2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Bi-Hadoop: Extending Hadoop To Improve Support For Binary-Input Applications

Xiao Yu and Bo Hong
School of Electrical and Computer Engineering
Georgia Institute of Technology

***Abstract*—The MapReduce programming model, along with its open-source implementation - Hadoop - has provided a cost effective solution for many data-intensive applications. Hadoop stores data distributively and exploits data locality by assigning tasks to where data is stored. Many data-intensive applications, however, require two (or more) input data for each of their tasks. Such applications pose significant challenges for Hadoop as the inputs to one task often reside on multiple nodes, and Hadoop is unable to discover data locality in this scenario. This often leads to excessive data transfers and significant degradations in application performance. In this paper, we present Bi-Hadoop, an efficient extension of Hadoop to better support binary-input applications. Bi-Hadoop integrates an easy-to-use user interface, a binary-input aware task scheduler, and a caching subsystem. Extensive experiments show that Bi-Hadoop can significantly improve the execution of binary-input applications by reducing the data transfer overhead, and outperforms existing Hadoop by up to 3.3x.**

***Keywords*-MapReduce; Hadoop; Data Locality;**

I. Introduction

Large-scale data intensive computing has become indispensable for many applications to gain insights from increasing volumes of data. The MapReduce [1] programming model, along with its open-source implementation Hadoop [2], has provided a cost effective solution for such data processing needs. Hadoop is designed to support data intensive applications on clusters of hundreds or thousands of compute nodes.

In Hadoop, applications consist of map and reduce tasks that operate on data stored on the HDFS file system [2]. HDFS breaks large files into a series of blocks and randomly distributes the blocks over the compute nodes. Fault tolerance is achieved by duplicating the blocks. Compute nodes can be labeled as *local* or *remote* for each block. Such locality information is utilized to schedule the tasks: priority is given to local nodes over remote nodes. This can lead to effective reduction in network transfer overheads, which is especially important for data-intensive applications due to the huge volumes of data that needs to be processed.

However, the locality-awareness of Hadoop is based on a relatively strong assumption that *a task is expected to work on a single data split*. In practice, a split typically consists of one data block, or a part of it. After all, this is what allows Hadoop to label a compute node as *local* or *remote* for scheduling purposes. This is in accordance with the MapReduce programming model, which defines one map task over each logical data split and thus requires users to describe the mapper function as a unary operator, applicable to only one single logical data split.

The unary-input requirement works well for many applications such as document processing. However, many other applications require more flexible operators. For example, a task in a pattern matching application would naturally take two inputs: one record of the template data, and another record of the stored data. For such applications, the unary input oriented Hadoop system has multiple limitations: (1) Developers need to work around the unary input requirement, which makes it less natural to program the applications. (2) When a workaround method is used, the built-in locality awareness of Hadoop becomes less effective or non-effective. (3) As binary-input tasks often share their data blocks, there are many unique locality optimization opportunities in these applications that cannot be exploited by existing Hadoop.

Motivated by the above observation, we study Bi-Hadoop, an extension of the Hadoop system to improve the execution of binary-input applications.

We make the following contributions in this paper:

1) We design an easy-to-use interface for users to describe the association between a task and its inputs.
2) We develop a task scheduling algorithm that is able to exploit data locality for binary-input applications.
3) We design and implement a caching mechanism to accelerate data reads. The caching mechanism is an integral part of our extension that materializes the improved data locality exposed by our scheduling algorithm.

Extensive experiments were conducted to verify the effectiveness of Bi-Hadoop. The performance of the scheduling algorithm is tested against a wide range of binary-input task patterns, which shows that our algorithm reduces remote data reads by up to 48% when compared with existing Hadoop scheduling algorithm. Experiments on two actual applications (a pattern matching application and PageRank) were conducted on a 64-node Amazon EC2 cluster. The results show that our method improve the execution speed of these applications by up to 3.3x over the native Hadoop system.

The rest of the paper is organized as follows: Section II reviews the background of the Hadoop system and its locality strategy and lists related works. We then presents an overview of our extension in Section IV. Section IV-C presents the detailed design of our extension. Section V illustrates the experimental results. Section VI presents our conclusion and future work.

978-0-7695-4996-5/13 $26.00 © 2013 IEEE
DOI 10.1109/CCGrid.2013.56

II. Background and Related Works

Due to its practical importance for data-intensive applications, the MapReduce programming model has attracted several implementation efforts (e.g. [3]–[5]), including Hadoop [2], which is a widely used open source implementation of MapReduce. Hadoop consists of two parts: Hadoop File System (HDFS) and the Hadoop MapReduce system.

HDFS is the default file system designed for fault-tolerant distributed storage. Documents in HDFS are partitioned into blocks, replicated and distributed to the compute nodes (called DataNodes) in the system. A centralized server node called the NameNode maintains the file system data structure and tracks the location of each replica block. In the Hadoop MapReduce system, a centralized job submission and scheduling server called the JobTracker is responsible for receiving client job submission and scheduling tasks onto TaskTrackers. TaskTrackers are services collocated on the same physical compute nodes as DataNodes, and are responsible for executing the tasks.

A job in the MapReduce system consists of a collection of map tasks and reduce tasks. Task inputs are in the form of key/value pairs, and are arranged into splits. Splits are typically smaller than or equal to data blocks in HDFS and are the basic indivisible unit to specify and schedule tasks. Hadoop segments the input HDFS data files into splits, and feeds them into the tasks. As discussed in Section I, this one-task-one-split assumption is critical for Hadoop to exploit locality-aware scheduling for unary-input applications.

As locality is key to the performance of Hadoop systems, it has attracted a lot of research attention recently. Haloop [6] extended Hadoop to improve the execution of iterative applications and Twister [7] proposed a light-weight MapReduce runtime for these applications. The two methods shared some similarities - they separated user data into a static set and a dynamic set so that the static user data can be cached and reused across iterations. Our work differs significantly as we target general applications with binary-inputs, rather than the specific type of iterative applications. In CoHadoop [8], an extension of HDFS was proposed to exploit co-locality among data blocks to reduce network data transfer. However, their method did not address the task scheduling problem, which is critical to systematically exploit the locality characteristics of applications. Instead, the co-locality information was manually discovered and configured. In Scarlett [9], the replication factors were adjusted for *hot* HDFS files to alleviate competition for such files. But like CoHadoop [8], this study did not consider the scheduling of tasks and cannot exploit locality for binary-input applications.

Several other research works have studied data placement in clusters. Heuristics are proposed in [10], [11] to optimize file access for scientific workflow in data centers. AllPairs [12] proposes a system for the all pair pattern applications and discusses data placement strategies. Our study also addresses the data placement problem, but we discuss the problem in the context of general MapReduce jobs that require two inputs for each task.

It is worth pointing out that many other aspects of Hadoop have also been studied to improve programmability (e.g. [13]), performance (e.g. [14]), or database applications(e.g. [15]).

III. Motivation for Bi-Hadoop

A. Lack of Support for Binary-Input Applications in Hadoop

While Hadoop has good support for unary input applications, it does not handle binary-input applications very well. Take tiled matrix vector multiplication algorithms for example, a task would naturally consist of one matrix block and one vector block. Since existing Hadoop only supports unary input tasks, users need to use one of the following workaround methods to program such applications:

- use an extra level of indirection in the input format to give the system an illusion of single input split that is actually one matrix block plus one vector block. This workaround method involves user defining a new InputFormat class which would significantly increase programming difficulty.
- use the Hadoop *distributed cache* utility to duplicate one input set at `all` the nodes so they can locally access this set of data, and Hadoop only needs to handle the other (one) input set. This method will not work when input data set exceeds the storage capacity of the distributed cache. Furthermore, duplication of an entire data set is often wasteful since each node will most likely access only part of the set.
- use an extra round of MapReduce job to concatenate the two input blocks for each task, and save the new *merged* data blocks onto HDFS for the actual MapReduce job. This workaround method needs to move a significant amount of data and cause expensive overheads. Additionally, if a block is to be shared by multiple tasks, the block will be duplicated multiple times in this data-preprocessing stage, which further increases the overheads.
- use two file system calls directly in the user-supplied mapper functions to read the two splits. This workaround method is easy to program. But the data reads are invisible to the Hadoop system. Without knowing which data blocks may be accessed by a task, the Hadoop scheduler cannot perform locality-aware scheduling, which will result in excessive data transfer overheads.

B. Data Locality in Binary-Input Applications

Tasks in binary-input applications often share their input data blocks. For example, in tiled matrix multiplication algorithms, the same block from matrix A will multiply with multiple blocks from matrix B. For another example, in a genome comparison application, an individual genome may need to be compared against multiple other genomes in a database. Apparently, the tasks (multiplying two matrix blocks, or comparing a pair of genomes) share their inputs.

This introduces a unique type of data locality: if tasks can be grouped together such that their overall data footprint can be minimized, then the group of tasks can benefit from reduced data transfer if they are co-assigned to the same compute node. The amount of data transfer can be further reduced if the assignment can utilize data blocks that are local to the compute

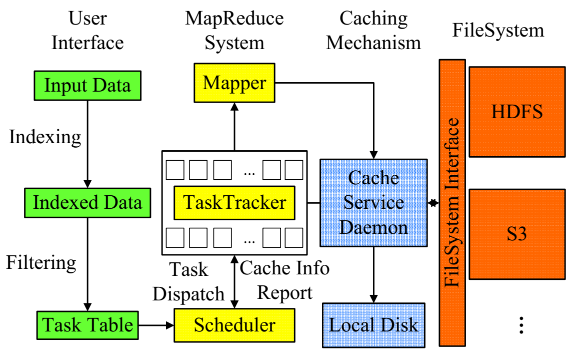


Fig. 1: Bi-Hadoop Extension System Overview

node. Note that existing Hadoop is unable to exploit such data locality as it is designed with unary-input applications in mind.

Clearly it is necessary to improve Hadoop to better support binary-input applications. In the following discussion, we will present our proposed Hadoop extension that can significantly improve the data locality for such applications.

IV. Bi-Hadoop Design

A. Design Challenges

Our extension aims to address the following design challenges:

- *Programmability*. We need to extend the programming interface so that users can specify the inputs to the tasks. We want the interface to be easy to program so that users can focus on the main functionality of their applications, rather than dealing with the interface itself.
- *Transparency*. The next goal is to make it easier for users to achieve high performance. For this reason, we do not want the users to be even aware of the data locality issues. Bi-Hadoop is simply an improved version of Hadoop that can execute binary-input applications faster. For this reason, we reject all design alternatives that require users to track/handle the locations of the data blocks.
- *Non-intrusiveness*. Bi-Hadoop takes certain amount of memory resources at the compute nodes to speed up binary-input applications. We want Bi-Hadoop to be fully bypassable when the system is executing unary-input applications, with close-to-zero overheads. Furthermore, if a user application tries to use up the available memory, we want Bi-Hadoop to gracefully disappear in the background and yield the resources to the user application.

B. Design Overview

Figure 1 illustrates an overview of Bi-Hadoop, which contains the following components: (1) the input interface, (2) the caching subsystem, and (3) the binary-input locality-aware scheduler.

- *Input interface*. This component assigns IDs to splits. Bi-Hadoop inherits the default Hadoop output format and adds a hook so that an ID can be designated to each split. In Bi-Hadoop, tasks are generated by calling a user-defined filter function that specifies which two splits would form a valid task. Tasks are internally represented as a 2-D matrix using either dense or sparse format depending on the application. The IDs assigned by the user will be used to identify the splits in the user filtering functions as well as by the scheduler.
- *Scheduler*. The scheduler first obtains the task representation by applying the filter function from the user interface, then gathers information about the locations of the input data blocks (which DataNode has which blocks), and subsequently calculates a locality-optimized execution schedule. During the course of the execution, the scheduler monitors the content of the caches on the fly, and fine-tunes the schedule according to dynamic locality information supplied by the caching subsystem.
- *Caching subsystem*. This component runs on each compute node and is designed to cache the input splits accessed by existing tasks (with the expectation that they will also be needed by subsequent tasks). The caching subsystem sits between MapReduce system and HDFS system, therefore it is user transparent. It supplies a split to the requesting map task if the split is in the cache, and will seamlessly resolve to the native HDFS data read mechanism when the requested split is missing in the cache.

In the following, we will discuss the details of Bi-Hadoop design, and analyze why it is capable of exploring data locality for binary-input applications.

C. Design Details

1) User Interface

The Bi-Hadoop user interface is designed to assign IDs to splits by letting the user name splits with strings.

```
// The Bi-Hadoop filter interface
public interface BiHFilter {
  public boolean accept(String split0Id, String
      split1Id);
}

// A usage example: matrix-vector multiply
public class MatVecMulFilter implements
    BiHFilter {
  public boolean accept(String split0Id, String
      split1Id) {
    if (!isAMatrix(split0Id)) return false;
    if (!isAVector(split1Id)) return false;
    colId = getMatrixColId(split0Id);
    rowId = getVectorRowId(split1Id);
    if (colId == rowId) return true;
    return false;
  }
}
```

Listing 1: Bi-Hadoop User Interface for Defining Tasks

An application can have one of the following input data formats: (1) Each user input file has a granularity small enough to define a split. In such case, file names can be naturally used as the split ID. Users can set a flag in Bi-Hadoop to specify such configuration. (2) Each input file is structured and contains multiple file splits. In this case, users just need to provide an ID file according to a predefined format, listing split IDs and the file segments that each split maps to. (3) The input files are unstructured. In this case, users need to

	1, V_0	2, V_1	3, V_2	4, V_3
1, M_{00}	1			
2, M_{10}	1			
3, M_{01}		1		
4, M_{11}		1		
5, M_{02}			1	
6, M_{12}			1	
7, M_{03}				1
8, M_{13}				1

Fig. 2: Incidence Matrix Example

do a preprocessing step to structure their input files. Similar methods was also used in other works [8], [15], [16] to pre-process unstructured data. The resulting structured files can then be handled as in case (2). We extend the default Hadoop OutputFormat class to provide simple utilities so that the name files can be easily generated along with the preprocessing step.

Users specify the map tasks by customizing a filter class, which returns true if a pair of split IDs form a task, and false otherwise. Listing 1 illustrate the simple interface and a usage example of matrix vector multiplication. Users can manipulate the ID strings in a customized fashion (such as $getMatrixColId()$ in Listing 1) to identify the file split and form the tasks.

2) The Locality-aware scheduler

The scheduler weaves all the components together in Bi-Hadoop. Once a MapReduce job is submitted, the scheduler first creates an internal presentation of the tasks. The scheduler will then monitor the locality of the data splits (disk replicas and copies in the cache subsystem), exploit data sharing pattern among the tasks, and assign tasks to optimize data localities for the tasks. Scheduling in Bi-Hadoop is performed in three phases:

Phase 1: Task Generation

Binary-input applications have two sets of input splits, A and B, and a task will take a split from A and another one from B. Note that A and B may overlap, either partially or completely.

In this phase, we run the user-supplied filter function (discussed in the user interface) and generate an internal presentation of the tasks in the form of an incidence matrix I . The matrix uses one row (and column) to represent a split in A (and B). If there exists a task whose input splits are $a \in A$ and $b \in B$, then we have $I(a, b) = 1$ indicating the presence of this task. Note that the matrix may be dense or sparse depending on the characteristics of the job. Subsequently, the storage of matrix I will take the dense or sparse forms accordingly.

Fig. 2 illustrates an example of the incidence matrix for matrix-vector multiplication. The matrix is of size 2 by 4 blocks and the vector is 4 blocks. Each value 1 in the matrix indicates a task that multiplies a matrix block with a vector block.

Phase 2: Static Task Grouping

Taking the incidence matrix as input, we partition the rows and columns into groups such that tasks within the same group share their input file splits. The algorithm is listed in Alg. 1.

In the algorithm, groups are formed such that a row should be included in a group if 80% of the columns from this row has the same value(0 or 1) as the columns from the representative

Algorithm 1 Algorithm for the Static Grouping Phase

```
Input:
    $taskMatrix$: the incident matrix of tasks.
Output:
    $groupList$: list of set of rows and columns

 1: for all $row \in taskMatrix$ do
 2:     $foundGroup \leftarrow False$
 3:     for all $group \in groupList$ do
 4:         $repr \leftarrow group.repr$
 5:         $smaller \leftarrow getSmaller(row, repr)$
 6:         if $size(row \cap repr) > threshold * size(smaller)$
            then
 7:             $group.add(row)$
 8:             $foundGroup \leftarrow True$
 9:         end if
10:     end for
11:     if $foundGroup == False$ then
12:         $groupList.addNewGroup(row)$
13:     end if
14: end for
```

row in the group. The representative row is chosen as the row in the group with the most 1's. Similarly, the algorithm is also used to form column grouping. Both row and column grouping results will be used in the next phase.

The static grouping phase provides an insight into the relation between tasks: tasks from the same group are likely to share (some) input splits, and if we assign them to a common compute node, we will see reduced data transfers.

Phase 3: Dynamic Task Dispatching

The dynamic task dispatching phase is executed during run time, it decides which node should execute which tasks, and the goal is to reduce data transfers while maintaining load balancing across the compute nodes. To achieve the goal, this phase uses the static grouping result as a guide to reduce data transfers, and further considers the following input information: (1) what replicas does each node have? (2) what splits is a node currently caching?

Before describing our dynamic dispatching algorithm, let us define a new term: a *task pack*, which is the set of tasks that will be executed together by a compute node. Task pack is the unit of actual task dispatching. Our scheduler takes two steps to form a task pack: first we choose a group from Phase 2 that most splits local, then we pick a pack from the group so that it can fit into the cache of the local compute node.

We use the following criteria when forming task packs: (1) the number of tasks in a pack should not exceed the total number of tasks dividing the number of nodes; (2) the difference between the number of row splits and the number of column splits is small; (3) at most half of the cache will be used for row splits or column splits. The first criterion is a heuristic to ensure load balance; the second criterion is to ensure data reuse and thus reduce data transfers; and the third criterion is a heuristic to the following optimization problem: maximize $n_1 * n_2$ subject to the constraint that $s_1 n_1 + s_2 n_2 \leq C$, where n_1, n_2 denote the number of row and column splits, $n_1 * n_2$ denote the number of tasks to be

Algorithm 2 Algorithm for the Dynamic Dispatching Phase

Input: $groupList$: list of set of rows and columns of task matrix
$cache$: cache status of the TaskTracker
$replica$: replica status of the TaskTracker
Output: $pack$: a pack of tasks

```
1: /* Find best group */
2: $local \leftarrow cache + replica$
3: $max \leftarrow 0$
4: for all $group \in groupList$ do
5:     $count \leftarrow \#blocks$ in both $local$ and $group$
6:     if $count > max$ then
7:         $best \leftarrow group$
8:         $max \leftarrow count$
9:     end if
10: end for
11: /* Packing */
12: calculate $maxRowSize, maxColSize, maxSize$
13: $numRows \leftarrow 0$
14: $numCols \leftarrow 0$
15: $size \leftarrow 0$
16: for all $rowsplit \in best.rows()$ do
17:     if $numRows \geq maxRowSize$ then
18:         break
19:     end if
20:     if $size \geq maxSize$ then
21:         break
22:     end if
23:     if not $replica.contains(rowsplit)$ then
24:         $size \leftarrow size + rowsplit.size()$
25:     end if
26:     $rowSplits.add(rowsplit)$
27:     $numRows \leftarrow numRows + 1$
28: end for
29: /* do the same for columns */
30: ...
31: for all $rowsplit \in rowSplits$ do
32:     for all $colsplit \in colSplits$ do
33:         if $hasTask(rowSplit, colSplit)$ then
34:             $pack.add(getTask(rowsplit, colsplit))$
35:         end if
36:     end for
37: end for
38: return pack
```

executed and s_1, s_2 denote the size of row and column splits and C denote the cache capacity.

Alg. 2 outlines this dynamic dispatching phase. Task packs will be created such that (1) tasks in a pack share their input splits; and (2) the input file splits are likely to be already has a local replica or in cache. Tasks in a pack are then scheduled onto the corresponding compute nodes (represented by its TaskTracker) one by one.

To take load balance into account, when no more packs can be formed for an idle TaskTracker, i.e., when all tasks have been already assigned to some pack, our scheduler falls back to the default Hadoop scheduling algorithm so that this idle TaskTracker can steal tasks from some pack that is assigned to some other TaskTracker. While this gives the TaskTracker some work to do, it may cause less than optimal data transfers (as this is an out-of-pack task). Nonetheless, this is no worse than what the native Hadopp would do.

3) Caching Subsystem

The caching subsystem has two components: a file handler object and a service daemon. The handler object is constructed when opening files. The service daemon sits on top of the file system abstraction of each compute node (between MapReduce and HDFS systems). We design it in this way because we want the service daemon to remember the caching history among jobs. Furthermore, the service daemon can function for any Hadoop supported file system.

When users want to open a cache-enabled file, they use an openCachedReadOnly function. The function returns our specialized file handler and users read data as usual using this handler. The openCachedReadOnly function accepts an optional versionID parameter besides the usual path parameter. We expect users to change this versionID if the data is modified. If the cached version is not equal to the user provided version, the block will be re-fetched. The handler checks if the current reading position is within the cached block or local replica boundary. If yes, the handler continues to read, otherwise, the handler sends a cache block request to the service daemon using a remote procedure call(RPC). Upon receiving the RPC response, the handler updates its status and proceeds to read.

The service daemon serves handlers' requests, manages the cached blocks and reports caching status to the TaskTracker for scheduling. When it receives a caching request, it checks if the required data is in the cache. If not, the daemon uses the usual file system API(such as HDFS API) to read the data and saves blocks into local file system. The maximum block size is fixed(default 64M) for our design. If the underlying file system has a block size larger than our cache block size configuration, that block is segmented into smaller blocks. The cached blocks are evicted using a least recently used policy if the capacity is reached. Each time when a TaskTracker needs to send a heartbeat, it also reports the status of the cache with the heartbeat.

D. Developer Transparency

With Bi-Hadoop, users will only need to perform one extra piece of work than with existing Hadoop: specifying which two splits form a task. Other than this, they just write Hadoop programs as they currently do: focusing on the mapper and reducer functions. The three phases of our scheduler as well as the caching subsystem are transparent to the users. The users do not need to know how tasks are grouped together to reduce their data footprint, or how task packs are formed to take advantage of local data replicas and cached splits, or how splits enters and leaves the caches. Bi-Hadoop executes in the background to accelerate the execution of binary-input applications.

E. *Non-intrusiveness*

The non-intrusiveness of Bi-Hadoop has two aspects:

1) If the application is unary-input as with existing Hadoop applications, our scheduler will detect the non-existence of the user-supplied input filter, and immediately hand everything over to the existing Hadoop scheduling subsystem. The overhead in this case will be minimal. The caching service will still run on the compute nodes. But they will not be activated and thus will only consume a minimum amount of resources.
2) Bi-Hadoop also consumes certain amount of memory when the caching subsystem attempts to cache file blocks at the nodes. To this end, Bi-Hadoop takes advantage of the existing Linux file caches, which by itself is elastic, meaning that it yields memory to user programs as they demand more memory. In such cases, Bi-Hadoop will seamlessly yield memory to user programs (by caching less). Note that the application will not benefit from the faster speed provided by caches, but they will still benefit from our task grouping process (with reduced data footprint).

V. Experimental Results

Extensive experiments were conducted to evaluate Bi-Hadoop. We performed (1) simulation-based studies to verify the effectiveness of our scheduling algorithm, and (2) execution of real applications to demonstrate the performance improvement over the existing Hadoop system.

A. *Effectiveness of the Scheduling Algorithm*

For this set of experiments, we studied a wide range of task sharing patterns to evaluate how well our scheduling algorithm can reduce remote data reads. We simulated a 64-node Hadoop system running HDFS with the following parameters:

- HDFS data blocks were 64MB (the default Hadoop setting), and were randomly distributed across the cluster with a uniform distribution.
- Compute nodes became available (and subsequently request new tasks) in a random order. This emulated random task execution time.

The following task input patterns were evaluated: (1) `All Pair Pattern(ap)`, which has a full incident matrix;(2) `Half All Pair Pattern(hap)`, which has a triangular incident matrix; (3) `Diagonal Block Pattern(db)`, which has a diagonal block incident matrix; (4) `Circular Shuffled Diagonal Block Pattern(csb)`, which circularly shuffles the rows and columns of diagonal block matrix; (5) `Random Shuffled Diagonal Block Pattern(rsb)`, which randomly shuffles the rows and columns of diagonal block matrix; (6) `Iterative Diagonal Block Pattern(idb)`, which is a series of jobs with Diagonal Block Pattern; (7) `Random Pair Pattern(rp0.x)`, which whether an entry is 1 or 0 is randomly generated according to the density level x; (8) `Iterative Random Pair Pattern(irp0.x)`, which is a series of jobs with Random Pair Pattern. Among the

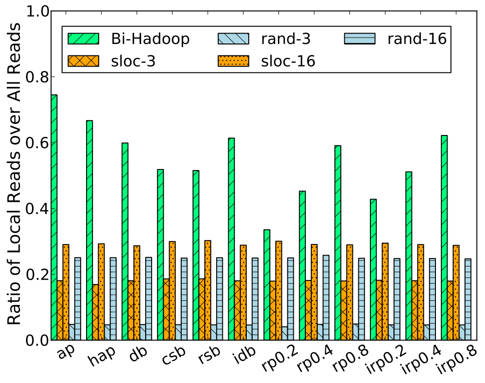


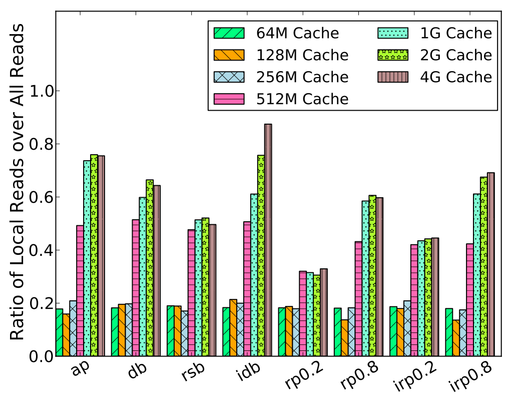
Fig. 3: Bi-Hadoop Scheduling Performance Comparison with Default Hadoop. `sloc-*` and `rand-*` are workaround methods of Hadoop with replication factor set to 3 and 16

patterns, All Pair Pattern, Half All Pair Pattern and Iterative Diagonal Block Pattern are extracted from real applications, the others are synthetic patterns.

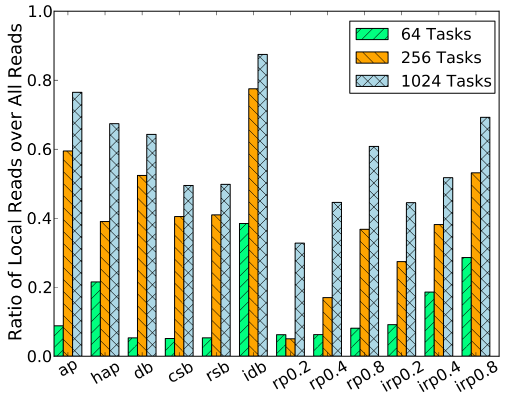
Fig. 3 compares the scheduling algorithm between Bi-Hadoop and the default Hadoop. As discussed previously, the default Hadoop needs to use a workaround method for binary-input applications. We tested the following two workaround methods: (1)`sloc`, which updates Hadoop's InputFormat such that Hadoop can schedule according to the locality of one data split (still cannot do anything about the second split), (2) `rand`, in this workaround method, the two splits are directly accessed using file system API calls inside the map functions, and Hadoop can only randomly assign the tasks because it does not know what data a task may request.

In this set of experiment, there were 1024 tasks, each having two 128MB input splits. The cache capacity was set to 1024M bytes. As is shown in Fig. 3, the Bi-Hadoop scheduling algorithm consistently outperforms the Hadoop scheduling methods (both sloc and rand), by up to 75%. In an attempt to help Hadoop increase the ratio of local reads, we increased the Hadoop replication factors from 3 to 16 (sloc-16 and rand-16 in Figure 3). But Hadoop only benefited from a moderate increase of local reads, which are still up to 60% lower than Bi-Hadoop scheduling. This is because our scheduling and caching mechanism tend to group data blocks that are co-accessed or re-used by the tasks. Blindly increase the replication factor does not help much, for example, the probability that two data blocks are co-located on one node is only 25% even for a replication factor high as 16 in a 64-node system.

We then studied the impact of cache capacity and the results are shown in Fig. 4(a). The experiments were configured with 1024 tasks, each having two 128MB input splits. The per node cache capacity varied between 64MB and 4096MB. The results show that the ratio of local reads stayed relatively low at 20% when the cache is smaller than 256MB, which is reasonable because 256MB is the minimum size to accommodate the input data for at least one task. Once cache capacity passes this threshold, the hit rate increases with larger caches, and can ensure that ∼50-70% data reads are locally satisfied when we



(a) Cache Capacity



(b) Total Number of Tasks

Fig. 4: Performance Impact of Cache Capacity And Number of Tasks

have 1024MB or larger caches.

The number of tasks also has an impact on our Bi-Hadoop scheduling algorithm. This is illustrated in Fig. 4(b). We varied the number of tasks from 64 to 1024, and the results show that a larger number of tasks leads to more local data reads. This is because more tasks per node provides more opportunity to exploit data sharing among the tasks.

In summary, this set of experiments show that, compared with the existing Hadoop task scheduling strategy, our Bi-Hadoop scheduling algorithm can significantly reduce the amount of remote data reads for binary-input applications. And this is achieved with a relatively low requirement on the cache capacity. Furthermore, our scheduling algorithm works better when the application scales up: the more tasks there are, the better our algorithm can exploit the data sharing characteristics among the tasks.

B. *Experiments with Actual Applications*

For this set of experiments, we ran real applications using a cluster of Amazon EC2 medium instances. Each instance was configured with 3.75GB memory, 2 EC2 compute units, 410 GB instance storage and runs 64-bit Amazon Linux AMI. We applied our extension on Hadoop stable release 1.0.4. HDFS replication factor is set to 3(default setting) if not mentioned otherwise.

Two applications were tested: (1) a pattern matching application that features the all pairs sharing pattern, and (2) PageRank where tasks share inputs with an iterative block diagonal matrix pattern.

1) *Pattern Matching*

This application is abstracted from the earthquake analysis problem studied in [17] where each template earthquake waveform is compared against each recorded waveforms (to automatically identify aftershocks). The programming implementation has two sets of floating-point arrays, A - the templates, and B - the recorded waveforms. Each array in A will perform a correlation analysis against each array in B. For the baseline Hadoop implementation, we implemented a workaround solution that combines two arrays from set A and B as a split. This has the similar effect as the `sloc` scheduling strategy we simulated.

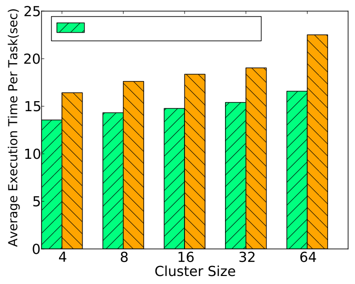
Figure 5(a) compares the performance between Bi-Hadoop and the baseline. In this set of experiments, all arrays were 256M bytes. Set A always contained 4 arrays. The number of arrays in set B was set to 4 times the number of nodes. Cache size was set to 2GB per node. Figure 5(a) shows that Bi-Hadoop out-performs the baseline in all cluster sizes by up to 26.3%. The speedup increases with cluster size (from 17.4% at 4 nodes to 26.3% at 64 nodes), which shows Bi-Hadoop scales better than the Hadoop baseline.

Fig. 5(b) shows the impact of cache size. Obviously the execution time decreases when the cache size increases. Furthermore, changing cache size from 256MB to 1GB already accounts for more than 70% of the speedup, which verifies the simulation results in Section V-A that our algorithm has a relatively low requirement on cache capacity to exploit the data sharing patterns. Figure 5(c) compares our extension with a naive strategy that simply increases the replication factor of HDFS. It can be seen that simply increasing the number of replication helps very little unless the replication factor reaches 12 replicas per block, which will unfortunately need an excessive amount of storage for the replicas and still cannot outperform Bi-Hadoop that only uses 1GB cache per node.

2) *PageRank*

PageRank is a well-known algorithm to rank web pages according to their significance. It is heavily used by search engines and is often implemented using MapReduce. The core of the algorithm is an iterative matrix vector multiplication, where the matrix represents the link connectivity among the web pages, and the vector represents the ranking of the pages.

The baseline Hadoop implementation is an optimized variation of a common implementation [6], [16], [18] that uses two MapReduce jobs in each iteration. As Bi-Hadoop extension naturally supports binary-input applications, we were able to design a matrix-vector multiplication based algorithm. The same Amazon EC2 Cloud instances were used for this set of experiments. We used a semi-synthetic dataset called


Average Execution Time Per Task(sec)
0
5
10
15
20
25
4
8
16
32
64
Cluster Size