(a) (b) (c)

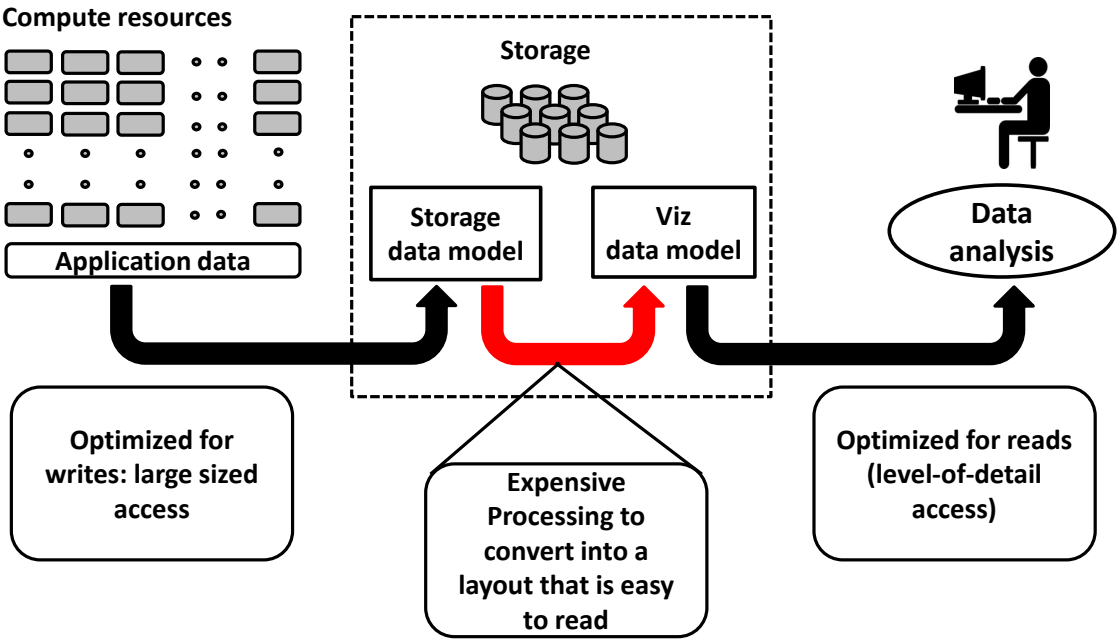
Figure 2.1: Snapshots of commonly used visualization tools– (a) Visit, (b) Paraview, and (c) Visus.

ViSUS[52, 53] is designed for streaming of massive data and uses hierarchical data representation technology, facilitating online visualization and analysis. Its design allows interactive exploration of massive datasets on commodity hardware, including desktops, laptops, and hand-held devices. Rather than running in a distributed environment, ViSUS supports thread-parallel operation. The IDX streaming data format is supported natively.

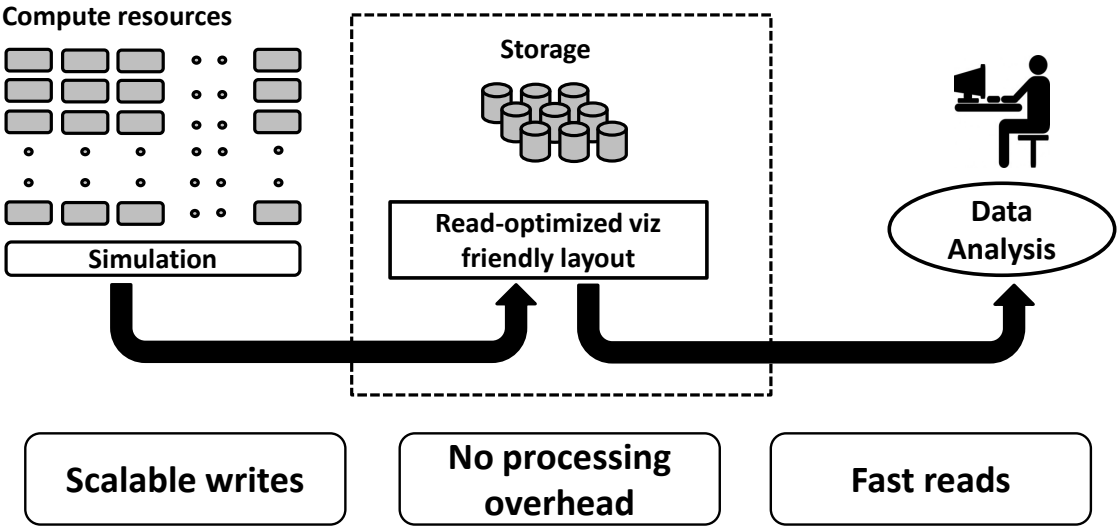
CHAPTER 3

FAST MULTIRESOLUTION DATA ACCESS

Dealing with large datasets presents several challenges, ranging from efficient parallel data writes to effective data exploration through analytics and visualization. The two different tasks– data generation (writes) and data exploration (reads)– have substantially different requirements. For instance, visualization and analysis mostly performed on computationally modest resources require fast access to localized subsets or multiresolution representations of the data. On the other hand, parallel writes, performed on supercomputers, often deploy aggregation strategies to permit large-sized contiguous access. Unfortunately, existing I/O solutions cater only to I/O needs on the write side, i.e., simulation dumps. Commonly used I/O systems such as parallel netCDF [4], HDF5 [5] and ADIOS [6] write data in the application layout, which typically is row or column major. These data typically need to be converted to a format that is more suitable for analysis and visualization. Traditionally, this conversion from storage model (data written from simulation) to visualization model (data read for exploration) is performed as a processing step right after simulation writes. This data flow is shown in Figure 3.1 (a). This processing step is both compute and storage intensive, and it also wastes precious user time. One method of addressing this problem is to reorganize data on the fly while it is been written from a simulation to a format that is more suitable for analysis and visualization. This approach gets rid of the processing step, as data written from simulations can be directly used for analysis and visualization. This data flow is shown in Figure 3.1 (b). The I/O library PIDX [10, 7, 11, 12, 13] has been developed to write data in the IDX format, which is both cache-oblivious and multiresolution, making it inherently suitable for analysis and visualization. Chapters in PART II detail the I/O transformations that enables PIDX to write data directly in the IDX format. This chapter talks about the format in detail.



(a)



(b)

Figure 3.1: An end-to-end data movement framework comprises of two components– dumps from simulations (writes) and analysis and visualization (reads) for scientific exploration. Both these components are shown in the figure using black arrows. (a) Traditional data movement framework involves an expensive processing step that converts data written from simulations to a format suitable for exploration (analysis and visualization). (b) Data movement framework that writes data directly in a format that is suitable for analysis and visualization, and hence, completely getting rid of the expensive processing phase.

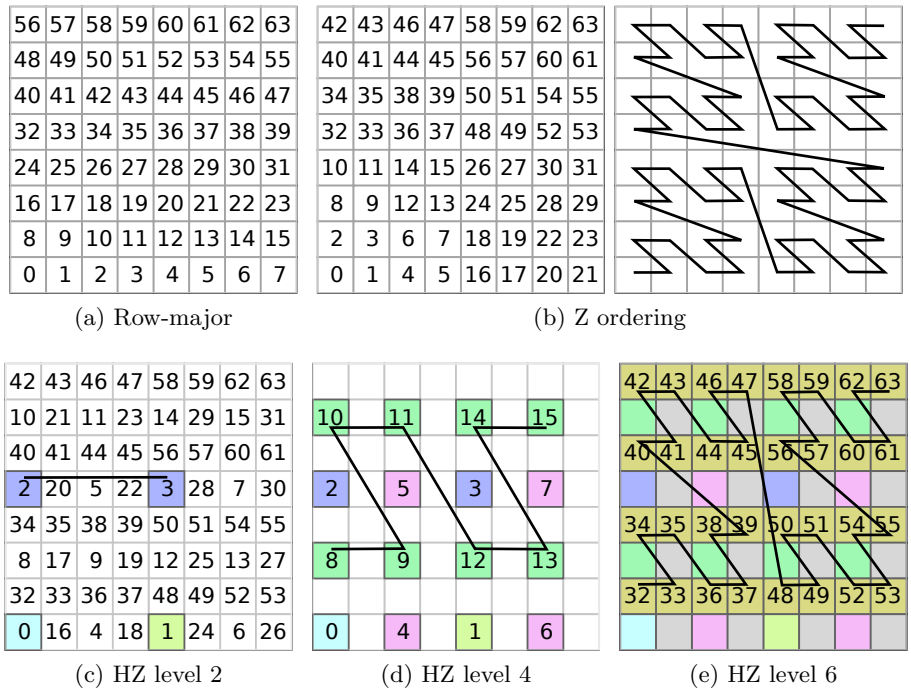
3.1 Background

Multiresolution data formats are very suitable for visualization for their ability to efficiently support adaptive level-of-detail algorithms. Previous work in multiresolution data formats for parallel processing environments includes that of Chiueh and Katz in developing a multiresolution video format for parallel disk arrays [54]. They leveraged Gaussian and Laplacian Pyramid transforms that were tailored to the underlying storage architecture to improve throughput. Their work focused on read-only workloads as opposed to write-heavy workloads. Chaoli, Jinzhu, and Han have presented work in parallel visualization [55] of multiresolution data. Their algorithm involves conversion of raw data to a multiresolution wavelet tree and focuses more on parallel visualization rather than a generic data format. Many researchers have proposed various techniques for multiresolution encoding and rendering of large-scale volumes [56]. Fewer studies were focused on designing parallel algorithms for generating data itself in a more interactive usable format. More recently, DynaM data representation supports convolution-based multiresolution data representation [57]. The work of Ahrens et al. enables multiresolution visualization by sampling existing data, and writes the multiresolution levels and meta-data to disk as independent files while keeping the full-resolution file intact [58].

3.2 IDX data format

Data can be encoded into a single-dimensional array for disk storage in a variety of ways (see Figure 3.2). A simple approach is to use row- or column-major ordering, but spatial locality is reasonable along only one dimension (the x-axis in the figure). Z-ordering (also known as Morton) [59] shows better spatial locality. The Z-index is efficient to compute – simply interleave the bits of the Cartesian coordinates. Hierarchical Z-ordering (HZ) extends Z-ordering by introducing hierarchy. The HZ-index is also efficient to compute using bit operations as described in [60]. The IDX data format uses HZ -indexing while storing data.

IDX is a multiresolution file format that enables fast and efficient access to large-scale scientific data. The format provides efficient, cache oblivious, progressive access to data by utilizing hierarchical Z-order for storage [14]. The IDX storage format uses HZ-ordering, which shows good locality both spatially and hierarchically. Conceptually, the scheme acts as a storage pyramid with each level of resolution laid out in Z-order. However, unlike traditional pyramids, IDX avoids replicating samples, which not only reduces the storage but also organizes the data hierarchically.



(a) Row-major (b) Z ordering

(c) HZ level 2 (d) HZ level 4 (e) HZ level 6

Figure 3.2: Different index orderings. (b) Row-major ordering has poor spatial locality. (a) Z ordering shows good spatial locality but has no concept of hierarchy or resolution adaptivity. (c)-(e) HZ ordering has both spatial and hierarchical locality. For an example of hierarchical locality, note that obtaining a $1/2^2$ resolution version of the grid requires a single disk read of elements 0-15 (best seen in (d)).

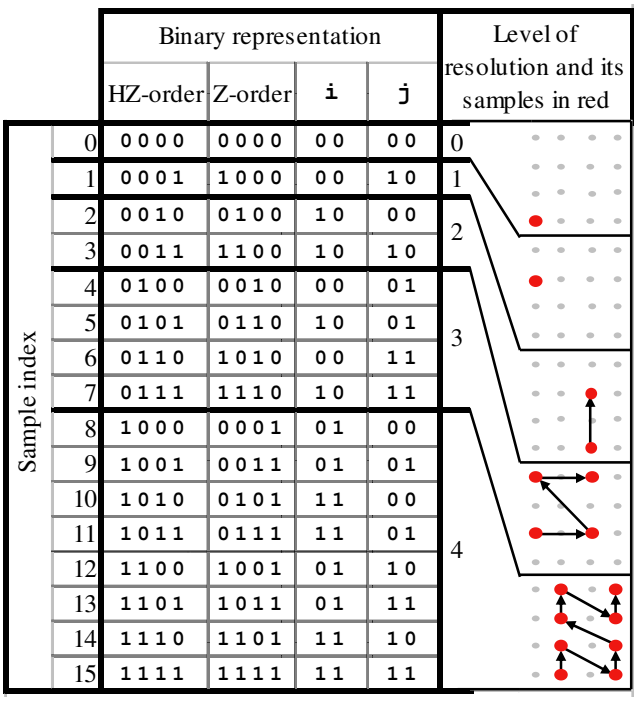
The IDX format provides an adaptive level-of-detail access of data that can be efficiently used in a data streaming environment. Traversing data hierarchically from the coarse to the fine resolutions and progressively updating output data structures derived from this data can provide a framework that allows for real-time access of the large-scale simulation data. Many of the parameters for interaction, such as display viewpoint, are determined by users at run time, and therefore, precomputing these levels of details optimized for specific queries is infeasible. Therefore, to maintain efficiency, a storage data layout must satisfy two general requirements: 1) if the input hierarchy is traversed in coarse to fine order, data in the same level of resolution should be accessed at the same time, and 2) within each level of resolution, the regions in spatial proximity are stored in proximity in memory. The IDX format meets both these requirements with its hierarchical Z-order data layout.

3.3 Z- and HZ-order background

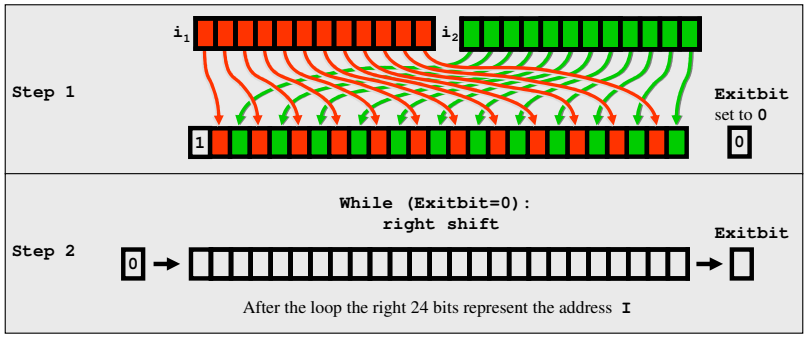
The data access routine of our system achieves high performance on data by utilizing a hierarchical variant of a standard Z-order (Lebesgue) space filling curve to lay out the application-layout data in one-dimensional memory. In the two-dimensional case, the Z-

order curve can be defined recursively by a Z-shape whose vertices are replaced by Z-shapes half its size (see Figure 3.3 (a)). Given the binary row-major index of a pixel ($i_n \ldots i_1\ i_0$, $j_n \ldots j_1\ j_0$), the corresponding Z-order index I is computed by interleaving the indices $I = j_n i_n \ldots j_1 i_1\ j_0 i_0$ (see Figure 3.3 (b) step 1).

Z-order exhibits good locality in all dimensions, but it does so only at full resolution and does not support hierarchical access. Instead, IDX format uses the hierarchical variant, called HZ-order, proposed by Pascucci and Frank [14]. This new index changes the standard



(a)



(b)

Figure 3.3: Example to convert row-order index to hierarchical Z-order index. (a) Levels of the hierarchical Z-order for a 4×4 array. The samples on each level remain ordered by the standard Z-order. (b) Address transformation from row-major index (i; j) to Z-order index I (Step 1) and then to hierarchical Z-order index (Step 2).

Z-order to be organized by levels corresponding to a subsampling binary tree, in which each level doubles the number of points in one dimension (see Figure 3.3 (a)). This pixel order is computed by adding a second step to the index conversion. To compute an HZ-order index I, the binary representation of a given Z-order index I is shifted to the right until the first 1-bit exits. During the first shift, a 1-bit is added to the left and 0-bits are added in all following shifts (see Figure 3.3 (b)). This conversion could have a potentially very simple and efficient hardware implementation. The software C++ version can be implemented as follows:

```
inline adhocindex remap(register adhocindex i){
  i |= last_bit_mask;  // set leftmost one
  i /= i&-i;           // remove trailing  zeros
  return (i>>1);       // remove rightmost one
}
```

3.4 Block-based storage

At storage within an IDX format, the HZ-ordered data samples are grouped in blocks of a constant size. A preset number of blocks is written into each binary file. A metadata *.idx* file contains all the required associated information of the IDX file. Any IDX file contains the bounding box *(box)*, number of elements per block *(elements_per_block)*, number of blocks per binary file *(blocks_per_file)*, the bitmask, and the filename template. Table 3.1, for example, shows block, HZ level as well as the file layout for an IDX file of *box* (0, 0, 0 : 16, 16, 16), 256 *elements_per_block*, and 2 *blocks_per_file*.

The HZ-ordering and corresponding distribution of data into different levels of resolution significantly reduces lag when zooming or panning a large-scale dataset. As can be seen in Table 3.1, block 1 contains data up to HZ level 8. In general, for each IDX file, the first block contains data up to level $\log_2(elements\_per\_block)$. Data in the first block span the entire volume in the lowest resolution. As the blocks are accessed in linearly increasing order, the resolution level starts to increase and spans a smaller part of the volume. A data query to an IDX dataset requires first checking the metadata to find the intersection of the queried data with the data blocks. Data can then be retrieved using lower-resolution data from initial blocks or higher-resolution data from later blocks. This scheme of access ensures an interactive and progressive data access.

Table 3.1: Block and file distribution of each HZ level in a 16^3 IDX dataset using 256 elements per block and 2 blocks per file. Levels 10 through 12 span multiple blocks.

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

PART II

I/O TRANSFORMATIONS

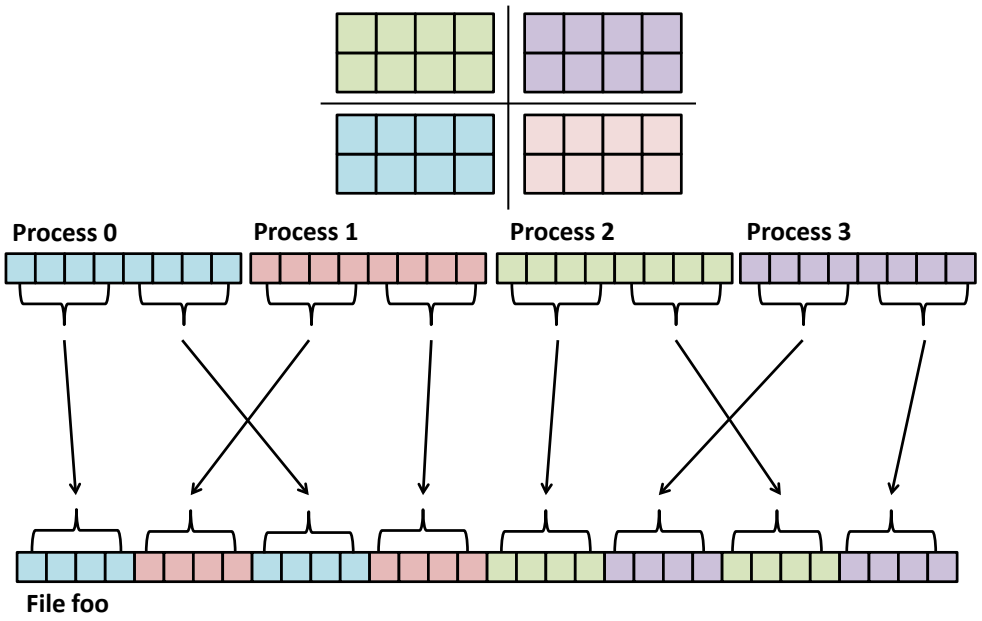
CHAPTER 4

RESTRUCTURING AND AGGREGATION FOR SCALABLE I/O

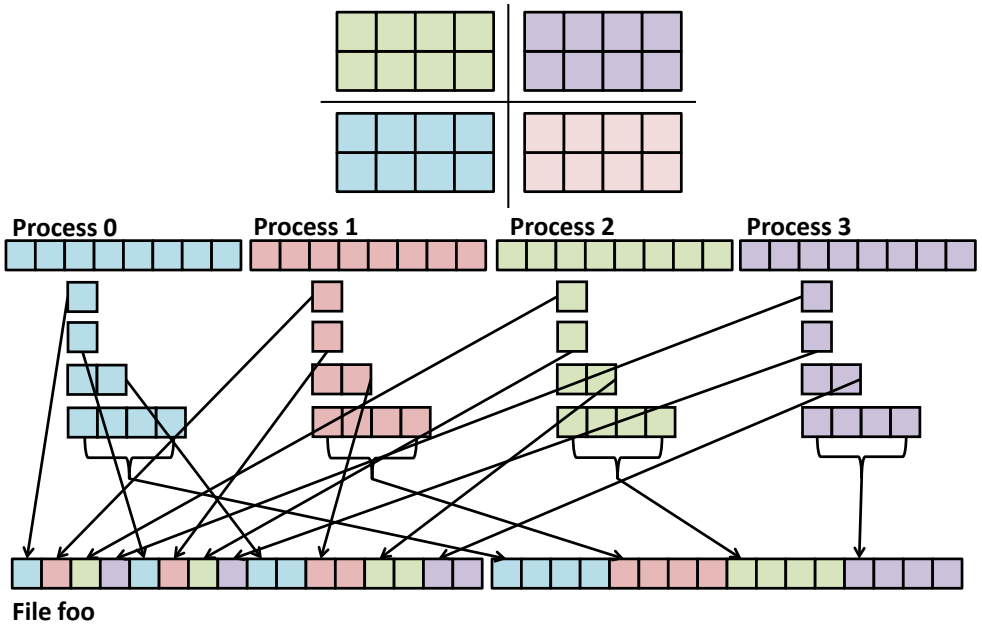
Parallel I/O can naively be performed with each process writing its own data directly to the proper location of a file. This kind of I/O is typically inefficient and does not scale due to the discontinuous access of sparse buffers within memory and the large number of small-sized accesses to the file system. This chapter details two I/O transformations– *data aggregation* and *data restructuring*– that address these issues. Section 4.1 discusses data aggregation, a transformation that reduces both the number of accesses to the parallel file system (aggregation) and avoids lock contention of files (subfiling). Section 4.2 discusses data restructuring, a transformation that increases data locality by aligning the application level layout to the storage level layout. Both these transformations are demonstrated within the framework of the PIDX I/O library, details of which are provided in Section 4.3 of this chapter.

4.1 Subfiling and data aggregation

Translation of application (simulation) layout into the data layout of an actual file typically leads to a large number of small-sized data accesses. A small-sized write access pattern is known to scale poorly, especially on leadership-class storage systems [61]. An example of this access pattern for two file formats is shown in Figure 4.1. A solution to this type of problem is to aggregate data before writing by using a two-phase I/O strategy [17]. In an MPI environment, this can usually be accomplished by simply issuing MPI-IO collective operations [16]. However, MPI-IO allows collective writes to only one file at a time within a given communicator. However, single-shared file I/O does not scale well on all leadership-class machines. The number of output files for an application has a significant effect on I/O performance. Existing I/O solutions often do not allow users to select the number of output data files. Instead, they often provide only two extreme options: one shared file or one file per process. The one-file-per-process strategy scales only until the large number of files overwhelms the file system with a high metadata operation cost, resulting in a major



(a)



(b)

Figure 4.1: Translation of application data from row-order layout into a file stored in (a) row-order layout and (b) multiresolution viz-friendly data layout (IDX file). (b) Is also an example of independent I/O done by one-phase implemetation (discussed in Section 4.1.2). Every process has a chunk of data in the application layout (row-order), which then is reorganized into a hierarchy of buffers in accordance with different levels of resolution. Every process then writes data for each resolution level in turn to the file (IDX dataset) using independent MPI-I/O write operations.

performance drawback. Moreover, downstream analysis tools would be forced to continue handling a large number of files with deterioration of their data access potential. On the other end, the single shared file mode can potentially create lock conflicts at the file system level, limiting the performance scalability. As a compromise, the third mode is subfiling, which allows for a tunable number of files intended to balance between the first two modes. Hence, in order to allow data aggregation to a tunable number of files, an aggregation algorithm is designed. This section details the aggregation mechanism where a subset of processes is responsible for combining data from all processes into large contiguous buffers before writing to a hierarchy of files. This I/O transformation thus reduces the number of accesses to the storage *(aggregation)* as well avoids lock contention of files *(subfiling)*. The customization in data aggregation grants a great degree of flexibility while adapting to the needs of a diverse variety of file systems.

The efficacy of the I/O transformation is demonstrated through the PIDX I/O library. The output of this library is a multiresolution data format (IDX) inherently suitable for analysis and visualization tasks. There are two major challenges in the design of the aggregation transformation: 1) translation of multidimensional, application-layout data (row-order or column-order) into the multiresolution layout of the analysis and visualization friendly format (Section 4.1.1) and 2) disk writes (Section 4.1.2).

4.1.1 Data reorganization (HZ encoding)

This step corresponds to on-the-fly transformation of application-layout data (row/column order) residing in the memory of parallel processes to an anlysis and visualization friendly, multiresolution data layout (IDX). It is essential for this transformation to happen as early and inexpensively as possible. The cost of data rerganization increases if delayed and delegated to the network or storage system. To this end, data reorganization is the first step performed and it takes place within the memory of the compute cores. The first step within data reorganization is to identify the number of resolution levels the dataset is going to have. For the IDX format, this is equal to the total number of HZ level (details on how to calculate the HZ level and HZ index are given in Section 3.3). Afterwards, each process allocates memory for each resolution level to hold the reorganized data; following which, the data samples (application layout) are placed at the appropriate location in the newly constructed buffer (storage layout). At the end of this step, every process ends up with an hierarchy of data buffer corresponding to different levels of resolution. A detailed example of the data reorganization step from row-order layout to multiresolution layout for an 8×8 two-dimensional datasets is shown in Figure 4.2.

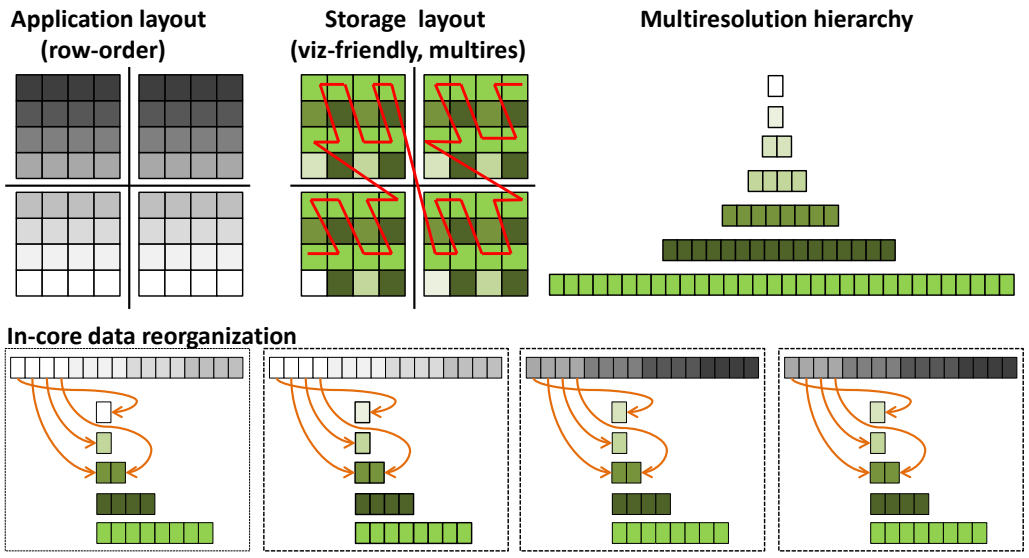


Figure 4.2: Top row, first image is a representative of data stored in row-order within memory of four processes. Top row, second image shows the row-order data transformed into a multiresolution data format (IDX). With the IDX format, data within every resolution level are also written following Z index ordering. For the purpose of simplicity and clarity, Z-order (red color curve) is shown only for the highest resolution level. The shades of green correspond to different levels of resolution (shown in the third figure of the first row). The second row shows data reorganization taking place in parallel within the four processes. This step copies the row-order data onto a hierarchy of buffers (corresponding to the different levels of resolution).

4.1.2 Independent I/O (one-phase I/O)

Once the data are reorganized into an hierarchy of buffers that matches with the layout of the visualization friendly data format, the next step is to write these buffers to a file. We adopted the commonly used approach of independent MPI-I/O to perform these writes. These writes were performed to interleaved portions of a file. Since the buffer created in data reorganization phase maps directly to the final data layout in the file, all processes could perform large-sized contiguous write operations. An example of one-phase I/O is shown in Figure 4.1 (b).

Using a constant per-process load of 128 MB, we investigated the weak scaling behavior of this technique on Surveyor [62]. The total load was then gradually increased linearly as the number of processes was varied from 1 to 512. The results are shown in Figure 4.3. From these results, it was seen that a single process achieves approximately 6.85 MiB/s, comparable to the speed achieved by a serial writer for an equal volume of data. The peak aggregate performance of 406 MiB/s is reached with 512 processes, which is approximately 60% of the peak IOR [44] throughput achievable (667MiB/s) on 512 cores of surveyor.

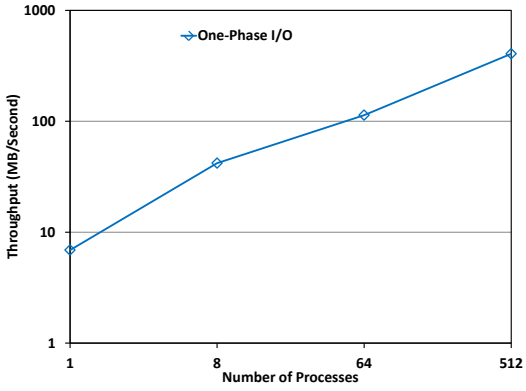


Figure 4.3: Weak scaling performance of independent I/O (one-phase I/O).

The weak scaling example using the independent I/O implementation hit a limit on scalability, falling short of the peak surveyor write performance achieved by IOR [44]. This behavior was further investigated by measuring the time required to write each level of the resolution hierarchy. Figure 4.4 shows the achievable throughput in MiB/s to write an 8 GB data volume consisting of 30 HZ levels on 64 nodes (one process per node) for each level in the resolution hierarchy. As the resolution level increases, the amount of data written doubles. Weak scaling was not able to achieve peak bandwidth as contention and metadata overhead caused the initial levels to take a disproportionate amount of time relative to the amount of data they were writing. In order to mitigate this problem, subfiling based data aggregation strategy was designed and adopted.

4.1.3 Two-phase I/O

The two-phase I/O aggregation scheme is used to mitigate the disk access inefficiencies due to small, noncontiguous file access by aggregating buffers into larger block-aligned buffers. The application data model is translated into an efficient movement of data to

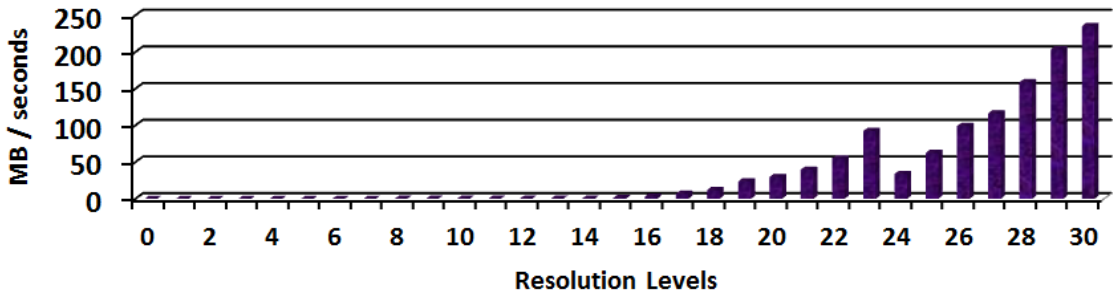


Figure 4.4: Time taken to write all resolution levels for a 8 GB data volume on 64 processes.

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

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

Figure 4.5 illustrates the evolution of the aggregation strategies. In Figure 4.5 (left), each process writes its own data directly to the appropriate underlying binary file, leading to a large number of small accesses to each file. In Figure 4.5 (center), each process uses `MPI_Put()` operations to transmit each contiguous data segment to an intermediate aggregator. Once the aggregator's buffer is complete, the data are written to disk using a single large I/O operation. In Figure 4.5 (right), we go one step further, by bundling

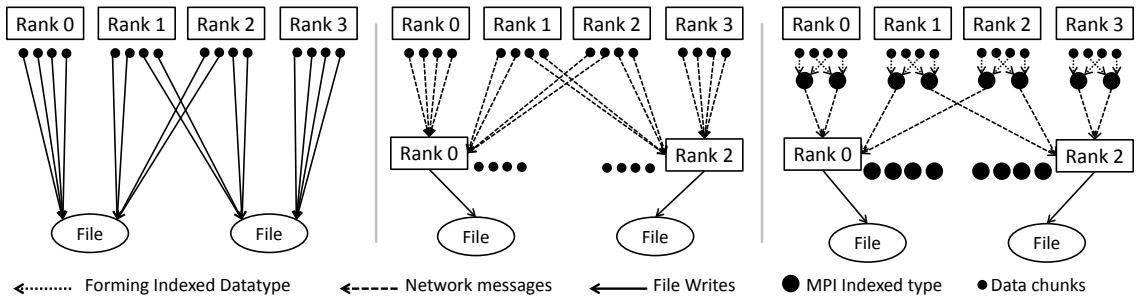


Figure 4.5: Schematic diagram of aggregation strategies: (left) no aggregation, (center) aggregation implemented with RMA, and (right) aggregation implemented with RMA and MPI_Datatypes

several noncontiguous memory accesses from each process into a single `MPI_Put()` using MPI indexed datatypes. This approach reduces the number of small network messages needed to transfer data to aggregators.

4.1.4 Performance evaluation

This section evaluates the performance of the aggregation algorithm compared to previous independent I/O implementation. To this end a microbenchmark was developed that was used to compare three techniques for writing data in parallel: 1) no aggregation is performed and all processes write directly to storage with MPI independent I/O, 2) aggregation is performed using a separate `MPI_Put()` operation for each contiguous region, and (3) MPI datatypes are used to transfer multiple regions using a smaller number of `MPI_Put()` operations. Figure 4.6 depicts performance of the three cases to write 10 time-steps as the number of processes are scaled from 256 to 8192. Each process writes out a $(64)^3$ subvolume with four variables. Aggregation with MPI datatypes yields a significant speed up for I/O performance in comparison to a scheme that uses no aggregation. At 256 processes, an 18-fold speed up is achieved, and at 2048 processes, a 30-fold speed up is achieved over a scheme with no aggregation. The aggregation strategy that utilized MPI datatypes yielded a 20% improvement over the aggregation strategy that issued a separate `MPI_Put()` for each contiguous region. An indexed datatype was used to describe all transfers to a given aggregator at each resolution level. The reason for the performance improvement is that the datatype reduced the number of small messages transferred during aggregation, therefore reducing network congestion. In future work, the step could be optimized further by creating datatypes that span multiple HZ levels.

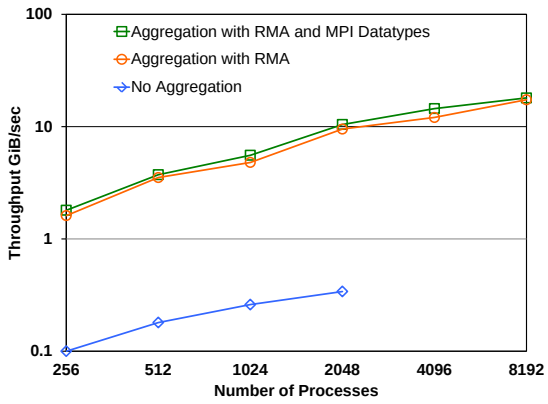


Figure 4.6: Weak scaling performance of independent I/O (blue trendline), two-phase I/O with data aggregation (orange trendline), and two-phase I/O with data aggregation with MPI datatypes (green trendline).

4.2 Data restructuring

Data layout optimized for parallel computing does not always match with the layout that is optimized for I/O and storage. The restructuring transformation increases data locality by aligning the application level layout to the storage layout. The phase works by altering the distribution of data among processes while keeping the data in its multidimensional application layout. Redistribution of data takes place based on a virtual grid imposed on the dataset.

The efficacy of the restructuring phase is demonstrated using the multiresolution data format. The levels of resolution are logically based on even powers of two, typical of level-of-detail schemes. If the original dataset is a power of two in each dimension (i.e., $2^x \times 2^y \times 2^z$), then the resulting ordering (data reorganization) is dense. However, many scientific simulations, such as S3D [2], Flash [63], and GCRM [64], do not normally produce datasets with even, power-of-two dimensions. For clarity in exposition throughout the chapter, the term *irregular* is used to describe datasets that have non-power-of-two dimensions (e.g., 22 $\times$ 36 $\times$ 22) and *regular* to describe datasets that have power-of-two dimensions (e.g., 32^3).

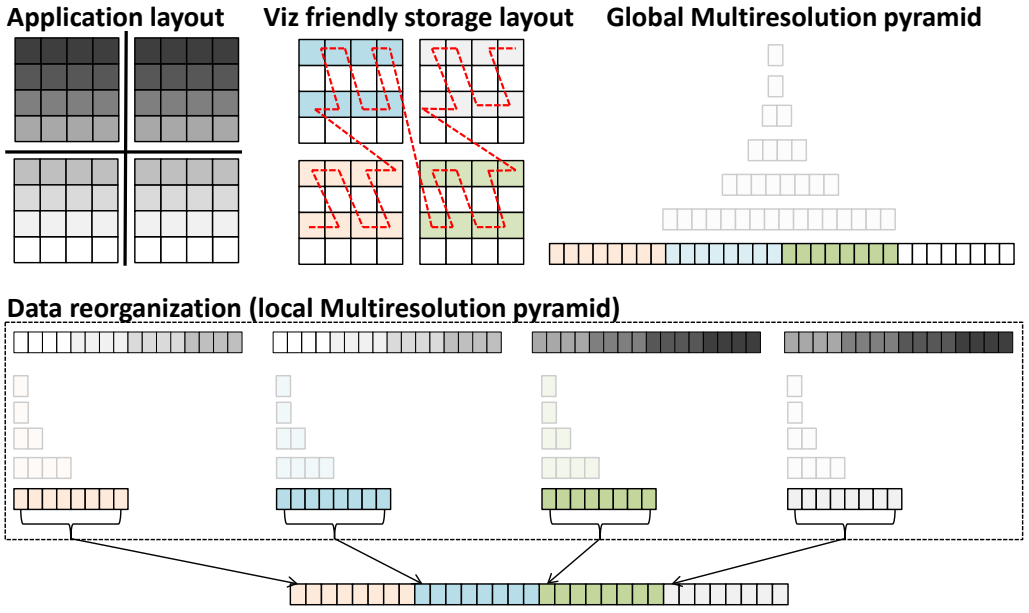
Naive parallel data reorganization of both regular and irregular datasets results in noncontiguous file access. This problem was addressed with aggregation strategies [7] presented in Section 4.1 that perform some amount of network I/O before disk I/O. However, data reorganization (HZ encoding for IDX data) of irregular datasets results in an additional challenge: not only is the file access noncontiguous, but the memory buffer for each process used for encoding is sparse and noncontiguous as well. Even if aggregation is used to improve the file access pattern, the memory buffer access pattern results in too many small messages

sent on the network during aggregation and too much wasted memory on each process to achieve acceptable performance for irregular datasets.

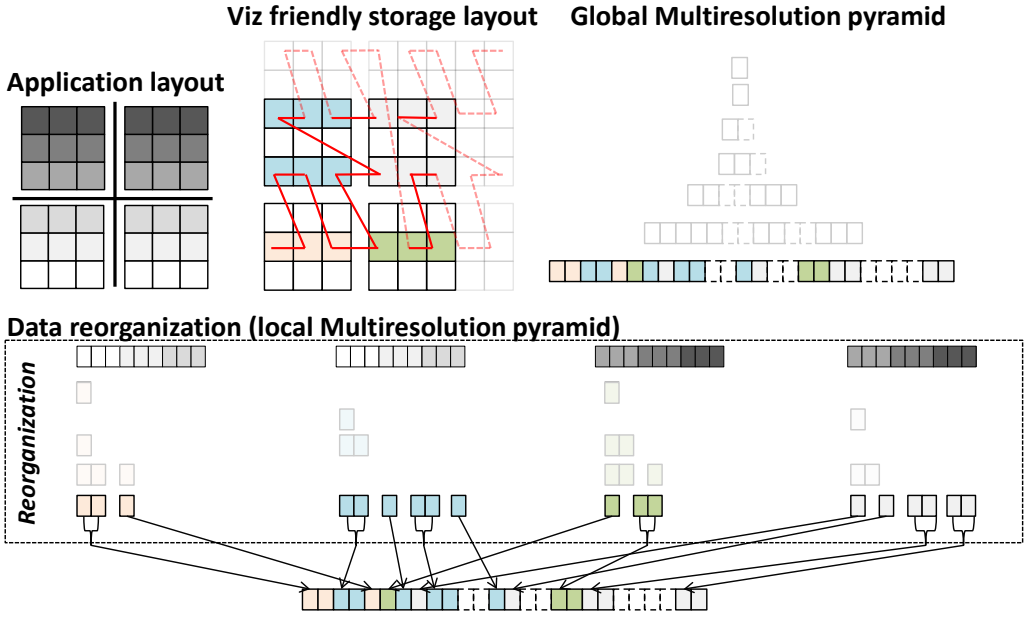
This section presents an I/O transformation that enables efficient writing of irregular datasets from a parallel application. This algorithm extends the two-phase writing (aggregation and disk I/O) used for regular data [7] with a three-phase scheme that restructures irregular data in preparation for aggregation.

Multiresolution data formats are logically based on even powers of two; hence, for regular datasets, the hierarchy of buffers produced after data reorganization is dense and contiguous and accesses noninterleaving regions of the file. An example of the buffer structure is illustrated in Figure 4.7 (a), which shows parallel conversion of a regular 8×8 two-dimensional row major dataset to visualization friendly multiresolution data format using four processes. Each process, indicated by a different color, handles a 4×4 block of data. For the purpose of simplicity and clarity, only indices for the highest resolution level are shown. The remaining resolution levels are not shown here; they are represented by empty boxes - the parallel conversion scheme can be similarly applied to these lower levels. From the figure, it can be seen that all four processes produce a dense, continuous, and noninterleaved mapping from the two-dimensional Cartesian space to the storage-level multiresolution space. This mapping ensures allocation of a minimal memory to store the reorganized data. In addition, the buffers produced by all processes are always nonoverlapping. This approach provides the opportunity for large, noncontentious I/O operations.

Unlike regular datasets, reorganization of application layout into the multiresolution layout of the visualization friendly format of irregular local data produces a sparse, noncontiguous buffer at each resolution level. Furthermore, the data buffers are interleaved across processes. This configuration causes two problems. First, it is wasteful of limited memory resources at each compute node. Second, it leads to inefficient memory access and a high volume of small messages during the aggregation phase of the encoded data (for two-phase I/O). Figure 4.7 (b) illustrates these two problems by showing a parallel conversion of an irregular 6×6 two-dimensional row order data to IDX format by four processes. Each process, indicated by a different color, handles a 3×3 irregular block of data. Since HZ encoding is based on even powers of two, a closest power-of-two (regular) box (8×8) must be used to compute the HZ indices, matching the HZ indexing scheme in Figure 4.7 (a), but skipping the dashed boxes. Examining the arrangement of data within each process, it is apparent that it does not lie on a single, connected piece of the Z curve. For example, process **P0** has only three sample points, sparsely placed in two pieces. Examining the



(a)



(b)

Figure 4.7: Example of data reorganization of a regular and an irregular dataset. (a) Data reorganization of an 8 × 8 two-dimensional regular data performed in parallel by four processes (four colors). Only data corresponding to the highest resolution level is highlighted. Within every process, buffers for all resolution levels are spanned by one Z curve and are dense and contiguous. (b) Data reorganization of a 6 × 6 irregular dataset. Dashed boxes are for nonexistent data points. Since multiresolution schemes require dimensions in powers of two, a sparse, interleaved, and overlapping disk level layout is produced.

disk-level layout clearly shows the sparse, noncontiguous, and overlapping data pattern of the four processes.

4.2.1 Three-phase I/O

The two-phase aggregation algorithm described in Section 4.1 distributes data to a subset of processes after data within every process is locally reorganized to match with the layout of the multiresolution data format. This technique is efficient only in cases in which the local data reorganization (application layout to multiresolution storage layout) at each process produces a dense buffer. To effectively handle irregular datasets that do not exhibit this characteristic, an additional *restructuring phase* is introduced to the algorithm before data reorganization and two-phase aggregation. This additional restructuring phase alters the distribution of data among processes while the data are still in their initial multidimensional format. The goal is to distribute the dataset such that each process holds a regular, power-of-two subset of the total volume. This distribution can be performed by using efficient, large, nearest-neighbor messages. Once the data have been structured in this manner, the subsequent local data reorganization at each process produces a dense contiguous and noninterleaving buffer that can be written to disk efficiently using two-phase aggregation.

Figure 4.8 further elucidates the effect of the data restructuring phase. Comparing Figure 4.8 with Figure 4.7 (b), it can be seen that the restructuring phase transforms the dimensions of process **P0**, from a *(3× 3) irregular* box to a *(4 × 4) regular* box. The HZ encoding of data in this layout produces an efficient, dense memory layout. The only exceptions are relatively small boundary regions of the data, which do not fit into even power-of-two subvolumes; but even in that case, the restructuring at least eliminates interleaving of HZ-encoded data across processes.

Figure 4.9 illustrates the three I/O phases in detail. Data restructuring is a localized phase involving only neighboring processes, at the end of which there is a set of processes handling regular boxes. With this scheme of data restructuring, barring a few boundary blocks, almost every irregular data block is transformed into a regular one. In the current scheme, independent I/O of IDX block sizes are used to write the irregular edge blocks. The implementation is both scalable and efficient because of the localized nature of communication among processes. For any imposed regular box, only a group of neighboring process can participate in the restructuring phase.

To restructure data, a virtual grid of regular boxes is imposed over the entire volume set. These regular boxes are then used to redistribute the irregularly stored data on each process. To resolve the different boundaries, data are judiciously exchanged among processes. Point-

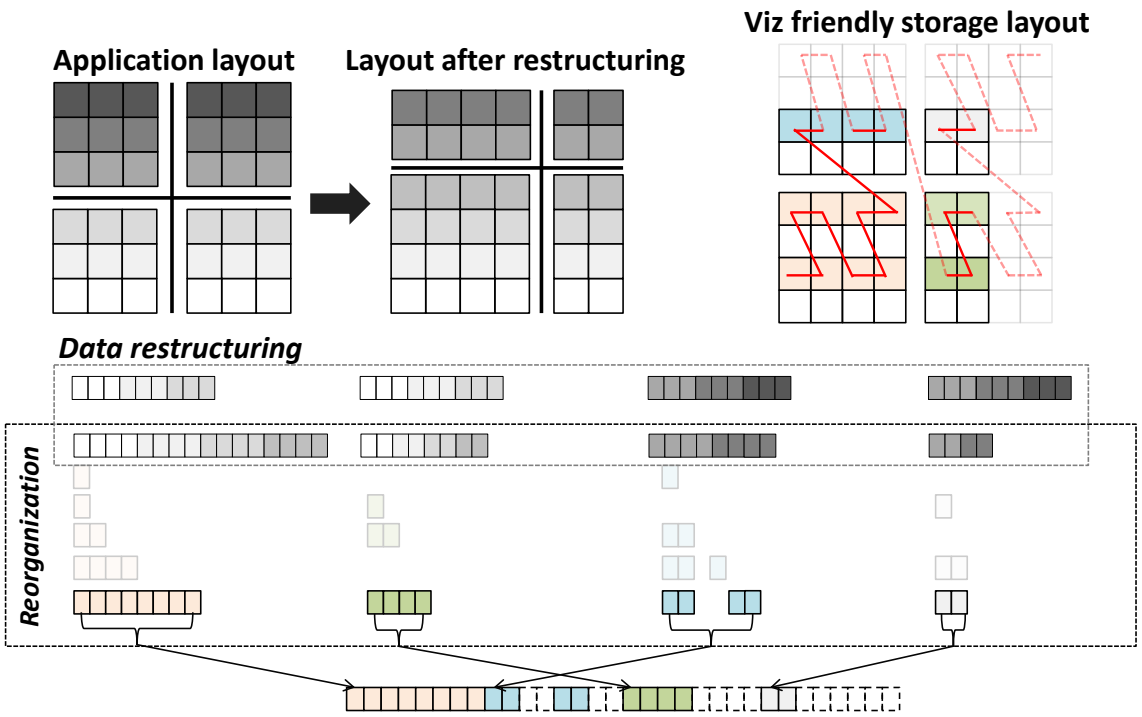


Figure 4.8: Data restructuring transforms the dimensions of process from a *(3× 3) irregular* box to a *(4 × 4) regular* box. The HZ encoding of data in this layout produces an efficient, dense memory layout. By restructuring the data, interleaving of the data of each process is also reduced despite the fact that loads on the boundary have different sizes.

to-point MPI communication (using MPI_Irecv and MPI_Isend) is used to transfer data among processes. This data restructuring leads to a grid of regular data boxes stored on some subset of processes.

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

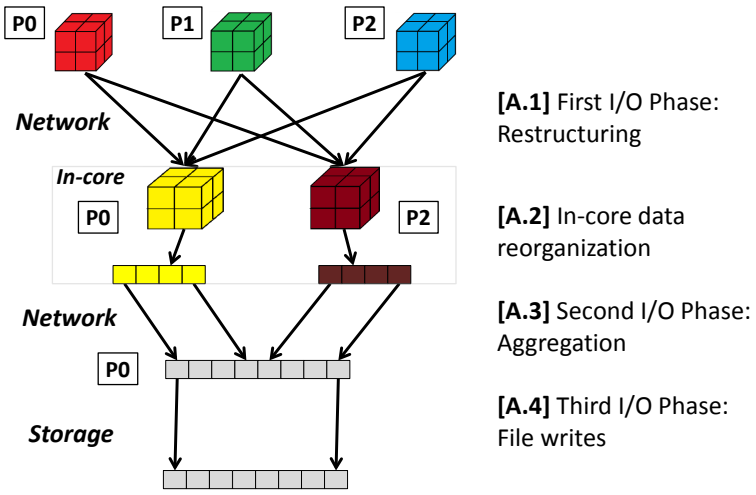


Figure 4.9: Three-phase I/O: [**A.1**] Data restructuring among processes transforms irregular data blocks at processes P0, P1, and P2 to regular data blocks at processes P0 and P2. [**A.2**] HZ encoding of regular blocks leading to dense and nonoverlapping data buffer. [**A.3**] Data transfer from in-memory HZ-ordered data to an aggregation buffer involving fewer large-sized data packets. [**A.4**] I/O writes from aggregation buffer to an IDX file.

point-to-point communication (using MPI_Irecv and MPI_Isend) is used among processes for data exchanges.

4.2.2 Performance evaluation

This section evaluates the performance of the restructuring algorithm compared to previous algorithms. To this end, a microbenchmark was developed to evaluate the performance of PIDX for various dataset volumes. The benchmark was used to compare three techniques for writing IDX data in parallel. The first is a naive implementation, in which each process performs an HZ encoding and writes its data directly to disk. The second uses two-phase

Algorithm 1 Data Restructuring

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

aggregation after the HZ-encoding step. The third algorithm introduces a data restructuring phase prior to HZ encoding. The aggregate performance on Hopper and the per process memory load of each implementation are shown in Figure 4.10. The per process data volume was set to 60^3, with each element containing two variables, with four floating-point samples per variable. This configuration produced 13.18 MiB of data per process.

The naive one-phase implementation performed poorly because of the combination of both discontiguous memory buffers after HZ encoding and discontiguous file access with small write operations. At 256 processes, it achieved 85 MiB/s. Each process consumed

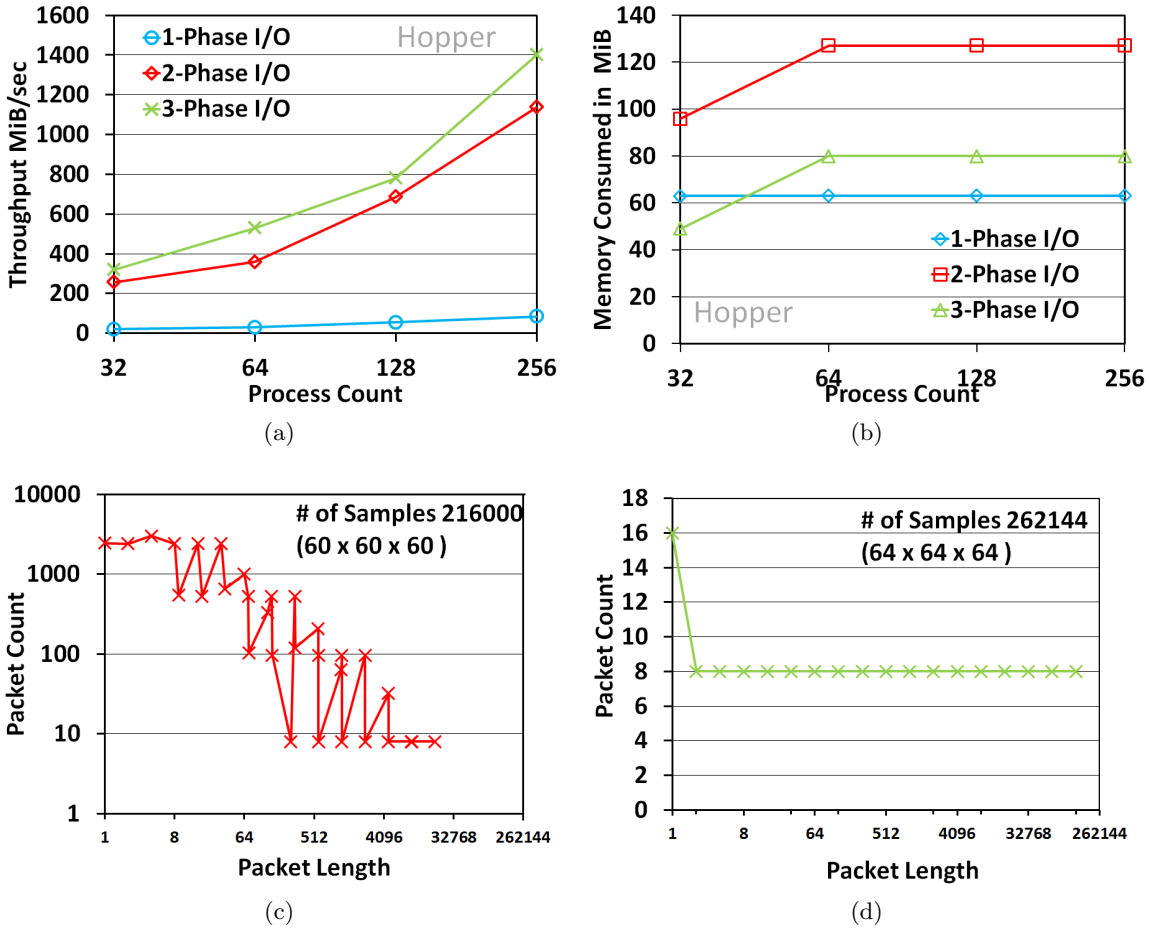


Figure 4.10: Effects of restructuring phase on performance and memory footprint. (a) Throughput for weak scaling of the three types of I/O (one-phase, two-phase and three-phase), with 60^3 block size. (b) Per process memory footprint of weak scaling, with a 60^3 block size . Size and number of data packets sent during the aggregation phase by process with rank 0, at process count 256 (from Figure 4.10a), for both (c) two-phase and (d) three-phase I/O. The effect of the data restructuring phase can be seen as all small data packets get bundled into a small number of large-sized packets.

approximately 64 MiB of memory (nearly five times the size of the original data volume) because of the sparse HZ encoding.

A two-phase I/O aggregation scheme mitigates the disk access inefficiencies due to small, noncontiguous file access by aggregating buffers into larger block-aligned buffers. This optimization improved throughput from 85 to 1100 MiB/s. The implementation performance still suffers from small network transfers during the aggregation phase due to the sparse HZ encoding. This is illustrated in Figure 4.10 (c), which shows the frequency and size of messages sent from rank 0 as an example. There are almost 10,000 messages of fewer than four bytes. In addition, the scheme consumes even more memory than the previous implementation because of the extra buffer used for aggregation. The aggregation buffer size varied from 32 to 64 MiB depending on the overall data volume.

By restructuring the data before aggregation and using a three-phase I/O scheme, significant performance gains can be made: specifically, additional 25% performance improvement over the two-phase aggregation algorithm, reaching 1400 MiB/s. Figure 4.10 (d) illustrates the frequency and size of messages sent from rank 0 during aggregation after the restructuring phase. The message size is significantly larger, leading to fewer total messages and significantly improved overall performance. In addition, the memory consumption per process is reduced in comparison to that of the two-phase algorithm because of the dense HZ encoding that is performed on restructured, power-of-two data.

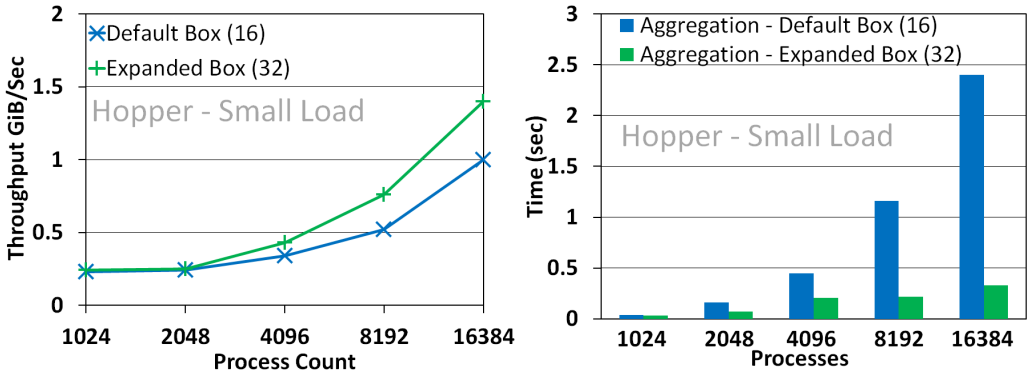
4.2.3 Parameter study

The tunable parameter in data restructuring that has the most profound impact on runtime behavior is the box size used for power-of-two restructuring. This parameter directly affects the network traffic, and thus can be effectively used to reflect the characteristics of the network on a given system. The restructuring phase provides some flexibility for controlling the size of the imposing regular box. Specifically, there is a parameter that switches between the default-imposed box and an expanded box. The default imposed box size is calculated at runtime by rounding up the data volume at each process to the nearest power of two. The expanded box size doubles the size of the box in each dimension. This parameter affects both the restructuring phase of the algorithm and the aggregation phase of the algorithm as follows:

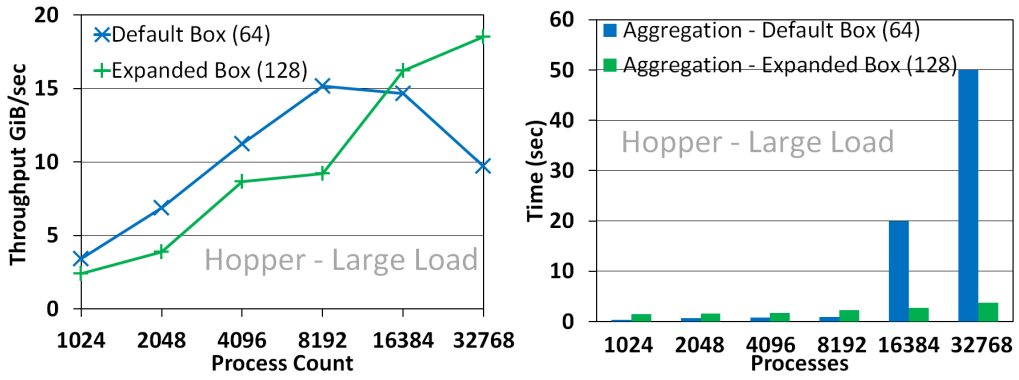
1. Data restructuring phase: change in the time needed to distribute the data as they are being transmitted to different numbers of intermediate nodes.
2. Data aggregation phase: change in time needed to send the data to the aggregator nodes as the data are coming from different numbers of intermediate nodes.

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

A set of experiments varying both scale and load was conducted to understand the behavior of this parameter. Both a small per process load of 15^3 and a relatively larger load of 60^3 were used. As in earlier experiments, each element consisted of two variables, each in turn containing four floating-point values. The small and large volumes were 0.2 MiB and 13.18 MiB, respectively. Figure 4.11 shows the performance of these two scenarios as the microbenchmark is scaled from 1,024 to 16,384 processes for the smaller load, and



(a)



(b)

Figure 4.11: Weak scaling results on Hopper as the restructuring box size is varied. (a) A small load of 15^3 per process with a default and expanded box size of 16^3 and 32^3, respectively. (b) A larger load of 60^3 per process with default and expanded box size of 64^3 and 128^3, respectively.

from 1,024 to 32,768 for the larger load; the box size is toggled between the default and expanded setting.

It was found that the time consumed by the restructuring phase was not greatly affected by the box size. For example, at 16,384 processes, it varied from 0.012 seconds with a default box and 0.018 seconds with an expanded box. The reason is that this algorithm is well distributed and involves communication with neighboring processes. As a result, we focused our attention on the impact of the aggregation phase. Figure 4.11 illustrates both the aggregate performance of the algorithm and the time consumed in the aggregation phase of the algorithm.

For the smaller load, both box sizes follow similar trends, but the performance gap between the default and expanded setting increases at scale. This result can best be explained by looking at the histogram of time spent during the aggregation phase with both the default and expanded box (Figure 4.11 (a)). From the graph, one can clearly see the increasing amounts of time spent performing aggregation with the default box. At process count 4,096, for example, the aggregation phase with the expanded box involves fewer processes (422) with relatively larger loads (2 MiB), as opposed to the default box where aggregation involves too many processes (3,375), each with a smaller load (256 KiB). Thus, the difference in performance can be attributed to the extra overhead caused by having too many nodes transmitting small data volumes during aggregation.

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

The results from a single system (Hopper) seem to indicate that this parameter could be tuned automatically based simply on the data volume and scale of an application. However, this is not necessarily the case when the experiment is repeated on Intrepid, which has a different network infrastructure. The results of this experiment are shown in Figure 4.12. In this case, the performance improvement from using an expanded box is negligible as the scale is increased, indicating that the Intrepid network is less sensitive to variations in the number and size of network messages. These results overall indicate that the restructuring

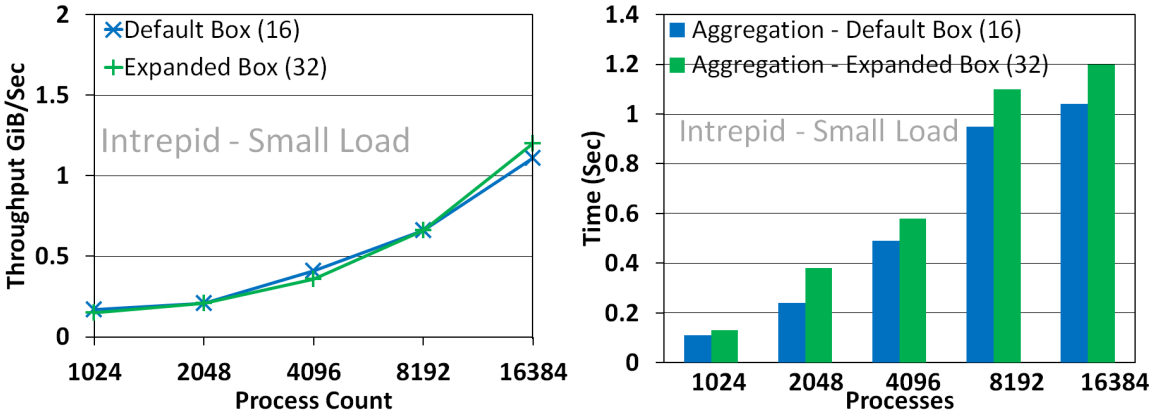


Figure 4.12: Intrepid (BG/P) results for weak scaling with per process data block of size 15^3. The default box size is 16^3, and the expanded box size is 32^3.

box size parameter should be selected to reflect characteristics of both the dataset and the computing system.

4.3 Parallel IDX (PIDX)

The efficacy of restructuring and aggregation has been empirically demonstrated in the form of the PIDX I/O library [10, 7, 11, 12, 13]. Parallel IDX (PIDX) enables simulations to write data directly in the visualization appropriate multiresolution IDX data format. It coordinates data access among participating processes so that they can write concurrently to the same dataset with coherent results. Previous I/O infrastructure [14] to read and write IDX data was serial in nature, which prevented parallel simulations from writing directly into the visualization friendly format. Data could be converted as a postprocessing step; however, this would significantly increase the analysis turn around time and make poor use of available parallel I/O resources available in todays leadership-class computing facilities. Parallel IDX (PIDX) library was therefore developed to enable simulations to write IDX data directly. The PIDX I/O library is open-source and can be downloaded from the url: https://github.com/sci-visus/PIDX. The data restructuring and aggregation strategy presented in the preceding subsections are two steps in the complete PIDX pipeline. The entire process can be summarized as follows:

1. Describe data model (see Section 4.3.1)
2. Create an IDX block bitmask
3. Create underlying file and directory hierarchy
4. Restructure data in its application layout (see Section 4.2)
5. Perform HZ encoding (see Section 4.1.1)

6. Aggregate data (see Section 4.1)
7. Write data to storage.

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

The IDX file and directory hierarchy is created in parallel before any I/O is performed. After creating files, data restructuring is invoked, at the end of which all processes end up having power-of-two sized boxes. Then the HZ encoding step is performed independently on each process. In order to minimize memory access complexity, all samples are copied into intermediate buffers in a linear Z ordering. Note that the Z ordered data from each process may span multiple HZ levels or multiple files within the overall IDX dataset. Aggregation is performed as described in Section 4.1. The final step is to write the HZ ordered data to disk. This step is performed using independent MPI-IO write operations. Due to explicit aggregation within PIDX, there is no need for collective I/O or derived data types at this step.

4.3.1 Expressing HPC data models with PIDX

It is essential to capture the application data model in a way that provides a complete picture of the intended data movement so that the I/O library can calculate an optimal strategy for its computation and I/O phases. For example, HPC simulations typically generate data for several related multidimensional variables at each simulation time-step.

If this collection of variables is stored in a naive manner, then there is a risk of incurring inefficiency from redundant HZ ordering calculations and suboptimal file access patterns. To this end, a API is tailored to the needs of HPC application data models.

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

To overcome this problem, an enhanced API was devised that decouples the definition of the application's data model from the actual transfer of data to storage. This technique has proven successful in a variety of existing high-level libraries such as HDF5 [5] and pNetCDF [4]. The PIDX API allows applications to store a collection of dense multidimensional variables. Each variable has its own type and can be written from an arbitrary memory layout on each process. An example of the use of this enhanced PIDX API is shown in Listing 4.2. The first step is to define a variable, which includes an indication of how many samples make up an instance of the variable and what MPI datatype will represent the variable. The second step is to describe how the variable is laid out in local memory, including any striding information. The third step is to add the variable to the dataset if the local process uses that variable. These initial three steps can be repeated as needed to describe all variables to be included in an IDX dataset. The final phase is the `PIDX_close()` command, which tells the library to transfer the entire dataset to storage. This organization allows PIDX to leverage as much concurrency as possible by taking all variables into account simultaneously. It also allows PIDX to reuse HZ calculations where possible for variables that share the same dimensions.

Listing 4.1: Enhanced PIDX API example

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

/* add local variables to the dataset */
PIDX_variable_write_data_layout(var1, offset, count, data);
```

```
/* describe memory layout */
PIDX_append_and_write_variable(dataset, var1);

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

4.3.2 PIDX performance evaluation

This section evaluates the performance of PIDX to directly write IDX datasets for each time-step of the S3D combustion simulation. S3D is a continuum scale first principles direct numerical simulation code that solves the compressible governing equations of mass continuity, momenta, energy, and mass fractions of chemical species including chemical reactions. The computational approach in S3D is described in [65]. In the S3D code, each rank is responsible for a piece of the three-dimensional domain; all MPI ranks have the same number of grid points and the same computational load under a Cartesian decomposition. S3D has been run successfully on up to near the full size (216,000 cores) of the NCCS XT5 jaguarpf, demonstrating near linear scaling up to approximately half of the machine (approximately 120k ranks) with the current validated production code base in the weak scaling limit. Similarly, S3D has demonstrated near linear scaling to 120,000 cores on the CrayXE6, Hopper2, at NERSC. S3D can be compiled with support for several I/O schemes, which are then selected at runtime via a configuration parameter. S3D I/O extracts just the portion of S3D concerning restart dumps, allowing us to focus exclusively on I/O characteristics. For our evaluation, an S3D I/O configuration was used wherein each process produced a 64^3 volume consisting of 4 fields (field 1 and 2 each of just 1 sample, field 3 with 3 samples and field 4 with 11 samples), which produces 32 MiB of data per process. Because PIDX is implemented in C, a wrapper was developed to facilitate the use of PIDX with S3D, which is a Fortran code. A custom I/O module in S3D was incorporated to enable the use of PIDX as an alternative to the existing schemes.

4.3.2.1 Experiment setup and platform

The experiments presented in this work have been performed on both the Hopper system at the National Energy Research Scientific Computing (NERSC) Center and the Intrepid system at the Argonne Leadership Computing Facility (ALCF). Hopper is a Cray XE6 with a peak performance of 1.28 petaflops, 153,216 cores for running scientific applications, 212 TB of memory, and 2 petabytes of online disk storage. The Lustre [21] scratch file system on Hopper used in evaluation consists of 26 I/O servers, each of which provides access to

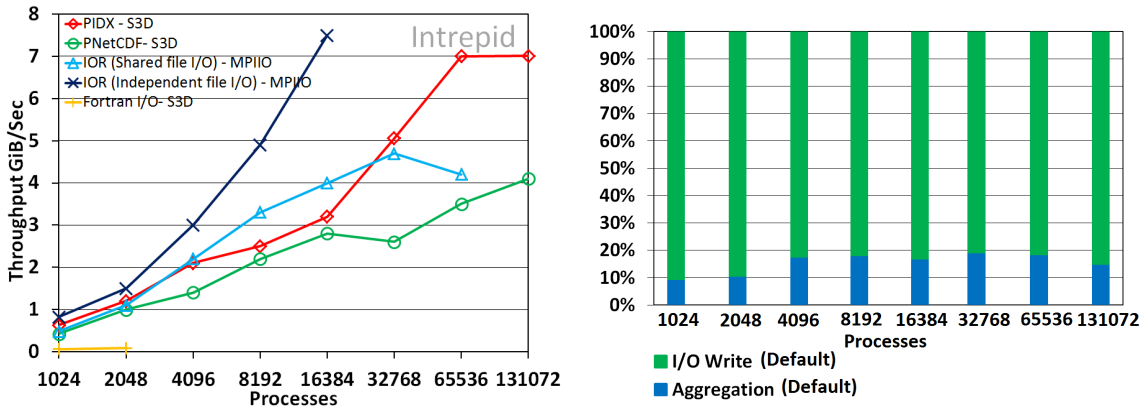
six object storage targets (OSTs). Unless otherwise noted, the default Lustre parameters which stripe each file across two OSTs are used. Intrepid is a Blue Gene/P system with a total of 164K cores with 80 terabytes of RAM and a peak performance of 557 teraflops. The storage system consists of 640 I/O nodes that connect to 128 file servers and 16 DDN 9900 storage devices. GPFS [20] file system is used on Intrepid for all experiments. The Intrepid file system was nearly full (95% capacity) during this evaluation. It is believed that this significantly degraded I/O performance on Intrepid.

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

4.3.2.2 Weak scaling

This section evaluates the weak scaling performance of PIDX when writing irregular (non-power-of-two) S3D datasets on both Intrepid and Hopper. In each run, S3D writes out 20 time-steps wherein each process contributes a 30^3 block of double-precision data (3.29 MiB) consisting of four variables: pressure, temperature, velocity (3 samples), and species (11 samples). On Hopper, the number of processes was varied from 1,024 to 65,536, thus varying the amount of data generated per time-step from 3.29 GiB to 210.93 GiB. On Intrepid, the number of processes as varied from 1,024 to 131,072, thus varying the amount of data generated per time-step from 3.29 GiB to 421.875 GiB.

Weak scaling results for PIDX and other parallel file formats on Intrepid are shown in Figure 4.13 (a). At the time of these experiments, the Intrepid file system was nearly full (95% capacity), which is believed to have seriously degraded I/O performance for all experiments. Although each of the output methods showed scaling, none of the output methods approached the expected throughput of Intrepid at scale. Default restructuring box size of 32^3 was used in all cases.



(a) (b)

Figure 4.13: Intrepid scaling results. (a) Weak scaling of different I/O mechanisms including PIDX, Fortran I/O, and PnetCDF generating irregular datasets with block size 30^3. The total data volume ranges from 3.29 GiB to 422 GiB. (b) The proportion of time taken by the aggregation phase and the I/O write phase at each scale.

Looking at the performance results in Figure 4.13 (a), it is observed that PIDX scales well up to 65,536 processes, and that for all process counts, it performs better than PnetCDF and Fortran I/O. This behavior is an artifact of PIDX aggregation, which finds a middle ground between shared-file and file-per-process I/O. Fortran I/O uses a unique file per process and places all files in the same subdirectory. This approach caused a high degree of metadata contention and serialization. In contrast, PIDX creates a hierarchy of subdirectories at rank 0 and coordinates file creation to avoid directory contention. The IOR unique file case appears to perform well because the measurements did not include directory creation time. IOR was configured with the *uniqueDir = 1*, which creates a separate subdirectory for each file in an attempt to show throughput in the absence of metadata contention at file creation time. However, this simply shifted the contention to directory creation time (which is not measured by IOR) instead of file creation time. As a result, is was possible to IOR with unique files only up to a scale of 16,384 despite its apparent performance because subdirectory creation was taking as much as 3 hours to complete at the largest scale.

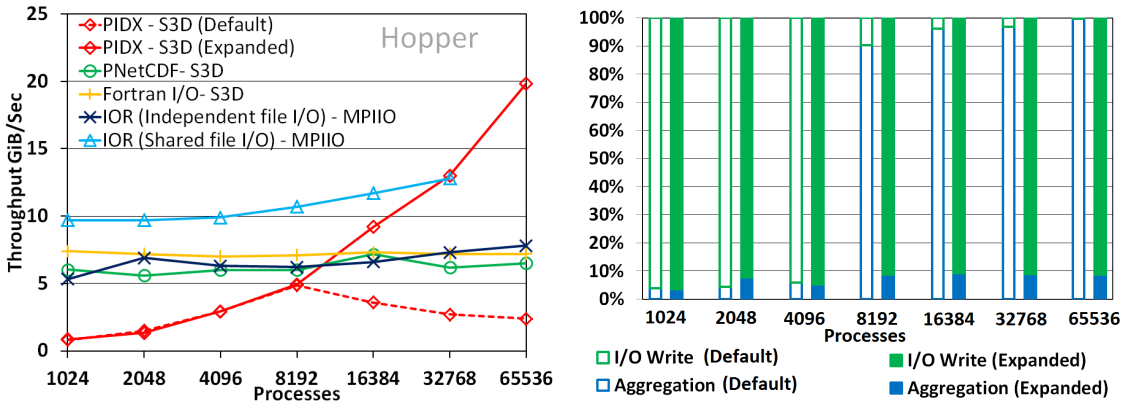
Table 4.1 shows the number of processes that are responsible for restructured data as well as the number of I/O aggregators. Both numbers grow linearly as the application scales. Scalability of aggregation on Intrepid can also be seen from Figure 4.13 (b), which shows the normalized timings of aggregation and I/O write phases. The figure indicates a low overhead of the PIDX network phases, as most of the PIDX write time is spent on disk I/O.

Table 4.1: The number of processes responsible for restructured data and the number of aggregator processes at each scale

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

Weak scaling results for Hopper are shown in Figure 4.14 (a). As described in Section 4.2.3, aggregation scaling on Hopper is affected by the size of the imposed regular box. Using an expanded box during the restructuring phase changes the distribution of data, effectively altering the number of processes participating in aggregation. Hence, two sets of experiments were performed on Hopper, using both a default imposed box and an expanded one, with dimensions 32^3 and 64^3, respectively.

Comparable performance for default and expanded boxes can be seen at lower process counts. This behavior corresponds to the case when the aggregation phase takes a similar amount of time, regardless of the number of processes participating, also supported by the



(a) (b)

Figure 4.14: Hopper scaling results. (a) Weak scaling of different I/O mechanisms including PIDX, Fortran I/O, and PnetCDF generating irregular datasets with block size 30^3. (b) The proportion of time taken by the aggregation phase and the I/O write phase as the number of processes is scaled from 1,024 (3.29 GiB) to 65,536 (210.93 GiB) for both default and expanded box.

phase-wise partition graph in Figure 4.14 (b). With increasing process counts, aggregation using the default box fails to scale, whereas aggregation with the expanded box continues to scale even at higher process counts. Table 4.2 indicates how the number of processes responsible for restructured data varies significantly with the default and the expanded box. For instance, at process count 32,768, the data are redistributed to 27,000 processes as opposed to only 3,375 processes with the expanded box. These two different process counts lead to aggregation phases with very different runtimes. As Figure 4.14 (b) shows, aggregation at 32,768 processes takes the majority of time when using the default box, whereas with the expanded box it takes under 10% of the total PIDX write time.

Compared with other file formats PnetCDF and Fortran I/O, PIDX appears to lag in the lower process count ranges ($\leq$ 8192); but as the number of processes increases, PIDX outperforms them. This lag in performance for the lower process counts can be attributed to a lack of aggregators. It can be seen in Figure 4.14 (a) how at lower process counts, Hopper yields higher throughput with relatively large number of aggregators controlled by blocks per file. For lower process counts, PIDX performed optimally with 64 blocks per file, transcending into a larger set of aggregators, whereas the experiments here used 512 blocks per file. Comparing performance numbers, it can be seen at process count 65,536, PIDX achieves a throughput of around 19.84 GiB/sec, which is approximately three times that of Fortran I/O (7.2 GiB/sec) as well as PnetCDF (6.5 GiB/sec).

4.3.2.3 PIDX example code

Listing 4.2 is an example to write a three dimensional volume of size $64 \times 64 \times 64$ using 8 processes each writing a $32 \times 32 \times 32$ chunk of volume.

Table 4.2: The number of processes responsible for restructured data when using the default (Def.) and expanded (Exp.) imposed box as compared with the number of aggregator processes.

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

Listing 4.2: PIDX example code (code.c) to write a three-dimensional volume.

```
#include <PIDX.h>

int main(int argc, char **argv)
{
  int rank = 0;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  int g_size[3] = {64, 64, 64};  //global size of the 3D volume
  int l_size[3] = {32, 32, 32};  //local size of the per-process volume

  // Calculating every process data's offset (l_offset)
  int slice = 0, sub_div[3], l_offset[3];
  sub_div[0] = (g_size[0] / l_size[0]);
  sub_div[1] = (g_size[1] / l_size[1]);
  sub_div[2] = (g_size[2] / l_size[2]);
  l_offset[2] = (rank / (sub_div[0] * sub_div[1])) * l_size[2];
  slice = rank % (sub_div[0] * sub_div[1]);
  l_offset[1] = (slice / sub_div[0]) * l_size[1];
  l_offset[0] = (slice % sub_div[0]) * l_size[0];

  // Synthetic simulation data
  float *data;
  int i, j, k, index;
  data = malloc(sizeof (float) * l_size[0] * l_size[1] * l_size[2]);
  for (k = 0; k < l_size[2]; k++)
    for (j = 0; j < l_size[1]; j++)
      for (i = 0; i < l_size[0]; i++)
      {
        index = (l_size[0] * l_size[1] * k) + (l_size[0] * j) + i;
        data[index] = rank;
      }

  // PIDX points
  PIDX_point p_g_size, p_l_offset, p_l_size;
  PIDX_set_point_5D(p_g_size, g_size[0], g_size[1], g_size[2], 1, 1);
  PIDX_set_point_5D(p_l_offset, l_offset[0], l_offset[1], l_offset[2], 0, 0);
  PIDX_set_point_5D(p_l_size, l_size[0], l_size[1], l_size[2], 1, 1);

  //  Creating parallel PIDX access
  PIDX_access access;
  PIDX_create_access(&access);
  PIDX_set_mpi_access(access, MPI_COMM_WORLD);

  // Creating IDX file
  PIDX_file file;
  PIDX_file_create("IDX_file.idx", PIDX_MODE_CREATE, access, &file);
  PIDX_set_dims(file, p_g_size);
  PIDX_set_current_time_step(file, 0);
  PIDX_set_variable_count(file, 1);

  // Create PIDX variable and add it to the dataset
  PIDX_variable var;
  PIDX_variable_create("IDX_variable", sizeof(float) * 8, FLOAT32, &var);
  PIDX_variable_write_data_layout(var, p_l_offset, p_l_size, data, PIDX_row_major);
  PIDX_append_and_write_variable(file, var);

  PIDX_close(file);             // Close IDX file
  PIDX_close_access(access);    // Close PIDX access

  free(data);
  MPI_Finalize();
  return 0;
}
```

The code can be executed in parallel using the command line argument `mpirun -n 8 code`. Note that the program is set up to run in parallel using 8 processes ($(64 \times 64 \times 64)$ / $32 \times 32 \times 32$). Both global volume size and local volume size can be changed by editing line numbers 8 and 9 of the code. The number of processes can then accordingly be calculated by dividing the global volume with the local volume. Detailed documentation of the code can be found at the url: https://github.com/sci-visus/PIDX.

4.4 Conclusion

This chapter has presented two I/O transformations, data restructuring and data aggregation, to write data directly from simulations into visualization-friendly IDX format. These transformations have been implemented within the framework of the PIDX I/O library. Using S3D I/O, PIDX has been shown to scale up to 131,072 processes on Intrepid BG/P and up to 65,536 on Hopper, with performance competitive with other commonly used parallel file formats, Fortran I/O and PnetCDF. In addition to its performance, one of the major benefits of PIDX is that it strikes a balance between efficient read I/O for multiresolution visualization as well as effective write I/O for large-scale simulation. Since PIDX implements a multiphase scheme, understanding the behavior of its parameters is important. Exploring the nature of how these parameters need to be chosen has indicated that they can be both application dependent and architecture dependent. In fact, knowing the effects of a combination of these two factors is often necessary in order to achieve high amounts of efficiency. For data that are unevenly distributed among processes, PIDX offers a generic scheme for evening out the distribution, which applies to data consisting of irregularly sized blocks, regularly sized blocks, and their combinations.

CHAPTER 5

DOMAIN PARTITIONING AND BLOCK-BASED COMPRESSION

This chapter presents transformations to enable I/O to efficiently scale up to 768K processes. This work builds on the PIDX I/O [10, 7, 11, 12, 13] framework (presented in Chapter 4), which utilizes subfiling, restructuring, and aggregation to efficiently write data in the hierarchical, multiresolution IDX file format [14]. Although PIDX has been demonstrated to scale better than other parallel I/O methods, the scaling performance of PIDX is still nonoptimal. Due to the nature of communication in the data aggregation phase, PIDX requires global synchronization, which ultimately limits its scaling efficiency.

This chapter presents an I/O transformation *domain partitioning* that alters the two-phase I/O algorithm in a way that eliminates all global communication and synchronization during data aggregation. This is done by limiting the required communication to local subgroups of processors by partitioning the global domain and writing the partitions as multiple independent datasets. This process isolates subgroups such that each file is written completely in parallel, resulting in a framework that is scalable up to the largest systems available today. Furthermore, a simple index shift enables writing the vast majority of the data relative to a (virtual) global index space. The few files written in local index space are then merged in a lightweight postprocess, recreating the single file access point most convenient for a user.

The second transformation is a block-based, fixed-bit compression strategy that enables data access at both varying spatial resolution and numerical precision. Data compression is an approach to both reduce the size of analysis and visualization dumps and facilitate improved scaling. Recent results with state-of-the-art, block-based compression schemes have shown that for many analysis and visualization tasks, sufficient results can be obtained using compression to as little as two to four bits per double precision value [66]. However, the hierarchical layout of the IDX format lowers the coherence of adjacent samples, rendering block-based compression less effective, especially at coarser resolutions. This problem is

tackled by introducing the notion of *chunks* - small local blocks of samples, e.g., 4×4×4 - that take the place of individual samples. Since chunks are spatially coherent by construction, they compress better than the files or disk blocks of the original format. Using the fixed-rate lossy compression recently introduced by Lindstrom [66], this method provides significant file reductions with minimal read and write overheads, enabling more frequent visualization and analysis dumps.

The PIDX framework with integrated partitioning and compression provides the first fully scalable I/O solution, which is demonstrated by showing for the first time parallel I/O scaling up to 768K cores.

5.1 Domain partitioning

An I/O-centric view of the typical simulation pipeline (see Figure 5.1) is as follows: the simulation domain is first divided into elements (usually pixels or voxels) and then elements are grouped into patches. Each patch is assigned a rank, or processor number. Rank assignment is done by the simulation software using a deterministic indexing scheme. For example, S3D assigns ranks in a row-major order, and Uintah assigns them in Z-order. Rank assignment is important at simulation time because interprocess communication will occur between patches close to each other in simulation space, but if their processors are distant in network space, then more communication overhead will occur.

When a data dump comes due, elements are encoded or assigned their IDX format index and then sent to designated aggregator nodes and written to disk. Aggregation is done before disk I/O in order to minimize the number of writes to disk. As shown in [24], the cost of increased network communication for aggregation is justified by significant disk I/O performance gains.

With an increasing number of core counts, the time spent during data exchange involved in the aggregation step starts to become significant, impeding scalability. This problem is demonstrated by the parallel HDF5 library that uses MPI collective I/O for communication (see Figure 5.2 (a), red trendline) and for PIDX, which has its own custom aggregation phase using one-sided RMA communication (see Figure 5.2 (b), red trendline). This overhead arises from a number of possible reasons: higher latency due to bigger networking space, internal overhead of MPI collective I/O, or a large number of target aggregators per process that can possibly be farther apart in the networking space. This work tackles the communication costs of aggregation by proposing a network partitioning scheme. The main idea is to partition the processor space and write one IDX file per partition in order to reduce the amount of internode communication in the aggregation step. Processors communicate

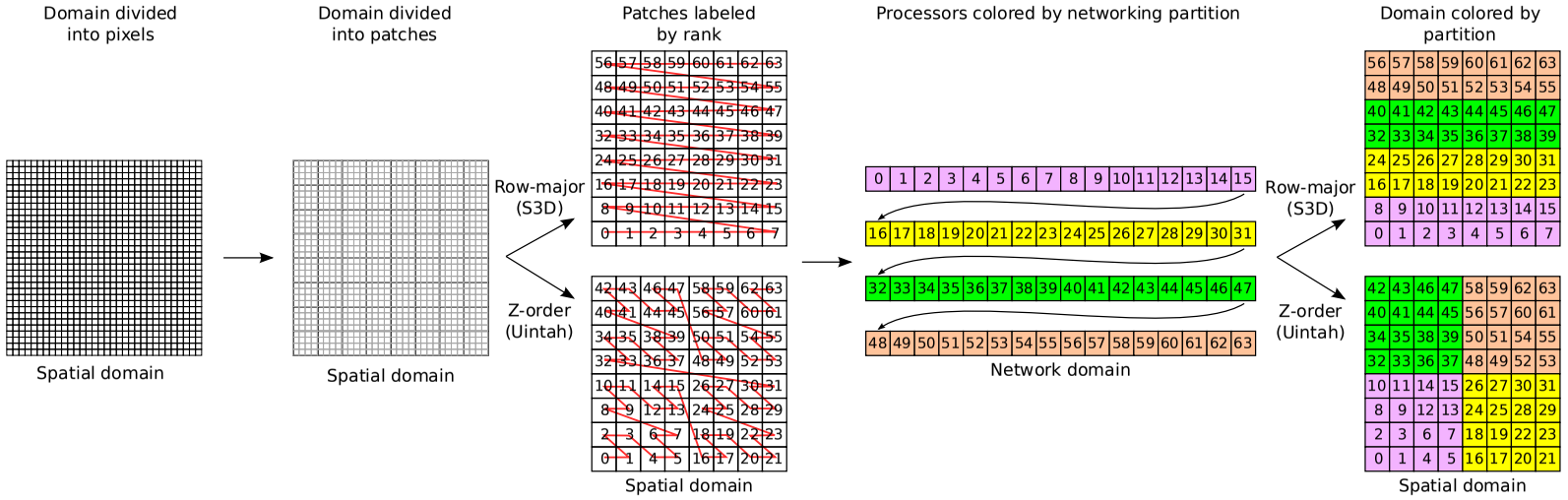


Figure 5.1: Simulation pipeline overview. The simulation domain is divided into elements and then into patches, which are assigned ranks. We use the ranks to partition the processes and each partition writes to its own files. The final panel shows the partition distribution in the simulation domain.

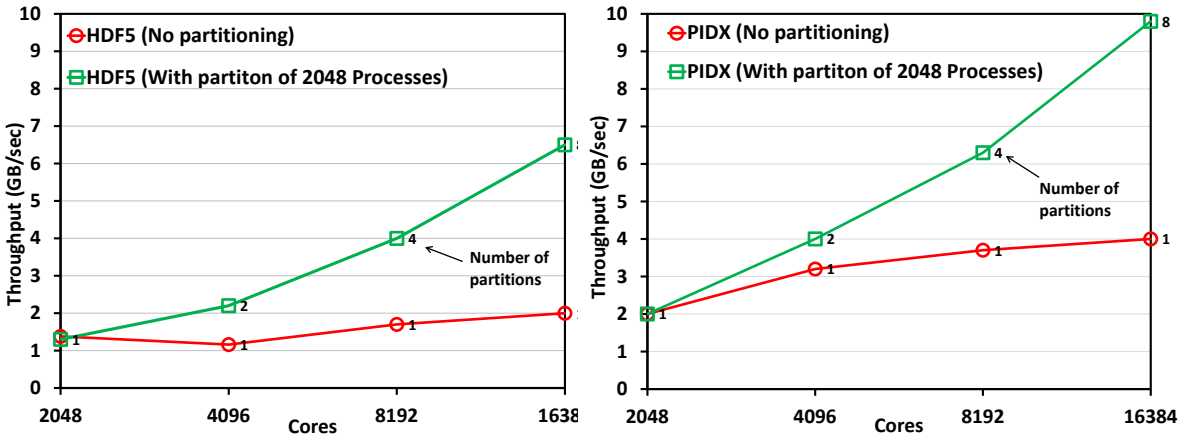


Figure 5.2: Weak scaling performance of HDF5 and PIDX with and without partitioning.

only within their partition during collective I/O, optimizing data dump preparation. Indeed, each partition writes its file entirely in parallel with the other partitions. Smaller partition size and more closely spaced processors in networking space result in less communication overhead in general. Partition size must be weighed against other factors, however, such as local versus global indexing, as well as the scheme used in partitioning.

5.1.1 Partition size

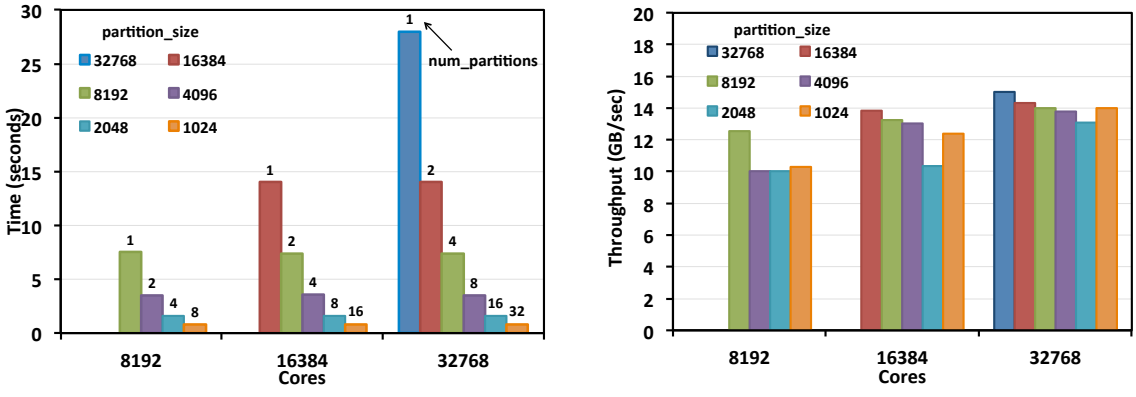
In the proposed method, the entire domain is first divided into several partitions. Then processes are grouped working on a partition using a local subcommunicator. Each of partitions is then written out as a separate IDX dataset (see Figure 5.1). The partition size is crucial as it directly influences the number of processes that need to be grouped together (more partitions imply fewer processes being grouped together and vice versa). The partition size significantly influences the way data get moved around during the aggregation phase. More partitions lead to a smaller networking space for the associated processes and also to a more localized nature of data communication. With PIDX, every new partition reduces the number of aggregators per process by half, hence significantly changing the data distribution pattern of the processes during the aggregation phase.

A set of scaling experiments was run to determine the ideal partition size. In the experiments, `partition_size` (or number of processes per partition) was varied at each core count. At each `core_count` = 8,192, 16,384, and 32,768 with partition sizes of `core_count` down to 1024 were tested. Note that

$$\texttt{num\_partitions} = \frac{\texttt{core\_count}}{\texttt{partition\_size}}$$

With this approach, the number of partitions will be 1 when `partition_size` equals `core_count` (corresponding to the base case where the entire domain gets assigned to one partition). In all our experiments, each process writes an 8 megabyte chunk of data. The aggregation and I/O times for the experiments are presented in Figure 5.3. Two direct inferences can be made from the aggregation time in Figure 5.3 (a). First, for all core counts, the aggregation time reduces by half as the partitions are doubled; for example, at 8192 cores, the aggregation time comes down from 7 seconds to 3.5 seconds as the `partition_size` goes down from 8192 to 4096. This behavior is observed because with every reduction of `partition_size`, the number of aggregators that every process has to send its data to also gets reduced by the same factor. Furthermore, with fewer target destinations, processes can send the data over the network in larger packets. The other key observation is that the aggregation time remains fixed for the same partition size at all scales. For example, it takes 1.6 seconds to perform aggregation in a 2,048 process partition at all core counts, which is critical for the aggregation phase to scale. What this observation signifies is that one can achieve scaling at even very high core counts by maintaining a small partition. The number of partitions will grow with core counts, but since the partition size remains fixed, aggregation time also remains roughly fixed.

The effect of I/O on varying the partition size can be seen in Figure 5.3 (b). The I/O timings do not have the kind of variation that was seen in the data aggregation phase, largely because the actual file I/O is bound by the physical I/O nodes of the machine to write the data. After data aggregation finishes, the aggregators write the data to the disk through the I/O nodes of the machine. On Mira, every 128 nodes (2,048 cores) share an I/O node. With a partition of 2,048 processes, there does not exist any contention over the



(a) (b)

Figure 5.3: Effect of partitioning on (a) data aggregation phase and (b) I/O phase.

I/O node if the ranks and the associated partitions are well aligned. However, when there is a partition with fewer or more than 2,048 processes, contention starts to creep in, with no partition having an exclusive access to an I/O node. Hence, for this reason, there is a dip in actual I/O time at 2,048 processes. The gain in I/O time with 2,048 core partitions over the 1,024 core partition is sufficient to amortize the gains seen in the data aggregation phase (see Figure 5.3 (a)). Also, it is expected that at much higher core counts, having partitions with fewer than 2,048 processes will cause a more prominent bottleneck. For all our scaling experiments, then, the partition size was fixed at 2,048 processes.

The result of combining aggregation and I/O can be seen in Figure 5.2 (b). Here a 4X improvement is observed in performance of the PIDX implementation using a partition size of 2,048 processes over the earlier implementation. In order to demonstrate the generic applicability of the approach, this idea of writing data in a partitioned index space was extended for the commonly used parallel HDF5 library. Similar to PIDX implementation, an extra layer was added on top of the existing I/O library, which partitioned processes into smaller domains with local communicators. Since the different partitions could be written only in the local index space, the offset of every process was adjusted so that the dataset could start from the origin. The HDF5 call was then invoked with the new communicator. The corresponding results are shown in Figure 5.2 (a). It can clearly be seen that with domain partitioning, the performance of the HDF5 library improved at all core counts. In this test, the resultant output files are written in the local index space. In Section 5.1.2, it is shown that with PIDX it is possible to write data from simulations in global index space as well.

5.1.2 Local and global indexing

Each partition writes to its own IDX file. There are two options for writing the IDX files - local or global indexing (see Figure 5.4). Using local indexing, each IDX file is separate and distinct from the others and has its own indexing. Local indexing achieves perfect storage efficiency, but this puts a hardship either on a postprocess or analysis software to merge the files. Global indexing is advantageous for downstream processing, but suffers from the same communication bottleneck as when writing the data with a single partition. We choose a hybrid indexing approach with a lightweight postprocess that takes advantage of the power of the IDX format.

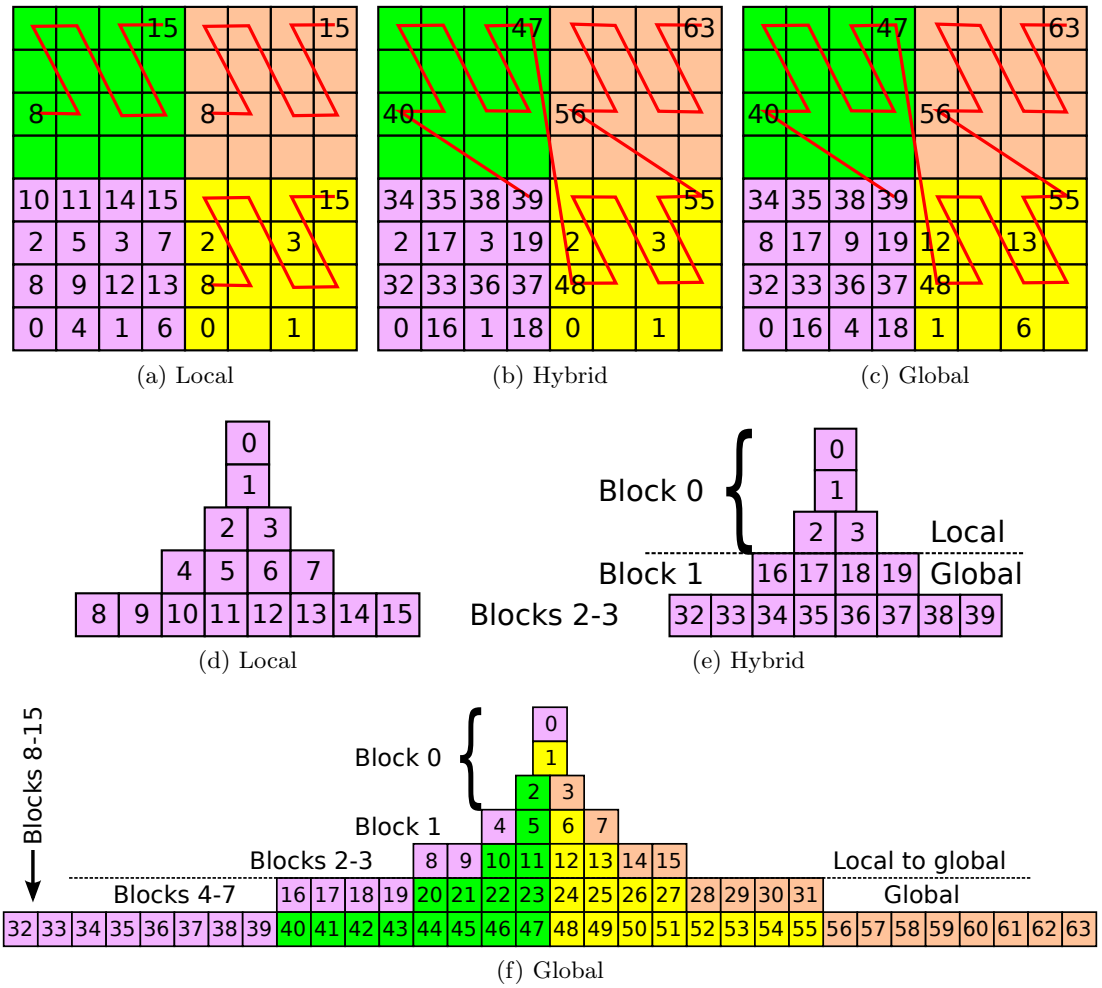


Figure 5.4: Local and global indexing with correspondences. Block size is 4. (a) Local indexing. Each partition indexes from 0 to 15. (b) Hybrid indexing. The first block of each partition is indexed locally. The remainder is indexed globally. (c) Global indexing. Each partition has interleaved indices with the other regions. (d) Local indexing shown hierarchically. (e) Hybrid indexing. The split between local and global occurs after the first block. (f) Global indexing of the entire domain. The only samples from the purple partition that belong to the mixed global blocks are 0, 4, 8, and 9, which form the first block of the local partition.

5.1.2.1 Hybrid indexing

Local indexing and global indexing have correspondences (see Figure 5.4), enabling a hybrid local/global index scheme. A full block is defined as one that contains only elements of a single partition. A block is called mixed if otherwise. Assume that each block has `block_size` elements, where `block_size` is a power of two. Assume also that all partitions are $2^j \times 2^j \times 2^j$ where $j \in \mathbb{N}$. As seen in the figure, if the elements of a partition are locally encoded, only the elements of the first block will be spread among multiple mixed blocks

in global indexing. For example, locally indexed elements $[0-3]$ are spread among the first three globally encoded blocks. Subsequent elements belong to their own full blocks.

Since exactly the first local block of each partition belongs to a mixed global block, the first `num_partitions` blocks using global indexing are mixed blocks. Our hybrid indexing scheme is as follows: each partition P_i encodes locally. The first block is written to disk as a locally indexed block. All remaining blocks $[B_1, B_2, \cdots, B_n]$ are written as globally indexed blocks by computing their global block number j from the local block number j'. Let $l \leftarrow 2^{\lfloor \log_2 j' \rfloor}$. Then

$$j = l * \texttt{num\_partitions} + (j' - l) \tag{5.1}$$

In Figure 5.4, block 1 in the hybrid representation maps to block 4 in the global representation using Equation (5.1). Similarly, blocks 2 and 3 map to blocks 8 and 9, respectively.

5.1.2.2 Merge and read

Writing IDX files using hybrid indexing results in files structured as shown in Figure 5.5. There is a set of mixed files, one for each partition, each of which begins with a single mixed, locally indexed block, followed by globally indexed blocks. The remaining files are labeled in the global index space. In order to treat these as a single IDX volume, the first block from each partition needs to merge, and the remaining blocks from the first file of each partition need to be reshuffled into their respective locations with respect to the global IDX volume. In practice, block reshuffling can be omitted by simply reading the desired blocks directly from their original files.

As seen in Figure 5.4, the first block of each partition is a mixed block that must be merged with its peers by moving each element to its correct location with respect to the global index space. This merge phase is done as a postprocess by visiting each mixed block,

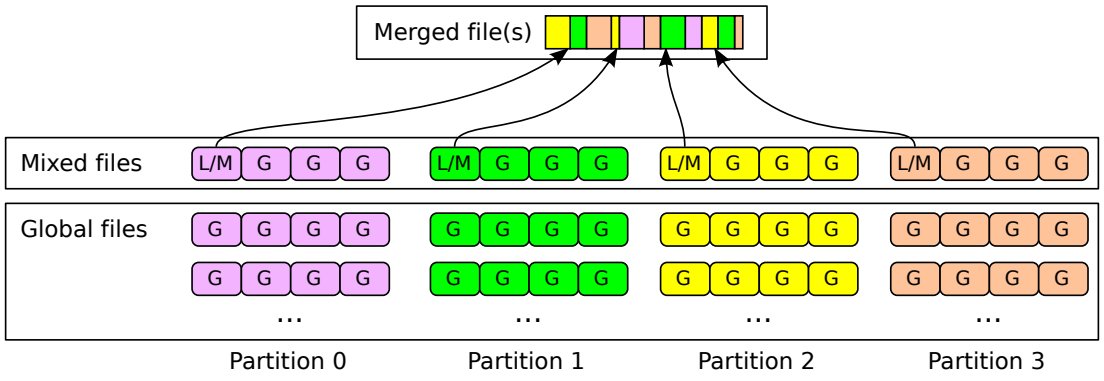


Figure 5.5: File structure for hybrid indexing. `blocks_per_file` = 4. M blocks are mixed blocks, L blocks are locally indexed, and G blocks are globally indexed.

computing the global index of each element, and writing the new set of merged files (see Figure 5.5). Figure 5.6 shows the times for this merge phase as the number of partitions increases. Since there are exactly `num_partitions` $*$ `block_size` elements to merge, which is a small subset of the total data, the merge step is extremely fast (e.g., our test on data from a 512K core simulation completed in 3.4 seconds).

After merging, the only blocks not in standard IDX placement are the remaining globally indexed blocks in the mixed files (see Figure 5.5). A small change to the IDX reader is required to access these blocks. Specifically, when the IDX file reader accesses block j, if $j <$ `num_partitions` or $j \geq$ `num_partitions` $*$ `blocks_per_file`, then it reads the block from its usual location in the global volume. Otherwise, it reads the block from the first file of its corresponding partition.

To determine which partition contains the requested block, let $hz = \lfloor \log_2 (j * \texttt{block\_size}) \rfloor$ be the HZ level of block j, and let $bpp = 2^{hz-1}/(\texttt{block\_size} * \texttt{num\_partitions})$ denote the number of blocks per partition at HZ level hz. The index of the partition is

$$p = \left\lfloor \frac{j - 2^{hz-1}/\texttt{block\_size}}{bpp} \right\rfloor$$

and the index of the specific block within partition p is

$$b = (j - 2^{hz-1}/\texttt{block\_size}) \ mod \ \texttt{bpp}$$

where *mod* is the modulus operator.

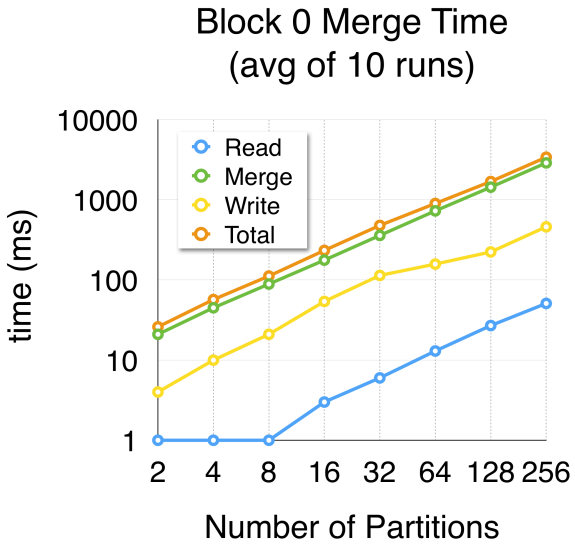


Figure 5.6: Merge step timings. For a simulation with 512K cores (256 partitions), block 0 merge time is 3.4 seconds per output variable.

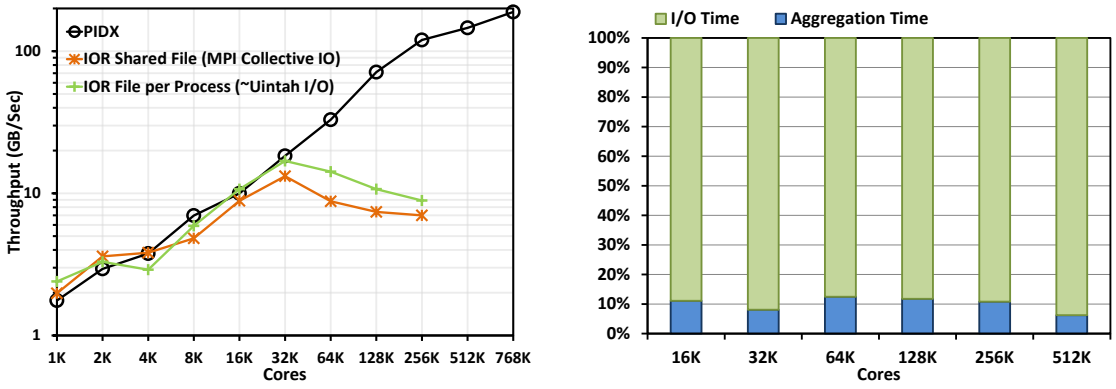
5.1.3 Partitioning evaluation

This section evaluates the performance of the domain partitioning approach using weak scaling. PIDX is integrated with the Uintah I/O simulator to directly dump data in the IDX format. In all runs, each process contributed a 32^3 block of double precision data (8 MB) consisting of 32 variables. The number of processes was varied from 1,024 to 524,288, thus varying the amount of data generated per time-step from 8 gigabytes to 6 terabytes. For all PIDX runs, the partition size was fixed at 2,048 cores, hence varying the number of partitions from 1 for 2,048 processes to 256 for 524,288 processes (see Table 5.1). IOR tests for each process count were alse executed in addition to partitioned PIDX runs. IOR is a general-purpose parallel I/O benchmark [44] that can be configured to perform both file-per-process as well as shared file I/O. In the experiments, for shared file I/O, all the processes wrote to a single file using MPI collective I/O.

Looking at the performance results in Figure 5.7 (a), it can be observed that PIDX scales well up to 524,288 processes, and that for all process counts, it performs better than IOR file-per-process and IOR shared file I/O. PIDX demonstrates almost linear scaling up to 262,144 cores whereas the performance of file-per-process and shared file I/O starts to decline after 32,768 cores. At 256K core counts, PIDX achieves an approximate speedup of 17X over shared file I/O and 14X speedup over file-per-process I/O. It achieves a max throughput of 190 gigabytes/second at 768K core counts. This behavior of PIDX can be attributed primarily to domain partitioning, as it breaks the global synchronization involved in data aggregation into multiple local synchronizations confined within each partition. PIDX performs data aggregation using one-sided RMA communication. One direct implication of performing I/O confined to a partition is in the reduction of the

Table 5.1: In each run, one partition is used for each 2K processes, and each partition writes 8 subfiles.

Total Processes	Number of Partitions	Total File Count
2048	1	8
4096	2	16
8192	4	32
16384	8	64
32768	16	128
65536	32	256
131072	64	512
262144	128	1042
524288	256	2048



(a) (b)

Figure 5.7: Weak scaling results using partitioning on Mira with Uintah I/O. (a) Results for weak scaling of PIDX I/O along with IOR shared file I/O and file-per-process benchmarks. The total data volume ranges from 8 GB to 6 TB. (b) The proportion of time taken by the aggregation phase and the I/O write phase at each scale.

number of aggregators to which each process has to send its data. In our study, as the number of partitions doubles with increasing core counts, the number of aggregators each process has to send its data to remains constant. As a result, we get a perfectly scalable data aggregation phase. This behavior can also be seen in Figure 5.7 (b), where the percent of time dedicated to data aggregation remains fixed at around 10% for all scales. Also, with localization during the data communication phase, the network outreach of each process becomes much smaller. This approach also distributes the MPI back-end overhead.

Another advantage of this approach is subfiling. Every partition gets written as a separate logical file since it is treated as a separate dataset. In the scaling study shown in Figure 5.7 (a), PIDX was configured to write 8 subfiles per partition, thereby varying the total number of files from 8 for 2,048 processes to 2,048 for 524,288 processes (see Table 5.1). Hence, with subfiling, PIDX is able to find a middle ground between shared file and file-per-process I/O, as it can be seen in Figure 5.7 (a) that too few files (with shared file) or too many files (file-per-process) both resulted in poor I/O performance. On Mira, I/O nodes manage the file metadata. Too many files per I/O node or too many I/O nodes sharing a file both lead to a bottleneck in metadata management.

5.2 Compression

Although the existing IDX format provides flexibility in accessing data at varying spatial resolutions, it does not support accessing data at varying numerical precision levels. This

section introduces this added flexibility of both storing and accessing the data at varying numerical precision by incorporating into IDX a fixed-bit rate floating-point compression scheme that also supports decompression at different bit rates. Using lossy compression, it is also possible to better utilize network and storage resources to reduce processing time.

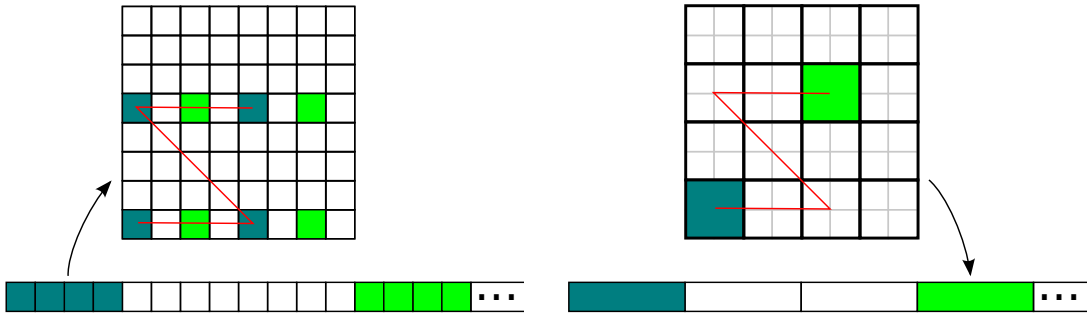
A traditional compression pipeline first encodes the data in some order such as the row-major order and then compresses it. Different ordering schemes result in different levels of compression efficiency due to the change in local data entropy. Entropy in data compression denotes the randomness of the data that are fed to the compression algorithm. The greater the entropy, the lower the compression ratio. If the data are encoded using row-major order, the locality in the x direction is excellent but entropy in the y and greater dimensions suffers. Z-ordering shows good locality in all dimensions. The HZ-ordering of the IDX file format also shows good locality, but only at high resolution. At low resolution, spatial coherence is poor because nearby samples at low HZ levels are spatially far from one another.

5.2.1 Chunking for compression

To achieve a low compression error with HZ-ordering, a chunking step is introduced where $n \times n \times n$ blocks of elements are "chunked" together and then compressed. These compressed chunks become the new atomic elements that are reordered according to their HZ indices and written to the IDX file (see Figure 5.8). This process can be thought of as reducing the number of levels in the HZ hierarchy. Since data in IDX format is always written and read in blocks, as long as the size of each chunk is smaller than the size of an IDX block, incorporating compressed chunks into the IDX format does not incur an I/O penalty. In practice, a chunk size of $4 \times 4 \times 4 = 64$ samples is used, whereas an IDX block typically contains hundreds or thousands of samples.

For compression we use zfp [66], a state-of-the-art, fixed-rate, lossy data compression library for floating-point arrays. As demonstrated in [66], many times, data can be encoded at a much lower bit rate than the typical bit rates of 32 or 64 without significant or noticeable quality loss in visualization and analysis. Using a lossy compression scheme makes it possible to have full control over the trade-off between data accuracy and the cost of data transfer and storage, and a fixed-rate format simplifies the handling of compressed data. Since the atomic element size is fixed, no further modifications (beside chunking) need to be introduced to the existing IDX format.

Unlike other floating-point compression schemes, zfp is optimized for compression of volumetric data stored in regular grids. It exploits spatial coherency in data by internally



(a) Encode, compress (b) Chunk, compress, encode

Figure 5.8: Chunking for compression. (a) Typical compression schemes compress on reordered elements. (b) Chunks of elements are compressed, and then they are treated as atomic elements in the IDX format.

grouping voxels of a three-dimensional grid into $4 \times 4 \times 4$ chunks and compressing each chunk independently. zfp compares favorably to other techniques in terms of compression errors, while having high throughput for both in-memory compression and decompression (280 MB/s and 400 MB/s, respectively, as reported in [66]). This level of throughput is sufficient in our case for the compression cost to be hidden by the networking and I/O costs.

Note that although chunking is performed before passing the data to the compressor, using a spatially-aware compression algorithm allows us to be flexible in our chunking scheme. On the other hand, since chunking is done independently of compression, data locality is taken care of, allowing any high-speed floating-point compression library to be used in place of zfp.

5.2.2 Compression evaluation

This section presents the impacts that compression has on PIDX, with regard to both performance and data integrity. Figure 5.9 shows weak scaling results performed at different bit rates using a $4 \times 4 \times 4$ chunk size. The S3D I/O simulator is used, where every process dumps 32^3 double precision data consisting of 16 fields. Varying the number of processes from 1K to 32K generated 4 GB to 128 GB of uncompressed data. The amount of data generated at each core count gets reduced in proportion to the reduction in bit rate. For example, at 32K cores, bit rates of 32, 16, 8, 4, and 1, respectively, produce 64, 32, 16, 4, and 1 GB of data. The reported time includes both chunking and compression.

Using compressed data significantly reduces the total processing time of PIDX. PIDX automatically increases the block size by a factor of $\log_2$ of the bit rate, reducing the number of files being created (see Table 5.2). There is almost a one half drop in time while going from uncompressed to a bit rate of 32. The linear decrease in total processing time as the bit

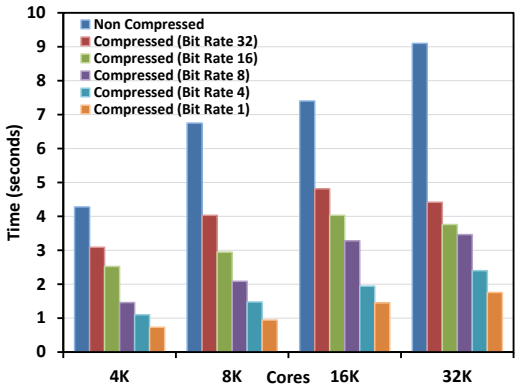


Figure 5.9: Compression experiments with S3D I/O. Total time for weak scaling while varying the compression bit rate from 32 to 1.

Table 5.2: Compression experiments with S3D I/O. Number of partitions, bit rate, and corresponding file counts.

Total Processes	Number of Partitions	Bit Rate	File Count
8192	4	8	4
8192	4	32	16
16384	8	8	8
16384	8	32	32
32768	16	8	16
32768	16	32	64

rate decreases implies the cost of compression is well compensated by the increase network and I/O bandwidths. Afterwards, a more gradual decrease in time is observed. Performance is mainly dominated by file access time instead of the actual I/O time. Overall, performance improvement between 4× to 8× is observed.

zfp is shown to have a very small error at low bit rates [66]; here we show that applying compression to chunks rather than HZ-encoded samples is important to retain low error rates. Figure 5.10 shows different renderings of a slice from a combustion simulation dataset. The slice shows a two-dimensional projection of the reaction of oxygen (O_2). This figure demonstrates the efficacy of the IDX format at accessing data at both low resolutions and low bit rates (compare Figure 5.10 (f) when the data are read at low resolution but full precision and Figure 5.10 (b) where the data are read at full resolution but with a reduced bit rate of 1).

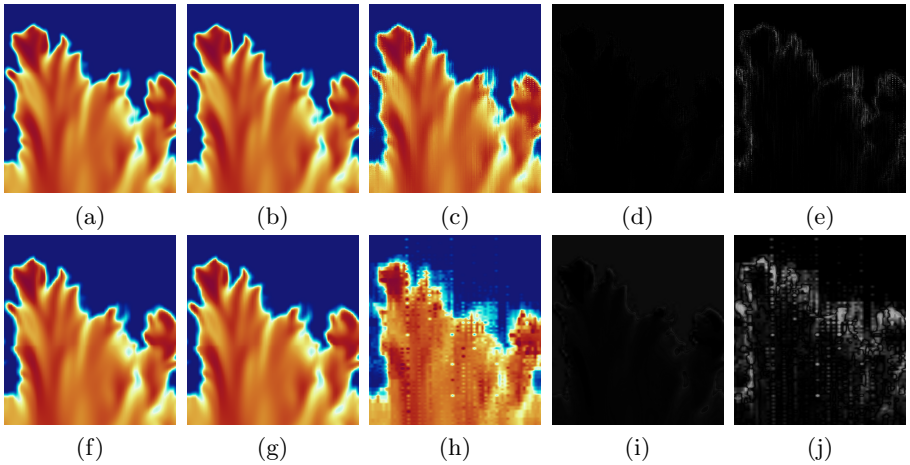
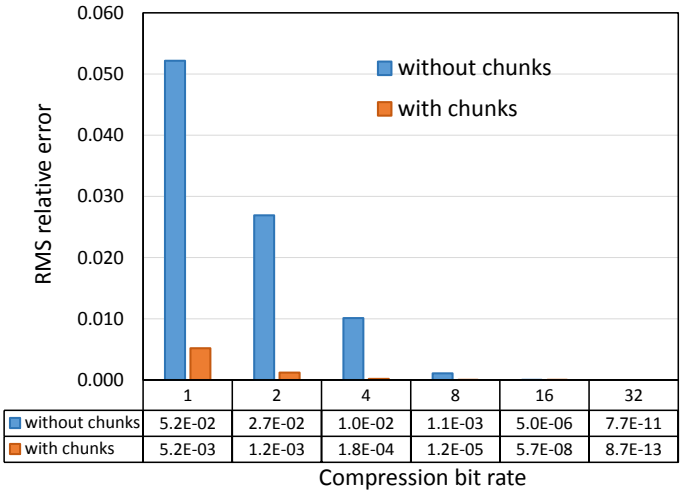


Figure 5.10: Effects of chunking and fixed rate lossy compression, at two resolutions (original and 1/64 the original resolution). All the figures correspond to a slice of the O_2 field of an S3D simulation. Every compression is done with a fixed bit rate of 1 bit per double. (a) No compression. (b) Compression with chunking. (c) Compression without chunking. (d) Difference image between (a) and (b). (e) Difference image between (a) and (c). (f) to (j): Low-resolution results (1/64 the native resolution), in the same order as (a) to (e).

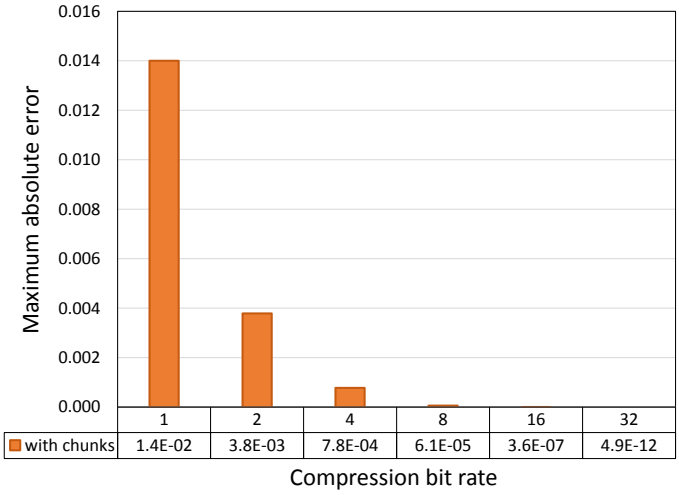
As also can be seen from the same figure, with chunking, even when using a compression bit rate of 1, there is no visible distinction when compared with data stored at full precision data. On the other hand, doing compression without chunking produces very clear artifacts because nearby, low-resolution samples in the HZ-encoded domain are far away from one another in the spatial domain. This effect can be seen clearly in Figure 5.10 (j). The errors are high not only in regions of high entropy (along the edge of the flame), but because of the hierarchical nature of IDX, the errors form a grid-like pattern where there is low entropy as well.

Figure 5.11 (a) compares the quantitative errors of compression with and without chunking at different bit rates. We compute the root mean square (RMS) of relative voxel error in a volume of size $N = 512 \times 256 \times 256$ of the O_2 field in a combustion dataset, using the formula $err_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\frac{v_i - v_i\prime}{v_i}\right)^2}$ where v_i is an original voxel and $v_i\prime$ is the corresponding decompressed voxel. As expected, the average error decreases exponentially as the bit rate increases. Also, compression without chunking produces errors that are from one to two orders of magnitude larger compared to those from compression with chunking. To complement the RMS errors, we also present the maximum of absolute errors for the same volume, at all bit rates in Figure 5.11 (b). The maximum of absolute errors is defined as $err_{max} = \max_{i=1}^{N} |v_i - v_i\prime|$. Absolute error is useful in choosing the most appropriate bit rate that balances data integrity and simulation performance.



	1	2	4	8	16	32
without chunks	5.2E-02	2.7E-02	1.0E-02	1.1E-03	5.0E-06	7.7E-11
with chunks	5.2E-03	1.2E-03	1.8E-04	1.2E-05	5.7E-08	8.7E-13

(a)



	1	2	4	8	16	32
with chunks	1.4E-02	3.8E-03	7.8E-04	6.1E-05	3.6E-07	4.9E-12

(b)

Figure 5.11: Error analysis of the fixed bit-rate compression scheme. (a) Comparison of RMS of relative errors between data written with and without chunks, computed on the O_2 field of an S3D combustion simulation dataset. (b) Maximum of absolute errors over the same volume of O_2, with chunks.

5.3 Conclusion

This chapter has demonstrated scaling of I/O to 768K cores by partitioning the network space. The efficacy of the partitioning scheme was demonstrated using both PIDX and HDF5. The nice mathematical properties of the HZ-ordering used by the IDX file format leads to our hybrid local/global indexing scheme, enabling fast I/O for both writes and reads. The indexing scheme lends itself perfectly to partitioned I/O. It has been shown that for IDX, a preindexing compression approach using chunking yields error rates that are orders of magnitude better than compressing after sample indexing.

CHAPTER 6

FAST READS OF LARGE-SCALE DATASETS

Massively parallel scientific simulations often generate large datasets that can range in size up to many terabytes. Reading this data are required for several reasons. Due to crashes, code modifications, or limited job time, simulations must often be restarted from an intermediate time-step. Restarts require reading of the entire saved state of the simulation at full-resolution for a given time. Similar to simulation restarts, comprehensive postprocessing analysis also requires reading of full-resolution data, although only a subset of variables may be necessary. The other major reasons to read data are for user-directed analysis and visualization. Unlike restarts and postprocessing, these tasks can often be performed using lower-resolution data with less spatial extent. For example, coarse-resolution data can be used to compute an approximation of comprehensive analysis results. Similarly, the limited size of display devices permits lower-resolution data to be shown with no perceptible difference in visual quality. Finally, spatial extent can be restricted since only data within a visible region needs to be loaded. Thus, two general classes of data reading are identified [13]:

1. full-resolution reads of an entire dataset as required for simulation restarts, and
2. partial-resolution reads of a subset of the data suitable for visualization and cursory analysis tasks.

Complete reads are generally performed using parallel systems whereas partial reads may be done in serial or parallel. Chapters 4 and 5 focused on writes-oriented I/O transformations. This chapter details transformations focusing reads, both serial and parallel.

6.1 View-dependent multiresolution serial reads

It is often desirable to utilize low-cost, low-power hardware to visualize massive simulation data as well as to perform cursory or user-directed analysis. For example, a portable device could be used to monitor the progress of a simulation. Utilizing a hierarchical

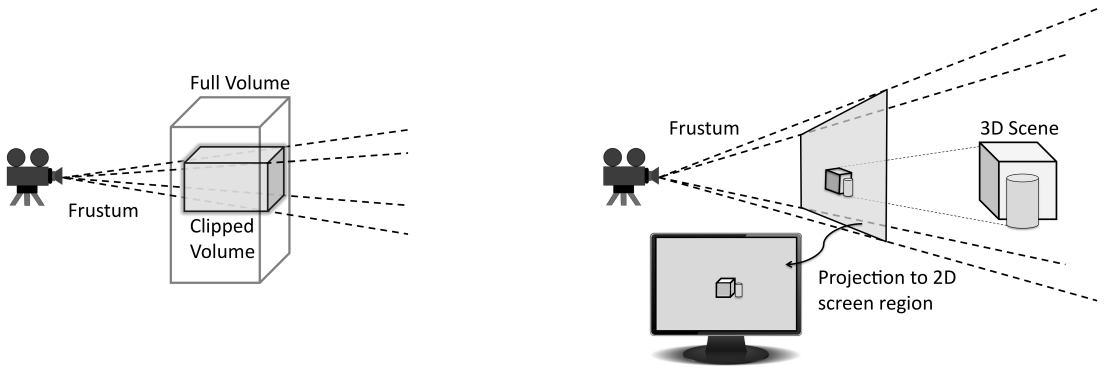
multiresolution data format enables view-dependent determination of the spatial extent and resolution of data to be loaded, permitting less data to be requested and facilitating interactive navigation of arbitrarily large datasets. This section explains the mechanism of view-dependent data loading and the implementation of this technique in the VisIt visualization application.

6.1.1 View-dependent data loading

All visualization systems incorporate some type of two-dimensional or three-dimensional camera model in order to transform data into a two-dimensional image on a screen. The camera model can be distilled down to a matrix multiplication applied to the individual components of the data in order for them to appear in the desired location when projected onto the two-dimensional viewing plane. A comprehensive description of the viewing pipeline is outside the scope of this work, but the process is explained in most introductory computer graphics texts such as [67, 68, 69]. The matrix resulting from the composition of model, viewing, and projection transformations, as well as the near and far clipping planes, creates a viewing "frustum," a parallelepiped oriented in the world space coordinate system. The frustum is used to crop data outside the view of the virtual camera (called "clipping"), and the ratio of the projected size of a data unit compared to a screen pixel can be used to determine the optimal resolution of data to be loaded.

View-dependent data loading begins by applying the frustum clip planes to the volume of data being visualized. Clipping the bounds of the data volume with the viewing frustum is how we determine the spatial extent of the data to be loaded. See Figure 6.1 (a). Next, we use the relative size of a display pixel compared to the size of a unit of the data volume projected to the screen in order to determine the minimum resolution necessary to visualize the data without artifacts. See Figure 6.1 (b). This technique is similar to the ubiquitous MIP-mapping technique of [70] used to decide what level of a given texture should be sampled based on its projection to screen space. Using the extent of the volume and relying on the fact that every HZ level reduces the resolution by half in one direction, we can determine the minimum level necessary for a unit of the data volume to cover approximately one pixel on the screen. Loading higher resolution data than this is a waste of time and memory because multiple data units are simply averaged over the same pixel, and loading a lower-resolution will result in an overly coarse or "jaggy" image.

The combination of these two techniques allows for interactive exploration of any size data using even modest hardware resources. We describe the mechanism by which we have incorporated view-dependent rendering into the VisIt visualization system.



(a) (b)

Figure 6.1: Example images showing the mechanism of view-dependent data reading. (a) Shows how the spatial extent of the data is clipped by the camera viewing frustum. Only data within the frustum must be loaded. (b) Shows the projection of a three-dimensional scene onto the two-dimensional camera viewing plane, which is copied to the display. The ratio of the area of a screen pixel to the projected area of a three-dimensional volume unit is used to determine the resolution (HZ) level that will be loaded. For example, if 4 volume units project to one pixel, 1/4 resolution data is sufficient to convey the scene.

6.1.2 Implementation in VisIt

Loading an entire large data volume is frequently excessive for simple visualization or cursory analysis and requires significant compute resources. For summary analysis and visualization, or to load extremely large datasets using more modest hardware, a coarse approximation of the data can be sufficient. To this end, the IDX format was incorporated as both a serial and parallel database reader plugin for VisIt. The implementation of the serial IDX reader in VisIt facilitates loading coarse levels of the data by selecting a subvolume and resolution sufficient to accommodate a given viewing frustum.

6.1.3 Performance evaluation

Interactivity is the overall performance goal of using view-dependent data loading. The key idea is to load as little data as possible at one time while still providing sufficient coverage of the viewing region. Efficient data loading is an important component of engineering an interactive application, but additional support is required at every level. For example, it is important to frequently check for user input even during a complicated analysis or visualization operation or while reading data.

Figure 6.2 shows the results of our experiments on Tukey. The times required by VisIt for loading a Uintah Data Archive (UDA) file versus the same simulation stored in the IDX format are plotted using increasing numbers of cores. In order to compare view-dependent

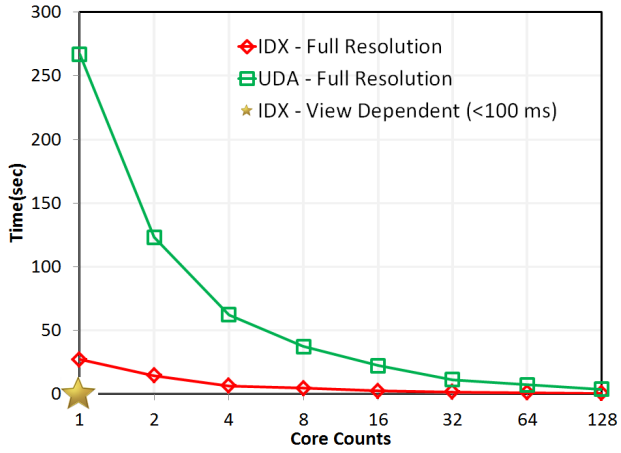


Figure 6.2: VisIt load times on Tukey [71] for a Uintah Data Archive (UDA) file versus the same simulation stored in the IDX format for increasing the numbers of cores. The star in the bottom left indicates the time for loading the data using the serial view-dependent method.

loading of multiresolution data to the existing practice of loading full-resolution data using a visualization cluster, the point noted with a star at the bottom of the figure shows the average time for loading the data using serial view-dependent visualization. All load times were gathered using the '-timings' argument to VisIt.

The dataset shown was computed using 9920 processors and consists of 19 variables on a 1323×335×290 domain over 171 time-steps, totaling approximately 4 TiB. The data shown in the figure are for loading a single variable of the dataset. Note that the differences observed in loading UDA versus IDX data at full-resolution are due to the nature of the two file formats. IDX data are stored in a cache-oblivious layout, whereas data in UDA data are stored in naïve row-major order using one file per process. Because the UDA format uses so many files, additional access time (open/close/seek) is incurred compared to the IDX format, which uses fewer files to store the same data. UDA is comparable to one file per process I/O, so as more cores are used the overhead incurred in accessing files is reduced because the number of files that are accessed per process decreases. Most importantly, as can be seen in the chart, view-dependent data reading is significantly faster than loading full-resolution data using any number of cores.

6.2 Parallel reads at full resolution

The previous section discussed the utility of multiresolution reads in serial mode. However, full-resolution parallel reads are still important for simulation restarts and comprehensive postprocessing analysis. This section describes transformations for parallel reads

implementation within the framework of the PIDX I/O library. It is then followed by in-depth evaluation showing both strong and weak scaling results. This section also compares PIDX performance with two common distributed parallel file formats, MPI collective I/O [16] and one file per process S3D I/O.

Parallel IDX reads involve translation from the hierarchical progressive data order of the IDX format to the conventional multidimensional data layout suitable for use by application processes. We use the experience with the PIDX writer to design the corresponding reader [10]. Firstly, a naïve implementation is described that uses one I/O phase to fetch data. Next, this strategy is improved by adding an additional data aggregation phase to achieve scalable performance by reducing noncontiguous data access.

For the one-phase parallel reader, every process first reads the data at each HZ level, and then correctly orders the data for application usage. This strategy is complementary to the approach followed in one-phase writes, where every process first calculates the HZ order and corresponding HZ level for its subvolume, and then uses independent MPI-I/O write operations to store each level. For both reading and writing, this method entails a high degree of noncontiguous data access of the file system, dramatically impeding scalability [7] (see Figure 6.3 (a)).

To improve upon the naïve one-phase approach, a two-phase I/O algorithm was devised to mitigate the problems caused by large numbers of small-sized disk accesses. With this approach, only a few select processes assigned as aggregators access the file system, making

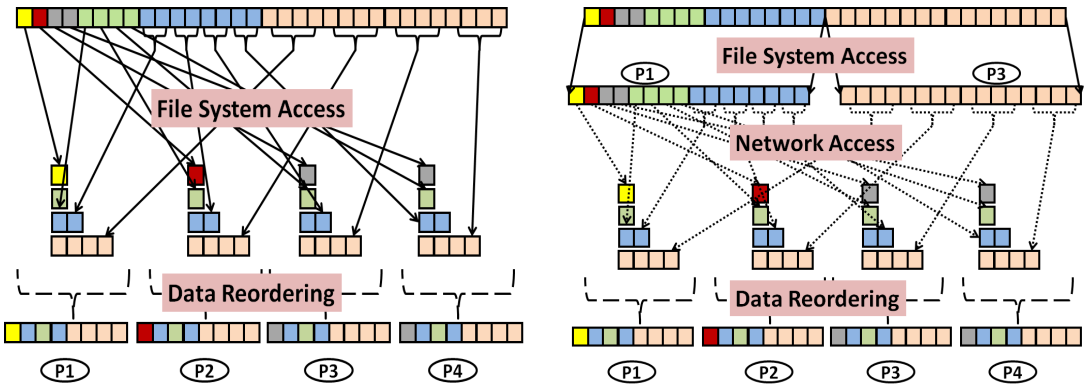


Figure 6.3: Reading IDX: data are read into a hierarchy of z-buffers and then reordered to the conventional multidimensional application layout. (a) The one-phase approach that requires many small noncontiguous disk accesses to load data into the z-buffer hierarchy of each process. (b) The improved two-phase method. First, large contiguous blocks of data are read from disk by a few aggregators. Next, the data are loaded via the network into the z-buffer hierarchies of each process.

large-sized contiguous read requests. Once the data are read by the aggregators, it is next transferred to every process over the network for reordering. As with PIDX writes, one-sided MPI communication is used for this purpose: after the aggregators finish reading data from the file system, all the processes use MPI_get() to gather data directly from the aggregators' remote buffers. Figure 6.3 (b) illustrates the two-phase I/O approach.

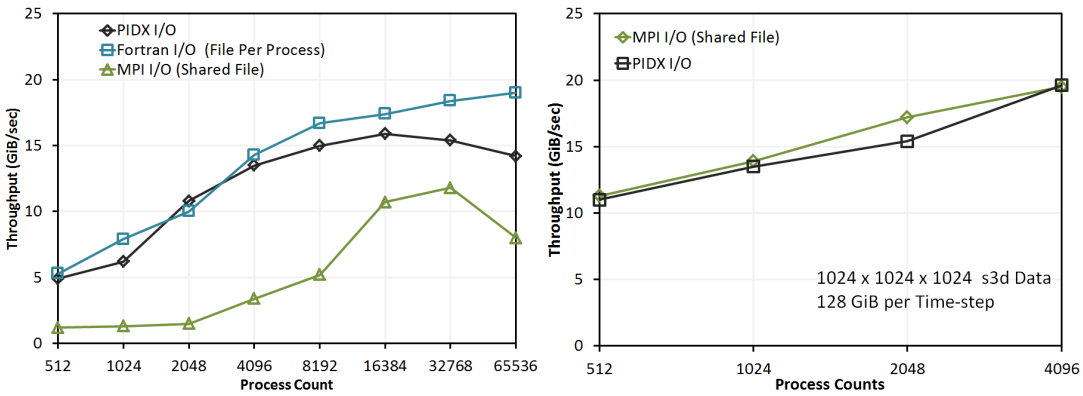
Two sets of experiments are performed: weak scaling and strong scaling. Weak scaling increases the data size proportional to the number of processors. It is a useful measurement for determining the efficiency of reading data for simulation restarts that typically require reading a complete dataset using many cores. Strong scaling increases the number of processors while maintaining the same problem size. It is a useful metric to determine how much a particular task can be accelerated by the addition of computational resources. Strong scaling is an indicator of parallel visualization and analysis performance because these tasks are often performed using fewer cores than the original simulation.

6.2.1 Weak scaling (simulation restarts)

We use the S3D I/O simulator to benchmark the performance of parallel reads. We compared PIDX performance with that of both the Fortran I/O and MPI collective I/O modules in S3D. With Fortran I/O, data are present in their native format organized into files equal to the number of processes used to create them. PIDX determines the number of files based on the size of the dataset rather than on the number of processes [7]. In the case of MPI I/O, data exist in a single, shared file. As opposed to Fortran I/O, both PIDX I/O and MPI I/O have an extensive data communication layer as part of the collective I/O phase. Default file system striping parameters were used for Fortran I/O, whereas for MPI I/O and PIDX I/O, the Lustre striping was increased to span all 96 OSTs available on that system.

In order to benchmark read performance, we first used the S3D simulator to create datasets of the corresponding format at each process count. We then invoked the S3D postprocessing module to enable parallel reads and conducted our experiments. For each run, S3D reads data corresponding to 25 time-steps. Every process reads a 32^3 block of double-precision data (4 MiB) consisting of four variables: pressure, temperature, velocity (3 samples), and species (11 samples). We varied the number of processes from 512 to $65,536$, thus varying the amount of data read per time-step from 4 MiB to 256 GiB.

Considering the performance results in Figure 6.4 (a), we observe that PIDX scales well up to $32,768$ processes, and that for all process counts, it performs better than MPI I/O. We also observe that both PIDX and MPI I/O do not perform as well compared to Fortran



(a) (b)

Figure 6.4: Results using Edison. (a) Weak scaling of different I/O mechanisms including PIDX, Fortran I/O, and MPI I/O reading datasets with a block size of 32^3. (b) Strong scaling for PIDX and MPI I/O for reading datasets of dimension 1024^3.

I/O because unlike PIDX and MPI I/O, Fortran I/O does not require any data exchanges across the network. The performance pattern of PIDX is mainly due to its aggregation phase, which finds a middle ground between the single-shared-file MPI I/O and one file per process I/O.

One new observation that we make for parallel I/O is the decline in scalability for both MPI I/O and PIDX at higher core counts, whereas Fortran I/O shows continued scaling. This decline is in contrast to parallel writes for which continued positive scaling results have been demonstrated for all I/O formats [12]. One possible explanation for this behavior is that network communication involved in the collective I/O data scattering phase may play a role in this effect. If this were the case, we note that Fortran I/O would not be affected since each process is independent of the network.

6.2.2 Strong scaling (postprocessing analysis)

The S3D I/O simulator was used to compare the strong scaling performance of PIDX with MPI I/O on Edison. The S3D simulator was first used to create input datasets of the corresponding format using 32768 cores with a per process domain size of 32^3. The S3D postprocessing module was then invoked to enable parallel reads. For each run, S3D reads data corresponding to 25 time-steps. Because the S3D simulator does not make available the capability to read multiple files per process, we were unable to acquire strong scaling numbers for Fortran I/O. Our experiments used a domain of dimension 1024^3. The number of processes was then varied from 512 to 4096, for which the block size per process ranged

from 128^3 (512 processes) down to 64^3 (4096 processes). The results of this study are shown in Figure 6.4 (b). We observe that PIDX and MPI I/O perform similarly at all core counts.

6.3 Parallel reads at varying resolution

For our final experiments, we explore parallel reading of multiresolution data. Providing parallel multiresolution data loading confers the advantage of faster coarse resolution data access while allowing for greater processing power and total aggregate memory available on larger clusters.

Parallel multiresolution reads can be useful for visualization and postprocessing analysis. As with serial multiresolution data reading, relatively small compute clusters can be utilized to perform analysis or to visualize much larger datasets than was previously possible. In addition, dedicated visualization clusters such as Tukey may have associated GPU arrays or be connected to large displays such as those used for power walls or immersive caves. These clusters may be utilized to produce ultra-high-resolution imagery using rendering techniques such as raytracing or to perform postprocessing analysis using a mixture of GPU and CPU resources.

One of the major challenges for performing multiresolution reads in parallel is ensuring stable scalable performance by limiting data reads. We made two significant changes to extend parallel full-resolution reads to support multiresolution capabilities. First, based on the desired level of resolution, we added the capability to directly select the appropriate HZ levels. Second, we made the aggregation phase self-balancing by maintaining the number of aggregators for varying levels of resolution. By balancing the load across aggregators, we ensure a uniform reduction of workload.

Due to limitations in aggregate memory, loading data at full resolution would not be feasible for some datasets, but with multiresolution support, data can be loaded at coarser resolutions, enabling approximate analysis or visualization for extremely large datasets.

We conducted these experiments using Edison, with stripe setting similar to our parallel full-resolution experiments. We first used the S3D simulator to create input datasets of global resolution 1024^3 and 2048^3. For the 1024^3 volume, we evaluate the performance of multiresolution reads by first reading the data at full resolution followed by reading the data at partial resolution. We decrease the resolution by half for each successive test down to 1/8 of the original volume size. For comparison, we also show the corresponding result for loading the full-resolution volume using MPI I/O (see Figure 6.5 (a)). In accordance with our strong scaling results, we observe that reading the full-resolution data requires approximately the same amount of time for both PIDX and MPI I/O. For multiresolution

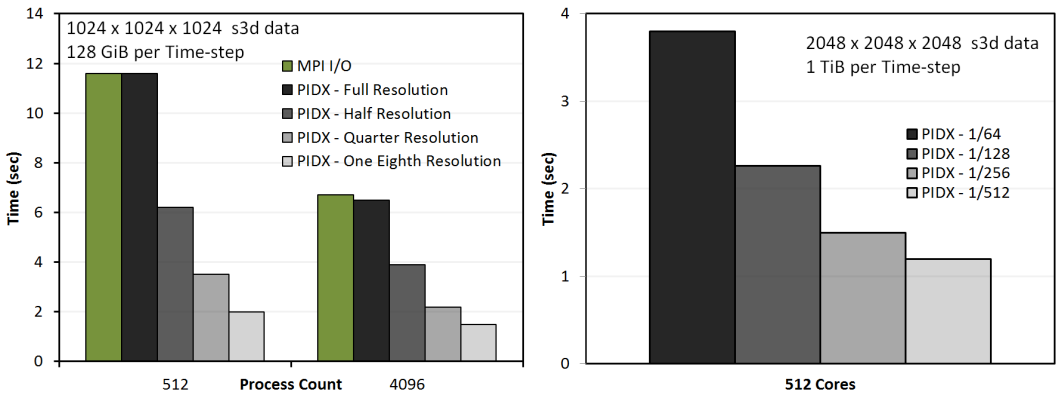


Figure 6.5: Results using Edison. (a) Timings for reading full-resolution 1024^3 volume data using PIDX and MPI I/O as well as timing for partial-resolution reads using PIDX. (b) Timings for reading a 2048^3 volume data at partial resolutions using PIDX, simulating an environment in which it is not possible or desirable to read the full-resolution data.

reads, we see almost perfect scaling when the resolution is reduced by half, but as the resolution continues to decrease, the efficiency begins to deteriorate due to the reduced workload of individual aggregators as the requested data size decreases.

We use the 1 TiB per time-step 2048^3 volume to simulate the situation in which the amount of data to load might be larger than the aggregate system memory. For this particular case, we begin reading data at 1/64 resolution and decrease by half for each run down to 1/512. The results for this experiment can be see in Figure 6.5 (b). From the figure, we observe that the read time using 512 cores is less than 4 seconds, which is fast enough to be used for cursory visualization or to compute approximate analysis on an otherwise unwieldy dataset. Also, as observed with the 1024^3 case, we notice a nearly perfect scaling for the first decrease in resolution but successive reductions are less optimal.

As conclusion, in this chapter, we identified two broad classes of data reading: full resolution reads for simulation restarts or postprocessing analysis and partial resolution reads of spatial subsets suitable for cursory analysis and visualization. Also, the technique of view-dependent reading was incorporated into the VisIt visualization framework.

PART III

MODELING I/O PERFORMANCE

CHAPTER 7

PERFORMANCE TUNING WITH CHARACTERIZATION AND MODELING

The performance of parallel I/O libraries such as PIDX is influenced by characteristics of the target machine, characteristics of the data being accessed, and the value of tunable algorithmic parameters. HPC systems vary widely in terms of network topology, memory, and processors. They may also use different file systems and storage hardware that behave differently in response to I/O tuning parameters. Although many of these properties can be empirically determined through characterization studies, it is still difficult to translate them into an optimal set of tuning parameters for a sophisticated I/O library. Hence, we propose the construction of a performance model of the system to aid in exploration of the parameter space. Moreover, the model can be used to suggest promising parameter configurations and auto-tune the system for higher levels of performance.

This chapter presents a detailed characterization study of the PIDX collective I/O algorithm including both network aggregation and file system I/O. The goal is to understand how performance is affected by combinations of fixed input parameters (system and data characteristics) and tunable parameters. The study then proceeds with modeling of behavior of the I/O library using machine learning techniques. Earlier work in high-performance research has proposed analytical models, heuristic models, and trial-and-error approaches. All these methods have known limitations [3] and do not generalize well to a wide variety of settings. Modeling techniques based on machine learning overcome these system limitations and build a knowledge-based model that is independent of the specific hardware, underlying file system, or custom library used. Based on the flexibility and independence of a variety of constraints, machine learning techniques have achieved tremendous success in extracting complex relationships from the training data itself.

Performance models are built using regression analysis on datasets collected during the characterization study. The regression models predict performance and identify optimal tuning parameters for a given scenario. The models have been trained on data obtained

from experiments conducted over a small number of cores. The models are first validated on (training) datasets from low core count. Then these models (and a small number of samples from simulations on high core count) are used for throughput prediction on test datasets from the high core count regime. The goal is to show that such a model would be useful for *approximately* predicting the behavior of a system in higher core count scenarios where brute-force sensitivity studies would be both costly and resource intensive. The samples from the high core count regime are obtained by augmenting the regression model with a sampling technique. Consequently, a new model is obtained that adaptively improves itself over multiple simulation timesteps, thus auto-tuning the system parameters to achieve higher performance over time.

The following key findings are reported from the characterization and modeling study. First, it was found that the Hopper network is more sensitive than is Intrepid to variations in the quantity and size of network messages. Second, owing to differences in ways job partitions are allocated, the two architectures exhibited different network scaling behaviors. Third, Hopper (Lustre) is more optimized to a unique file per process I/O approach than is Intrepid (GPFS), whose I/O is optimized for fewer shared files. Moreover, for varying data load, Hopper showed variation in performance pattern with similar parameter configuration. In the modeling study, small validation errors were observed for throughput prediction on a low number of cores and comparatively higher error on high core count experiments. Overall, it as found out that Hopper was more difficult to model (particularly at high core counts) compared with Intrepid.

7.1 Case study using PIDX

PIDX is used as the case study for characterization and modeling in this work. Section 7.4.5.3 discusses how the techniques can be applied to other I/O software stacks. PIDX deploys a three-phase I/O approach to write data in parallel in IDX file format. The first phase involved restructuring of simulation data into large blocks (powers of two) while preserving the original multidimensional format. The restructuring phase facilitates optimized HZ ordering, followed by efficient I/O aggregation (second phase) and ending with actual disk-level I/O writes (third phase). By adopting this three-phase I/O, PIDX is able to mitigate the shortcomings of both small disk accesses as well as unaligned and discontinuous memory access. Figure 7.1 illustrates the three phases of PIDX.

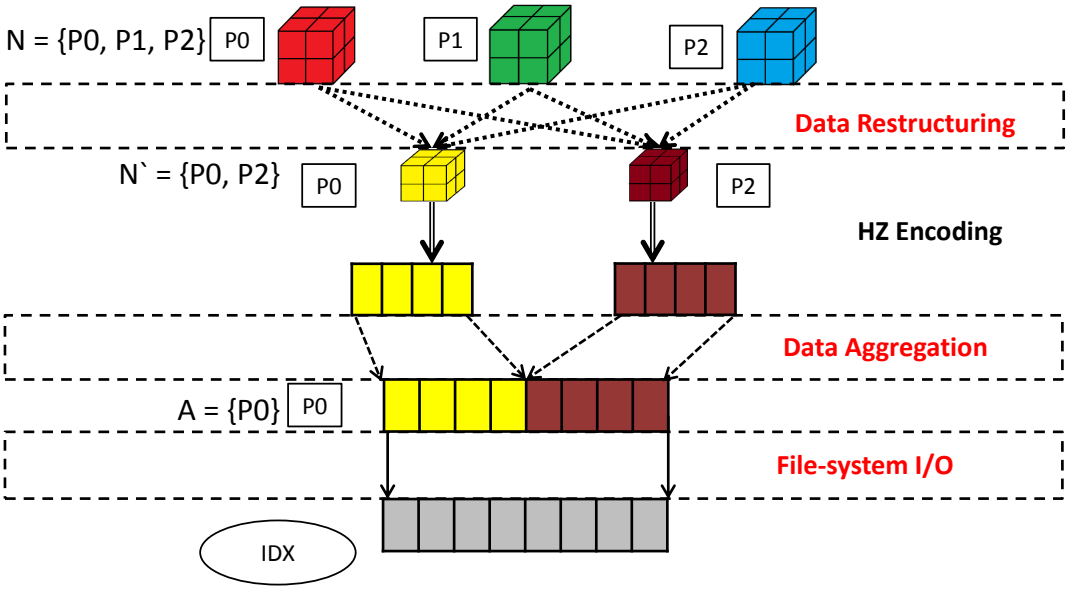


Figure 7.1: Block diagram showing the three I/O phases of PIDX – restructuring, aggregation, and file I/O.

7.1.1 Three phases of I/O

In the first I/O phase (data restructuring), data are redistributed from initial N processes to N' processes in larger power-of-two sized blocks. PIDX allows two values of N': 1) default, N (or $\sim N$), where the extent of the restructured block has dimensions rounded to the closest power-of-two number (of the per-process volume), and 2) expanded, $N/8$ (or $\sim N/8$), where the dimension (or length) of the restructured block is twice that of the default case. In the second I/O phase (aggregation in Figure 7.1), data after restructuring are aggregated from N' processes to a chosen set of A aggregators. The PIDX aggregation phase can be tuned in two ways: by varying the number of processes participating in aggregation and by varying the number of aggregators. The first is a consequence of the restructuring phase that controls the number of processes participating in aggregation ($N' \sim N$ or $N/8$). The second can be directly tuned with PIDX. The default number of aggregators in PIDX is a product of tunable parameter F (file count) and input parameter V (variable count). However, this scheme, which corresponds to the case where an aggregator is responsible for writing all data for one variable in one single file, lacks flexibility. To overcome this lack of flexibility, an additional parameter a_f (aggregation factor) is introduced. Instead of having a fixed number of aggregators as in the default case, a_f provides the flexibility to use all available cores as aggregators. So now the aggregator count A is expressed as $a_f \times F \times V$,

such that $A \leq N$. The third I/O phase (disk-level write) is directly influenced by both aggregator count (A) and file count (F). Note that the domain partitioning transformation presented in Chapter 5 has not been included in this piece of work for two reasons: 1) Chapter 5 successfully does tuning of the partitioning algorithm that possibly can be used elsewhere and 2) it is the first transformation to be applied on the software stack with the three I/O phases following, and due to this, it is rather more important to tune the more detailed layer of the I/O system that directly deals with the network and the storage.

7.1.2 High-dimensional parameter space

Table 7.1 summarizes the parameter space for PIDX. The table is divided into input, output, and tunable parameters. Input parameters are preset at the start of the simulation design. Tunable parameters can be adjusted to improve and optimize performance over multiple simulation runs. In the tunable parameter column, the numerical superscript refers to the I/O phase. For example, N' belongs to restructuring (the first I/O phase). The dependent output parameters are the network throughput, I/O throughput, and the combined throughput of network and I/O. Since PIDX has its own customized collective I/O implementation involving both data aggregation and disk-level I/O phases, both phases are examined separately as well as together.

7.2 Experimental platforms

The experiments presented in this chapter were performed on Hopper at the National Energy Research Scientific Computing (NERSC) Center and Intrepid at the Argonne Leadership Computing Facility (ALCF). Hopper is a Cray XE6 with a peak performance of 1.28 petaflops, 153, 216 compute cores, 212 TiB of RAM, and 2 PiB of online disk storage. All experiments on Hopper were carried out using a Lustre scratch file system composed of 26 I/O servers, each of which provided access to six Object Storage Targets (OSTs). Unless otherwise noted, the default Lustre parameters were used that striped each file across two OSTs. Intrepid is a Blue Gene/P system with a peak performance of 557 teraflops, 167, 936 compute cores, 80 TiB of RAM, and 7.6 PiB of online disk storage. All experiments on

Table 7.1: Summary of parameter space. Numerical superscript refers to the I/O phase.

Independent Variables (Input)	**Parameters (Tunable)**	**Dependent Variables (Output)**
N: Initial cores count	N': Core counts after re-structure[1]	Network throughput
D: Data size/resolution	A: Aggregator count $(a_f \times V \times F)$[2,3]	I/O throughput/File
V: Number of variables	a_f: aggregation factor [2,3]	Combined throughput
M: Machine specs/memory	F: File counts[3]	

Intrepid were carried out using a GPFS file system composed of 128 file servers and 16 DDN 9900 storage devices. Intrepid also uses 640 I/O nodes to forward I/O operations between compute cores and the file system. The Intrepid file system was nearly full (95% capacity) during our evaluation study. It is believed that this situation significantly degraded I/O performance on Intrepid.

7.3 Performance characterization

Two-phase I/O algorithms consist of 1) data aggregation (shuffling) over the network and 2) file system level I/O writes. Both phases are studied independently. This approach helps to isolate artifacts, bottlenecks, and patterns that are specific to each phase. First the characterization methodology is described, followed by the results of applying that methodology to two HPC platforms. In addition, findings are highlighted that can be applied outside the scope of PIDX, both for the data aggregation phase as well as the file system level I/O. Because the data aggregation phase is independent of the PIDX file format, results from both network and I/O characterization could be applied to other parallel I/O libraries as well. Also, the I/O phase of PIDX is capable of writing data to shared files, unique files per process, and various configurations between those two extremes. This choice is independent of the network characterization.

The most critical tunable parameter for both the network and I/O phases of PIDX is the selection of the number of aggregator processes (A). This parameter impacts the overall algorithm in three ways: 1) it affects the distribution of data over the network when aggregating data from the N' cores that hold restructured data, 2) it dictates the number of cores that will access the file system during the I/O phase, and 3) it controls the maximum amount of memory that will be consumed on a core. The memory consumption is particularly important for applications that have already been tuned to use a large fraction of memory on each core for computation purposes. In this work, it is assumed that all data will be transferred to the aggregators before being written to the file system.

7.3.1 Methodology

The number of aggregators A in the study is varied from N' (where every process holding restructured data is also an aggregator) to $N'/32$ (where one of 32 processes holding restructured data is also an aggregator). The $A = N'$ case uses the least amount of memory per process, whereas $A = N'/32$ requires the most memory. There is also some flexibility in varying the value of N'; it affects two aspects of the network phases: 1) how data is redistributed from the compute cores (N) to restructuring cores (N') for HZ encoding

during the restructuring phase and 2) how many nodes the data must be aggregated from during the aggregation phases. The default value of N' is N, but an *expanded* restructuring configuration is also evaluated in which $N' = N/8$. The value of N' has no effect on the I/O phase of the algorithm.

Two types of experiments are used to characterize network and I/O performance. In the first type, which is referred to as *data scaling*, the number of cores is fixed at 4096 with all cores participating in aggregation (*default* case), while the per-process data size is exponentially varied from 256 KiB to 64 MiB. Table 7.2 lists the values, for resolution (D) and variable count (V), used to achieve the range of data size within each process. In the second type, which is *weak scaling*, the per process data resolution is fixed to $64 \times 64 \times 64$ ($V = 1, 2, 4$), and the number of cores is varied between 1024, 4096, and 8192.

The above choice of parameters represents a wide range in the parameter space that can reasonably be investigated. In addition, scaling studies for both load and core will highlight varying trends that are specific to the choice of load or the number of cores used. Most importantly, as will be shown later, these diverse parameter settings and variety of datasets generated lead to a rich training set that helps build an accurate and robust system model.

7.3.2 Network characterization

In this section, we study the performance of data aggregation from N' processes to A aggregators to characterize the interconnect network. This phase involves data being exchanged among processes over the internode communication channel hence characterizing the underlying network. One-sided remote memory access (RMA) is used for all communication in this phase of the PIDX algorithm.

Table 7.2: Different configuration of resolution and variables of dataset used in our experiment for data scaling along with corresponding memory allocated.

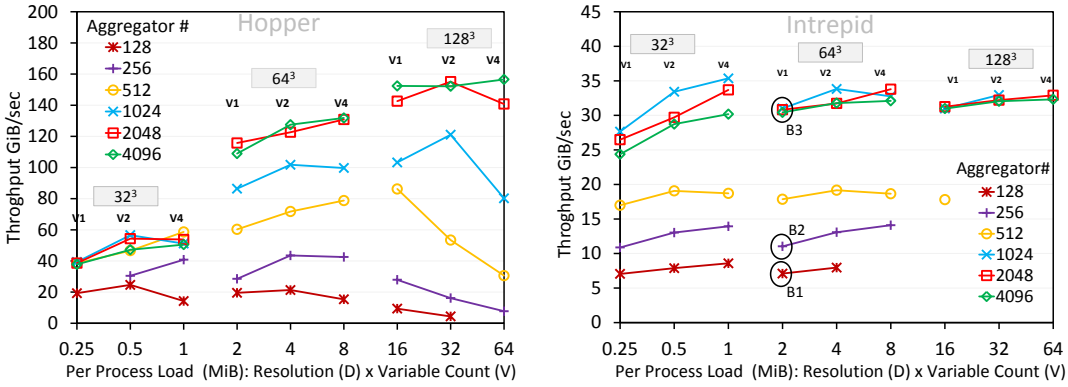
Resolution (D)	Variables (V)	Memory
$32 \times 32 \times 32$	1	256 KiB
	2	512 KiB
	4	1 MiB
$64 \times 64 \times 64$	1	2 MiB
	2	4 MiB
	4	8 MiB
$128 \times 128 \times 128$	1	16 MiB
	2	32 MiB
	4	64 MiB

7.3.2.1 Data scaling

Network scaling results for Hopper and Intrepid can be seen in Figure 7.2. Note that the figure has disconnected trend lines. They are grouped by resolution values (annotated by a gray box at the top of each column). Overall, an increase in aggregator count typically leads to an improvement in performance on both machines and across all data loads (32^3, 64^3 and 128^3 with $V1, V2, V4$).

In order to better understand the impact of different aggregation schemes, network counter data for Intrepid were also collected. The data correspond to all links of the torus for the three aggregator counts (4096, 256, and 128) and for data size 64^3 with one variable (see Figure 7.3). Projection of the three-dimensional network topology provided by Boxfish [72] was used. Boxfish is an integrated performance analysis and visualization tool developed at Lawrence Livermore National Laboratory. Each image of Figure 7.3 shows all the network links along two torus dimensions aggregated into bundles along the third dimension. Two observations can be made from the figures: 1) fewer data packets are transmitted across the network with increasing aggregators, and 2) a reduced aggregator count results in skewed data distribution across network links. Hence, for Intrepid, the higher aggregator counts of N', $N'/2$, and $N'/4$ perform much better than the lower aggregator counts $N'/8$, $N'/16$, and $N'/32$. Also, for Intrepid, almost similar performance for N', $N'/2$, and $N'/4$ can be attributed to the architecture of the machine, where it has four cores on a node.

Boxfish visualization data are not available on Hopper, but the performance analysis for Intrepid holds for Hopper as well, as it demonstrates similar performance ordering with varying aggregator counts (see Figures 7.2 (a) and 7.2 (b)).



(a) (b)

Figure 7.2: Throughput of data aggregation phase with varying data loads for (a) Hopper and (b) Intrepid.

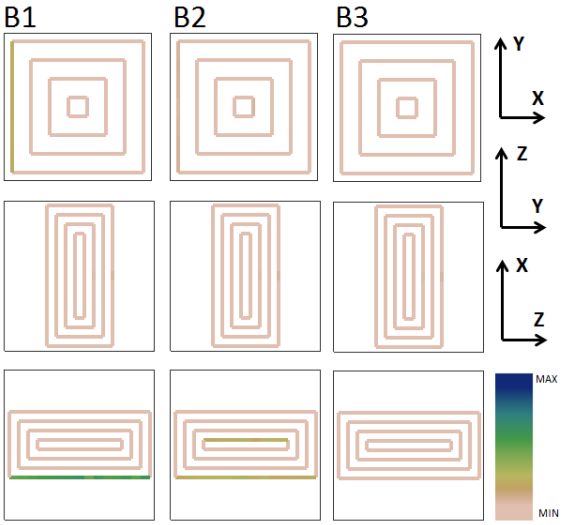


Figure 7.3: Visualizing the Intrepid network flow with Boxfish for aggregator count 128 (B1), 256 (B2), and 4,096 (B3) (denoted as black circles in Figure 7.2 (b)).

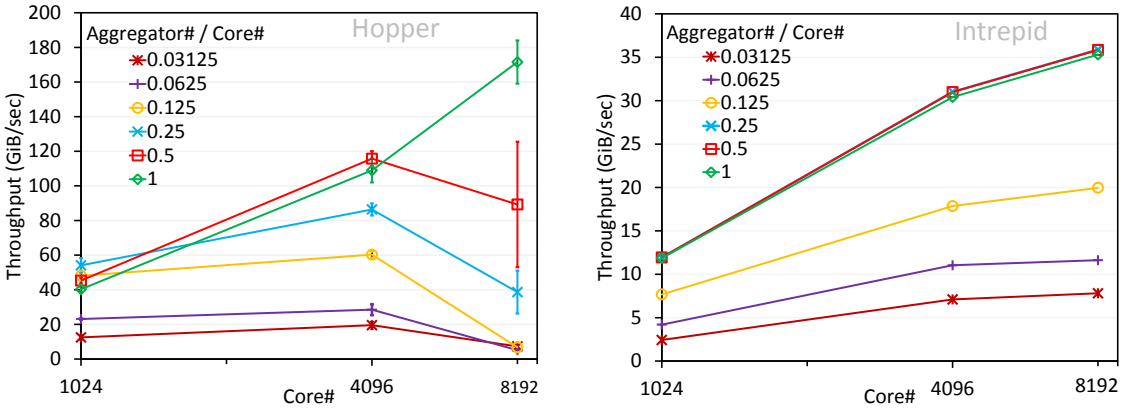
It can also be observed that the range of aggregator counts from 128 to 1,024 exhibits a degrading trend on Hopper as the data volume is increased. In contrast, Intrepid exhibits fewer variations across data volumes regardless of the number of aggregators. This result can be attributed to the fact that Hopper is more sensitive to variations in the quantity and size of network messages being transmitted between nodes.

It is important to choose an aggregator configuration that yields optimal throughput both for data aggregation and the more dominant I/O phase. In addition to optimal throughput, memory requirements should be taken into consideration while choosing an appropriate aggregation configuration.

7.3.2.2 Weak scaling

Figure 7.4 shows network scalability on Hopper and Intrepid while varying the total number of cores from 1,024 to 8,192 with a fixed data volume per process of 64^3. The trendlines correspond to the ratio of the number of aggregators and the number of cores. Six different configurations with ratios of $1/32 = 0.03125$, $1/16 = 0.0625$, $1/8 = 0.125$, $1/4 = 0.25$, $1/2 = 0.5$, and 1 are used. A ratio of 1 implies aggregator count equal to the total number of cores.

In Figure 7.4 (a), Hopper shows scalability with 8K cores only when the number of aggregators and processes participating in aggregation is equal. In contrast, Intrepid shows an upward trend for all core counts in Figure 7.4 (b) regardless of the aggregator ratio. These differences in behavior are a result of the different network topologies in Hopper and



(a) (b)

Figure 7.4: Scaling of throughput performance for data aggregation phase on (a) Hopper and (b) Intrepid.

Intrepid. Intrepid has a dedicated network partition for each application, while Hopper has a shared network and nodes may be allocated in a physically distant manner. With fewer aggregators and a larger number of processes participating in aggregation, a packet must travel a greater distance, which increases the likelihood of interference from other running jobs.

An additional restructuring phase led to the scaling of data aggregation phase for Hopper, as seen in Figure 7.5. All the results are for $8,192$ cores while varying the number of aggregators from 256 to $4,096$. Results for two variable counts, 1 and 4 for data with

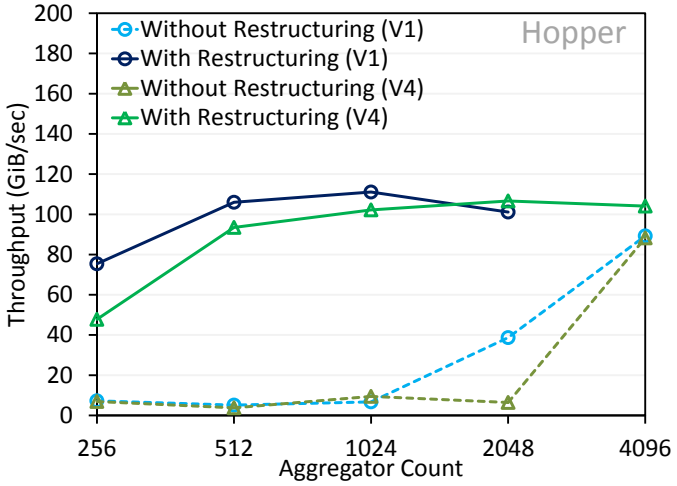


Figure 7.5: Throughput performance after restructuring data (before aggregation), and hence reducing the numbers of processes participating in aggregation to $8,192/8 = 1,024$.

resolution, 64^3 are shown. The per-process load varies from 2 MiB to 8 MiB. Restructuring data using an *expanded* box before the aggregation phase reduced the number of processes participating in aggregation by one-eighth ($1,024$ from $8,192$), which indirectly reduced the outreach of nodes, thereby compressing the networking space. As can be seen in the figure, at an aggregator count $4,096$, there is almost a 10-fold improvement in performance for aggregation with restructuring over aggregation without restructuring.

7.3.3 I/O characterization

This section details the performance of file I/O as A aggregator nodes write to the parallel file system. As in the network study, both load and core counts are varied in order to evaluate different aspects of the storage system. Excerpts from the study that exhibit interesting behavior are presented here.

7.3.3.1 Data scaling

For Hopper, at 4,096 cores, global volumes of 512^3, $1{,}024^3$ and $2{,}048^3$ were used. Each global volume setting consists of one variable with per process resolution of 32^3, 64^3, and 128^3 and varying data sizes of 256 KiB, 2 MiB, and 16 MiB, respectively. Two sets of experiments were performed on Hopper. In the first case, the total number of files was varied while keeping the aggregation factor constant, and for the second, the aggregation factor was varied while keeping the number of files constant.

In the first set of experiments with both a_f and variable count equal to 1, the number of aggregators is equal to the number of files. This configuration is referred to as Case U because of its similarity to a unique file per process I/O strategy. In the second set of experiments, the number of files is constant (set to the minimum file count seen in Case U), and instead the number of aggregators is varied using the variable aggregation factor a_f. This configuration is referred to as Case F, since the number of files that are constant is kept fixed. As in network characterization, the number of aggregators is exponentially from N to $N/32$ ($4,096$ to 128).

Figure 7.6 shows the Case U and Case F results on Hopper. Case U exhibits superior peak performance compared with that of Case F for all three data volumes, but the optimal number of aggregators varies in each case. Focusing on the 1024^3 volume as an example, two key trends can be observed. First, the performance degrades as the aggregator count increases for both Case U and Case F. This result can be explained by looking at burst sizes for the different aggregator configurations in Table 7.3. A large number of aggregators lead to smaller I/O write sizes as the data are distributed over more processes. For example,

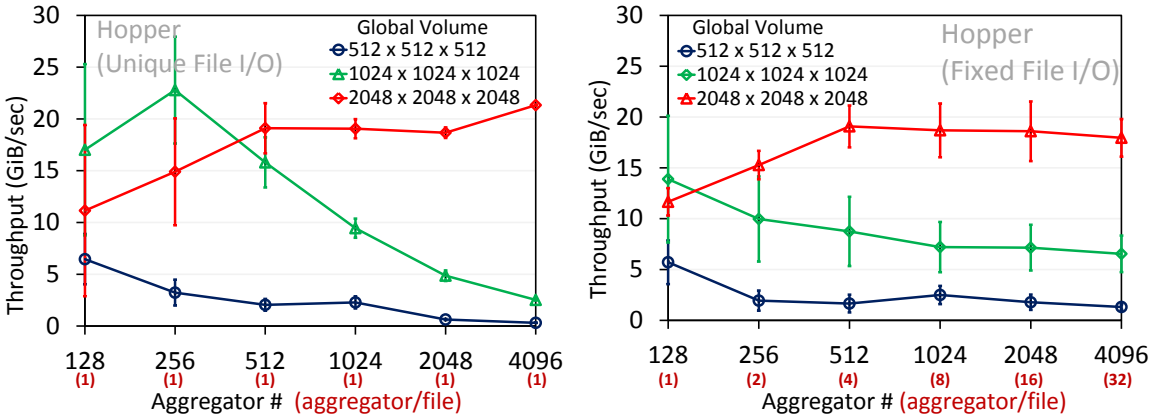


Figure 7.6: Hopper results for throughput vs. aggregator count for (a) Case U (unique File I/O) and (b) Case F (fixed File I/O).

Table 7.3: I/O burst size for all aggregation combination with all different loads.

Agg Count	IO Burst Size (MiB) 32^3 (V1)	IO Burst Size (MiB) 64^3 (V1)	IO Burst Size (MiB) 128^3 (V1)
128	8	64	512
256	4	32	256
512	2	16	128
1024	1	8	64
2048	0.5	4	32
4096	0.25	2	16

for the 32^3 dataset with 128 aggregators, the I/O burst size is 8 MiB, whereas with 4,096 aggregators, it is only 256 KiB. The small write access pattern scales poorly. Second, Case U exhibits more rapid degradation than does Case F as the number of aggregators is increased. This difference is due to the overhead in creating a larger number of files in Case U. For example, 4,096 unique files must be created when using 4,096 aggregators in Case U. Although these files are created in parallel, the file creation cost constitutes a larger portion of the I/O time relative to the cost of the actual file I/O. The 2048^3 global volume example does not exhibit this trend even though it is creating the same number of files because the I/O volume is large enough to amortize the file creation overhead. Therefore, a steady improvement in performance is observed as the aggregator count is increased at larger data volumes.

The same experimental setup was used for measurements on Intrepid. Unlike Hopper, Intrepid shows better performance for Case F than for Case U for all data volumes because

of differences in file system architecture between the systems. Better I/O performance is achieved on Intrepid by using fewer files (shared) and avoiding serialization points from creating multiple files in the same directory.

7.3.3.2 Weak scaling

This analysis retains the Case F and Case U configurations from the *data scaling* analysis. The total number of cores is varied while keeping the data volume per process fixed at 64^3 with one variable (2 MiB per process). Trendlines are plotted corresponding to the ratio of number of aggregators and the number of cores. Six different configurations with ratios of $1/32 = 0.03125$, $1/16 = 0.0625$, $1/8 = 0.125$, $1/4 = 0.25$, $1/2 = 0.5$, and 1 are used. A ratio of 1 implies aggregator count equal to the total number of cores.

On Intrepid for both Case F and Case U, the weak scaling shown in Figure 7.7 reveals distinct performance patterns. As can be seen in Figure 7.7 (a), for Case U where the number of files equals the number of aggregators, two important trends can be seen: improvement in performance with decreasing aggregator count for all cores and no scaling in performance with increasing core counts. These observations agree with the results seen in the previous study (*data scaling*). The trends are due to large I/O writes leading to better disk access patterns, as well as smaller overhead in creating the hierarchy of files. Straight trend lines across cores represent poor scaling performance, which can be explained by the increasing overhead in creating more files. In Figure 7.7 (b), the performance scales according to the quantity and size of the I/O operations with no change in file creation overhead.

A similar weak scaling experiment on Hopper for Case U can be seen in Figure 7.7 (c). Performance behavior is similar to that of Intrepid with throughput improving with increasing aggregator count. The flat trendlines for the aggregator count and core count ratio 0.5 and 0.25 indicate no scaling due to the overhead in creating the hierarchy of files. A key point to note is the performance gain at every core while reducing the number of aggregators. Reducing the number of aggregators by half yields a twofold performance improvement (from 2.5 GiB/sec to 5 GiB/sec) due to both better disk-access pattern from large I/O writes and smaller overhead in creating the hierarchy of files. As can be seen from Table 7.3, for the 64^3 dataset with one variable and 4,096 aggregators, the I/O burst size is only 2 MiB as compared with relatively larger (and more favorable) burst sizes of 4 MiB and 8 MiB with aggregator counts 2,048 and 1,024. Besides better disk accesses with fewer aggregators, there are fewer files to write, and the overhead in creating the files is reduced

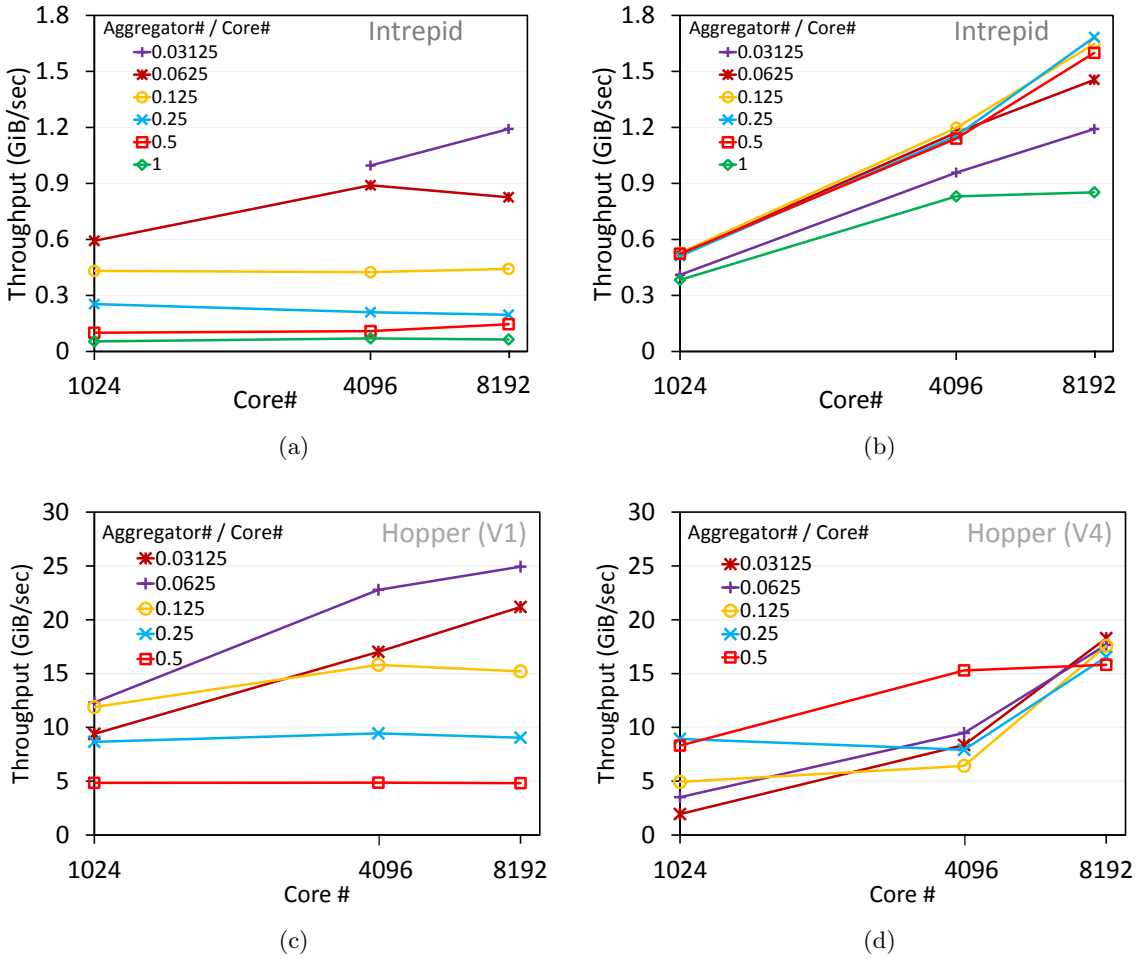


Figure 7.7: Throughput for Intrepid with increasing core counts; trend lines correspond to the ratio of aggregator count and core count for (a) Case U and (b) Case F. Throughput for Hopper with increasing core counts; trend lines correspond to the ratio of aggregator count and core count for (c) Case U (for V=1) and (d) Case U (for V=4).

as well. File per process access patterns on this system are most effective if the data volume is large enough to amortize file creation costs.

Instead of showing results for Case F on Hopper, we present a new set of experiments in which the number of variables is changed to four while otherwise retaining the load configurations from Figure 7.7 (c) and Case U. The configuration illustrates the performance of the system with a larger load. The results can be seen in Figure 7.7 (d). The key observation is the reversal of order of aggregator while going from 1024 cores to 8192 cores. Having fewer aggregators along with fewer files results in better performance at higher scale because of the reduced contention and file creation overhead.

The above examples, specifically the last one, illustrate how dramatically system behavior can vary at different data scales. This situation motivates the need to build a performance model to aid in parameter exploration.

7.4 Performance modeling

This section goes through the process of creating machine learning-based system models using regression analysis. As will be showed, the learned models accurately predict the performance and parameters for a wide range of system configurations and for two different computing architectures, Hopper and Intrepid. Moreover, the models used are independent of PIDX and can be used for other existing parallel I/O libraries.

7.4.1 Model description

To model a system, training data is collected, following which informative features or attributes are selected from the training data that are then used to train different models. The best-performing model is then chosen. Apart from studying the system behavior, another benefit of the characterization study is generating large amounts of data that can be used to build a prediction model. Here each simulation run is a data point and the different parameter choices are the attributes (or attributes/fields/dimensions) of the data point. The total number of simulation runs defines the number of data points (or training data size, say N). Since the number of parameters is the same for each data point (say, D), all datasets are of the same N and attribute size D. Table 7.4 presents a typical example of a single data point with nine attributes from our training dataset. One can think of the data point as a vector with D components. Let x_i denote the ith data point, and let x_{i1} to x_{iD} be its D attributes. In our example, x_{i1} would be "GlobalV," x_{i2} would be "LocalV," and so on. Each data point x_i is associated with an output y_i, which in our case is the throughput. Given a dataset $\{x_i, y_i\}$ of $i = 1, \ldots, N$ and $x_i \in \mathbb{R}^D$, a general machine learning model (call it θ) aims to learn the relationship between the D variables x_{i1} to x_{iD} and the observed outcome y_i. The learned model θ predicts the outcome $\hat{y}_i = \theta(x_i)$ for

Table 7.4: Example data point showing nine attributes.

Independent (input) Variables					**Tunable Parameters**			
total data block dimensions	data block dimensions per core	partici-pating processes	number of fields	system memory	number of aggre-gators	aggre-gation factor	number of files	whether using re-structuring
GlobalV	LocalV	#Cores	V	Mem	AGP	AF	F	R
1024×1024×512	64×64×64	2048	4	1	64	4	4	0

each data point x_i and in the process incurs a loss $\sum_{i=1}^{n} loss(\hat{y}_i, y_i)$. Loss quantifies the deviation of the predicted outcome from the true outcome and can be of various forms such as squared, absolute, log, or exponential. The goal of the learning process is to formulate an objective or cost function (that includes the loss) and then choose a model that best minimizes the objective (and hence the loss). The general form of the optimization problem to be minimized is

$$\arg\min_{\theta} \sum_{i=1}^{n} loss(\hat{y}_i, y_i) = \arg\min_{\theta} \sum_{i=1}^{n} loss(\theta(x_i), y_i),$$

and this can be solved by using standard techniques from the optimization literature (for example, gradient descent). In algebraic terms, the dataset can be thought of as an $N \times D$ matrix $\mathbf{X}$ and the outcome as an $N \times 1$ vector $\mathbf{Y}$; and the goal is to solve for the expression $\mathbf{Y} = \mathbf{X}\theta$ to obtain the $D \times 1$ model description vector θ.

Based on the problem domain or the data type, machine learning models can be broadly categorized into two types: 1) classification and 2) regression. In classification problems, the outcomes y_i are categorical variables, whereas for regression, the output is continuous variable. In our case, because of the continuous nature of the throughput output variable, a regression based model is built. Regression models have the general form

$$y_i = \theta(x) = \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \ldots + \beta_D x_{iD} + \varepsilon_i,$$

where the model θ is represented by D coefficients β_j $(j = 1, \ldots, D)$ and ε_i represents random noise in the data. As earlier, the goal is to solve for the regression coefficients β_js using algebraic analysis or optimization schemes.

7.4.2 Model selection

Various regression models have been proposed in the literature. In this study, I experimented with a number of different regression models [73] that included 1) linear models, such as linear regression, ridge regression, lasso, lars (least angle regression), elastic net, SGD (stochastic gradient descent), and support vector regression (with linear kernel): and, 2) nonlinear models, such as decision trees, support vector regression (with polynomial and RBF kernels), and gaussian processes. Oftentimes, an *ensemble* of classifiers outperforms a single classifier, which has led to the popularity of ensemble models, such as bagging and boosting. Bagging ensemble models, such as random forests and gradient boosted decision trees (GBDT), were also tried for the model.

The comparative performance of different regression models over all datasets for Intrepid is presented in Table 7.5. After testing all models, a tree-based regression model was

Table 7.5: Comparison of model performances, showing average error of all experiments.

Model	Error (in %)	Model	Error (in %)
Linear Reg	19.6	SVM Reg (Lin)	21.2
Ridge Reg	20.2	Decision trees	9
Lasso	18.9	SVM Reg (Poly)	16
Lars	20.34	Gaussian Processes	13
Elastic Net	21.68	Random forest	8.2
SGD	16.7	GBDT	8.1

chosen as it resulted in the lowest test error across all datasets tested. Tree-based models include decision tree (for standalone classifiers) and random forests and GBDT (for ensemble models). Tree-based models are simple and intuitive to understand because the decision at each step (node of the tree) is based on a single attribute or feature (in our case system, parameter) of the dataset, which involves a quick look-up operation along the depth of the tree. In contrast, other machine learning algorithms build models in dual space or solve a complex optimization problem, which makes it difficult to judge the relative usefulness of specific dataset attributes. Moreover, unlike most machine learning models, tree-based models do not require much parameter tuning. Indeed, tree-based models such as random forests and GBDT are a popular choice for model building and data analytics and have proved useful in related HPC applications [3]. In the discussions below, results using tree-based models are presented. For the experiments, Python-based standard regression packages from the open-source machine learning toolkit *scikit-learn* [74] were used. For all the experiments, multiple runs are performed with the mean value being reported.

7.4.3 Training data

Our training data consists of performance figures for different parameter settings collected during the characterization study (see Section 7.3). We collect data from two phases: data aggregation (shuffling) and data I/O. We combine these datasets to construct training data for the entire two-phase I/O. Our system modeling and performance predictions are based on this combined dataset.

7.4.4 Attribute selection

A wide range of attributes was selected to improve the discriminative power of the model. To start with, application information was extracted in terms of global and local data resolution, number of variables (or fields), and file count. System-level information, such as core count, aggregator count, aggregation factor, memory, and whether restructuring

was used or not, was also used. Overall, there were nine attributes, which are listed in Table 7.4 along with a brief description of each. Table 7.4 also presents an example line from our combined dataset. Attribute selection or attribute reduction procedures [3] to select a smaller set of useful attributes are usually beneficial for large attribute spaces. In this case, since the number of attributes was already small and moderately reasonable, attribute selection or reduction was not performed.

7.4.5 Results

This section shows results to demonstrate the accuracy of the model trained on the combined dataset. The model has been trained on training data obtained on small cores (1,024, 4,096 and 8,192) for the microbenchmarking application. The model is initially validated on test data also obtained from the same small core regime. Performance validation results are presented for microbenchmarking and S3D applications, where aggregate throughput is predicted (given a set of parameters). Each run of S3D generates four variables: pressure, temperature, velocity (3 samples), and species (11 samples).

7.4.5.1 Performance validation

Figure 7.8 presents results of model validation on low core count regimes for microbenchmarking (2,048 cores) and S3D applications (4,096 cores). The throughput for different values of the aggregator count is predicted. In all cases, the throughput prediction of the model is close to that of the original throughput values. Note that for S3D, the prediction performance is particularly good despite the fact that the model is trained on the microbenchmark data. Thus, the trained model is sufficiently general and performs well across different target applications. In Figure 7.8, it can be seen that the overall average percentage error for Hopper is $\sim 32\%$ and for Intrepid is about 20%.

7.4.5.2 Parameter prediction

In this section, the model is used to predict parameters that maximize throughput. To this end, an adaptive modeling framework is proposed (shown in Figure 7.9). To start with, the model is trained on a labeled dataset. Next, a set of candidate points $p_1, p_2, \ldots, p_n$ is sampled, and performance throughput of each point in the set is predicted using model t^0. The point p^k that has the maximum performance throughput is selected. Thereafter, the machine is set to the parameter values obtained using the selected point p^k and is made to run for the next time-step of the simulation. The output of the simulation process yields the true observed throughput p^k, which is fed along with the point p^k to the model (via

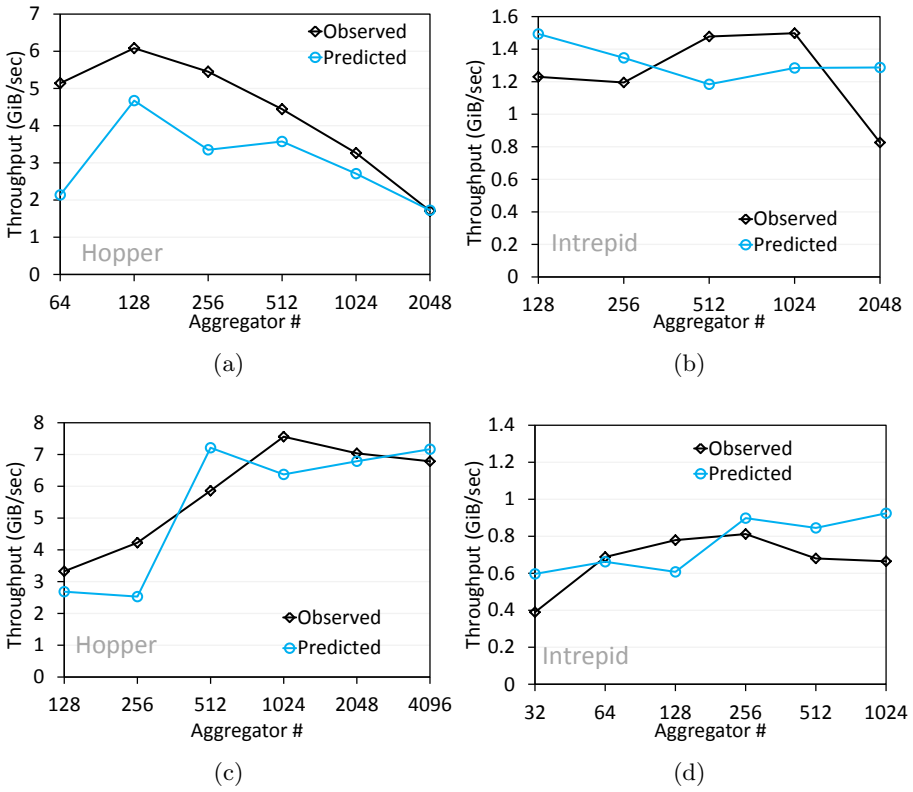


Figure 7.8: Validation results for microbenchmark (top row) and S3D (bottom row) for Hopper (left panels) and Intrepid (right panels).

the dotted loop) and retrains the model. This iterative process continues until a predefined number of time-steps is reached or the predicted throughput does not change in subsequent iterations, indicating that the model has converged or is close to convergence.

For verification, the model is tested on Hopper at 4,096 cores and 64^3 data load. The model is able to identify the best A value 128. Note that this model is fully automatic and adapts its performance at each iteration to find parameter settings that improve the overall throughput performance. In the future, it is planned to test this model on larger core counts and use it to auto-tune the system performance fully automatically and without any manual intervention.

7.4.5.3 Prediction for high core counts

In validation results, performance of our prediction was tested on the same regime from which training data were collected. In this section, the model is used to study the performance of our prediction on high core count regimes (16K, 32K, and 64K). Note that since the model has not seen points in a high core count regime, it would be useful to update them after prediction on each data point. This adaptive approach caters to *online* machine

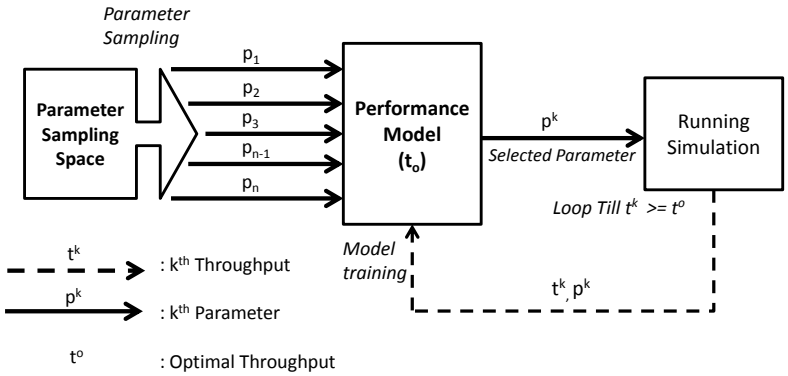


Figure 7.9: Block diagram for adaptive modeling.

learning techniques (where the model is updated after seeing each point) and is in contrast to the *offline* learning models used in the validation section.

For performance prediction, different test cases on Hopper and Intrepid are considered. For Hopper, different data loads 32^3 and 64^3 are used, whereas for Intrepid, the load is fixed at 64^3 while varying the parameter settings. Figure 7.10 shows the modeling results for S3D on Hopper and Intrepid. Observe that for Hopper, the prediction is better for a 32^3 data load than a 64^3 data load. On Intrepid, the prediction for $A = N/4$ is better than for $A = N$. In all cases, the predictions are reasonably close, showing the benefits of an adaptive model that gradually improves itself over multiple prediction time-steps. For a 32^3 data load, the errors on Hopper and both cases of Intrepid are less than 30%. For a much larger 64^3 data load, however, the error increases drastically. As can be seen, tree-based nonlinear models have been unable to capture this behavior of Hopper. Characteristics of Hopper at high data loads change significantly from that on low data loads and hence might

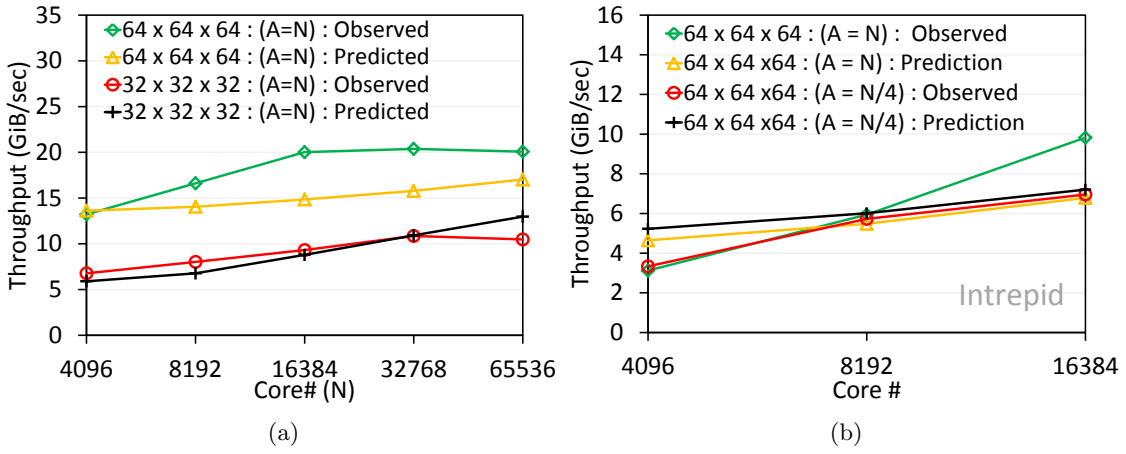


Figure 7.10: Prediction results at high core counts for S3D on (a) Hopper and (b) Intrepid.

need additional investigations for characterization and modeling. One reason could be that the network phase in Hopper is much less well behaved than in Intrepid.

In Figure 7.8 and Figure 7.10, it is seen that the modeling error is high in only a few cases (for example, for 64 aggregator core count in Figure 7.8 (a)). In addition, it is observed that the overall average percentage error was particularly high for Hopper (32% in Figure 7.8) but much less for Intrepid (20% in Figure 7.8). These results lead us to conclude that Intrepid is more well behaved and simpler to model than is Hopper. Similarly, in characterization study plots, Hopper exhibited higher variability in error than did Intrepid. One reason for the difference in behavior is that the network phase, which is a component of the combined throughput, is more well behaved on Intrepid and hence is easier to model and predict. This is primarily to do with how jobs are allocated by default on Blue Gene vs Cray machines. On Blue Gene, the communication traffic is isolated, whereas on Cray it can interfere with the traffic of other jobs.

Our proposed model can be applicable to other parallel I/O library file formats, such as pnetCDF, parallel HDF5, and Fortran I/O. As for PIDX, each case needs to identify discriminating attributes that would be useful in training the model. For PIDX, such information in the form of aggregator count, number of cores, and number of files is used. Similarly, for pnetCDF one can use the number of files created [75] presented with a subfiling approach for pnetCDF (where performance is demonstrated with respect to the number of files written), or any other parameter-affecting performance. Once the informative attributes are identified, one can collect these feature values for different data points and create a training dataset to train a model. Thus, the underlying model remains unchanged, and one needs only to vary the attributes or fields in order to make the model applicable to different file formats.

In this work, the data from our initial characterization study were used to build a regression-based performance model. This performance model is beneficial for exploring parameter spaces that are difficult to cover in characterization studies. For instance, in Figure 7.10, we use the trained model to approximate performance up to 64K core counts. Our model also reinforces the performance pattern observed in the characterization study; for example, Figure 7.7 (c) (of the characterization study) and Figure 7.8 (a) (of the modeling) demonstrate similar trends of declining throughput with increasing aggregator counts.

7.5 Conclusion

This chapter talks about the characterization study and performance modeling of the parallel I/O library PIDX. It was observed that because of differences in ways job partitions are allocated, the two architectures exhibited different network scaling behaviors; for Hopper the network fails to scale at higher core counts. On the other hand, the Intrepid network is more stable, scalable, and less responsive to varying message sizes than Hopper. From the I/O characterization study, it was observed that small details such as file creation time add substantial overhead. Hopper (Lustre) is more optimized to a unique file per process I/O approach than is Intrepid (GPFS), whose I/O is optimized for fewer shared files. It was also observed that optimizing I/O at varying scale requires the proper choice of aggregators, which again varies according to the machine. Further, the datasets from the characterization study were used for training machine learning models. The models show that throughput and parameters can be accurately predicted using regression analysis techniques. It was also shown that models trained on datasets from low numbers of cores perform reasonably well on high numbers of cores, albeit with some online adaptive updates. Another key aspect of this approach is that the proposed models are independent of the characteristics of the target machine, the underlying file-system, and the custom I/O library used and thus can be applied to other I/O application scenarios.

PART IV

ADAPTIVE I/O

CHAPTER 8

SPARSE DATA STORAGE AND I/O

With simulations increasing in size and complexity, it is difficult for I/O and storage systems to keep pace with the increasing amount of data that scientists need to store. Increasing I/O time and storage costs leads to a curtailment in the frequency of analysis and checkpoint dumps, and it also slows down postprocessing analysis tasks. One way to effectively tackle this problem of increasing I/O time and storage costs is to perform sparse data dumps from simulations. This can be done by transforming the typical dense multidimensional array output of a simulation into a light-weight sparse representation, which can then be written instead of storing the data in their entirety. This chapter presents two use-cases of sparse data dumps suited for uniform resolution simulations: 1) region-of-interest (ROI) (see Figure 8.1 (a) and Figure 8.1 (c)) and 2) reduced resolution (see Figure 8.1 (b)). ROI data dumps correspond to an I/O and storage methodology that allows regions/subsets of any simulation to be written at different resolutions. This method is very effective as many simulations are heavily padded to avoid boundary artifacts, and often the phenomena of interest, e.g., ignition, extinction, etc., are confined to a comparatively small part of the domain. Therefore, writing data that stores the ROI according to some importance measure significantly reduces the overall datasize without impacting the results. With reduced-resolution dumps, a down-sampled version of the entire domain is written out. The two approaches lead to a significant improvement in both the I/O time as well as the storage footprint of the simulation.

Existing HPC storage softwares such as pnetCDF [4], parallel HDF [5], and ADIOS [6] have supports only for dense, multidimensional arrays. This chapter presents a novel, storage, and I/O methodology for performing sparse data dumps (ROI and reduced resolution). There are two key challenges to tackle when performing sparse data writes. First is to identify an appropriate data format that incurs minimal storage overhead. Different data formats can lead to a varying degree of storage overheads. Section 8.1 demonstrates blocking and storing of data in fixed block sizes as an effective approach to minimize storage overhead. The second challenge with sparse data writes, especially ROI dumps, is that of

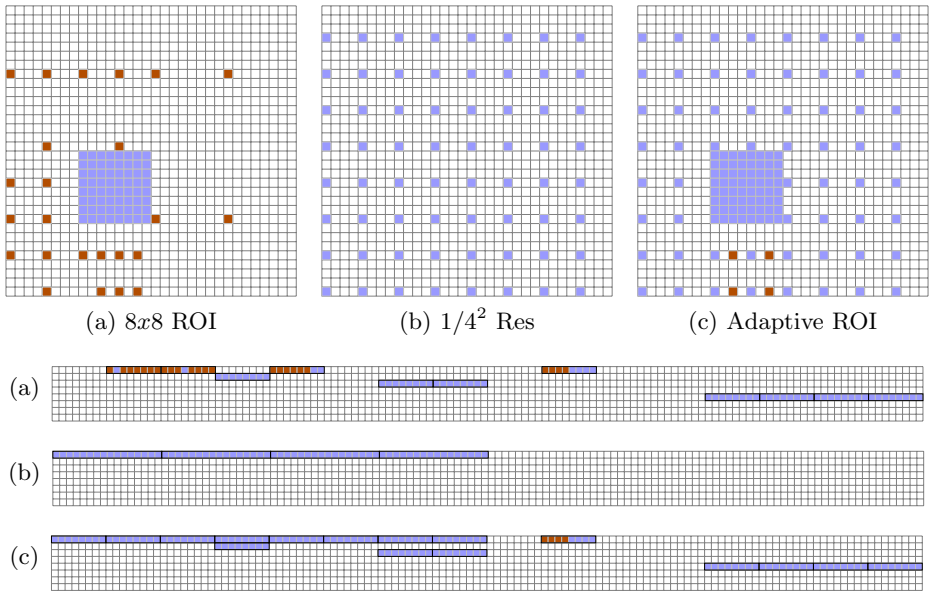


Figure 8.1: Spatial and disk-level layouts of ROI and reduced resolution grids. The top row shows spatial grids with different write patterns. The lower grids show the element layout on disk. For disk-level storage, a block size of 8 is used for (a) and (c), and 16 for (b). Blue elements are primary elements; orange elements are secondary elements, or elements that are written as a by-product. Only primary and secondary elements are actually written to disk – blank elements are not written. (a) ROI write. Efficiency is 0.73. (b) Reduced resolution write. Data on disk are not fragmented for reduced resolution writes, so efficiency is 1 irrespective of block size. (c) Another example of ROI, combining (a) and (b). Secondary elements for the ROI overlap with primary elements of the reduced resolution write, so data are less fragmented, for efficiency of 0.97.

load balancing. In an ROI dump, subsets of the simulation are adaptively written at different resolutions. Load imbalance incurs as processes have different amounts of data to write, as opposed to the same amount of data written by processes in a typical simulation dump. Section 8.2 presents a technique, *layout-aware data aggregation*, to tackle the load imbalance problem. The efficacy of the technique is empirically demonstrated using PIDX on data derived from S3D combustion simulation [65], and is shown to be more efficient than current techniques that store data in its entirety.

8.1 Sparse data storage format

This section presents two sparse storage strategies: ROI and reduced resolution, both of which are attractive alternatives to the traditional raw file dumps. Both of these sparse-data storage techniques lead to a significant reduction in I/O and storage footprint, providing opportunities to increase the frequency of data dumps from simulations. Most simulations perform I/O writes at intervals of several hundred time-steps, partially due to

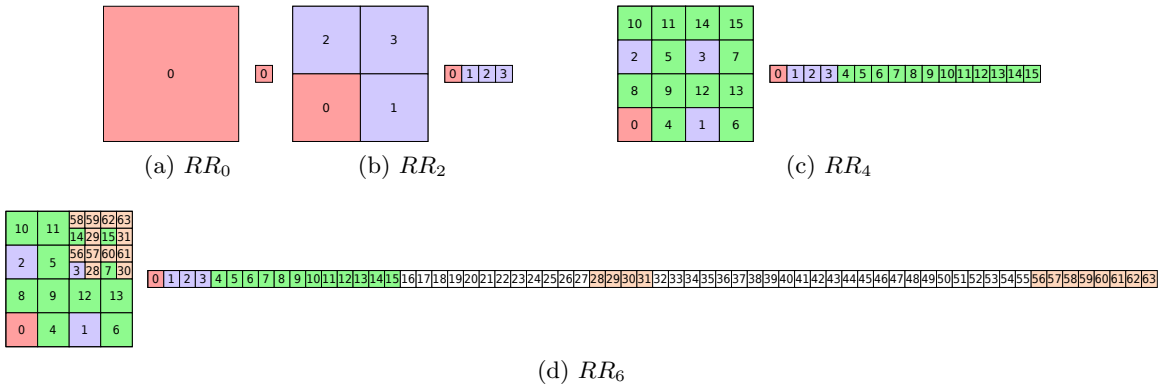
the expensive I/O cost. Sparse data dumps can be used to effectively insert several ROI and reduced-resolution writes between any two full-resolution ones, when more frequent writes are desired, which leads to better tracking and monitoring of the simulation. Following are the two sparse-storage strategies described in detail:

- *Regions-of-interest:* With this method of storage, a subset of the domain is written at the native simulation resolution while the remainder of the region of the simulation is either discarded (see Figure 8.1 (a)) or saved at a lower resolution (see Figure 8.1 (c)). A challenge for these kinds of writes is to identify the ROI before they are written to disk. To this end, we first perform some in-situ analysis that can help the identify regions-of-interest. Both automated and sophisticated techniques such as merge or contour trees and techniques such as range thresholding can be used for this purpose.

- *Reduced resolution:* With this method of storage, the entire domain is written at a reduced resolution (see Figure 8.1 (b)). The resolution can be controlled by the user. Generating simulation results at reduced-resolution allows us to speedup I/O for both parallel writes and exploratory visualization. In addition, this approach allows simulations to output data more frequently.

Multidimensional dense arrays from simulations can be stored to disk in a variety of ways. The simplest approach is to use row- or column-major ordering, but spatial locality is reasonable along only one dimension (the x-axis in the figure). Z-ordering (also known as Morton) [59] shows better spatial locality. Hierarchical Z-ordering extends Z-ordering by introducing a hierarchy. The IDX storage format uses HZ-ordering, which shows good locality both spatially and hierarchically, as shown in Figure 8.2 (a-c). Conceptually, the scheme acts as a storage pyramid with each level of resolution (called HZ level) laid out in Z-order. Figure 8.2 (a-c) shows an example of a 4×4 grid mapped to a linear index using IDX. This section explores the suitability of the format to sparse data and also an extension to the format to support such datasets.

Figure 8.2 (d) shows an adaptively refined version of this grid alongside the now partially occupied index space. Note that even for this adaptively sampled grid, a lower-resolution version of the entire domain can be obtained by reading the contiguous 0-15 block. To understand the impact of the partially occupied index space, a resolution region RR_i at HZ level i is defined. It is the spatial region in the domain stored at resolution i (see Figure 8.2). The region may be of any shape and need not be connected. Since IDX does not replicate samples, each element e of an arbitrary resolution grid with HZ index $HZ(e)$



(a) RR_0 (b) RR_2 (c) RR_4

(d) RR_6

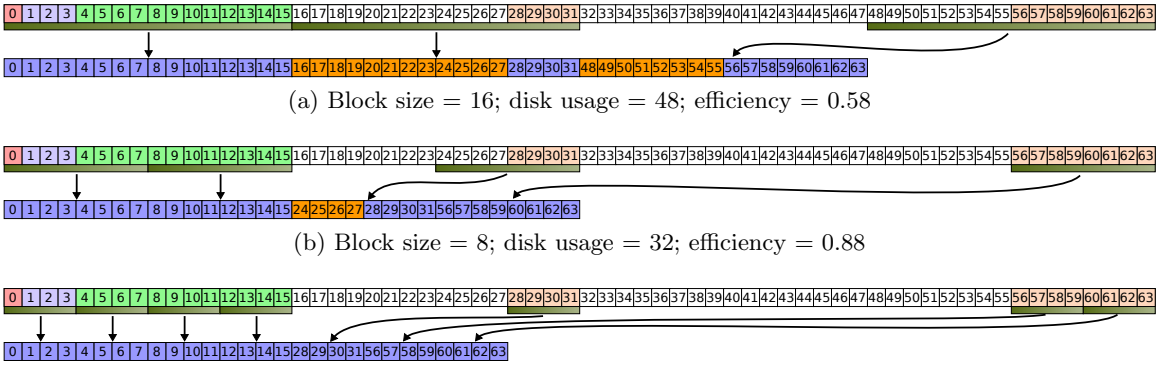
Figure 8.2: HZ encoding of resolution regions. Elements are colored by hierarchy level (HZ level in parentheses): pink (0), blue (1-2), green (3-4), orange (5-6). (a)-(c) Because of hierarchical locality, there is no wasted space when storing the entire domain at reduced resolution. (d) Storing an ROI smaller than the domain introduces fragmentation that will be handled when writing to disk. Elements in white need not be written to disk.

has a uniquely defined HZ level, $HZLevel(e)$, and can potentially be part of all RR_is with $HZLevel(e) \leq i$. Consider again Figure 8.2 (d): RR_4 covers the entire domain and RR_6 only the top-right corner. Even with the good spatial locality of the HZ order, RR_6 gets split into two blocks of memory with indices 28-31 and 56-63, respectively. Storing all the samples in the index space would lead to a waste of storage. Hence, to minimize the empty spaces that need to be stored, the entire dataset does not get stored in a single chunk but instead is broken into blocks that are written to (and read from) disk.

Due to the fragmentation of the index space and the disk blocking, files may contain some "samples" not part of the original grid. These are called *secondary* samples, whereas samples that are part of the original grid are referred to as *primary* samples (see Figure 8.3). Each block that contains at least one primary sample is written and all other blocks are skipped. Figure 8.3 shows the blocking and the resulting data on disk for different block sizes for the example of Figure 8.2 (d). Note that, as the block size decreases, the efficiency increases. The number of secondary samples can vary based on the block size. The (storage) efficiency of a particular scheme is defined as:

$$E = \frac{P}{P + S} \tag{8.1}$$

with P the number of primary and S the number of secondary elements. Note that any lower-resolution write will always have perfect efficiency as long as it covers the entire domain. Furthermore, if there exists an RR_i that does not cover the domain, adding lower-resolution data actually improves the efficiency.



(a) Block size = 16; disk usage = 48; efficiency = 0.58

(b) Block size = 8; disk usage = 32; efficiency = 0.88

(c) Block size = 4; disk usage = 28; efficiency = 1.0

Figure 8.3: Compaction of HZ-indexed elements. The single-dimensional array (above) is divided into write blocks and skip blocks. Only write blocks are stored on disk (below). Elements in the HZ-indexed array are shown colored by HZ level, and elements in the disk array are colored blue for primary and orange for secondary elements. Decreasing block size decreases disk space usage and increases efficiency.

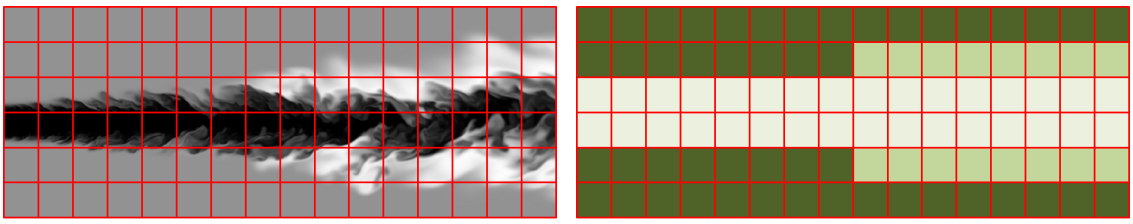
Examples of spatial and disk level layouts of ROI and reduced-resolution grids are shown in Figure 8.1. The top row shows spatial grids with different write patterns. The lower grids show the element layout on disk. For disk-level storage, a block size of 8 is used for (a) and (c) and 16 for (b). Blue elements are primary elements; orange elements are secondary elements, or elements that are written as a by-product. Only primary and secondary elements are actually written to disk – blank elements are not written. Figure 8.1 (a) corresponds to an ROI example. Here only a chunk of data is stored while discarding the rest; this configuration could also be stored using row-order data formats, although that would require implicit tracking of the offset and extent of each chunk. Moreover, handling ROI chunks of varying sizes could also possibly lead to waste of storage. Using the IDX data format with a block size of 8 provides a storage efficiency of 0.73. For reduced-resolution writes in Figure 8.1 (b), since data on disk is not fragmented, we get a storage efficiency of 1 irrespective of block size. Figure 8.1 (c) shows another example of ROI where the ROI is written at full resolution and the remainder is written at 1/16 resolution. In this example, the secondary elements for the ROI region overlaps with primary elements of the reduced-resolution, so data are less fragmented, for an efficiency of 0.97.

The primary technical challenge of ROI I/O is to attain a high throughput even though the data are distributed unevenly and potentially consists of many small isolated regions, which can lead to fragmented accesses to both memory and disk. The following section explores how using different block sizes and the on-the-fly aggregation capabilities of PIDX reduces these problems and leads to high throughput I/O.

8.2 Layout-aware data aggregation

The biggest challenge while performing sparse data writes, especially ROI dumps, is that of load balancing. As opposed to an equal amount of data written by processes in a typical simulation dump, in an ROI dump, load-imbalance is incurred as processes have different amounts of data to write. With the ROI approach, the *interesting* regions of a simulation are typically stored at the full simulation resolution, whereas other regions are either completely ignored or are dumped at lower resolution. This storage pattern makes ROI dumps an inherently load-imbalanced problem as processes having the interesting regions have more data to write as opposed to the remainder of the processes. This load imbalance is shown in Figure 8.4. The grid in red color corresponds to different processes. As can be seen in Figure 8.4 (b), processes in dark green correspond to regions of simulation that are not storing any interesting information, as opposed to processes in light green that store the main crux of the simulation. While performing an ROI dump for this particular simulation, processes in dark green can either discard their data or write the data at a lowered resolution; either ways, it leads to load imbalance among processes.

The first step towards solving the load-imbalanced problem is not to store the entire data in a single chunk but instead break the array into blocks that can be written to disk. This way, each block that contains at least one primary sample is written and all other blocks are skipped. Figure 8.3 shows the blocking and the resulting data on disk for different block sizes for the example of Figure 8.2 (d). Based on the position of the ROI, many of these blocks are sparsely populated. Each block may contain samples from many different processes in an HPC simulation, especially in the initial low-resolution HZ levels. As a result, each process holds a variety of data elements that are destined for noncontiguous file regions. One solution to this problem is to aggregate data before writing by using a two-phase I/O strategy. With data aggregation, a subset of processes is responsible for combining data from all processes into large contiguous buffers before writing to storage.



(a) (b)

Figure 8.4: Load imbalance while performing ROI writes. Processes in dark green have much less data to write as compared to processes in light green.

We wrote a customized *layout-aware* data aggregation strategy for ROI writes. The first step in performing sparse data dumps is identification of IDX blocks that need to be written to disk and accordingly perform aggregation and disk I/O. For the uniform resolution datasets (dense arrays), there are no secondary elements; hence, all blocks get written to the disk. Besides, all processes have the same amount of data to write. As a result, the write-load by default gets equally balanced among the aggregators. This is not true with ROI data, where blocks have to be skipped from being written to the disk. Hence, for sparse data I/O, before performing data aggregation and actual disk writes, it is essential to identify the blocks that need to be written and then uniformly send the data corresponding to these write-blocks to the aggregators. Identification of the *write-blocks* takes place in two steps. In the first step, every process locally identifies the write-blocks that it will populate. To this end, blocks are labeled as a write or a skip block by allocating a buffer that acts as a bitmask for which blocks to write. The size of the buffer is determined by the highest HZ level. For a given processor P, let h be the highest HZ level represented by an element. P then iterates through all blocks in all HZ levels $\leq h$. If a block B intersects the spatial region covered by P, then B is marked as a write-block. A write-block is represented by one in the bitmask (non-write/skip-blocks are 0). At the end of this step, all processes have a local copy of the bitmask. After this step, we perform an MPI_Allreduce on the bitmask with a bitwise OR operand. This step leads to the calculation of the global block bitmask that is shared by all processes. Every *one* in the global bitmask corresponds to the block that needs to be populated, and a *zero* corresponds to the blocks that need to be skipped. With the write-blocks identified, the information is used to set up the aggregation buffers for the aggregation step. This step helps avoid any unnecessary I/O (for skip-blocks) and achieves load balance.

The aggregation algorithm was implemented within the PIDX framework. MPI one-sided RMA communication is used for aggregation, with each aggregator presenting an RMA window in which to collect data. The clients place data directly into appropriate remote buffer locations according to a global view of the dataset. Note that this global view is created in the first phase where processes identifies all the write-blocks. Details on aggregation selection and aggregatior placements can be found in Section 4.1.

8.3 Performance evaluation

The aim of this section is to evaluate the time and storage efficiencies of various layouts of regions of interest, data and weak scaling results on reduced-resolution writes, and time

and disk usage results from combined ROI and reduced-resolution writes of combustion simulation output using S3D.

8.3.1 Experiment platform

The experiments presented in this work were performed on Edison at the National Energy Research Scientific Computing (NERSC) Center and Mira at the Argonne Leadership Computing Facility (ALCF). Edison is a Cray XC30 with a peak performance of 2.39 petaflops, 124,608 compute cores, 332 TiB of RAM, and 7.5 PiB of online disk storage. We used Edison Lustre file system (168 GiB/s, 24 I/O servers, and 4 Object Storage Targets). Default striping was used with the Lustre file system. The Mira system contains 48 racks and 768K cores and has a theoretical peak performance of 10 petaflops. Each node has 16 cores, with 16 GB of RAM per node. I/O and interprocessor communication travels on a 5D torus network. All 128 compute nodes have two 2 GB/s bandwidth links to two different I/O nodes, making 4 GB/s bandwidth for I/O at most. I/O nodes are connected to file servers through QDR IB. Mira uses a GPFS file system with 24 PB of capacity and 240 GB/s bandwidth.

8.3.2 Region-of-interest I/O

This section provides an empirical analysis of full-resolution ROI writes, with the goal of understanding the tradeoffs between storage efficiency and performance. For most of the experiments conducted in this section, the amount of data written is varied among 1%, 5%, 10%, 20%, 40%, 60%, and 80% of the entire data volume, which is achieved by making the given percentage of processes generate and write data, whereas others remain idle. The IDX block size is varied among 2^{14}, 2^{15}, 2^{16}, 2^{17}, and 2^{18}. Recall that smaller block size leads to better storage efficiency (see Figure 8.3). Furthermore, two scenarios (configurations) are identified for the distribution of processes in the global domain: *clustered*, where processes generating data are spatially close to each other (see Figure 8.5 (a)), and *scattered*, where processes generating data are spatially far from each other (see Figure 8.5 (b)).

For a given percentage of data to write, the clustered layout improves the storage efficiency; this is due to the existence of fewer secondary elements across all processes, because most of the secondary elements of a process are primary elements of an adjacent process (see Figure 8.5 (a)). Recall from Section 8.1 that secondary elements are an artifact of usage of blocks. Similarly, the scattered case leads to poor storage efficiency, due to the presence of large numbers of secondary elements across all processes (see Figure 8.5 (b)). Storage efficiency improves with reduced block sizes for both layouts.

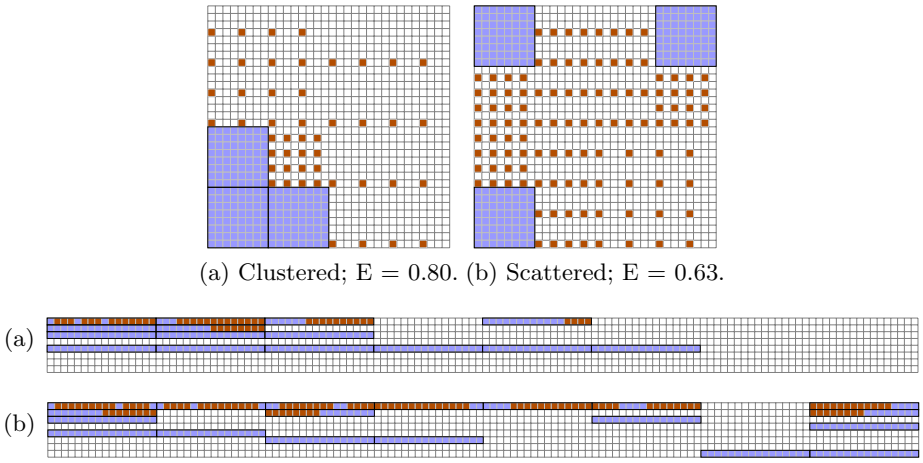
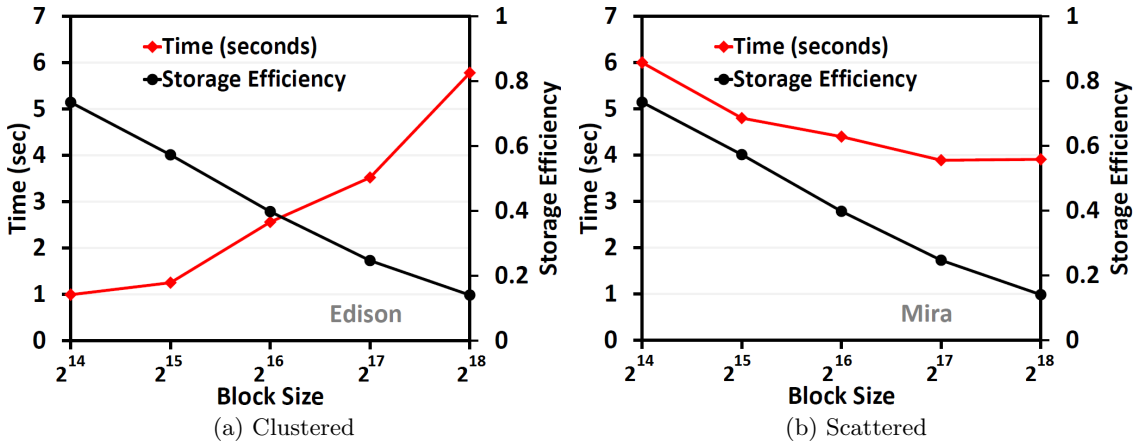


Figure 8.5: Effect of scattering of processes on efficiency. If ROI are located apart from each other, then efficiency goes down as potential secondary elements from one ROI are less likely to correspond to primary elements of another ROI. 32×32 grid; 8×8 regions; block size = 16. Efficiency is denoted as E.

8.3.2.1 Storage and performance trade-offs

The first set of experiments measures the tradeoffs between storage efficiency and performance for writing ROI data as a result of variation in block size. For this purpose, the scattered layout is used while fixing the amount of data written at 1%. This configuration in particular is interesting. It can be treated as the worst case ROI I/O scenario: a small percent of data is being written, and storage efficiency is low because the writing processes are spatially scattered. Block size is exponentially varied from 2^{14} to 2^{18}.

All experiments are conducted at a fixed core count of 4096 using the S3D I/O simulator. The S3D I/O extracts just the portion of S3D combustion simulation code concerned with restart dumps, allowing us to focus exclusively on I/O characteristics. Each of the 40 processes (1% of 4096) produced a 64^3 subvolume of double precision floating point data. This configuration produced 32MB of data within each of the 40 processes. The number of blocks per file was fixed to 256 for all runs, implying fewer files with a large block size and vice versa. Results for both Mira and Edison can be seen in Figure 8.6. Performance measured as time taken to perform the I/O operation (red trendline) is shown on the primary Y-axis (left), and the corresponding storage efficiency (black trendline) is plotted on the secondary Y-axis. A key insight is that storage efficiency improves with a smaller block count no matter what machine is used. Hence, the black trendline for storage efficiency shows an inclining trend with decreasing block size for both Mira and Edison. Interestingly,, the performance (time to write) improves with storage efficiency on Edison, as opposed to a decline on Mira. This is largely due to the metadata contention associated with writing



(a) Clustered (b) Scattered

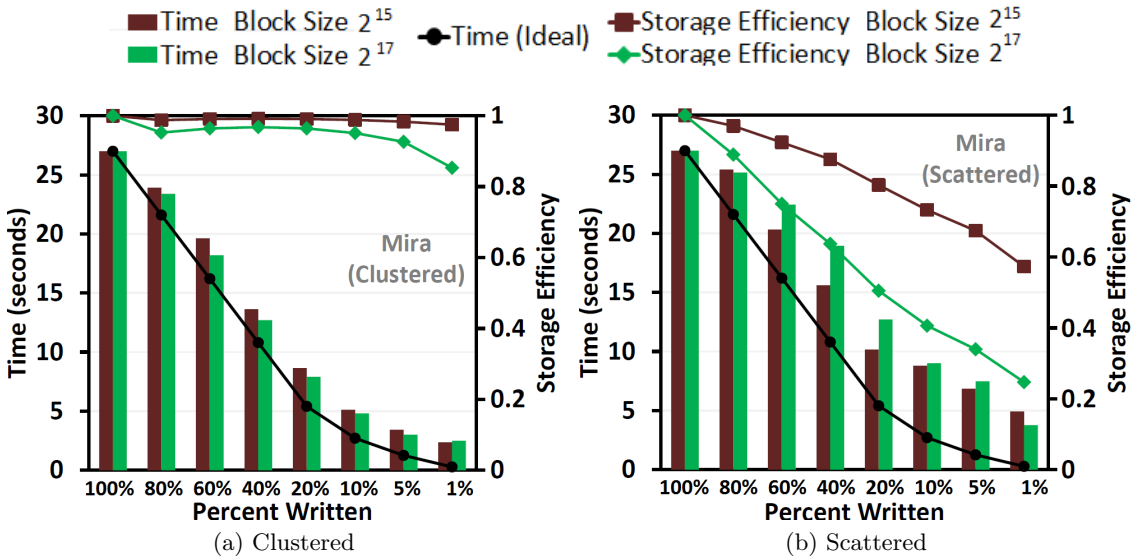
Figure 8.6: Performance evaluation and storage efficiency with varying block sizes for writing 1% of scattered layout data using 4096 cores on (a) Edison and (b) Mira.

large numbers of files on Mira. For example, a block size of 2^{14} writes 256 files as opposed to writing only 16 files for 2^{18} block size.

8.3.2.2 Storage efficiency versus performance

This section evaluates the efficacy of PIDX when performing full-resolution ROI writes with varying region sizes. An S3D I/O simulator is used to carry out all experiments. The number of core counts was fixed to 4096, while varying the percent of processes performing I/O to 1%, 5%, 10%, 20%, 40%, 80%, and 100%. For each of these cases, the processes generating data each contributed a 64^3 block of double-precision data (32MB). On Mira, block sizes of 2^{15} and 2^{17} were used that correspond to relatively lower and higher storage efficiency, respectively. For both Mira and Edison, both scattered and clustered layout were used for each percentage of data written. The results for Mira can be seen in Figure 8.7. The brown and green histograms correspond to I/O time for blocks of size 2^{15} and 2^{17}, respectively, and are shown on the primary Y-axis (left). Similarly, the green and brown trendlines correspond to the storage efficiency for the blocks sizes 2^{17} and 2^{15} shown on the secondary Y-axis. The black trendline corresponds to ideal time showing a linear decrease in time with decreasing write percentage.

For Mira with decreasing write volumes, the percent reduction in time for the clustered layout is better than the scattered layout, because the clustered layout has higher storage efficiency. Another observation is that for the clustered layout, performance for writes up to 40% of the entire volume is comparable to the ideal time. Performance starts to decrease for smaller write percents, mainly due to the lack of workload. Comparing the performance pattern across block sizes with similar storage efficiencies (i.e., the workload is similar for



(a) Clustered (b) Scattered

Figure 8.7: Performance and storage measured with varying percent write for block size 2^{15} and 2^{17} with (a) clustered layout, and (b) scattered layout.

both layouts), it can be seen that with the clustered layout, larger block size (2^{17}) results in slightly better performance compared to block size 2^{15}. This can be attributed to less metadata contention associated with creating fewer files with block size 2^{17}. The scattered layout, on the other hand, presents slightly different behavior for the two block sizes. Due to a large difference in the storage efficiency of the two blocks, the I/O performance is dominated by the total load involved. Hence, for percent writes of 60%, 40%, and 20%, the smaller block with better storage efficiency and relatively lesser work load shows better performance. The trend again starts to reverse around 10% writes, when the overall work load becomes small, resulting in better performance for block size of 2^{15} compared to 2^{17}.

8.3.3 Reduced-resolution I/O

Storing simulation results at reduced resolution can be used for fast exploratory visualization. In addition, this capability can also allow simulations to dump more frequent visualization checkpoints. The majority of simulations perform I/O dumps at intervals of several hundred times-teps, partially due to the expense incurred in parallel I/O. With PIDX, we can effectively have several reduced-resolution dumps inserted between any two full-resolution dumps. The more frequent dumps can lead to better tracking and monitoring of the simulation. Figure 8.8 shows the lifted ethylene jet dataset stored at full (top) and 1/64 (bottom) resolutions. The lower-resolution image is more efficient to store and requires less disk space, and still can give a general idea of how the simulation is progressing.

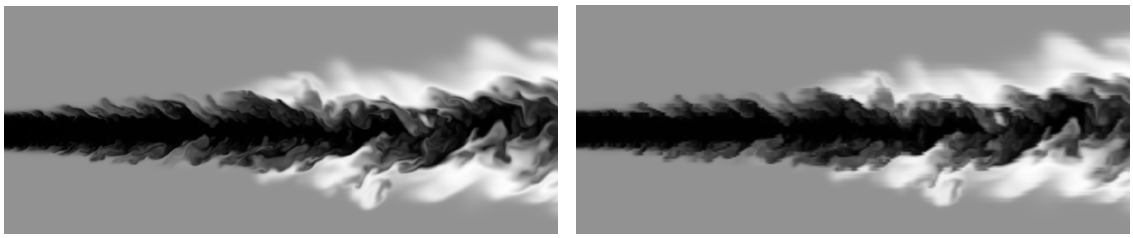
The layout of data in IDX format make it inherently efficient for performing parallel I/O at a lowered resolution. Data are laid out in increasing resolution, so access up to a given resolution level does not encounter any secondary elements, which leads to contiguous access of data. Absence of any secondary element also ensures storage efficiency is ideal. In the following section, we evaluate the performance of reduced-resolution writes.

8.3.3.1 Data scaling

This section evaluates the efficacy of writes at varying resolutions. Similar to ROI, experiments in this section were also carried out using the S3D I/O simulator. Twenty time-steps were written for each run. The number of cores was fixed at 4096, while each process contributed a 64^3 block of double-precision data (32MB at full-resolution). Resolution level was exponentially decreased from 1 to 1/64. At full resolution, each process dumps 64^3 elements; at 1/64 resolution each process dumps 16^3 elements. The experimental results for Mira and Edison are shown in Figure 8.9. Except for the last two refinements of 1/32 and 1/64, it can be seen that for Mira, write time is cut almost in half with each resolution level, demonstrating near perfect efficiency. On the other hand, with Edison, although write time does reduce with resolution, the improvement is not as high as that of Mira. This difference in behavior of the two machines can largely be attributed to the presence of dedicated I/O nodes on Mira, as opposed to shared I/O channel on Edison.

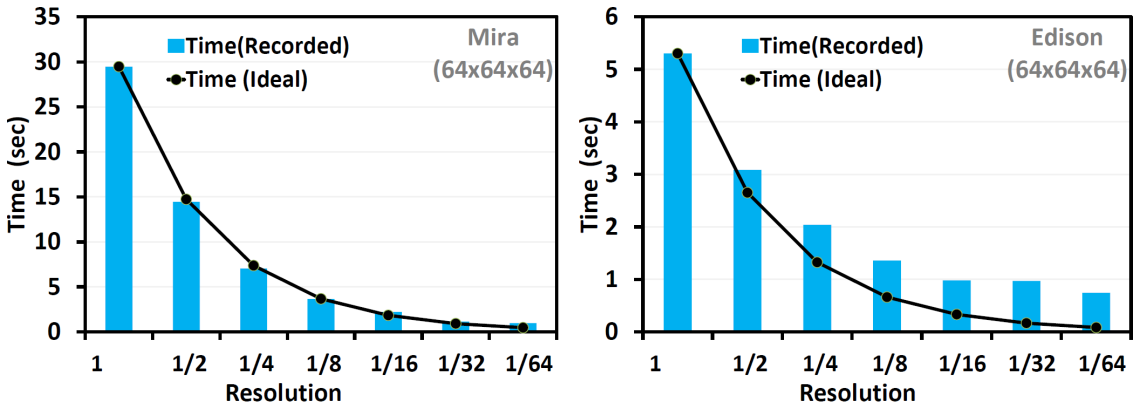
8.3.3.2 Weak scaling

This section evaluates the weak scaling performance when writing S3D datasets at reduced resolution on both Mira and Edison. In each run, S3D writes out 20 time-steps. With Edison, each process contributes a 64^3 block of double-precision data and for Mira, a process contributes a 32^3 block of double-precision data. On Edison, the number of processes was varied from 1024 to 32768, thus varying the amount of data generated per



(a) Full resolution (b) Low resolution

Figure 8.8: Slice rendering of the temperature field of the lifted ethylene jet. (a) Full-resolution data and (b) data at 1/64 resolution level.



(a) Mira (b) Edison

Figure 8.9: Reduced-resolution I/O. The histogram corresponds to the time recorded to write the dataset at varying resolution levels. The trendline corresponding to ideal time shows the ideal, linear decrease in time with decreasing resolution, and is calculated by dividing the recorded time for full-resolution writes by two for every resolution level. (a) With 4096 cores on Mira, the plot shows reduced-resolution write timings where each core has 64^3 elements. (b) With 4096 cores on Edison, the plot shows reduced-resolution write timings where each core has 64^3 elements.

time-step from 32GB to 1 TB. On Mira, the number of processes was varied from 1024 to 16384, thus varying the amount of data generated per time-step from 4 GB to 128 GB. Weak scaling runs are conducted for three resolution levels: full resolution, 1/8, and 1/64. The results for Mira and Edison can be seen in Figure 8.10. Two key trends can be observed in these results. First, parallel I/O performance scales with all resolution writes, and second, decline in throughput for 1/8 resolution writes is much more prominent than the decline in throughput of 1/64. This behavior can be attributed to the lack of adequate load to leverage the benefits of the optimizations of the PIDX API.

The weak scaling experiments show that dumps at reduced resolution are scalable, and our data scaling experiments indicate that runtime performance gains can be achieved for varying loads. When combined with the fact that disk usage has ideal efficiency, reduced-resolution writes become a compelling option for simulation monitoring and other applications where a full data dump is not necessary.

8.4 ROI I/O for S3D combustion simulation

The primary usefulness of ROI and reduced-resolution I/O is when they are combined into Adaptive ROI I/O. To demonstrate Adaptive ROI, the lifted ethylene jet, one of the largest combustion simulations performed by S3D (see Figure 8.11), is used. In particular, the temperature field is used as an initial test case. Two thresholds are used to define regions

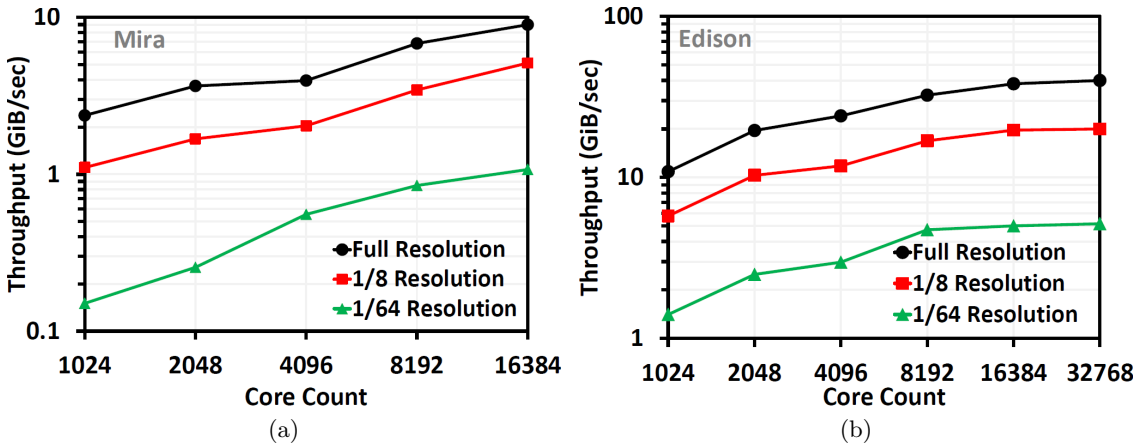


Figure 8.10: Reduced-resolution I/O. (a) Weak scaling on Mira. The number of cores increases from 1024 to 32768, while keeping the per process load fixed at 64^3, and varying the resolution as 1, 1/8 and 1/64. (b) Weak scaling on Edison. The number of cores increases from 1024 to 16384, while keeping the per process load fixed at 32^3, and varying the resolution as 1, 1/8, and 1/64.

of high, medium, and low temperature and save these at full, 1/64, and 1/512 resolution, respectively, which results in a very complex arrangement well suited to stress the system. The two flame sheets most easily distinguished on the left side of Figure 8.11 burn very hot and thus get preserved at full resolution. The outside coflow on the top and bottom is heated by the central flame and thus resides in the medium temperature region. Finally, the channel in between the sheets contains the relatively cool fuel stream, which gets classified as low temperature. Together, this configuration creates many sharp resolution drops and isolated regions as well as a significantly uneven data distribution. The resulting adaptive resolution IDX output takes only 39% of the full-resolution output time, while writing 30% of the 12.8 GB of full resolution data. As shown in the middle of Figure 8.11, the resulting volume rendering (using up-sampling to create a uniform resolution grid) preserves the ROI almost perfectly while showing the expected artifacts, especially in the center of the flame. The bottom of Figure 8.11 shows the same adaptive data without up-sampling to highlight the block distribution.

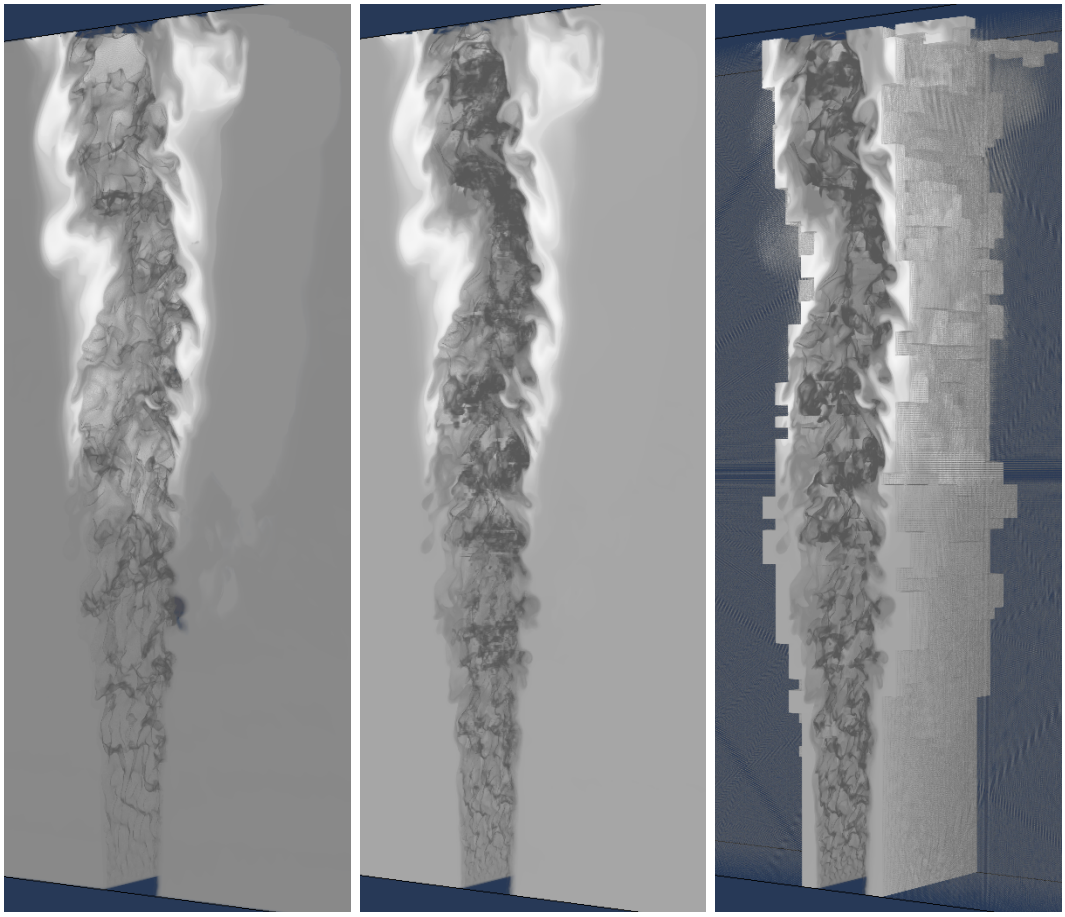


Figure 8.11: Volume rendering of the temperature field of the lifted ethylene jet. (top) Full resolution data; (middle) adaptively sampled data, up-sampled to create a uniform image; and (bottom) adaptively sampled data without up-sampling to highlight the data distribution.

CHAPTER 9

PARALLEL I/O FOR ADAPTIVE RESOLUTION SIMULATIONS

There is currently a marked trend of simulations moving towards adaptive-resolution techniques, e.g., Adaptive Mesh Refinement (AMR), to better manage multiple scales and couple detailed dynamics with large-scale behaviors. Most current high-end I/O frameworks [4, 5] are optimized for uniform grids and, in fact, adaptive-resolution grids are often simply represented and written as a collection of uniform grids at different resolutions. Such representations can result in fragmented and thus inefficient I/O. Furthermore, for convenience, many approaches unnecessarily replicate data on multiple levels, increasing the data footprint and decreasing I/O performance. This chapter presents a novel storage and I/O system for adaptive-resolution grids of an AMR simulation. In the presented approach, all AMR grids are merged into a single sparsely populated grid with data being stored adaptively. This approach to storing data has several advantages. It leads to faster data access, avoids any unnecessary data replication, and also provides both spatial and hierarchical locality. All these properties make our way of storing AMR data more suitable for analysis and visualization tasks, thus improving the overall productivity of the application developer. The chapter presents an I/O technique to write data directly from AMR simulations into the unified sparse grid. The efficacy of the method is empirically demonstrated within the framework of the parallel IDX (PIDX) I/O library. The approach is further evaluated using the Uintah block-structured AMR simulation environment [1, 76, 77].

9.1 Background on I/O strategies for AMR simulations

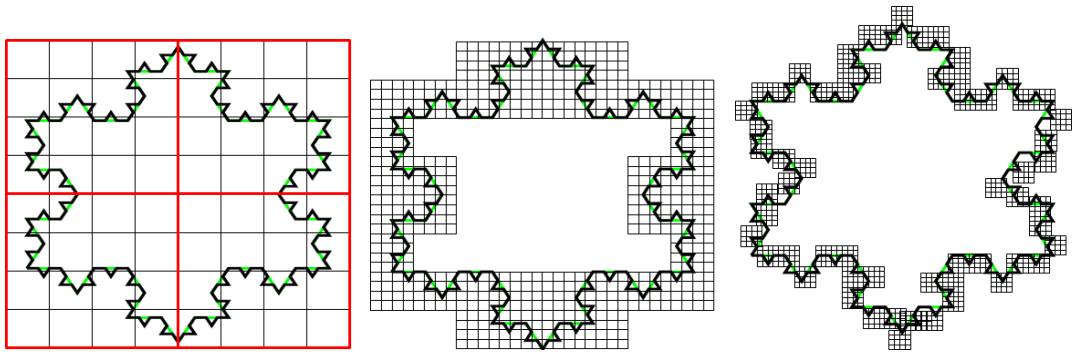
An adaptive mesh refinement simulation typically starts with a base coarse grid. As the solution proceeds, the simulation identifies the regions requiring more resolution by some parameter characterizing the solution. Finer subgrids are superimposed only on these regions. Finer and finer subgrids are added recursively until either a given maximum level of refinement is reached or the local truncation error has dropped below the desired level. Thus,

in an adaptive mesh refinement computation grid, spacing is fixed for the base grid only and is determined locally for the subgrids according to the requirements of the problem. An example of the grid structure in an AMR simulation is shown in Figure 9.1; this is an example with three *AMR levels* and an *AMR resolution ratio* of 16. Resolution ratio is defined as the ratio between the grid size of two adjacent meshes, in this case $(64\times64)/(8\times8)$. In this example, the coarse level AMR grid is decomposed into four patches. A patch is an n-dimensional rectilinear block of data; it is the smallest computational unit of a simulation. Computation and I/O tasks of a patch can be performed by only one process.

Performing efficient I/O and storage for AMR simulations is a major challenge. Most current high-end I/O frameworks are optimized for uniform grids and, in fact, adaptive resolution grids are often simply represented and written as a collection of uniform grids at different resolutions. The different AMR grids are stored separately using either file-per-process approach or the shared file approach. In the former approach, every process writes out the patches it is working on to a separate file (see Figure 9.2, left) and in the latter approach, data for every AMR grid is written to one single shared file (see Figure 9.2 (right)). Both these approaches are very inefficient; they result in fragmented inefficient I/O. The file-per-process approach fails to scale with increasing core and patch count, whereas the shared-file approach fails to scale due to global dependencies.

9.2 I/O methodology

For checkpoint dumps, it is essential to write data for all AMR grids in their entirety; however, for analysis and visualization dumps, it is sufficient to store the regions of the simulation only at their finest resolution-level. This approach makes it possible to represent all AMR grids into one flat, sparse, and adaptive grid, avoiding any data replication across



(a) AMR level 0 (b) AMR level 1 (c) AMR level 2

Figure 9.1: AMR simulation with 3 AMR levels and resolution ratio of 16. Grid size of AMR level 0, 1, 2 is 8×8, 32×32, 128×128, respectively.

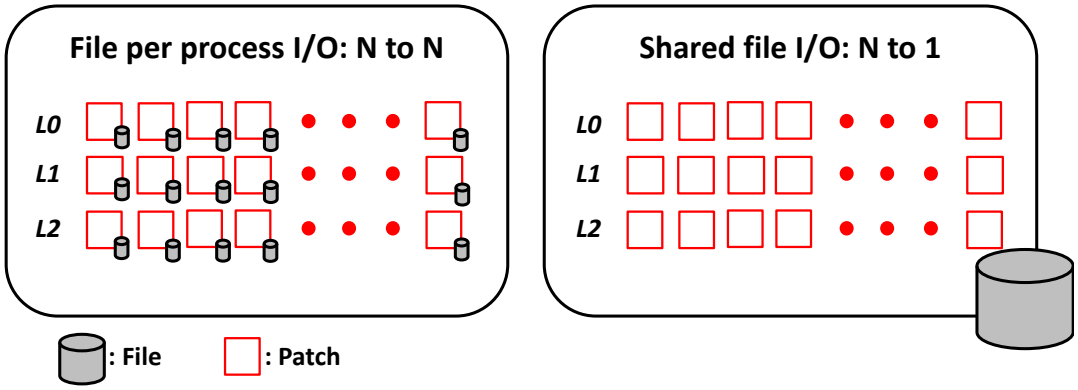


Figure 9.2: Storage strategies for AMR simulation data. (left) File per process storage strategy. (right) Shared file I/O storage strategy.

resolution levels and improving I/O performance. For data exploration, scientists are concerned only with the finest, most resolved resolution levels; therefore, besides being more optimal on write performance, this way of storing data also improves application developer productivity by providing fast and efficient read access to the finest resolution level data. Therefore, instead of storing the AMR grids as separate datasets, we merge them into one large sparsely populated adaptive grid. An example of the approach is shown in Figure 9.3.

Storing the adaptive grid can be done using different kinds of data formats. In Section 8.1 focusing on ROI dumps, we showed the efficacy of the IDX format for storing sparse datasets. At the storage level, both ROI dumps and AMR dumps are identical. They differ from each other only by the way the data are generated. With AMR dumps, adaptivity is implicit. It is something that comes inherently from the nature of the simulation, whereas with ROI dumps, adaptivity is explicit; uniform resolution data are processed in situ. Afterwards, regions are identified and then adaptively stored at different resolutions. Given

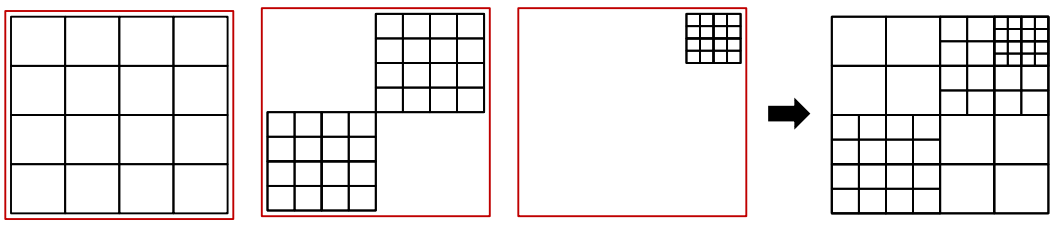


Figure 9.3: Instead of storing the AMR grids as separate datasets, they are merged in one large sparsely populated grid, with data being stored adaptively.

the similarities between the storage format of the AMR simulations grids and the ROI grids, we chose to use the IDX data format. Details on the format can be found in Chapter 3.

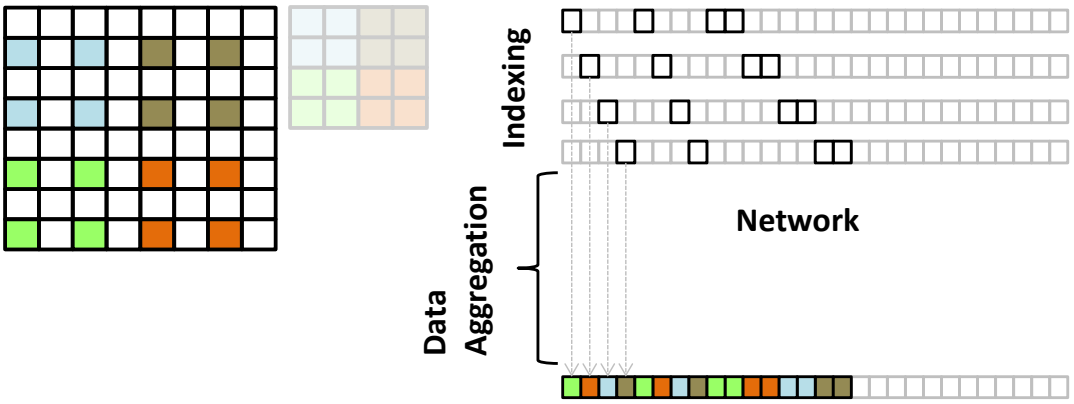
9.2.1 Data aggregation across AMR levels

With the entire simulation domain being stored only at the highest resolution level, the coarse resolution samples of the refined regions get overwritten by data from the finest resolution grid, also illustrated in Figure 9.4 using a two-level AMR grid. The size of the coarse level grid is 8×8, and the top right corner of the grid is refined with a resolution ratio of 4, yielding a second (fine) resolution grid. The four colors correspond to four processes. The processes start by working on the four patches of the coarse grid and then eventually progress to the four patches of the fine AMR grid. Note that the coarse resolution grid is shown with respect to its position in the final adaptive grid. In Figure 9.4 (b), in the final adaptive grid, the top-right four samples of the coarse resolution get overwritten by the four samples of the fine resolution grid.

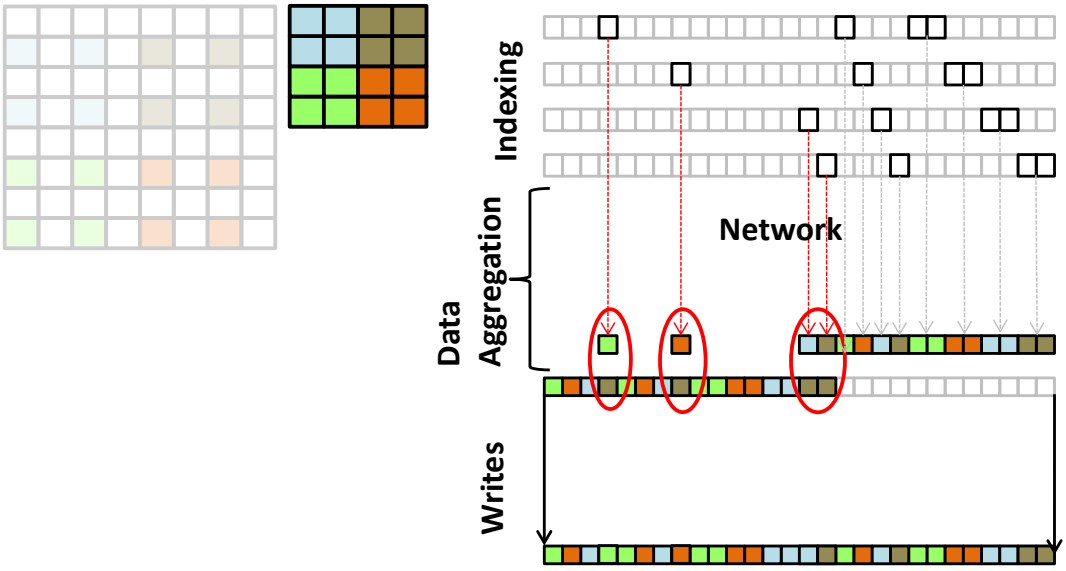
In the first prototype implementation, the AMR levels were written to disk in turn, starting from coarse to fine resolution levels. The first step is in-core data reorganization, where every process identifies the final position of all its sample in the final output file. After the reorganization phase, data are ready to be written to the file. Note that a distinct data reorganization and I/O takes place for every AMR level.

In the field of parallel I/O, a very common optimization technique is to perform data aggregation before writing by using two-phase I/O strategy. However, this approach can be applied only for the coarsest AMR level. The finer AMR levels need to overwrite some of the samples written by the coarse grids (see Figure 9.4). These samples in the finer AMR level therefore share a part of the coarsest level index space, preventing finer AMR levels of large-sized contiguous data accesses. Data at the fine AMR levels hence cannot be aggregated and is written only by making small-sized accesses. This kind of write access pattern fails to scale on big machines. We tackle this problem by developing a customized data aggregation phase spanning all AMR levels.

The Store algorithm (Figure 9.4) starts by setting up aggregation buffers for all AMR levels. It then aggregates the entire dataset before performing the disk write. Store requires memory, as the aggregation buffers store elements of all AMR levels at once, but gains performance advantages because it avoids the multiple fragmented disk writes of finer AMR levels, and instead makes fragmented data transfers from processing nodes to aggregation nodes. Store is the default algorithm used by PIDX. We compared the performance of this approach with the one that does not deploy aggregation. We performed the experiments on



(a)



(b)

Figure 9.4: Example demonstrating data aggregation across AMR levels. Data reorganization and aggregation of (a) coarse AMR level and (b) fine AMR level is followed by disk accesses. Data aggregation of the fine AMR level overwrites samples of the coarse AMR level on the fly, following which large-sized disk accesses are made.

Edison (see Section 9.3.1) with a coarse AMR level with 25^3 elements per core and a refined level at a refinement ratio of 2. Our approach with aggregation performed I/O roughly an order of magnitude faster in all cases. With deployment of aggregation strategy, at 32/64/128 cores, we saw performance go up from 4/8/12 seconds to 0.6/1/2 seconds. The key point to note is the order of iteration through AMR levels is important. Levels must be processed from low resolution to high resolution so that more refined data overwrite coarser approximations rather than vice versa.

Algorithm Store

1: **for** Each AMR level $l \in \{0, \ldots, n\}$ **do**
2: Determine write blocks
3: **end for**
4: Initialize aggregation buffers
5: **for** Each AMR level $l \in \{0, \ldots, n\}$ **do**
6: HZ encoding
7: Aggregation
8: **end for**
9: Disk write

9.2.2 Expressing AMR simulation data models with PIDX

The enhanced PIDX API allows an application to store a collection of dense multidimensional buffers corresponding to the hierarchy of the patches and levels for every variable. An example of the use of this PIDX API is shown in Listing 9.1. First is the initialization step `PIDX_create()`, where most of the relevant metadata, including the number of AMR levels and the corresponding bounds, is specified. The second step is to define a variable, using information such as its datatype and number of samples that it contains. The third step loops through all the AMR levels, defining every patch associated within the level. The patch here is also associated with the variable defined in the previous step. While remaining in the same loops, the next two steps, respectively, add the patch to the dataset and pass memory buffers and the layout to the library. These three steps can be repeated as needed to describe all variables to be included in an IDX dataset. The final phase is the `PIDX_write()` command, which tells the library to transfer the entire dataset to storage.

Listing 9.1: Example of PIDX API for AMR simulation support

```
/* defines idx for processes */
idx = PIDX_create(.....,amr_level_count, *amr_extents...);

/* define variables for processes */
var1 = PIDX_variable_global_define(``var1'', samples, datatype);

/* define patches for the variable */
patch1 = PIDX_patch_define(idx, var1, level_number, patch_number);

/* add patch1 to the dataset */
PIDX_patch_local_add(dataset, var1, patch1, global_index, count);

/* describe memory layout of patch*/
PIDX_patch_local_layout(dataset, var1, memory_address, datatype);

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

The API design provides the flexibility of having any number of AMR levels and patches, and it also continues to retain its usability for uniform resolution simulations. This organization allows PIDX to leverage as much concurrency as possible by taking all AMR

levels into account simultaneously. Note that this API can also be used to create separate IDX files for each of the AMR levels by going through the entire write cycle (`PIDX_create()` to `PIDX_write()`) level-wise.

9.2.3 Parallel HDF5

PIDX differs from HDF5 in terms of the storage format and in the read and write techniques. Whereas IDX is a data format naturally suited to multiresolution AMR datasets as well as visualization and analysis, HDF5 is, in contrast, a container, making data layout specification the application developer's responsibility. A well-designed HDF5 layout can be suitable, but is not included in the HDF5 specification. Further, IDX does not need to store metadata associated with AMR levels or adaptive ROI; hierarchical and spatial layout characteristics are implicit, whereas HDF5 requires metadata to describe data layout and extent.

The read and write techniques of PIDX also contribute to better performance. While HDF5 performs shared file I/O, PIDX breaks data into multiple files. The number of files to generate can be adjusted based on the file system. This approach extracts more performance out of parallel file systems and is customizable to specific file systems. Further, PIDX utilizes a customized aggregation phase that spans all AMR levels, leveraging concurrency and leading to more optimized file access patterns.

9.3 Performance evaluation

Our AMR experiments are done within the Uintah simulation environment. Whereas Uintah is a block-structure code [78], the presented format is suitable for any hierarchical structured data, including octree and overlapping grids.

9.3.1 Experiment platform

The experiments presented in this work were performed on Edison at the National Energy Research Scientific Computing (NERSC) Center and Mira at the Argonne Leadership Computing Facility (ALCF). Edison is a Cray XC30 with a peak performance of 2.39 petaflops, 124,608 compute cores, 332 TiB of RAM, and 7.5 PiB of online disk storage. We used Edison Lustre file system (168 GiB/s, 24 I/O servers, and 4 Object Storage Targets). Default striping was used with the Lustre file system. The Mira system contains 48 racks and 768K cores and has a theoretical peak performance of 10 petaflops. Each node has 16 cores, with 16 GB of RAM per node. I/O and interprocessor communication travels on a 5D torus network. All 128 compute nodes have two 2 GB/s bandwidth links to two different I/O nodes, making 4 GB/s bandwidth for I/O at most. I/O nodes are connected to file

servers through QDR IB. Mira uses a GPFS file system with 24 PB of capacity and 240 GB/s bandwidth.

9.3.2 Uintah simulation and I/O framework

The Uintah framework uses a structured adaptive mesh refinement (SAMR) grid to execute the solution of partial differential equations (PDEs). The component developer typically uses either a finite difference or finite volume algorithm to discretize the PDEs on the grid. The grid can be thought of as a container of AMR levels. Each level is described by a collection of patches that are distributed to individual processors/cores via a load balancing algorithm in the runtime system. See Figure 9.5. Each patch can be thought of as the fundamental unit of work that contains a region of cells at a given resolution.

In the existing I/O framework of Uintah, every process writes data belonging to a patch into a separate file. This form of I/O is an extension to the file-per process style of I/O commonly adopted by many simulations. Each MPI rank collects all the patches it is assigned. The simulation variable data for each patch is written out into a separate file along with its associated metadata file. The metadata file stores type, extent, and bounds of all the variables. For AMR with multiple levels, a directory structure is created. For relatively small numbers of patches ($< 2K$) and core counts, the I/O framework works well. However, I/O performance degrades significantly for typical simulations with several hundreds of thousands of patches/processors. The overwhelmingly large number of small files causes both writes and reads to become extremely expensive.

Including metadata files, the total number of files created every time-step is twice the total number of patches across all AMR levels. The number of patches usually exceeds the product of cores and AMR levels, leading to an allocation of multiple patches per AMR

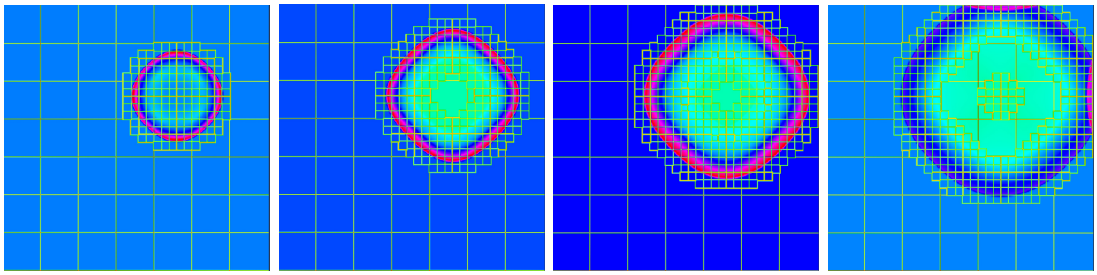


Figure 9.5: Progression of an AMR Uintah simulation of a two-level problem in time. Data were written to disk using PIDX and visualized with ViSUS. The experiment was run on Edison with a coarse grid domain of size 200×200×200. The figure demonstrates regridding of the finer AMR level, leading to an increasing number of patches. Each rectangle corresponds to a single patch. Note that the last time-step corresponds to two disconnected regions: center and ring of the fine AMR level.

level for every process. Creation of such a large number of files leads to a big overhead in metadata management. The problem intensifies as a simulation progresses, leading to the generation of newer patches at finer AMR resolutions (see Figure 9.5). Note that it is structurally not possible to coalesce patches into a bigger buffer to obtain better disk access patterns and fewer files because patches within an AMR level are not guaranteed to be spatially close to each other, and further, may form irregular shapes (see Figure 9.5).

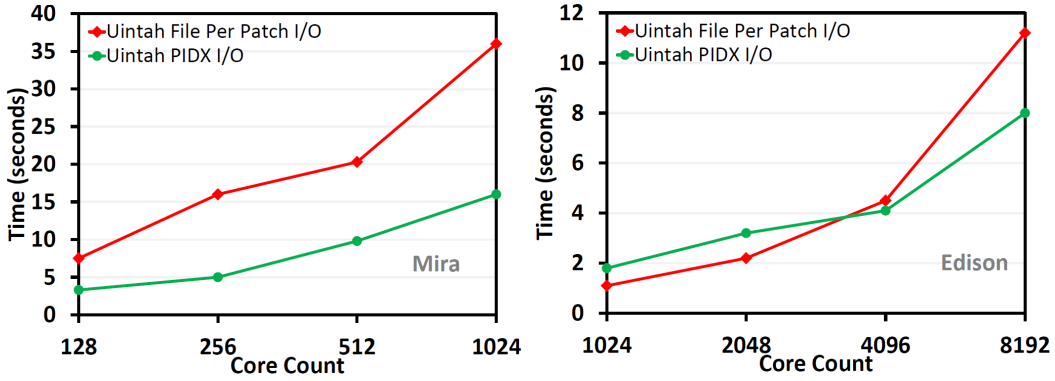
9.3.3 Weak scaling results

This section evaluates the weak scaling performance of our I/O framework when writing AMR data with two levels of refinement from Uintah simulations on both Mira and Edison. The simulation used a refinement ratio of 2, i.e., each voxel in the coarse level was refined into eight voxels at the finer refinement level. In each run, Uintah wrote out 15 time-steps consisting of four variables: pressure, temperature, density, and mixture fraction. The simulation dumps data at every 10th time-step. The patch size for the coarse refinement level is 25^3, and the patch size for the finer refinement level is 12^3. All the processes uniformly span the coarse-level grid, hence writing one coarse-level patch of size 25^3. For this particular simulation, the number of fine-level patches at the start of the run is approximately equal to the number of cores, but as the simulation progresses, the number gradually increases with regridding of the fine level.

All experiments were performed on both Edison and Mira. On Edison, the number of processes was varied from 1024 to 8192 and used a block size of 2^{14}. On Mira, processes ranged from 128 to 1024 with a block size of 2^{17}. The weak scaling results comparing Uintah with PIDX I/O and Uintah with the traditional file-per-patch on Mira and Edison can be seen in Figures 9.6 (a) and 9.6 (b), respectively.

On Mira, it can be seen in Figure 9.6 (a) that at all core counts, Uintah with PIDX I/O performs better than the traditional file-per-patch mechanism. At 1024 cores, the file-per-patch scheme takes approximately 35 seconds compared to 15 seconds by PIDX I/O. Broadly, there are two reasons for this performance behavior. 1) Metadata congestion in creating the hierarchy of files for the file-per-patch method leads to degraded nonscalable performance. 2) Fewer file creations along with the custom Store2 algorithm used in PIDX lead to a better disk access pattern and improved scalibility.

It can be seen in Figure 9.6 (a) that PIDX I/O trails Edison in performance on core counts of 1024 and 2048, primarily because the Lustre filesystem of Edison is more adept at handling large numbers of files compared to the GPFS file system of Mira. Hence, for lower-core counts, the number of files generated by file-per-patch approach is within



(a) Mira (b) Edison

Figure 9.6: Scaling results for a two-level AMR simulation. (a) Mira results for weak scaling of Uintah with PIDX I/O and traditional file-per-patch I/O. We do not report numbers to 8192 cores on Mira because the default Uintah I/O scheme fails at higher core counts (Uintah's file-per-patch approach overwhelms the I/O nodes). This problem is currently being addressed by Uintah developers. (b) Edison results for weak scaling of Uintah with PIDX I/O, and traditional file-per-patch I/O.

reasonable limits, but as the number of cores increases, the number of files generated reaches a limit that starts to saturate the metadata server. Hence, there is an observed degradation in performance at 4096 core counts. PIDX, on the other hand, generates roughly two orders of magnitude fewer files than file-per-patch I/O schemes (see Table 9.1), and this ratio remains steady even as the number of cores increases. The improved performance with PIDX can again be attributed to the custom aggregation phase that assures favorable disk access patterns.

9.4 Conclusion

Motivated by the growing need for storage techniques and I/O frameworks that support massive, adaptive datasets in a parallel setting, PIDX has been demonstrated to be an effective framework for data I/O in both AMR and uniform grid simulation environments. The experiments show that IDX is storage-efficient, amenable to I/O optimizations, and scalable. Because of its previously demonstrated spatial and hierarchical locality, reads are resolution-progressive and cache-efficient, and it is thus an excellent choice for visualization, analysis, and monitoring applications. This chapter has presented algorithms that build on existing aggregation methodologies for efficient writes, and shown that the extensions to the PIDX framework are viable for adaptive data.

Table 9.1: The number of patches generated for Edison runs, and corresponding number of files generated with PIDX I/O and file-per-patch I/O approach. With the latter approach, there is a metadata file for every patch taking the file count tally to twice the total number of patches. With PIDX, there are fewer files controlled by parameters block size and blocks per file, which are set as 32768 and 128, respectively.

Total processes	Avg patch count	File count	
	L0 + L1	PIDX	File-per-patch
1024	1024 + 1412	24	4872
2048	2048 + 3201	44	10498
4096	4096 + 5835	88	19682
8192	8192 + 10857	156	38098

PART V

CONCLUSION

CHAPTER 10

CONCLUSION AND FUTURE WORK

The increase in computational power of supercomputers is enabling unprecedented opportunities to advance science in numerous fields, such as climate science, astrophysics, cosmology, and material science. These simulations routinely produce larger quantities of raw data. A key requirement is to analyze these data and transform them into useful insight. A critical bottleneck being faced by analysis applications is the I/O time to read and write data to storage. This dissertation tackles key challenges in the field of I/O. PART I of this dissertation focuses on developing new algorithms to effectively facilitate data movement from compute resources to the storage system. We particularly demonstrate the efficacy of the transformations towards design of the Parallel IDX (PIDX) I/O library, which writes data directly in an analysis friendly data format. Highly scalable performance of the I/O library has been demonstrated on a number of HPC platforms. In addition to its performance, one of the major benefits of PIDX is that it strikes a balance between analysis-appropriate reads and large-scale writes from simulations. PART II tackles the problem of I/O tuning with the help of performance characterization and modeling. PART III presents storage and I/O techniques for adaptive-resolution data, which is demonstrated in the context of both uniform-resolution and adaptive-resolution simulations. With adaptive-resolution dumps, not only are we able to save on storage space and I/O time, but we also significantly expedite the postprocessing analysis and visualization tasks.

As we move towards exascale, I/O will pose even tougher challenges. It is expected to see higher capacity and density memory technology *(burst buffers)*, such as SSD and NVRAM, that has demonstrated better energy savings over HDDs. SSDs and NVRAMs provide significant performance advantages for random I/O access, making them good candidates for buffering, caching, and increasing memory capacity over DRAMs for tackling larger problem sizes. Although incorporating a hardware layer of faster storage devices provides the I/O software developers an opportunity for performance improvement, research on how the new devices should be used is still in its infancy.

Another key challenge as we move towards exascale is that of *resiliency*, which describes the reliability of both software and hardware in the field of HPC. Reliability has become a bigger problem as systems scale to use exascale, and the predicted component counts for systems of the future will only exacerbate this issue. Also, performance, power, and reliability are interrelated and although strict requirements are being set for performance (exaflop) and power (20 MW), reliability requirements are less constrained. The current supercomputers almost exclusively address this problem through checkpointing, but there is a concern that future systems will need more elaborate tools to achieve the goal of resilience.

Focusing on the research done during the course of this dissertation, there are several potential topics to extend this work. The core of this research has focused on rectilinear grids. Potentially new transformations need to be developed for simulations with particle data. Similar to rectilinear meshes, a big challenge while handling particle data will be to balance the trade-off between large-sized writes and analysis-appropriate reads. There needs to be a storage strategy that can facilitate fast and efficient access of particle data. Research needs to be done to further optimize and scale I/O for AMR simulations. This dissertation tackles AMR simulations that have aligned grids, but newer algorithms need to be designed to tackle unaligned grids. Another possible research avenue would be to leverage different compression strategies to support I/O scaling. One direction would be making the current framework work with lossless compression. Also, data access can be optimized further by using an adaptive bit rate for each chunk, depending on the local entropy distribution of the data.

In conclusion, the research done as part of this dissertation has led to the development of an efficient, flexible, and high-performing I/O software layer. This development has been made possible with I/O transformations that expose tunable parameters that can control the nature of flow of data across the network, all the way to the storage system. Using the customizable I/O transformations for a multiresolution, analysis-friendly data format, we also established for the first time a fast and interactive read component (optimized for latency) in the otherwise write-heavy (optimized for bandwidth) I/O software stack.

REFERENCES

[1] J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *Ninth IEEE International Symposium on High Performance and Distributed Computing*. IEEE, Piscataway, NJ, November 2000, pp. 33–41.

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

[3] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud *et al.*, "A practical method for estimating performance degradation on multicore processors, and its application to hpc workloads," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 83:1–83:11.

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

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

[6] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE '08*. New York: ACM, June 2008, pp. 15–24.

[7] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli *et al.*, "PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets," in *IEEE International Conference on Cluster Computing*, 2011.

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

[9] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla *et al.*, "In-situ feature extraction of large scale combustion simulations using segmented merge trees," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 1020–1031.

[10] S. Kumar, V. Pascucci, V. Vishwanath, P. Carns, R. Latham *et al.*, "Towards parallel access of multi-dimensional, multiresolution scientific data," in *Proceedings of 2010 Petascale Data Storage Workshop*, November 2010.

[11] S. Kumar, V. Vishwanath, P. Carns, J. Levine, R. Latham *et al.*, "Efficient data restructuring and aggregation for I/O acceleration in PIDX," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–11.

[12] S. Kumar, A. Saha, V. Vishwanath, P. Carns, J. A. Schmidt *et al.*, "Characterization and modeling of pidx parallel I/O for performance optimization," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 67:1–67:12.

[13] S. Kumar, C. Christensen, J. Schmidt, P.-T. Bremer, E. Brugger *et al.*, "Fast multiresolution reads of massive simulation datasets," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer International Publishing, 2014, vol. 8488, pp. 314–330.

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

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

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

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

[18] "Intrepid home page," https://www.alcf.anl.gov/intrepid.

[19] "Mira home page," https://www.alcf.anl.gov/mira.

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

[21] "Lustre home page," http://lustre.org.

[22] "Hopper home page," https://www.nersc.gov/users/computational-systems/hopper/.

[23] "Edison Dragonfly topology," https://www.nersc.gov/users/computational-systems/edison/configuration/interconnect/.

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

[25] "Preparing applications for mira, a 10 PetaFLOPS IBM blue gene/q system," http://www.alcf.anl.gov/files/PrepAppsForMira_SC11_0.pdf.

[26] "Cray XT5," http://en.wikipedia.org/wiki/Cray_XT5.

[27] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala *et al.*, "Looking under the hood of the IBM Blue Gene/Q network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 69.

[28] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski *et al.*, "Plfs: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 21:1–21:12.

[29] V. Vishwanath, M. Hereld, and M. Papka, "Toward simulation-time data analysis and I/O acceleration on leadership-class systems," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, Oct 2011, pp. 9–14.

[30] W. Yu, J. S. Vetter, and H. S. Oral, "Performance characterization and optimization of parallel I/O on the cray XT," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2008, pp. 1–11.

[31] M. Fahey, J. Larkin, and J. Adams, "I/O performance on a massively parallel cray XT3/XT4," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2008, pp. 1–12.

[32] B. Xie, J. Chase, D. Dillow, O. Drokin, S. Klasky *et al.*, "Characterizing output bottlenecks in a supercomputer," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 8:1–8:11.

[33] W. Yu, S. Oral, J. Vetter, and R. Barrett, "Efficiency evaluation of cray XT parallel io stack," in *Cray User Group Meeting (CUG 2007)*, 2007.

[34] P. C. Roth, "Characterizing the I/O behavior of scientific applications on the cray XT," in *International workshop on Petascale data storage (PDSW)*, ser. PDSW '07. New York, NY, USA: ACM, 2007, pp. 50–55.

[35] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms *et al.*, "I/o performance challenges at leadership scale," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 40:1–40:12.

[36] M. Oberg, H. M. Tufo, and M. Woitaszek, "Exploration of parallel storage architectures for a blue gene/l on the teragrid," in *9th LCI International Conference on High-Performance Clustered Computing*, 2008.

[37] U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: modeling the characteristics of rigid jobs," *J. Parallel Distrib. Comput.*, vol. 63, no. 11, pp. 1105–1122, Nov. 2003.

[38] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, ser. HPCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 340–351.

[39] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS*, 2010, pp. 129–142.

[40] B. Lee, R. Vuduc, J. Demmel, and K. Yelick, "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," in *International Conference on Parallel Processing (ICPP)*, 2004, pp. 169–176 vol.1.

[41] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 557–558.

[42] S. Lakshminarasimhan, D. A. Boyuka, S. V. Pendse, X. Zou, J. Jenkins *et al.*, "Scalable in situ scientific data encoding for analytical query processing," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 1–12.

[43] B. Behzad, J. Huchette, H. Luu, R. Aydt, Q. Koziol *et al.*, "Abstract: Auto-tuning of parallel io parameters for hdf5 applications," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, 2012, pp. 1430–1430.

[44] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of hpc applications using a parameterized synthetic benchmark," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008, pp. 1–12.

[45] E. K. Lee and R. H. Katz, "An analytic performance model of disk arrays," in *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '93. New York, NY, USA: ACM, 1993, pp. 98–109.

[46] A. Uselton, M. Howison, N. Wright, D. Skinner, N. Keen *et al.*, "Parallel I/O performance: From events to ensembles," in *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[47] P. Hanuliak, "Analytical method of performance prediction in parallel algorithms," *Open Cybernetics & Systemics Journal*, vol. 6, pp. 38–47, 2012.

[48] K. Barker, K. Davis, and D. Kerbyson, "Performance modeling in action: Performance prediction of a cray xt4 system during upgrade," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2009, pp. 1–8.

[49] D. Feng, Q. Zou, H. Jiang, and Y. Zhu, "A novel model for synthesizing parallel I/O workloads in scientific applications," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2008, pp. 252–261.

[50] "VisIt home page," https://wci.llnl.gov/codes/visit/.

[51] "ParaView home page," http://www.paraview.org/.

[52] V. Pascucci, G. Scorzelli, B. Summa, P.-T. Bremer, A. Gyulassy *et al.*, "The ViSUS visualization framework," in *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, E. W. Bethel, H. Childs, and C. Hansen, Eds. CRC Press, 2012.

[53] V. Pascucci, G. Scorzelli, B. Summa, P.-T. Bremer, A. Gyulassy *et al.*, "Scalable visualization and interactive analysis using massive data streams," *Advances in Parallel Computing: Cloud Computing and Big Data*, vol. 23, pp. 212–230, 2013.

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

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

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

[57] Y. Tian, S. Klasky, W. Yu, B. Wang, H. Abbasi *et al.*, "Dynam: Dynamic multiresolution data representation for large-scale scientific analysis," in *Networking, Architecture and Storage (NAS), 2013 IEEE Eighth International Conference on.* IEEE, 2013, pp. 115–124.

[58] J. P. Ahrens, J. Woodring, D. E. DeMarle, J. Patchett, and M. Maltrud, "Interactive remote large-scale data visualization via prioritized multi-resolution streaming," in *Proceedings of the 2009 Workshop on Ultrascale Visualization*, ser. UltraVis '09. New York, NY, USA: ACM, 2009, pp. 1–10.

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

[60] V. Pascucci and R. J. Frank, "Hierarchical indexing for out-of-core access to multi-resolution data," in *Hierarchical and Geometrical Methods in Scientific Visualization.* Springer, 2003, pp. 225–241.

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

[62] "Surveyor home page," https://www.alcf.anl.gov/surveyor.

[63] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale *et al.*, "FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes," *Astrophysical Journal Supplement*, vol. 131, pp. 273–334, 2000.

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

[65] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes *et al.*, "Terascale direct numerical simulations of turbulent combustion using s3d," in *Computational Science and Discovery Volume 2*, January 2009.

[66] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization & Computer Graphics*, no. 12, pp. 2674–2683, 2014.

[67] P. Shirley and S. Marschner, *Fundamentals of Computer Graphics*, 3rd ed. Natick, MA, USA: A. K. Peters, Ltd., 2009.

[68] D. Hearn and M. P. Baker, *Computer graphics, C version.* Prentice Hall Upper Saddle River, 1997, vol. 2.

[69] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice (2Nd Ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[70] L. Williams, "Pyramidal parametrics," *SIGGRAPH Comput. Graph.*, vol. 17, no. 3, pp. 1–11, Jul. 1983.

[71] "Tukey home page," https://www.alcf.anl.gov/tukey.

[72] A. Landge, J. Levine, A. Bhatele, K. Isaacs, T. Gamblin *et al.*, "Visualizing network traffic to understand the performance of massively parallel simulations," vol. 18, no. 12, 2012, pp. 2467–2476.

[73] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[74] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion *et al.*, "Scikit-learn: Machine Learning in Python ," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[75] K. Gao, W.-K. Liao, A. Nisar, A. Choudhary, R. Ross *et al.*, "Using subfiling to improve programming flexibility and performance of parallel shared-file I/O," in *International Conference on Parallel Processing, 2009. ICPP '09*, September 2009, pp. 470–477.

[76] S. G. Parker, J. Guilkey, and T. Harman, "A component-based parallel infrastructure for the simulation of fluid-structure interaction," *Engineering with Computers*, vol. 22, pp. 277–292, 2006.

[77] S. G. Parker, "A component-based architecture for parallel multi-physics PDE simulation." *Future Generation Computer Systems*, vol. 22, pp. 204–216, 2006.

[78] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah DAG-Based runtime system on PetScale supercomputers," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, pp. 96:1–96:12.