MRAP: A Novel MapReduce-based Framework to Support HPC Analytics Applications with Access Patterns

Saba Sehrish[1], Grant Mackey[1], Jun Wang[1], and John Bent[2]
[1]University of Central Florida
ssehrish, gmackey, jwang@eecs.ucf.edu
[2]Los Alamos National Laboratory
johnbent@lanl.gov

ABSTRACT

Due to the explosive growth in the size of scientific data sets, data-intensive computing is an emerging trend in computational science. Many application scientists are looking to integrate data-intensive computing into computational-intensive High Performance Computing facilities, particularly for data analytics. We have observed several scientific applications which must migrate their data from an HPC storage system to a data-intensive one. There is a gap between the data semantics of HPC storage and data-intensive system, hence, once migrated, the data must be further refined and reorganized. This reorganization requires at least two complete scans through the data set and then at least one MapReduce program to prepare the data before analyzing it. Running multiple MapReduce phases causes significant overhead for the application, in the form of excessive I/O operations. For every MapReduce application that must be run in order to complete the desired data analysis, a distributed read and write operation on the file system must be performed. Our contribution is to extend Map-Reduce to eliminate the multiple scans and also reduce the number of pre-processing MapReduce programs. We have added additional expressiveness to the MapReduce language to allow users to specify the logical semantics of their data such that 1) the data can be analyzed without running multiple data pre-processing MapReduce programs, and 2) the data can be simultaneously reorganized as it is migrated to the data-intensive file system. Using our augmented MapReduce system, MapReduce with Access Patterns (MRAP), we have demonstrated up to 33% throughput improvement in one real application, and up to 70% in an I/O kernel of another application.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed Programming;
D.1.3 [**Concurrent Programming**]: Parallel Programming;
H.3.4 [**Systems and Software**]: Distributed Systems;

General Terms

I/O Performance of HPC Applications, Large-scale data processing systems

Keywords

HPC Analytics Applications, HPC Data Access Patterns, MapReduce

1. INTRODUCTION

Today's cutting-edge research deals with the increasing volume and complexity of data produced by ultra-scale simulations, high resolution scientific equipment and experiments. These datasets are stored using parallel and distributed file systems and are frequently retrieved for analytics applications. There are two considerations regarding these datasets. First, the scale of these datasets [6, 33] that affects the way they are stored (e.g. metadata management, indexing, file block sizes, etc). Second, in addition to being extremely large, these datasets are also immensely complex. They are capable of representing systems with high levels of dimensionality and various parameters that include length, time, temperature, etc. For example, the Solenoidal Tracker at the Relativistic Heavy Ion Collider (STAR; RHIC) experiment explores nuclear matter under extreme conditions and can collect seventy million pixels of information one hundred times per second [10]. Such extraordinarily rich data presents researchers with many challenges in representing, managing and processing (analyzing) it. Many data processing frameworks coupled with distributed and parallel file system have emerged in recent years to cope with these datasets [3, 16, 17, 20].

However, the raw data obtained from the simulations/sensors needs to be stored to data-intensive file systems in a format useful for the subsequent analytics applications. There is an information gap because current HPC applications write data to these new file systems using their own file semantics, unaware of how the new file systems store data, which generate unoptimized writes to the file system. In HPC analytics, this information gap becomes critical because source of data and commodity-based systems do not have the same data semantics. For example, in many simulations based applications, data sets are generated using MPI datatypes [21] and self describing data formats like netCDF [7, 25] and HDF5 [4, 8]. However, scientists are using frameworks like MapReduce for analysis [18, 24, 26, 27, 30] and require datasets to be copied to the accompanied distributed and

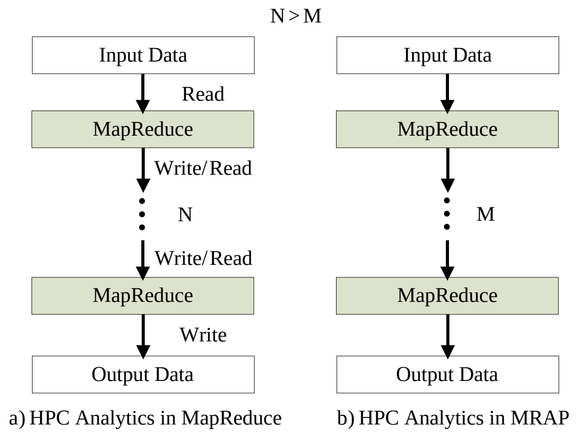


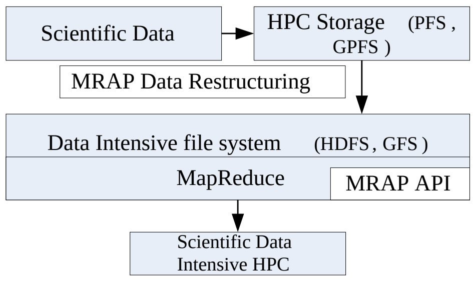
Figure 1: Steps performed in writing HPC analytics using both MapReduce and MRAP.

parallel file system e.g. HDFS [13]. The challenge is to identify the best way to retrieve data from the file system following the original semantics of the HPC data.

One way to approach this problem is to use a high level programming abstraction for specifying the semantics and bridging the gap between the way data was written and the way it will be accessed. Currently, the way to access data in HDFS like file systems is to use the MapReduce programing abstraction. Because MapReduce is not designed for semantics based HPC analytics, some of the existing analytics applications use multiple MapReduce programs to specify and analyze data [24, 30]. We show the steps performed in writing these HPC analytics applications using MapReduce in Figure 1 a). In Figure 1 a), N MapReduce phases are being used; the first MapReduce program is used to filter the original data set. If data access pattern is complex, then a second MapReduce program will perform another step on the data to extract another dataset. Otherwise, it will perform the first step of analysis and generate a new data set. Similarly, depending on the data access pattern and the analysis algorithm, multiple phases of MapReduce are utilized.

For example, consider an application, which needs to merge different data sets followed by extracting subsets of that data. Two MapReduce programs are used to implement this access pattern. That is, the first program will merge the data set and the second will extract the subsets. The overhead of this approach is quantified as 1) the effort to transform the data patterns in MapReduce programs, 2) number of lines of code required for MapReduce data pre-processing and 3) the performance penalties because of reading excessive datasets from disk in each MapReduce program.

We propose a framework based on MapReduce, which is capable of understanding data semantics, simplify the writing of analytics applications and potentially improve performance by reducing MapReduce phases. Our aim in this project is to utilize the scalability and fault tolerance benefits for MapReduce and combine them with scientific access patterns. *Our framework, called **MapReduce with Access Patterns (MRAP)**, is a unique combination of the data access semantics and the programming framework (MapReduce), which is used in implementing HPC analytics applications.* We have considered two different types of access patterns in the MRAP framework. The first pattern is for the matching, or similar analysis operations where the input data is required from two different data sets. These applications access data in smaller contiguous regions. The second pattern is for the other types of analysis that uses data in multiple smaller non-contiguous regions. MRAP framework consists of two components to handle these two patterns; MRAP API and MRAP data restructuring.



Scientific Data
HPC Storage (PFS , GPFS)
MRAP Data Restructuring
Data Intensive file system (HDFS , GFS)
MapReduce
MRAP API
Scientific Data Intensive HPC

Figure 2: High-Level System View with MRAP

MRAP API is used to specify both access patterns. It is flexible enough to specify the data pre-processing and analytics in fewer MapReduce programs than currently possible with traditional MapReduce as shown in the Figure 1 b). Figure 1 shows that MRAP is utilizing M phases, and $M < N$, where N is the number of corresponding phases in a MapReduce based implementation. With MRAP, $M < N$ is possible because it allows the users to describe data semantics (various access patterns based on different data formats) in a single MRAP application. As mentioned in the previous example, two MapReduce programs are used to describe an access pattern which requires merge and subset operations on data before analyzing it. In MRAP, only one MapReduce program is required, because data is read in required merge pattern in the Map phase, and subsets are extracted in the reduce phase.

MRAP data restructuring reorganizes data with the copy operation to mitigate the performance penalties resulting from non-contiguous small I/O requests (second access pattern). Our prototype of MRAP framework includes basic implementation of functions that allow users to specify data semantics and data restructuring to improve the performance of our framework. Figure 2 shows the high level system view with MRAP. Our results with a real application in bioinformatics and an I/O kernel in astrophysics, show up to 33% performance improvement by using MRAP API, and up to 70% performance gain by using data restructuring.

This paper is organized as follows: Section 2 discusses the motivation for our MRAP approach. Section 3 describes the overall framework and its design, and how API effects the data layout and how optimization like data restructuring is used in the framework. Results are presented and analyzed in Section 4. Section 5 describes the related work in large scale data processing, access patterns in scientific applications and optimization on these patterns. Conclusion and future work is in Section 6.

2. MOTIVATION FOR DEVELOPING MRAP

In this section we describe the motivation for developing MRAP. Data for analysis comes in all types of formats, file semantics, and comes from many various sources; whether it be weather data to high energy physics simulations. Within these patterns, we also need to provide for efficient file accesses. Current approaches show that in order to implement these access patterns, a series of MapReduce programs is utilized [24, 30], also shown in the Figure 1.

In a MapReduce program, the map phase always reads data in contiguous chunks, and generates data in the form of (key, value) pairs for the reduce phase. The reduce phase then combines all the *values* with the same *keys* to output the result. This approach works well when the order and sequence of inputs do not affect the output. On the other hand, there are algorithms that require data to be accessed in a given pattern and sequence. For example, in some image pattern/template matching algorithm, the input of the function must contain a part of the original image and a part of the reference image. Using MapReduce for this particular type of analysis would require one MapReduce program to read the inputs from different image files and then combine them in the reduce phase for further processing. A second MapReduce program would then analyze the results of the pattern matching. Similar behavior has been observed in two other application as shown in the Figure 3 and Figure 7 a). Figure 3 describes a distributed *Friends-of-Friends* algorithm; the four phases are MapReduce programs [24]. Figure 7 a) is discussed in the Section 4. Both these figures show the multi-stage MapReduce applications developed for scientific analytics.

Using this multi-stage approach, all the intermediate MapReduce programs write data back to the file system while subsequent applications read that outputted data as input for their task. These excessive read/write cycles impose performance penalties and can be avoided if initial data is read more flexibly as compared with traditional approach of contiguous chunks read. Our proposed effort, MRAP API, is designed to address these performance penalties and to provide an interface that allows these access patterns to be specified in fewer MapReduce phases. Hence, the goal of the MRAP API is to deliver greater I/O performance than the traditional MapReduce framework can provide to scientific analytics applications.

Additionally, some scientific access patterns generally result in accessing multiple non-contiguous small regions per task, rather than the large, contiguous data accesses seen in MapReduce. Due to the mismatch in sizes of distributed file system (DFS) blocks/chunks and the logical file requests of these applications, small I/O problem arises. The small I/O requests from various access patterns impact the performance by accessing excessive amount of data, when implemented using MapReduce. The default distributed file system chunk used in current setup is either 64/128 MB. Most of the scientific applications store data with each value comprising of a few KBs. Each small I/O region (a few KBs) in the file may map to large chunks (64/128 MB) in the file system. If each of the requested small I/O region is 64 KB, then a 64 MB chunk will be retrieved to process only 64 KB making it extremely expensive.

A possible approach can be to decrease the chunk size to reduce the amount of extra data accessed. These smaller chunks increase the size of metadata [5]. Also, the small I/O accesses also result in a large number of I/O requests sent to the file system. In MapReduce framework, smaller chunks and large number of I/O requests become extremely challenging because there is a single metadata server (*NameNode*) that handles all the requests. These small I/O requires efficient data layout mechanisms because the patterns can be used to mitigate performance impact that would otherwise arise. Hence, providing semantics for various access patterns which generate small I/O requires optimizations at both the programming abstraction and the file system levels.

3. DESIGN OF MRAP

The MRAP framework consists of two components; 1) MRAP API, that is provided to eliminate the multiple MapReduce phases used to specify data access patterns, and 2) MRAP data restructuring, that is provided to further improve the performance of the access patterns with small I/O problem. Before we describe each component of the MRAP framework, it is important to understand the steps that existing HPC MapReduce applications perform in order to complete an analysis job. These steps are listed as follows:

- Copy data from external resource (remote storage, sensors, etc) to HDFS/similar data-intensive file system with MapReduce for data processing.
- Write at least one data pre-processing application (in MapReduce) to prepare data for analysis. The number of MapReduce applications depends on the complexity of initial access pattern.
    - This data preparation can be a conversion from raw data to scientific data format or in general filtering of formatted data.
- Write at least one MapReduce application to analyze the prepared datasets. The number of MapReduce applications for analysis varies with the number of steps in the analytics algorithm.

As compared with this existing MapReduce setup, our MRAP framework will allow the following steps:

- Copy data from external resource to HDFS/similar file system (perform MRAP data restructuring if specified by the user).
- Write an MRAP application to specify the pattern and subsequent analysis operation.
    - If data was restructured at the copy time, then map/reduce phases will be written only for the analysis operation.

We discuss the two components, i.e. MRAP API and MRAP data restructuring in detail in the next subsections.

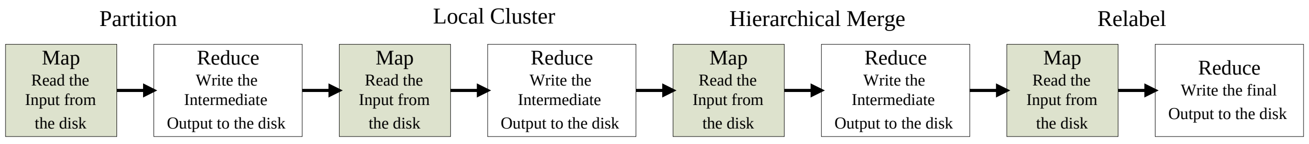

Partition
Local Cluster
Hierarchical Merge
Relabel
Map Read the Input from the disk
Reduce Write the Intermediate Output to the disk
Reduce Write the final Output to the disk

Figure 3: Overview of the distributed FoF algorithm using 4 MapReduce phases [24].

3.1 MRAP API

The purpose of adding this data awareness functionality is to reduce the number of MapReduce programs that are used to specify HPC data access patterns. We provide an interface for two types of data access patterns; 1) applications that perform match operations on the data sets, and 2) applications that perform other types of analysis by reading data non-contiguously in the files. At least one MapReduce program is used to implement either 1 or 2. *Note: There are some complex access patterns which consist of different combinations of both 1 and 2.* We first briefly describe how MapReduce programs work, and why they require more programming efforts when it comes to HPC access patterns.

A MapReduce program consists of two phases: a Map and a Reduce. The inputs and outputs to these phases are defined in the form of a **key** and a **value** pair. Each map task is assigned a split specified in *InputSplit* to perform data processing. A *FileSplit* and a *MultiFileSplit* are the two implementations of the interface *InputSplit*. The function *computeSplitSize* calculates the split size to be assigned to each map task. This approach guarantees that each map task will get a contiguous chunk of data. In the example with merge and subset operations on data sets, following two MapReduce programs will be used.

- MapReduce1:
  `(inputA, inputB, functionA, outAB)`
- functionA (merge)
  `map1(inputA, inputB, splitsizeA, splitsizeB, merge, outputAB)`
  where merge has different criteria, e.g. merge on exact or partial match, 1-1 match of inputA and inputB, 1-all match of inputA and inputB, etc.
  `reduce1(outputAB, sizeAB, merge_all, outAB)`
- MapReduce2:
  `(outAB, functionB, outC)`
- functionB (subset)
  `map2(outAB, splitsizeAB,subset)`
  where subset describes the offset/length pair for each split of size splitsizeAB.
  `reduce2(subset, outC)`

We now describe MRAP API, classes and functions, and how it allows the user to minimize the number of MapReduce phases for data pre-processing. Then, we describe the provided user templates that are pre-written MRAP programs. The user templates are useful in the cases when the access patterns are very regular and can be specified by using different parameters in a configuration file *e.g, a vector access pattern can be described by the size of data set, size of a vector, number of elements to skip between two vectors, and number of vectors in the data set.*

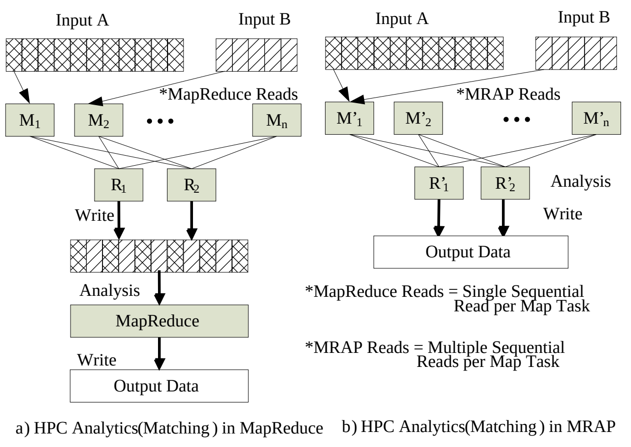
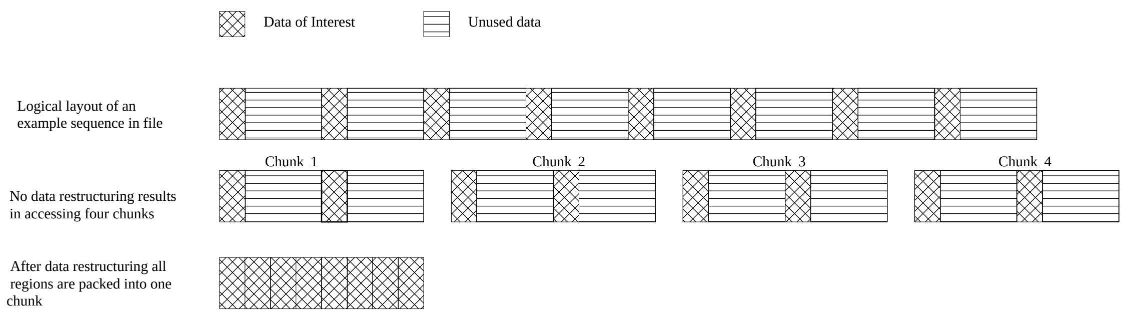

Input A
Input B
*MapReduce Reads
M1
M2
Mn
R1
R2
Write
Analysis
MapReduce
Write
Output Data
a) HPC Analytics(Matching) in MapReduce
Input A
Input B
*MRAP Reads
M'1
M'2
M'n
R'1
R'2
Analysis
Write
Output Data
*MapReduce Reads = Single Sequential Read per Map Task
*MRAP Reads = Multiple Sequential Reads per Map Task
b) HPC Analytics(Matching) in MRAP

Figure 4: A detailed View of comparing MapReduce and MRAP for applications that perform matching.

In MRAP, we provide a set of two classes that implement customized InputSplits for the two aforementioned HPC access patterns. By using these classes, users can read data in one of these two specified patterns in the map phase and continue with the analysis in the subsequent reduce phase. The first pattern for the applications that perform matching and similar analysis is defined by $(inputA, inputB, splitSizeA, splitSizeB, function)$. $inputA/inputB$ represents both single or multi-file inputs. Each input can have a different split size, and then the *function* describes the sequence of operations to be performed on these two data sets. The new *SequenceMatching* class implements *getSplits()*, such that each map task reads the specified *splitSizeA* from *inputA* and *splitSizeB* from *inputB*. This way of splitting data saves one MapReduce data pre-processing phase for the applications with this particular behavior as shown in the Figure 4.

The second access pattern is a non-contiguous access, where each map task needs to process multiple small non-contiguous regions. This pattern is defined as $(inputA, (list\_of\_offsets/maptaks, list\_of\_lengths/maptask), function)$. These patterns are more complex to implement, because these patterns are defined by multiple offset/length pairs. Each map task can either have the same non-contiguous pattern, or different pattern. Current abstraction for *getSplits()* either return an offset/length pair or a list of filenames/lengths to be processed by each map task. The new *AccessInputFormat* class with our new *getSplits()* method implements the above-mentioned functionality to the MapReduce API.



Data of Interest
Unused data
Logical layout of an example sequence in file
Chunk 1
Chunk 2
Chunk 3
Chunk 4
No data restructuring results in accessing four chunks
After data restructuring all regions are packed into one chunk

Figure 5: An example showing the reduction in number of chunks per map task after data restructuring

These patterns are generally cumbersome to describe using MapReduce existing split methods because the existing splits are contiguous, and require the pattern to be transformed to the MapReduce (key, value) formulation.

The above mentioned access pattern with merge and subset operations, is specified in one MRAP program, as follows:

- `(inputA, inputB, functionAB, outC)`
- functionAB (merge, subset)
  `map1(inputA, inputB, splitsizeA, splitsizeB, merge, outAB)`
  Read data from all sources in the required merge pattern
  `reduce1(outAB, sizeAB, subset, outC)` *Extract data of interest from the merged data*

We now explain the user templates. User templates are prewritten MRAP programs using new input split classes that read data according to the pattern specified in configuration file. We provide template programs for reading the astrophysics data for halo finding, and general strided access patterns. In the configuration file (see Appendix), we require type of application, e.g. any MPI simulation, or astrophysics to be specified. Each of these different categories generate data in different formats and access data in different patterns. By providing a supported application type, MRAP can generate a template configuration file to the user.

We give a few examples of the access patterns that can be described in a configuration file. A *vector based* access pattern is defined as if each request having the same number of bytes (i.e. a vector) and is interleaved by a constant or variable number of bytes (i.e. a stride) and number of vectors (i.e. a count). A slightly complex vector pattern is a *nested* pattern, that is similar to a vector access pattern. However, rather than being composed of simple requests separated by regular strides in the file, it is composed of *strided segments* separated by regular strides in the file. That is, a nested pattern is defined by two or more strides instead of one as in a vector access pattern. A *tiled* access patterns is seen in datasets that are multidimensional, and are described by number of dimensions, number of tiles/arrays in each dimension, size of each tile, size of elements in each tile. An *indexed* access pattern is more complex, it is not a regular pattern and describe a pattern using a list of offsets and lengths. There are many more patterns used in HPC applications, but we have included the aforementioned patterns in this version of MRAP.

3.2 MRAP Data Restructuring

MRAP data restructuring is provided to improve the performance of access patterns which access data non-contiguously in small regions to perform analysis. Currently, there is a little support for connectivity of HDFS to resources that generate scientific data. In most cases, data must be copied to a different storage resource and then moved to HDFS. Hence, a copy operation is the first step performed to make data available to MapReduce applications. We utilize this copy phase to reorganize the data layout and convert small I/O accesses to large by packing small (non-contiguous) regions into large DFS chunks (also shown in the Figure 5). We implement data restructuring as an MRAP copy operation that reads data, reorganizes it and then writes it to the HDFS. It reorganizes the datasets based on user specification when data is copied to HDFS (for the subsequent MapReduce applications). The purpose of this data restructuring when copying data is to improve small I/O performance.

The MRAP copy operation is performed in one of two ways. The first method copies data from remote storage to the HDFS along with a configuration file, the second method does not have a configuration file. For the first method, in order to reorganize datasets, MRAP copy expects a configuration file, defined by the user, describing the logical layout of data to be copied. In this operation, the file to be transferred to HDFS and the configuration file are submitted to MRAP copy. As the file is written to HDFS, MRAP restructures the data into the new format defined by the user. When the copy operation is completed, the configuration file is stored with the restructured data.

This configuration file is stored with the file because of a very important sub-case of the MRAP copy operation, on-the-fly data restructuring during an MRAP application. In this case, a user runs an MRAP application on a file that was previously restructured by the MRAP copy function. In this job submission, another configuration file is submitted to define how the logical structure of the data in said file should look before the current MRAP application can begin. As shown in Figure 6, the configuration file stored with data during the initial restructuring is compared with the configuration file submitted with MRAP application. If the two configuration files match, that is, the logical data lay-

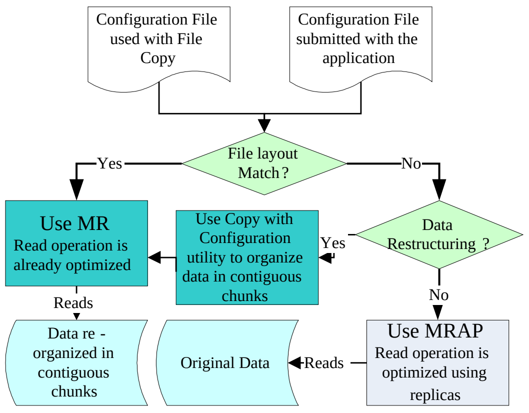

Configuration File used with File Copy
Configuration File submitted with the application
File layout Match?
Yes
No
Use MR
Read operation is already optimized
Use Copy with Configuration utility to organize data in contiguous chunks
Yes
Data Restructuring ?
No
Reads
Data re - organized in contiguous chunks
Original Data
Reads
Use MRAP
Read operation is optimized using replicas

Figure 6: Flow of operations with data restructuring

out of the stored file matches what the MRAP application is expecting, then the MRAP operation begins. Else, data restructuring occurs again and the MRAP application runs once the file is restructured.

In the case of the second copy method, data is copied to HDFS from remote storage without any file modification. This option would be used when the user wants to maintain the original format in which the data was written. Hence, option two in MRAP will be performed as a standard HDFS copy. As discussed in the paragraph above, this option is amenable to use if the file in question is constantly being restructured. The approach used to optimize file access for this case is discussed later in the section.

As mentioned earlier that MRAP copy operation performs data restructuring, we now explain the data restructuring algorithm. The data restructuring converts small non-contiguous regions to large contiguous regions, and can be formulated as a *bin packing* problem where different smaller objects are packed in a minimal number of larger bins.
Definition: Given a set of items with sizes $s_1, \dots, s_n$, pack them into the fewest number of bins possible, where each bin is of size V.
This problem is a *combinatorial NP-hard* and there are many proposed heuristics to solve this. In data restructuring each item from bin packing problem corresponds to a smaller non-contiguous region, whereas each bin corresponds to a DFS chunk of size V. We use First-fit algorithm to pack the smaller regions into chunks.

The time to perform data restructuring based on Algorithm 1 is determined by the following: number of regions in the access patterns that are needed to be combined into chunks is m, and size of each region is s_i. time to access each region T_{readS}, number of chunks after restructuring p where size of each chunk is V, time to write one region to the chunk is T_{chunk} and time to update metadata for the new chunk is T_{meta}. The time to perform data restructuring will be
$T_{dr} = (m \times T_{readS}) + (p \times (V/s_i) \times T_{chunk}) + (p \times T_{meta})$.
We can also determine the execution time of the application with M tasks, that will involve the time to read any access pattern $(M \times m \times T_{readS})$ and processing time T_p,
$T_{comp} = T_p + (M \times m \times T_{readS})$.

Algorithm 1 Data Restructuring Algorithm

Input: A set U which consist of all smaller region size required by a map task in a MapReduce Application, $U = \{s_1, s_2, ..., s_m\}$, where s_i corresponds to size of i^{th} region, and m is the number of non-contiguous regions requested the task. A set C of empty chunks $C = \{c_1, c_2, ..., c_p\}$, where capacity of each c_x is V. $p = V/s_1 \times m$ when all s_i are of the same size else p is unknown.
Output: Find minimal p such that all smaller regions are packed into p number of chunks.
Steps:

```
for i is 1 to $m$, [Iterate through all the elements in set $U$]
$\forall c_j \in C$ = empty, $0 < j < p$
if $\sum_i s_i \leq V$)
Add $i^{th}$ element to $c_j$
else Add $c_j$ to $C$
increment $j$ i.e. start a new chunk
end for
$p = j$, since $j$ is keeping track of when a new chunk is added.
```

Data restructuring will be not beneficial if $T_{dr} > T_{comp}$. However, $T_{dr} < T_{comp}$ will result in contiguous accesses, significantly improving the performance.

The benefit of data restructuring is that when an application is run multiple times and requires only one data layout, the read operation has been highly optimized, making the total execution time of the operation much shorter. It minimizes the number of chunks required by a map task because after restructuring all smaller regions that are scattered among various chunks are packed together as shown in the Figure 5. However, if each application uses the file in a different way, that is, the file requires constant restructuring, then data restructuring will incur more overhead as compared to the performance benefits.

4. RESULTS AND DISCUSSION

In the experiments, we demonstrate how MRAP API performs for the two types of access patterns as described in Section 3.1. The first access pattern performs matching operation on the data sets, whereas the second access pattern deals with the non-contiguous accesses. We also show the performance improvement due to data restructuring for the second access pattern.

The most challenging part of this work is to fairly evaluate MRAP framework using various data access patterns against the same patterns used in the existing MapReduce implementations. Unfortunately, most HPC analytics applications which could enunciate the benefit from MRAP still need to be developed. Also, there are no established benchmarks currently available to test our design. We have used one application from the bioinformatics domain, an open source MapReduce implementation of the "read-mapping algorithm", to evaluate the MRAP framework. This application performs sequence matching and extension of these sequences based on the given criteria and fits well with the description of the first access pattern. For the second access pattern, we use both MRAP and MapReduce to read astrophysics data in tipsy binary format that is used in many applications for operations like halo finding. In the next subsection, we describe our testbed and benchmark setup.

4.1 Testbed and Benchmarks Description

There are 47 nodes in total with Hadoop 0.20.0 installed on it. The cluster nodes configurations are shown in the

Table 1: CASS Cluster Configuration

15 Compute Nodes and 1 Head Node	
Make& Model	Dell PowerEdge 1950
CPU	2 Intel Xeon 5140, Dual Core, 2.33 GHz
RAM	4.0 GB DDR2, PC2-5300, 667 MHz
Internal HD	2 SATA 500GB (7200 RPM) or 2 SAS 147GB (15K RPM)
Network Connection	Intel Pro/1000 NIC
Operating System	Rocks 5.0 (Cent OS 5.1), Kernel:2.6.18-53.1.14.e15
31 Compute Nodes	
Make& Model	Sun V20z
CPU	2x AMD Opteron 242 @ 1.6 GHz
RAM	2GB - registered DDR1/333 SDRAM
Internal HD	1x 146GB Ultra320 SCSI HD
Network Connection	1x 10/100/1000 Ethernet connection
Operating System	Rocks 5.0 (Cent OS 5.1), Kernel:2.6.18-53.1.14.e15
Cluster Network	
Switch Make & Model	Nortel Nortel BayStack 5510-48T Gigabit Switch

Table 1. In our setup, the cluster's master node is used as the *NameNode* and *JobTracker*, whereas the 45 worker nodes are configured to be the *DataNodes* and *TaskTrackers*.

The first application, CloudBurst consists of one data format conversion phase and three MapReduce phases to perform read mapping of genome sequences as shown in the Figure 7 a). The data conversion phase takes an ".fa" file and generates a sequence file with a format following HDFS sequence input format. It breaks the read sequence into 64KB chunks and write sequence in the form of these pairs $(id, (sequence, start\_offset, ref/read))$. The input files consist of a reference sequence file and a read sequence file. During the conversion phase, these two files are read in to generate the pairs for ".br" file. After this data conversion phase, the first MapReduce program takes these pairs in the map phase and generates *mers* such that, the resulting (key, value) pairs are $(mers, (id, position, ref/read, left\_flank, right\_flank))$. The flanks are added to the pairs in the map phase to avoid random reads in HDFS. These key, value pairs are then used in the reduce phase to generate *SharedMers* as $(read\_id, (read\_position, ref\_id, ref\_position, read\_left\_flank, read\_right\_flank, ref\_left\_flank, ref\_right\_flank))$. The second MapReduce program generates Mers per read. The Map phase does nothing and reduce phase groups the pairs generated in the first MapReduce program by *read_id*.

In MRAP, this whole setup is performed such that there is one map and one reduce phase as shown in the Figure 7 b). We do not modify the conversion phase in generating the .br file. The first phase reads from both reference and read sequence files to generate the SharedMers and they are coalesced and extended in the reduce phase. The only reason we can allow a the map phase to read chunks from multiple files is because, MRAP API allows for a list of splits per map task. Essentially, each mapper reads from two input splits and generate $(read\_id, (read\_position, ref\_id, ref\_position, read\_left\_flank, read\_right\_flank, ref\_left\_flank, ref\_right\_flank))$ for the shared mers. The reduce phase aligns and extends the shared mers. It results in a file, which contains every alignment of every read with at most some defined number of differences.

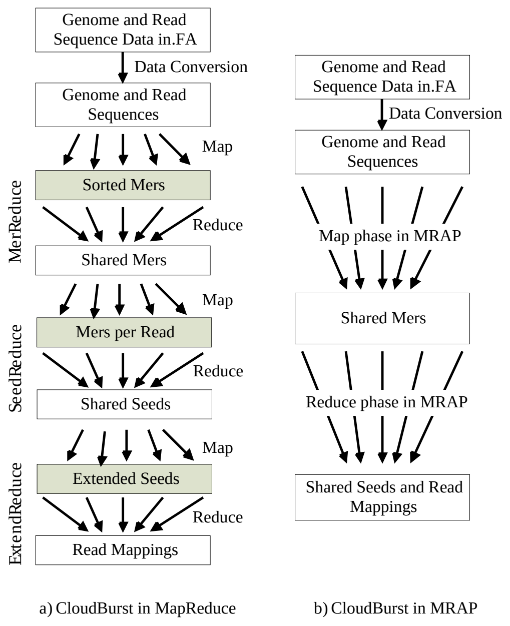


Figure 7: a) Overview of the Read-Mapping Algorithm using 3 MapReduce cycles. Intermediate files used internally by MapReduce are shaded [30]. b) Overview of the Read-Mapping Algorithm using 1 MapReduce cycle in MRAP.

In the second case, we perform a non-contiguous read operation on astrophysics data set used in halo finding application, followed by the grouping of given particles. There are two files in the downloaded data set: particles_name, which contains the positions, velocities and mass of the particles. In addition to the particles_name file, an input_data file summarizing cosmology, box-size etc and halo catalogs (ascii-files), containing: mass, position and velocity in different coordinates are also provided [2, 1].

Finally, we used a micro benchmark to perform small I/O requests using MRAP to show the significance of data restructuring. We use three configurable parameters to describe a simple strided access pattern [15]. These parameters are *stripe*, *stride* and *data set size*. We show the behavior of changing the stripe size with various data sizes, where the stride was dependent on the number of processes and stripe size. The stripe size is the most important parameter for these experiments because it determines the size and the number of read requests issued per process. We write a MapReduce program to perform the same patterned read operation. In the map phase each process reads a contiguous chunk, and marks all the required stripes in that contiguous chunk. In the reduce phase, all the stripes required by a single process are combined together.

4.2 Demonstrating Performance for MRAP

Bioinformatics Sequencing Application: We compare the results of MRAP and the existing MapReduce implementation of the read-mapping algorithm. As shown in the Figure 7, the traditional MapReduce version of the CloudBurst algorithm requires three MapReduce applications in

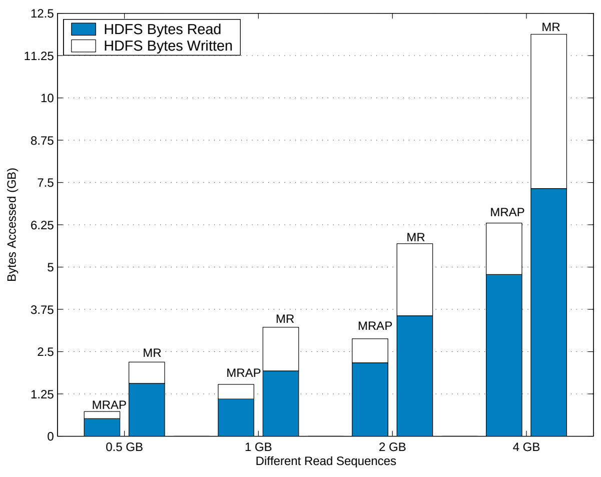


Figure 8: This graph compares the number of bytes accessed by the MapReduce and MRAP implementation of Read-mapping, and shows that MRAP accesses $\approx 47\%$ less data.

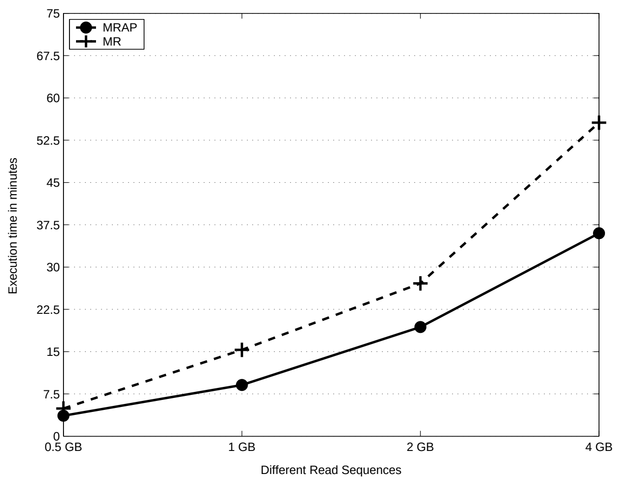


Figure 9: The graph compares the execution time of the Read-mapping algorithm using MRAP and MapReduce (MR).

order to complete the required analysis as compared with one MapReduce phase required in MRAP. Because the MRAP implementation requires only one phase of I/O, we anticipate that it will significantly outperform the existing CloudBurst implementation. We first show the total number of bytes accessed by both MRAP and MapReduce implementations in the Figure 8. Each stacked bar shows the number of bytes read and written by MRAP and MapReduce implementation. The number of bytes read is more than the number of bytes written because reference data sequences are being merged at the end of reduce phases. The number of bytes accessed in the MRAP application are on average 47.36% less than the number of bytes accessed in the MapReduce implementation as shown in Figure 8. This reduced number of I/O accesses result in an overall performance improvement of upto 33%, as shown in the Figure 9.

We were also interested in looking at the map phase timings because each map phase in MRAP was reading its input from two different data sets. Further break down of the execution time showed that map task took $\approx 55sec$ in each phase to finish, and there were three map phases making this time equal to $2min, 45sec$. In MRAP, this time was $\approx 1min, 17sec$, because MRAP reads both read and reference sequence data in the map phase, as opposed to reading either read or reference sequence in the map phase. Hence, the time of a single map task in MRAP is greater than the time of a single map task in MapReduce, but the benefit comes from multiple stages in MapReduce based implementation.

The MRAP implementation of CloudBurst application accesses multiple chunks per map task. Multiple chunks per map task generate remote I/O requests if all the required chunks are not present on the scheduled node. In this test, $\approx 7-8\%$ of the total tasks launched caused remote I/O accesses, slowing down the MRAP application. Hence, supplemental performance enhancing methods, such as dynamic chunk replication or scheduling multiple chunks, are required to be implemented with MRAP framework. Since, this particular access pattern does not access data non-contiguously, data restructuring is not an appropriate optimization for it.

Astrophysics Data sets: We ran a set of experiments to demonstrate the performance of MRAP over MapReduce implementation for reading non-contiguous data sets. In this example, we use an astrophysics data set and show how MRAP deals with non-contiguous data accesses. As described earlier the halo catalog files contains 7 attributes, i.e. mass (m_p), position (x, y, z) and velocity (V_x, V_y and V_z) for different particles. In the setup, we require our test application to read these seven attributes for only one particle using the given catalogs, and scan through the values once to assign them a group based on a threshold value. Since MRAP only reads the required data sets, we consider this case where less data as compared with total data set is required by the application. The MapReduce application, reads through the data set and marks all the particles in the Map phase. The reduce phase filters out the required particle data. We assume that the data set has 5 different particles, and at each time step we have attributes of all the 5 particles. Essentially, the map phase reads entire data set, and the reduce phase writes only $1/5^{th}$ of the data sets. The second MapReduce application, scans through this filtered dataset and assigns the values based on the halo mass. The MRAP implementation, reads the required $1/5^{th}$ data in the map phase, and assigns the values in the reduce phase. We show the results in Figure 11 and 10 by testing this configuration on different data sets.

The application has access to data sets from $1.5-10$ GB in the form ≈ 6.3 MB files, and the data of interest is $\approx 0.3-2$ GB. Figure 10 shows that amount of data read and written in an MRAP application is $\approx 75\%$ less than the MapReduce implementation. The reason is that MRAP API allows to read the data in smaller regions, hence, instead of reading the full data set only data of interest is extracted in the map phase. This behavior anticipates significant improvement in the execution time of the MRAP application when compared

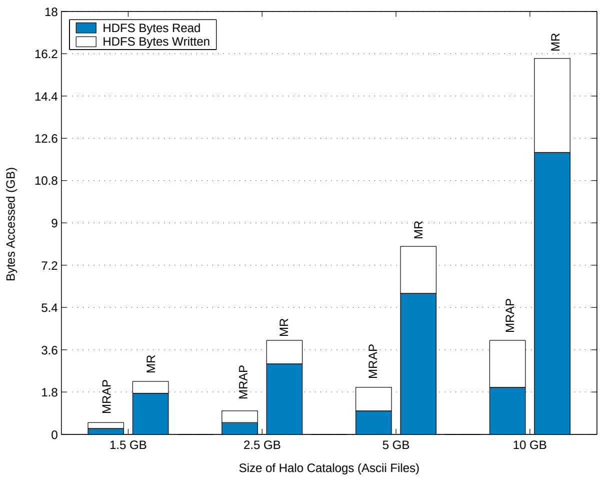


Figure 10: This graph shows the number of effective data bytes Read/Written using MRAP and MR. MRAP only reads the requested number of bytes from the system, as compared with MR, and shows $\approx 75\%$ fewer bytes read in MRAP.

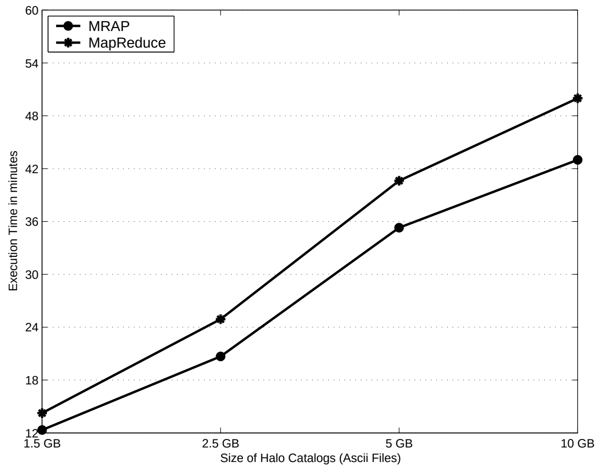


Figure 11: The graph compares the execution time of an I/O kernel that read Astrophysics data using MRAP and MapReduce. MRAP shows an improvement of up to $\approx 14\%$.

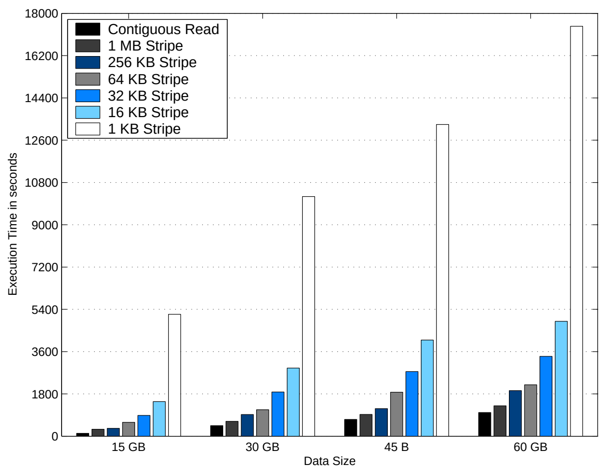


Figure 12: This figure shows the performance penalties due to small I/O by running a micro benchmark. The non-contiguous read operation with smaller stripe sizes has more performance penalties because of the amount of excessive data read and the number of I/O requests made.

with MapReduce. The results are shown in Figure 11, and they only show an improvement of $\approx 14\%$, because HDFS is not designed for small I/O requests. In the next subsection, we further elaborate on the small I/O problem, and show the results of proposed solution i.e. data restructuring.

4.3 Data Restructuring:

In the section 4.2, we saw the performance benefits of MRAP for applications with different data access patterns, where it minimized the number of MapReduce phases. Some patterns e.g. non-contiguous accesses incur an overhead in the form of small I/O, as we demonstrate in the Figure 12. We used random text generated data sets of 15 GB, 30 GB, 45 GB and 60 GB in this experiment to show that small I/O degrades performance of read operations. We have used 15 map tasks, each map task reads 1, 2, 3 and 4 GB using small regions (stripe sizes) ranging from 1 KB to 1 MB. We choose this range for stripe sizes because 1) there are many applications that store images which are as small as 1 KB [14], 2) 64 KB is a very standard stripe size used in the MPI/IO applications running on PVFS2 [9] and 3) 1 MB to 4 MB is the striping unit used in GPFS [31]. We have used the default chunk size of 64 MB in this experiment.

Figure 12 shows that smaller stripe sizes have larger performance penalties because of the number of read requests that are issued for striped accesses. 1 KB depicts the worst case scenario, where for 1 GB per map task will have 1,048,576 read calls which results in much more calls for larger data sets. Figure 12 also shows the time it takes to perform contiguous read for the same 1, 2, 3 and 4 GB per map task. Overall larger stripe sizes tend to perform well because as they approach the chunk size, they issue fewer read requests per chunk. These requests become contiguous within a chunk as the stripe size becomes equal to or greater than the chunk size.

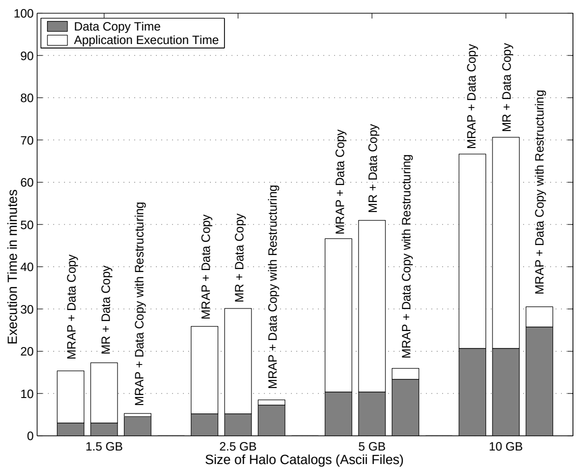


Figure 13: This figure shows the execution time of the I/O kernel for halo catalogs, with three implementations. MRAP API with data restructuring outperforms MR and MRAP API implementations.

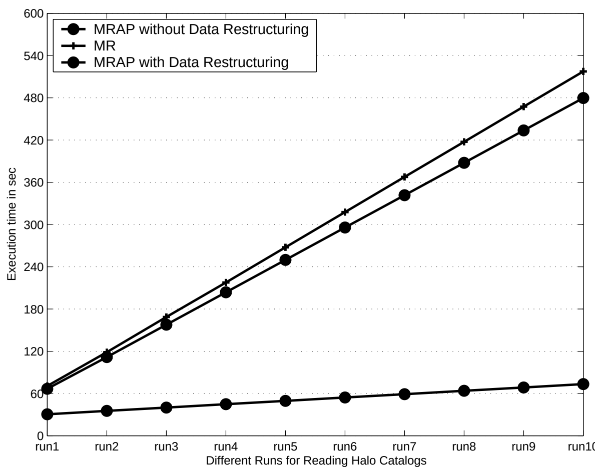


Figure 14: This figure shows the benefits of Data Restructuring in the long run. Same application with three different implementations (MR, MRAP, MRAP + data restructuring) is run over a period of time.

A contiguous read of 1 GB with 64 MB chunks will result in reading 16 chunks. On the other hand, with 1 MB stripe size, there will be 1024 stripes in total for 1 GB set. The upper bound of the number of chunks that eventually provides these 1024 stripes is 1024. Similarly, for 1 KB stripe size, there are 65536 stripes that generate as many read requests, and may map to 65536 chunks in the worst case. In short, we could use some optimizations to improve this behavior, such as data restructuring, which are studied in this paper.

We run a test by restructuring astrophysics data, and then read the data to find the groups in the given particle attributes. We restructure data such that the attributes at different time steps for each particle are stored together. In the example test case, we run the copy command and restructure data. The overhead of running copy command is shown in the Figure 13. After that we run the application to read the same amount of data as it was reading in Figure 11 and show the time it took to execute that operation. It should be noted that the amount of data read and written is the same after data restructuring. Data restructuring organizes data to minimize the number of I/O requests not the size of total requested data. In these tests, size of each request was $\approx 6.3MB$, and the number of I/O requests generated, for example for a $10GB$ data set is 1625. When data is restructured, 10 small regions each of 6.3 MB are packed into a single chunk of 64 MB, and reduce the number of I/O requests by 10 times. In the figure, we can see that data restructuring significantly improve the performance by up to $\approx 68.6\%$ as compared with MRAP without data restructuring and $\approx 70\%$ as compared with MapReduce. The overhead of data restructuring includes time to read the smaller regions, and put them into contiguous data chunks.

We would also like to describe that once restructured, subsequent runs with the same access patterns will perform contiguous I/O and have further performance improvement over non-restructured data. We present this case in the Figure 14, and show that data restructuring is useful for the applications with repeated access patterns. For the same configuration used in the Figure 13, we run the same application on 10 GB data set after data restructuring. It is evident from the graph, that even with the overhead as shown in Figure 13, data restructuring is giving promising results.

5. RELATED WORK

Large scale data processing frameworks are being developed because of the information retrieval for web scale computing. Many systems like MapReduce [16, 17], Pig [28], Hadoop [3], Swift [29, 36], Dryad [22, 23, 35] and many more abstractions are there that allow large scale data processing. Dryad has been evaluated for HPC analytics applications in [19]. However, our approach is based on MapReduce, which is well-suited for the data parallel applications where data dependence does not exist and applications run on a shared-nothing architecture. Some other approaches like CGL MapReduce [18] also propose a solution to improve the performance of scientific data analysis applications developed in MapReduce. However, their approach is fundamentally different from our work. In CGL MapReduce they do not address decreasing the number of MapReduce phases, rather they mitigate the file read/write issue by providing an external mechanism to keep read file data persistent across multiple map reduce jobs. Their approach does not work with HDFS, and relies on a NFS mounted source.

Scientific applications use high level APIs like NetCDF [7], HDF5 [4] and their parallel variants [8, 25] to describe the complex data formats. These APIs and libraries work as an abstraction with the most commonly used MPI framework by utilizing the *MPI File views* [11]. NetCDF (Network Common Data Form) is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of *array-oriented scientific data* [7]. The data model represented by HDF5 support very complex data objects, metadata and a completely portable file format with no limit on the number or size of data objects in the collection [4]. We develop ways of specifying access patterns similar to *MPI datatypes* and *MPI File views* within MapReduce. The motivation for our approach is to facilitate data-intensive HPC analytics particularly the ones with access patterns and are developed using MapReduce.

To improve the small I/O problem, many approaches have been adopted in HPC community particularly for the applications using MPI/MPI-IO. These techniques are supported both at the file system and programming abstraction level. Data sieving allows the processes to read excessive contiguous data set in a given range instead of making small I/O requests to multiple non-contiguous chunks. The limitation of this approach is that each process reads excessive amount of data [34]. Similarly, collective I/O also allows a process to read a contiguous chunk of data but then using MPI's communication framework, it redistributes the data among multiple processes as required by them [34]. In large-scale systems with thousand of processes, collective I/O with its two-phase implementation results in communicating a large amount of data among processes. Other approaches are for checkpointing applications like PLFS, which adds a layer between the application and the file systems and re-maps an application's write access pattern to be optimized for the underlying file system [12]. DPFS provides striping mechanisms that divide a file into small pieces and distribute them across multiple storage devices for parallel data access [32]. Our approach of data restructuring is significantly different from these approaches because we re-organize data such that processes are not required to communicate with each other, and maintain shared-nothing architecture for scalability.

6. CONCLUSION AND FUTURE WORK

We have developed an extended MapReduce framework to allow users to specify data semantics for HPC data analytics applications. Our approach reduces the overhead of writing multiple MapReduce programs to pre-process data before its analysis. We provide functions and templates to specify the sequence matching, and strided (non-contiguous) accesses in reading astrophysics data, such that access patterns are directly specified in the map phase. For experimentation, we ran a real application from bioinformatics and an astrophysics I/O kernel. Our results show a maximum throughput improvement up to 33%. We also studied the performance penalties due to the non-contiguous accesses (small I/O requests) and implemented data restructuring to improve the performance. Data restructuring uses a user-defined configuration file and reorganizes data such that all non-contiguous chunks are stored contiguously, and show a performance gain of up to 70% for the astrophysics data set.

These small I/O requests also map to multiple chunks that are assigned to a map task, and require schemes to improve performance by selecting optimal nodes for scheduling map tasks on the basis of multiple chunk locations. The study of improving chunk locality is left for the future work. In the future, we would implement the dynamic chunk replication and scheduling schemes on a working Hadoop cluster to address the data locality issue. We would also like to develop more real world HPC data analytics applications using MRAP, and also explore new applications with their different access patterns than the ones described in this paper.

7. ACKNOWLEDGMENTS

This work is supported in part by the US National Science Foundation under grants CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413, US Department of Energy Early Career Principal Investigator Award DE-FG02-07ER25747, and National Science Foundation Early Career Award 0953946.

8. REFERENCES

[1] Astrophysics - Hashed Oct-tree Algorithm. http://t8web.lanl.gov/people/salman/icp/hot.html.

[2] Cosmology Data Archives. http://t8web.lanl.gov/people/heitmann/arxiv/codes.html.

[3] Hadoop. http://hadoop.apache.org/core/.

[4] HDF5. http://www.hdfgroup.org/hdf5/.

[5] Hdfs metadata. https://issues.apache.org/jira/browse/hadoop-1687.

[6] http://www.cisl.ucar.edu/dir/09seminars/roskies 20090130.ppt.

[7] netCDF. http://www.unidata.ucar.edu/software/netcdf/.

[8] Parallel HDF5. http://www.hdfgroup.org/hdf5/phdf5/.

[9] Parallel virtual file system version 2. http://www.pvfs.org/.

[10] Relativistic Heavy Ion Collider. http://www.bnl.gov/rhic.

[11] MPI-2: Extensions to the message-passing interface. http://parallel.ru/docs/parallel/mpi2, July 1997.

[12] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Supercomputing, 2009 ACM/IEEE Conference*, Nov. 2009.

[13] Dhruba Borthaku. The Hadoop Distributed File System: Architecture and Design.

[14] Philip Carns, Sam Lang, Robert Ross, Murali Vilayannur, Julian Kunkel, and Thomas Ludwig. Small-file access in parallel file systems. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, April 2009.

[15] Avery Ching, Alok Choudhary, Kenin Coloma, Wei keng Liao, Robert Ross, and William Gropp. Noncontiguous I/O accesses through MPI-IO. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 104, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Jeffrey Dean. Experiences with mapreduce, an abstraction for large-scale computation. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 1–1, New York, NY, USA, 2006. ACM.

[17] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[18] J. Ekanayake, S. Pallickara, and G. Fox. Mapreduce for data intensive scientific analyses. In *eScience, 2008. eScience '08. IEEE Fourth International*

Conference on, pages 277–284, 2008.

[19] Jaliya Ekanayake, Thilina Gunarathne, Geoffrey Fox, Atilla Soner Balkir, Christophe Poulain, Nelson Araujo, and Roger Barga. Dryadlinq for scientific analyses. In *E-SCIENCE '09: Proceedings of the 2009 Fifth IEEE International Conference on e-Science*, pages 329–336, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[21] William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *EuroSys '07: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.

[23] Michael Isard and Yuan Yu. Distributed data-parallel computing using a high-level programming language. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 987–994, New York, NY, USA, 2009. ACM.

[24] YongChul Kwon1, Dylan Nunley2, Jeffrey P. Gardner3, Magdalena Balazinska4, Bill Howe5, and Sarah Loebman6. Scalable clustering algorithm for n-body simulations in a shared-nothing cluster. Technical report, University of Washington, Seattle, WA, 2009.

[25] Jianwei Li, Wei 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 *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39, Nov. 2003.

[26] Xuhui Liu, Jizhong Han, Yunqin Zhong, Chengde Han, and Xubin He. Implementing WebGIS on Hadoop: A Case Study of Improving Small File I/O Performance on HDFS. In *IEEE Cluster '09*, 2009.

[27] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 222–229, Washington, DC, USA, 2008. IEEE Computer Society.

[28] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[29] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: a fast and light-weight task execution framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[30] Michael C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[31] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[32] Xiaohui Shen and Alok Choudhary. DPFS: A Distributed Parallel File System. *Parallel Processing, International Conference on*, 0:0533, 2001.

[33] Volker Springel, Simon D. M. White, Adrian Jenkins, Carlos S. Frenk, Naoki Yoshida, Liang Gao, Julio Navarro, Robert Thacker, Darren Croton, John Helly, John A. Peacock, Shaun Cole, Peter Thomas, Hugh Couchman, August Evrard, Jorg Colberg, and Frazer Pearce. Simulations of the formation, evolution and clustering of galaxies and quasars. *Nature*, 435(70422):629–636, June 2005.

[34] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, page 182, Washington, DC, USA, 1999. IEEE Computer Society.

[35] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In Richard Draves and Robbert van Renesse, editors, *OSDI*, pages 1–14. USENIX Association, 2008.

[36] Yong Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206, July 2007.

APPENDIX

A sample configuration file for a strided access pattern is as follows.

```
<configuration>
Defining structure of the new file
<property>
<name>strided.nesting_level</name>
<value>1</value>
<description>It defines the nesting levels. < /description>
<property>
<name>strided.region_count</name>
<value>100</value>
<description>It defines the number of regions in a non-contiguous
access pattern. < /description>
</property>
<property>
<name>strided.region_size</name>
<value>32</value>
<description>It defines the size in bytes of a region, i.e. a stripe
size. < /description>
</property>
<property>
<name>strided.region_spacing</name>
<value>10000</value>
<description>It defines the size in bytes between two consecutive
regions, i.e. a stride. < /description>
</property>
</configuration>
```