Optimizing FastQuery Performance on Lustre File System

Kuan-Wu Lin
National Tsing Hua Univeristy
Hsinchu, Taiwan
vidcina@lsalab.cs.nthu.edu.tw

Jerry Chou
National Tsing Hua Univeristy
Hsinchu, Taiwan
jchou@lsalab.cs.nthu.edu.tw

Surendra Byna
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
SByna@lbl.gov

Kesheng Wu
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
KWu@lbl.gov

ABSTRACT

FastQuery is a parallel indexing and querying system we developed for accelerating analysis and visualization of scientific data. We have applied it to a wide variety of HPC applications and demonstrated its capability and scalability using a petascale trillion-particle simulation in our previous work. Yet, through our experience, we found that performance of reading and writing data with FastQuery, like many other HPC applications, could be significantly affected by various tunable parameters throughout the parallel I/O stack. In this paper, we describe our success in tuning the performance of FastQuery on a Lustre parallel file system. We study and analyze the impact of parameters and tunable settings at file system, MPI-IO library, and HDF5 library levels of the I/O stack. We demonstrate that a combined optimization strategy is able to improve performance and I/O bandwidth of FastQuery significantly. In our tests with a trillion-particle dataset, the time to index the dataset reduced by more than one half.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

I/O performance, Tuning, FastQuery

1. INTRODUCTION

Many scientific applications produce and consume large amounts of data. For instance, the Intergovernmental Panel on Climate Change (IPCC) multi-model CMIP-3 archive is about 35 TB in size. The next generation CMIP-5 archive, which will be used for the AR-5 report [1] is projected to contain over 10 PB of data. Large scale experimental facilities produce equally impressive amounts of data. The LHC experiment is capable of producing 1 TB of data in a second, many gigabytes of which are recorded for future analyses. The Large Synoptic Survey Telescope (LSST) will record many terabytes of data per night. The torrents of data are expected to overwhelm our capacity to make sense of them [14].

While data sizes of scientific applications are large, in many cases, the essential information is contained in a relatively small number of data records. For example, in the IPCC data of petabytes in size (10^{15} bytes) critical information related to important events such as hurricanes might occupy no more than a few gigabytes (10^9 bytes). Capabilities for accessing only the necessary information-rich data records, instead of going through all of them, can significantly accelerate scientific discoveries. The requirement for efficiently locating interesting data records is indispensable to many data analysis procedures.

The class of data structure for quickly locating selected data records is known as indexes [22] and the well-known implementation of indexing techniques are inside the commercial database systems. Instead of asking the scientists to load into commercial database systems, we advocate an approach of using an indexing library with common scientific data format libraries. This strategy directly works with user data files and allows the scientists to stay with their existing data management and analysis software. They can gradually adapt the indexing techniques without disrupting their on-going work.

In this work, we make use the FastQuery system we built on top of the FastBit indexing package to accelerate data selection based on arbitrary range conditions defined on the available data values, e.g., "energy $> 10^5$ and temperature $> 10^6$" [29, 30]. The FastQuery system provides two key features that are lacking in FastBit. FastQuery defines a common data access layer for working with many popular scientific data formats; and FastQuery provides automatic data partition so that FastBit can work with a very large number of data records using parallel processing.

There are many different scientific data formats in use currently. For example, there is a large community using HDF5 [26]; the IPCC data are in NetCDF format [27] or pNetCDF

[18]. Recently, many large-scale simulations have adapted ADIOS BP format [20]. One common feature of these disparate file formats is that they are all based on the array data model. Therefore, FastQuery uses an array model as the abstract interface to work with these different formats. Currently, our implementation of the interface supports a wide range of scientific data formats including HDF5, NetCDF, pNetCDF, and ADIOS-BP. We have demonstrated this flexibility in a series of published work [8, 9, 10, 17, 4].

In our previous implementation of FastQuery [10, 8], reading and writing of data from file system used default system settings. It is possible to obtain better performance by using parallel I/O bandwidth more efficiently. FastQuery performs significant amount of I/O in reading data to compute indexes and writing the indexes to disk. Obtaining peak I/O performance in these two important file system activities is essential to improve FastQuery performance. Figure 1 shows a contemporary parallel I/O software stack with HDF5 [26] as high-level I/O library, MPI-IO as middleware layer, and Lustre as the parallel file system. Each layer offers tunable parameters for improving performance, and hopes to provide reasonable default settings for the parameters. In this paper, we present a systematic approach of choosing parameters that can achieve significant portion of the peak I/O bandwidth.

The key contributions of this work are:

- redesigning the implementation of FastQuery to use multicore processors efficiently;
- performing a systematic study of parallel I/O performance at each level of the parallel I/O stack and identified local optimal parameters to achieve significant performance improvement;
- providing a set of practical lessons optimizing the performance of I/O operations.

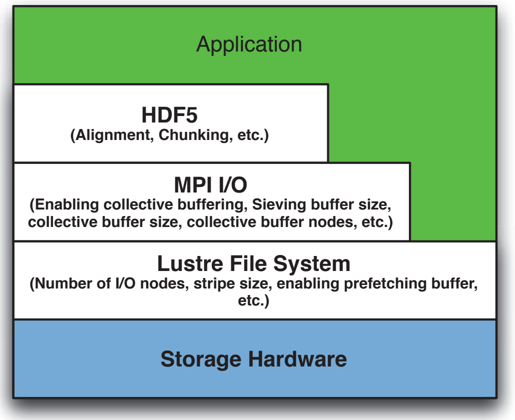

Application
HDF5
(Alignment, Chunking, etc.)
MPI I/O
(Enabling collective buffering, Sieving buffer size, collective buffer size, collective buffer nodes, etc.)
Lustre File System
(Number of I/O nodes, stripe size, enabling prefetching buffer, etc.)
Storage Hardware

Figure 1: Parallel I/O Stack and various tunable parameters

In the reminder of this paper, we present a brief introduction to scientific data, indexing, and bitmap indexing in Section 2. In this section, we also discuss an application use case of FastQuery. We explain a large trillion particle simulation dataset to motivate the need for indexing and querying with FastQuery. In Section 3, we explain the design of FastQuery. Sections 4 and 5 detail the experimental set up we used to test performance of FastQuery and the analysis and selection of optimization parameters of the parallel I/O stack, respectively. We conclude the paper in Section 6 with a discussion of future work.

2. BACKGROUND AND MOTIVATION

2.1 Searching in Scientific Data

To bring indexing and searching capabilities to scientific users, FastQuery uses FastBit bitmap indexing technology and supports multiple scientific data formats. Since many scientific applications use NetCDF, HDF5, and ADIOS data formats, FastQuery is a vastly useful system. We will briefly discuss indexing for scientific data formats and bitmap indexing technology before listing a few parallel I/O tuning related efforts.

Most research efforts on indexing and searching techniques are designed for commercial database applications. However with the recent explosion of scientific datasets, researchers are extending indexing techniques for scientific applications as well [24]. Traditional indexing techniques, such as B-tree, are designed primarily to accelerate access to individual data records, such as looking for a customer's bank record [11]. In contrast, a query on scientific data typically returns a fairly large number of records. For instance, a search for accelerated particles in a Laser Wakefield particle accelerator might result in thousands of data records corresponding to thousands of particles. Furthermore, scientific datasets are often produced or collected in bulk, and are never modified. A class of indexing methods that can take full advantage of these characteristics is called the bitmap index.

A bitmap index logically contains the same information as a B-tree index. A B-tree consists of a set of pairs of key value and row identifiers; however, a bitmap index replaces the row identifiers associated with each key value with a bitmap. Because the bitmaps can be operated efficiently, this index can answer queries efficiently as demonstrated first by O'Neil [23]. The basic bitmap index uses one bitmap for each distinct key value. For scientific data where the number of distinct values can be as large as the number of rows (i.e., every value is distinct). The number of bits required to represent an index may scale quadratically with the number of rows. In such a case, an index for 10^9 rows may require 10^{18} bits. Such an index is much larger than the raw data size and is not acceptable except for the smallest datasets.

A number of different strategies have been proposed to reduce the bitmap index sizes and improve their overall effectiveness. Common methods include compressing individual bitmaps, encode the bitmaps in different ways, and binning the original data [24]. FastBit [29] is an open-source software package that implements many of these methods. FastQuery chooses to use FastBit as a representative of general indexing methods. FastBit has been shown to perform well in a number of different scientific applications [29].

2.2 Tuning parallel I/O performance

By using FastBit, FastQuery system can compute bitmap indexes on parallel computers and compress them very efficiently. However, the time spent in these computational steps are typically a relatively small portion of the overall execution time. The remaining time needed to perform reading original data and writing the computed indexes typically require more time. Therefore it is important to make effective use of the underlying parallel I/O subsystem. Various optimization strategies have been proposed to tune performance of I/O operations on a parallel system for a specific application or an I/O kernel. We will not discuss the exhaustive list of these research efforts, but will discuss a few efforts that tried tuning multiple layers of the parallel I/O stack (shown in Figure 1).

Panda project [7, 6] studied automatic performance optimization for collective I/O operations where all the processes of an application synchronize I/O operations such as reading and writing an array. The Panda project searched for disk layout and disk buffer size parameters using a combination of a rule-based strategy and randomized search-based algorithms. The rule-based strategy is used when the optimal settings are understood and simulated annealing is used otherwise. The simulated annealing problem is solved as a general minimization problem, where the I/O cost is minimized. The Panda project also used genetic algorithms to search for tuning parameters [5]. The optimization approach proposed for in this project were applicable to the Panda I/O library, which existed before MPI-IO and HDF5. The Panda I/O is not in use now and the optimization strategy was not designed for parallel file systems that are in current use.

Yu et al. [31] characterize, tune, and optimize parallel I/O performance on Lustre file system of Jaguar, a Cray XT supercomputer, at Oak Ridge National Laboratory (ORNL). This effort tuned data sieving buffer size, I/O aggregator buffer size, and the number of I/O aggregator processes. This study manually ran a selected set of codes several times with different parameters. Howison et al. [15] also perform tuning of various benchmarks that select parameters for HDF5 (chunk size), MPI-IO (collective buffer size and the number of aggregator nodes) and Lustre parameters (stripe size and stripe count) on Hopper supercomputer at the National Energy Research Scientific Computing center (NERSC). These two studies prove that tuning parallel I/O parameters can achieve better performance. Behzad et al. [2] recently explored detecting I/O tunable parameters using genetic algorithms at multiple layers of the stack. In this study to tune I/O performance of FastQuery, we use the knowledge of these research efforts and come up with a local optimum based strategy for selecting optimization parameters.

2.3 Application Use Case: VPIC

We now discuss an application use case where FastQuery was successfully used [4] recently. We also use a subset of data from this application in tuning FastQuery I/O performance in Section 5. This application models a core mechanism of space weather, the collisionless magnetic reconnection. Collisionless magnetic reconnection is an important mechanism that releases energy explosively as field lines break and reconnect in plasmas. This phenomenon can release massive energy to damage satellite communication equipments and produce beautify aurora borealis. Such a reconnection also plays an important role in a variety of astrophysical applications involving both hydrogen and electron-positron plasmas.

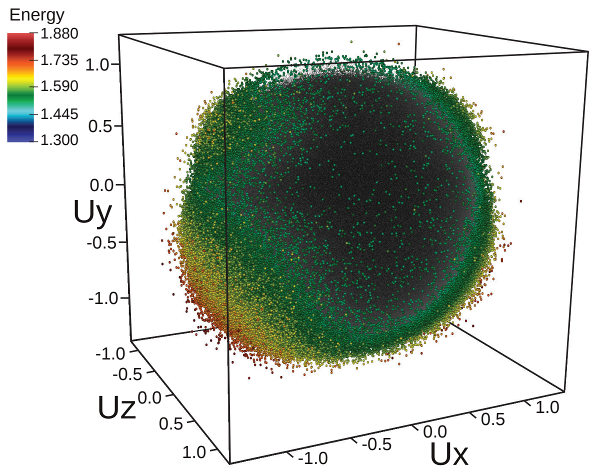


Figure 2: Visualization of the 1 trillion electron dataset showing all particles with energy value > 1.3 (gray). In addition, all particles with energy value > 1.5 are shown in color, with color indicating energy. The queries result in $164,856,597$ **particles with energy > 1.3 and** $423,998$ **particles with energy > 1.5.**

We recently ran a particle simulation that simulates collisionless magnetic reconnection with open boundaries in 3D. The simulation code is the highly optimized particle code, called VPIC [3]. The simulation tracks two trillion particles on Hopper supercomputer at NERSC. Particle properties of interest in this simulation include spatial location (x,y,z), energy, and projection of velocity components on the directions parallel and perpendicular to the magnetic field $U_{\|}$, $U_{\perp,1}$, and $U_{\perp,2}$. The data size of each particle is $32 bytes$, and the number of particles increases as the simulation progresses. Data written in each time step varied between $30TB$ to $42TB$ data.

The analysis of VPIC dataset requires only part of the whole dataset where interesting features for science lie. The interesting feature scientists look for in the trillion particle data is the set of particles that have energy values greater than a specified threshold, such as 1.1, 1.3, 1.5, and generating various types of histograms. Assuming these subsets of data with different thresholds and histograms are computed separately, the traditional technique needs to read all the trillion particles multiple times. This technique is highly inefficient because of reading $40TB$ dataset is very time consuming. With FastQuery and FastBit, one can build indexes of the data and search for the threshold and generate histograms by reading only the data that satisfy the thresholds. In our recent study, we have shown that indexing a trillion particles takes 10 minutes and querying the indexed data takes about 3 seconds. This is a few orders of magnitude faster than the traditional scanning approach. Figure 2 shows a visualiza-

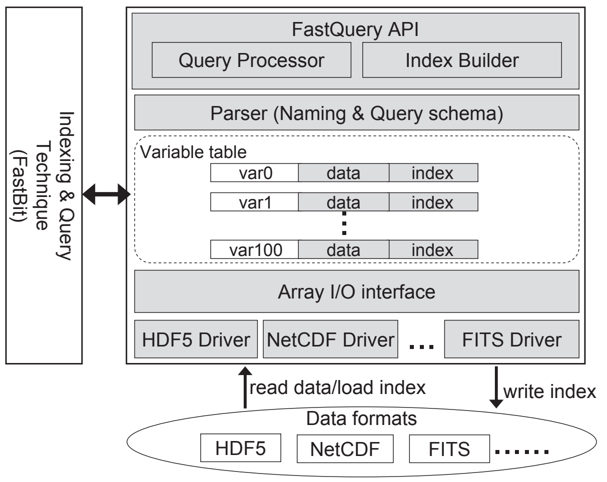


Figure 3: An illustration of the overall design of FastQuery.

tion of the phase space of particles with energies greater than 1.3 from the 1 trillion particle dataset. Generation of this figure uses FastQuery plugin to the VisIt visualization software [28, 25].

Even though FastQuery performed nicely in the previous studies, we see an opportunity to systematically tune the I/O operations to further improve performance of FastQuery. In the next section, we will describe the options we plan to examine.

3. TUNING OPTIONS

On today's supercomputer systems with their complex hardware and deep software stacks, it is often difficult to reach optimal performance. As observed from our previous studies [9, 10, 4], indexing is a time consuming operation that could easily take more than a few hours without careful tuning or configuring the libraries along the I/O path. Hence, in this section, we describe the parallel strategies involved during indexing, and explain the parameters that could be tuned to improve the overall system performance to be discussed later in Section 5.

3.1 FastQuery overview

The objective of FastQuery is to support data selection on scientific data formats based on arbitrary range conditions defined on the available data values, e.g., "energy $> 10^5$ and temperature $> 10^6$". It has two main functionalities: *indexing* and *querying*. FastQuery provides a unified array interface to various scientific data formats by using FastBit bitmap indexing and querying technology to accelerate queries. An illustration of the overall design of FastQuery is shown in Figure 3. Our current implementation supports high-level data formats including HDF5 [26], NetCDF [27], pNetCDF [18] and ADIOS-BP [20].

The indexing function builds bitmaps of data in a file and stores them into the same file or a dedicated index. This indexing operation contains three main steps: (1) read the original data values from the file, (2) construct bitmap indexes data structure in memory, and (3) write the bitmaps to the file. The querying function evaluates different queries by accessing the data and the indexes. If the necessary indexes have not been built, FastQuery simply scans through the data values for evaluation. But typically, when the necessary indexes are available, the querying process involves two main steps: (1) load the bitmaps from file, and (2) evaluate the indexes with the given query.

In order to process massive datasets such as the ones shown in the above application use case, we designed FastQuery to exploit parallelism at both computation and I/O levels. To take advantage of distributed memory nodes and multiple CPU cores systems, FastQuery divides a full dataset into multiple fixed size subarrays, and builds the indexes of those subarrays iteratively by assigning each subarray to a single thread from an MPI domain as shown in Figure 4. It can be noted that an MPI domain is similar to an MPI task, but each task contains multiple threads. Previously, we have implemented parallel FastQuery using MPI alone [9, 10]. In a recent exercise, we use Pthreads to make each MPI task spawn multiple threads [4]. In that work, we explained reasons for choosing this hybrid parallel approach and opting Pthreads instead of software packages such as OpenMP, TBB, and so on. The current work continues to use the same MPI and Pthreads combination, but aims to systematically study the tuning parameters to improve the overall performance.

The indexes of the subarrays are built, collected together, and then stored into an index file by going through layers of software stack and I/O libraries described. FastQuery first initiates I/O operations using the scientific data format library that manages the logical view of a file. Specifically, in this paper, we choose HDF5 as the data format library for our study. The HDF5 library issues I/O requests to the file through a MPI-IO layer which enable multiple MPI tasks operate on a single file in parallel. Finally, at the lowest level, the I/O requests are handled by a parallel file system that controls the physical layout of the file across multiple disks, and performs the actual data read/write operations in parallel. Except for the HDF5 specific tuning parameter described in Section 3.3, the tuning process is same for all the file formats FastQuery supports. Therefore, this study can produce useful parameter choices for all FastQuery users.

3.2 FastQuery Parallel I/O Strategy

Parallelism of FastQuery can be controlled by three key parameters. First parameter is the *subarray_size*, the length of subarray for dividing the dataset. Since the size of indexes is often proportional to the size of its data, choosing a balanced subarray size is vital for obtaining high performance. For example, smaller subarray size results in each thread reading or writing small amounts of data. But, given the same dataset and number of cores, smaller subarray size also creates more subarrays, and thus requires more iterations to finish. In other words, smaller subarray size reduces the I/O size from each thread, but increases the number of I/O requests. Therefore, as shown in our results in Section 5, larger subarray could achieve better performance by minimizing the number of I/O requests. However, the size of subarray is limited by the number of cores in a multicore CPU and the size of available memory for each core. During

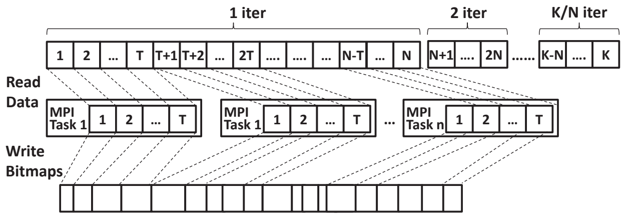
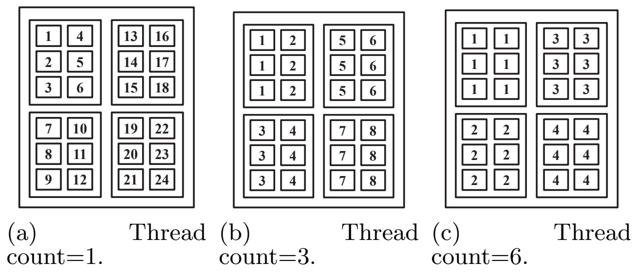


Figure 4: In the hybrid parallelism design of FastQuery, each subarray is assigned to one of the threads from a single MPI task. Threads individually read data and write bitmaps to file. It is worth noting that the size of bitmap from each thread is not uniform and Pthreads provide more flexibility in dealing with non-uniform data than OpenMP or TBB.



(a) Thread count=1. (b) Thread count=3. (c) Thread count=6.

Figure 5: Placement of MPI tasks with the support of threading. When *thread_count* $= 1$**, one MPI task is launched on each core. With** *thread_count* $= 3$**, two MPI tasks are launched on each NUMA node, and each MPI tasks creates three threads. When** *thread_count* $= 6$**, only a single MPI task runs each NUMA node, but it is still able to utilize all the cores by creating 6 threads.**

index building all the associated data of a subarray including its data values and indexes must fit into the memory during the indexing operation. Since the newer generation is expected to have less memory per core than the current generation of supercomputers, it is useful to consider the performance impact of working with smaller subarrays.

The second FastQuery tuning parameter is *thread_count*, which is defined as the number of threads spawned from each MPI task in FastQuery. As shown in Figure 4, FastQuery allows each MPI task to create a fixed number of threads. The MPI tasks are only responsible for holding shared resources among threads, such as the MPI token for inter-process communication and the memory buffer for collective I/O, while the threads do the actual processing tasks of creating indexes and evaluating queries. Figure 5 shows three possible settings on a 24-core compute node consisted of four 6-core NUMA nodes. As shown in the figure, with the support of threading, the number of MPI tasks can be reduced without sacrificing the degree of parallelism. Reducing the number of MPI tasks also reduces the amount of system resources needed to manage the MPI tasks.

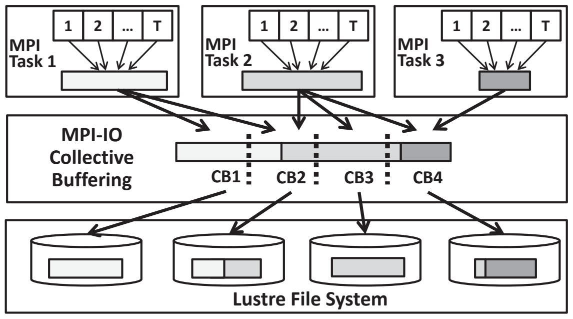


Figure 6: FastQuery I/O aggregation among threads to utilize collective buffering provided by the MPI-IO library.

Finally, we implemented an I/O aggregation technique to collect from and distribute data to all the threads before writing or reading, respectively. The aggregation of data reduces the number of I/O requests from FastQuery. Figure 6 illustrates the aggregation technique. FastQuery assigns subarrays to threads in such a way that the threads from an MPI task always read/write a continuous data region of a dataset. Therefore, if the aggregation is enabled, the master thread of each MPI task simply allocates a memory space to collect the data that needs to be read or written from other threads and makes a single I/O request to handle the whole data region from an MPI task. In contrast, without the aggregation, each thread would make its own I/O request call individually, which degrades I/O performance significantly. While aggregation can reduce the number I/O requests, it also involves an extra memory between the threads. However, this user-level aggregation may also incur certain costs. For example, when writing indexes for different subarrays, the aggregation procedure needs to copy the bitmaps and associated metadata to a single large buffer. This increases the memory requirement and produces another copy of the in memory data. On reading, the raw data comes into a single buffer which might be associated with one of the processor core and increases the memory access cost of other cores.

3.3 HDF5 and MPI-IO Collective I/O

HDF5 library is designed to support hierarchical object-database representation of scientific data. It is also designed to minimize performance contention on large HPC systems by coordinating I/O to single, shared files. To improve performance and hide the complexity, HDF5 encapsulates several general purpose optimizations as discussed in [16]. According to that study, the most useful feature is the collective buffering built upon the MPI-IO routines for collective and independent I/O operations.

The collective buffering technique of HDF5 [19] attempts to improve I/O workload distribution with respect to the underneath parallel file system stripe boundaries as shown in Figure 6. In the example, 4 nodes (i.e., CB1 $\sim$ CB4) are chosen to be the collective I/O nodes. Each collective buffering (CB) node is responsible for the I/O of a single physical I/O block (i.e. Lustre stripe) in the parallel file system by collecting its content from multiple MPI tasks. Thus, each physical I/O block only interacts with a single

CB node resulting in reduced locking overhead at the file system level. However, as indicated from the previous studies [12], the communication overhead associated with this technique could still outweigh the saving in I/O time under some circumstances. For example, it has been shown that small writes that spread far apart relative to the stripe size will result in poor I/O performance regardless using collective buffering or not.

In this work, the collective buffering algorithm evaluated in our experiment is the version CB 2. It has been integrated into the Message Passing Toolkit (Cray MPT 3.2) on the Cray systems. The CB 2 algorithm implements a Lustre-optimized scheme that uses static-cyclic and group cyclic Lustre stripe aligned methods described by [19] Cray implementation of these methods merged the Lustre "abstract device-driver for I/O" (ADIO) code from Sun Microsystems with its own code to provide additional tunable parameters to optimize performance. We explain the details of Lustre striping in the following subsection. The *cb_nodes* MPI hint decides the number of collective buffer nodes. By default, *cb_nodes* is set equal to the stripe count of a file, so that each CB node maps to a disk (i.e. Object Storage Target or OST in Lustre file system). The cb_buffer_size determines the memory size of collective buffer on each collective buffer node. Since each I/O request is to read/write a single Lustre stripe, the default value is set equal to the size of a Lustre stripe. Finally, the feature can be disabled by setting the MPI-IO hint *romio_cb_write = disable*.

3.4 Lustre Striping

At the raw I/O level, the parallel file system used in our study testbed is Lustre, which is commonly used by many supercomputers. Lustre provides I/O parallelism by striping data across multiple disks (i.e. OSTs). The striping of a file is controlled by two parameters: *stripe size* and *stripe count*.

Stripe count is the number of OSTs to be used for storing a file, and stripe size is the number of bytes written on one OST before cycling to the next. In Cray MPT 3.2, those two parameters are given as MPI-IO hints at file creation. The parameter $striping\_factor$ sets the Lustre stripe count and *striping_unit* sets the Lustre stripe size. In general, larger stripe count provides better parallelism because the data can be read/write from more disks simultaneously. However, if the parallel file system is shared, using larger stripe count also increases the possibility of I/O contention with other users.

The stripe size setting often depends on the I/O size. If the stripe size is larger than the I/O size, multiple I/O requests is likely to access the same Lustre stripe and suffer from locking overhead. On the other hand, if the stripe size is smaller than the I/O size, multiple Lustre stripes must be accessed to serve a single I/O request. As observed from previous studies and our experimental results, the striping parameters have significant impact on the parallel I/O performance.

4. EXPERIMENTAL SETUP

4.1 Testbed

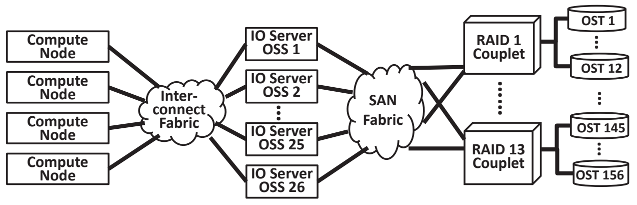


Figure 7: The set up of Lustre file system in our testbed. It consists of 13 LSI disk controller, 26 I/O servers (OSSs) and 156 storage targets (OSTs).

We have conducted our experiments on the NERSC Cray XE6 supercomputing system named Hopper [1]. The system has ≈6,500 compute nodes, with 24 CPU cores and 32GB memory per node. In our experiments, we control the number of MPI tasks and threads to always use all 24 cores on each compute node. In all the experiments discussed in Section 5, we use the same dataset for indexing at a fixed concurrency of 5136 cores.

The filesystem used by Hopper is Lustre as shown in Figure 7. It consists of 13 LSI 7900 disk controllers. Each disk controller is served by two I/O servers, called OSSs (Object Storage Servers), and each OSS hosts six OSTs (Object Storage Target). Hence, there is a total of 156 OSTs which can be considered as a software abstraction of a physical disk. Striping data across multiple OSTs instead of the default stripe count of 2 OSTs allows users to potentially increase their read and write I/O bandwidth.

4.2 Datasets

Our performance study is based on the dataset collected from a 1.2-trillion particle simulation run of the highly optimized particle code VPIC [3] as described in Section 2.3. The dataset is stored in a HDF5 file with 8 variables, and each variable is a one-dimensional floating point array of 1.2 trillion elements. The total file size of the dataset is 34TB. The Lustre stripe size and count is set during file creation with the stripe size = 64MB and stripe count = 144.

We use two subset datasets from the file for our tuning study and performance evaluation in Section 5 to represent datasets of different sizes. We refer the two subset datasets as `Large dataset` and `Small dataset` through the rest of the paper. Both Large and the Small datasets contain only one variable from the original HDF5 file. The Large dataset has about 1 trillion particles (i.e., 963 million to be exact), but the Small dataset only has 7.7 million particles which is 1/128th of the Large dataset size. The odd size of the Large dataset is to make sure the number of elements in the array can be evenly divided by the number of processors so as to minimize the load imbalance among the CPU cores and reduce the variation in I/O performance.

Since input data is only read by FastQuery, their Lustre stripe size and count settings cannot be changed by FastQuery, and remains with 64MB and 144, respectively.

[1]`http://www.nersc.gov/nusers/systems/hopper2/`

dataset name	file size	indexes size	subarray size	read size per core	write size per core
Large	3.76TB	2.98TB	16M	64MB	44MB
Small	30.1GB	23.7GB	128K	512K	378K

Table 1: Descriptions of the Large and the Small datasets used in our evaluations.

4.3 Methodology

In our experiments, we report the execution time and I/O bandwidth from indexing the Small and the Large datasets using different subarray sizes. The subarray size used for the Large dataset is 16 million particles, and the subarray size for the Small dataset is 128 thousand particles. We carefully choose those numbers so that the total number of particles are evenly distributed among the subarrays and the subarrays are then evenly divided among the cores (or threads). Given the 5136 cores used in our test, bitmap indexes in both datasets are built in exactly 12 iterations.

The indexing precision used in the experiments is 1-digit [29]. Since the subarray size is fixed, all cores read the same amount of data in every iteration. That is 64 MB per core for the Large dataset and 512 KB per core for the Small dataset. The sizes of bitmaps computed from each subarray are varied depending on the values, but they are roughly the same as we observed. For the Large dataset the size of bitmaps is about 44 MB per subarray, and for the Small dataset it is about 378 KB. Overall, each experiment involves reading a total of 3.76 TB data and writing bitmaps of 2.98 TB for the Large dataset. For the Small dataset, the data read is 30.1 GB and the bitmaps written are of 23.7 GB size. We summarize the description and characteristics of the two subset datasets in Table 1.

We observed significant variability in the performance results collected from our experiments. This is a typical behavior because the file system used in our testbed is a shared resource and therefore prone to contention from other users. In addition, as observed in a previous study [21], many other issues, such as the complexity of the I/O software stack and the contention of non-IO MPI communication, could also contribute to the variability. To obtain reasonable peak I/O performance values, we repeat each experiment at least 5 times over the course of several weeks. As suggested by Howison et al. [16], we use the Lustre Monitoring Tool (LMT) [13] to detect conspicuous cases of contention from other user applications and eliminate those timing results from our study. In all the plots for our experimental results, we report the medium number from our repeating measurements as our best approximation of the I/O bandwidths. We also use the error bar on the figures to indicate the 25^{th} and 75^{th} percentile numbers resulting from the experiments.

5. EXPERIMENTAL RESULTS

In this section, we first present the overall performance improvement from our tuning in Section 5.1. Figure 8 shows the overall performance improvement that reports the total time to index the selected datasets after we tune each of the parameters. In the following subsections, we provide the details of our tuning process by examining one parameter at a time from the lowest level of the I/O stack, i.e. Lustre parallel file system, to the highest level of the stack, i.e. FastQuery itself, and we discuss the performance impact from each of the parameters. At the end, tuned FastQuery implementation achieved ≈20 GB/s I/O bandwidth on Lustre file system. Compared with our previous base implementation, the tuned FastQuery achieved an overall performance speedup of 3.7X for the Large dataset, and that of 11.6X for the Small dataset.

5.1 Overall Performance Evaluation

The default setting of Lustre stripe size on Hopper is 1 MB and the default stripe count is 2. By using the default settings, writing indexes for both datasets took longer than 1 hour limit we set. Hence, as shown in Table 2, we use a baseline setting to start our tuning with the Lustre stripe count of 10, the stripe size of 4 MB, and disabling all the optimization options including collective buffering, threading, and thread aggregation. Performance with these settings are much better than that with the default setting.

As described in Section 3, indexing involves three major operations: (1) read data values; (2) compute indexes; and (3) write indexes. More specifically, writing indexes includes writing the bitmaps and the associated metadata of bitmaps. Hence, in Figure 8 we show the time spent on each of these four operations separately in different color.

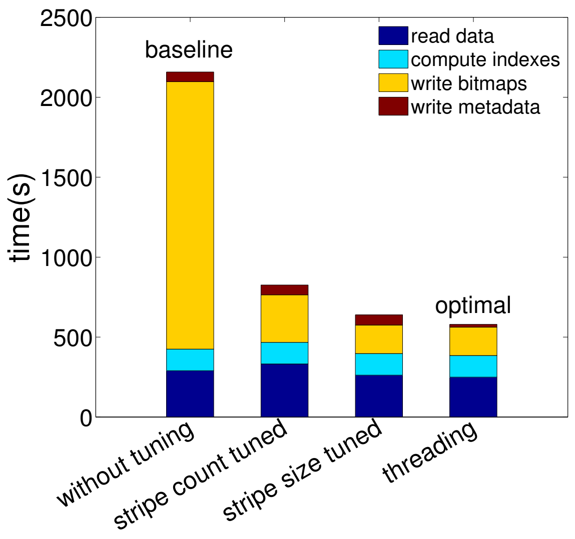
For both the Large and the Small datasets, we clearly observe that the total indexing time is dominated by the I/O, especially for the baseline case without tuning. Since we do not change the number of cores used in our test, the index computation time remains constant throughout our experiments. The data read time was not affected by striping optimizations because we are reading a data file that was written to file system. For writing the index file, we varied stripe count, stripe size, MPI-IO collective buffer parameters, and the number of FastQuery threads per MPI process. For reading the original data, we varied collective buffer and FastQuery threads per MPI process. We observe that the main contribution to performance improvement comes from writing index file portion (write bitmaps).

Specifically, for the Large dataset shown in Figure 8 (a), we first change the stripe count from 10 to 144 to exploit the I/O parallelism at file system level, and successfully reduce the time from 2158s to 826s. Then we increase the stripe size from 4 MB to 64 MB, so that each Lustre stripe can roughly match to the FastQuery I/O size (i.e. 44 MB). As a result, the time further reduced from 826 seconds to 639 seconds. Finally, we enable threading and aggregation options in FastQuery to reach our optimal time of 580s. We did not enable collective I/O and buffering because enabling them actually increases the total index building time on the Large dataset. Overall, the indexing time reduces by a factor of 3.7 from 2193s to 580s.

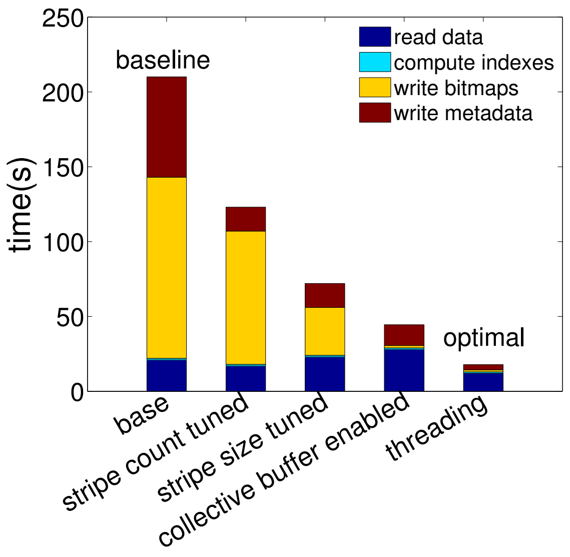
For the Small dataset shown in Figure 8 (b), the time is also significantly reduced from 210s to 123s and then 72s, after we increase the stripe count to 144 and reduce the stripe size to 1 MB. Different from the Large dataset, because each I/O size is small from FastQuery, enabling collective I/O reduces locking overhead in Lustre and reduces the time from 72s to 45s. Finally, again with threading, the indexing time eventually reduces to 18s. The index writing time reduces by 11.6X from 210s to 18s.

	setting	Lustre		MPI-IO	FastQuery		total running time	speedup
		striping count	striping size	collective buffering	thread aggregation	thread count		
Large dataset	baseline	10	4 MB	disable	disable	1	2158	-
	optimal	144	128 MB	disable	enable	6	580	**3.7**
Small dataset	baseline	10	4 MB	disable	disable	1	210	-
	optimal	144	1 MB	enable	enable	6	18	**11.6**

Table 2: The setting and performance results of both the Large and the Small datasets before and after our tuning.



(a) Large dataset.



(b) Small dataset.

Figure 8: Overall performance (indexing time) improvement after tuning each of the parameters to the optimal setting for the Large and the Small datasets.

5.2 Stripe Count

We first study the performance impact of Lustre stripe count in writing bitmap index file. For the "baseline" testing, we have set the stripe count of 10 and the stripe size of 64 MB for the Large dataset. We have set these parameters to be 10 and 4 MB for the Small dataset, respectively. The rest of the parameter settings are the same as the baseline described in Table 2.

As shown in Figure 9, the I/O bandwidth (larger the better) improves as stripe count increases for both the Large and the the Small datasets. For the Large dataset, when stripe count is 10, the I/O bandwidth is only 1.95 GB/s, but when stripe count is 156, the I/O bandwidth improves to 19 GB/s, which is $\approx 9.7X$ improvement. This result is expected because larger stripe count allows higher degree of I/O parallelism among Object Storage Targets (OSTs) of Lustre file system. However, we also found that the improvement becoming smaller for the stripe count values larger than 120. This is likely due to the fact that there are only 26 I/O servers (i.e. OSSs) available to serve the I/O requests. As the load gets closer to the theoretical peak bandwidth (i.e. ≈ 35 GB/s in this testbed), more resource contentions and locking overhead is expected.

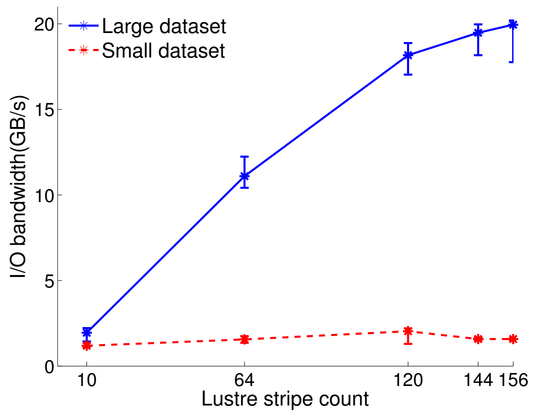
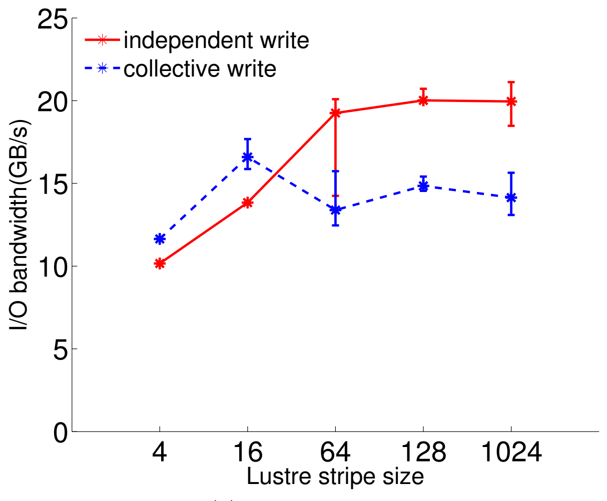
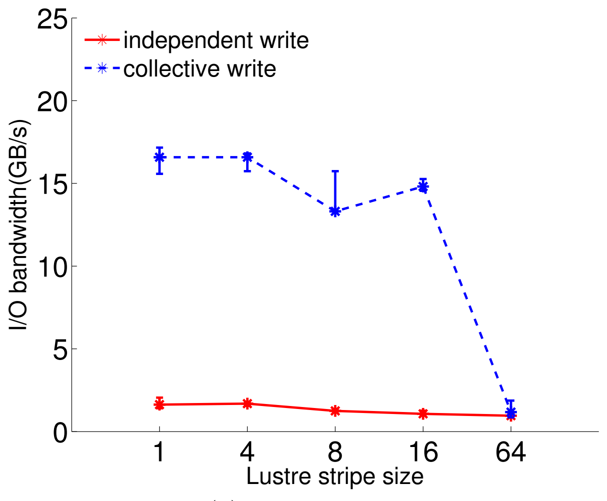


Figure 9: I/O bandwidth vs. stripe count. We select 144 for both the Large and the Small datasets for future tests.



(a) Large dataset.



(b) Small dataset.

Figure 10: I/O bandwidth vs. Lustre stripe size. We choose 128MB stripe size with independent I/O for Large dataset, and 1MB stripe size with collective I/O for Small dataset.

For the Small dataset, the I/O bandwidth is much worse than the Large dataset because of its small I/O size per request. The performance improvement as we increase the stripe count was marginal. The bandwidth slightly drops when the stripe count is greater than 120. This poor I/O performance and the drop is because the amount of data accessed in the Small dataset is not large enough to utilize the parallelism from the file system. Accessing smaller amount of data is also affected by interference from other applications accessing the I/O subsystem. As choosing all 156 OSTs may have a possibility for I/O contention, we decide to choose 144 as the tuned stripe count for the rest of tuning process.

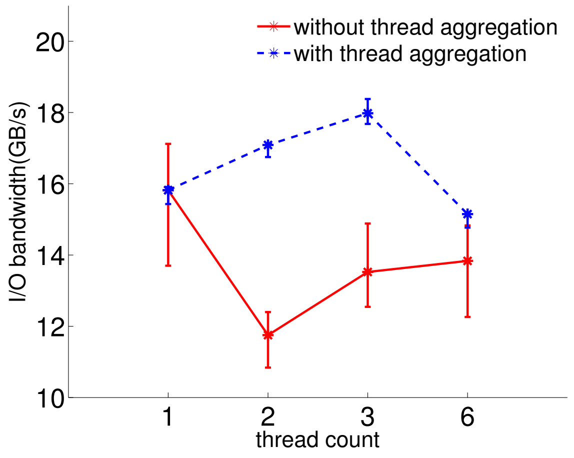
5.3 Stripe Size & Collective Buffering

We then evaluated the performance impact from tuning the Lustre stripe size. All the parameter settings are the same as the default values shown in Table 2, except the stripe count is set to 144 according to the decision from the previous subsection. Because the stripe size of the input dataset file is fixed, here we only report the I/O bandwidth of writing indexes. The results of both with and without using MPI collective buffering are presented in Figure 10.

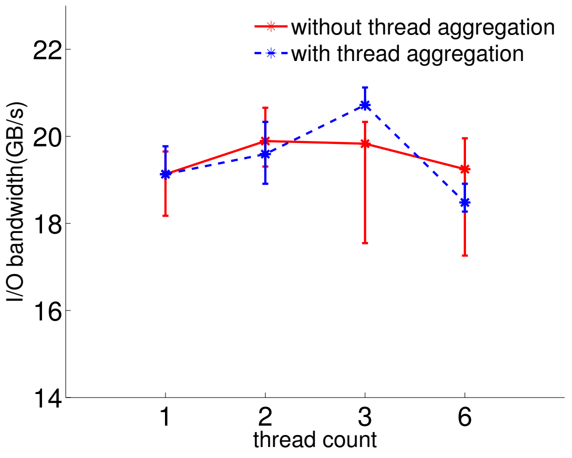
We first examine the results of the Large datasets shown in Figure 10 (a). With calling HDF5 independent I/O API without using MPI-IO collective buffering, the I/O bandwidth gradually improved as the stripe size increases. But once the stripe size is larger than 64 MB, the performance seems to be remaining the same. The likely explanation of this trend is the size of bitmaps that needs to be written from each core is roughly 44 MB under the given subarray size of 16 million elements. Hence, a smaller Lustre stripe size will force to divide a single I/O request from the FastQuery into multiple Lustre physical data block I/O requests. As a result, more I/O contentions and queuing delay could occur. On the other hand, if the stripe size is larger than the I/O size from FastQuery, multiple I/O requests is likely to access the same Lustre stripe, and may suffer from the locking overhead. However, we do not observe such performance degradation in this experiment. We believe it is because each I/O size is relatively larger (i.e. 44 MB), thus the data transfer time would dominate overall performance comparing to the locking overhead.

In contrast, the dash line in Figure 10 (a) plots the results of calling the HDF5 collective I/O API and using the MPI-IO collective buffering the CB 2 algorithm [19]. We found the performance is relatively un-correlated to the Lustre stripe size. It is because the CB 2 collective buffering algorithm is designed with an attempt to alter the I/O pattern and size from users to match the stripe boundaries of Lustre. Therefore, the collective buffering can accommodate the changes of Lustre stripe size to optimize I/O performance. However, comparing the two plots in Figure 10 (a), we observe the collective I/O only performs better than the independent I/O with smaller stripe size (i.e. less than 16 MB). That means the communication cost of collective buffering could be exceeding its I/O benefit with a larger I/O size and the stripe size due to the limited locking overhead under this circumstance.

Figure 10 (b) plots the results of the Small dataset. Because the subarray size used for the Small dataset is only 128K elements, the bitmaps built from each core and written to file is also reduced from 44 MB to around 378 KB. With such small I/O size from FastQuery, we found collective buffering can substantially improve the I/O bandwidth in any of the stripe sizes we tested. But the performance still decreases with larger stripe size, and especially when the stripe size is as large as 64 MB. Therefore, for the rest of study, we decide to pick stripe size 128 MB with independent I/O for the Large dataset, and pick stripe size 1 MB with collective I/O for the Small dataset. Those setting also results in the peak performance we observed from Figure 10.



(a) Read performance.



(b) Write performance.

Figure 11: I/O bandwidth vs. the thread count per MPI task for the Large dataset. We choose stripe count=6 with enabling thread aggregation for the Large dataset.

5.4 Thread Count per MPI Task

Finally, we evaluate the performance benefit from the two optimization techniques that implemented in FastQuery: hybrid parallelism and thread aggregation.

As mentioned in Section 3, FastQuery parallelizes its workload by combining both MPI and threads. The ratio between MPI tasks and threads is tunable by a parameter called *thread_count* whose value determines the number of threads created from each MPI task. While the *thread_count* can be given arbitrary, its setting is highly depending on the CPU architecture. Take Hopper's compute node as an example where each compute node is consisted of 4 NUMA nodes, and each NUMA node has 6 cores. To minimize memory access latency, all threads from a MPI task must be assigned to the same NUMA node. Furthermore, over commit a single core to multiple threads should not be allowed to prevent performance degradation. Therefore, in order to utilize all the cores without conflicting to the above requirements, the only feasible settings in our testbed are: 1, 2, 3, and 6 as illustrated in Figure 5.

For the Large dataset, Figure 11 summarizes the read and write I/O performance under various setting for thread count. In the plots, we also include the results of with and without using thread aggregation technique. With thread aggregation, it aggregates the I/O data from all the threads and makes a single HDF5 MPI-I/O call for each MPI task. Overall, thread count=6 performs relatively well for both read and write. Specifically, for write performance, we observed the best result is at thread count=3 with thread aggregation. The reason is likely because of the Lustre stripe size is 128 MB, and the output bitmap size from each thread is only around 40 MB. So, by aggregating the output data from 3 threads, we can match the I/O size of each write request to the Lustre stripe size. Once the thread count increases to 6, the I/O size becomes larger than the stripe size, and the bandwidth drops down immediately, from 20.72 GB/s to 18.48 GB/s. We also observed thread aggregation seems to benefit the read performance more than the write, but we are still investigating the causess.

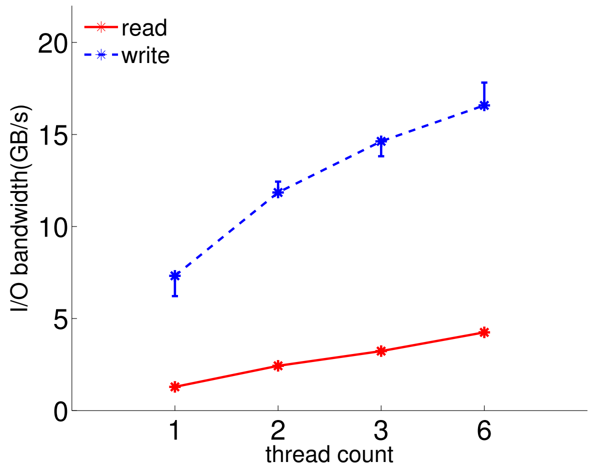


Figure 12: We choose thread count=6 with enabling thread aggregation for the Small dataset.

For Small dataset, we plot the results in Figure 12. Because we chose to use collective I/O at the MPI-IO level from the previous tuning study, thread aggregation must be enabled to guarantee one I/O request per MPI tasks in our current FastQuery implementation. Hence we only show the results with enabling thread aggregation for the Small dataset. Unlike the result for the Large dataset, we observed higher thread count clearly can improve I/O bandwidth for both read and write. Specifically, when the thread count increases from 1 to 6, the write bandwidth also grows from 7.36 to 16.58, a factor of 2.26. Similarly, the read performance was significantly increased from 1.29 to 4.25, a factor of 3.29 improvement. This result indicates that replacing MPI tasks with threads can effectively reduce the overall communication cost. But the communication overhead will only be a factor if the I/O size is smaller. Therefore, we weren't able to observe the same benefit on the the Large dataset in Figure 11. Finally, the read performance is very poor in Figure 12 because the file stripe size (i.e. 64 MB) is much larger than the data size of each subarray (i.e. 496 KB) in the experiment. As we have shown in the stripe size study in Figure 10 (b), the performance will be worse with larger stripe size for small I/O. Therefore, the same conclusion is verified in the experiment as well.

6. CONCLUSIONS

Selecting informative records from the massive data generated from large scale scientific application simulations has been a difficult challenge for scientific data analysis. To address the problem, we have developed FastQuery, a parallel indexing and query system, with the objective to accelerate the data selection process by utilizing indexing and parallel I/O techniques. In this paper, we describe the parallel strategies and tunable parameters applied in each of the I/O stack of FastQuery including at the level of application, HDF5, MPI-IO and Lustre file system. Then we present a systematic approach of choosing parameters that can achieve significant portion of the peak I/O bandwidth based on the observations and analysis of our study.

We found the optimal setting is highly dependent on the data size of each I/O request from FastQuery, and that size is controlled by the subarray size for dividing workload in FastQuery. At the parallel file system level, we suggest to maximize the Lustre stripe count without using all OSTs, and pick Lustre stripe size that is larger but closer to the subarray size. At the HDF5 and MPI-IO level, we found collective I/O and buffering can indeed improve performance for the small size I/O, but not for the large size I/O. Hence, we should disable collective buffering and use HDF5 independent I/O when using larger FastQuery subarray size. Finally, we developed two parallel strategies inside FastQuery to support two parallel strategies: hybrid parallelism and thread aggregation. We found both strategies can effectively improve I/O performance in most settings, especially in the case of smaller subarray size.

In the future, we would like to base on the finding in this paper to provide auto-tuning of FastQuery, so that users can receive good performance without requiring the detailed understanding of the complex I/O stack used by FastQuery.

7. ACKNOWLEDGMENTS

This work was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing (NERSC) Center. We gratefully acknowledge the help received from many NERSC staff, particularly David Turner.

8. REFERENCES

[1] IPCC Fifth Assessment Report. `http://en.wikipedia.org/wiki/IPCC_Fifth_Assessment_Report`.

[2] B. Behzad, J. Huchette, H. Luu, R. Aydt, S. Byna, Y. Yao, Q. Koziol, and Prabhat. A framework for auto-tuning hdf5 applications. In *HPDC*, 2013. `https://sdm.lbl.gov/~sbyna/research/papers/hpdc2013.pdf`.

[3] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):7, 2008.

[4] 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 *SC*, pages 59:1–59:12, 2012.

[5] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization using genetic algorithms. In *HPDC*, pages 155 –162, jul 1998.

[6] Y. Chen, M. Winslett, Y. Cho, S. Kuo, and C. Y. Chen. Automatic Parallel I/O Performance Optimization in Panda. In *In Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 108–118, 1998.

[7] Y. Chen, M. Winslett, S.-w. Kuo, Y. Cho, M. Subramaniam, and K. Seamons. Performance modeling for the panda array I/O library. In *SC*, 1996.

[8] J. Chou, K. Wu, and Prabhat. FastQuery: A general indexing and querying system for scientific data. In *SSDBM*, pages 573–574, 2011.

[9] J. Chou, K. Wu, and Prabhat. FastQuery: A parallel indexing system for scientific data. In *Proceedings of Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2011.

[10] 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 *SC*, pages 30:1–30:11, 2011.

[11] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[12] Cray. Getting Started on MPI I/O, Dec. 2009. CrayDoc S-2490-40.

[13] C. M. Herb Wartens, Jim Garlick. LMT - The Lustre Monitoring Tool. `https://github.com/chaos/lmt/wiki`.

[14] T. Hey, S. Tansley, and K. Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft, Oct. 2009.

[15] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre File Systems. In *Proceedings of Workshop on Interfaces and Abstractions for Scientific Data Storage*, Heraklion, Crete, Greece, Sept. 2010. LBNL-4803E.

[16] M. Howison, Q. Koziol, D. Knaak, J. Mainzer, and J. Shalf. Tuning HDF5 for Lustre File Systems. In *Proceedings of Workshop on Interfaces and Abstractions for Scientific Data Storage*, Heraklion, Crete, Greece, Sept. 2010. LBNL-4803E.

[17] J. Kim, H. Abbasi, L. Chacón, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu. Parallel in situ indexing for data-intensive computing. In *LDAV*, pages 65–72. IEEE, 2011.

[18] J. Li, W. keng 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 *SC*, page 39, 2003.

[19] W.-k. Liao and A. Choudhary. Dynamically adapting file domain partitioning methods for collective i/o

based on underlying parallel file system locking protocols. In *SC*, pages 3:1–3:12, 2008.

[20] 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 *CLADE*, pages 15–24, 2008.

[21] J. Mache, V. Lo, and S. Garg. The impact of spatial layout of jobs on I/O hotspots in mesh networks. *JPDC*, 65(10):1190–1203, Oct. 2005.

[22] P. O'Neil and E. O'Neil. *Database: principles, programming, and performance*. Morgan Kaugmann, 2nd edition, 2000.

[23] P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, London, UK, UK, 1989. Springer-Verlag.

[24] A. Shoshani and D. e. Rotem. *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman & Hall/CRC Press, 2009.

[25] K. Stockinger, J. Shalf, W. Bethel, and K. Wu. Query-driven visualization of large data sets. In *IEEE Visualization*, pages 167–174, Oct. 2005.

[26] The HDF Group. HDF5 user guide. `http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.user.html`.

[27] Unidata. The NetCDF users' guide. `http://www.unidata.ucar.edu/software/netcdf/docs/netcdf/`.

[28] VisIt Visualization Tool. `https://wci.llnl.gov/codes/visit/`.

[29] K. Wu. FastBit: an efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 16:556–560, 2005.

[30] K. Wu, E. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31:1–38, 2006.

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