Book author

Book title goes here

List of Figures

List of Tables

Contents

Chapter 1

In Situ Exploratory Data Analysis for Scientific Discovery

Sriram Lakshminarasimhan
North Carolina State University

Kanchana Padmanabhan
North Carolina State University

Zhenhuan Gong
North Carolina State University

John Jenkins
North Carolina State University

Neil Shah
North Carolina State University

Eric Schendel
North Carolina State University

Isha Arkatkar
North Carolina State University

Rob Ross
Argonne National Laboratory

Scott Klasky
Oak Ridge National Laboratory

Nagiza F. Samatova
North Carolina State University (samatova@csc.ncsu.edu)
Oak Ridge National Laboratory

1.1 Introduction

Peta-scale simulations utilize the massive computational power available from supercomputers to simulate scientific phenomena at previously unseen levels of detail. Resulting from simulations being performed at such a scale is the ability to easily generate several terabytes of a data in a single day. Unfortunately, the rate at which data is generated across simulations exceeds the bandwidth available to ingest data into external storage devices. This disparity oftentimes leads to simulations being delayed while waiting for I/O operations to complete.

For example, the computational capability of leadership computing facilities grew from 14 TFlops to 1.8 PFlops (the Cray Jaguar at Oak Ridge National Laboratory) from 2004 to 2009, an increase of over 100× [27]. This trend is likely to continue into the exascale, and in 2011, the K computer in Japan has reached 10.51 PFlops. On the other hand, from 2004 to 2009, the aggregate I/O rate on leadership computing facilities has only increased from 140 GB/s (the ASC Purple at LLNL) to 200 GB/s (the Cray Jaguar) [27], through

the support of high-performance parallel file systems, such as IBM General Parallel File System (GPFS) [57], Parallel Virtual File System (PVFS) [9], and Lustre [58]. At these performance trajectories, leadership-class facilities will quickly hit a performance ceiling, where increases in computational capability will go unutilized due to the much slower increases in I/O. Thus, it is imperative that we remove or at least lessen the severity of I/O limitations.

There are many ways to help address I/O performance limitations, including developing new storage technologies such as solid-state drives (SSDs), optimizing parallel file system implementations to provide peak disk bandwidth, and overlapping I/O and computation. However, more to the point of this chapter, we can rather exploit this gap in performance by using the otherwise idle CPU cycles on other meaningful computation. This allows us to not only make use of the computational resources available, but also perform operations on the data previously considered to be post-processing, while the original data is still in memory.

This process of operating on data in memory at simulation time is called *in situ* processing. While technically any parallel algorithm that can be inserted into a simulation code can be considered *in situ*, there are a number of optimization goals particular to the high-performance computing field.

1. First and foremost, *in situ* algorithms must minimize the time added to the simulation as a whole. Project budgets are typically strict about computational time allocated.

2. Related to the first point, *in situ* algorithms should minimize or avoid global communication, opting for local communication or even being communication free. Requiring idling while waiting for network operations to complete (e.g., global communication) defeats the purpose of performing *in situ* computations.

3. *In situ* algorithms should also leave a minimal memory footprint. In systems with hundreds of thousands of cores, available memory per core is limited, with simulation codes using the majority of it.

There are a number of ways that *in situ* methods can satisfy these goals, but they are, of course, application dependent. This brings us to another question: what kinds of computation are applicable for performing *in situ*? To answer this question, it is useful to review the categories of data produced by applications:

Check-point and restart data (C & R): This data is written out by simulations at routine intervals (e.g., every hour) so that simulations can avoid having to restart from scratch in the case of any execution failure. This data, which is usually voluminous, can benefit from the application of lossless compression techniques to reduce the burden on I/O.

Analysis and visualization data (A): This data is often written in more

TABLE 1.1: Summary of GTS output data by different categories.

Category	Write Freq.	Read Access	Total Size	*in situ* Algorithms
C&R	1-2 hours	Once or never	≈TBs	Data Reduction
A	10^{th} timestep	Many times	≈TBs	Data Reduction Transformation, Analysis & Visualization
V&V	2^{nd} timestep	A few times	≈GBs	Analysis

frequent intervals than C & R data, and is repeatedly accessed by visualization and analysis routines. Unlike C & R data, data used for analysis is inherently lossy as scientists typically sample data on some fixed timestep interval to keep the data size within manageable proportions. While lossless compression techniques can be applied to this data, it usually requires repeated access and could benefit from lossy compression algorithms that can provide higher compression ratios, as long as the loss is within an acceptable bound. This data is mined for exploration and analysis, leading to scientific discovery. Some simulations treat the checkpointed and analysis data the same.

Verification and validation data (V): This data is written out every few timesteps to check the sanity of the running simulation. This data is usually small, amounting to a few MBs every timestep.

Table 1.1 shows example statistics on each type of data generated by the GTS [70] simulation, a particle-based simulation for studying plasma microturbulence in the core of magnetically confined fusion plasmas of toroidal devicessimulation.

From these types of data, there are two broad types of computations applicable for *in situ* computation, which we will focus on in this chapter. One is *data analysis*, performed on analysis and visualization data. There are two sides to this coin: one involves analyzing the data for specific phenomena without user interaction, computable through data mining algorithms. See Section 1.2. For example, *feature extraction*, where the definition of a feature is dependent on the application domain (such as vortices in a fluid-flow simulation), can be performed *in situ* and without user guidance. The other side of the coin instead requires user interaction and resides in the space of database optimization. That is, the data can be prepared for user-defined *query processing*–the retrieval of data given constraints in the spatio-temporal domain as well as on variable values. Query processing on the data *in situ* is a complex problem, one that would require running queries on data that might or might not exist in the simulation working set, and one that produces a number of interesting research directions, such as simulation steering, validation, and diagnostics at simulation-time. A more clear-cut goal is to build the global database *in situ*, performing data reorganization, indexing, reduction, etc.

to optimize post-simulation query processing, rather than doing all of these data-intensive operations from disk. This is described in Section 1.5.

The second type of computation applicable is *data transformation*, which overlaps with the (future) analysis portion through *data reorganization*, which prepares the data for efficient future access, and *data reduction*, through saving only the analysis results, such as features obtained from feature extraction. However, there is another important class of reorganization that is not addressed in the analysis space: compression. Compression not only has the potential of reducing the overall data footprint and thus the overall I/O cost, but it also exploits the growing compute-I/O performance gap. Therefore, with some appropriate level of overlap, compression (lossy or lossless), obviously *in situ*-capable, has the potential of improving overall application performance. This is discussed in Section 1.3.

Finally, an important question for system designers to answer is: where in the software stack should *in situ* computation be enabled? Scientific codes can reach into the hundreds of thousands of lines: a naïve inclusion of *in situ* computation would require an enormous programming effort from the application designers to include the computation and efficiently handle compute, network, and I/O resource utilization to allow efficiency. Obviously, this is unacceptable. Furthermore, without the ability to decouple the computation of simulations and their corresponding I/O, *in situ* algorithms will only add to the overall simulation time, rather than be properly overlapped. Therefore, we must enable *in situ* computations at varying subsystems of the scientific code, so we can allow the scientific code itself to incur minimal changes, while allowing for a large degree of flexibility and resource utilization. The following subsystems need to be looked at in greater detail to allow this enabling to occur:

High-level I/O APIs: APIs such as ADIOS [37] are highly extensible, allowing for the definition of custom operations to occur while the data is "in-flight," or on its way to disk. Libraries such as these represent the entry point of *in situ* codes, plugging in libraries and frameworks that can include *in situ* algorithms.

I/O Optimizing Middleware: A popular I/O optimization is to dedicate a subset of nodes in a compute cluster to be responsible for all I/O in the system, and even an additional layer of nodes to control resource scheduling; this way, compute nodes send data, often asynchronously, across a higher-bandwidth network rather than directly to disk, reducing idle time and enabling efficient overlap. *In situ* code can be enabled in these middlewares, hooked into the high-level I/O APIs, so that the compute nodes need not incorporate the additional code nor perform the actual *in situ* computation, leading to the low-latency, transparent integration of *in situ* algorithms.

Section 1.4 discusses both the high-level I/O APIs that many scientific applications use and the enabling middleware technologies.

The benefits of answering these questions and providing robust, efficient *in situ* computation will not only benefit high-performance computing, but will also find much benefit for other "big data" application domains and online/streaming analysis scenarios. For example, the ever-increasing sample rate of sensor data in both the climate and physics domains necessitates the usage of analysis methods that can analyze and process data in real time [18]. While the conditions by which the data is generated may be different, the components necessary to work with distributed data before outputting to storage share numerous similarities with the *in situ* processes and frameworks discussed in this chapter.

1.2 *In Situ* Data Mining

In any scientific simulation, the application scientist is looking to understand certain phenomena (or patterns) that may not be easy to analyze in the physical world. An example of such a pattern is the vortex of a fluid flow, including fluids such as liquid, gas, and plasma. *Vortices* are spinning flows of fluids that are typically turbulent. Vortices have certain physical properties; for instance, their speed and rate of rotation is the highest in the center and gradually decreases when moving away from the center and they exhibit a minimum fluid pressure in the center. Scientists may be interested in the vortices themselves, which in climatology signify cyclones or hurricanes, or in more generic patterns such as regions with abnormally high temperature or low pressure, where low pressure systems may signify a coming storm.

Scientific simulations not only help to understand a physical phenomenon but provide information that will help replicate the phenomenon under controlled settings. One example is thermo-nuclear fusion, widely considered the "Holy Grail" of renewable energy generation. Fusion energy production could provide an alternative, environmentally friendly, and renewable energy source for our planet, but it is an extremely complex problem in the realm of physics and chemistry. The technical challenge for fusion energy production is stabilizing the flow of heated plasma in magnetic fields inside a fusion energy reactor–scientifically, this problem is oriented towards controlling plasma instability, or turbulence. Scientists have conducted simulations for several years now to model fusion reactions, but lacked appropriate technology to find the onsets of turbulence (called *fronts*) from analysis data [59].

Using some known properties of the physical patterns, analysis and visualization data produced by a simulation can be mined for regions or volumes that satisfy these properties, known as *features*. The process of identifying these features is called *feature extraction*. In simple terms, feature extraction is the process of separating the *important* regions from the *background* [63]. Traditionally, this analysis is performed as a post processing step. However,

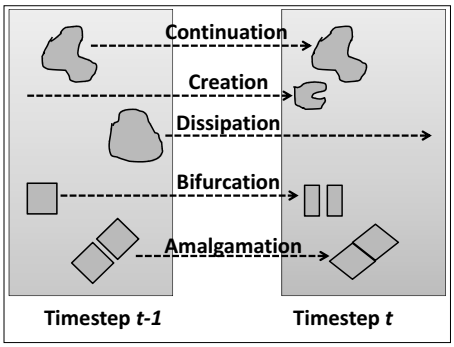


FIGURE 1.1: Feature tracking events.

the ability to perform these *in situ* would have numerous benefits. First, features in many cases are the only important data scientists wish to analyze, so storing only the features results in huge data reduction [38, 79]. Second, feature extraction done at simulation time allows *real-time* analysis of features, that is, analysis of the data while the simulation is running. This is especially important for simulations with running times in the order of days and weeks, or for future applications that allow scientists to steer or verify the simulation as the simulation is running.

Scientific simulations deal with dynamic phenomena, that is, phenomena that change over time. Hence, performing feature extraction alone is not sufficient. We also need to be able to study the changes the features undergo over time. This process is called *feature tracking*. Typically both feature extraction and tracking are performed to better understand the simulation.

Feature tracking involves looking for the following events [54, 63], illustrated in Figure 1.1:

Continuation: A feature identified in timestep $t-1$ exists in timestep t.

Creation: A new feature appears in timestep t.

Dissipation: A feature from timestep $t-1$ is no longer present in timestep t.

Bifurcation: A feature from timestep $t-1$ separates into two more features in t.

Amalgamation: Two or more features from timestep $t-1$ merge into one feature in timestep t.

1.2.1 Property-based Methods

Features could be defined as volumes in the data that satisfy a threshold for certain attributes (e.g., pressure, temperature). The region of interest may

be constrained by a single attribute or a combination of several attributes. There are two strategies that are typically deployed for feature extraction, similar to density-based clustering techniques that separate the useful regions from noise.

Region growing (or filling) [63] is an iterative method that uses certain data points (data cells, voxels, pixels, or particles) in the current timestep as seeds, and the feature is "grown" by utilizing the neighboring points until the value of the attribute falls below (or rises above) a certain threshold. *Seed points* are typically defined as points with simulation-specific extreme values. For example, in combustion simulations to identify high-temperature regions, the simulation points with the highest temperatures are chosen as seeds and the feature is grown using the neighbors of the seeds, stopping when the temperature falls below a certain user-defined threshold.

Connected Components [76] is a method where all data points that satisfy certain criteria (e.g., pressure below a threshold) are identified, followed by grouping neighboring points into one feature. Neighboring points here are defined as those not separated by other data that do not satisfy the criteria.

Once the features are identified in some timestep, they need to be tracked in the following timesteps. As discussed, this involves looking for any of the 5 events defined. Feature extraction takes place at each timestep and then the lists from the current and previous timesteps are correlated. This is called the *correspondence problem*. A brute-force method would be to compare each feature identified in the previous timestep with all features and all combinations of features (*bifurcation* event) in the current timestep [63]. The comparison is based on certain domain-specific attributes. Because the brute-force method must perform comparisons on all subsets of the feature set, the number of comparisons is exponential, which is certainly not applicable for *in situ* computation. A more reasonable method is to split the correspondence into two steps. The first step is to look for those feature pairs that overlap [54, 63], as shown in Figure 1.2. An efficient way to perform this overlap is to sort the feature lists in two consecutive timesteps and then merge the two lists [10, 11, 12]. Using this method, the amount of overlap between a pair of features can more easily be determined. This gives us candidates for corresponding features. The second step is to identify for each feature the *best-matched* feature pair. This can be done by calculating the *normalized correspondence metric (NCM)* [63, 64], originally defined for overlapping volumes. Let f_i and f_j be features occupying a volume in space. The NCM for the pair is given by:

$$NCM(f_i, f_j) = \frac{Volume(f_i \cap f_j)}{\sqrt{Volume(f_i) * Volume(f_j)}} \tag{1.1}$$

In the sorting-based strategy, the entire list of features from each timestep has to be calculated before the feature is tracked by solving the correspondence problem. This requires us to analyze all the data produced in every timestep. In reality, only a few features out of the entire list will be of interest. There-

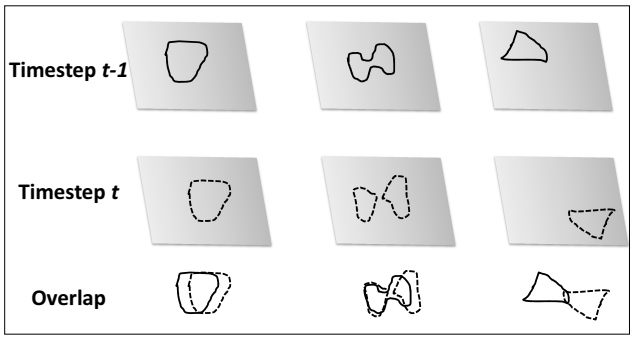


FIGURE 1.2: Feature tracking using overlap.

fore, instead of segmenting the entire data in the current timestep, a more efficient method would be to take advantage of the feature information from the previous timestep(s) to predict the position of the feature in the current timestep [43].

This problem can be mathematically modeled using one of three variations (Figure 1.3). The *direct method* merely performs direct projection of the same feature in the previous timestep to obtain the feature in the current timestep. However, this does not account for features migrating during simulations. To predict the current position of a feature more accurately two additional methods can be used. The *linear method* uses the projection of the feature in the previous frame but the position is offset using the difference in centers of the feature's region in the previous two timesteps, using a linear model. The *quadratic method* also uses the projection of the feature in the previous frame but the position is offset by modeling a quadratic curve that passes through the centers of the same feature in the previous three timesteps. From the description it is fairly clear that feature tracking using the direct method can begin in the second timestep, the linear can begin only in the third timestep and the quadratic can only begin in the fourth timestep.

A drawback for these kinds of predictions is that they only consider motion with respect to the feature centers, ignoring other parameters such as angular motion, change in volume, or the merge or split that can occur in subsequent timesteps. Furthermore, these projection-based methods cannot identify new features in the current timestep.

The prediction made is only an estimate; in order to identify the actual feature, an additional step called *region morphing* is performed. Region morphing is similar to region growing but includes both boundary growing and shrinking. It uses a simple breadth-first search strategy to include neighboring data points that satisfy the attribute thresholds and excludes those data points from the feature that do not satisfy the attribute thresholds. The search begins with a seed point in the feature region.

The algorithms discussed in this section so far deal with identifying fea-

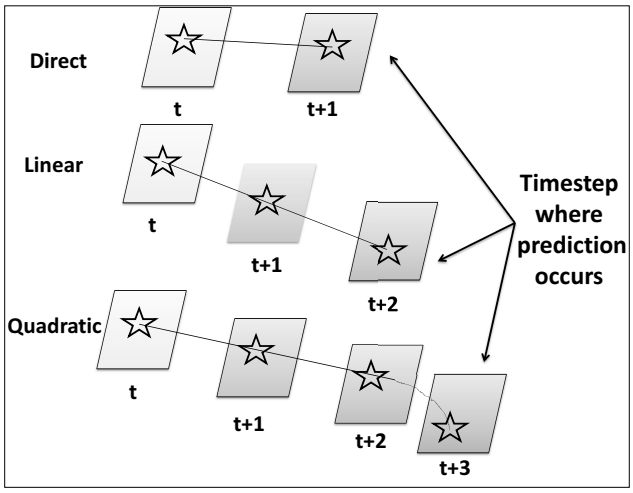


FIGURE 1.3: Feature prediction in the current timestep utilizing information from previous timesteps.

tures based on some attribute value. There are also some application-specific features that may not be dependent on the attribute value but only on the distance or location with respect to other particles. In dark matter cosmological simulations, one particular phenomenon of interest is the evolution of *dark matter halos*. These halos are of great importance because it is said that almost all the mass in the universe ends up in these halos, which are objects with dynamical equilibrium. These also have the property that smaller halos over time merge to form larger halos. Objects like the galaxies are said to form and evolve in these halos [32]. However, there is not much known or understood about these except that they are clusters of particles in space where all particles are within a certain threshold distance from all the other particles in the halo. This threshold is known as the *linking length*.

The implementation to identify these features uses a *friend-of-friend (FOF)* algorithm, that employs these linking-length to identify halos. A *friend* of a particle is a particle within the linking length. It is sufficient to identify the halos by then comparing friends of the friends, and clustering them together. This lends itself to an efficient parallel implementation [4, 3, 74], where the overall space is partitioned among the processes (when *in situ*, this is predetermined) and spatial overlap is utilized to minimize local communication, only needing to update particle locations after each timestep. This is similar to *stencil computation*, which defines a set of *ghost cells* on array boundaries, which are updated each step in the computation. In addition to the identification, the halos are also further classified based on the number of particles they contain. The *light halo* has about 10 to 40 particles, the *medium halo* has about 41 to 300 particles, the *heavy halo* has about 301 to 2,500 particles, and

the *extra heavy halo* has more than 2,500 particles [3]. The data reduction by storing only the halos is significant: up to 50% of the non-halo data can be thrown out.

1.2.2 State Change Detection

In the previous section, we identified *salient* features by analyzing some specific attribute values. There are other feature extraction algorithms where the features represent specific significant events that take place within the system and, in some cases, it represents the system moving from one state to another. For example, in the Lifted Ethylene Jet Flame Simulation, the extinction and reignition events are important and complementary. The flame may get extinguished because the heat release rate cannot keep up with the heat losses and if the extinguished regions continue obtaining a supply of fuel and oxygen, the flame could reignite. These events are important and require further analysis [33].

The regions in each timestep in the Lifted Ethylene Jet Flame Simulation can be characterized by some subset of chemical species that is present in the input. Each timestep is divided into data blocks and *Principle Component Analysis (PCA)* is performed on each block [69]. The eigenvalues and eigenvectors provide enough information to understand the chemical species that the block correlates with. This information, when taken over the entire simulation, can characterize the various possible events that have taken place during the simulation. However, the number of such states may be very large: hence, some local clustering using the chemical species as attributes is used to provide a more abstracted view. This view can also help understand the timesteps where the ignition or extinction events took place. In order to reduce the dimensionality of the data block, PCA is used to project the data onto a lower dimensional space by removing highly correlated dimensions. However, instead of performing it per block, the global (per timestep) PCA is calculated using the local covariance matrices of each block combined using an update function [51]. The global projection is then applied to each datablock. This reduces the problem to performing one single PCA.

Another example of a feature representing the underlying system's state change is a *front*. Fronts, as discussed in the introduction, signify the beginning of a turbulence. Thus, there is a change in the state of the system which needs to be captured. An algorithm has been developed that enables automatic turbulent front detection, and is able to track and quantify front propagation over spatio-temporal regions [59]. Mathematically, fronts tend to occur at points during the simulation where the potential energy function reaches maximum curvature. However, calculating curvature numerically (via second derivative) is unreliable and ineffective, as it produces noisy patterns that challenge finding the points of interest. Thus, due to the inherent complexity of the simulation data, automated detection of fronts calls for a statistically robust and productive method.

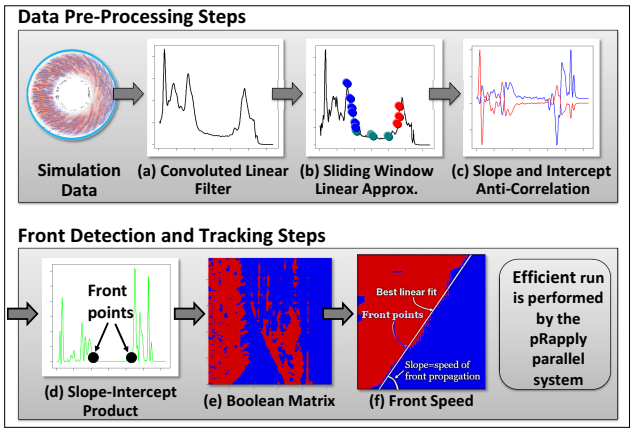


FIGURE 1.4: An end-to-end, multi-step analytical pipeline for spatio-temporal turbulent front detection and tracking.

The front detection method is shown in Figure 1.4. The method first smoothes raw simulation data to reduce noise and make analysis more manageable (Figure 1.4.a). The smoothing technique uses a convoluted linear filtering algorithm, where values of the smoothed function are dependent on the weighted average of surrounding values. Next, the data is traversed in a sliding fashion and linear regression is applied on each window of points (Figure 1.4.b). The slope and intercept values are collected and normalized to the $[-1, 1]$ range due to their varying magnitudes. The collected values suggest that slope and intercept values over the windows will be anti-correlated–a positive slope likely results in a negative intercept, and vice versa (Figure 1.4.c). This strongly anti-correlated pattern suggests that multiplying these slope and intercept values to create a SIP (slope-intercept-product) "signal" could effectively enable identification of the curvature points of interest. This signal is made entirely positive in order to further amplify the differences in the data's direction and magnitude. Next, a low ($T \approx 0.01$) threshold is applied for all such signals over varying timesteps (Figure 1.4.d). The theoretical justification behind this process is that points of curvature will be found where the data changes from having a slope parallel to the y-axis to the x-axis, and vice versa. At these points, the slope will be near 0, and thus the product will also be near 0. The thresholding operation converts all of the SIP signals to matrices of Boolean TRUE/FALSE values, where TRUE corresponds to signal values v, $v \geq T$, and FALSE represents values for which $v < T$. Finally, a two-color heatmap is constructed, representing TRUE and FALSE, respectively (Figure 1.4.e). The heatmap clearly depicts propagations of turbulent fronts over data. This analysis technique also enables users to specify regions of interest, and accordingly calculates the direction, duration, and speed of

propagation of fronts located within that region. This entire process is simply parallelized over different timesteps using the `pRapply` software.

1.2.3 Importance-driven Feature Extraction

A strategy for feature extraction that is different from those previously defined is to quantify the various regions in the data based on the amount of information they convey. An information-theoretic method quantifies the significance of a feature based on the amount of information it conveys by itself and the amount of information it conveys in comparison to the same feature in a previous timestep [71]. Each timestep is divided into spatial datablocks, corresponding to processors at simulation time. Each datablock is characterized by a set of attributes, for example, pressure, temperature, etc., which is used to build a *multidimensional histogram* for each block. Each bin in the histogram contains the data (voxels, pixels, particles) that satisfy a certain combination of attribute values. This histogram is used to calculate the *entropy* that will quantify the amount of information a datablock holds on its own. A high entropy value signifies a more important block. The *mutual information* is used to quantify the amount of information a data block has with respect to the block in the same spatial location but in the previous timestep. This is calculated by building a joint *feature temporal* histogram. A low mutual information value signifies a more important block. The normalized heights of each bin in the histogram provide the required probabilities to calculate both mutual information and entropy. The importance of a timestep is calculated as a function of the importance of each datablock it contains.

This method helps separate out regions that offer significant information. Additionally, this method helps bring to attention the abnormal events that took place in the spatio-temporal simulation. It helps the questions of *when* and *where* the abnormal (or interesting) events happened in the simulation. The timesteps with the highest importance scores answer the *when* question. Within this timestep, the datablocks with the highest significance values answer the *where* question.

1.2.4 Communication Strategies

An important optimization goal of *in situ* analysis techniques is to minimize the amount of inter-process communication at simulation time. Interprocess communication is required to analyze features spanning multiple processors. In the worst case, these features could span every processor. There are three general inter-process communication strategies found in the literature. In the *complete merge* [10, 11, 12] strategy (Figure 1.5), each processor identifies its local set of features and then communicates the information to all the other processors, using *all-to-all* communication methods. Using tree-based communication algorithms, logarithmic communication steps are needed for the all-to-all communication, relative to the processor count. At the end of

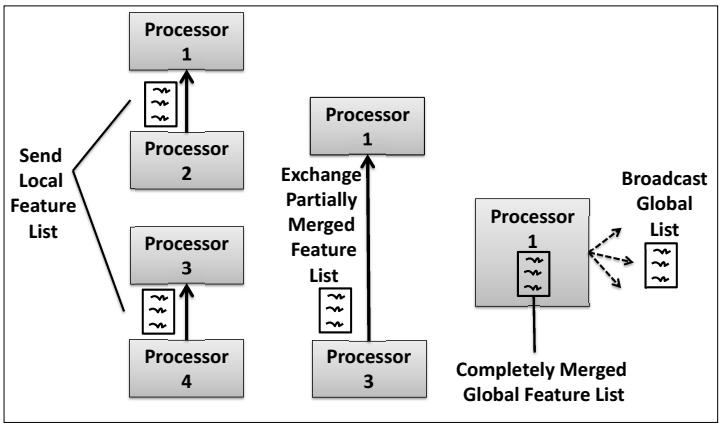


FIGURE 1.5: Complete merge strategy.

this communication, all processors have the *global* feature table with correlated information. Obviously, this is highly communication-intensive, which is undesirable for *in situ* computations.

In the *partial merge* [10, 11, 12] strategy, each processor only communicates the feature information with its immediate neighbors. At this stage, the partially merged feature tables are sent to another server (called *viz-accumulator*) that takes care of the merging. This server may then pass on the globally merged feature table back to each processor or to a visualization system. The communication is much less intensive, though it still requires global communication. Both the complete and partial merge strategies have been deployed by the algorithms [10, 11, 12] discussed in Section 1.2.1. The third strategy is called *no merge*. In this strategy, each processor identifies the features at each timestep but no merging takes place during the simulation run. As a post-processing step, the entire simulation feature set is analyzed and merging takes place. This maintains simulation speed, but as a result moves the problem of merging elsewhere, so using no merge will be application-dependent.

In conclusion, we find that *in situ* data mining is an intelligent way to extract useful information from the raw simulation data. The features identified characterize important physical patterns and provide a way to analyze and understand them better. Apart from the techniques discussed in this section, parallel implementations for several existing data mining algorithms [13, 31, 36, 40, 44, 45, 46, 48, 60, 61, 62, 72, 73] could potentially be translated for *in situ* data analysis.

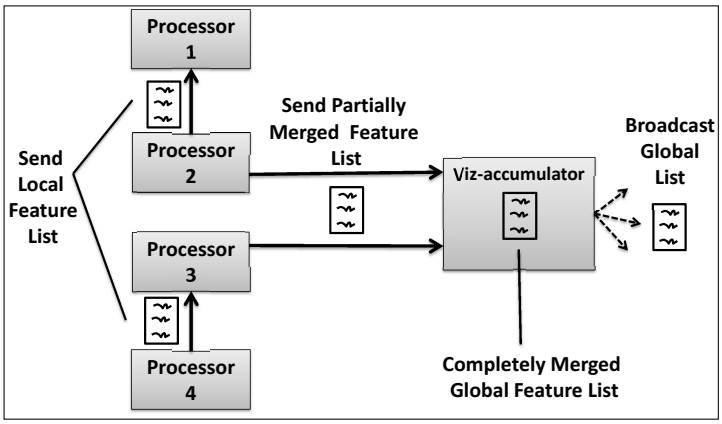


FIGURE 1.6: Partial merge strategy.

1.3 *In Situ* Compression of Scientific Data

The last decade in the high performance computing realm has witnessed an increasing imbalance between computational power and file system bandwidth. As this imbalance is expected to grow even further in the exascale era, data compression becomes a necessity rather than an option. With compression being co-located with the simulation process, idle CPU cycles can be utilized to bridge the gap between data generation, and ingestion.

However, compression of single and double-precision spatio-temporal scientific data has proven to be a challenge due to its inherent complexity and high entropy associated with the values. General lossless compression libraries, such as BZIP2 and ZLIB, tend to fare poorly on such scientific datasets, offering less than 10% reduction in data size in majority of the simulation data [56]. In order to alleviate this problem, several compression techniques have been proposed, both in the lossless and lossy front, specifically designed to work on data from scientific simulations. Although suitable for compressing in the post-processing phase for archival and storage, some of the techniques are inefficient for *in situ* application. In this section we evaluate the effectiveness of some of the existing lossy and lossless compression techniques, and analyze the applicability of compression algorithms in *in situ* processing environments.

1.3.1 Lossless Compression

Lossless compression of floating-point data is the only applicable type of compression in checkpoint and restart data, since the copy of the data must be exact for reproducibility. Lossless compression is also applicable to high-

fidelity simulation data or for analysis methods sensitive to noise, such as Fourier analysis. However, lossless compression requires a more complex processing scheme which general purpose compressors traditionally do not exploit. To understand what makes floating-point compression hard, one can look at the bit-level probability distribution values on scientific datasets. An ideal distribution on a bit for compression would be every bit position in the data holding the same value (0 or 1), while the least-ideal distribution would be equal occurrences of each bit. Highly repeating values, represented by higher probability distribution values closer to 1, can be easily predicted, and compressed. However, an equal probability distribution (closer to 0.5) makes the bits unpredictable and hence hard-to-compress (HTC). Of course, other properties, such as spatio-temporal correlations in bit probabilities, may be present. Compressors can take advantage of these properties, though property discovery for general-purpose compression is a nontrivial problem, and least significant bits in double-precision variables tend to have nearly no correlation of any kind with each other.

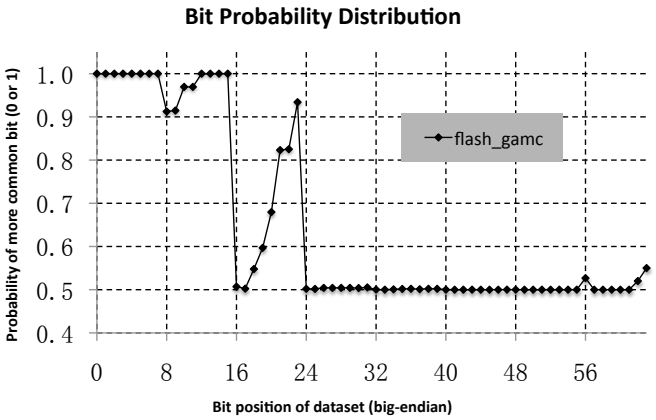


FIGURE 1.7: Plot of probability distribution of each bit position from the `gamc` variable in FLASH simulation data.

Figure 1.7 shows the bit-level probability distribution patterns for a typical floating point variable in scientific datasets (the one given is from FLASH astrophysics simulation code [5]). Widely used general purpose compressors like ZLIB and BZIP2 are oblivious to this fact, and perform poorly as a result. In this section we study some of the state-of-the-art compression utilities, namely FPC [7], FPZIP [34], and ISOBAR [56], and their applicability in *in situ* processing environments. A summary of the strategies behind these algorithms is given in Table 1.2.

TABLE 1.2: Summary of different compression algorithms.

Compression	Lossless?	Strategy
ISOBAR	Yes	Frequency Analyzer, Black-box Compressor
FPC	Yes	FCM / DFCM Predictor
FPzip	Yes	Lorenzo Predictor, Arithmetic Encoder
BZIP2	Yes	Burrows-Wheeler Transform
ZLIB	Yes	LZ77 and Huffman Encoder
Wavelets	No	Wavelet Transform
ISABELA	No	Sort pre-conditioner, $B-$Spline reduction

1.3.1.1 ISOBAR

The In Situ Orthogonal Byte Aggregate Reduction (ISOBAR) Compression utilizes a preconditioner to extricate compressible data from HTC datasets. ISOBAR is comprised of two main components to provide improved compression efficiency:

1. The ISOBAR-analyzer, which identifies the portions of the data that can be compressed by an entropy-based encoder vs. the portions that cannot.

2. The ISOBAR-partitioner, which reorganizes the compressible and incompressible portions of the input data identified by the analyzer.

ISOBAR-compress is highly flexible in that varying compression libraries such as ZLIB or BZIP2 or even FPC and FPzip can be utilized to encode compressible bytes, based on user preferences (compression ratio vs. speed). The overall workflow of ISOBAR compression is shown in Figure 1.8.

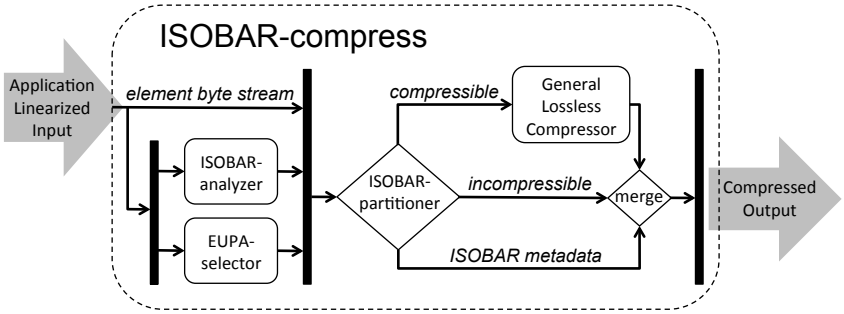


FIGURE 1.8: ISOBAR-compress preconditioner workflow.

ISOBAR-analyzer The objective of ISOBAR-analyzer is to identify byte-column clusters in the floating point dataset that are ineffective when compressed. That is, in a double precision dataset, each significant byte in all doubles in the stream is considered for compression together, rather than compressing on a double-by-double basis. ISOBAR-analyzer works on a byte-level

granularity to efficiently identify high-entropic byte-columns that cannot be efficiently compressed. The ensuing ISOBAR-compress operation re-orders the compressible and incompressible portions and pipelines them to be used by general purpose compressors. This byte-level ordering ensures better compression ratios with entropy encoding.

In order to ascertain which bytes are compressible, ISOBAR-analyzer first transforms an $n-$dimensional input data into a linearized stream of floats, and then generates a frequency distribution for each byte column in the set of floats. For example, consider the case when ISOBAR-analyzer works on a 1-dimensional dataset of N elements of w bytes each ($w = 8$ for double-precision, $w = 4$ for single-precision float). The analyzer starts with the initialization of a frequency counter array of length w, with zeroes. The data is scanned through, and the frequency counter is updated to calculate the distribution of each of the possible $2^8 = 256$ unique bit patterns for every byte-column. When this value is less than $N/256$, ISOBAR considers the column-based dataset to negatively affect the overall compressibility. Removing this set of bytes in the compression phase would invariably result in both better compression ratios and throughput.

ISOBAR-partitioner Once the compressible byte-columns are identified by ISOBAR-analyzer, the ISOBAR-partitioner determines whether the column would contribute to improved compression ratios. Columns identified as *improvable* are subsequently passed to the compress phase after re-partitioning. In some cases, it is possible that the analyzer concludes that either none or all the columns in the dataset are compressible. When this happens, during the analyzer phase, the partitioner classifies the compressibility type of the dataset to be *undetermined*, and passes the entire dataset to the compressor.

The ISOBAR-partitioner essentially converts the input data into two segments. One is the compressible byte columns that are packed together and fed to the underlying compression library. The other is the set of incompressible bytes that are stored as-is. In this phase, different linearization strategies can be employed like row-order or Hilbert or Z-order mapping (see Section 1.5.1.3), before passing the data on to the compressors.

The partitioner realigns the data, as determined by the linearization strategy. For example, suppose ISOBAR-analyzer returns that columns 1, 2, and 4 are compressible, and row-major linearization order is used. The ISOBAR-partitioner would cluster all 1^{st} byte values belonging to the N input elements, and reorganize them in row-major order. This data would then be appended with the 2^{nd} column, followed by the 4^{th}. So, in this case, only $3 \times N$ bytes are passed to the compression process (a ratio of $\frac{3}{w}$), guaranteeing an improvement in both throughput by compressing less, and compression performance by keeping high-entropy content out of the compression algorithm.

1.3.1.2 FPC

FPC is a fast lossless floating-point compression algorithm for 64-bit double precision data. Unlike other compression algorithms like ZLIB and BZIP2, FPC is a single pass linear-time algorithm that is designed to deliver high-throughput on both compression and de-compression. One important characteristic of the FPC algorithm is that it does not depend on the underlying structure of the data. FPC compresses a linear sequence of doubles by predicting each encountered value, and storing the XOR difference between the predicted and actual value. The leading zeros in the difference are then encoded to achieve compression.

The prediction algorithm used in FPC is a variant of the FCM [78] and DFCM [19] predictors, using finite context models of order n to predict the next occurrence in a stream of values based on the preceding n occurrences. For example, a five-order FCM would predict an element using the previous five values. In the case of a differential model, the context is modeled as the difference between successive occurrences. The FCM and DFCM predictors have shown accurate performance in predicting and prefetching instructions, thus eliminating dependencies and improving parallel execution. With FPC, this concept is extended to predict bit patterns of the first two significant bytes of floating-point numbers containing the sign bit, exponent bits and top 4 mantissa bits in an efficient manner. FPC, for each prediction, uses both FCM and DFCM and selects the best predicting one, using an additional bit per prediction to store the choice. The primary feature of FPC is its efficient implementation that results in high compression throughputs. To accomplish this, FPC is *cache-aware*, bounding the memory needed by the hash functions of FCM and DFCM to store the recent m contexts (where $m < 26$, and user specified) to fit within the cache. Because difference coding is used, FPC can fail to provide adequate compression performance on data from peta-scale simulation application that contain little or no repeating values, or those without much point-to-point correlation.

1.3.1.3 FPzip

The FPzip compression utility was designed to compress values from 2D and 3D structured grids, fields, and unstructured meshes. FPzip compression processes data in a coherent fashion, and employs a Lorenzo predictor to predict values, and the difference between the predicted and the actual values is encoded using a high speed entropy encoder.

The *Lorenzo predictor* [24], used in FPzip, is a generalization of parallelogram prediction algorithms to an arbitrary number of dimensions. The prediction algorithm relies on the values of its neighbours in a grid to predict the current value. This is in contrast to FPC, which only implicitly uses spatial correlation in its prediction, if the linearization scheme captures it. For example, with a 2D Lorenzo predictor, a point $\langle x, y \rangle$ whose value is given by

$f(x, y)$ is predicted as

$$f'(x, y) = f(x, y - 1) - f(x - 1, y) + f(x - 1, y - 1) \tag{1.2}$$

For an n-dimensional grid, an $(n - 1)$-dimension slice must be maintained in memory. Since the addition and subtraction of a large number of floating-point numbers might lead to overflow and underflow, the floating point values are monotonically mapped to unsigned integers and extra care is taken to identify the propagation of the carry bit. In the case a lossy compression is desired, FPZIP allows some of the least significant bits to be discarded during the mapping phase. FPZIP exploits an inherent characteristic of data where exponent values can be correlated with neighbouring values, and with a light-weight metadata for the predictor, FPZIP performs effectively on several datasets.

1.3.2 Lossy Compression

Lossy compression techniques are irreversible operations that trade some form of loss in information or precision to achieve significant reduction in data sizes. Some of the common forms of reduction include quantization, histogram binning, subsampling, and transform-based techniques like Wavelet transform, and Discrete Cosine Transform [39]. The advantage of lossy compression techniques over lossless ones is the ability to provide multi-fold reduction even on HTC datasets. In cases where domain knowledge about the generated data can be utilized, reduction by over an order of magnitude space is not uncommon. Additionally, space savings can be incorporated when it is known *apriori* that the data will be used for visualization and analysis routines that do not require high fidelity or full precision data. This amounts to considerable savings in data storage, and time otherwise spent in expensive data movement.

1.3.2.1 Subsampling

Subsampling in the spatial or temporal resolution is a common method employed by application scientists to achieve data reduction. For example, in the case of fusion simulations such as GTS [70], the data that is used for analysis is saved to storage only every 10^{th} timestep. In the case of astronomical simulations like Supernova, this number can be as high as 100. Although skipping timesteps help keep the output data within manageable proportions, and alleviate the bottleneck on I/O, this technique is not scalable. Skipping a large number of timesteps can result in extreme events or features being missed from identification during the analysis phase.

1.3.2.2 Quantization

Quantization in its simplest form can be defined as a process that maps a set of values representing a large, possibly infinite, range into a smaller range of values. The smaller range of values is called a *codebook vector*, and

each individual value is called a *code word*. While converting the values into codewords results in loss of information, its subsequent encoding results in data compression. The design of the quantizer function determines the amount of compression achieved, and information loss that is incurred. The simplest form of quantization is the scalar quantization, which applies a quantizer function to each input value, individually. An example scalar quantizer is a function *round()* that maps values in the real domain to integral domain. Encoding the values in the integral domain provides better compression ratios than its original form.

Scalar quantization techniques are usually easy to apply, and its fast operation makes it an attractive technique for *in situ* processing. The quantizer function can be further classified as either uniform or non-uniform. Uniform quantization uses the same number of bits to encode every value in the input. This can lead to varying error rates, with regions in the input value that have low entropy incurring a higher penalty. Therefore, a non-uniform quantization can be applied that can adapt to different levels of compression in regions with varying information content, thus leading to more accurate approximation. While non-uniform quantization requires additional metadata to reflect the level of compression, superior quantization performance usually justifies the trade-off. Methods like the Lloyd-Max method [22], which ensures the quantizer boundaries are changed to match the data statistics, or adaptive scalar quantization that selects quantization boundaries "on the fly" to reflect data statistics in local context are effective examples of non-uniform quantization.

An extension of scalar quantization in higher dimensions is vector quantization, which has been shown to be a promising approach for compression. The principle behind vector quantization (VQ) is the fact that coding blocks of spatially correlated values provide more compression than when coding individual values. The overhead involved in sampling and training the input data to generate the codebook vector is non-trivial. To overcome this overhead, some variants of this technique employ no codebook training, and adapt the codebook based on local statistics. In either case, while compression rates provided by VQ are usually high, the slow convergence nature of the algorithm, along with issues in precision of computation, complicates its deployment as an *in situ* processing method.

1.3.2.3 Discrete Transforms–Cosine and Wavelet

The Discrete Cosine Transform (DCT) and Discrete Wavelet Transform (DWT), namely Haar and Daubechies [14], are two extensively used techniques in visualization and multi-media routines to achieve high compression ratios. DCT works by subdividing the input data into blocks, and traversing the values in a coherent order, such as zig-zag (to take advantage of spatial correlation), and then converting the values in each block in its spatial domain to its frequency domain. The resulting compact representation stores

a large amount of information in a small number of coefficients. Coefficients with lower values can be eliminated by thresholding, and the remaining few coefficients are optionally quantized and encoded, thus achieving high compression rates. The decoder part reads the frequency values, and applies the inverse transform to obtain the data in its original form. Since the majority of the information is contained in a smaller number of coefficients, the decompression results in data that exhibit good visual fidelity compared to the original.

Discrete Wavelet Transform (DWT) differs from DCT, in which the transformation occurs from the spatial domain to the time-frequency domain. With DWT, the transform allows good localization in both the time and spatial frequency domains, which enables DWT to achieve high compression rates. Wavelets is predominantly used for multi-resolution analysis (MRA) where wavelet transformation can be applied recursively within each block. The compression properties still hold at higher resolutions, and visualization functions take advantage of these properties by allowing different levels of detail to be selected during runtime. Both transform-based compression techniques are ideally suited for *in situ* data reduction since they offer high compression throughput.

1.3.2.4 Transform Pre-conditioners – ISABELA

The core idea behind applying DWT and DCT is that the transformation from the spatial domain to a more natural frequency-based domain yields values that are highly compact in its representation. The coefficients (values in the transformed domain) are clustered together, leading to an ideally optimal encoding. Unfortunately, due to inherent high-entropy content in many scientific datasets, the transformations generate a scattered representation which lowers the compression rate that can be achieved. This means that existing transform-based reduction methods are effective in their natural form only when they can exploit spatial and temporal correlation in the underlying data. In order to circumvent this problem, new methods have emerged that apply a *pre-conditioner* to the data before domain transformation, leading to a more natural, high compressible representation upon doing so.

Pre-conditioners are known to be effective tools to tackle a large number of problems in linear algebra. For example, to calculate the determinant of a large matrix, a transform may be applied to convert it to a lower or upper-diagonal form, thus simplifying the problem into an easier task of multiplying diagonal elements.

In the case of data reduction, what would be an ideal pre-conditioner that transforms the high-entropy spatio-temporal data into a function of high global regularity? One example of a state-of-the-art lossy compressor using pre-conditioners is ISABELA [28], which is capable of providing high-accuracy compression with multi-fold reduction of scientific data. ISABELA uses a sorting pre-conditioner that sorts the noisy original input signal into

a monotonically increasing curve. The exponent values in scientific floating-point datasets exhibit low entropy, as seen in Figure 1.7, thereby producing a gradually increasing smooth curve of high regularity upon sorting. This global regularity leads to rapidly vanishing moments in the transformed space, and compared to the data approximation without the above pre-conditioner, the transformed signal can now be modeled with high precision using a smaller number of coefficients.

ISABELA Methodology The core methodology behind ISABELA is quite similar to DWT and DCT based encoding. The data from the input signal is divided into smaller chunks, transformed and subsequently quantized and encoded. With ISABELA, a sorting pre-conditioner is applied to the input data, which produces a curve whose rate of change in values is the slowest. A smooth curve of this sort can be approximated efficiently using curve fitting techniques. ISABELA compression utilizes cubic $B-$splines, which are piecewise-polynomial functions, to approximate the sorted curve. In contrast to higher order non-linear polynomial functions, $B-$splines model complex curves by dividing the curve into piecewise parametric curves of lower order, and hence the shape of the curve can be controlled locally without affecting the other parts of a curve. Since cubic $B-$splines use only a third order polynomial function, they are fairly efficient in the time taken for both curve approximation and interpolation.

Compared to DCT based reduction, which performs well on smaller chunks that exhibit a large degree of spatial correlation (neighbouring pixels in an image usually have little variation in intensity values), pre-conditioning a larger chunk ensures a larger number of spatially clustered values. This results in the need to store only a few $B - spline$ coefficients to model the entire data accurately. Thus the ideal scenario for accurate modeling is when the entire data is sorted. However, the overhead introduced to maintain a large index that tracks the position from the original to the sorted signal, eliminates any gain in compression achieved by using a smaller number of coefficients. To balance between the compression ratio and the accuracy levels, ISABELA divides the data into smaller chunks, or *windows*, of fixed sizes. Within each window, a sorting pre-conditioner is applied, and the curve is then reduced to a set of $B-$spline coefficients.

The total storage used by ISABELA is the sum of a heavy-weight index (I), and a significantly reduced number of constant coefficients (C) which is light-weight. The index $I = \{i_1, i_2, ..., i_N\}$ is a bijective mapping that translates the position in the input array to the new location in the sorted array, where N is the number of elements. Since the range of values in I is a permutation of numbers from 1 to N, each value in the index can be represented using $log_2(N)$ bits. Clearly, the storage taken up by I is determined the number of elements in a compression window W. Using the compression ratio metric of original data size divided by compressed data size, the compression ratio of a 64-bit double precision dataset in each window (and hence the entire data) is

given by:

$$CR = \frac{W \times 64}{W \times log_2(W) + C \times 64} \tag{1.3}$$

An optimal choice of window size is one that balances the storage taken up by I and incorporates a sufficient number of elements to generate a smooth curve when conditioned. Therefore, to balance the compression ratio and accuracy, the ideal strategy for choice of the input parameters would be to choose the smallest window size.

1.4 Middleware for *In Situ* Processing

As noted, *in situ* processing is best performed below the application level of the scientific software stack for numerous reasons. Efficiently adding *in situ* capabilities to scientific codes relies on a few I/O-related optimizations:

- First, compute nodes in current HPC architectures perform *I/O Forwarding* [47]; rather than directly interacting with parallel filesystems and causing costly disk and metadata server contention, data is sent to a smaller number of dedicated I/O nodes across a much faster network interconnect. This not only simplifies supercomputer architectures by cutting down on the number of nodes connected to the I/O hardware, but opportunities exist for allowing the I/O nodes to coordinate and optimize file reading/writing. Compute nodes can then initiate I/O requests which, from the compute node point of view, finish much more quickly compared to direct disk interaction. Using an asynchronous interface, the compute nodes may also move on with their computation, providing a higher degree of autonomy.

- While I/O forwarding provides numerous advantages, it still cannot increase the aggregate throughput restricted by the I/O architecture. Furthermore, I/O patterns of the applications may still cause problems in the I/O forwarding pipeline, such as resource exhaustion on the I/O nodes. Therefore, while it mitigates much of the problems in allowing thousands or more nodes unrestricted access to the filesystem, the sheer size and scope of the data make I/O prohibitively expensive and the use of a smaller number of I/O nodes introduces unintended consequences. Therefore, large gains in efficiency can be made by adding a layer of *staging nodes* [1] between the compute and I/O nodes. The benefits here are twofold. First, more advanced resource management protocols can be used to minimize the effect of undesirable I/O patterns on compute node stalling. Second, these nodes can additionally perform analytics

and visualization "in-flight," while the data is still in memory, reducing the I/O pressure on postprocessing scenarios on the data, an especially important task when working with petabyte datasets. A large degree of operators on the data can be performed here, such as those discussed in the previous sections, and can be pipelined with application computation and I/O.

To take advantage of these hardware capabilities, efficient abstractions must be provided that both allow *transparency* of the particular architecture to the user and allow *flexibility* in allowing users to use the underlying hardware in interesting and efficient ways and not restricting them to an overly concise set of functionalities. For instance, it would be unacceptable to force application designers to perform manual bookkeeping across each layer of nodes; instead, an I/O operation should look like an I/O operation from the compute node point of view. At the same time, users who have prior knowledge about their application demands should be able to use the same libraries in a fashion optimal for their application. These software abstractions called *middleware*, and this section describes numerous aspects of middleware, such as system design, interface to the application designer, challenges in bridging the gap between simplicity and optimality, etc. These technologies allow application designers to tap into current-and-future generation hardware capabilities while avoiding the high development costs of writing optimal code directly targeting the hardware.

Before a discussion on middleware ensues, it is important to describe some *enabling technologies*, technologies that the middlewares rely on to provide their functionality to application designers. These are described in the next section. Once we have a brief familiarity with these software codes, we will move on to middleware implementing the I/O forwarding and built on top of these enabling technologies, and finally discuss staging middleware. These are described in Sections 1.4.2 and 1.4.3, respectively.

1.4.1 Enabling Technologies

1.4.1.1 Collaborative I/O with MPI-IO

MPI-IO [23] is a set of portable I/O APIs that are specifically designed for high-performance parallel MPI programs. It incorporates MPI features, such as inter-process communication and synchronization, into its I/O routines for cooperative reading and writing among a large number of processes. It applies collective I/O to group small noncontiguous accesses to improve data access performance. It is also optimized for parallel file systems to achieve high-throughput parallel data access. MPI-IO usually refers to the interface; there are a number of implementations. One in particular, ROMIO [68], is a high-performance and portable implementation of MPI-IO. It is optimized for noncontiguous data access patterns by data sieving and collective I/O [67]. For multiple read requests, instead of reading each piece separately, ROMIO

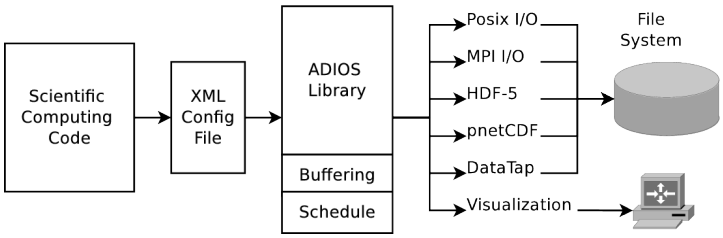


FIGURE 1.9: ADIOS architecture.

reads a contiguous chunk of data which starts from the first requested byte until the last requested byte into a temporary memory buffer. It then extracts the portions the user requested from the temporary buffer and copies them to the users buffer. ROMIO has been included as part of MPICH2 and MPICH-1 MPI implementations.

1.4.1.2 Hardware-agnostic I/O Optimization with ADIOS

Existing I/O routines used in scientific codes vary from the standard POSIX I/O and MPI-IO interfaces to higher-level I/O libraries providing advanced data layout functionality. Two examples of advanced data formatting are HDF-5 [17] and parallel netCDF [30], which provide a set of tools and libraries to help users create, organize, and manage high-volume complex datasets using their data models and formats. They support and optimize for various I/O and storage systems, and are portable across different platforms. Different optimization techniques are applied based on the I/O libraries and file systems used. When running the same scientific code in different environments, different I/O routines and configurations need to be applied to achieve optimal performance, which could require significant changes to the application code.

The Adaptive I/O System (ADIOS) [37] allows us to handle all of these differing architecture and configuration environments gracefully. Figure 1.9 shows the architecture of ADIOS and its interaction with scientific codes and I/O libraries. ADIOS provides a set of standard I/O routines, which can be dynamically configured by XML configuration files, without the need to rewrite and re-compile the code. Through this design, ADIOS provides extreme flexibility in allowing application scientists to tailor their code to a particular I/O optimization technique without modifying their source code, as well as achieve optimal performance on different platforms through dynamic configurations.

More importantly, for staging and *in situ* analytics middleware, ADIOS is extensible to include various auxiliary tools, such as analytic and monitoring codes, which provide a framework for *in situ* data analysis and processing. These analytics functions can be embedded within the I/O calls themselves to

provide transparent *in situ* analytics and metadata generation to applications, again without requiring source code changes.

ADIOS accomplishes this by describing data in the BP file format, a portable format compatible with popular scientific file formats such as HDF-5 and netCDF. The BP file format is designed to support *delayed consistency* (performing synchronization operations only at the start and end of each I/O operation to reduce synchronization delay and improve I/O throughput), lightweight *in situ* data characterization, and resilience. Each process writes its own output into a process group slot. These slots are variably sized based on the amount of data required by each process. Data characteristics for the variables, such as user-defined analytics, are included in process output. These slots can be padded to parallel file system stripe sizes to improve access performance.

1.4.1.3 Resource Overlap with Asynchronous Operations

Regardless of I/O forwarding or data staging, synchronizing I/O requests, even just for transfers from compute to I/O nodes, produces idle times for the compute nodes, something which must be avoided in peta-and-exascale applications. Therefore, asynchronous communication and I/O are of paramount importance for efficient large-scale applications. Underlying the need for asynchronous I/O and communication requests is the desire to utilize all resources concurrently, and minimize resource-to-resource dependencies. In the context of computing, this means to overlap transfers on the network and to disk with computation that further advances the application.

In hardware, the mechanism of separating the requirement of CPU involvement in memory transactions is called *Direct Memory Access (DMA)*. DMA-enabled hardware subsystems (such as disk controllers and network cards) can access memory independently of the CPU state, though cache coherence on DMA-enabled systems becomes more complicated. Expanding the concept of DMA across computational units, *Remote Direct Memory Access (RDMA)* enables devices on one node (in particular, the network adapter) to access the memory on another node, without CPU involvement. RDMA enables zero-copy, high-throughput, and low-latency networking for high-performance parallel compute clusters. Examples of modern RDMA infrastructures in supercomputing environments include InfiniBand, iWarp, Quadrics Elan, IBM BlueGene Interconnect, and Cray SeaStar [1].

To test the benefits of asynchronous operations in empirical terms, a detailed evaluation was conducted to study strategies of resource overlap in MPI-IO [50]. Different strategies are considered and evaluated for the overlap of I/O with computation and communication to show the performance benefits of asynchronous operations in a distributed context. Specifically, the following overlap strategies are presented and experimentally evaluated: 1) Overlapping I/O and communication; 2) Overlapping I/O and computation; 3) Overlapping computation and communication; and 4) Overlapping I/O,

communication, and computation. Experiments show that all of these techniques are effective, and bring performance improvements averaging 25% or greater on a number of applications and maxing out at 85% or greater, parameterized by buffer sizes and number of processors.

1.4.2 I/O Forwarding Middleware

As mentioned, I/O forwarding replaces direct disk access from compute nodes with transfers over the network to dedicated I/O nodes, where the data is subsequently read/written. Typically, as in the Blue-Gene architectures, there is a static compute-to-I/O node mapping. Thus, the primary goal of middleware implementing I/O forwarding is to make the operations as transparent as possible, as well as to supply users with common optimizations such as asynchronous operators. In other words, the user should be able to issue I/O commands from compute nodes as if the nodes had direct connections to the filesystem, and without knowledge of where the data goes en-route to/from the disk. A number of questions arise from these relatively simple goals:

1. Given the static compute-to-I/O node mapping, what is the best way to schedule the transfer of data to I/O nodes in collective I/O requests?
2. What opportunities exist for I/O nodes to optimize collective I/O from an arbitrary collection of compute nodes to arbitrary locations in disk?
3. What are some I/O specific optimizations (such as prefetching) that I/O nodes can perform to minimize I/O request completion time?
4. What application-specific phenomena, such as bursty I/O patterns that typically occur in scientific applications [41], can lead to problems such as exhausting memory resources in the pipeline? How does I/O forwarding handle these phenomena?

One example of a software implementation of I/O forwarding is the I/O Delegate Cache System (IODC) [47]. Figure 1.10 shows the architecture of the IODC system. The IODC divides system cores into two categories, Application Nodes and I/O Delegate Nodes (IOD nodes), using ROMIO [68] as its backend. IODC intercepts all I/O requests initiated by applications running on application nodes, and redirects them to IOD nodes. These nodes store data in a collective cache system that aggregates small redirects into larger ones for higher throughput. The IOD nodes run a *request detector*, intercepting I/O requests from application nodes. Upon interception, the IODs update their in-memory cache using MPI collective communication between themselves, and perform the redirect operation. To simplify cache coherence, the caching system partitions files into pages corresponding to the filesystem block size and only holds a single cached copy of files among the nodes. The IODC system is embedded in MPI-IO, allowing an abstraction of the underlying parallel file system and all redirection/cache operations to the application

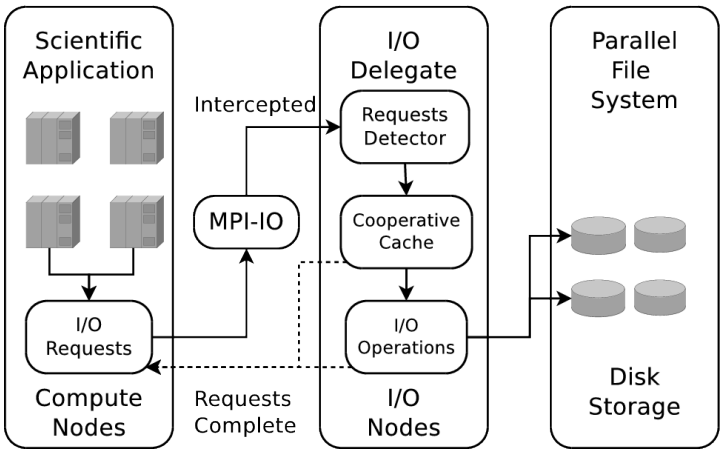


FIGURE 1.10: IODC architecture.

nodes. Therefore, any application using MPI-IO requires no changes to the source code.

1.4.3 Data Staging Middleware

The I/O forwarding mechanisms provide an efficient method of handling data coming to or from disk, while minimizing idle time for compute nodes. In this sense, the addition of data staging nodes represents an evolution of I/O forwarding into a model with a higher degree of flexibility, allowing more complex operations on the data as it makes its way to disk. Part of the expressive power of data staging includes the ability to allow users to define functionality on the staging nodes, opting for in-flight operations such as data analytics, reduction, and reorganization.

However, with this added flexibility comes an added responsibility for an application designer, as well as concerns for library developers. Note that many characteristics of data staging coincide with I/O forwarding, so the questions presented in Section 1.4.2 apply here as well. A sampling of concerns specific to data staging include:

1. How much additional effort must application designers expend in order to make efficient use of the staging nodes, both on compute nodes (data packaging or preprocessing for staging operators) and staging nodes (custom staging operators)?

2. What are optimal ways of taking arbitrary input from compute nodes, load-balancing the staging operator workload among multiple staging nodes, and utilizing I/O forwarding efficiently and in a pipelined manner?

The simplest staging middlewares exist to combat issues that come up in scientific application I/O patterns such as bursty I/O behavior. Decoupled and Asynchronous Remote Transfers (DART) [15] is an example framework to improve I/O through staging that consists of three components: the client layer, streaming server, and receiver. The client layer runs on compute nodes and links with the application. It notifies the streaming server when the application has data ready to transfer. The streaming server runs on service nodes and extracts data from compute nodes after receiving notification. It then transfers the data to the receiver running on remote nodes. It can also directly write data to local storage if the receiver is not specified. Using an additional layer between compute and I/O nodes allows DART to control in a finer-grain manner how to transfer data from the compute nodes to the I/O nodes, leading to smoother access patterns and resource utilization. A similar system is PDIO (Portals Direct I/O) [66], which adopts the three-tier architecture and supports wide-area network (WAN) data transfer. PDIO runs daemons on externally connected I/O nodes. The daemons receive messages and data from the clients on the compute nodes, aggregate them into memory buffers with optimal size, and asynchronously send them to remote receivers over the WAN via parallel TCP/IP data streams.

DataStager [1] provides an extensible framework for staging and I/O forwarding, providing mechanisms for scheduling and load-balancing transfers as well as allowing user-defined computation. The staging nodes serve as another layer to forward I/O for compute nodes, and also perform *in situ* data capture and analysis for visualization [16] and metadata generation. DataStager leverages RDMA-based network infrastructures and enables the staging area to take charge of data processing jobs that require high throughput, including synchronization, aggregation, collective processing, and data validation. The processing results can be used as input to a variety of extensible service pipelines such as data storage on disk, visualization, and data mining.

DataStager has two components: a library called DataTap and a parallel staging service called DataStager. DataTap is a client-side library co-located with the compute application. It is implemented with the ADIOS API in order to support asynchronous I/O and keeps application code changes to a minimum. DataTap uses an efficient, self-describing binary format, FFS [8] to mark up its binary data. The FFS format makes it possible for binary data to be analyzed and modified in transit, and enables the graph-structured data processing overlays, termed I/OGraphs [2]. The overlays can be customized for a rich set of backend uses, including *in situ* processing, online data visualization, data storage, and transfer to remote sites.

The staging service, DataStager, is comprised of server-side processes to actively read data from DataTap clients using RDMA techniques. It applies server-directed I/O for asynchronous communication to fetch data from compute nodes. Server-directed I/O lets the I/O nodes control the data transfers and hence the resources based on their capacity, which allows for smoother access patterns. In addition, the server controlled data transfer causes mini-

mal runtime impact by allowing the application to progress without actively pushing the data out. The server-directed I/O is particularly useful in HPC environments where a small partition of I/O nodes serves a large number of compute nodes. The disparity in the sizes of the partitions, accompanied with the bursty behavior of most scientific application I/O [41], can exhaust memory and I/O resources on the I/O nodes. DataStager uses resource-aware schedulers to select I/O operations to carry out. The selection of an operation is based on the memory space on I/O nodes and the status of the application nodes and schedulers (idle vs. non-idle, for example). If all schedulers agree to issue a transfer request, an RDMA read request is issued to the originating application node. Multiple requests may be serviced simultaneously if resources permit. Once the RDMA read is completed, the staging handler is notified and will then process the message according to the configuration. It can direct write data to disk, forward data to network, query and analyze data online, and so on.

Based on DataStager, an *in situ* data processing middleware called Preparatory Data Analytics (PreDatA) [80] has been built. PreDatA is an approach for preparing and characterizing data produced by the large-scale simulations running on peta-scale machines in an *in situ* manner. It exploits the computational power of staging nodes, and provides a pluggable framework for executing user-defined operations such as data re-organization, real-time data characterization, filtering and reduction, and select analysis. Data can be treated as streams and the operations can be specified in ways natural to the "streaming" context, so that streaming data processing techniques can be applied in PreDatA with little additional porting effort.

PreDatA middleware can be placed in both the compute nodes or the staging nodes. Data analysis and operations can be plugged and hosted in either location. When application performs I/O operations, PreDatA acquires output data through ADIOS and stages data from compute nodes to staging nodes. In-transit data processing is performed along the data flow. PreDatA schedules asynchronous data movement from compute nodes to staging nodes to minimize interference with the simulation. PreDatA supports user-defined data operations such as buffer management, scheduling, data indexing and query, and data exchange and synchronization across staging nodes. It also provides a pluggable framework for end users to specify, deploy, and debug data processing and analysis functions.

1.5 *In Situ* Data Layout Optimization

The data staging model provides opportunities for *in situ* data processing and analysis, which provides huge benefits for understanding the data without performing costly disk reads over the raw data for analytics. However, *in situ*

data processing has its limits. For instance, there simply is not enough memory available for analysis in a *global context*, which is essential for *exploratory data analytics*. Therefore, it is infeasible to perform all data analysis at simulation time, and data may have to be read multiple times for post-simulation analytics in a global context.

With this requirement in mind, there are a number of ways we can accelerate future analytics operations. Preferably in the staging process, applications can use data reorganization to optimize future post-processing. One scenario for this occurs in scientific databases, emphasizing query processing with heterogeneous constraints.

Scientific database technologies, especially when considering *in situ* computation, involve a number of considerations. First, what types of queries and data access patterns are these database systems optimized for (Section 1.5.1.1)? Second, what optimization techniques do databases currently apply for different access patterns? (Section 1.5.1.2 and Section 1.5.1.3)?

1.5.1 Basic Layout Optimization Techniques

Scientific simulation codes usually generate multi-dimensional spatio-temporal data, which is quite different from structured data stored in traditional relational databases, and thus requires different optimization techniques. Data accesses are usually accompanied by value and spatial constraints. For example, for a climate dataset, the user might want to know what regions within certain latitudinal and longitudinal ranges have abnormally high temperature values.

Different query types lead to different data access patterns, but contiguous access patterns are the most efficient. Thus, the goal of data layout optimization is to store data based on potential access patterns to achieve more contiguous access patterns in queries. But first, a discussion of the access patterns themselves, defined by query semantics, must be discussed.

1.5.1.1 Query Types and Data Access Patterns

We summarize common query types and data access patterns on scientific spatio-temporal data as follows:

Value-constrained: Queries that request spatial regions and/or their corresponding variable values, subject to constraints on those or other variable values. E.g., what (latitude,longitude) pairs at some time in a simulation have an abnormally high temperature? What are those temperature values? These queries are also known as range queries.

Region-constrained: Queries that request spatial regions and/or their corresponding variable values, subject to regional constraints. E.g., what are the temperature values within North Carolina at some time?

Value-and-region-constrained: Queries that request spatial regions and/or

their corresponding variable values, subjects to constraints on both the regions as well as variable values. E.g., what regions within North Carolina have an abnormally high temperature?

Besides the query types listed above, multi-variate and multi-resolution data access should also be considered. In multi-variate data access, multiple variables may be accessed with constraints on different variables. In multi-resolution data access, only part of the data in certain resolution is accessed to satisfy coarse-grained analytic requirements. These are also common access patterns for scientific spatio-temporal data, and require different layout optimization techniques to achieve best access performance.

1.5.1.2 Optimization for Value-constrained Queries

To optimize the data layout for value-constrained access, points with similar values should be stored together to achieve more contiguous data access. Data can be partitioned into *bins*, where each bin represents a set of similar values. *Value-constrained binning* is applied to assign data points in different bins based on their values, and data points within the same bins are stored together on the storage space to achieve contiguous data access patterns for value-constrained access.

A database popularized by fast value-constrained query performance is FastBit [75], a state-of-the-art bitmap indexing scheme, that applies different binning techniques to optimize for range queries. Bitmap indexing techniques traditionally employ any combination of 3 tasks: binning, encoding, and index compression. FastBit binning maps variables to bins so that variables of a similar property or value are co-located for quick lookup. For example, given a bin B that represents a range $[0, 5)$, for each record, a bit is used to represent whether the record falls into a bin or not. A bitmap vector is encoded using this technique for each defined bin. Since the space taken up by the bitmap vectors becomes unmanageable for large datasets, FastBit employs a Word-Aligned-Hybrid compression scheme [77], based on run-length encoding, to reduce the index size. To support and optimize for multi-variate queries, FastBit applies fast bitmap operations such as AND and OR among bitmaps of different variables to achieve fast selection. However, building the bitmap indexes is an expensive operation, lowering the capability for performing it *in situ*. Furthermore, the storage requirements for the indexes are on the order of the dataset size, adding significantly to I/O and storage costs.

ISABELA-QA [52] is a query engine based on ISABELA-compressed data (see Section 1.3.2.4), designed to occupy far less space than existing database technologies and indexes. Similar to FastBit, ISABELA-QA employs binning of values to cluster data with similar values. However, ISABELA-QA, rather than building bitmaps to identify particular values with bins, colocates the ISABELA-generated permuted indices for each window on a per-bin basis, taking advantage of the sorted nature of the windows to minimize the ensuing metadata. When processing queries, bin metadata

is fetched, compression windows are identified, and the $B-$spline coefficients and permuted indices are read. One particular advantage of this setup is that data points can be interpolated independently of one another, so only a subset of the window need to be decompressed. Furthermore, the storage footprint is nearly the same as a linear organization of the compression windows, occuring additional storage costs only for the metadata.

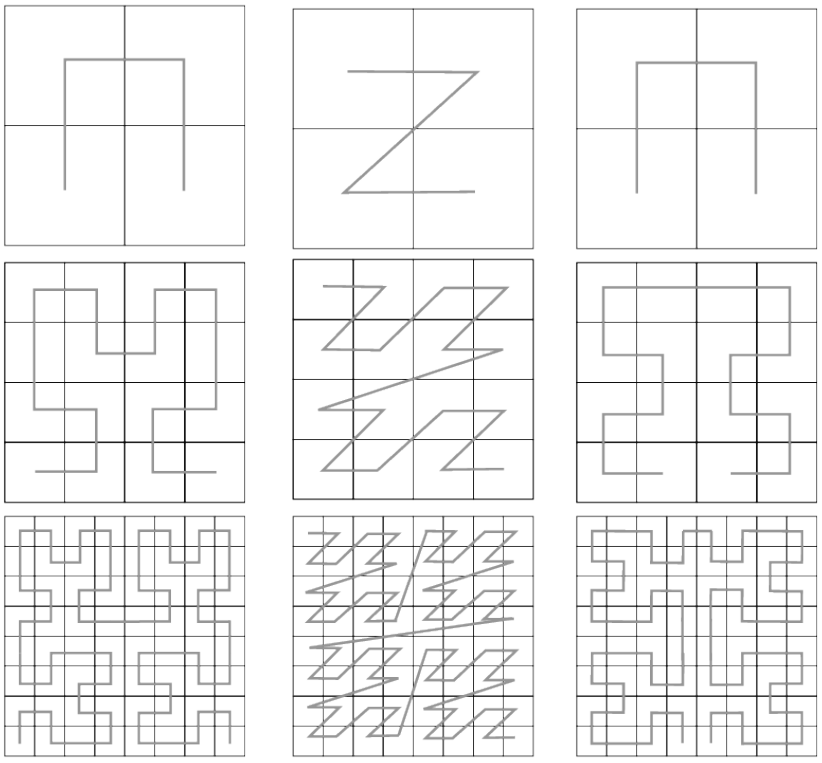
In a similar vein, ALACRI^{2}TY [25] fuses lossless compression and database indexing to produce a lightweight data and index representation. The method relies on the representation of floating point data in memory, consisting of an fractional mantissa component exponentiated by an exponent component, and given an explicit sign bit. The most significant bytes, containing the sign, exponent, and most significant mantissa components, are removed from the dataset and unique value encoding is used to compress them. These unique values then form the bin edges, and data is reorganized into bins corresponding to their high-order bytes. An *inverted index* map bins to record IDs to maintain the original mapping.

1.5.1.3 Optimization for Region-constrained Queries

Region-constrained queries usually access sub-volumes of multi-dimensional data, in which the points are spatially contiguous. However, multi-dimensional data needs to be linearized before being written to disks since storage space is one-dimensional. If data is linearized as it is stored in memory (e.g., row-major order for 2-D data), there will be high performance disparity when accessing data along different dimensions. Thus, the key challenge is how to linearize data to achieve better spatial locality.

One common technique is to divide multi-dimensional data into chunks. The chunking technique divides each dimensions into equal- or variate-sized partitions. The SciDB system [6] is a distributed spatial database system which applies chunking techniques to optimize for spatial-constrained queries [65]. Chunks are distributed over multiple nodes to achieve parallel data access. SciDB includes special optimizations for sparse matrices, and introduces overlapping areas to optimize for certain calculations.

A more fine-grained optimization is to use *space-filling curves (SFC)* [53] to linearize data on disks to improve spatial locality. Rather than storing data blocks in row-major or column-major order, data are re-organized in SFC order. The SFC order helps to ensure that spatially contiguous points in multi-dimensional space are also placed contiguously on disks, thus reducing potential seek operations in queries accessing certain spatial regions. Popular SFCs include Hilbert SFC, Z-order SFC, Moore SFC, and so on. An illustration of these are shown in Figure 1.11. Among all SFCs, Hilbert SFC has been demonstrated to have best preservation of spatial locality properties. A detailed analysis and comparisons of clustering achieved by Hilbert curves, Z-order curves, and Gray code curves is presented in [42] The Hilbert curve reduces the number of clusters by 50% compared with Z-order curve and Grey



(a) Hilbert curve. (b) Z-order curve. (c) Moore curve.

FIGURE 1.11: The first three iterations of Hilbert/Z-order/Moore space-filling curves.

code curve for 2-dimensional square and sphere access. For 3-dimensional cube access, Hilbert curve reduces the number of clusters by 45%, and for sphere access it reduces the number of clusters by 30% compared to the other space-filling curves.

Since space-filling curves are usually defined in a recursive manner, some SFCs can additionally be thought of as a hierarchical representation to support *multi-resolution* data access. A hierarchical indexing scheme based on Z-order SFCs [49] has been introduced. For each level of resolution, a subset of points is stored together in Z-order and lower levels of resolution contain fewer points than the higher resolutions. Higher resolution access can be always achieved by fetching data that are contiguously located on storage space.

1.5.2 Towards Heterogeneous Access Patterns

One major drawback of existing storage layout techniques is that they primarily focus on optimizations for a *particular* access pattern(s). For example, the SFC's mentioned in the previous section only improve performance for access patterns induced by spatial constraints on sub-planes/sub-volumes of the data space. For value-constrained access patterns, the entire dataset must be scanned to select qualified points. Similarly, FastBit and ISABELA-QA are optimized only for value-constrained accesses and cannot handle spatially-constrained accesses efficiently. A naïve approach to support multiple access patterns would be creating copies of dataset that favor different access patterns. However, the ever-increasing sizes of simulation data make multiple replications infeasible at extreme scale.

One potential way to address these challenges are to utilize multiple value-constraint-optimizing and region-constraint-optimizing techniques. One example of this is in a configurable, layered layout optimization technique, which iteratively applies SFC and ISABELA-compression to provide query processing performance improvements for numerous access patterns [20]. The layout scheme combines Hilbert space-filling curves and value-constrained binning, through ISABELA-QA, together. It divides data into blocks and interleaves these blocks across value-constrained bins along Hilbert space-filling curves to optimize for both value and region queries with both value and spatial constraints. Based on this scheme, MLOC, a flexible multi-level framework has been presented [21]. MLOC applies a flexible, hierarchical multi-level architecture. Multiple fine-grained optimization techniques can be flexibly placed within the framework, to achieve optimizations for multiple access patterns in user-defined priority. Users can specify the order of the optimization techniques within the framework to generate a different layout on storage based on the frequency of access patterns induced by real queries.

1.6 Limitations and Future Directions

While there is a large amount of pioneering work being performed to bring robust, efficient *in situ* processing to reality, the problems necessitating *in situ* processing, namely, I/O bottlenecks in large-scale systems, remains far from being solved. Specifically, there are a number of limitations with the current state of the art that must be improved upon in the coming years as compute capability continues to scale at a faster rate than I/O capability.

1.6.1 Data Analysis

In situ data analysis methods, some of which are identified in this chapter, have shown impressive progress in taking advantage of large-scale data resident in memory to accelerate knowledge discovery. In spite of this, it is infeasible to perform *all* data analytics at simulation write time, especially for interactive processes such as query processing. Furthermore, the act of aligning simulation code and staging architectures with scalable analytics kernels is a non-trivial problem.

For data analysis methods which are feasible to perform *in situ*, it is crucial to continue advancing the state of the art, developing and optimizing scalable, robust analysis algorithms, especially for increasingly complex simulation datasets and analytics scenarios.

1.6.2 Data Reduction

Recent compression algorithms, focused on scientific data, have advanced both compression speed and compression ratio for *hard-to-compress* double-precision datasets, which scientific simulations predominantly use for accurate computation. However, it has been observed that average lossless data reduction rates for these datasets, so far, fall in the range of 30% [56]. While this degree of reduction is significant for datasets in the terabytes, petabytes and beyond, it does not come close to bridging the performance trends gap of compute and I/O. While lossy compression can be a solution to this, with far higher data reduction rates, the loss of precision may not be acceptable to application scientists, simulations requiring pristine data for checkpointing, and analysis functions sensitive to changes in precision.

In order to achieve higher compression ratios, it is necessary to exploit all possible relationships in the data, creating a *context-sensitive* approach to lossless compression. For example, FPZIP takes advantage of spatial correlations between data points for prediction and ISOBAR takes advantage of similarities in the double-precision data representation for compression. These techniques are a good start, but more research taking advantage of these and other characteristics need to be performed. For instance, is it possible to use temporal relationships in compression, and if so, how to maintain the necessary information between simulation timesteps within a staging framework? Are there correlations between grid nodes of a particular level of resolution in Adaptive Mesh Refinement (AMR) simulations?

For lossy compression, it is crucial that the loss of precision be able to be quantified and bounded so that application scientists can reason about the tradeoffs between data accuracy, analysis accuracy, and data reduction. For instance, ISABELA provides such a bound at compression time through error encoding, trading off additional storage cost for guaranteed accuracy bounds [29]. As with lossless compression, increasing the context-sensitivity

of compression techniques are necessary to see huge reductions of data without suffering too much loss of precision.

Furthermore, compression methodologies must be compatible with staging architectures in order for the benefits of data reduction to translate into I/O cost reductions. Effective pipelining of compression and I/O operations is a necessity to hide compression costs in terms of CPU time. Initial work, for example, has been done pipelining ISOBAR-compression of byte columns with I/O operations using ADIOS [55], resulting in both reduced storage footprints and write times. The trick behind the performance seen in the ISOBAR pipelining method is the semantic division of tasks (e.g., multiple compression streams for both compressible and incompressible byte streams).

1.6.3 Architecture

The march of new hardware trends is a constant in the computing world, and heterogeneity is increasing in HPC systems as a result. Nearly all compute clusters are comprised of multi-core processors, and alternative computing hardware such as GPUs and FPGAs are highly active in the research community as a method to achieve a high computation rate with lower relative power consumption. There are a number of implications for *in situ* computation as a result. First, how do we best utilize and integrate emerging hardware architectures, such as GPUs and solid state drives (SSDs), within a robust *in situ* framework? Second, access to CPU main memory is required for network and I/O operations, memory transfer to co-processors (such as GPUs, with discrete main memory), not to mention the simulation computations themselves. With so many competing resources for main memory, how does one additionally include *in situ* operations in such a way as to not be bottlenecked by memory bandwidth and to optimize multicore/memory hierarchy usage?

On an intra-node level, the increasing complexity of multicore processors requires new ways of thinking about how computation maps to resources and how to best utilize the cache hierarchy, which can be performed at both the application and compiler levels [26, 35]. *In situ* processing techniques must take advantage of these architectural advances and utilize richer forms of inter- and intra-node parallelism (e.g., MPI and OpenMP).

On an off-chip level, new technologies such as SSDs, GPUs, etc. pave the way towards new and exciting *in situ* capabilities. SSDs, while being more expensive and having less data density than traditional HDDs, have low power consumption and far better random I/O rates, making them well suited for local disk caches on compute nodes. These would allow for *in situ* computations and analysis requiring too much memory to be performed in memory alongside simulation data. GPUs provide the possibility to accelerate compute-intensive *in situ* operations and allow CPUs to instead perform data management tasks.

Conclusion

It is highly unlikely that there will be a "silver bullet" solution to the growing disparity between computational capability and I/O bandwidth. This means that new programming paradigms must be adopted to continue the phenomenal increases in the ability to simulate, solve, and/or analyze larger and more complex problems into the exascale. *In situ* processing of data is a strong first step in this direction. Rather than cutting corners to allow scientific codes to complete in a reasonable amount of time, *in situ* algorithms can exploit the far superior compute performance to allow meaningful computation at simulation time, eliminating the need for costly post-processing.

There are many such cases where *in situ* computation makes sense in the data analysis realm, which have been presented here; the take-away point is that we get insightful analysis of data at minimal cost by performing the operations while waiting on I/O. Furthermore, we can prepare the data *in situ* for future analyses, which is essential for efficient database operations such as query processing. Rather than incurring the enormous cost of reorganizing the whole database from disk, the data can be shuffled at write time for a far more efficient database construction methodology. Finally, we can directly optimize the application on multiple fronts using *in situ* compression techniques: lossless (and especially lossy) compression reduces the I/O pressure and storage costs, while making use of the huge aggregate throughput of compute clusters.

Given the number of operations we can perform *in situ*, we can ensure that, when I/O becomes rate-limiting for scientific codes, there are ways in which we can enable maximum utilization of the underlying hardware and continue the relentless pace of scientific discovery provided by supercomputing.

Bibliography

[1] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. 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.

[2] H. Abbasi, M. Wolf, and K. Schwan. LIVE data workspace: A flexible, dynamic and extensible platform for petascale applications. *Cluster Computing, IEEE International Conference on*, 0:341–348, 2007.

[3] J. Ahrens, K. Heitmann, M. Petersen, J. Woodring, S. Williams, P. Fasel, C. Ahrens, H. Chung-Hsing, and B. Geveci. Verifying scientific simulations via comparative and quantitative visualization. *IEEE Computer Graphics and Applications*, pages 16 –28, 2010.

[4] J. Ahrens, J. Woodring, S. Williams, C. Brislawn, S. Mniszewski, P. Fasel, J. Patchett, and L. Lo. Visualization and data analysis challenges and solutions at extreme scales. In *Proceedings of the Scientific Discovery through Advanced Computing Conference*, 2011.

[5] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, Nov. 2000.

[6] P. G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 963–968, New York, NY, USA, 2010. ACM.

[7] M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. In *IEEE Data Compression Conference*, pages 293–302, 2007.

[8] F. Bustamante, G. Eisenhauer, and K. S. Widener. Efficient wire formats for high performance computing. In *In Proceedings of Supercomputing 2000*, pages 39–39, 2000.

[9] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th annual Linux Showcase and Conference - Volume 4*, pages 28–28, Berkeley, CA, USA, 2000. USENIX Association.

[10] J. Chen, Y. Kusurkar, and D. Silver. Distributed feature extraction. In *Proceedings of the Conference on Visualization and Data Analysis*, pages 189–195, 2002.

[11] J. Chen, D. Silver, and L. Jiang. The feature tree: Visualizing feature tracking in distributed AMR datasets. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 103 –110, 2003.

[12] J. Chen, D. Silver, and M. Parashar. Real time feature extraction and tracking in a computational steering. In *Proceedings of the High Performance Computing Symposium*, pages 155–160, 2003.

[13] A. Choudhary, D. Honbo, P. Kumar, B. Ozisikyilmaz, S. Misra, and G. Memik. Accelerating data mining workloads: Current approaches and future challenges in system architecture design. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, pages 41–54, 2011.

[14] I. Daubechies. Orthonormal Bases of Compactly Supported Wavelets II: Variations on a Theme. *SIAM J. Math. Anal.*, 24:499–519, March 1993.

[15] C. Docan, M. Parashar, and S. Klasky. DART: A Substrate for High Speed Asynchronous Data IO. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 219–220, New York, NY, USA, 2008. ACM.

[16] N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. E. Jansen. The ParaView coprocessing library: A scalable, general purpose in situ visualization library. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, October 2011.

[17] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, New York, NY, USA, 2011. ACM.

[18] A. Ganguly, J. Gama, O. Omitaomu, M. Gaber, and R. Vatsavai. *Knowledge Discovery from Sensor Data*. CRC Press, Taylor & Francis, 2009.

[19] B. Goeman, H. Vandierendonck, and K. D. Bosschere. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *Seventh International Symposium on High Performance Computer Architecture*, pages 207–216, 2001.

[20] Z. Gong, S. Lakshminarasimhan, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. F. Samatova. Multi-level layout optimization for efficient spatio-temporal queries on ISABELA-compressed data. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '12, 2012.

[21] Z. Gong, T. Rogers, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. 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.

[22] R. Gonzalez and R. Woods. *Digital Image Processing (2nd Edition)*. Prentice Hall, 2002.

[23] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[24] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak. Out-of-core compression and decompression of large n-dimensional scalar fields. *Computer Graphics Forum*, 22:343–348, 2003.

[25] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, N. Shah, E. R. Schendel, S. Ethier, C. Chang, J. Chen, H. Kolla, S. Klasky, R. Ross, and N. F. Samatova. Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. *Proceedings of the 23rd International Conference on Database and Expert Systems Applications (DEXA)*, 2012.

[26] M. Kandemir, T. Yemliha, S. Muralidhara, S. Srikantaiah, M. Irwin, and Y. Zhang. Cache topology aware computation mapping for multicores. *Programming Language Design and Implementation*, pages 74–85, 2010.

[27] S. Klasky, H. Abbasi, J. Logan, et al. In situ data processing for extreme scale computing. In *Proceedings of the Scientific Discovery through Advanced Computing Program (SciDAC)*, Denver, CO, USA, 2011.

[28] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In *Euro-Par*, 2011.

[29] S. Lakshminarasimhan, N. Shah, S. Ethier, S.-H. Ku, C. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. ISABELA for effective in-situ compression of scientific data. *Concurrency and Computation: Practice and Experience*, 2012. (In Press).

[30] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[31] J. Li, Y. Liu, W. k. Liao, and A. Choudhary. Parallel data mining algorithms for association rules and clustering. In *Handbook of Parallel Computing: Models, Algorithms, and Applications*. CRC Press, 2007.

[32] Y. Li. *Dark Matter Halos: Assembly, Clustering and Sub-halo Accretion*. PhD thesis, University of Massachusetts - Amherst, 2010.

[33] D. Lignell, J. Chen, T. Lu, and C. Law. Direct numerical simulation of extinction and reignition in a nonpremixed turbulent ethylene jet flame. *Western States Section Meeting of the Combustion Institute*, 2007.

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

[35] J. Liu, Y. Zhang, W. Ding, and M. Kandemir. On-chip cache hierarchy-aware tile scheduling for multicore machines. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 161 –170, 2011.

[36] Y. Liu, W. K. Liao, and A. Choudhary. Design and evaluation of a parallel HOP clustering algorithm for cosmological simulation. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, pages 82–89, 2003.

[37] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

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

[39] K. L. Ma, C. Wang, H. Yu, and A. Tikhonova. In-situ processing and visualization for ultrascale simulations. *Journal of Physics: Conference Series*, 78(1):012043, 2007.

[40] A. Mascarenhas, R. W. Grout, P. T. Bremer, E. R. Hawkes, V. Pascucci, and J. H. Chen. Topological feature extraction for comparison of terascale combustion simulation data. In *Topological Methods in Data Analysis and Visualization*. Springer Berlin Heidelberg, 2011.

[41] E. L. Miller and R. H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 567–576, New York, NY, USA, 1991. ACM.

[42] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13:124–141, 2001.

[43] C. Muelder and K. L. Ma. Rapid feature extraction and tracking through region morphing. Technical Report CSE-2007-25, University of California at Davis, 2007.

[44] H. Nagesh, S. Goil, and A. Choudhary. A scalable parallel subspace clustering algorithm for massive datasets. In *Proceedings of the International Conference on Parallel Processing*, pages 477–483, 2000.

[45] H. Nagesh, S. Goil, and A. Choudhary. Parallel algorithms for clustering high-dimensional large-scale datasets. In *Data Mining for Scientific and Engineering Applications*. Kluwer Academic Publishers, 2001.

[46] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A benchmark suite for data mining workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 182 –188, 2006.

[47] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press.

[48] B. Ozisikyilmaz, R. Narayanan, J. Zambreno, G. Memik, and A. Choudhary. An architectural characterization study of data mining and bioinformatics workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pages 61 –70, 2006.

[49] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, pages 2–2, New York, NY, USA, 2001. ACM.

[50] C. M. Patrick, S. Son, and M. Kandemir. Comparative evaluation of overlap strategies with study of I/O overlap in MPI-IO. *SIGOPS Oper. Syst. Rev.*, 42:43–49, October 2008.

[51] P. Pebay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. Technical Report SAND2008-6212, Sandia National Laboratories, 2008.

[52] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S-H Ku, S. Ethier, J. Chen, C.S. Chang, S. Klasky, R. Latham, R. Ross and N. F. Samatova. ISABELA-QA: Query-driven data analytics over ISABELA-compressed scientific data. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2011)*, Seattle, Washington, 2011.

[53] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY.

[54] R. Samtaney, D. Silver, N. Zabusky, and J. Cao. Visualizing features and tracking their evolution. *Computer*, pages 20–27, 1994.

[55] E. Schendel, S. Pendse, J. Jenkins, D. Boyuka II, Z. Gong, S. Lakshminarasimhan, Q. Liu, H. Kolla, J. Chen, S. Klasky, R. Ross, and N. Samatova. ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization. *HPDC*, pages 61–72, 2012.

[56] 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. *International Conference on Data Engineering (ICDE)*, 2012.

[57] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *First USENIX Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Monterey, CA, 2002.

[58] P. Schwan. Lustre: Building a file system for 1,000-node clusters. In *Proceedings of the Linux Symposium*, 2003.

[59] N. Shah, Y. Shpanskaya, C. S. Chang, S. H. Ku, A. V. Melechko, and N. F. Samatova. Automatic and statistically robust spatio-temporal detection and tracking of fusion plasma turbulent fronts. In *Proceedings of the Scientific Discovery through Advanced Computing Conference*, 2010.

[60] S. Shekhar, V. Gandhi, P. Zhang, and R. R. Vatsavai. *Availability of Spatial Data Mining Techniques*, pages 63–85. SAGE Publications, 2009.

[61] S. Shekhar, R. R. Vatsavai, and M. Celik. Spatial and spatiotemporal data mining: Recent advances. In *Next Generation of Data Mining*. CRC Press, 2008.

[62] S. Shekhar, P. Zhang, Y. Huang, and R. Vatsavai. Trends in spatial data mining. *Science*, pages 357–379, 2003.

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

[64] D. Silver and X. Wang. Tracking scalar features in unstructured datasets. In *Proceedings of the IEEE Conference on Visualization*, pages 79–86, 1998.

[65] E. Soroush, M. Balazinska, and D. Wang. ArrayStore: A storage manager for complex parallel array processing. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, New York, NY, USA, 2010. ACM.

[66] N. T. B. Stone, D. Balog, B. Gill, B. Johanson, J. Marsteller, P. Nowoczynski, D. Porter, R. Reddy, J. R. Scott, D. Simmel, J. Sommerfield, K. Vargo, and C. Vizino. PDIO: High-performance remote file I/O for Portals enabled compute nodes. In *in Proceedings of the 2006 Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV*, 2006.

[67] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–189, Washington, DC, USA, 1999. IEEE Computer Society.

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

[69] D. Thompson, R. Grout, N. Fabian, and J. Bennett. Detecting combustion and flow features in situ using principal component analysis. Technical Report SAND2009-2017, Sandia National Laboratories, 2009.

[70] 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(9):092505, 2006.

[71] C. Wang, H. Yu, and K. L. Ma. Importance-driven time-varying data visualization. *IEEE Transactions on Visualization and Computer Graphics*, pages 1547–1554, 2008.

[72] J. Wei, H. Yu, R. Grout, J. Chen, and K. L. Ma. Dual space analysis of turbulent combustion particle data. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 91 –98, 2011.

[73] J. Wei, H. Yu, and K. L. Ma. Parallel clustering for visualizing large scientific line data. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization*, pages 47–56, 2011.

[74] J. Woodring, K. Heitmann, J. Ahrens, P. Fasel, C. H. Hsu, S. Habib, and A. Pope. Analyzing and visualizing cosmological simulations with ParaView. *The Astrophysical Journal Supplement Series*, 2011.

[75] K. Wu. FastBit: An efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 16(1):556, 2005.

[76] K. Wu, W. Koegler, J. Chen, and A. Shoshani. Using bitmap index for interactive exploration of large datasets. In *Proceedings of the International Conference on Scientific and Statistical Database Management*, pages 65–74, 2003.

[77] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, SSDBM '02, pages 99–108, Washington, DC, USA, 2002. IEEE Computer Society.

[78] S. Yiannakis and J. E. Smith. The predictability of data values. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 248–258, Washington, DC, USA, 1997. IEEE Computer Society.

[79] H. Yu, C. Wang, R. Grout, J. Chen, and K. L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Application*, pages 45–57, 2010.

[80] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA: Preparatory data analytics on peta-scale machines. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.