ABSTRACT

JENKINS, JONATHAN PAUL. End-to-end Noncontiguous Access Pattern Optimization for Extreme-scale Scientific Data Analytics. (Under the direction of Dr. Nagiza F. Samatova.)

In high-performance computing (HPC) environments, numerous factors conspire to make efficiently accessing large-scale scientific data difficult: the continually growing imbalance between compute and I/O capabilities, the data intensive nature of current and future scientific simulations, the distribution of data among many discrete storage locations in parallel filesystems, and the complexity of data access patterns in I/O workloads.

Driven by these observations, two worthwhile goals to consider for the purpose of removing bottlenecks from, and otherwise optimizing, I/O in an HPC context is reducing the size of access through data reduction and reducing the complexity of mapping noncontiguous data accesses to storage through data organization policies in parallel filesystems. Techniques for the former can aid in mitigating the compute-I/O gap, while techniques for the latter can help extract the maximum performance from the underlying storage system.

In this thesis, we propose complementary approaches to reduce both the size and complexity of data accesses to parallel storage, focusing on post-data-generation analysis workloads (i.e., I/O read optimization) across the I/O software stack. Additionally, we develop methodologies to efficiently process noncontiguous data, a common occurrence in HPC data workloads, informed by current architectural trends (e.g., GPUs). For data reduction, we explore techniques of level-of-detail analysis, which aims to reduce read costs for data analysis at the cost of reduced analysis precision. While existing level-of-detail methods, such as hierarchical Z-order sampling and wavelet multiresolution analysis, have proven useful in a number of analysis tasks, they do not provide both hard bounds on data precision and full-context views of the data, both of which are essential for a robust level-of-detail methodology.

Based on these limitations, we present a *precision-based* level-of-detail methodology (APLOD) for scientific floating-point data, which utilizes the floating-point format to provide well-defined I/O-accuracy tradeoffs. Data layout complexity is reduced by a deterministic partitioning of data, with low computational overhead and bounded per-point errors.

The data processing required to implement APLOD induces noncontiguous access patterns, which are ubiquitous in scientific computing, commonly seen in array subvolume accesses, spatio-temporal accesses, etc. Processing these accesses, specifically for I/O, have been explored at various levels of the I/O software stack. For APLOD in particular, we have shown scalable integration with the ADIOS high-level I/O library.

At the middleware level, specifically MPI and MPI-IO, we show the ability to map the APLOD format and data transformations onto an MPI datatypes representation, which enables

the MPI runtime to efficiently communicate noncontiguous data without user intervention. One significant use-case missing from such noncontiguous data processing is the ability to efficiently process data residing in graphics processing unit (GPU) memory. Hence, we develop an MPI datatypes processing algorithm optimized for GPU-resident data, utilizing the massively parallel nature of GPUs. We demonstrate low processing overhead for both regularly-structured data and irregular data, compared to methods based on PCIe direct memory access (DMA).

Finally, reducing the magnitude of accesses is but one piece of the puzzle – complex, noncontiguous access patterns are exceptionally difficult to process effectively, especially at scale. With this problem in mind and spurred on by recent developments and optimization opportunities in HPC storage systems, we explore optimizations for complex access patterns, such as those exhibited by APLOD, in the space between middleware I/O drivers (e.g., MPI) and object-based storage systems (e.g., PVFS). We do this by exploiting direct object-storage semantics to dynamically create partially replicated data optimized for differing access patterns, as well as by leveraging integrated I/O tracing and analysis to make intelligent decisions to drive our replica-based optimizations. Our method is shown to be effective at improving I/O performance for several common noncontiguous access workloads.

End-to-end Noncontiguous Access Pattern Optimization for Extreme-scale Scientific Data Analytics

by
Jonathan Paul Jenkins

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

2013

APPROVED BY:

Dr. Xiaosong Ma	Dr. Kemafor Anyanwu
Dr. David Thuente	Dr. Anatoli V. Melechko

Dr. Nagiza F. Samatova
Chair of Advisory Committee

BIOGRAPHY

Jonathan Jenkins earned his Bachelor of Science in Computer Science from Lafayette College, Easton, PA in 2010. He entered North Carolina State University's Graduate Program in the following fall of 2010 with Dr. Nagiza Samatova as his advisor. During his doctoral study, Jonathan additionally had two research aide appointments at Argonne National Laboratory.

ACKNOWLEDGEMENTS

There are many people that played an integral role in the completion of this dissertation, whether via direct collaboration, guidance roles, technical support or emotional support.

First and foremost, I am eternally grateful to my thesis advisor, Dr. Nagiza Samatova, for supporting me throughout the entirety of my graduate study. It goes without saying that my success as a graduate student is both a direct and indirect result of her guidance, and my future successes in the field of computer science will be a result of the personal and professional growth acheived under her tutelage. Her drive, her students-first policy, and her crystal-clear research and analytical vision is a constant source of inspiration, for which I have learned and benefitted immensely from.

I am greatly thankful of my thesis committee members: Professors Kemafor Ogan, Xiaosong Ma, Anatoli Melechko, and David Thuente, for their valuable comments and support that have been very helpful in completing and improving upon this thesis. I am further grateful to Dr. Douglas Reeves for admission and financial support upon my entry to the NC State PhD program.

Over the period of this dissertation I have had the pleasure of collaborating with several experts at various institutions in the field of computing. At Argonne National Laboratory, I have collaborated closely with Robert Ross and Dries Kimpe, as well as Robert Latham, Pavan Balaji, Tom Peterka, and James Dinan. Additionally, the completion of my study was performed while at Argonne National Laboratory, with support by the MCS division as well as LCRC. I am also thankful of Scott Klasky at Oak Ridge National Laboratory, C.S. Chang and Stephane Ethier at Princeton Plasma Physics Laboratory, and Jackie Chen and Hemanth Kolla at Sandia National Laboratory for collaboration in various components of this thesis.

I owe a debt of gratitude to Dr. Samatova's research group, for which I had the pleasure of daily collaboration with – both the work presented in this thesis and numerous other works outside the scope of this thesis would not have been possible without their help. Sriram Lakshminarasimhan, David Boyuka, and Eric Schendel each played integral roles in much of this thesis, and in many other projects I have been involved in. I am especially thankful to Xiaocheng Zou and Houjun Tang for the work on RADAR. Additionally, I would like to thank Kanchana Padmanabhan for collaboration on the "Practical Graph Mining with R" book. Finally, I am also grateful for collaborations with Isha Arkatkar, Zhenhuan Gong, and Terry Rogers.

I would like to thank my family for the support that was essential throughout my graduate studies: my mother Susan, father Douglas, my brothers Jeremy, Jeffery, and sister Myra. I am truly blessed to have had their support throughout my life.

Last but not least I am deeply grateful to my girlfriend Jennifer Nunn's support and en-

couragement throughout my graduate career. She has always been there for me through both the best and worst moments of graduate school, and has been a ever-present voice willing me to success. I can think of noone else I would have shared my graduate experience with.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A well-known problem in the HPC community is the ever-increasing rate of data production by large-scale scientific simulations, coupled with diverging rates of growth between dedicated compute resources and shared storage resources [53]. These conditions lead to I/O becoming a limiting factor in many data-intensive applications. This is not only a simulation-time problem, where data to be written is being generated, but also a post-simulation analysis/visualization of data, made especially important by the fact that the data flow for large-scale simulation data is typically write-once, read-many.

To mitigate the performance gap, numerous avenues of research have been explored. On the one hand, architectural advancements such as the usage of SSDs and burst buffers [58] have been closely examined to continue to increase I/O system bandwidth and responsiveness. Additionally, software-based solutions and data workflows have been developed to reduce or even eliminate the need to perform I/O, via *in-situ* and *in-transit* computations which perform analysis operations while data is in cluster memory [61, 125, 123] or by bypassing the filesystem and forwarding data directly from the simulation to dedicated analysis/visualization resources [108]. While having been shown to be effective approaches, not all analyses are able to be done at run-time for practical reasons – data cannot perpetually be in memory, and analyses such as query-driven analysis require user input across a potentially global data context.

On the other hand, a broad range of data transformation and reorganization techniques have been developed to accelerate/ease access to, reduce the footprint of, and/or augment data in storage. There are many examples, such as reducing the storage footprint via compression techniques (both lossy [49, 50] and lossless [85, 10]), indexing the dataset for fast future access [39, 42], focusing on performing such transformations *in-situ* [87, 47], or hybrid combinations of the mentioned methods [48, 30, 29].

In this thesis, we examine data transformations and reorganizations in storage across the I/O software stack in support of optimizing I/O read workloads. Specifically, we focus our work on

the following two problems: 1) read-time data reduction techniques, reducing the I/O necessary to perform accurate data analysis (related but not equivalent to reducing the physical storage size of said data), and 2) generating efficient data distributions in parallel storage based on user workload characteristics. There are numerous challenges in developing such methodologies, with both usability and efficiency concerns:

1. Integration with existing I/O libraries. Data reduction that is fully "out-of-band" would render usage and adoption in applications highly difficult.

2. Transformation speeds capable of *in-situ* usage. Compression libraries such as bzip2 may be too heavyweight for *in-situ* application on scientific datasets (as seen by Schendel et al. [85]) and may not present enough "value-added" to justify large-scale usage.

3. Non-intrusive data modification. Related to the first point, successful data transform and reorganization techniques must attempt to avoid highly disruptive changes in data format; otherwise they risk creating non-trivial interoperability concerns.

1.1 Hypothesis

Given our I/O optimization motivations and in mind of the specific challenges behind our proposed work, we derive the following hypothesis which will be tested throughout our thesis:

For maximum I/O read performance and hence minimum time-to-analysis, compute-efficient, read-time data reduction and distribution must be considered across the entire I/O software stack. Furthermore, techniques must be able to work in tandem to optimize I/O in two non-exclusive domains – reducing the size of data read necessary to perform efficient analysis with guaranteed accuracy bounds, and distributing datasets in such a way that translates complex user access patterns into simple and efficient filesystem access patterns.

1.2 Proposed Approaches

This thesis contributes new approaches that support the given hypothesis. Specifically, our contributions are the following: a fast, bounded-error, lossless level-of-detail methodology, a non-contiguous data processing algorithm capable of quickly packaging noncontiguous data layouts such as that produced by our level-of-detail methodology, and a framework that uses filesystem abstractions, specifically, the object storage abstraction, to build specialized data layouts given user access patterns.

1.2.1 Precision-based Level-of-detail Processing of Big Data

In the context of the growing amount of scientific simulation data, data reduction methodologies are important tools for the computational gains of current and future supercomputers to not be outweighed by the slower rate of growth by I/O subsystems. Compression, data reduction via feature selection at run-time [61, 83, 92, 123], and other methods are active areas of research; one methodology in particular that focuses on analysis-time data reduction is *level-of-detail* analysis, which refers to analyzing a subset or approximation of a full context dataset. Level-of-detail analysis allows scientists to trade off data precision and accuracy for a smaller data footprint, improving I/O performance.

Problems and Challenges

For extreme-scale data and simulations, there are a number of restrictions necessitating new level-of-detail methodologies. The most important of these is *bounded per-point error*, to provide a guarantee of data accuracy – sampling-based analysis and wavelet multiresolution analysis (MRA) do not provide this guarantee. Furthermore, post-processing large-scale data is increasingly viewed as undesirable, as evidenced by the growth of *in-situ*, or simulation-time, data processing methodologies and frameworks, such as DataTap/DataStager [3] and PreDatA [126]. Since in-situ support is necessary, a *low-overhead data transform* is required. Finally, to aid in integration as well as to ensure communication and I/O efficiency, the data layout of the level-of-detail method must be *sufficiently simple* so as to *minimally disturb existing communication and I/O patterns* – general lossless compression methods are ill-suited for this.

Previous methodologies do not map well to this restricted problem space. Sampling-based level-of-detail methodologies, whether statistical database sampling [16, 32, 72, 75] or spatial sampling through data layouts such as the Z-order space-filling curve [78], do not provide a full-context view of the data, and may lose small features or sharp transitions in the data. Wavelet (MRA) [23], used heavily in image-processing and included in standards such as JPEG 2000 [99], provide strong average errors, but do not provide per-datum bounds on error. More generally, data compression techniques designed specifically with scientific data in mind, such as FPC [9, 10], FPZip [57], and ISOBAR [21], change both communication patterns for collective I/O and data layout patterns to produce data chunks not aligned to disk stripe boundaries.

To summarize, our goal is to create a level-of-detail methodology that presents a full-context view of the data with bounded per-point error. The method must additionally incur low computational overhead, and produce a simple data layout that can be easily integrated into existing tools and do not significantly change communication-I/O patterns.

Approach and Results

In order to meet these requirements, we developed an analytics-driven *precision* level-of-detail (APLOD) methodology [41]. Built for floating-point data, APLOD enables configurable level-of-detail access through specification of significant byte boundaries, which we call a *component vector* (CV). Querying data at a parameterized level of precision, along with characteristics defined in the IEEE 754 floating-point format [2], allows us to give exact upper bounds on per-point data accuracy. Furthermore, the deterministic splitting of the input floating-point data means access patterns are not significantly modified.

Initial results, compared against wavelet multiresolution analysis (MRA) using the GNU Scientific Library (GSL) [25], were positive from both a performance and accuracy standpoint. Performance-wise, the rate of transforming to and from the APLOD format is numerous times faster than the GSL implementation (greater than 10x for larger buffers). Accuracy-wise, the strict upper bounds on APLOD error allow analysis errors for numerous simple and more complex operations to scale with the degree of precision, while outliers in the wavelet-reconstructed data prevent the same from occurring in some of the metrics.

Additionally, we demonstrated how to represent the APLOD-formatted data within the popular Adaptable I/O System (ADIOS) [59], which uses a simple XML configuration format to configure data layout. By splitting variables within the XML configuration, the data layout is handled by ADIOS rather than by users, provided that the data passed to ADIOS is APLOD-transformed. For a 4x reduction in data read size via precision level-of-detail, we showed I/O performance to scale at a factor of 3.5–4x.

1.2.2 Representing and Processing Noncontiguous Data Layouts at the Middleware Level with MPI Datatypes

While the simplicity of the APLOD data layout implies that application integration is a simple matter, it nevertheless places an additional data management burden on the user, and is thus undesirable. In the simplest case, each simulation-defined "chunk" of data, whether per-core or some aggregate, can be transformed according to the APLOD CV, and iteratively written to disk. In this case, disk striping patterns, and the communication necessary to write the data to disk would change, though not to the degree that other data processing methodologies such as data compression would. Furthermore, with the increased reliance on GPU technologies such as Nvidia's Compute Unified Device Architecture (CUDA) [71], it is preferable for higher-level abstractions to have a degree of synergy between data located on the GPU and data located on the CPU.

Problems and Challenges

One question that arises is a usability versus performance one. What higher-level abstractions exist that are capable of representing APLOD operations, and what effect do the use of those frameworks have on APLOD performance? For this, we have identified two data movement middlewares that are capable of performing APLOD storage and communication. The first of which, ADIOS, we already discussed in Section 1.2.1. The second is through the Message Passing Interface's (MPI) *derived datatypes* [65, 81].

The usage of MPI derived datatypes allows us to implicitly perform the APLOD transform through the communication layer. Through derived datatypes, which defines a noncontiguous data space to perform communication and I/O, we showed how to map any APLOD format into an equivalent MPI datatype. However, it was shown that the representation resulted in an up-to 10x performance regression, and is hence not suitable for APLOD transformation efficiency.

Despite the poor performance of an MPI-based method, recent computing architecture changes necessitate revisiting the datatypes approach. Specifically, many-core, massively parallel architectures such as graphics processing units (GPUs), having made great strides in programmability, have been deployed in numerous HPC systems. These strides have been seen in the rapid development of many-core APIs and programming languages, such as CUDA, the Open Compute Language (OpenCL) standard [43], and compiler-driven approaches, such as the OpenACC Application Programming Interface [77]. Hence, we must also take into account simulation data residing in GPU space, which currently is distinct from CPU RAM. MPI datatype processing algorithms are inherently serial due to the single-process-per-rank mapping of MPI tasks to processing elements, creating a mismatch between the algorithms and GPU best-practices. While recent research has investigated communicating *contiguous* data in GPU memory through MPI [97, 110], as well as communicating noncontiguous strided data [109] new processing methodologies must be developed in order to package the full gamut of noncontiguous data, and subsequently perform APLOD operations on the GPU.

Approach and Results

To address the lack of efficient noncontiguous data transfer from data in GPU memory, we developed an MPI datatypes processing system which is capable of efficiently processing arbitrary datatypes directly on the GPU. We designed a means for converting conventional datatype representations into a GPU-amenable format, exposing parallelism. Fine-grained, element-level parallelism was then utilized by a GPU kernel to perform in-device packing and unpacking of noncontiguous elements.

As MPI derived datatypes are widely used in applications containing noncontiguous, complex data structures, we performed extensive experimentation across a range of possible data lay-

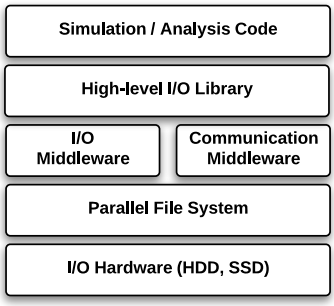


Figure 1.1: I/O Software/Hardware Stack

outs and scenarios, including APLOD evaluation. We demonstrated a several-fold performance improvement for noncontiguous column vectors, 3D array slices, and 4D array subvolumes over CUDA-based alternatives. Compared with optimized, layout-specific implementations, our approach incurs low overhead, while enabling the packing of datatypes that do not have a direct CUDA equivalent. These improvements are demonstrated to translate to significant improvements in end-to-end, GPU-to-GPU communication time. For APLOD in particular, we show that MPI-based solutions, whether within GPU or CPU memory, are not an efficient mapping of computation, necessitating other avenues of optimization.

1.2.3 Dynamic, Filesystem-level Data Layout Optimization

Consider the scientific software stack, shown in Figure 1.1. In general, data pre- and post-processing can be performed at numerous levels: application developers can manually incorporate them into simulations, semantics can be added into the various I/O and communication middlewares that abstract the data processing, and filesystems can optimize access to modified data formats. For example, compression has been integrated into I/O forwarding middleware [86, 115]), requiring no application-level modifications. As discussed, APLOD was originally implemented as a combination of application-level and I/O library-level modifications, showing both I/O scalability and low transform overhead. However, HPC-centric, filesystem-level optimizations, taking advantage of the semantic qualities of distributed storage, have not been explored, leaving a number of research questions (such optimizations have been explored in other contexts; see Section 4.5).

Problems and Challenges

Within the context of APLOD in particular, and analytics-induced noncontiguous access patterns in general, optimizations using parallel filesystem semantics (as opposed to, for example,

POSIX I/O) have not been explored, except indirectly through the typical filesystem tuning process and through I/O library configurations. There are opportunities for optimization at the filesystem level which requires answering pressing research questions. These research directions are driven by the shift towards more decentralized filesystem designs resembling HDFS [91] and GoogleFS [27], in order to continue to scale in terms of cost, reliability, and performance. Most importantly, the primary fault-tolerance mechanism in these systems is data replication. Significant performance improvements can be made by exploiting this form of fault-tolerance, tailoring different replications to different access patterns. From this goal, numerous research questions can be derived:

- What role does access semantics play in enabling access-pattern-based optimization and to what degree do we need to make these semantics available to the filesystem? As an APLOD example, different analysis scenarios require different degrees of data precision, implying different "optimal" layouts for the task at hand.
- How do we capture and optimize access patterns across replications in a *dynamic* fashion? For example, the aggregate access patterns of initial analyses of data may not match those of later analyses. Furthermore, scientists may run many different analysis applications on a single dataset - static methodologies or optimizations driven locally, rather than globally, lack the flexibility needed to handle these cases.
- What effects do data reorganization have on maintaining synchronicity between different copies of the data? If there are non-trivial data transformations, then the modification of these copies of data is also non-trivial.

However, investigating these questions solely within the context of APLOD creates a mismatch in levels of abstraction: enabling and performing optimizations at the filesystem level specific to APLOD would have minimal impact on the larger storage community that is not applicable to other types of data layouts. Hence, we must consider higher-level problem abstractions to optimize using filesystem-level semantics, which can be applied to other problem domains (including APLOD).

Approach and Results

In this work, we investigate the first two research questions, focusing on access patterns induced by multidimensional matrices (the third point is a subject of future work, see Chapter 5.1). We choose multidimensional arrays based on numerous motivating factors. First, the APLOD format can be represented as a two-dimensional array, with one dimension being the precision components. Second, the use of multidimensional matrices is ubiquitous in scientific computing.

Not only can spatio-temporal data be represented as multidimensional matrices, but simulation variables themselves can be represented similarly. In databases, this distinction is made between a *row-store*, where each tuple is stored contiguously, and *column-store*, where each variable within a tuple is stored contiguously.

Given the likely future makeup of I/O systems, the use of data replication in parallel filesystems can be seen in three different contexts: a strict *performance* context, where data can be replicated to increase concurrent bandwidth to the file, a *resilience* context, where data can be recovered and accessible in case of system error (hardware or software), and an *access pattern* context, where different replications can organize data to optimize various methods of access (such as row-major or column-major accesses). We initially focus our optimizations on the use of replications in the latter context, with the former two to be tackled in future work (see Chapter 5.1).

Optimizing these accesses in a dynamic fashion is a difficult problem, one requiring numerous steps. First, for the purpose of prototyping, we decompose the problem into two domains: a filesystem-level data layout scheme and a middleware-level replica management and I/O-driver (implemented as an MPI-IO extension). This allows us to experiment with replica layout policies and apply lessons-learned for future development of more tightly-integrated filesystem semantics with respect to replication. Second, we use the latest experimental developments in the PVFS filesystem [14, 31] to develop a direct, object-level data and replica layout policy, enabling a single container for all related data/metadata, all without modifying the original data layout in physical storage. Third, we develop an integrated tracer for MPI-IO capable of tracking collective optimizations, an important use case for I/O workloads, and utilize the IOSig trace analysis algorithm [11] to gather access-pattern information. Finally, we use a *model-driven* approach to determine effective replica layouts for a given set of access-patterns and an *aging mechanism* based on frequency-of-access to prioritize replica decisions.

As discussed, we investigate multiple types of volumetric decomposition, showing promising results for the ability of partial replication at the object-storage layer to improve I/O performance based on access pattern detection.

Chapter 2

Byte-precision Level of Detail Processing for Variable Precision Analytics

2.1 Introduction

Data reduction in extreme-scale, scientific simulations is a quickly emerging necessity to reduce current and especially future I/O bottlenecks, as compute performance continues to increase at a far higher rate than I/O performance [54]. I/O bottlenecks are especially pronounced when running analysis on simulation-generated data, a typically read-only process performed numerous times by multiple application scientists, often on dedicated analysis clusters with less computational power than the machines the data was generated on.

In the context of extreme-scale computing, data reduction technologies face a number of unique architectural, algorithmic, and application-specific challenges which complicate the emergence of an efficient solution that is highly applicable across application contexts. First, scientific data is notoriously hard-to-compress, due to the utilization of double-precision floating-point variables. These variables tend to have highly entropic mantissa bits, leading to data reduction only on the order of $10 - 30\%$. Achieving higher compression ratios with lossless compression methods require the discovery of non-trivial patterns within the typically spatio-temporal data, making these methods unsuitable for *in-situ* processing. While state-of-the-art lossless compression utilities such as ISOBAR [21] and FPC [10] have been making headway into fast lossless compression of scientific data, achieving high degrees of data reduction while retaining full precision is still best suited to a post-processing scenario, where a full-context approach is possible.

Lossy compression, on the other hand, can greatly increase the compression ratio, making

it more suitable for alleviating I/O bottlenecks *in-situ*. However, application scientists spend an enormous amount of effort ensuring accurate and precise simulation results. While the loss of precision may be acceptable for some simulations, it will not be for all.

Regardless of method, compression as a data reduction strategy introduces new, less optimal access patterns to the I/O system, on both a software and hardware level. On the software level, non-uniform compressed buffer sizes would necessitate global communication between I/O nodes, introducing additional latency costs. On the hardware level, applications optimized for certain striping patterns may lose performance due to the non-uniform buffer sizes, leading to disk contention and lost I/O bandwidth.

Given the challenges that data reduction impose for simulation codes, instead of focusing on reducing data at *write-time*, we argue it is highly beneficial to reduce data at *read-time* through partial-precision analytics. The key insight here is that many types of analysis functions may produce acceptably accurate results even with a greatly reduced amount of precision. A common invocation of this principle can be found in *multiresolution analysis* (MRA) of wavelet-compressed data, traditionally used in the graphics and visualization communities. However, wavelet-based MRA has no bounds on errors at any resolution, and is technically not lossless for double-precision data (though it can be in some cases for single-precision data [104]). Furthermore, wavelet compression standards such as JPEG 2000 [99] have been successfully used to compress single-precision climate data [118], but requires the quantization of single-precision floating-point data, something which may not work well on double-precision datasets crossing a wide range of exponent values.

To achieve these ends, we propose a *analytics-driven precision level of detail* (APLOD) preprocessing methodology. Our approach is inspired by the bit-level format of double-precision variables, and the fact that truncation of the mantissa component leads to low, bounded maximum errors based on the number of mantissa bits kept (see Table 2.1). To promote high efficiency as well as application-specific tuning based on the accuracy needs of scientific simulation analyses, we enable a generalized partitioning of double-precision data along byte-boundaries, described by a byte-level *component vector* (CV). In other words, based on user preferences, datasets are partitioned into groups of contiguous significant bytes, such as the most significant two bytes. Datasets are stored contiguously by most significant bytes so that only data at a required level of precision can be loaded into memory. There are numerous benefits to this approach that, to our knowledge, have not been utilized by other analysis-level data-reduction methods:

- APLOD processing enables *configurable* byte-level decompositions, providing simple access to a range of precisions, including full precision, based on application needs. Each degree of precision provides a hard bound on per-point relative error, as opposed to wavelet

MRA.

- APLOD processing minimally disturbs existing parallel I/O access patterns. If I/O patterns in an application are communication-free, then the patterns with APLOD processing are communication free. Buffer sizes are deterministic, given the original data's buffer sizes. The storage barrier to APLOD processing is low, requiring at most tweaking to disk striping parameters.

- APLOD processing is a low overhead operation in both the *shuffling* of a double-precision buffer to the decomposition defined by a CV as well as the *reconstruction* of the partitioned data back into original (or truncated) form. Even for the finest grain decomposition, the transform operations achieve a throughput of 600MB/s. Wavelet transforms, however, perform at a maximum of 434MB/s, which degrades significantly for very large buffers. When reconstructing partial-precision data from a packed significant byte representation, transform time is decreased in direct proportion with the data reduction. On a per-core basis, the transform throughput far exceeds hard disk bandwidth, making APLOD processing an ideal candidate for *in-situ* integration with applications.

- APLOD processing is orthogonal to existing data reduction and I/O optimization methods. Lossless compression can be applied to each byte-component (oftentimes resulting in higher compression ratios [86]), and I/O optimizations exploiting data layout patterns need not be significantly changed to incorporate APLOD layout changes. Since wavelet MRA has stricter data layout requirements, applying complex layout optimizations to wavelet-transformed data can be nontrivial.

- The programming overhead required to express APLOD operations is minimal and simple to express using well-known I/O libraries, such as ADIOS [59].

This paper is organized as follows. First, we discuss related works in Section 2.2. Next, we discuss the transformation methodology, including different ways of expressing the operations (manually or using MPI datatypes), as well as simple ways of expressing the I/O operations through ADIOS, in Section 2.3. In Section 2.4, we evaluate the methodology in a number of benchmarks, including ADIOS I/O performance with and without partial precision decompositions enabled (Section 2.4.1) and transform overhead for both manually coded and MPI-datatypes-based APLOD representations (Section 2.4.2). Finally, while we do not provide an exhaustive or theoretical treatment of analysis algorithm accuracy, we empirically test a number of algorithms for varying precisions on real-world large-scale scientific applications GTS [111], S3D [17], and XGC-1 [82], in Section 2.4.3.

Table 2.1: Maximum per-point percent errors on partial-precision IEEE 754 doubles, masking the remaining bytes with the quantity 0x7F...FF.

Significant Bytes	Max Underest. Error	Max Overest. Error
2	-1.5e0%	3.1e0%
3	-6.1e-3%	1.2e-2%
4	-2.4e-5%	4.8e-5%
5	-9.3e-8%	1.9e-7%
6	-3.6e-10%	7.3e-10%
7	-1.4e-14%	2.8e-12%

2.2 Background

The concept of *level of detail processing*, or operating on a subset or approximation of the full context dataset, has been explored in numerous contexts. Examples include statistical sampling techniques, arising from the database community, I/O optimization frameworks in HPC that enable sampling on a spatio-temporal domain, and a wide range of signal and image processing techniques, using transforms such as various families of wavelets that enable MRA. Note that each level of detail processing method discussed is compatible with APLOD processing through applying the operations on each significant byte component, though seek costs, effects on wavelet accuracy, etc. need to be considered from the partitioning of byte-components.

Sampling in databases is a well-studied area, with much early work on general sampling queries [75, 72], sampling on an index [73], and sampling in spatial databases [74]. More recent work has also treated the issue of minimizing sampling error due to data skew [16, 32]. Database sampling methods typically do not change the underlying data layout, leading to a large number of seeks, a problem in HPC environments, where seeking in parallel file systems is a high-latency operation. Furthermore, statistical sampling runs the risk of losing small features or sharp transitions in the data, especially since most scientific datasets exist in some spatio-temporal domain. Finally, statistical sampling, such as random sampling, provides for data analysis errors as a distribution, rather than as a bound; while improbable, there is still the chance for significant skew depending on the data distribution.

To tie data access patterns to level of detail processing and achieve high-performance sampling, Pascucci and Frank [78] take a different approach and focus on I/O efficiency instead of statistical rigor. They break the data into progressively coarser, mutually exclusive subsets according to the Z-order space filling curve. Each subset is stored contiguously, allowing efficient access to coarse-grained data subsamples, corresponding to the spatial patterns exhibited in the space-filling curve. By retrieving some number of contiguous layers of the hierarchy, data can be returned at a particular sampling rate with only a single sequential read operation. Related methods in the context of visualization are described elsewhere [113, 51], and also use hierarchi-

cal data layouts. This method also provides a speed/accuracy tradeoff for data analysis, but has some potential drawbacks. Aside from the mentioned statistical sampling problems, the fixed sampling procedure of hierarchical data layouts is prone to selection bias, though for analysis operations such as visualization this is less of an issue.

Discrete wavelet transforms and MRA [23] are particularly popular for level of detail processing, commonly used in image standards such as JPEG 2000 [99] for high compression ratios while maintaining image integrity. In general, discrete wavelet transforms convert an input signal into detail coefficients (from a *high-pass filter*) and approximation coefficients (from a *low-pass filter*), recursively transforming the approximation coefficients. The hierarchical nature of the transform allows reconstruction of the original signal from a subset of the approximation coefficients with high accuracy. Thus, wavelet MRA is the closest applicable method to our proposed APLOD processing. However, while low average errors from MRA are shown (see Section 2.4), there is no upper bound on the errors, and accuracy depends on spatio-temporal relationships in the data. Compared to APLOD, wavelet MRA allows for a much finer-grained decomposition of data. However, at least for the double-precision datasets we test on, using even one-half of the coefficients leads to unacceptable errors, rendering this capability mostly ineffective with such datasets.

2.3 Methodology

As mentioned, the goal of enabling efficient variable-precision analytics requires a modified data representation, one that is capable of being queried by varying degrees of precision. The reduced precision format should directly result in reduced I/O costs; otherwise, there is little tangible benefit to using such a representation aside from perhaps a reduced memory footprint. Furthermore, overhead for performing the transformation to and from the data representation must be small enough to not bottleneck the application.

Based on these restrictions and differing analysis needs of applications, we define a simple, parameterized data decomposition model based on *component vectors* (CVs). We treat a buffer of data as a matrix of bytes, then define column slices (components) based on significant byte boundaries. These slices are transformed to occupy a contiguous buffer, which can then be read into memory separate from other levels of precision. CVs define these column slices, allowing users to choose a data decomposition that makes sense for their analysis needs.

The basis for the component vector is the representation of double-precision floating point data in memory. Recall that the IEEE 754 standard [2] represents double-precision values as a single sign bit, 11 exponent bits and 52 mantissa bits (see Figure 2.1). The value represented

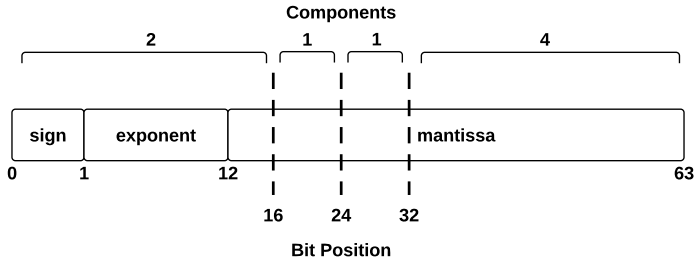


Figure 2.1: The partitioning of a IEEE 754 double-precision value by the CV $\{2, 1, 1, 4\}$.

by double-precision data (not including subnormal values) is given by:

$$\text{value} = (-1)^s \times (1 + \sum_{i=1}^{52} (m_i 2^{-i})) \times 2^{e-1023}, \tag{2.1}$$

where s is the value of the sign bit, m_i is each mantissa bit in decreasing order of significance, and e is the unsigned integer interpretation of the exponent bits.

Given this representation, we make two observations that drive our partial-precision representation. First, we observe that the mantissa bits represent a fractional component in the overall value. That is, the entire mantissa component with the implicit one bit, between the minimum (of all zeroes) and maximum (of all ones), represents a factor in the resulting double-precision value in the range $[1, 2)$. Second, we observe that each less significant mantissa bit contributes an exponentially smaller amount to this multiplier. Given Equation 2.1, we can easily bound the effects of truncation or replacement of the mantissa bits, the results of which can be seen in Table 2.1.

Combined with the use of the exponent bits, truncation of the less significant mantissa bits thus provides a good opportunity to reduce the amount of data at small error rates. By comparison, truncation is typically not feasible for integer data since the bits encoding the integer are not exponentiated by a set of exponent bits. Given the double-precision format, the smallest byte-wise CV must always contain the exponent portion; otherwise, unacceptably high error rates would be generated. Hence, since we use byte-boundaries for efficiency reasons, the first component in any CV is restricted to be at least two bytes, as shown in Figure 2.1. As shown in Table 2.1, truncating to the most significant two bytes, along with masking the discarded mantissa bits, leads to a maximum relative error of 3.1%.

Given the definition of the byte-level partitioning of double-precision data, a high level system overview is given in Figure 2.2. An application defines a CV suitable for its partial-precision analysis needs, the original data is *shuffled* into the new representation and written to disk. At analysis time, users read partial (or full) precision data on a per-component basis,

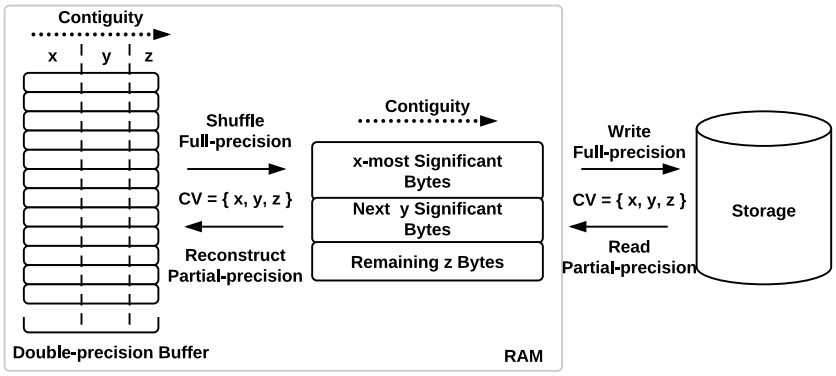


Figure 2.2: Byte-precision level of detail partitioning, based on a generic component vector (CV) splitting groups of significant bytes.

according to the needed level of precision. If necessary, the data is then *reconstructed* back to the original format, though with the missing significant bytes masked with an appropriate value to reduce the average error.

An example CV defined over a double-precision value is shown in Figure 2.1. For this CV ($\{2, 1, 1, 4\}$), the first two bytes of each double-precision value are stored contiguously. As mentioned, these bytes contain the sign bit, all 11 exponent bits and the four most significant mantissa bits. If a buffer of size 8MB was being shuffled, the result would consist of four buffers: a buffer containing 2MB of most significant two bytes, two 1MB buffers of the next two significant bytes, and a 4MB buffer containing the remaining significant bytes.

2.3.1 Component Vector Representation and Operations

The shuffling and reconstructing operators can be defined in two ways, through using the Message Passing Interface (MPI) Standard to specify MPI datatypes and transforming implicitly through MPI communications and I/O routines [65], or as standalone operators. Each component in a CV has a number of semi-regular structures, sharing an element-to-element stride as well as a total number of elements, differing only in the "width" of each datum. These can be simply represented as a set of MPI `vector` types, with `blocklengths`, or number of elements per stride, as the number of bytes in the component. A `vector` type is defined per component, and the set of vectors with necessary offsets are encoded into a single MPI `struct` type, which specifies a set of distinct datatype, offset pairs. Figure 2.3 shows the overall type for an example CV. Given the three components X, Y, and Z, three MPI `vectors` are used (with possible repetition for components with equal byte-widths), and packaged together as a single MPI `struct`. These types can then be directly used within MPI I/O or communication routines, leaving the MPI library to handle the packaging of the data.

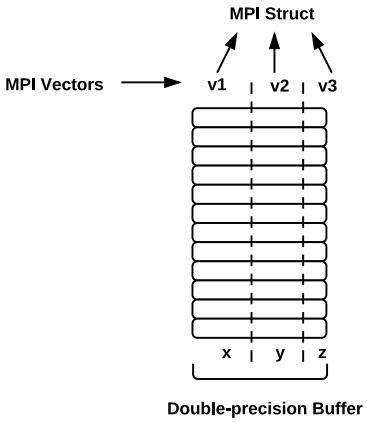


Figure 2.3: Partial-precision level of detail transformation using MPI datatypes, for CV $\{x, y, z\}$.

However, there are a number of issues with a purely MPI-based representation. First, the packing algorithm, which places the non-contiguous data into a contiguous buffer given a datatype specification, may not be efficient compared to a low-level implementation which can take advantage of vectorizing and loop unrolling. See Section 2.4.2. Second, in order to reconstruct at varying degrees of precision, it is necessary to create a different datatype per number of components to load. For example, for the CV $\{2, 1, 1, 4\}$, four MPI `structs` need to be created, one for the first component, one for both the first and second components, and so on. Furthermore, when reconstructing partial-precision data, it is necessary to perform an additional memory setting operation to zero out or mask the significant bytes not loaded in, adding to the overhead. A specialized implementation is able to roll this memory set operation into the reconstruction process.

A manual representation is much simpler, consisting merely of the CV. The shuffling and reconstruction operations can be seen as a generalization of a matrix transpose, where each column is of variable width. This would suggest usage of efficient matrix transpose algorithms. However, in our case, one dimension is always on the order of a few bytes. Such a restricted problem space makes using complex matrix transpose algorithms unnecessary: they break down to following the same access patterns as the naive transpose algorithm. Hence, the naive algorithm is sufficient, consisting of a single loop per component, setting the data in the APLOD format to its correct location in its original format, while taking advantage of component widths and necessary memory setting operations along the way.

```
<!-- original variable -->
<var name="phi"   type="double"          ... />
<!-- variable in APLOD format, CV=2,1,1,4 -->
<var name="phi.c1" type="unsigned short" ... />
<var name="phi.c2" type="unsigned char"  ... />
<var name="phi.c3" type="unsigned char"  ... />
<var name="phi.c4" type="unsigned int"   ... />
```

Figure 2.4: A double-precision variable in the ADIOS XML configuration, in both unmodified and in APLOD format.

2.3.2 Partial-precision I/O

While it is difficult to provide a generic I/O analysis for each simulation code, there exist a number of parallel I/O middlewares that abstract file storage details from the application while ensuring high performance. A particularly popular I/O abstraction is the Adaptable I/O System (ADIOS) [59]. ADIOS is a state-of-the-art componentization of the I/O system that, with a simple change to an entry in an XML configuration file, changes codes to use numerous I/O backends, called "transports," without requiring application recompilation. For example, POSIX, MPI-IO, parallel HDF5 [120], PnetCDF [56], and numerous others are supported transports. Given the flexibility of I/O methods to use and the possibility for defining the CV through the XML configuration file, we chose to use ADIOS to show that data reduction through our partial precision reorganization of data can translate directly to improved I/O costs, and can do so under a wide range of application contexts.

Enabling APLOD processing using ADIOS requires two simple changes. The first change is at the configuration level, where the ADIOS XML configuration file is modified to support the retrieval of partial-precision data. Figure 2.4 shows an example. For each variable being reorganized using APLOD, new variables are created for each CV component to replace the original variable. For multivariate data, there are two ways to place the CV components in the configuration file. The first is to interleave the CV components for each variable, optimizing the access of multiple variables at a particular degree of precision in one fell swoop. The other is to place all CV components consecutively for each variable, optimizing for univariate access of data at various degrees of precision.

The second change required is at the code level, where the I/O operations are modified to handle the partial-precision data. Following the data transform, a write/read is made for each APLOD component, depending on the order described in the previous paragraph, replacing the original I/O calls. Since the ADIOS API fetches variables from file using string identifiers, this process can be automatized using an appropriate naming metric, such as *variable-name.component*, and thus can be hidden behind wrapper functions.

2.4 Experimental Evaluation

To evaluate the APLOD methodology in a leadership-class HPC environment, we perform all experiments on Oak Ridge National Laboratory's Jaguar cluster (Cray XK6 architecture), consisting of a single 16-core AMD Opteron 6200 processor and 32GB of memory per node. For I/O benchmarks, we use ADIOS version 1.3.1 on the Lustre parallel file system.

We look at datasets from a number of real-world simulations. GTS [111] and XGC-1 [82] are both particle-in-cell simulations of nuclear fusion devices, with GTS studying microturbulence in the plasma core and XGC-1 studying microturbulence at the edge. We examine the *potential* (ϕ) variable of a single timestep from GTS and the *temperature* varaible of a single timestep from XGC-1. Finally, we look at S3D [17], a direct numerical simulation of reacting flows in combustion. We examine the *velocity* variable of a single timestep in two dimensions (*uvel* and *vvel*).

As a primary source of comparison, we use wavelet multiresolution analysis. Specifically, we use the D4 Daubechies wavelet provided by the GNU Scientific Library (GSL) [25]. Compared to the other wavelet options available in the GSL (such as Haar and Spline), the D4 wavelet was the most accurate for the datasets used in this paper and had minimal performance differences. For the GTS potential variable, which is linearized from a toroidal structure, we use the one-dimensional wavelet transform. For the other two-and-three-dimensional datasets, we divide the data spatially into 32×32 blocks and use the standard two-dimensional transform on each, which interleaves each level of the transform between rows and columns. For comparison against APLOD reorganization, we then load the most significant wavelet coefficients and reconstruct the data, replacing with zeroes the remaining coefficients.

We perform a number of benchmarks to evaluate our methodology. First, we test whether reading reduced data correlates to lower I/O costs using a parallel read of both unmodified double-precision data and APLOD-reorganized data. Second, we examine the APLOD transform overhead for numerous CV configurations, comparing against an MPI datatypes representation as well as against the D4 transform. Finally, we examine analysis functions of varying degrees of complexity to evaluate the effects of partial precision reconstruction and wavelet MRA on analysis accuracy. Note that, as opposed to reconstructing the data as double-precision variables with a masked mantissa portion, it is possible to reconstruct in single-precision format or even using a fixed-point representation, as proposed by Narayanan [67], to optimize analysis performance, energy efficiency, etc. These methods are complementary to the APLOD methodology, using APLOD reorganization to reduce the I/O costs while using the alternate reconstructions to optimize various metrics. However, to evaluate the effectiveness of our partial precision representation with respect to analysis accuracy, we reconstruct the data in double-precision format to remove any additional effects on accuracy that the alternative representations would

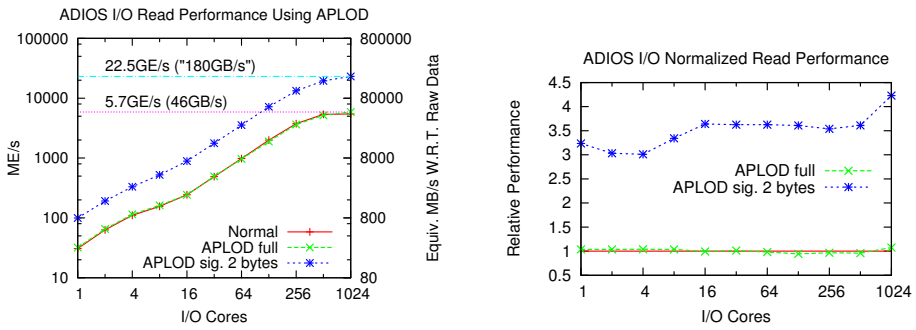


Figure 2.5: Parallel I/O read performance (actual and relative) using ADIOS, with and without APLOD-reorganization. ME/s - millions of elements per second. GE/s - billions of elements per second.

induce, in effect keeping the data in as close to original form as possible.

2.4.1 I/O Performance

Using ADIOS as a driver for our APLOD I/O performance benchmarking, we investigate parallel read performance under a number of scenarios. On a per-reader basis, we chose a partition size of 512MB for two reasons: first, data reading scenarios tend to use many less cores than writing for analysis purposes due to different resource allocation for the respective tasks, and second, we wish to target bandwidth-bound analysis scenarios, and thus minimize the effect of access latency. Furthermore, we chose to test on the CV $\{2, 1, 1, 4\}$, as our accuracy results in Section 2.4.3 indicate that the first four bytes of double-precision variables are sufficient to perform many types of analysis operations, while the latter four bytes are used where full precision is needed (*e.g.*, checkpoint-restart data).

Three scenarios for read performance are shown in Figure 2.5. Our base case is ADIOS performance without data reorganization, which tops out in our tests at about 46GB/s, or 5.7 billion double-precision elements. This is closely matched by the ADIOS read performance of full precision data using reorganized APLOD data with the CV $\{2, 1, 1, 4\}$, providing evidence that the reorganization of data does little to disturb read access patterns at the disk level. However, for manually tuned I/O read patterns, more care would be needed. Finally, we show the effects of reading only the first APLOD component, corresponding to the two most significant bytes. While the time taken by a single compute node (16 cores) is degraded somewhat, increasing the number of cores shows between 3.5 and 4 times the improvement in I/O performance, relative to the number of "elements" read. At 1024 cores, the partial-precision read operation occurs at 22.5 billion elements per second, or a relative "180GB/s".

2.4.2 Transform Performance

For a number of CVs, we test APLOD transformation performance using both our manual implementation and the MPI datatypes representation. For each benchmark, we measure the slowest possible transformation, corresponding to the finest grain CV $\{2, 1, 1, 1, 1, 1, 1\}$, as well as a few others, to show the effect of partitioning with varying degrees of coarseness on transform overhead.

Furthermore, we compare against the one-dimensional transform provided by the GSL for a few reasons. Most notably, the output format of the one-dimensional transform may be directly used for multiresolution analysis and is thus most applicable for comparison against APLOD. The two-dimensional transform, in order to support the contiguous layout of each "level" of the transform, would require additional data reorganization. Combined with the need to transform non-contiguous column data, the overhead becomes unacceptably large.

First, Figure 2.6 shows both full-precision shuffling and reconstruction operations. Asides from very small sizes (less than one KB), the manual implementation at the finest grain CV shows a throughput of 600MB/s for a 1MB buffer, and a maximum of 744MB/s for an 8KB buffer. For coarser grain CVs, a clear performance advantage is seen due to the implementation taking advantage of larger byte widths. At a very coarse grain partitioning (CV $\{4, 4\}$, where the first component represents approximately the level of precision of a single-precision floating point number), the transform is extremely fast, operating with a throughput of up to 2GB/s. For the reconstruction, the throughput experienced a small drop-off due to the data size exceeding available cache space. MPI datatypes in this case (via a call to `MPI_Pack`) are not able to provide the performance of the manual implementation, operating at an approximately order of magnitude lower throughput. The wavelet transform's performance lies between the two APLOD implementations, showing a throughput of 275MB/s for a 1MB buffer and a maximum throughput of 434MB/s for a 2KB buffer. As evidenced in the graph, the performance of the wavelet transform experiences larger regressions than the other methods for larger buffer sizes.

Next, Figure 2.7 shows the throughput of the reconstruction process, in terms of the amount of output data produced. The rate of transformation is increased in proportion to the precision of data transformed. For instance, reconstructing a buffer from the most significant two bytes produces the data at a rate of up to 4.3GB/s. Similar trends can be seen in reconstructing the most significant three and four bytes, including performance increases from using coarse-grained CVs. As with the full-precision shuffling and reconstruction, the MPI packing methodology achieves approximately five to ten times slower performance. Performance metrics for wavelets are not shown here, as the inverse transform performs equivalent operations regardless of the proportion of coefficients used, meaning that wavelet multiresolution analysis only takes advantage of a reduced data representation for I/O operations.

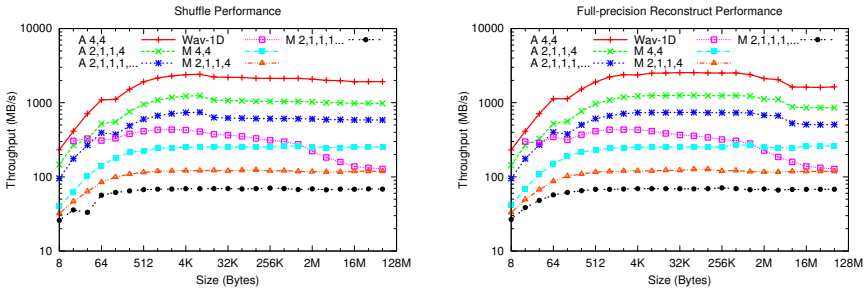


Figure 2.6: Performance of transforming from original data layout to component-level contiguous chunks, and vice versa. *A* - APLOD. *M* - MPI datatypes. *Wav-1D* - one-dimensional wavelet transform.

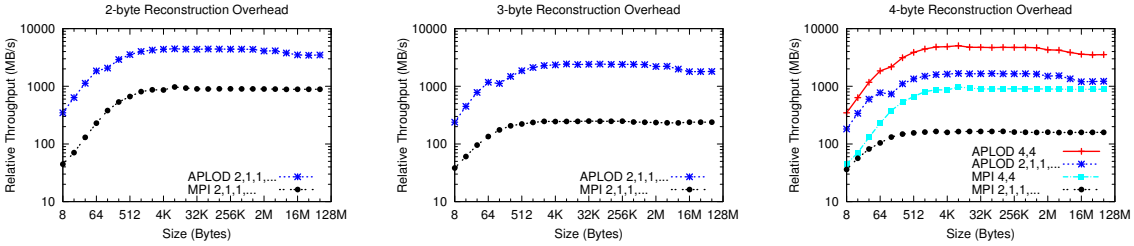


Figure 2.7: Performance of partial-precision value reconstruction.

2.4.3 Partial Precision Analysis Accuracy

Basic Measures

While the benefits of partial precision analysis to I/O costs are clear and intuitive, it must be verified that partial precision analytics is a viable methodology to pursue. More specifically, it is necessary to determine at which levels of precision analysis functions provide adequately accurate results. As the scope of analysis functions on scientific data is much too large for an exhaustive survey, we choose a number of simple and more complex analysis scenarios to determine the potential for large-scale partial-precision analytics.

First, we ensure that the general structure of the data is consistent, which we expect to see from the low maximum errors presented in Table 2.1. To that end, we calculate mean and maximum per-point relative error in the dataset, Pearson correlation, and the relative errors of mean/standard deviation between the original datasets and the partial precision datasets. Each of these metrics are shown in Tables 2.2, 2.3 and 2.4, respectively. These results show that the datasets are roughly equivalent in the aggregate, with low errors for the mean and

Table 2.2: Per-point relative errors (absolute values).

		Per-point relative error (%, absolute)			
		Median		Maximum	
Variable	**Parameter**[1]	**APLOD**	**D4**	**APLOD**	**D4**
potential	2	1.08e0	1.10e1	3.12e0	2.12e6
	3	4.24e-3	-	1.22e-2	-
	4	1.65e-5	4.55e0	4.76e-5	5.57e5
temp	2	1.02e0	1.30e-1	3.12e0	7.10e1
	3	4.24e-3	-	1.21e-2	-
	4	1.63e-5	5.34e-2	4.73e-5	4.08e1
uvel	2	1.08e0	4.44e-2	3.12e0	3.84e5
	3	4.15e-3	-	1.22e-2	-
	4	1.62e-5	1.23e-2	4.77e-5	1.31e5
vvel	2	1.08e0	6.34e-1	3.12e0	1.30e7
	3	4.24e-3	-	1.22e-2	-
	4	1.66e-5	1.87e-1	4.77e-5	2.26e6

[1]Proportion of data used. For APLOD, the number of significant bytes. For the D4 wavelet, the proportion of coefficients brought in for multiresolution analysis, as a fraction of eight.

standard deviation values and near equivalent (1.0) Pearson correlations. Furthermore, the average absolute relative error per-point is low, though the maximum error for two bytes of precision reaches the maximum possible. In comparison, the wavelet-reconstructed data has much more variability. The most significant difference between the D4 wavelet and APLOD is the presence of large relative errors in the wavelet-reconstructed data, that persist even for a large number of coefficients. Also, the errors seen in the wavelet-regenerated data see a lesser degree of change when moving from lower precisions to higher precisions, both due to the "vanishing" nature of the less significant coefficients as well as the persistence of high maximum errors throughout.

Distributional Analysis: k-means and Histogram

Since there is little error between points of data, and that error is distributed across the dataset, the previous metrics were expected to be highly accurate. However, small changes to a variable may change the global behavior of analysis algorithms that, for example, partition the data. In the worst case, edge conditions between the partitions can be sensitive to changes in precision, producing different results. To test the possibility of small local changes yielding large global changes, we performed two experiments on the data: generating an equal-interval histogram based on the data with a constant number of bins, and clustering the data via the k-means algorithm with randomized centroids.

Table 2.5 show the *misclassification rate* of running the k-means algorithm on the *uvel* and *vvel* variables. The misclassification rate is computed by using the same randomly chosen

Table 2.3: Pearson Correlation between full and partial-precision data.

		Pearson Correlation	
Variable	**Parameter**[1]	**APLOD**	**D4**
potential	2	1-8.59e-5	0.972
	3	1-1.34e-9	-
	4	1-0.00e0	0.996
temp	2	1-5.75e-04	0.954
	3	1-9.04e-09	-
	4	1-1.34e-13	0.984
uvel	2	1-6.59e-04	0.992
	3	1-9.97e-09	-
	4	1-4.77e-14	0.998
vvel	2	1-9.27e-05	0.993
	3	1-1.42e-09	-
	4	1-0.00e0	0.998

[1]Proportion of data used. For APLOD, the number of significant bytes. For the D4 wavelet, the proportion of coefficients brought in for multiresolution analysis, as a fraction of eight.

Table 2.4: Partial-precision relative errors for mean and standard deviation.

		Mean Error (%)		Std. Dev. Error (%)	
Variable	**Parameter**[1]	**APLOD**	**D4**	**APLOD**	**D4**
potential	2	3.88e0	1.60e-11	5.53e-2	2.77e0
	3	2.88e-2	-	2.70e-5	-
	4	2.06e-4	9.78-11	1.54e-7	4.24e-01
temp	2	6.87e-2	7.91e-14	2.92e-1	4.64e0
	3	6.46e-6	-	1.87e-4	-
	4	4.57e-8	9.10e-13	1.56e-9	1.64e0
uvel	2	1.90e-2	8.47e-12	9.85e-2	8.16e-1
	3	3.81e-6	-	1.95e-5	-
	4	4.64e-9	1.27e-11	1.39e-8	2.09e-1
vvel	2	4.26e-2	2.68e-12	6.30e-2	6.90e-1
	3	1.04e-5	-	6.80e-6	-
	4	2.30e-8	3.64e-12	5.81e-8	1.55e-1

[1]Proportion of data used. For APLOD, the number of significant bytes. For the D4 wavelet, the proportion of coefficients brought in for multiresolution analysis, as a fraction of eight.

Table 2.5: Clustering errors, measured as the misclassification rate compared to full-precision. The *uvel* and *vvel* variables from S3D are partitioned into 10 clusters.

	K-means Error (%)	
Parameter[1]	**APLOD**	**D4**
2	5.41e0	3.27e0
3	2.52e-1	-
4	1.30e-3	3.31e0

[1]Proportion of data used. For APLOD, the number of significant bytes. For the D4 wavelet, the proportion of coefficients brought in for multiresolution analysis, as a fraction of eight.

centroids for both the full and partial precision data and computing the proportion of partial-precision points assigned to different clusters than the corresponding full precision points. For two bytes of precision, 5.41% of points are assigned to a different cluster than when using full-precision values. These points are likely near the "edges" of the original clusters, which are vulnerable to membership switching through small changes in the cluster centers. The error quickly disappears when using a higher degree of precision, however. Once again, in the wavelet-reconstructed data we see a relatively consistent level of error due to the existence of outliers. At the lowest degree of precision tested, the error is less than that of APLOD, but the persistent presence of outliers regardless of the proportion of wavelet coefficients keeps the errors relatively consistent, while increasing the bytes of precision with APLOD quickly overcomes such errors.

Figure 2.8 superimposes the partial-precision histogram on the full-precision version. We display only the XGC-1 temperature histogram as it is the most illustrative of the trends shown in the datasets considered (S3D has similar patterns, but to a lesser degree, and the exponential component of GTS data differs by far too much for a equal-interval histogram to capture useful distributional information). Note that the selection of data we are using has little difference in the exponential component, meaning that the fractional component is a relatively more important component of the data.

For APLOD-based reconstruction, as Figure 2.8 shows, the problems with losing precision (truncating the six least significant mantissa bytes) are apparent. There is a clear "clustering" effect whereby there is not enough precision to sufficiently separate values into bins when equal-interval binning is used, hence the oscillatory pattern between empty bins and much larger bins. One solution is to use different parameters based on the degree of precision used, which is undesirable; ideally, reduced precision should only introduce noise into the analysis, instead of requiring a revisiting of the analysis function. In any case, with three bytes of precision, these problems all but disappear, suggesting that three bytes is accurate enough to represent this dataset. Data with small ranges that primarily occupy the mantissa portion of the double-precision data are likely to need additional precision.

The wavelet-reconstructed data also has distributional problems, though not of the variety

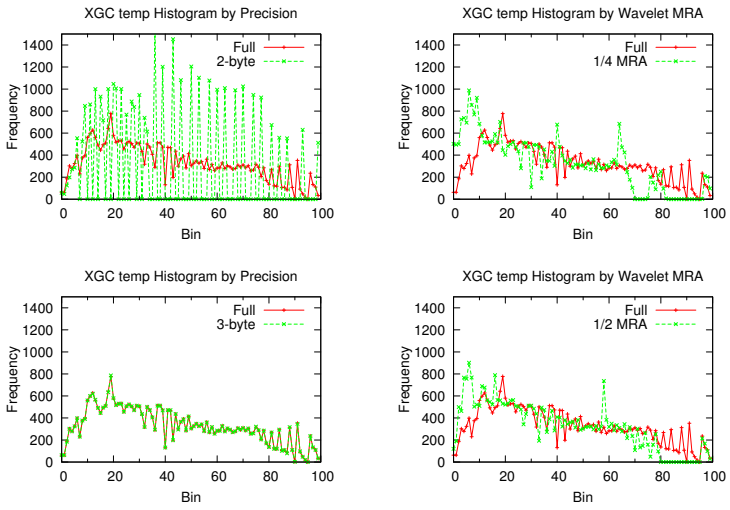


Figure 2.8: XGC-1 100-bin histograms.

that the APLOD data has. Figure 2.8 shows that the existence of outliers shift the range of the data, leading to a different distribution of values among them. However, even if the bins were "shifted" from lower to higher bins in Figure 2.8, skewed results would still appear, as evidenced in the spikes in values compared to the three bytes of precision in APLOD, which matches perfectly against the full-precision data.

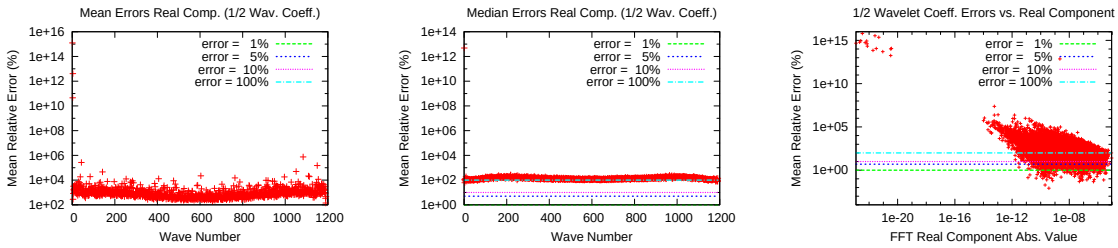
Fourier Analysis Accuracy

Finally, some analysis functions involve transformations of the data into a different space. In particular, many applications rely on signal processing techniques for data analysis, built on transformations such as wavelet and Fourier transforms. The data in this case may be especially prone to *error propagation* since new points of data are produced by sometimes complex operations on the full or subselected data set, which are then analyzed.

For GTS simulations in particular, Fourier Transforms (FFT) of the grid-based electrostatic potential are carried out to analyze the spectral characteristics of turbulence in the fusion core. Gradients in the hot fusion plasma generate so-called "drift waves", which couple with each other through non-linear effects. Charged particles forming the plasma continuously exchange energy with these waves as turbulence develops. Spectral analysis through FFTs allows the identification of the most important modes in the system (i.e., fastest growing modes) and how they couple with each other to form other modes.

To examine the effect of running FFTs on partial-precision data, we perform the transform on the full-precision and partial-precision GTS potential data. With the resulting set of complex

numbers, we generated three metrics for both APLOD and wavelet MRA. First, we organize the data by drift wave number and calculate the mean/median relative error vs. the full-precision-generated FFT data, to examine if there is a distributional relation between the errors. Second, we plot the absolute values of the full-precision FFT coefficients against the corresponding error seen to examine whether there is a relation between exponent value and error. Figures 2.9 and 2.10 show these metrics for the wavelet-generated and APLOD data, respectively, while Table 2.6 gives a numerical distribution of the errors. The real component is shown for these metrics; in comparison, the complex component shows similar trends and the magnitude of the complex numbers shows order-of-magnitude improvements in accuracy.



a *b* *c*

Figure 2.9: For the D4 wavelet, *a,b* - Mean, median errors of FFT data along each drift wave (real component), and *c* - FFT errors plotted against real component value.

Based on these error measurements, we observe two trends. First, the wavelet-generated data, even when using one-half of the wavelet coefficients, shows unacceptable error in all metrics. As noted in the previous sections, we believe this to be the result of outliers skewing the results across every FFT component. Second, errors seen for the APLOD data are much higher for the lowest precision representation than in the previous metrics. The combination of each point in FFT space being a linear combination of all others and the huge differences in the exponent component leads to the propogation of errors. While the largest of outliers are persistent for multiple bytes of precision (where the real and complex component are in the 10^{-20} range), the remaining data approaches acceptable error at four bytes of precision. For this particular application, approximate measures based on aggregate values at wave numbers may be able to use a lesser degree of precision (three bytes).

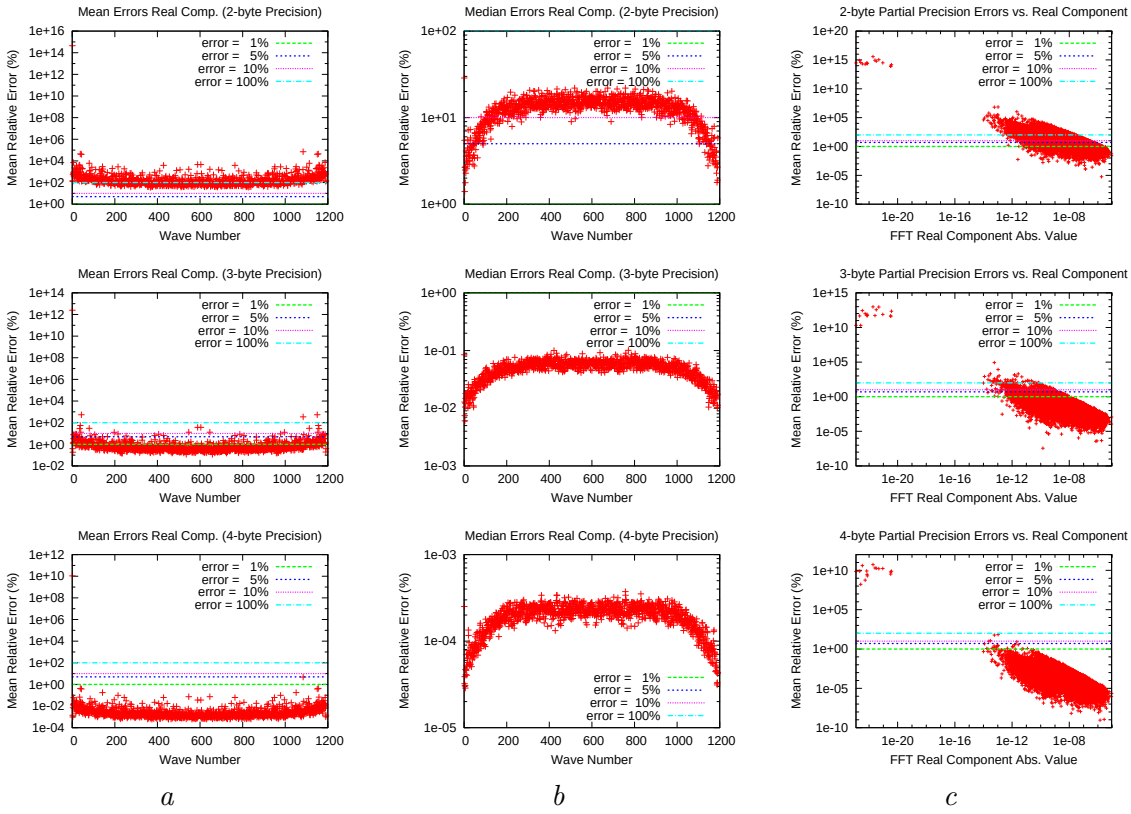


Figure 2.10: For varying APLOD precisions, *a,b* - Mean, median errors of FFT data along each drift wave (real component), and *c* - FFT errors plotted against real component value.

2.5 Conclusion

Extreme-scale scientific applications follow the write-once, read-many paradigm, following a cycle of production-level simulation runs followed by analysis by multiple scientists. Rather than focusing on optimizing the writing of simulation data or reducing it through compression, which may disrupt I/O patterns, we instead chose to reorganize the data to optimize read-level performance by trading off variable data precision for data reduction. Through our method of APLOD processing, this functionality is provided in a simple and low-overhead manner. Perhaps most importantly, it avoids the primary problems of data reduction through compression by providing a globally deterministic partitioning of data that respects existing data layouts and avoids non-uniform buffer sizes. We believe our approach is a low-barrier, high-applicability solution for applications that wish to speed up their analysis processes without sacrificing performance with respect to the original data layout or spend significant effort redesigning I/O and data layout methodologies. Our claim to applicability is strengthened by the use of ADIOS,

Table 2.6: Distribution of relative errors for real, complex, and magnitude components of FFT data generated from the GTS phi data. Total number of points is 191751. *A* refers to APLOD, *W* refers to wavelets, and the number refers to the proportion of the full dataset used (as a fraction of eight).

Comp.	**Rel. Err. (%)**	**A-2**	**A-3**	**A-4**	**W-4**
Real	$x \leq 1$	27912	177681	191558	752
	$1 < x \leq 5$	36193	10235	92	3041
	$5 < x \leq 10$	22306	1688	10	3632
	$10 < x \leq 100$	77202	1789	10	66114
	$100 \leq x$	28138	358	81	118212
Complex	$x \leq 1$	32420	179648	191547	719
	$1 < x \leq 5$	43955	8887	88	2969
	$5 < x \leq 10$	23652	1322	14	3802
	$10 < x \leq 100$	67067	1525	8	65491
	$100 \leq x$	24657	369	94	118770
Magnitude	$x \leq 1$	52841	190283	191751	1445
	$1 < x \leq 5$	60856	1410	0	6086
	$5 < x \leq 10$	26606	44	0	7534
	$10 < x \leq 100$	46830	12	0	122837
	$100 \leq x$	4618	2	0	53849

a widely-used I/O library in the extreme-scale sciences.

Chapter 3

Enabling Fast, Noncontiguous GPU Data Movement in Hybrid MPI+GPU Environments

3.1 Introduction

A great amount of interest in the HPC community has been centered on the capabilities of graphics processing units (GPUs) as inexpensive, many-core accelerators. Evidence of this is seen in recent Top500 lists of supercomputers [1], where GPU accelerators are gaining in popularity due to their effectiveness over a wide range of computational loads and a favorable FLOPs to power ratio.

A number of technical challenges arise from the addition of a fundamentally different computing architecture to existing systems. Aside from the cost of developing, porting, and optimizing codes to run on the GPU, there is a greater concern about integrating them into algorithms with non-trivial point-to-point and collective communication patterns. The currently prevailing GPU accelerator model consists of discrete graphics processing hardware with memory separate from the CPU's RAM. Hence, any communication operation involving data resident in GPU memory requires moving data between GPU and CPU memories, effectively adding another "hop" to the communication graph. Since the MPI Standard [65] does not define MPI's interaction with GPU memory managed by, for example, OpenCL [43] or CUDA [71], the burden of managing distinct memory spaces, especially of non-contiguous communication, falls on the application developers.

Enabling MPI to interact directly with data stored in GPU memory is an important step toward providing transparent and efficient integration of GPUs into HPC applications. Beyond communication operations that transfer contiguous chunks of GPU data, a more general and

challenging problem is the communication of *noncontiguous* data. MPI datatypes enable such communication for data in CPU memory, allowing the programmer to define an arbitrary layout for data to be sent or received (or input/output for MPI I/O operations) using a single MPI operation. A common use of data types in scientific computing is the transfer of noncontiguous array slices from GPU to GPU in applications such as stencil computations, which require array boundary updates (cell exchange) between processes [63, 69, 84].

For the computational benefit of using the GPU to outweigh the cost of data transfer into CPU main memory, these communication operations must be performed with minimal overhead. The naive solutions of transferring point-by-point and transferring the entire noncontiguous buffer to the CPU are unacceptable from a performance point-of-view, suffering from unacceptably high latencies and wasted bandwidth, respectively. The transfer granularity between the GPU and the CPU and over the network must be sufficiently coarse to fully utilize the PCIe bus and network interconnect. To achieve this granularity when working with noncontiguous data, it is necessary to *pack* the data into a contiguous buffer prior to transfer. Effective implementations exist for packing noncontiguous data residing in CPU memory [81], but there exists no generalized implementation of packing noncontiguous data residing within GPU memory that takes advantage of GPU parallelism and memory bandwidth.

In this work, we present the design of an efficient, in-GPU noncontiguous datatype processing system. We focus on NVIDIA's CUDA interface, though techniques presented are broadly applicable across accelerator hardware and programming models. Our approach defines a datatype representation that exposes fine-grain parallelism and utilizes a GPU kernel to leverage this parallelism to accelerate data movement. We demonstrate that our approach is comparable or better at packaging noncontiguous data when compared with CUDA's built-in transfer routines, and has low overhead compared to hand-coded packing kernels. We demonstrate an up-to 700% end-to-end latency improvement for performing large, noncontiguous vector data communication. In addition, our system supports arbitrary datatypes for which, to our knowledge, no equivalent exists. Finally, we evaluate the impact of resource contention for GPU cores and access to the PCIe bus. To realize these design goals, we identify and address three key challenges in enabling efficient processing of noncontiguous MPI datatypes in GPU memory:

1. *Datatype Representation in GPU Memory*: High memory utilization on GPUs requires highly regular, contiguous access patterns exploiting register memory and the small user-controlled cache. Thus, as a first step towards building an efficient packing algorithm, we develop a GPU-optimized serialized datatype representation for arbitrary MPI datatypes in GPU memory, separated into a cacheable, constant-length parameter space, and a variable-length parameter space.

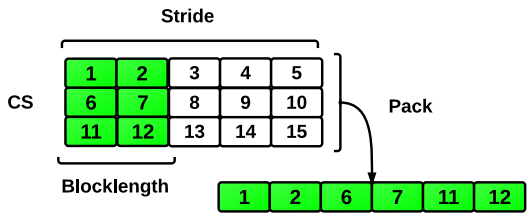


Figure 3.1: Array slice with a width of two elements, an MPI vector datatype `CS` encoding it, and the slice's subsequent packed form.

2. *Parallel GPU Packing Kernel*: An efficient datatype processing algorithm must take advantage of GPU hardware characteristics, such as a fine degree of parallelism and low context-switch overhead. We identify a *fine-grain*, *dependency-free* parallel packing strategy based on canonical datum identification and a traversal algorithm based on the packing strategy and datatype representation.

3. *Packing in the Presence of Resource Contention*: The scheduling policy of GPU kernels and PCIe activity prevents resource sharing to the degree operating systems and CPUs allow; a packing operation could starve in the presence of another resource-intensive kernel. Different communication patterns may necessitate different packing strategies. We identify algorithm patterns for which the packing operation interferes with application performance and provide experimentation showing their effects.

This paper is organized as follows. In Section 3.2 we provide an overview of MPI datatypes and their optimized processing in CPU memory, as well as necessary concepts in efficient GPU algorithm design. Section 3.3.1 discusses the optimization of the datatype representation, while Section 3.3.2 discusses the packing algorithm, given the GPU datatype representation. A detailed evaluation of GPU datatype processing is given in Section 3.4. Finally, we review related work in Section 3.7 and provide concluding remarks and discussion in Section 3.8.

3.2 Background

3.2.1 MPI Datatypes Specification

The Message Passing Interface (MPI) Standard [65] specifies the definition of *datatypes*, allowing users to portably communicate noncontiguous data between processes with minimal effort, increasing productivity while efficiently utilizing network resources. For instance, a noncontiguous column vector can be defined using a simple `vector` type, as shown in Figure 3.1. In this example, the datatype `CS` has a *stride* of five elements and a *blocklength* of two elements.

```
// Specify layout of sender's buffer, a vector of vectors,
//       with base type of double.
// The vector function signature is:
// MPI_Type_vector(count, blocklength, stride,
//       old_type, new_type).
MPI_Datatype CSvec, CS;
MPI_Type_vector(4, 1, 2, MPI_DOUBLE, &CSvec);
MPI_Type_vector(3, 2, 5, CSvec, &CS);
// commit the type description
MPI_Type_commit(&CS);
// perform communication, using intermediate packing
MPI_Send(buffer, 1, CS, ...);
```

Figure 3.2: Defining and communicating a vector-of-vectors.

The stride encodes the distance between consecutive *blocks* while the blocklength encodes the number of datatype children per block.

The most powerful aspect of the datatypes specification is support for *composition*, layering datatypes to create complex selections of data within a simple and concise API. For instance, the "elements" of `CS` could themselves be datatypes such as array subvolumes, and the packing operation would pack, for each "element" of `CS`, the data specified by the datatype. As another example of composition, Figure 3.2 shows a code snippet defining a vector-of-vectors, each with a unique stride, that cannot be easily defined by a single datatype. *Primitive* datatypes, such as integer and floating-point variables, form the basis for *derived* datatypes, such as MPI `vector`s, which can be defined in terms of either primitive or other derived types.

The datatype encodings provided through MPI are driven by the data layout of the application. For example, simulations utilizing arrays commonly use the `vector` or `subarray` types for column vectors or subvolumes. This allows users to define subsets of their data to communicate to other processes, rather than manually preparing the data for communication. The most common datatypes used include a *strided* `vector` of *blocks*, a `subarray` defining an n-dimensional subvolume, an `indexed` set of location-blocklength pairs with a homogenous underlying datatype, and a `struct` consisting of location-blocklength-datatype tuples. A block refers to a contiguous chunk of datatypes, and the *blocklength* refers to the number of "child" datatypes that a block contains.

The definition of datatypes is used within MPI applications to provide a simple interface for the communication of noncontiguous data to/from both disk and other compute elements. However, initiating an I/O operation or a network transfer for each individual piece of data is

expensive and poorly utilizes resources. To address this challenge, MPI implementations *pack* the data into contiguous buffers prior to performing the network or I/O operation.

For noncontiguous datatype processing to be useful in large-scale applications, a simple *datatype representation* must be provided that allows for fast *traversal* of the datatype. Datatype traversal refers to processing the datatype, computing offsets in the input buffer for each primitive defined by the datatypes. The specification of datatypes, while formally described as a list of $\langle$*type*, *displacement*$\rangle$ pairs, can be encoded using a tree structure, where each node in the tree represents a datatype. This structure, as well as necessary parent-child relationships, is captured in the MPICH implementation of *dataloops* [81], which records type-specific parameters and propagates information about datatypes necessary for a simple traversal. Specifically, the information needed for a traversal is the *extent* and *size* of the child datatype, where the extent is the distance between successive child data types and the size is the amount of data encoded by the type, if stored contiguously.

Given an encoding of a noncontiguous datatype, MPICH traverses the datatype by representing the traversal as a buffer-filling operation. MPICH unrolls a depth-first search on the tree structure and uses a concise stack-based representation of the traversal. Each stack element represents type-specific parameters, such as how many `vector` blocks have been traversed. The extent and size at each level of the tree are used to compute offsets from the raw data into the contiguous buffer, and type-specific optimizations can be utilized to prevent revisiting nodes more than necessary, such as substituting specialized memory copy functions for `vector` types.

3.2.2 GPU Architecture and Programming Model

Nvidia's Compute Unified Device Architecture (CUDA) defines a programming abstraction for general purpose computation on GPUs (GPGPUs) [71]. For this paper, we focus on CUDA and Nvidia GPUs, though the concepts and algorithms can be easily applied to other libraries, such as OpenCL.

CUDA presents the GPU as a CPU-driven coprocessor, where the CPU issues asynchronous parallel *kernels* on the GPU. Kernel launches and memory copies between CPU memory and separate GPU memory are performed across the PCIe bus, a high-latency, high-bandwidth operation; and direct memory access (DMA) enables both kernel calls and memory operations to be performed asynchronously.

GPUs have multiple streaming multiprocessors (SMs), each consisting of a multiple scalar processors (SPs), giving hundreds of total available cores for computation at a given time. The threading model provided is *single instruction multiple thread*, or SIMT, which executes a group of threads (a *warp*, typically 32) in lockstep. SIMT, unlike SIMD (single instruction multiple data), allows threads to *diverge* on branch instructions, where each branch is executed serially

until a convergence point is reached. Threads are grouped in three-dimensional grids, or *thread blocks*, where each block is statically allocated register and cache memory and scheduled on an SM. Compared to CPU threads, GPU threads are extremely lightweight and far less powerful, but make up for it in sheer parallelism potential and extremely low context switch overhead.

The main memory in GPUs are optimized for parallel access in large chunks (typically 128B) that are *coalesced* by adjacent threads in a warp; if adjacent threads access adjacent memory, the operations are combined into a single memory transaction. While the main memory is a high-latency, high-bandwidth resource with a small L2 cache, each multiprocessor also contains a fast but small user-controlled scratch cache, called *shared memory*.

Given these components, there are a number of optimization goals when devising GPU algorithms. First, PCIe bus activity should be minimized, because of high latency and transfer rates that pale in comparison to GPU hardware specifications. Second, memory access patterns on the GPU should be regular and exhibit locality with respect to threads. Third, the shared memory space should be used as much as possible to minimize main memory accesses. Additionally, GPU algorithms must exhibit fine grain parallelism so that the hardware can utilize context switching to hide main memory access latency and stalls in the instruction pipeline.

3.2.3 GPU-GPU Communication in MPI Frameworks – MVAPICH

Recently, the MVAPICH team has utilized key developments in recent CUDA frameworks to enable the transparent MPI communication of buffers in GPU memory [110]. Using CUDA unified virtual addressing (UVA), they enable the communication of data in GPU memory space through the existing MPI routines by checking address origin (GPU vs. CPU memory) at runtime. More recently, this functionality has been extended to two-dimensional vector data by utilizing CUDA's built in strided copy routines [109]. Currently, their methodology is based solely on existing CUDA library functions (specifically, one-and-two-dimensional memory copies) and thus cannot be extended to other datatypes; we provide a framework capable of representing and packing arbitrary datatypes built on top of each other. Our methodology can be integrated into their buffer-pool-based framework in a simple manner, however.

3.3 In-GPU Datatype Processing

The communication data flow driving our datatype processing is shown in Figure 3.3, using as an example the `CS` datatype from Figure 3.1. Given a datatype definition, the data is packed within GPU memory using a kernel, then transferred to CPU memory to be communicated. In order to optimize this flow, we organize the datatype representation in GPU memory to be efficiently accessed by GPU threads. Furthermore, we use a packing algorithm that fully utilizes

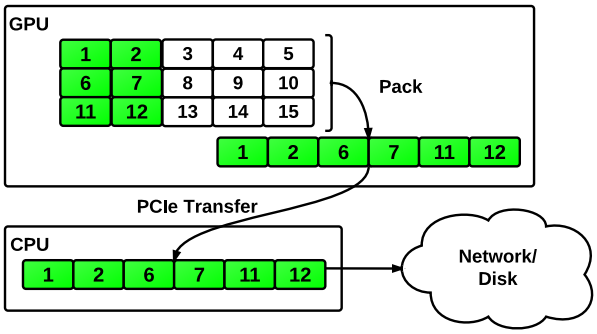


Figure 3.3: Communication pattern necessitating GPU packing. Reversing the arrow directions produces the pattern necessitating GPU unpacking.

GPU threading resources, so that each thread reads a noncontiguous element and places it into contiguous space free of inter-thread dependencies. For illustrative purposes, we assume that `CS` is composed of a second `vector` type `CSvec`, as shown in Figure 3.2. In other words, `CSvec` is a child datatype of `CS`.

3.3.1 MPI Datatype Encoding in GPU Memory

As opposed to the dynamic tree structure that the MPI datatypes specification would imply, GPU best practices suggest storing the type representation contiguously, preferably loading into shared memory once upon kernel invocation. However, many datatypes have a variable-length encoding, such as the `indexed` and `struct` types. This presents a problem because hundreds, if not thousands, of threads may be resident on a single SM, and we cannot assume that the available shared memory is sufficient to store the full type representation for types with variable-length encoding.

Thus, we enforce a cache policy that all GPU threads can benefit from, caching only the fixed-length parameter space of the datatype(s). To facilitate this, the datatype representation is separated into fixed and variable length parameter spaces, using a serialization order corresponding to a preorder traversal of the type tree. With variable-length datatype fields, such as blocklengths and displacements for the `indexed` type, left aside, we observe that the remaining type tree can be stored in shared memory, as each type otherwise requires a small amount of fixed-length memory to encode. This separation creates a *cacheable, fixed-length parameter space* and a *variable-length parameter space*, both residing in GPU memory. See Table 3.1 for a listing of datatypes with their fixed and variable length parameters.

Figure 3.4 shows an example type tree of arbitrary types. As illustrated in the figure, the type tree is traversed preorder, storing the fixed-length parameters contiguously. The variable-length parameters are stored in a separate contiguous buffer, called the *lookaside buffer*. For

Table 3.1: MPI datatypes and their fixed/variable length parameters. The "Common" row contains parameters common to all datatypes in our implementation. The lookaside offset is added to point to the variable type parameters upon serialization.

Type	Fixed	Variable
Common	count size extent child primitives	
`contiguous`		
`vector`	stride blocklength	
`subarray`	dimension lookaside offset	array sizes subarray sizes start offsets
`indexed`	lookaside offset	blocklengths displacements
`indexedblock`	blocklength lookaside offset	displacements
`struct`	lookaside offset	blocklengths displacements child types

each datatype with a variable-length parameter, a pointer to the lookaside buffer is included into the type's fixed-length parameters. We call this the *lookaside offset*. To control traversal and remove the explicit encoding of primitives, a bitfield is used to specify the node type (leaf vs. non-leaf), encoding the primitive type if the node is a leaf (e.g., integer, floating-point). This bitfield is also included in the fixed-length parameter space.

Since the type tree is preorder-serialized, a top-down traversal to a single datum requires no additional linkage information for nearly every type. The only exception is when there are `struct` types with multiple derived datatype children, requiring additional pointers in the struct variable-length parameters to differentiate where in memory the children types are.

For most derived datatypes, the encoding is quite simple. For example, the encoding for `CS` is simply the fixed parameters in rows Common and `vector` in Table 3.1, followed by the same parameters encoding `CSvec`. The resulting representation requires a small number of bytes of storage. A single `indexed` type is equally simple, though different from an implementation point of view. It has a similarly small fixed-length storage size, followed by a potentially large list of blocklengths and displacements, requiring storage in GPU main memory.

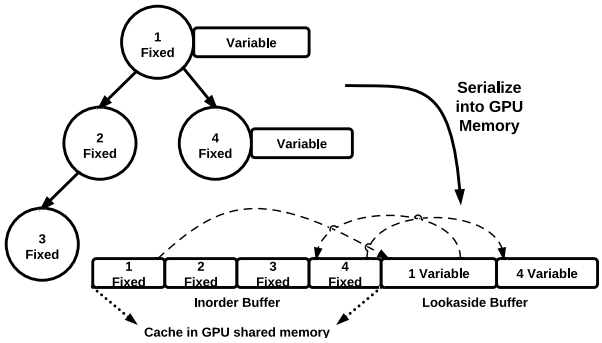


Figure 3.4: Example type tree in CPU memory, separated and serialized preorder into GPU memory by its fixed-and-variable-length parameters. Branches in trees only appear for `struct` types.

3.3.2 Parallel GPU Packing Kernel

There are technical challenges discouraging a straightforward "port" of current CPU-based datatype processing algorithms to the GPU. In particular, CPU-side datatype processing implementations are based on filling buffers from noncontiguous data in CPU memory, leaving a possibility for the coarse-grain parallelism of filling multiple buffers. This runs contrary to best practices on the GPU, where a finer grain of parallelism is critical to performance. Section 3.3.2 addresses the mismatch in parallel packing strategies, while Section 3.3.2 discusses the algorithm itself, based on the parallel processing strategy and optimized datatype representation.

Parallelism via Point-Based Retrieval

Current MPI implementations are focused on efficiently traversing the datatype by size, that is, filling up a fixed-size buffer to be sent over the network. When considering parallelizing the process, such as for data residing on the GPU, current implementations can be parallelized in terms of size. Given a number of buffers and a buffer size, each buffer can be filled in parallel using the information encoded in the dataloops implementation. However, this will not map well to the GPU, as efficient GPU computation necessitates a finer degree of parallelism.

To enable this finer degree of parallelism, we enrich the datatypes encoding with minimal additional knowledge about child datatypes to produce a *dependency-free* parallel traversal. In addition to caching the size and extent of child datatypes, the *number of primitives* can be similarly cached, allowing for fine-grained parallelism on a per-primitive level. When defining types (and thus building the type tree), the number of primitives encoded by a type gets propagated upwards, so that the parent type (e.g., `CS`) records the number of primitives in each instance of the child datatype (e.g., `CSvec`). Using this encoding, we can base the traversal solely

on which primitive to fetch, requiring no additional information besides the type representation. Without this encoding, the type representation can only define the location of a primitive with respect to all previous primitives encoded by the type, which is undesirable for the level of parallelism we are considering.

GPU Datatype Traversal Algorithm

The datatype representation on the GPU, combined with the parallel datatype traversal strategy, yields a straightforward packing algorithm for noncontiguous GPU data with two favorable properties: per-thread storage required, asides from the shared datatype representation, only requires register memory, and there are no inter-thread dependencies and hence no communication or synchronization between threads.

The traversal algorithm assigns a single primitive datum to a thread based on the number of primitives to be packed and traverses the type tree in a top-down fashion, using the datatype's extent, size, and number of child type primitives to update read and write offsets, After the "leaf" derived datatype is encountered, the offsets point to the locations in memory of both the element to pack and where to place it. Algorithm 1 shows the general process. In Line 12, we can change packing to unpacking by merely switching the direction of the read/write. In Line 17, pointer-jumping is only necessary for `struct` types with multiple derived children; see Section 3.3.1. One particular aspect to note is that adjacent threads are assigned adjacent primitives defined by the datatype, so locality between adjacent primitives enables coalesced memory operations on them. Furthermore, on the most common MPI datatypes (`vector`, `subarray`, `blockindexed`), threads experience no branch divergence due to a single code path.

The functions `inc_read` and `inc_write` are type-dependent. Thankfully, they are simple to compute for the `contiguous`, `vector`, `subarray`, and `blockindexed` types, as they have a very regular structure. All but the `subarray` type have an $O(1)$ complexity, and the subarray type has an $O(d)$ complexity, where d is the number of dimensions. The `inc_read` and `inc_write` functions for the `vector` type computation are shown together in Algorithm 2. The general strategy is to compute the block that the primitive resides in, update the offsets appropriately, then "recurse" on the child type.

For the composite types `CS` and `CSvec`, Procedure 3 shows the execution trace of a single thread traversing to its corresponding primitive. One thread is launched for each of the four primitives in the datatype. Note that the execution trace for this type is the same across all threads launched.

For the datatypes with variable-length parameters, such as `indexed`, the process is more nuanced. To avoid performing a per-thread linear scan of of the blocklengths, preprocessing is performed to allow a logarithmic-time binary search. A prefix-sum is performed on the indexed

Algorithm 1: Point-based traversal and packing of arbitrary datatype. Refer to Table 3.1 for fields of the variable `type`.

```
input  : user_buffer: buffer with data to pack
input  : type: serialized datatype, starting at root
input  : ID: element to pack, in canonical order
output : pack_buffer: packed buffer
1  // in, out: location in user/packed buffer, respectively
2  in ← 0, out ← 0
3  Load type fixed-length parameters into cache
4  while true do
5      // increment buffer offsets based on datatype
6      in ← in + inc_read(ID, type)
7      out ← out + inc_write(ID, type)
8      // compute element ID w.r.t. child type
9      ID ← ID % type.elements
10     if type is leaf then
11         // finished processing datatypes, perform r/w
12         pack_buffer [out] ← user_buffer [in]
13         break
14     else
15         // process child type; for non-struct,
16         //   translates to type +=sizeof(type)
17         type ← type.child
```

type's list of blocklengths as a preprocessing step. Then, given a count of n and a list of prefix-summed blocklengths $b_0, b_1, \ldots, b_n$, the terminating condition for thread (primitive) i in the binary search is

$$b_h \leq i/e < b_{h+1} \tag{3.1}$$

where $0 \leq h < n$ and e are the number of elements in the child datatype. The additional b_n term is needed to check the condition at $h = n - 1$.

There is one optimization in particular that we apply to dramatically improve the packing operation. This optimization makes use of the observation that all writes are performed into a contiguous buffer and are thus highly coalesced by adjacent GPU threads. Given this insight, we enable *zero-copy* memory transactions on the GPU. Instead of packing the data into GPU main memory, then performing a bulk copy on the packed buffer, current-generation GPUs can utilize *memory-mapping* of CPU memory into the GPU's memory space. Then, the streaming multiprocessors can, in effect, write directly across the PCIe bus into CPU main memory. Since threads write exactly once and at the end of their traversal, memory mapping is a perfect opportunity to obtain additional performance for minimal effort, by avoiding the GPU main

Algorithm 2: Read/write offset computation for the `vector` type. Refer to the Common and `vector` rows of Table 3.1 for the fields stored in a vector type.

input : type: `vector` datatype
input : ID: primitive to pack, in canonical order
output: in_inc, out_inc: read/write offset increments

```
1  // offset w.r.t. child datatypes
2  count_offset ← ID / type.elements
3  // offset w.r.t. vector blocks
4  block_offset ← count_offset / type.blocklength
5  // for each block, advance by stride bytes
6  // for each child datatype in block, advance by extent
7  in_inc ← block_offset ∗ type.stride + type.extent * (count_offset % type.blocklength)
8  // for each child datatype, advance by child size
9  out_inc ← count_offset ∗ type.size
10 return in_inc, out_inc
```

memory and implicitly pipelining the computational and PCIe loads.

3.3.3 Packing in the Presence of Resource Contention

The methodology for packing was discussed with an underlying assumption of availability of resources and without consideration of scenarios where packing could actually be detrimental to overall performance. For instance, what if a user initiates a send for data residing on the GPU while a fully occupant kernel is running? In the worst case, the scheduling policy of current GPUs, which schedules blocks to run to completion and only allows for architectural reasons a single kernel to be run on each multiprocessor, can easily lead to starvation of a packing kernel. This, in turn, can lead to unacceptably high wait times.

There are a number of communication patterns which could introduce resource contention, centered around concurrently performing communication and other operations. Contention can occur on the computational level, when the communication is performed asynchronously to enable computational overlap. Thus, the packing operation would coincide with that computation. Furthermore, PCIe transfers could be occurring while a communication operation is being performed, such as in CPU-moderated algorithms that follow an iterative setup-compute-collect model. Thus, both PCIe directions could coincide with the packing operation and transfer of results. A combination of these can also occur, such as when multiple users or MPI processes are accessing the same underlying hardware. Communication patterns utilizing global synchronization, such as in stencil codes, will not run into resource contention, however.

With resource contention, the best case occurs when we are working with types such as

Procedure 3: Execution trace of vector-of-vectors traversal for a single thread.

```
   input     : user_buffer: buffer to pack
   input     : ID: thread/datum ID
   output    : pack_buffer: packed buffer
 1 in ← out ← 0
 2 Coordinated load of CS, CSvec into shared memory
 3 type ← CS
 4 Increment in, out using Alg. 2, with ID, type
 5 ID ← ID % type.elements
 6 Is type a leaf type? (no)
 7 Increment type pointer by sizeof (vector type)
 8 //   type ← CSvec
 9 Increment in, out using Alg. 2, with ID, type
10 ID ← ID % type.elements
11 Is type a leaf type? (yes)
12 pack_buffer [out] ← user_buffer [in]
```

`vector` or two- or three-dimensional `subarray`. CUDA and OpenCL allow for the transfer of regularly strided two- and-three-dimensional subarrays, in addition to contiguous buffers, avoiding multiprocessor usage. While useful for the common case of array processing on the GPU, it is nevertheless a special case that cannot be relied on for all applications.

When the datatype is nontrivial and there is resource contention preventing a packing kernel from being run, there are a number of methods that can be used to get the data onto the CPU. The two simplest ones are transferring by extent and transferring point-by-point. Both of these, of course, are highly inefficient. Transferring the entire extent of a datatype, except in cases where the extent and size are approximately equal, wastes bandwidth and still requires packing on the CPU end. Transferring point-by-point suffers from the high latency of initiating copies from the CPU. Both have the potential for interfering with user kernels that rely on host-device transfers. A more intelligent method would involve a hybrid of the two, transferring sections with a low extent-to-size ratio in bulk and transferring point-by-point otherwise. However, such a method would need more complex processing and memory management on the CPU-side and would still have the problems of both methods, albeit reduced in severity. For some types, CUDA and OpenCL allow for the transfer of regularly-strided two-and-three-dimensional subarrays. While this is very useful for the common case of array processing on the GPU, it is nevertheless a special case which cannot be relied on for all applications. Another option is to devote a *persistent kernel* for use by MPI operations and utilize signaling and polling to initiate packing, similar to Stuart *et al.*'s implementation of message-passing on many-core processors [97]. However, since we show latency costs to be

extremely important when performing the packing operation and their method produced an increase in these costs, we do not consider this approach (see Sections 3.4.2 and 3.4.3).

Unfortunately, there is currently no way within the CUDA or OpenCL interfaces to query the level of resource utilization on the GPU, complicating the act of choosing a globally efficient strategy (kernel versus memory-copies) without having application-specific knowledge. Since the overarching goal of this research is to provide transparent GPU data management from within MPI, solutions such as hijacking user kernel calls to collect statistics and infer utilization are, while interesting problems, not addressed by this paper.

3.4 Evaluation - Microbenchmarks

We evaluate our datatypes processing methodology using microbenchmarks of packing performance on numerous MPI datatypes, comparing against appropriate CUDA alternatives as well as optimized type-specific packing kernels. We additionally break down the costs of our packing algorithm, including PCIe and memory operations to and from the input buffer, and give the full context improvement in GPU-to-GPU communication by implementing a ping-pong test on noncontiguous data. Finally, we examine the effects of GPU resource contention on packing and memory copy operations by modifying the issuing order of packing and other operations. For all tests, we used a Nvidia C2050 GPU with version 4.1 of CUDA, connected to an AMD Opteron 6128 at 800MHz. For the communication benchmarks, we used two such nodes, connected by QDR Infiniband, and MVPICH version 1.8. We pin CPU memory used in transfers to enable DMA and enable zero-copy for all datatypes but the `struct` type during packing. For the communication benchmarks, we use two such nodes, connected by QDR Infiniband.

3.4.1 Test Datatypes

To measure the kernel overhead against "bare metal" DMA and provide an upper bound on packing performance, we perform a baseline comparison against the `contiguous` datatype, which can be satisfied with a single memory copy call (`cudaMemcpy`).

To benchmark two-and-three dimensional subarrays such as column vectors, we use a `vector` type, against the CUDA alternative of `cudaMemcpy2D`. We fix the stride between blocks to 512 bytes, which enables maximum performance of the CUDA operation; unaligned arrays greatly hamper CUDA's performance in this regard. Furthermore, we vary the blocklength to experiment the performance implications of block width. For example, if we wanted to transfer the rightmost two columns of a two-dimensional matrix, we would set the blocklength to two doubles, or 16 bytes.

To benchmark array types outside the scope of `vector` representation, we use a four-

dimensional subvolume encoded as a `subarray` type, compared against iterative calls to `cudaMemcpy3D`. We fix the containing volume to be $64 \times 64 \times 64 \times 64$ and pack/transfer an four-dimensional hypercube of increasing size.

To benchmark an `indexed` type, for simplicity, we use the same data format as in our test `vector` type, meaning a constant blocklength and a regular displacement pattern. Other datatypes would be used in practice and be much more efficient, but this benchmark is a reasonable indicator of `indexed` performance; varying blocklengths would cause less divergence than the uniform blocklength, and a regular displacement allows us to control coalescence in a fine-grain manner. For comparison, we transfer the data block-by-block using `cudaMemcpy`.

To show the effect of the `indexed` type's binary search implementation on performance, we additionally evaluate the `indexedblock` type (abbreviated as `idxblock` in the experiments), which is similar to the `indexed` type but has a uniform blocklength, rendering the need to perform a binary search unnecessary. Once again, for simplicity, we use the same data format as the `indexed` and `vector` types. For comparison, we transfer the data block-by-block using `cudaMemcpy`.

Finally, we use a `struct` type to test the effect of thread divergence on writing. We use a simple *C*-style struct consisting of an 8-byte `double`, two 4-byte `ints`, and a `character`, which amounts to 24 bytes with padding. For comparison we copy the extent of each `struct` using `cudaMemcpy`. Furthermore, we disable the use of zero-copy for this type, as the uncoalesced write pattern induced by thread divergence leads to the issuance of a PCIe transaction for each `struct` member, causing significant performance regression.

3.4.2 Noncontiguous Packing Performance

For each datatype presented in Section 3.4.1, we evaluate the general performance of packing from GPU memory into CPU memory, with respect to the size of the packed buffer. Figures 3.5 shows these experiments, comparing against their respective CUDA alternatives. Furthermore, we compare against hand-coded packing routines to test the overhead of our generic packing methodology, shown in Figure 3.6.

A number of interesting trends can be observed for the different datatypes. First, since there is a relatively large gap between the command latency and peak throughput, transfers on the lower KB level are latency-bound, and thus very small absolute differences are seen between the CUDA API calls and the packing kernel. Given the current architecture of discrete GPUs, little can be done to improve these results. Emerging architectures that combine CPU and GPU functionality, such as AMD's Fusion [7], show promise in bridging this performance gap in the future. Furthermore, the latency of issuing kernels is slightly larger than that of issuing memory copies, due to. Thus, the performance of kernel-based packing is adversely affected for smaller

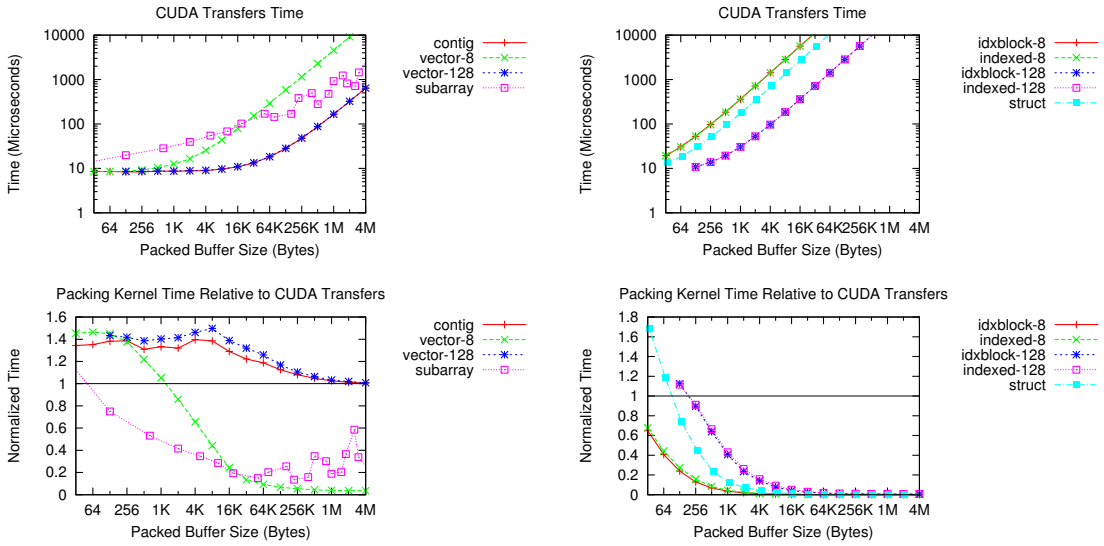


Figure 3.5: Baseline packing time for several MPI datatypes using the CUDA API, and relative performance of packing against CUDA.

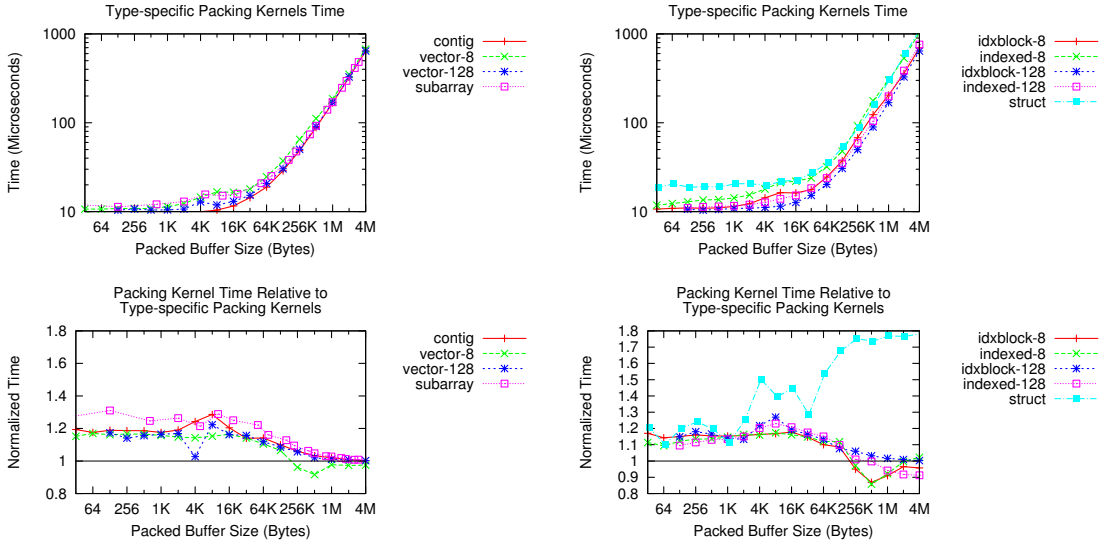


Figure 3.6: Hand coded packing kernel times and relative generalized pack performance.

input sizes, performing worse than the alternative CUDA-based methods (though only on the order of microseconds).

Second, the packing kernel is clearly preferable for types that do not have a CUDA equivalent (e.g. `cudaMemcpy2D`), because of the latency in initiating each blockwise memory copy. Blockwise memory copies, such as for the `indexed` and `indexedblock` types, could compete with the packing kernel only for extremely large block sizes.

For the types that do have a CUDA-equivalent, the results are more nuanced. Aside from latency considerations which affect both methods, performance is largely a function of the data layout: for two-dimensional memory copies, each block must be wide enough to saturate the bus for best performance. For single columns corresponding to a blocklength of 8 bytes, the two-dimensional memory copy performs very poorly, while the packing kernel performs approximately 20 times faster. For a larger number of contiguous columns (16 `doubles` per stride in Figure 3.5), the memory copy outperforms the packing kernel in all cases, especially for small and medium-sized inputs, because of the additional kernel latency. For larger-sized inputs, both the copy and the packing kernel approach the bandwidth limit, so the relative performance difference begins to converge.

The four-dimensional `subarray` type, despite being reasonably mapped to the CUDA API, nevertheless sees major performance improvements when moving to a kernelized packing operation. Since the three-dimensional memory copies must be made iteratively to transfer the entire type, the latency is aggregated through the copies and hurts overall performance.

Compared with type-specific implementations, the generic packing algorithm performs well, with little discernible difference in performance. The differences in normalized performance between the type-specific and generic algorithms are due to the overhead of loading the type representation and instruction overhead from supporting arbitrary type representations. This overhead, however, amounts to between about two and five microseconds for most inputs. The differences in the `struct` implementations are a result of hard-coding the relative location of each `struct` primitive, benefiting from compiler optimization and greatly simplified traversal logic.

The `vector` type is one of the more widely used MPI datatypes, and different parameterizations lead to significantly different performance, so the performance gap in the different `vectors` in Figure 3.5 needs to be further explored. Figure 3.7 fixes the number of blocks in the `vector` and compares the completion times of the packing kernel and the two-dimensional memory copy, varying by the blocklength. As seen in the figure, the performance of CUDA is highly dependent on the blocklength. Blocklengths that are multiples of 32 bytes perform best, but all others experience significant performance regression. Similar performance characteristics are seen when varying the stride parameter, though this is not shown in the paper. Note that an intelligent MPI datatype processing implementation can easily check for these cases, given

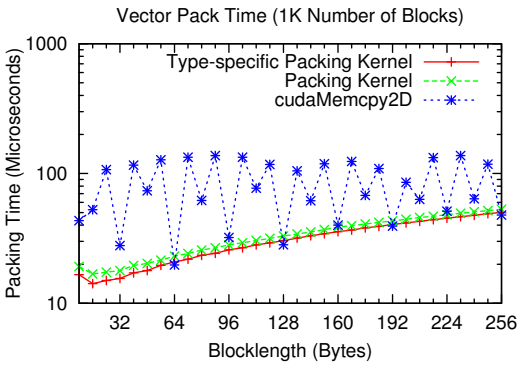


Figure 3.7: `vector` pack performance vs. `cudaMemcpy2D`, with varying blocklengths.

information about the type and hardware configuration.

For three-dimensional arrays in particular, a single `vector` type can be used to send each face of the array: the fully contiguous X-Y face, the contiguous-per-row X-Z face, and the noncontiguous Y-Z face. Together, these operations represent the communication step of a variety of stencil codes, meaning that performance differences from the transfers would be similarly seen within an application context. Table 3.2 shows the transfer rate of each face for different array sizes, using the packing kernel and CUDA's two-dimensional memory copy. The results largely agree with those previously presented; contiguous chunks of data are more effectively transferred using built-in CUDA copies (though there is only an approximately 10-15% difference), while packing is dramatically better for getting non-contiguous data. Note that the CUDA memory copy seems to degrade in performance for the X-Z plane transfer in the $512 \times 512 \times 512$ case. We cannot currently explain this behavior.

3.4.3 Noncontiguous Packing Performance by Component

The performance metrics in Section 3.4.2 give a good overview of the relative performance of the different types, but some information is still missing. For instance, what are the costs of PCIe transfers? What is the effect of memory layout on the overall performance? To answer these questions, Figure 3.8 shows the performance under three contexts: the full context as presented in Section 3.4.2, the completion time of packing into GPU memory (avoiding PCIe transfers), and the datatype traversal time. Note that the packing operations for small-sized messages are latency bound, meaning the issuing of the packing kernel is the dominant cost.

For medium and large-sized messages, the efficiency of the traversal operation is largely dependent on the complexity of the type used. For instance, the `vector` and `contiguous` types, when only traversing the type, complete quickly due to the simplicity of the traversal logic. The `subarray` type, however, suffers in performance due to the additional logic and integer

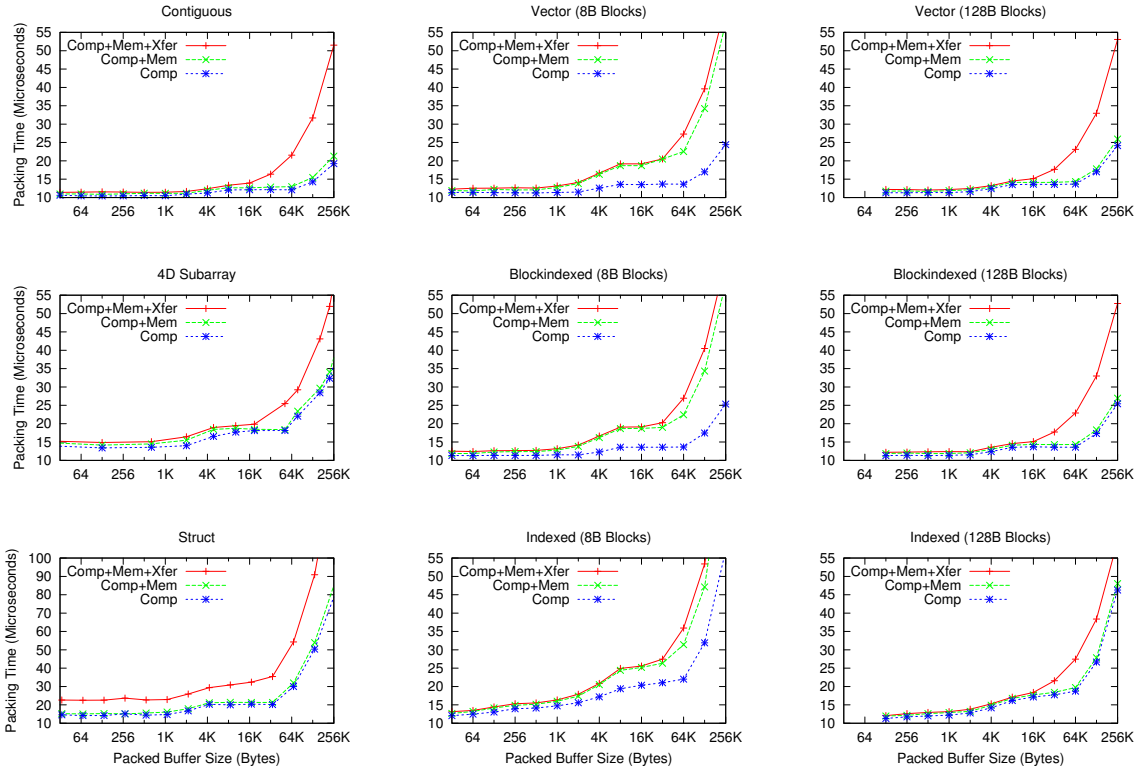


Figure 3.8: Packing time, by component. "Comp" refers to traversing the type, computing input/output offsets. "Mem" refers to performing the read/write operation at the end of the traversal operation. "Xfer" refers to sending the packed data across the PCIe bus.

Table 3.2: Transfer of face of three dimensional matrix of double-precision values to CPU, versus cudaMemcpy2D. X-Y: fully contiguous. X-Z: z sets of x contiguous doubles. Y-Z: fully non-contiguous.

		Throughput (MB/s)	
Size	**Face**	**Pack**	**CUDA**
$64 \times 64 \times 64$	X-Y	923	**1062**
	X-Z	937	**1097**
	Y-Z	**865**	186
$128 \times 128 \times 128$	X-Y	2573	**2854**
	X-Z	2554	**2868**
	Y-Z	**2131**	209
$256 \times 256 \times 256$	X-Y	4567	**4842**
	X-Z	4553	**4845**
	Y-Z	**3728**	216
$512 \times 512 \times 512$	X-Y	5790	**5841**
	X-Z	**5792**	1645
	Y-Z	**4816**	218

computation compared to types such as `vector` necessary to represent and pack a subarray of arbitrary dimension. However, for cases such as a four-dimensional subvolume, multiple `vectors` would have to be used, which would reduce performance, so one cannot merely replace the types and get higher performance.

For types with variable-length parameters, such as `indexed`, the problem becomes memory-bound with respect to the input type, and thus sees lesser performance on the traversal. The indexed type, performing a binary search, still must pay the penalty of accessing GPU main memory for every point retrieved, which is a high-overhead operation. Some aspects, such as coalescence between adjacent threads in the search, help reduce the severity, but the cost regardless is high enough to reduce the performance substantially. Note that the worst case for `indexed` occurs when there is a large set of approximately uniform blocklengths. Not only does this increase the size of the variable length parameter space for the type, but it also maximizes branch divergence and non-locality in the search. Similar trends are seen in the `struct` type, though to a higher degree due to an even higher reliance on the variable length parameters to perform the point retrieval; each block can be a separate datatype (see Table 3.1). The cost of performing the binary search for these types can be seen by comparing the `indexed` and `indexedblock` types. Specifically, the binary search implementation of `indexed` type traversal causes significant overhead, though for packed buffer sizes less than 64 KB, the absolute overhead is no more than 9 microseconds.

The impact of the read/write stage of packing on performance is determined by the data

layout and whether the type has variable-length parameters. The best example is shown in the `indexed` and `vector` types. With a small blocklength and thus high noncontiguity, reading the values is the bottleneck of the datatype processing. With a large blocklength and thus a higher degree of contiguity, the reading is an efficient process because of the much higher degree of coalescence. If the type has variable-length parameters, then the traversal is the primary cost; but significant overhead can still be seen when packing highly noncontiguous data, such as with the `indexed` type with 8-byte blocks.

Adding the PCIe bus activity into the packing adds overhead and ultimately bottlenecks the faster packing operations for larger buffer sizes. Zero-copy keeps the overhead small for medium-sized buffers. As mentioned in Section 3.4.1, zero-copy is not used in the `struct` type, causing a higher relative performance degradation than seen in the other types due to the serialization of the packing and PCIe operations.

3.4.4 Full Evaluation: GPU-to-GPU Communication

Now that the performance of packing on various datatypes has been studied, we consider it within the context of MPI point-to-point communication. Because of the extremely inefficient performance of CUDA-based methods on irregular data (e.g., `indexed`, `struct`), for this benchmark we only consider the packing of a `vector` type of varying blocklength; an MPI_Send where data is packed at the rate of 4 MB per second will obviously not perform well. Furthermore, we do not consider the use of GPUDirect for our method, a kernel patch provided by Nvidia that allows both Infiniband and CUDA to pin the same block of memory (it is used by MVPICH, however). This will be the focus of further research and evaluation, though the integration of it will equally benefit the packing algorithm and CUDA alternatives. Figure 3.9 shows the completion time of a GPU-to-GPU ping-pong benchmark. The sender packs the `vector` data from GPU memory into contiguous CPU memory, immediately followed by a send operation, while the receiver receives and unpacks the `vector` into GPU memory. This process is then repeated back to the original sender.

The efficiency of the communication is dependent, as expected, on the data layout. A small blocklength and large buffer size, which favors the packing operation, causes a large relative performance increase compared with using the two-dimensional memory copy. A larger blocklength causes the memory copy to be largely equivalent to the packing operation. For small message sizes, GPU-to-CPU latency is the primary cost, which in this benchmark is felt four times over. Network latency, by comparison, was much lower. For medium to large-sized messages, the measured network bandwidth of 2.0 GB/s formed the bottleneck, which is much lower than the packing and memory copy throughput.

Compared to MVAPICH, our packing methodology performs roughly equivalently for small-

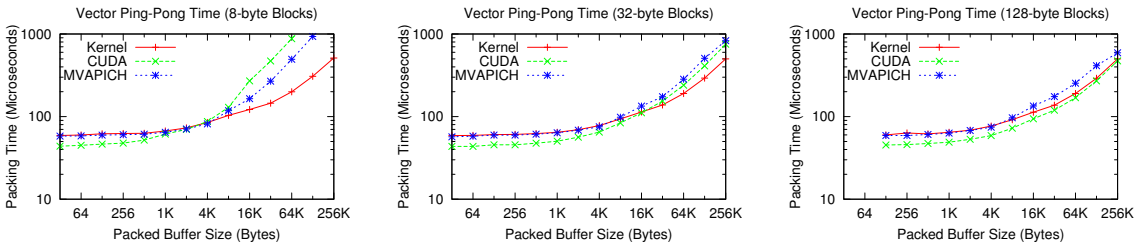


Figure 3.9: GPU-to-GPU ping-pong test, on the `vector` type with 8, 32, and 128 byte blocks, against `cudaMemcpy2D`. The `vector` stride is aligned to maximize CUDA performance.

and medium-sized buffers, and begins to outperform MVAPICH's vector communication algorithm for large-sized buffers. MVAPICH uses an optimized communication routine for vectors, performing a two-dimensional memory into GPU memory, then transferring the now-contiguous data to the CPU. While this avoids poor PCIe utilization from narrow vector blocklengths as seen from two-dimensional copying directly to the CPU, the approach is more memory intensive, using two sets of memory operations. Furthermore, no overlapping of PCIe and packing activity is performed. Through our use of zero-copy, both of these problems are overcome. The latency in MVAPICH's two memory copies is roughly equivalent to kernel launch and execution overhead, explaining the equivalence in communication performance for small- and medium-sized buffers.

3.4.5 Resource Contention Effects on Packing

To induce the contention scenarios discussed in Section 3.3.3, we use a few simple operations to stress the resource in question. We call these the application (app) operations. For both directions of PCIe activity, we merely issue a memory copy. For SM contention, we utilize a vector add operation. The reason we do this is to tie it closely to a packing operation (using the `vector` type), with packing time similar to the application operation time.

To form a baseline for comparison, we time each operation in isolation. To measure contention effects on the pack/copy operation, we initiate the application operation, then initiate and time the noncontiguous pack/copy. To measure contention effects on the application operation, we initiate the pack/copy operation, then initiate and time the application operation. Regardless of the operation, we measure the amount of time it takes to finish both, in order to see the degree of overlap occurring in the operations.

The parameter space for an experiment of this variety is enormous, so we have chosen an exemplar that is representative of the trends as a whole, and has processing times that are reasonably close for every operation. For each of the following experiments, we used a vector of

Table 3.3: User workloads in contention with the pack kernel and CUDA API calls, using the `vector` type, in milliseconds. The Workload column shows the order in which the operations are initiated, while the Proc. column shows the time between initialization of the packing/CUDA operation and its completion. Section 3.4.5 discusses the parameters.

	SM			PCIe (CPU→GPU)			PCIe (GPU→CPU)		
Workload Order	**User**	**Proc.**	**Total**	**User**	**Proc.**	**Total**	**User**	**Proc.**	**Total**
Serialized (Pack)	1.00	2.55	3.55	3.34	2.55	5.89	2.56	2.55	5.11
Serialized (CUDA)		2.96	3.96		2.97	6.31		2.97	5.53
User→Pack	-	3.52	3.55	-	3.65	4.08	-	3.18	5.09
User→CUDA	-	3.00	3.03	-	3.66	4.06	-	5.53	5.54
Pack→User	3.53	-	3.56	4.08	-	4.11	5.08	-	5.11
CUDA→User	1.03	-	3.00	4.05	-	4.07	5.53	-	5.53

total size 16 MB, and defined the `vector` datatype to have a count of 262,144, a blocklength of 8, and a stride of 64 bytes. Rather than choosing more realistic parameter sets (these cover the entire buffer), we chose these values to best show the effects of resource contention due to each operation having a similar run time. In the case of GPU contention, we show that operation scheduling plays an integral role in creating contention among various operations, so we expect similar results for other datatypes and operations, though on a different scale.

Table 3.3 shows these exemplars. For the SM experiment, the order of initiation is critically important. When using the packing kernel, either operation, when initiated after the other, gets starved out, only starting when there are available SMs. The two-dimensional memory copy, avoiding the SMs entirely, does not suffer this problem, and sees no degradation in performance. In other words, the Direct Memory Access (DMA) engine handles the copy operation, leaving the GPU's SMs untouched.

For the PCIe experiment from GPU to CPU, both the application operation and the pack/memory copies suffer, as both must use the same lane of the bridge. However, a very interesting finding can be seen in the app-then-pack case. Since the packing operation utilizes zero-copy for all but the `struct` type (e.g. memory mapping GPU memory into CPU memory), we notice that the scheduling mechanism seems to treat the SM-issued bus transactions more favorably. Using CUDA memory copies instead of the pack does not overlap at all with the application memory copy and vice versa, since the transfers are completely serialized on the CPU end (regardless of using different CUDA streams).

For the CPU to GPU PCIe experiment, while we would expect an insignificant degree of contention due to the operations using different PCIe lanes (PCIe is full duplex), we actually see some degradation in the time taken, though the totals for issuing both concurrently are much less than that for the completely serial case. We unfortunately cannot explain this behavior with absolute certainty, but we hypothesize it to be an artifact of the scheduler, or a small degree of contention with respect to transferring kernel parameters.

```
struct vblock_t {
      int num_verts, num_cells;
      int num_cell_verts, num_complete_cells;
      int num cell_faces, num_face_verts;
      int num_orig_particles;
      float mins[3], maxs[3];
      float *vertices, *sites;
      float *areas, *vols;
      int *cells, *face_verts;
      int *num_cell_faces, *num_face_verts;
};
```

Figure 3.10: Parallel Voronoi tesselation data structure for HACC.

More complex contention scenarios, such as mixed PCIe/SM loads and multiple users, are not shown because of the countless possibilities they entail, though we can make a few observations. For algorithm patterns that interleave PCIe transfers and kernels, there is more flexibility for the scheduler to insert other operations between them. Therefore, the starvation would not be as strict as that occurring in some of the cases in Table 3.3. Perhaps, in future GPU architectures, advanced schedulers would be able to enable resource sharing on a finer grain level, increasing the fairness with respect to performance of multiple application contexts hitting on the same hardware.

3.5 Evaluation - Distributed Voronoi Tessellation for HACC

The next application benchmark is taken from the analysis of cosmological simulations. The HACC [79] cosmology code is a framework for N-body particle simulations of dark matter tracer particles. Some analysis tasks such as identifying cosmological voids are enabled by the conversion of raw particle data to a Voronoi tessellation, which converts a point cloud to a polyhedral mesh. When executed in a spatially-decomposed data-parallel manner, each MPI process computes the data structure shown in Figure 3.10.

The structure contains a combination of integer and floating point scalars and vectors, and pointers that need to be followed to access the actual data members. Each process contains a different number of particles, hence different lengths of buffers that need to be fetched. When writing and reading results from parallel storage using MPI-IO, the above data are accessed using a single custom MPI data type by each MPI process. Traversing the data type is a set of contiguous pieces combined in a noncontiguous fashion.

To assess the performance of packing this datatype, we first logged the memory accesses of

Table 3.4: Packing times in milliseconds for 8 MPI ranks. *CPU Pack*: reference packing time of data resident in CPU RAM. *Copy-only*: GPU-to-CPU packing time using memory copies for each GPU buffer. *Kernel*: GPU-to-CPU packing time using the packing kernel.

Rank	CPU Pack	Copy-only	Kernel
0	0.96	0.43	0.35
1	0.40	0.25	0.16
2	1.68	0.65	0.57
3	1.53	0.61	0.53
4	0.92	0.42	0.34
5	0.26	0.19	0.11
6	0.83	0.39	0.31
7	0.55	0.29	0.20

the CPU packing done by MPI for a test run of 32,768 dark matter tracer particles converted to a Voronoi mesh using eight MPI processes. Each process produced a trace that logged the base type, quantity, and starting address associated with fetching each structure member.

We then regenerated the identical memory access pattern in our benchmark and compared the performance of three versions of data type packing. Table 3.4 shows those results. "CPU Pack" is the time to pack the original MPI data type using the CPU only. The "Copy-only" column is the time to manually pack a single buffer using a sequence of GPU-to-CPU copies, one for each `struct` field solely using `cudaMemcpy`. The "Kernel" column is our GPU packing kernel version. Our results show a 13-43% reduction in time-to-CPU by using packing, with a median reduction of 20.8%. We attribute this to the reduced latency costs in issuing a single kernel versus multiple copies.

3.6 Evaluation - APLOD in GPU Memory

Finally, we evaluate our methodology on APLOD processing in GPU memory, compared against a hand-coded methodology. Recall that the APLOD transforms can be performed implicitly using datatypes (see Figure 2.3). Specifically, a `struct` of varied-width `vectors` can be used, with widths corresponding to CV values. Figure 3.11 shows the performance of packing (corresponding to shuffling) and unpacking (corresponding to reconstructing), including intermediate PCIe transfers. As it can be seen, the hand-coded method is much more efficient than the generic packing methodology. We attribute this to the order of processing implied by our packing compared to the order used by the hand-coded method. In the generic packing method, adjacent threads process adjacent elements on a *per-vector* basis, incurring per-warp strided reads. However, the hand-coded method is optimized to read the buffer into memory first then iteratively

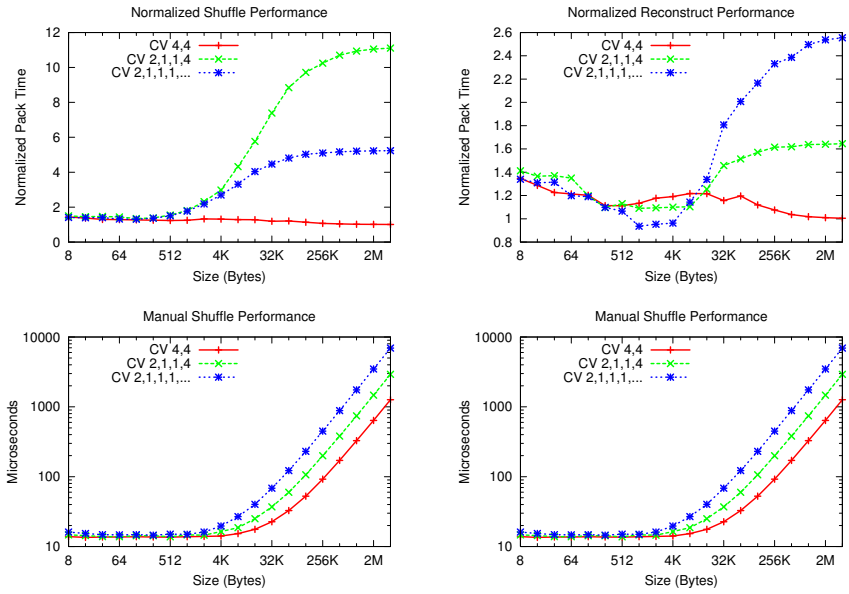


Figure 3.11: APLOD full precision GPU-to-CPU shuffle and CPU-to-GPU reconstruction performance.

write out contiguous components on a per-warp level. Hence, the hand-coded method is far more efficient by taking advantage of the APLOD semantics, whereas the datatypes method does not have this information.

3.7 Related Work

There have been a number of efforts to integrate GPU functionality into an HPC environment, with modifications at the application, programming model, and library levels to account for a discrete GPU main memory space.

At the application level, algorithms that use both MPI and GPUs, such as Jacobsen *et al.*'s flow computation algorithm [38], are modified to allow efficient GPU computation, such as changing the problem space partitioning to benefit GPU access patterns. However, since no library support is enabled, the algorithms end up losing efficiency on the handling of memory as well as the amount of programming effort. Furthermore, MPI datatypes differ from these specialized data structures in that the datatypes efficiently encode a subset of the data structures used, for use in communication and I/O routines.

At the programming model level, Gelado *et al.* created the Asymmetric Distributed Shared Memory model (ADSM) to provide a single GPU address spaces across a cluster [26]. From the CPU standpoint, all GPU memory is in a single address space, but GPUs are only aware of

their resident memory space. Their consistency model is designed for and allows operating and processing on the shared address space in contiguous chunks with memory coherence, and would have to become much more complex to enable the transfer and consistency of noncontiguous data or partial data within a contiguous buffer. Since our method is based on the message passing model with no consistency enforcements, our work does not apply here, though an interesting problem could be the combination of ADSM with noncontiguous datatypes.

Zippy [22] combines the message passing and shared-memory models (based on Global Arrays) and provides a single address space for all GPUs in the cluster, using MPI as its backend. Zippy works specifically on array-based data, as compared to our aims of supporting a generalized data representation. Since their dimensionality support extends beyond the two-and-three dimensional arrays representable by CUDA, our work is applicable to both representing an area that needs to be transferred (such as noncontiguous array boundaries) and to subsequently packaging that data efficiently.

At the library level, Distributed Computing for GPU Networks (DCGN) [97] extends MPI and partially implements the standard with a highly threaded design, utilizing signaling/polling mechanisms to allow for GPU-sourced communication. It also uses existing MPI libraries as a backend, meaning our work can directly benefit theirs. Unfortunately, given the current architectural constraints, the signaling and polling operations are cycle-consuming and lead to high latencies in GPU-sourced communication routines. This likely means that, while future architectures will allow for efficient GPU-sourced communication, the model used in today's applications will be a CPU "push-pull" model, that is, the CPU initiates the communication routine and invokes the memory management operations on the GPU.

Similarly, `cudaMPI` works on top of MPI, focusing on performance implications of different memory types, such as pinned vs. not pinned [55]. Specifically, they focus on the application of the latency/bandwidth performance model, which comes into play when doing anything GPU-related, which tends toward high-latency, high-bandwidth operations. Additionally, they briefly discuss noncontiguous memory transfer onto the CPU, but only as an application-specific column-vector transfer, and do not take into consideration MPI datatypes in general. Similar to our method, they issue a kernel to pack this data. Our work thus directly applies to their framework.

3.8 Concluding Remarks

Since GPUs are expected to continue evolving to be capable of more general purpose computations, it is extremely important to integrate them into widely used libraries in the HPC community, such as MPI. We have presented one important aspect towards this end, the processing of arbitrary, non-contiguous datatypes describing data residing in GPU memory. In particular,

we found that kernelizing the packing operation leads to huge performance improvements in datatypes that describe two non-exclusive data layouts: highly non-contiguous data, and irregularly located data. These cases are particularly important for future applications because there is a large degree of research into new ways of using GPU hardware to perform complex operations. With these complex operations come more complex communication patterns. Relaxing the data layout requirements necessary for quickly getting the data from the GPU to the CPU and across nodes is helpful from an optimization standpoint: algorithms could have local access patterns that differ from global communication patterns, and if there is efficient packing available, applications could focus more towards optimizing the local patterns.

Overall, we view our method as complementary towards the goal of full, robust integration of GPU technology into high-performance data movement frameworks such as MPI, as well as a baseline for future MPI library implementations. A complete solution to GPU data movement within MPI would not only minimize internal memory copies and fully utilize current/future versions of GPUDirect (such as in the work for MVAPICH), but also be able to flexibly determine the best methodology for transferring the data, especially noncontiguous data.

Determining how to transfer noncontiguous data in an "optimal" way would take into account the degree of noncontiguity of the data, dynamic replacement of the generic packing algorithm with type-specific kernels or CUDA alternatives when a performance gain could be realized, while taking into account competing operations for limited GPU resources. While determining the degree of resource contention would require additional support within the CUDA/OpenCL runtimes, the other factors could easily be decided based on the information encoded in the datatype. For example, CUDA DMA can be used in place of the generic packing algorithm for a single `vector` type (e.g., `CSvec`) by merely analyzing the strides/blocklengths, while a type-specific kernel could replace generic packing for single `vector` types with parameters unfavorable to CUDA DMA performance.

Another common optimization to make is the pipelining of data in communication codes to increase network utilization, such as in MVAPICH [109]. While we did not explicitly explore this in our work (our hardware configuration was bottlenecked by PCIe latency for small messages and network bandwidth for large messages), it is straightforward to provide this functionality. Given a pipeline unit of an arbitrary size, we can modify the point-to-thread mapping and the input/output offsets in Algorithm 1, signifying an *element offset* and dynamically assigning threads and thread blocks to the number of elements to read. We can also simply select the number of elements that can fit into any given buffer using the existing datatype encoding and a single-pass traversal, similar in style to Algorithm 1. For systems with increasingly high network capabilities, it is important for this functionality to be available, and our design is capable of performing pipelining with little change to the underlying methods.

Furthermore, we have shown the need, through experiments on resource contention, for

more complex resource scheduling and management on the GPU. As communication patterns get more complex and multiple threads and MPI processes access the same GPU hardware, there is nothing a user can do to prevent resource contention other than fine-tuning and organizing the code to explicitly minimize contention. Thankfully, the MPI Standard allows hints in the form of attributes to be passed to datatypes, which presents an non-intrusive path towards allowing users to suggest communication strategies. While there may be no way to avoid resource contention, at least the user could be able to have some say in the handling of it. To enable a wider range of applications to efficiently use the GPU, providing scheduling capabilities, such as a priority-based scheduler for performance critical workloads such as packing, will become an increasingly important aspect of overall GPU adoption and use, especially for complex applications.

Acknowledgements

This work was supported in part by the U.S. Department of Energy under contract DE-AC02-06CH11357, and additionally by the National Science Foundation under Grant No. 0958311.

Chapter 4

RADAR: Runtime Asymmetric Data Access-driven Replication

4.1 Introduction

In HPC systems and parallel filesystems such as PVFS [14], Lustre [89], and GPFS [88], the distribution of data across multiple storage devices is a difficult problem, for which numerous works have been dedicated to solving. The combination of high-dimensionality (multiple variables distributed in a spatio-temporal domain) and distributed requests over many processes complicates making an informed decision about how to place data to result in high performance. The problem is exacerbated when noncontiguous access patterns are induced on disk, such as subvolume access. Even optimizations made to reduce/eliminate noncontiguous disk access, such as two-phase collective I/O [100], create new access patterns for which the data distribution may not be optimized for.

Previous works have looked at data layout optimization in an HPC context in two general respects: modifying the logical layout of data with the goal of producing highly specialized data organizations for a specific usage (e.g., range-query processing on scientific data [48, 29, 30, 39, 40]), and optimizing the physical distribution of datasets to better match the mapping of process requests to I/O servers usage [94, 124, 95], either in place or as separate entities in storage, and with varying degrees of adaptability. However, these works have some combination of the following potential problems that we wish to mitigate: modified logical formats introduce both interoperability concerns and difficulties related to manual management of the custom format; works that provide multiple data layouts or replicate data in multiple formats rely on creating directories/files for each, leading to a large number of files to process any time the dataset is used; the distribution formats are either fixed or optimize for a single metric, e.g., disk thrashing via DiskSim [8], requests to a single segment of file, etc.

Given these problems, we present a model-driven, adaptive layout optimization framework using direct parallel filesystem semantics. Our layout optimization is based on partial replication, allowing a small, controllable increase in dataset sizes in exchange for I/O performance optimization. Furthermore, as opposed to previous works, which fix either the regions of data to replicate or the replication format, we allow variability in both. In particular, we present the following contributions:

1) Adaptive, storage-aware replica management policy. Given a set of I/O access patterns, our replica layout manager (Section 4.3.4) uses an I/O performance modeling approach to 1) create replicas with varied layouts for performance optimization of input access patterns, and 2) to rank replicas for inclusion under storage-limited scenarios. Furthermore, our approach is capable of gracefully handling imbalances in both server loads and client loads, using performance modeling to account for the former and distribution heuristics to account for the latter. Using a prototype MPI-IO driver, our method is shown to be effective at accelerating common subvolume decomposition tasks, showing speedups of up to $1.5 - 3.5$.

2) Single-container, non-intrusive dataset storage. All replica data and metadata is stored alongside the original, unchanged data in a single file container, achieved through direct object-storage semantics (Sections 4.3.2). Distribution of replica data among a fixed set of replica objects is enabled through a combination of sparse-file capabilities and a object-slice-based allocation scheme (Section 4.3.5).

3) Datatype- and collective-aware MPI-IO tracing. As an enabling technology, we develop a tracer capable of collecting full logical I/O requests with low overhead at the MPI-IO-level (Section 4.3.3). It is configurable to collect either pre-collective or post-collective optimizations (or both). As a result, our layout manager is able to generate performance-improving replicas for applications already using collective optimizations, while previous methods typically do not handle this case.

4.2 Background

4.2.1 MPI-IO, ROMIO, and ADIO

In the MPI Standard, MPI-IO exposes a datatype-driven file decomposition and operating method ("views"). That is, datatypes are defined much like they would for communication (see Section 3.2), except the locations are with respect to a linear byte store beginning at offset 0. This allows users to easily map data structures in memory to those in file. For example, reading and distributing multidimensional arrays can be simply performed by use of the `subarray` type for each process.

ROMIO is the implementation of MPI-IO [102] used in MPICH. ROMIO is implemented

by mapping MPI-IO calls to the ADIO interface [101], which exposes necessary functionality in order to portably implement MPI-IO and allows separation between MPI functionality and the underlying filesystem calls (e.g., POSIX vs. native PVFS). Furthermore, the design of ROMIO is such that filesystem-specific ADIO targets can be dynamically specified through the filename argument by prepending the implementation name followed by a colon, e.g., "lustre:" and "pvfs2:." This implementation is enabled by loading tables of function pointers of the particular filesystem's implementation of ADIO based on either system defaults or user requests.

4.2.2 PVFS and EOF

Recently, the "End-of-Files" (EOF) [31] extension to PVFS [14], a high-performance parallel filesystem, was created to expose the object storage abstraction directly into the client space, allowing a richer and more elegant mapping from application datasets to parallel storage systems. The object storage abstraction presents uniquely identified (typically via a 64-bit ID), extendable linear byte arrays as the elemental unit of storage, and most parallel filesystems in usage today distribute a file to a set of objects using a distribution function, or *striping*. EOF cuts out the middleman between file and object set and allows applications to forward I/O operations directly to individual objects. For example, dataset metadata can be forwarded to a single object, while the data itself can be assigned distinct objects based on timesteps, variables, etc.

4.3 Method

4.3.1 Overview

Figure 4.1 shows a general overview. The system workflow, in which access patterns are garnered from applications and replicated data layouts are created to optimize for those access patterns, is realized by a number of components. First, we develop a datatype-aware, collective-aware "pass-through" ADIO trace layer that captures I/O requests and forwards to other ADIO implementations. We then process the trace using a pattern analyzer, outputting access patterns of interest, such as strided access. Our replica layout manager ingests these patterns, and along with previously generated patterns, determines what data to replicate and in what format. Finally, our replica-aware, EOF-based ADIO implementation matches I/O requests to replications, redirecting the subsequent EOF object operations.

As a preliminary, Section 4.3.2 describes the data management policies employed by our method. Section 4.3.3 briefly discusses the tracer and generation of access patterns. Section 4.3.4 describes in detail the replication layout manager. Finally, the replica-aware ADIO driver is discussed in Section 4.3.5.

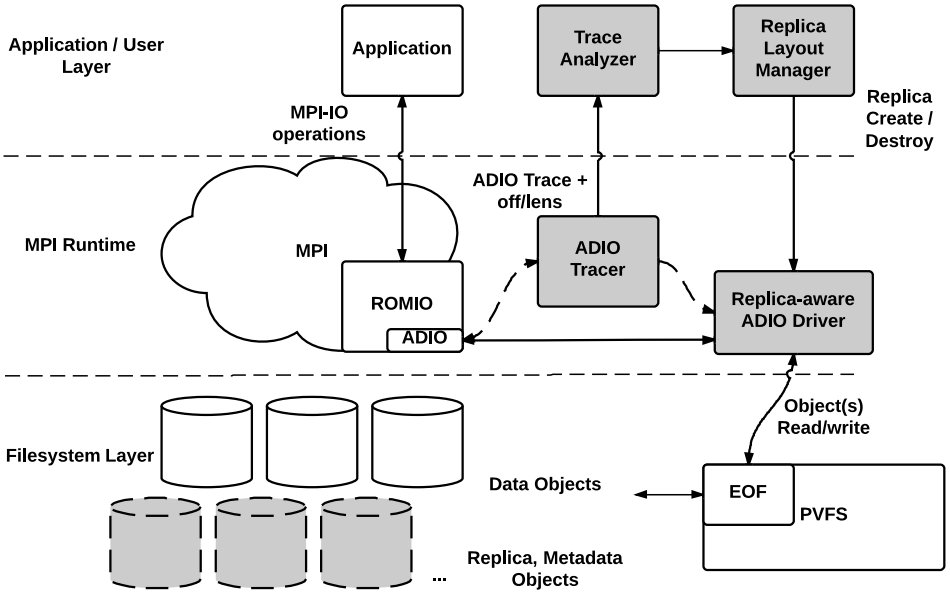


Figure 4.1: RADAR components, across the I/O software stack. The shaded figures delineate our contributions.

4.3.2 EOF Data Management

As EOF exposes object-based storage and most components of our method operate on EOF objects, we first discuss the object layout of the various data/metadata in our system. More specific details can be found in the system components' respective sections. Figure 4.2 shows the various types of objects used in RADAR, all under a single filename. The original dataset is stored as it would be on any distributed filesystem, striped by some distribution across multiple storage devices (objects, in our case). We include a file metadata object since distribution in EOF is relegated to the user. For RADAR, at file create time we allocate a number of replica objects equal to the number of data objects. We do this both for practicality reasons (limits on per-file concurrency) and for semantic reasons (currently, EOF cannot dynamically add or remove objects from a file container). A replica metadata object is used to store the mapping of replicas to object locations. Finally, the set of generated access patterns processed by RADAR is placed in a dedicated object for the results to persist across multiple application runs.

Under this data management scheme, only two sets of data exist outside of the MPI/EOF file container used by RADAR: the trace output and trace analysis results. For completeness, we intend to integrate these intermediate datasets into the RADAR format, though an option for external output is still important as a tracer of the given granularity can potentially be useful for general I/O performance analysis.

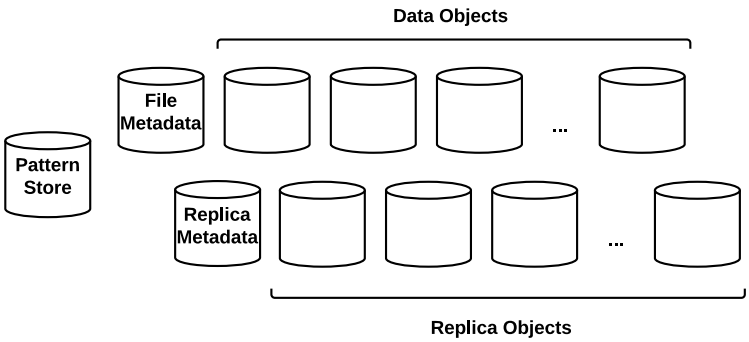


Figure 4.2: EOF object layout for RADAR.

4.3.3 I/O Tracer and Analyzer

ADIO Tracer

With respect to MPI, our layout optimization techniques need to be aware of how MPI-IO calls translate into actual I/O requests. For example, optimizations such as two-phase or data sieving may be applied transparently to the user. PMPI-based (or POSIX-based) tracers make this difficult – neither have the sufficient information to distinguish the difference between logical accesses and physical (a PMPI tracer could conceivably perform this, but would require reverse engineering of collective optimizations).

Hence, our tracer is designed as an ADIO implementation, outputting the ADIO function calls as well as offset/length pairs (either before or after applying collective optimization). 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 [13, 12], or even related works that modify data layouts (see Section 4.5). Within the MPICH source, we enable this primarily through function pointer swapping and auxiliary data structures added to the MPI file data structure. Hints passed via the `info` object enable and set options for the tracer, and, if requested, open calls are synchronized to allow for comparing times of operations to determine concurrency of accesses.

In order to enable trace functionalities in a generic manner, we simulate I/O calls from within the trace functions. While this may incur some computational overhead compared to directly embedding within specific ADIO implementations (such as our own), we feel that the generic ROMIO-level tracing can find use beyond this work, and will be explored further. Simulating I/O involves iterating over the file datatype, generating offset/length pairs corresponding to file accesses.

Through simple configuration options (either MPI hints passed though an `info` object or environment variables), we can easily allow our simulated I/O to be applied immediately upon

reaching a collective I/O request (before collective optimizations), only in the underlying serial I/O operations, or both, taking advantage of the fact that a single collective read/write is composed of an optional data exchange phase followed by numerous serial read/writes.

One issue of concern for tracing and trace analysis is determining whether operations on different processes are concurrent, and possibly conflicting. To provide sufficient information for this, we perform a barrier at file open time, take the wall-clock time, and only record per-process time deltas. This avoids performing clock synchronization while giving us a reasonable indication of when operations occur relative to one-another, sufficient for this work.

The output of the tracer includes all ADIO calls, as well as any offset/length pairs generated through reading or writing, separated on a per-process basis. Currently, the trace is in plain-text format and we do not attempt compression; compression methodologies such as inline pattern analysis and/or off-the-shelf compressors (e.g., Zlib [24]) will be explored in future work (see Section 5.1).

Trace Analyzer

The access patterns we are primarily concerned with are contiguous access patterns (sequential access of a large space with fixed or average-size request sizes) and k-d strided access patterns (accesses that differ in offset by a fixed value, or stride), both of which are common in HPC I/O workloads. These accesses occur over a linear address space of bytes, but in practice typically correspond to accesses along spatio-temporal domains or across multiple variables (e.g., temperature, pressure in a climate simulation).

To gather the desired access patterns, we built a variant of the IOSig trace analysis software [11, 121]. We similarly use a template matching approach, but, as our tracer works at the ADIO level and additionally processes datatypes, the processing of the traces has been rewritten. For more discussion pertaining to access pattern categorization and discovery, see [11, 121].

4.3.4 RADAR Layout Manager

The goal of RADAR's layout manager component is to create replicas with a layout that improves I/O performance under a given set of access patterns. Given this goal, there are a number of questions to consider: 1) How do we generate a data layout to best improve performance under a particular access pattern? 2) How do we measure performance "improvement" itself? 3) How do we augment our choices with both temporal information and limits on storage overhead?

To provide a framework capable of answering these questions in a flexible manner, the layout manager uses the following strategies. 1) To optimize in the presence of concurrent accesses, we generate replicas for *time-delimited pattern sets*. 2) To quickly generate and evaluate candidate replica sets, we adopt a *two-phase* performance-modeling approach, using a coarse-

grain performance model to quickly produce and select candidate replica sets and using a fine-grained performance model to compare against the original data layout. 3) Given user-driven constraints space-constrained scenarios, we employ a configurable *aging mechanism* to aid in selecting new replicas for inclusion, alongside the potential performance improvement calculated by the performance models. Algorithm 4 gives an overview of the replica creation and decision process, and the following sections discuss the individual components.

Algorithm 4: RADAR layout manager overview

in-out : DS (data store): RADAR MPI file consisting of pattern store PS, replication store RS, replica metadata RM
input : Pats: set of access patterns to process
input : Δt: time window size
input : $\mathcal{M}_f$, $\mathcal{M}_c$: fine, coarse-grained perf. models
input : γ: decay function
input : σ: storage upper bound

// split patterns into sets of concurrent accesses
1 $\beta \leftarrow$ partitioning of Pats into buckets by Δt
2 processed $\leftarrow$ { }
// process each pattern subset
3 **for** $\flat \in \beta$ **do**
    // use coarse-grain model to make candidate replica
4   reps $\leftarrow$ `opt_replicas`($\mathcal{M}_f$, $\flat$)
    // use fine-grain model to estimate perf. benefit
5    $C_{\text{diff}} \leftarrow \mathcal{M}_f(\flat)$ - $\mathcal{M}_f$(reps)
6   append $\{C_{\text{diff}}$, reps $\}$ to processed
// read existing (benefit, age, replicas) tuples, applying decay
7 existing $\leftarrow$ [$\{\gamma(c, \text{age}), R\} \mid (c, \text{age}, R) \in$ PS]
8 merge $\leftarrow$ `sort`(processed $\cup$ existing)
9 write replica sets in decreasing order from merge, evicting replicas from merge in increasing order as required by σ, until no more can be added

Pattern Preprocessing

Preprocessing of the patterns is driven by our optimization goals: create a replica enabling efficient access of the pattern, while being aware of concurrent system operations. The latter has been examined in previous work by converting all accesses into log-structured, effectively creating a one-to-one process-to-server mapping [4, 124]. Here we are looking at flexible distribution among multiple servers for read optimization, rather than write optimization.

Each pattern in our analyzed traces has starting and ending times. Given these and a value δ, the patterns are partitioned into *buckets*, each corresponding to a time window of length δ.

Each bucket is then considered a single entity for the purposes of performance modeling and optimization. As each bucket need not be sorted in our method, the overall process is linear in the number of patterns.

Replica Generation and Ranking – Performance Modeling

As mentioned in the overview, we use a simple, constant-time performance model to generate candidate replicas, and a more involved model to give a relative performance comparison between the original data layout and candidate layouts. This design decision is made in order to quickly generate replicas for testing, while retaining the ability to evaluate against unbalanced access patterns with respect to either the amount of data requested per-process or the amount of data processed by each I/O server.

Preliminaries Our models both use latency/bandwidth measurements over both network and storage, assuming serialization of requests at the node level (both client and server) and requests to storage. Table 4.1 shows the relevant variables. Furthermore, we make a few simplifying assumptions across both models that, while harmful to general-purpose high-accuracy performance prediction, still allow us to make valid measures for comparative purposes over time-gated accesses in a manner that is computationally reasonable. Firstly, we assume no pipelining of network and storage operations, insulating us from false positives arising from slightly different access schedules but giving a pessimistic view of system capabilities; we consider both types of costs equally from an optimization point-of-view. Secondly, resource contention is measured through the aggregation of request latencies and, in the fine-grained performance model, through penalization terms on nodes based on the number of distinct requests. This misses some phenomena observed in real runs or in full system/subsystem simulators [8], such as disk head thrashing.

Coarse-grained Model The coarse-grained model is a generalization of the cost model created by Song et al. [94] to optimize accesses under the following characteristics: 1) uniform access sizes, 2) perfect access distribution among servers corresponding to PVFS data layouts, and 3) single time of issuance across all processes. This model, while not created for general purpose I/O modeling, has proven useful for HPC applications with regular access patterns, and is appropriate for driving our replica placement method, given that we are in full control of replica placement and can produce such regular accesses. First, we discuss the model and generalization, then we discuss how we find effective replica layouts using it.

The cost model by Song et al. have four separate costs which are summed to find the final result: T_e, the establishment time for all network operations, T_x, the time to transfer all request data across the network, T_s, the "startup" time for all storage accesses, and T_{rw}, the read/write

Table 4.1: Performance Model System Parameters

System parameters	
n	I/O servers
ℓ_{net}	I/O request (network) latency
b_{net}	Network per-byte transfer time
ℓ_{sto}	I/O request (disk) latency
b_{sto}	Storage per-byte transfer time
r_s	Local storage readahead
Per-pattern-set inputs	
$\mathbb{P}$	Set of access patterns with process mappings
p	I/O participants (clients)
m	I/O participants per node
Coarse-grain model input/outputs	
B	Total request size across all patterns
n_p	Servers contacted per client
r	Average request size per client per server $= B/(pn_p)$

time for all storage accesses. These costs are computed separately based on the PVFS data layout being used, which they fix to be a one-to-one process to server mapping (1-DV), a one-to-all mapping (1-DH) and a process-group to server-group mapping (2-D). Refer to the respective paper for more details. We collapse this mapping based on a simple observation: the parameter being varied across each of the models is the *servers contacted per client*. Making this an explicit variable n_p allows us to collapse the equations into a single set:

$$T_e = \max(mn_p, \lceil \frac{pn_p}{n} \rceil)\ell_{net} \tag{4.1}$$

$$T_x = \max(mn_p, \lceil \frac{pn_p}{n} \rceil)rb_{net} \tag{4.2}$$

$$T_s = \lceil \frac{pn_p}{n} \rceil \ell_{sto} \tag{4.3}$$

$$T_{rw} = \lceil \frac{pn_p}{n} \rceil rb_{sto}. \tag{4.4}$$

Note that the n_p term is within the ceiling/maximum functions, and not factored out, to ensure the model will remain accurate for low-client-count scenarios (i.e., $pn_p < n$).

Coarse-grained Model Usage Given the definition of the coarse-grain model, we derive a simple replica creation process, using the following strategy: assume the underlying accesses are regular and uniform, compute $\min_{n_p}(T_e + T_x + T_s + T_{rw})$, then resolve any load balances by over/under provisioning replica striping across the servers. After computing B and an average m, simply calculate model values for $n_p \in \{1, 2, \ldots, n\}$ and choose the minimum. Then, perform

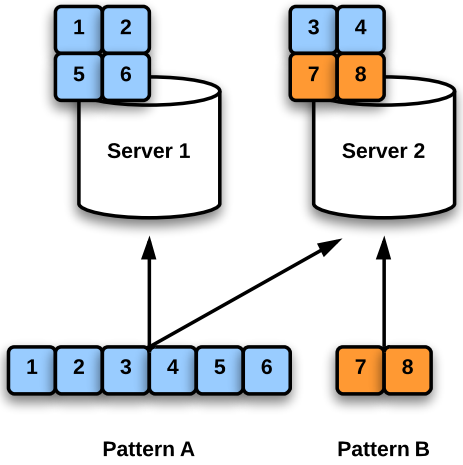


Figure 4.3: Access pattern over and under provisioning based on model optimization on balanced accesses (for $n_p = 2$)

the logical striping under the assumption that r is the actual request size per client. Finally, compute for each pattern which servers its data resides on. An example mapping is shown in Figure 4.3. The intuition behind this layout heuristic is that patterns with balanced accesses will be optimized as normal, and overprovisioning for unbalanced accesses with larger relative sizes will be made up for by underprovisioning for accesses with smaller sizes, mapping degree of concurrency to relative access size.

Fine-grained Model The fine-grained model shares similarities to the coarse-grain model, using latency/bandwidth modeling at both the network and storage level to generate an overall cost. However, whereas the coarse-grained model makes some significant assumptions about access characteristics, we need a more robust model capable of capturing load imbalance. Our approach consists of the following two steps, with the underlying steps of mapping each pattern in the pattern set to its respective client process/node and set of contacted servers: 1) compute a localized T'_e, T'_s and T'_{rw} for each server, and 2) calculate the total time-to-receipt T'_t for each client node based on the server calculations, with the maximum among them being the total request time.

The computation of T'_s and T'_{rw} are relatively straightforward, with T'_s being the number of noncontiguous blocks (measured at a page granularity and taking into account readahead) accessed times the storage access latency, and T'_{rw} being the total size of all requests to the server times the inverse of the bandwidth. For T'_{rw}, we additionally adjust the performance to account for readahead – if two consecutive requests are within a readahead window (default

128KB on Linux), then the disk latency cost is avoided at the cost of consuming the bytes separating the two requests.

The computation of T'_e and T'_t are more nuanced, as we must consider both access latency and wait times for request/response receipt. As computing these wait times exactly would require a known access schedule (as well as more architecturally-accurate simulation), we instead compute an approximate. For T'_e, we compute the network latency times the number of incoming requests, with a *penalization term* ϵ_e. For this, we take the node contacting the server with the maximum number of outgoing requests, and assume that the request to the server occurs after all of its other requests. Similarly for T'_t, the penalization term ϵ_t is computed by assuming that the data requested by the given node is issued after the server's access schedule, for the contacted server with the largest load.

Considering the full set of pattern accesses, the computational cost of evaluating the model is best described using a bipartite graph $G = (M, N, E)$, where M represents the client nodes, N represents the server nodes, and E the mapping of client nodes to contacted server nodes. Generating the graph requires time proportional to the number of contiguous blocks encoded by the pattern set times the average number of servers containing each block (at most the average vertex degree). Evaluating the model, akin to traversing each vertex and its corresponding neighbor set, is linear in $|E|$.

Replica Aging

An important consideration in the RADAR methodology is how to represent the "age" of a particular pattern (or set of patterns), with the implication that the layout manager can prefer "younger" accesses to "older" ones. While raw timestamps can easily be gathered by tracers, real time as a measure of age is noisy and rife with pitfalls: machines can go down, scientists can go on vacation or not run applications over the weekend, etc. Hence, we need a more robust measure of time.

To provide a robust measure of age relative to application runs, RADAR defines age to be the amount of data read from the dataset, normalized by the original dataset size. For example, consider a simulation that writes X bytes of analysis data. Afterwards, consider three read-only analysis application runs (in order of execution): A, B, and C. Under our system, patterns generated during application A's run would have an age of zero, patterns generated during B would have an age of $\text{read}(A)/X$, and patterns generated during C would have an age of $(\text{read}(A) + \text{read}(B))/X$. The metadata required to maintain the ages, namely, the total amount of data read from a dataset, can be stored in the file metadata of the RADAR MPI file. While this method has some issues of its own, such as a bias towards larger applications, it nevertheless sufficiently captures the temporal context necessary to compare the relative age

of access patterns.

Given this formulation of age, we use a configurable *decay* function, applying on the performance benefit of already existing pattern sets. That is, given a benefit measure b and the difference in age a between the current dataset state and the age at which the pattern set was generated, the result of the decay function $\gamma(b, a)$ is used in place of b in the replica selection process. Examples are the identity function ($\gamma(b, a) = b$) and an exponential decay function such as *half-life* ($\gamma(b, a) = b2^{-a/X}$, the half life in this case corresponds to a reading of the full dataset).

4.3.5 Replica-aware ADIO Driver

The responsibilities of the RADAR ADIO driver is to interface with EOF, maintain the semantically varying sets of objects, and to remap I/O requests into the replica space, as appropriate. Asides from the remapping portion, the rest of the processing is relatively simple, corresponding to loading/distributing file and replica metadata, updating the bytes-read component (for later use by the layout manager), and driving some RADAR-specific operations, such as performing the replication. The replication process itself is discussed in the next section, while the request remapping is presented in the section thereafter.

Replication Operations

A yet undiscussed component of the RADAR process is how exactly to distribute a replica. As discussed earlier, we currently implement two types of replicas: a contiguous replica, and a strided replica. Both are striped in the replica object space. The main problem here arises from the fact that we are using a shared set of replica objects to store multiple objects.

In order to both simplify design and minimize usage of space, we exploit sparse-file capabilities in the local filesystems employed by PVFS. Essentially, file blocks not written to are not stored on the disk. The typical example of this is a file that is created on open, and written to once at byte offset 1GB. The file size reported by stat will simply be the last byte offset written, while the actual storage used will be a single disk block, along with metadata.

We divide all objects into allocation units we call *object domains* (ODs), an example of which is shown in Figure 4.4. An OD corresponds to the full set of replica objects, spanning a per-object address space with fixed, large-granularity sizes. Each replication is placed within a single OD, as shown in the example. To avoid biasing replica placement towards one object or another, replicas are assigned starting objects in round-robin order. In the figure, for example, the replica shown begins addressing at the leftmost object, while the next replica created will begin addressing at the next leftmost object.

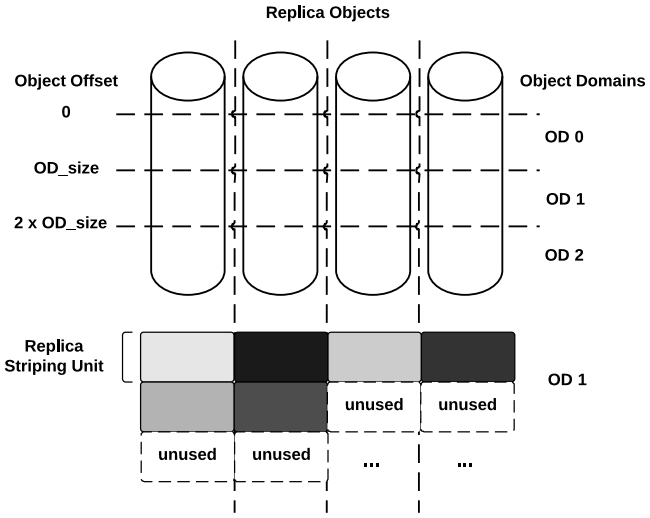


Figure 4.4: Allocation units in RADAR ("Object Domains," or ODs), and replica layout in an OD.

I/O Processing in the Presence of Replicas

Given a set of replicated data layouts and a set of I/O requests (e.g., generated by a call to `MPI_File_read`), the most difficult task for the RADAR I/O driver is to determine which, if any, replica to read/write from. This problem can be separated into two tasks: matching requests (offset/length pairs in file) to replicas, then choosing which among the replicas to issue the operation to, or none at all. Note that, for this work, we only consider full matches, in which each contiguous block of a request can be fully satisfied through reading from a single replica. Partial replica mapping, resulting in reading partly from a replica and partly from the original data, is a focus for future work; our intuition is that splitting a single request into multiple locations will result in increased latency costs, especially in a multi-reader environment. However, answering this concern fully requires more sophisticated performance modeling.

Replica Matching The matching itself, given an offset/length pair and a replica, is a simple matter, but the main concern is coping with an increasing number of replications. For n replicas, it is not preferable to have to do a naive linear test of each one: for a large number of small, localized replicas, this approach would clearly not scale. Furthermore, arbitrary replicas over strided data (multiple dimensionalities, block sizes) make using classical spatial partitioning data structures such as interval and R-trees unwieldy: the strided pattern would need to be flattened into its elemental blocks, expanding the tree size considerably. Replicas over contiguous data can benefit from these structures, however.

As a compromise between flattening the strided pattern representation and inducing a linear

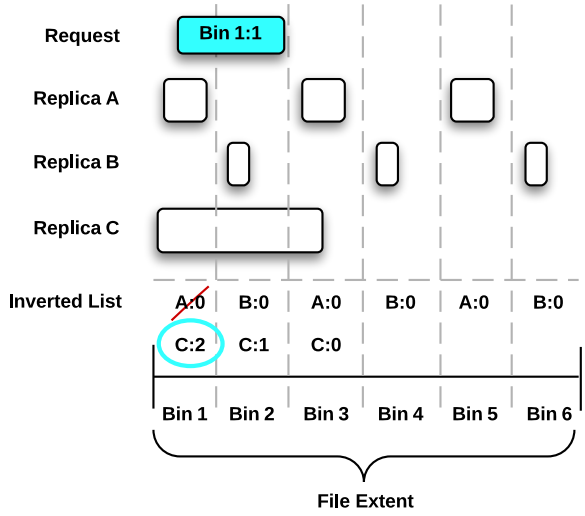


Figure 4.5: Replica lookup using inverted index with bin extension

scan of the full replica set, we build an *inverted index* [117, 127] over the file. As the name suggests, an inverted index maps regions of file to a list of replicas overlapping with the regions, rather than mapping replicas to regions of file. Typically, inverted indexes have been used to accelerate searches such as large-scale document searching – search terms match to a set of containing documents, but they have also proven useful in other scenarios [39, 40] First, we partition the extent of the file into *bins*, where each bin represents a distinct contiguous region of the file. Then, for each bin we list the replicas for which locations encoded by the replica overlap with the bin's byte boundaries. Hence, queries (i.e., finding the replicas which may satisfy a given offset/length pair) need only look at and process the list of replicas in the bins that intersect with the request. This representation is shown in Figure 4.5.

One problem with the inverted index approach is that multiple bins may need to be examined, and their replica lists intersected, to fully process queries. To avoid paying this additional cost, we perform a simple modification of the per-bin replica list to enable full querying using only a single bin. Namely, for each entry in the list of per-bin replicas, we record the *bin extension*, or the number of bins that the overlapping contiguous block extends across (in increasing byte order). For replicas over strided data, we record the maximum extension over all blocks that overlap in the bin. Then, given an offset/length pair to query on, we need only look at the intersecting bin with the smallest byte boundary and compare the bin extension of the request to the bin extension of the replica. This modification is also represented in Figure 4.5, with the numbers next to the identifiers representing the bin extension.

Finally, to help mitigate a worst-case linear search time of the inverted index (all replicas

overlap with a particular bin) for every request, we additionally keep a one-element history of replica matches, with the assumption that it is more likely than not that a replica will be read multiple times in sequence by a client. For processing a request, the previous replica read is matched against first, followed by a query against the inverted index if the request does not match.

Replica Selection Given an I/O request and a set of replicas that can satisfy the request, a significant problem is how to select which replica among the set to read from that would result in the best performance. This choice is nonexistent for writing, as all replicas must be updated: as noted earlier, our primary use-case is accelerating read performance for multiple analysis workloads; mixed read-write workloads would not fare well under our methods. Complicating the selection is that the available information to make this decision is inherently local (requests occur on a per-process or possibly per-MPI-aggregator level), consisting of only the replica metadata.

Our solution to this problem is a simple heuristic we call *smallest containing block* (SCB). The idea behind SCB is that of *specialization*: we consider replicas with a finer granularity to be more specialized than those without, and hence should be prioritized in the replica selection process. This generally means that replicas over strided data will more than likely be selected over replicas over contiguous data, as each of the strided data would be more sparse.

Other methods, such as performance models, could certainly be "swapped" in to make the replica selection as well, as the data layout is known to all. However, inherently local performance models cannot have a big-picture view of the system by which many I/O operations could be happening at any given time. In this sense, we are relying on the RADAR layout manager to make informed choices pertaining replicas, and allowing the I/O mechanism itself to be simpler.

4.4 Experimental Evaluation

4.4.1 Setup

All experiments were ran on the LCRC Fusion cluster at Argonne National Laboratory. Each node contains two quad-core Intel Xeon processors at 2.53 GHz with 32GB RAM, and nodes are connected by Infiniband QDR. Each node in Fusion contains local hard-disk storage (250GB IBM iDataPlex). Additionally, our implementation of RADAR is based on MPICH 3.0.2 and PVFS2 2.8.1, patched with EOF. Due to issues with Infiniband support for PVFS on Fusion, both MPI communication and PVFS client-server commuinication is performed via TCP over Infiniband.

Table 4.2: Performance Model Variables

System parameters		
n	8	I/O servers
r_s	128KB	Local storage readahead
ℓ_{net}	$32.9\mu s$	I/O request (network) latency
ℓ_{sto}	$6.20ms$	I/O request (disk) latency
b_{net}	$0.00112\mu s$ (867MB/s)	Network per-byte transfer time
b_{sto}	$0.0212\mu s$ (44.98MB/s)	Storage per-byte transfer time

4.4.2 RADAR-specific Setup

As we use a modified version of PVFS and each node in Fusion has local storage, we assign a subset of the nodes to serve as PVFS I/O servers and use the remaining as I/O clients. We use eight I/O servers in all experiments. Hence, each server initially contains 8 GB of data, striped using 1 MB blocks.

Table 4.2 shows the performance model parameters we gathered via microbenchmarks on Fusion. We use the BMI `pingpong` utility in PVFS to gather network performance through PVFS, where BMI (Buffered Message Interface) is PVFS's client/server communication interface. We use simple read benchmarking via co-locating a PVFS client with a server to gather storage performance parameters. Note that the microbenchmark result for disk bandwidth is much lower than expected: the bandwidth when not going through PVFS is 90MB/s. We were unable to eliminate this discrepancy, but we believe it to be a result of internal threading and buffering overheads on the PVFS server.

For our inverted list acceleration structure for replica lookup, we divided the file into 1024 bins, each of which covering a 64MB extent of data.

4.4.3 Benchmarks

We evaluate our layout optimization work within the context of multidimensional array decomposition, a common set of accesses seen in numerous analysis workloads. Specifically, we define a three-dimensional volume of data over a fixed number of contiguous timesteps and use MPI-IO to read selections of the resulting dataset using four types of decompositions, shown in Figure 4.6: row-wise (distribute volume by contiguous plane), column-wise (distribute volume by non-contiguous plane), block-wise (distribute volume by 3D subvolume) and timestep-wise (distribute single subvolume from a range of timesteps). The row-wise decomposition induces contiguous patterns at each process, while the remaining decompositions induce multidimensional strided patterns at each process, or contiguous patterns at each aggregator process in the case of collective I/O. Note that these access patterns are a superset of the access patterns

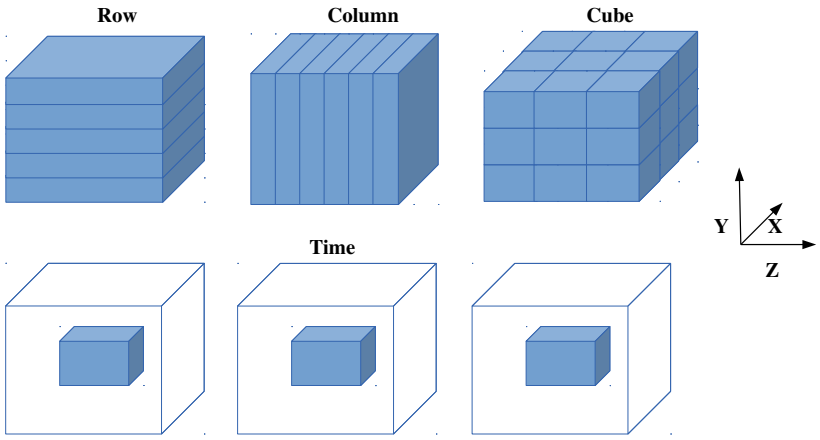


Figure 4.6: Subvolume decompositions used in our evaluation (contiguous in order Z, Y, X, time).

exhibited by several well-known benchmarks such as MPI-Tile-I/O [66], IOR [37], and PIO-bench [90], all of which perform accesses with regular (single- or multi-dimensional) strides.

For all experiments, we used a subvolume of (time, X, Y, Z) dimensions (in row-major) $128 \times 256 \times 256 \times 256$, each element of which is a 32-byte structure (e.g., four C `doubles`). The total size of this dataset is 64GB.

4.4.4 Decomposition Performance

We test each decomposition using the following MPI process configurations with respect to performing I/O: independent I/O with all processes on each node participating, independent I/O with one process per node participating, and collective node with one aggregator per node. The first is a "naive" approach for parallel I/O, but one that is still used as a result of its simplicity as well as for access patterns such as log-structured, which structures data in the order written (typically for checkpointing/restarting, but some libraries such as ADIOS use log-structured with respect to each process's writes). The second is similar to the first, but represents the computation model of MPI+threads or MPI+accelerator, in which a single process on each node performs communication and I/O while intra-node parallelism is used for computation. The third is more common, as it allows optimizations such as two-phase I/O [100], though with the overhead of communication and data movement between I/O participants.

Figures 4.7, 4.8, 4.9, and 4.10 show performance under the different decompositions both before and after RADAR data replication. For these runs, we synchronize prior to running the decomposition and calculate bandwidth with respect to the maximum elapsed time for each individual read. There are a few points of interest in general about these experiments:

1. All decompositions except the time-based decomposition decompose the same overall data size of 2GB (four timesteps of 512MB volumes). That means, with an increasing number of clients, the average request size decreases and the number of requests increase, leading to potentially less efficient access when not using collective I/O.

2. The time-based decomposition, as mentioned, defines a fixed-size subvolume for each client to read of size 64MB. As the number of clients increase, the per-client requests remain the same, leading to an increase in the total request size.

3. The cube decomposition divides the volume into perfect cubes ($1, 8, 27, 81, 125$, etc.) no less than the number of clients, and clients are assigned multiple blocks to read, resulting in varying request granularities based on the number of clients. For instance, a four-client run will divide the subvolume into eight blocks and assign two blocks to each client. This can lead to both load-imbalance (processes can be oversubscribed blocks compared to others) and varied access patterns due to the possibility of multiple smaller blocks combining into a single, large, contiguous block.

4. Finally, the difference between having a single client per node and having up to eight is marginal, due to the much larger performance potential of the network vs. storage.

Figure 4.7 shows the time-based decomposition. Here, I/O aggregation was disabled by ROMIO due to the per-process data being both non-interleaved and separated in storage, and thus is not shown. Without replication, the aggregate performance is far below peak performance due to the noncontiguous accesses. The use of data reorganization through replication enables high performance across the spectrum, though tapering off once I/O servers begin processing requests from multiple clients.

Figure 4.8 shows the cube-based decomposition. As mentioned, this decomposition results in block sizes of high variance with a changing number of clients, which is a significant factor in the overall performance. As seen in the figure, the performance implications can easily be seen between the four-client and eight-client decomposition for the non-replication case, and the eight-client and 16-client decomposition for the RADAR case. Regardless, the use of RADAR helps to smooth out the performance characteristics as a result of making the strided accesses contiguous per client and over/under-provisioning of replicas based on load. Additionally, for the single-client case, we notice a large performance gap between the no-replication and replication case. We are currently unable to diagnose this difference; the generated layout by RADAR is the same and the I/O driver follows largely the same code path. Unfortunately, collective I/O results were unable to be gathered; we encountered crashes in the ROMIO implementation and/or MPI datatype processor when attempting to run our workload.

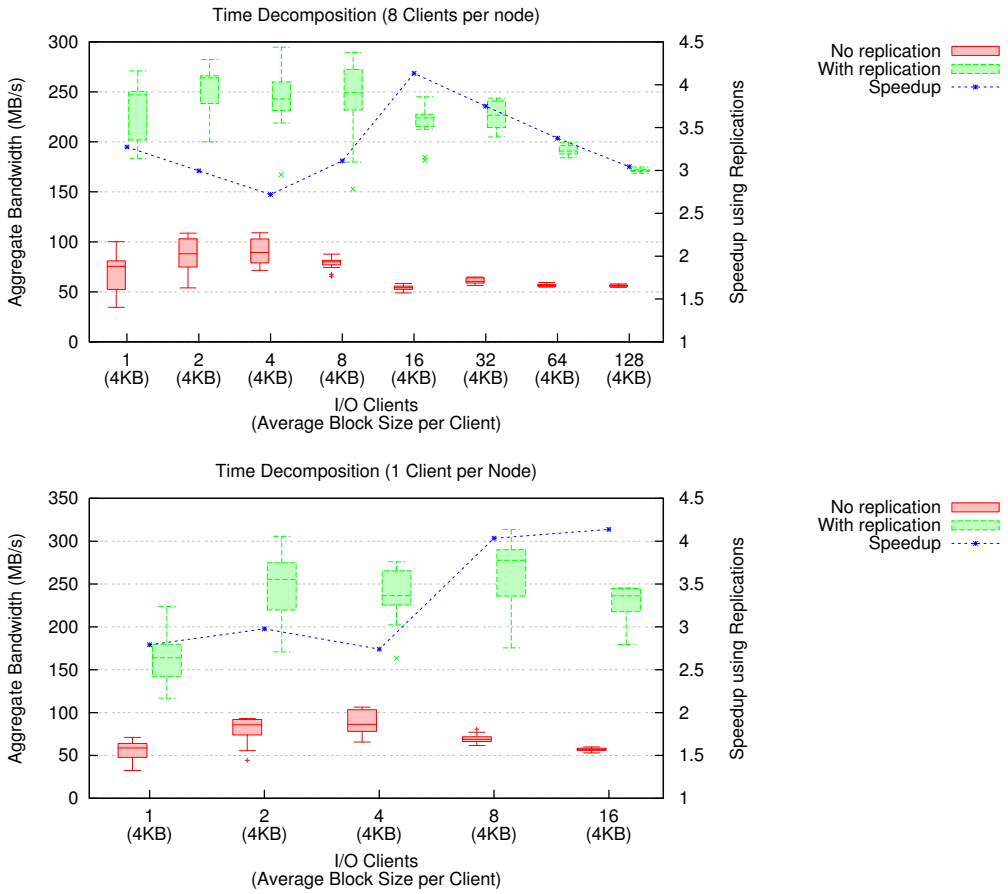


Figure 4.7: Subvolume-over-time-decomposition results with different process configurations

Figure 4.9 shows the column-based decomposition performance. This represents a pathological case of I/O, as seen by the average contiguous block sizes. Hence, performance without collective optimizations or RADAR is far worse than any of the other decompositions as the number of clients increase. RADAR is capable of greatly improving performance over the original data layout, but tapers off for increasing client counts. The reason for this, asides from the increased number of requests per server, may also be implementation-based: the RADAR MPI-IO driver (along with the native PVFS2 driver) enter processed datatype offset/length pairs into a queue, then use PVFS2 *list I/O* [18] to send the list of requests to the I/O server in one go (list I/O is most similar to the previously-proposed POSIX I/O extensions `readx`/`writex` [107], an interface for specifying sets of noncontiguous file accesses to noncontiguous memory). The block size is so small in this case that the queue becomes full and issues correspondingly small requests. Note that, for fairness, we use the same queue depth of 64 as used by the PVFS2 ADIO implementation (non-configurable). As for collective I/O, RADAR performs similarly

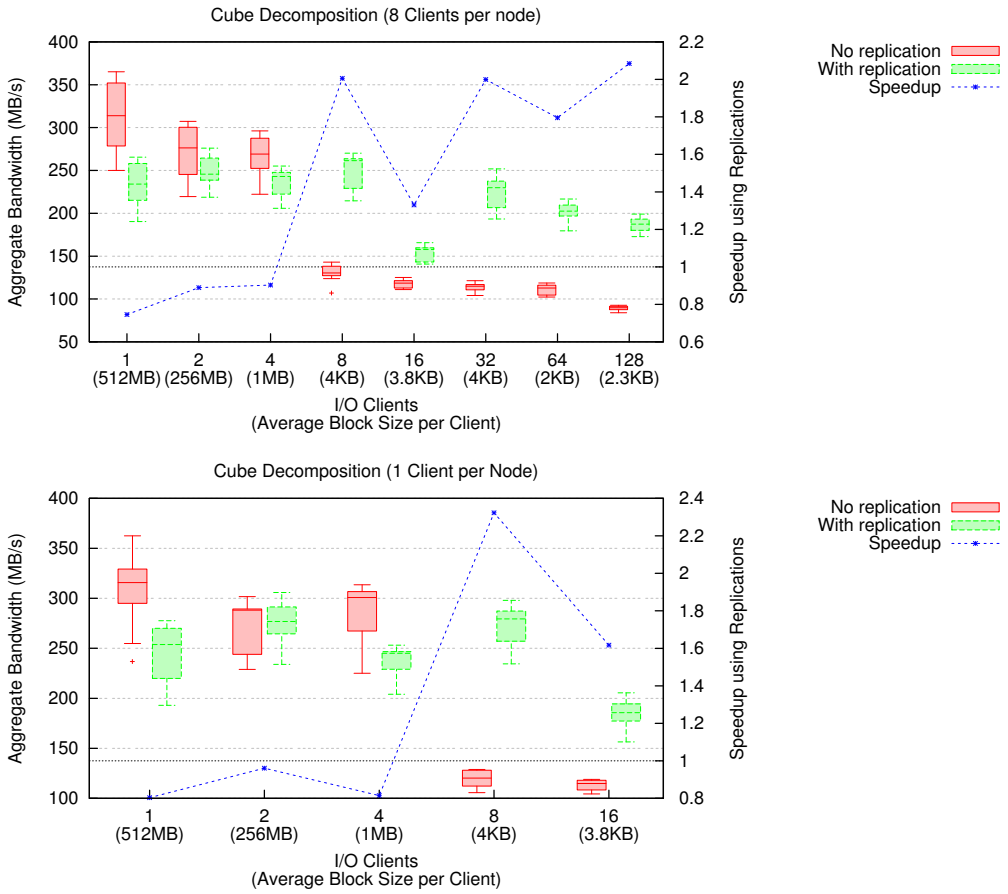


Figure 4.8: Cube-decomposition results with different process configurations

to collective I/O, due to the large-granularity block sizes induced by the two-phase algorithm (16MB). Performance for large numbers of clients drops likely because of underestimating the network capability when choosing the replica layout: see the following paragraphs on model performance.

Figure 4.10 shows the row-based decomposition performance, representing the "best case" for parallel I/O without reorganization: large, contiguous, non-overlapping blocks. Here, as the storage is the primary bottleneck and block sizes are very large, RADAR is not shown to have any benefit. We also show collective I/O results, but in this case performance is either unchanged or reduced due to the data movement overheads.

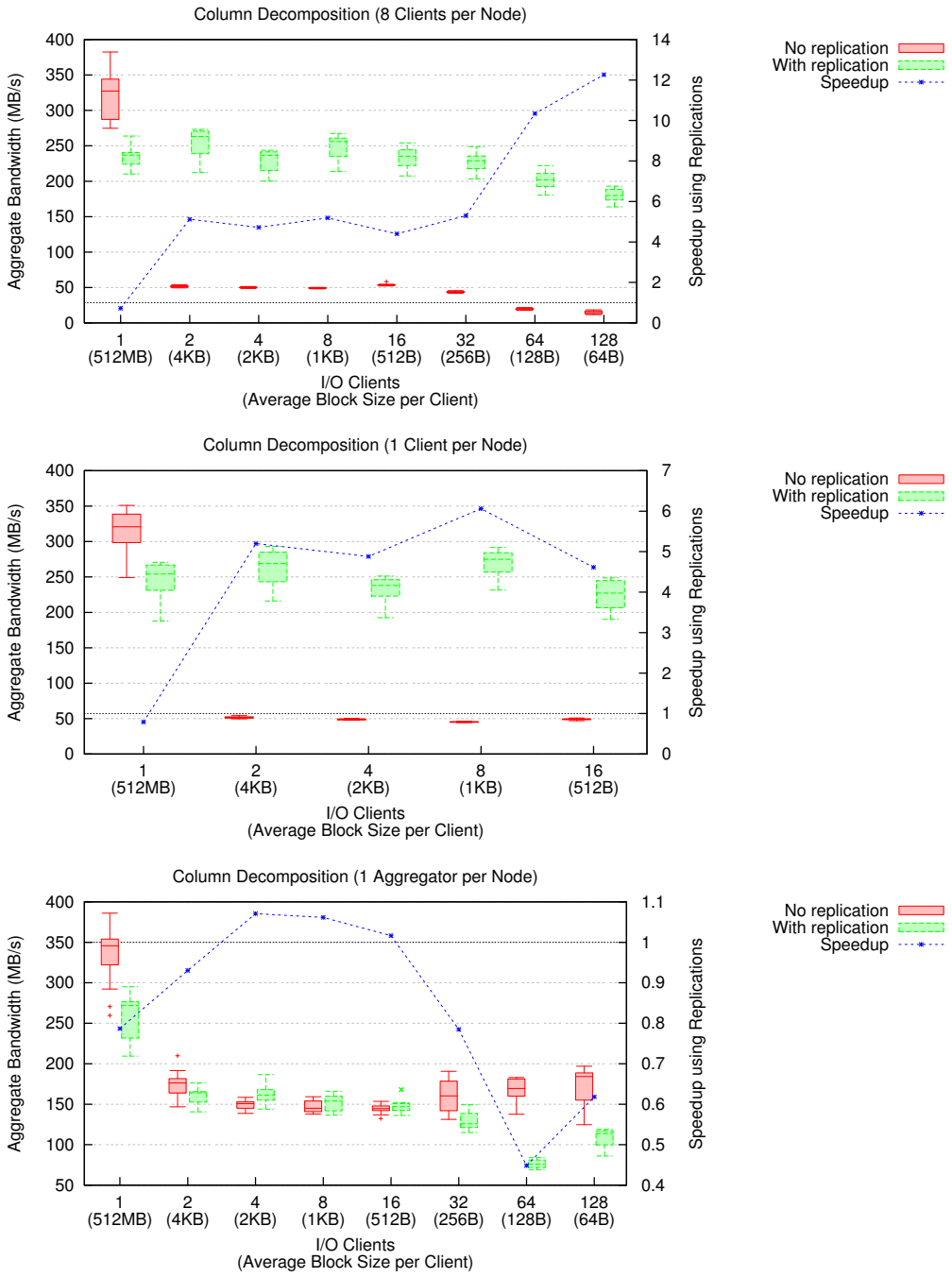


Figure 4.9: Column-decomposition results with different process configurations

4.4.5 Model Verifications

Given the performance shown in Section 4.4.4, we now look at how the performance modeling approach compares. As previously discussed, the goal of the performance models is to capture if

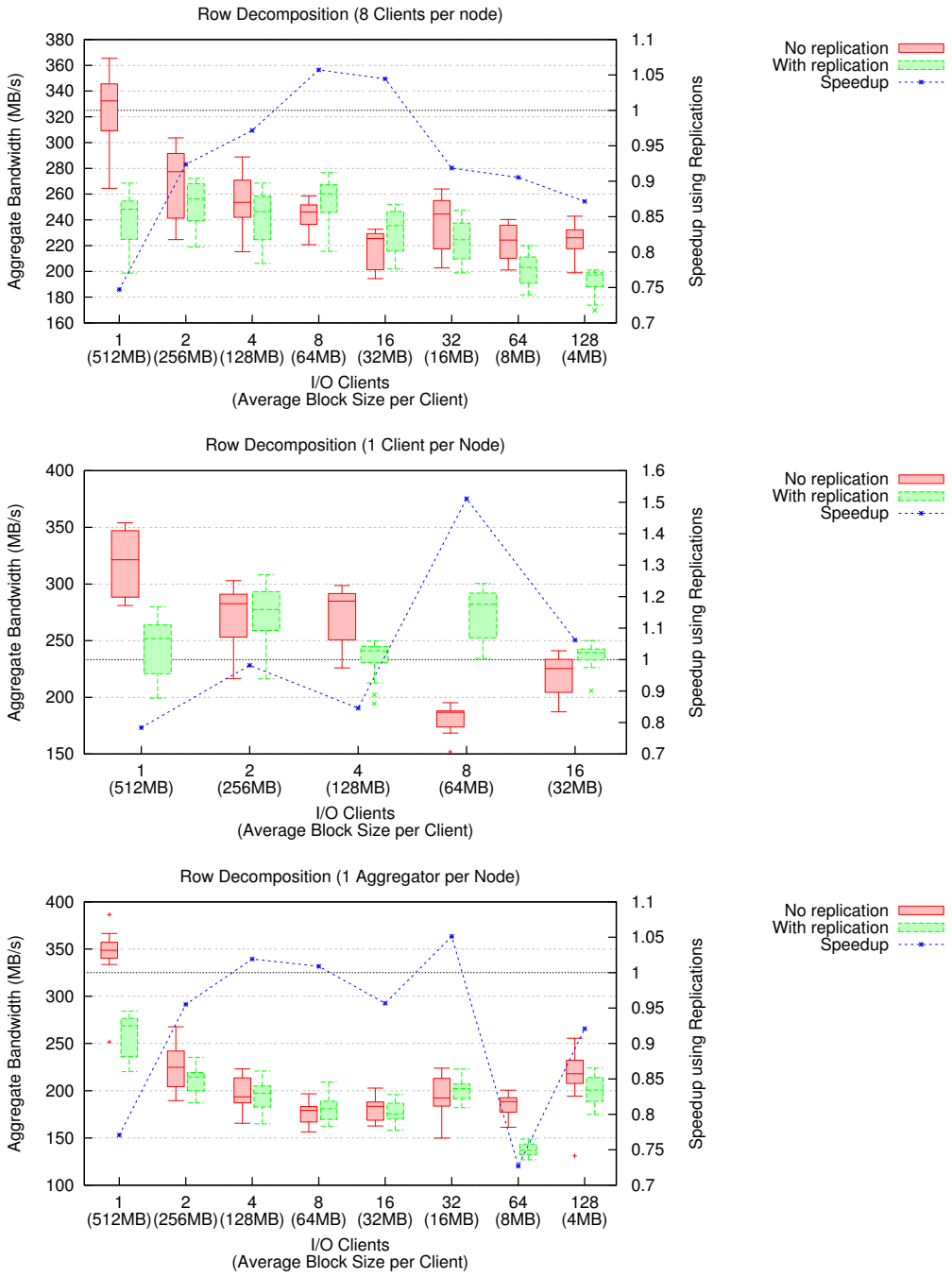


Figure 4.10: Row-decomposition results with different process configurations

a specific data layout can be improved by a modified data layout via replication. Note that this is a different goal than strict performance accuracy: here, the primary measurement of interest

is the accuracy of relative performance between two layouts (one with replicas, one without). As discussed, the models do not perform full system simulation, making them unsuitable for general-purpose performance prediction.

Figures 4.11, 4.12, and 4.13 show the results, comparing the model-derived performance of both the original layout and the layout under replication to the median of the performance shown in Section 4.4.4. We additionally show the estimated performance using the coarse-grained model corresponding to the best layout. In general, the "best layout" found corresponded to the heuristic of spreading each pattern's data across as many servers as possible until overlap occurs, in which case the distribution contracts accordingly. Also, as discussed before, results for time-based and cube-based collective decomposition are not shown.

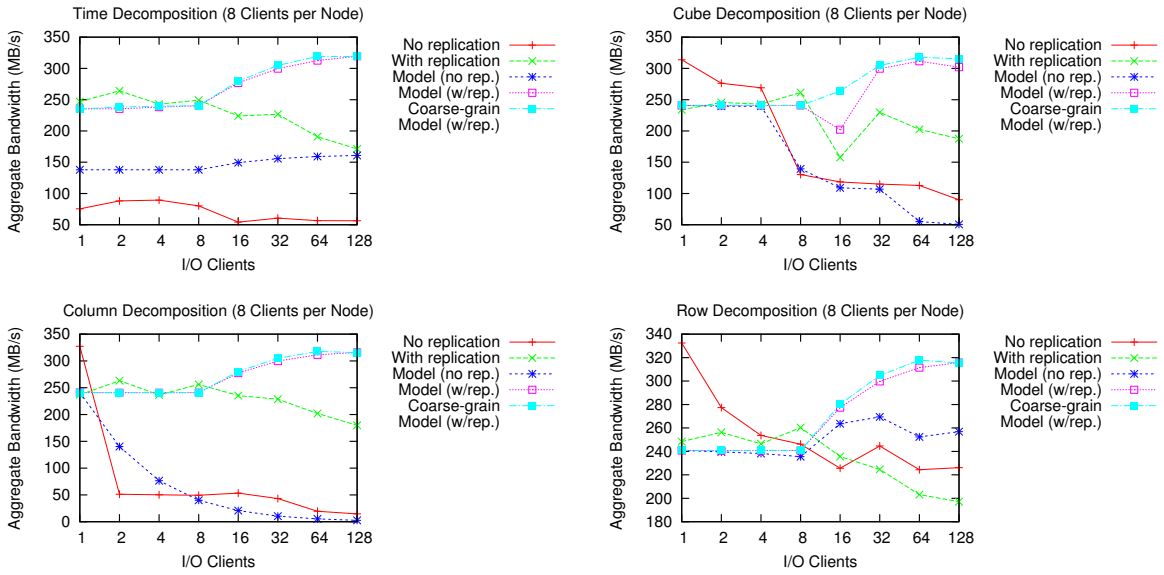


Figure 4.11: Model results against median empirical performance (8 clients per node).

Overall, the model results capture some degree of performance difference between the original layout and the candidate replica set layout. Large, contiguous accesses are shown to have little difference between both the original and replicated layouts, implying that creating the replica set would result in minimal, if any, gain. Alternatively, smaller, noncontiguous accesses are correctly shown to have a large degree of benefit by the models.

There are a few nontrivial aspects of the test system and software preventing the models from performing more accurately. First, there does not seem to be any performance benefit arising from the addition of nodes – the model assumes that additional nodes and servers corre-

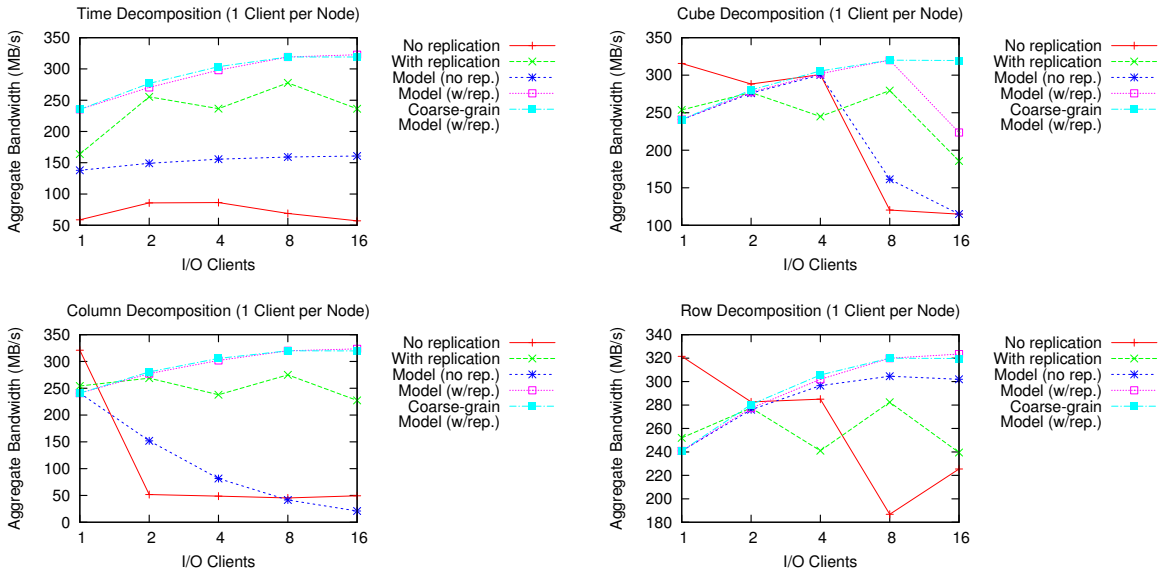


Figure 4.12: Model results against median empirical performance (1 clients per node).

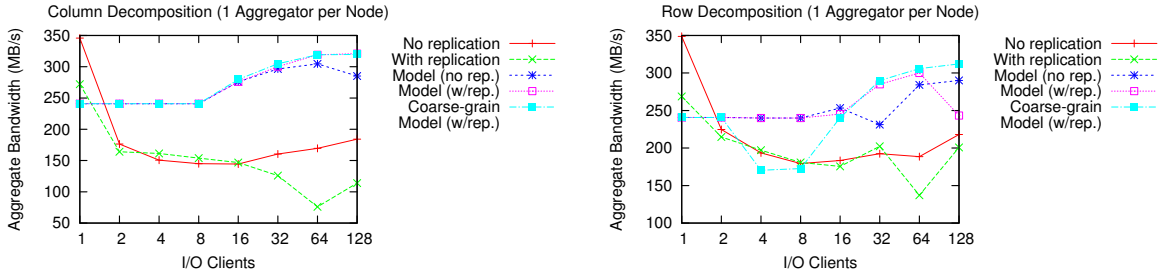


Figure 4.13: Model results against median empirical performance (1 aggregator per node).

sponds to additional network resources to draw upon. As seen in the time-based decomposition (fixed access sizes per client), this does not seem to be the case. Furthermore, the cost models generally overestimate the performance: we believe this to be largely the cause of our read-ahead simulation being optimistic in its ability to effectively cache pages without interfering with normal system performance. Finally, our performance models do not model the communication phase of optimizations performed during collective I/O, and hence overestimate the overall performance.

Table 4.3: Median replica entries per non-empty inverted list bin

Decomp.	Config.	Clients			
		16	32	64	128
Col	ind., 8/node	16.0	32.0	64.0	128.0
	ind., 1/node	16.0	N/A	N/A	N/A
	collective	N/A	1.0	2.0	4.0
Row	ind., 8/node	2.0	4.0	8.0	16.0
	ind., 1/node	2.0	N/A	N/A	N/A
	collective	N/A	N/A	1.5	2.0
Cube	ind., 8/node	6.0	8.0	16.0	36.0
	ind., 1/node	6.0	N/A	N/A	N/A
Time	ind., 8/node	1.0	1.0	1.0	1.0
	ind., 1/node	1.0	N/A	N/A	N/A

4.4.6 Replica Inverted List Performance

To test the efficacy of our inverted list approach to replica lookup, we measured the median entries-per-bin for all of the experiments shown in Section 4.4.4. Note that, for these measurements, bins having no entries were discarded from the computation – we wish to measure effectiveness only for regions of file accessed by the read benchmarks. Furthermore, we used an internal threshold of ten for enabling the inverted list, hence, no results for run configurations that did not produce enough replications.

The results can be seen in Table 4.3. For the individual I/O case of the read benchmarks tested, each process's read workload reduces to a multidimensional strided access pattern, and hence produces exactly one replica, meaning that the upper bound on the number of replicas is the core count. The effectiveness of the method depends on the degree of interleaving: fine-grained interleaving at the level of bytes, as seen by the column decomposition, obtain no benefit, as each replica touches all non-empty bins. By contrast, the other decompositions see a larger degree of benefit, especially the time-based decomposition, in which each process accesses a distinct, nonoverlapping portion of file with a large separating stride.

4.4.7 Performance with APLOD

To explore the performance implications of combining optimizations at multiple levels of the I/O software stack, we now show layered optimization, using both APLOD and RADAR in tandem to optimize partial-precision I/O on a subvolume. We use the same volume dimensions of $128 \times 256 \times 256 \times 256$, except instead of having 32-byte elements (corresponding to four C `doubles`), we have 8-byte elements (corresponding to an APLOD decomposition with CV $\{2, \ldots\}$). We read the modified volume using the given decompositions and performed the

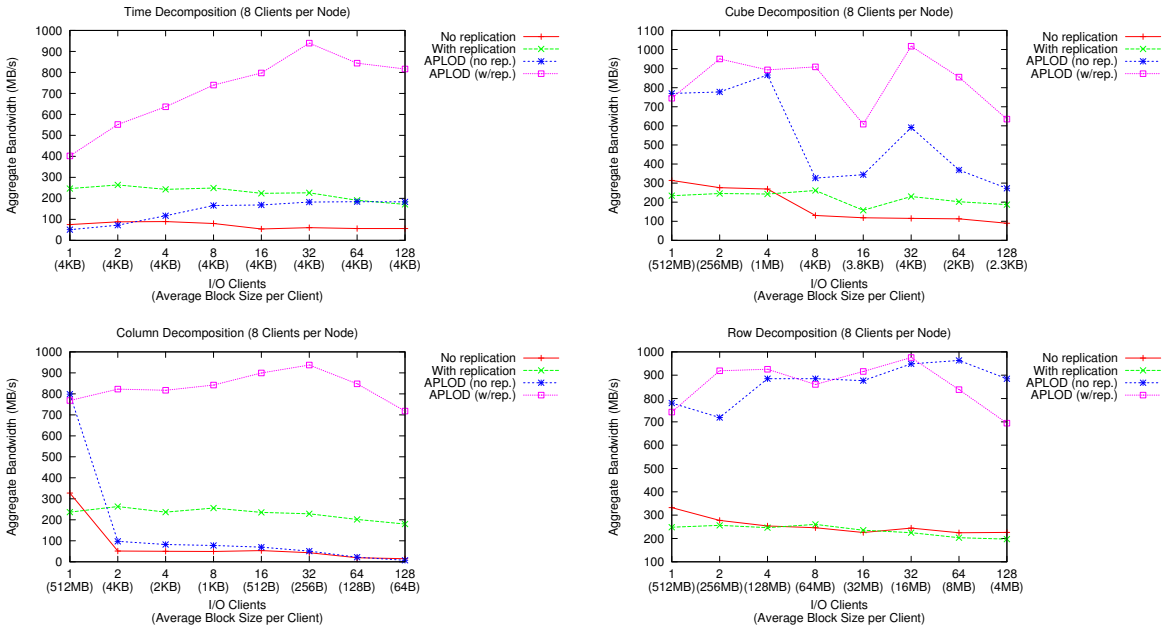


Figure 4.14: APLOD (two bytes precision) performance with and without RADAR. Note that the average block size for APLOD is one-fourth of the listed size.

APLOD reconstruction operation, reporting median aggregate bandwidth with respect to the original data size.

The APLOD read results are shown in Figure 4.14. As shown, the performance benefit from using APLOD alone depends on the contiguity of the data. The time-based and column-based decompositions show little benefit of using APLOD on the original data; the performance bottleneck in those cases is the noncontiguous access patterns, rather than the volume of data read. For the row-based and some cases of the cube-based decompositions, the performance benefit is roughly similar to the data reduction.

Where the APLOD method really shines is in the combination of the data reduction provided by the APLOD transform with the data reorganization provided by RADAR. This allows reads using APLOD to be performed under the best possible case: large, contiguous chunks.

4.5 Related Work

4.5.1 Replication in Storage Systems

Data replication in storage systems is a well-researched topic in many domains. Many parallel/distributed filesystems, such as Hadoop File System (HDFS) [91] and Google File System (GFS) [27, 64], built for task-centric, data-intensive workloads, as well as the Ceph filesys-

tem [112], have data replication as a first-order feature. Furthermore, for current filesystems used in HPC (e.g., PVFS [14], Lustre [89], PanFS [114]) where fault-tolerance is typically provided through hardware redundancy, data replication is beginning to be explored. For example, Tantisiriroj et al. developed a *shim* layer for PVFS for the purpose of executing Hadoop workloads, enabling readahead buffering, making striping information available to the Hadoop scheduler, and modifying PVFS's data placement policy to provide data replication [98]. In addition, local filesystem replication has been explored in local filesystems in a performance context by reorganizing data to minimize rotational latency and maximize locality [5, 34].

Database systems also widely use replication, both as a fault-tolerance method similar to filesystems, and as a performance optimizer by replicating "chunks" of the database using query history as a guide [68, 116]. Also, recent works in MonetDB [6] have enabled dynamic replication of columns via a method called *database cracking* [36, 35], which replicates columns and reorganizes/indexes the replicated data as queries are performed on it.

Finally, other areas such as *on-demand* (e.g., video) services [19, 46] and Data Grid systems [52, 96] have used replication to maximize data availability and throughput.

4.5.2 I/O Middleware and User-level Replication

Recently, the use of replication to ensure high availability and/or improve performance have been explored outside of the storage system; that is, as either new high-level libraries or alongside I/O middleware. Son et al. [93] show the possibility of handling replication-based fault tolerance at the middleware level, performing file block replication using the PMPI interface. For MPI-based applications that use a one-to-one, process-to-file mapping, Song et al. [94] creates three replications with different file-to-disk mappings: file-per-storage-node, full striped files, and partially striped files. Yin et al. extended Song's model to handle one-dimensional and two-dimensional strided accesses [122]. Zhang et al. [124] used replication through PMPI to minimize disk head thrashing by playing back local disk traces with DiskSim [8].

4.5.3 Capturing and Detecting I/O Access Patterns

As gathering and systematically deriving system usage information from an application or set of applications is a highly important task, many solutions have been developed that attack the problem from a multitude of directions. Note that the majority of these methods cover more areas than just I/O. As Numerous methods have been developed to work specifically on MPI and PMPI (MPI's profiling interface), such as the MPI Parallel Environment (MPE) [15] for full MPI event tracing and mpiP [105] for lightweight, statistical measures. Dynamic instrumentation methods include automatically instrumenting at compile time through source code analysis [45], as well as runtime binary instrumentation through IOPin [44], based on

the Pin [60] framework. The Scalatrace family of MPI tracers focus on compressed trace generation [70, 80, 106, 119], using histogram generation and a combination of intra-node and inter-node trace compression. Additionally, Darshan [13, 12] focuses on *center-wide* usage patterns by combining local, subsystem metrics (such as block device profiling with the Sysstat [28] and fsstat [20] tools) and application-level metrics (instrumented through POSIX and MPI-IO).

Once acceptable profiles or logs of application/system performance is gathered, they can be mined for emergent patterns. Statistical learning methods can be used in a general sense to capture high level patterns such as block-to-block association [62, 76, 103, 5]. Recent methods specifically for HPC have been developed, again typically through the MPI/MPI-IO layers. For example, Byna et al. developed an MPI-based I/O prefetching methodology based on detecting multidimensional striding patterns of varying sizes [11]. Also, He et al. investigated PLFS index compression using pattern recognition under a checkpointing use-case [33]. The IOSig [11] trace analyzer convert I/O operations to compact and parameterized representations called I/O signatures using a *template matching* approach, which iteratively attempts to match specific patterns (e.g., regularly strided) to the sequence of I/O accesses.

4.6 Conclusion

Effective data distribution in large-scale analysis systems is an integral component of achieving high performance I/O, especially in the presence of complex, noncontiguous workloads such as the volume decompositions we have presented. Through the tight coupling with a filesystem view of the data as a set of distinct objects, we were able to create arbitrary data layouts optimized for the access patterns induced on the dataset, all in a single container. RADAR is a promising step in the direction of automated specialization of data layouts based on application-specific needs and access patterns, providing both increased performance and an initial ability to reason about the "worth" of layouts for the purpose of marshalling usage of limited space for optimized data distributions. Finally, we have shown that optimization-aware tracing methodologies (i.e., MPI-IO two-phase aware) can be an effective tool for adaptive layout optimization works, alleviating the optimization layer from having a deep understanding of intermediate optimizations.

Chapter 5

Conclusion

5.1 Future Work

Our work in optimizations grounded in and related to noncontiguous access patterns has hit upon multiple layers of the I/O software stack, from the application level, through high-level I/O libraries and I/O middleware, and finally touching upon the filesystem layer. Our planned future work primarily touches upon the latter two components, being concerned with extensions and improvements to the RADAR framework (at both middleware and filesystem levels) and, informed by our experiences with the RADAR work, more general directions in storage systems and HPC filesystems research. We additionally plan to research advanced topics related to data transformations (specifically APLOD), their place in the I/O software stack, and ways to improve integration and interoperability with the existing HPC I/O ecosystem.

5.1.1 RADAR Future Work

For RADAR in particular, there are a number of future research directions that would aid in solidifying the framework and enabling the maximum flexibility and performance benefits.

Filesystem-driven Replication Operations As a significant methodological change, allowing the parallel filesystem to be in charge of the data movement aspects of the framework, that is, the creation of replicas in storage given user (or RADAR) input would be an invaluable addition to both the RADAR framework, and to other future work (see Section 5.1.2). The primary benefit of doing so, asides from easing the client-side load and converting the replication creation operations from an active client program to a passive filesystem operation, is that the filesystem may perform the operations at times of low load. As shown by Carns et al. [13, 12], there is ample opportunity for the filesystem to perform layout optimizations without interrupting normal operation, as filesystems typically do not sustain peak utilization

over long periods of time (if BlueGene/P activity over the period studied is indicative of typical parallel filesystem operation). Hence, the use of idle time to perform layout optimizations, such as those explored in RADAR, is a promising direction.

Investigation of RADAR Task Separation Related to the previous point, an open question for dynamic data layout optimizations such as those performed by RADAR is what the proper separation of tasks in a multi-data-source environment should be. In the RADAR work, we directly use filesystem abstractions (via object-storage), but much of the heavy lifting occurs at the I/O middleware level. To summarize the components of RADAR, there is an observational component (i.e., tracing and trace analysis) a decision component (i.e., deciding what replicas to create and how), and a storage component (i.e., storage policy of replicas and data using EOF, as well as means to access them programmatically). Research can be performed in identifying how these general categories can be executed at various levels of the I/O stack. For instance, synchronization of information such as replication metadata, currently guaranteed (though not explicitly enforced) by MPI file access consistency semantics, could be pushed to the filesystem level. Furthermore, giving the filesystem semantic knowledge of replications enables the fusion of replicas-as-optimizers and replicas-as-fault-tolerance.

Incorporate Runtime Usage Statistics Into Replica Selection There are a few pieces of information not yet incorporated into the RADAR layout manager that could prove very useful for a more heterogeneous set of workloads. Namely, these pieces are frequency of access to existing replicas and performance. Currently, performance of reading replicas is captured implicitly through the performance modeling step of the layout manager, but having real usage numbers could prove to be a more accurate solution, though steps would need to be taken to mitigate deviations in performance for reasons out of the user's control – storage systems are shared resources, after all. Frequency of access can also help refine the aging process, as currently a replica that is accessed frequently is aged in the same manner as a replica that is rarely accessed.

Inline Trace Analysis Currently, the output size and thus overhead of the trace methodology is dependent on the total number of I/O requests made. For long-running applications with large numbers of processes and I/O requests, it would be more feasible from both a memory standpoint and a write-performance standpoint to incorporate inline trace compression. This can take the form of access pattern discovery via IOSig, application of other, related works, such as the ScalaTrace family of trace collection/compression techniques [70, 106, 119, 80], as well as general-purpose compression techniques.

5.1.2 Storage System Future Work

Our experiences in RADAR have inspired us to look more deeply into filesystem semantics concerning replication, especially at the intersection of performance-enhancing replication policies and replication policies meant to mitigate the various types of software/hardware failures that occur in large-scale storage systems. This requires a in-depth look into replication and I/O server communication protocols, as adding asymmetry into redundant data stores has the potential of breaking up the one-to-one mapping between the location of a chunk of data and the location of its replicas that is assumed in most technologies today.

By combining both replication-based fault-tolerance protocols and heterogeneous data layouts, the data redundancy can be used in multiple, performance-enhancing contexts. As a simple example, different data striping parameterizations can be used, which, as shown in the RADAR work, can result in a higher degree of performance. A more complex example is maintaining write-optimized data (e.g., log structured in the vein similar to the format in the Parallel Log-structured File System, or PLFS [4]) while maintaining read-optimized data in a replica. All of these are promising avenues of research, and could come in the form of user-provided hints and flags to file operations or as part of a feedback loop similar to what RADAR does. The primary research questions are how to ensure synchronicity of data as well as how to design replication protocols to handle both forwarding data reads/writes to their correct locations as well as to gracefully cope with software/hardware faults.

5.1.3 Advanced Data Transform Techniques in the I/O Software Stack

Two significant problems when considering the application of data transformation methodologies such as APLOD, compression, etc. on scientific datasets is 1) how the physical representation of transformed data on disk is accessed by users in a simple and effective manner, and 2) how transformed data interoperates with other libraries and data formats. In the worst case, the transformed data and necessary metadata can be stored on disk using custom POSIX-based I/O drivers, requiring all applications and libraries interacting with the data to support the custom format. With an already large ecosystem for processing scientific data in an HPC context, this approach is likely not scalable.

Given these concerns, it is of utmost importance to develop *generic* data transformation frameworks that decouple a dataset's *logical* view of the data from the *physical* view. This is certainly not a new concept (PLFS abstracts per-process log-structured files away to present a clean, "shared" view of a file for write optimization, and databases are designed specifically around this concept), but there are a few HPC-specific areas for which more research is necessary. For example, the high-level I/O libraries ADIOS [59], HDF5 [120], and PnetCDF [56] all expose array-based models with varying levels of complexity, and the logical dataset view

(multidimensional, time-varying arrays) are translated by the libraries into a layout in physical storage (e.g., via MPI-IO). The "next level" of this idea is to further decouple the data models and allow advanced data transformations within the libraries themselves. By incorporating operations such as compression as plugins that sit between the logical and physical data layouts, interoperability becomes easier (these libraries are widely used and supported), data transformation roles become clear (the high-level I/O library is in charge of all transforms), and the burden on end-users to support the transforms are minimal (merely enable the transform when configuring the dataset). Furthermore, the design of a user-configurable plugin architecture within said libraries has the potential to give users an effective combination of ease-of-use (defining the logical structure, enabling/disabling the transform) and data control (developing or using existing plugins to allow fine-grained, application-specific augmentation of data).

To this end, we are currently investigating transform frameworks in the ADIOS library; we are developing a plugin architecture to allow operations including general-purpose compression and level-of-detail methods such as APLOD. Given ADIOS's architecture (see Section 2.3.2), such plugins can be easily enabled by the user through addition of a single XML entry.

REFERENCES

[1] Top 500 supercomputing sites. `http://www.top500.org`.

[2] IEEE standard for floating-point arithmetic. *IEEE Standard 754-2008*, 2008.

[3] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 39–48, New York, NY, USA, 2009. ACM.

[4] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[5] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization for self-optimizing storage systems. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 183–196, Berkeley, CA, USA, 2009. USENIX Association.

[6] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Communications of the ACM*, 51:77–85, December 2008.

[7] Nathan Brookwood. AMD fusion family of APUs: Enabling a superior, immersive PC experience. *Insight*, 64:1–8, 2010.

[8] John S. Bucy, Jiri Schindler, Steven Schlosser, Gregory Ganger, and Contributors. The disksim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University Parallel Data Lab, 2008.

[9] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *Proceedings of the Data Compression Conference (DCC)*, pages 293–302, 2007.

[10] M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, 2009.

[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 *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[12] 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 Transactions on Storage (TOC)*, 7(3):8:1–8:26, October 2011.

[13] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *IEEE International Conference on Cluster Computing*, Cluster'10, pages 1–10, 2009.

[14] Philip H. Carns, Walter B. Ligon, III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *In Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.

[15] Anthony Chan, William Gropp, and Ewing Lusk. User's guide for MPE: Extensions for MPI programs. Technical Report ANL/MVS-TM-ANL-98/xx, Argonne National Laboratory, 2003.

[16] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 534–542, 2001.

[17] J.H. Chen, A. Choudhary, B.De. Supinski, M. DeVries, E.R. Hawkes, S. Klasky, W.K. Liao, K.L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C.S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(015001), 2009.

[18] Avery Ching, Alok Choudhary, Wei-Keng Liao, Rob Ross, and William Gropp. Noncontiguous i/o through pvfs. In *IEEE International Conference on Cluster Computing*, pages 405–414, 2002.

[19] Tat-Seng Chua, Jiandong Li, Beng-Chin Ooi, and Kian-Lee Tan. Disk striping strategies for large video-on-demand servers. In *Proceedings of the Fourth ACM International Conference on Multimedia*, MULTIMEDIA '96, pages 297–306, New York, NY, USA, 1996. ACM.

[20] Shobhit Dayal. Characterizing HEC storage systems at rest. Technical Report CMU-PDL-09-109, Carnegie Mellon University Parallel Data Laboratory.

[21] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C.S. Chang, S.H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. ISOBAR preconditioner for effective and high-throughput lossless data compression. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 138–149, 2012.

[22] Zhe Fan, Feng Qiu, and Arie E. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum*, 27(2):341–350, 2008.

[23] M.W. Frazier. *An Introduction to Wavelets through Linear Algebra*. Springer-Verlag, 1999.

[24] Jean-loup Gailly and Mark Adler. Zlib general purpose compression library. http://zlib.net/, Jan. 2012.

[25] M. Galassi et al. *GNU Scientific Library Reference Manual*, 3rd edition. ISBN:0954612078.

[26] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ASPLOS '10 Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 347–358, 2010.

[27] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[28] Sebastien Godard. Sysstat utilities home page. http://sebastien.godard.pagesperso-orange.fr/index.html.

[29] Zhenhuan Gong, David A. Boyuka II, 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 *the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'13)*, Delft, The Netherlands, 2013.

[30] 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 *Proceedings of the 41st International Conference on Parallel Processing*, ICPP '12, 2012.

[31] David Goodell, Seong Jo Kim, Robert Latham, Mahmut Kandemir, and Robert Ross. An evolutionary path to object storage access. In *Proceedings of the Seventh Workshop on Parallel Data Storage*, PDSW '12, 2012.

[32] Peter J. Haas and Christian König. A bi-level bernoulli scheme for database sampling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 275–286, New York, NY, USA, 2004. ACM.

[33] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. Discovering structure in unstructured I/O. In *Proceedings of the Seventh Workshop on Parallel Data Storage*, PDSW '12, 2012.

[34] Hai Huang, Wanda Hung, and Kang G. Shin. Fs2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 263–276, New York, NY, USA, 2005. ACM.

[35] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels*. PhD thesis, University of Amsterdam, 2010.

[36] Stratos Idreos, Martin Kersten, and Stefan Manegold. Database cracking. In *Proceedings of the 3rd International Conference on Innovative Data Systems Research*, CIDR'07, 2007.

[37] Interleaved or random (IOR) parallel filesystem I/O benchmark.

[38] Dana A. Jacobsen, Julien C. Thibault, and Inanc Senocak. An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. In *Proceedings of the 48th AIAA Aerospace Sciences Meeting*, 2010.

[39] John Jenkins, Isha Arkatkar, Sriram Lakshminarasimhan, Neil Shah, Eric R. Schendel, Stéphane Ethier, Choong-Seock Chang, Jacqueline H. Chen, Hemanth Kolla, Scott Klasky, Robert B. Ross, and Nagiza F. Samatova. Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In *Database and Expert Systems Applications (DEXA)*, pages 16–30, 2012.

[40] John Jenkins, Isha Arkatkar, Sriram Laksminarasimhan, David A. Boyuka II, Eric R. Schendel, Neil Shah, Stephane Ethier, C.S. Chang, Jackie Chen, Hemanth Kolla, Scott Klasky, Robert Ross, and Nagiza F. Samatova. ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. *Transactions on Large Scale Data and Knowledge Centered Systems (TLDKS)*, 8220:95–114, 2013.

[41] John Jenkins, Eric Schendel, Sriram Lakshminarasimhan, David A. Boyuka II, Terry Rogers, Stephane Ethier, Robert Ross, Scott Klasky, and Nagiza F. Samatova. Byte-precision level of detail processing for variable precision analysis. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, Salt Lake City, UT, USA, 2012.

[42] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. Otoo, V. Perevoztchikov, A. Poskanzer, Prabhat, O. Rubel, A. Shoshani, A. Sim, K. Stockinger, G. Weber and W-M Zhang. Fastbit: interactively searching massive data. *Journal of Physics: Conference Series*, 180(1):012053, 2009.

[43] Khronos OpenCL Working Group. *The OpenCL Specification Version 1.1*. Khronos Group, 2011. `http://www.khronos.org/opencl/`.

[44] Seong Jo Kim, Seung Woo Son, Wei-keng Liao, Mahmut Kandemir, Rajeev Thakur, and Alok Choudhary. IOPin: Runtime profiling of parallel I/O in HPC systems. In *7th Parallel Data Storage Workshop*, PDSW'12, 2012.

[45] Seong Jo Kim, Yuanrui Zhang, Seung Woo Son, Ramya Prabhakar, Mahmut Kandemir, Christina Patrick, Wei-keng Liao, and Alok Choudhary. Automated tracing of i/o stack. In *Proceedings of the 17th European MPI Users Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'10, pages 72–81, Berlin, Heidelberg, 2010. Springer-Verlag.

[46] Jan H.M. Korst. Random duplicated assignment: An alternative to striping in video servers. In *Proceedings of the Fifth ACM International Conference on Multimedia*, MULTIMEDIA '97, pages 219–226. ACM, 1997.

[47] 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, HPDC '13, pages 1–12, New York, NY, USA, 2013. ACM.

[48] Sriram Lakshminarasimhan, John Jenkins, Isha Arkatkar, Zhenhuan Gong, Hemanth Kolla, Seung-Hoe Ku, Stephane Ethier, Jackie Chen, C. S. Chang, Scott Klasky, Robert Latham, Robert Ross, and Nagiza F. Samatova. ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC, pages 31:1–31:11, New York, NY, USA, 2011. ACM.

[49] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Robert Latham, Robert Ross, and Nagiza F. Samatova. Compressing the incompressible with ISABELA: in-situ reduction of Spatio-Temporal data. In *17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)*, Bordeaux, France, 8 2011.

[50] Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Seung-Hoe Ku, C. S. Chang, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. Isabela for effective in situ compression of scientific data. *Concurrency and Computation: Practice and Experience*, 25(4):524–540, 2013.

[51] Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of the Conference on Visualization*, pages 355–361, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[52] Houda Lamehamedi, Boleslaw Szymanski, Zujun Shentu, and Ewa Deelman. Data replication strategies in grid environments. In *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, ICA3PP'02, pages 378–383. IEEE, 2002.

[53] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 40:1–40:12, New York, NY, USA, 2009. ACM.

[54] Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms, and William Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 40:1–40:12, New York, NY, USA, 2009. ACM.

[55] O.S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *IEEE Cluster PPAC Workshop*, pages 1–8, 2009.

[56] Jianwei Li, W.K. Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, page 39, New York, NY, USA, 2003. ACM.

[57] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.

[58] Ning Liu, Jason Cope, Phillip Carns, Cristopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST)*, pages 1–11, 2012.

[59] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the International Workshop on Challenges of Large Applications in Distributed Environments (CLADE)*, pages 15–24. ACM, 2008.

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[61] Kwan Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Application*, pages 14–19, 2009.

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

[63] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proc. of the 2011 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[64] Marshall Kirk McKusick and Sean Quinlan. GFS: Evolution on fast-forward. *Queue*, 7(7):10:10–10:20, August 2009.

[65] MPI Forum. MPI-2: Extensions to the message-passing interface. Technical report, Univ. of Tennessee, Knoxville, 1996.

[66] Parallel I/O benchmarking consortium. `http://www.mcs.anl.gov/research/projects/pio-benchmark/`.

[67] Ramanathan Narayanan, Berkin Özışkyılmaz, Gokhan Memik, Alok Choudhary, and Joseph Zambreno. Quantization error and accuracy-performance tradeoffs for embedded data mining workloads. In *Proceedings of the International Conference on Computational Science (ICCS)*, pages 734–741, Berlin, Heidelberg, 2007. Springer-Verlag.

[68] Sivaramakrishnan Narayanan, Umit Catalyurek, Tahsin Kurc, Vijay Shiv Kumar, and Joel Saltz. A runtime framework for partial replication and its application for on-demand data exploration. In *High Performance Computing Symposium, SCS Spring Simulation Multiconference*, HPC '05, 2005.

[69] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern CPUs and GPUs. In

Proc. of the 2010 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis, pages 1–13, 2010.

[70] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, August 2009.

[71] Nvidia CUDA compute unified device architecture. `http://developer.nvidia.com/category/zone/cuda-zone`.

[72] F. Olken. *Random sampling from databases*. PhD thesis, University of California, 1993.

[73] F. Olken and D. Rotem. Random sampling from B+ trees. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 269–277, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[74] F. Olken and D. Rotem. Sampling from spatial databases. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 199–208, 1993.

[75] Frank Olken and Doron Rotem. Simple random sampling from relational databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 160–169, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[76] 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*, ICS '02, pages 147–155, New York, NY, USA, 2002. ACM.

[77] OpenACC directives for accelerators, 2011. `http://www.openacc-standard.org`.

[78] Valerio Pascucci and Randall J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, page 2, New York, NY, USA, 2001. ACM.

[79] A. Pope, S. Habib, Z. Lukic, D. Daniel, P. Fasel, K. Heitmann, and N. Desai. The accelerated universe. *Computing in Science Engineering*, 12(4):17 –25, july-aug. 2010.

[80] Prasun Ratn, Frank Mueller, Bronis R. de Supinski, and Martin Schulz. Preserving time in large-scale communication traces. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 46–55, New York, NY, USA, 2008. ACM.

[81] Robert Ross, Neill Miller, and William Gropp. Implementing fast and reusable datatype processing. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 404–413. Springer Berlin / Heidelberg, 2003.

[82] S. Ku, C.S. Chang, and P.H. Diamond. Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion*, 49(115021), 2009.

[83] Ravi Samtaney, Deborah Silver, Norman Zabusky, and Jim Cao. Visualizing features and tracking their evolution. *Computer*, pages 20–27, 1994.

[84] Andreas Schafer and Dietmar Fey. High performance stencil code algorithms for GPGPUs. In *International Conference on Computational Science (ICCS)*, 2011.

[85] E.R. Schendel, Ye Jin, N. Shah, J. Chen, C.S. Chang, Seung-Hoe Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N.F. Samatova. Isobar preconditioner for effective and high-throughput lossless data compression. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 138 –149, april 2012.

[86] Eric R. Schendel, Saurabh V. Pendse, John Jenkins, David A. Boyuka, Zhenhuan Gong, Sriram Lakshminarasimhan, Qing Liu, Hemanth Kolla, Jackie Chen, Scott Klasky, Robert Ross, and Nagiza F. Samatova. ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization. In *Proceedings of the ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 61–72, 2012.

[87] Eric R. Schendel, Saurabh V. Pendse, John Jenkins, David A. Boyuka, II, Zhenhuan Gong, Sriram Lakshminarasimhan, Qing Liu, Hemanth Kolla, Jackie Chen, Scott Klasky, Robert Ross, and Nagiza F. Samatova. Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 61–72, New York, NY, USA, 2012. ACM.

[88] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, 2002. USENIX Association.

[89] Philip Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, 2003.

[90] Frank Shorter. Design and analysis of a performance evaluation standard for parallel file systems. Master's thesis, Clemson University, 2003.

[91] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[92] Deborah Silver and Xin Wang. Tracking and visualizing turbulent 3D features. *IEEE Transactions on Visualization and Computer Graphics*, pages 129–141, 1997.

[93] Seung Woo Son, Robert Latham, Robert Ross, and Rajeev Thakur. Reliable MPI-IO through layout-aware replication. 2011.

[94] Huaiming Song, Yanlong Yin, Yong Chen, and Xian-He Sun. A cost-intelligent application-specific data layout scheme for parallel file systems. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 37–48, New York, NY, USA, 2011. ACM.

[95] Huaiming Song, Yanlong Yin, Xian-He Sun, Rajeev Thakur, and Samuel Lang. A segment-level adaptive data layout scheme for improved load balance in parallel file systems. In *Cluster, Cloud and Grid Computing (CCGrid), IEEE/ACM International Symposium on*, pages 414–423, 2011.

[96] Heinz Stockinger, Asad Samar, Koen Holtman, Bill Allcock, Ian Foster, and Brian Tierney. File and object replication in data grids. *Cluster Computing*, 5(3):305–314, 2002.

[97] Jeff A. Stuart and John D. Owens. Message passing on data-parallel architectures. In *Proc. of the 23rd IEEE Int's Parallel and Distributed Processing Symposium*, May 2009.

[98] Wittawat Tantisiriroj, Seung Woo Son, Swapnil Patil, Samuel J. Lang, Garth Gibson, and Robert B. Ross. On the duality of data-intensive file system design: Reconciling HDFS and PVFS. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'11, pages 67:1–67:12, New York, NY, USA, 2011. ACM.

[99] David Taubman and Michael Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

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

[101] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '96, pages 180–187, Washington, DC, USA, 1996. IEEE Computer Society.

[102] Rajeev Thakur, Robert Ross, Ewing Lust, and William Gropp. Users guide for ROMIO: A high-performance, portable mpi-io implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[103] Nancy Tran and Daniel A. Reed. Automatic ARIMA time series modeling for adaptive I/O prefetching. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):362–377, April 2004.

[104] A. Trott, R. Moorhead, and J. McGinley. Wavelets applied to lossless compression and progressive transmission of floating point data in 3-D curvilinear grids. In *Proceedings of the Conference on Visualization*, pages 385–388, 1996.

[105] Jeffrey S. Vetter and Michael O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 123–132, New York, NY, USA, 2001. ACM.

[106] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 26–31, New York, NY, USA, 2009. ACM.

[107] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward. Extending the posix i/o interface: A parallel file system perspective. Technical report, Argonne National Laboratory, 2008.

[108] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E. Papka. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.

[109] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Xiangyong Ouyang, Sayantan Sur, and Dhabaleswar K. Panda. Optimized non-contiguous MPI datatype communication for GPU clusters: Design, implementation and evaluation with MVAPICH2. In *IEEE International Conference on Cluster Computing*, Cluster '11, 2011.

[110] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K. Panda. MVAPICH2-GPU: Optimized GPU to GPU communication for infiniband clusters. In *International Supercomputing Conference*, ISC '11, 2011.

[111] W.X. Wang, Z. Lin, W.M. Tang, W.W. Lee, S. Ethier, J.L.V. Lewandowski, G. Rewoldt, T.S. Hahm, and J. Manickam. Gyro-kinetic simulation of global turbulent transport properties in Tokamak experiments. *Physics of Plasmas*, 13(092505), 2006.

[112] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[113] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmermann, and Thomas Ertl. Level-of-detail volume rendering via 3D textures. In *Proceedings of the IEEE Symposium on Volume Visualization*, pages 7–13, New York, NY, USA, 2000. ACM.

[114] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association.

[115] B. Welton, D. Kimpe, J. Cope, C.M. Patrick, K. Iskra, and R. Ross. Improving I/O forwarding throughput with data compression. In *International Conference on Cluster Computing*, CLUSTER '11, pages 438–445. IEEE, 2011.

[116] Li Weng, Umit Catalyurek, Tahsin Kurc, Gagan Agrawal, and Joel Saltz. Servicing range queries on multidimensional datasets with partial replicas. In *IEEE International Symposium on Cluster Computing and the Grid*, volume 2 of *CCGrid '05*, pages 726–733. IEEE, 2005.

[117] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, second edition, 1999.

[118] J. Woodring, S. Mniszewski, C. Brislawn, D. DeMarle, and J. Ahrens. Revisiting wavelet compression for large-scale climate data using JPEG 2000 and ensuring data precision. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 31–38, 2011.

[119] Xing Wu, Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C. Roth. Probabilistic communication and I/O tracing with deterministic replay at scale. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 196–205, Washington, DC, USA, 2011. IEEE Computer Society.

[120] MuQun Yang, Robert E. McGrath, and Mike Folk. HDF5 - a high performance data format for earth science. In *Proceedings of the International Conference on Interactive Information Processing Systems (IIPS) for Meteorology, Oceanography and Hydrology*, 2005.

[121] Yanlong Yin, Surenda Byna, Huaiming Song, Xian-He Sun, and Rajeev Thakur. Boosting application-specific parallel i/o optimization using IOSIG. In *Cluster, Cloud and Grid Computing (CCGrid)*, pages 196–203, 2012.

[122] 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 *IEEE International Symposium on Parallel and Distributed Computing*, IPDPS'13, pages 345–356, 2013.

[123] Hongfeng Yu, Chaoli Wang, R.W. Grout, J.H. Chen, and Kwan Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Application*, pages 45–57, 2010.

[124] 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*, ICS '10, pages 223–232, New York, NY, USA, 2010. ACM.

[125] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Scott Klasky, Qing Liu, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. PreDatA–preparatory data analytics on peta-scale machines. In *IPDPS*, Atlanta, GA, April 2010.

[126] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. PreDatA: Preparatory data analytics on peta-scale machines. In *IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'10, pages 1–12, April 2010.

[127] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), July 2006.