Scalable In Situ Scientific Data Encoding for Analytical Query Processing

Sriram Lakshminarasimhan [1,2,+], David A. Boyuka II [1,2,+], Saurabh V. Pendse [1,2], Xiaocheng Zou [1,2], John Jenkins [1,2], Venkatram Vishwanath [3], Michael E. Papka [3,4], Nagiza F. Samatova [1,2,*]

[1] North Carolina State University, Raleigh, NC 27695, USA

[2] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA

[3] Argonne National Laboratory, Argonne, IL 60439, USA

[4] Northern Illinois University, DeKalb, IL 60115, USA

[*] Corresponding author: samatova@csc.ncsu.edu [+] Authors contributed equally

ABSTRACT

The process of scientific data analysis in high-performance computing environments has been evolving along with the advancement of computing capabilities. With the onset of exascale computing, the increasing gap between compute performance and I/O bandwidth has rendered the traditional method of post-simulation processing a tedious process. Despite the challenges due to increased data production, there exists an opportunity to benefit from "cheap" computing power to perform query-driven exploration and visualization during simulation time. To accelerate such analyses, applications traditionally augment raw data with large indexes, post-simulation, which are then repeatedly utilized for data exploration. However, the generation of current state-of-the-art indexes involve a compute- and memory-intensive processing, thus rendering them inapplicable in an *in situ* context.

In this paper we propose DIRAQ, a parallel *in situ*, *in network* data encoding and reorganization technique that enables the transformation of simulation output into a query-efficient form, with negligible runtime overhead to the simulation run. DIRAQ begins with an effective core-local, precision-based encoding approach, which incorporates an embedded compressed index that is $3-6$x smaller than current state-of-the-art indexing schemes. DIRAQ then applies an *in network* index merging strategy, enabling the creation of aggregated indexes ideally suited for spatial-context querying that speed up query responses by up to 10x versus alternative techniques. We also employ a novel aggregation strategy that is topology-, data-, and memory-aware, resulting in efficient I/O and yielding overall end-to-end encoding and I/O time that is less than that required to write the raw data with MPI collective I/O.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing Methods— *inverted index, aggregation*; D.4.2 [**Storage Management**]: Secondary storage—*data compression, parallel storage*

HPDC'13, June 17–21, 2013, New York, NY, USA.

Keywords

exascale computing; indexing; query processing; compression;

1. INTRODUCTION

In high-performance computing (HPC), the concept of *in situ* processing, or processing data at application run time and in application memory, is one of increasing importance. The traditional approach of performing data processing, analysis, etc. as a post-processing step is becoming a rate-limiting factor as application data sizes increase faster than I/O capabilities. Recent research has been investigating the design space and implications of *in situ* processing and data staging frameworks to facilitate this model [1–3, 25, 33]

While the concept of *in situ* processing has been realized in such areas as visualization [19, 24, 31] and analysis frameworks [33], in this paper we focus specifically on index generation. Such indexing enables the acceleration of tasks, such as exploratory and query-driven analysis, that may not themselves be amenable to *in situ* processing, thus indirectly reducing time-to-analysis. This approach to supporting *query-driven analytics* for large-scale data has only just begun to be studied. Recently, the bitmap indexing technique FastBit [21, 28] has been applied in parallel with FastQuery [5, 8, 9] and extended to demonstrate indexing in an *in situ* context [17].

However, in order to extend *in situ* indexing to the production context of high core count application runs, several challenges must first be overcome. Most index generation processes are both computationally expensive and storage intensive, incurring significant processing and I/O overhead. These are opposed to one of the central goals of *in situ* computation: to minimally disturb application run time. Furthermore, as indexing in a global context is prohibitively expensive due to the need for global coordination, current methods of index generation produce *fragmented* indexes across compute resources, which significantly increases query response time. Related to these overheads is the memory-intensive nature of indexing, placing hard constraints on the memory overhead of indexing and limiting the degree of aggregation that can take place.

To address these challenges, we propose a methodology for Data Indexing and Reorganizing for Analytics-induced Query processing (DIRAQ). The following contributions enable us to make inroads towards a storage-lightweight, resource-aware data encoding technique that incorporates a query-efficient index:

Storage-lightweight, Query-optimized Data Encoding We describe an encoding technique that converts raw floating-point data into a compressed representation, which incorporates a *compressed inverted index* to enable optimized query access, while also exhibiting a total storage footprint *less than* that of the original data. We exploit the spatio-temporal properties of the data by leveraging our previous work with ALACRITY [16], augmenting it with a highly-compressed inverted index using a modified version of the PForDelta compression [34] algorithm.

Scalable, Parallel Data Reorganization For fixed-size groups of processes, we "defragment" indexes to optimize query performance by developing an *in network* aggregation and merge technique tailored to our encoding, which distributes the computation equally among all compute cores in the group and allows arbitrary selection of aggregator cores to gather/write the resulting data. This way, we avoid the pitfalls of serializing the encoding process at various stages.

Resource-aware Aggregation We additionally make our group-wise indexing *resource-aware*, dynamically learning optimal data paths and choices of aggregators through per-group neural network modeling that supports online feedback. The optimization space for the model is constrained by the available memory, ensuring memory constraints are not violated.

Our proposed method shows promising results on 9 datasets from the FLASH astrophysics simulation [11] and 4 datasets from S3D combustion simulation [7]. Our encoding reduces the overall storage footprint versus the raw data by a factor of 1.1–1.8x, and versus FastQuery-indexed data by 3–6x. Our scalable reorganization and aggregation method combined with our encoding allows up to 6x to-disk throughput improvement compared to MPI-IO on the raw data. Finally, query performance on our defragmented indexes is improved by up to 10x versus FastQuery-generated bitmap indexes.

2. RELATED WORK

In this section we cover previous work related to *in situ* processing, *in situ* indexing and aggregation strategies for I/O.

The onset of petascale and exascale computation has seen a significant growth in works that encompass simulation-time processing relating to *in situ* and *in network* processing [4, 19, 31], along with several data staging systems such as JITStaging [1], DataStager [3], DataTap [2], PreDatA [33] and GLEAN [25] that explore movement of simulation data to co-located clusters for processing. The DIRAQ pipeline has been carefully designed to complement staging-driven approaches for generating indexes; parts of the index merging process can be offset to staging routines, but is not the focus of this paper.

Distributed and parallel indexing itself has been well researched in the database community. The majority of the indexing techniques are variants of the commonly used B-Tree indexing technique, which have been shown to have sub-optimal performance for many workloads over read-only scientific datasets, when compared to other techniques such as bitmap indexing [27]. The parallel indexing scheme FastQuery [5,8,9] and subsequent *in situ* work [17], which extend the WAH-compressed bitmap indexing method FastBit [21,28], share the same overarching goal as DIRAQ. However, DIRAQ differs in that it utilizes *in situ* index aggregation over a larger spatial context, instead of concatenating indexes from each core, making it more suitable for analytics, and because it explicitly addresses issues such as including network aggregation and limited I/O throughput.

In contrast to a post-processing context, performing indexing *in situ* demands special attention to scalable I/O, an area with much prior work. MPI collective I/O is the canonical approach to this problem, which typically incorporates a two-phase I/O technique [23] to *aggregate* data into fewer, larger requests to the filesystem. This principle has been refined in various ways, including a 3-phase collective I/O technique with hierarchical Z-order encoding [18], and pipelined aggregation enabling the overlap of computation and threaded I/O [12, 13]. However, while these approaches are well-suited to a variety of common I/O patterns, indexing introduces an irregular access pattern. To overcome this, we opt for a customized I/O aggregation strategy that includes *in network* merging of core-local encoded data.

As for optimizing the I/O aggregation process, recent work has been performed on auto-tuning the number of aggregators involved in MPI collective I/O [6]. Depending on the amount of data written out with each group, they either merge or split process groups (indirectly changing the aggregator ratio) to better utilize the available I/O bandwidth. In this paper, we group compute processes that belong to the same processor set (*pset*) since I/O forwarding is done on a per-*pset* level. While they use process mapping based on topology, we employ the aggregator placement to be topology-aware as well. Additionally, our method tunes aggregators within a group, rather than changing the underlying group size.

3. BACKGROUND

3.1 ALACRITY - Indexing for Scientific Data

We use our previous work with ALACRITY [16] as the starting point for indexing in DIRAQ. ALACRITY is a storage-lightweight indexing and data reduction method for floating-point scientific data that enables efficient range query with position and value retrieval. Specifically, ALACRITY is optimized to identify positions satisfying range conditions (e.g. "temperature > 2500") and efficiently retrieves the values associated with those points. It achieves this by utilizing a byte-level binning technique to simultaneously compress and index scientific data. Because ALACRITY integrates data reduction with indexing, it exhibits a much lower storage footprint relative to existing indexing methods. For instance, while ALACRITY's total footprint (data + index) is consistently about 125% of the raw data for double-precision datasets, a typical FastBit [26] encoding over the same data requires $\approx 200\%$, and a B+-Tree may require more than 300% [27].

The key observation in ALACRITY is that, while floating-point datasets have a large number of unique values, the values are still clustered and do not span the entire floating point domain. If we examine the most significant k bytes of these floating point values (typically $k = 2$), this value clustering translates into a list with much lower cardinality. This is because the IEEE floating point format defines the highest k bytes (which we denote as *high-order bytes*) to contain the sign bit, exponent bits, and most significant mantissa bits of the value. The high-order bytes will therefore exhibit lower cardinality than the *low-order bytes*, which typically contain much more variation and noise.

ALACRITY leverages this observation by binning on the high-order bytes of the data. Because the exact high-order bytes are stored as bin "header values," this information does not need to be repeated for each value, and so the data is substantially reduced by storing only the low-order bytes for each datapoint. As a property of the floating-point format, each bin contains points belonging to a single, contiguous value range. Therefore, by properly ordering the bins, range queries can be answered by reading a contiguous range of bins in a single, contiguous read operation, significantly

reducing the number of seeks necessary to support value retrieval.

However, because the binning operation rearranges the values, an index is required to maintain the original ordering. In the original paper, we explore two alternatives: a "compression index" and an inverted index. The compression index was shown to be effective in providing data compression, but is not appropriate for querying, and so we do not consider it in this paper. The inverted index, while larger, is still lightweight, with a storage requirement of only 50% of the original data size for double-precision data, and is effective for querying.

While ALACRITY works well in the context of serial indexing, we must develop additional methods in order to support parallel indexing in DIRAQ. In particular, although ALACRITY's total storage size is notably smaller than previous methods, it still represents a non-trivial I/O overhead beyond raw data, and would be expensive if applied as-is for *in situ* indexing. Additionally, index merging was not previously considered, as ALACRITY operated using full-context data. However, in this paper, we analyze the end-to-end performance of indexing, starting from core-local data generation to data capture on storage devices.

3.2 PForDelta - Inverted Index Compression

PForDelta [34] (standing for Patched Frame-of-reference Delta) is a method for efficiently compressing inverted indexes, and is frequently used in the context of indexing and search over unstructured data, such as documents and webpages [29,32]. As ALACRITY presents a viable inverted index-based method for encoding scientific data, it presents a perfect opportunity to further reduce storage overhead by applying PForDelta.

PForDelta encoding operates on a stream of sorted integer values, divided into fixed-size chunks. Each chunk is transformed to encode the first value (the frame of reference) and the differences between consecutive values. A fixed bit width b is then selected to encode the majority of deltas – those remaining are stored as *exceptions* in a separate *exception list*. The majority of deltas typically require a far fewer number of bits to encode than the original data, and a relatively small chunk size (128 elements in the original work) enables a highly adaptive, per-chunk selection of b.

4. METHOD

4.1 Overview

Figure 1 illustrates the logical flow behind DIRAQ *in situ* indexing. Our method consists of the following components:

1. **Storage-lightweight, query-optimized data encoding** that produces a compressed form of the data with an integrated index, combining both data *and* index compression. By achieving a low storage requirement while also supporting optimized range queries, this encoding approach enables us to surmount the obstacle that the HPC I/O bottleneck represents to *in situ* indexing. This component is explained in Section 4.2.

2. **Scalable, parallel data reorganization**, which reduces I/O time and improves post-simulation query performance by aggregating encoded data, without incurring unacceptable global communication costs. For comparison, existing indexing methods produce either a single, global index, or a highly-fragmented set of per-core indexes. Unfortunately, the former can only be accomplished with post-processing or expensive global communication, and the latter results in degraded query performance (as demonstrated in Section 5.2). In contrast, our group-level index aggregation technique largely avoids both of these drawbacks. We also leverage the hardware-level RMA support in modern HPC clusters to perform fast *in network* merging. These aspects of DIRAQ are discussed in Section 4.3.

3. **Resource-aware dynamic aggregator selection**, which incorporates the inherent *network topology* and *available memory* constraints of a running system to improve the choice of aggregator cores at runtime. Leveraging this information helps to achieve improved performance over MPI-IO two-phase I/O [10]. More detail is given in Section 4.4.

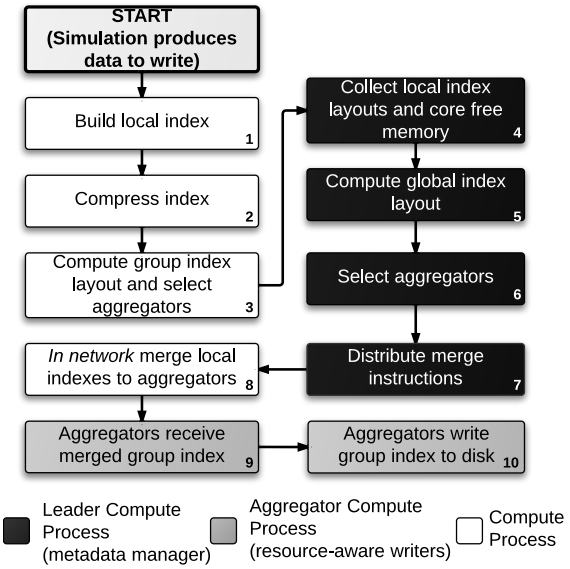


Figure 1: Overview of the DIRAQ indexing methodology, with lightweight encoding[1,2], scalable index aggregation[8,9] and resource-aware dynamic aggregator selection [6].

4.2 Lightweight, Query-optimized Data Encoding

The first step for DIRAQ is to address the fundamental challenge of *in situ* processing, and indexing in particular: because simulation I/O blocking time is already a serious concern today, any *in situ* process must ensure this metric is not substantially impacted. However, while there exist parallel indexing methods for scientific data, no current method has addressed this issue sufficiently to operate within the context of a running simulation. The root of the problem is two-fold. First, current indexing methods inherently *increase* total storage footprint (data + index), commensurately increasing I/O times. Second, computation time for current indexing methods dominates even the I/O time.

We address this problem in DIRAQ by extending ALACRITY, a storage-lightweight indexing method with integrated *data compression*, by augmenting it with *index compression* to achieve even higher overall compression. By thus reducing both components of the output, we reduce the storage footprint to **55%-90%** (a compression ratio of 1.1-1.8x) for 9 variables from FLASH [11] simulation data (Table 1). Note that these compression ratios include both index *and* data; this implies the data encoding can simultaneously support efficient range queries *while also reducing total simulation I/O*, a win-win scenario.

As with ALACRITY, the DIRAQ query-optimized encoding format enables two-sided range queries of the form $a < var < b$ to be evaluted efficiently (a or b may also be $-\infty$ or $+\infty$, respectively,

allowing one-sided range queries, as well). Given such a constraint, DIRAQ can retrieve the matching values (value retrieval) and/or the matching dataset positions (position retrieval). Additionally, DIRAQ supports an approximate query mode that accepts a bounded per-point error in order to perform an index-only query, which has benefits to query performance. These query types are discussed in more detail in a previous paper [16]. In pseudo-SQL syntax, the index supports queries of the forms:

```
SELECT var [and/or] position WHERE a < var < b
SELECT ~var [and/or] position WHERE a < ~var < b
```

An overview of the encoding format produced by DIRAQ is depicted in Figure 2. The process of binning is largely similar to that presented in the ALACRITY paper [16], as reviewed in Section 3.1, although some modifications have been made (including a generalization to support bit-level, rather than byte-level, adjustment of the high/low-order division point). However, the major extension that enables this encoding methodology to be used for in situ indexing is the incorporation of index compression, and thus this component is described in detail next.

4.2.1 Inverted Index Compression

In DIRAQ, an inverted index is used to map binned values to their (linearized) positions in the original dataset, referred as "record IDs," or "RIDs." Within each bin, we keep the low-order byte elements sorted by ascending RID, which enables us to apply PForDelta to the inverted index of each bin. Thus, the problem of index compression in DIRAQ can be reframed as compressing sorted lists of unique integers; the chosen method can then be applied to each bin's inverted index independently to achieve overall index compression.

Instead of using general-purpose compression routines, we use a modified version of PForDelta to leverage the specific properties of our inverted index. As stated in Section 3.2, PForDelta operates on a sorted list of integers, which matches the nature of DIRAQ's inverted indexes. Furthermore, PForDelta achieves high compression ratios for clustered elements; in DIRAQ, the inverted index RIDs typically exhibit good clustering, as these spatial identifiers are grouped by bin, and thus correspond to similar-valued points, which tend to cluster spatially.

We make modifications to PForDelta to achieve higher compression ratios for our application. The first difference is in the method for specifying the positions of exceptions in the delta list for patching. As opposed to the original approach, which uses a linked list of relative offsets embedded in the delta list, we instead leverage the fact that all RIDs are unique, implying that all deltas are strictly positive. This permits us to place 0's in the delta list to mark exception positions, as they do not normally appear, thus eliminating the need for an embedded linked list and forced exceptions. These 0's have low overhead, as they are bit-packed along with the deltas.

The second modification we implement is to deal with overhead when compressing small chunks. While our implementation maintains the fixed 128-element chunk sizes used in the original PForDelta work, the last chunk may have fewer elements. If the original input stream (inverted index bin, in our case) has far fewer elements than one chunk (128), PForDelta's chunk metadata may become dominant, reducing compression ratios or even slightly *inflating* the index. This situation occurs for datasets with high variability (when modeling turbulence for example), which leads to a large number of small bins. To prevent this issue from unduly harming overall compression, we add a dynamic capability in PForDelta to selectively disable compression of chunks that are not successfully reduced in size.

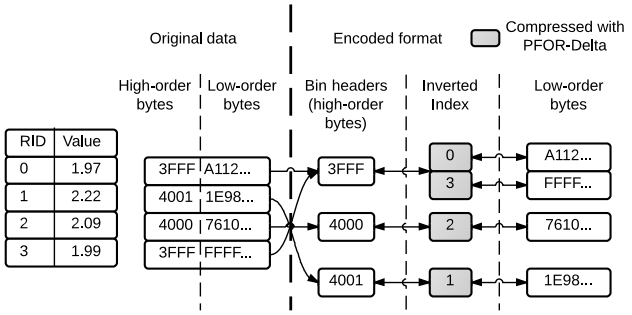


Figure 2: An overview of how raw floating-point data is encoded by DIRAQ for compact storage and query-optimized access. CII stands for compressed inverted index.

4.2.2 Effectiveness of Index Compression

As a preliminary microbenchmark, we evaluate the overall throughput and compression ratios of DIRAQ's encoding on 6 single-precision and 3 double-precision datasets from the FLASH [11] simulation. The test platform is the Intrepid BlueGene/P supercomputer. For each dataset, we take 256 compute cores worth of data, and encode each independently (simulating the core-local indexing that will be used in the next section), reporting the mean statistics in Table 1.

Compression Ratio: In our microbenchmarks, we observed index compression ratios from 3x to 20x across different variables and different refinement levels in the FLASH dataset. We observe that the compression ratio directly correlates with the number of bins in the dataset encoding, as shown in Table 1. With fewer bins, each bin's inverted index contains a larger portion of the RIDs for the dataset. This yields a denser increasing sequence of RIDs per bin, and thus smaller delta values, which ultimately results in a high compression ratio due to PForDelta bitpacking with 1 or 2 bits per element. Even with less compressible datasets like *vely*, *velz*, we observe ≈ 2.5x compression of the index.

Overall, this level of compression results in substantial I/O reduction, and also improves aggregation performance (as discussed in Section 4.3). For comparison, when encoding these same datasets using the original ALACRITY method, total storage footprints (data + index) are all around 150% for the single-precision datasets, and 125% for those with double-precision. Over the datasets we evaluate, we observe effective I/O throughput to be increased by a factor of $1.5-2.5$x based on this data encoding method

Throughput: The indexing throughput is also dependent on the number of bin values, but to a lesser degree as compared to the compression ratio (max-over-min variation of about 1.4x vs. about 10x). This is primarily due to the use of fixed-size chunks during index compression, each of which is processed independently. However, since the compression is considerably faster than I/O access and has the net result of reducing the data footprint, the variation in indexing throughputs of DIRAQ do not contribute to a noticeable variation in end-to-end times , unlike with other compute-intensive indexing techniques [8, 9].

4.3 Scalable, Parallel, In Situ Index Aggregation

In the simplest case, we can parallelize our indexing method by applying it to all compute cores simultaneously, generating *core-local* indexes, similar to FastQuery. However, this method of parallelization produces *fragmented* indexes, which are poor for query performance. Queries must necessarily process each core-local in-

Table 1: Effect of dataset entropy (number of bins) on index compression ratio and total size on 256 processes each indexing 1 MB of data on BG/P.

Dataset	Average Bins	Index Compression Ratio	Total Size (% of Raw)	Encoding Throughput (MB/s)
flam	8	19.8	55	18.7
pres	1	20.6	54	18.7
temp	7	17.1	55	18.7
velx	339	4.3	78	17.6
vely	1927	2.4	90	14.0
velz	1852	2.5	89	14.1
accx*	172	3.3	88	15.1
accy*	166	3.3	88	15.0
accz*	176	3.3	88	15.0

* double-precision datasets.
+ total data written out as a % of original raw data.

dex in turn, incurring numerous disk seek costs and other overheads. Thus, it is desirable to build an index over aggregated data. However, performing global data aggregation is not scalable due to expensive global communication, so we instead consider *group-level* index aggregation, where compute nodes are partitioned into fixed-size groups, balancing the degree of index fragmentation (and thus query performance) with I/O blocking time (and thus simulation performance).

A particular challenge in building an index over a group of processes is the distribution of computation over the group. Forwarding per-core data to a small number of "aggregator cores" to be indexed poorly utilizes compute resources. Instead, we make the observation that core-local DIRAQ indexes can be merged efficiently. However, merging the indexes solely using aggregator cores again underutilizes the remaining compute cores, which sit idle during that time.

Given these challenges, we opt for a third option, shown in Figure 3, that eliminates compute bottlenecks at the aggregator cores. After each core in the group locally encodes its data, a group "leader" core collects the local index metadata and computes the group index metadata. The group index metadata is then used by each compute core to implicitly merge the indexes *in network* by using Remote Memory Access (RMA), thereby materializing the group index fully formed across any number of chosen aggregator cores. Using this method, aggregators and all the other group cores perform the same degree of computation, as well as avoid per-core file writing, at the small cost of a metadata swap (which would be necessary, regardless).

Having given an overview of the aggregation method in DIRAQ, we now examine each step in more detail. Note that, while we focus on *index aggregation* this section, because DIRAQ's encoding tightly integrates the index and low-order bytes, an equivalent aggregation and merging process is also applied to the low-order bytes. Thus, unless specifically noted, every step in the follow procedure is applied to both the index and low-order bytes simultaneously, but we discuss in terms of the index for brevity.

4.3.1 Building the group index

The DIRAQ encoding technique is especially suited for *in situ* index aggregation, as it enables efficient determination of the group index layout before any actual data or index is transferred. Recall that our encoding bins data according to values that share the same most significant bits (called the bin's "header value"). Because this number of significant bits is fixed across all cores, if bins with the same header value appear on multiple cores, all these bins will have

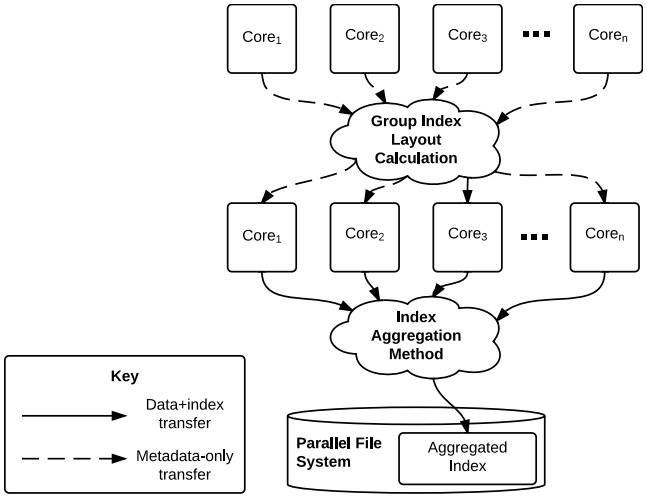


Figure 3: A logical view of group index aggregation and writing.

equivalent value boundaries, and will furthermore be part of the same bin in the group index, greatly simplifying the index merge operation. Furthermore, because the data within a bin is ordered by RID, and we assign a contiguous range of RIDs to each core, merging several local bins into a group bin is a simple matter of concatenation. Note that "bin" here refers to both the bin's low-order bytes and its associated inverted index.

The process for building the group layout on the leader core is shown in Figure 4. First, the layouts for all core-local indexes within the group are collected to a "group leader" core. Next, the leader takes the union of all bin header values for the group, and then determines the offset of each local index bin within the corresponding group index bin, with bins from cores with lower RID ranges being filled first. Finally, the newly-constructed group layout is disseminated back to the compute cores, along with the set of bin offsets specific to each core (referred to as "bin placements"). Note that indexes could also be aggregated across groups, by exchanging bin metadata between group leaders. We expect to study this possibility in future work.

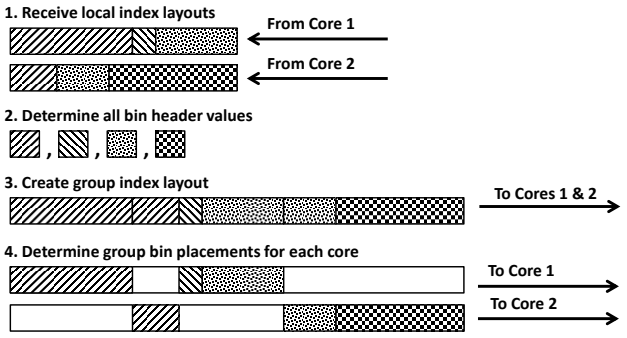


Figure 4: Steps for building a group index layout on the leader core (shown for a group size of 2, generalizable for more cores).

The end result is that each core obtains a complete view of the group index to build, as well as precise locations for its local index bins within that group index. In other words, the cores now collectively have a logical view of how the aggregation should proceed. It is important to note that this cooperative layout construction pro-

cess constitutes only a small part of the overall end-to-end indexing time, and so we focus the bulk of our optimization efforts on the more time-intensive aggregation and write phases next.

4.3.2 *Index aggregation via* in network *memory reorganization*

After all cores in the group receive the group index layout and core-specific bin placements, the cores proceed to transfer their local bins to the appropriate locations within the group index. This mapping is straightforward at this point; the bin placements include an offset and length within the group index for each local bin. However, simply writing at these offsets to a shared file will result in $N \cdot B$ qualitatively random write operations hitting the parallel file system servers at once (where N is cores per group, and B is average local index bins per core). With $N \approx 64$ and $B \approx 4,000+$, typical in our experiments with FLASH, the resultant $250,000+$ I/O requests *per group* would be prohibitively expensive, and an alternative approach must be sought. Furthermore, the straightforward usage of MPI-IO would necessarily require numerous, small writes and high communication overhead, corresponding to the highly interleaved mapping between core-local and group indexes.

Our solution leverages a cooperative *in network memory reorganization* technique to utilize the RMA capabilities of modern network infrastructure to perform index merging during network transfer. Using this method, the system can materialize the group index fully-formed across the memory of a subset of the compute cores in the group, called the "aggregator cores." The process proceeds as follows:

1. A set of aggregator cores with sufficient memory to collectively accommodate the group index is selected. This is done by piggybacking available memory statistics from each core on the existing local layout messages sent to the group leader, and having the leader disseminate the selection along with the group index layout and bin placements. We examine the importance of selecting topologically-separated and memory-balanced aggregators in Section 4.4; for now, the specific selection criteria are not pertinent.

2. After this selection, the memory commitment needed for the group index is balanced as evenly as possible across the aggregators, which then expose the required memory to all the compute cores for RMA operations.

3. Finally, all compute cores treat the exposed memory on the aggregators as a single, contiguous memory buffer, and use their bin placements and knowledge of the group index layout to inject their local index bins onto the proper aggregators at the right offsets. These data transfers utilize MPI one-sided RMA operations, using relaxed consistency and reduced-locking hints to achieve increased performance.

4. After RMA synchronization is completed, the group index is fully formed on the aggregators, which then simply write out the entire contents of their exposed memory window in sequential chunks on disk, completing the process.

By circumventing the need for an in-memory index merge, the computational load on the aggregators is reduced (which is important because there are few of them relative to the number of cores in the group). Furthermore, this approach also eliminates the need for temporary swap buffers required during an in-memory merge, and so is more robust in the face of limited spare memory available under many real-world simulations.

4.4 Optimizing index aggregation using memory- and topology-awareness

Algorithm 1: Topology-aware aggregator assignment on group leader processes

Input : n: Number of processes in the group.
Input : $M[1,...,n]$: Free memory size of each process in the group.
Input : d: Average amount of data per-core.
Input : b: Average number of bins per-core.
Output: a: Estimated ideal number of aggregators.
Output: t: Estimated neural-net time.
Output: $R[1,...,a]$: Ranks of aggregators.
Output: O: Start and end offsets associated with each aggregator.

```
minAggs = getMinAggregators(M,n,d)
maxAggs = getMaxAggregators(M,n,d)
t = ∞
// Estimate the optimal number of
// aggregators using trained model
for numAggs = {minAggs, ..., maxAggs} do
    estimatedTime = NNEstimate(numAggs,d,b)
    if t > estimatedTime then
        t = estimatedTime
        a = numAggrs
    end
end
// Try placing aggregators using
// topology-aware settings
for aggSet ∈ {topologyAwareAggregatorSetList} do
    if fit (a,aggSet,M) == TRUE then
        R = assign (a,aggSet)
        O = generateOffsets (a,R,d)
        return {a,t,R,O}
    end
end
// Generate a valid random placement
R = generateRandomAggregators (a, M)
O = generateOffsets (a,R,d)
return {a,t,R,O}
```

One of the most common techniques employed for I/O is two-phase MPI collective I/O, which performs a data aggregation phase prior to writing. However, the number and placement of aggregators within MPI, which can be tuned using "hints," does not take topology considerations into account, leading to network hotspots and other performance degradations [25]. Hence, recent works have explored topology-aware mapping for BlueGene/P [25] and tuning the aggregation ratio [6, 25].

However, these static techniques are not directly applicable within DIRAQ for the following reasons. First, the use of index compression results in varying data sizes written out by process groups. Second, with DIRAQ, the aggregation phase not only includes a simple data transfer, but also an *in network* index merging strategy. Thus, the *in network* aggregation performance is based on a number of changing parameters, such as differing bin layouts across write phases, and so requires a more dynamic approach.

To account for these *time-variant* characteristics in DIRAQ, as well as the highly interleaved (core-local) I/O access patterns DIRAQ produces, data aggregation requires a strategy in which the

number of aggregators/writers evolve according to simulation data characteristics. In DIRAQ, the group leaders are a natural fit for driving this process, as they have a complete view of the aggregated index and are responsible for distributing aggregator information to the other cores in the group.

We build an optimization framework that can dynamically select the number of aggregators, given the group index and low-order byte layout, while leveraging the work done in GLEAN [25] to contol aggregator placement. Since this layout has numerous, interacting characteristics, our initial study found that rigid, linear models insufficiently captured the relationship between group layout and I/O. Hence, we train a neural network, bootstrapping offline to model and optimize DIRAQ aggregation parameters. Our choice of using neural-network-based learning is based on the fact that it is suitable to learn representations of the input parameters that capture the characteristics of the input distribution and subsequently carry out function approximation, especially when modeling non-linear functions [14, 30]. The topology- and memory-aware strategy is described in the Algorithm 1. The details of the performance modeling are explained in the following section.

Both the neural network and the list of topology-aware aggregator sets are determined simultaneously using a set of offline microbenchmarks. The usage of the neural network warrants further discussion (See Section 4.4.1). We estimate the execution time of the aggregation process as a function of the number of aggregators. Then, after pruning the possible number of aggregators based on memory constraints, we run the neural network over each possible number of aggregators, and choose the number that is predicted to have the minimal completion time. By estimating the completion time, the leader can perform error propagation based on the actual time taken. Furthermore, we observed negligible computational overhead (on the order of milliseconds), even when running multiple iterations of the neural network estimation.

4.4.1 Performance Model

The goal of the neural network-based performance model is to accurately predict the performance of three components: the index and low-order byte aggregation times as well as the I/O times for DIRAQ, both with and without inverted index compression. Given these predictions, we can apply the model on each group to determine at run-time a well-performing set of aggregators. Furthermore, we focus on the BlueGene/P architecture, though our methods can be applied to other cluster architectures. Table 2 gives the necessary parameters.

Given the BlueGene/P architecture, the DIRAQ indexing framework consists of three components, namely the compute cores, the aggregators, and the BlueGene I/O pipeline, which consists of the I/O forwarding cores and a Myrinet switch which provides connectivity to file server cores of a cluster-wide file system [25]. We assume an aggregation group of size ρ. An aggregation group is defined as a logical group of ρ compute cores (with one of them also as the group leader) and corresponding a aggregator cores. Each aggregation group forms an MPI communicator in our implementation. We model the aggregation and I/O process taking place in a single aggregation group.

In order to build an accurate model, we must take into account the RMA contention at the aggregators. To do so, we ran a set of microbenchmarks to measure the aggregation and I/O times, for varying parameters ρ, B, d, and a (refer to Table 2). Linear regression is not suitable for modeling the non-linear relationship $t_{agg\_io} = f(\rho, d, b, a)$. Therefore, we used a 3-layered neural network with ρ, B, d, and a as inputs, 40 neurons in the hidden layer and the t_{agg_io} as the output. This is further used to determine the optimal number of aggregators in Algorithm 1.

Table 2: Parameters for the performance model.

Fixed Input Parameters	
ρ	Number of compute cores per MPI communicator
e	Unit element size (e.g. 32 bits for single-precision)
s	Number of significant bits used in DIRAQ encoding
γ	Data reduction due to indexing $(= s/(8 \cdot e))$
Run-time Input Parameters	
a^*	Number of aggregators per MPI communicator
d	Average size of core-local data
B	Average number of core-local bins
l	Average local inverted index size
σ	Average inverted index compression ratio
Bootstrapped Input Parameters	
μ_w	Disk write throughput per aggregator (determined using microbenchmarks)
Output Parameters	
$t_{agg\_io\_index}$	Index aggregation and I/O time (sec)
$t_{agg\_io\_LOB}$	Low-order byte aggregation and I/O time (sec)

* Iterated over by leader node for optimization.

We collected measurements for the aggregation and I/O times for various combinations of ρ, d, B, and a. We then trained the neural network using the FANN neural network library [20] with a total of 630 such samples. We used a $85-15\%$ division into the training and testing subsets. With this configuration, we obtained a mean squared error of $1.15e^{-4}$ and an R^2 statistic of 0.9812 on the test data using the iRPROP training algorithm [15] and the symmetric sigmoid (tanh) activation function. On the contrary, a simple linear regression model resulted in a R^2 statistic of 0.553.

4.4.1.1 Case 1 : Without compression.

In this case, the local layout of the index corresponding to the target variable is first built on every compute core. Then, the local index generation takes place which is followed by the process of building the global index layout and transferring the index metadata. Note that the former is purely a computation step, while the latter primarily involves communication between the compute cores. The index and low-order bytes are then aggregated followed by the initiation of the index and low-order bytes I/O operations. During I/O, the compute cores produce the data for the next time step and initiate the corresponding local layout generation.

The index and low-order byte aggregation steps involve the compute cores writing at known offsets in the aggregator core's memory via one-sided RMA calls. The indexing scheme reduces the data by about $\gamma = \frac{s}{8 \cdot e}$ of its original size. Thus,

$$t_{agg\_io\_index} = f(\rho, l, B, a) \tag{1}$$

$$t_{agg\_io\_LOB} = f(\rho, (1-\gamma)d, B, a) \tag{2}$$

4.4.1.2 Case 2 : With compression.

This scenario includes an additional index compression stage. In this case, we add the index compression phase after the local index generation at every compute core. The low-order byte aggregation and I/O remains the same. However, the index aggregation and I/O take place over the compressed index. Thus,

$$t_{agg\_io\_index} = f\left(\rho, \frac{l}{\sigma}, B, a\right) \tag{3}$$

Using the above equations for index and low-order byte aggre-

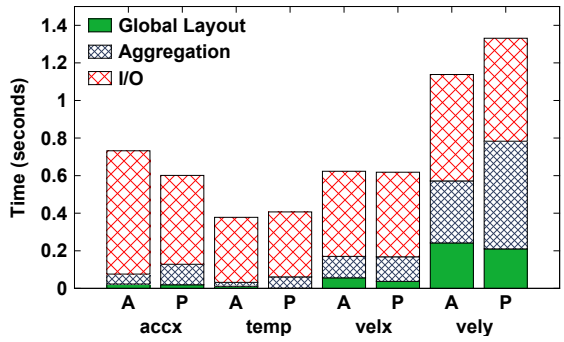


Figure 5: Accuracy of the performance model prediction on Group Layout, Aggregation, and I/O timings with compression for a fixed compute to aggregator ratio of 8:1. A, P stand for actual and predicted timings, respectively.

gation, and the values of ρ, d, and B as input, we determine the optimal number of aggregators for both the index and low-order bytes as the one which results in the minimum t_{agg_io} times :

$$\hat{a}_{LOB} = \underset{a}{\operatorname{argmin}}\, t_{agg\_io\_LOB}(a) \quad (4)$$

$$\hat{a}_{index} = \underset{a}{\operatorname{argmin}}\, t_{agg\_io\_index}(a) \quad (5)$$

4.4.2 *Accuracy of Performance Model*

In this section, we analyse the accuracy of the trained neural network in two scenarios. One, where the model predicts the aggregation and I/O times when the aggregation ratio is fixed, and the other when model picks the aggregation ratio. This is done at the group leader nodes, which has the necessary parameters like the number of bins, compression ratios and access to the trained neural network to make smart decisions. Figure 5 shows the predicted accuracy on component timings when the aggregation ratio is fixed for 4 variables from FLASH simulation, with 1 MB of data indexed and compressed at each core.

5. EXPERIMENTAL EVALUATION

With regards to our parallel, *in situ* indexing methodology, there are three primary performance metrics we evaluate. The first, query performance, serves two purposes: it gives a comparison between DIRAQ and FastQuery indexes, and it underscores the need for defragmented indexes to accelerate query-driven analytics. The second, index performance, evaluates our group-wise index aggregation methodology, looking at index size, generation speed, and scalability. The final metric, end-to-end time (which is broken down stage-by-stage for better insight), evaluates the effectiveness of our memory- and topology-aware optimizations, as based on dynamic neural network application, as well as our methodology as a whole.

5.1 Experimental Setup

Our experiments were conducted on the "Intrepid" system at the Argonne Leadership Computing Facility (ALCF). It is a 40K node quad-core BlueGene/P cluster consisting of 80 TB memory delivering a peak performance of 557 TFlops. Unless otherwise specified, evaluations were done on the General Purpose Filesystem (GPFS) [22] with the default striping parameters. Each experiment below was repeated 5 times and the reported numbers correspond to the median times for each of the results below. All experiments were performed in the "VN" mode in which one MPI process is launched on every core on the chosen set of nodes.

Before we describe the results, we would like to point out that the GPFS storage utilization was over 96%, at the time of our evalution. As a result, the practically observed I/O throughput (Figure 9) is only a fraction of the theoretical maximum throughput.

We execute DIRAQ with the Sedov white-dwarf FLASH simulation [11] and analyze its *in situ* performance on 9 different datasets (6 single-precision and 3 double precision). However, for the sake of brevity, we present the results on 4 datasets. Based on the dataset entropy (Table 1), the chosen single-precision variables $temp, velx, vely$ can be considered representative samples of all the datasets used in this paper. Since, the number of unique values (bins) in the index directly relates to the compression ratio we take datasets that have low $(100-200)$, medium $(1000-5000)$ and high (>10000) number of *global* bins, which correspond to *temp*, *velx*, and *vely*, respectively. Additionally, we choose one double precision dataset *accx* as well.

Additionally, to analyze the performance of index compression and querying across simulations, we show results on 4 datasets from S3D combustion simulation as well. We do not include scaling results from S3D in this paper, since S3D does not run on Intrepid, and our optimizations related to topology-aware aggregations exploit BlueGene/P-specific hardware. We analyze the serial query performance over S3D datasets on the Lens Analysis Cluster at Oak Ridge National Laboratory. Each node of Lens consists of four 2.23 GHz AMD Opteron quad-core processors and is equipped with 64 GB of RAM. We run the queries over the Lustre File System with the default striping parameters.

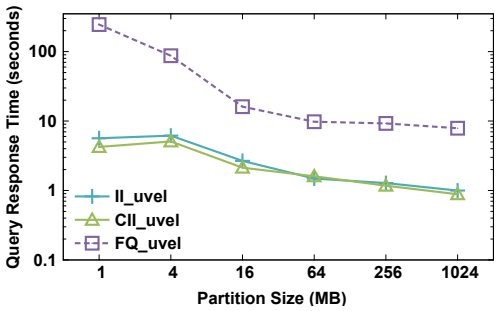
5.2 Query Performance

We first demonstrate the importance of aggregation for post-simulation query performance. Figure 6 depicts the serial query performance of DIRAQ and FastQuery with full-precision value retrieval (meaning exact values are returned). We use single-sided range queries of the form $var < b$, where b is selected to induce a particular query selectivity. We perform this experiment using multiple partition sizes (that is, amounts of data indexed as a unit), ranging from 1 MB to 1024 MB on a 2 GB dataset, while fixing query selectivity (i.e., the fraction of data satisfying the query) at 0.1%. Note that FastQuery is constrainted to a partition size equal to the amount of data available per variable per core, as the algorithm produces a local index for on core (though all such indexes are stored contiguously on disk). In contrast, DIRAQ can produce larger partition sizes by increasing its aggregation ratio, even when per-core data is low.

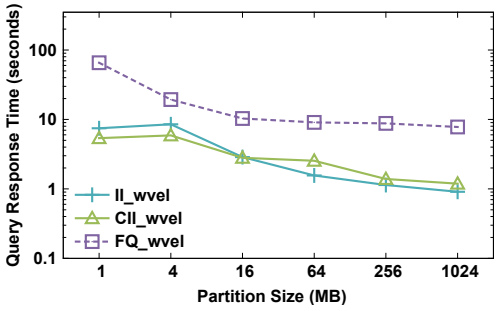
The figures show two trends. First, the DIRAQ indexing scheme, being lightweight in both computation and storage, outperforms FastQuery's method given a particular partition size. Second, for both methods, query performance is directly proportional to index partition size, presumably because the number of seeks induced is inversely proportional to the partition size, while the sizes of any contiguous reads are reduced. This performance characteristic of indexing in general is precisely our motivation for opting to perform index aggregation, rather than core-local indexing.

5.3 Indexing Performance

The performance of an indexing methodology can be considered in numerous contexts: a *storage* context, a *computational* context, and a *scalability* context. The following sections explore these contexts, providing a finer grain performance measure and analysis of individual tasks in DIRAQ.

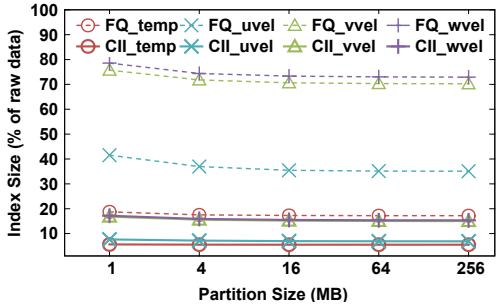


(a) uvel from S3D simulation.

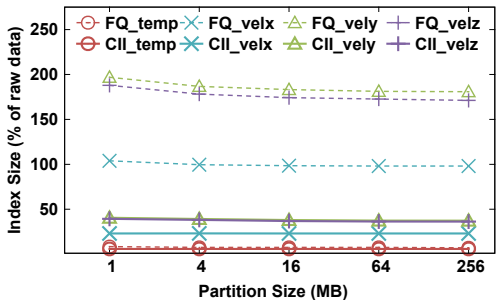


(b) wvel from S3D simulation.

Figure 6: Comparison of response times of DIRAQ compressed (CII) and uncompressed (II) indexes with FastQuery, over various aggregation sizes, on queries of fixed-selectivities.



(a) Double-precision datasets from S3D simulation.



(b) Single-precision datasets from FLASH simulation.

Figure 7: Resulting index sizes (as a % of raw data) on varying amount of data aggregated per-core with FastQuery (FQ) and DIRAQ (CII) indexing techniques.

5.3.1 Index Size

Figure 7 shows the index size generated for both single-precision FLASH variables as well as double-precision S3D variables over 256 MB of raw data. As with the query processing results, we experiment with multiple partition sizes to ascertain the effect of index fragmentation on index size. FastQuery's index size, as a proportion of the raw data, slightly increases for smaller partition sizes, and the difference in resulting index sizes using DIRAQ is near imperceptible. While the index metadata size increases in proportion to decreasing partition sizes, the overall effect is negligible. This means that, when considering the size of aggregation groups in DIRAQ, overall index size can be disregarded as an optimization parameter.

5.3.2 Indexing Speed

The core-local indexing throughput was shown in Table 1. In this section, we instead look at end-to-end indexing performance through scalability metrics, as well as study the stage-by-stage timing breakdown of DIRAQ with and without compression.

5.3.2.1 Scalability.

It is crucial that DIRAQ exhibit favorable scalability properties, as it aims to index large-scale data. To this end, we have performed both strong and weak scalability benchmarks, simulating the indexing of one variable during one timestep. In both benchmarks, we keep the aggregator group size constant. In the strong scalability experiment, the overall data size is kept constant at 2 GB, whereas in the weak scalability experiment, the data size per core is kept constant at 1 MB per core.

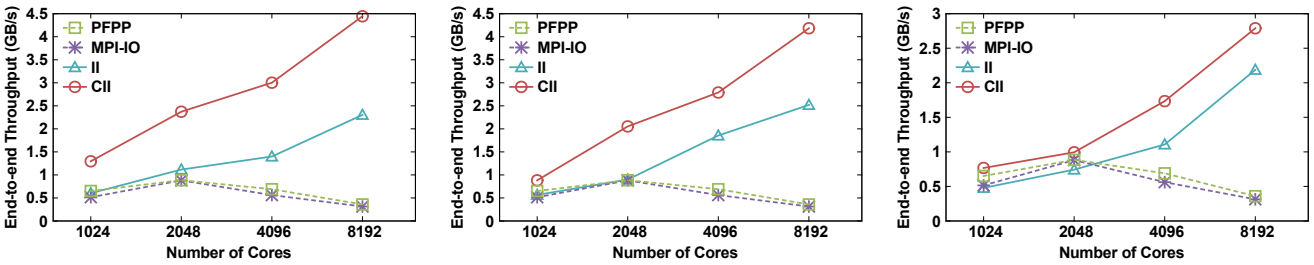
The weak scaling results vary as shown in the Figure 9. We observe that the throughput for DIRAQ with compression increases almost linearly with the number of cores in the simulation for all the variables, due to increasing utilization of I/O resources. We compare DIRAQ with the baseline case of writing the raw simulation output using MPI-I/O and POSIX file-per-process approaches, both evaluated using the IOR benchmark. With DIRAQ we observe throughput gains of approximately 5x and 6x on the *velx* and *temp* variables respectively, and close to a 2x improvement for the least compressible *vely* variable. On the other hand, DIRAQ without compression yields a similar performance as these baseline approaches. Note that, without index compression, DIRAQ writes out 1.5x more than the original raw output, but performs comparably to the other write schemes, likely due to our topology-aware aggregation.

The strong scaling results vary as shown in the Figure 8. Under strong scaling, the amount of data indexed per core reduces at larger scales. While the index size generated by DIRAQ is proportional to the input data size (Figure 7), and so remains constant, the data movement stage now generates smaller message sizes per bin. This slightly increases the network contention at larger scales, but since the available I/O throughput increases as we request more cores, and this is the dominant component, we see improvements in the end-to-end times.

5.3.2.2 Performance Breakdown.

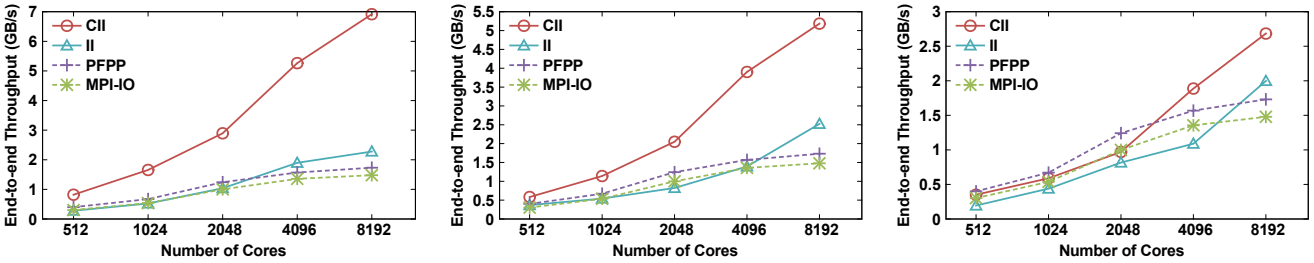
We breakdown the performance of DIRAQ by analyzing the execution time of each stage in the pipeline for the 4 datasets, namely *temp*, *velx*, *vely*, and *accx*, with and without compression. We componentize the stages of DIRAQ into *Encoding, Group Layout, Aggregation,* and *I/O*.

The *Encoding* stage gives the total time involved in encoding the core-local data, including the optional PForDelta compression of the inverted index. The *Group Layout* involves sending the core-local layout to the group leader, which constructs the global layout and assigns aggregators in charge of performing I/O. The *Aggregation* component includes the time taken to send both the index and



(a) temp (b) velx (c) vely

Figure 8: Strong scaling on the effective end-to-end throughput (original data size / end-to-end encoding time) on 3 FLASH simulation datasets indexing a total of 2 GB, on Intrepid. PFPP (POSIX file-per-process) and MPI-IO (two-phase collective) perform raw data writes with no overhead of indexing.



(a) temp (b) velx (c) vely

Figure 9: Weak scaling on the effective throughput (original data size / end-to-end indexing time) for 3 FLASH simulation datasets indexing 1 MB on each core on Intrepid. PFPP, MPI-IO performs raw data writes with no overhead of indexing.

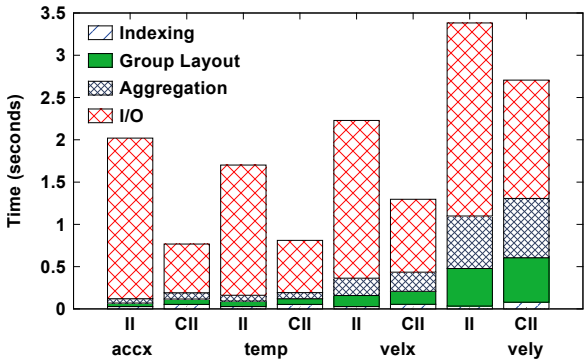


Figure 10: Cumulative time spent on each stage of DIRAQ for 4 datasets from FLASH simulation on Intrepid, with each process indexing 1 MB of data using an uncompressed index (II), and a compressed inverted index (CII).

low-order bytes to its corresponding aggregator, and the *I/O* component includes the time taken to perform the I/O operation on the aggregators, and the waiting-time on the non-aggregator processes.

Figure 10 shows the total time involved in creating aggregated indexes for the chosen datasets, with and without compression. Each process operates on 1 MB of data and a group, consisting of 256 processes uses 32 aggregators (writers). In almost all the cases, compression has a positive impact by reducing the amount of data moved within the network and I/O. However, the time spent in each stage of the pipeline varies with the entropy of the candidate dataset. For example, the time spent in the aggregation and group layout phases of the *temp* variable (highly compressible) is significantly less than the same for the *vely* variable (least compressible). We further discuss the breakdown results for the *temp* and *vely* variables since they are representative of the dataset entropy spectrum. We observe an average case performance for the *velx* and *accx* variables.

- *Encoding*: The encoding time at local cores is affected by the number of unique bin values present in the data. The dataset *vely*, which has 100x more unique bins than *temp*, requires around 2x more time to complete the encoding process. Even with the application of compression, which roughly doubles the amount of encoding time, this stage consumes less than 5% of the total execution time.

- *Group Layout*: Unlike the encoding process, which is local to each core, the group layout stage requires M:1 communication within each group for merging the bin layouts from each process. However, the group-leader performs more work when the dataset has a larger number of unique bins, since calculating the offsets for each core in the group-aggregated layout requires M-way list-merge-like operations that induce loop branching and semi-random access. This explains why *vely* has a pronounced *Group Layout* stage compared to the other datasets. Activating index compression adds another 10% to the group layout time, since additional index compression metadata must be communicated and merged.

- *Aggregation*: This step essentially converts a set of distributed, non-contiguous, core-local views of the index/low-order bytes into a smaller set of aggregated, contiguous segments of the index/low-order bytes on the aggregators. To achieve this, each process would need to make B_{local} RMA calls, where B_{local} is the number of local bins at that process. For datasets with few unique bins, very few RMA calls are required to

cluster the index and low-order bytes by bin, which occurs quickly on the 3D torus network.

- *I/O and end-to-end times*: The number of bins, and the clustering factor of values determine the final compression ratio. For example, the variable *pres*, and to a lesser extent $flam$, possesses little variation in the indexed values on a single core. These datasets have indexes that are compressed by as much 20x, and thus have $\approx 22\%$ and 35% less data to write to disk when compared with *velx* and *vely*, respectively. When compared with using an uncompressed inverted index, the amount of data written is reduced by as much as 2.7x, leading to end-to-end completion times that are up to 2.2x faster.

5.4 Resource Awareness

Figure 11 shows the performance of the aggregator selection mechanism on the dataset *vely* at 4096 cores. The resource-aware aggregation algorithm chooses a well-performing number of aggregators. On the hard-to-compress dataset *vely*, for example, the neural network predicts within a group, 8 aggregators for the index aggregation, and 16 for the low-order bytes aggregation, as opposed to a single, fixed aggregation ratio. Compared with static aggregation strategies, this results in 10-25% improvement in average throughput when writing to disk. The additional benefit from this scheme is that the variation in end-to-end times across groups is reduced as well, thereby reducing the idle time on some groups waiting to synchronize after I/O.

The neural net is inclined to pick a higher number of aggregators when indexes are less compressible. For other variables such as *temp*, which are highly compressible, an aggregation ratio of 64:1 (4 aggregators) enables aggregator writers to avoid making very small I/O requests to the filesystem. Because of topology-aware aggregator placement along with an aggressive compression scheme, aggregation times generally do not present a bottleneck when compared with I/O times.

6. CONCLUSION

This paper describes DIRAQ, an effective parallel, *in situ* method for compressing and indexing scientific data for analysis purposes during simulation runtime. DIRAQ produces a compressed index that is significantly smaller than state-of-the-art indexes. The combination of index compression and data reduction results in an encoding that, in many cases, is actually smaller than the original, unindexed data. By using a high-throughput local indexing and compression scheme followed by an effective *in network* index merging and aggregation strategy, DIRAQ is able to generate group-level indexes with minimal overhead to the simulation. For our application, a custom aggregation scheme, along with an adaptive approach to choosing aggregator ratios, results in better performance compared to MPI collective I/O routines. Overall, DIRAQ presents an analysis-efficient data encoding that is smaller than the raw data in majority of the cases, offers faster query processing time than current indexing schemes, and can be generated *in situ* with little-to-no overhead (and possibly an I/O performance improvement) for simulation applications.

7. ACKNOWLEDGMENT

We would like to thank the FLASH Center for Computational Science at the University of Chicago for providing access to the FLASH simulation code and both the FLASH and S3D teams for providing access to the related datasets. We would like to acknowledge the use of resources at the Leadership Computing Facilities at Argonne National Laboratory and Oak Ridge National Laboratory, ALCF and OLCF respectively. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under Contract DE-AC05-00OR22725. This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs). The work of MEP and VV was supported by the DOE Contract DE-AC02-06CH11357.

8. REFERENCES

[1] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky. Just in time: adding value to the IO pipelines of high performance applications with JITStaging. In *Proc. Symp. High Performance Distributed Computing (HPDC)*, 2011.

[2] H. Abbasi, J. Lofstead, F. Zheng, K. Schwan, M. Wolf, and S. Klasky. Extending I/O through high performance data services. In *Proc. Conf. Cluster Computing (CLUSTER)*, Sep 2009.

[3] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: scalable data staging services for petascale applications. In *Proc. Symp. High Performance Distributed Computing (HPDC)*, 2009.

[4] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proc. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[5] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *Proc. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[6] M. Chaarawi and E. Gabriel. Automatically selecting the number of aggregators for collective I/O operations. In *Proc. Conf. Cluster Computing (CLUSTER)*, 2011.

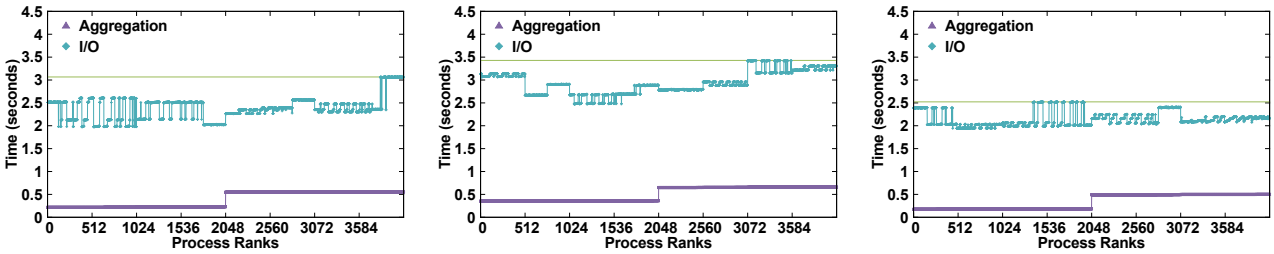
[7] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Journal of Computational Science & Discovery (CSD)*, 2(1), 2009.

[8] J. Chou, K. Wu, and Prabhat. FastQuery: a parallel indexing system for scientific data. In *Proc. Conf. Cluster Computing (CLUSTER)*, 2011.

[9] J. Chou, K. Wu, O. Rübel, M. Howison, J. Qiang, Prabhat, B. Austin, E. W. Bethel, R. D. Ryne, and A. Shoshani. Parallel index and query for large scale data analysis. In *Proc. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011.

[10] J. M. del Rosario, R. Bordawekar, and A. Choudhary. Improved parallel I/O via a two-phase run-time access strategy. *ACM SIGARCH Computer Architecture News*, 21(5):31–38, Dec 1993.

[11] 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. *Astrophysical Journal Supplement Series*, 131:273–334, Nov 2000.



(a) 32:1 (fixed) (b) 64:1 (fixed) (c) NN-driven (dynamic)

Figure 11: Comparison of different aggregation strategies (number of compute to aggregator process) on I/O and aggregation timings, with process ranks sorted by aggregation times. 1 MB of the dataset *vely* is indexed by each of the 4096 processes.

[12] J. Fu, R. Latham, M. Min, and C. D. Carothers. I/O threads to reduce checkpoint blocking for an electromagnetics solver on Blue Gene/P and Cray XK6. In *Proc. Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2012.

[13] J. Fu, M. Min, R. Latham, and C. D. Carothers. Parallel I/O performance for application-level checkpointing on the Blue Gene/P system. In *Proc. Conf. Cluster Computing (CLUSTER)*, 2011.

[14] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Proc. Conf. Neural Networks*, Jul 1989.

[15] C. Igel and M. Hüsken. Empirical evaluation of the improved Rprop learning algorithm. *Journal of Neurocomputing*, 50:2003, 2003.

[16] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, N. Shah, E. R. Schendel, S. Ethier, C.-S. Chang, J. H. Chen, H. Kolla, S. Klasky, R. B. Ross, and N. F. Samatova. Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In *Proc. Conf. Database and Expert Systems Applications, Part II (DEXA)*, 2012.

[17] J. Kim, H. Abbasi, L. Chacon, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu. Parallel in situ indexing for data-intensive computing. In *Proc. Symp. Large Data Analysis and Visualization (LDAV)*, Oct 2011.

[18] S. Kumar, V. Vishwanath, P. Carns, J. A. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M. E. Papka, J. Chen, and V. Pascucci. Efficient data restructuring and aggregation for I/O acceleration in PIDX. In *Proc. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[19] K. L. Ma. In situ visualization at extreme scale: challenges and opportunities. *Journal of Computer Graphics and Application (CG&A)*, pages 14–19, 2009.

[20] S. Nissen. Implementation of a fast artificial neural network library (fann). Technical report, Department of Computer Science University of Copenhagen (DIKU), Oct 2003. http://fann.sf.net.

[21] O. Rübel, Prabhat, K. Wu, H. Childs, J. Meredith, C. G. R. Geddes, E. Cormier-Michel, S. Ahern, G. H. Weber, P. Messmer, H. Hagen, B. Hamann, and E. W. Bethel. High performance multivariate visual data exploration for extremely large data. In *Proc. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[22] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proc. Conf. File and Storage Technologies (FAST)*, Jan 2002.

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

[24] T. Tu, H. Yu, J. Bielak, O. Ghattas, J. C. Lopez, K.-L. Ma, D. R. O'Hallaron, L. Ramirez-Guzman, N. Stone, R. Taborda-Rios, and J. Urbanic. Remote runtime steering of integrated terascale simulation and visualization. In *Proc. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, 2006.

[25] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Proc. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2011.

[26] K. Wu. FastBit: an efficient indexing technology for accelerating data-intensive science. In *Journal of Physics: Conference Series (JPCS)*, volume 16, page 556, 2005.

[27] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. Conf Very Large Data Bases (VLDB)*, 2004.

[28] K. Wu, R. R. Sinha, C. Jones, S. Ethier, S. Klasky, K.-L. Ma, A. Shoshani, and M. Winslett. Finding regions of interest on toroidal meshes. *Journal Computational Science & Discovery (CSD)*, 4(1), 2011.

[29] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. Conf. World Wide Web (WWW)*, 2009.

[30] R. M. Yoo, H. Lee, K. Chow, and H.-H. S. Lee. Constructing a non-linear model with neural networks for workload characterization. In *Proc. Symp. Workload Characterization (IISWC)*, Oct 2006.

[31] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In situ visualization for large-scale combustion simulations. *Journal of Computer Graphics and Applications (CG&A)*, 30(3):45 –57, May-Jun 2010.

[32] J. Zhang, X. Long, and S. Torsten. Performance of compressed inverted list caching in search engines. In *Proc. Conf. World Wide Web (WWW)*, 2008.

[33] 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 *Proc. Symp. Parallel Distributed Processing (IPDPS)*, Apr 2010.

[34] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. Conf. Data Engineering (ICDE)*, 2006.