Enhancing MapReduce via Asynchronous Data Processing

Marwa Elteir, Heshan Lin and Wu-chun Feng
Department of Computer Science
Virginia Tech
{maelteir, hlin2, feng}@cs.vt.edu

***Abstract*—The MapReduce programming model simplifies large-scale data processing on commodity clusters by having users specify a *map* function that processes input key/value pairs to generate intermediate key/value pairs, and a *reduce* function that merges and converts intermediate key/value pairs into final results. Typical MapReduce implementations such as Hadoop enforce barrier synchronization between the map and reduce phases, i.e., the reduce phase does not start until all map tasks are finished. In turn, this synchronization requirement can cause inefficient utilization of computing resources and can adversely impact performance.**

Thus, we present and evaluate two different approaches to cope with the synchronization drawback of existing MapReduce implementations. The first approach, hierarchical reduction, starts a reduce task as soon as a predefined number of map tasks completes; it then aggregates the results of different reduce tasks following a tree structure. The second approach, incremental reduction, starts a predefined number of reduce tasks from the beginning and has each reduce task incrementally reduce records collected from map tasks. Together with our performance modeling, we evaluate different reducing approaches with two real applications on a 32-node cluster. The experimental results have shown that incremental reduction outperforms hierarchical reduction in general. Also, incremental reduction can speed-up the original Hadoop implementation by up to 35.33% for the wordcount application and 57.98% for the grep application. In addition, incremental reduction outperforms the original Hadoop in an emulated cloud environment with heterogeneous compute nodes.

***Index Terms*—Distributed Computing, Cloud Computing, MapReduce, Hadoop, Asynchronous processing.**

I. Introduction

Today, many commercial and scientific applications require the processing of large amounts of data, and thus, demand compute resources far beyond what can be provided by a single commodity processor. To address this, various high-end computing platforms — including supercomputers, graphics processors, clusters of workstations, and cloud computing environments — have been architected to facilitate parallel processing at different scales. However, efficiently exploiting the hardware parallelism provided by these platforms requires parallel programming skills that are difficult to master by common application developers. Even for well-trained experts, the engineering and debugging of parallel applications can be time consuming.

MapReduce [4] is an effort that seeks to democratize parallel programing. Originally proposed by Google, MapReduce has been used to process massive amounts of data in web search engines on a daily basis. With the maturity of Hadoop, an open-source MapReduce implementation, the MapReduce programming model has been widely adopted and found effective in many scientific and commercial application areas such as machine learning [2], bioinformatics [8], astrophysics [7] and cyber-security [5]. Because of its ease of use and built-in fault tolerance, MapReduce is also becoming one of the most effective programming models in cloud computing environments, e.g., Amazon EC2 [1].

A MapReduce job consists of two user-provided functions: map and reduce. As shown in Figure 1, the run-time system creates concurrent instances of map tasks that split and process the input data. The intermediate results generated by map tasks are then copied to the reduce tasks, which in turn reduce the intermediate results and produce the final output. In a MapReduce system, all the above data is stored as key/value pairs for efficient data indexing and partitioning.

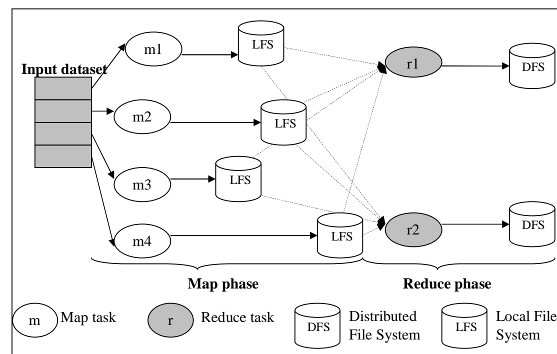

Fig. 1. MapReduce framework

Existing MapReduce implementations (e.g., Hadoop) enforce a barrier synchronization between the map phase and the reduce phase, i.e., the reduce phase only starts when all map tasks are completed. There are two cases where this barrier synchronization can result in serious resource underutilization. First, on heterogeneous environments, the faster compute nodes will finish their assigned map tasks earlier, but these nodes cannot proceed to the reduce processing until all the map tasks are finished, thus wasting resources. The resource heterogeneity can originate from different hardware

configurations or resource sharing through virtualization.[1] Second, even in homogeneous environments, a compute node may not be fully utilized by the map processing because a map task alternates between computation and I/O. Also, there is additional scheduling overhead between the execution of different tasks. In this case, overlapping map and reduce processing can considerably improve job response time, especially when the number of map tasks is relatively large compared to the cluster size. For instance, the number of map tasks of a MapReduce job is typically configured to be 100 times the number of compute nodes at Google [4].

In this paper, we focus on addressing the above synchronization problem for a specific class of MapReduce jobs, called *recursively reducible* MapReduce jobs. For this type of MapReduce jobs, a portion of the map results can be reduced independently, and the partial reduced results can be recursively aggregated to produce global reduce results. One example of such a job is word counting. More details about recursively reducible MapReduce jobs will be discussed in Section II-B.

The unique characteristic of recursively reducible MapReduce jobs is that there is no inherent synchronization requirement between the map phase and the reduce phase. Consequently, we propose and compare two asynchronous data-processing techniques to enhance resource utilization and performance of MapReduce for recursively reducible jobs. The first approach, hierarchical reduction (HR), overlaps map and reduce processing at the inter-task level. This approach starts a reduce task as soon as a certain number of map tasks complete and aggregates partial reduced results following a tree hierarchy. The second approach, incremental reduction (IR), exploits the potential of overlapping data processing and communication within each reduce task. It starts a designated number of reduce tasks from the beginning and incrementally applies reduce function to the intermediate results accumulated from map tasks.

We evaluate the proposed approaches with analytical models and experiments on a 32-node cluster. The experimental results demonstrate that *both* approaches can effectively improve the MapReduce execution time — with the incremental reduction approach consistently outperforming hierarchical reduction. In particular, incremental reduction can outperform the original Hadoop implementation by 35.33% for the wordcount application and 57.98% for the grep application.

The rest of the paper is organized as follows. Section II provides background information about Hadoop and recursively reducible MapReduce jobs. Section III presents the design of the proposed approaches, in particular, hierarchical reduction and incremental reduction. Section IV evaluates the performance of the proposed approaches using an analytical model. The experimental results are discussed in Section V. Section VI presents the related work. Finally, Section VII concludes the paper.

[1] Zahria et al. reported that there can be a 2.5-fold performance difference among various virtual machine instances on Amazon EC2 [13].

II. Background

Here we describe Hadoop, an open-source Java implementation of the MapReduce framework, as well as the notion of recursively reducible jobs.

A. Hadoop

Hadoop can be logically segregated into two subsystems, i.e., a distributed file system called HDFS and a MapReduce run-time system. The MapReduce run-time system follows a master-slave design. The master node is responsible for managing submitted jobs, i.e., assigning map and reduce tasks of every job to the available workers. By default, each worker can run two map tasks and two reduce tasks simultaneously.

At the beginning of a job execution, the input data is split and assigned to individual map tasks. When a worker finishes executing a map task, it stores the map results as intermediate key/value pairs locally. The intermediate results of each map task will be partitioned and assigned to the reduce tasks according to their keys. A reduce task begins by retrieving its corresponding intermediate results from all map outputs (called the *shuffle* phase). The reduce task then sorts the collected intermediate results and applies the reduce function to the sorted results. To improve performance, Hadoop overlaps the retrieving and sorting of finished map outputs with the execution of newly scheduled map tasks.

B. Recursively Reducible Jobs

Word counting is a simple example of recursively reducible jobs. The occurrences of a word can be counted first on different splits of an input file, and those partial counts can then be aggregated to produce the number of word occurrences in the entire file. Other recursively reducible MapReduce applications include association rule mining, outlier detection, commutative and associative statistical functions, and so on. In contrast, the square of the sum of values is an example of a reduce function that is *not* recursively reducible, because $(a+b)^2 + (c+d)^2$ does not equal $(a+b+c+d)^2$. However, there are some mathematical approaches that can transform such functions to benefit from our solution.

It is worth mentioning that there is a *combiner function* provided in typical MapReduce implementations including Hadoop. The combiner function is used to reduce key/value pairs generated by a *single* map task. The partially reduced results, instead of the raw map output, are delivered to the reduce tasks for further reduction. Our proposed asynchronous data-processing techniques are applicable to all applications that can benefit from the combiner function. The fundamental difference between our techniques and the combiner function is that our techniques optimize the reducing of key/value pairs from *multiple* map tasks.

III. Asynchronous MapReduce Data Processing

In this section, we present the design details of our two proposed asynchronous data-processing techniques: hierarchical reduction and incremental reduction.

A. Hierarchical Reduction (HR)

1) Design and Implementation: Hierarchical reduction seeks to overlap the map and reduce processing by dynamically issuing reduce tasks to aggregate partially reduced results along a tree-like hierarchy. As shown in Figure 2, as soon as a certain number (i.e., defined by the aggregation level σ_H) of map tasks are successfully completed, a new reduce task is created and assigned to one of the available workers. This reduce task is responsible for reducing the output of the σ_H map tasks that are just finished. When all map tasks are successfully completed and assigned to reduce tasks, another stage of the reduce phase is started. In this stage, as soon as a certain σ_H reduce tasks are successfully completed, a new reduce task is created to reduce the output of the σ_H reduce tasks. This process repeats until there is only one remaining reduce task, i.e., when all intermediate results are reduced.

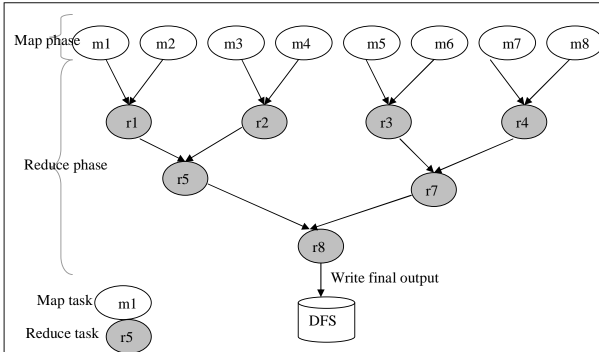

Fig. 2. Hierarchical reduction with an aggregation level of two.

Although conceptually the reduce tasks are organized as a balanced tree, in our implementation a reduce task at a given level does not have to wait for all of the tasks at the previous level to finish. In other words, as soon as a sufficient number of tasks (i.e., σ_H) from the previous level becomes available, a reduce task from the subsequent level can begin. Such a design can reduce the associated scheduling overhead of HR.

2) Discussion: One advantage of HR is that it can parallelize the reduction of a single reducing key across multiple workers, whereas in the original MapReduce framework, the reduction of a key is always handled by one worker. Therefore, this approach is suitable for applications with significant reduce computation per key. However, HR incurs extra communication overhead in transferring the intermediate key/value pairs to reduce tasks at different levels of the tree hierarchy, which can adversely impact the performance as the depth of the tree hierarchy increases. Other overheads include the scheduling cost of reduce tasks generated on the fly.

The fault-tolerant design of the original Hadoop needs to be modified to accommodate HR. In particular, the *JobTracker* should keep track of all created reduce tasks, in addition to the tasks assigned to be reduced by these reduce tasks. Whenever a reduce task fails, another copy of this task should be created, and the appropriate tasks should be assigned again for reduction.

B. Incremental Reduction (IR)

1) Design and Implementation: Incremental reduction aims to start the reduce phase as early as possible within a reduce task. Specifically, the number of reduce tasks are defined at the beginning of the job similar to the original MapReduce framework. Within a reduce task, as soon as a certain amount of map outputs are received, the reduction of these outputs starts and the results are stored locally. The same process repeats until all map outputs are retrieved.

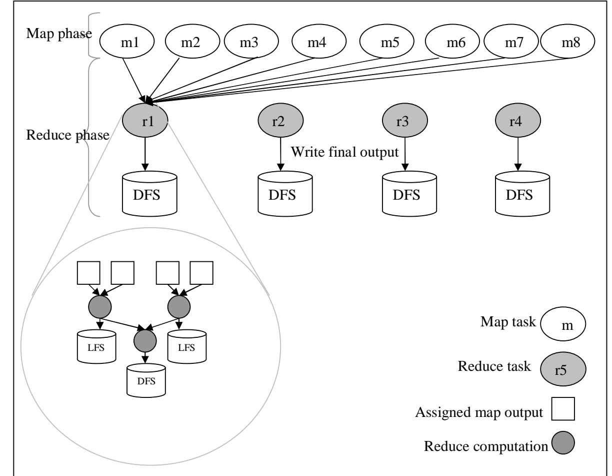

Fig. 3. Incremental reduction with an reduce granularity of two.

In Hadoop, a reduce task consists of three stages. The first stage, named shuffling, copies the task's own portion of intermediate results from the output of all map tasks. The second stage, named sorting, sorts and merges the retrieved intermediate results, according to their keys. Finally, the third stage applies the reduce function to the values associated with each key. To enhance the performance of the reduce phase, the shuffling stage is overlapped with the sorting stage. More specifically, when the number of in-memory map outputs reaches a certain threshold, *mapred.inmem.merge.threshold*, these outputs are merged and the results are stored on-disk. When the number of on-disk files reaches another threshold, *io.sort.factor*, another on-disk merge is performed. After all map outputs are retrieved, all on-disk and in-memory files are merged, and then the reduction stage begins.

In our IR implementation, we make use of *io.sort.factor* and *mapred.inmem.merge.threshold*. When the number of in-memory outputs reaches the *mapred.inmem.merge.threshold* threshold, they are merged and the merging results are stored on the disk. When the number of on-disk outputs reaches the *io.sort.factor* threshold, the incremental reduction of these outputs begins and the reducing results are stored instead of the merging results. When all map outputs are retrieved, the in-memory map outputs are reduced along with the stored reducing results. The final output data is written to the distributed file system. The entire process is depicted in Figure 3.

2) Discussion: IR incurs less overheads than HR for two reasons. First, the intermediate key/value pairs are transmitted

once from the map to reduce tasks instead of several times along the hierarchy. Second, all of the reduce tasks are created at the start of the job, and hence, the scheduling overhead is reduced.

In addition to the communication cost, the number of writes to local and distributed file system are the same (assuming the same number of reduce tasks) for both the original MapReduce and IR. Therefore, IR can outperforms the original MapReduce when there is sufficient overlap between the map and reduce processing.

The main challenge of IR is to select the right threshold that triggers an incremental reduce operation. Too low a threshold will result in unnecessarily frequent I/O operations, while too high a threshold will not be able to deliver noticeable performance improvements. Interestingly, a similar decision, i.e., the merging threshold, has to be made in the original Hadoop implementation as well. Currently we provide a run-time option for users to control the incremental reduction threshold. In the future, we plan to investigate self-tuning of this threshold for long-running MapReduce jobs.

It is worth noting that since the map and reduce tasks in this approach are created in the same manner as in Hadoop, the fault-tolerant scheme of Hadoop works in IR as well.

IV. Analytical Models

Here we derive analytical models to compare the performance of the original MapReduce (MR) implementation of Hadoop and the augmented implementations with hierarchical reduction (HR) and incremental reduction (IR) enhancements. Table I presents all the parameters used in the models.

Without loss of generality, our modeling assumes the number of reduce tasks r is smaller than the number of execution slots $2n$. In fact, the Hadoop documentation recommends that 95% of the execution slots is a good number for the number of reduce tasks for typical applications. However, our analysis can be easily generalized to model the cases where there are more reduce tasks than the number of execution slots. Furthermore, the model assumes that the number of map tasks m is greater than the number of available execution slots in the cluster $2n$ (recall that there are two execution slots per node by default). When the number of map tasks is less than the number of execution slots, all map tasks are executed in parallel and completed simultaneously, so there is no way to overlap map and reduce phases.

For simplicity, we consider two different cases. The first case assumes that there is a high degree of overlap between the map and reduce procedures, where the processing (including retrieving, merging, and sorting) of almost all intermediate results in a reduce task can be overlapped with map computations. The second case, representing a low degree of overlap between the map and reduce procedures, assumes that the processing of only a portion of intermediate results in a reduce task can be overlapped with map computations.

A. MR

The execution of the original Hadoop can be illustrated by the left part of Figure 4. When the overlapping degree is high,

Parameters	Meaning
m	Number of map tasks
n	Number of nodes
k	Total number of intermediate key/ value pairs
r	Number of reduce tasks of the MR framework
t_m	Average map task execution time
t_{rk}	Average execution time of reducing values of a single key
σ_H	Aggregation level used in HR
C	Communication cost per key/value pair
C_{MR}	Communication cost from m map tasks to r reduce tasks in MR
C_{HR}	Communication cost from the assigned σ_H map tasks to a reduce task in HR

TABLE I
Parameters used in the performance model

the merging phase of MR can be eliminated. So, the total time becomes *Map time* + *Reduce time*, assuming the final stage merging is neglected.

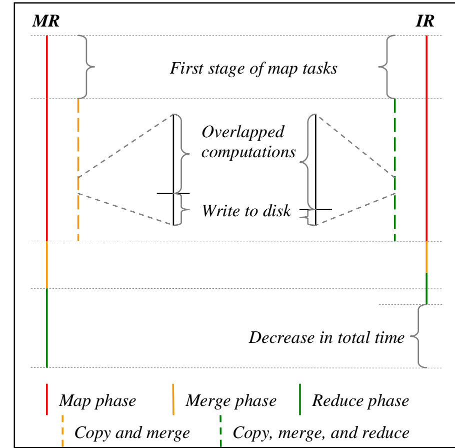

Fig. 4. Execution of MR and IR

For low degree of overlap, if the reduce tasks occupy all nodes i.e., the number of reduce tasks (r) is larger than or equal to n, only a portion of the merging of the intermediate results can be overlapped with the map computation, and the total time can be expressed by Equation (1), where o represents the reduction in the merging time.

$$T_{MR} = \textit{Map time} + (\textit{Merge time} - o) + \textit{Reduce time} \quad (1)$$

However, when the reduce tasks do not occupy all nodes, more merging can be overlapped with the map computations due to the load balancing effect i.e., the nodes with running reduce tasks process a smaller number of map tasks compared to the other nodes. This is because Hadoop uses greedy task scheduling; the nodes without running reduce tasks process map tasks faster, and in turn, are assigned more map tasks. As a result, the *Map time* is increased and the merging time is decreased as shown in Equation (2), where l represents the load balancing effect, o' represents the overlapping effect, and $o' > o$. As r increases, l and o' keeps decreasing, until reaching 0 and o respectively when $r = n$ (Equation (1)).

$$T_{MR} = (\textit{Map time} + l) + (\textit{Merge time} - o') + \textit{Reduce time} \quad (2)$$

B. HR

For hierarchical reduction, map and reduce processing at different stages can be overlapped, as shown in Figure 5. To compare the performance of HR with that of MR, we consider more detailed modeling.

For HR, when all map tasks are finished, the remaining computations are to reduce the un-reduced map tasks in addition to combining the results of this reducing stage with other partial reduced results. Specifically, the total execution time of MR, and HR can be represented by the following equations, where C'_{MR} is the communication required to retrieve the remaining map outputs since MR's communication is overlapped with the map computations, and s is the remaining number of reduce stages in the HR's hierarchy:

$$T_{MR} = \mathit{Map\ time} + C'_{MR} + \lceil\frac{k}{r}\rceil log(\lceil\frac{k}{r}\rceil) + t_{rk} \times \lceil\frac{k}{r}\rceil \quad (3)$$

$$T_{HR} = \mathit{Map\ time} + (C_{HR} + \frac{\sigma_H k}{m} log(\frac{\sigma_H k}{m}) + t_{rk} \times \frac{\sigma_H k}{m}) \times s \quad (4)$$

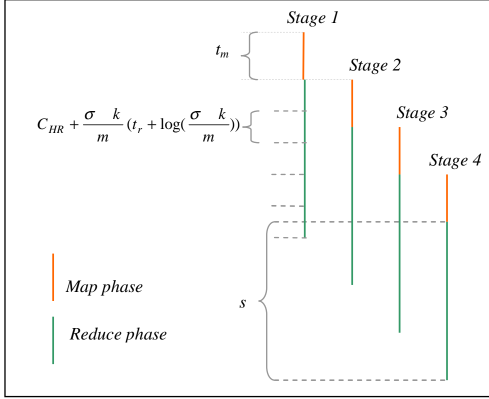

Fig. 5. Execution of HR framework when $m = 8n$

Assuming every map task produces a value for each given key, then C'_{MR} is $\frac{k \times 2n}{r \times m} \times C$, then C_{HR} is $\frac{\sigma_H k}{m} \times C$, and C_{HR} equals $\frac{\sigma_H r}{2n} \times C'_{MR}$, where C is communication cost per key/value pair. By substituting C_{HR} in Equation (4) by the previous value, the equation becomes:

$$T_{HR} = \mathit{Map\ time} + (\frac{\sigma_H r}{2n} \times C'_{MR} + \frac{\sigma_H}{m} \times (klog(\frac{\sigma_H k}{m}) + kt_{rk})) \times s \quad (5)$$

When the overlapping degree is high, s can be replaced by $(log_{\sigma_H}(2n) + 1)$ in Equation (5), which represents the reduction of the map tasks of the final stage. Moreover, the merging time can be eliminated from Equation (3). So, for significantly large m, the reducing part of Equation (5) is smaller than that of Equation (3). If this term occupies a significant portion of the total time of MR, and the communication overhead is small, then HR will behave better than MR as we will see in the experimental evaluation. However, when the overlapping degree is low, s can be very deep, and the performance of HR can be worse than MR.

C. IR

The execution of IR relative to MR is represented by Figure 4. Particularly, when the overlapping degree is high, the final merging, and reducing phase of IR can be eliminated. So the total execution time can be represented by *Map time*. By comparing this to MR, we can conclude that IR behaves better than MR especially when the reducing time of MR is significant.

For low overlapping degree, if r is larger than n, then merging and reducing of only a portion of the intermediate results can be overlapped with the map computations, and the total time can be expressed by Equation (6). Where o_m and o_r represents the reduction in the merging and reducing time respectively.

$$T_{IR} = \mathit{Map\ time} + (\mathit{Merge\ time} - o_m) + (\mathit{Reduce\ time} - o_r) \quad (6)$$

To compare this with Equation (2), we consider the details of the overlapping computations in MR and IR. The main difference is that IR performs reducing after merging and writes the results of reduce rather than the results of merge to disk. Assuming the reduce function changes the size of the input by a factor of x and the reduce function is linear, then the overlapped computations of MR (O_{MR}) and IR (O_{IR}) can be represented by the following equations, where I is the size of intermediate key/value pairs to be merged and reduced during the map phase, $IlogI$ is the average number of compare operations executed during the merge, p_s is the processor speed, d_s is the disk write speed.

$$O_{MR} = \frac{I_{MR} logI_{MR}}{p_s} + \frac{I_{MR}}{d_s} \quad (7)$$

$$O_{IR} = \frac{I_{IR} logI_{IR} + I_{IR}}{p_s} + \frac{I_{IR} \times x}{d_s} \quad (8)$$

Given the same overlapping degree, if $x < 1$, which is valid for applications like wordcount, grep, linear regression etc., then IR is able to conduct more merging in addition to reducing overlapped with map computations. So, the merging and reducing terms in equation 6 is less than the same terms in equation 1. So, IR can behave better than MR given the reduce computations is signifiant as illustrated by Figure 4. On the other side, if $x \geq 1$ as in sort application, then the performance of IR highly depends on the complexity of the reduce function compared to the merging, in addition to the size of the intermediate key/value pairs.

By applying the previous analysis to the case where $r < n$, we can conclude that IR can behave better than MR in this case given also the reducing time occupies a significant portion of the total execution time.

V. Experimental Analysis

In this section, we present a rigorous performance evaluation of our proposed techniques. The details of the system configurations are given along with the conducted experiments.

A. Experimental Platform

We ran our experiments on System X at Virginia Tech, comprising Apple Xserve G5 compute nodes with dual 2.3GHz PowerPC 970FX processors, 4GB RAM, and 80GB hard drives. The compute nodes are connected with a Gigabit Ethernet interconnection. Each node is running the GNU/Linux operating system with kernel version 2.6.21.1. The proposed approaches are developed based on Hadoop 0.17.0.

B. Overview

We use two applications i.e., *wordcount* and *grep* in the experiments. Wordcount is an application that parses a document or a number of documents, and produces for every word the number of its occurrences. Grep accepts a document or a number of documents and an expression as input, and it matches this expression along the whole documents and produces the number of occurrences for every match.

We first aim at studying the scalability of the different reducing approaches in terms of the dataset size. Next, we seek to understand the reasons of performance difference between different approaches by profiling the behaviors of wordcount and grep under various configurations. Finally, we evaluate various reducing approaches on emulated cloud environments.

The major performance metric in all experiments is the total execution time in seconds. Moreover, for a fair comparison, we follow the guidance given in the Hadoop documentation regarding the number of map and reduce slots per node as well as the thresholds that control the frequency of merging and sorting intermediate results (i.e., *mapred.inmem.merge.threshold* and *io.sort.factor* as discussed in Section III-B). Furthermore, we enable the combiner function in all experiments In addition, the aggregation level of the hierarchical reduction approach is set to 4, which produces the best results on the 32-node cluster. Before executing an experiment, we flush the Linux file system cache by having each node read a dummy file that is larger than the size of the memory. This avoids the I/O performance inconsistency caused by the Linux file caching.

C. Scalability with the Dataset Size

In this experiment, we aim at studying the scalability of the three reducing approaches with regard to the size of the input dataset. We run wordcount and grep using data sets of two different sizes, i.e., 16GB and 64GB. For wordcount, the number of reduce tasks was set to 4 and 8 (a broader range were used in Section V-D). For grep, we use a query that produces results of moderate size. The performance of queries generating various sizes of output will be investigated in Section V-E.

As shown in Figure 6, generally, as the size of the input dataset increases, the performance improvement of IR over MR increases. Specifically, for wordcount, as the size of the input increases to 64GB, IR outperforms MR by 34.5% and 9.48% instead of 21% and 5.97% for 16GB input using 4 and 8 reduce tasks, respectively. In addition, for grep, increasing the input size to 64GB improves IR's performance gain over MR; IR is better than MR by 16.2% (for 64GB input) instead

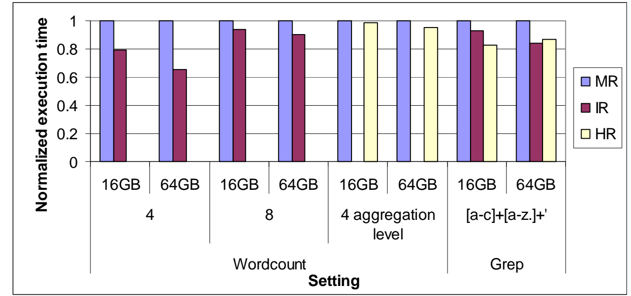

Fig. 6. Scalability with dataset size using wordcount and grep

of 7.1% (for 16GB input). The scalability of IR attributes to two reasons that provide more room for overlapping map processing and reduce computations. First, as the dataset size increases, the map phase has to perform more I/O. Second, for 64 GB, the number of map tasks increases from 256 to 1024, thus increasing the scheduling overhead.

Similarly, the performance improvement of HR over MR increases by 9.96% for wordcount as the size of input dataset increases. However, the speedup of HR over MR decreases (by 4.13%) for grep when larger input is used. This is because the extra communication cost caused by the increase of input data can be compensated by the corresponding increase in the map processing time in wordcount but not grep.

Since the normal Mapreduce jobs process large amounts of input data, all of the subsequent experiments will use input datasets of 64 GB.

D. Wordcount Performance

In a cluster of 32 nodes, we run wordcount on a dataset of size 64 GB. The number of map tasks is set to 1024, and the number of reduce task is varied from 1 to 64. As shown in Figure 7, as the number of reduce tasks increases, IR's improvement over MR decreases. Specifically, for one reduce task, IR behaves better than MR by 35.33%. When the number of reduce tasks is increased to 4, IR behaves better by 34.49%. As the number of reduce tasks increases, the processing time of a reduce task decreases, thus providing little room for overlapping the map and reduce processing. Specifically, when the number of reduce tasks is 32, a reduce task only consume a mere 6.83% in the total execution time as shown in table II.

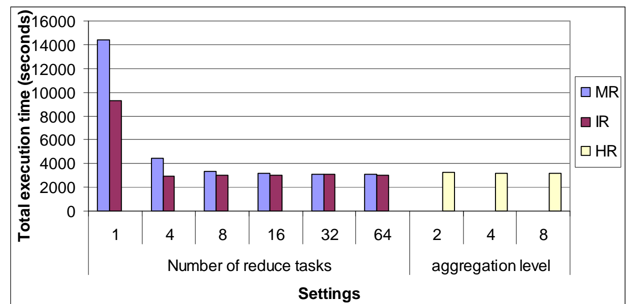

Fig. 7. Performance of MR vs. IR using wordcount

Reduce Tasks	Incremental merges	Reduce Time	Map Time
1	46	11736	2690
4	55	1699.5	2732
8	13	524	2787
16	2	315	2845
32	0	210	2865

TABLE II
MR AND IR EXECUTION PROFILE.

Reduce Tasks	MR	IR
1	21	21
4	22	25 + 1S
8	26 + 2S	28
16	30 + 2S	28 + 2S
32	31	32

TABLE III
NUMBER OF MAP TASKS EXECUTED ON A CODE WITH REDUCE TASK(S) RUNNING. *S* MEANS A SPECULATIVE TASK.

Furthermore, IR achieves its best performance at 4 reduce tasks because this provides the best compromise between level of parallelism controlled by the number of reduce tasks and overlapping map with reduce. Specifically, in this case, IR conducts 55 incremental merges overlapped with the map computations compared to 0 merges with 32 reduce tasks, as shown in Table II. As a result, the nodes executing a reduce task executes 26 map tasks instead of 32 map tasks in case of 32 reduce tasks, recall the *load balancing effect* discussed in section IV. Specifically, when only a small portion of the nodes are executing reduce tasks, these nodes will have less resources to spend on the map processing, and thus more map tasks are "pushed off" to the other nodes without reduce tasks.

The best performance is achieved at 32 and 4 reduce tasks for MR, and IR respectively. With the best performance for both MR and IR, IR is better by 5.86%.

On the other side, using an aggregation level of 4, HR behaves better than MR with 8 reduce tasks by 5.13%. We changed the aggregation level from 2 to 8 as shown in Figure 7. The best performance is achieved for the aggregation level 4, when a best balance is achieved between the overlap of map and reduce processing and the reduce overhead. For example, with an aggregation level of 2, a reduce operation can be triggered sooner, but the reducing hierarchy is also deeper.

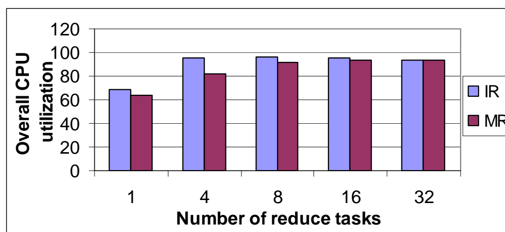

Fig. 8. CPU utilization throughout the whole job using wordcount

To better understand the benefits of the incremental reduction approach, we measured the CPU utilization throughout the job execution, and the number of disk transfers per second during the map phase for both MR and IR. As shown in

Concurrent jobs	MR Execution Time (seconds)	IR Execution Time (seconds)
1	3107.5	2925.5
2	6064	5687
3	9025.5	8303

TABLE IV
MR AND IR PERFORMANCE WITH CONCURRENT JOBS.

Figure 8 and 9, the CPU utilization of IR is greater than MR by 5% on the average. In addition, the average disk transfers of IR is less than MR by 2.95 transfers per second. This attributes to the smaller amount of data written to disk by IR, because it reduces the intermediate data before writing it back to disk. In doing so, IR also reduces the size of data read from disk at the final merging and reducing stage.

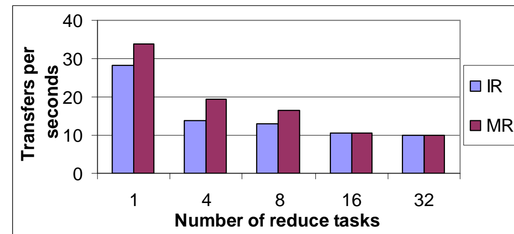

Fig. 9. Number of disk transfers per second through the map phase using wordcount.

To conclude, for any number of reduce tasks, IR achieves either better or same performance as MR. And the best performance for IR is achieved using only 4 reduce tasks, This means that IR is more efficient in utilizing the available resources. So, we expect IR to achieve better performance when several jobs are running at the same time, or with larger amounts of reduce processing. Particularly, when running three concurrent jobs of wordcount, the best configuration of IR behaves better than the best configuration of MR by 8.01% instead of 5.86% as shown in table IV.

E. Grep Performance

In a cluster of 32 nodes, we run grep on a dataset of 64 GB. The number of map tasks is set to 1024, and the number of reduce tasks is set to the default value i.e., one. Grep runs two consecutive jobs; one returns for each match the number of its occurrence, and the other is a short job that inverts the output of the previous job so that the final output will be sorted based on occurrence of the matches instead of alphabetically. In this experiment, we focus on the first longer job. We used five different queries each produces a different number of matches and hence different numbers of intermediate and final key/value pairs.

As shown in Figure 10, IR and HR deliver very similar performance. In addition, for the first query, all reducing approaches have the same performance. For subsequent queries, HR and IR outperform MR, and the performance improvement of HR/IR over MR increases along with the number of matches.

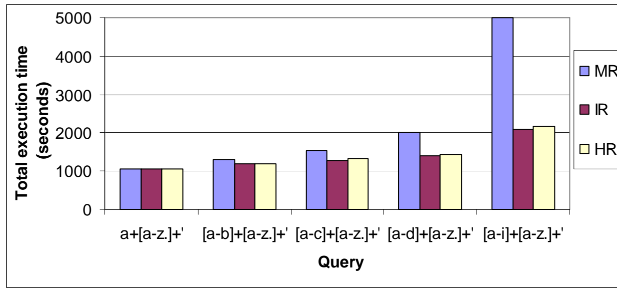

Fig. 10. Performance of MR, IR, and HR using grep.

Query	Reduce Time (seconds)	Intermediate Data Size (records)
a+[a-z.]+	135	37,285,680
[a-b]+[a-z.]+	250	55,897,216
[a-c]+[a-z.]+	351	78,196,736
[a-d]+[a-z.]+	742	113,039,360
[a-i]+[a-z.]+	1569	306,921,472

TABLE V
CHARACTERISTICS OF DIFFERENT QUERIES

F. Heterogeneous Environment Performance

Nowadays data centers are becoming incrementally heterogeneous, due to the use of virtualization and/or machines from different generations. In this experiment, we aim at studying the robustness of MR, HR, and IR to the heterogeneity of the target cluster. In a cluster of 32 nodes, we manually slow down several nodes i.e., 10 nodes to mimic a heterogeneous cluster We continuously run *dd command* to convert and write a large file (e.g. 5.7 GB) to disk in order to slow down a given node. This approach was used in a related study by Zahria et al. [13].

We expect in these environments, the map phase time gets longer due to the effects of the slow nodes. So, if the reduce tasks are appropriately assigned to the fast nodes, then utilizing the extra map time in reduce computations could improve the performance of the proposed approaches. Using wordcount, we run MR, and IR with the best configuration achieved in Section V-D i.e., 32 reduce tasks for MR and 4 reduce tasks for IR. As shown in Figure 11, when the reduce tasks are assigned to the fast nodes, IR becomes better than MR by 10.33% instead of 5.86%. However, when they are randomly assigned, IR becomes better than MR by only 2.32%. This is expected because the I/O and computing resources available for reduce tasks in this case are limited, preventing IR from taking advantage of overlap between the map and reduce processing. We argue that if the heterogeneity originates from different generations of hardware or from virtualization, it is possible to identify the fast nodes and assign more reduce tasks to these nodes.

Note that HR's performance drops significantly when running in the heterogeneous environment. This can be attributed to the large number of generated reduce tasks in HR. In addition, it is indeterministic where these tasks will be run, so it is not possible to avoid the effect of the slow nodes.

To simulate a cloud environment with slower I/O performance because of virtualization, we slow down 32 nodes and repeat the experiment. As we can see, IR still outperforms MR by 7.14%.

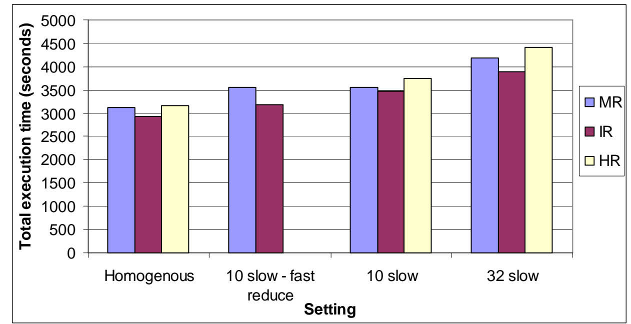

Fig. 11. Wordcount performance in homogeneous and heterogeneous environments. In the second setting, 10 nodes are slowed down, and the reduce tasks are scheduled on the fast nodes. Currently, where reduce tasks will be run in HR is not controllable.

VI. Related Work

Several research efforts have been done to enhance the MapReduce framework. Sawzall is a programming language built on top of MapReduce [9]. It aims at automatically analyzing huge distributed data files. The main difference between Sawzall and the standalone Mapreduce framework is that Sawzall distributes the reduction in a hierarchical topology-based manner. In [10], the authors improved the performance of Google's MapReduce framework by pipelining disk and network I/O. Particularly, they aimed at streaming intermediate data as it is generated and uses local storage as a write-ahead log for network transfer. In [12], the authors believe that the original MapReduce framework is limited in supporting applications like relational data processing. So they presented a modified version of the MapReduce named MapReduceMerge, where the reduce workers produce a list of key/values pairs that are transmitted to a set of merge workers for further processing.

Moreover, Valvag et al. developed a high-level declarative programming model and its underlying runtime, Oivos, which aims at handling applications that require running several MapReduce jobs [11]. This framework has two main advantages compared with MapReduce. First, it reduces the overheads associated with such type of applications that includes monitoring the status and progress of each job, determining when to re-execute a failed job or start the next one, and specifying a valid execution order for the MapReduce jobs. Second, it removes the extra synchronization when these applications are executed using the traditional MapReduce framework, i.e., every reduce task in one job should complete before any of the map tasks in the next job can start.

Steve et al. realized that the loss of intermediate map outputs may result in a significant performance degradation [6]. And although using HDFS (Hadoop Distributed File System) improves the reliability, it results in considerably increase in the job completion time. As a result, they proposed some

design ideas for a new intermediate data storage system. Zahria et al. [13] proposed a speculative task scheduling named LATE (Longest Approximate Time to End) to cope with several limitations of the original Hadoop's scheduler in heterogeneous environments such as Amazon EC2[1].

Finally, Condie et al. [3] recently extended the MapReduce architecture to work efficiently for online jobs in addition to batch jobs. Instead of materializing the intermediate key/value pairs within every map task, they proposed pipelining these data directly to the reduce tasks. They further extended this pipelined MapReduce to support interactive data analysis through online aggregation, and continuous query processing.

To conclude, our proposed solutions are orthogonal to the above extensions to MapReduce framework and can complement their improvements.

VII. Conclusions and Future Work

In this paper, we designed and implemented two approaches to reduce the overhead of the barrier synchronization between the map and reduce phases of Hadoop, an open source implementation of MapReduce. In addition, we evaluated the performance of these approaches against Hadoop using analytical model and experiments on a 32-node cluster.

The first proposed approach is the hierarchical reduction, which overlaps map and reduce processing at the inter-task level. It starts a reduce task as soon as a certain number of map tasks complete and aggregates partial reduced results following a tree hierarchy. This approach can be effective when there is enough overlap between map and reduce processing. However, this approach has some limitations due to the overheads of creating reduce tasks on the fly, in addition to the extra communication cost of transferring the intermediate results along the tree hierarchy. To cope with these overheads, we proposed the incremental reduction approach, where all reduce tasks are created at the start of the job, and every reduce task incrementally reduces the received map outputs. The experimental results have shown that this approach consistently outperforms the hierarchical reduction approach and the original Hadoop implementation.

In the future, we plan to evaluate our techniques to a broader class of applications. We will also investigate in generalizing our techniques to support applications which are not recursively reducible in their original form. Finally, we will evaluate the performance of the proposed techniques on real cloud environments.

Disclaimer

This paper is based on version 0.17.0 of Hadoop. We completed the major design and development of our approaches in June 2009. Recently, we found that in concurrence with our project development, Hadoop extended the implementation of the original combiner function, using a design similar to the incremental reduction (IR) approach proposed in our paper.

In the most recent version of Hadoop, the combiner function is triggered in a reduce task during the in-memory merge of map outputs. In our design, the incremental reduction occurs at *both* in-memory merge and on-disk merge. We also present an alternative design, i.e., hierarchical reduction. Finally, in addition to our proposed designs, we presented a rigorous performance evaluation and analytical analysis of incremental reduction (IR) and hierarchical reduction (HR).

Acknowledgement

This research is supported in part by NSF grant CNS-0915861.

References

[1] Amazon.com. Amazon elastic compute cloud. http://www.amazon.com/gp/browse.html?node=201590011.

[2] S. Chen and S. Schlosser. Map-reduce meets wider varieties of applications meets wider varieties of applications. Technical Report IRP-TR-08-05, Intel Research, 2008.

[3] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, University of California at Berkeley, 2009.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems, Design & Implementation, OSDI*, 2004.

[5] M. Grant, S. Sehrish, J. Bent, and J. Wang. Introducing map-reduce to high end computing. 3rd Petascale Data Storage Workshop, Nov 2008.

[6] S. Ko, I. Hoque, B. Cho, and I. Gupta. On Availability of Intermediate Data in Cloud Computations. In *12th Workshop on Hot Topics in Operating Systems (HotOS XII)*, 2009.

[7] G. Mackey, S. Sehrish, J. Bent, J. Lopez, S. Habib, and J. Wang. Introducing mapreduce to high end computing. In *Petascale Data Storage Workshop Held in conjunction with SC08*, 2008.

[8] A. Matsunaga, M. Tsugawa, and J. Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics. Microsoft eScience Workshop, 2008.

[9] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[10] Alex Rasmussen and Maximus Daehan Choi. Improving performance in mapreduce through pipelining. 2006.

[11] S.V. Valvag and D. Johansen. Oivos: Simple and efficient distributed data processing. In *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, pages 113–122, Sept. 2008.

[12] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.

[13] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI: USENIX Symposium on Operating Systems Design and Implementation*, 2008.