Abstract

TANG, HOUJUN. Dynamic Data Prefetching and Layout Optimizations for High Performance Heterogeneous Data Access. (Under the direction of Dr. Nagiza F. Samatova.)

The advancement toward exascale computing is producing massive amount of data. The data is then accessed by analysis and visualization applications to explore and reveal knowledge in it. There is a growing demand for efficient data access for these data-intensive applications as the I/O performance often dominates the overall execution time. However, the existing I/O sub-systems in high performance computing (HPC) only provide general purpose optimizations, and cannot satisfy the dynamic and diverse data accesses from various applications. These accesses often exhibit certain regularities of I/O behavior, which can be characterized as data access patterns. Understanding and utilizing these patterns to optimize for efficient data accesses becomes a challenging task.

With the recognition of data access patterns, data prefetching and layout optimization can be performed to improve the data access efficiency. Data prefetching can bridge the performance gap between memory and data storage devices by making predictions and bringing data before its use. As a result, the CPU stall is masked and data access latency is hidden. As a complement to prefetching, data layout optimization is another promising approach to speedup data accesses. In leadership computing facilities, data is typically stored on the parallel file system with hard disk drives (HDDs). The data organization (layout) is crucial to the access performance. The read performance suffers if it involves a large number of non-contiguous data block accesses, due to the high latency of seek and read operations by the HDDs. Unfortunately, it is common that analytic and visualization applications only require a subset of the data file, with an access pattern that can be very different from how data is written.

We propose a framework that performs data prefetching and layout optimization with the goal to extract the maximum data access performance in modern HPC architectures. We first present an online analyzer that is capable of detecting various heterogeneous data access patterns at an application's runtime with low computational and memory overhead. Combining our pattern detection with prefetching, high prefetch accuracy is achieved and read performance is improved.

To further optimize data access, we present a dynamic I/O framework that recognizes the data based on access patterns, replicates the data of interest in multiple reorganized layouts that would benefit frequently used read patterns, and makes runtime decisions on selecting a favorable layout for current read request. This framework supports reading individual elements as well as chunks of a multi-dimensional array data.

Finally, as simulations adopt Adaptive Mesh Refinement (AMR) technique, the data records from a region of interest could be widely scattered on storage devices and accessing interesting regions results in significantly reduced I/O performance. We study the organization of block-structured

AMR data on storage devices in order to improve read performance of spatio-temporal data accesses. An in situ data layout optimization framework is developed, which is able to automatically selects from a set of candidate layouts based on a performance model, and reorganizes the data before writing to storage.

Dynamic Data Prefetching and Layout Optimizations for High Performance Heterogeneous Data Access

by
Houjun Tang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2016

APPROVED BY:

Dr. Rada Chirkova	Dr. Steffen Heber
Dr. Kemafor Anyanwu Ogan	Dr. Ranga Raju Vatsavai

Dr. Nagiza F. Samatova
Chair of Advisory Committee

Biography

Houjun Tang received his bachelor's degree in Computer Science and Technology from Shenzhen University, China in 2012. He began his Ph.D. study in the following August at North Carolina State University under the direction of Dr. Nagiza F. Samatova. During his doctoral study, he has spent three summers working at Argonne National Laboratory and Lawrence Berkeley National Laboratory.

Acknowledgements

First and foremost, I am grateful to my advisor, Dr. Nagiza Samatova. I definitely could not have reached this point without her consistent guidance and support in the past four years. I am very thankful to my committee members, Dr. Rada Chirkova, Dr. Steffen Heber, Dr. Kemafor Ogan, and Dr. Raju Vatsavai, for taking their valuable time to serve on my thesis committee, and for offering their insights in this dissertation. I would also like to thank Dr. Rui Mao, my undergraduate advisor at Shenzhen University. Without his guidense and encouragement, I could not have achieved my dream of studying in the United States.

This dissertation would not have been completed without the help from Dr. Samatova's research group. I am thankful to Eric Schendel and John Jenkins for letting me participate in their research projects in my first year. I am especially thankful to Xiaocheng Zou for his help both in my research and in my life. Additionally, I would like to thank Steve Harenberg and Stephen Ranshous for their help with my research paper writing.

I have had the pleasure of collaborating with researchers at national laboratories: Drs. Suren Byna, Kesheng (John) Wu, Bin Dong, Oliver Rubel, Kristofer Bouchard, Dan Martin, Hans Johansen, and Dharshi Devendran from Lawrence Berkeley National Laboratory, and Scott Klasky from Oak Ridge National Laboratory.

I am thankful to my parents for their unconditional love and support. My mother Hai Zhang, and father Hezhong Tang have been enormously supportive of me throughout my life.

The work was supported by the U.S. Department of Energy, Office of Science (SciDAC SDM Center) and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

Table of Contents

List of Tables

List of Figures

CHAPTER

1

INTRODUCTION

As computer systems and technology continue to evolve, the size of data being produced is growing increasingly larger. Table 1.1 [65] shows the data requirements of representative scientific applications run at Argonne Leadership Computing Facility through the INCITE program from the Department of Energy. The total size of the data produced by simulations and experiments has already surpassed the terabyte scale, now into petabyte.

However, data access speed have not kept up with the growth of computing power and data production rate. As a result, a huge gap emerged between data production and data access performance. Figure 1.1 shows the number of CPU cycles required to access register, cache, memory, and hard disk storage. A significant gap between memory and disk can be seen, which is primarily caused by the relatively slow disk access speeds.

Multivariate, spatio-temporal datasets at the tera- and peta-byte scale bring new challenges to the I/O subsystem for modern high performance computing (HPC). With the huge amount of data that needs to be read, the data access time can dominate the analysis or visualization application's execution time. Optimizing I/O performance becomes a crucial task for these I/O-bound applications.

Many different approaches have been proposed to address this issue. For example, in-situ data processing techniques [2, 44, 92] enable data analytics during the run time of a simulation

Table 1.1: Data requirements for representative INCITE applications

Project	On-Line Data	Off-Line Data
FLASH: Buoyancy-Driven Turbulent Nuclear Burning	75TB	300TB
Reactor Core Hydrodynamics	2TB	5TB
Computational Nuclear Structure	4TB	40TB
Computational Protein Structure	1TB	2TB
Performance Evaluation and Analysis	1TB	1TB
Climate Science	10TB	345TB
Parkinson's Disease	2TB	50TB
Plasma Microturbulence	2TB	10TB
Lattice QCD	1TB	44TB
Thermal Striping in Sodium Cooled Reactors	4TB	8TB
Gating Mechanisms of Membrane Proteins	10TB	10TB

or experiment, when the data is being generated, reducing data movement. Data compression is a complementary approach, reducing the size of the data being transfered, thus reducing I/O time, at the cost of having to compress and decompress the data [39, 46, 73]. However, though successfully reducing the required data movement, these approaches have their limitations: the time required to compress and decompress data at the PB scale may be substantial, surpassing the gains from reduced I/O; in-situ processing assumes complete prior knowledge of the analytical tasks to be performed, and requires most of the data to fit in memory. When scientists are performing exploratory analytics or visualization, based on multiple iterations of a diverse set of analytical tasks, having all the data stored permanently on file system is typically preferred.

A typical parallel I/O system has several layers: applications, high-level I/O library, I/O middleware, parallel file systems, and the underlying storage devices. In order to bring data from the file system into main memory efficiently, and narrow the gap between computation and I/O across the software stack, each layer has introduced different general purpose optimizations. At the highest level are I/O libraries such as ADIOS [52], NetCDF [64], Parallel netCDF [47], and HDF5 [81]. These high level libraries are built on I/O middleware such as MPI/MPI-IO [35, 80], which in turn are built on parallel file systems (PFS) such as GPFS [71], Lustre [8], and PVFS [66]. However, specialized optimizations are left to the domain scientists.

While domain scientists may have the best knowledge of how the data would be accessed, it requires a tremendous level of effort, or is simply impractical to manage and organize the data manually, given the complexity and uniqueness of the layers in the parallel I/O stack. Even though scientists know their applications thoroughly, we cannot expect that they are familiar with all the low

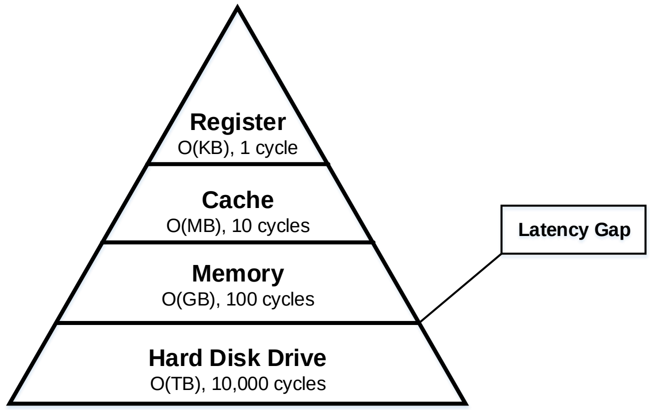


Figure 1.1: Data access/storage hierarchy

level details. Thus, it is desirable to have a framework that can perform optimizations to improve data access efficiency automatically, based on either prior knowledge or data access history.

In this thesis, we present our approaches that address this issue. To understand how data is accessed, we first explore data access patterns [51, 53], which play an important role and serve as the fundamental step in our optimizations. For example, the read performance of applications with qualitative and recognizable access patterns, such as 2d-strided read requests, can benefit greatly from various optimization techniques if the pattern can be identified.

Data prefetching and layout optimization are two effective approaches to improve data access efficiency. Prefetching hides the data access latency, masks the CPU stall, and can achieve high accuracy levels if the runtime access patterns can be identified online timely and correctly. While prefetching happens in the I/O middleware layer, data layout optimization focuses on the storage level, as shown in Figure 1.2. The goal for layout optimization is to store the data in such a way that frequently accessed data are organized closely are stored contiguously. In this way, the number of seek and read operations required to satisfy a read request is minimized.

1.1 Challenges

To summarize the problems mentioned in the above sections, we elaborate specifically on the challenges in improving data access efficiency.

In order to achieve high prefetching accuracy, it is necessary to acquire comprehensive knowledge of the application's access patterns. Various methods have been proposed [11, 19, 48, 60]; however, these tools are all offline-based, and are not capable of detecting complex access patterns (such as composition-based unstructured access patterns). Offline-based tools assume the availability of complete access history of one or more previous runs, which is unrealistic to obtain for scientific applications that run for hours or even days. In addition, offline-based algorithms cannot be directly applied to online analysis as they assume the presence of full access history, which may not fit in memory or be available; and only detect a pattern after its full occurrence, which provides no useful information for the current optimization.

Once a scientific dataset is produced, it's storage layout typically remains the same. The layout is determined by, and optimized for, the producer that is writing the data to storage devices. A variety of analytical applications may access the dataset, each exhibiting access patterns much different from the writer. While the initial layout is ideal for access patterns similar to that of the writer, it could lead to poor performance (2X to 100X slower) and dominate the total run time. To address these issues, the following challenges must be addressed:

1. **Recognizing data usage patterns and identifying an optimal physical layout.** The system needs efficient and accurate analysis of various data access patterns. With a number of partial replicas of different layouts, selecting the optimal layout from these replicas becomes crucial for performance improvement.

2. **Adapting to changes in read patterns.** Read patterns are often dynamic, changing over time as the user explores the data and focuses on different regions of interest.

3. **Evaluating costs and benefits in reorganizing data.** The system should evaluate the cost to create and store replicas with different layouts, together with the benefits of having this reorganized data, and use this information to help make the creation decision.

4. **Handling duplicate data replicas and budgeted storage space for replicas.** When the storage space of replicas reaches an upper bound, a decision has to be made about which of the previous replicas to drop (if any).

While previous work focus on data layout optimization for uniform grid data, the advent of adaptive mesh refinement (AMR) brings new challenges, rendering existing approaches unsuitable or open to improvement. The multi-level organization of AMR datasets, with multiple boxes (patches) of refinement levels, add a new dimension of complexity to data access patterns. Moreover, the size of the boxes varies, requiring the optimization technique to be adaptive, and boxes are usually small in size (less than 1MB), potentially resulting in more non-contiguous accesses.

1.2 Proposed Approaches

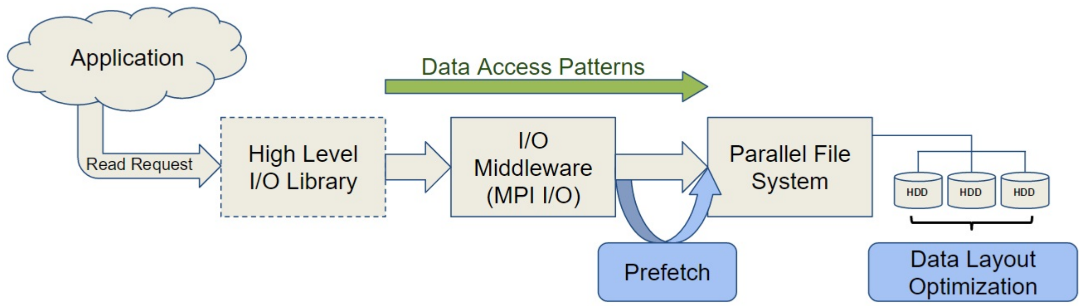


Figure 1.2: Hierarchical I/O stack and optimizations in HPC

In this thesis, we address the problem of improving data access efficiency by proposing two approaches:

- Data access pattern guided prefetching;
- Data layout optimization.

We first present an online access pattern analyzer (Chapter 2) that recognizes patterns during an application's runtime, and uses the patterns to guide prefetching. Based on the accumulated knowledge of frequent access patterns, we present a dynamic layout optimization framework (Chapter 3) for uniform mesh data. Our framework selects suitable data layouts for heterogeneous patterns, reorganizes the data accordingly, and is entirely transparent to the user. Furthermore, to support the emerging adaptive mesh refinement simulations, and the access of their produced data, we propose an AMR data specific layout optimization approach (Chapter 4). We briefly summarize the contributions of our work.

1.2.1 Access Pattern Guided Prefetching

We proposed an online access pattern analyzer that supports both structured and unstructured access patterns, while having high accuracy and low computational and memory overhead. The framework performs access pattern analysis during an application's runtime, and utilizes the pattern information to guide prefetching for better performance. We adopt a "pattern growth" approach, which enables online analysis: as new accesses arrive, they are compared to the current active pattern before being inserted to the trace buffer. The pattern library consists of detection and coalesce rules for detecting structured and unstructured access patterns. Each time a read request is made, the tracer extracts the read request's information while it is being passed to the prefetcher. The requested data are copied to user buffer if found in the prefetch cache, otherwise a normal file read is issued to the parallel file system. Figure 1.2 shows the modified parallel I/O stack that includes our prefetcher, shown as the shaded shape between the middleware and PFS layers.

We used PIO-Bench [75], a widely used synthetic parallel file system benchmark suite, to conduct experiments with various structured access patterns. For unstructured access patterns, we developed a micro-benchmark that simulates the file read behavior of an application that performs 3D visualization of climate datasets. With the pattern-aware prefetching, our method results in up to 26% run-time reductions on top of less than 5% overhead, with both kind of access patterns, in 22 benchmark evaluations.

1.2.2 Uniform Grid Data Layout Optimization

We proposed a framework that selects the most suitable layout among the common layout reorganization techniques based on detected data usage patterns. It is capable of performing storage-efficient optimizations for heterogeneous patterns, from both bounding box and element data selections. The framework is composed of a *Trace Analyzer, Layout Decision Maker, Pattern and Layout Knowledge Base*, and *Data Reorganization Manager*.

The Trace Analyzer uses a binary instrumentation method to trace I/O read calls and to identify data access patterns. We have developed in this work a cost model to predict the number of contiguous blocks accesses by a read access pattern. The Layout Decision Maker analyzes the cost of accessing data using the available layouts of the requested data, and selects the layout that would give the best performance. The Data Reorganization Manager uses suggestions of improved layouts to replicate and reorganize data. An advanced user can also initiate a request to the Data Reorganization Manager. When multiple replicas of the data with different layouts are available, the Layout Decision Maker dynamically redirects the read call to the selected replica for obtaining the best performance. The metadata related to the available layouts and data access pattern history are managed in the Layout and Pattern Knowledge Base.

To measure the performance of our proposed framework, we used I/O kernels extracted from four different scientific applications or datasets from various science domains, including:

- querying a 188 billion particle plasma physics dataset produced by a Vector Particle-In-Cell (VPIC) simulation of magnetic reconnection phenomenon, to demonstrate the support for element selection and partial match;

- accessing climate model and observation data, used for detecting Atmospheric Rivers (AR), to demonstrate the ability of applying historical optimization strategies on new datasets;
- accessing Electrocorticography (ECoG) data, to demonstrate the support for non-regular patters with an ability to perform optimizations for different data regions;
- accessing data from Mass Spectrometry images to show that our framework is able to support and manage different layout reorganization techniques at the same time.

Our framework yields up to 90X time speedup in the plasma physics queries that uses element selection, and up to 8X speedup for other experiments using bounding box selection.

1.2.3 AMR Data Layout Optimization

Adaptive mesh refinement (AMR) has gained its popularity in recent years, with the ability to dynamically adapt simulation resolution across space and time. It greatly improves the efficiency of computational resources, meeting the acceptable error levels for numerical convergence at the same time. However, the hierarchical and complex data structure of AMR differs greatly from uniform grid data, and brings new challenges to I/O optimization.

AMR libraries like Chombo do not provide functions to access AMR data with spatial selection. Instead, the entire dataset must read to access any region, which is extremely inefficient for analyzing a small region of interest. Even with support for reading data subsets, performance could still be poor, as boxes of data that are selected are scattered across the file, and may cause contention on the parallel file system. Moreover, other files that are accessed together with the AMR dataset, such as index files, could yield high data access latency.

Towards enabling efficient AMR data accesses, we present an I/O framework that addresses the issues of runtime data reorganization, supporting the complex structure of AMR data. We propose a new clustering-based approach to reorganize the AMR box storage layout, with consideration of the Lustre parallel file system's bulk I/O RPC size, which is shown to be more effective than space-filling curves. As the spatial location information of AMR boxes are contained within the dataset, no prior knowledge of the dataset is required with our optimization. Our framework automatically selects from a set of candidate layouts based on a performance model, and reorganizes the data before writing to storage.

We evaluate our framework with three AMR datasets and access patterns derived from scientific applications. Our performance model is able to identify the best layout scheme and yields up to a $3X$ read performance improvement compared to the original layout. Though it is not possible to turn all read accesses into contiguous reads, we are able to achieve 90% of contiguous read throughput with the optimized layouts on average.

1.3 Overview of the dissertation

This thesis is organized as follows. Chapter 1 introduced the challenges in the high performance heterogeneous data access and discussed the overview of the thesis work. Chapter 2 introduces the data access patterns of scientific applications, and demonstrates the benefits of access pattern guided prefetching. Chapter 3 shows the design and evaluation of a dynamic data layout optimization framework for uniform grid data. In Chapter 4, we propose a new layout reorganization technique to address the problem of layout optimization for AMR data. We conclude this dissertation with Chapter 5.

CHAPTER

2

ACCESS PATTERN GUIDED PREFETCHING

In this chapter, we first introduce an online analyzer capable of detecting both simple and complex access patterns with low computational and memory overhead. We then utilize the analyzer to guide data prefetching and show the performance improvement with different I/O benchmark configurations.

2.1 Introduction

Scientists who work with simulations such as S3D combustion [17] and GTS core plasma fusion [84] spend a significant amount of time analyzing the massive amount of data generated. With the increasing gap between CPU and I/O, the performance of scientific analysis and visualization applications are often I/O-bound [97], thus read performance becomes a key area for optimization. An essential component of this process is to better understand the application's I/O behavior or its access patterns.

An access pattern is a sequence of accesses that exhibits a certain regularity. Many common access patterns occur as a result of iterative computations [27]. For example, if a matrix is stored in row-major format, reading consecutive rows of the matrix results in a contiguous pattern, whereas reading one column induces a simple-strided pattern with the file pointer incremented by the same amount (row size) between each request. Scientific applications exhibit these patterns and others, including higher dimensional strided access patterns and composition-based or correlation-based unstructured access patterns.

Recognizing access patterns in an application is a key to potentially reducing future file read time. Scientific applications often read and analyze data alternately, thus by overlapping the two phases with prefetching can significantly reduce the overall execution time of the application. Accurate prefetching can be achieved with access pattern analysis.

In order to achieve high prefetching accuracy, it is necessary to acquire comprehensive knowledge of the application's access patterns. Various methods have been proposed [11, 19, 48, 60], however, these tools are all offline-based and not capable of detecting complex access patterns (such as composition-based unstructured access patterns). Offline-based tools assume access history of one or more previous runs beforehand, which is unrealistic to obtain for scientific applications nowadays that run for hours or even days. In addition, offline based algorithms cannot be directly applied to online analysis as 1) they assume the presence of full access history, which may not fit in the memory; and 2) they detect a pattern after its full occurrence, which provides no useful information for the current optimization strategy.

We propose a method for online analysis that requires no prior information of the application. To the best of our knowledge, our method is the first one capable of performing online analysis of various complex access patterns. Our framework collects an application's file read requests during its run-time and perform online analysis to guide prefetching. The contributions of this work are as follows:

1. **Online, low-overhead pattern analysis with high accuracy.** We adopt a "pattern growth" approach and efficient pattern detection algorithms to enable online analysis with overhead less than 5% in all test cases. The overall run-time reduction is up to 26% via pattern-aware prefetching with accuracy up to 99%;

2. **Support for various access patterns.** We develop an analyzer capable of detecting structured access patterns as well as composition-based and correlation-based unstructured access patterns;

3. **Low memory footprint.** To retain low memory footprint during run-time, I/O traces are merged with their corresponding access patterns in a compact format and keep a limited number of recent trace records in memory.

2.2 Background

Many I/O access patterns classification approach have been proposed [11, 23, 54]. Compared with them, we additionally support unstructured access pattern. Although the access pattern classification is similar, the algorithms to detect the patterns are different for offline and online analysis, as explained in Section 3.1.

2.2.1 Structured Access Pattern

Structured access patterns include contiguous, simple-strided, and kd-strided patterns. Fig. 2.1 illustrates the former two kinds. A contiguous pattern occurs when consecutive read requests are accessing a contiguous region of data in a file. It can be further divided into uniform and variable size patterns. For strided patterns, a stride is the difference between starting offsets of consecutive requests, and is fixed within each dimension. Simple-strided pattern is a special case of kd-strided when $k=1$. A kd-strided pattern can be viewed as a series of $k-1$d-strided segments with its k dimensional stride. For example, a 2d-strided pattern with the following offsets: {1, 3, 5, 11, 13, 15, 21, 23, 25}, is composed of three simple-strided segments {1,3,5}, {11,13,15}, and {21,23,25}, with the second dimensional stride of 10. Kd-strided pattern is often found when accessing a sub-volume or sub-plane of multi-dimensional data.

2.2.2 Unstructured Access Pattern

Unstructured access patterns are accesses that exhibit patterns with less regularity compared to structured ones. The number of accesses is linear to the number of parameters representing them, while exponential for structured ones. There are two particular instances that we found useful for scientific applications, which are referred as composition-based and correlation-based unstructured

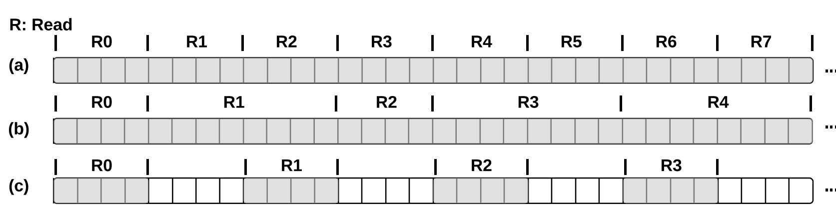


Figure 2.1: Each block represents 1 byte of data stored in row-major format, with shaded blocks being accessed. (a) Contiguous with uniform size: 8 requests ($R0$ to $R7$) each access 4 bytes. (b) Contiguous with variable size: 5 requests with sizes of $4,8,4,8,8$ bytes. (c) Simple-strided: 4 requests each access 4 bytes of data with 8 bytes between the starting offsets of consecutive requests

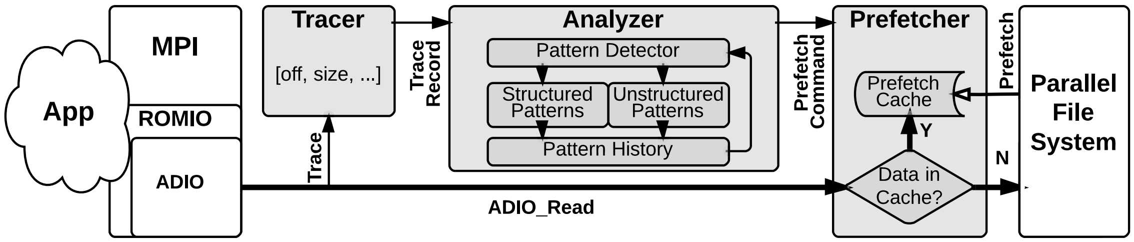


Figure 2.2: An overview of our framework: each time a read request is made, the tracer extracts the read request's information while it is being passed to the prefetcher. The requested data are copied to user buffer if found in the prefetch cache or a normal file read is issued to the parallel file system, the components added are in shaded shapes

access patterns. The composition-based patterns capture the repeating intervals between structured patterns or individual accesses, which is further explained in Section 3.3. Previous research in [48] exploited block correlations in storage systems. We include this kind of pattern and referred it as correlation-based unstructured access patterns. For example, from an offset sequence of {10, 20, 30, 40, 50, 10, 70, 20, 30, 80, 10, 40, 20, 30}, the correlation-based pattern is {10|20,30} and {20|30}, meaning that the data starting from offset 20,30 is frequently accessed after 10, while 30 is often accessed after 20. The threshold value of frequent accesses is 3, which is the number of times an offset occurs to be considered in a pattern. The request size is omitted for simplicity.

2.3 Method

Our online analyzer performs access pattern analysis of applications during their run-time and utilize the pattern information to guide prefetching for better performance. Fig. 2.2 illustrates the overview of our framework.

2.3.1 Tracer

To provide the analyzer the current data accesses, we need to collect the trace of actual I/O requests. High-level wrappers like PMPI are difficult to obtain important information such as collective optimizations. While low-level wrappers (POSIX-based) are file system specific and has no information regarding MPI. To preserve MPI-IO optimizations like data sieving and collective I/O, we implemented the tracer in the “middle-level" ADIO layer. By doing this, we gain the flexibility of examining the I/O request mapping induced by collective optimizations, while freeing up the namespace for other PMPI-based methods, such as Darshan [15, 16]. We modified the ADIO implementation part of MPICH [9] 3.0.4 source code.

2.3.2 Online Access Pattern Analysis

We adopt a rule-based model for access pattern detection in our online analyzer, which is the key component the framework. We maintained a “pattern library” that contains a collection of rules. These rules provide a concise description of the access sequences that are recognized as access patterns. The input is a sequence of accesses and the output is the detected access patterns and corresponding prefetching instructions.

Each time a read request is traced, the analyzer first performs a lookup in the pattern history to decide whether to activate a previously detected pattern and start prefetching or use it for analysis. The pattern analysis procedure includes the following steps:

1. Create a new pattern if current records in the trace buffer match any detection rules in the pattern library;
2. “Grow” the current pattern if the following accesses belong to it and inform the prefetcher to prefetch data that are predicted to be accessed next;
3. Commit the access pattern to the pattern history when the new access do not fit in;
4. Attempt to coalesce the current pattern with previous structured ones to form a higher level pattern;
5. Look back in the pattern history and check if there is any pattern that matches the current one.

More details of this procedure are explained in later examples of structured and unstructured pattern analysis.

Unlike offline analysis with a complete access history, online analysis must be incremental to detect a pattern during its occurrence. Thus we adopt the above “pattern growth” approach: as new accesses arrive, they are compared to the current active pattern before being inserted to the trace buffer. The pattern library consists of detection and coalesce rules for detecting structured and unstructured access patterns. The difference between them are the objects they operate on: detection rule operates on offset of accesses while coalesce rule operates on patterns. The analysis is performed periodically instead of upon every new request to reduce computation overhead. Three threshold values (T_{struct}, T_{corr}, and T_{comp}) are used to trigger the analysis of structured, correlation-based unstructured, and composition-based unstructured access patterns.

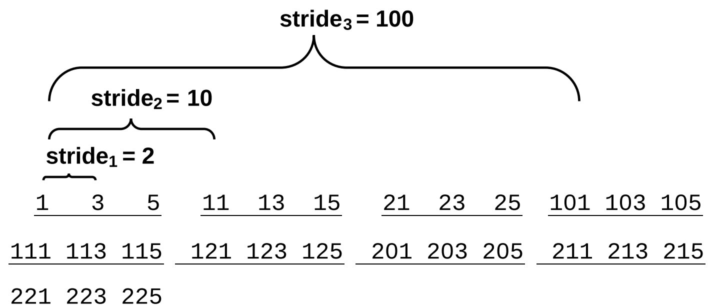


Figure 2.3: Offsets of a 3d-strided pattern

2.3.3 Structured Access Pattern Analysis

Different detection rules are used for contiguous and simple-strided access patterns. A contiguous pattern is determined by having at least 3 consecutive accesses with no gap in between. A simple-strided pattern comes with same offset differences (stride) between at least 3 consecutive accesses with identical request size. Kd-strided pattern is composed of $(k-1)$d-strided segments and is detected by the coalesce rule, which checks the stride and the number of accesses of two strided patterns with the same dimension. Note that each dimension of a kd-strided pattern must have at least three $(k-1)$d-strided segments.

Take a 2d-strided pattern with the following offsets {1, 3, 5, 11, 13, 15, 21, 23, 25, 31, 33, 35} as an example. The second dimensional stride can not be determined until 31 is accessed that signals the end of the third simple-strided segment. With three simple-strided segments detected and committed, they are coalesced to a 2d-strided one(step 1 to 4 of the pattern analysis procedure). An earlier detection is possible if a previous 2d-strided pattern with the same stride and number of accesses of first dimension exists in the pattern history: we temporarily mark the current simple-strided pattern of {1, 3, 5} as the 2d-strided one and start prefetching (step 5). Once a mismatch happens, it is restored to the previously detected pattern and continue the analysis procedure. Only the most recent pattern that qualifies is used in case multiple candidates exist, as same pattern tends to occur close in time.

To further illustrate how the analyzer identifies a higher dimensional kd-strided pattern, we adopt an example as shown in Fig. 2.3. until offset 101 is accessed, we only know the current pattern is a 2d-strided one. With this in mind, whenever a simple strided pattern emerge, we will look back of the previous detected patterns and check if a kd-strided pattern can be formed with the simple strided ones during the analysis.

The time complexity the above pattern detection process is $O(n \times T_{struct})$, and for the coalesce rule is $O(N_{spattern})$, where n is the number of total accesses, and $N_{spattern}$ is the number of detected structured access patterns. Though the time complexity depends on the whole trace and could be quite large, the frequency of the analysis is expected to be high and as a result for each analysis procedure the workload is relatively small.

2.3.4 Unstructured Access Pattern Analysis

Previous analyzers usually deal with access patterns build from individual accesses. However, when accessing time-series data generated by scientific simulations, a higher level of pattern often exists between the accesses of different time steps. For example, if a scientist wants to visualize a climate dataset with hourly recorded data at the times when the daily low/high temperature occurs (usually 5-6am and 2-3pm) for 30 days. The corresponding visualization application would read data of time step 5, 6, 14, 15, 29, 30, 38, 39, 53, 54, 62, 63, etc. and for each time step, structured access patterns could exist if a sub-volume decomposition is used for parallel processing. State-of-the-art analyzer like IOSIG [11] is only able to detect the structures ones within each time step, while not recognizing the higher level of composition-based unstructured pattern with time step intervals repeating 29 times of $\{1,8,1,14\}$.

The detection rule for composition-based pattern detection is to find offset delta (the difference between any two consecutive offsets) sequences that repeat at least twice. Two separate delta sequences are created from the offset of accesses and the starting offset of structured access patterns. To efficiently detect such patterns, we build suffix trees incrementally that has linear time and space complexity. The corresponding pattern can be easily obtained from its suffix tree after each time of analysis. Since the number of patterns is significantly smaller than that of accesses, this process is expected to be of low computation overhead.

For correlation-based access patterns, steps 2 and 4 are skipped because a correlation-based pattern stays the same once generated. In step 5, patterns are merged into one if a previous pattern with the same "entry" is found. Only accesses with request size larger than R_{size} are considered because the cost of analyzing those accesses outweighs the cost brought on by prefetching. In addition, we only focus on frequent accesses (occurs more than T_{freq} times) with their next N_{next} accesses. And the time complexity is $O(n \times N_{next})$ The frequent access is referred as the "entry" of a pattern. A candidate set of accesses that have the potential of becoming frequent, which have a frequency between $T_{freq} - \epsilon$ and T_{freq}, is maintained for incremental analysis. The analyzer then forms the pattern of each frequent access as the entry and a list of its following frequent accesses. Each time the entry is accessed, this pattern is activated and the following accesses are prefetched as much as possible.

The limitation of non-structured access patterns is that it is only effective when there is sufficient amount of accesses accumulated during an application's run time. To help user better understand the access pattern and improve the analysis performance, we provide the option of post-run analysis. Access patterns can be saved to files after all data loading process have completed. The output file can be used as the meta-data file and be loaded in later runs. This is useful when the same data files are used multiple times with same or similar access patterns.

Since the detected patterns are per-process based, we will form global patterns when possible. One use case of this is when each process of an application reads only part of the data and the total data accessed is a contiguous region. When running that application with different number of processes, the local pattern of each process will change but the global patterns remain the same, and we can infer the pattern of each process from the global one.

2.3.5 Trace Storage with Low Memory Footprint

Our framework requires limited additional memory usage during application's run-time. The tracer extracts useful information from read requests and passes them to the analyzer to determine whether to store them in the trace buffer. Trace records are compressed to a pattern representation if possible. The memory used for structured access patterns are significantly reduced due to its regularity. A 2d-strided pattern with 1024 accesses needs approximately 102KB in memory while only 134B with a pattern representation. The unstructured access patterns require more storage than structured but still use much less memory than keeping all its accesses.

In addition, since online analysis focuses on current access patterns, only recent trace records are kept in the trace buffer. The tracer is implemented in the ADIO layer of MPI-IO, on which MPI optimizations like data sieving can be captured and utilized, as well as allowing the usage of other PMPI-based methods, such as Darshan [16]. The trace record contains information of its MPI rank, file ID, offset, request size, starting time, and operation type. Patterns are stored in a compact format: the structured access pattern contains file ID, pattern type, start time, start offset, and request size.

2.3.6 Informed Prefetching

The prefetcher prefetches data informed by the analyzer and checks if data in the prefetch cache can be used for current request. Depending on the accecc pattern, the size of prefetched data varied, and we only consider relatively large data size ($> 1KB$) as smaller request sizes do not benefit from prefetching. It is also implemented in ADIO layer and prefetches data per MPI process using a prefetching thread. To avoid extra overhead caused by communication between processes, both

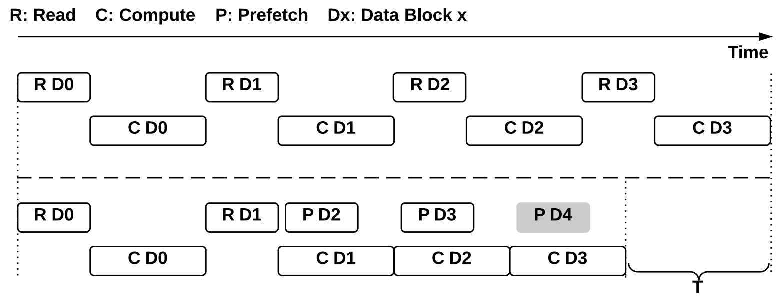


Figure 2.4: Prefetching is fully masked by computation

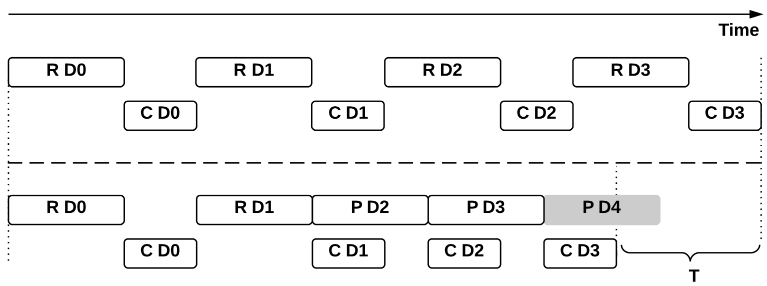


Figure 2.5: Prefetching is partially masked by computation

the analysis and prefetching are per-process based. We adopted a conservative prefetching strategy to minimize the cost of mis-prefetching: the prefetcher starts to prefetch data when a stable access pattern is detected and stops immediately when the previously prefetched data is not used, which indicates the detected access pattern is terminated.

2.4 Experimental Results

2.4.1 Experimental Setup

Our experiments were conducted on Argonne LCRC Fusion cluster. Each node is equipped with Intel Xeon 8-core (dual quad-core) 2.53 GHz processor, 36 GB memory, and 250GB local disk. The attached local disk to each node enables us to set up our own PVFS2 servers and create an isolated environment. We used 8 server nodes running PVFS2 2.8.2 file system with default strip size of 64KB. These nodes are connected with InfiniBand QDR and Gigabit Ethernet. Additionally, we implemented our framework based on MPICH 3.0.4.

Table 2.1: Ratio of computation time to read time for a given size of data

Size	min	min100	mean	sd	sort
128KB	0.027	0.061	0.183	0.353	1.388
1MB	0.028	0.031	0.221	0.428	1.899
16MB	0.034	0.027	0.244	0.473	2.586

Table 2.2: Prefetching accuracy of three structured access patterns

Pattern Type	Size	Read #	Accuracy
Contiguous / Simple-strided	128KB	1024	99.9%
	1MB	512	99.8%
	16MB	32	96.5%
2d-strided	128KB	1024	99.8%
	1MB	512	99.6%
	16MB	32	92.0%

2.4.2 Structured Access Pattern Performance

We used the PIO-Bench [75], a widely used synthetic parallel file system benchmark suite, and conducted experiments with contiguous, simple-strided, and 2d-strided access patterns to evaluate the performance with structured access pattern detection. The sequential (contiguous in our classification), simple-strided, nested strided (2d-strided), segmented (contiguous) and Tiled I/O (simple-strided with different parameters) access patterns fall in structured access patterns in our classification.

As mentioned in Section 1, the benefit of prefetching comes from overlapping I/O and computation. Fig. 2.4 and Fig. 2.5 illustrate four periodic read (R D0 to R D3) that are fully and partially masked by the computations via informed prefetching and the total time of T_1 and T_2 is reduced. To mimic real application's behavior, we insert computation time between each file read operation of PIO-Bench. To determine the computation time, we collected the time of running GNU Scientific Library functions such as find minimum number, first 100 smallest numbers, mean, standard deviation, and sorting. The ratio of computation time to read time for different size of data are shown in Table 2.1. We found the ratio of 0.5, 1.0, and 2.0 could represent different scenarios of real computation time and thus are used in our experiments. The results of simple-strided is similar to those of 2d-strided and we show the results using ratio of 0.5 and 1.0 that represent I/O intensive and compute intensive scenarios, contiguous and 2d-strided access pattern, and read request of 128KB and 1MB.

From the results shown in Fig. 2.6 we can see a reduction in the application's total running time in all cases with the percentage of up to 26% and an average of 17% for contiguous access pattern and 16% for 2d-strided. The performance gain of the informed prefetching with access pattern analysis are more pronounced when the computation to read time ratio is 1.0 because read time is fully masked by computation. For ratio with 2.0, the time reduction percentage is between that of 1.0 and 0.5, which is expected because the potential of run-time reduction is less when computation takes most of the time. Compared with offline methods such as that in [11], we achieve approximiately the same IO improvement.

Since the read size of each experiment is fixed, we could also see a trend of time increase with the number of processes increases, this is due to more requests to the parallel file system slows down its performance. We also observed that for the cases with 8 processes that takes longer time than 4 or 16, we believe this is caused by network congestion since all 8 processes are on the same node and send read requests concurrently.

2.4.3 Unstructured Access Pattern Performance

The random strided pattern of PIO-Bench is a composition-based unstructured access pattern, however, this pattern is too simple compared to real scientific applications. Thus we developed a micro-benchmark with both structured and unstructured access patterns. We found the results for correlation-based patterns are similar to those in [48].

The micro-benchmark simulates the file read behavior of an application mentioned in Section 2.3.3, which performs 3D visualization of climate datasets with hourly data at time steps when daily low/high temperature occurs. A sub-volume decomposition is used to perform parallel I/O for each time step. We experimented with two types of decompositions: row-wise and column-wise, as shown in Fig. 2.7. For each time step, the 3D data is broke into "slices" and each process reads one slice. The resulting access pattern contains both structured (simple/$2d$-strided within each time step) and composition-based unstructured pattern (repeating kd-strided with time step interval rotates from $\{1,8,1,14\}$). Similar to the previous experiments, we set the computation time to the average time of each file read. In addition to using plain row-major file layout, we also tested with files stored with block layout. Scientific applications like ScaLAPACK benefits from this kind of layout as they use blocks as the unit for communication and computation. The normal row-major file layout can also be viewed as the block layout with block size of 8B (the size of double).

The total data size of each time step read by all processes is 1GB and we vary the decomposition type, file layout type, and the number of processes. All processes are synchronized before the first read and the maximum elapsed time is reported. Fig. 2.8 compares the performance results by row and column decomposition with different file layout types. The row decomposition of different

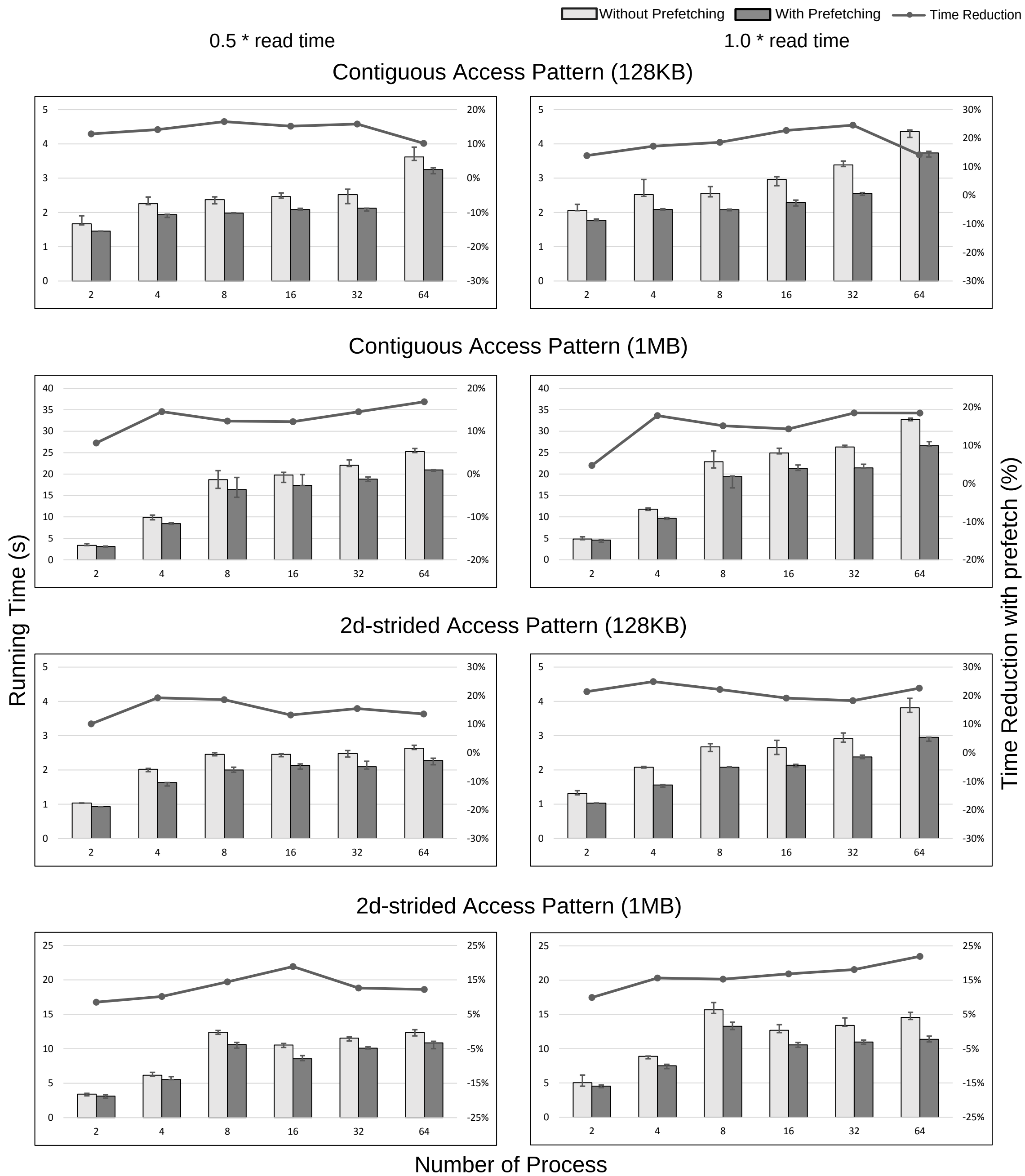


Figure 2.6: Performance of contiguous and 2d-strided access patterns

block sizes have similar results, and column decomposition with row-major layout takes much longer time since it has most dis-contiguous accesses. For all cases, we observe the time reduction

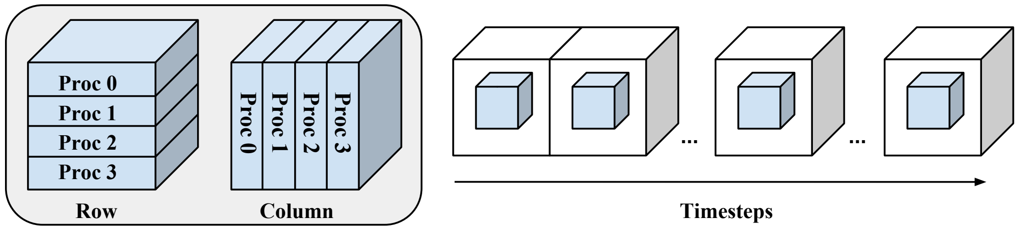


Figure 2.7: Two types of domain decomposition used in our evaluation

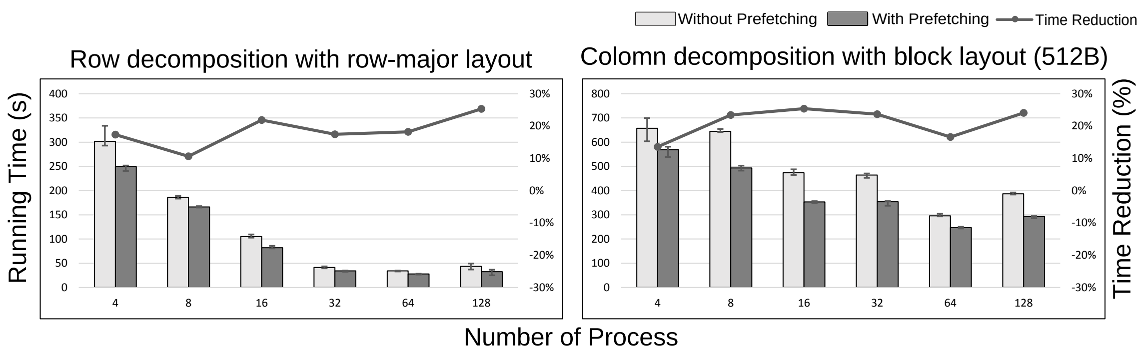


Figure 2.8: Performance of row/column domain decomposition with different block size

ranges from 13% to 26% with prefetching, which proves the effectiveness of the analyzer.

2.4.4 Overhead of Trace Collection and Access Pattern Analysis

The overhead of our trace collector and analyzer is defined as the time difference between the two runs with our framework and with original MPICH. To test the overhead of trace collection and analysis, we run with the previous configurations by setting the computation time to, the median of 10 different runs is used. We show results of two different cases in Fig. 4.9. Similar overhead is observed in other cases and all are less than 5%.

2.4.5 Accuracy of Access Pattern Detection

To evaluate the effectiveness of our pattern detection algorithm, we use prefetching accuracy as a metric. It is calculated by dividing the amount of subsequently used and prefetched data by the total used data. The high accuracy means the prediction of analyzer is accurate. Table 2.2 shows the prefetching accuracy of three structured access patterns. The high percentage is expected as these patterns are highly structured and remain stable for a period of time.

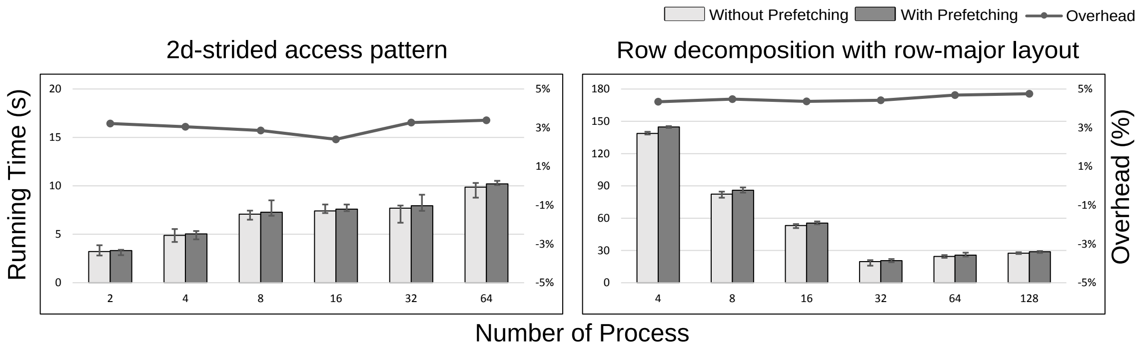


Figure 2.9: Overhead of trace collector and analyzer with 2d-strided and unstructured access pattern

2.5 Related Work

Various methods have been proposed to utilize access patterns for I/O optimization. In HPC community, tracing and profiling tools that are used for analysis and visulization purposes such as TAU [74], Jumpshot [94], Periscope [31], Upshot [37], and KOJAK [56]. Gong et. al [32] proposed a parallel run-time layout optimization framework to speed up queries on large complex scientific datasets. In database community, utilizing access patterns to guide prefetching proves to be effective [36]. Unlike their methods that deal with file layout organization and database objects, respectively, our work only involves MPI-IO and is on byte level. These tools mainly focus on profiling and analyzing an individual application's parallel processing performance, MPI messaging between computing nodes and processes/threads, and I/O behaviors in main memory layer instead of file storage layer.

To understand the I/O behavior of the data access from various applications, Reed et al. have categorized several common data access patterns of parallel applications [53–55].

Existing access pattern analyzers [11, 19, 48, 60] perform analysis in the offline-based fashion. In [11], a notation called I/O signature that represents access patterns is proposed. However, their pattern analysis only focus on structured ones. Oly et al. used a Markov model [60] built from access history to predict future accesses and prefetch data. C-Miner [48] uses a frequent sequence mining algorithm named CloSpan to discover block correlations, and utilizes the detected information for prefetching and reorganizing data layout. Choi et. al applied probabilistic latent semantic analysis with deterministic annealing [19] to discover file or variable access patterns. These methods require prior knowledge of the application and can not be directly applied to online analysis. We enabled our trace collection and analysis to be online, which is more desirable for scientific applications nowadays. Our analyzer can also be used in offline manner that generates same access patterns as offline-based ones.

Another method is streaming frequent itemset mining [14]. However, the frequent itemset mining algorithm are used with large-scale transaction data while our data is a single long sequence of accesses. Breaking the sequence into multipleshort sequences and viewing them as transactions will cause information to be lost, as occurrences could be split into two sequences.

Data prefetching is an effective latency-hiding solution for improving efficiency of parallel I/O and has been extensively studied and widely used [3, 5, 24, 25, 28, 30, 49, 59, 89]. However, the traditional prefetching strategies such as file-system level approaches are conservative. Even with advanced parallel file systems such as PVFS [66], Lustre [8] , and GPFS [71], high bandwidth is not achieved when only simple patterns such as contiguous or simple strided are detected. They cannot provide satisfactory performance for the modern scientific simulations with a large number of complex access patterns. Patterson et al. proposed informed prefetching [62], but this requires developers to add I/O hints to the program. Unlike their method, our framework requires no code modification.

While the above research work focus on access patterns of data within a file, to detect access patterns of files have also been investigated. One approach is to use graphical models to represent unstructured access pattern. The main idea is to record number of times that one item a is accessed within a time window after another b as the weight of edge connecting vertex a and b. Kuenning et al. used probability [42, 43] to capture file correlations in file systems. Bhadkamkar et al. proposed BORG [6], a self-optimizing storage system that performs automatic block reorganization based on the directed process access graph build with traces obtained from profiler at the block layer. Our current work is only applied on patterns within a file, but it can be extended to detect file access patterns as well. Besides, one major problem when using graph is that it is very likely to consume too much memory storing the graph. In our method, patterns stored in a much more compact way that requires much less space.

Darshan [15, 16] focuses on center-wide usage patterns by combining local, subsystem metrics and application-level metrics. To ensure scalability and small traces containing salient information, Darshan only captures statistical usage and cumulative timing information, which limit its usage to general picture of I/O behavior. HPCT-IO [72], IOT [67], and ScalaIOTrace [57, 88] are application level tracing tools, these tools provide detailed and specific I/O operations, but traces can not be directly obtained during the application's run time. Thus we implemented our own tracer similar to those.

2.6 Summary

In this chapter, we presented an online access pattern analyzer that supports both structured and unstructured access patterns with high accuracy and low computation and memory overhead. With the pattern-aware prefetching, our method results in up to 26% run-time reductions on top of less than 5% overhead with both kind of access patterns in 22 benchmark evaluations.

CHAPTER

3

UNIFORM GRID DATA LAYOUT OPTIMIZATION

In previous chapter we proposed an online analyzer that is capable of detecting both simple and complex access patterns with low computational and memory overhead, by combining our pattern detection with prefetching, we have achieved high prefetch accuracy and accelerate runtime I/O. However, through our experiments we found that another aspect that has a significant impact on the I/O is the data storage layout. In this chapter, we focus on dynamic layout reorganization to improve the locality of uniform grid data accesses and achieve multi-fold I/O performance improvement.

3.1 Introduction

Large-scale scientific simulations and experiments produce massive volumes of data. This data is typically stored on a parallel file system in an organization that is optimal for writing and remains fixed afterwards. However, scientific data is often written once and read many times and the organization (layout) of the written data may not be efficient for the read patterns used in data analysis operations. For example, scientific simulations such as S3D combustion [17] and GTS core plasma fusion [84] write the data of *all* variables by time steps, yet analysis processes often read a *subset* of variables within a specific region over a number of time steps. Such mismatches between a write layout and a read pattern lead to poor read performance due to a large number of seek and read operations to hard disk-based file systems. This issue is exacerbated by the advancement towards exascale computing, leading to ever-increasing dataset sizes and thus presenting challenges with data management and I/O optimization for efficient data accesses.

To address this data layout mismatch issue, many layout reorganization methods have been proposed to increase the number of contiguous I/O accesses. For instance, *space-filling curves*, such as Hilbert-curve and Z-curve, are used to reorganize the original data [82]; array *transposition* is applied to create multiple full replicas [50] of data; and *merging* of multiple non-contiguous data blocks to a single contiguous chunk to create partial replicas [40, 90]. Each of these techniques has its own advantages and disadvantages. Space-filling curves bring performance benefits to sub-region accesses by reorganizing the dataset and they require no additional storage when only the original data is reorganized. Array transposition leads to better performance for accesses that have significantly larger sizes along one dimension. However, transpositions may require multiple replicas of the data. Specialized merging with partial replication results in better performance, as the previously non-contiguous data accesses become contiguous.

Despite various advantages of reorganization, none of the strategies alone can provide near-optimal read performance for hetergeneous patterns of analytic applications. To support multiple read patterns, there is a need for managing different layout strategies. These organizations shall facilitate commonly used spatial selections defined by multi-dimensional bounding boxes as well as by random element (point) selections. As storage space for managing multiple full replicas is expensive, support for managing partial replicas considering the storage budget is necessary. Transparent redirection of accesses to the available replicas that may match fully or partially are required as well. To the best of our knowledge, a framework supporting these requirements is absent in scientific data management. These optimizations are common place in the relational database management [1]. However, scientific data is often stored on parallel file systems using file formats such as HDF5 and NetCDF that do not provide such data reorganizations.

In this paper, we present the dynamic data reorganization framework that performs dynamic data access pattern tracing and identification functions, efficient storage of partial replicas to support multiple read patterns, and redirection of read accesses to a favorable layout at runtime. Our data reorganization framework shows a broader applicability compared to existing methods, enabled by the following main contributions of this work.

1. **Dynamic pattern identification.** Our framework automatically traces read accesses and identifies the data usage patterns during an applications' runtime. The current implementation

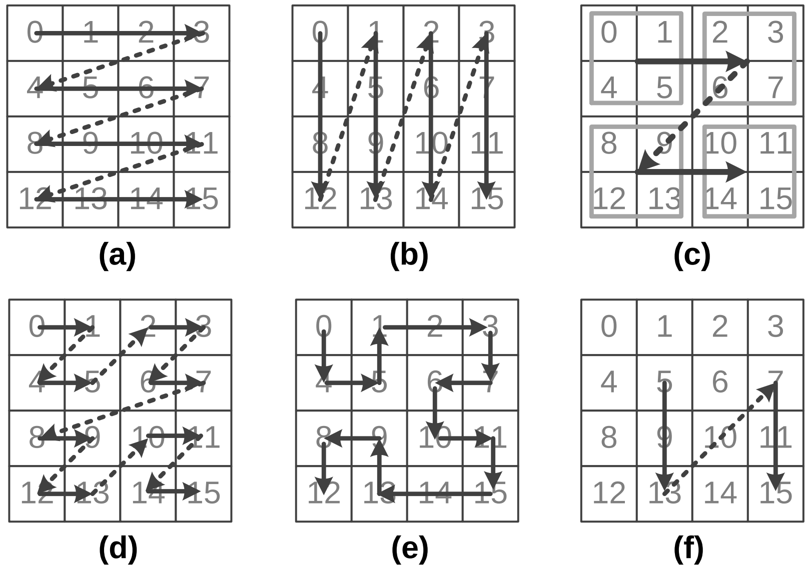


Figure 3.1: Data reorganization techniques that our framework supports. The numbers in each cell are the starting offsets of original data, and the arrow lines are the order of the reorganized offsets: (a) (original) row-major layout, (b) column-major (transposition) layout, (c) blocked (chunking) layout, used as a pre-processing step before applying (d) and (e), (d) z-curve, (e) Hilbert-curve, (f) custom merging of a subset (data at offsets 5,9,13,7,11,15)

supports the HDF5 [81] library in tracing bounding box selections, known as *hyperslabs* in HDF5, and element selections.

2. **Flexible multi-layout management with storage budgets.** Instead of using only one layout reorganization technique, we provide more flexibility by supporting multiple layout reorganization techniques among those shown in Figure 3.1. Based on the user-specified storage constraint and current usage patterns, our framework evaluates the costs of reorganization and the benefits with accessing the reorganized data to select the most suitable technique.

3. **Runtime decision making with partial match and redirection.** By allowing partial matches between read patterns and the reorganized replicas, we extend the usability of existing layouts compared to the exact match strategy from previous work [40]. For a more accurate cost estimation, a page-level (instead of byte-level) cost model is used for estimating data access cost during the decision making process. Further, we enabled automatic read redirection to the model-selected layout for improved performance.

The remainder of this paper is organized as follows: We briefly discuss the related work in Section 4.4. In Section 3.2, we present a high-level overview of the proposed data reorganization framework and describe the functionality of different components. We demonstrate the application of the framework in Section 4.3 using read patterns from multiple real applications and conclude our discussion in Section 3.5. various automated scientific data management optimizations [29, 86]. In this paper, we present the dynamic data reorganization service framework that performs dynamic data access pattern tracing and identification functions, efficient storage of partial replicas to support multiple read patterns, and to direct read accesses to optimal layouts at runtime.

3.2 Method

We present an overview of the proposed dynamic data reorganization framework in Figure 3.2. The main components of the framework are *Tracer*, *Trace Analyzer*, *Layout Decision Maker*, *Pattern and Layout Knowledge Base*, and *Data Reorganization Manager*.

The tracer traces the accessed files, variables, and the offsets (data locations) of the application's reads and pass them to access pattern analyzers to identify the read patterns. The analyzed results are stored as metadata and managed by SDS Metadata Manager, which implemented using Berkeley DB. With the identification of the data usage patterns, the layout reorganizer will create replicas with optimized layout for the patterns. These replicas will be used for future accesses that have same or similar patterns.

The Trace Analyzer uses a binary instrumentation method to trace I/O read calls and to identify data access patterns. Our current implementation supports the HDF5 library to trace hyperslab (a subset of a multi-dimensional array) definitions that access bounding box and element selections. We have developed in this work a cost model (Section 3.2.2) to predict the number of disk drive page accesses by a read access pattern.

The Layout Decision Maker analyzes the cost of accessing data using the available layouts of the requested data and selects a layout that would give the best access performance. The supported layout reorganization techniques, as shown in Figure 3.1, are designed as plugins so new layouts can be easily added for more specialized optimization.

The Data Reorganization Manager uses the suggestions of improved layouts to reorganize and replicate data with optimized layouts. An advanced user can initiate a request to the Data Reorganization Manager to reorganize data. When multiple replicas of the data in different layouts are available, the Layout Decision Maker dynamically redirects the read calls to the selected replica for obtaining the best performance. The metadata related to the available layouts and data access pattern history are managed in the Layout and Pattern Knowledge Base. We discuss each of these

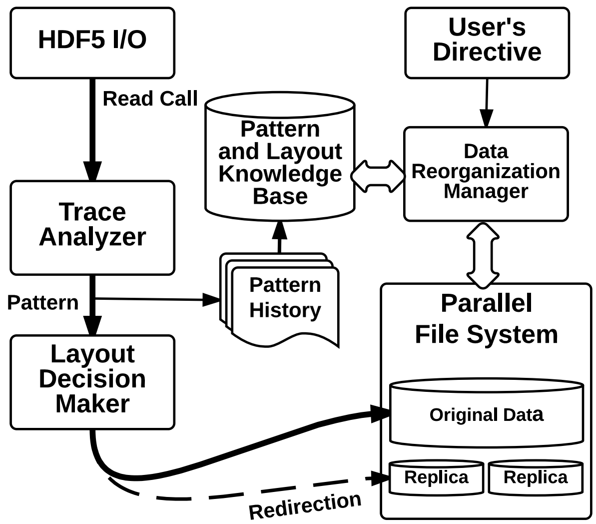


Figure 3.2: At runtime, the framework traces and detects patterns of I/O and decides on optimal data layouts. The layout management, i.e., replica creation according to the optimal layouts is performed offline. When optimal layouts are available, redirection of file read calls to the replicated data is performed at runtime using binary instrumentation. The right side of the figure shows the components of the dynamic reorganization framework

components in detail in the following subsections.

3.2.1 Trace Analysis and Pattern Detection

The first step to understand the data usage of applications is tracing the I/O read calls and identifying patterns. Motivated by existing work [40, 77, 90], we characterize data usage patterns in accessing a particular dataset, by focusing on three major aspects: (1) variables within a dataset being accessed, (2) accessed region (one or more sub-planes or whole plane, one or more sub-volume or whole volume, scattered points) of variables, and (3) the size of the requests.

The runtime pattern detection operation is performed first by extracting the relative information from HDF5 read calls issued by the running application. Similar to our previous work [77], this operation is performed within each MPI process and we keep the related information in an auxiliary data structure. We then analyze the data selection information to identify patterns. For HDF5 and other I/O libraries such as NetCDF [64], PNetCDF [47] and ADIOS [52], element (point) and bounding box selections are the two typical types of data selection provided to users that result in different patterns. We use a compact representation for the identified patterns, as shown in 3.2.3.

To identify different patterns induced by element and hyperslab selection, our framework first checks the selection type and then records the data selection information during runtime. This information is then used for selecting high performant layout (Section 3.2.2) and creating new replica in offline layout management (Section 3.2.4).

Bounding Box Selection

Many analysis applications read data from a variable that is bounded by spatial locations defined by multi-dimensional array coordinates. As categorized by Lofstead et al. [51], in a 2D array, this bounding box region is referred as sub-plane or a whole plane and in a 3D dataset, the region is called sub-cube. In HDF5, the bounding box selections are called *hyperslabs*.

A HDF5 hyperslab selection can be regarded as a complex bounding box selection. It allows users to select multiple bounding boxes with arbitrary regions using set operations (e.g. intersection, union, etc.). Such flexibility simplifies users effort to read their interested data regions in one read function call. Dealing with complex definitions of hyperslab challenges the existing work (such as [40, 82, 90]), which deals with one bounding box selection at a time. One such example is when accessing a labeled dataset, where the data is partitioned into chunks and each chunk has a different label. The data of one label is scattered in a file and is determined by an auxiliary index (See Section 3.3.5 for more details).

Element Selection

Element selection is commonly used when a query library is involved, where the coordinates of typically scattered elements are given and the corresponding data need to be read from file. The coordinates can often be obtained fast with indexing techniques such as FastBit [85] and ISABELA-QA [45]. However, reading the data often results in extremely low I/O throughput due to the large number of non-contiguous reads with small request sizes. The capability to optimize for such patterns would bring huge read performance improvements and thus motivates us to explore the methods for such optimization.

To optimize data reads, we only assume the coordinates of the data selected as input, specifically, we do not require that the high level criteria on which the selection was based (e.g., range query) is known. As a result, our optimization is generic and can benefit the existing indexing techniques directly. For example, in the analysis and visualization of the VPIC dataset [12], only the particles with high energy are of interest. And thus a small subset of elements would be repeatedly accessed in a sequence of queries with value constraints such as $Energy > 1.3$, $Energy > 1.5$, $Energy > 1.8$, etc. By clustering those scattered elements with an intrinsic correlation into a contiguous chunk

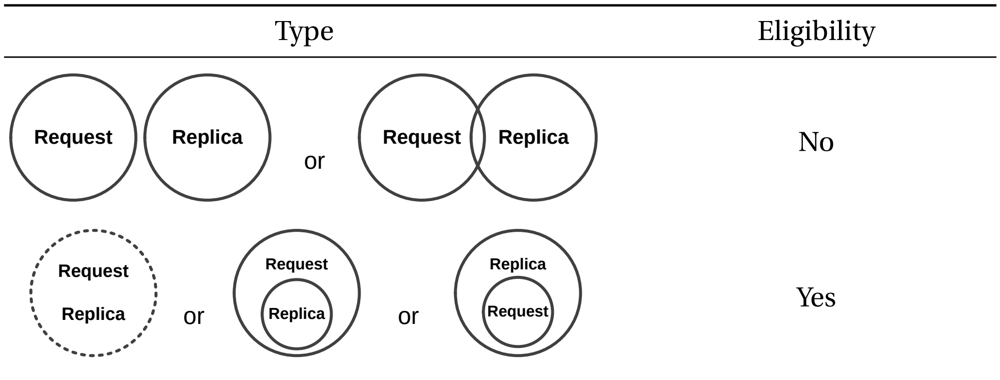


Figure 3.3: The eligibility for a replica to be a candidate is determined by how much overlap it has with the read request. A replica is not eligible with no or partial overlap region while eligible in other cases

of data in file, and storing their original offsets, a large amount of time can be saved when future accesses include these elements. More details about the "clustering" part will be elaborated in Section 3.2.4.

3.2.2 Layout Decision Making

The layout decision maker uses the pattern information recorded in the detection process and attempts to find the best matching replicas. The layout decision making process consists of two main steps:

Step 1: Candidate Selection

The layouts that cannot satisfy the request are first pruned to avoid the potentially large overhead of iterating through all layouts and loading their metadata. To be I/O efficient, a storage-lightweight catalog containing the start and end offsets for each existing layouts is maintained and used for the first round of coarse-grained pruning. The coarse-grained pruning prevents loading all metadata files. Another round of fine-grained pruning is performed, which loads the rest of metadata (the exact regions of data that a replica contains) of the remaining layouts and compared them with the requested data regions. A candidate set of potential replicas is generated using the following rules shown in Figure 3.3.

Note that we consider replicas that partially overlap with the requested data as not eligible. This is because the overlapping regions cannot be estimated accurately without loading the metadata of a replica. Accessing the metadata, especially for element selections (mappings to the original dataset), results in a non-negligible I/O cost as the size of metadata grows linearly with the data.

Though it is best when the data of a replica is exactly the same as a request, we found performance improvements using the replica in two cases: 1) when the request region is larger than the replica, splitting the read request to read the entire replica and the rest is still beneficial especially when the overlapping region is relatively large (see § 3.3.2); and 2) when the replica contains more data than the request, using the replica results in more contiguous accesses than using the original dataset and is expected to provide better read performance before a specific replica for that pattern is created.

Step 2: Layout Ranking Via Cost Model

The candidate replicas are ranked and the final decision is made via our page-level cost model. Inspired by the previous research work from [40, 90], we adopt a similar model with adjustments that better estimate the costs. As opposed to byte-level cost model, we chose to use a page-level cost model as it more accurately reflects the file read cost, especially in cases of element selection. The estimated read time T_r for replicas with different layouts is defined as follows (parameters defined in Table 3.1):

$$T_r = \max\{(N^i_{pg} * T_{pg} + N^i_{chk} * T_{chk}) N^i_{sp} \mid \forall i \in \mathcal{O}\} + T_{lat} \tag{3.1}$$

where $\mathcal{O}$ is the set of Object Storage Targets (OSTs). Based on the request and the layout, we "flatten" the requested region into linear space and calculate N^i_{pg} and N^i_{chk} within the data stored on each OST. This cost model estimates the total time needed when reading data across multiple OSTs, and assumes each OST offers the same I/O rate as well the network and storage latency. T_{pg}, T_{chk}, and T_{lat} are measured at the start of each application runs to reflect the dynamic I/O performance in a shared system. We compare the cost for all eligible replicas and the one with lowest T_r is selected.

I/O Redirection in HDF5

Once the layout decision is made, our framework automatically directs the read to the chosen replica. The replica's metadata such as the file and variable's path and name and the mapping of the layout to the original dataset is stored as part of the metadata, and becomes effective for the actual read. We have modified the related HDF5 read functions with our data selection procedure. When no replica is available, the normal HDF5 read process is used. If any replica is selected by our framework, the normal HDF5 read is skipped and the corresponding data in the replica is read instead.

Table 3.1: Parameters in the cost analysis model

Symbol	Meaning
N_{sp}^{i}	Number of I/O client processes accessing OST i
N_{pg}^{i}	Number of contiguous 4KB blocks need to be accessed within OST i
N_{chk}^{i}	Number of non-contiguous chunks need to be accessed within OST i
T_{pg}	Average cost of reading contiguous blocks of data per 4KB page
T_{chk}	Cost of reading non-contiguous chunks on one storage node
T_{lat}	Network and storage latency

3.2.3 Pattern and Layout Knowledge Base

To prepare the information needed for layout reorganization, our framework performs historic usage pattern analytics each time new patterns are discovered during runtime. It is an offline incremental analysis process that extracts and analyzes data usage patterns. It is based on the previous results from runtime pattern analysis (Section 3.2.1) with two more aspects added: (1) when and how many processes are issuing read requests together, (2) total size and I/O throughput. These are obtained after a read call completes. Based on the above aspects, our framework adopts a data usage patterns representation as {variable name/path, selection type and spatial region, process IDs, start/end time, total size, I/O rate}. Each read request results in one such record and is inserted into the pattern history. The most important aspect of the above is the accessed region.

A pre-processing step is performed to generate global patterns by merging the local patterns of each MPI process. The global pattern provides necessary information for the later data placement among OSTs (Section 3.2.4). Analysis of these patterns produces new information such as the "hot" data regions and pattern frequency for a dataset. This information, together with the metadata from existing layouts (replicas), is maintained in the "Knowledge Base". The layout metadata includes the replica's original file and reorganization information. Our framework automatically loads information from the knowledge base when the application starts. For offline layout management, the knowledge base supplies information to layout manager for layout creation and deletion.

3.2.4 Layout Reorganization

Layout management includes three tasks crucial to read performance: *replica creation* (when and how to create a new layout), *replica eviction* (which to remove when exceeding a user's storage budget), and *replica placement* (how the data is distributed among OSTs). This management occurs in an offline fashion, when the application terminates, to avoid runtime I/O contention. We assume that the data resides in parallel file systems such as Lustre, and the replicas with their metadata are stored by creating a special directory in the same folder with the original dataset.

Replica Creation

The layout manager makes the decision of when and how to create a new layout given the information from the knowledge base. This knowledge base can initialized with two options:

1. our framework can "learn" and decide what and how to perform layout optimizations, which takes effect after a few runs;

2. users can instrument our framework with the patterns from their knowledge, allowing performance improvements at the first use.

Three common replica creation scenarios are considered with the corresponding strategy that our framework automatically selects:

1. **The original dataset can or cannot be reorganized with the limited additional storage allowed:** "Concatenation" is used to create partial replicas, with partial match and replica deletion enabled. That is, a new layout will not be created when the existing replicas cover a non-trivial portion of the request (e.g., Section 3.3.2).

2. **The original dataset cannot be modified, but unlimited storage space is allowed:** "Concatenation" is used to create replicas as many as possible (e.g., Section 3.3.5).

3. **The original dataset can be reorganized but no additional storage is allowed:** Transposition or space-filling curves are used (e.g., D2 scenario 1 of Section 3.3.3).

The use of transposition and space-filling curves is thoroughly discussed in existing research [50, 82]. We provide more information on concatenation, which was explored in [40, 90], but only for single bounding box selection with high regularity of spatial patterns (kd-strided). To complement their work, our method adds support for both hyperslab selection with non-regular spatial patterns as well as element selection. We describe concatenation as follows: with a data selection that contains multiple chunks of non-contiguous data, the data between the chunks are removed and all the chunks are concatenated into one big contiguous chunk. By storing this big chunk as a partial replica, when the same (or overlapping subset) data selection occurs, the big chunk can be read all at once and thus brings read performance improvements.

Replica Eviction

Storage efficiency is achieved through the analysis of overlapping regions between new patterns and existing ones, and the deletion of old, less frequently used replicas. When the additional storage reaches the user-defined budget, the replicas are ranked according to a combination of their recent usage, size, and effectiveness (performance improvement $time_{old}/time_{new}$); older and less effective replicas are dropped to make space for new ones. As with maintaining layouts, each replica is associated with one "metadata" file containing the mapping to the original file. A separate "range" file is maintained for each dataset and stores the start and end offset of each replica for fast pruning as discussed in Section 3.2.2.

OST-Aware Replica Placement

Even when the right layout organization technique is selected, read performance can still be far from ideal when treating the parallel file system (PFS) as a black box. Popular PFSs, such as Lustre and PVFS, use striping for data distribution among multiple storage devices. In Lustre file system, Object Storage Targets (OSTs) are for storing data. The data distribution on Lustre is decided by the stripe count (how many OSTs to use) and the stripe size (size of data to write to one OST before moving to the next). Even with manual adjustment, such simple striping strategy can not provide efficient read accesses. Processes reading contiguous data with size comparable to $stripe\_count * stripe\_size$ would lead to the scenario where one OST is accessed by many processes, yielding contention (see Figure 3.4) that significantly degrades the overall read performance.

To avoid OST contention, the best practice is to make each OST be contacted by as few processes as possible–a guiding principle in our framework. Our framework analyzes the usage patterns (Section 3.2.3) and rearranges the data so that future accesses that are the same, or similar, will have low read contention, as shown in Figure 3.5.

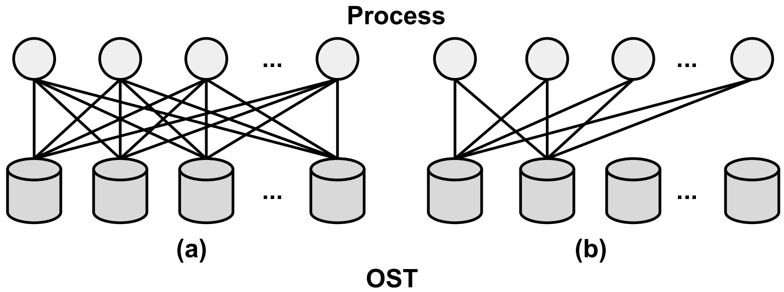


Figure 3.4: Two bad cases where each OST is accessed by multiple processes. (a) Each process access data from all OSTs. (b) Each process access data from a subset of OSTs

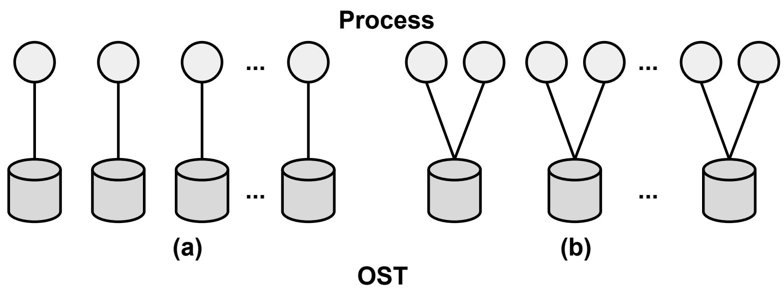


Figure 3.5: Two ideal cases where each OST is only accessed by minimal number of processes. (a) The number of processes equals to that of OSTs. (b) The number of processes is larger than that of OSTs

3.3 Results

3.3.1 Overview of Evaluation

To demonstrate the effectiveness of our framework, we evaluated read performance of kernels extracted from four different scientific applications or datasets from various science domains. We ran all our experiments on a Cray XC30 supercomputer named “Edison” at the National Energy Research Scientific Computing Center (NERSC). The data was stored on a Lustre filesystem equipped with 36 Object Storage Servers (OSSs) and 144 Object Storage Targets (OSTs). All OSTs were utilized and each experiment was repeated multiple times to obtain consistent performance results. To avoid effects of caching, multiple copies of the data were created to guarantee that the same file is not accessed by any two consecutive experiments.

I/O kernels from four different real scientific applications were used in our experiments. These include: (1) querying a 188 billion particle plasma physics data produced by Vector Particle-In-Cell (VPIC) simulation of magnetic reconnection phenomenon to demonstrate the support for

Table 3.2: Data selections used in the experiments for the plasma physics dataset

Test case	High-level representation	Accessed data	Replica overlap %	Original time (s)	Optimized time (s)	Overhead time (s)	Speedup
A1	Energy >1.7	588 MB	100%	15.9	1.5	0.7	10.6
A2	Energy >1.6	1151 MB	100%	27.4	2.8	1.9	9.8
A3	Energy >1.5	2378 MB	100%	115.2	5.1	3.1	28.1
A4	Energy >1.4	5441 MB	100%	677.7	7.5	4.2	90.3
B1	Energy >1.4	5441 MB	44%	677.7	387.5	5.7	1.7
B2	Energy >1.5	2378 MB	48%	115.2	64.6	3.6	1.8
B3	Energy >1.5	2378 MB	25%	115.2	89.7	2.0	1.3

element selection and partial match (Section 3.3.2); (2) accessing climate model and observation data, used for detecting Atmospheric Rivers (AR), to demonstrate the ability of applying historical optimization strategies on new datasets (Section 3.3.3); (3) accessing Electrocorticography (ECoG) data to demonstrate the support for non-regular patters with an ability to perform optimizations for different data regions (Section 3.3.5); (4) accessing data from Mass Spectrometry images to show that our framework is able to support and manage different layout reorganization techniques at the same time (Section 3.3.4). These kernels represent the read accesses of several scientific data analysis applications.

For each experiment, we compare the I/O time between accessing the data in its original layout (row-major), with unmodified HDF5, and that of the reorganized layout that is replicated and selected by our framework. In the climate and mass-spectrometry imaging accesses, we further compare the performance with two scenarios: with and without the availability of additional storage. In the case of no additional storage available, our framework selects one from transposition, Z-curve, and Hilbert-curve based on the pattern. This new layout replaces the original data file.

We demonstrate replica creation only in the AR experiment, and omit this process in the other experiments. Our framework initially creates replicas in an aggressive way, when the allowed storage space is sufficient. That is, data is replicated with the selected layout even if it is accessed once. When the total available storage becomes low with the increase of replicas, only the more frequently accessed data regions are reorganized and replicated. The oldest and less frequently accessed replicas are replaced. This entire process yields a shorter initial “training time”, while maintaining good read performance for the true frequent patterns in the long run.

3.3.2 Plasma Physics Particle Data

The plasma physics particle dataset is generated from the first principles 3D electromagnetic relativistic kinetic particle-in-cell code [12], called VPIC to simulate collisionless magnetic reconnection phenomenon of two trillion particles. This dataset contains properties of particles that include the location (X, Y, Z), kinetic energy ($Energy$), and individual components of particle velocity (U_x, U_y, and U_z). The dataset used for our experiments was a subset of the trillion particle dataset based on the condition of $Energy > 1.1$. We experimented with four variables: "Energy," "X," "Y," and "Z," which are each stored in a $1D$-array using 4096 MPI processes. Each variable contains about 189 billion elements with a size of 700 GB.

In our experiments, we use the queries similar to those of a previous analysis of this data [12] for visualizing the highly energetic particles. The corresponding queries are selecting variables with value constraints such as $Energy > 1.7$ and $Energy > 1.6$. We split the experiments by running queries on the plasma physics data in two groups: A and B, as shown in Table 3.3. Group A ($A1$ to $A4$) represents the best cases where the created replica exactly matches the request. While for group B ($B1$ to $B3$), only a partially matched replica exists, which simulates when storage space can only afford replicas that cover part of the requests. In such cases, our framework would automatically select the replica with the largest overlapping region during runtime.

Table 3.2 compares the read time of accessing the original dataset and the read time of accessing the replicas that use concatenation. The framework overhead is included in the "Optimized time", since the time to load the metadata (mapping from reorganized layout to original layout) is non-negligible. We can see a substantial performance improvement when accessing the replicas for various queries. The speedup increases with the total data accessed in Group A, with the highest speedup being $90X$ in $A4$. The reason for this speedup is that with more elements selected, more seek and read operations are involved, causing contention in the parallel file system and resulting in low

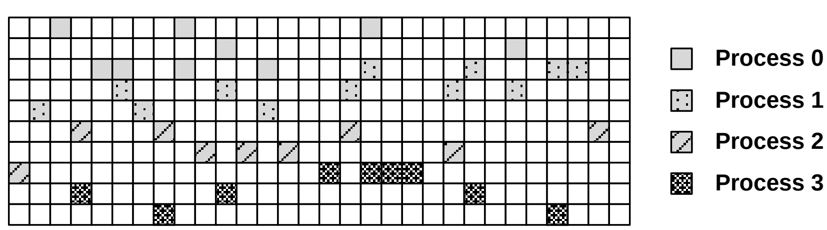


Figure 3.6: An example of the typical accesses for the plasma physics dataset: each box corresponds to one element, each process reads a number of elements with their locations scattered throughout the dataset

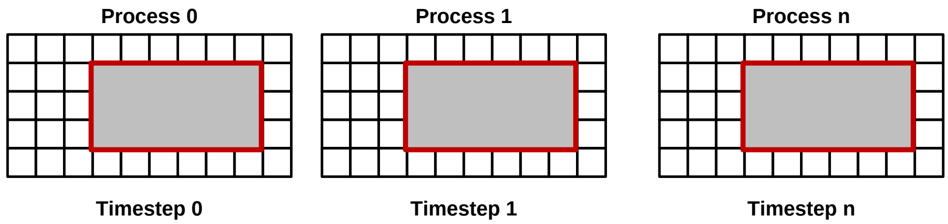


Figure 3.7: An example of AR detection code read accesses in a file. A subset of the rectangular region would result in multiple same-sized blocks with same size gaps in between when data is stored in row-major format

I/O throughput (less than 10MB/s). In contrast, the reorganized replicas are accessed contiguously, resulting in faster performance. For Group B, the speedup increases with the “overlap percentage”, as more data is accessed contiguously with high throughput.

3.3.3 Atmospheric Rivers Detection

Extreme precipitation events on the western coast of North America are often traced to an extreme weather phenomenon, known as atmospheric rivers (ARs). We focus only on the I/O phase of the detection process [13], which is as follows: each MPI process reads and analyzes the integrated water vapor (IWV) variable pertaining to one timestep. Each timestep represents a daily or hourly average of IWV. For each MPI process, a typical data read pattern consists of obtaining data related to an ocean basin. For instance, with the climate dataset for the entire globe, studying the AR events in the US western coast requires reading data corresponding to a rectangular region in a mesh (with the HDF5 hyperslab selection of the $2D$ dataset), as shown in Figure 3.7.

In this experiment, we are simulating the I/O part of AR detecting in two climate datasets. With the same interested region, the data selections are identical when accessing the two datasets. The two datasets $D1$ and $D2$ each have 4096 timesteps and contain 1536×2304 double values per timestep. This resolution represents a 0.25° climate model or observation output. We used 4096 MPI processes and 4 different read requests, $Q1$ through $Q4$, that read 2D sub-planes with 5%, 10%, 15%, 20% of the total dataset size. In the first half of runs with $Q1-Q4$ and dataset $D1$, our framework has no prior knowledge and no replica is available; it only reads from original datasets. The time differences between the first 4 bars in Figure 3.8 are the overhead of our framework. After each of the first 4 queries, with sufficient storage space, our framework automatically creates replicas with reorganized layouts of the previously accessed data. The user is also able to choose to apply such optimizations to another dataset. In our test, we chose to apply on dataset $D2$, and the performance

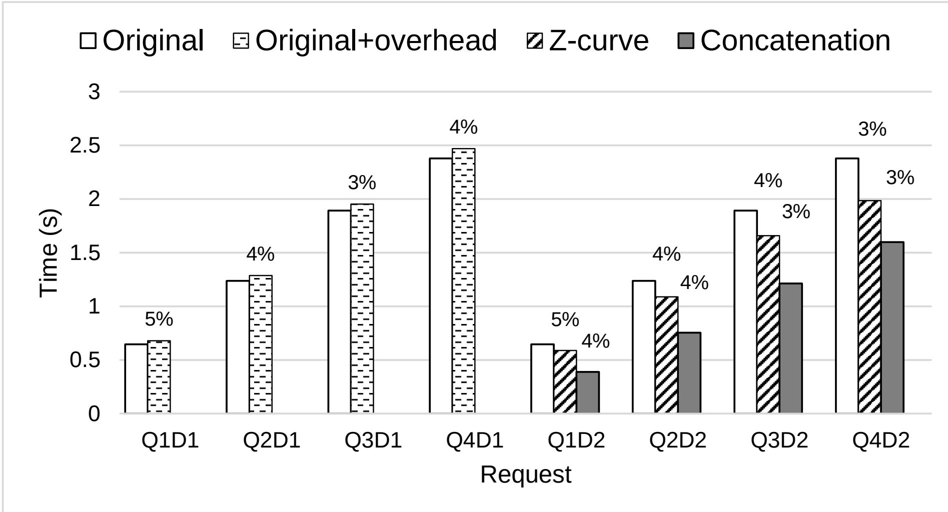


Figure 3.8: $Q1D1$ to $Q4D1$ compares the time of reading from the original dataset $D1$ with ("Original+overhead") and without ("Original") our framework. $Q1D2$ to $Q4D2$ are combined results of two sceneries: (1) no additional storage allowed and our framework selects a "Z-curve" layout and replaces the original dataset with it, (2) sufficient additional space and our framework selects "Concatenation" and creates corresponding partial replicas. The percentage number labels are the framework's runtime overhead

for accessing with $D2$ with reorganized replicas are shown as the last 4 bars of Figure 3.8. From the second half of the results, we can see that even with this small-sized dataset, our framework is still able provide speedups of $1.5X$ to $1.7X$ with concatenation, and $1.1X$ to $1.2X$ with Z-curve.

3.3.4 Mass Spectrometry Imaging

Mass spectrometry imaging (MSI) enables researchers to directly probe endogenous molecules within the architecture of the biological matrix. The data for each position is represented as a profile of intensity values over a corresponding range of mass-to-charge (m/z) values. There are three types of frequently used patterns when accessing an MSI dataset: (1) m/z slice selection ([:, :, zmin:zmax]); (2) spectra selection ([xmin:xmax, ymin:ymax, :]); and (3) 3D sub-volume selection ([xmin:xmax, ymin:ymax, zmin:zmax]), as described in [68].

A 2D MSI dataset can be described as a three-dimensional cube of $(x, y, m/z)$. Each spectrum describes the distribution of masses (m/z) at a given image location (x, y). For the experiments, we used a dataset with size $394 \times 518 \times 133092$. However, with the relatively small dataset, in order to avoid noise in I/O operations, only 144 MPI processes were used so that each process reads a

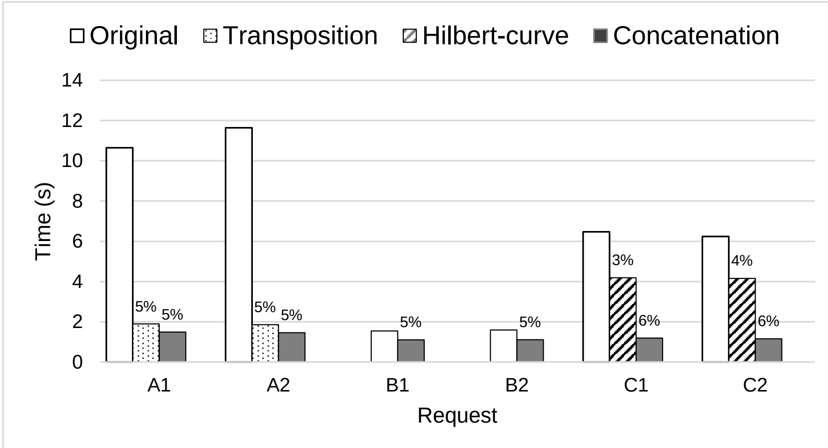


Figure 3.9: Group A compares the time to read from the original dataset, a reorganized layout with "Transposition", a reorganized layout with "Concatenation", where 25 consecutive m/z slices per process are read; Group B compares the time to read from original dataset with the time to read from partial replicas reorganized with "Concatenation", where each process reads 3×3 spectra; Group C compares the time to read from original layout, a reorganized layout with "Hilbert-curve", and a reorganized layout with "Concatenation", where each process reading $20 \times 20 \times 10000$ sub-volumes. The percentage number labels are the framework's runtime overhead

non-trivial amount of data. We simulate the process of exploring the MSI dataset with 6 executions, including each of the above mentioned common patterns. For each type of pattern (referred as "Group"), we vary the location of the selected region and the results are shown in Figure 3.9.

For Group A and C, the accesses include much more non-contiguous chunks when reading from the original row-major layout as the data selection is not aligned with the data linearizion. A new layout that uses transposition or a Hilbert-curve better matches the read pattern and thus offers more contiguous accesses and speedups ranging from $1.5X$ to $8X$. The concatenation further improves the read performance as a result of OST-aware replica placement, which is not supported for transposition (Transposition and Hilbert-curves are applied to the entire dataset, and can only use the parallel file system's striping parameters for data placement). For Group B, the data accesses match the row-major layout and are near-optimized, leaving the only option for performance improvement on data placement, which brings close to a $1.4X$ speedup. Each group has two tests with same data selection but at different locations (e.g. A1 and A2 are different only with the start location).

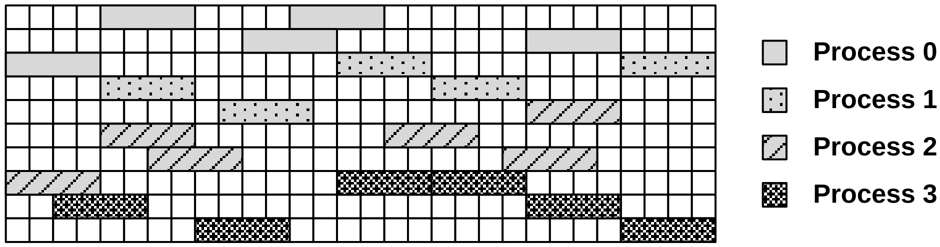


Figure 3.10: ECoG dataset: data having the same label are stored in a file with same-size blocks but variant gaps in between when stored in row-major format

3.3.5 ECoG Benchmark

An electrocorticography (ECoG) experiment records the electrical activity from the cerebral cortex when the patient is reading different words at different times. We used a $1GB$ dataset containing a $2D$-array of 541241344×256 64-bit double-precision floating point numbers. Associated with the data are three datasets of indices and labels and are used to find the target data regions. The pattern of the ECoG benchmark is non-regular (varied-size strided) with queries from users such as "select all data labeled with KEE1." Due to the small size, we only used 24 processes for the parallel reads. Note that data labeled with a certain type such as "KEE1" are scattered blocks stored in the file, as it is recorded at different times (Figure 3.10); their location is determined by the "Event_EIndx" associated with the dataset.

In our experiment, we simulate the access behavior of the ECoG dataset by selecting multiple labels and read their corresponding data. The 6 different experiments are selected with different sets of query labels, while varying the total request size ranging from 5% to 30% of the entire dataset. The results are shown in Figure 3.11. In this case, our framework selects concatenated dataset as the only candidate reorganization strategy. As the data associated with the different labels has different distributions in the file, and has different impact to the overall performance, it is expected that the time to read from the original dataset does not grow linearly with the read size. While using concatenated dataset for reorganization, it is guaranteed that each of the requests is reading a contiguous chunk, thus the time grows close to linearly. Note that we have observed that reading larger contiguous chunk yields higher throughput, which explains why it is not exactly linear growth. The speedups of using our framework range from $2X$ to $6X$, as concatenation makes the previously non-contiguous data accessed from the original layout a big contiguous chunk in the replica.

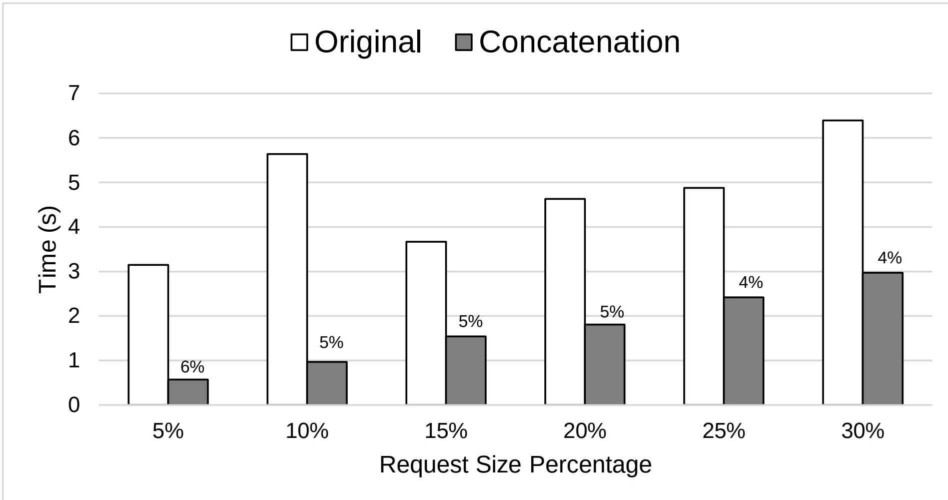


Figure 3.11: Read time comparison with different query selections accessing the ECoG dataset. With non-regular data selection, our framework selects "Concatenation" to reorganize and create partial replicas. The percentage number labels are the framework's runtime overhead

Table 3.3: Cost model validation

Selection	SEL0	SEL1	SEL2	SEL3	SEL4	SEL5	SEL6	SEL7
Dimension	1	2	2	2	3	3	3	3
Candidate Layout	R0, C0	R1, T1	R2, T2, H2, Z2	R3, T3, H3, C3	R4, T4x, T4y, T4z	R5, T5y, T5z, H5	R6, T6y, T6z, H6	R7, T7y, T7z, H7, C7
Selected	C0	S1	Z2	C3	T4y	T5z	H6	C7
Measured Best	C0	S1	Z2	C3	T4y	T5z	H6	C7

R: Original row-major. Tu: Transposition with u as first dimension.
Z: Z-curve H: Hilbert-curve. C: Concatenation.

3.3.6 Validation of the Cost Model

To Validate the correctness and effectiveness of the runtime decision making process that we proposed in Section 3.2.1, we have designed a series of queries with different patterns. We prepared three synthetic datasets with one, two, and three dimensional structure. For the $1D$ dataset, only concatenated datasets are supported, while transposition, and space-filling curves are supported for $2D$ and $3D$ datasets. The data selection type of the patterns (all are non-contiguous accesses) that we use are subset, sub-plane, sub-volume, and element selection.

For layout decision making, the model only needs to provide information to choose between candidates. As experiments are performed in a shared environment, the I/O performance can be affected by many different factors, such as other users accessing the parallel file system and the network. Thus, instead of comparing the absolute performance, we only compare the ranks of different layouts computed by the cost model; this is sufficient for runtime uses.

We varied the data selection types and layout candidates as shown in Table 3.3. $SEL0$ selects a subset of the $1D$ array; $SEL1$ selects a subset along y dimension; $SEL2$ and $SEL3$ select a $2D$ sub-plane; $SEL4$ selects a $2D$ sub-plane with a significantly larger size along y dimension; $SEL5$ selects a $3D$ sub-volume with a significantly larger size along y dimension; $SEL6$ and $SEL7$ select a $3D$ sub-volume with comparable dimension size of all dimensions. The concatenated replicas always yields the best performance as they provide the most contiguous accesses, but at the cost of additional storage space. To simulate the cases that no extra space is allowed, we intentionally include it only once in the candidate layout. We can see that our cost model can make the proper decision among multiple layouts that contain the data requested with different kinds of patterns.

3.3.7 Overhead

The storage overhead used to optimize for bounding box selections is about the same size of the created replicas (the associated metadata are in a compact representation of near-constant size). While for element selection, with the non-negligible metadata size, the total storage overhead is about 1.3 times replica's size (assuming the 32-bit elements in the original dataset, and in the metadata 16-bit integers are used together with compression). Note that when using space-filling curves, the original dataset is replaced and thus requires only constant space with the metadata containing the parameters of the used curve and with no replicas created.

The runtime overhead comes from the three parts in our framework. We labeled the overhead numbers above each bars in the result figures. From our measurement, the total overhead of trace analysis and I/O redirection is relatively small, less than 3% in all test cases. While depending on the data selection type as well as the number of available replicas, the overhead for metadata loading and decision making varied by a great deal. For bounding box selection, it ranges from 3% to 5%, while for element selection, due to the read for the much larger metadata information, it may takes 10% or more depending on the number of candidate replicas after the first round of pruning as mentioned in the first Step of Section 3.2.2. Note that even with such overheads of metadata loading, we observe substantial performance improvements as the resulting read from replica is much more contiguous than from original dataset, as shown in the plasma physics queries.

3.4 Related Work

To optimize data access performance, numerous methods have been proposed in parallel I/O libraries. Two-phase I/O [7, 78], data sieving and collective I/O [79], and datatype I/O [18] have all achieved their success. However, these techniques focus on merging small noncontiguous I/O requests into large contiguous ones, and not suitable for optimizations on large data accesses.

The linearization and organization of the extreme-scale datasets on parallel file systems are crucial to the scientific application's performance. SciDB [10] addresses sub-volume access patterns by applying array slicing, joining operations, and array division into regular/irregular chunks for multi-dimensional scientific data. EDO [82] optimizes sub-plane and sub-volume access patterns for spatial locality through Hilbert space-filling curves reordering and sub-chunking. However, SciDB and EDO both reorganize the original data layout and provide average performance for sub-region accesses. In contrast, we present a framework that maintains replicas with layout optimized for the common access patterns in scientific data explorations.

Chunking is another layout optimization technique that splits the dataset into multiple chunks and improves performance when operating on a subset of the data [38]. However, current approaches [58, 70], such as those supported by HDF5, are not flexible enough to meet the need for the dynamic patterns discussed in this work. The chunking is applied with fixed chunk dimensions and cannot be modified afterward, limiting its applicability for non-regular spatial patterns. OpenMSI [68] adopted chunking, compression, and data replication to improve the data access efficiency for MSI datasets. While it focuses on a specific domain and pre-generates all the reorganized replicas, our framework covers broader types of data usage patterns and dynamically adapts to the change of patterns.

MLOC [33, 34] proposed a parallel layout optimization framework to achieve better performance for queries on scientific datasets with heterogeneous access patterns. Though multiple layouts are discussed, it focuses on one layout at a time and does not provide either runtime decision making or layout management. In contrast, our work supports multiple layouts chosen based on the users' specification as well as runtime decision making. FastBit [85] and ISABELA [45, 46] used binning for value-constraint query optimization with indexing.

Given additional storage, creating multiple partial replicas, each optimized for a specific kind of access pattern, can greatly improve the read performance and meet the need of heterogeneous access patterns of scientific applications. InterferenceRemoval [96] analyzes and identifies segments of files with interfering accesses and replicates them to designated I/O nodes. PDLA [90] explored data replication for patterns with high regularity and selected from three layouts: 1-DH, 1-DV, and 2-D on the parallel file system. Earlier we introduced RADAR [40], which maintains partial replicas and selects the most optimized one for current access pattern during run-time. However, both approaches are limited to regular spatial patterns, with no optimizations for patterns induced by element selection.

The Scientific Data Services (SDS) system proposes to apply data management optimizations transparently without placing burden on scientific application developers [29, 86]. One of the services SDS proposes is to reorganize and to replicate data on parallel file systems. SDS has a client-server architecture. The server would analyze the access patterns of I/O read calls, identify the data layouts that benefit the read patterns, perform data reorganizations, and manage the metadata of the reorganized datasets. SDS requires to traces the accessed files, variables, and the offsets (data locations) of the application's reads and pass them to access pattern analyzers for identifying read patterns. The analyzed results can then be stored as metadata and managed by SDS Metadata Manager, which implemented using Berkeley DB. With the identification of the data usage patterns, the layout reorganizer will create replicas with optimized layout for the patterns. These replicas will be used for future accesses that have same or similar patterns. While SDS has capabilities and a framework to perform reorganizations and to redirect data accesses to suitable layout, several components are yet to be developed. For instance, components for capturing data accesses, for analyzing and detecting read patterns dynamically, and for estimating costs and benefits are still missing in SDS. The methods developed in this study, i.e., trace capturing and analyzing, layout decision making, and layout reorganization methods, are planned to be integrated into SDS to provide a broader applicability than its current implementations.

3.5 Summary

In this chapter, we proposed a framework that selects the most suitable layout among the common layout reorganization techniques based on detected data usage patterns. It is capable of performing storage-efficient optimizations for heterogeneous patterns from both bounding box and element data selections. On datasets of scientific applications from various domains, our framework yields $1.3X$ to $90X$ time speedup in the plasma physics queries, $1.1X$ to $1.7X$ speedup in the AR detection,

$1.4X$ to $8X$ in MSI, $2X$ to $6X$ in ECoG experiments.

CHAPTER

4

AMR DATA LAYOUT OPTIMIZATION

In Chapter 3, we presented the design of a layout optimization framework for uniform grid data. In this chapter, we study the organization of block-structured AMR data on storage to improve performance of spatio-temporal data accesses. To enable efficient AMR read accesses, we develop an in situ data layout optimization framework. Our framework automatically selects from a set of candidate layouts based on a performance model, and reorganizes the data before writing to storage. We then evaluate this framework with three AMR datasets and access patterns derived from scientific applications.

4.1 Introduction

As scientific simulations move towards exascale, parallel I/O continues to be a primary performance bottleneck in HPC systems. An effective strategy for alleviating this bottleneck is to perform data layout optimizations, where both the *physical distribution* of the data onto storage devices as well as the *logical layout* of data is optimized. Parallel file systems, such as Lustre [8], GPFS [71], and PVFS [66] provide high I/O performance with "striping" to *physically distribute* blocks of data to multiple storage devices; however, these require the users to understand the I/O workload of their application as well as the underlying parallel file system. Moreover, the selected policy is often fixed

Table 4.1: Data storage layout optimization approaches

Design Principle	Approach	Uniform Mesh	Cell-based AMR	Block-structured AMR
Logical Layout	Space-filling curve	Hilbert-curve [34, 82]	Hilbert-curve [41]	**Enabled by this paper**, limited performance improvement in certain cases (Figure 4.3).
	Array transposition	Full replication in [50].	Applicable, but limited to specific access patterns (large size along one dimension)	
	Concatenation	Partial replication [40, 90]	Applicable, require prior knowledge of specific data access regions and additional storage for creating replicas.	
	Cluster-based	Becomes chunking [10].	Different implementation.	**Proposed in this paper**
Physical Distribution	Striping	Supported by parallel file systems such as Lustre [8], GPFS [71], and PVFS [66], strategies such as 1DH, 1DV, and 2D [76]. Static policy and most useful for write optimization.		

and is costly to change. This is exacerbated for spatio-temporal data accesses (accessing a subset of time-series data) that result in non-contiguous access patterns, which are dynamic and difficult to predict.

On the other hand, *logical layout optimizations*, which focus on increasing the data locality, have proven to be effective for improving spatio-temporal data read accesses. For example, space-filling curves (SFCs) [69], such as the Hilbert curve, can be used to linearize multidimensional data [34, 82]. In addition, array transpositions [50] or concatenations of multiple non-contiguous segments into one contiguous block [40, 90] can be performed to create replicas and redirect the read request to the appropriate specialized layout at runtime (*runtime* and *in situ* are used interchangeably in this paper).

However, existing logical layout optimization techniques primarily focus on uniform mesh data and fall short when applied to AMR data, which has a dynamic and hierarchical data structure. The Hilbert SFC has been used in an AMR cosmology application, to map between spatial and on-disk locations of the root cells and improve the I/O performance [93]; however, this was *cell-based* [41] AMR, not *block-structured* [4] AMR (Figure 4.1). In cell-based AMR, the refinement is performed on a cell-by-cell basis: one cell is refined into uniform-sized smaller cells if higher resolution is required. The finer level children cells fill the entire space of the refined root cell and can be treated as an individual piece of uniform mesh.

Block-structured AMR inserts smaller blocks (boxes) at places where high resolution is needed. These boxes are grouped into a hierarchy of refinement levels that adapt throughout the computation as the areas of interest move. Unlike cell-based AMR where boxes of the same refinement level are aligned, the boxes of block-structured AMR can be located *arbitrarily* in the problem domain. The AMR boxes also come with varying sizes and one finer level box can span multiple coarser level

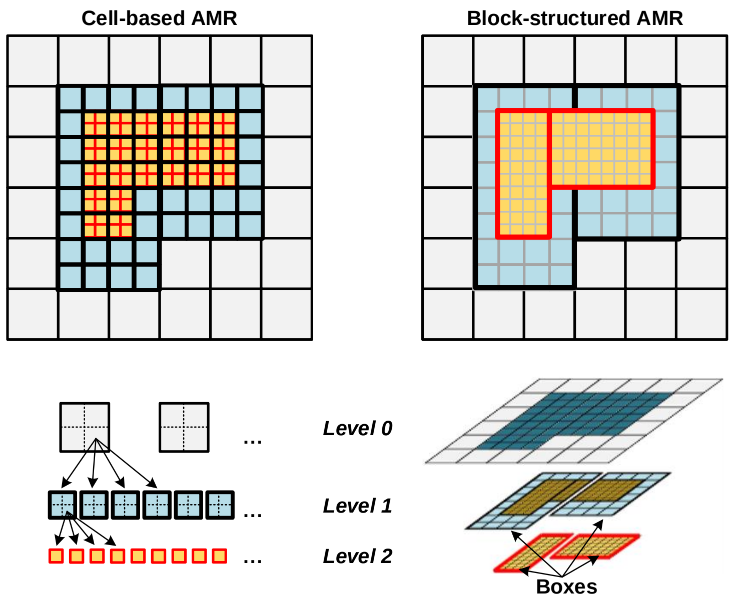


Figure 4.1: An example of the mesh configurations for cell-based and block-structured AMR data, with three levels and a refinement ratio of 2

boxes. As a result of these characteristics, the SFC-based approach used in [41] requires further adaption in order to work. Even with SFC adapted to block-structured AMR, there are cases (as shown in Figure 4.3) that achieve poor read performance with spatial data accesses. We summarize the applicability of the existing work in Table 4.1.

To enable efficient block-structured AMR spatio-temporal read access, we propose an in situ block-structured AMR data layout optimization framework. Our layout optimization directly operates on in-situ block-structured AMR data, dynamically selects a preferable layout, and writes out the data with the optimized layout to the storage system transparently to users. Our framework encompasses SFC approaches as well as a new cluster-based approach that we propose. To the best of our knowledge, this work is the first to focus on layout optimizations for *block-structured* AMR data. In summary, we make the following contributions:

- We create a framework that is applicable to block-structured AMR and is integrated into the Chombo AMR infrastructure [20], enabling automatic and transparent layout optimizations along with I/O aggregation for improved read performance.
- We enable two SFC approaches and introduce a cluster-based layout reorganization approach for block-structured AMR. We also propose a dynamic prediction-based layout decision

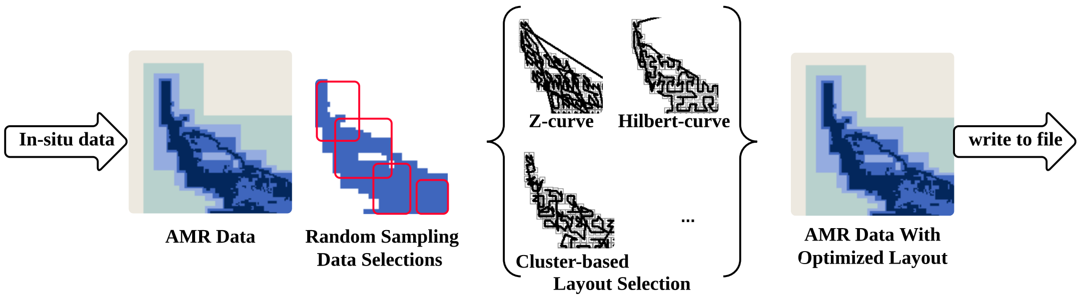


Figure 4.2: Overview of the run time data layout reorganization process. With the AMR data in memory, our framework first selects one AMR level (level 3) data and randomly select a number of sub-regions (bounding box selection), it then applies different layout on the data and compare the estimated read time using a performance model. The most suitable one is selected based on both performance and processing overhead, and applied to all levels of the AMR data before writing to file

making strategy for efficient spatio-temporal data read access.

- We evaluate our framework using three AMR applications and datasets and show a read performance improvement over the datasets' original layout along with low runtime processing and small storage overhead.

The remainder of this paper is organized as follows: In Section 4.2, we present the proposed AMR data reorganization framework and describe the functionality of different components. We demonstrate the application of the framework in Section 4.3 using real AMR applications and datasets, and evaluate the effectiveness of the framework as well as the overhead. Finally, we discuss the related work in Section 4.4 and conclude the paper in Section 4.5.

4.2 Method

4.2.1 Approach Overview

Our proposed approach aims at enabling efficient block-structured AMR spatio-temporal read accesses. With the AMR data generated and stored in memory, our approach selects a preferable layout automatically, before the data is written to the storage system. This selected storage layout will benefit future spatio-temporal data read accesses.

Our proposed runtime layout optimization framework can be separated into two components: *data layout optimization* and *I/O direction*. Data layout optimization, as shown in Figure 4.2, focuses on the selection of a favorable layout based on the characteristics of the AMR data. While the discussion in this paper centers on the runtime process, our method could also be used as a post-processing tool to reorganize previously written AMR data. The second component, I/O direction, is in effect when the framework detects the current accessing file has a reorganized layout. For convenience purposes, we also provide a new API for reading the data with a user-specified spatial selection. We have integrated our work into the Chombo infrastructure, making the layout optimization transparent to users: they only need to re-compile with layout optimization enabled in Chombo.

4.2.2 Runtime AMR Data Layout Optimization

The runtime layout reorganization process is triggered at data write time after the AMR hierarchy is generated and the data (stored as AMR boxes) are distributed across the processors. Each processor has a local copy of the entire AMR hierarchy, along with other metadata (such as AMR level and all the AMR boxes' locations), while only holding the data of AMR boxes that are assigned to it. The assignment of AMR boxes to the processors is determined by Chombo, and we do not rely on any particular assignment strategy. All the layout optimization approaches operate on the level basis, as the AMR data is organized and stored by each level in Chombo. Below, we first introduce the two proposed layout reorganization strategies and then describe the decision making process for selecting a target layout.

SFC approach

Using a space-filling curve (SFC) to reorganize data layout is an effective way to increase data locality. A unique SFC index is assigned to each data element according to its spacial coordinates in the overall computational space. By using this SFC index as the new linearization order (layout) to organize data in the storage devices, the nearby data elements are stored more closely, on average, than if they were stored with the commonly used row-major or column-major layout.

Existing work that uses SFCs for layout reorganization assumes to operate on regular mesh data, where all the data elements have uniform size and fill the entire problem domain. SFCs can be directly applied to cell-based AMR, where the refined cells have identical sizes on the same levels. Specifically, a refined cell is filled with r^d finer cells, where r is the refinement ratio and d is the dimension of data, common r values are 2 and 4, and common d values are 2 and 3. However, block-structured AMR has data organized in a very different way. As shown in Figure 4.1, block-structured AMR refines regions of interest on a level basis. One finer level box can span multiple coarser level boxes and can be inserted at any location with different sizes, resulting in a non-regular mesh. As a result, it requires extra work to process the data and adapt the SFC methods for block-structured AMR.

To enable SFCs for block-structured data, we propose an approach that maps each AMR box to a coordinate, computes their respective SFC index, and uses this index for layout reorganization. To map the AMR boxes, we first create a “virtual” regular mesh using a unit size equal to the average size of all boxes on the current level. Through our evaluations, we have found that the average size provides better improvements than others such as the maximum or minimum size. Each box is mapped to one unit of the new grid based on the overlap percentage. In cases where one large box covers multiple units, the decision is made based on the element that would result in most adjacent boxes. In this way, each AMR box is represented by only one coordinate and the SFC index can be computed. This approach also prevents a sparse data distribution that one would get if using the coordinates (e.g., the coordinate of the lower left corner) of the boxes directly. The last step is to sort the boxes according to their SFC index because all units in the virtual grid may not have a mapped box.

With the implementations of the Hilbert- and Z- curve-based layout reorganization strategies, we found that these SFC approaches would lead to a number of back and forth “jumps” in cases such as those shown in Figure 4.3. Each of these “jumps” results in a non-contiguous access if only one box is selected for read. Consider each AMR box as a node in a graph and the order of the boxes forming edges of a path (linearization order) containing all nodes. With a bounding box selection, the resulting number of non-contiguous accesses is essentially the number of edges between the selected nodes (boxes) and the rest of the graph. As a result, the more “jumps” that exist, the more non-contiguous accesses that are likely to happen.

Cluster-based approach

To reduce non-contiguous accesses in cases like those shown in Figure 4.3, we propose a cluster-based layout optimization approach. This approach uses *hierarchical clustering*[1] to group AMR boxes into *square-shaped, bulk I/O RPC-sized* clusters. *Spiral order* is used for the order of boxes within each cluster, and the order between clusters is based on a *depth-first traversal* of a *minimum spanning tree*. We explain the rationale behind each choice in the following paragraphs.

Data clustering is a common technique in data mining to group similar objects into clusters. As we would prefer to group AMR boxes that are close in space and store them as contiguous as possible on storage devices, hierarchical clustering (HC) becomes a good choice, as objects are grouped based on their distances. Choosing a distance function is an important factor for HC. We initially tried

[1] Pairwise average-linkage clustering

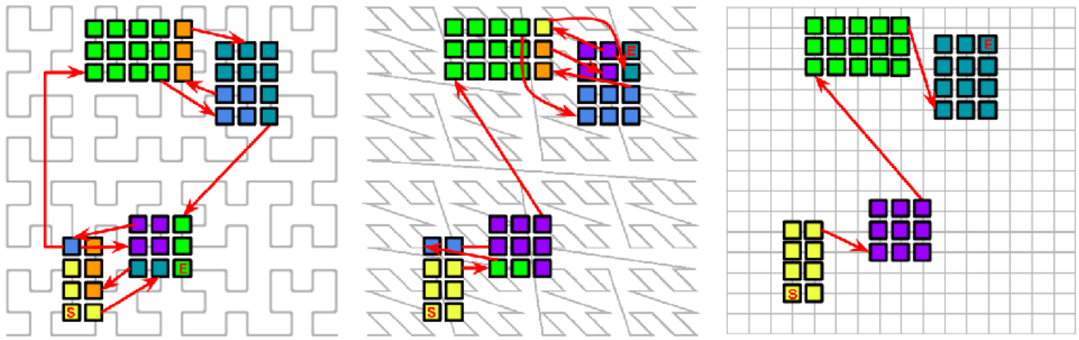


Figure 4.3: Examples of applying three different layout reorganization approaches to the AMR data. The example data has four disjoint components, each with a number of AMR boxes (represented as squares). The red arrows are the "jumps" in the new order defined by the SFC, and boxes that are contiguous have the same color within each component. "S" and "E" represents the start and end box as a result of the SFC

Euclidean distance (2-norm distance, $D(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$); through our experiments, however, we found it does not offer good performance. Round-shaped clusters would be formed and, as the bounding box selections request rectangular regions, the cluster is often "cut" into pieces, resulting in more non-contiguous reads. To solve this problem, we switched to *Chebyshev distance* (infinity-norm distance, $D(p,q) = \max_i(|p_i - q_i|)$) to form *square-shaped* clusters that align with the bounding box selection. Additionally, using a bottom-up approach for HC provides us the flexibility to control the size of the clusters.

With HC, there are two more important aspects to consider, the number of objects and the total data size of each cluster. Too few or too many boxes in one cluster will not significantly improve the data locality. As with the total data size, it would be preferable to have it match with the file system's readahead size. For example, the Lustre parallel file system provides the option to configure its *bulk I/O RPC size*, which is the smallest data block to access at one time (1MB by default). The common size of AMR boxes ranges from 8×8 to $16 \times 16 \times 16$ of floating-point numbers, and through our experiments we have observed good I/O performance using the bulk I/O RPC size as the threshold size for each cluster. Additionally, clusters of bulk I/O RPC size also alleviates the "cut" problem we mentioned previously, as each cluster is likely to be read entirely by the file system, even if not all its boxes are selected.

With the clusters formed, the final step is to generate the order of AMR boxes within clusters and between clusters. We use *spiral order* for intra-cluster AMR box ordering; this minimizes the potential effect of the "cut" as mentioned previously. To generate the order between clusters, we find it effective to generate a *minimum spanning tree* and base the order on a *depth-first traversal* of the tree. Algorithm 1 describes the process for our cluster-based approach. Figure 4.4 compares the results of applying the three approaches to an AMR dataset.

Algorithm 1: Cluster-based Layout Optimization

Data: n AMR boxes with coordinates L and sizes S; $thres$ is the total data size of a cluster
Result: A new AMR box order sequence s

Patition the problem domain into sub-regions;
Assign the partitions to different MPI processes;
Each of the participated MPI process performs the following cluster algorithm;
label n boxes as n unprocessed clusters, S_i is the box size for C_i, $C_{finished} = empty$;
while *there exist a* C_i *labeled unprocessed* **do**
    Find the closest pair of clusters $C_i \notin C_{finished}$ and $C_j \notin C_{finished}$, based on their Chebyshev distance;
    Mark C_i and C_j as processed.;
    Merge C_i and C_j into a new cluster C_k;
    Assign the sum $S_t = S_i + S_j$ of C_i and C_j to C_k;
    if $S_t > thres$ **then**
        label C_k as finished;
Adjust on the border boxes between adjacent partitions;
Generate the spiral order of each cluster;
Generate a Minimum Spanning Tree (MST) of the clusters;
Perform depth-first traversal of the MST to generate the new AMR box order O;

One drawback of the cluster-based approach is that it comes with a much higher time complexity; namely, $O(n^3)$ with sequential processing, where n is the number of total AMR boxes. Compared with the $O(n)$ complexity of SFC-based approaches, this computational overhead is much higher, even when parallelizing the HC algorithm. Table 4.2 compares the data layout reorganization time for the three different approaches that have been mentioned thus far. We observe a comparable performance between Hilbert- and Z- SFCs. However, the cluster-based approach yields a much higher reorganization time.

A mechanism that allows dynamic runtime layout selection is required due to the significant processing time difference between using SFC and cluster-based approach, considering that the SFC only results in worse data locality in some cases (see Figure. 4.3). Such a decision making process will select the cluster-based optimization when it observes that applying a space-filling curve will

Table 4.2: Data reorganization time of different approaches

Level	# boxes	Reorganization Time (s)		
		Hilbert-curve	Z-curve	Cluster
L0	36	1.80E-05	1.50E-05	1.80E-05
L1	557	1.84E-04	8.40E-05	1.74E-04
L2	662	4.38E-04	4.98E-04	2.23E-04
L3	1512	1.40E-03	1.27E-03	5.38E-01
L4	3262	6.25E-03	5.93E-03	2.17E+00

result in too many "jumps". We will discuss our performance model-based layout selection process in the following sub-section.

Performance Model

To compare the performance of different AMR data layout optimization approaches, we first introduce a performance model that focuses on the number of non-contiguous accesses considering the parallel file system's bulk I/O size. In addition to the number of non-contiguous counts that affect the read performance, we also found that the distance between successive non-contiguous accesses is another factor with non-trivial performance impact. We have performed various experiments on Cori's Lustre file system (more details about Cori in Section 4.3) to confirm such an impact.

We benchmarked the read time of five categories, each of which reads the same 32 non-contiguous blocks with sizes between $1MB$ to $10MB$ (typical block size for individual AMR data read). In keeping with the experiments in Section 4.3, we use union of multiple hyperslabs in HDF5 [81] to read

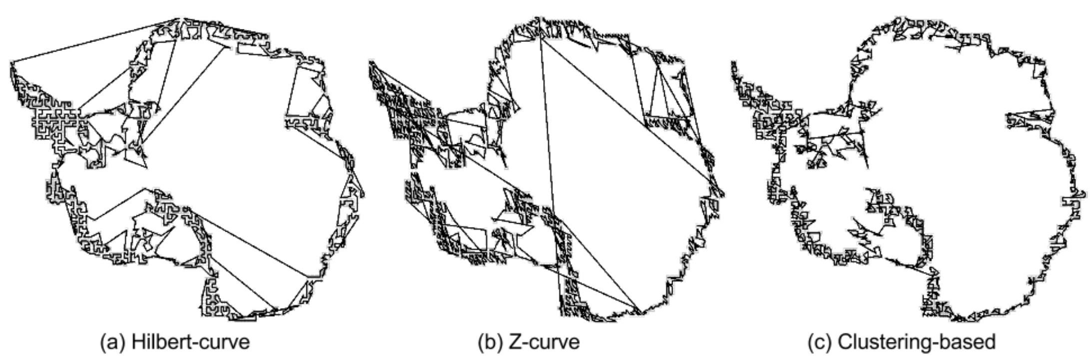


Figure 4.4: Comparison for the layout reorganization results of applying three approaches to the level 3 of the Antarctica AMR dataset

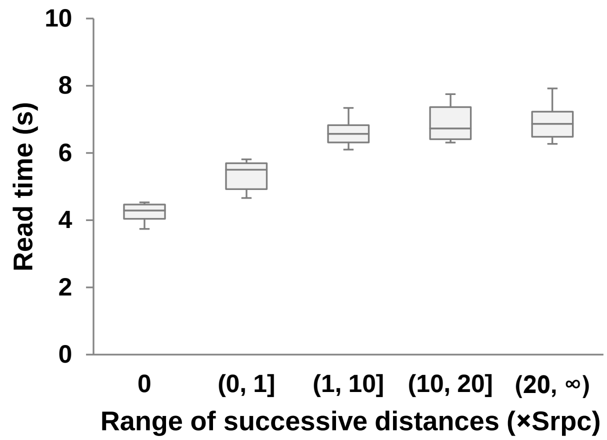


Figure 4.5: Comparison of read time with different ranges of successive non-contiguous read offset distances

all 32 blocks at once. We set the offset distance between successive reads randomly (but within a given range, e.g. $(0,1] \times {S_{rpc}}^2$) for each category, and repeat them 10 times for the performance comparison box plot shown in Figure 4.5. We can see that there is a relatively high performance difference between category "0" (all contiguous reads) and "(0,1]", "(0,1]" and "(1,10]". We also obverse that the read time becomes similar after the size of successive distance exceeds 20 times the bulk I/O size. We estimate the read time T_r of a particular data selection on an AMR dataset as follows:

$$T_r = T_t + \max\{\alpha C_s^i + \beta C_m^i + \gamma C_{lx}^i \mid \forall i \in \mathcal{O}\} \tag{4.1}$$

where $\mathcal{O}$ is the set of Object Storage Targets (OSTs) used in the Lustre file system, C_s^i, C_m^i, and C_{lx}^i are the counts of $(0,1] \times S_{rpc}$, $(1,10] \times S_{rpc}$, and $(10,\infty) \times S_{rpc}$ non-contiguous accesses. T_t is the estimated time for reading the entire data selection, if the data were stored contiguously in a file. The α, β, and γ are weights based on the estimated I/O cost of the three categories. Their exact values depend on storage hardware characteristics and are determined dynamically at application start time to reflect the dynamic changes in the shared system. We do not determin them before each access as it involves too much overhead and invalidate our optimization. We assume that small-sized time-series datasets are stored in a single OST, and larger datasets ($>$ 10GB) are striped across all the OSTs.

This performance model estimates the total time for reading data from multiple OSTs and

[2] We use S_{rpc} for bulk I/O RPC size

assumes each OST offers the same I/O rate as well the network and storage latencies. The C^i terms are obtained for a given selection and layout by flattening the requested region into linear space, calculating the distances of the successive non-contiguous accesses and keeping a count for each distance category. Since we are comparing the time estimate for the same data selection over different layout reorganization strategies, T_t is regarded as a constant and is eliminated during comparisons.

Target Layout Decision Making

As we have previously discussed, the cluster-based approach is expected to produce better read performance improvement than the SFC approaches in cases shown in Figure 4.3. However, the computation overhead of cluster-based organization is much higher than that of SFC in all the cases. To take advantage of the fast processing time using SFC when the expected improvement is similar, (within 10% using our performance model), we introduce the layout decision making process.

Our layout decision making process is based on an estimation of possible data accesses. Since we do not assume the knowledge of the exact future data accesses for the AMR dataset at the time of reorganization, an *optimal* decision cannot be made. However, we are able to approximate the “optimal” decision by using a data sampling approach.

The sampling process is performed as follows: in each round of sampling, a random AMR box is selected and its centroid is used as the centroid of a bounding box selection; then the size of the bounding box is determined with each dimension a random length between $5X$ the box dimension to half of the problem domain on that dimension. The two numbers are chosen based on our observation of real AMR spatial selections. With the bounding box selection determined, we use the performance model described earlier to calculate the estimated time for n such selections using each of the layout reorganization approaches (we found n =the number of boxes on that level sufficient for the decision making). The layout with lowest score from the model is expected to provide the most performance improvement for future spatial data read accesses.

In order to minimize the overhead of this process, we only perform it on either the second finest level (or the finest level if no more than 3 levels). This is based on the fact that the second finest level is already of high resolution, and the refined region is similar to the finest level. In our experiments with the BISICLES dataset (description in Section 4.3.1) of 5 AMR levels, we found almost identical estimated time using the performance model; while for the other datasets with only 3 levels, the finest level is needed for accurate estimation.

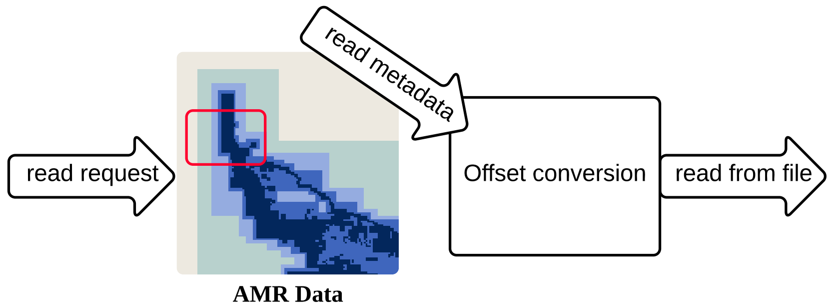


Figure 4.6: Overview of runtime I/O direction

4.2.3 Runtime I/O direction

With our framework enabled at data read time, the opened file is first checked and, if identified to have an optimized layout, the corresponding AMR box offset array (see Section 4.3.2) is read and used, as shown in Figure 4.6. Otherwise, the original read operation is performed. We aggregate the AMR box selections using the union of multiple hyperslabs in HDF5 [81] to avoid small I/O accesses.

4.3 Results

4.3.1 Experiment Overview

To evaluate the effectiveness of our framework, we have conducted experiments on “Cori” Phase 1, a Cray XC40 system at the National Energy Research Scientific Computing Center (NERSC). Cori’s Lustre file system has 248 OSTs, with a peak performance of 744 GB/s.

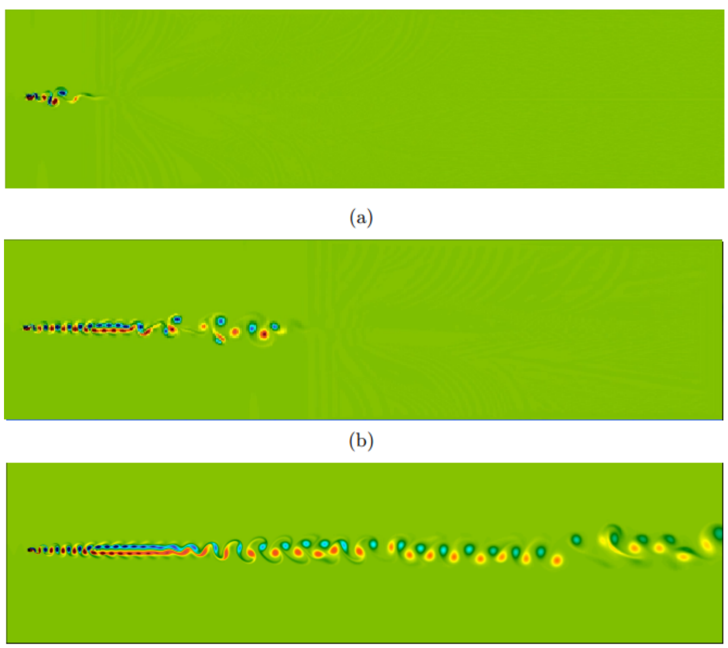
In our experiments, we have used all the 248 OSTs. For obtaining consistent performance results, we repeated each experiment 10 times. To avoid any effect of caching while data is being read, we have used multiple copies of the datasets to guarantee that one file is never read by any two consecutive experiments. All the read experiments retrieve data from all AMR levels.

Datasets

In our evaluation, datasets from two real applications and an I/O benchmark are used. We provide their description as follows:

BISICLES - Antarctica Ice Sheet Modeling: The BISICLES code [21] models ice sheet dynamics in regions such as Antarctica and Greenland. We have setup the BISICLES simulation to generate the ice sheets of Antarctica with 5 AMR levels, a base problem domain of 768×768, and a refinement ratio of 2. The generated AMR dataset is about $1.3GB$ for each timestep and the finest level has ≈3200 boxes (varied in different timesteps).

Turbulent Flow in 2D: This simulation code is the modeling of the two dimensional flow past a cylinder [95] during transition to turbulence, known as the Karman vortex street as shown in Figure 4.7. The simulation depicts secondary, tertiary, and quaternary wake structures. These structures are captured at high resolution where only the vortices are refined on the grid using AMR combined with a cut cell method. The wake extends approximately 250 cylinder diameters and total time simulated is 112 seconds. The simulation produces 478 timesteps with increasing data sizes (ranges from 31MB to 344MB), each of stored as one files on one OST. The dataset has 3 AMR levels, with the coarsest level being 1024×256. We carry out the experiment with a spatial selection of the middle region (a 512×256 rectangular region) as an "observation window" of the flow activity. We use 478 MPI processes and each process only reads data from one file. Note that in the first few timesteps the flow has not reached to the selected region, thus no data is read in those files.



(a)

(b)

(c)

Figure 4.7: 2D flow past a cylinder at Re = 300 with extended wake (l = 32). (a), (b), and (c) are the visualization of three different timesteps

Chombo I/O benchmark We also used the Chombo I/O benchmark to generate even larger AMR

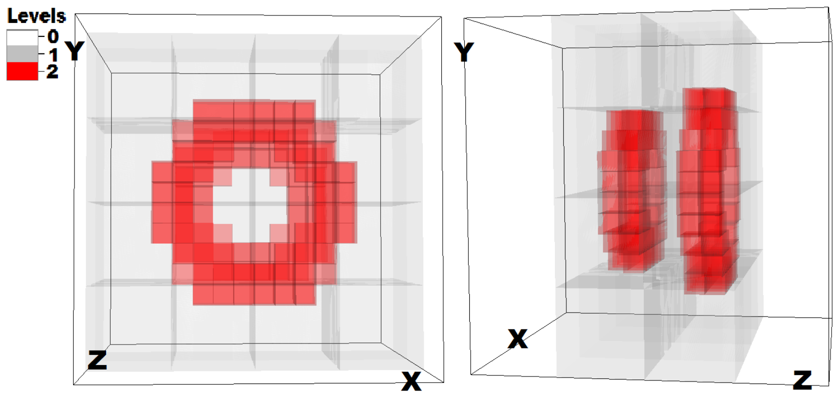


Figure 4.8: Two views of the unit data region generated by the Chombo I/O benchmark

datasets. This I/O benchmark provides the function to generate synthetic AMR datasets based on user specifications, and replicate a unit data region along different dimensions to create arbitrary data file size. Figure 4.8 illustrates the unit region generated, we set the replication factor to "8, 8, 8", which means the resulting data file has 8 replications of the unit region along all three dimensions. The generated file has a problem domain of $256 \times 256 \times 256$ on its coarsest level, 3 total levels with 512, 16381, 98304 boxes on each one, and about 338GB in total size.

4.3.2 Evaluation of data organization overhead

First we analyze the overhead involved in deciding which layout optimization strategy to use as well as the corresponding reorganization time. For these experiments, we used sub-region data selections from the BISICLES dataset [21].

Computational Overhead

The decision making process for selecting a target layout is the first step of our layout optimization method. As stated in Section 15, this process is only performed on *one* level, and the total time for computing the score using the performance model on takes less than 5% of the write time (e.g. about $0.5s$(3.5%) for the BISICLES dataset in Section 4.3.4). The layout reorganization overhead will depend on the selected layout optimization strategy. Both of the space-filling curve has $O(n)$ a time complexity and the cluster-based approach is $O(n^3)$, where n is the number of total AMR boxes.

In Figure 4.9, we show the total time comparison between different layout optimization approaches and the baseline case (original). The total overhead ranges from 25% to 75% for cluster-based approach and 5% to 20% for SFC approaches, compared to the total write time. With the

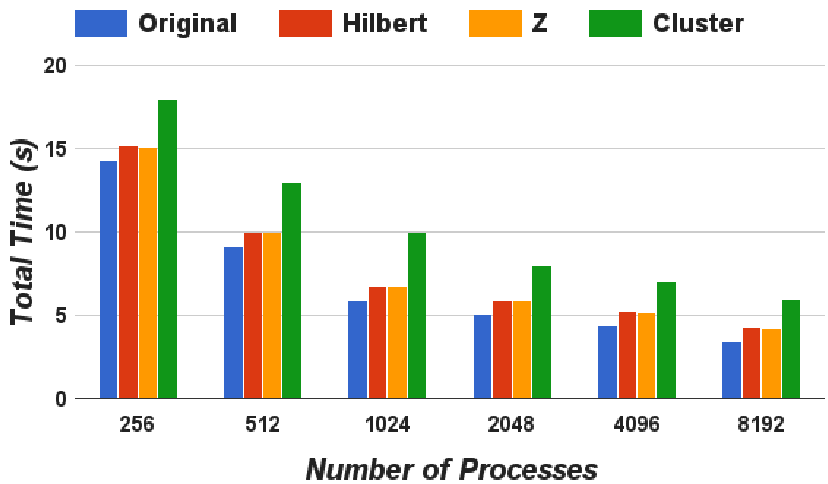


Figure 4.9: Runtime layout optimization overhead. "Original" is the total write time, and the other three bars each represents the total elapsed time for one approach. The time difference between the three approaches and the "original" is the overall computation overhead of our framework

consideration that the AMR dataset is write-once-read-many, the performance benefits overweight the runtime overhead. In the case of applying the layout optimization to existing dataset via post-processing, it needs to read data from storage system (compared to the in-memory data for runtime processing) before the layout optimization and then write the optimized layout.

Storage Overhead

Our layout optimization work directly operates on the AMR dataset and no data is replicated. However, Chombo linearizes and stores the AMR box data as one 1D array and uses another $1D$ array to keep the starting offsets of each AMR box. This mapping array does not explicitly include the size of each AMR box, which is needed when reading from the reorganized layout. Thus, we added such information into the dataset. The amount of storage space needed to store this extra information is minimal. The total storage overhead is $n \times 8$ bytes, where n is the total number of AMR boxes, and is insignificant compared to the typical AMR simulation dataset sizes.

4.3.3 Evaluation of Performance Model

To evaluate the correctness of our performance model, we used 25 different sub-region selections and compare the estimated cost calculated by our performance model and the actual read time. We used the same BISICLES datasets as we did in the previous section.

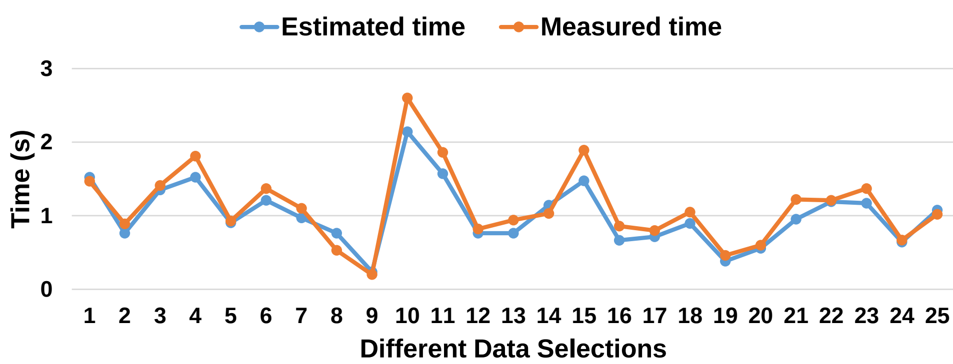


Figure 4.10: Comparison between model predicted time and measured read time of different data selections

Figure 4.10 shows the estimated time is generally close to the actual read time for the 25 different data selections. The model overestimates in some cases such as 10 and 15, this is because the model estimates the time in categories rather than individual values. The average time difference between the two is less than 15%, with a variance of 0.014 and a standard deviation of 0.12.

4.3.4 Read Performance Evaluation - BISICLES

As mentioned in Section 15, the target layout selection is based on user's priority. If the user prefers the most performance improvement regardless of the overhead, the cluster-based approach is always selected. Otherwise our framework will select Hilbert-curve as the estimated performance for the three methods do not differ significantly.

In this set of experiments, we have used 11 sub-region selections that correspond to major ice shelves in the Antarctica. We compare the read time of the AMR dataset with different layout reorganizations and the results are shown in Figure 4.12. This read time comparison also includes the construction of AMR structure along with the data read process. Further, the I/O throughput, calculated only with the data read time, is shown in Figure 4.13. We observe a speedup of $1.5X$ to $3X$ compared with using the original layout, and the cluster-based approach achieves 88% of the throughput with contiguous accesses on average.

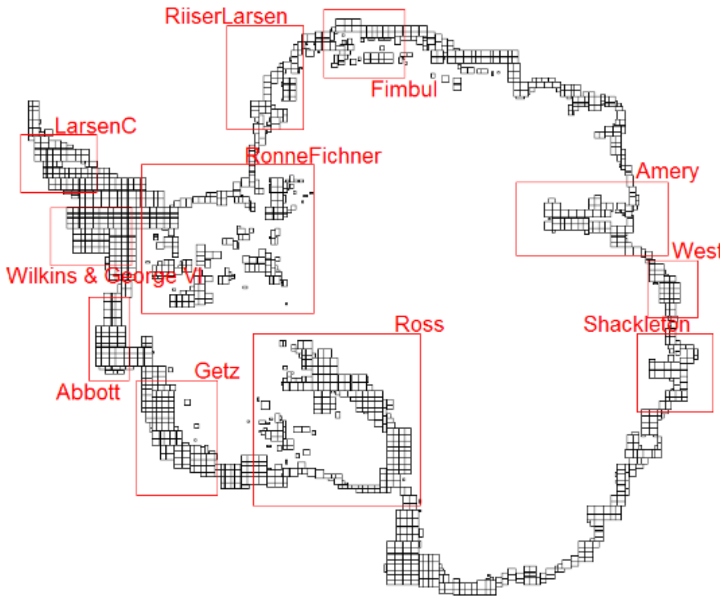


Figure 4.11: Sub-region data selections of the Antarctica ice sheets dataset produced by BISICLES simulation

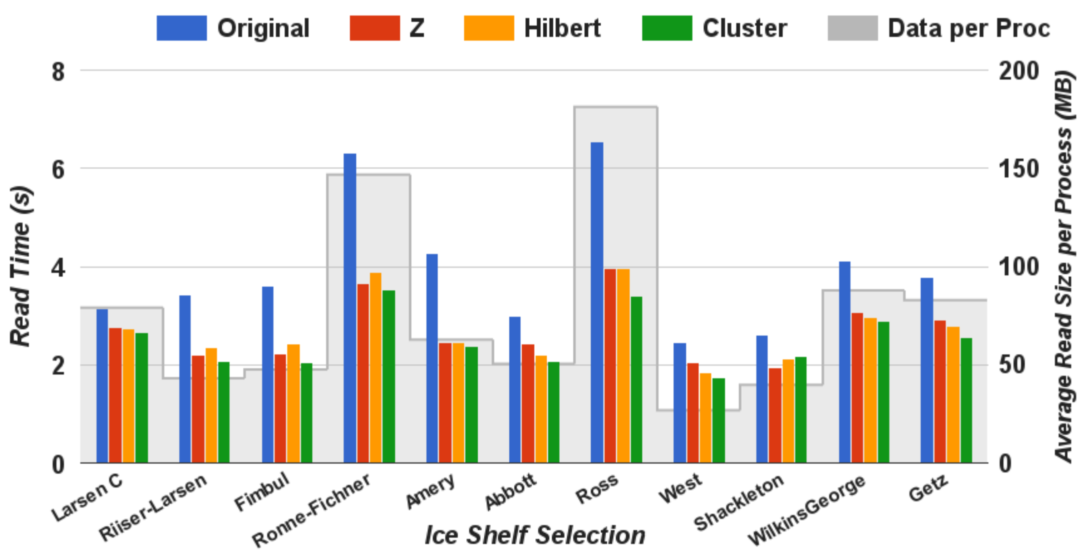


Figure 4.12: Read time comparison of the different layout strategies using Z-curve, *Hilbert-curve* (selected by framework), and Cluster-based on a climate dataset. The different sub-region selections are based on the regions of major ice shelves of Antarctica. The gray skyline shows the average data read per process (right axis). The total time also includes the construction of AMR structure

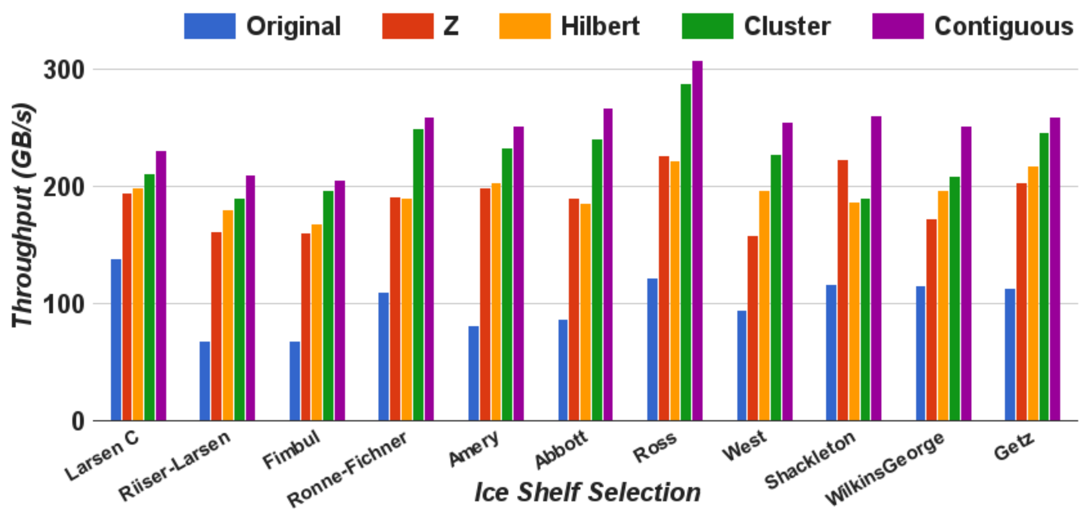


Figure 4.13: The corresponding aggregated data read throughput of the different layout strategies, also compared with reading the same amount of contiguous data. The throughput is calculated by using only the AMR data read time, excluding the AMR construction time

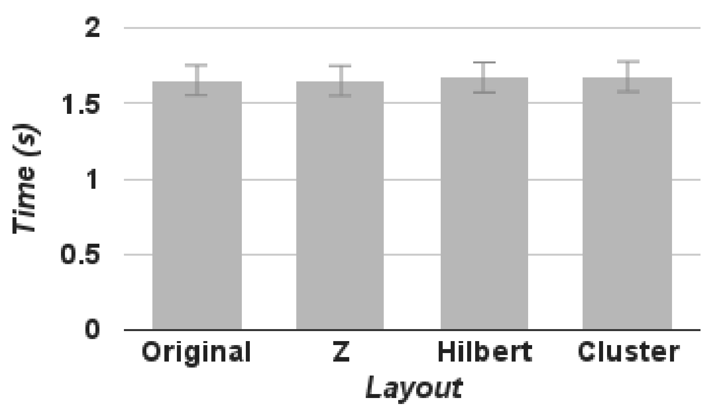


Figure 4.14: Comparison of read times among different layout reorganization approaches when applied to the flow past a cylinder dataset, *original layout* is selected by framework

4.3.5 Read Performance Evaluation - Turbulent Flow

For reading the 2D flow simulation data, our framework does not alter the original layout as the layout selection process determined that none of the other layout reorganization strategies can improve the read performance significantly. Figure 4.14 shows the actual read time of different layouts applied to the datasets. The four different layout reorganization techniques have almost identical read performance, which is expected due to the characteristics of the AMR dataset. This dataset can be regarded as having a single component (though it consists a few disjoint parts), and since the accesses of this dataset does not split the component vertically, the original layout that orders the AMR boxes by column-major order is already optimized for such accesses.

4.3.6 Evaluation using Chombo I/O Benchmark

In our experiments of the Chombo I/O benchmark dataset, we compare the read performance using five different bounding box selections with total selection size ranging from approximately 5% to 25%. To minimize intra-node I/O contention, we use 128 MPI processes (4 from each compute node) to read data concurrently.

Figures 4.15 and 4.16 show the experimental results for reading the generated dataset with different spatial selections. We fixed the number of processes that read the file concurrently and vary the spatial selection size and locations. The total speedups for different selections range from $1.3X$ to $2X$. The cluster-based approach achieves more than 94% of the throughput with contiguous accesses. Note that this throughput is lower compared to the BISICLES dataset in the previous experiments; however, this is expected as fewer processes are used (128 compared to 2480) and Iess amount of data accessed .

4.4 Related Work

The linearization and organization (layout) of the extreme-scale datasets on parallel file systems are crucial to the performance of scientific applications. SciDB [10] addresses sub-volume access patterns by chunking the multi-dimensional scientific data. EDO [82] improves spatial locality through Hilbert SFC. With additional storage, read performance can be improved by creating multiple replicas optimized for specific access patterns, as proposed in [40, 90]. Other works in computer graphics such as out-of-core rendering [22, 91] also address data organization issues. However, new challenges are created with the advent of AMR, making the existing approaches either unsuitable or improvable.

AMR simulation codes are able to dynamically adapt simulation resolution across space and time, which improves efficiency of computational resources drastically while meeting the acceptable error levels for numerical convergence. It has gained its popularity in various domains such as astrophysics [61], climate [21], and chemistry [26]. There are mainly two approaches to implement AMR applications: block-structured [4] and cell-based [41]. Cell-based AMR engines typically use an Octree data structure to handle the tree metadata. In block-structured AMR, the AMR boxes are grouped into a hierarchy of levels of refinement that adapt throughout the computation as the areas of interest move. Each box is defined as an independent computational entity with its own solution storage

The multi-level organization of AMR datasets with multiple boxes of refinement levels add a new dimension of complexity to data access patterns. Another complexity is that the size of the boxes varies, requiring the optimization technique to be adaptive. In addition, these boxes are usually small in size (less than 1MB), potentially resulting in more small non-contiguous read accesses with spatial selection.

Chombo [20] is a block-structured AMR software used by many scientific applications [21,63,83]. Chombo has a general, flexible interface to optimized kernels for rectangular-grid stencil operations, allowing the cost of less efficient irregular operations to be offset by a large number of efficient regular-mesh operations. The finer levels can evolve at a finer computational timestep than the coarser levels. These levels may be distributed across processors for parallel processing. It linearizes the AMR boxes with Lexicographical order based on their coordinates for storage. However, such layout can be inefficient for spatio-temporal read accesses as it could lead to a large number of non-contiguous accesses. For example, reading data of a m by n sub-region of a $2D$ dataset with such layout would result in $O(m)$ non-contiguous accesses. With a properly organized layout, the number of non-contiguous reads could be significantly reduced.

4.5 Summary

In this chapter, towards obtaining efficient spatio-temporal reads from block-structured AMR datasets, which is challenging due to the complex AMR data structure, we propose an in situ AMR data storage layout optimization framework. For selecting the best performing layout, we use a random sampling approach with a performance model for making selection decisions at runtime before writing the data to the storage system. Our framework supports two SFC and a clustering-based approaches to provide read performance improvements for future data accesses. Our evaluations based on read accesses of three AMR datasets show up to $3X$ speedups compared to reading from original layout, and achieve 90% on average of the contiguous read throughput. The extensions of this work include further investigation of the performance model and other layout optimization approaches.

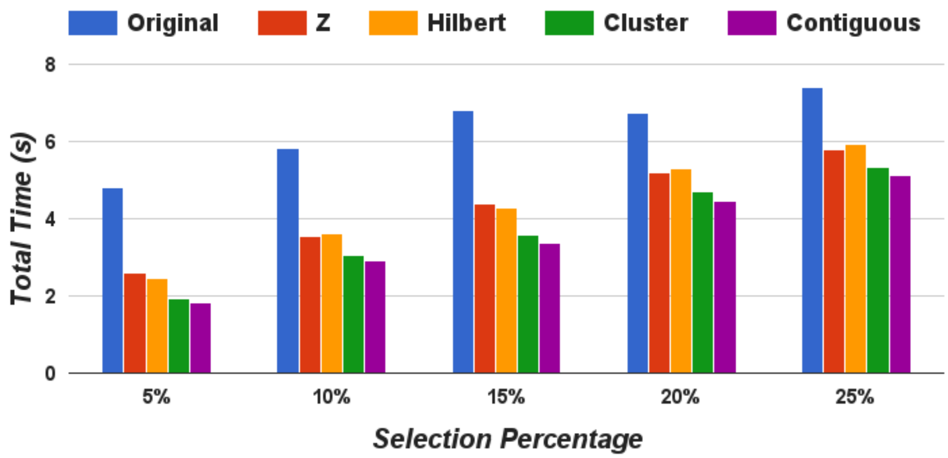


Figure 4.15: Read time comparison of the different layout strategies on a 3D AMR dataset with three levels and total size of 338GB. The total time also includes the construction of AMR structure, *cluster-based* approach is selected by framework

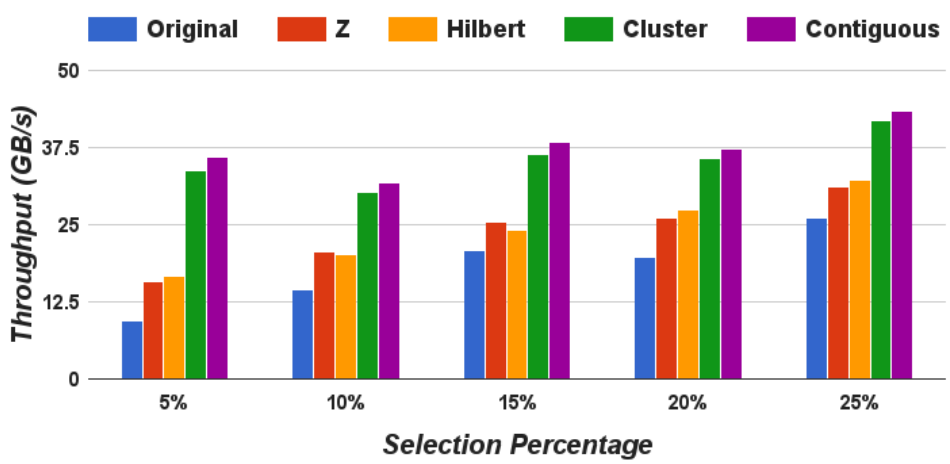


Figure 4.16: The corresponding aggregated data read throughput of the different layout strategies, also compared with reading the same amount of contiguous data. The throughput is calculated by using only the data read time, excluding the AMR construction time

CHAPTER

5

CONCLUSION AND FUTURE WORK

We conclude this dissertation by stating the importance of I/O performance bottleneck and how our research effort has provide solutions on different layers of HPC I/O stack. We present our future research directions in I/O bottleneck identification and optimization, and scalable data movement with deep memory hierarchy.

5.1 Conclusion

I/O performance is crucial to the overall performance of data-intensive analysis and visualization applications in HPC environments. With the advancement towards exascale computing, the volume of data is increasing, as is the performance gap between processors and data storage. To alleviate this I/O bottleneck, we focus on effective and efficient parallel I/O and data management. We have designed and implemented methods including online access pattern analysis, pattern guided prefetching, and data layout optimizations for both uniform grid data and AMR data. These techniques are able to achieve multi-fold data read performance improvements as demonstrated by various applications from different scientific domains, such as plasma physics, climate, and chemistry.

In Chapter 2, to recognize the application's access patterns and utilize them to prefetch data, thereby overlapping computation and I/O and save I/O time, we proposed an online access pattern analyzer that supports both structured and unstructured access patterns with high accuracy and low computation and memory overhead. With the pattern-aware prefetching, our method results in up to 26% run-time reductions on top of less than 5% overhead with both kind of access patterns in 22 benchmark evaluations.

We proposed a framework that selects the most suitable layout among the common layout reorganization techniques based on detected data usage patterns in Chapter 4. It is capable of performing storage-efficient optimizations for heterogeneous patterns from both bounding box and element data selections. On datasets of scientific applications from various domains, our framework yields $1.3X$ to $90X$ time speedup in the plasma physics queries, $1.1X$ to $1.7X$ speedup in the AR detection, $1.4X$ to $8X$ in MSI, $2X$ to $6X$ in ECoG experiments.

In Chapter 4, towards obtaining efficient spatio-temporal reads from block-structured AMR datasets, which is challenging due to the complex AMR data structure, we propose an in situ AMR data storage layout optimization framework. For selecting the best performing layout, we use a random sampling approach with a performance model for making selection decisions at runtime before writing the data to the storage system. Our framework supports two SFC and a clustering-based approaches to provide read performance improvements for future data accesses. Our evaluations based on read accesses of three AMR datasets show up to $3X$ speedups compared to reading from original layout, and achieve 90% on average of the contiguous read throughput. The extensions of this work include further investigation of the performance model and other layout optimization approaches.

5.2 Future Work

Modern computing exhibits several important trends. In the HPC area, the DOE Exascale Computing Initiative will focus on enabling the deployment of exascale computing platforms by 2023. This will cause a hundred-fold increase in sustained performance over existing computing capabilities. The driving technologies, including the emerging heterogeneous, many-core processors, and deep memory hierarchy, to build exascale systems have not been sufficiently explored. Such unprecedented scale of computing power comes with big challenges; for example, data volume will be increased to the PB and EB-level while I/O will continue to challenge applications, effectively limiting the amount of scientific computing that can be performed on HPC systems.

I/O Bottleneck Identification and Optimization. Towards efficient parallel l/O and data management, we plan to further explore and identify the I/O bottlenecks across the HPC parallel I/O software and hardware stack. Our previous work focus mainly on the parallel file system and I/O middleware layer, however, other layers also have their own bottlenecks, new optimization strategies can be developed to enable efficient parallel I/O in exascale computing. In addition, utilizing data usage semantics is another key to I/O optimization. The semantic information in existing scientific storage formats typically include data types , dimensions, and other descriptions. To facilitate the scientific discovery process, analyzing advanced semantic information such as data access patterns and causalities between variables or datasets is an promising way.

Scalable Data Movement with Deep Memory Hierarchy. Existing and future HPC systems continue to employ a deep, hierarchical memory and storage structure to supply fast data access for extreme-scale computing. In addition to standard caches and main RAM memory, deeper layers including non-volatile RAMs, SSDs, and shared disks are present. Such a deep memory hierarchy offers another angle for efficient data movement. We will develop a scalable data movement mechanism which fully exploits the deep memory hierarchy, and explore offline, online adaptive, and dynamic optimizations.

This mechanism uses data layout optimization and prefetching techniques can also be combined with data compression and indexing. Data compression techniques are able to reduce the total amount of transmitted between different memory layers. We will explore more selective compression techniques that are based on the characteristics of different layers. Data indexing has been extensively studied for exploratory scientific data analysis [87]. With the existing indexing technologies, an effective index placement strategy is needed so that scientific applications are able to locate the interesting features of data efficiently.

References

[1] Daniel J Abadi, Daniel S Myers, David J DeWitt, and Samuel R Madden. Materialization strategies in a column-oriented dbms. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 466–475. IEEE, 2007.

[2] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, Karsten Schwan, and Scott Klasky. Just in time: adding value to the io pipelines of high performance applications with jitstaging. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 27–36. ACM, 2011.

[3] J-L Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 176–186. IEEE, 1991.

[4] Marsha J Berger and Phillip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics*, 82(1):64–84, 1989.

[5] David Bernstein, Doron Cohen, and Ari Freund. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the IFIP WG10. 3 working conference on Parallel architectures and compilation techniques*, pages 19–26. IFIP Working Group on Algol, 1995.

[6] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORganization for Self-optimizing Storage Systems. In *FAST*, volume 9, pages 183–196. Citeseer, 2009.

[7] Rajesh Bordawekar, Juan Miguel Del Rosario, and Alok Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 452–461. ACM, 1993.

[8] Peter J Braam and Rumi Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[9] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skjellum. User's guide to MPICH, a portable implementation of MPI. *Argonne National Laboratory*, 9700:60439–4801, 1995.

[10] Paul G Brown. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[11] Surendra Byna, Yong Chen, Xian-He Sun, Rajeev Thakur, and William Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC 2008*, pages 1–12. IEEE, 2008.

[12] Surendra Byna, Jerry Chou, Oliver Rübel, Homa Karimabadi, et al. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *SC*, page 59. IEEE Computer Society Press, 2012.

[13] Surendra Byna, Michael F Wehner, Kesheng John Wu, et al. Detecting atmospheric rivers in large climate datasets. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, pages 7–14. ACM, 2011.

[14] Toon Calders, Nele Dexters, and Bart Goethals. Mining frequent itemsets in a stream. In *Data Mining, 2007. ICDM 2007. Seventh IEEE International Conference on*, pages 83–92. IEEE, 2007.

[15] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM TOC*, pages 8:1–8:26, October 2011.

[16] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *Cluster'10*, pages 1–10, 2010.

[17] Jacqueline H Chen, Alok Choudhary, B De Supinski, M DeVries, ER Hawkes, S Klasky, WK Liao, KL Ma, J Mellor-Crummey, N Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):015001, 2009.

[18] Avery Ching, Alok Choudhary, Wei-keng Liao, Robert Ross, and William Gropp. Efficient structured data access in parallel file systems. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 326–335. IEEE, 2003.

[19] Jong Youl Choi, Hasan Abbasi, David Pugmire, Norbert Podhorszki, Scott Klasky, Cristian Capdevila, Manish Parashar, Matthew Wolf, Judy Qiu, and Geoffrey Fox. Mining hidden mixture context with adios-p to improve predictive pre-fetcher accuracy. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–8. IEEE, 2012.

[20] P Colella, DT Graves, TJ Ligocki, DF Martin, D Modiano, DB Serafini, and B Van Straalen. Chombo software package for AMR applications-design document, 2000.

[21] Stephen L Cornford, Daniel F Martin, Daniel T Graves, Douglas F Ranken, et al. Adaptive mesh, finite volume modeling of marine ice sheets. *Journal of Computational Physics*, 232(1):529–549, 2013.

[22] Wagner T Correa, James T Klosowski, and Claudio T Silva. Visibility-based prefetching for interactive out-of-core rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 2. IEEE Computer Society, 2003.

[23] Phyllis E Crandall, Ruth A Aydt, Andrew A Chien, and Daniel A Reed. Input/output characteristics of scalable parallel applications. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 59–59. IEEE, 1995.

[24] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *ICPP 1993*, volume 1, pages 56–63. IEEE, 1993.

[25] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *TPDS*, 6(7):733–746, 1995.

[26] Ralf Deiterding. A parallel adaptive method for simulating shock-induced combustion with detailed chemical kinetics in complex domains. *Computers & Structures*, 87(11):769–783, 2009.

[27] Andrea Di Biagio, Ettore Speziale, and Giovanni Agosta. Exploiting thread-data affinity in openmp with data access patterns. In *Euro-Par 2011*, pages 230–241. Springer, 2011.

[28] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In *USENIX Annual Technical Conference*, volume 7, pages 261–274, 2007.

[29] Bin Dong, Surendra Byna, and Kesheng Wu. SDS: a framework for scientific data services. In *Proceedings of the 8th Parallel Data Storage Workshop*, pages 27–32. ACM, 2013.

[30] Ilya Ganusov and Martin Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 350–360. IEEE, 2005.

[31] Michael Gerndt and Michael Ott. Automatic performance analysis with periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, 2010.

[32] Zhenhuan Gong, DA Boyuka, Xiaocheng Zou, Qing Liu, Norbert Podhorszki, Scott Klasky, Xiaosong Ma, and Nagiza F Samatova. Parlo: Parallel run-time layout optimization for scientific data explorations with heterogeneous access patterns. In *CCGrid 2013*, pages 343–351.

[33] Zhenhuan Gong, Sriram Lakshminarasimhan, John Jenkins, Hemanth Kolla, Stephane Ethier, Jackie Chen, Robert Ross, Scott Klasky, and Nagiza F Samatova. Multi-level layout optimization for efficient spatio-temporal queries on isabela-compressed data. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 873–884. IEEE, 2012.

[34] Zhenhuan Gong, Terry Rogers, John Jenkins, Hemanth Kolla, Stephane Ethier, Jackie Chen, Robert Ross, Scott Klasky, and Nagiza F Samatova. Mloc: Multi-level layout optimization framework for compressed scientific data exploration with heterogeneous access patterns. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 239–248. IEEE, 2012.

[35] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[36] Wook-Shin Han, Yang-Sae Moon, and Kyu-Young Whang. Prefetchguide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbmss. *Information Sciences*, 152:47–61, 2003.

[37] Virginia Herrarte and Ewing Lusk. Studying parallel program behavior with upshot. Technical report, Argonne National Lab., IL (United States), 1991.

[38] Mark Howison. Tuning hdf5 for lustre file systems. In *Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS10),, Heraklion, Crete, Greece, September 24, 2010*, 2012.

[39] John Jenkins, Isha Arkatkar, Sriram Lakshminarasimhan, David A Boyuka II, Eric R Schendel, Neil Shah, Stephane Ethier, Choong-Seock Chang, Jackie Chen, Hemanth Kolla, et al. Alacrity: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems X*, pages 95–114. Springer, 2013.

[40] John Jenkins, Xiaocheng Zou, Houjun Tang, Dries Kimpe, Robert Ross, and Nagiza F Samatova. RADAR: Runtime asymmetric data-access driven scientific data replication. In *Supercomputing*, pages 296–313. Springer, 2014.

[41] Hua Ji, Fue-Sang Lien, and Eugene Yee. A new adaptive mesh refinement data structure with an application to detonation. *Journal of Computational Physics*, 229(23):8981–8993, 2010.

[42] Geoffrey H Kuenning. The design of the seer predictive caching system. In *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on*, pages 37–43. IEEE, 1994.

[43] Geoffrey H Kuenning and Gerald J Popek. *Automated hoarding for mobile computers*, volume 31. ACM, 1997.

[44] Sriram Lakshminarasimhan, David A Boyuka, Saurabh V Pendse, Xiaocheng Zou, John Jenkins, Venkatram Vishwanath, Michael E Papka, and Nagiza F Samatova. Scalable in situ scientific data encoding for analytical query processing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 1–12. ACM, 2013.

[45] Sriram Lakshminarasimhan, John Jenkins, Isha Arkatkar, Zhenhuan Gong, et al. 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*, page 31. ACM, 2011.

[46] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *Euro-Par 2011 Parallel Processing*, pages 366–379. Springer, 2011.

[47] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[48] Zhenmin Li, Zhifeng Chen, Sudarshan M Srinivasan, and Yuanyuan Zhou. C-Miner: Mining Block Correlations in Storage Systems. In *FAST*, pages 173–186, 2004.

[49] Wei-Fen Lin, Steven K Reinhardt, and Doug Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 301–312. IEEE, 2001.

[50] Jialin Liu, Surendra Byna, Bin Dong, Kesheng Wu, and Yong Chen. Model-driven data layout selection for improving read performance. In *IPDPSW*, pages 1708–1716. IEEE, 2014.

[51] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. Six degrees of scientific data: reading patterns for extreme scale science I/O. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 49–60. ACM, 2011.

[52] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible I/O and integration for scientific codes through the adaptable I/O system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.

[53] Tara M Madhyastha, Daniel Reed, et al. Exploiting global input output access pattern classification. In *Supercomputing, ACM/IEEE 1997 Conference*, pages 9–9. IEEE, 1997.

[54] Tara M Madhyastha, Daniel Reed, et al. Learning to classify parallel input/output access patterns. *Parallel and Distributed Systems, IEEE Transactions on*, 13(8):802–813, 2002.

[55] Tara M Madhyastha and Daniel A Reed. Input/output access pattern classification using hidden markov models. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 57–67. ACM, 1997.

[56] Bernd Mohr and Felix Wolf. Kojak–a tool set for automatic performance analysis of parallel programs. In *Euro-Par 2003 Parallel Processing*, pages 1301–1304. Springer, 2003.

[57] Frank Mueller, Xing Wu, Martin Schulz, Bronis R De Supinski, and Todd Gamblin. Scalatrace: tracing, analysis and modeling of hpc codes at scale. In *Applied Parallel and Scientific Computing*, pages 410–418. Springer, 2012.

[58] Beomseok Nam and Alan Sussman. Improving access to multi-dimensional self-describing scientific datasets. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 172–179. IEEE, 2003.

[59] Kyle J Nesbit and James E Smith. Data cache prefetching using a global history buffer. In *Software, IEE Proceedings-*, pages 96–96. IEEE, 2004.

[60] James Oly and Daniel A Reed. Markov model prediction of I/O requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, pages 147–155. ACM, 2002.

[61] Brian W O'shea, Greg Bryan, James Bordner, Michael L Norman, Tom Abel, Robert Harkness, and Alexei Kritsuk. Introducing enzo, an AMR cosmology application. In *Adaptive mesh refinement-theory and applications*, pages 341–349. Springer, 2005.

[62] R Hugo Patterson, Garth A Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. *Informed prefetching and caching*, volume 29. ACM, 1995.

[63] Kalyan S Perumalla, Richard M Fujimoto, Prashant J Thakare, et al. Performance prediction of large-scale parallel discrete event models of physical systems. In *Simulation Conference, 2005 Proceedings of the Winter*, pages 9–pp. IEEE, 2005.

[64] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990.

[65] Robert Ross, Robert Latham, Marc Unangst, and Brent Welch. Parallel I/O in practice, tutorial notes. *SC'08*, 2008.

[66] Robert B Ross, Rajeev Thakur, et al. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[67] Philip C Roth. Characterizing the I/O behavior of scientific applications on the cray xt. In *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*, pages 50–55. ACM, 2007.

[68] Oliver Rübel, Annette Greiner, Shreyas Cholia, Katherine Louie, E Wes Bethel, Trent R Northen, and Benjamin P Bowen. OpenMSI: A high-performance web-based platform for mass spectrometry imaging. *Analytical chemistry*, 85(21):10354–10361, 2013.

[69] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[70] Sunita Sarawagi and Michael Stonebraker. Efficient organization of large multidimensional arrays. In *Data Engineering, 1994. Proceedings. 10th International Conference*, pages 328–336. IEEE, 1994.

[71] Frank B Schmuck and Roger L Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[72] Seetharami Seelam, I-Hsin Chung, Ding-Yong Hong, Hui-Fang Wen, and Hao Yu. Early experiences in application level I/O tracing on blue gene systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008*, pages 1–8. IEEE, 2008.

[73] Neil Shah, Eric R Schendel, Sriram Lakshminarasimhan, Saurabh V Pendse, Terry Rogers, and Nagiza F Samatova. Improving I/O throughput with PRIMACY: Preconditioning ID-Mapper for compressing incompressibility. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 209–219. IEEE, 2012.

[74] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[75] Frank Shorter. *Design and analysis of a performance evaluation standard for parallel file systems*. PhD thesis, Clemson University, 2003.

[76] Huaiming Song, Yanlong Yin, Yong Chen, and Xian-He Sun. A cost-intelligent application-specific data layout scheme for parallel file systems. In *HPDC 2011*, pages 37–48. ACM, 2011.

[77] Houjun Tang, Xiaocheng Zou, John Jenkins, David A Boyuka II, Stephen Ranshous, Dries Kimpe, Scott Klasky, and Nagiza F Samatova. Improving read performance with online access pattern analysis and prefetching. In *Euro-Par*, pages 246–257. Springer, 2014.

[78] Rajeev Thakur and Alok Choudhary. An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming*, 5(4):301–317, 1996.

[79] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[80] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.

[81] The HDF Group. Hierarchical Data Format, version 5, 1997-2015. http://www.hdfgroup.org/HDF5/.

[82] Yuan Tian, Scott Klasky, Hasan Abbasi, Jay Lofstead, Ray Grout, Norbert Podhorszki, Qing Liu, Yandong Wang, and Weikuan Yu. EDO: improving read performance for scientific applications through elastic data organization. In *CLUSTER*, pages 93–102. IEEE, 2011.

[83] D Trebotich, P Colella, GH Miller, A Nonaka, T Marshall, S Gulati, and D Liepmann. A numerical algorithm for complex biological flow in irregular microdevice geometries. In *Technical Proceedings of the 2004 Nanotechnology Conference and Trade Show*, volume 2, pages 470–473, 2004.

[84] WX Wang, Z Lin, WM Tang, WW Lee, S Ethier, JLV Lewandowski, G Rewoldt, TS Hahm, and J Manickam. Gyro-kinetic simulation of global turbulent transport properties in tokamak experiments. *Physics of Plasmas*, 13:092505, 2006.

[85] Kesheng Wu. FastBit: an efficient indexing technology for accelerating data-intensive science. In *Journal of Physics*, volume 16, page 556. IOP Publishing, 2005.

[86] Kesheng Wu, Surendra Byna, Doron Rotem, and Arie Shoshani. Scientific data services: a high-performance I/O system with array semantics. In *Proceedings of the first annual workshop on High performance computing meets databases*, pages 9–12. ACM, 2011.

[87] Kesheng Wu, Ekow Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 24–35. VLDB Endowment, 2004.

[88] Xing Wu, Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C Roth. Probabilistic communication and I/O tracing with deterministic replay at scale. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 196–205. IEEE, 2011.

[89] Chuan-Kai Yang, Tulika Mitra, and Tzi-cker Chiueh. A decoupled architecture for application-specific file prefetching. In *USENIX Annual Technical Conference, FREENIX Track*, pages 157–170, 2002.

[90] Yanlong Yin, Jibing Li, Jun He, Xian-He Sun, and Rajeev Thakur. Pattern-direct and layout-aware replication scheme for parallel I/O systems. In *IPDPS*, pages 345–356. IEEE, 2013.

[91] Sung-Eui Yoon, Peter Lindstrom, Valerio Pascucci, and Dinesh Manocha. Cache-oblivious mesh layouts. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 886–893. ACM, 2005.

[92] Hongfeng Yu, Chaoli Wang, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, (3):45–57, 2010.

[93] Yongen Yu, Douglas H Rudd, Zhiling Lan, Nickolay Y Gnedin, Andrey Kravtsov, and Jingjin Wu. Improving parallel I/O performance of cell-based AMR cosmology applications. In *IPDPS 2012*, pages 933–944. IEEE, 2012.

[94] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with jumpshot. *International Journal of High Performance Computing Applications*, 13(3):277–288, 1999.

[95] Hong-Quan Zhang, Uwe Fey, Bernd R Noack, Michael König, and Helmut Eckelmann. On the transition of the cylinder wake. *Physics of Fluids (1994-present)*, 7(4):779–794, 1995.

[96] Xuechen Zhang and Song Jiang. InterferenceRemoval: removing interference of disk access for MPI programs through data replication. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 223–232. ACM, 2010.

[97] Yifeng Zhu, Hong Jiang, Xiao Qin, Dan Feng, and David R Swanson. Improved read performance in a cost-effective, fault-tolerant parallel virtual file system (ceft-pvfs). In *CCGrid 2003*, pages 730–735. IEEE, 2003.