Taming the Scientific Big Data with Flexible Organizations for Exascale Computing

by

Yuan Tian

A dissertation proposal submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
Aug 4, 2012

Keywords: Data Organization, Temporal and Spatial Aggregation, ADIOS, Climate Modeling, S3D, Parallel File System

Approved by

Weikuan Yu, Assistant Professor of Computer Science and Software Engineering
Scott Klasky, Group Lead of Scientific Data Group at Oak Ridge National Laboratory
Xiao Qin, Associate Professor of Computer Science and Software Engineering
David Umphress, Associate Professor of Computer Science and Software Engineering

Abstract

The last five years of supercomputers has evolved at an unprecedented rate as High Performance Computing (HPC) continue to progress towards exascale computing in 2018. These systems enable scientists to simulate scientific processes with great complexities and consequently, often produce complex data that are also exponentially increasing in size. However, the growth within the computing infrastructure is significantly imbalanced. The dramatically increasing computing power is accompanied with the slowly improving storage system. Such discordant progress among computing power, storage and data has led to a severe I/O bottleneck for the advancing of scientific computing. While intensive research for the next generation storage is undergoing, a revolutionary upgrade to current back-end storage systems is not foreseeable in the near future. As a result, applications become more reliant on I/O software in hoping to alleviate the performance bottleneck through driving the storage system at its full speed. Efficient I/O for scientific big data is crucial for a successful transition into exascale for HPC.

However, providing a high performance I/O at software layer is nontrivial. The large volume, high complexity and mismatch between the organization of scientific data and underlying storage system pose grand challenges for I/O software design. This dissertation investigates the characteristics of scientific data and storage system as a whole, and explores the opportunities to drive the I/O performance for petascale computing and prepare it for the exascale. To this end, a set of flexible data organization and management techniques are introduced to address the I/O challenges from five directions, namely system-wide data concurrency, in-node data organization, complex I/O patterns, time dimension analytics and asynchronous compression. For these purposes, four key techniques are designed to exploit the capability of the back-end storage system for processing and storing scientific big data

with a fast and scalable I/O performance. It has been shown that these techniques can contribute to the real world scientific applications with enhanced I/O performance and scalability for end-to-end data flow. It also contributes as part of the solution towards scalable data management techniques while high performance computing is progressing into exascale.

Acknowledgments

First and formost, I would like to thank my advisor, Dr. Weikuan Yu for his insightful guidance and generous support of my doctoral career. His invaluable comments and feedbacks of my research are what made this thesis possible. It has been a privilege working with him.

Additionally I'd like to give special thanks to my mentor Dr. Scott Klasky. He introduced me to research in scientific computing and helped focus my ideas through his knowledge of science. I would also like to extent my deepest gratitude to Dr. Umphress. As my Master's program advisor and the committee member of doctoral program, he has always supported me with his guidance, patience and encouragement. I also would like to thank my committee member Dr. Xiao Qin and my university reader Dr. Shiwen Mao for their time, patience and suggestions that led to me improving this work. I would also gratefully thank Dr. Kai Chang for his generous support throughout my years at Auburn.

I would like to thank my fellow students at PASL lab for their generous help on my research and kindly encouragement. It has been a great pleasure working with this group of wonderful students. I would like to thank ADIOS team for their help to my research and moral support. I am also deeply grateful for my dearest friends Bing Qi, Chun Guo and Wei Ren. You girls made Auburn home to me.

My deepest gratitude goes to my parents Liaofa Tian and Xiaoheng Zhou, my brother Shan Tian, and sister-in-law Wenxi Zhuang. Your love and support have always been there with me, and are the reasons I could pursuit my dreams. Finally, I am deeply grateful for my best friend, Charles Joseph, for holding my hands through the darkest nights. Your love lights up my world.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's competitive research environment, high performance computing plays a significant role in driving the cutting edge research, such as high energy physics, climate and earth system simulation, new material development, etc. With the fast-growing computational power of supercomputers, scientific simulations are able to realize much finer granularity in their simulations, thereby opening up new horizons for making more scientific discoveries. Massive data is generated and consumed by large-scale simulation codes, from checkpoint-restarts, monitoring, analysis, and visualization work flow. However, the evolving of leadership scale infrastructure is significantly imbalanced. The increasingly powerful computing platform is accompanied with the slowly improving back-end storage system. Such growth mismatch becomes even more distinct with the continuous progressing of silicon technology. As a result, the disparity between the load of the storage system and its capability has became a severe bottleneck for the further advance of scientific computing.

To alleviate the such bottleneck, parallel file systems such as Lustre [20], GPFS [83], PVFS [68], and Panasas file system [23], are designed to provide scalable parallel I/O solution. Similar efforts can be found in hardware vendors such as DDN, NetApp and IBM. They continue to provide highly concurrent storage hardware to increase the aggregated throughput. However, the scaling of current storage system does not match with the computing front-end, mainly due to the energy and cost constrains. Therefore, the limitation of such techniques becomes evident as system continues to scale. While much effort has been devoted to the next generation storage hardware, e.g. Solid State Device (SSD) citessd and Non-volatile random-access memory (NVRAM) [29], a massive replacement of magnetic disks in back-end storage systems is not foreseeable in the near future. Recent emerging

co-design paradigm is to bring the computer scientists and domain scientists together to design the computer hardware, software and algorithms that accommodate the computational requirements of applications. However, the existing scientific codes cannot benefit from such technologies. Instead, simulations are becoming increasingly reliant on software and algorithmic advances to maintain acceptable I/O costs.

Significant effort has been undertaken to design I/O softwares that can improve parallel I/O performance. Earlier examples include MPI-IO [91], ROMIO [7]. Numbers of I/O libraries, e.g HDF-5 [57], NetCDF4 [94], ADIOS [1], PnetCDF [47] are also designed to further speedup I/O. Such effort has shaded merits in a large number of applications. Particularly the write performance has been driven to close to system peak for many applications. However, there are still a wide range of applications with complex I/O patterns that are not adequately addressed by these techniques. Moreover, majority of current I/O approaches only focus on the write side of issues as it is directly related to the simulation run-time. Read performance is often overlooked, despite its importance in driving scientific insight through scientific simulation. Scientific tasks such as analyzing the simulation output to gain scientific understanding, finding critical features within the dataset and producing meaningful results are inherent parts of the scientific data workflow. Therefore, it is also important to provide a good read performance in supporting efficient data post-processing.

Addressing aforementioned issues are challenging on current storage systems, particularly at scale. The reasons are detailed in Chapter 2. However, it should be noted, that the major issue comes from the discrepancy between data and storage. The implication is that the storage system does not well support current organization and management of data, leading to limited I/O performance. For writing, current I/O techniques are able to drive the peak performance when aggregated bandwidth of storage is well utilized for output, e.g. through constructing the storage-friendly large sequential writes. However, for cases where such approaches fail to bridge the disparity, e.g. complex I/O patterns, a significant performance degradation is observed. With much less attention being given, the read performance

is further exacerbated due to data is laid out in storage without sophisticated planing, or with a strategy that is only favorable towards write performance. The result of such oblivion is that the storage system is, in fact, *working against* the data, instead of *working with* the data.

To provide a high performance I/O service for both data generation and consumption, this dissertation investigates the challenges and opportunities to find and develop the *harmony* between data and storage. In particular, it address the aforementioned issues from five directions include: 1) data concurrency of storage system; 2) in-node data organization; 3) efficient data analysis in time dimension; 4) applications with challenging I/O patterns; 5) enabling compression for scientific data. To this end, this dissertation introduce a set of system-oriented data organization and management techniques that are designed to develop the a harmonious organization between data and storage. It focuses on coordinating the massive volume of data to fully exploit the capability of storage system. It consists of four key I/O techniques:

1. **Space Filling Curve Based Data Reordering** is designed to ensure a new-maximum data concurrency from storage system for common access patterns of scientific application.

2. **System-Aware Two-Level Data Organization** further guide the data organization on a specific storage node under the governs of an Optimized Chunking model.

3. **Spatial and Temporal Aggregation** is a novel technique that is proposed to improve the I/O performance for applications that are not yet benefit from current I/O techniques. Meanwhile an efficient data analytics on time dimension is enabled.

4. **Nearline Data Compression and Decompression** is to investigate enabling computationally-heavy compression algorithms for scientific data during simulation runtime through asynchronous I/O technology.

The rest of the dissertation is organized as follow. Chapter 2 gives a detail discussion of background and motivation. Chapter 3 provides an overview of related work. Chapter 4 to Chapter 7 present the detail design and evaluation of each data organization and management technique. Chapter 8 discusses the future work of this study. Finally, the dissertation is concluded in Chapter 9.

Chapter 2

Statement of The Problems

This chapter presents a detail discussion of the challenges to be answered in this dissertation. Firstly, the insufficiency of common practices in addressing the write and read issues is presented. Then it illustrates the organization disparity between scientific data and storage system, followed with the performance issues because of such mismatch. Finally the opportunities of further improving I/O performance through asynchronous compression is discussed.

2.1 Why Common Practice Are Not Sufficient?

In the past few years, many scientific applications are being transmitted to large-scale systems by adopting I/O middleware that are designed to leverage the parallel storage system. Recent emerging co-design paradigm is to bring the computer scientists and domain scientists together to design the computer hardware, software and algorithms that accommodate the computational requirements of applications. However, many applications have complex data characteristics that are not well supported by existing parallel I/O libraries. For example, applications may generate large number of small variables. In parallel, each process only holds very small amount of data for each variable. It is challenging to provide a good I/O speed for both writing and reading.

NASAs climate and weather models such as GEOS-5 (the Goddard Earth Observing System Model) and Model-E are such applications. To achieve high resolution in their simulations, and to accumulate fine-grained observational data, they require very efficient computational and I/O resources for data generation, management, movement, analysis, reanalysis, assimilation, and visualization. An I/O technique that can significantly enhance

the spatial and temporal resolution of these climate and weather models can systematically lift the capability of climatologists and meteorologists in addressing critical climate issues. For example, as one of the most important models, GEOS-5 desires to routinely run high-resolution simulations (e.g., 3km and beyond). It demands a very efficient parallel I/O infrastructure to reduce the I/O time of simulations.

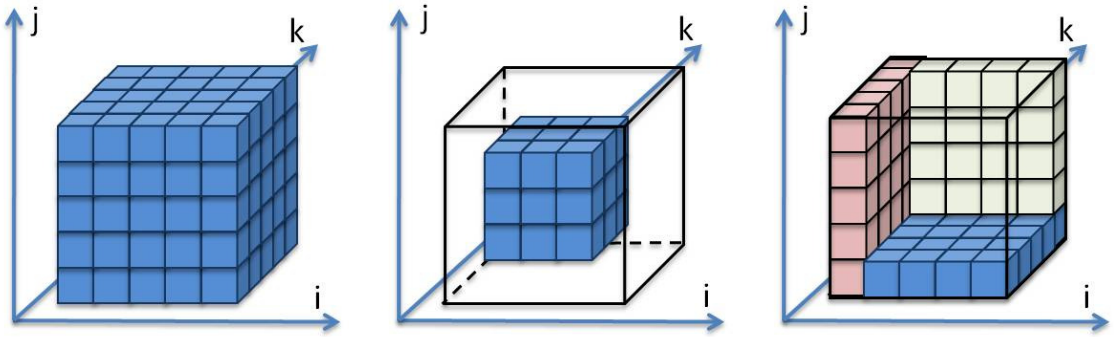
Moreover, it is also quite common for data post-processing to examine data along time dimension, e.g. observing the change of temperature within one hour. But current common data layouts do not provide a good support for such access pattern. A normal practice is to store data of different time steps in different files, or in different blocks within the same file. Because the data segments are scattered within the file, a large number of read requests have to be made to retrieve data back, degrading the storage performance. An efficient data organization that facilitates the temporal exploration is highly demanded.

2.2 How Data is Accessed?

Large-scale simulation codes can generate massive multidimensional datasets, from checkpoint-restarts, monitoring, analysis, and visualization output. Their execution is often bottlenecked by the cost of I/O because of their gigantic datasets. Many efforts have focused on decreasing the application turnaround time by studying the output side of the problem, but few have systematically examined the read performance of scientific applications on large-scale supercomputers, despite the importance of read performance to scientific simulation and analysis workflows.

To improve read performance, a thorough understanding of application access patterns is crucial. Based on our direct experience with many application teams in the U.S. and beyond, including combustion (S3D [16]), fusion (GTC [39], GTS [97], XGC-1 [13]), earthquake simulation (SCEC [22]), MHD (pixie3D [12]), numerical relativity codes (PAMR [75]), and supernova (Chimera [58]) codes, there are four main fundamental reading patterns for application data analysis:

- Read all of a single variable (c.f. Figure 2.1(a)). This would be representative of reading the temperature across a simulation space, for example.
- Read an arbitrary orthogonal subvolume (c.f. Figure 2.1(b)).
- Read an arbitrary orthogonal full plane (c.f. Figure 2.1(c)).
- Read multiple variables together. This would be representative of reading the components of a magnetic field vector, for example.



(a) Variable (b) Subvolume (c) 2-D Planes

Figure 2.1: A $5 \times 5 \times 5$ Array (k: fastest dimension)

Other reading patterns are either composed of a mixture of these patterns or are minor variations. For example, reading entire checkpoint-restart datasets can be perceived as an extended case of reading multiple variables together. Figure 2.1 gives an example of a $5 \times 5 \times 5$ 3-D array (Figure 2.1(a)), a $3 \times 3 \times 3$ subset (Figure 2.1(b)) of the array and three 5×5 planes (Figure 2.1(c)) in three dimensions: i, j, and k; where i is the primary, i.e., the slowest varying dimension, and k is the tertiary and fastest varying dimension. Data in the array is stored first along the fastest dimension k, then along the slow dimensions, j and i, on disk. Among these patterns, reading orthogonal planes has been the least studied. However, it is a very commonly used data pattern by scientific applications. For example, for combustion studies with S3D [32], the computation was targeted at a variable of $1408 \times 1080 \times 1100$ points (12GB), but the majority of analysis is performed on the orthogonal planes of the variable (either 1408×1080 or 1080×1100 points). However, the performance of reading such planes

(a.k.a *planar read*) is often bottlenecked by the extremely poor performance in retrieving data along slow dimensions from multidimensional arrays [18].

2.3 Why Read is Slow?

There are two main issues faced by aforementioned access patterns of multidimensional arrays. The first is the disparity between the order of storage and the order of access for data points in a multidimensional variable. When data is not traversed in the order in which it is stored, reading cannot benefit from techniques such as data prefetching, caching, etc. For example, when reading an ij plane, data points are stored non-contiguously along these slow-varying dimensions. Current disk does not provide a good support for access in this manner, so that the read performance degrades significantly. However, this does not happen to jk planes whose data points are stored along the fast-varying dimensions. Such disparity leads to a phenomenon is often referred as "dimension dependency" [86], where the read performance depends on the dimensions involved in a query, rather than just the data size of the query. A current popular solution to this problem is to store multiple copies of the same data with a different dimension being used as the primary dimension in each copy. For example, climate researchers at the Geophysical Fluid Dynamics Laboratory (GFDL) make multiple replicas of all datasets with x, y, z and *time* as the fastest dimension, respectively. Such workarounds help reduce the reading time [26], but increase the total storage size by 4 times. Second, there is a lack of data concurrency when a subset of data points is retrieved from a multidimensional variable. When the subset of data points is not logically contiguous, it is often concentrated on only a few storage devices among a large number of total devices that are used to store the entire variable. For example, with the logically contiguous data layout, a plane in the fast dimension will be located on only one storage target if the plane size is less than the stripe size. In this case, applications can not make use of aggregated bandwidth from all devices, and are then limited to the bandwidth available from a single storage device.

Currently there are two popular data organizations: logically contiguous (LC) and chunking. A collection of research has been carried out for better approaches on efficiently storing and indexing multidimensional array. Among them, a strategy called *Chunking* has become a standard for multidimensional array storage. This approach partitions an array into many smaller units called chunks. Chunking has been proven to be able to alleviate the dimension dependency by providing an opportunity to cache the entire data chunk for request on the slow dimension(s). The rational behind such layout is the cost efficiency of the disk processing overhead. However, chunking still has its limitations when applying on scientific data. First, chunking is lack of support for two common cases in scientific applications: 1) For applications that generate gigantic datasets per process, chunks can grow very large to the extent in which data points inside a chunk are sparsely located for certain visualization and analytics tasks. For example, retrieving a column from a 2-D chunk with 1200*1200 data points requires reading more than 94% redundant data or 2000 seek and read operations. 2) For applications that generate small data per process, small chunks can be sparsely retrieved within the logical space. Such as a 2-D 1200*1200 logical space consists of 100*100 chunks, a range query could result more time to spend on seeking across small chunks than actually retrieving data. Moreover, the traditional sequential placement of data chunks on storage targets has a high possibility to result in limited data concurrency for many common access patterns. To overcome these issues of chunking and enable fast access of multidimensional scientific datasets without dimension dependency, it is important to investigate a new data organization.

2.4 Data Reduction and I/O

Scientific applications running on large-scale systems often have a great deal of redundancy in their data sets. For example, the checkpoint data from different time steps may contain substantial similarity. The observations show that the datasets from a real S3D simulation have compression ratios ranging from 1.5 to 876 via the gzip utility. Data with such

internal redundancy is being transmitted as applications go through many phases of data collection, transport, analysis and visualization. A good portion of network I/O bandwidth is squandered during different phases of application execution, while many of CPU cores spend their time waiting on the data movement to be completed.

With the growing gap between the speed of multicore processors and that of I/O devices, it would be desirable if this disparity can be mitigated by leveraging the computing power to compress and consolidate data, thus saving the network bandwidth and relieving the data pressure on I/O devices. Along this line of considerations, scientists often undertake cumbersome post-processing steps to reduce the size of their simulation data. The back-end archival and storage system for a supercomputer can also be used to identify and purge redundant data through special hardware and software offerings. Both these approaches are typically performed in an offline manner, i.e., outside the normal execution of scientific applications. They can neither benefit the I/O efficiency at runtime, nor reduce the consumption of network bandwidth for applications. Other parallel I/O techniques, such as HDF5 [57] and NetCDF-4 [38, 62], support data compression. But the compression is carried out in an isolated manner by individual processes. Worse yet, data compression (while computationally intensive) is performed inline with respect to the productive computation, degrading the overall performance of applications. Thus it is critical for the I/O software stack to have a solution that can take advantage of the computing power from multicore and many-core processors, achieve asynchronous data compression, and reduce the consumption of network bandwidth when storing data to back-end storage devices. By mitigating part of the I/O pressure to computing cores, both write and read performance can be speeded up.

2.5 Summary

In summary, this thesis seeks to provide a high-performance and scalable I/O performance for scientific big data through addressing following directions:

- The system-wide data concurrency

- The in-node data organization
- The multi-dimensional aggregation
- The data layout facilites the data analytics on time dimension
- Data reduction through asynchronous I/O pipeline

The solutions are:

- Space Filling Curve Based Data Organization
- Two-level System-aware Data Organization
- Spatial and Temporal Data Aggregation
- Nearline Data Compression and Decompression

Chapter 3

Related Work

Improving I/O performance on large scale systems has been one of the major research topics in HPC. This section gives a review for the past efforts that emphasized on the topics that are related to the research presented in this thesis.

3.1 Parallel I/O

Significant efforts have been undertaken to improve the performance of parallel I/O. ROMIO [7] provides the most popular implementation of the parallel I/O interface, MPI-IO [91]. It implements a variety of techniques including extended two-phase I/O [90], split-phase collective I/O [25] and disk-directed I/O [41]. MPI-IO/GPFS [76] and MPI-IO/BlueGene [83] have introduced MPI-IO optimizations that are designed to take advantage of the specific features of General Parallel File System (GPFS) and BlueGene [6]. Bent et al. [10] from Los Alamos National Laboratory developed a parallel log-structured file system (PLFS) to support efficient checkpointing of data-intensive applications on the RoadRunner system [9]. PLFS makes use of FUSE as an interposition layer to support concurrent writes to a logically shared file with many physically separated files. This is to avoid the serializing side effects of the original Panasas file system [23] in handling concurrent writes to a shared file. PLFS does not address the issue of concurrent writes from multiple processes. It also requires the insertion of a kernel module to interpret and translate metadata. between PLFS and Panasas file system.

There are a number of recent attempts to optimize parallel I/O on large-scale systems. Yu et al. [102, 100] have carried out a series of optimization studies on the Cray XT platforms. Yu et al. [101] also show the benefit of hierarchical file striping in allowing data access to

multiple subfiles instead of a single shared file. A similar approach has been explored by Liao et al. [48] in the Parallel-NetCDF project. This paper builds on top of the earlier work of hierarchical striping to provide elastic and evenly distributed subfiles. It organizes individual data files (subfiles) in a hierarchical and elastic manner so that compressed data chunks can be striped to all storage devices, without causing the overhead of wide striping. Tatebe et al. [88] have exploited the concepts of local file view to maximize the use of local I/O bandwidth in the design of a distributed file system for the Grid environment. Klasky et al. [40] and Ma et al. [55] have investigated the I/O performance benefits of multithreading. Abbasi et al. [4] and Zheng et al. [104] have showed the work of expediting scientific applications by I/O staging.

Some research have investigated how to improve the read performance of data-intensive applications. Lofstead et al evaluate and understand the performance of many of the reading patterns for extreme scale science applications [52]. Many approaches have explored data staging and caching to either bring data *a priori*, or buffer data temporarily, respectively, in anticipation of performance savings of future data access. For example, staging has been exploited for grid-based scientific computing [45, 17]. The staging approach adopted for grid environments is quite different from what is applicable to closely integrated systems such as supercomputers. The PreDatA [104] system creates a staging area in which data can be prepared through annotation, filtering, indexing, and organization, for efficient post-analysis. Zazen [93] makes intensive use of more storage devices, and caches simulation data as a series of small files across multiple disks of a networked analysis cluster. In doing so, it improves the read performance of data analysis. However, neither PreDatA or Zazen examines the performance of data reordering strategies for I/O performance improvements.

3.2 Data Organization

Researchers have investigated different data organizations for multidimensional scientific data. Same interest is shared in other domains, particularly in database domain, where

multidimensional array is commonly used for data storage. For example, log-based data organization is exploited for databases [31] and various file systems [78, 85, 96]. Sarawagi et al. [80] categorized the strategies for efficient organization of large multidimensional arrays into four classes, namely chunking, reordering, redundancy, and partitioning. While chunking has been commonly recognized as an efficient data layout for multidimensional arrays because of its capability of alleviating dimension dependency [86].

A line of work indicated in order to ensure good I/O performance, reorganizing data chunks is necessary. Many multidimensional array declustering algorithms [67], [74], [14], [15] were proposed to improve common access patterns of a multidimensional array. Schlosser et al [82] explored the chunk placement strategy at the disk level. Much more efficient access was observed by putting data chunks on the adjacent disk blocks. Deshpande et al [24] examined how chunking the large dataset and caching the chunks with a modified LRU algorithm can speedup reading. However, the future access pattern for scientific application varies and may not be known *a priori*. A data layout should accommodate any access pattern. Fan et al. [28] proposed a latin cube strategy to put neighborin data points into one shared memory module to improve I/O performance. Because of the natural thinking of the disk traversal, the logically contiguous format has been adopted by many popular I/O libraries including NetCDF versions 3 [62] and 4 [94], HDF5 [57], and PnetCDF [47]. Through the terascale era, this worked extremely well. In fact, HDF5 can achieve excellent performance [103]. With the size and complexity of modern storage arrays, the read performance for 3-D arrays for restart purposes does not measure up for the logically contiguous format, compared to a log-based format [72]. Chunking can be further combined with partitioning, a.k.a *subchunking*, which further decomposes the data chunk. In [80], Sarawagi and Stonebraker gave an initial guidance of what the proper way for chunking would be. Then Sawires et al [81] proposed a multilevel chunking strategy to further improve the performance for range queries on a multidimensional array. Otoo et al [66] mathematically calculated the optimal size of subchunks from a combination of system parameters. However, the study was based on very limited

resource, and did not reflect the reality on modern petascale systems. A group of researchers proposed SciDB [11] as a data management system particularly intended for applications involving large scale array processing. They used an overlapping chunks to speed up queries on the subset of data. The performance was gained by more requirement for storage without introducing more complexity to applications. In [84], Sorouch et al proposed ArrayStore, a two-level regular and irregular data chunking layout to speedup queries in database system.

Space filling curves are widely used [8] because of their good spatial locality properties, especially in spatial database researches such as [77, 30]. Lawder et al. [46] have explored different kinds of space filling curves to develop indexing schemes for data layout and fast retrieval in multidimensional databases. Pascucci and Frank [70] have used a global indexing scheme to reorder regular grids based on Lebesgue's space filling curves so that the performance of progressively rendering of multidimensional datasets was improved. Jagadish et al. [37] have examined the linearization of multidimensional data chunks through space filling curves. They have concluded that the Hilbert Curve outperformed all other curves. Moon et al. [59] have demonstrated that the Hilbert Curve can achieve better clustering than both z- and gray-coded curves in two-dimensional and three-dimensional spaces. Hu et al. [35] have used a Hilbert curve for data reorganization and achieved substantial improvements in the performance of fine-grained irregular applications on shared memory systems. Kuo [43] et al. also used Hilbert curve to reorganize the multidimensional datasets. However, they focused on examing the subarray, which is less impacted by the data distribution, and the scale of study is not comparable to today's petascale systems.

3.3 Data Deduction

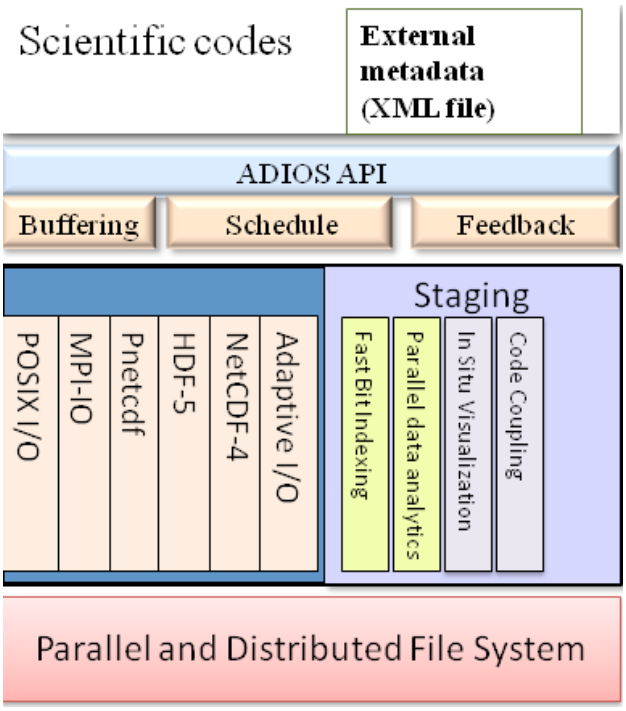
Many studies have examined how to improve I/O through data compression. Park et al. [69] developed a CZIP compression scheme in a file system. It is based on the Content-based naming (CBN) technique to eliminate redundant chunks. Vilayannur et al. [95] employed the content-addressable concept into the design and implementation of a file system,

CAPFS (Content Addressable Parallel File System). Lakshminarasimhan et al [44] proposed an in-situ compression approach for spatial-temporal data. Potential gains of data compression can be greater at the MPI-IO level for scientific applications because it reduces the requirements on both networking bandwidth and storage capacity. Our work represents an attempt in this direction.

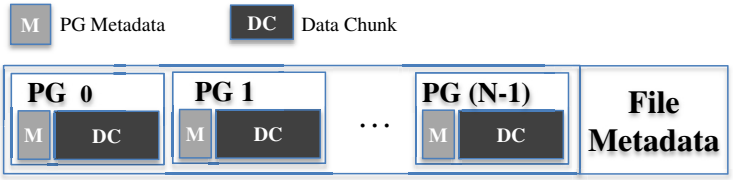
Multiresolution techniques have been widely studied in image processing and visualization domain. The fundamental idea is to abstract a much larger dataset with into limited size of data. Among all the algorithms, Wavelet [19] is a technique to abstract the signal of an image or a dataset into a small subregion and store the filtered signal residuals in the other hierarchical regions, leaving the space for high compression ratio. It was first introduced by Mallat in [56]. Since its successful application in JPEG2000 [89, 87], Wavelet has been widely used in large image and large volume data compression and visualization. Poulakidas et al [73] proposed a wavelet based image compression algorithm for fast image subregion retrieval. Ihm [36] et al introduced a 3-D wavelet algorithm for large volume data rendering and achieved a very good compression ratio. In [33], Guthe et al investigated interactively rendering of very large data sets on standard PC hardware. The data was compressed using a hierarchical Wavelet Transform and decompressed on-the-flay while rendering. Yu et al [99] et al proposed a wavelet-based time-space partition (WTSP) tree algorithm to achieve high compression ratio for large datasets on time domain. [21] proposed a toolkit named VAPOR which applied wavelet transform on scientific data for multiresolutional purpose. In general, most of the wavelet-based algorithms require a preprocessing step for data preparation and are computationally expensive. For scientific applications which have restrict I/O performance requirement, data representation is not the only concern. Pascucci et al proposed IDX [71] for visualization of scientific data through downsampling along Z-order Space Filling Curve. They showed good read and write performance [42]. However, the issue with this work along with other downsampling-based algorithms is, the points of interest may be lost during the subsampling.

3.4 Adaptable I/O System

A significant work of this thesis is built upon Adaptable I/O System (ADIOS) [64], an I/O middleware from Oak Ridge National Laboratory. An overview of ADIOS and its BP (Binary Packed) file format is provided in this section.



(a) ADIOS Architecture



(b) BP File Format

Figure 3.1: Adaptable I/O System

ADIOS provides a simple, flexible way for scientists to manage the data that may need to be written, read or processed during simulation runs. The key techniques used by ADIOS such as external XML data description file, run-time selection of I/O transport method, data caching, etc have successfully helped ADIOS to gain high I/O performance on many

petascale systems such as Jaguar [60], NERSC [61]. ADIOS employes a data caching to buffer the metadata and real data until overflows. Thus a large amount of small writes are avoid. An external XML file is used to describe user data, e.g., data types, sizes and I/O operations. A key feature of this XML file is the selection of which transport method, such as MPI-IO, POSIX, or HDF5, should be used for each grouping of data in the application. As such, the user can select at runtime how to process the data without requiring any code changes. In particular, the XML includes data hierarchy, data type specifications, process grouping and how to process the data. This configuration is read on code startup and, based upon the settings in this XML file, the data will be processed differently. For example, the user could select MPI-IO to write out a single shared file for analysis output and POSIX I/O to write out one file per process for diagnostic data. Different setups enable the I/O to behave differently without having to either change or recompile the source code. The primary goal of this I/O system is to offer a level of adaptability such that the scientist can change their I/O transport simply by changing a single attribute in the XML file. A high-level illustration of ADIOS is shown in Figure 7.2.

ADIOS has demonstrated significant performance benefits for a number of petascale scientific applications [4, 49, 54, 104, 72]. It uses a default file format called BP. In this format, ADIOS applies the chunking strategy for storing multidimensional datasets. Each process is assigned one data chunk from one dataset after domain decomposition. If the application needs to output multiple datasets, each process will have multiple data chunks, one from each dataset. A group of chunks from one process, along with their attributes such as data size and offsets, are grouped together and stored as a larger unit, called *Process Group* (PG) in ADIOS. , also referred as *Data Group*. An example of BP file output for one dataset written by N processes is shown in Figure 3.1(b). All PGs are placed within BP file in the order of process IDs.

Chapter 4

Space Filling Curve Based Data Reordering

To enable fast access of multidimensional scientific datasets, it is important to investigate a strategy that can 1) provides maximum aggregated bandwidth for common access patterns of scientific application, and 2) overcome the disparity between data organization of scientific data and physical storage. In this study, we propose an I/O framework named Elastic Data Organization (EDO) that can support flexible data organization strategies for different scientific applications. EDO addresses the challenging issues faced by common access patterns, particularly planar reads, through its elastic data organization algorithms. By default, EDO uses a chunking based data organization to balance the cost of seek operations and extra data retrieval. At chunk level, EDO uses Hilbert Space Filling Curve (SFC) to balance the distribution of data groups (a.k.a process groups) across storage targets. As we will discuss in Section 4.2, such strategy distributes data elements across parallel storage targets to aggregate their bandwidth without file system restrictions and improve concurrency for the access patterns discussed in section 2.2. The data reordering algorithm does not precludes portability nor requires any application-level changes. They are used to decide the placement of data elements for optimal concurrency and better exploitation of bandwidth from storage devices.

In this chapter, we first discuss the motivation of this work in Section 4.1. We then describe the design of EDO in Section 4.2. Section 4.3 provides a mathematical analysis of data concurrency using different data organization strategies. Section 4.4 further validates our strategy through a comprehensive set of experimental results. Finally, we summarize this work in Section 4.5.

4.1 Motivation

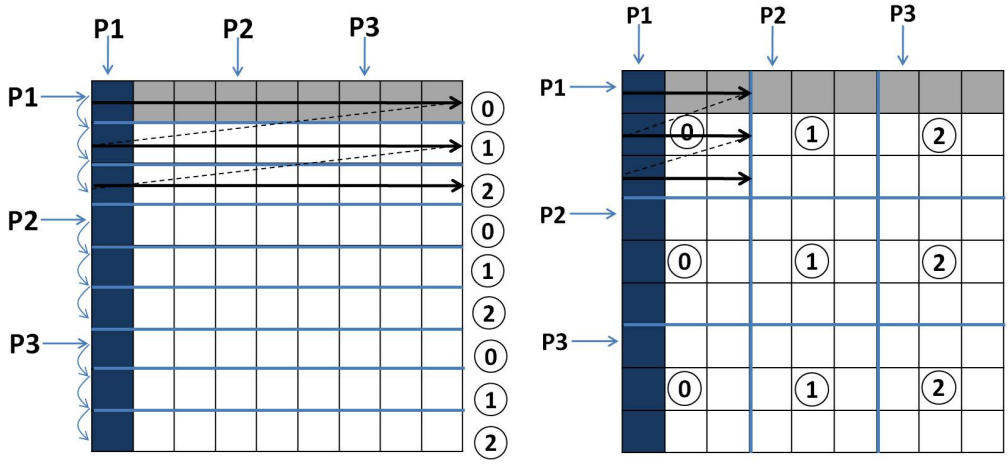
In this section, we present a short discussion of existing data organizations and their performance issues.

When an application needs to retrieve data elements from a multidimensional dataset, two main factors affect the read performance. One is the contiguity of these data elements, another is the number of concurrent storage devices that are supplying the data. The former determines the maximum number of seek operations, though the actual number of seeks may be reduced by reading extra data between data elements. The latter determines the *concurrency* of storage access in an application.

Currently there are two popular data organizations: logically contiguous (LC) and chunking. Figure 4.1 compares these two data organizations and show how the read performance can be different between these organizations. In the figure, a 2-D array with 9×9 integer elements is written on three storage targets using LC and chunking, respectively. The stripe width is equal to 36 bytes. The arrowed lines represent the order in which these data elements are stored on storage devices, e.g., Object Storage Targets (OSTs) in the case of Lustre file system. The circled numbers indicate targets on which data elements are located; the shaded squares are the requested data elements. If we read in the row-major order with three processes, for both organizations, each process needs 1 seek operation and 1 read operation to retrieve the data. However, chunking is expected to be 3 times faster than LC since LC serializes read requests from three processes to one OST.

Concurrency issues are also observed for LC when retrieving a column, as shown in Figure 4.1. Furthermore, the performance is degraded because each process either has to perform 3 seeks and 3 reads to retrieve the data, or one read to get 19 elements at a time. The former is slow due to frequent expensive seek calls; the latter is also inefficient since 84% extra data is retrieved. Chunking can not help for such retrieval patterns as well. Every process either needs the same number of seek and read operations to obtain the requested data, or retrieves 67% extra data. Meanwhile, chunking suffers from similar concurrency

issues. Processes are contending at a small number of storage targets (only *OST0* in this case).



(a) LC (b) Chunking

Figure 4.1: Current Data Organizations

4.2 Design of SFC-based Elastic Data Organization

In view of the performance issues of existing data organizations, we design EDO as an extension of ADIOS to support elastic data organization algorithms. EDO retains the chunking data organization of ADIOS, and many of its salient features , including NSSI [65] for staging, DataTap [5] for asynchronous I/O, and Dataspace [27] for memory-to-memory code coupling. Figure 4.2 shows the software architecture of EDO. It focuses on enabling data reordering algorithms for different scientific applications. Varying strategies are provided as selectable algorithms to determine the placement of data units in EDO. These include the default linear placement, Hilbert Space Filling Curve (SFC). It also provides a flexible interface to enable other algorithms such as Z-order, random placement. As Hilbert curve has been proven to show best declustering capability among all the Space Filling Curves [37]. This study focuses on this algorithm. In the rest of the section we discuss the Hilbert SFC algorithm in detail, and describe how it is used to determine the data organization in EDO.

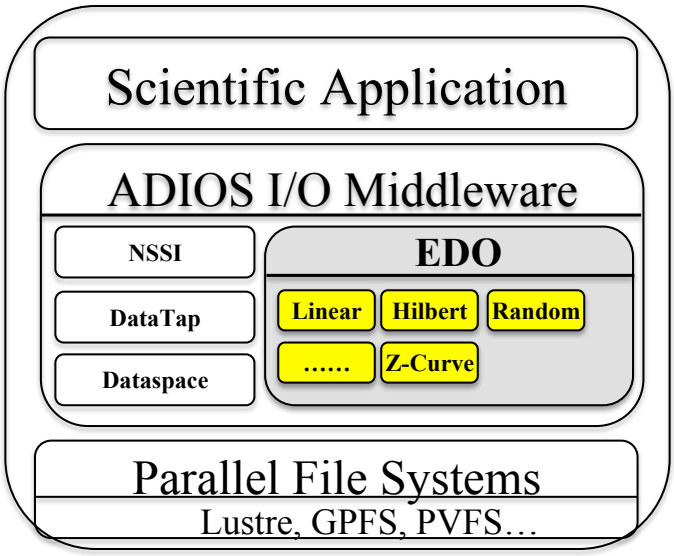


Figure 4.2: EDO Architecture

4.2.1 Hilbert Curve Data Reordering

EDO provides a Hilbert Space Filling Curves (SFC) based chunk ordering strategy to balance the distribution of data groups (i.e., PGs) across storage targets.

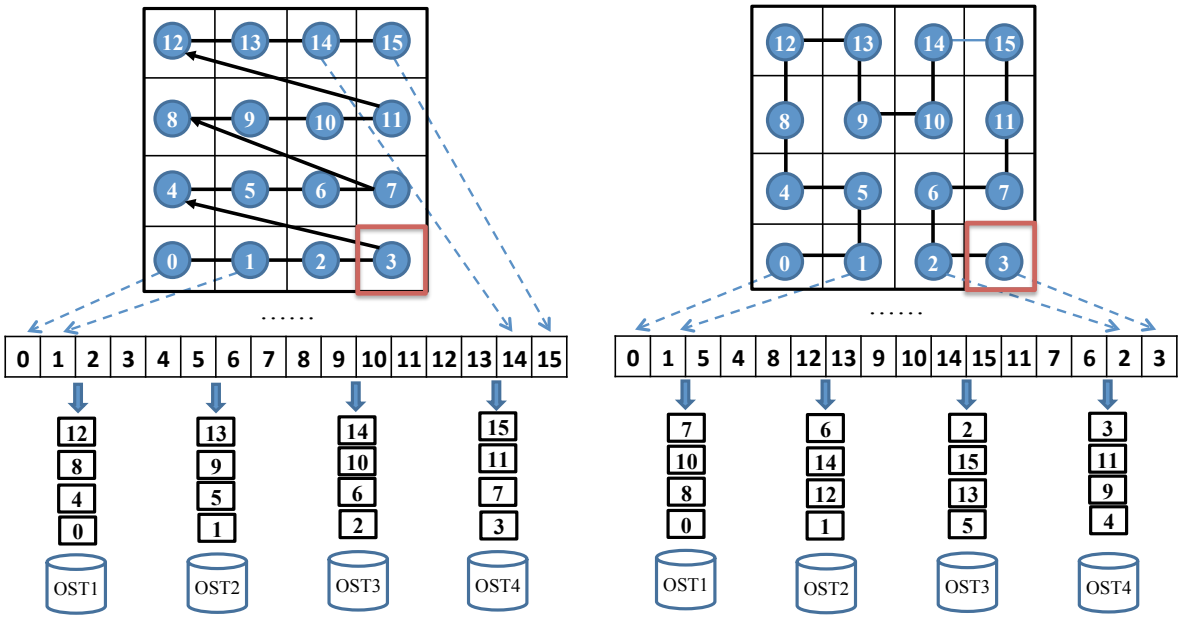
Figures 4.3(a) and 4.3(b) compare data organization between the linear placement (as in ADIOS) and the Hilbert SFC-based placement as introduced in EDO. A 2-D array of 16 chunks is used here as an example to simplify the description. These chunks are written to 4 storage targets (OSTs) via 16 processes.

In the original ADIOS, each PG is placed on one storage device in a round-robin fashion. Good concurrency can be achieved in row-major order because sequential PGs are placed on different OSTs, leading to good data distribution. However, such placement may face severe concurrency issues when data is accessed along the slow-varying dimensions, similar to the case in Figure 4.1(b), .

A better method is needed to order PGs so that data chunks on any dimension can be clustered, i.e., achieve good data locality. We use the Hilbert *Space Filling Curve* (SFC) [34] because it is a strategy to map a multidimensional space onto a one-dimensional space, and

has been used in a wide variety of applications, especially when data locality is of concern. The Hilbert curve guarantees the best geometric locality properties [63]. Additionally, the cost of transforming the index of data units is low [37], a strength also shared by other strategies such as the morten (z-curve) index.

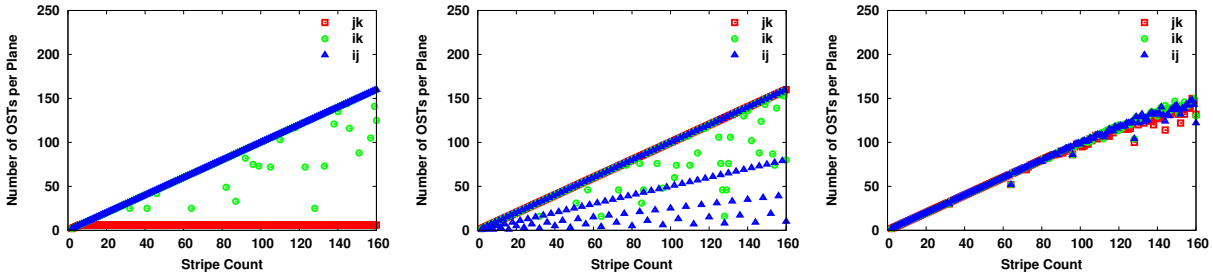
The 2-D array on the left of Figure 4.3(b) shows the placement of PGs using Hilbert curve. Using this algorithm, data chunks are shuffled among all OSTs. For example, instead of writing to OST2, chunk 5 will be placed on OST3, while chunk 14 will be placed on OST2 instead of OST3. This strategy does not impact the read performance of the entire array because the number of total storage targets remains the same. A slight impact is observed for the row-major order, such as the first and second rows, where data chunks are now placed on fewer storage targets. However, a significant difference is shown for data access in the column-major. Instead of striding among multiple rows, many data chunks become sequentially organized. Thus when one column is requested, the targeted data chunks will spread to 3 OSTs, compared to only 1 OST under the original data organization. Thus the Hilbert curve can improve concurrency for data planes from slow-varying dimensions.



(a) Linear Placement in ADIOS (b) SFC-based Placement with EDO

Figure 4.3: Comparison of Different Data Organizations

4.3 Analytical Modeling of Data Concurrency



(a) Logically Contiguous (b) Linear Placement (c) Hilbert Curve

Figure 4.4: Concurrency Modeling of a 2-D Plane on 3 Dimensions

To validate the function of new organization algorithms in EDO, we analytically model the performance of different data organizations, where a 2-D plane is retrieved from a 3-D dataset. We first introduce one formula to quantify the data concurrency. Our study is based on the Lustre [20] file system. The concurrency of a plane is determined by the number of OSTs that data elements are placed. While the placement of data elements are determined by their offsets within the file and striping parameters. Thus, we introduce the following formula to calculate the concurrency:

$$Concurrency = |\alpha_0 \bigcup \alpha_1 \bigcup ... \bigcup \alpha_i|, where$$
$$\alpha_i = \frac{offset_i}{stripe\_width} \% stripe\_count, i \in [0...(n-1)] \quad (4.1)$$

where n is the number of data elements on the plane. Note that the offset of data element is different under different data organizations.

Based on the above formula, several programs are developed to calculate the concurrency of reading 2-D planes from a 3-D data array. We then compare the average concurrency across three dimensions. Figure 4.4 shows the result of concurrency when reading 2-D planes from an $800 \times 800 \times 800$ 3-D global array. The initial array is created by 4,096 processes. Each process writes $50 \times 50 \times 50$ elements either as a 3-D chunk or a contiguous segment,

depending on the data organization. The stripe size is 1MB. With such dataset, we vary the stripe count from 2 to 160, the maximum stripe count allowed by Lustre, and compare the data concurrency on three dimensions using different organization strategies. Note that the theoretical maximum concurrency for a plane equals to the configured stripe count for the output file. An optimal data organization should show a concurrency that is equal or close to the maximum concurrency on all dimensions.

Under the logically contiguous organization, shown by Figure 4.4(a), the average number of OSTs for jk planes is consistently small. Severe contention will occur if such plane is retrieved by many processes in parallel. The data concurrency of ik planes varies. In some cases, It drops to less than half of the stripe count. The ij planes have the highest concurrency. Note that the highest concurrency does not necessarily mean the best performance which also depends on the contiguity of the read operations.

Figure 4.4(b) shows the data concurrency using the linear placement of data chunks. The jk planes have the highest concurrency, i.e., they are able to use the maximum number of OSTs. Due to high variances of concurrency on the ik and ij planes, low concurrency for these two planes is common. For example, when a stripe count is set to the maximum 160 on Lustre, where the highest aggregated write bandwidth is expected, the resulting concurrency for planar reads is rather low to 80 and 10 for the ik and ij planes, respectively.

Figure 4.4(c) shows the data concurrency of Hilbert curve. All three types of planes show close-to-optimal concurrency, more than 76% of the available storage targets are utilized. There is little variation among different types of planes. Thus, a balanced performance can be expected. When the stripe count for an array is set to 160, the Hilbert curve organization is able to achieve a concurrency of 132, 130 and 122 OSTs, respectively for jk, ik, and ij planes. There are cases that the linear placement can utilize more OSTs than the Hilbert curve, for instance when the stripe count is 121. But such sweet spots require either a sophisticated calculation that involves the number of read processes, the stripe count, the stripe size, the size of dataset, and/or an extensive set of tuning experiments. Worse yet,

one set of parameters will not fit for different multidimensional data arrays. Application scientists would rather be relieved from having to understand such calculation or go through time-consuming tuning experiments. The Hilbert curve-based ordering frees them from such performance concerns. Application scientists can then intuitively choose the striping parameters for their datasets, and/or expect a consistent and well balanced read performance in return.

4.4 Experimental Performance Results

We deploy EDO on the *Jaguar* supercomputer at Oak Ridge National Laboratory (ORNL) to evaluate its performance. Jaguar is currently the second fastest supercomputer in the world [60]. (Note that the experiments are conducted before it is upgraded to current Cray XK6 system). Jaguar is a massively parallel, distributed memory system composed of a 2.3 PetaFlop/s Cray XT5 partition and a 263 TeraFlop/s Cray XT4 partition, a 5 PB file system known as *Spider*. The Cray XT5 partition is used for our experimental evaluation. It contains 18,688 compute nodes besides login/service nodes. Each compute node contains dual hex-core AMD Opteron 2435 (Istanbul) processors running at 2.6GHz, 16GB of DDR2-800 memory, and a SeaStar 2+ router. The entire partition contains 224,256 processing cores and 300TB of memory. The Spider file system is the largest Lustre file system in the world, with 672 storage targets (OSTs) on its widow-1 section and over 26,000 clients, and it is the fastest Lustre file system in the world with a demonstrated bandwidth of 240 GB/s.

A self-contained I/O kernel for S3D [16] from Sandia National Laboratories is used in our experiments. S3D is a high-fidelity, massively parallel solver for turbulent reacting flows. It employs a 3-D domain decomposition to parallelize the simulation of combustion. S3D generates datasets of different sizes. Four test cases: small (S), medium (M), large (L) and extra large (X) are shown in Table 4.1.

According to the previous practice with ADIOS on Jaguar, the stripe size is set as the size of PG, a technique that maximizes concurrency and reduces false sharing on the Lustre

Table 4.1: Test Cases Written by 16×16×16 Writers

	Per Process		Entire Array	
	Elements	Data Size	Elements	Data Size
S	20^3	62.5KB	320^3	250MB
M	50^3	0.95MB	800^3	3.8GB
L	100^3	7.6MB	1600^3	30.5GB
X	250^3	119.2MB	4000^3	476.8GB

file system. A separate test program is created to read planes and subvolumes from the logically contiguous file format.

We measure the read performance using three different types of data organization strategies: Logically Contiguous (LC), linear placement of PGs by the original ADIOS (ORG), and EDO (EDO). Each test case is run 10 times for every data point. The median of top five results is chosen to remove the transient effect.

4.4.1 Performance of Planar Reads with PG-level Reordering

We first evaluate the performance of planar reads. In this test case, only PG-level organization using the Hilbert curve is applied by EDO. As shown in Figure 4.4, the most variation is observed for linear placement when the stripe count is divisible by 2. Linear placement can achieve better concurrency when the stripe count is a prime number. We set the stripe count to two representative cases, 128 and 137, respectively. Table 4.2 shows the theoretical concurrency achievable by different data organizations on three dimensions for two cases: S and X. The numbers are calculated based on the formula we introduced in section 4.3. The maximum or the exact number of OSTs are listed wherever applicable.

As discussed in section 4.1, the read performance is also impacted by the number of seek operations. Table 4.3 gives measured numbers of seeks required by each process under different organizations when 64 processes (readers) are used. The numbers may vary when the number of readers changes. ORG and EDO both are based on ADIOS BP format, resulting in identical numbers of seek operations. Because ADIOS uses aforementioned

strategy that reads in redundant data to avoid frequent seek operations, it requires much less seeks compared to LC. When 64 (8×8 on one plane) readers read out the data written by 4096 (16×16 on one plane) writers, each reader needs to retrieve 4 (2×2) PGs. Thus 4 seeks are required for each process. We omit one seek operation needed to consult the variable metadata. Because the metadata is only read by the first process, and then passed to the rest of processes. This does not affect the overall analysis.

Table 4.2: Concurrency of Planes (number of OSTs)

(a) stripe_count=128

	jk	ik	ij
EDO (Max/Min)	100/92	104/56	104/74
ORG	128	16	8
LC (S/X)	13/1	10/128	28/128

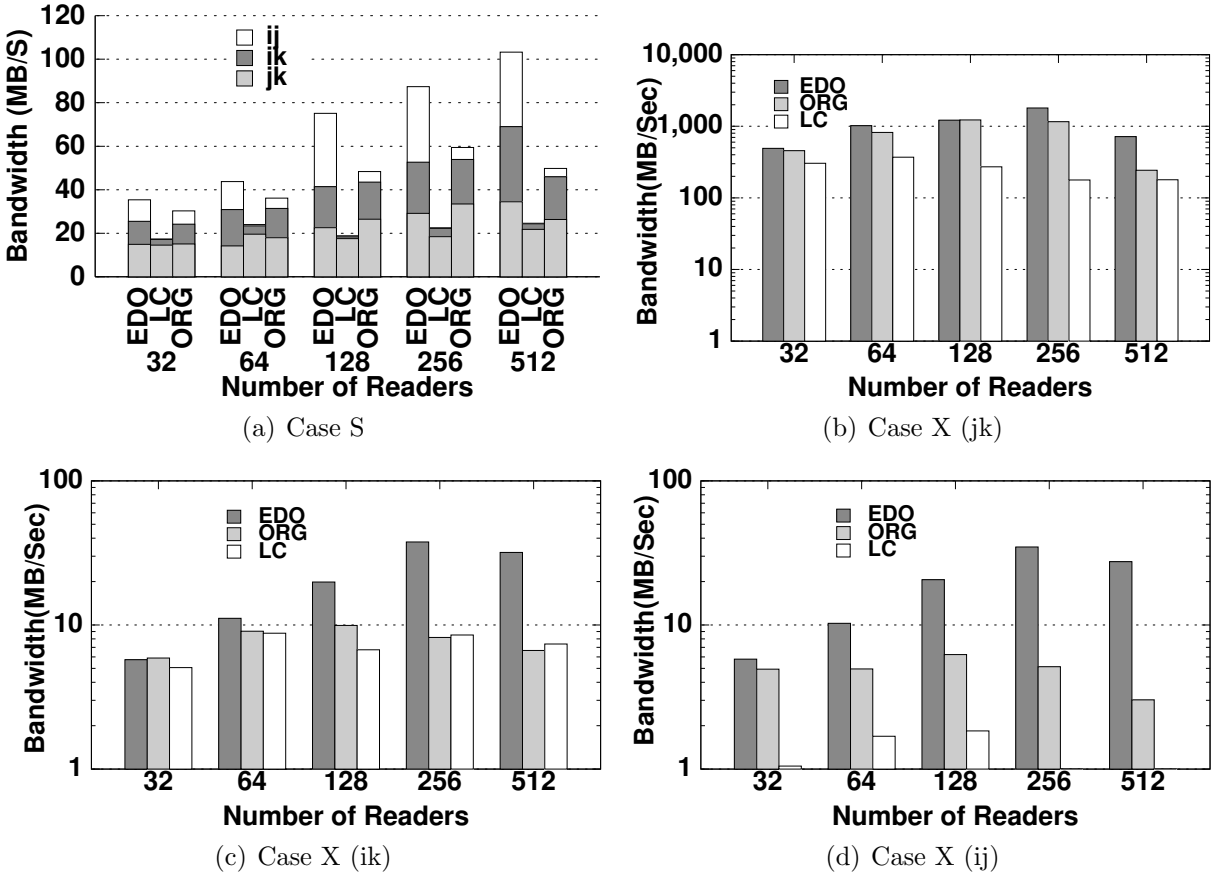
(b) stripe_count=137

	jk	ik	ij
EDO (Max/Min)	124/115	135/105	134/107
ORG	137	137	137
LC (S/X)	13/1	137/137	137/137

Table 4.3: Number of Seeks per Reader (64 Readers)

	S		X	
	LC	ADIOS	LC	ADIOS
jk	40	4	250	4
ik	40	4	250	4
ij	160	4	62500	4

The performance results for a stripe count of 128 are shown in Figure 4.5. For both cases S and X, LC has better or close performance compared to the other two organizations with small numbers of readers on jk planes. This is because LC places these planes as large, sequential data blocks to a few storage targets while the same plane spreads across many OSTs in small units under the other two organizations. Small number of readers also



(a) Case S

(b) Case X (jk)

(c) Case X (ik)

(d) Case X (ij)

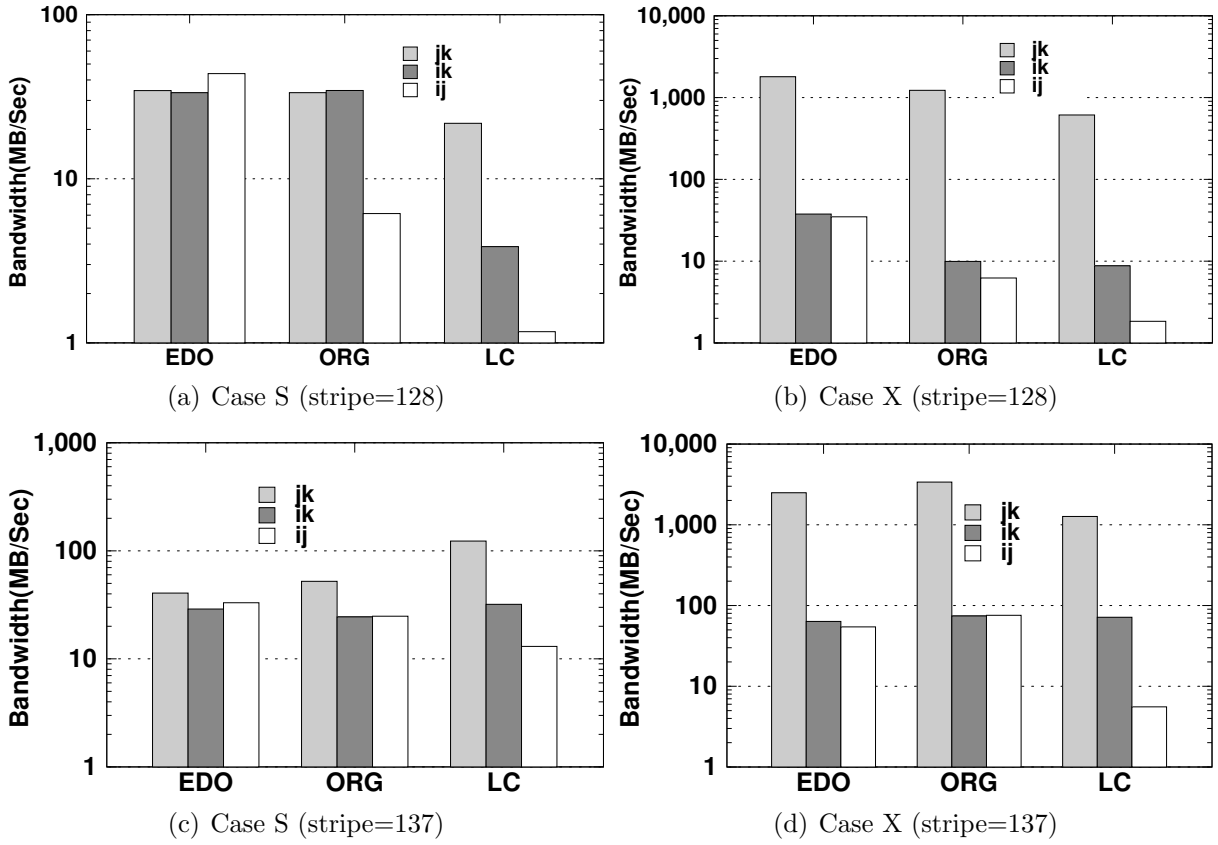
Figure 4.5: Planar Performance (writers=4096, stripe=128, plane size (S/X) = 800KB/122MB)

result in more seeks for each process with the ADIOS BP format. Thus, LC particularly favors a small number of reading processes to retrieve data. More processes suffers from contention because read requests will conflict with each other on a few OSTs. The story is different for ik and ij planes. Because a large number of seek and read operations are required, and because data spreads across a small number of OSTs, the performance of LC drops significantly. In contrast, ORG and EDO are able to bring the read performance of ik and ij planes close to that of the jk planes in Case S. The decreasing number of OSTs leads to a performance loss to ORG, especially for the ij planes. In Case X, ORG performs similarly compared to LC on ik and ij planes because small concurrency and large amount of read overhead unnecessarily consuming a lot of I/O bandwidth. Even though EDO suffers from the same amount of read overhead on these two dimensions, good concurrency helps it achieve much higher read performance.

Figures 4.6(a) and 4.6(b) show the peak performance of two test cases. Good concurrency and fewer seek and read operations help EDO achieve a maximum speedup of 37 times and 7 times for planes of slow-varying dimensions compared to LC and ORG, respectively. We also observe that read overhead of chunking do not impact the performance of EDO for Case S. But distinct performance differences are observed for Case X across three planes.

We conduct the same evaluation when the stripe count is changed to 137 with maximum of 512 readers. We show peak performance in Figures 4.6(c) and 4.6(d). Adding more OSTs help three data organizations improve their bandwidth. Particularly, ORG delivers the best performance because of good concurrency as shown in Table 4.2. Even so, EDO is able to achieve performance close to ORG, without sophisticated user efforts in determining that 137 storage targets are optimal for this combination of data size, writing process count and data organization.

In view of its consistent and balanced performance results for all planes with different stripe counts, we believe EDO can be used as a better data organization strategy, particularly beneficial in supporting user convenience and consistent near-optimal performance.



(a) Case S (stripe=128)

(b) Case X (stripe=128)

(c) Case S (stripe=137)

(d) Case X (stripe=137)

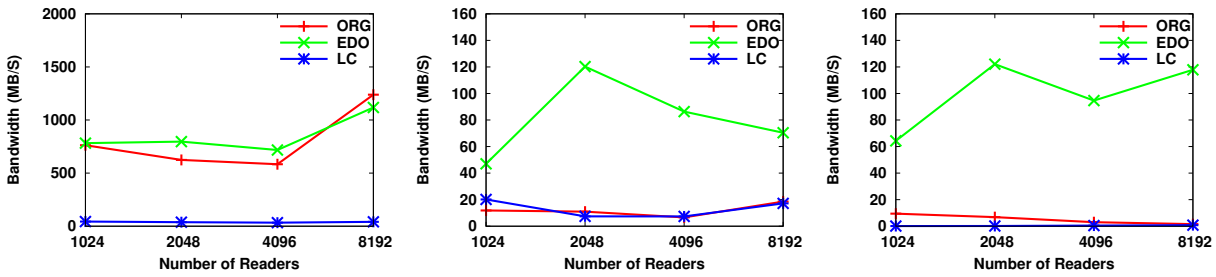
Figure 4.6: Performance Differences Across Different Planes (Logscale)

4.4.2 Scalability of Planar Reads

We also examine the performance of planar reads with an increasing number of writers. In this experiment, the number of readers is fixed to 512, while the data being read is written by different number of processes ranging from 1,024 to 8,192. Thus the amount of data read by each process increases accordingly. Both Hilbert curve and subchunking are used for EDO in this case. The results for Case X are shown in figures 5.6. Once again EDO is able to maintain good and consistent performance.

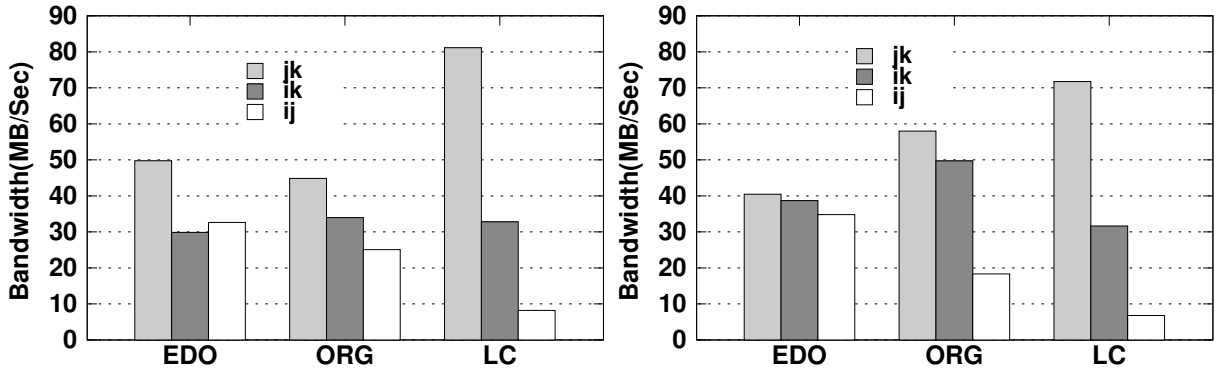
4.4.3 Planar Reads of Multiple Variables

Another common access pattern from scientific codes is to read planes from multiple variables, for example, reading a slice of data from 9 variables out of 14 variables. The ratio



(a) jk (b) ik (c) ij

Figure 4.7: Planar Performance with Scaling of Writers(Readers=512, stripe=128)



(a) 36 Readers (b) 72 Readers

Figure 4.8: Read Planes from Multiple Variables

of writers to readers is normally 10 : 1 or 20 : 1. To evaluate the performance of EDO for such patterns, we design a test case that has 8 double precision 3-D arrays written by 720 $(10 \times 6 \times 12)$ processes. For each variable, a process writes a $22 \times 36 \times 22$ 3-D array resulting in a global size of $264 \times 216 \times 220$. So the total data size is 765.7 MB. One plane is 445.5 KB. We set the stripe count as 40 and the stripe size as 2 MB so that data created by one process is written to one OST. Only Hilbert curve is used for PG-level organization within EDO.

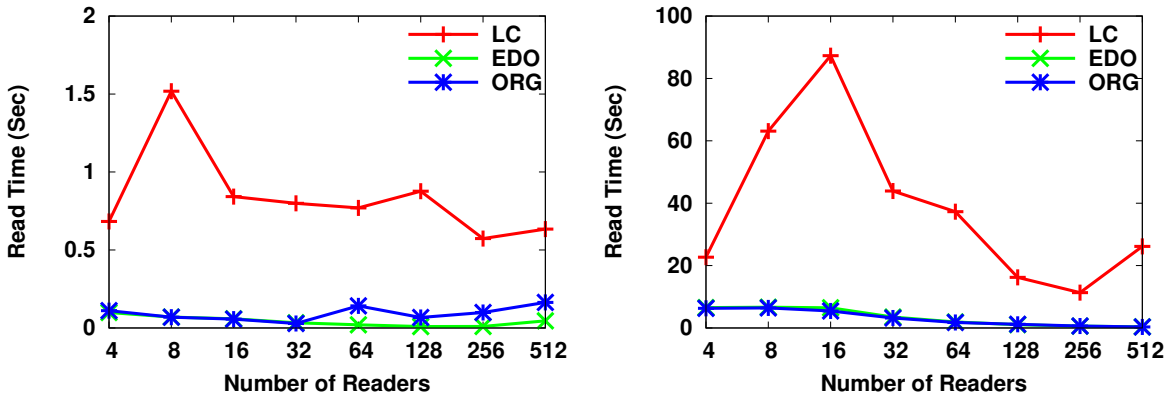
We evaluate the performance of reading 2-D planes from 5 out of 8 variables. 36 and 72 processes are used, respectively, corresponding to 1/20 and 1/10 of the original 720 writers. The results are shown in Figure 5.10. LC has the best performance for jk planes. But because of contention, adding more read processes decreases the bandwidth. More seek

operations again degrade the performance of ik and ij planes. ORG achieves nearly the same performance on jk and ik planes because their data chunks spread to all 40 OSTs. The difference is due to more extraneous data is retrieved for ik planes. The performance drops significantly for ij planes, especially for 72 processes. This is because such planes are only located on 10 OSTs, compared to the maximum of 35 OSTs using EDO. With 36 processes, the contention is not as severe as that of 72 processes, thus relatively higher bandwidth is observed. EDO performs slightly lower than but comparable to ORG for jk and ik planes. This can be attributed to the difference between the number of OSTs. Overall, EDO delivers consistent and balanced read performance for all planes under both cases and with any choice of striping parameters.

4.4.4 Read Performance of Subvolume

A subvolume is an orthogonal, rectangular cube within a multidimensional variable. For these experiments, the small and large data cases are examined using 128 storage targets for all three data organizations. To represent an arbitrary subvolume, a volume that containing one-eighth of the total data size is read from the center of the logical simulation area. Each dimension of the subvolume is half of the maximal dimension size. Only Hilbert curve is used for PG-level organization within EDO. Figure 5.11 shows the experimental results.

In both cases, EDO is able to cover more OSTs than ORG, i.e. 128 and 64, respectively. 64 OSTs are enough to serve up to 512 readers effectively. Thus we observe similar performance between EDO and ORG, with EDO performing slightly better. Even though LC is able to use all OSTs in both cases, its performance suffers because reading a subvolume requires more seek operations. Overall, EDO is able to provide consistent and balanced read performance for planar reads. For cases like reading a subvolume where a dataset covers enough storage targets, EDO is able to achieve the peak read performance.



(a) Case S (b) Case L

Figure 4.9: Subvolume Performance (writers=4096, stripes=128, subvolume size (S/L) = 31.25MB/3.81GB)

4.4.5 Impact to Data Generation

To examine possible performance impact of EDO to write operations, we evaluate its overhead in terms of time increment to data generation between ORG and EDO. Overall, a maximum of 5% overhead is observed among all the test cases. Detailed results are not included. This test indicates that EDO has negligible impact to the generation of multidimensional datasets in scientific applications.

4.5 Summary

In this work, we have designed and developed a Space Filling Curve based Elastic Data Organization for efficient data retrieval from scientific multidimensional datasets. Multiple organization strategies are supported as selectable algorithms for data ordering at two different levels. EDO uses Hilbert curve for distributing and ordering ADIOS data groups. By using this strategy, data from scientific multidimensional arrays can be distributed in a balanced manner across all storage devices, so that the aggregated bandwidth can be effectively aggregated and exploited for challenging read access patterns, particularly planar reads. By doing so, EDO is able to deliver a more consistent and balanced read performance

for all types of planar reads from multidimensional arrays. We have mathematically validate the performance benefits of EDO on large-scale parallel file systems. We have also experimentally evaluated the performance of EDO using the Jaguar supercomputer at Oak Ridge National Laboratory. Our results demonstrate that EDO can improve the performance of planar reads by as much as 37 times.

Chapter 5

Two-level System-Aware Data Organization

In Chapter 4, we have demonstrated that the read performance can be improved for common access patterns of scientific applications through a SFC-based data placement strategy. However, significant difference is still observed for large chunks, as shown in Figure 4.6. This is due to the large amount of extra data retrieved for reading on the slow dimension with chunking data layout. So the question is, is there a *correct* chunk size that achieves the best balance between the cost of seek operations and extra data retrieval? If so, how do we find it?

To enable efficient I/O, another issue needs to be addressed is the complexity of the simulation output. One simulation dataset is normally a collection of many multidimensional variables. Each variable possesses distinct characteristics. For example, a simulation may generate one variable that is on the order of hundreds of gigabytes, while another variable is only hundreds of megabytes. After domain decomposition, a common parallelization strategy in scientific applications, such differences become even more pronounced. It is difficult to achieve the optimal performance by applying a uniform data organization strategy to all variables. Therefore, a technique that can dynamically organize each variable to match with the underlying storage system is desired.

To guide a *corret* data organization for complex simulation output, we introduce a theoretical **Optimized Chunking** model to derive the *ideal* chunk size and guide data organization during the simulation run-time. Such a model is established upon a collection of system parameters. So the organization is able to adapt to the underlying system. To enable optimized chunking, we design a two-level data organization scheme. The first level is focused on the construction of *ideal* sized chunks. Based on our study in Chapter 4, a

Space Filling Curve based data placement strategy is used to ensure near-maximum data concurrency at the second level. We have implemented this multi-layer strategy as a light-weight middleware called **Smart-IO**.

In this chapter, we first introduce the Optimized Chunking model in Section 5.1. The design details of Smart-IO are described in Section 5.2. Section 5.3 validates our strategy through a comprehensive set of experimental results. Section 5.4 gives a summary of this work.

5.1 Optimized Chunking Model

In order to find the *good* chunk size, we first need to theoretically investigate what is the *ideal* data organization for multidimensional data on a parallel storage system. The I/O performance on a large-scale system is influenced by many factors, such as the communication cost, the number of writers, the size of the chunks, the number of storage targets, etc. As mentioned earlier, for a chunk based data organization, the size of chunks plays a critical role in determining the read performance for query on a data subset where data points are not contiguously stored. Thus optimizing read performance requires a model to find an effective chunk size that works well under the constraints of HPC system characteristics.

In general, the response time for a read request on a large scale system can be expressed as a five collection of five terms,

$$RT = (T_{comm} + T_{seek} + T_{io} + T_{local}) \times \alpha$$

Here T_{comm} is the time for each client to send out the request to the storage system over the network. T_{seek} is the total time to perform seek operations to locate the data on disk. It is the sum of the seek time, disk head settling time and rotational latency, which are all dependent on the physical characteristics of the disk and the physical locality of the data on the disk. Without taking into account the physical placement of data on a particular

disk, we use the return time of seek calls at the file system level to quantify the seek time. T_{io} is the time taken to read out all the requested data from disk. Since storage systems are commonly attached to the computation partition as separate units, this time is taken from the perspective of the client. Thus we include the data transfer time for the network to return data. The last part, T_{local}, is the time for client to perform local processing such as the memory copy. T_{local} depends on the contiguity of requested data in memory, but is significantly smaller than the other costs of I/O. α represents the interference factor on the large-scale system. However, to determine this factor needs a thorough study of the system and it is not the focus of this work. For a simplified analytical model, the external and internal interferences to the storage system are ignored. Such first principle guided modeling can help pinpoint a solution that enables near-optimal I/O performance tuning in a timely fashion.

Given the read time as describe by Equation (5.1), we now expand the model to calculate a chunk size that can provide near-optimal performance. Assume this optimized chunk size is **OCS** and the original chunk size is **CS**. A data chunk that is larger than OCS needs to be divided, while smaller data chunks require aggregation to the proper size. Therefore, we have

$$OCS = \begin{cases} \frac{CS}{N_{ocs}}, \text{when dividing large chunks} \\ CS \times N_{ocs}, \text{when aggregating small chunks,} \end{cases} \tag{5.1}$$

where N_{ocs} is the number of chunks of size **OCS**.

Assume a read request contains K optimized chunks, according to the Equation (5.1), the read time RT on the slow dimension(s) can be expressed as

$$RT_{slow} = K \times (CC + T_s + \frac{OCS}{BW_{io}} + T_{local}) \tag{5.2}$$

,where CC is the communication cost unit, $BW_{i/o}$ is the I/O bandwidth, T_s is the time unit for each seek operation, and T_{local} is time to perform memory copy operation for each chunk. The relationship between OCS, K, N_{ocs} and T_{seek} can be described as:

$$OCS \propto \frac{1}{K} \propto \frac{1}{N_{ocs}} \propto \frac{1}{T_{seek}}.$$

As we do not have a numerical relationship between K and N_{ocs}, there are two scenarios to be considered separately in order to solve Equation (5.2).

I/O Bounded: When OCS is too large, the time spent on I/O outweighs the cost of communication and seek operations. In this case, the entire read time is dominated by T_{io}, resulting in T_{seek} and T_{cc} being obviated in the expression. The read time on the slow dimension RT_{slow} can be represented as:

$$\begin{aligned} RT_{slow} =& K \times (\frac{OCS}{BW_{io}} + T_{local}), \\ & \text{when } K \times (CC + T_s) << \frac{K \times OCS}{BW_{io}}. \end{aligned} \tag{5.3}$$

Seek Bounded: When OCS is too small, i.e., N_{ocs} is large. Because $N_{ocs} \propto T_{seek}$, response time is more dominant by T_{seek}, and T_{io} will not appear in the expression. Therefore, we have

$$\begin{aligned} RT_{slow} =& K \times (CC + T_s + T_{local}), \\ & \text{when } K \times (CC + T_s) >> \frac{K \times OCS}{BW_{io}}. \end{aligned} \tag{5.4}$$

Notice the combination of Equations (5.3) and (5.4), which is the continuous function Equation (5.2), can be visualized using the dark red curved line in Figure 5.1. The left part of the curve represents the Equation (5.3), while the right side of the curve represents the Equation (5.4).

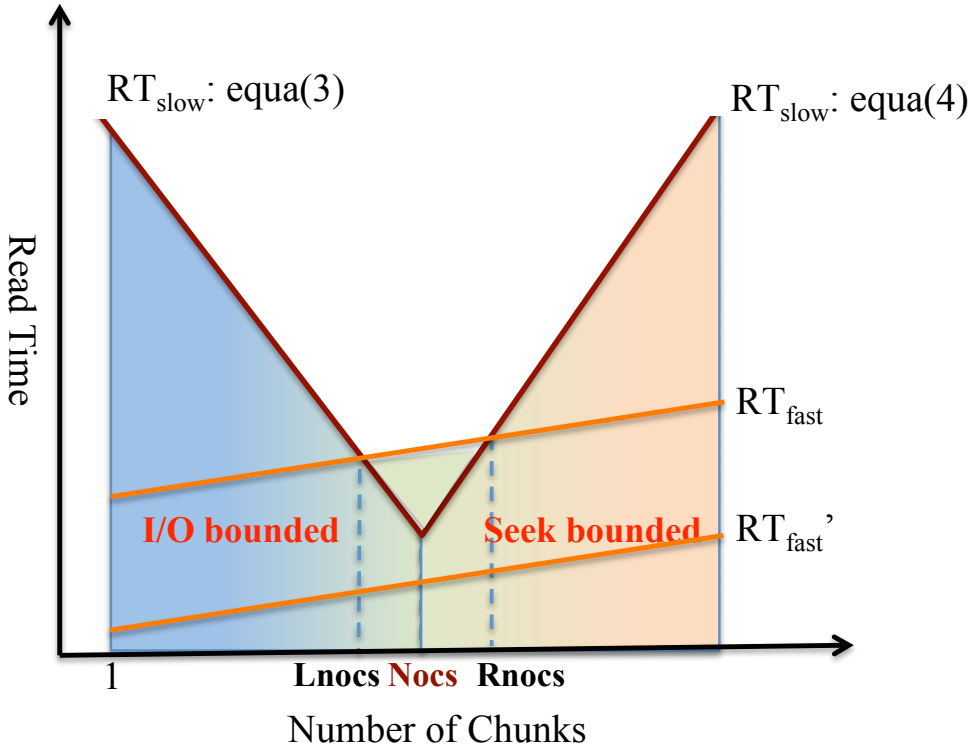


Figure 5.1: The Read Time vs. the Number of Chunks

The function's minimum is achieved at the point of discontinuity, that is, when:

$$K \times (CC + T_s) = \frac{K \times OCS}{BW_{io}} \tag{5.5}$$

Canceling K out from both sides of the equation, we have the Optimized Chunks Size:

$$OCS = BW_{io} \times (CC + T_s) \tag{5.6}$$

Correspondingly, N_{ocs} can be calculated as:

$$N_{ocs} = \begin{cases} \frac{CS}{BW_{io} \times (CC + T_s)}, \\ \text{when dividing large data chunks} \\ \\ \frac{BW_{io} \times (CC + T_s)}{CS}, \\ \text{when aggregating small data chunks} \end{cases} \tag{5.7}$$

It should be noted that the strategy of chunking is to improve the read performance when accessing the slow dimension(s) by sacrificing the performance on the fast dimension. The original contiguous data on the fast dimension is divided into discontiguous segments. Therefore, the read time on the fast dimension can be expressed as:

$$\begin{aligned} RT_{fast} =& K \times (CC + T_s + \frac{ReqSize}{K \times BW_{io}} + T_{local}) \\ =& K \times (CC + T_s + T_{local}) + \frac{ReqSize}{BW_{io}} \end{aligned} \tag{5.8}$$

, where ReqSize represents the size of the query. As we have $K_{ocs} \propto N_{ocs} \propto RT_{fast}$, therefore the read performance degradation grows linearly with N_{ocs}. This may cause the read performance on the fast dimension to become slower than the read performance on the slow dimension! This relationship is represented in Figure 5.1 by two orange straight lines. As mentioned earlier, in view of the general performance for any access pattern, the fastest total response time may not incur at the point N_{ocs} but still within the *Optimized Region* of L'_{nocs} and R_{nocs}". The performance difference inside of the Optimized Region is within a small range. As this study is aimed at finding an optimized chunk size, we use our solution of N_{ocs} as the guidance for data organization. As presented later, our experimental results demonstrate that this value provides satisfactory performance.

5.2 Design of Two-level System-Aware Data Organization

To enable the Optimized Chunking model and speed up scientific data analytics, we introduce Smart-IO, a light-weighted software layer that can dynamically construct the multidimensional scientific data into an optimized organization with the guidance of the model. Figure 5.2 shows the software architecture of Smart-IO and its components.

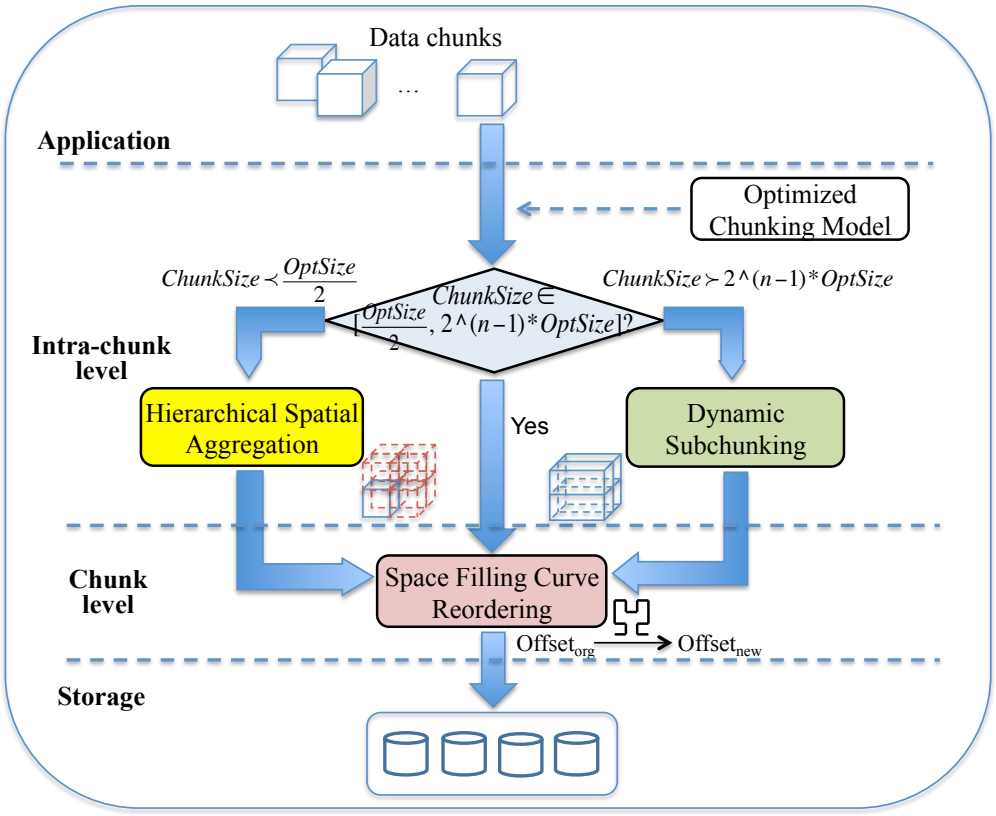


Figure 5.2: Two-Level Data Organization of Smart-IO

From high level, Smart-IO sits between the application layer and the storage system. It provides a two-level data organization. The first level is intra-chunk level, which focuses on building the data chunks into *optimized* size (OptSize). OptSize is a system-aware value derived from our *Optimized Chunking* model, which can achieve a good balance between the data transfer efficiency and processing overhead through specific system parameters. For data chunk that does not satisfy the OptSize, it needs to be reconstructed accordingly. To serve this purpose, two strategies are designed, namely *Hierarchical Spatial Aggregation*(HSA)and

Dynamic Subchunking(DYS). At the second level, which is chunk level, a default SFC (Space Filling Curve)-based reordering is used to distribute data chunks among storage devices to ensure the close-to-maximum data concurrency from the storage system.

Thus for a n-dimensional array, the two-level data organization with Optimized Chunking can be described as:

- If $ChunkSize < \frac{OptSize}{2}$, use Hierarchical Spatial Aggregation to consolidate small chunks;
- If $ChunkSize > OptSize \times 2^{n-1}$, use Dynamic Subchunking to decompose large chunks;
- If $\frac{OptSize}{2} < ChunkSize < OptSize \times 2^{n-1}$, use Space Filling Curve for chunk-level reordering.

Under such organization, a data chunk has three paths moving towards the storage system, as shown in Figure 5.2.

As we can see, a decision window is constructed as $[\frac{OptSize}{2}, OptSize \times 2^{n-1}]$. The rational of this window is provided in section 5.2.4 to determine the path of the data chunk.Essentially the goal of the HSA and DYS is to construct data chunks that fall into the window, where the SFC-based chunk ordering is applied. The rest of this section describes the design of these components in detail.

5.2.1 Hierarchical Spatial Aggregation

Even though the scientific applications normally generate a gigantic amount of data, it is not rare for an output dataset contains one or few small variables. A small variable is turned into even smaller pieces after domain decomposition. For example, an application running with 8,000 processes generates a 1-gigabyte variable. Such variable is divided into 8,000 131KB segments distributing within the output file. A significant number of seeks and memory operations are required for common access patterns, correspondingly leading to limited read performance. Such seek operations can only be avoided when the same

number of processes are used to read the variable. However, it is not a common practice in data analytics. Worse yet, the read performance is still not guaranteed as small requests from 8,000 processes can overwhelm the storage system. Aggregation is a technique that widely used to converge small pieces of data. However, simply concatenating small chunks does not solve the problem. Because the number of disk and memory operations remains the same for reading. Thus, we design a Hierarchical Spatial Aggregation strategy which aggregates data chunks in a way that their spatial localities are reserved. For every spatially adjacent 2^n processes, an *Aggregation Group* (AG) is formed. Within each AG, one process is selected as the aggregator for one variable. If there is more than one variable needs to be aggregated, the aggregator process will be selected in a round-robin fashion within the same group to achieve load balancing. The aggregator aggregates data from 3 closest neighbor processes and constructs them into a larger data chunk. If aggregated chunk size still does not fall into the decision window for output, a second level of aggregation will be performed among the first level aggregators who have hold all the data in their memories. Figure 5.3 shows an example of aggregating one variable from 16 processes in a 2-D space. For every spatially adjacent 4 processes, an *Aggregation Group* is formed. Process 0 is selected as the first aggregator. In our case, process 0 is chosen as the second level aggregator. After aggregation, only the aggregators will be writing out the data. Figure 5.4 gives an example of data movement and file output for 3 variables where 2 of them, var2 and var3, qualify for the HSA. After HSA, only process 0 needs to write out var2 and process 1 needs to write out var3. Such design causes slightly different output size for different processes. However, the upperbound of such difference is $OptSize \times 2^{(n-1)}$, which does not cause the significantly prolonged write time for the aggregators. While the amount of requests and seek operations are reduced by $level \times 2^n$ times, where level is the level of HSA performed.

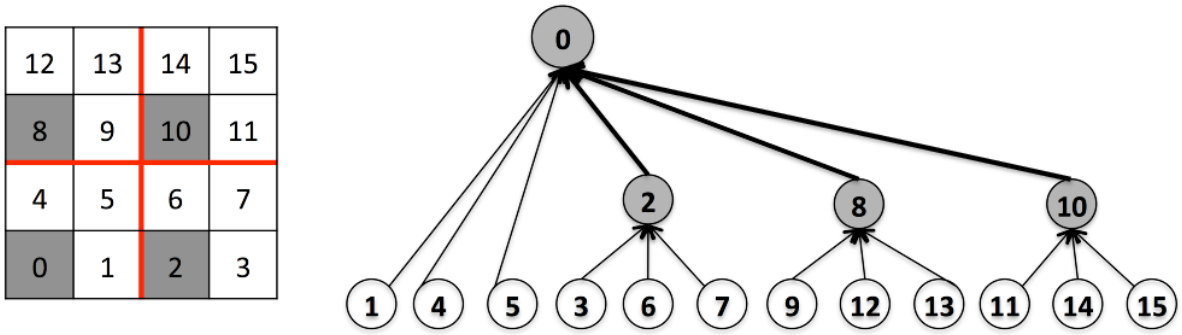


Figure 5.3: Hierarchical Spatial Aggregation

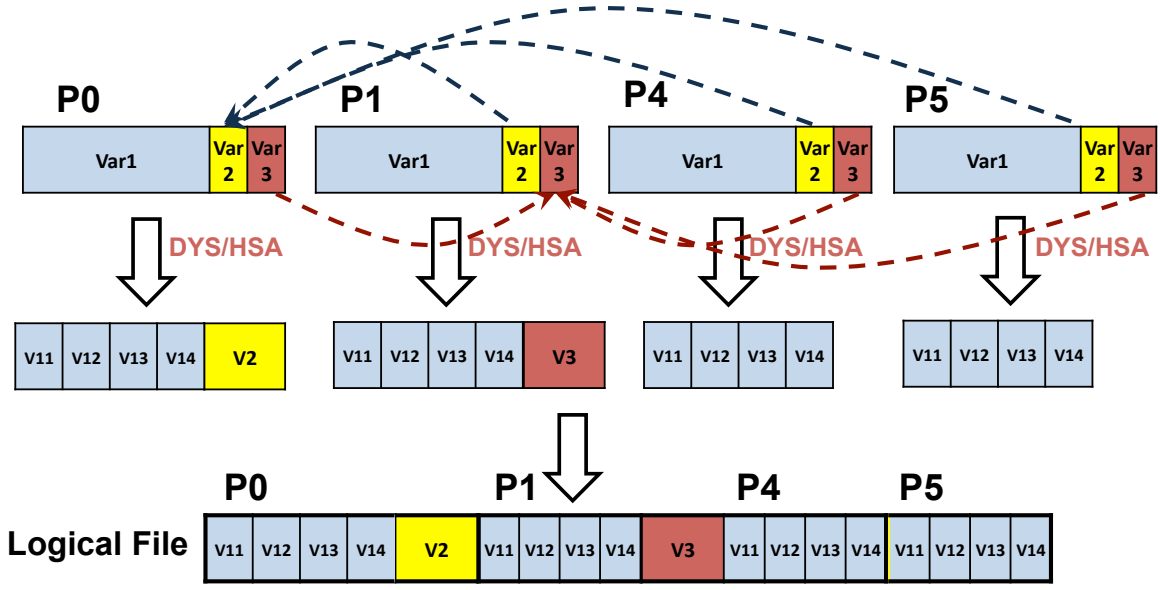


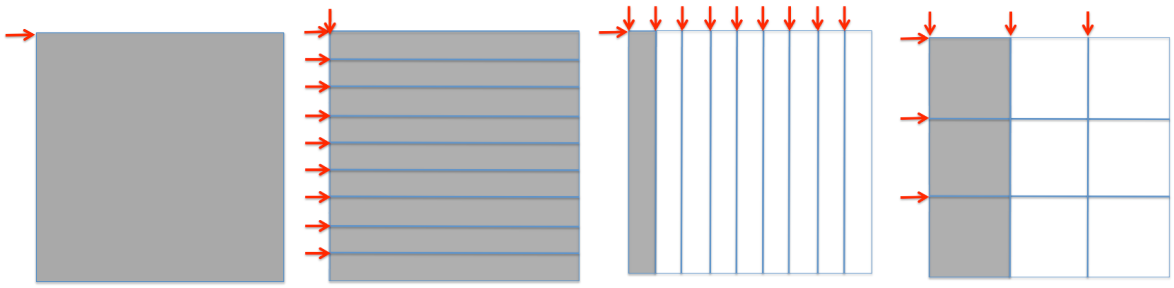
Figure 5.4: Data Movement and File Output

5.2.2 Dynamic Subchunking

Chunking has shown its capability to alleviate the *dimension dependency* for reading. However, because data points are laid out sequentially within a chunk, dimension dependency can be significant again when a data chunk itself is large. A range query on a slow dimension either has to perform a large amount of seek operations, or to read in a lot of redundant data from the start to the end point of request, as shown in Figure 5.5(a). Even though the latter option is more preferable on parallel file system, fundamentally neither approaches is efficient. As OptSize provides the good balance between the size of data transfer and the processing overhead, each large data chunk can be decomposed into *subchunks* with the size

of OptSize to further alleviate dimension dependency. Subchunking the large data chunk is important for applications that access data with a high degree of locality.

However, how to decompose a chunk needs to be investigated. Assume a 2D chunk is divided into 9 subchunks, there are three common options for such decomposition. Figure 5.5(b) to Figure 5.5(d) provide examples of these options. The red arrow represents seek operation. The shaded region represents the amount of data needs to be read in for a request on slow dimension. The row major is the fast dimension and column major is the slow dimension.



(a) Original Chunk (b) option 1 (c) option 2 (d) option 3

Figure 5.5: Comparison of Domain Decomposition for Data Subchunking

As we can from Figure 5.5(b) to Figure 5.5(d), subchunking on the slow dimension does not benefit reading on that dimension instead introducing more seek operations. The amount of data overhead is determined by the amount of subchunking on the fast dimension, which also is proportional to the performance degradation on the fast dimension. However, reading on fast dimension normally is always efficient compared to the slow dimension(s) because data is laid out contiguously. For example, reading 120MB data on Jaguar is expected to cost less than half second with one process. Assume read time is proportional to the number of seeks, which normally can be optimized by using more readers, subchunking into 9 subchunks along the fast dimension will increase the read time to 4.5 second. This is still within the tolerable margin, comparing to more than 60 seconds for read time on the slow dimension as measured. Thus, we choose option 2 as subchunking distribution strategy for our design. Essentially,

subchunking will be performed on all the dimensions except the slowest dimension for an n-dimensional data chunk. Number of subchunks on each dimension will be as balanced as possible.

In the following sections, we use a series number X_1-X_2...-$X_{(n-1)}$ to represent the number of subchunks on an n-dimensional array, where n represents the slowest dimension.

5.2.3 Data Organization based on Space Filling Curve

After the Optimized Chunks are constructed, a Hilbert Space Filling Curve ordering is used to rearrange the placement of data chunks on storage, as shown in Figure 4.3. The rational of such strategy is based on our work described in Chapter 4.

5.2.4 Optimized Chunk Size Decision Window

For a n-dimensional array, the decision window is constructed as $[\frac{OptSize}{2}, OptSize \times 2^{n-1}]$. The upperbound of this window is decided by our subchunking strategy. Because we only apply partition on n-1 dimensions as described in 5.2.2, and the minimum partition on each dimension is 2. Therefore given a data chunk, the minimum number of subchunks is 2^{n-1}, leading to the minimum chunk size qualifies for subchunking to be $OptSize \times 2^{n-1}$. On the other hand, the Hierarchical Spatial Aggregation is performed among the closest neighbors on n dimensions, leading to minimum 2^n chunks to be aggregated. However, we do not want to over aggregate data chunks that result in a chunk size where subchunking is required. As the upperbound of the chunk size is $OptSize \times 2^{n-1}$. So we have the lowerbound as $\frac{OptSize \times 2^{n-1}}{2^n}$, which is $\frac{OptSize}{2}$.

5.2.5 Smart-IO Implementation

In view of the performance issues of existing data organizations, we choose to implement Smart-IO as a part of Adaptable I/O System (ADIOS). The description of ADIOS is provided

in Section 3.4. Applications can easily adapt the system-aware data organization provided by Smart-IO through the flexible interface of ADIOS.

5.3 Experimental Results

We have implemented and evaluated Smart-IO on the *Jaguar* and *Sith* supercomputers at ORNL. Jaguar is currently the third fastest supercomputer in the world [60]. It is equipped with Spider (an installation of Lustre) for the storage subsystem, with a demonstrated bandwidth of 240 GB/s. Spider has three partitions named Widow 1, Widow 2 and Widow 3, respectively. Each partition contains 336 storage targets (OSTs). In our experiments, we used its Widow 2 partition. Sith is another smaller system at ORNL with 1,204 processing cores. Sith has a local Lustre file system which contains 12 OSTs. Most of the experiments are performed on Jaguar, unless it is otherwise stated.

The I/O bandwidth on both Spider and the local file system of Sith are approximately 250MB/Sec per OST, the average seek time is 8ms, and the communication cost is about 1.9ms. Thus, the OCS is calculated as 2.5MB using Equation (5.6). The decision window size is $[1.25MB, 10MB]$ for Optimized Chunking policy. Based on the previous practices of the ADIOS team on Jaguar, the stripe size is set as the size of ADIOS process group. This practice can maximize data concurrency, reduce false sharing on the Lustre file system, and reduce the internal and external interferences [53].

S3D [16] combustion simulation code from Sandia National Laboratories is used in our experiments. S3D is a high-fidelity, massively parallel solver for turbulent reacting flows. It employs a 3-D domain decomposition to parallelize the simulation space. We set up the output file to contain 4 variables (Var1, Var2, Var3 and Var4) with distinct sizes. Table 5.1 shows the data chunk size after the original 3-D domain decomposition and the exemplary variable sizes with 4,096 ($16 \times 16 \times 16$) processes. The Smart-IO operations performed on each chunk based on the decision window are also listed.

Table 5.1: Test Variables (Elements/Size)

	Var1	Var2	Var3	Var4
Chunk	256^3/128MB	128^3/16MB	64^3/2MB	32^3/256KB
Variable	4096^3/512GB	2048^3/64GB	1024^3/8GB	512^3/1GB
Operations	DYS/SFC	DYS/SFC	SFC	HSA/SFC

The performance evaluation of Smart-IO is mainly focused on the I/O performance of planar read, which is the most common yet very challenging access pattern. We measure the read performance among three types of data organization strategies: Logically Contiguous (LC), the chunking strategy of the original ADIOS (ORG), and two-level data organization of Smart-IO (Smart). A separate test program is created to evaluate the I/O performance of logically contiguous data layout. Each test case is run 10 times for every data point. The median of top five numbers is chosen as the result.

5.3.1 Data Generation

One of the design considerations of Smart-IO is to constrain the performance impact on data generation. As shown in Table 5.1, Dynamic Subchunking is performed on Var1 and Var2, while one level of Hierarchical Spatial Aggregation is performed on Var4. The write time of using 4,096 processes to output the entire file along with the time breakdown is shown in Table 5.2.

Table 5.2: Write Time Break Down

	I/O	Subchunking	Aggregation	Total
ORG	41.49	0	0	41.49
Smart	41.67	0.87	0.13	42.67

As we can see, subchunking and aggregation do not cause significant delays to the write time. Only 2.8% overhead is observed to the total write time. We also evaluate the weak scaling of data generation. As shown in Figure 5.6, very limited overhead is introduced in all cases.

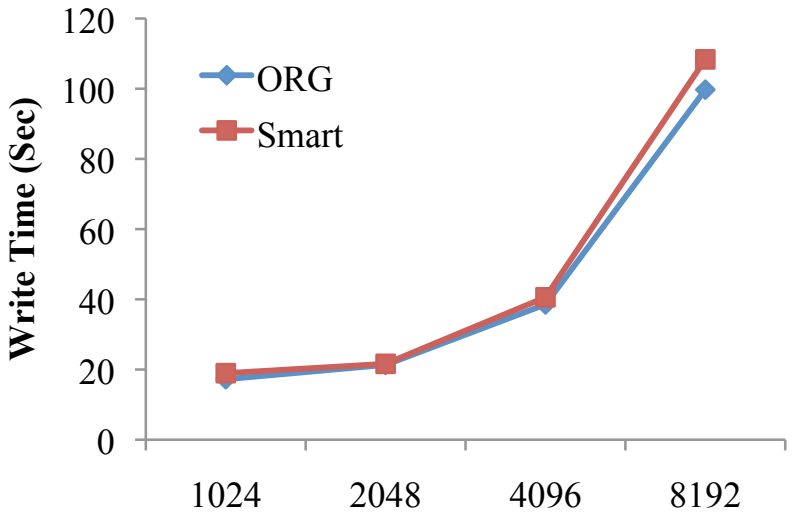


Figure 5.6: Weak Scaling of Data Generation

5.3.2 Dynamic Subchunking

As the decision of Dynamic Subchunking is based on the value of OCS. Thus, we evaluate the performance of subchunking as an index of the accuracy of our algorithm. Based on OCS value and Equation (5.6), subchuking is performed on Var1 and Var2. As both cases exhibit the same information. We only show the test results for Var1 as representative case. With Smart-IO, a 7-7-1 (49 total) subchunking is performed on Var1. To evaluate the accuracy of our model, we then vary the number of subchunks on each dimension to a range of $[N_{OCS} - 2, N_{OCS} + 2]$, that is $[5, 9]$ in our case. This leads to four other different number of subchunks 25, 36, 64 and 81. A planar read is performed on each of three dimensions. The number of readers varies from 32 to 512, so to follow the tradition that application scientists often spend only 10% of the writers to read. The total read time on three dimensions are shown in Figure 5.7. The number of X-axis represents the number of chunks on k, j, and i dimensions respectively, while O-7-7-1 represents the number of subchunks calculated by the OCS model.

As we can see, O-7-7-1 delivers the best read time on the slow dimensions in most cases. Increasing or decreasing 1 subchunk on each dimension, that is 6-6-1 and 8-8-1 in our test

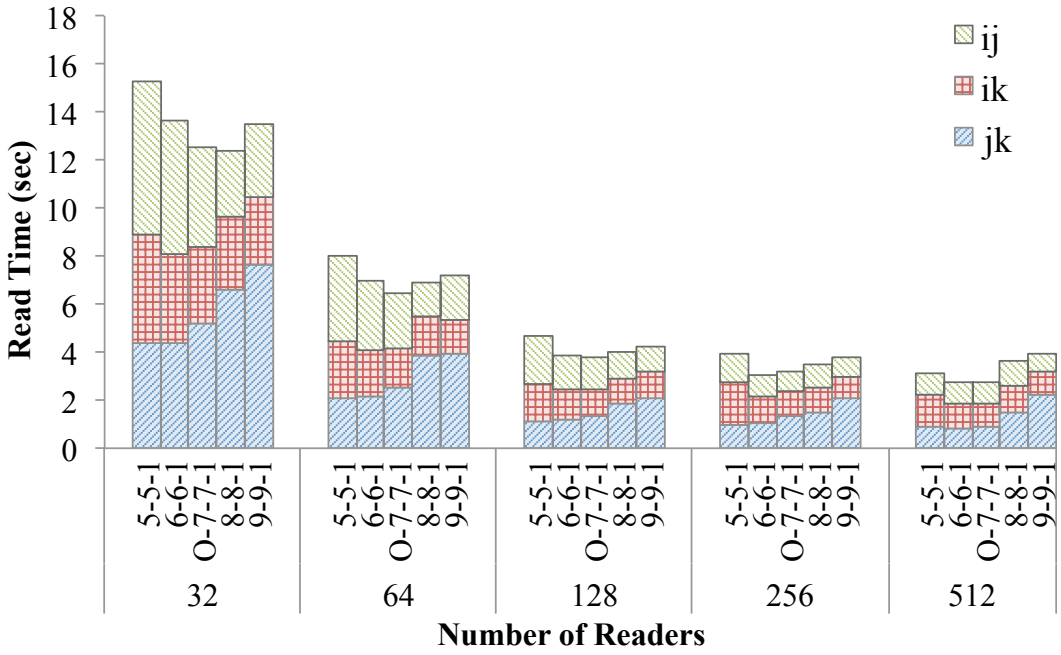
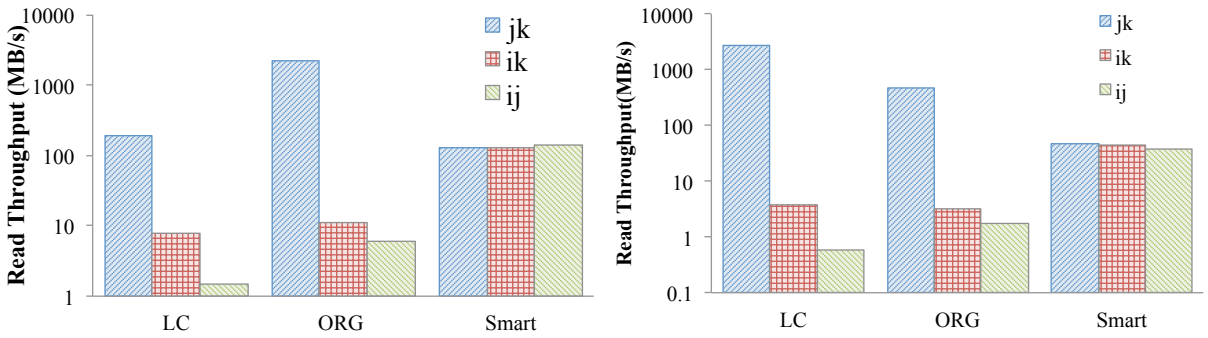


Figure 5.7: Dynamic Subchunking Performance; O-7-7-1 (49 subchunks) is our calculated value using the Optimized Chunking formula. We compare the performance with other variation of chunking from 25 (5-5-1) to 81 (9-9-1) subchunks

case, does not achieve a significant performance improvement. Further changing the number of subchunks suggests more performance degradation caused by either data overhead or processing overhead. Even though O-7-7-1 does not give the best performance for overall read time in some cases due to the overhead on the fast dimension *jk*, it is still able to deliver a close-to-optimal performance. As described in 5.1, the overall best performance can be achieved anywhere within the *optimized region*. Therefore, our formula is able to provide a satisfactory result for a close-to-optimal performance, if not the optimal.

We also examined the read performance of the original ADIOS and the Logically Contiguous data layouts. The peak performances of three different data organization on both Jaguar and Sith supercomputers are shown in Figure 5.8. Smart-IO achieves a maximum improvement of 12 times compared to the original ADIOS, and 72 times compared to LC on Jaguar, 22 times and 66 times speedup are observed on Sith. At the same time a more balanced read performance is achieved in three dimensions on both systems.

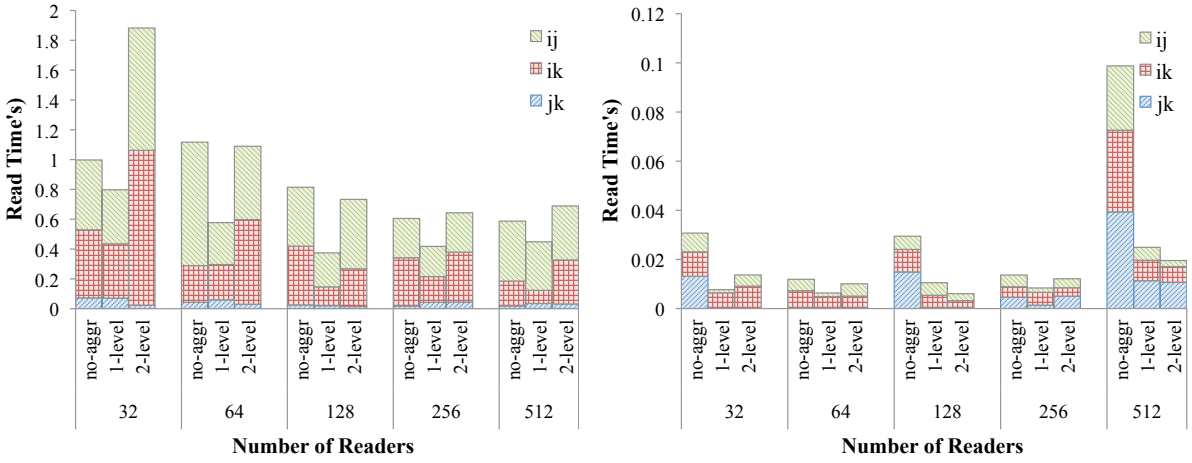


(a) Peak Performance (Jaguar, 512 Readers, Log scale) (b) Peak Performance (Sith, 256 Readers, Log scale)

Figure 5.8: Dynamic Subchunking Peak Performance on Planar Read

5.3.3 Hierarchical Spatial Aggregation

We then examine the performance of Hierarchical Spatial Aggregation by a given OCS. In our experiment, we focus on examining the threshold for performing HSA and its performance impact on planar reads. The calculated lower bound 1.25MB causes an one-level HSA (represented by 1-level in Figure 5.9 for Var4 with an aggregated chunk size of 2MB. To evaluate the performance of HSA, we manually set the HSA threshold to 4MB and 256KB, which lead to two-level (2-level) of HSA and no HSA (no-aggr) performed respectively. The results on both Jaguar and Sith are shown in Figure 5.9.



(a) Jaguar (b) Sith

Figure 5.9: Planar Read Performance of HSA

As shown in the Figure 5.9, 1-level of HSA provides the best performance on both systems. The performance of no-aggr suffers from frequent seek operations while 16MB aggregated chunk size of 2-level HSA introduces large data overhead. 1.25MB threshold of Smart-IO achieves a good balance between the number of seek operations and the amount of redundant data retrieval on the slow dimension. The read performance becomes closer with larger number of processes because the amount of overhead is reduced per process.

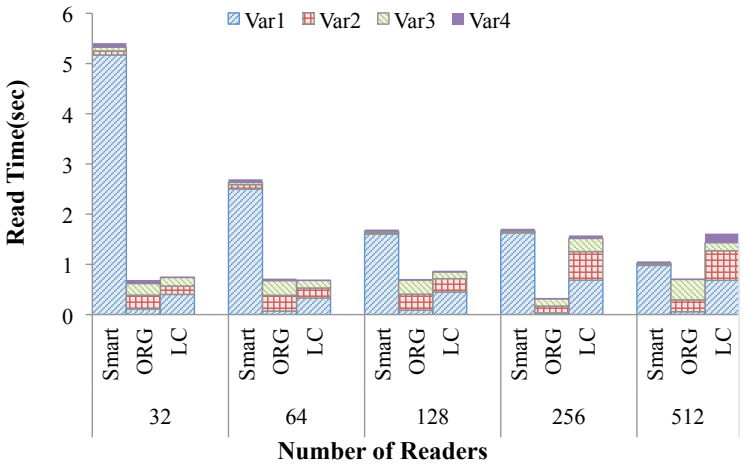
5.3.4 Planar Reads of Multiple Variables

We then evaluate the performance of reading one plane from all the variables within the test file. The evaluation is mainly focused on the comparison of data organization between Smart-IO, original ADIOS and LC. The total read time of planar reads for four variables are shown in Figure 5.10.

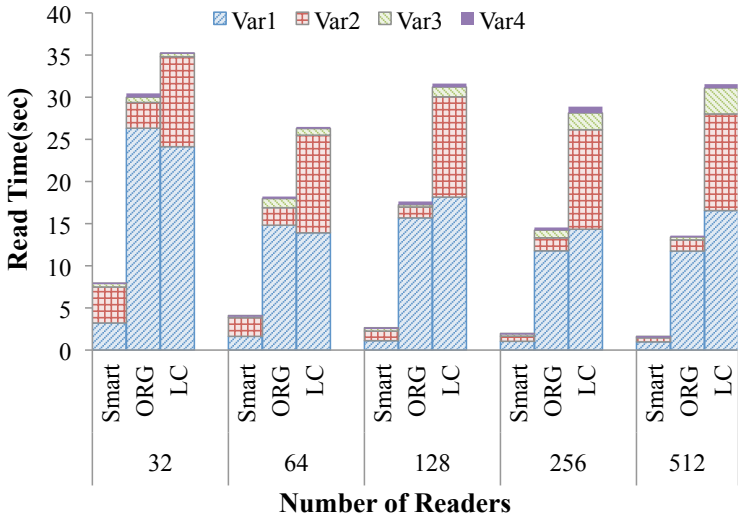
As we can see, LC shows advantage only when data is contiguous and smaller amount of readers are used, as shown in Figure 5.10(a). Due to limited data concurrency on fast dimension, the performance of LC degrades when more readers are used. Smart-IO showed small performance degradation compared to the original ADIOS on fast dimension. However, by using an OCS-based two-level data organization, Smart-IO is able to outperform the original ADIOS significantly on the slow dimensions. Overall, a maximum of 8 times and 43 times speedup of total read time is achieved compared to the original ADIOS and LC, respectively.

5.3.5 Read Subvolume

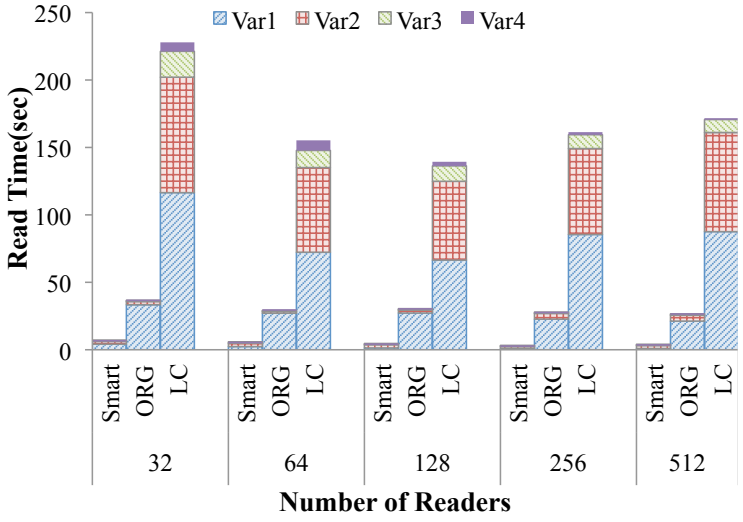
Finally we evaluate the performance of reading a subvolume from a variable. For these experiments, we use variables Var1 and Var4 as the representative cases. A volume containing one-eighth of the total data size is read from the center of the logical simulation area. Each dimension of the subvolume is half of the global dimension size. Figure 5.11 shows the experimental results for variable Var1 and Var4.



(a) jk

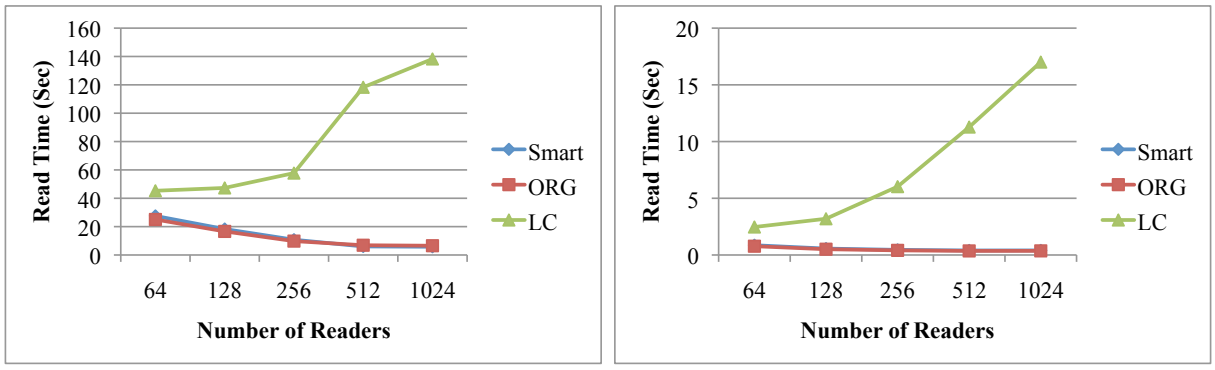


(b) ik



(c) ij

Figure 5.10: Planar Read Performance of Multiple Variables



(a) Var1 (Subvolume = 64GB) (b) Var4 (Subvolume = 128MB)

Figure 5.11: Subvolume Performance

As shown in the Figure 5.11, Smart-IO and the original ADIOS achieve close performance as the dimension dependency is less obvious for access pattern such as subvolume. The performance of LC suffers because data within a subvolume after 3-D domain decomposition is not contiguous, causing significant performance degradation in both cases.

5.4 Summary

The complexity of gigantic scientific data, and the physical limitation of current storage system that is built on the magnetic disks pose a grand challenge on providing an efficient I/O method. This work tries to address such issue from the read side of problem and focuses on providing a system-aware data organization that can effectively utilize the underlying storage system. To guide the proper data organization for multidimensional scientific data, we developed the Optimized Chunking model to find the best balance between the data transfer efficiency and processing overhead. Such model takes a collection of system parameters into consideration, so that the data chunk size can adapt to the underlying system. To further enable such model for scientific applications, we designed a light-weighted I/O framework named Smart-IO. Smart-IO provides two levels of data organization to address the challenges from a single storage target and the overall parallel storage system through Optimized Chunking model and a Hilbert Space Filling Curve ordering. By applying such

two-level data organization, we significantly alleviate the dimension dependency for multi-dimensional scientific data. A much more balanced and consistently good read performance is ensured for scientific post-processing. We evaluated Smart-IO on Jaguar and Sith Super-computers at Oak Ridge National Laboratory. The experimental results show that Smart-IO is able to achieve a maximum of 72 times and 22 times speedup to the planar reads of S3D compared to the Logically Contiguous and chunking data layout, respectively.

Chapter 6

Spatial and Temporal AggRegation

In the era of petascale computing, more and more scientific applications are deployed on leadership scale computing platforms to enhance the scientific productivity. However, many mission critical applications have challenging I/O access patterns that are not addressed adequately with the existing I/O techniques, and therefore cannot leverage the peak I/O performance of leading computing platforms. For example, scientific applications may generate datasets that are composed of many small data elements along various spatial and temporal dimensions. Such datasets are very challenging to be analyzed efficiently as a series of temporally or spatially related data.

In this study, we examine such access patterns in detail and propose a novel I/O scheme named STAR (Spatial and Temporal AggRegation) to enable high performance reads and writes. STAR is designed to identify the spatial and temporal relationships among data variables, and accordingly aggregate them into a multi-dimensional data structure before storing to the disks. In doing so, it is able to not only reduce the turnaround time, but also facilitate the common access patterns of data analytics. In particular, STAR is able to enable efficient data queries along the time dimension, a practice that is common in scientific analytics but not yet supported by existing I/O techniques. As a case study, we have extended GEOS-5 (the Goddard Earth Observing System Model) from NASA, to leverage the benefits of STAR.

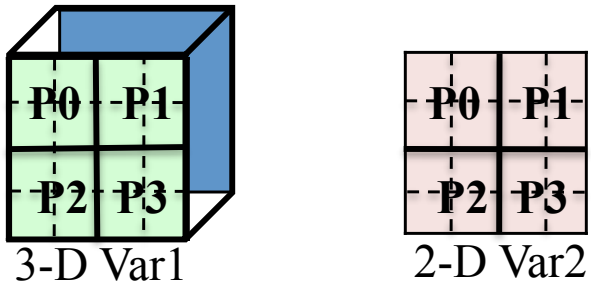
In this chapter, we first discuss the background and motivation of this work in detail in Section 6.1. We then introduce the design of STAR in Section 6.2. Section 6.3 validates our strategy through a comprehensive set of experimental results. Finally, a summary is provided in Section 6.4.

6.1 Motivation

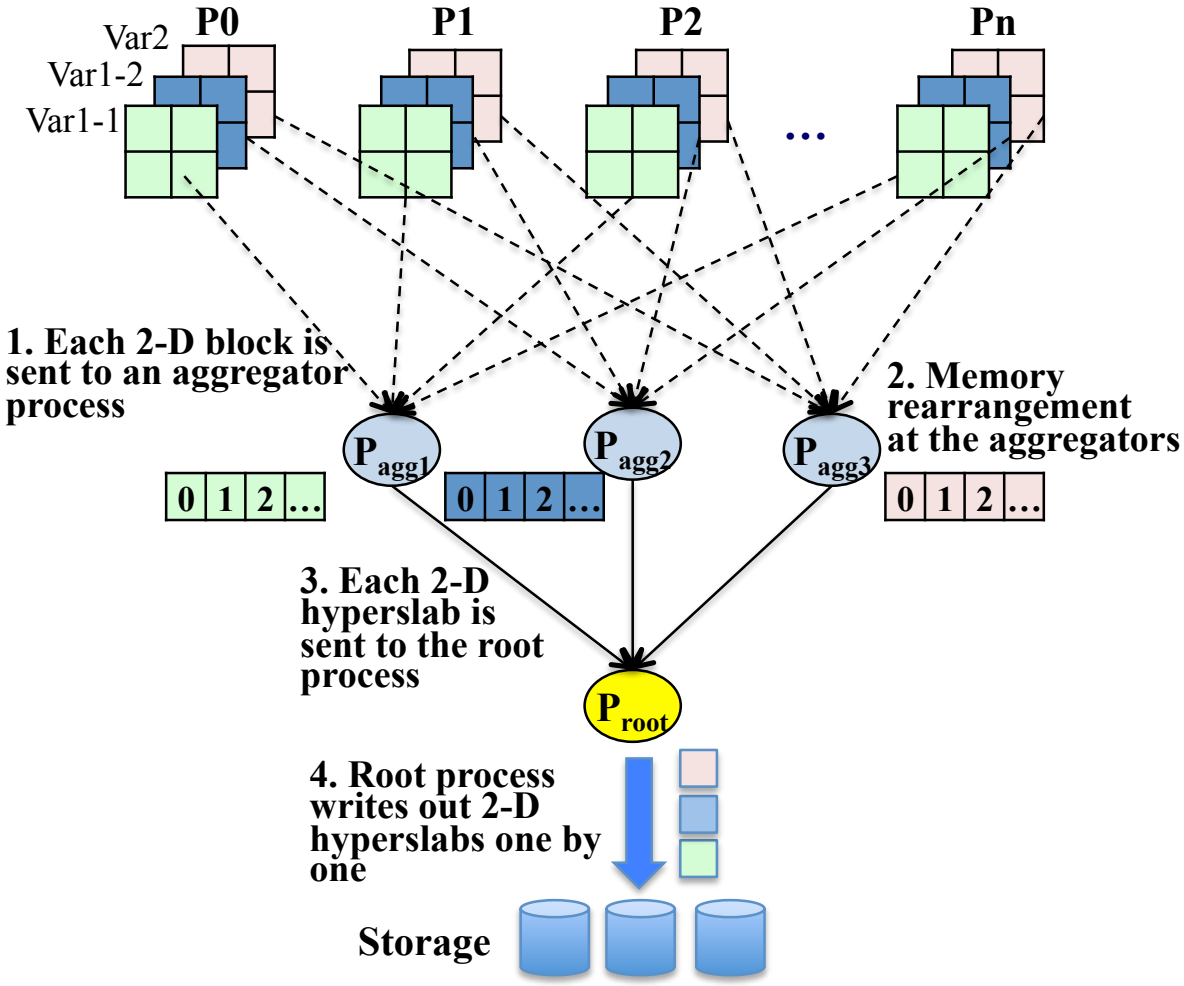
6.1.1 Analyzing the Legacy GEOS-5 I/O Flow

The Goddard Earth Observing System Model, Version 5 (GEOS-5) [2] is designed by NASA to simulate climate variability on a wide range of time scales, from hours of small time scales to multi-century climate change. Data variables of GEOS-5 are organized as *bundles*, one for each physics model such as moist, turbulence, etc. GEOS-5 uses the serial version of the NetCDF-4 I/O library, which provides a logically contiguous data layout for each variable in a self-describing file format. Because GEOS-5 applies a 2-D domain decomposition on the simulation space, a significant number of data rearrangements in memory and across network are incurred to ensure the contiguity of data before storing it to storage.

Figure 6.1(a) shows data decomposition along two dimensions for a 3-D variable Var1 and 2-D variable Var2. The data movement to write these two variables with n processes using current NetCDF-4/HDF-5 method is shown in Figure 6.1(b). As we can see, a multi-dimensional variable is written out in the unit of 2-D hyperslabs. One progress is designated as the *aggregator* for one hyperslab. For data variables composed of more than one 2-D hyperslab, e.g 3-D variables, the aggregator is chosen in a round robin fashion among all processes for the purpose of balancing the load. For example, Figure 6.1 shows that 3 different processes are chosen as the aggregators for 3 hyperslabs, respectively. After the 2-D hyperslabs are formed, one *Root* process receives them from the aggregators according to their logical position in the global array. It then stores them to the storage. From the movement of data, we can observe a series of sequential memory and communication operations that would impact the write performance. For example, data blocks from each process are sequentially sent to the aggregators. In the same manner, the hyperslabs are pushed to the storage from the Root process. A large amount of *many to many shuffling* and *many to one fan-in* communication operations are required for data transfer, which likely degrades the performance and constrains the scalability. Moreover, significant memory is required at the



(a) 2-D Domain Decomposition



(b) Data Movement

Figure 6.1: Legacy GEOS-5 Data Movement Among n Processes

aggregators in order to accommodate the newly formed hyperslabs from 2-D data blocks. Such overhead grows linearly with the increasing volume of the output variables, as well as

the number of processes. Moreover, the performance is limited to the bandwidth of the Root process, unable to fully exploit the aggregated bandwidth of large storage systems.

6.1.2 Issues for Efficient I/O

The design of current I/O flow of GEOS-5 is to facilitate the use of one kind of data layout called *logically contiguous*. This desire is shared by many other applications that organize data in such fashion. Another layout called *Chunking* [80] allows data to be stored in its original multidimensional structure along with domain decomposition. This layout provides an opportunity to enable a parallel I/O stream for applications. In fact, chunking has demonstrated to outperform the logically contiguous (LC) as a data organization for multidimensional scientific data [80, 92]. However, simply applying chunking is not effective to speedup I/O for applications such as GEOS-5. The challenges are two-fold.

Large Total Data Volume, Small Time Step Output

Many applications produce a large amount of data over the course of a simulation. The total volume of generated data is massive but such gigantic dataset can also be the collection of small different outputs, a.k.a time steps. For example, a diagnosis output for GEOS-5 simulation ith a half degree resolution consists of 265 variables ranging from 800KB to 40MB, leading to the total size for one time step to be 3.1GB. When such simulation is run at scale, each process controls only a very small fraction of the data for each of the 265 variables. With a chunking data layout, a large number of small write requests will be sent to storage at each time step, causing ineffective utilization of the network and degrading the storage performance.

Data Access Across Time Steps

There exists also an issue with the current GEOS-5 data layout for data access. Our earlier work has studied the data organization and categorized data access patterns during post-processing [92] into the following categories.

- Read in **all** of a single variable.
- Read an arbitrary orthogonal **subvolume**
- Read an arbitrary orthogonal **full plane**

That prior work focused on the read performance for a specific time step. However, such patterns are also commonly performed across time dimension. Particularly, an application may read a subset of data variables from different time steps. Long-running applications such as GEOS-5 normally consist of many time steps. It is a common practice of data post-processing for scientists to explore the physical change of simulated natural systems over a good span of time. However, the expensive I/O costs leads to significantly prolonged simulation execution time. The scientists often have to reduce the number of time steps to maintain the overall cost. Such compromise limits the scope of scientific exploration and in turn restrains the productivity. Even after the desired number of time steps are written out, currently a normal practice is to store data of different time steps in different files, or in different blocks within the same file. Because the data segments are scattered within the file, a large number of read requests have to be made to retrieve data back, degrading the storage performance. The performance is further aggregated if data blocks are stored in different files as metadata overhead can become significant. Such overhead grows linearly with the increasing number of time steps. This makes the analysis of scientific data very inefficient.

To address the aforementioned I/O issues for applications such as GEOS-5, we need to come up with a systematic I/O strategy that can support fast I/O for simulation run-time, and efficient analysis in both temporal and spatial dimensions.

6.2 Design of STAR

To address the I/O issues described in Section 6.1, we propose an I/O technique called Spatial and Temporal AggRegation (STAR). The design of STAR focuses on achieving three goals for scientific applications:

1. To enable a fully parallelized and high performance I/O flow for scientific applications;
2. To provide a data organization that facilitates the common access patterns of data post-processing; and
3. To require minimal modification of the application.

From a high level, STAR sits between the application and the underlying storage system. It uses a chunking-based data organization to enable high-performance parallel I/O flow for both writing and reading. To accommodate applications that have small output and a large number of time steps, the design includes a dual-aggregation strategy that not only aggregates data spatially, but also temporally. This strategy offers another dimension for constructing bulk data output with limited network communication, while facilitating the common access patterns of data post-processing. Two aggregation algorithms, *Spatial Aggregation* (SAR) and *Temporal Aggregation* (TAR) are proposed for such purpose. An example of data movement using STAR to output a 2-D variable of 3 time steps is shown in Figure 6.2. As we can see, the 2-D blocks of the variable are first merged into a 3-D block with *time* as a new dimension at each process. Then SAR is performed among the processes to further merge data chunks before they are pushed to storage. Such merging is performed hierarchically to maintain the spatial locality of the data points, while reducing the amount of write requests. We describe STAR in more detail in the rest of this section.

6.2.1 Temporal Aggregation

It has been a common practice to use some extra memory as a temporary data buffer to consolidate small data blocks into one large sequential block. However, the scope of

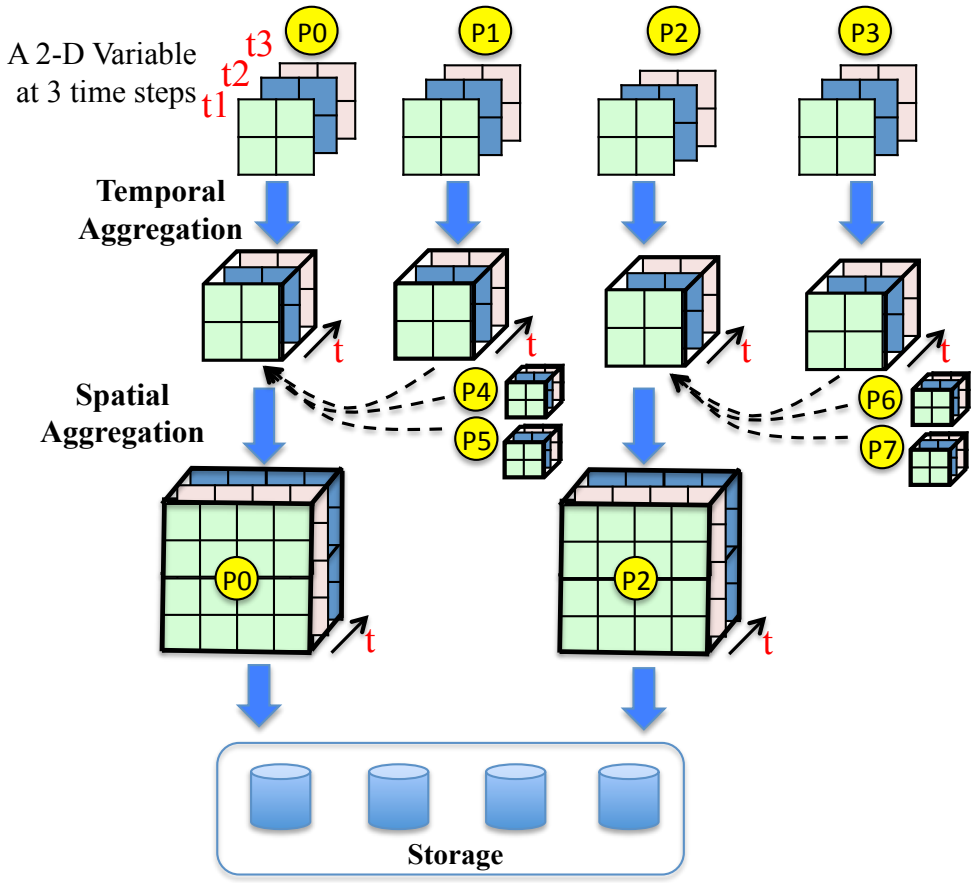


Figure 6.2: Data Movement Between 8 Processes on a 2-D Variable (2-D domain decomposition, 3 time steps output)

such buffering strategy is typically limited to a single compute node and thus not very effective when the data size per process is rather small on a single node. For example, one time step of the half degree simulation of GEOS-5 generates about 3.1GB data. When the simulation is run with 4,096 processes, each process contains only 0.78MB data for 265 variables. For a compute node consisting of 16 cores, one node can only accumulate a maximum of 12.5MB of data. To further merge data, communication based aggregation is frequently used to consolidate data across compute nodes. However, this requires extensively more aggregation in order to achieve a large enough block of data, thus incurring significant network communication costs. Both effectiveness and scalability of such network dependent techniques are limited when the simulation is run at scale.

In addition, the existing data layouts, such as chunking or LC, normally organize the output of different time steps in different locations within one or more files. Such layouts cannot efficiently support data analytics along the time dimension. Assuming the default stripe count is 3 on a file system, the left part of Figure 6.3 shows an example in which six time-step versions of one variable are stored individually. As we can see, for a query that examines the data across 6 time steps, 6 requests need to be made to retrieve data from the storage targets. Even though such requests can be served in parallel, a large number of requests with an increasing number of time steps and readers cause contention and serialization at the storage targets, leading to degraded performance and limited scalability. In general, the cost of such access pattern can be expressed as:

$$Cost_{org} = N_{ts} \times (T_{Request} + \frac{DataSize_{ts}}{BW_{io}}) \times \alpha \tag{6.1}$$

, where N_{ts} is the number of time steps of query. $T_{Request}$ represents the overhead to initiate a read request to storage target. $DataSize_{ts}$ is the requested data size per time step. BW_{io} is the I/O bandwidth of the storage target. α represents the interference factor of the system. As we can see, the cost of performing data analytics in time dimension grows linearly with the increasing number of time steps. In particular when the requested data is small, majority of the I/O time will be spent on initiating the I/O operations, which is highly inefficient.

Temporal Aggregation is designed to open up another horizon for data consolidation, while facilitating efficient data analytics in time dimension. As shown in Figure 6.2, instead of pushing the variable to storage at the end of each time step, three 2-D variable data blocks from different time steps are stored as a contiguous memory segment. The memory segment is then passed to the next stage of processing. Logically three 2-D data blocks are now stored as one 3-D data chunk with time as the new dimension. Such strategy gives more opportunity for data to be further merged into larger segments without introducing inter-process communication overhead. Meanwhile, it reduces the frequency of I/O requests to storage and alleviates the network contention for both writing and reading. For a d_{tar}

degree temporal aggregation, all of the data from N_{ts} time steps is merged. Therefore the number of I/O request is reduced to $\frac{N_{ts}}{d_{tar}}$. Reading from 6 time steps with the degree of TAR equals to 3 is shown in the right part of Figure 6.3. As we can see, only two requests are required to retrieve data. Such strategy significantly reduces the read overhead and network contention. The number of requests can be further reduced with higher degree of TAR. The cost of a query in time dimension can be expressed as:

$$Cost_{STAR} = (\frac{N_{ts}}{d_{tar}} \times T_{Request} + \frac{N_{ts} * DataSize_{ts}}{BW_{io}}) \times \alpha \tag{6.2}$$

, where d_{tar} is the degree of TAR. d_{tar} is dependent on the total size of each time step and the amount of free memory on node. When the available memory is not sufficient to hold more time steps, buffered data is flushed to storage and the temporal aggregation process starts again from the following time step. Figure 6.3 shows an example when d_{tar} equals 3 time steps. Note that TAR does not impact the performance of read operations for a specific time step because the organization at each time step is not changed.

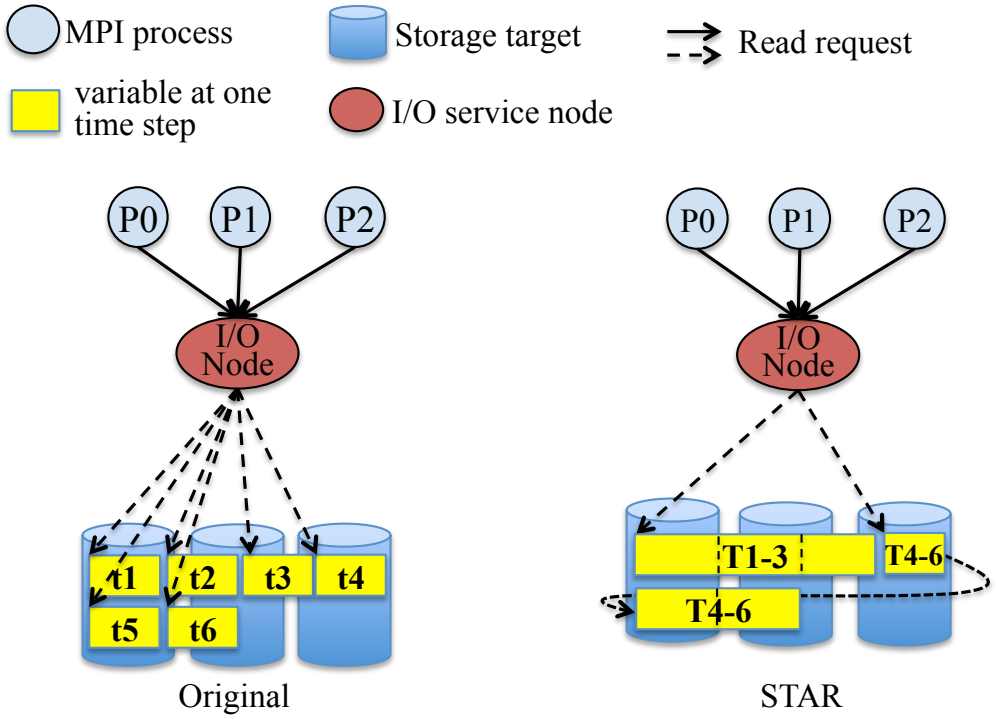


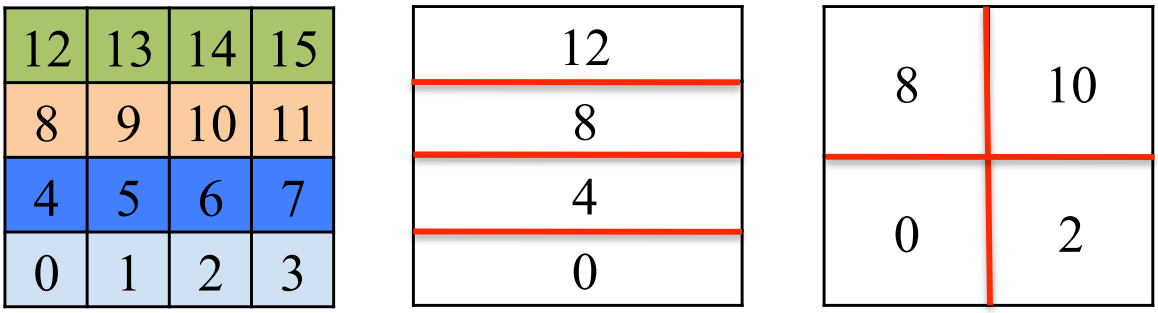
Figure 6.3: Comparison of Reading on Time Domain for Different Data Organizations

6.2.2 Spatial Aggregation

STAR employs a chunking organization to store multidimensional datasets. Even though temporal aggregation increases the size of the chunks that are written, the total size of each variable remains fairly small. As we discussed in section 6.1.2, chunking breaks the continuity of data. For example, a 3-D variable from a half degree simulation of GEOS-5 has a global dimension of $576 \times 361 \times 48$. After a 2-D domain decomposition from 4,096 ($128 \times 32 \times 1$) processes, the local dimension for each process is only about $5 \times 12 \times 48$. Such small chunks cause significant overhead for reading because the access time is dominated by numerous seek operations to reach these chunks. Reading a 2-D plane on the fastest dimension would require a total of 4,096 seek operations. Even though such overhead can be amortized by using more readers, the number of read requests remains the same, causing contention at the storage.

To address the data access issue, we use our previous study Hierarchical Spatial Aggregation (SAR) introduced in Section 5.2.1. The aggregation is performed in a way that the spatial locality of each data point is maintained. After aggregation, only the final aggregators writes out the data. With SAR, the contention at compute node and storage is alleviated because fewer processes are participating in the writing. The read performance is greatly improved, because the number of read requests and seek operations on any dimension are reduced by $(2^n)^{sl}$ times, where sl denotes the actual level of SAR. The detail description of this strategy can be found in Section 5.2.1. However, the control of the level of aggregation is different in STAR. Instead of following the Optimized Chunk Size guidance, SAR is conducted based on the availale memory at the aggregators. Since both SAR and TAR require the use of additional memory, the memory resource needs to be balanced among them. The coordination of SAR and TAR requirements is described in Section 6.2.3.

Besides the read performance for decomposed global variables, another consideration of hierarchical aggregation is to maintain the capability of alleviating the dimension dependency of chunking data organization. Assume each compute node has 4 CPU cores, Figure 6.4(a)



(a) Original 2-D Chunk (b) Intra-node (c) Hierarchical

Figure 6.4: Comparison of Aggregation Strategy

shows a 2-D chunk written by 16 processes. Different shading represents different compute nodes. One popular aggregation strategy is to merge data from processes within one compute node. Therefore process 0, 4, 8 and 12 are chosen as aggregators. While with hierarchical aggregation, process 0, 2, 8 and 10 are chosen as aggregators. As shown in Figure 6.4, intra-node aggregation reduces network communication between compute nodes. However, the result of such aggregation is a 2-D chunk with logically contiguous data organization as shown in Figure 6.4(b). While hierarchical aggregation is able to maintain the chunking data organization of the original 2-D data, as shown in Figure 6.4(c). Therefore SAR enables faster read operations on the slow dimensions, a.k.a alleviating the dimension dependency for multidimensional data. Such capability is important to enable efficient read operations for data analytics [86].

6.2.3 Coordination of Temporal Aggregation and Spatial Aggregation

With the trend of increasing number of cores per compute node and decreasing amount of memory per core, the coordination of SAR and TAR needs to be well planned to ensure a good usage of precious memory resources. Given a memory with size of $Memsize$ that is allocated for STAR, the relationship between the memory capacity and the aggregation parameters can be expressed by the following equation:

$$MemSize = d_{tar} \times DataSize_{ts} \times (2^n)^{sl}, d_{tar} \leq Total_{ts} \tag{6.3}$$

, where $MemSize$ is the amount of allocated memory, d_{tar} the degree of Temporal Aggregation, $DataSize_{ts}$ the amount of data per time step for each process, n the number of domain decomposition, and sl again the level of Spatial Aggregation.

To ensure a balance between TAR and SAR, by default STAR enables one level of SAR, that is $sl = 1$. Under such circumstances, if the memory cannot hold all of the data from all time steps, which means $Total_{ts} > d_{tar}$, the output will be divided into $\frac{Total_{ts}}{d_{tar}}$ times. When memory is large enough to buffer data from all the time steps, that is $Total_{ts} \leq d_{tar}$, sl can be calculated as:

$$sl = \lfloor \log_{2^n}(\frac{MemSize}{Total_{ts} \times DataSize_{ts}}) \rfloor \tag{6.4}$$

STAR allows user to specify the amount of memory allocated for STAR based on the application and system parameters. The calculation of d_{tar} and sl are performed within STAR automatically.

6.2.4 STAR Implementation and Incorporation with GEOS-5

GEOS-5 [2] is a system of models integrated in the Earth System Modeling Framework (ESMF). The GEOS-5 Data Assimilation System integrates the GEOS-5 Atmospheric Global Climate Model (GEOS-5 GCM) with the Gridpoint Statistical Interpolation atmospheric analysis (ANA) developed jointly with National Oceanic and Atmospheric Administration (NOAA) /National Centers for Environmental Prediction. GCM components consists of three major components, namely Atmospheric Global Climate Model (AGCM), Oceanic Global Climate Model (OGCM), and HISTORY which is the I/O component in charge of the output for diagnosis data. It is also the focus of this study. Current HISTORY component provides two types of output format: the GrDAS *flat* binary data output and

self-describing NetCDF-4/HDF-5 file format. so to enable a self-describing file format with high performance for both simulation run-time output and data post-processing.

We have designed and implemented STAR first as part of the Adaptable I/O System (ADIOS)[1], ADIOS applies the chunking strategy for storing multidimensional datasets. We built STAR as a component of ADIOS to leverage its penetration among the existing scientific applications. ADIOS allows users to specify the amount of memory for data buffering, we use such feature to direct the coordination of TAR and SAR as we described in the previous section.

As a case study, we have enabled STAR within GEOS-5 as an extension of its HISTORY I/O component, as shown in Figure 6.5. The output format of GEOS-5 is defined within its input configuration file, where users can easily switch to STAR and leverage its high-performance I/O for writing and reading.

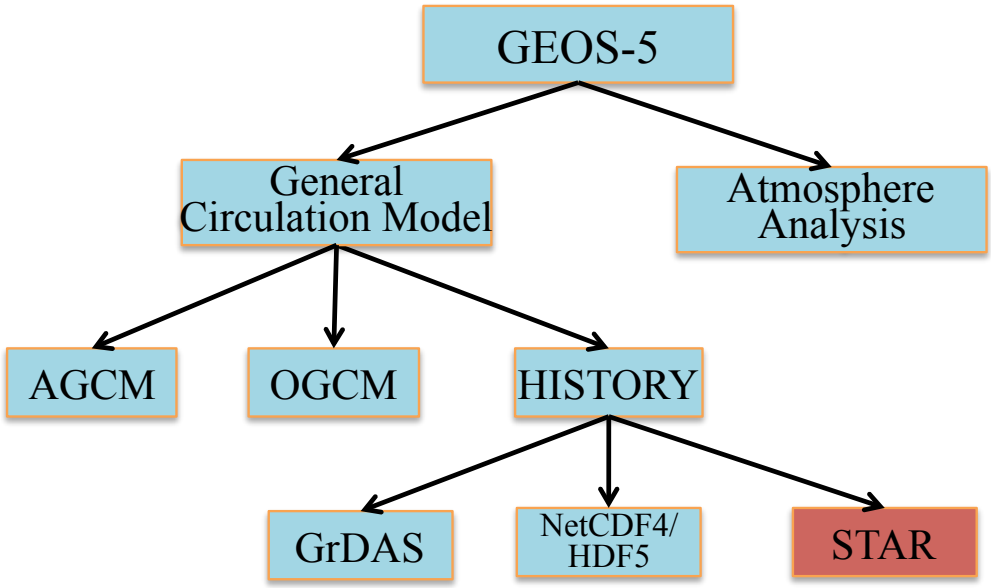


Figure 6.5: GEOS-5 Model Architecture

6.3 Experimental Results

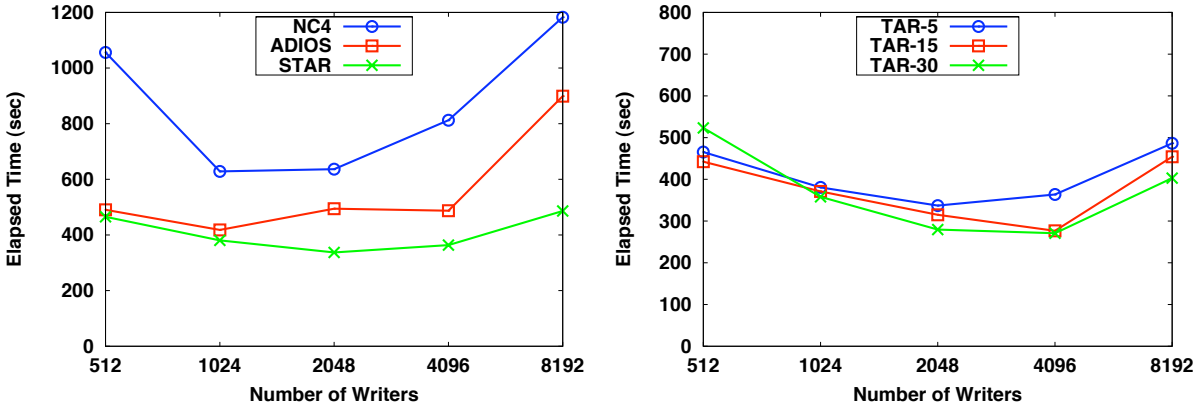
We have evaluated the performance of STAR on Jaguar [60] supercomputer at ORNL. Jaguar is currently the third fastest supercomputer in the world [60]. It is equipped with 18,688 compute nodes. Each node contains one 16-core Opteron processor and 32GB memory. Jaguar is connected to Spider (an installation of Lustre) as its storage subsystem. Spider has three partitions named Widow 1, Widow 2 and Widow 3, respectively. In our experiments, we use its Widow 2 partition which contains a total of 336 storage targets (OSTs).

GEOS-5 is used as our test application. In our experiment, GEOS-5 is configured as half degree simulation unless otherwise stated. The output is configured with 7 variable bundles, which include 185 2-D variables and 80 3-D variables. The simulation resolution is 576×361 with 72 vertical levels. It is regridded to 48 levels during output. Such configuration leads to the data size per time step as 3.12GB. GEOS-5 applies a 2-D domain decomposition on the simulation space, leaving the vertical resolution undivided. A 2-D variable with time as its third dimension can be expressed as var(lon, lat, t), where *lon* represents the longitude and *lat* represents the latitude. A 3-D variable with time as its fourth dimension can be expressed as var(lon, lat, alt, t), where *alt* represents the altitude, a.k.a., the vertical levels. The evaluation studies both write and read performance of three different data organizations: the original GEOS-5 NetCDF-4 I/O method (NC4), the original ADIOS (AIOS), and STAR (STAR). Every test case is ran 10 times and the second best result is reported, so as to remove the transient effect.

6.3.1 Write Performance of STAR with Temporal Aggregation

We first evaluate the write performance of STAR with GEOS-5. The application is configured as a 30 hour simulation with diagnosis data output every hour, resulting 30 time steps. The time is measured by the embedded timer within the HISTORY component of GEOS-5. We compare the write performance between different output methods including

the original NetCDF-4, ADIOS, and STAR. To measure the performance of STAR with temporal aggregation only, we manually turned off Spatial Aggregation in this test case. The degree of TAR is set to be 5 time steps. The number of writers varies from 512 to 8,192 and the test results are shown in Figure 6.6(a).



(a) Degree of Temporal Aggregation=5 time steps (b) Comparison of Different Degree of Temporal Aggregation

Figure 6.6: Data Output Elapsed Time (30 time steps, Spatial Aggregation disabled)

As shown in Figure 6.6, both ADIOS and STAR are able to reduce the I/O time significantly compared to the original NetCDF-4 method. By aggregating 5 time steps of data, STAR is able to reduce the number of write requests, and write data at big blocks. Therefore it achieves the best performance compared to NC4 and ADIOS. Note the I/O time becomes longer with 8192 processes for all three cases. This is the increasing contention from more processes to a fixed number of storage targets [50]. Notably, among the three, the original NetCDF-4 method is the worst because of its serial management of data aggregation to a single processes.

To further evaluate the performance of temporal aggregation, we increase the degree of TAR to 15 and 30 time steps and evaluate the write performance. The results are shown in Figure 6.6(b). Among all the TAR test cases, aggregating all of the 30 time steps achieves the most significant improvement. That's because the number of I/O operations reduces in proportion to the increasing degree of TAR.

6.3.2 Write Performance of STAR with Duo-Aggregation

Next we evaluate the write performance of STAR when both TAR and SAR are enabled. In this experiment, we fix the degree of TAR to 30 time steps and the memory allocation to be 800MB at the aggregator. Base on Equation 6.4, this leads to 1 level of SAR for 512 and 1,024 processes, and 2 level of SAR for the rest of test cases. We compare the write performance of STAR with NC4, ADIOS, and SAR only. The write performance is shown in Figure 6.7.

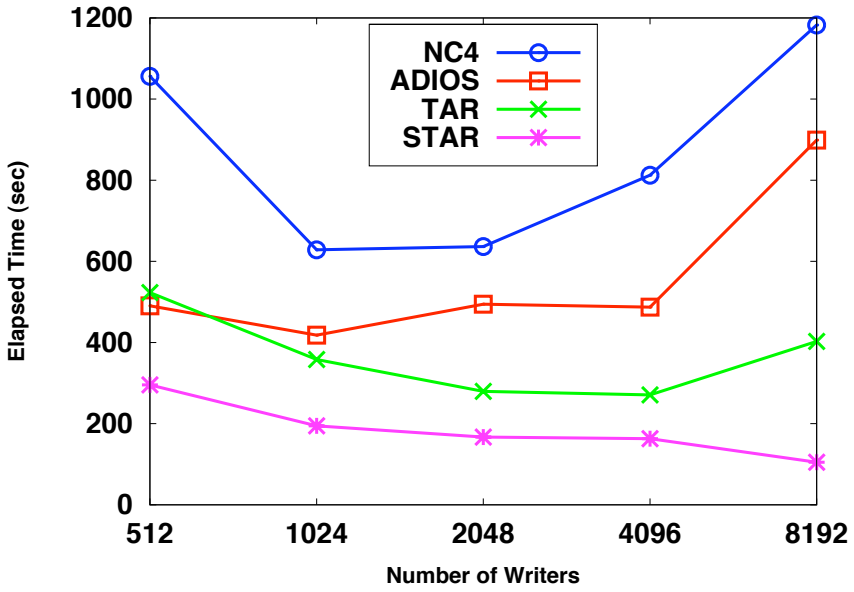
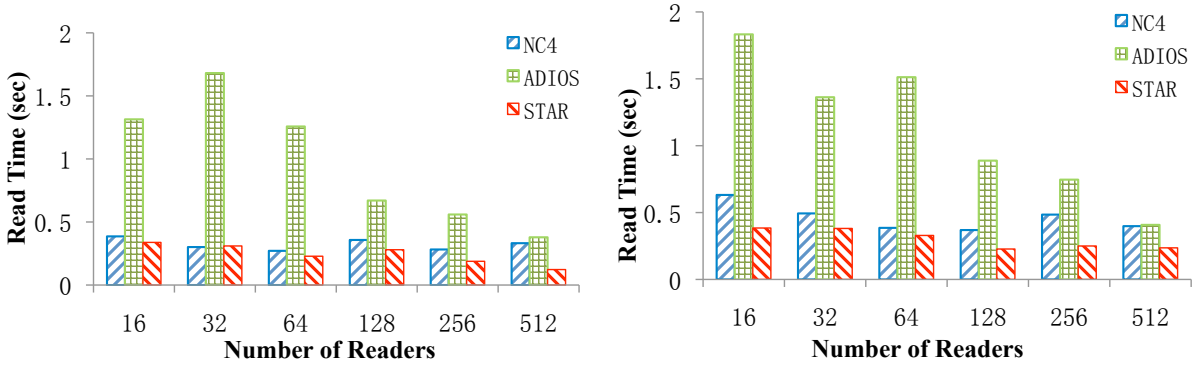


Figure 6.7: Data Output Elapse Time with Duo-aggregation (30 time steps, Temporal Aggregation=30 time steps)

As shown in Figure 6.7, the I/O time is further reduced by using TAR and SAR together. This is because SAR reduces the number of write requests by the order of 4 for 2-D domain decomposition, leading to further reduced contention at the storage. More importantly, good scalability is demonstrates by using such strategy. Even with the maximum of 8,192 processes, performance improvement is still observed. This is because the number of write processes is reduced to 512 by using a 2-level SAR. A maximum of 11 times speedup is achieved compared to NC4, and the speedup is only 4 times with only TAR.

6.3.3 Planar Read of 1 Time Step

For read performance, we mainly focus on the performance of reading a plane of data elements from a multidimensional variable. Such planar read is the most common and very challenging access pattern in data post-processing [92]. In this experiment, the variable is initially written by 4,096 processes. Three 2-D slices, namely (lat, lon), (lat, alt) and (lon, alt), are read from a 3-D variable on three different dimensions. Using ADIOS, the variable is divided into 4,096 chunks after 2-D domain decomposition. Note that temporal aggregation does not change the read performance at each time step. Therefore this experiment mainly reflects the effectiveness of spatial aggregation. A 2-level SAR is applied on the variable produced by 4,096 processes with STAR. We vary the number of readers from 16 (the core count of one compute node on Jaguar) to 512. These choices are made based on the understanding that application scientists typically use 10% or less processes for reading out of the original number of writing processes. We evaluate the performance of reading three 2-D slices from a half degree variable and a quarter degree variable. The results are shown in Figure 6.8.



(a) Half Degree 3-D Variable (Dimension: 576x361x48)

(b) Quarter Degree 3-D Variable (Dimension: 1152x761x48)

Figure 6.8: Planar Read Performance Within 1 Time Step (Total read time of 3 2-D planes on 3 dimensions)

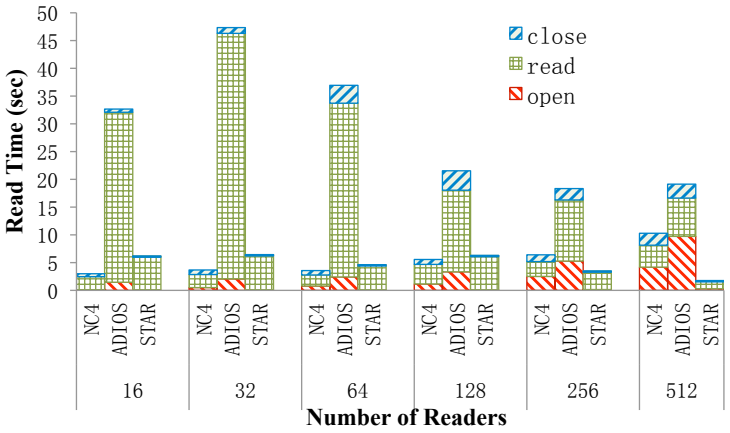
As expected, intensive chunking on small variables causes the performance of ADIOS to suffer due to a large number of seek operations. Increasing the number of readers helps alleviate such overhead. By applying SAR, the number of seek and read operations is reduced

by a factor of 4. With level-2 SAR, such operations are reduced to the 1/16 of the original ADIOS. Accordingly a fast read is observed for STAR. Such good performance is also scalable with an increasing number of readers. Similar trend is observed when reading a quarter degree variable, as shown in Figure 6.8(b). Overall, STAR demonstrates up to 3 times speedup compared to NC4, and 6 times speedup compared to the original ADIOS.

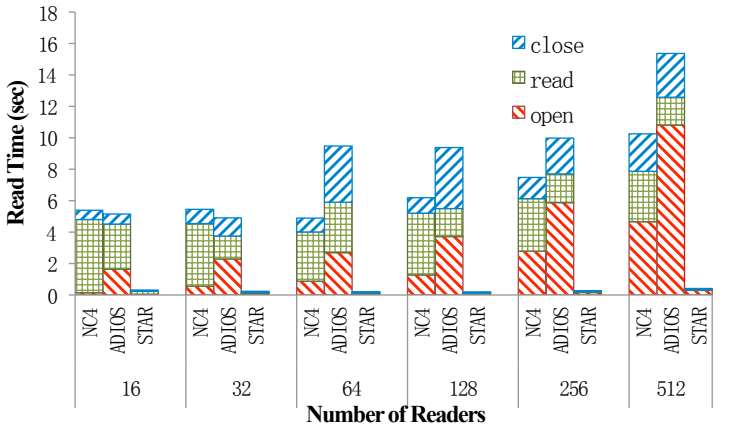
6.3.4 Planar Read of 30 Time Steps

Examining the changing values of data variables over time is a common pattern for data analytics. In this experiment, we evaluate the performance of reading a 2-D plane from 3 dimensions from a 3-D variable across all 30 time steps. That logically means that data is read in three kinds of subsets, namely (lon, lat, t), (lon, alt, t) and (lat, alt, t). The initial data was generated by 4,096 processes. With STAR, a 2-level SAR and 30 Time Steps aggregation are applied to the variable during data generation. The original GEOS-5 simulation produces one file per time step. Since temporal aggregation does not push out data immediately at the end of each time step, the number of output files is determined by the degree of time aggregation N, that is $\frac{30}{N}$. Therefore, only 1 file is generated for STAR.

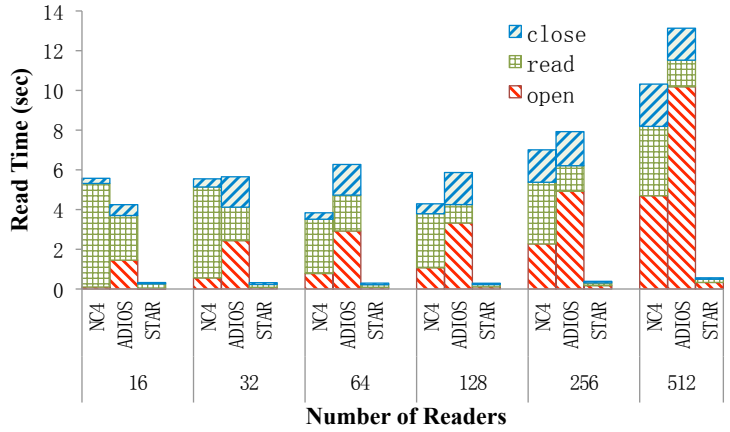
The time to retrieve a 2-D slice on three dimensions across 30 time steps together is shown in Figure 6.9. The time breakdown is also shown in the figure. As we can see, the original NC4 data layout exhibits good performance in the case of (lon, lat, t), for which data is contiguously stored. The time on the other two dimensions are more dominated by I/O because the contiguity is broken. The performance of NC4 is not scalable. To analyze data across 30 time steps requires each process to retrieve data from 30 files. This causes a big number of file open and close operations, which become a bottleneck with the increasing number of readers. Thus the performance of the original ADIOS suffers further from the overhead of metadata operations such as open and close. STAR only generates one output file therefore its performance is less affected by such metadata operations. More importantly, unlike the case of NC4 in which the majority of the time is spent on reading the



(a) var(lon, lat, t), constant alt



(b) var(lon, alt, t), constant lat



(c) var(alt, lat, t), constant lon

Figure 6.9: Planar Read Performance of 30 Time Steps (Multiple Files, Half degree variable)

data, STAR demonstrates very efficient read operations with temporal aggregation. Overall, STAR achieves the best performance on three dimensions. A maximum of 39 times speedup is achieved by STAR compared to the NetCDF4 method in the original GEOS-5.

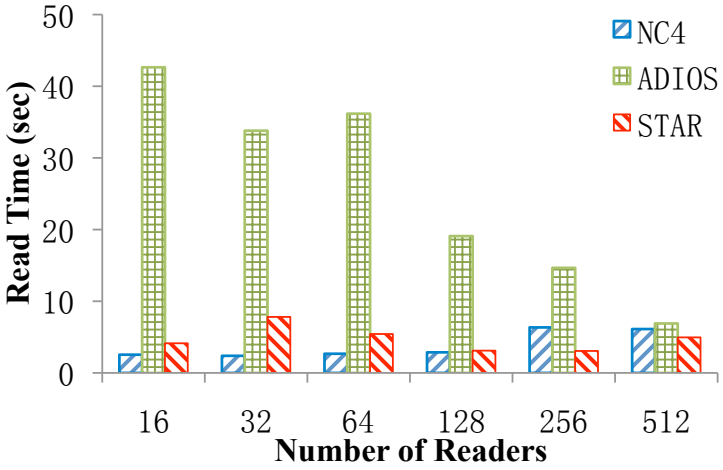
There are also cases where applications generate one single file by appending the time step outputs. In this case, the overhead of metadata operations are the same for different data organizations. Thus the performance difference is due to the actual reading of data. We examined the performance of planar reads across 30 time steps for the quarter degree variable. Data from all time steps are stored in one file. Figure 6.10 shows the experimental results. As we can see, STAR again achieves the best performance A maximum of 64 times speedup is achieved compared to the case of NC4.

6.3.5 Read Performance of 1-D Subset on 30 Time Steps

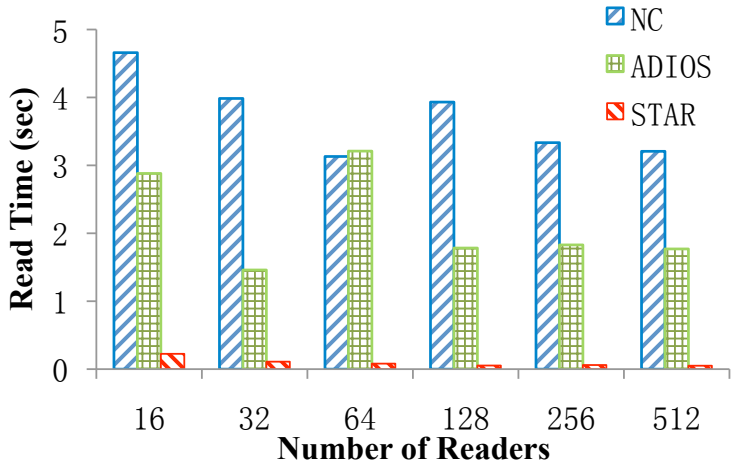
Another common access pattern for data analytics is to read a 1-D subset of variables across time steps. Such access pattern can be expressed as reading a subset of var at (lon, t), where lat and alt are constant, or reading a subset at (lat, t), where lon and alt are constant. Data variables of GEOS-5 only have 48 data points in alt dimension, and reading in (alt, t) dimension is also rare in practice, therefore we do not include it in this test case. Figure 6.11 shows the results for such access patterns. As we can see, while the performance of NC4 and ADIOS suffers from storage contention and a large number of read requests, STAR demonstrated a significantly improved performance in both cases. A maximum of 73 times speedup is achieved compared to NC4.

6.3.6 Visualization Results

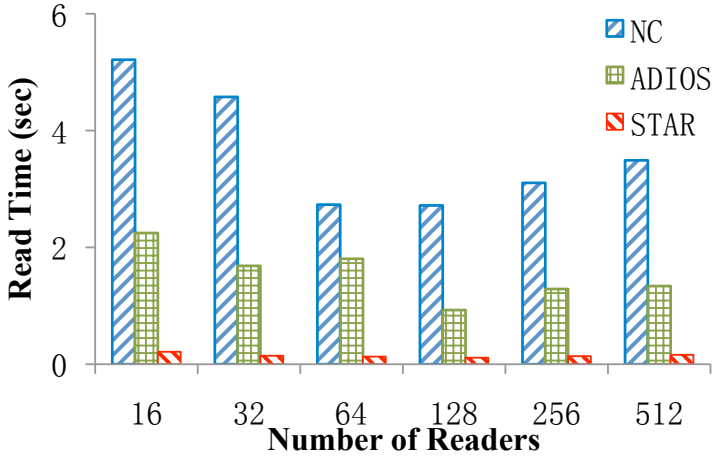
Figure 6.12 compares the performance of reading 1-D subset across time steps, that is (lat, t), using the temperature variable [3]. A fraction of a 1-day simulation output that consists of 10 time steps is shown in the figure. To retrieve such data for visualization in a half degree GEOS-5 simulation, STAR is able to render these images 11 times faster compared to the original NetCDF4 method in GEOS-5.



(a) var(lon, lat, t), constant alt

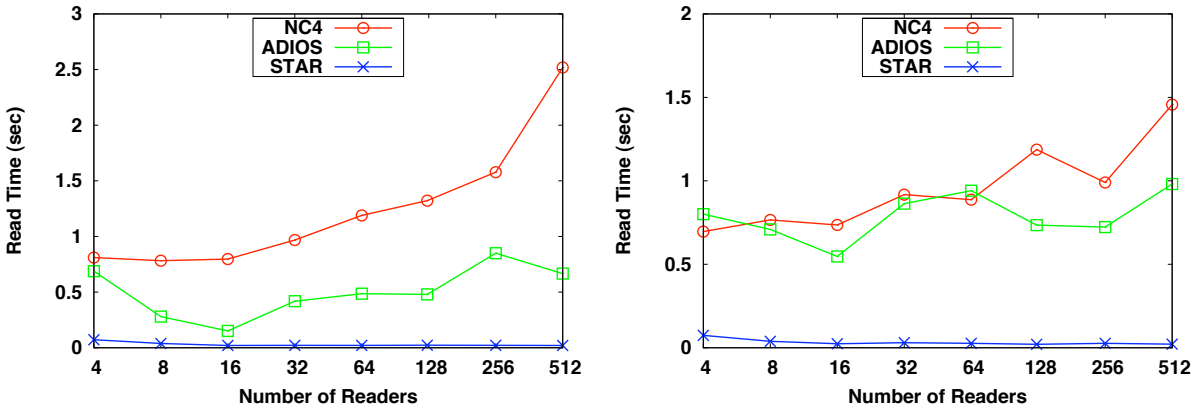


(b) var(lon, alt, t), constant lat



(c) var(alt, lat, t), constant lon

Figure 6.10: Planar Read Performance of 30 Time Steps (1 Files, Quarter degree variable)



(a) var(lon, t), constant (lat, alt)

(b) var(lat, t), constant (lon, alt)

Figure 6.11: 1-D Read Performance of 30 Time Steps (1 Files, Quarter degree variable)

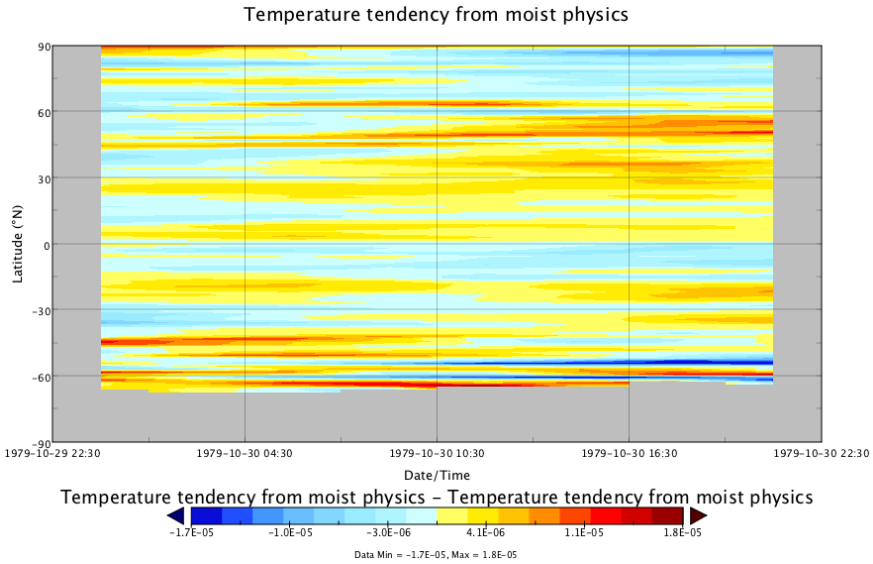


Figure 6.12: A Visualization Output of temperature(lat, time) for 1 Day Simulation (10 Time Steps)

6.4 Summary

To enable high performance I/O for scientific applications and to provide fast read performance for data analytics on time dimension, we propose Spatial and Temporal AggRegation (STAR) I/O scheme. STAR dynamically reorganizes data for scientific applications. Such strategy improves the performance for mission critical applications such as GEOS-5 for NASA climate modeling. Two key strategies are proposed. First, spatial aggregation with a hierarchical topology is designed to consolidate small data blocks with limited network communication so that large writes can be consolidate to fewer I/O processes, at the same

time alleviating the overhead for reading. Second, temporal aggregaion is designed to open up another dimension of data merging to further construct large data blocks. This novel strategy also provides an efficient read performance for analytics with a temporal series of data. The coordination of spatial aggregation and temporal aggregation is carefully designed to better utilize the memory resources. STAR has been designed and implemented within the ADIOS I/O middleware so it can be easily adopted by any application. As a case study, we have enabled STAR for the Goddard Earth Observing System Model (GEOS-5) from NASA by extending its diagnosis data I/O component. With a much simplified I/O flow and a system-friendly data organization, an efficient I/O speed is demonstrated for both writing and reading. Our experimental results on Jaguar supercomputer at ORNL have demonstrated a maximum of 11 times speedup for the write performance of GEOS-5, and 73 times speedup for the performance of data post-processing.

Chapter 7

Nearline Data Compression and Decompression

Advances on multicore technologies lead to processors with tens and soon hundreds of cores in a single socket, resulting in an ever growing gap between computing power and the available memory and I/O bandwidth for data handling. It would be beneficial if some of the computing power can be transformed into gains of I/O efficiency, thereby reducing this speed disparity between processors and I/O devices.

A scheme named NEarline data COmpression and DECompression (neCODEC) for scientific applications is proposed in this chapter. NeCODEC is designed to enable data compression at the MPI-IO level. It aims to provide a portable utility for many applications. Several techniques are introduced in neCODEC, including a nearline thread, elastic file representation, distributed metadata handling, and balanced subfile distribution. First, neCODEC is designed with a service thread for nearline data compression. It avoids burdening the main thread of applications with the onus of computationally intensive compression tasks. Secondly, to organize compressed data for efficient storage and easy retrieval, neCODEC is designed with an elastic file representation. A neCODEC file is composed of an elastic number of data files (a.k.a subfiles), and a metafile that stores index records to data blocks in the subfiles. Thirdly, to provide scalable management of metadata records, distributed metadata handling is provided in neCODEC. Lastly but also importantly, neCODEC supports balanced subfile distribution on parallel file systems such as Lustre [20]. With a balanced distribution of subfiles, neCODEC ensures that bandwidth from all storage devices can be effectively leveraged

In this chapter, we first discuss the motivation of this work in Section 7.1. Then we describe the design and implementation of neCODEC in Sections 7.2 and Sections 7.3, respectively. We then provide our experimental results in Section 7.4. Finally, a summary of this study is provided in Section 7.5.

7.1 Motivation

The execution of a scientific simulation normally consists of multiple iterations of two major phases: computation and I/O. The computing cores perform specified mathematic calculation during the computation phase. The generated data then is written out during I/O phase. Multiple time steps involve the repeated execution of these two phases, as shown in Figure 7.1 where three time steps of simulation is executed. However, due to the slow storage devices, often times the computing cores have to stay idle and wait for the completion of data movement, which is inefficient. The core design of neCODEC is to bridge such gap by leveraging the computing power to reduce the amount of data for output, and free the computing cores from waiting through asynchronous I/O technique. As shown in Figure 7.1, a compression phase is inserted after computation. Once the compression is complete, the data will be passed to a dedicated service thread for I/O. While the computing cores can move on to the next step of computation. Even though the overhead is introduced because of compression, the simulation is speeded up through reduced I/O time and I/O pipelining. Similar overhead is introduced to the reading. The read cost now is the sum of the I/O time and decompression time. However, performance gain can be expected by using an efficient compression algorithm which also produces high compression ratio. Finding the efficient compression algorithm is not the focus of this work. Instead, we mainly study a good framework that efficiently enables the compression algorithms for scientific data.

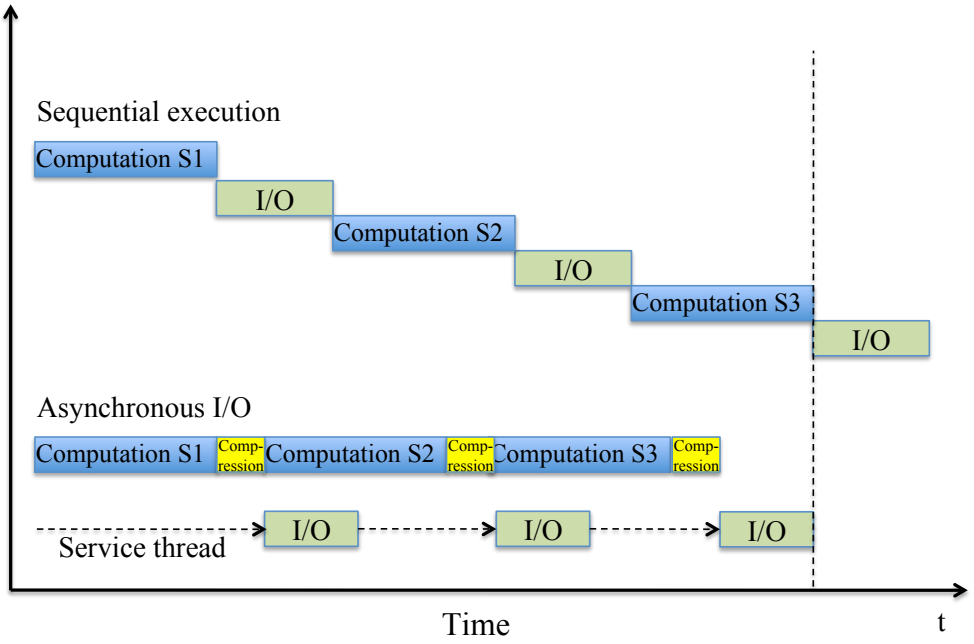
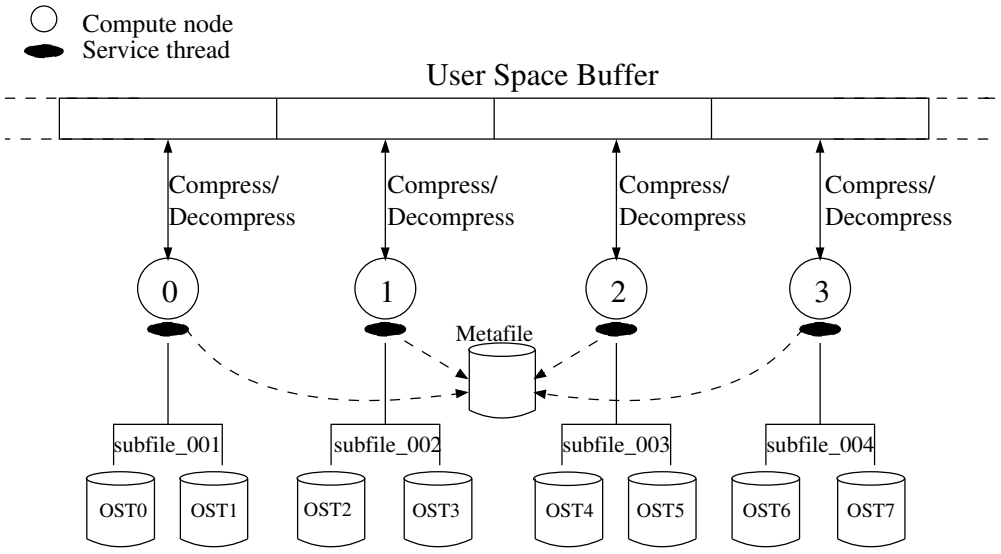


Figure 7.1: The Execution of Scientific Simulation of 3 Time Steps

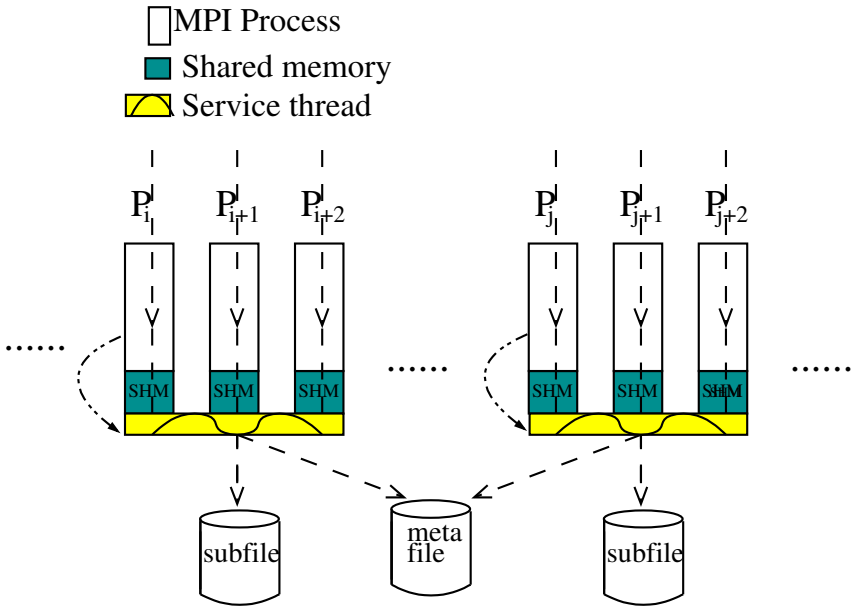
7.2 Design of NeCODEC

To reduce data redundancy in scientific applications, we propose a data reduction strategy named neCODEC for nearline data compression and decompression. neCODEC uses the Optimized Chunk Size as compression unit, and provides a distributed file representation and balanced subfile dissemination. neCODEC is targeted for systems that consist of a set of compute nodes, all connected through an interconnection network to a set of storage nodes. neCODEC requires no software modification at the storage side.

In the following of this section, we first describe software components of neCODEC and then discuss the design of neCODEC in detail, focusing on elastic file representation, data segmentation and compression, balanced subfile distribution, and distributed metadata handling.



(a) Architecture



(b) Interaction of Threads and Processes

Figure 7.2: Software Components of neCODEC

7.2.1 Software Components of neCODEC

Figure 7.2 shows the software components of neCODEC. neCODEC is targeted for systems that consist of a set of compute nodes, all connected through an interconnection network to a set of storage nodes. neCODEC requires no software modification at the storage

side. For data originally to be stored as a file, neCODEC will create a metafile and several separated data files (subfiles). neCODEC creates a dedicated service thread per node to support nearline (asynchronous) data compression. The service thread is spawned by the first process on a node. Main efforts in neCODEC will be focused on offloading all I/O related data processing from MPI processes to this service thread in an efficient manner. As shown in Figure 7.2(a), application data is first divided into chunks among MPI processes. Then all MPI processes pass their chunks to the service thread on the same node for compression. Figure 7.2(b) shows the service threads on two compute nodes and their interaction with MPI processes through shared memory.

7.2.2 Elastic File Representation

File representation of neCODEC is a conceptually simple design. It divides application data into small chunks, and then compresses them using a configurable algorithm, currently zlib. Alternative algorithms such as bzip2 or other lossless compression algorithms for scientific data can be plugged into neCODEC.

Figure 7.3 shows the overall file format of one metafile and two subfiles. The metafile carries the original file name which is "perf". The subfile name is a combination of the original file name and a subfile index. It is formatted as: .OriginalName_xxx. "xxx" stands for the subfile index. Thus two subfiles are named .perf_001 and .perf_002, respectively. Subfiles are dynamically created based on the growth of the original file. For each chunk, a record is created to store its attributes such as subfile location, size, and offsets. This record is then inserted into the metafile. The compressed data chunk is stored into a subfile.

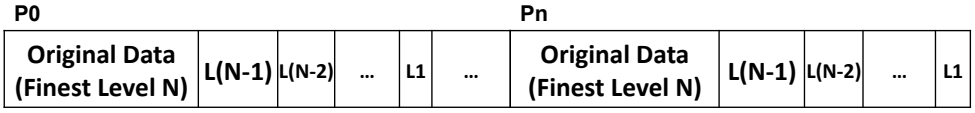


Figure 7.3: The Format for a neCODEC File

A neCODEC metafile consists of two types of records. It starts with a fixed-size header record, followed by an array of metadata records, one per data chunk. Below we describe their formats in detail.

Format of Header

A metafile consists many fixed-size records. The first record is called a metafile header. It is padded to the same size as the other regular metadata records. The header enables quick access to general information about the file. Currently, it contains three fields:

- **Magic Number** – A predefined number used to identify the neCODEC file format.
- **File Size** – The size of the original file.
- **Max Index** – The biggest index number of all metadata records.

Format of Metadata Records

Metadata records contain the information indicating the location and attributes of compressed data chunks in separated subfiles. The length of each record is 64 bytes. The fields include:

- **Segment Index** – The global index of the chunk in the original file. It is calculated by the chunk offset in the original file (*Original Offset*) divided by the chunk size.
- **Segment Length** – The actual length of the chunk after compression.
- **Offset** – The beginning offset of a compressed chunk inside its subfile.
- **Original Offset** – The beginning offset of a chunk in the original file (without compression).
- **Subfile Index** – The index of the subfile that contains the compressed chunk.

- **Miscellaneous Attributes** – A one-byte field that stores two attributes. The first bit is a flag that is set when the data chunk is compressed. The remaining 7 bits are used to store the version number. These attributes are described further in Section 7.2.3.
- **Padding** – This is currently unused space. It can be used in future to store a hashed digest for the data chunk.

7.2.3 Data Segmentation and Compression

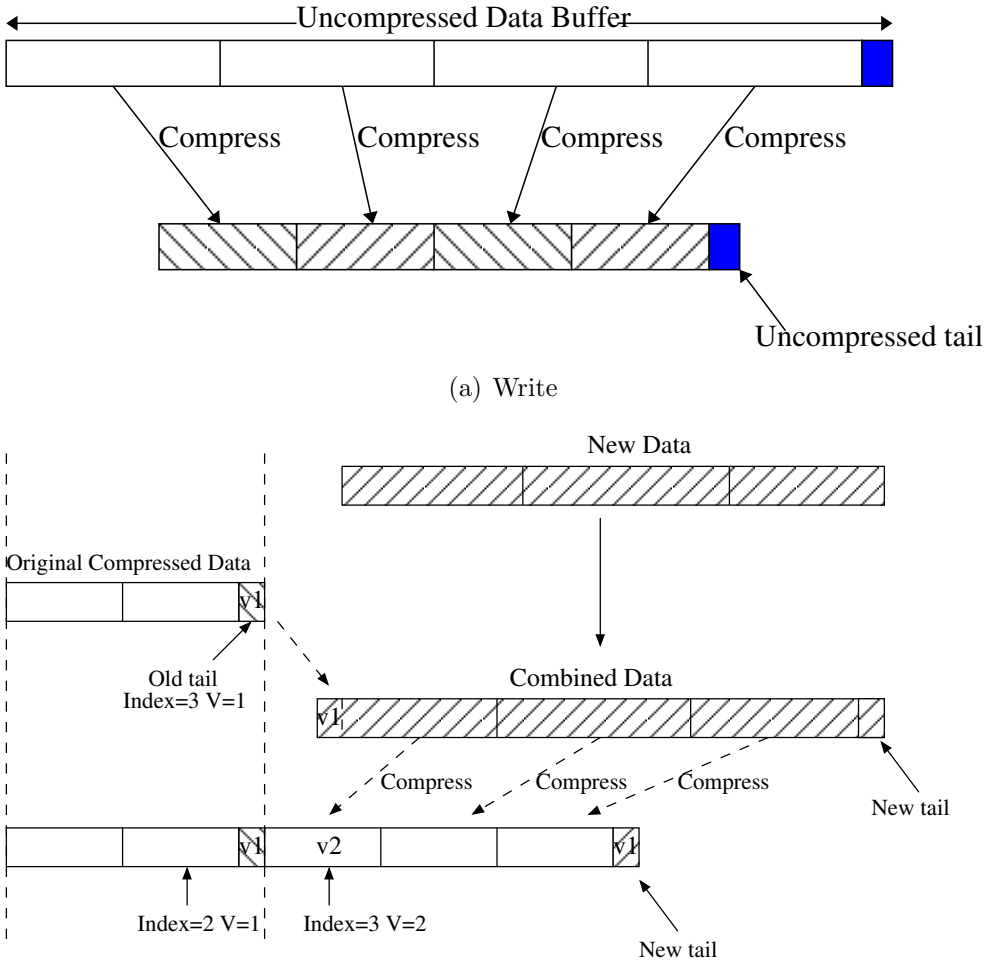


Figure 7.4: Data Compression during Write

Figure 7.4(a) shows data segmentation and compression in neCODEC. Application data is divided into fixed-size data chunks. The actual chunk size is adjustable during run-time

(by using MPI hints). Each data chunk is compressed before it is stored into a subfile. A metadata record is generated for each compressed data chunk, and then stored into the metafile. When the data buffer cannot be divided into chunks evenly, the flanking regions will be written into the subfile directly without compression. To read the data back, a process first retrieves the metadata record. Then it locates the corresponding subfile, offset, and length for the data chunk. Data is then read from the subfile, decompressed, and returned to the application.

Uncompressed partial chunks caused by unaligned flanking regions require special handling. Particularly, when the remaining data arrives, the partial chunk needs to be combined with new data to form a new chunk. We adopt a virtual file appending approach as shown in Figure 7.4(b). Every chunk in the file is tagged with a version number and a flag that indicates whether it is compressed. When a new chunk is formed, its metadata record is updated accordingly. To distinguish two versions of a data chunk with the same offset, a version number is stored inside the metadata record. This allows us to purge the stale chunks at a later point of time. Similar approach is adopted when a file segment is to be overwritten. An updated chunk will be marked with an increasing version number. When the file size grows to a certain level, a chunk purging operation can be performed to remove old chunks. These old chunks are found by traversing metadata records.

Balanced Subfile Distribution

Subfiles are dynamically created based on the growth of data. Any subfile can be selected to host incoming data chunks, thereby enabling elastic file growth. In addition, on Lustre file system, a balanced distribution is enabled for subfiles to leverage aggregated bandwidth from all storage devices. The striping pattern of a subfile consists of three parameters: stripe size, stripe width, and stripe index. Based on our earlier study [101], we organize subfiles in a hierarchical manner so that all subfiles are evenly distributed and striped to all storage devices, without causing the overhead of wide striping. Figure 7.2 gives an example of the

distribution of subfiles. All subfiles have a stripe width 2, with the first subfile being striped from the first storage device. Stripe indices of other subfiles are set accordingly.

7.2.4 Distributed Metadata Handling

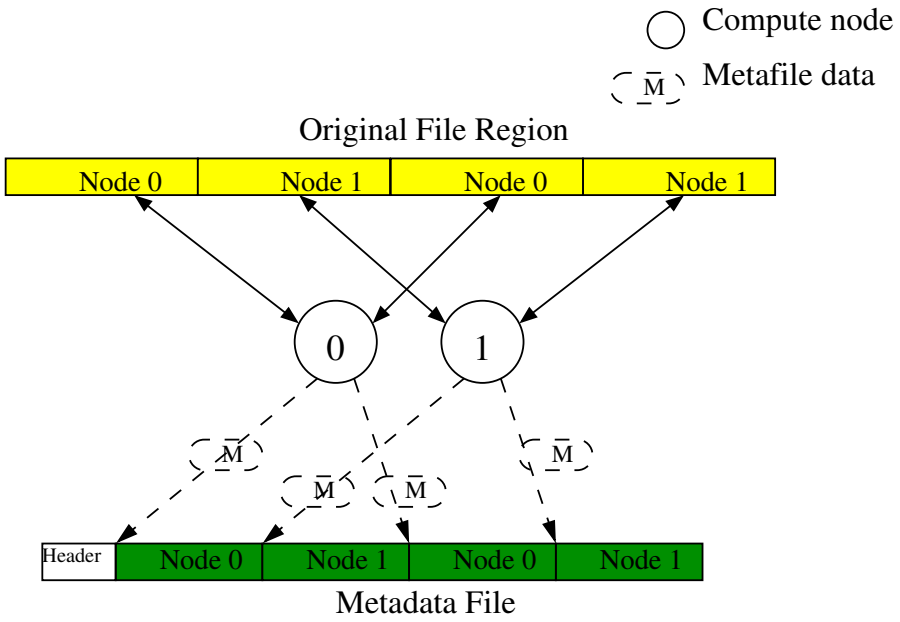


Figure 7.5: Metadata Distributed on 2 Nodes

In neCODEC, every node creates one service thread which writes metadata records to a shared metafile. When there are a large number of processes performing I/O, it is critical to enable an efficient approach for metadata access. To meet this requirement, we developed a distributed metadata handling scheme. The entire metadata for a file is fully distributed to all service threads. All service threads allocate a block of memory as the metadata buffer for storing transient metadata records. When the buffer usage reaches a threshold percentage, e.g., 80%, the service thread will flush all existing metadata records into the on-disk metafile. In the meantime, new records can still be inserted into the other 20% of the buffer. In this scheme, all metadata is stored as large and sequential blocks to the disk, allowing efficient disk access. For a read request, a service thread will first search for metadata records from its metadata buffer. If it is not found, the service thread

will retrieve a number of contiguous metadata records, along with the required ones. Such design allows easy integration with data staging or prefetching techniques. With reasonable locality, this scheme combines multiple metadata records into one, and avoids the cost of frequent metadata access.

Note that, as our initial effort to study for the benefits of compression, we strictly divide the content of a file among MPI processes. Thus each service thread is managing a distinct set of metadata records. The consistency issue among metadata buffers from different threads is then avoided through this rigid distribution. In the future, we plan to investigate the feasibility of dynamic partitioning methods.

7.3 Implementation

We have implemented a prototype of neCODEC by modifying the MPI-IO implementation of MPICH2. In this prototype, several other implementation details are worth mentioning here. First, the majority of the code is introduced as an extension to the ADIO implementation of Lustre file system. Second, the service thread is created by the first MPI process, and it is not created until the first call to open (create) a file. This thread stays alert for upcoming I/O requests from any process on the same node. It will not exit until the MPI_Finalize() is called. Third, data cached in the service thread will be flushed either when the corresponding file is closed by the application, or when an explicit file synchronization request is made.

Because disk access is expensive, we support data caching in neCODEC to minimize the frequency of disk access. Two shared memory buffers are allocated for each process. One is a metadata buffer (currently 2MB), whose usage is described in Section 7.2.4. Another is the cache for compressed data chunks. Similar to the metadata cache, the data chunks will be flushed to the disk file when they have taken up more than 50% of the total data cache.

7.4 Performance Evaluation

Our experiments were conducted on the QueenBee cluster from Louisiana Optical Network Initiative (LONI). QueenBee consists of 668 nodes. Each node is equipped with dual-socket quad-core Intel Xeon 64-bit 2.33GHz processors, 512KB cache, 8GB physical memory (1GB per core). All nodes run Red Hat Enterprise Linux 4 operating system. The storage system is a 58TB Lustre file system with 16 Lustre OSTs. In this paper, we focus on the performance of write operations, while initial read performance is also evaluated to examine the performance impact of metadata handling.

7.4.1 Independent I/O

For independent I/O performance measurements, we use *perf.c* from the ROMIO distribution. This program performs concurrent writes to a shared file. Each process writes a contiguous 24MB data at disjoint offsets based on its rank in the program. We fill in the test data with random alpha-numerical characters. Our measurement indicates that such data has a compression ratio of 2.3. Compression ratio is defined as the ratio between the size of the original data and the size after compression (via zlib). This generates a maximum of 3GB data size with 128 processes. The average time taken for each iteration is recorded. Because neCODEC uses a dedicated asynchronous service thread to perform I/O, data chunks are buffered in the service thread when MPI_File_Write() returns. The service thread later flushes data chunks when the buffered data reaches a threshold or when the file is closed. So we calculate the write time as the sum of MPI_File_Write() return time and the actual I/O time inside the service thread. Because neCODEC uses a dedicated I/O thread, which may impact performance when MPI process layout varies, we thoroughly examine the write performance with different process arrangement.

Each node on QueenBee is equipped with 8 compute cores. When a program is launched to use all 8 cores, the service thread of neCODEC will compete with the regular MPI processes for compute cores. To examine the effect of using one service thread, neCODEC

is tested with different numbers (7 and 8) of cores (processes) per node. The "original" case for ROMIO is always run with 8 processes per node as theoretically its performance is not influenced by the MPI process arrangement.

Figure 7.6 shows the performance comparison between ROMIO and neCODEC. Lustre enables client data cache in the kernel. To mitigate caching effects, we calculate the I/O time of ROMIO by forcing a file sync operation after the write operation. For complete comparison, we also include the performance of independent I/O operations without file sync. As shown in Figure 7.6, ROMIO outperforms neCODEC in most cases with caching effects. However, the scalability issue is also observed. Hierachical striping helps neCODEC achieves good scalability and eventually surpasses the original ROMIO (no sync) on 128 processes. When file sync operations are used to reduce caching effects, neCODEC outperforms the original ROMIO in all cases, with either 7 or 8 processes per node. When neCODEC is run with 8-process per node, the performance is constrained by the competition for CPU between service thread and processes. With 7-process per node, neCODEC benefits greatly from the dedicated core for compression and delivers significantly higher bandwidth. Compared to ROMIO (with sync), neCODEC achieves a bandwidth improvement up to 3 times with 128 processes.

Taken together, the performance evaluation of independent I/O tests, we conclude that neCODEC is able to improve I/O bandwidth for highly compressible data by leveraging a portion of the CPU power. A dedicated compute core will help neCODEC significantly. So one compute core is reserved on each compute node in the following experiments unless otherwise indicated.

7.4.2 Performance of MPI-Tile-IO

MPI-Tile-IO [79] tests the performance of tiled access to a two-dimensional dense matrix, simulating the type of workload that exists in some visualization applications and numerical applications. In our experiments, each process renders a 1×1 array of displays, each with

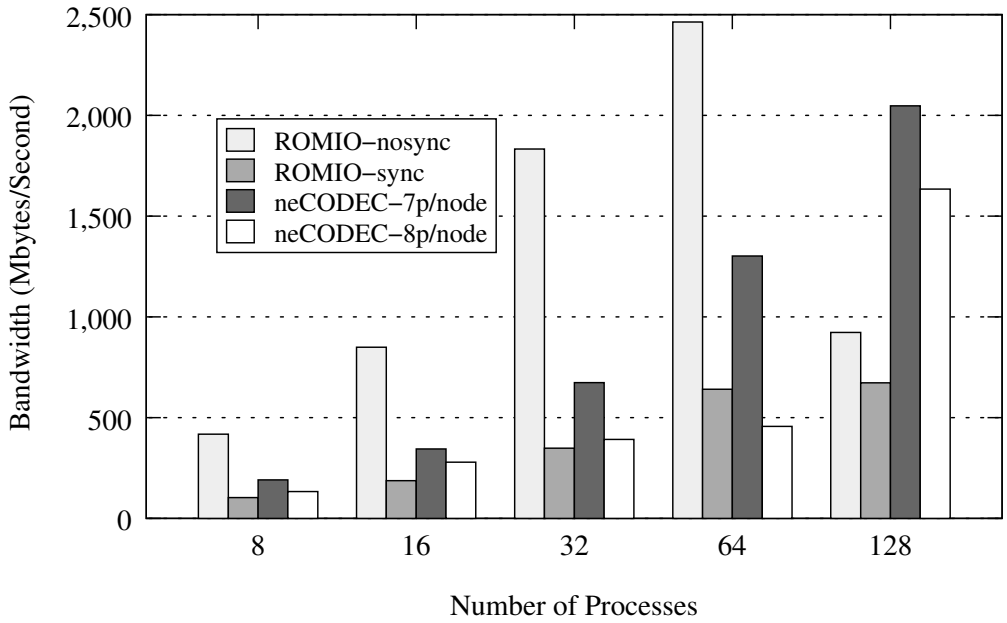
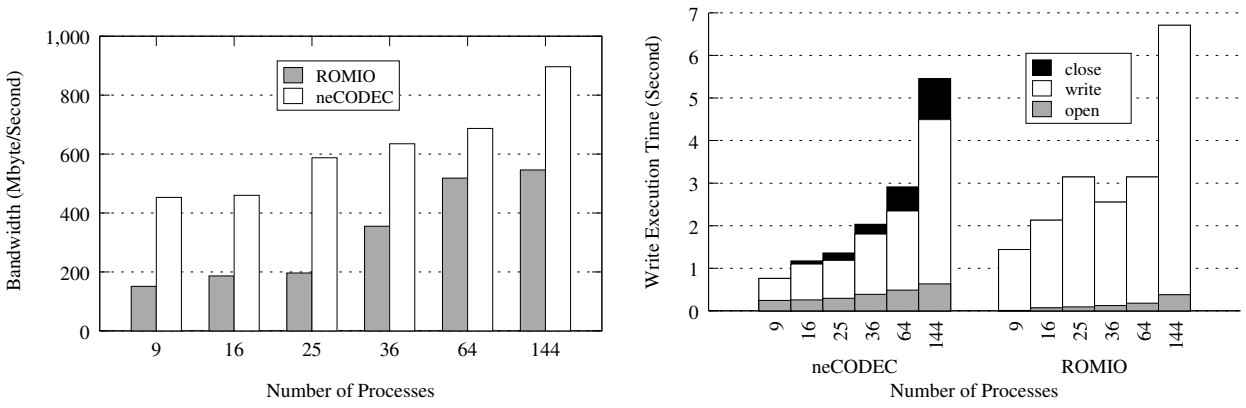


Figure 7.6: Independent I/O Performance

2048×1536 pixels. The size of each element is 8 bytes, leading to a file size of $24 * N$ MB, where N is the number of processes.



(a) Write

(b) Write Time Breakdown

Figure 7.7: MPI-Tile-IO Write Performance

Figure 7.7(a) shows the performance improvement of neCODEC compared to ROMIO. We observe that neCODEC outperforms the original ROMIO in all cases. A maximum of 3

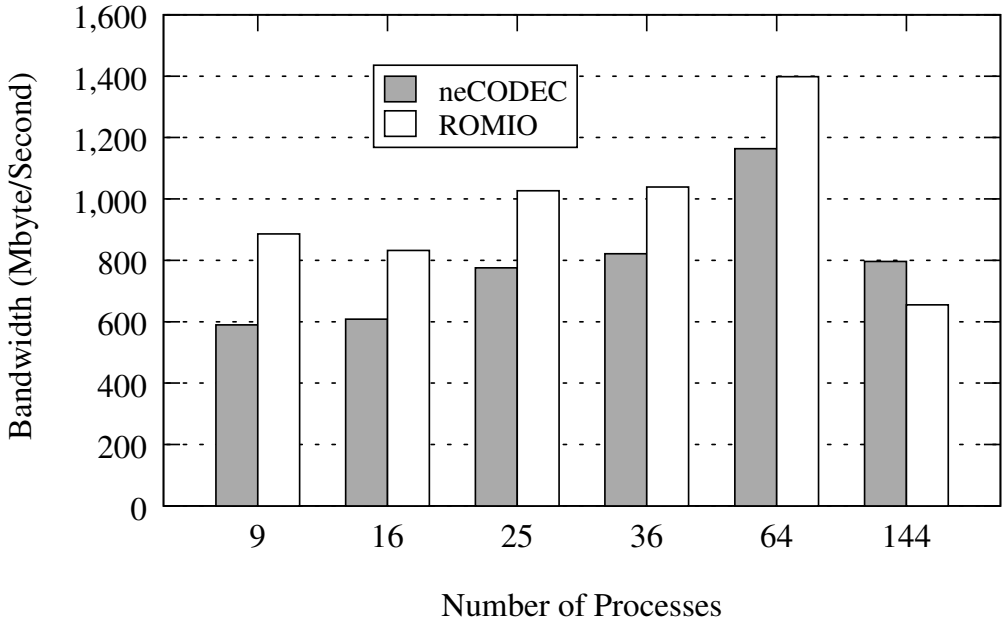


Figure 7.8: MPI-Tile-IO Read Performance

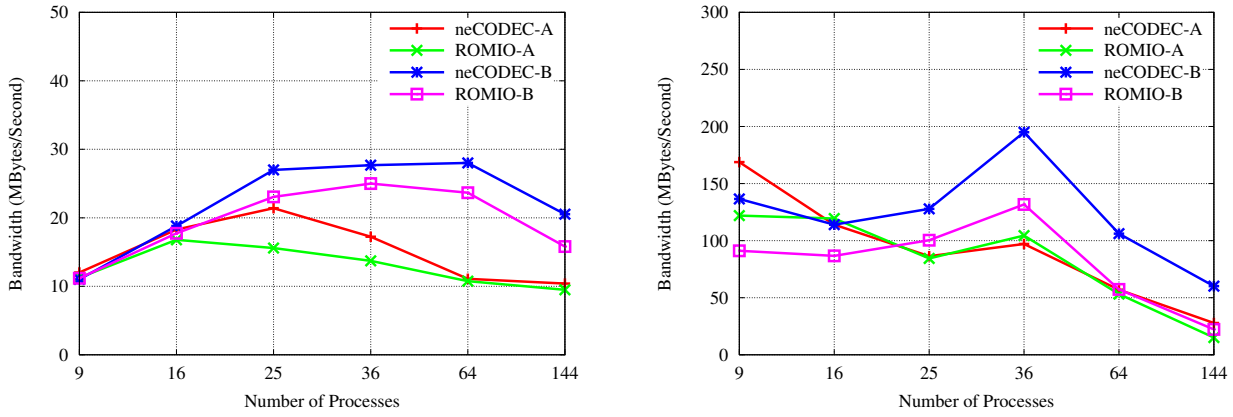
times improvement is achieved with 25 processes, while 1.6 times bandwidth improvement is achieved with 144 processes. Moreover, neCODEC presents a good overall scalability.

We further examine the time breakdown of MPI-Tile-IO. In the case of neCODEC, the file *open* time includes the time to spawn the service thread; the *write* time is a combination of MPI_File_Write() and the write() system call operation inside the service thread; and the *close* time includes the time to flush cached data and metadata records to the disk. Figure 7.7(b) shows the comparison of timing breakdown between ROMIO and neCODEC. Spawning service threads does not significantly increase the file open time. However, when a file is closed, much time is spent by the main thread to synchronize with the service thread, which has to flush cached data and metadata into storage. However, compressed data leads to much reduced time in writing the file. The savings from the file write time actually outweighs the overheads from file open and close calls. Therefore, this results in better I/O performance for neCODEC compared to ROMIO.

Figure 7.8 shows the read performance of MPI-Tile-IO when the data is initially in the disk. In this case, each process has to retrieve the metadata before reading data from

disk. We observe that, due to the overhead of metadata retrieving, neCODEC performs worse than the original ROMIO except when there are 144 processes. While neCODEC is able to support good read performance at scale, we plan to incorporate techniques such as data staging to avoid the initial metadata overhead in neCODEC. As an initial prototype, neCODEC already exhibits good performance benefits, and presents a viable I/O technique for data-intensive applications.

7.4.3 Performance of NAS BT-IO



(a) Write (b) Read

Figure 7.9: Performance of BT-IO

NAS BT-IO [98] is developed at NASA Ames Research Center based on NAS BT (Block-Tridiagonal) parallel benchmark. The entire dataset undergoes diagonal multi-partitioning and is distributed among a square number of MPI processes. Its data structures are represented as structured MPI datatypes and written to a file periodically, typically every 5 timesteps. There are several different BT-IO implementations, which vary on how its file I/O is carried out among all the processes. In our experiments, we use an implementation that performs I/O using MPI-IO collective I/O routines, so called *full mode* BT-IO. In this mode, BT-IO performs 40 iterations of collective writes, followed by a verification phase

which performs 40 iterations of collective reads. We run BT-IO with 9 to 144 processes and two different classes. BT-IO Class A generates 400 MB and Class B 1697.93 MB.

Figure 7.9(a) shows the write performance of BT-IO. NeCODEC is still able to improve the effective I/O bandwidth for BT-IO even when the data is not highly compressible. Bandwidth improvements of 37% and 18% are observed for Class A and Class B, respectively. The peak bandwidth of Class A is achieved at 16 processes. This is because the Class A data size is relatively small. More writers lead to finer grained data elements for each process. Hence many smaller I/O requests are generated to Lustre, degrading the I/O performance. A similar trend is observed for the performance of Class B. For both classes, neCODEC exhibits better performance compared to the original ROMIO.

The read performance of BT-IO is shown in Figure 7.9(b). As BT-IO performs read immediately after write, neCODEC is able to retrieve its metadata from cache. Accordingly we observe better performance for neCODEC. A maximum of 49% improvement is achieved for Class B. For Class A, the performance gain is smaller.

7.4.4 S3D Combustion Simulation Application

S3D [16] is a combustion simulation application using direct numerical simulation (DNS) solver developed at Sandia National Laboratories. It solves fully compressible Navier-Stokes, total energy, species and mass continuity coupled with detailed chemistry. This code traditionally runs on a large number of processors, on many of the largest supercomputers in the Department of Energy and at the National Science Foundation centers. One of the more taxing parts of the simulation is spent in the I/O, and our group has done lots of research to expedite this process [51, 54, 4, 104].

Typically the data that is produced in the simulation includes three-dimensional Cartesian mesh points of four global arrays, (mass, velocity, pressure, and temperature), along with many chemical species. We learn from our interaction with S3D scientists that datasets are usually written as either small (20^3) or medium (50^3), or large (100^3) data points per

MPI process. In our performance evaluation we use the medium data size, which produces about 15.3 MB of data per process per checkpoint. Therefore, with 125 processes in our experiments, the total amount of data is 1.9 GB per output. We measure the time for ten checkpoints, and show the results in Figure 7.10. In order to make the experiment comparable to "real" S3D data, we validate our runs with the early time steps of the S3D simulation. The datasets are confirmed to be highly compressible.

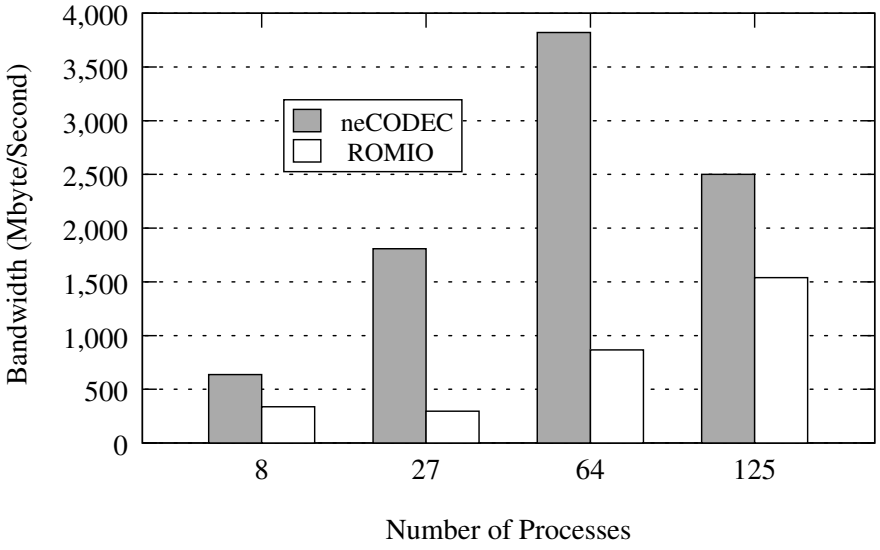


Figure 7.10: S3D Performance

We observe significant performance improvements with neCODEC compared to the original ROMIO in all cases. With 64 processes, neCODEC achieves about 3.8GB/sec bandwidth, which is more than 4.4 times improvement of the original ROMIO. The bandwidth drops at 125 processes. We suspect that it can be partly attributed to the increasing number of metadata records and the metadata management overhead. Thus we further evaluate the performance of S3D with different chunk sizes.

Tuning neCODEC Chunk Size for S3D

To further examine the impact of data management overhead, we measure the performance of S3D with different chunk sizes. Figure 7.11 shows the performance of ROMIO and

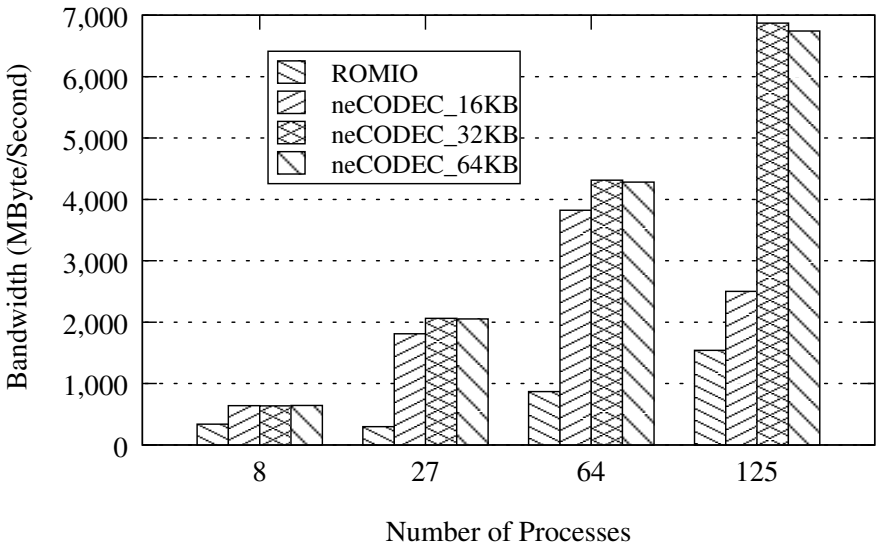


Figure 7.11: The Performance of S3D with Varying Chunk Sizes

neCODEC with different chunk sizes. As shown in the figure, the chunk size of 32KB leads to the best bandwidth, an improvement of 5 times compared to ROMIO on 64 processes, and 4.5 times on 125 processes. Compared to the chunk size of 16KB, 32KB leads to better compression ratio, and less number of data chunks, therefore more efficient metadata handling. The 64KB chunk size performs slightly lower than 32KB. This is because that an even larger chunk size at 64KB slows down compression, therefore degrading the overall performance.

7.5 Summary

In this work, we have explored the feasibility to design an I/O compression framework to leverage a portion of the computing power to compress and consolidate scientific datasets, and mitigate the speed disparity between multicore processors and I/O devices. To this aim, we have designed NEarline data COmpression and DECompression (neCODEC) to deliver efficient I/O for data-intensive scientific applications. A prototype implementation of neCODEC has been accomplished. Inside neCODEC, a number of salient techniques

are implemented accordingly, including a nearline service thread, elastic file representation, balanced subfile distribution, and distributed metadata handling. The performance of neCODEC is evaluated using a set of data-intensive microbenchmarks and scientific applications. Our experiments demonstrate that neCODEC can achieve an overall performance improvement for scientific datasets with different compositions and varying compression ratios. For a combustion simulation application, S3D, we show that neCODEC increases its effective bandwidth by more than 5 times.

Chapter 8

Future Work

There are three topics lays in the future work of this study.

8.1 Platform Migration

In previous chapters, a set of techniques to improve I/O performance for scientific applications have been represented. The majority of the study was performed on Cray XT series platforms which used Lustre file system. To gain general acceptance for wider range of applications, it is important for these techniques to be tailored and evaluated on other types of supercomputers and file systems. For example, the Intrepid system at Argonne National Laboratory is one of the fastest supercomputer in US. It is an IBM Blue Gene/P system equipped with GPFS file system. Intrepid has been serving as the platform for a large number of mission critical applications. Successfully adopting to such system can help our techniques contribute to more scientific codes within HPC community. The challenge lays in how to accommodate different networks and file systems. In particular, the data striping strategy of file system has great impact on the design of many of our techniques. Part of the efforts in the future work will be addressing these issues during the migration of the platforms.

8.2 Optimized Chunking Based Data Indexing

In chapter 5 presented the *optimized* data organization for multidimensional data. Such data organization provides a consistently good and balanced performance for common access patterns. One horizon to explore based on such technique is the integration with data indexing techniques. Data indexing allows the important scientific characteristics to be abstracted

and stored for an unit of data subset. Simple examples of such characteristics include the maximum, the minimum and the mean value of the subset. More sophisticated characteristics allows for more complex mathematical operation during data post-processing. With the help of such metadata, data analytics can be performed more efficiently by simply loading the data subsets that match with the query criteria. However, the granularity of the data unit is important. Because the finer granularity means more data indices will be generated and stored, which can become significant for gigantic volume of scientific output. While coarser granularity leads to larger data unit, and consequently more extra data retrieval even when query is only on a very small fraction of data unit. By building the data indexing based on our OCS model, reading is ensured to be efficient. The metadata overhead is also constrained to a limited range.

8.3 Next Generation Hardwares

Current design of our data organization techniques are all built upon the magnetic disk based storage systems. Even though replacing such storage system with the next generation storage hardware is not taking place instantly, our techniques need to be proactive to welcome the soon coming exascale computing. This intrigues us to investigate the validation of our design on new hardwares, and the corresponding strategies for new challenges. For example, Solid State Devices have average equal access time for random reads. Such hardware characteristic diminishes the dimension dependency for multidimensional data. However, the overhead of writing is more significant due to read-and-write operations of data blocks. What is the implication of such hardware organization to data organization? And how severe the data concurrency issue remains if the future storage system still can not match with the scale of the computation partition? Investigating these questions help us to pave our path to the upcoming exascale era.

Chapter 9

Conclusion

The discord among fast growing computing power, explosively increasing data volume, and slowly improving storage system has led to a severe I/O bottleneck for the advancing of scientific computing. To bridge the disparity between the mismatched components in scientific computing so a high performance I/O service can be achieved, this dissertation investigates the opportunities in finding and developing a harmonious organization between data and storage, so that the performance of the storage system can be best utilized for fast I/O services. The scope of this study covers issues during both data generation and consumption, and exploits the opportunities from the entire storage system as well as one single storage node. Four key data organization and management techniques are introduced for efficient scientific data organization and management. For system-wide performance, a Space Filling Curve based data reordering is proposed to shuffle the placement of the data chunks on storage nodes. By doing so, the bandwidth of the storage system can be effectively aggregated and exploited for challenging read access patterns of data analytics, particularly planar reads. For in-node performance, an Optimized Chunking model is introduced that provides a system-aware approach for finding the *correct* chunk size on a specific system. To enable the OCS model, a two-level data organization framework named SMART-IO is designed. For applications that are not yet benefit from current I/O techniques due to its challenging I/O patterns, an I/O scheme STAR is presented which utilizes Spatial Aggregation and Temporal Aggregation to bring their I/O performance to next level. In particular, the Temporal Aggregation is a novel technique that facilitates the data analytics on time dimension, a common practice in data post-processing but not yet well supported by current data layouts. Additionally, I examine the opportunity to take advantage of the performance

gap between computation and I/O. An Nearline Data Compression and Decompression strategy is proposed to leverage the computing power to consolidate data through compression algorithms. By mitigating the pressure from storage device to computing cores, both write and read performance are improved.

Majority of the study described in this dissertation has also been integrated with the ADIOS I/O component framework, allowing the large number of users of ADIOS easy access to this new technology. Implementing the data organization techniques within the framework of ADIOS, provides the technology with a large number of adopters in the scientific computing domain. For example, the Earth System Model GEOS-5 from NASA has addressed many challenging I/O issues by using the techniques described in this dissertation. Most of the Data Cheroraphy techniques are also being targeted for inclusion in the official release of ADIOS in November 2012, allowing any user of ADIOS to utilize such system-oriented data management framework, with a minimal level of source code modification.

In preparation for the upcoming exascale computing, a thorough understanding of the correlation between data and infrastructure is the key to a successful transition. This study has demonstrated to be part of the solution towards scalable data management techniques for the next generation of I/O frameworks. The combination of intelligent data placement, system-aware data organization, exploration in time dimension and asynchronous compression has showcased the opportunities and benefits of developing a *harmonious* organization between data and storage in High Performance Computing. This research has already gained significant adoption in the scientific domain and has extended the venue of I/O research for the next generation computing.

Chapter 10

Acknowledgement

This work is funded in part by UT-Battelle grants UT-B-4000099498 and UT-B-4000086682 to Auburn University, in part by National Science Foundation awards CNS-0917137 and CNS-1059376, and in part by a NASA award NNX11AR20G.

This research is sponsored in part by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. This research is conducted with high performance computational resources provided by the Louisiana Optical Network Initiative (http://www.loni.org). It also used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

Bibliography

[1] Adaptable I/O System. http://www.nccs.gov/user-support/center-projects/adios .

[2] NASA. http://gmao.gsfc.nasa.gov/systems/geos5.

[3] NASA. http://disc.sci.gsfc.nasa.gov/mdisc/data-holdings.

[4] H. Abbasi, G. Eisenhauer, M. Wolf, and K. Schwan. Datastager: scalable data staging services for petascale applications. In *HPDC '09*, New York, NY, USA, 2009. ACM.

[5] H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky. Extending i/o through high performance data services. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, 31 2009-sept. 4 2009.

[6] N. Adiga, G. Almasi, G. Almasi, and et al. An overview of the bluegene/l supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[7] Argonne National Laboratory. ROMIO: A High-Performance, Portable MPI-IO Implementation. http://www-unix.mcs.anl.gov/romio/.

[8] T. Asano, D. Ranjan, T. Roos, and E. Welzl. Space filling curves and their use in the design of geometric data structures. In *Lect. Notes Comput. Sc.*, volume 911, pages 6–44. 1995.

[9] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[10] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. Plfs: a checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[11] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[12] L. Chacón. A non-staggered, conservative, $\nabla \dot{s} B \rightarrow = 0$, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries. *Computer Physics Communications*, 163:143–171, Nov. 2004.

[13] C. S. Chang, S. Klasky, J. Cummings, R. Samtaney, A. Shoshani, L. Sugiyama, D. Keyes, S. Ku, G. Park, S. Parker, N. Podhorszki, H. Strauss, H. Abbasi, M. Adams, R. Barreto, G. Bateman, K. Bennett, Y. Chen, E. D. Azevedo, C. Docan, S. Ethier, E. Feibush, L. Greengard, T. Hahm, F. Hinton, C. Jin, A. Khan, A. Kritz, P. Krsti, T. Lao, W. Lee, Z. Lin, J. Lofstead, P. Mouallem, M. Nagappan, A. Pankin, M. Parashar, M. Pindzola, C. Reinhold, D. Schultz, K. Schwan, D. Silver, A. Sim, D. Stotler, M. Vouk, M. Wolf, H. Weitzner, P. Worley, Y. Xiao, E. Yoon, and D. Zorin. Toward a first-principles integrated simulation of tokamak edge plasmas - art. no. 012042. *Scidac 2008: Scientific Discovery through Advanced Computing*, 125:12042–12042, 2008.

[14] C.-M. Chen, R. Bhatia, and R. Sinha. Multidimensional declustering schemes using golden ratio and kronecker sequences. *Knowledge and Data Engineering, IEEE Transactions on*, 15(3):659 – 670, may-june 2003.

[15] C.-M. Chen and C. T. Cheng. From discrepancy to declustering: near-optimal multidimensional declustering strategies for range queries. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 29–38, New York, NY, USA, 2002. ACM.

[16] J. H. Chen et al. Terascale direct numerical simulations of turbulent combustion using S3D. *Comp. Sci. & Disc.*, 2(1):015001 (31pp), 2009.

[17] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *J. Netw. Comput. Appl.*, 23:187–200, 1999.

[18] H. Childs. Architectural challenges and solutions for petascale postprocessing. *J. Phys.*, 78(012012), 2007.

[19] C. K. Chui. An introduction to wavelets. 1992.

[20] Cluster File System, Inc. Lustre: A Scalable, High Performance File System. http://www.lustre.org/docs.html.

[21] J. Clyne. The multiresolution toolkit: Progressive access for regular gridded data. In *Proceedings of the Visualization, Imaging, and Image Processing*, 2003.

[22] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, and P. Maechling. Scalable earthquake simulation on petascale supercomputers. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–20, Washington, DC, USA, 2010. IEEE Computer Society.

[23] A. M. David Nagle, Denis Serenyi. The Panasas ActiveScale Storage Cluster – Delivering Scalable High Bandwidth Storage. In *Proceedings of Supercomputing '04*, November 2004.

[24] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 259–270, New York, NY, USA, 1998. ACM.

[25] P. M. Dickens and R. Thakur. Improving collective i/o performance using threads. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 38–45, Washington, DC, USA, 1999. IEEE Computer Society.

[26] C. H. Q. Ding and Y. He. Data organization and I/O in a parallel ocean circulation model. In *Proc. SC99*. Society Press, 1999.

[27] C. Docan, F. Zhang, M. Parashar, J. Cummings, N. Podhorszki, and S. Klasky. Experiments with memory-to-memory coupling for end-to-end fusion simulation workflows. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:293–301, 2010.

[28] C. Fan, A. Gupta, and J. Liu. Latin cubes and parallel array access. In *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, pages 128 –132, apr 1994.

[29] R. Freitas and W. Wilcke. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, 2008.

[30] G. Ghinita, P. Kalnis, and S. Skiadopoulos. Prive: anonymous location-based queries in distributed mobile systems. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 371–380, New York, NY, USA, 2007. ACM.

[31] J. Gray et al. The recovery manager of the system R database manager. *ACM Comput. Surv.*, 13(2):223–242, 1981.

[32] R. W. Grout, A. Gruber, C. Yoo, and J. Chen. Direct numerical simulation of flame stabilization downstream of a transverse fuel jet in cross-flow. *P. Combust. Inst.*, 2010. In press.

[33] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *IEEE Visualization*, 2002.

[34] D. Hilbert. Ueber die stetige abbildung einer line auf ein flächenstück. *Math. Ann.*, 38:459–460, 1891.

[35] Y. Hu, A. Cox, and W. Zwaenepoel. Improving fine-grained irregular shared-memory benchmarks by data reordering. In *Proc. SC00*, pages 33 – 33, Nov. 2000.

[36] I. Ihm and S. Park. Wavelet-based 3d compression scheme for very large volume data. In *In Proceedings of Graphics Interface '98*, pages 107–116, 1998.

[37] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD Rec.*, 19(2):332–342, 1990.

[38] H. L. Jenter and R. P. Signell. Netcdf: A public-domain-software solution to data-access problems for numerical modelers, 1992.

[39] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid -based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 24, Washington, DC, USA, 2003. IEEE Computer Society.

[40] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid -based parallel data streaming implemented for the gyrokinetic toroidal code. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 24–, Washington, DC, USA, 2003.

[41] D. Kotz. Disk-directed i/o for mimd multiprocessors. *ACM Trans. Comput. Syst.*, 15(1):41–74, 1997.

[42] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout. Pidx: Efficient parallel i/o for multi-resolution multi-dimensional scientific datasets. In *CLUSTER '11: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, Washington, DC, USA, 2011. IEEE Computer Society.

[43] S. Kuo, M. Winslett, Y. Cho, J. Lee, and Y. Chen. Efficient input and output for scientific simulations. In *In Proceedings of I/O in Parallel and Distributed Systems (IOPADS)*, pages 33–44. ACM Press, 1999.

[44] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Isabela-qa: query-driven analytics with isabela-compressed extreme-scale scientific data. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 31:1–31:11, New York, NY, USA, 2011. ACM.

[45] E. Laure, H. Stockinger, and K. Stockinger. Performance engineering in data grids. *Concurr. Comp-Pract. E.*, 17:4–171, 2005.

[46] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Rec.*, 30(1):19–24, 2001.

[47] J. Li et al. Parallel netCDF: A high-performance scientific I/O interface. In *Proc. SC03*. ACM, 2003.

[48] W.-k. Liao and A. Choudhary. Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[49] J. Lofstead et al. Managing variability in the io performance of petascale storage system. In *Proc. SC10*, New York, NY, USA, 2010. IEEE Press.

[50] J. Lofstead et al. Managing variability in the IO performance of petascale storage systems. In *Proc. SC10*, New York, NY, USA, 2010. ACM.

[51] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible i/o and integration for scientific codes through the adaptable i/o system (adios). In *6th International Workshop on challenges of Large Applications in Distributed Environments*, Boston, MA, 2008.

[52] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, and M. Wolf. Six degrees of scientific data: Reading patterns for extreme scale science io. In *Proceedings of the 20th International ACM Symposium on High-Performance Parallel and Distributed Computing, to appear*, HPDC'11. ACM, 2011.

[53] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. Six degrees of scientific data: reading patterns for extreme scale science io. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 49–60, New York, NY, USA, 2011. ACM.

[54] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

[55] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving mpi-io output performance with active buffering plus threads. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 10 pp., 2003.

[56] S. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(7):674 –693, jul 1989.

[57] The HDF Group. Hierarchical data format version 5, 2000–2010. http://www.hdfgroup.org/HDF5.

[58] O. E. B. Messer, S. W. Bruenn, J. M. Blondin, W. R. Hix, A. Mezzacappa, and C. J. Dirk. Petascale supernova simulation with chimera. *Journal of Physics: Conference Series*, 78, July, 2007.

[59] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE T. Knowl. Data En.*, 13(1):124–141, 2001.

[60] NCCS. http://www.nccs.gov/computing-resources/jaguar/.

[61] NERSC. http://www.nersc.gov.

[62] NetCDF. http://www.unidata.ucar.edu/software/netcdf/.

[63] R. Niedermeier and P. Sanders. On the manhattan-distance between points on space-filling mesh-indexings. Technical report, 1996.

[64] Oak Ridge National Laboratories. http://www.nccs.gov/user-support/center-projects/adios.

[65] R. Oldfield, P. Widener, A. Maccabe, L. Ward, and T. Kordenbrock. Efficient data-movement for lightweight i/o. *Cluster Computing, IEEE International Conference on*, 0:1–9, 2006.

[66] E. J. Otoo, D. Rotem, and S. Seshadri. Optimal chunking of large multidimensional arrays for data warehousing. In *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*, DOLAP '07, pages 25–32, New York, NY, USA, 2007. ACM.

[67] E. J. Otoo, A. Shoshani, and S. won Hwang. Clustering high dimensional massive scientific dataset. In *SSDBM*, pages 147–157, 2001.

[68] P. H. Carns and W. B. Ligon III and R. B. Ross and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[69] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with czip compression. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[70] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Proc. SC01*, pages 45–45, Nov. 2001.

[71] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *SC*, page 2, 2001.

[72] M. Polte et al. ...and eat it too: High read performance in write-optimized HPC I/O middleware file formats. In *In Proceedings of Petascale Data Storage Workshop 2009 at Supercomputing 2009*, 2009.

[73] A. S. Poulakidas, A. Srinivasan, Ö. Egecioglu, O. H. Ibarra, and T. Yang. Image compression for fast wavelet-based subregion retrieval. *Theor. Comput. Sci.*, 240(2):447–469, 2000.

[74] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, and A. El Abbadi. Efficient retrieval of multidimensional datasets through parallel i/o. In *High Performance Computing, 1998. HIPC '98. 5th International Conference On*, pages 375 –382, dec 1998.

[75] F. Pretorius. Evolution of Binary Black Hole Spacetimes. *Phys. Rev. Lett.*, 95:121101, 2005.

[76] J.-P. Prost, R. Treumann, R. Hedges, B. Jia, and A. Koniges. MPI-IO/GPFS, an Optimized Implementation of MPI-IO on Top of GPFS. In *Proceedings of Supercomputing '01*, November 2001.

[77] J. A. Rice. *Hilbert R-tree: An improved R-tree using fractals.* Duxbury Press, Ibrahim Kamel, Christos Faloutsos, 1995.

[78] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-Structured file system. *ACM T. Comput. Syst.*, 10:1–15, 1991.

[79] R. Ross. Parallel I/O Benchmarking Consortium. http://www-unix.mcs.anl.gov/rross/pio-benchmark/html/.

[80] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proc. 10th Int. Conf. on Data Eng.*, pages 328–336, Houston, TX, 1994.

[81] A. Sawires, N. E. Makky, and K. Ahmed. Multilevel chunking of multidimensional arrays. *Computer Systems and Applications, ACS/IEEE International Conference on*, 0:29–I, 2005.

[82] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger. On multidimensional data and modern disks. In *FAST*, 2005.

[83] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *FAST '02*, pages 231–244. USENIX, Jan. 2002.

[84] T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011.* ACM, 2011.

[85] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin. An implementation of a log-Structured file system for UNIX. In *USENIX'93*, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association.

[86] T. Shimada, T. Tsuji, and K. Higuchi. A storage scheme for multidimensional data alleviating dimension dependency. In *Digital Information Management, 2008. ICDIM 2008. Third International Conference on*, pages 662 –668, nov. 2008.

[87] A. Skodras, C. Christopoulos, and T. Ebrahimi. The jpeg 2000 still image compression standard. *Signal Processing Magazine, IEEE*, 18(5):36 –58, sep 2001.

[88] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid datafarm architecture for petascale data intensive computing. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 102, Washington, DC, USA, 2002. IEEE Computer Society.

[89] D. Taubman and M. Marcellin. Jpeg2000: standard for interactive imaging. *Proceedings of the IEEE*, 90(8):1336 – 1357, aug 2002.

[90] R. Thakur and A. Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, Winter 1996.

[91] R. Thakur, W. Gropp, and E. Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[92] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, N. P. R. Grout, Q. Liu, Y. Wang, and W. Yu. Edo: Improving read performance for scientific applications through elastic data organization. In *CLUSTER '11: Proceedings of the 2011 IEEE International Conference on Cluster Computing*, Washington, DC, USA, 2011. IEEE Computer Society.

[93] T. Tu et al. Accelerating parallel analysis of scientific simulation data via zazen. In *FAST'10*, pages 129–142. USENIX Association, 2010.

[94] Unidata. http://www.hdfgroup.org/projects/netcdf-4/.

[95] M. Vilayannur, P. Nath, and A. Sivasubramaniam. Providing tunable consistency for a parallel file store. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[96] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual log based file systems for a programmable disk. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 29–43, Berkeley, CA, USA, 1999. USENIX Association.

[97] W. X. Wang and et al. Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. *Physics of Plasmas*, 13(9):092505, 2006.

[98] P. Wong and R. F. Van der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division.

[99] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

[100] W. Yu and J. Vetter. ParColl: Partitioned Collective I/O on the Cray XT. In *International Conference on Parallel Processing (ICPP'08)*, Portland, OR, 2008.

[101] W. Yu, J. Vetter, R. Canon, and S. Jiang. Exploiting Lustre File Joining for Effective Collective I/O. In *7th Int'l Conference on Cluster Computing and Grid (CCGrid'07)*, Rio de Janeiro, Brazil, May 2007.

[102] W. Yu, J. Vetter, and H. Oral. Performance Characterization and Optimization of Parallel I/O on the Cray XT. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, Miami, FL, April 2008.

[103] W. Yu, J. Vetter, and H. Oral. Performance characterization and optimization of parallel I/O on the cray XT. *IPDPS*, pages 1–11, April 2008.

[104] F. Zheng et al. Predata - preparatory data analytics on peta-scale machines. In *IPDPS*, Atlanta, GA, April 2010.