SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats

Yi Wang, Wei Jiang, Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University Columbus OH 43210
{wayi,jiangwei.agrawal}@cse.ohio-state.edu

Abstract—Despite the popularity of MapReduce, there are several obstacles to applying it for developing scientific data analysis applications. Current MapReduce implementations require that data be loaded into specialized file systems, like the Hadoop Distributed File System (HDFS), whereas with rapidly growing size of scientific datasets, reloading data in another file system or format is not feasible.

We present a framework that allows scientific data in different formats to be processed with a MapReduce like API. Our system is referred to as SciMATE, and is based on the MATE system developed at Ohio State. SciMATE is developed as a customizable system, which can be adapted to support processing on any of the scientific data formats. We have demonstrated the functionality of our system by creating instances that can be processing NetCDF and HDF5 formats as well as flat-files. We have also implemented three popular data mining applications and have evaluated their execution with each of the three instances of our system.

I. INTRODUCTION

Gaining insights from data to facilitate scientific discoveries has emerged as the fourth paradigm for research, in addition to theory, experimentation, and computation. At the same time, the amount of data collected from instruments or generated from simulations is increasing at a massive rate. Thus, there is a growing need for tools that can help the development of data analysis applications, which can then be executed in a scalable fashion on large datasets.

In both scientific and commercial sectors, there has been a lot of interest in data-intensive computing. Much of the activity in this area has been around the MapReduce paradigm for implementing data-intensive applications [7]. MapReduce and its variants are quite effective in simplifying the development of data-intensive applications, through the use of a simple API, and with a robust runtime system.

Despite the popularity of MapReduce, there are several obstacles to applying it for developing scientific data analysis applications. Current MapReduce implementations require that data be loaded into specialized file systems, like the Hadoop Distributed File System (HDFS). On the other hand, high performance computing systems where large-scale simulation data is produced use file systems like PVFS. Though certain recent developments could enable execution of Hadoop jobs on PVFS [18], [5], scientific data tends to be stored in formats like NetCDF, HDF5, or ADIOS. Hadoop is not capable of transparently processing data stored in these formats. Moreover, with rapidly growing size of scientific datasets, reloading data in another file system or format is not feasible [4].

We present a framework that allows scientific data in different formats to be processed with a MapReduce like API.

Our system is referred to as SciMATE, and is based on the MATE system developed at Ohio State [14], [13]. SciMATE is developed as a customizable system, which can be adapted to support processing on any of the scientific data formats. Particularly, it has a data format adaption API, which can be used to create an instance of the system for processing data in a particular format. Once such an instance has been created, data processing applications developed using a MapReduce-like API can be executed. Applications can be developed assuming a simple logical view of data. Thus, data analysis application developed for our system is independent of the specific data format used for storing the data.

We have demonstrated the functionality of our system by creating instances that can be processing NetCDF and HDF5 formats as well as flat-files. We have also implemented three popular data mining applications and have evaluated their execution with each of the three instances of our system.

II. BACKGROUND

This section provides background information on the MapReduce model and the MATE system, as well as the popular scientific data formats, including NetCDF and HDF5.

A. MapReduce and MATE

MapReduce [7] was proposed by Google for scalable application development for data-centers. With a simple interface of two functions, map and reduce, this model has a great suitability for the parallel implementations of a variety of applications, including search engine support and machine learning [6], [10]. The map function takes a set of input instances and generates a set of corresponding intermediate output (key, value) pairs. The MapReduce library groups together all of the intermediate values associated with the same key and shuffles them to the reduce function. The reduce function, also written by the users, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values.

In recent years, our research group also designed a system we refer to as MapReduce with AlternaTE API (MATE) [14], [13]. The distinctive characteristic of MATE is that it enables the users to explicitly declare a reduction object, and uses this to overcome a key inefficiency in MapReduce, which is of storing and shuffling a large number of key-value pairs.

B. Scientific Data Formats

For efficient storage and I/O on large-scale scientific data, a number of scientific data formats have been proposed,
including CDF (Common Data Format), FITS (Flexible Image Transport System), GRIB (GRid In Binary), HDF and HDF5 (Hierarchical Data Format), HDF-EOS (Earth Observing System extensions to HDF), and NetCDF (Network Common Data Format). Among these scientific data formats, NetCDF and HDF5 seem to be the most popular ones at the current time.

NetCDF [2] comprises software libraries and a self-describing, portable data format, with the goal of supporting creation, access, and sharing of scientific data. It is commonly used in climatology, meteorology, and GIS applications. A NetCDF dataset comprises of dimensions, variables, and attributes, which all have both a name and an ID number by which they are identified.

HDF5 [1] is another widely used data model and file format for storing and managing data. It is designed for flexible and efficient I/O and for high volume and complex data. HDF5 is portable and extensible, allowing applications to evolve in their use of HDF5. HDF5 files are organized in a hierarchical structure, with two primary structures: groups and datasets.

III. SYSTEM DESIGN

In this section, we discuss the design and implementation of the SciMATE system. After giving a brief overview of the system, we describe an API for integrating a new data format into the system.

A. System Overview

SciMATE’s execution is closely based on the precursor system, MATE, which has been described in our earlier publications [14], [13]. The key new feature is the scientific data processing module, which is responsible for partitioning input scientific datasets, loading partitioned data into memory blocks, and possible data restructuring while loading non-contiguous data.

Figure 1 gives an overview of the execution flow of a typical application using SciMATE in a distributed environment. First, given an input scientific dataset, the scientific data processing module will invoke a specific partitioning function, which corresponds to the data format involved. This function divides the input into a number of partitions. The partitioning function needs to be implemented for each specific data format, and it interacts with the corresponding library to help retrieve dataset information such as dataset dimensionality/rank, the length of each dimension, the number of units, and the total size of dataset. Additionally, while performing partitioning, data locality is considered so that each node can load most of the input partitioned data from its own local disk.

Figure 2 shows additional details of the scientific data processing module. Since NetCDF and HDF5 are two of the most popular scientific data formats, we have implemented data processing support for these two formats in the current version of SciMATE. Although SciMATE is able to process large NetCDF/HDF5 datasets, all the nitty-gritty NetCDF/HDF5 APIs details are abstracted within the data processing module and transparent to application developers. Thus, the users are allowed to develop applications on the on-disk scientific datasets as if they are ordinary arrays located in the memory.

The most important component of the scientific data processing module is a block loader, which is scheduled by the runtime to load partitioned data into memory blocks. As Figure 2 illustrates, the block loader is connected with both a data format selector and an access strategy selector so that the module can tackle two options specified by the users: the data format of input datasets and access strategy. The current data format selector provides three system-defined options: NetCDF, HDF5 and flat-file, i.e. binary data. Once the users select one of the three options, the runtime will dynamically bind a corresponding adapter through the data format selector. The launched adapter translates calls to the generic interface of block loader, into calls to the original scientific file format libraries (NetCDF/HDF5) or the standard file I/O library. On the other hand, the current access strategy selector provides two options: full read and partial read, which will be introduced later in Section IV.

B. API for Integrating a New Data Format

Besides supporting NetCDF, HDF5, and flat-files, our system allows for enhanced reusability and extensibility by allowing a new data format adapter to be easily added. Thus, the same data analysis application can be executed even on a dataset which has a different data format. The three system-defined data format adapters can be viewed as instances of a
generic adapter interface. Given a new data format, a third-party adapter can be developed by implementing the block loading prototypes that interact with the corresponding library.

The API set for scientific data processing is summarized in Table I. To integrate a new data format, a customized partition function is required, which computes both the starting location and the size of each partition. The same function then sends these results to assigned computing nodes. While some critical information like the number of nodes can be retrieved using certain MPI functions, other important information is obtained via data-format specific functions. These functions include get_dimensionality, get_dataset_size, get_dimlens and get_unit_num, which help obtain dataset dimensionality/rank, dataset size, the length of each dimension and the number of units, respectively.

Moreover, the API set also allows the use of data-format specific libraries to implement different access strategies, including full read and partial read. The full_read function reads complete data records. The partial_read function specifies how to load a single contiguous subregion. Additionally, partial_read can also be implemented via certain access patterns: the partial_read_by_block function is for the strided pattern which will be discussed later in Section IV, the partial_read_by_column function allows the users to select required columns that are mapped to certain dimensions, regardless of column contiguity, and finally, the partial_read_by_list function serves to retrieve a list of discrete elements in the input datasets.

To show how this API is used for different data formats, we take the full_read function as an example, and demonstrate how NetCDF and HDF5 calls are implemented. For double precision data, the steps involved are: 1) open the file and retrieve the file id by file name; 2) open the dataset and retrieve the dataset id by file id and dataset name; 3) check if the data type is double; 4) retrieve the dataset rank by dataset id; 5) retrieve the length of each dimension; 6) define the data space and memory space with the same rank and the same length of each dimension; 7) load the entire dataset into read buffer; and 8) release the file id and other descriptors if necessary. The NetCDF and HDF5 calls implementing these steps are listed in Table II. Note that because of a slight difference between NetCDF and HDF5 programming models, steps 4-6 can be skipped for the NetCDF adapter. Based on this, we expect that an adapter for any other array-based scientific data format can be implemented in a similar fashion.

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>DESCRIPTIONS OF THE FUNCTIONS IN THE SCIENTIFIC DATA PROCESSING API</th>
</tr>
</thead>
<tbody>
<tr>
<td>int partition (char*, char*, int, size_t**, size_t**)</td>
<td>Partitions the input dataset for each computing node by computing the starting location and size of each partition</td>
</tr>
<tr>
<td>int get_dimensionality (char*, char*)</td>
<td>Retrieves dataset dimensionality/rank. The arguments are file name and dataset name</td>
</tr>
<tr>
<td>size_t get_dataset_size (char*, char*)</td>
<td>Retrieves dataset size. The arguments are file name and dataset name</td>
</tr>
<tr>
<td>int get_dims (char*, char*, size_t*)</td>
<td>Retrieves the length of each dataset dimension. The arguments are file name, dataset name, and dataset dimensionality</td>
</tr>
<tr>
<td>unsigned long get_unit_num (char*, char*)</td>
<td>Retrieves the number of units in the dataset. The arguments are file name and dataset name</td>
</tr>
<tr>
<td>int full_read (char*, char*, void*)</td>
<td>Reads the entire input dataset. The arguments are file name, dataset name, and read buffer</td>
</tr>
<tr>
<td>int partial_read (char*, char*, void*, size_t, size_t*)</td>
<td>Reads a contiguous selection of the input dataset. The arguments are file name, dataset name, read buffer, starting location, and buffer size</td>
</tr>
<tr>
<td>int partial_read_by_block (char*, char*, void*, size_t, size_t, size_t, size_t, size_t, size_t, size_t)</td>
<td>Reads a non-contiguous selection of the input dataset by regular block</td>
</tr>
<tr>
<td>int partial_read_by_columns (char*, char*, void*, size_t, size_t, int*, int*)</td>
<td>Reads a non-contiguous selection of the input dataset by column</td>
</tr>
<tr>
<td>int partial_read_by_list (char*, char*, void*, size_t**, int*)</td>
<td>Reads a non-contiguous selection of the input dataset by specifying a list of discrete elements</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE II</th>
<th>INVOKED CALLS IN NETCDF/HDF5 LIB FOR full_read IMPLEMENTATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step#</td>
<td>Invoked NetCDF Call</td>
</tr>
<tr>
<td>1</td>
<td>nc_open</td>
</tr>
<tr>
<td>2</td>
<td>nc_inq_varid</td>
</tr>
<tr>
<td>3</td>
<td>nc_inq_vartype</td>
</tr>
<tr>
<td>4</td>
<td>(nc_inq_varndims)</td>
</tr>
<tr>
<td>5</td>
<td>(nc_inq_dimlen)</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>nc_get_var_double</td>
</tr>
<tr>
<td>8</td>
<td>nc_close</td>
</tr>
</tbody>
</table>

IV. SYSTEM OPTIMIZATION

In this section, we discuss optimizations that have been implemented in SciMA TE system to help improve the performance of reading data.

A. Data Access Strategies and Patterns

In our framework, we have considered two different types of access strategies: Full Read and Partial Read. In terms of the partial read strategy, it can be implemented by different access patterns.

**Full Read:** Full read involves loading the entire original dataset for processing. It is the simplest case during the block loading, where all the block data is required for computations. Therefore, no subsetting or merge operations are required.

445
Partial Read: Partial read refers to loading non-contiguous subsets of input dataset. The need for such a read operation can arise because of many reasons. First, not all data records may be analyzed by a particular application. Similarly, not all dimensions may be needed. There is also a possibility that the data elements needed to construct a single record may not be stored contiguously. The key issue with a partial read is that a simple read of all required data elements may lead to poor performance, due to frequent non-contiguous and/or small I/O requests. Such non-contiguous and small reads may be replaced by contiguous or large reads. SciMATE performs this replacement after input data is partitioned and partitions are assigned to each node, but before each partition is loaded into a memory block. Note that all the details of such restructuring are transparent to the users.

To perform partial read, the users only need to specify the layout of a high-level view of the required data subsets, which usually can be described by a certain high-level access pattern. In other words, partial read is performed based on a specific high-level access pattern explicitly determined by the users beforehand. The different access patterns we support are as follows:

- **Strided Pattern**: If the required non-contiguous regions have a regular pattern, a strided pattern can be specified, via starting dimension/record, the access stride vector within each dimension/record subset, the number of dimension/record subsets, as well as the number of dimensions/records within each subset. All the details related to the low-level data layout are mapped from the high-level view of datasets related to record/dimension and hence hidden from the users. The strided patterns can be categorized into a simple-strided pattern and a nested-strided pattern. Simple-strided pattern refers to a series of I/O requests where each request is for a segment that contains the same number of elements, and each segment starting location is incremented by the same factor in a single dimension/record growth direction. Nested-strided pattern is more complex. It is composed of strided segments separated by regular strides, i.e., a nested-strided pattern is defined by two or more strides in different dimensions and/or record growth direction, instead of one stride as in the simple-strided access pattern.

- **Column Pattern**: If the required data consists of a set of arbitrary columns, a column pattern can be specified, using that set of columns. This is because that in many HPC data analysis applications, only a subset of dataset dimensions may be needed for a certain type of data analysis, and SciMATE can map each dimension to a single column during computation.

- **Discrete Point Pattern**: If the required regions cannot be described by a regular pattern, a discrete point pattern can be specified, using a vector of element locations and the number of required elements.

B. Optimizing Different Types of Accesses

The following optimizations are performed in our framework. For strided patterns, appropriate strided reads are invoked from NetCDF/HDF5 libraries. We appropriately map the high-level parameters such as dimension/record index, to the low-level layout like column/row index. The optimization for column pattern is described in details below. For discrete point pattern, no optimization has been implemented currently.

When only a subset of columns are needed, the first question is whether to use full read or partial read. It turns out that when a contiguous subset of columns are needed, partial read can be beneficial. However, when a non-contiguous subset is needed, the performance can depend upon the particular layout being used.

It also turns out that, NetCDF has a better tolerance for column non-contiguity than HDF5. If the loaded columns are composed of multiple discrete regions, the performance of HDF5 partial read is likely to get worse. In comparison, the loading time of any NetCDF partial read, including reading a number of discrete column regions, will take at most twice as long as the full read (while reading the same amount of data). Thus, if a small subset of data is to be loaded, partial read is advantageous for NetCDF.

The second issue is to optimize partial read with column accesses. We choose from two approaches for reading a set of columns, which are fixed-size column read and contiguous column read.

**Fixed-size Column Read**: Fixed-size column read is a naive strategy. This strategy allows only a fixed number of columns to be read at a time, and the system default value of this fixed number is 1. Thus, the data loading amount for each time will be the same, resulting in a balanced workload distribution.

**Contiguous Column Read**: Contiguous column read strategy reads a contiguous column set at a time. Instead of a fixed number, the number of loaded columns in this strategy is determined by column contiguity at runtime. Contiguous column read takes advantage of the column contiguity so that the number of column reads equals the number of contiguous regions. This strategy can minimize the number of reads and significantly reduce the overhead resulting from frequent small I/O requests.

The choice of strategy in our system is guided by the observations from a number of experiments we performed. Here, we use k-means clustering to illustrate the influence of column contiguity in the performance of column read. We executed k-means application on 8 GB 10-dimensional NetCDF and HDF5 datasets. We considered several different possible application scenarios, where different numbers of
contiguous/discrete columns were read from the datasets. The results are shown in Figure 3. In the horizontal axis, different scenarios are shown. Particularly, “1” means only 1 single column was read, “2C” indicates that 2 contiguous columns were required, “3C” stands for 3 contiguous columns, “2D” denotes 2 discrete columns to be read, “2C+2C” represents 2 discrete column sets, where each consists of 2 contiguous columns, and finally, “3D” refers to 3 discrete columns.

First, we observed that the data loading performance could be categorized into 3 levels: with 1, 2C, and 3C being the first level, 2D and 2C+2C being the second level, and 3D being the third level. To simplify the analysis, we ignored the influence brought by the number of loaded columns, especially between “1” and “2C” in the experiments on the HDF5 datasets. Although the numbers of loaded columns differed in the first 3 scenarios, their data loading time were close, because the column reads were performed only once. Similarly, it took 2 column reads to complete data loading in “2D” and “2C+2C”, so the data loading time was longer than the first three, while being shorter than “3D”. Overall, we can conclude that the performance primarily depends upon the number of separate reads, and not on the volume of data read.

Second, the results showed NetCDF had a better column non-contiguity tolerance than HDF5. For instance, sometimes reading 3 discrete HDF5 columns could be quite expensive, even over 3 times slower than reading a single column, while reading discrete NetCDF columns caused a relatively small overhead. Additionally, we also observed that the strides among all the discrete regions also had some impact on the performance.

To summarize, we use the observations made from our experiments to guide the selection of appropriate column access approach, i.e., contiguous column read. Note, however, we expect a future development to improve the attractiveness of fixed-size column read. Once NetCDF and/or HDF5 libraries allow parallel data accesses from multiple threads in a single process, the performance of fixed-size column read may improve, since a better load balance can be achieved.

V. EXPERIMENTAL RESULTS

In this section, we evaluate the functionality and scalability of SciMATE system on a cluster of multi-core machines. We show the performance achieved with three different data formats: NetCDF, HDF5, and flat-file. Last, we present the results from optimization evaluation, including comparing the performance of partial read against full read, as well as the performance of fixed-size column read against contiguous column read.

A. Applications and Environmental Setup

We used three popular data analysis algorithms, which are k-means clustering (k-means), principal components analysis (PCA), and k-nearest neighbor search (kNN). K-means is one of the most popular data mining algorithms. PCA is a popular dimensionality reduction method that was developed by Pearson in 1901. Its goal is to compute the mean vector and the covariance matrix for a set of data points that are represented by a matrix. KNN is a type of instance-based learning method [3] for classifying objects based on closest training examples in the feature space, which has been successfully applied in diverse domains like protein function prediction and image de-noising [16].

The datasets and the application parameters we used are as follows. The sizes of all the datasets for the evaluation in Section V-B and V-C were 16 GB and 8 GB, respectively. With k-means, we used 10-dimensional points, and the number of clusters, k, was set to be 100. For both PCA and kNN, the points were 1000-dimensional. In kNN, the number of clusters, k, was set to 100, and the number of nearest neighbors, was set to be 30.

Our experiments were conducted on a cluster of multi-core machines. The system uses AMD Opteron(TM) Processor 8218 with 4 dual-core CPUs (8 cores in all). The clock frequency of each core is 2.6 GHz and the system has an 8 GB main memory. We have used up to 128 cores (16 nodes) for our study. The NetCDF version is 4.1.1, and the HDF5 version is 1.8.7.

B. Functionality Evaluation

In this section, we evaluate the data processing performance and the data loading performance of SciMATE by executing the three data analysis algorithms in a way that involved full reads. For each of the applications, we use 16 GB datasets that are stored in three different data formats: NetCDF, HDF5 and flat-file (FLAT).

Data Processing Times: Figures 4 through 6 show the comparison results for three applications as we scale the number of threads used on a single node. Figures 7 through 9 show the results as we scale the number of nodes used with 8
threads on each node. We can see that the data processing performance on these three different data formats is quite similar, and moreover, scales with both number of threads and nodes. This shows that SciMATE is capable of supporting different data formats, and scaling the performance of data analysis applications.

**Data Loading Times:** Figures 10 and 11 show the data loading times for k-means and PCA, respectively. The results indicate that loading NetCDF/HDF5 datasets is faster than loading flat-file datasets, demonstrating up to 62.6% throughput improvement. We believe this is because the highly structured nature of scientific data can facilitate the parallel I/O in distributed environments. Compared with the unstructured flat-file datasets, the data layout of NetCDF/HDF5 datasets is described in the header information, so certain unnecessary scan for each partition or split can be avoided. Moreover, the results also show that loading NetCDF datasets is slightly faster than loading HDF5 datasets. This is related to the performance of the libraries, and mostly unrelated to SciMATE. Specifically, NetCDF has a linear data layout, i.e. the data arrays are either stored in contiguous space (with a predefined order) or interleaved with a regular pattern. Such regularity in the layout helps improve MPI-IO performance. In comparison, HDF5 chooses hierarchical data layout, which is more flexible but leads to more irregular accesses. Particularly, HDF5 uses dataspace and hyperslabs to map and transfer data between the memory and the file system. This causes buffer packing/unpacking in a recursive way, and results in performance loss to a certain degree. Another factor is that NetCDF has a smaller header I/O overhead than HDF5. In NetCDF, only one header is stored, while in HDF5 header metadata is dispersed in separate header blocks for each object.

**C. Optimization Experiments**

In this section, we evaluate two of the optimizations proposed in Section IV, i.e. use of partial reads, and use of contiguous column reads. Our third optimization, which is the use of APIs for strided patterns, is not evaluated here, since it is directly supported in NetCDF and HDF5.

**Comparing Partial Read with Full Read:** To evaluate the benefits of partial read, we designed the following experiment. We executed PCA application with 4 nodes and 1 thread per node, on both NetCDF and HDF5 datasets (8 GB each). We considered the following cases: 200, 400, 600, and 800 contiguous columns were required by the data analysis applications, when the dataset had a total of 1000 columns. For these four cases, we compared the performance of partial read against full read.

The results reported in Figure 12 show that if the number of contiguous columns to be loaded is no more than half of the total number, partial read can outperform full read (shown as right-most charts, i.e. when all 1000 columns are read). Otherwise, full read may be a better option. This is because that both NetCDF and HDF5 datasets have row-major ordering, and therefore, the data corresponding to a subset of contiguous columns actually is not stored contiguously in the dataset files. Thus, a partial read involves multiple I/O requests for loading segments from disk files to the read buffer. Therefore, unless the volume of data to be read is quite small,
the overhead of separate I/O requests slows down the read operation.

Moreover, compared with the full read, the partial read only loads required data into memory, leading to a better data locality and lower memory consumption. Therefore, sometimes partial read can also accelerate the later reduction process because of the better data locality.

**Comparing Fixed-Size Column Read with Contiguous Column Read:** We compare the performance of fixed-size column read against that of contiguous column read, by executing kNN application with 2 threads on one node, on both NetCDF and HDF5 8 GB datasets. In the experiments we designed, the SciMATE reads three contiguous column sets with the same number of columns in the 1000-dimensional datasets. Each contiguous column set contains 100, 200, and 300 columns respectively in three different comparison experiments.

Figure 13 and Figure 14 show that contiguous column read outperforms fixed-size column read, even the number of contiguous regions is quite limited. Contiguous column read can gain a speedup of about 3.0 on NetCDF datasets, and a speedup of about 30.0 on HDF5 datasets.

**VI. RELATED WORK**

The topics of data-intensive computing and MapReduce have attracted a lot of attention in the past few years. Because of the large volume of work in this area, we will restrict ourselves to work specific to scientific data processing.

At Indiana, CGL-MapReduce [8] has been developed as an extension to MapReduce. It uses streaming for all the communications, and thus improves the performance to some extent. More recently, a runtime system, Twister [9], has been developed to optimize iterative applications. Somewhat similar optimizations are already implemented in MATE.

Integrating MapReduce systems with scientific data has been a topic of much interest recently [11], [20], [21], [15], [5], [12], as also summarized by Buck et al. [5]. The Kepler-Hadoop project [20], as the name suggests, combines MapReduce processing with Kepler, a scientific workflow platform. Its limitation is that it cannot support processing of data in different scientific formats. In another system [21] NetCDF processing using Hadoop is allowed, but the data has to be converted into text, causing high overheads. SciHadoop [5] integrates Hadoop with NetCDF library support to allow processing of NetCDF data with MapReduce API. We expect to compare the performance of our system with SciHadoop in the future. We will like to note, however, that our underlying system, MATE, has been shown to outperform Hadoop by a large factor in previous studies [14], [13]. Moreover, their current implementation is restricted to handling NetCDF. The authors plan on extending their work to support HDF5 (and list several challenges in accomplishing this), whereas our current implementation already handles HDF5. MARP [17] is another MapReduce-based framework to support HPC analytical applications, which optimizes for certain access patterns. However, it cannot directly operate on scientific data formats.

Research has been conducted towards processing and querying data in formats like NetCDF, with other APIs (i.e. besides MapReduce) as well. The NetCDF Operator (NCO) library has been extended to support parallel processing of NetCDF.
files [19]. However, a specialized API has to be used which is format-specific. To enable querying of large-scale scientific data, FastQuery framework has been developed [12], but their focus is querying (subsetting) of data rather than processing of data.

VII. CONCLUSIONS

With growing importance of analysis of large-scale data in various scientific areas, two trends have become clear. First, data collected from instruments and simulations is going to be stored in certain specialized data formats, depending upon the specific domain (e.g., climate data is almost always stored in NetCDF). Second, with the rapid increase in dataset sizes, it is not feasible to reformat the data, or load it in a different file system or a database. Thus, we need an ability to develop scalable data analysis applications that can work with the popular scientific data formats.

This paper has presented such a framework. Our system, SciMATE, has an API which allows it to be customized for any new scientific data format. We have also developed several optimizations, particularly driven by the observations that most scientific datasets have a large number of attributes, but only a subset of those are needed by any particular application. We have created three instances of the system, for NetCDF, HDF5, and flat-files, and have evaluated them using three popular data analysis tasks. Our results show that each of the three instances of the system scales well on a multi-core cluster.

Acknowledgements

This work is supported by NSF grants CCF-0833101 and IIS-0916196.

REFERENCES


Fig. 14. KNN: Comparison between Fixed-size Column Read and Contiguous Column Read on HDF5 Datasets