SIDR: Efficient Structure-Aware Intelligent Data Routing in SciHadoop

Joe Buck, Noah Watkins, Greg Levin, Adam Crume,
Kleoni Ioannidou, Scott Brandt, Carlos Maltzahn, Neoklis Polyzotis

Dept. of Computer Science, University of California - Santa Cruz
1156 High Street, Santa Cruz, CA, USA 95064
{buck,jayhawk,glevin,adamcrume,kleoni,scott,carlosm,alkis}@cs.ucsc.edu

Abstract—**MapReduce is a popular framework for distributed, parallel computation that has started to be used in domains quite different from the web applications for which it was designed, including the processing of big structured data, e.g., scientific and financial data. Previous work on using MapReduce to process scientific data did not incorporate knowledge of this structure in its internal communications. We show that performance gains can be realized by leveraging knowledge of the structure of the data to minimize and localize communications between nodes, guarantee workload balance across processing nodes, ensure that Reduce tasks start as soon as possible, and create balanced, contiguous output. We implemented these improvements in SciHadoop, a version of the open-source Hadoop MapReduce framework designed for structured scientific data. Our results show total query execution time reductions of up to 29% over SciHadoop with initial results available with only 6% of the query completed, and the resultant output is more efficiently organized, compared to Hadoop.**

I. Introduction

MapReduce is a simple framework for enabling the parallel processing of large data sets. Its rise in popularity has led to its use in problem domains and with types of data beyond those for which it was originally designed, including scientific computing [1, 2, 3]. For large-scale scientific datasets, *MapReduce* is an attractive parallel processing framework due to its simple programming model, ability to scale well on commodity hardware, and the fact that many problems in scientific computing translate well to its type of parallelization.

In *MapReduce*, the framework handles many aspects of parallelizing a computation, including the routing of data during program execution. *MapReduce*'s communication model makes a worst-case assumption about data dependencies between *Map* and *Reduce* tasks that leads to significant run-time and communications inefficiencies — it assumes that all *Reduce* tasks may be assigned data from any *Map* task regardless of the actual data dependencies. The barrier created by this assumption is well known and several research efforts [4, 5, 6] have made progress in ameliorating its impact, however they did so for certain classes of queries or with resource limitations, after which the system devolved to standard *MapReduce*.

Scientific data is often highly structured (modeled as an array, a sphere, or some other shape [7, 8, 9]) and stored in file-formats where meta-data describing that structure resides along side the actual data. Previous efforts in applying *MapReduce* to scientific data typically fall into two camps: 1) those that make little direct use of this additional structure, rather preferring to process entire files [10, 11], thereby ignoring locality concerns and potentially facing scalability limitations or, 2) those that extract data from its native format and store it in some other layout [12, 13, 14], likely achieving performance gains while forfeiting access to the data in its native format.

This paper presents SIDR (**S**tructure-aware **I**ntelligent **D**ata **R**outing), an extension of SciHadoop [15], whose core idea is to elevate the data abstraction at which Hadoop partitions, routes and accounts for intermediate data from the byte-stream to the logical coordinate (n-tuple indicating data position within the dataset). SIDR's focus on the partitioning of intermediate data and its routing builds upon SciHadoop's focus on the partitioning of input data and the effects of input splits on the performance of a *MapReduce* query. SIDR represents a transition in Hadoop from naively moving bytes to understanding the structure of the data being processed. With SIDR, results are available prior to the completion of all *Map* tasks and, in contrast to previous research, the results are correct, rather than approximations. This is accomplished via a general method that is not limited to distributive functions or in terms of data-size.

SIDR, and SciHadoop, provide *MapReduce* with efficient, locality-aware access to scientific data in its original format while leveraging knowledge of the structure of the data to realize substantial performance gains.

This paper presents two primary contributions:

1) A formal model of internal communications in *MapReduce* that enables reasoning about communications and workload balance throughout a *MapReduce* program
2) Three key performance improvements to the *MapReduce* processing mode:
    a) A data-structure aware partitioning function for intermediate data;
    b) Extending the *MapReduce* framework so that *Reduce* tasks can derive their specific data dependencies and understand when those have been met. This added knowledge enables the production of correct early-results; and
    c) A modified scheduler that enables co-execution of

TABLE I: Summary of Common Symbols

Sets	
$\mathcal{T}$	Input to a *MapReduce* job
$\mathcal{I}$	Set of all input splits
$\mathcal{O}$	Output from a *MapReduce* job
$v'_{k'}$	the set of all values in v' that are part of a key/value pair where the key is k'
$\mathbb{K}^{\mathcal{T}}$	set of keys that actually exist in K for a *MapReduce* job
$\mathbb{K}_i^{\mathcal{T}}$	set of keys that actually exist in I_i for a *MapReduce* job
$\mathbb{K}'^{\mathcal{T}}_{\ell}$	set of keys in K' that are assigned to KEYBLOCK$_\ell$ for a given job
Elements	
I_i	the i^{th} input split
k, k'	a key in spaces K or K', respectively
KEYBLOCK	a partition of K'
KEYBLOCK$_\ell$	the ℓ^{th} KEYBLOCK
$\langle k, v\rangle$	a $key/value$ pair in $K \times V$
$\langle k', v'\rangle^i$	a $key/value$ pair in $K' \times V'$ created by the *Map* task processing I_i
RR(I_i)	the application of a RECORDREADER to an I_i
r	the number of *Reduce* tasks
m_i	a particular *Map* task

Reduce tasks with the *Map* tasks that they depend on for intermediate data

We use the formal model to prove the correctness of our *MapReduce* changes and then evaluate their performance via an implementation in SciHadoop, our version of the Hadoop *MapReduce* processing framework. Our results show up to a 29% reduction in total query execution time over SciHadoop with initial results available with as little as 6% of the query completed, the production of dense, contiguous output, and a 10x - 1,000x reduction in required network connections.

II. BACKGROUND

A. Scientific Data

Scientific data is typically stored in binary file formats that provide higher-level abstracts for accessing data. A common abstraction used by these libraries is a coordinate-based system. Data is read and written from files via functions that take coordinate arguments in lieu of byte-offsets and then translate those coordinates into accesses of the underlying file. Common scientific file formats include NetCDF [8], HDF5 [9], FITS [7], and GRIB [16] (for the rest of this paper, we use NetCDF notation). Given scientific libraries requirement that data accesses be done via coordinates, SciHadoop, which this work is based on, specifies its data via pairs of n-dimensional coordinates specifying a corner and a shape in the total set of data being processed (e.g., corner: {100,0,0} shape: {20, 50, 50} would specify a 50,000 element cube with its origin at {100,0,0}).

The process by which scientific libraries translate from coordinates to byte-offsets in the underlying file is typically opaque, leaving the application and user oblivious as to the actual layout of data on storage. To contrast that approach, unstructured data is accessed by directly specifying byte-extents in a file; the connection between data requested and that datas' location in the file is explicit. Furthermore, most scientific libraries only support accessing data via the coordinate system; it is not typically possible to request data from a scientific library by specifying a range of bytes to be read. This failure to provide a byte-stream oriented interface results in *MapReduce* programs processing scientific data having to either specify their input in terms of coordinates or pre-partition their data so every input is defined as an entire file or set of files. By specifying an entire file, a RECORDREADER can use the access library to query the file for the requisite meta-data and construct the coordinates that represent the entirety of the content of that file. The latter approach can create new data management issues as well as limit the cases where *MapReduce* can efficiently process scientific data.

B. MapReduce Overview

An illustration of the data flow for a *MapReduce* job (also called a "query" or "program") is shown in Figure 1. When a *MapReduce* job begins executing, a central coordinator partitions the specified input data, $\mathcal{T}$, into a set $\mathcal{I}$ consisting of units(subsets) called $InputSplit$s, denoted I_i. An I_i is typically defined as byte-ranges in one or more files (e.g., bytes 1024 - 2048 in file "dataFile1"). Each split is assigned to, and processed by, one *Map* task. The *Map* task then employs a file-format specific library, called a RECORDREADER, that reads the *Map* task's assigned I_i and outputs key/value pairs where the keys are in keyspace K. Those key/value pairs are consumed by said *Map* task which then outputs new key/value pairs, referred to as intermediate data, with keys in a logically distinct keyspace, K' ("logically distinct" indicates that specific values in this domain do not necessarily correspond to the same values in a different domain (e.g., the data at coordinate {2,4} in the input in not necessarily related to the data at coordinate {2,4} in the intermediate data). A partition function then maps the key for every intermediate key/value pair to a specific KEYBLOCK (a "KEYBLOCK" is a partition of the keyspace K' for the given *MapReduce* job).

Each *Reduce* task is assigned a KEYBLOCK and processes all intermediate data where the key portion of the key/value pair is within said KEYBLOCK. Prior to the application of the *Reduce* function, *Reduce* tasks perform a merge sort of all their data, combining all key/value pairs with the same k' key into a pair consisting of a single instance of the key and a list containing all the values (denoted $v'_{k'}$). The sorting and merging ensures that all values corresponding to a given key will be processed by the *Reduce* function at the same time.

As a *Reduce* task applies the *Reduce* function, it emits a new set of values. The total set of data written as output from all *Reduce* tasks is denoted $\mathcal{O}$.

MapReduce makes a few simple guarantees:

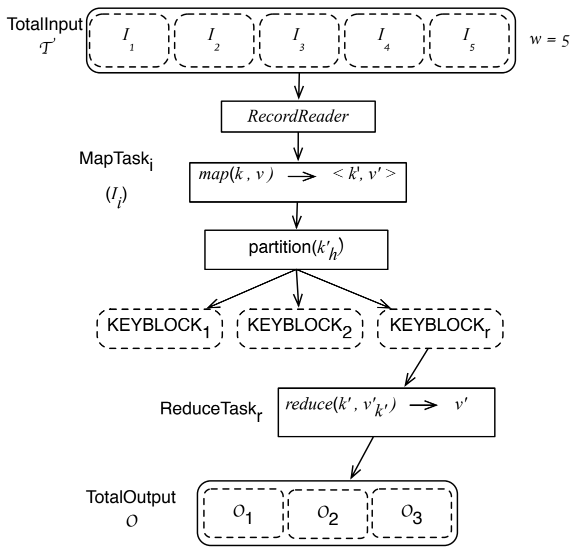

Fig. 1: Annotated data flow for a *MapReduce* program.

1) All data in the input will eventually be assigned to some *Map* task
2) All values for the same k' will be processed by a single *Reduce* task and at the same time
3) The framework handles fault tolerance (partial results from a failed task are discarded and the task is rescheduled)

The *MapReduce* framework is free to partition data and schedule tasks as it seems fit, typically taking data locality into consideration, as long as it fulfills these guarantees. The lack of ordering guarantees among tasks of the same type (*Map* or *Reduce*) as well as the lack *Map* task atomicity enables a higher degree of flexibility than is possible with more rigid parallel frameworks.

C. *The* MapReduce *Barrier*

The primary ordering constraint in *MapReduce* is that a *Reduce* task only processes data for a given key when it has all of the key/value pairs with that particular key. Given an inability to make assumptions about the behavior of the partition function for intermediate data, and the resulting assignment of data to KEYBLOCKs, a worst-case assumption of any *Map* task creating output for any *Reduce* task must be used. This necessitates a barrier between the end of the last *Map* task and the beginning of any *Reduce* task for a given *MapReduce* program. The barrier guarantees that all values for a given key will be processed at the same time with the side effect of preventing *Reduce* tasks from beginning to process their assigned data until the output from all *Map* tasks is available. The left portion of Figure 2 depicts the barrier, which is explored in more detail in Section IV-C.

III. *MapReduce*'s Communication Model

While a significant body of work relating to formal definitions of the *MapReduce* job model and its relation to other computing models exists [17, 18, 19, 20], little attention has been given to *MapReduce*'s internal communication model. In general, this is reasonable as there are points in the data flow of a *MapReduce* job where it is not possible to reason about the relationship between a function's input and output, making the use of worst-case assumptions necessary. By constraining ourselves to queries over structured scientific data and leveraging the meta-data included in scientific file formats, SIDR can correlate the inputs with outputs and therefore can make better decisions about data routing.

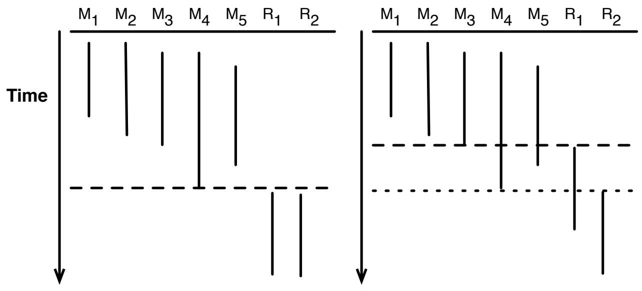
M$_1$ M$_2$ M$_3$ M$_4$ M$_5$ R$_1$ R$_2$
Time
M$_1$ M$_2$ M$_3$ M$_4$ M$_5$ R$_1$ R$_2$
(a)
(b)

Fig. 2: **a**: In general, *Reduce* tasks must wait for all *Map* tasks to complete prior to beginning their execution. **b**: When *Reduce* tasks understand their actual data dependencies (e.g., R_1 depends on M_1, M_2, M_3 while R_2 depends on M_3, M_4, M_5), they can start after the last *Map* task they depend on finishes.

A. *Blackboxes in the* MapReduce *Data Flow*

There are two points in the flow of data through a *MapReduce* program where it is difficult, or impossible, to correlate input with output (or vice versa). These points present a roadblock to any meaningful optimization of communications within *MapReduce* without sacrificing generality.

The first point at which the *MapReduce* infrastructure masks the flow of data is in the RecordReader. In running a *MapReduce* job, the user specifies a set of input data and a RecordReader. The *MapReduce* framework splits up the specified data into a set $(I_1, I_2, ..., I_i)$, each of which will be read by an instance of RecordReader that will output key/value pairs for use as input to a *Map* function.

In practice, the input to a RecordReader is usually expressed as byte-ranges while the output is a set of key/value pairs where the key is in some logical keyspace. This mismatch prevents reasoning about relationships between the two without taking drastic steps, such as invoking the RecordReader, providing it with a byte-range to process and observing its output. A definition for the data flow through a RecordReader is shown in Formulation (1).

$$\begin{aligned} &\text{for } m_i = |I_i| \, , \\ &\text{RR}(I_i) \; = \left\{ \langle k_1, v_1 \rangle^i , \langle k_2, v_2 \rangle^i , ..., \langle k_{m_i}, v_{m_i} \rangle^i \right\} \qquad (1) \\ &\langle k_j, v_j \rangle^i \in K \times V \quad \forall j \in \{1, 2, ..., m_i\} \end{aligned}$$

The second point at which the *MapReduce* data flow is opaque is the assignment of intermediate key/value pairs to KEYBLOCKs. This process represents a partitioning of the

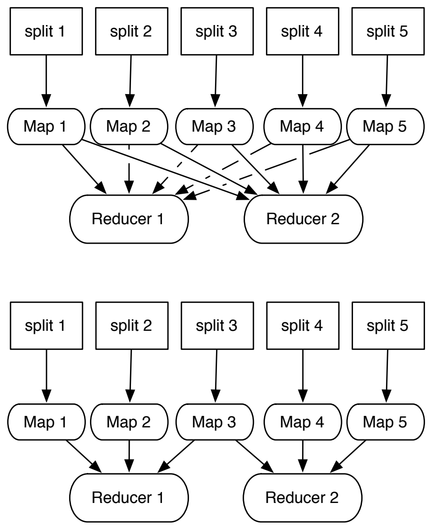

Fig. 3: **Top**: default communications pattern for the *MapReduce* program in Figure 2. **Bottom**: actual data dependencies for the same program.

keyspace for the intermediate data (K') where each subset (KEYBLOCK) will be assigned to a *Reduce* task. In Hadoop, the assignment of a given key/value pair of intermediate data to a KEYBLOCK is done via modulo arithmetic over a binary representation of the key. Like the RECORDREADER, it is difficult, if not impossible, to reason about the relationship between this function's input and output. In this case, the difficulty is caused by the interaction of the partition function with the key and the implementation specifics that come into play. A definition of the partition function is shown in Formulation (2).

$$partition : K' \rightarrow \{\text{KEYBLOCK}_\ell \mid \ell \in \{1, 2, ..., r\}\} \quad (2)$$

The goal of this partitioning is to create a roughly even distribution of data across KEYBLOCKs. A side effect of using a modulo function (the default in Hadoop) is that it precludes reversing the process with any accuracy; it is not possible to take a key/value pair in a KEYBLOCK and derive which *Map* task generated that data.

IV. HADOOP

In our discussion of *MapReduce*, we reference Hadoop [21] for implementation details, as it is the predominant publicly available *MapReduce* framework. The following subsections highlight a few implementation details that are relevant to the explanation of SciHadoop and SIDR.

A. Input Split Generation

The Hadoop run-time is tasked with partitioning the user-specified input files into sets of I_i. This process typically splits $\mathcal{T}$ linearly into I_i that are roughly equal in size to the underlying filesystem's block size. A side effect is that every I_i should be approximately the same size. This is a desirable property because a large difference in size, or skew, between different I_i would likely lead to commensurate imbalances between *Map* task run-times that could extend the run-time of the *MapReduce* job (see Figure 2 which shows the longest running *Map* task as a limiting factor for total *MapReduce* job performance).

B. KEYBLOCK *Sizes*

KEYBLOCK sizes are a function of the distribution of the observed intermediate keys ($\mathbb{K}'^{\mathcal{T}}$) combined with the partition function. Assuming a modulo-style partition function, a relatively even distribution of keys in $\mathbb{K}'^{\mathcal{T}}$ translates into a similarly even distribution among KEYBLOCKs.

Skew in KEYBLOCK sizes can cause divergence in run-times between *Reduce* tasks. Given that a *MapReduce* job typically has many times fewer *Reduce* tasks than *Map* tasks and that the global barrier causes *Reduce* tasks to all start at the same time, a disparity across *Reduce* task run-times typically has a larger impact on total *MapReduce* job run-time than *Map* task skew. Section IX has references to previous work on avoiding skew that provide further details on the effect of skew on *MapReduce* job run-time.

C. Communication in Hadoop

Every *Map* task produces a (potentially empty) data set for every *Reduce* task and each *Reduce* task requests its data set from every *Map* task some time after said *Map* task has successfully completed. This communication pattern, based on a worst-case assumption of global data dependencies, is shown in the top portion of Figure 3.

In Hadoop, *Reduce* tasks opportunistically copy intermediate data from completed *Map* tasks and merge them with previously copied data. By copying data as *Map* tasks complete, network IO can be overlapped with the computation of other *Map* tasks, resulting in performance improvements. As noted in SectionII-C, *Reduce* tasks cannot begin processing this data until after all *Map* tasks have completed (Figure 2(a)).

A *Map* task's generation of intermediate data highlights the fact that the *Map* and *Reduce* tasks process key/value pairs while the framework naively moves bytes. As intermediate key/value pairs are produced, the key for each pair is passed through the partition function, the output indicating which KEYBLOCK the data is destined for and therefore which data set the key/value pair should be written to. When a *Map* task finishes processing its assigned input, it closes the file containing intermediate data and writes out a small amount of meta-data for each KEYBLOCK, including the number of bytes in the set, to a header. When a *Reduce* task has acquired its set of intermediate data from a *Map* task, it is not aware of the contents of the data set, merely the data contained in the header. It is not until the *Reduce* task starts processing its data that the file is read and the identity of the keys contained within the file are known.

V. SCIHADOOP

SIDR builds upon previous work on integrating structured data into Hadoop [15], including: 1) expanding the use of logical coordinates in Hadoop to input split generation, 2) creating an extraction shape to explicitly describe how data in the initial input maps to intermediate data, and 3) enabling Hadoop to leverage available meta-data describing the structure of the data being processed to make more informed decisions during input split generation.

A. Expanding the Use of Logical Coordinates

As described in Section IV-A, Hadoop partitions the input into a set of I_i. Each of these is read by a RECORDREADER that emits key/value pairs for consumption by *Map* tasks.

SciHadoop defines its I_i in terms of logical coordinates, creating a situation where both RECORDREADER input and output are defined at the same level of abstraction and also in the same logical space (coordinates in the logical space of K). This alignment of abstraction levels enables SciHadoop to reason between a given I_i and the keys it will produce for the corresponding *Map* task as well as correlating *Map* task inputs to a given I_i. In fact, I_i and the set of all keys that $\mathrm{RR}(I_i)$ will produce are equivalent. Given that we can translate from the total input to the set of I_i that will be consumed by the set of all *Map* tasks, the system can, at the initialization of a Hadoop job, calculate the set of keys in K that will actually be processed by the set of all *Map* tasks, denoted $\mathbb{K}^{\mathcal{T}}$. This newfound knowledge is significant as $\mathbb{K}^{\mathcal{T}}$ can be used in place of the entire keyspace K when reasoning about the flow of data, thereby enabling SciHadoop to make decisions based on the keys that actually exist in the data set being processed, rather than the set of all valid keys.

It is worth noting that defining I_i in terms of logical coordinates, rather than the default of byte-ranges, complicates attempts at achieving high rates of data locality while scheduling *Map* tasks, leaving SciHadoop the added task of creating splits that result in decent data locality. This point was addressed [15] and the results indicate that achieving good data locality while defining I_i via logical coordinates is feasible.

B. Extraction Shape

The extraction shape, described in [15], is a concrete representation of how a given *Map* function translates keys in its input, K, into keys in its output, K'. Specifically, the extraction shape is logically tiled, in a given order, over $\mathcal{T}$ with each instance representing a unique k' key in K'. This means that *MapReduce* can now translate any $k \in K$ to a $k' \in K'$. Defining an extraction shape for a *MapReduce* program that is applying a query over structured scientific data is straightforward and can be represented efficiently via a set of n-dimensional arrays [15].

As an example, consider a query over a 2-dimensional data set that is downsampling by taking every disjoint 2x2 region of the input data and outputting the average value of the 4 data points. In this example, the extraction shape would be $\{2, 2\}$ (indicating every 2x2 input shape translates into a single element in the output). An example of this translation can be seen in Figure 5(b) as well as an extraction shape that represents an upsampling in Figure 5(a). Strided access (reading data at regularly spaced intervals) can be described by adding an additional n-dimensional array indicating the stride lengths (and other patterns, such as nested strides, can be represented as well).

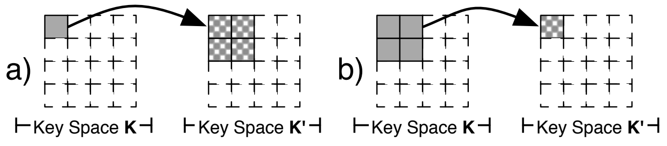

Fig. 4: **a**: A single value in K translating into 4 different values in K' **b**: A {2,2} extraction shape translating 4 different values in K into 1 value in K'

For value-based queries, such as requesting all of the data where the value exceeds some threshold, the extraction shape can be a single-cell or a collection of cells (in the latter case, a list of values may be returned if more than one value in the shape exceeds the threshold).

C. Input Meta-data

Scientific file formats typically encode structural meta-data alongside data in a single file. The meta-data is typically exposed by a function call that returns the dimensions and type of data being stored. An example of meta-data for a file that we use in our experiments can be seen in Figure 5.

Leveraging this meta-data allows Hadoop to translate from the specified input for the *MapReduce* program to $\mathbb{K}^{\mathcal{T}}$. The same information also enables several performance optimizations including reading from disk only the data required for a query [15] (as opposed to reading all of the data and then discarding the unwanted portions).

```
dimensions:                variables:
   time = 7200;               int windspeed(time, lat, lon, elev);
   lat = 360;
   lon = 720;
   elev = 50;
```

Fig. 5: An example of meta-data stored in a file containing scientific data. The file contains a single 4-dimensional variable named "windspeed" with lengths {7200, 360, 720, 50}.

VI. ENHANCING THE MAPREDUCE COMMUNICATIONS MODEL FOR STRUCTURED DATA

The availability of meta-data describing the structure of the data being processed combined with the extraction shape's succinct representation of the *Map* tasks translation of keys between the initial keyspace K and the intermediate keyspace K' clarifies *MapReduce*'s internal communications model. We created an enhanced partition function that incorporates this additional knowledge to partition the space of keys that actually exist ($\mathbb{K}^{\mathcal{T}}$) rather than the entire keyspace (K). This new function can provide absolute bounds on KEYBLOCK size skew, rather than the probabilistic guarantee normally provided, as well as create KEYBLOCKS consisting of contiguous ranges of keys in K', rather than keys spread throughout the total space. Furthermore, we can make the new partition function invertible, allowing *Reduce* tasks to calculate the set of keys that will be assigned to their KEYBLOCK (KEYBLOCK_ℓ). Given KEYBLOCK_ℓ, a *Reduce* task can determine when its actual data dependencies are met, rather than pessimistically deferring processing until all *Map* tasks finish (global barrier).

A. Replacing the Hash Function

Given that SciHadoop has access to $\mathbb{K}^{\mathcal{T}}$ and can map keys in that space onto $\mathbb{K}'^{\mathcal{T}}$ via the extraction shape it can also reason about the total output space ($\mathcal{O}$). For a particular $\mathbb{K}^{\mathcal{T}}$ and extraction shape, $\mathbb{K}'^{\mathcal{T}}$ can be calculated by dividing the length of each dimension in $\mathcal{T}$ by the entry in the corresponding dimension of the extraction shape. Since an extraction shape can map a single value in K into multiple values in K', multiple values in K into a single value in K', or a single value in K into a single value in K', $\mathcal{O}$ may be smaller, larger or the same size as $\mathcal{T}$, respectively. The $\mathcal{O}$ for a given $\mathcal{T}$ and extraction shape, abbreviated to 'es', is denoted $\mathcal{O}^{\mathcal{T}}_{\text{es}}$.

In SIDR, Hadoop's default partition function is replaced with one that incorporates knowledge of $\mathbb{K}'^{\mathcal{T}}$, the extraction shape, and $\mathcal{O}^{\mathcal{T}}_{\text{es}}$. This new function, referred to as partition+, computes $\mathbb{K}'^{\mathcal{T}}$ and then partitions that, rather than K', into r KEYBLOCKs (where r is the number of *Reduce* tasks).

Given that $\mathcal{O}^{\mathcal{T}}_{\text{es}}$ for a query over structured data is a fixed size (for the queries we're considering), partition+ can guarantee an upper bound on the skew in sizes between KEYBLOCKs. This is accomplished by selecting said upper bound (possibly user-defined or derived from query details), creating an n-dimensional shape whose total size is smaller than the upper bound, determining the total number of instances of the shape that exist in $\mathcal{O}^{\mathcal{T}}_{\text{es}}$, and then dividing the total count of those instances by the number of reducers. The final result is the number of instances of our shape that should make up each KEYBLOCK. This process guarantees that each KEYBLOCK varies in size by at most one instance of the shape that was selected, which was chosen to be smaller than the selected permissible variance.

Given that partition+ knows not only the size but also the contents of $\mathbb{K}'^{\mathcal{T}}$, it can also produce KEYBLOCKs consisting of keys that are contiguous in K'. This is accomplished by partitioning $\mathbb{K}'^{\mathcal{T}}$ in contiguous ranges in K', rather than using a modulo operator. Creating KEYBLOCKs that will write dense arrays, as opposed to writing randomly throughout $\mathcal{O}$, simplifies interacting with the total output at a later date and has performance benefits as a result of producing fewer writes than in the default case. The latter point assumes that the scientific library will convert logically dense arrays into efficient file accesses.

B. Altering the MapReduce *Communications Model*

Combining the additional knowledge provided by the meta-data from scientific access libraries, the extraction shape and our partition+ function, it is possible to extend Hadoop to calculate the set of keys in K that will produce intermediate data for a given KEYBLOCK as well as the specific set of data that will be assigned to a given *Reduce* task. This ability to reason about elements in the initial input and their translation, via the *Map* and partition functions, into elements assigned to *Reduce* tasks is a powerful new feature.

The value $\mathcal{I}_\ell$ represents the set of I_i that, when processed by a RECORDREADER and its associated *Map* task, will produce at least one intermediate key/value pair that will be assigned to KEYBLOCK_ℓ. $\mathcal{I}_\ell$ is the actual data dependency of KEYBLOCK_ℓ and in using that as a barrier, rather than assuming a dependency on all I_i, *Reduce* tasks can begin processing data assigned to their KEYBLOCK as soon as the last I_i in their $\mathcal{I}_\ell$ completes while still guaranteeing correctness. A definition of $\mathcal{I}_\ell$ is shown in Formulation (5) and the process by which $\mathcal{I}_\ell$ is computed is described in the remainder of this section. The ability to calculate $\mathcal{I}_\ell$ enables the more precise communications model depicted in the bottom portion of Figure 3. The effect of this new model on scheduling is shown in Figure 2(b) where *Reduce* task R_1 is allowed to start prior to *Map* tasks M_4 and M_5 completing because all of its data dependencies are fulfilled.

When a *MapReduce* job begins, the input is specified and the range of keys in K that are present ($\mathbb{K}^{\mathcal{T}}$) is determinable via the meta-data provided by the scientific access libraries. Once $\mathbb{K}^{\mathcal{T}}$ is partitioned into $\mathcal{I}$, then a given I_i is itself $\mathbb{K}^{\mathcal{T}}_i$. With this additional information, we can re-frame Formulation (1) as Formulation (3).

$$
\begin{aligned}
&\text{for } m_i = |I_i| \,, \\
&\quad \text{RR}(I_i) \;= \\
&\qquad\qquad \left\{ \langle k_1, v_1 \rangle^i , \langle k_2, v_2 \rangle^i , ..., \langle k_{m_i}, v_{m_i} \rangle^i \right\} \qquad (3) \\
&\quad \langle k_j, v_j \rangle^i \in \mathbb{K}^{\mathcal{T}}_i \times V \quad \forall j \in \{1, 2, ..., m_i\}, \\
&\quad \bigcup_{j=1}^{m_i} k_j = \mathbb{K}^{\mathcal{T}}_i = I_i
\end{aligned}
$$

An explicit extraction shape enables SIDR to easily understand how a key in K translates into key(s) in K' and likewise from $\mathbb{K}^{\mathcal{T}}$ to $\mathbb{K}'^{\mathcal{T}}$. The term $\mathbb{K}'^{\mathcal{T}}_i$ denotes the keys in $\mathbb{K}'^{\mathcal{T}}$ that result from applying the *Map* function to the key/value pairs in the corresponding $\mathbb{K}^{\mathcal{T}}_i$. In practice, $\mathbb{K}'^{\mathcal{T}}_i$ is easily calculated by combining the extraction shape with $\mathbb{K}^{\mathcal{T}}_i$.

The set of all keys in $\mathbb{K}'^{\mathcal{T}}$ that will be assigned, via the partition+ function, to KEYBLOCK_ℓ is denoted $\mathbb{K}'^{\mathcal{T}}_\ell$. When *Reduce* tasks start up, they are supplied an ID, indicating their role amid the total set of *Reduce* tasks. A *Reduce* task can use that ID, the partition+ function, and $\mathbb{K}^{\mathcal{T}}$ to calculate its $\mathbb{K}'^{\mathcal{T}}_\ell$, which is necessary to determine I_ℓ for a KEYBLOCK, as shown in Subsection VI-C.

C. Ensuring Correctness When Starting Reduce *Tasks Early*

As has been discussed, an extraction shape represents how a key in K is mapped to a key in K'. This mapping is accomplished by a series of multiplications involving the K key and the extraction shape. Given this, it is trivial to also map from K' to K; the multiplication is simply inverted. The function that maps from K' to K is referred to as $ExSh^{-1}()$ (short for "inverse extraction shape"). This ability to both map from K to K' as well as from K' to K is not possible in the general case of Hadoop due to the default partition function being a modulo operator.

Combining $\mathbb{K'}_{\ell}^{\mathcal{T}}$ (the set of all keys in K' that will be assigned to KEYBLOCK$_{\ell}$) with the ability to map from K' to K, a *Reduce* task can compute the set of keys in $\mathbb{K}^{\mathcal{T}}$ that produce intermediate data destined for its KEYBLOCK, denoted $\mathbb{K}_{\ell}^{\mathcal{T}}$. A definition of $\mathbb{K}_{\ell}^{\mathcal{T}}$ is shown in Formulation (4).

$$\text{for KEYBLOCK}_{\ell}, \qquad (4)$$
$$\mathbb{K}_{\ell}^{\mathcal{T}} = \bigcup k \mid \exists k' \in \mathbb{K'}_{\ell}^{\mathcal{T}} \text{ such that } ExSh^{-1}(k') = k$$

The ability to calculate $\mathbb{K}_{\ell}^{\mathcal{T}}$ is requisite for maintaining correctness when starting *Reduce* tasks early in the case where multiple elements in K map to a single element in K' (Figure 4(b)). The set of keys in K that map to the same point in K' may fall in different I_i (generating multiple $\langle k', v'\rangle$) or the same I_i (generating a single $\langle k', v'\rangle$). Since the *Reduce* task doesn't know how many $\langle k, v\rangle$ were combined to produce a given $\langle k', v'\rangle$, it cannot begin processing after receiving a particular $\langle k', v'\rangle$ without risking the production of an answer based on insufficient input. This is a significant issue, as a pessimistic solution would require reverting back to the original global barrier.

$$I_{\ell} = \forall I_i \mid \exists\ k \text{ such that } k \in \mathbb{K}_{\ell}^{\mathcal{T}} \ \wedge\ k \in I_i \qquad (5)$$

The issue of the ambiguity as to the number of $\langle k, v\rangle$ that were combined to form a particular $\langle k', v'\rangle$, and therefore KEYBLOCK$_{\ell}$, can be resolved in two different ways:

1) $\mathcal{I}_{\ell}$, the set of inputs that will produce data destined for KEYBLOCK$_{\ell}$, can be computed and that can be used as a proxy for $\mathbb{K}_{\ell}^{\mathcal{T}}$, since it is a super-set (see Formulation (5)).

2) each key/value pair in the K' space can be annotated to include the number of elements in the K space that it represents. Each *Reduce* task can then keep a running tally of the number of $\langle k, v\rangle$ represented by the $\langle k', v'\rangle$ it receives. When it has accumulated data representing all $\langle k, v\rangle$ in its $\mathbb{K}_{\ell}^{\mathcal{T}}$, processing can safely begin. SIDR uses the former method and also implemented the annotations required for the latter method as a means of validating the system's correctness.

In Hadoop, all $\langle k', v'\rangle$ produced by a particular *Map* task and assigned to the same KEYBLOCK are written into the same file. We added a field to the header data for that file indicating how many $\langle k, v\rangle$ are represented by the set of all $\langle k', v'\rangle$, for a given KEYBLOCK, in the file. As mentioned in Subsection IV-C, when a *Reduce* task retrieves its intermediate data, it only has access to the information in the header; with our additions to the header data, a *Reduce* task can now track the set of all $\langle k, v\rangle$ represented by the contents of the files containing its intermediate data without having to read and parse the data.

D. Benefits of the Enhanced Model

The ability for *Reduce* tasks to begin processing data as soon as possible allows for additional overlapping of computation with IO beyond that already present when *Reduce* tasks copy intermediate results from completed *Map* tasks while other *Map* tasks execute. Previous research [4, 5, 6] indicates that this additional overlapping will translate into shorter total run-times for *MapReduce* jobs. The resources that are typically consumed by a *Reduce* task while waiting for unnecessary *Map* tasks to complete are now made available for other work and the storage occupied by intermediate data for the completed KEYBLOCK can be released prior to job completion.

VII. Altering task scheduling in Hadoop

Alterations to the default Hadoop scheduler, *org.apache.hadoop.mapred.JobQueueTaskScheduler*, were required in order to fully realize the benefits made possible by our previously described changes. When a Hadoop job begins, all *Map* tasks are added to a tree structure with the lowest level of nodes corresponding to specific servers in the cluster and higher level nodes being aggregations of servers (racks, etc.). Tasks are first added to the node(s) in the tree representing servers where the data for that task is stored (to achieve data locality). That same task is then added to the parent of those nodes (representing the rack that each data-local server resides in) and then to the parent of that node (representing the pool of all servers in the cluster). When a server requests a new *Map* task, that tree is traversed, starting at the leaf-node representing that server and working its way up, with the first runnable task encountered being returned. This results in tasks that would be local to that server being returned if they exist. Barring that, a rack-local task is returned if one exists or else, as a last resort, any non-local, runnable task is returned. In contrast, *Reduce* tasks are scheduled in monotonically increasing order of their IDs as slots become available.

In terms of *Reduce* tasks starting once their data dependencies are met, the interaction between the two default scheduling polices results in those dependencies being met haphazardly since no attempt is made to co-schedule *Reduce* tasks with the *Map* tasks they depend on. In practice, the odds of a *Reduce* task having its data dependencies met was mostly a function of the number of *Map* tasks it depended on combined with how many *Map* tasks had completed up to that point; in other words it was strictly probabilistic.

In SIDR, the scheduling process was altered so a *Map* task was only available to be scheduled if at least one *Reduce* task that depended on it was already running (dependency information was generated by the *FileInputFormat* class during job submission and then read by *JobInProgress* when scheduling tasks). The actual scheduling algorithm was unchanged, but rather was not informed of *Map* tasks until those tasks would contribute to a running *Reduce* task. This co-scheduling creates the desired effect of correlating *Map* tasks with running *Reduce* tasks.

VIII. Evaluation

Experimental Setup. Our experiments were conducted on a cluster of 25 nodes, each with two 2.0GHz dual-core Opteron 2212 CPUs, 8GB DDR-2 RAM, four 250GB Seagate 7200-RPM SATA hard drives, and Gigabit Ethernet, running Ubuntu 10.10. SIDR built upon the SciHadoop code [22] that we ported from Hadoop 0.23 to Hadoop 1.0. Our Hadoop cluster has a single node acting as both the NameNode and JobTracker while the other 24 nodes serve as both DataNodes and TaskTrackers. The 24 DataNode/TaskTracker nodes use one hard drive for the OS, supporting libraries and for temporary storage while the other 3 hard drives are dedicated to HDFS. All nodes have a single Gigabit network connection to an Extreme Networks' Summit 400 48-t switch. HDFS is configured with 3x replication and 128 MB block size. Each TaskTracker is configured with 4 *Map* task slots and 3 *Reduce* task slots (testing showed these were the optimal settings for our cluster).

A. Early Results

Using the approach described in Subsection VI-C, we augmented *FileInputFormat* code to calculate the data dependencies (I_ℓ) for each *Reduce* task. The Hadoop task scheduler was modified as described in SectionVII to schedule *Reduce* tasks and then add the elements in each scheduled *Reduce* task's I_ℓ to the pool of *Map* tasks to be scheduled. A *Reduce* task starts processing its data once it has acquired the intermediate data from the set of *Map* tasks in its I_ℓ.

1) Query 1: This experiment applied a median function over a 4-dimensional data set of sizes $\{7200, 360, 720, 50\}$ and using an extraction shape of $\{2, 36, 36, 10\}$ (meta-data representing this file appears in Figure 5). The data set can be thought of as 300 days worth of hourly windspeed measurements at a resolution of 0.5°Longitude by 0.5°Latitude at 50 different elevations with the query representing finding a median value, over 2 consecutive days, for each 18°Longitude by 36°Latitude region, in cross-sections of 10 elevation steps.

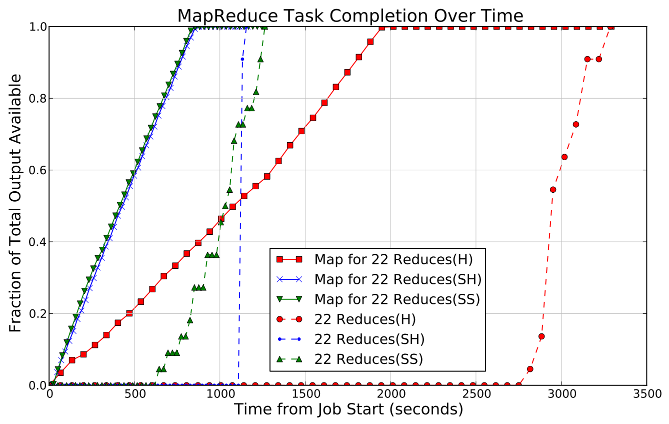

Fig. 6: *Map* and *Reduce* task completion for the same query run with Hadoop(H), SciHadoop(SH) and SIDR (SS)

We first compare Hadoop, SciHadoop and SIDR with the total number of *Reduce* tasks configured to 22, with the choice of 22 *Reduce* tasks owning to best practices dictating that 90% of the node count as a reasonable number of *Reduce* tasks. Figure 6 shows *Map* and *Reduce* task completion over time. SIDR starts producing results around 625 seconds while SciHadoop's first result arrives just after 1,132 seconds and Hadoop's first result coming at 2,797 seconds. The difference in the slopes of both *Map* and *Reduce* tasks between Hadoop and SciHadoop owe to the efficiencies gained by intelligent input split generation and data locality enabled by SciHadoop [15]. The gap between the first result in SIDR and the first results in SciHadoop and Hadoop clearly shows a benefit resulting from using the actual data dependencies present in the data being processed. The query executing with SIDR completes at 1,264 seconds while SciHadoop completes slightly sooner, at 1,250 seconds. SIDR's slightly longer run-time is due to the denseness of its output. The last SIDR *Reduce* task has to copy, merge and process all of the data in the last 4.5% (1/22nd) of the *Map* tasks. In SciHadoop, the output of those *Map* tasks would be spread evenly across all *Reduce* tasks. In summary, SIDR produces its first result twice as soon as the query run against SciHadoop and more than four times faster than Hadoop.

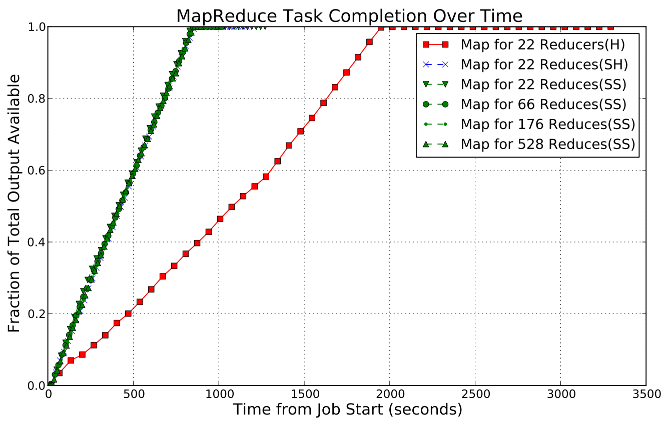

Fig. 7: *Map* task completion for a fixed query run with Hadoop using 22 *Reduce* tasks, SciHadoop using 22 *Reduce* tasks and SIDR running 22, 66, 176 and 528 *Reduce* tasks.

Next, we present the same query and dataset while varying the number of total *Reduce* tasks configured for each execution. By fixing the query and the input, the output produced by the query will likewise remain fixed. Increasing the number of *Reduce* tasks will result in each *Reduce* task having a smaller amount of data assigned to its KEYBLOCK, which will reduce its data dependencies. Figures 7 and 8 show these experiments representing the query run with 22 for all implementations as well as 66, 176 and 528 *Reduce* tasks for SIDR. Given the size of our dataset (348 GB) and the configured HDFS block size (128 MB), the SciHadoop partitioning scheme creates 2,781 input splits for this job.

Figure 7 shows a graph of *Map* task execution time as the number of *Reduce* tasks is varied. This graph does not change in any appreciable manner as the number of *Reduce* tasks varies, with the exception of Hadoop being less efficient than the other approaches. This result indicates that our changes, which alter the communications between *Map* and *Reduce* tasks as well as *Reduce* task behavior and scheduling (to a lesser extent), do not have a detrimental impact on *Map* task performance. It also shows that varying the number of *Reduce* tasks does not affect the run-time of *Map* tasks.

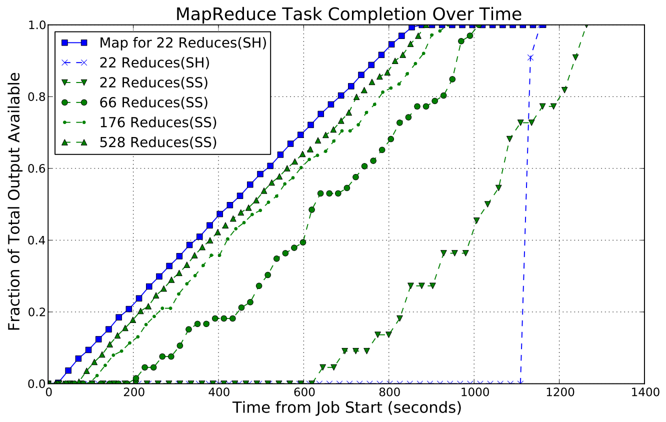

Fig. 8: *Reduce* task completion for a fixed query run with Hadoop using 22 *Reduce* tasks, SciHadoop using 22 *Reduce* tasks and SIDR running 22, 66, 176 and 528 *Reduce* tasks.

The results in Figure 8 show the effects on *Reduce* task run-time as the number of *Reduce* tasks is scaled. With SIDR, as the number of *Reduce* tasks is increased, the time to first result and the total job execution time both decrease. The same query run with SIDR finishes 29% faster than with Hadoop and nearly three times faster than Hadoop (shown in Figure 6). These performance improvements are attributable to decreasing the size of the data dependency for each *Reduce* task allowing *Reduce* tasks to start sooner and overlap more IO with computation. Results for Hadoop are omitted to allow the graph to present a finer resolution, thereby more readily showing the variations between the different experiments.

As has been previously shown in Subsection VI-C, a given *Reduce* task cannot begin processing its data until all of its data dependencies have been satisfied. Given this requirement, the ideal graph for the *Reduce* tasks would be a line that paralleled the graph for *Map* task completion, shifted to the right by the average amount of time it took a *Reduce* task to process its assigned data (which will vary with the number of *Reduce* tasks, given a fixed amount of output). Figure 8 shows that as the number of *Reduce* tasks increases, the corresponding graph line converges towards that ideal with 528 *Reduce* tasks being very close to optimal.

2) Query 2: Applying the same techniques to a different class of queries yields similar results in that both the time to first result and total run-time are decreased. For the second query, we created a data set of the same size, $\{7200, 360, 720, 50\}$, with values generated from a Gaussian distribution via *java.util.Random* class. The query applies a filter to the data that emits only values more than three standard deviations greater than the mean value. Query 2 uses an extraction shape of $\{2, 40, 40, 10\}$ out of convenience; results will contain a list of all the values in the extraction shape that are over the threshold. Since we're only returning values greater than three standard deviations then the total output should be approximately 0.1% of the total data set (93.31 million values out of a 93.31 billion values data set).

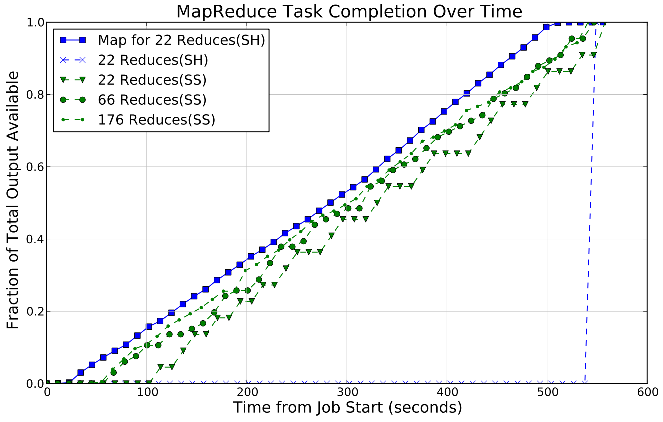

Fig. 9: *Reduce* task completion for a filter query run with SciHadoop using 22 *Reduce* tasks and SIDR running 22, 66 and 176 *Reduce* tasks.

Figure 9 shows the results for the second query when varying the number of *Reduce* tasks with SIDR and SciHadoop with 22 *Reduce* tasks (the *Map* task completion results for 22 *Reduce* tasks is show for reference). Again, Hadoop results are omitted in favor of showing more detail via a smaller x-axis.

Query 2 outputs significantly less total data than Query 1. Because of this, the *Reduce* task completion lines converge towards the optimal line with fewer total *Reducer* tasks than Query 1 since each *Reduce* task has less data to process and therefore finishes more quickly, freeing up that server to process another *Reduce* task. Also, since the *Reduce* tasks represent such a small fraction of the total query execution time to start with (indicated by the slope of the graph for SciHadoop with 22 *Reduce* tasks), there is little room for improvement. For these reasons, the reduction in total query time for Query 2 is much smaller than it was for Query 1.

By contrasting the results of the two queries on data of the same size, it is evident that the nature of the query being applied has a direct effect on number of *Reduce* tasks required to approach optimal performance. In general, the ideal number of *Reduce* tasks for a given query would be the minimum number required to achieve performance within some delta from the optimal line.

B. Contiguous Output

The partition+ function leverages query-specific knowledge to partition the output space into KEYBLOCKs that are both balanced in size and contain contiguous portions of the output space. This latter point is a significant improvement over the default when writing out structured data in a *MapReduce*

TABLE II: Scaling of Individual Reduce Write Times and Size

Hadoop Reduce Write Scaling		
Total Reduces	Avg Time in Seconds (Std Dev)	Output Size (MB)
20	6 (.6)	494
40	11.4 (.9)	988
80	24.2 (3.2)	1976
SIDR Reduce Write Scaling		
*	0.3 (.02)	24.8

TABLE III: Network Connection Scaling

Hadoop Reduce Write Scaling		
Map / Reduce Count	Hadoop (# Connections)	SIDR (# Connections)
2781/22	61,182	2,820
2781/66	183,546	2,905
2781/132	367,092	3,031
2781/264	734,184	3,267
2781/528	1,468,368	3,760
2781/1024	2,936,736	5,106

program, assuming that writing logically contiguous data translates into contiguous file accesses.

The default partitioning approach taken by Hadoop results in each KEYBLOCK consisting of data residing throughout the total output space. Writing a collection of data spread throughout a space (sparse data) is non-trivial. There are a few common approaches, one of which is to create a file representing the entire space and using sentinel values for the missing data points. If we consider a *MapReduce* program that took this approach, a few issues become apparent. Firstly, the size of the file written by each *Reduce* task is the same, namely the size of the total output, which means that increasing the number of reducers writing out a fixed amount of output will likewise increase the total amount of bytes written but not the amount of useful data. This creates a disincentive to increase the degree of parallelization of the query, which conflicts with the results in Subsection VIII-A. Secondly, the time taken by a *Reduce* task to write its data will increase as the number of reducers increases, due to larger seeks between writes. Table II shows the results of a micro-benchmark that simulates these issues by writing sparse data to a NetCDF file. For the experiment, we fixed the amount of data a single *Reduce* task would write and then scaled the total amount of output along with the number of configured *Reduce* tasks (i.e., between measurements we doubled the size of the total output and also doubled the number of simulated reducers). The table shows the size of the file created by one reduce task and the time it took for the reduce task to write its data (data generation and meta-data operations were not included in the timing). All time measurements are averages across 10 runs, with standard deviation shown in parenthesis.

The bottom entry in the table shows the same test with a single *Reduce* task writing a contiguous portion of the output containing the same amount of useful output data as the other simulations. Since the single *Reduce* task will write the same amount of data, regardless of the number of *Reduce* tasks, the output file size and write time is constant across tests.

There are other methods for storing spare data, such as creating a collection of points and the corresponding data. This approach would create storage overhead for data written (since both the data and its coordinate are explicitly stored, rather than the coordinate being implicit, as is normally the case) that would be a function of the total output size but that overhead would be a constant scalar relative to the amount of useful data and independent of the number of *Reduce* tasks. Alternatively, many file formats provide compression, which would ameliorate the impact of having many sentinel values to some degree. However, the creation of KEYBLOCKs of equal sizes and also consisting of contiguous data presents a superior solution to these alternatives.

C. More Efficient Network Resource Usage

Table III shows how the total number of network connections between *Map* and *Reduce* tasks scales as the number of *Reduce* tasks is increased. The efficiency of SIDR is the result of each *Reduce* task only contacting *Map* tasks that have produced data assigned to its KEYBLOCK; all other *Map* tasks are ignored. Compare this to Hadoop, where every *Reduce* task connects to every *Map* task. Additionally, there is a limit of the number of concurrent connections per *Reduce* task (10 be default in Hadoop 1.0), which can create an undesirable serialization of communication. The reduction in network connections required by SIDR diminishes the likelihood of this serialization occurring in our work.

D. Variance in Reduce Completion Times

Even with the Hadoop scheduler changes described in Section VII, there is still a lot of flexibility in the how tasks are scheduled in SIDR. 10 shows the amount of variance for *Map* tasks and *Reduce* tasks for 22 and 88 *Reduce* tasks, averaged over 10 runs with error bars representing the standard deviation for each data point. In SIDR, data dependencies are small(er) global barriers, so a *Reduce* task will have at least as much variance as the set of all *Map* tasks that it depends on. Increasing the number of *Reduce* tasks makes those dependencies smaller (per *Reduce* task) and reduces the likely of a given *Reduce* task depending on several long-running *Map* tasks.

E. partition+ Performance

The partition+ function displays comparable runtimes to the default partition class used by Hadoop, *org.apache.hadoop.mapred.lib.HashPartitioner*. We created a micro-benchmark that loaded a 25 MB file of HDF5 data consisting of 6,480,000 elements into memory and then hashed them with both functions. This test represents the time taken for a *Map* task to determine the KEYBLOCK for 6.48 Million intermediate key/value pairs. Over ten runs, the default partition function has an average time of 200 ms and a standard deviation of 18.8 ms. Our partition+ function had an average execution time of 223 ms with a standard deviation of 21 ms, representing an approximately 11% decrease in

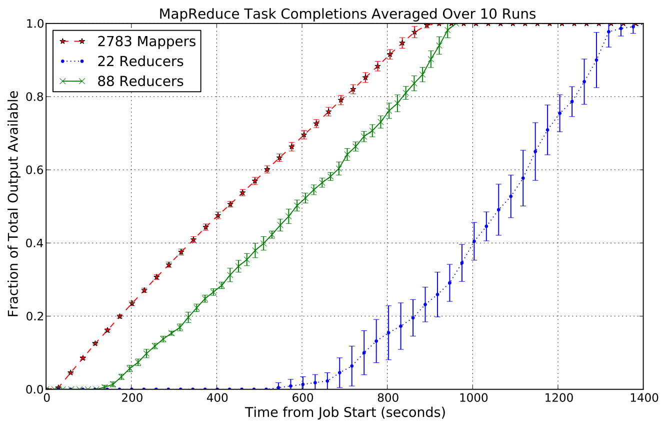

Fig. 10: SIDR *Map* task completion for 2783 tasks along with *Reduce* task completion for 22 and 88 *Reduce* tasks, averaged over 10 experiments with standard deviation presented as error bars

performance when using our partition+ function. The effect in absolute terms is negligible, given that the measured execution times of *Map* tasks ranges from tens of seconds to tens of minutes. However, we will look into optimizing the partition+ function as part of our future work.

IX. Related Work

MapReduce Online[4] focused on bringing *online aggregation*, a research topic from the database community, to *MapReduce* systems with their Hadoop Online Prototype (HOP). HOP altered the processing model of *MapReduce* so that *Map* tasks pipe their output directly to waiting *Reduce* tasks, rather than writing it to local storage from which *Reduce* tasks acquire it. In HOP, the *Reduce* task processes data as soon as it is received, with correct results made available only after all *Map* tasks complete. This approach persists the pessimistic assumption of global data dependencies; it merely begins processing data at the *Reduce* task without waiting to accumulate all of the values for a given key. By processing data as it arrives, HOP is limited to using functions that are distributive or the adoption of algorithms that produce approximate results for non-distributive functions. The authors point out several resource requirements imposed by their system and state that HOP devolves to the standard *MapReduce* processing model if any of those requirements are not met, limiting the applicability of HOP.

HOP provides early results in the form of approximations of the total output given some portion of the input having been processed(e.g., 25%, 50%, etc.). These data are useful for gaining high-level intuition about the result of the computation but any program consuming these approximate answers will need to be re-executed each time a more accurate result is produced, thereby requiring additional resources.

Another project [6] built upon HOP's approach and added the ability to process larger data sets by extending the original in-memory data structures used by *Reduce* tasks to spill to disk if the main memory on a node was exhausted. This allowed the work to be applied to larger data sets than was possible with the original HOP approach, but is otherwise similar in its benefits and limitations.

These projects found that diminishing the *MapReduce* barrier yielded performance benefits (up to 25% run-time improvement with pipe-lining tasks in [5] and 25% on average, and up to 87%, in [5, 6]). The shorter execution times are primarily attributed to a reduction in the time a *Reduce* task spent idle, as well as increased opportunity to overlap computation with IO.

Systems that store scientific data in more structured formats [23, 12, 24, 25, 10] do not face the same set of issues shown in this work because they have complete control, and therefore complete knowledge, of the data being stored. They also have their own set of drawbacks in terms of processing large-scale scientific data, including time for data ingest, the inability to reuse existing tools/code because the data is no longer available in the original format or additional storage if the original file must be preserved (perhaps for archival or analysis with other tools). Other attempts at processing scientific data with *MapReduce* have opted to convert the data into text files or to strictly interact with files as entire entities [13, 1, 2, 3, 11], requiring users to split large files into smaller files if data locality is desired. Neither of these present feasible long-term solutions as they require additional data movement and reformatting or interact with data at too coarse of granularity and thereby suffer from sub-optimal performance.

SciDB [14] currently requires data to be ingested and stored in its internal formats but they aspire to eventually support in-situ processing of scientific data in its native file-format.

We consider this work to be complementary to more expressive models for distributed computation, such as Pregel[26] and Dryad[27], as they can leverage the methods presented in this paper within their sub-computations.

X. Conclusions and Future Work

The work presented in this paper explores the benefits of extending Hadoop to more efficiently partition data, derive the actual data dependencies between *Map* and *Reduce* tasks, and using that knowledge to make more informed scheduling decisions. By augmenting Hadoop to do this, it can start *Reduce* tasks once their actual dependencies are met, thereby producing final results for portions of the output space while only a fraction of the input has been read. Total job run-time is reduced and the resultant output is organized into dense structures of approximately equal sizes. We have also shown how the *MapReduce* model is amenable to this approach to structured data without sacrificing generality or correctness.

Building upon the work in this paper, we plan on investigating methods of providing failure recovery that do not require intermediate data to be persisted to disk-based storage, as is currently done in Hadoop. SIDR's ability to produce early, complete results for portions of the total output space should incorporate well into pipe-lined computations, which we will also investigate as part of our next project. We are expanding the SciHadoop code to support additional file formats as well as compatibility with the Ceph distributed file system [28]; the latter enabling Hadoop programs to write their output to scientific file formats (Hadoop's present limitation of only offering append semantics when writing files precludes creating output via most scientific file formats).

Acknowledgments

We would like to thank our collaborators: Meghan (Wingate) McClelland, Gary Grider, James Nunez and Aaron Torres at Los Alamos National Laboratory; Maya Gokhale at Lawrence Livermore National Laboratory; John Bent at EMC; Russ Rew at Unidata; and our colleagues in the Systems Research Lab and Institute for Scalable Scientific Data Management at UC - Santa Cruz. Funding provided by DOE grant DE-SC0005428 and (partially) NSF grant #1018914.

References

[1] J. Ekanayake, *et al.*, "Mapreduce for data intensive scientific analyses," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, dec. 2008.

[2] T. Gunarathne, *et al.*, "Mapreduce in the clouds for science," *Cloud Computing Technology and Science, IEEE International Conference on*, vol. 0, 2010.

[3] H. Zhao, *et al.*, "Parallel accessing massive NetCDF data based on MapReduce," in *Web Information Systems and Mining*, ser. Lecture Notes in Computer Science, 2010.

[4] T. Condie, *et al.*, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. USENIX Association, 2010.

[5] ——, "Online aggregation and continuous query support in mapreduce," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10. ACM, 2010.

[6] A. Verma, *et al.*, "Breaking the mapreduce stage barrier," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, sept. 2010.

[7] Fits homepage. http://fits.gsfc.nasa.gov/.

[8] Netcdf homepage. http://www.unidata.ucar.edu/software/netcdf/.

[9] Hdf5 homepage. http://www.hdfgroup.org/HDF5/.

[10] S. Loebman, *et al.*, "Analyzing massive astrophysical datasets: Can pig/hadoop or a relational dbms help?" in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, 31 2009-sept. 4 2009.

[11] Y. Wang, *et al.*, "SciMATE: a novel MapReduce-like framework for multiple scientific data formats," ser. CCGrid '12. ACM, 2012.

[12] M. Ivanova, *et al.*, "MonetDB/SQL meets SkyServer: the challenges of a scientific database," in *Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, ser. SSDBM '07, 2007.

[13] J. Ekanayake, *et al.*, "DryadLINQ for scientific analyses," in *e-Science, 2009. e-Science '09. Fifth IEEE International Conference on*, dec. 2009, pp. 329 –336.

[14] P. G. Brown, "Overview of SciDB: large scale array storage, processing and analysis," in *Proceedings of the 2010 International Conference on Management of Data*, ser. SIGMOD '10, 2010.

[15] J. B. Buck, *et al.*, "Scihadoop: array-based query processing in hadoop," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011.

[16] Grib homepage. http://www.ecmwf.int/products/data/software/grib_api.html.

[17] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA, 2004.

[18] M. Felice Pace, "BSP vs MapReduce," *ArXiv e-prints*, Mar. 2012.

[19] H. Karloff, *et al.*, "A model of computation for mapreduce," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '10, 2010.

[20] S. Lattanzi, *et al.*, "Filtering: a method for solving graph problems in mapreduce," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11, 2011.

[21] Hadoop homepage. http://hadoop.apache.org/.

[22] SciHadoop source code on github.com. https://github.com/four2five/SciHadoop/.

[23] E. Soroush, *et al.*, "Arraystore: a storage manager for complex parallel array processing," in *Proceedings of the 2011 international conference on Management of data*, ser. SIGMOD '11, 2011.

[24] P. Baumann, *et al.*, "The multidimensional database system RasDaMan," *SIGMOD Rec.*, pp. 575–577, June 1998.

[25] A. van Ballegooij, *et al.*, "Distribution rules for array database queries," in *Database and Expert Systems Applications*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3588.

[26] G. Malewicz, *et al.*, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 international conference on Management of data*, ser. SIGMOD '10, 2010.

[27] M. Isard and Y. Yu, "Distributed data-parallel computing using a high-level programming language," in *Proceedings of the 35th SIGMOD international conference on Management of data*, ser. SIGMOD '09. ACM, 2009.

[28] S. A. Weil, *et al.*, "Ceph: a scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06, 2006.