Optimal Algorithms for Cross-Rack Communication Optimization in MapReduce Framework

Li-Yung Ho
Institute of Information Science
Academia Sinica,
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
Email: lyho@iis.sinica.edu.tw

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Taipei, Taiwan
Email: wuj@iis.sinica.edu.tw

Pangfeng Liu
Department of Computer Science and Information Engineering,
Graduate Intitute of Networking and Multimedia,
National Taiwan University
Taipei, Taiwan
Email: pangfeng@csie.ntu.edu.tw

***Abstract*—MapReduce is a widely used data-parallel programming model for large-scale data analysis. The framework is shown to be scalable to thousand of computing nodes and reliable on commodity clusters. However, research has shown that there is room for performance improvement of the MapReduce framework. One of the main performance bottlenecks is caused by the all-to-all communication between mappers and reducers, which may saturate the top-of-rack switch and inflate job execution time. Reducing cross-rack communication will improve job performance. In current MapReduce implementation, the task assignment is based on the pull-model, in which cross-rack traffic is difficult to control. In contrast, the MapReduce framework allows more flexibility in assigning reducers to the computing nodes.**

In this paper, we investigate the reducer placement problem (RPP), which considers the placement of reducers to minimize cross-rack traffic. We devise two optimal algorithms to solve RPP and implement the algorithms in the Hadoop system. We also propose an analytical solution for this problem. Our experiment results with a set of MapReduce applications show that our optimization achieves 9% to 32% performance improvement compared with the unoptimized Hadoop.

***Keywords*-cloud computing, MapReduce optimization, cross-rack communication, optimal algorithm, network load balancing**

I. Introduction

Data parallelism programming models, such as MapReduce [1], Hadoop [2] and Dryad [3], have been widely used in analyzing Tera-byte scale data as well as routine data processing in cloud environment nowadays. In particular, Hadoop, an open-source implementation of MapReduce, has become more and more popular in organizations, business companies and institutes. For example, the A9 of Amazon [4] uses Hadoop to power search in goods and books, New York Times uses Amazon EC2 to run Hadoop cluster to convert images from Tiff to pdf [5], Terrier team of University of Glasgow [6] uses Hadoop for information retrieval research.

Although Hadoop has gained popularity for large-scale data processing, it still have room for improvement. Many research efforts have been devoted to improving performance and reliability of Hadoop, including intermediate data fault tolerance, stage pipelining, in-memory store of intermediate data, etc. This paper focuses on reducing cross-rack communication, which is one of the critical factors that affect the performance of Hadoop.

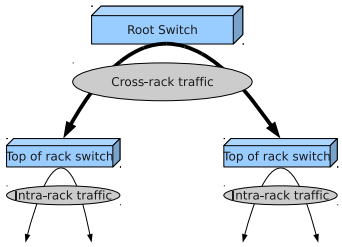


Figure 1. intra-rack and inter-rack traffics

In Hadoop framework, user needs to provide two functions, mapper and reducer, to process data. Mappers produce a set of files and send to all the reducers, a reducer will receive files from all the mappers, which is a all-to-all communication model. Cross-rack communication happens if a map and a reduce reside in different racks, which is very often in today's data center environment. Typically, Hadoop runs in a datacenter environment in which machines are organized in racks. Each rack has a top-of-rack switch and each top-of-rack switch are connected to a root switch. Every cross-rack communication needs to travel through the root switch and hence the root switch becomes a bottleneck. Figure 1 illustrates the intra-rack and inter-rack traffics in a datacenter.

As the cross-rack communication saturates the root switch, the packets sent by a mapper will be lost and have

to be re-sent, which increases data transfer time as well as job completion time. The delay caused by saturation will become more significant in large-scale data processing because such application generates large-size intermediate data that need be transferred between racks. Intermediate data are usually larger than the input data in size. For instance, in the wordcount application, the mapper will generate extra tuple attributes in the intermediate data.

The location of a Mapper is decided by the location of its input chunk, which means the mapper is dispatched to the server that stores the input chunk (file is split into chunks). The reason is to achieve data locality such that mappers do not have to fetch the input data from other servers through the network. In contrast, the location of a reducer is decided randomly in current Hadoop implementation. However, it has been reported that proper placement of reducers also plays a crucial role in reducing cross-rack communication.

The distribution of reducers among racks should be dependent on the distribution of mappers. In the extreme case, if all mappers are in the same rack, the best strategy to eliminate cross-rack communication is to place all reducers in the same rack that the mappers reside in. However, in practice, such case will not happen because (1) HDFS distributes data chunks in datacenter more or less evenly to achieve load balance. Hence the mappers are distributed across racks, and (2) in large scale data processing, all mappers cannot be placed in the same rack because of the computing capability of a rack.

The main contributions of this paper are described as follows. First, we formulate the placement of reducers to reduce cross rack communication of mappers and reducers as an optimization problem. We develop a linear-time greedy algorithm to find the optimal solution for the problem. We also propose an analytical solution to the optimization problem. Second, we implement the optimal algorithm in Hadoop, and our experiment results show that our algorithm reduces cross-rack communication and achieves 9% to 32% performance improvement on average for a set of benchmark/application programs.

The rest of the paper is organized as follows. Section II surveys related works on optimization for MapReduce. Section III introduce the background knowledge of MapReduce framework. In Section IV, we formulate the reducer placement as an optimization problem and present our algorithm for finding the optimal solution in section V. In Section VI, we report our experiment results on a server cluster spanned over four racks. Section VII gives some concluding remarks.

II. RELATED WORK

Many approaches to optimization in Hadoop framework have been proposed. Sandholm et al [7] presented a system that improves the job scheduling in Hadoop in three way. First, the system uses regulated and user-defined priorities to offer service to jobs. Second, the system will dynamically adjust resource allocation to jobs based on the different job stages. Finally, the system automatically detects the bottleneck and eliminates it within a job. Their experimental results show a 11-31% improvement in completion time for MapReduce jobs.

Condie et al [8] propose a modified version of Hadoop to support stream processing, online aggregation and continuous queries. Their system, Hadoop Online (HOP), uses a producer-consumer framework in tasks. Producer pushes the results to consumer and this model leads task pipelining naturally. Their system allows user to see the "early returns" from a job and can reduce the completion time of a MapReduce job.

Hadoop assumes the cluster nodes are homogeneous and tasks make progress linearly. With this assumption, Hadoop can discover straggler tasks and speculatively re-execute them. However, the assumption does not always hold. Zaharia et al [9] proposed a speculative task scheduling algorithm called Longest Approximate Time to End (LATE) that improve the Hadoop response times by a factor of 2 in a heterogeneous environment. LATE speculatively executes the task first that is predicted to finish last. The completion time of a task is predicted by tracking task progress instead of percentage of work completed.

Seo et al [10]uses two optimization schemes, prefetching and pre-shuffling, to improve the performance of Hadoop and reduce the execution time by 73%. They use the idle network resource to fetch the next key-value pair when working on current key-value pair. On the other hand, the current task can prefetch the data for next task in the queue in its computation.

Ganesh Ananthanarayanan et al [11] develop a system named *M*antri to rein the stragglers in MapReduce clusters. They proposed three main approaches to eliminate the effect of stragglers and one of them is *N*etwork-Aware Placement, which is related to our work. This approach balances the network load by placing reduce tasks among racks. Its algorithm computes the optimal value over all reduce tasks placement permutations and minimize the maximum data transfer time.

The major difference between *M*antri and our work is that *M*antri uses exhausted search to find the optimal reduce tasks placement, which requires very high time complexity of $O(N^R)$, where N is the number of racks and R is the number of reducers. In contrast, our greedy algorithm finds the optimal solution with very low time complexity of $O(N \times R)$.

III. BACKGROUND

In this section, we give a brief overview of the MapReduce framework. Most of our discussion focuses on Hadoop, the open source implementation of MapReduce.

A. Hadoop Distributed File System

A Hadoop system runs on top of a distributed file system called Hadoop Distributed File System (HDFS). The input data of a Hadoop job is stored as files in HDFS. A file in HDFS is split into chunks and stored dispersedly in the system. Since the Hadoop system is designed to process large volume of data, the size of chunk is typically 64 MB. HDFS replicates the chunks to multi-nodes to achieve data fault tolerance. The consistency of replicas is not considered because the files in HDFS are assumed to be append-only and the chunks are written once and read many times. HDFS employs master-slave architecture. The master node stores metadata of chunks in the memory and the slaves store the chunks. As a client request a chunk, the master processes the request, find out which slave stores the chunk and dispatches the request to that slave. All the metadata of chunks are stored in the memory of master node to ensure fast response of chunk request.

B. MapReduce Framework

The MapReduce framework is a runtime system that utilizes a set of machines to process large volume of data. This framework employs the data-flow programming model that data flow through multiple phases in parallel. There are three phases in the MapReduce framework : map phase, shuffle phase and reduce phase. Map task executes user-provided map function and read a chunk as input data which contains multiple key-value pairs records. The map task combines the records with the same key as a tuple and write it to an intermediate file on the local disk. The Map task produces multiple intermediate files and each of them will be forwarded to a recue task later. After the map phase, the shuffle phase supported by the framework, sort the tuples in a intermediate file by the key and transfer the file to a reduce task. When a reduce task receives all intermediate files from all map tasks, it applies user-provided reduce function and process the tuples in all intermediate files and finally write an output file on HDFS. As described above, the shuffle phase heavily utilizes network due to its all-to-all communication nature and is very likely the bottleneck of performance, meanwhile, to reduce network traffic, the map task is executed on the node which contains the input chunk, that is, send the computation to the data. Moreover, the intermediate files are stored on the local disk to eliminate network traffic.

IV. Model

In this section, we define the cross-rack communication model and formulate the reducer placement problem.

Suppose we run a MapReduce job in a data center which consists of N racks and there are totally M mappers and R reducers for the job. The size of intermediate data produced by a mapper is k bytes. We consider the up and down traffics of a rack.

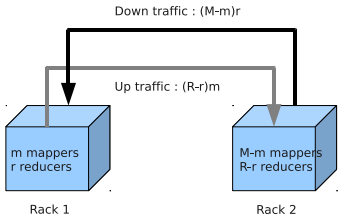


Figure 2. Up and down traffic between two racks

For a rack S_i, $i \in \{1, \ldots, N\}$, the amount of data sent out (up traffic) and in (down traffic) the rack are decided by the number of mappers and reducers in the rack. Suppose there are m_i mappers and r_i reducers on rack S_i, the up traffic is generated by the mappers in S_i sending the intermediate data to the reducers residing outside rack S_i. Meanwhile, the reducers in S_i receive the intermediate data from mappers in remote racks which causes the down traffics. Therefore, the amount of data sent by up and down traffic can be computed by

$$\textbf{up} : m_i \cdot (R - r_i) \cdot k \qquad \textbf{down} : r_i \cdot (M - m_i) \cdot k$$

Figure 2 illustrates the up and down traffics between two racks.

To simplify our model, we set k to 1. Hence, the total data traffic of rack S_i is

$$m_i \cdot (R - r_i) + r_i \cdot (M - m_i)$$

To achieve input data locality, MapReduce framework assigns mappers to machines which store the corresponding input chunks, therefore, the number of mappers of a rack S_i (m_i) is decided approximately by the number of input chunks stored in the rack S_i. However, it is only true when the degree of replication of chunk is 1, since with degree of replication larger than 1, we do not know which replica the framework chooses as input. Here we assume the degree of replication is 1.

Hence, with the knowledge of input chunks distribution among racks, which can be obtained via metadata service of Hadoop distributed file system (HDFS), we can know each m_i of rack $S_i \quad \forall i \in \{1, \ldots, N\}$. In another way, this information of number of mappers of each rack can also be acquired from master node at run-time.

The total amount of data going in and out from a rack S_i is then a linear function of the number of reducers in the rack.

$$f_i(r_i) = m_i \cdot (R - r_i) + r_i \cdot (M - m_i) = (M - 2m_i) \cdot r_i + R \cdot m_i$$

We want to minimize the cross-rack communications of a data center, which is equivalent to placing the reducers

to minimize the maximum total traffic for all racks in the data center when given the number of mappers in each rack. This problem can be formulate as a min-max optimization problem:

$$min\{ \quad \underset{r_i, i\in\{1,\cdots,N\}, \sum_i r_i = R}{\text{argmax}} \quad \{f_1(r_1), \cdots, f_N(r_N)\} \quad \}$$

where $f_i(r_i)$ is the total traffic of rack S_i, $M = \sum_i m_i$ and $R = \sum_i r_i$ are predefined constants.

In conclusion, we propose the *reducers placement problem*. Given N racks, total number of reducers R and a mapper configuration $(m_1, m_2, \ldots, m_N), \sum_i m_i = M$, we want to find a reducer configuration $(r_1, r_2, \ldots, r_N)$ such that the maximum of $\{f_1(r_1), f_2(r_2), \ldots, f_N(r_N)\}$ is minimum, with the restriction $\sum_i r_i = R$ and $f_i(r_i) = (M - 2m_i) \cdot r_i + R \cdot m_i \forall i \in \{1, \ldots, N\}$.

We describe an optimal algorithm and an analytical solution to this problem in next section.

V. Algorithms

We devise two optimal algorithms to solve the *reducer placement problem*, (*RPP*) in this section. We also develop a mathematical method to obtain the analytical solution of RPP.

A. Greedy Algorithm

Our greedy algorithm assigns one reduce task to a rack at a time. When assigning a reduce task to a rack, it chooses the rack which incurs minimum total traffic (up and down) if the reduce task is assigned to that rack. That is

$$min_{i\in\{1,\ldots,N\}} f_i(r_i + 1), \text{where} f_i(r_i) = (M - 2m_i) \cdot r_i + m_i R$$

The r_i is the number of reducers currently assigned to rack i. Here is an example. Suppose we have three racks, $N = 3$, and the number of mappers of the first rack is 2, the second rack is 3 and the third rack is 4, that is, $m_1 = 2$, $m_2 = 3$ and $m_3 = 4$ and $M = 9$. We now have $R = 4$ reducers to schedule. By our formulation, the traffic functions of these three racks are

$$\begin{cases} 5r_1 + 8 & (rack1) \\ 3r_2 + 12 & (rack2) \\ 1r_3 + 16 & (rack3) \end{cases} \tag{1}$$

We use a state tuple (r_1, r_2, r_3) to represent the current reducer assignment, initially, the tuple is $(0, 0, 0)$. To assign the first reducer, because $5*1+8 = 13(rack1) < 3*1+12 = 15(rack2) < 1*1+16 = 16(rack3)$, we assign the first reducer to rack 1. The state tuple now becomes $(1, 0, 0)$. In the second step, we assign next reducer to rack 2 due to $3*1+12 = 15(rack2) < 1*1+16 = 16(rack3) < 5*2+8 = 18(rack1)$. And the state tuple becomes $(1, 1, 0)$. The steps continue and finally we get the state tuple $(2, 2, 0)$ which means we place 2 reducers on rack1, 2 reducers on rack2 and 0 reducer on rack3.

The following is the pseudo code of the greedy algorithm.

Algorithm 1 Greedy Algorithm for RPP

Require: The number of mappers on each rack : $\{m_1, m_2, \ldots, m_N\}$
Ensure: A reducer state tuple : $\{r_1, r_2, \ldots, r_N\}$
  $N \leftarrow$ number of racks
  $M \leftarrow$ number of total mappers
  $R \leftarrow$ number of total reducers
  $state\_tuple[N] \leftarrow \{0, 0, \ldots, 0\}$
  for $i = 1$ to R **do**
    $minimal \leftarrow \infty$
    for $j = 1$ to N **do**
      $traffic = (M - 2m_j) \cdot (state\_tuple[j] + 1) + m_j R$
      if $traffic < minimal$ **then**
        $candidate = j$
      end if
    end for
    $state\_tuple[candidate] + +$
  end for
  return $state\_tuple$

We next prove that the greedy algorithm for *RPP* is optimal.

Theorem 1: The greedy algorithm for *RPP* is optimal.

Proof: The proof is by induction. For $R = 1$, the reducer is placed in the rack with minimum traffics by the greedy algorithm, thus we minimize the maximum traffic. Suppose $R = k$ we have minimized the maximum traffic, we prove that for $R = k+1$ we also minimize the maximum traffic. For the $(k+1)_{th}$ reducer, the algorithm places it in the rack with minimum traffic by the linear equations and hence for $R = k+1$ we also minimize the maximum traffic. ■

B. Binary Search

The second algorithm, called `Binary Search`, uses binary search to find the minimum bound of the traffic function for each rack, and then use that minimum bound to find the number of reducers on each rack. Formally, we represent the traffic of racks by a set of linear functions bounded by B

$$\begin{cases} (M - 2m_1)r_1 + m_1 R \leq B \\ (M - 2m_2)r_2 + m_2 R \leq B \\ \ldots \\ (M - 2m_N)r_N + m_N R \leq B \end{cases} \tag{2}$$

where $r_1 + r_2 + \ldots + r_N = R$. We solve the linear programming by binary searching the minimum bound B. Once the bound B is found, we use B and the constrain $r_1 + r_2 + \ldots + r_N = R$ to solve r_i for $i \in 1, \ldots, N$. In the

above example, we use binary search to find that B is 18, and then solve r_i for $i \in \{1, 2, 3\}$.

$$\begin{cases} r_1 = (18-8)/5 = 2 \\ r_2 = (18-12)/3 = 2 \\ r_3 = R - r_1 - r_2 = 4 - 2 - 2 = 0 \end{cases} \tag{3}$$

Hence the state tuple is $\{2, 2, 0\}$. The following is the pseudo code of the binary search algorithm. To simplify description, we define the linear system of equations(2) as $\Gamma(r_1, r_2, \ldots, r_N, B)$.

Algorithm 2 Binary Search Algorithm for RPP

Require: The number of mappers on each rack : $\{m_1, m_2, \ldots, m_N\}$
Ensure: A reducer state tuple : $\{r_1, r_2, \ldots, r_N\}$
  $N \leftarrow$ number of racks
  $upper \leftarrow M \times R$
  $lower \leftarrow 0$
  $state\_tuple[N] \leftarrow null$
  $\Gamma(r_1, r_2, \ldots, r_N, B)$: the target linear system
  while $upper \neq lower$ **do**
    $B = \frac{upper+lower}{2}$
    $state\_tuple[N] =$
    $solve\_linear\_programming(\Gamma(r_1, r_2, \ldots, r_N, B))$
    if $state\_tuple[N] = null$ **then**
      $lower = B$
    else
      $upper = B$
    end if
  end while
  $B = upper$
  $state\_tuple =$
  $solve\_linear\_programming(\Gamma(r_1, r_2, \ldots, r_N, B))$
  return $state\_tuple$

C. Analytical Solution

In this subsection, we introduce the analytical solution of RPP. For a set of linear equations,

$$\begin{cases} f_1(r_1) = (M - 2m_1)r_1 + m_1 R \\ f_2(r_2) = (M - 2m_2)r_2 + m_2 R \\ \cdots \\ f_N(r_N) = (M - 2m_N)r_N + m_N R \end{cases} \tag{4}$$

we want to minimize the maximum of $f(r_i)$ where $i \in \{1, \ldots, N\}$ with the constraint $\sum_i r_i = R$. We first rewrite the linear system as a set of N-dimension hyperplanes according to the constrain of $\sum_i r_i = R$.

$$\begin{cases} z &= f_1(r_1, r_2, \ldots, r_{N-1}) = (M - 2m_1)r_1 + 0r_2 + \ldots \\ &+ \quad 0 \cdot r_{N-1} + m_1 R \\ z &= f_2(r_1, r_2, \ldots, r_{N-1}) = 0 \cdot r_1 + (M - 2m_2)r_2 + \ldots \\ &+ \quad 0 \cdot r_{N-1} + m_2 R \\ \cdots & \\ z &= f_{N-1}(r_1, r_2, \ldots, r_{N-1}) = (M - 2m_N)r_N + m_N R \\ &= (M - 2m_N)(R - r_1 - r_2 - \ldots - r_{N-1}) + m_N R \end{cases} \tag{5}$$

And we have the matrix representation.

$$\begin{bmatrix} M-2m_1 & 0 & \cdots & M-2m_1 & -1 \\ 0 & M-2m_2 & \cdots & M-2m_2 & -1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & M-2m_{N-1} & -1 \\ -(M-2m_N) & -(M-2m_N) & \cdots & -(M-2m_N) & -1 \end{bmatrix} \times \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_{N-1} \\ z \end{bmatrix} = \begin{bmatrix} -m_1 R \\ -m_2 R \\ \vdots \\ -m_{N-1} R \\ -(M-m_N)R \end{bmatrix} \tag{6}$$

To minimize the maximum of the linear system, it is equivalent to find the minimum intersection point of all hyperplanes. We next show the optimality and uniqueness of the intersection point of all planes.

Theorem 2: The equation (5) of hyperplanes has unique and minimum intersection point.

Proof: We begin with its matrix representation of equation (6). We transform the coefficient matrix by Gaussian elimination process and show its determinant is not zero by the assumption of $m_i < \frac{M}{2} \quad \forall i \in \{1, \ldots, N\}$.

$$\begin{vmatrix} M-2m_1 & 0 & \cdots & 0 & -1 \\ 0 & M-2m_2 & \cdots & 0 & -1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & M-2m_{N-1} & -1 \\ -(M-2m_N) & -(M-2m_N) & \cdots & -(M-2m_N) & -1 \end{vmatrix} \sim \begin{vmatrix} M-2m_1 & 0 & \cdots & 0 & 0 \\ 0 & M-2m_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & M-2m_{N-1} & 0 \\ 0 & 0 & \cdots & 0 & \triangle \end{vmatrix} \neq 0 \tag{7}$$

$$where \quad \triangle = \frac{M-2m_N}{M-2m_1} \cdot (-m_1 R) + \frac{M-2m_N}{M-2m_2} \cdot (-m_2 R) + \cdots + \frac{M-2m_N}{M-2m_{N-1}} \cdot (-m_{N-1} R) - (M - m_N)R \tag{8}$$

Hence, the matrix equation has unique solution and the hyperplanes has only one intersection point.

The optimality is shown by partial differentiating at the intersection point.

$$\frac{\partial f_i(r_1, r_2, \cdots, r_{N-1})}{\partial r_i} = \begin{cases} & (M - 2m_i) \\ - & (M - 2m_N) \end{cases} \quad i \in \{1, \cdots, N-1\} \tag{9}$$

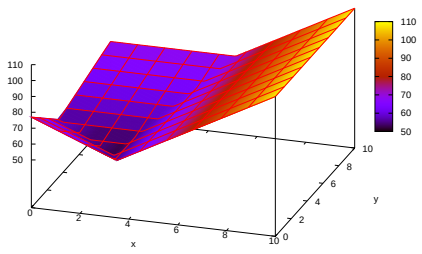


Figure 3. An illustration of intersection of hyperplanes

The intersection point is an extreme because there are two derivatives with different sign (+,-) of each direction.

■

For illustration, we take $N = 3, M = (4, 7, 5), R = 7$ for an example. The set of hyperplanes is $\Gamma(r_1, r_2, z)$:

$$\begin{cases} 8r_1 + 0r_2 - z = -28 \\ 0r_1 + 2r_2 - z = -49 \\ -6r_1 - 6r_2 - z = -77 \end{cases} \quad (10)$$

Figure 3 shows the intersection point of three hyperplanes is the minimum solution. The intersection point can be obtained by solving equation(10) and we get $r_1 = 2.94, r_2 = 1.28$ and $z = 51.58$, which means the traffic load is 51.58 and the number of reducers of each rack $\{r_1, r_2, r_3\}$ is $\{2.94, 1.28, 2.78\}$. This analytical method finds the global minimum of the linear system, however, the solution solved by this method may not be feasible. It provides a lower bound of cross-rack traffics in the data center. We obtain the feasible and optimal solution by using the greedy algorithm or binary search.

VI. Experiment

The experiment are performed on a four-track cluster. The racks are connected by a gigabits switch and the network topology form a one-level tree. The CPU types of Rack A, B, C and D are Intel Xeon E5504, E5520, E5620 and E5620 respectively. The memory of all nodes is 8 GB. The number of nodes in rack A, B, C and D are 7, 5, 4, 4 respectively. The nodes in a rack are homogeneous and the master node resides in Rack A.

To evaluate our reducer placement algorithm, we need a benchmark suite to be the testing programs. HiBench [12] is a benchmark suite for Hadoop developed by Intel, China, however, it is not open-source. Since there is no standard benchmark for Hadoop evaluation, we choose five popular MapReduce applications as our benchmark programs. These applications are

- WordCount
- Grep
- PageRank
- K-mean Cluster
- Frequently Pattern Match

WordCount and Grep are included in the Hadoop distribution and are used widely in benchmarking Hadoop. PageRank is an important and representative applications in web searching, and we obtain the application from the Cloud[9] [13] library. K-mean cluster and frequent pattern match applications are included in Mahout [14], an open-source machine learning library to run on Hadoop. We use these two applications because they are popular and representative in machine learning areas.

Our Hadoop cluster is configured as eight mapper slots and four reducer slots per node. The default value of reducers of a job is set as 50 by the suggestion of Hadoop (0.7x to 0.9x of total reducer capacity). We use the speedup of elapsed time of total reduce tasks as our performance metric. The metric is defined by

$$\frac{original - optimized}{original} \times 100\%.$$

A. Result

Table I summarizes the performance improvement by our algorithm on each benchmark. The improvement ratio ranges from 2% to 32% by different benchmarks. The best performance improvement is 32%, in the case of PageRank 10G. The frequent pattern matching application does not have significant improvement. Other applications improve about 10% in average. In the case of PageRank 10G, we further investigate the source of improvement. Table II shows the average execution time of shuffle, merge and reduce phase of a reduce task compared with original Hadoop. Moreover, Table III shows the comparison of distribution of reducers among 4 racks. Our algorithm achieves a more balanced reducer placement. Note that in PageRank 10G, the application generates 101 reducers to be scheduled.

Benchmark	Speedup (%)
Grep	9.35
WordCount	12.37
PageRank	32.84
K-mean	14.7
FPM	1.76

Table I
SPEEDUP OF BENCHMARKS

To compare with the result of network-aware reducer placement in Mantri [11], they use exhausted search to find

Phase	Hadoop	Ours
Shuffle	105.5s	80.98s
Merge	6.2s	5.7s
Reduce	8.1s	7.5s

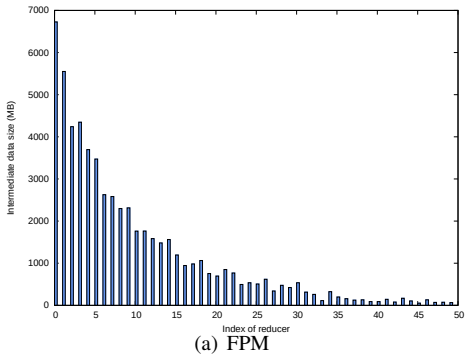
Table II
AVERAGE EXECUTION TIME OF PHASES

Rack	Hadoop	Ours
A	37	20
B	25	22
C	18	28
D	21	31

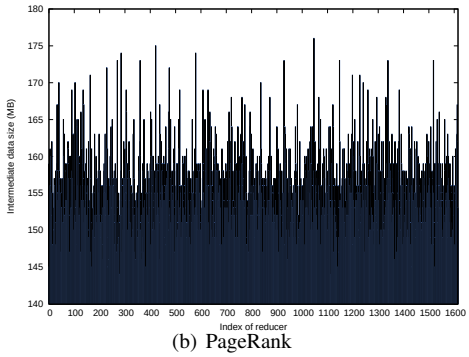
Table III
NUMBER OF REDUCERS ON EACH RACK

the optimal reducer placement and their experiment demonstrates GroupBy can improve 32%. Our work achieves the same improvement in PageRank by solving RPP optimally.

Our traffic model assumes the amount intermediate data of each mapper sent to a reducer is a constant. This is not the case in FPM application. As Figure 4(a) shows, the amount of input data of each reducer is very different, ranges from 7 GB to 50 MB. The elapsed time of reduce tasks in FPM is dominated by the reducer with the largest input. Our algorithm only decides the number of reducers in each rack but does not consider the variation in data sizes. However, in other four applications, the intermediate data size is approximately the same. For example, in Figure 4(b) in the case of PageRank 200G, the size is in a short range of 155 MB to 165 MB. Grep and WordCount use random text writer as input and have similar distribution of intermediate data size.



(a) FPM



(b) PageRank

Figure 4. Intermediate Data Size

We next examine the effect of varied input data size on our algorithm. The results are presented in Figure 5. WordCount and K-means have similar trend, as the input data becomes larger, the speedup improves. This is because when the input data becomes larger, so does the intermediate data, and our algorithm keep a balanced load of network. Note that the number of reducers in WordCount and K-means are fixed at 50 when varying input data size.

The behavior of PageRank is very different. The reason is that PageRank sets the number of reducers as the same with the number of mappers, so the number of reducer becomes larger as input size increases. Our algorithm calculates the number of reducers of each rack and limits the number of reducers running on a rack. As the number of reducers becomes larger, more reducers wait for a specific rack to run on, and some reduce slots in other racks are idle. This lowers the utilization of reduce slots and inflates the job execution time. On the other hand, Hadoop uses pull model to assign reduce tasks to rack, so reduce slots never become idle. As a result, the speedup decreases as the amount of input increases.

The speedup of Grep fluctuates as varying input data size. The reducer of Grep reads the selected lines by mappers and writes them all into HDFS. This induces heavy I/O operations and dominates the execution time of reducers.

Based on the above investigation, Our algorithm is suitable for the applications with constant intermediate data size and heavy network traffics (shuffle bound), like PageRank. Our algorithm achieves network load balancing by placing reducers among racks and improves the performance of shuffle phase.

VII. CONCLUSION

MapReduce employs all-to-all communication model between mappers and reducers. This results in saturation of network bandwidth of top-of-rack switch in shuffle phase and straggles some reducers and increases job execution time. In this paper, we model the traffics in multiple-racks environment and propose a *Reducer Placement Problem*(*RPP*) to balance cross-rack traffics by placing reducers in racks. We propose two optimal algorithms to solve *RPP* and an analytical method to find the minimum (may not feasible) solution of *RPP*.

We select five MapReduce applications to be the evaluation benchmark. The experiment shows the performance improvement by equipping our network load balancing algorithm can achieve 32% performance improvement. We also examine the relation between input data size and performance, and we find the applications with approximately the same intermediate data size and heavy network traffics is suitable for our algorithm, because our algorithm keep a balanced network traffics and prevent reducers from becoming stragglers.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th*

conference on Symposium on Opearting Systems Design and Implementation, ser. OSDI 04, vol. 6. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] "Hadoop," http://hadoop.apache.org/.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 59–72.

[4] "A9," http://a9.com/.

[5] "Nytime," http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/.

[6] "Terrier," http://terrier.org/.

[7] T. Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '09. New York, NY, USA: ACM, 2009, pp. 299–310.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 21–21.

[9] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.

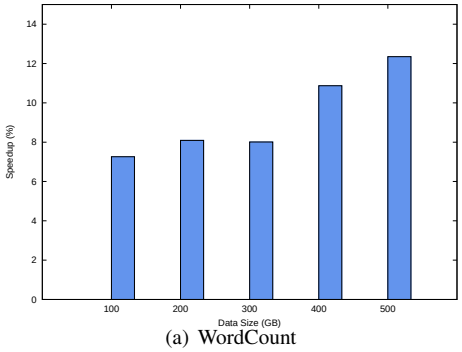
[10] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng, "Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment," in *IEEE International Conference on Cluster Computing*, ser. CLUSTER '09, 2009, pp. 1–8.

[11] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.

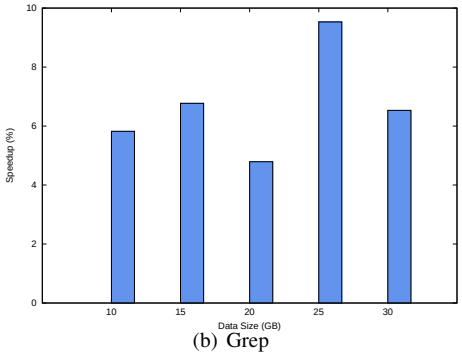
[12] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, ser. ICDEW'10, 2010, pp. 41 –51.

[13] "Cloud9 : Mapreduce library for hadoop," http://www.umiacs.umd.edu/ jimmylin/cloud9/docs/index.html.

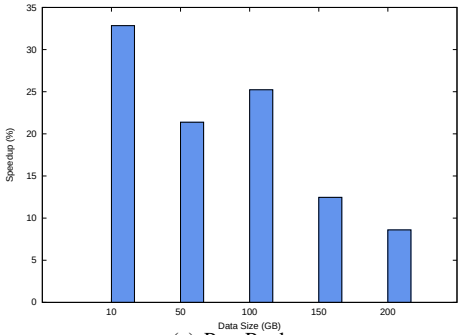
[14] "Apache mahout : A scalable machine learning library," http://mahout.apache.org/.



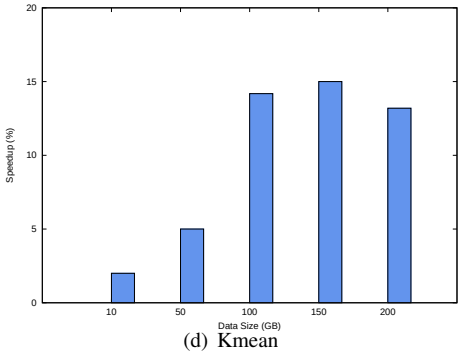
(a) WordCount



(b) Grep



(c) PageRank



(d) Kmean

Figure 5. Speedup with varying data size