Cross-Phase Optimization in MapReduce

Benjamin Heintz, Chenyu Wang, Abhishek Chandra, and Jon Weissman
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, MN
{*heintz,chwang,chandra,jon*}*@cs.umn.edu*

***Abstract*—MapReduce has been designed to accommodate large-scale data-intensive workloads running on large single-site homogeneous clusters. Researchers have begun to explore the extent to which the original MapReduce assumptions can be relaxed including skewed workloads, iterative applications, and heterogeneous computing environments. Our work continues this exploration by applying MapReduce across widely distributed data over distributed computation resources. This problem arises when datasets are generated at multiple sites as is common in many scientific domains and increasingly e-commerce applications. It also occurs when multi-site resources such as geographically separated data centers are applied to the same MapReduce job. Using Hadoop, we show that the absence of network and node homogeneity and locality of data lead to poor performance. The problem is that interaction of MapReduce phases becomes pronounced in the presence of heterogeneous network behavior. In this paper, we propose new cross-phase optimization techniques that enable independent MapReduce phases to influence one another.**

We propose techniques that optimize the push and map phases to enable push-map overlap and to allow map behavior to feed back into push dynamics. Similarly, we propose techniques that optimize the map and reduce phases to enable shuffle cost to feed back and affect map scheduling decisions. We evaluate the benefits of our techniques in both Amazon EC2 and PlanetLab. The experimental results show the potential of these techniques as performance is improved from 7%–18% depending on the execution environment and application.

***Keywords*-MapReduce, Cloud, Distributed, Scheduling**

I. Introduction

The scale of data being generated globally has been increasing dramatically over the past few years. MapReduce [1] has emerged as a popular programming paradigm for data processing and analysis over large datasets in large-scale platforms. Fueling this growth is the usage of Hadoop [2], the open-source MapReduce variant, as well as the emergence of cloud computing and services such as Amazon EC2 [3].

MapReduce is traditionally deployed on local, tightly coupled clusters, with the input data pre-placed within the cluster. It is assumed that the input is either generated close to the cluster, or can be moved to it easily. However, with the popularity of Cloud storage services and the improvement of personal storage devices, user data are now distributed around the world. Moreover, many Internet services such as Google and Facebook consist of multiple data centers, and Content Distribution Networks such as Akamai also place their data close to the users in a highly distributed manner. Scientific datasets—such as in climate, geology, and health-care domains—are generated in a geographically distributed manner due to the distributed nature of instruments, sensors, and patient records. Thus, *it is increasingly common for data to be generated at multiple sites spread around the globe.*

Besides the data, computing resources themselves are increasingly distributed. This is true for many cloud services today that utilize multiple data centers and sites (such as Amazon EC2 regions). Scientific computing has used this model traditionally through the use of Grids, which inherently consist of federated, multi-site computing resources.

Given the distributed nature of data and compute resources, a key question is whether MapReduce is well suited for data analysis in a highly distributed environment. Relaxing the assumptions of the original MapReduce in terms of execution domain and suitable applications is an active area of pursuit by many researchers. Our line of inquiry continues along this path. As we show through experiments in Section II, under a widely distributed environment with high network heterogeneity, Hadoop does not always perform well. We find that the main reason for this performance degradation is the interaction and heavy dependency across different MapReduce phases. This happens because the data placement and task execution are highly coupled in the MapReduce paradigm (because MapReduce attempts to assign tasks to nodes that already host input data for those tasks[1]). Thus, the decisions on where to place data severely impact the scheduling decision on where map and reduce tasks are executed, and vice versa. In a heterogeneous environment, particularly one with slow wide-area links, such coupling can severely impact the end-to-end performance by creating bottleneck links and nodes in the execution. We show that Hadoop's default data placement and scheduling mechanisms do not take such cross-phase interactions into account, and while they may try to optimize individual phases, they could result in globally bad decisions, resulting in poor overall performance. The problem is not that a given phase cannot take into account

[1]When this is not possible, tasks must read their inputs remotely, and scarce network bandwidth becomes a limiting factor.

decisions made by a prior phase, but rather the opposite. Upstream decisions may limit the flexibility of later phases, and thus, earlier phase decisions must account for their impact later on.

In order to overcome these limitations, we propose techniques to achieve this type of cross-phase optimization in MapReduce. The key idea behind our proposed techniques is to consider not only the execution cost of an individual task or computational phase, but also its impact on the performance of subsequent phases. We propose two sets of techniques:

- *Map-aware Push:* Traditional MapReduce assumes that the input data are pushed to compute nodes before execution starts. We instead propose making the input data push aware of the cost of map execution, based on the source-to-mapper link capacities as well as mapper node computation speeds. We achieve this by overlapping the data push with map execution, which provides us with two benefits. The first benefit is a pipelining effect which hides the latency of data push with the map execution. The second benefit is a dynamic feedback between the map and push that enables nodes with higher speeds and faster links to process more data at runtime.
- *Shuffle-aware Map:* In traditional MapReduce, the typical shuffling of intermediate data from mappers to reducers is an all-to-all operation. However, in a heterogeneous environment, a mapper with a slow outgoing link can become a bottleneck in the shuffle phase, slowing down the downstream reducers. We propose map task scheduling based on the estimated shuffle cost from each mapper to enable faster shuffle and reduce execution.

We have implemented our techniques in the Hadoop framework. We evaluate the benefits of our techniques in both Amazon EC2 and PlanetLab. The experimental results show the potential of these techniques as performance is improved from 7%–18% depending on the execution environment and application.

This paper is organized as follows. In Section II, we empirically demonstrate the limitations of MapReduce execution under a wide-area computation environment. We then present our proposed techniques: *Map-aware Push* in Section III and *Shuffle-aware Map* in Section IV. We show experimental results by putting these techniques together in an end-to-end manner in Section V. We present related work in Section VI and conclude in Section VII.

II. MapReduce Performance in Widely Distributed Environments

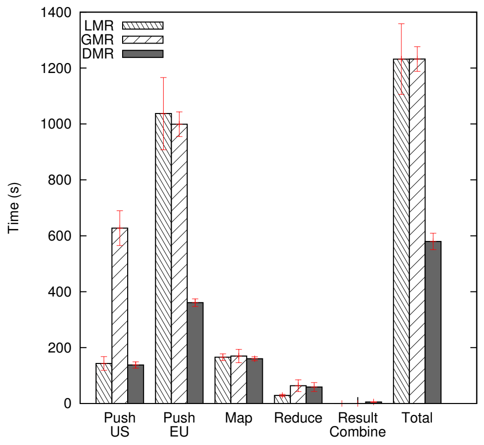
In this section, we empirically demonstrate the challenge of efficiently executing MapReduce over a widely distributed environment through experiments conducted on the PlanetLab and Amazon EC2 platforms. We present only a subset of results from our experiments here due to space constraints, and more details are available in another paper [4].

A typical MapReduce job executed in a cluster environment consists of three main phases: (i) *Map*, where map tasks execute on their input data; (ii) *Shuffle*, where the output of map tasks (intermediate key-value pairs) are disseminated to reduce tasks; and (iii) *Reduce*, where reduce tasks are execute on the intermediate data to produce the final outputs. It is typically assumed that the input data are already available on the compute nodes before the job execution is started. Such *data push* is usually achieved through file system mechanisms such as those provided by HDFS. However, when data sources are geographically distributed, such as across multiple data centers, the process of pushing data to the compute nodes may itself be costly, and hence, must be considered as a separate phase of the overall computation.

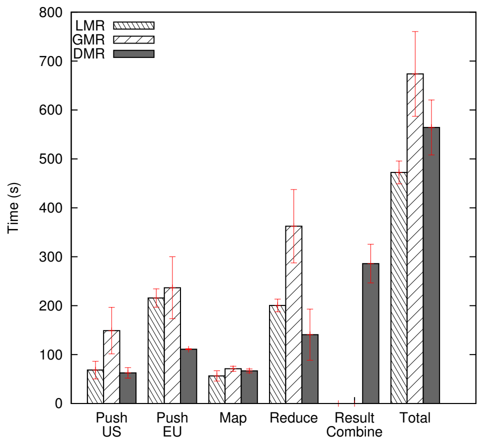
To illustrate this problem, we compare three possible architectures for MapReduce execution: (i) *Local MapReduce (LMR)*, where all data are first moved into a centralized cluster followed by the execution of a local MapReduce job within that cluster; (ii) *Global MapReduce*, where all the widely distributed compute resources are considered as a single MapReduce cluster without considering the network heterogeneity; and (iii) *Distributed MapReduce (DMR)*, where multiple MapReduce jobs are first executed close to the data sources, followed by a final centralized combination of their outputs. Note that LMR is the typical way in which MapReduce computation is done today.

In our experimental setup, we used a total of eight PlanetLab nodes in two widely separated clusters—four nodes in the US, and four nodes in Europe. In addition, we used one node in each cluster as a data source. For each cluster, we chose tightly coupled machines with high inter-node bandwidth (i.e., they were either co-located at the same site or share some network infrastructure). The intra-cluster bandwidth was between 1.5–2.5 $\mathrm{MB/s}$, while the inter-cluster bandwidth between any pair of nodes (between US and EU) was around 300–500 $\mathrm{KB/s}$.

Figures 1(a) and 1(b) show the results of executing WordCount on plain-text and random input data respectively in this environment. These two scenarios correspond to different application/data characteristics: one (WordCount with plain-text) where input data are aggregated into much smaller intermediate data, and the other (WordCount with random data) where input data expand into larger intermediate data, increasing the shuffle costs. Within each graph, we show the time of individual phases: Push US/EU corresponding to data push from the US and EU data sources respectively, followed by the map and reduce phases. The Result Combine phase is the combine phase for the DMR architecture only, comprising the data transmission plus combination costs, and this phase corresponds to a logical shuffle/reduce of the intermediate results.



(a) WordCount on 800 MB plain-text data



(b) WordCount on 250 MB random data

Figure 1. Hadoop execution on PlanetLab. In (a), DMR finishes the fastest by avoiding the transfer of large input data over slow links, while in (b), LMR finishes faster since it minimizes the larger intermediate and output data transfer costs.

From the results, we make the following observations. From Figure 1(a), we see that in a wide-area environment, the cost of moving input data to the compute nodes (the data push phase) for LMR can be significant, which impacts the overall execution time of the job despite the map and reduce phases being relatively efficient. In fact, these datasets are small by MapReduce standards, but the size of data and its distributed nature present orthogonal challenges; here we focus on the challenges arising from widely distributed data.

Since the choice of mapper nodes depends on where data are pushed, pushing data to nearby nodes (as for DMR) is much more efficient when the volume of intermediate data is small relative to that of the input data. On the other hand, from Figure 1(b), we see that when the volume of intermediate data is much larger than that of the input data, the cost of combining intermediate results can be the dominant factor. In particular, in this case, shuffling and merging the intermediate results close to the mappers (as for LMR) is much more efficient. GMR performs poorly in both cases, as it does not use network locality either in the push or the result combine phases. Our experiments on the Amazon EC2 environment showed similar results [4].

Overall, these results illustrate the close interdependency between the different stages of a MapReduce execution. In particular, the choice of mapper nodes to which inputs are pushed impacts both how long the data push takes, as well as where the intermediate data are generated. This in turn impacts the performance of the data shuffle to the reducers. This problem is particularly severe for wide-area environments, since they are typically heterogeneous in terms of node and link capacities. Therefore, it is important to optimize the overall end-to-end computation as a whole while taking into account the network and platform characteristics. We now describe the techniques that achieve our aims.

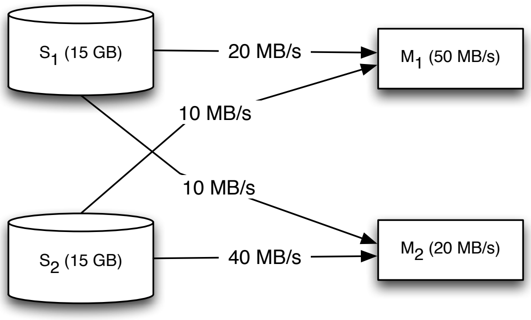


Figure 2. A simple example network with two data sources and two mappers

III. Map-aware Push

The first opportunity for cross-phase optimization in MapReduce lies at the boundary between the push and map phases. A typical practice is what we call a *push-then-map* approach, where input data are imported in one step, and computation begins only after the data import completes. This approach has two major problems. First, by forcing all computation to wait for the slowest communication link, it introduces waste. Second, separating the push and map phases deprives map tasks of a way to demand more or less work based on their compute capacity. This makes scheduling the push in a map-aware manner more difficult. To overcome these challenges, we propose two changes: first, overlapping—or pipelining—the push and map phases; and second, inferring locality information at runtime and driving scheduling decisions based on this knowledge.

Before discussing how we implement our proposed approach in Hadoop and showing experimental results, we describe in more detail the problems of a push-then-map approach and how our proposed *Map-aware Push* technique addresses them. To begin, consider the simple example environment shown in Figure 2, comprising two data sources S_1 and S_2 and two mappers M_1 and M_2. Assume that each data source initially hosts 15 GB of data and that the link bandwidths and mapper computation rates are as shown in the figure.

A. *Overlapping Push and Map to Hide Latency*

With these network links, the following push distribution would optimize the push (i.e., minimize push time): S_1 pushes 10 GB to M_1 and 5 GB to M_2, and S_2 pushes 3 GB to M_1 and 12 GB to M_2. If we were to use this distribution with a push-then-map approach, then the entire push would finish after 500 s even though source S_2 would finish its part after only 300 s. Map computation would begin after 500 s and continue until 1350 s.

If we instead were to overlap the push and map, allowing map computation to begin at each mapper as soon as data begin to arrive, then we could avoid this unnecessary waiting. For example, assuming that we used this same optimal push distribution, then mapper M_2, with slower compute capacity than its incoming network links, would be the bottleneck, finishing after 850 s. By simply overlapping the map and push phases, we could reduce the total push and map runtime by about 37% in this example.

B. *Overlapping Push and Map to Improve Scheduling*

The second problem arises due to the lack of feedback from the map phase to the push phase. Without this feedback, and absent any *a priori* knowledge of the map phase performance, we are left with few options other than simply optimizing the push phase in isolation. Such a single-phase optimization favors pushing more data to mappers with faster incoming network links. In our example, however, it is the mapper with *slower* network links (M_1) that is actually better suited for map computation. Unfortunately, by pursuing an optimal push phase, we end up with more data at M_2 and in turn roughly 3.3x longer map computation there than at M_1. For better overall performance, we need to weigh the two factors of network bandwidth and computation capacity and trade off between faster push and faster map. Continuing with our simple example, we should tolerate a slightly slower push in order to achieve a significantly faster map by sending more data to mapper M_1. In fact, in an optimal case, we would send 60% of all input data there, yielding a total push-map runtime of only 600 s, or a 55% reduction over the original push-then-map approach.

C. *Map-aware Push Scheduling*

Of course, this raises the question of how we can schedule push and map jointly, respecting the interaction between the two phases. As we have argued, this is difficult to do when the push and map phases are separated. With overlapped push and map, however, the distribution of computation across mapper nodes can be demand-driven. Specifically, whereas push-then-map first pushes data from sources, our approach logically *pulls* data from sources on-demand. Using existing Hadoop mechanisms, this on-demand pull is initiated when a mapper becomes idle and requests more work, so faster mappers can perform more work. This is how our proposed approach respects map computation heterogeneity.

To respect network heterogeneity, our *Map-aware Push* technique departs from the traditional Hadoop approach of explicitly modeling network topology as a set of racks and switches, and instead infers locality information at runtime. It does this by monitoring source-mapper link bandwidth at runtime and estimating the push time for each source-mapper pair. Specifically, let d be the size of a task in bytes (assume for ease of presentation that all task sizes are equal) and let $L_{s,m}$ be the link speed between source node s and mapper node m in bytes per second. Then we estimate the push time $T_{s,m}$ in seconds from source s to mapper m as

$$T_{s,m} = \frac{d}{L_{s,m}} \ . \tag{1}$$

Let S denote the set of all sources that have not yet completed their push. Then when mapper node m requests work, we grant it a task from source $s^* = \arg\min_{s \in S} T_{s,m}$. Intuitively, this is equivalent to selecting the closest task in terms of network bandwidth. This is a similar policy to Hadoop's default approach of preferring *data-local* tasks, but our overall approach is distinguished in two ways. First, rather than *reacting* to data movement decisions that have already been made in a separate push phase, it *proactively* optimizes data movement and task placement in concert. Second, it discovers locality information dynamically and automatically rather than relying on an explicit user-specified model.

D. *Implementation in Hadoop*

Now we can discuss how we have implemented our approach in Hadoop 1.0.1. First, the overlapping itself is possible using existing Hadoop mechanisms, but a more creative deployment. Specifically, we set up a Hadoop Distributed File System (HDFS) instance comprising the data source nodes, which we refer to as the "remote" HDFS instance and use directly as the input to a Hadoop MapReduce job. Map tasks in Hadoop typically read their inputs from HDFS, so this allows us to directly employ existing Hadoop mechanisms.[2]

Our scheduling enhancements, on the other hand, require modification to the Hadoop task scheduler. To gather the bandwidth information mentioned earlier, we add a simple

[2]To improve fault tolerance, we have also added an option to cache and replicate inputs at the compute nodes. This reduces the need to re-fetch remote data after task failures or for speculative execution.

Table I
MEASURED BANDWIDTHS IN THE EC2 EXPERIMENTAL SETUP

From	To	Bandwidth (MB/s)
Source EU	Worker EU	8
Source EU	Worker US	3
Source US	Worker EU	3
Source US	Worker US	4
Worker EU	Worker EU	16
Worker EU	Worker US	2
Worker US	Worker EU	5
Worker US	Worker US	2

network monitoring module which records actual source-to-mapper link performance and makes this information accessible to the task scheduler. For Hadoop MapReduce jobs that read HDFS files as input, each map task corresponds to an InputSplit which in turn corresponds to an HDFS file block. HDFS provides an interface to determine physical block locations, so the task scheduler can determine the source associated with a particular task and compute its $T_{s,m}$ based on bandwidth information from the monitoring module. If there are multiple replicas of the file block, then $T_{s,m}$ can be computed for each replica, and the system can use the replica that minimizes this value. The task scheduler then assigns tasks from the closest source s^* as described earlier.

E. Experimental Results

We are interested in the performance of our approach, which overlaps push and map and infers locality at runtime, compared to a baseline push-then-map approach. To implement the push-then-map approach, we also run an HDFS instance comprising the compute nodes (call this the "local" HDFS instance). We first run a Hadoop DistCP job to copy from the remote HDFS to this local HDFS, and then run a MapReduce job directly from the local HDFS. We compare application execution time using these two approaches. Because we are concerned primarily with push and map performance at this point, we run the Hadoop example WordCount job on text data generated by the Hadoop example randomtextwriter generator, as this represents a map-heavy application.

We run this experiment in two different environments: Amazon EC2 and PlanetLab. Our EC2 setup uses eight EC2 nodes in total, all of the m1.small instance type. These nodes are distributed evenly across two EC2 regions: four in the US and the other four in Europe. Each node hosts one map slot and one reduce slot. Two PlanetLab nodes, one in the US and one in Europe, serve as distributed data sources. Table I shows the bandwidths measured between the multiple nodes in this setup.

Figure 3(a) shows the execution time of the WordCount job on 2 GB of input data, and it shows that our approach to overlapping push and map reduces the total runtime of the push and map phases by 17.7%, and the total end-to-end runtime by 15.2% on our EC2 testbed.

Table II
MEASURED BANDWIDTHS IN THE PLANETLAB EXPERIMENTAL SETUP

From	To	Bandwidth (MB/s)
All sources	All workers	1-3
Workers A-C	Workers A-C	4-9
Workers A-C	Worker D	2
Worker D	Workers A-C	0.2-0.4

Next, we run the same experiment on PlanetLab. We continue to use two nodes as distributed data sources, and we use four other globally distributed nodes as compute nodes, each hosting one map slot and one reduce slot. Table II shows the bandwidths measured between the multiple nodes in this setup. Due to the smaller cluster size in this experiment, we use only 1 GB of text input data.

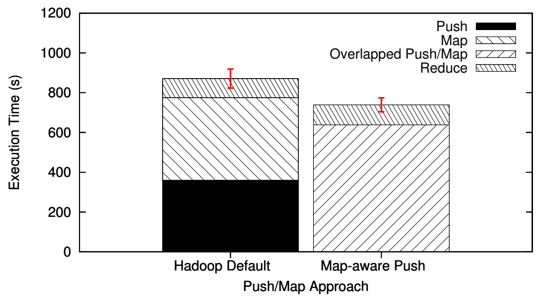
Figure 3(b) shows that push-map overlap can reduce runtime of the push and map phases by 21.3% and the whole job by 17.5% in this environment. We see a slightly greater benefit from push-map overlap on PlanetLab than on EC2 due to the increased heterogeneity of the PlanetLab environment.

IV. SHUFFLE-AWARE MAP

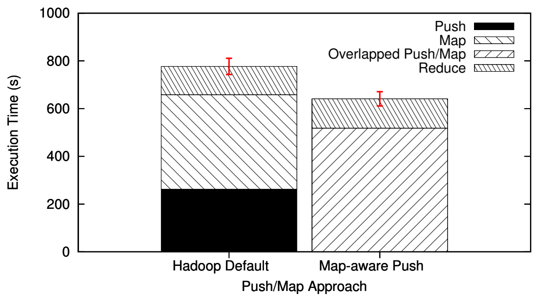
In the previous section, we showed how the map phase can influence the push phase, in terms of both the volume of data each mapper receives as well as the sources from which each mapper receives its data. In turn, the push determines, in part, when a map slot becomes available for a mapper. Thus, from the perspective of the push and map phases, a set of mappers and their data sources are decided. This decision, however, ignores the downstream cost of the shuffle and reduce as we will show. In this section, we show how the set of mappers can be adjusted to account for the downstream shuffle cost. This was also motivated in Section II as we illustrated the importance of shuffling and merging intermediate results close to mappers, particularly for shuffle-heavy jobs.

In traditional MapReduce, intermediate map outputs are shuffled to reducers in an all-to-all communication. In Hadoop, one can control the granularity of reduce tasks and the amount of work each reducer will obtain. However, these decisions ignore the possibility that a mapper-reducer link may be very poor. For example, in Figure 4, the links between mapper C and reducers D and E are poor, thus raising the cost of shuffle. For applications in which shuffle is dominant, this phenomenon can greatly impact performance, particularly in heterogeneous networks.

Two solutions are possible: changing the reducer nodes, or reducing the amount of work done by mapper C and in turn reducing the volume of data traversing the bottleneck links. We present an algorithm that takes the latter approach. In this way, the downstream shuffle (or reduce) can impact the map. This is similar to the *Map-aware Push* technique where the map influenced the push.



(a) WordCount on 2 GB text data on EC2



(b) WordCount on 1 GB text data on PlanetLab

Figure 3. Runtime of a Hadoop WordCount job on text data for the push-then-map approach and the *Map-aware Push* approach on globally distributed Amazon EC2 and PlanetLab test environments. Error bars reflect 95% confidence intervals.

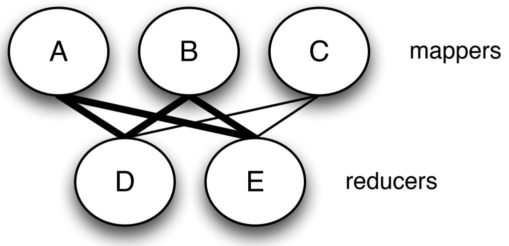


Figure 4. An example network where links from mapper C to reducers D and E are shuffle bottlenecks

As in typical MapReduce, we assume the reducer nodes are known *a priori*. We also assume that we know the approximate distribution of reducer tasks: i.e., we know the volume of intermediate data allocated to each reducer node. This allows us to know how much data must travel on the link from a mapper node to each reducer, which our algorithm utilizes. The distribution can be estimated using a combination of reducer node execution power and mapper-reduce link speed, pre-profiled. This estimate can be updated during the map phase if shuffle and reduce are overlapped.

A. *Shuffle-aware Map Scheduling*

To estimate the impact of a mapper node upon the reduce phase, we first estimate the time taken by the mapper to obtain a task, execute it, and deliver intermediate data to all reducers (assuming parallel transport). The intuition is that if the shuffle cost is high then the mapper node should be throttled to allow the map task to be allocated to a mapper with better shuffle performance. We estimate the finish time T_m for a mapper m to execute a map task as follows: T_m = T_m^{map} + $T_m^{shuffle}$, where T_m^{map} is the estimated time for the mapper m to execute the map task, including the time to read the task input from a source (using the *Map-aware Push* approach), and $T_m^{shuffle}$ is the estimated time to shuffle the accumulated intermediate data D_m up to the current task, from mapper m to all reducer nodes $r \in R$, where R is the set of all reducer nodes. Let $D_{m,r}$ be the portion of D_m destined for reducer r, and $L_{m,r}$ be the link speed between mapper node m and reducer node r. Then, we can compute

$$T_m^{shuffle} = \max_{r \in R} \left(\frac{D_{m,r}}{L_{m,r}} \right). \qquad (2)$$

The *Shuffle-aware Map* scheduling algorithm uses these T_m estimates to determine a set of *eligible mappers* to which to assign tasks. The intuition is to throttle those mappers that would have an adverse impact on the performance of the downstream reducers. The set of eligible mappers M_{Elig} is based on the most recent T_m values and a tolerance parameter α:

$$M_{Elig} = \{m \in M | T_m \leq \min_{m \in M} T_m + \alpha\}, \qquad (3)$$

where M is the set of all mapper nodes.

The intuition is that if the execution time for a mapper (including its shuffle time) is too high, then it should not be assigned more work at present. The value of the tolerance parameter α controls the aggressiveness of the algorithm in excluding slower mappers (in terms of their shuffle performance) from being assigned work. At one extreme, $\alpha = 0$ would enforce assigning work only to the mapper with the earliest estimated finish time, intuitively achieving good load balancing, but leaving all other mappers idle for long periods of time. At the other extreme, a high value of $\alpha > (\max_{m \in M} T_m - \min_{m \in M} T_m)$ would allow all mapper nodes to be eligible irrespective of their shuffle performance, and would thus reduce to the default MapReduce map scheduling. We select an intermediate value:

$$\alpha = \frac{(\max_{m \in M} T_m - \min_{m \in M} T_m)}{2}. \qquad (4)$$

The intuition behind this value is that it biases towards the upper half of mappers in terms of their shuffle performance. This is but one possible threshold; future research will explore other possibilities.

We note that the algorithm makes its decisions dynamically, so that over time, a mapper may become eligible or ineligible depending on the relation between its T_m value and the current value of $\min_{m \in M} T_m$. As a result, this algorithm allows a discarded mapper node to be re-included later should other nodes begin to offer worse performance. Similarly, a mapper may be throttled if its performance starts degrading over time.

B. Implementation in Hadoop

We have implemented this *Shuffle-aware Map* scheduling algorithm by modifying the task scheduler in Hadoop. The task scheduler now maintains a list of estimates T_m for all mapper nodes m, and updates these estimates as map tasks finish. It also uses the mapper-to-reducer node pair bandwidth information obtained by the network monitoring module to update the estimates of shuffle times from each mapper node. Every time a map task finishes, the task tracker on that node asks the task scheduler for a new map task. At that point, the scheduler uses (3) to determine the eligibility of the node to receive a new task. If the node is eligible, then it is assigned a task from the best source determined by the *Map-aware Push* algorithm described in Section III. On the other hand, if the node is not eligible, then it is not assigned a task. However, it can request for work again periodically by piggybacking on heartbeat messages, when its eligibility will be checked again.

C. Experimental Results

We now present some results that show the benefit of *Shuffle-aware Map*. Here we run our InvertedIndex application, which takes as input a set of eBooks from Project Gutenberg [5] and produces, for each word in its input, the complete list of positions where that word can be found. This application shuffles a large volume of intermediate data, so it is an interesting application for evaluating our *Shuffle-aware Map* scheduling technique.

First, we run this application on our EC2 multi-region cloud as described in Table I. In this environment, we use 1.8 GB of eBook data as input, and this produces about 4 GB of intermediate data to be shuffled to reducers. Figure 5(a) shows the runtime for a Hadoop baseline with push and map overlapped, as well as the runtime of our *Shuffle-aware Map* scheduling technique, also with push and map overlapped.

The reduce time shown includes shuffle cost. Note that in *Shuffle-aware Map* the shuffle and reduce time (labeled "reduce" in the figure) are smaller than in stock Hadoop. Also observe that in *Shuffle-aware Map* the map times go up slightly—this algorithm has decided to make this tradeoff resulting in overall better performance. On our wider-area PlanetLab setup (see Table II) we use 800 MB of eBook data and see a similar pattern, as Figure 5(b) shows. Again, an increase in map time is tolerated to reduce shuffle cost later on. This may mean that a slower mapper is given more work since it has faster links to downstream reducers. For this application, we see performance improvements of 6.8% and 9.6% on EC2 and PlanetLab, respectively.

V. Putting it all Together

To determine the complete end-to-end benefit of our proposed techniques, we run experiments comparing a traditional Hadoop baseline, which uses a push-then-map approach, to an alternative that uses our proposed *Map-aware Push* and *Shuffle-aware Map* techniques. Taken together, we will refer to our techniques as the *End-to-end* approach. We carry out these experiments on the same PlanetLab and EC2 test environments introduced in Table I and Table II, respectively. We focus here on the InvertedIndex application from Section IV as well as a new Sessionization application. This Sessionization application takes as input a set of Web server logs from the WorldCup98 trace [6], and sorts these records first by client and then by time. The sorted records for each client are then grouped into a set of "sessions" based on the gap between subsequent records. Both the InvertedIndex and Sessionization applications are relatively shuffle-heavy, representing a class of applications that can benefit from our *Shuffle-aware Map* technique.

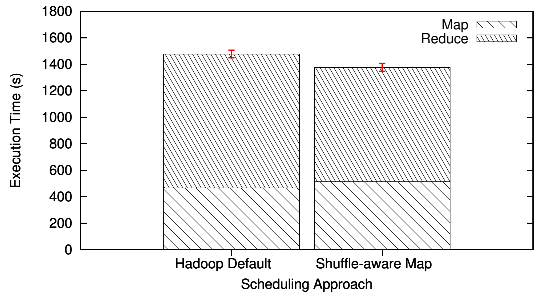
A. Amazon EC2

First, we explore the combined benefit of our techniques on our EC2 test environment (see Section III for details), comprising two distributed data sources and eight worker nodes spanning two EC2 regions. Figure 6(a) shows results for the InvertedIndex application, where we see that our approaches reduce the total execution time by about 9.7% over the traditional Hadoop approach. There is little difference in total push and map time, so most of this reduction in runtime comes from a faster shuffle and reduce (labeled "reduce" in the figure). This demonstrates the effectiveness of our *Shuffle-aware Map* scheduling approach, as well as the ability of our techniques to automatically determine how to tradeoff between faster push and map phases or faster shuffle and reduce phases.

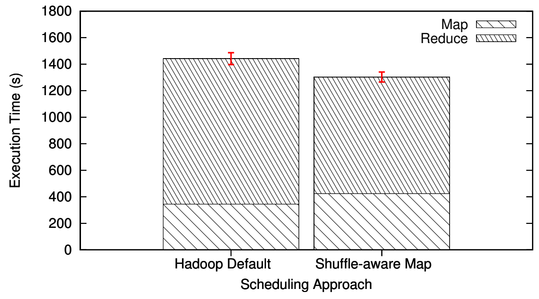
Now consider the Sessionization application, which has a slightly lighter shuffle and slightly heavier reduce than does the InvertedIndex application. Figure 6(b) shows that for this application on our EC2 environment, our approaches can reduce execution time by 8.8%. Again most of the reduction in execution time comes from more efficient shuffle and reduce phases. Because this application has a slightly lighter shuffle than does the InvertedIndex application, we would expect a slightly smaller performance improvement, and our experiments confirm this.

B. PlanetLab

Now we move to the PlanetLab environment, which exhibits more extreme heterogeneity than the EC2 environment. For this environment, we consider only the InvertedIndex application, and Figure 7 shows that our approaches

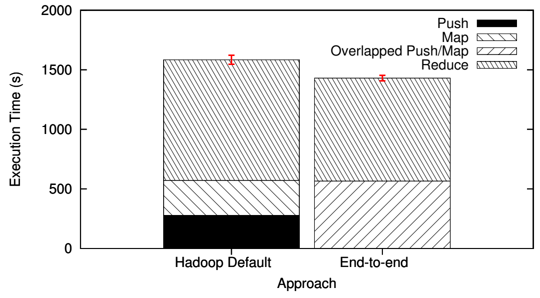


(a) InvertedIndex on 1.8 GB eBook data on EC2

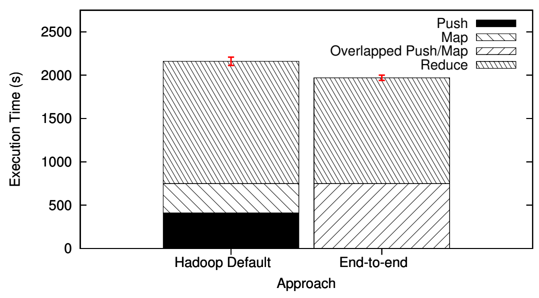


(b) InvertedIndex on 800 MB eBook data on PlanetLab

Figure 5. Runtime of the InvertedIndex job on eBook data for the default Hadoop scheduler and our *Shuffle-aware Map* scheduler. Both approaches use an overlapped push and map in these experiments. Error bars reflect 95% confidence intervals.



(a) InvertedIndex on 1.8 GB eBook data on EC2



(b) Sessionization on 2 GB text log data on EC2

Figure 6. Execution time for traditional Hadoop compared with our proposed *Map-aware Push* and *Shuffle-aware Map* techniques (together, *End-to-end*) for InvertedIndex and Sessionization applications on our EC2 test environment. Error bars reflect 95% confidence intervals.

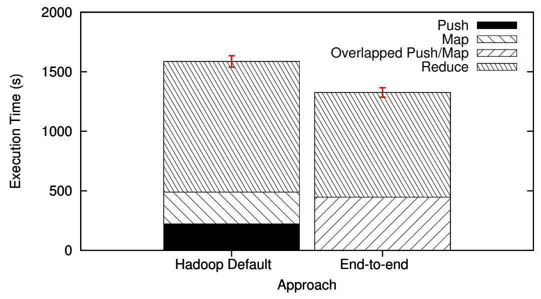


Figure 7. Execution time for traditional Hadoop compared with our proposed *Map-aware Push* and *Shuffle-aware Map* techniques (together, *End-to-end*) for for the InvertedIndex application with 800 MB eBook data on our PlanetLab test environment. Error bars reflect 95% confidence intervals.

can reduce execution time by about 16.4%. Although we see a slight improvement in total push and map time using our approach, we can again attribute the majority of the performance improvement to a more efficient shuffle and reduce.

To more deeply understand how our techniques achieve this improvement, we record the number of map tasks assigned to each mapper node, as shown in Table III. We see that both Hadoop and our techniques assign fewer map tasks to Mapper D, but that our techniques do so in a much more pronounced manner. Network bandwidth measurements revealed that this node has much slower outgoing network links than do the other mapper nodes; only about 200–400 KB/s compared to about 4–9 MB/s for the other nodes (see Table II). By scheduling three map tasks at that node, Hadoop has effectively "trapped" intermediate data there, resulting in a prolonged shuffle phase. Our *Shuffle-aware Map* technique, on the other hand, has the foresight to avoid this problem, and it does so by refusing to grant Mapper D additional tasks even when it becomes idle and requests more work.

Table III
NUMBER OF MAP TASKS ASSIGNED TO EACH MAPPER NODE IN OUR PLANETLAB TEST ENVIRONMENT

Scheduler	Mapper A	Mapper B	Mapper C	Mapper D
Hadoop Default	5	4	5	3
End-to-end	5	5	6	1

VI. Related Work

Traditionally, the MapReduce [1] programming paradigm assumes a tightly coupled homogeneous cluster applied to a uniform data-intensive application. Previous work has shown that if this assumption is relaxed, then performance suffers.

Zaharia et al. [7] showed that, under computational heterogeneity, the mechanisms built into Hadoop for identifying straggler tasks break down. Their LATE scheduler provides better techniques for identifying, prioritizing, and scheduling backup copies of slow tasks. In our work, we also assume that nodes can be heterogeneous since they belong to different data centers or locales. Chen et al. [8] report techniques for improving the accuracy of progress estimation for tasks in MapReduce. Ahmad et al. [9] demonstrate that despite straggler optimizations, the performance of MapReduce frameworks on clusters with computational heterogeneity remains poor as the load balancing used in MapReduce causes excessive and bursty network communication and the heterogeneity further amplifies the load imbalance of reducers. Their Tarazu system uses a communication-aware balancing mechanism and predictive load-balancing across reducers to address these problems. Mantri [10] explores various causes of outlier tasks in further depth, and develops cause- and resource-aware techniques to identify and act on outliers earlier, and to greater benefit, than in traditional MapReduce. Such improvements are complementary to our techniques. Previous work mainly focuses on computational heterogeneity within a single cluster. Our work, however, targets more loosely coupled and dispersed collections of resources with bandwidth heterogeneity and constraints.

Several works have targeted MapReduce deployments in loosely coupled environments. MOON [11] explored MapReduce performance in volatile, volunteer computing environments and extended Hadoop to improve performance under loosely coupled networks with unreliable slave nodes. In our work, we do not focus on solving reliability issues; instead we are concerned with performance issues of allocating compute resources to MapReduce jobs and relocating source data. Costa et al. [12] propose MapReduce in a global wide-area volunteer setting. However, this system is implemented in the BOINC framework with all input data held by the central scheduler. In our system, we have no such restrictions. Luo et al. [13] propose a multi-cluster MapReduce deployment, but they focus on more compute-intensive jobs that may require resources in multiple clusters for greater compute power. In contrast, we consider multi-site resources not only for their compute power, but also for their locality to data sources.

Other papers have addressed MapReduce data flow optimization and locality. Gadre et al. [14] optimize the reduce data placement according to map output locations, which might still end up trapping data in nodes with slow outward links. Kim et al. [15] present a similar idea of shuffle-aware scheduling, but do not consider widely distributed data sources that are not co-located with computation clusters. Their ICMR algorithm could make use of our mechanisms to improve the performance of MapReduce under such environments. Pipelining MapReduce has been proposed in MapReduce Online [16] to modify the Hadoop workflow for improved responsiveness and performance. It assumes, however, that input data are located with the computation resources, and it does not address the issue of pipelining push and map. MapReduce Online would be a complementary optimization to our techniques since it enables the shuffling of intermediate data without storing it to disk. Our mechanisms could be used to decide where data should flow and their technique could be used to optimize the transfer. Similarly, Verma et al. [17] discuss the specific challenges associated with pipelining the shuffle and reduce stages. Our *Map-aware Push* technique could also be applied to pipeline shuffle and reduce, though we have not yet done so. The Purlieus system [18] considers MapReduce in a single cloud, but is unique in that it focuses on locality in the shuffle phase. It emphasizes the coupling between the placement of tasks (in their case virtual machines) and data. However, these works do not provide an end-to-end overall improvement of the MapReduce data flow.

Other work has focused on fine-tuning MapReduce parameters or offering scheduling optimizations to provide better performance. Sandholm et al. [19] present a dynamic prioritization system for improved MapReduce runtime in the context of multiple jobs. Our work is concerned with optimizing a single job relative to data source and compute resource locations. Babu [20] proposes algorithms for automatically fine-tuning MapReduce parameters to optimize job performance. Starfish [21] proposes a self-tuning architecture which monitors runtime performance of Hadoop and tunes the configuration parameters accordingly. Such work is complementary to ours, however, as we focus on mechanisms to directly change task and data placement rather than tune configuration parameters.

Finally, work in wide-area data transfer and dissemination includes GridFTP [22] and BitTorrent [23]. GridFTP is a protocol for high-performance data transfer over high-bandwidth wide-area networks, and BitTorrent is a peer-to-peer file sharing protocol for wide-area distributed systems. These protocols could act as middleware services to further reduce data transfer costs and make wide-area data more accessible to wide-area compute resources.

VII. Conclusion and Future Work

Many emerging data-intensive applications are widely distributed, either due to the distribution and collection of datasets, or by the provision of multi-site resources such as multiple geographically separated data centers. We show that in such heterogeneous environments, MapReduce/Hadoop performance suffers as the impact of one phase upon another can severely impact performance along bottleneck links. We show that Hadoop's default data placement and scheduling mechanisms do not take such cross-phase interactions into account, and while they may try to optimize individual phases, they can result in globally bad decisions, resulting in poor overall performance. To overcome these limitations, we propose techniques to achieve cross-phase optimization in MapReduce. The key idea behind our proposed techniques is to consider not only the execution cost of an individual task or computational phase, but also its impact on the performance of subsequent phases. We propose two sets of techniques. *Map-aware Push* enables push-map overlap to hide latency and enable dynamic feedback between the map and push phases. Such feedback enables nodes with higher speeds and faster links to process more data at runtime. *Shuffle-aware Map* enables a shuffle-aware scheduler to feed back the cost of a downstream shuffle into the map process and affect the map phase. Mappers with poor outgoing links to reducers are throttled, eliminating the impact of mapper-reducer bottleneck links. For a range of heterogeneous environments (multi-region Amazon EC2 and PlanetLab) and diverse data-intensive applications (WordCount, InvertedIndex, and Sessionization) we show the performance potential of our techniques, as runtime is reduced by 7%–18% depending on the execution environment and application.

ACKNOWLEDGMENT

The authors would like to acknowledge NSF Grant IIS-0916425 and NSF Grant CNS-0643505, which supported this research.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of OSDI*, 2004, pp. 137–150.

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

[3] "Amazon EC2," http://aws.amazon.com/ec2.

[4] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman, "Exploring MapReduce efficiency with highly-distributed data," in *Proceedings of MapReduce*, 2011, pp. 27–33.

[5] "Free eBooks by Project Gutenberg," http://www.gutenberg.org/.

[6] M. Arlitt and T. Jin, "Workload characterization of the 1998 World Cup Web Site," HP Labs, Tech. Rep. HPL-1999-35R1, Sep. 1999.

[7] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proceedings of OSDI*, 2008, pp. 29–42.

[8] Q. Chen, D. Zhang, M. Guo, Q. Deng, S. Guo, and S. Guo, "SAMR: A self-adaptive MapReduce scheduling algorithm in heterogeneous environment." in *Proceedings of IEEE CIT*, 2010, pp. 2736–2743.

[9] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing MapReduce on heterogeneous clusters," in *Proceedings of ASPLOS*, 2012, pp. 61–74.

[10] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters," in *Proceedings of OSDI*, 2010, pp. 265–278.

[11] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "MOON: MapReduce on opportunistic environments," in *Proceedings of ACM HPDC*, 2010, pp. 95–106.

[12] F. Costa, L. Silva, and M. Dahlin, "Volunteer cloud computing: MapReduce over the internet," in *Proceedings of IEEE IPDPSW*, 2011, pp. 1855–1862.

[13] Y. Luo, Z. Guo, Y. Sun, B. Plale, J. Qiu, and W. W. Li, "A hierarchical framework for cross-domain MapReduce execution," in *Proceedings of ECMLS*, 2011, pp. 15–22.

[14] H. Gadre, I. Rodero, and M. Parashar, "Investigating MapReduce framework extensions for efficient processing of geographically scattered datasets," in *Proceedings of ACM SIGMETRICS*, 2011, pp. 116–118.

[15] S. Kim, J. Won, H. Han, H. Eom, and H. Y. Yeom, "Improving Hadoop performance in intercloud environments," in *Proceedings of ACM SIGMETRICS*, 2011, pp. 107–109.

[16] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proceedings of NSDI*, 2010, pp. 313–327.

[17] A. Verma, N. Zea, B. Cho, I. Gupta, and R. H. Campbell, "Breaking the MapReduce stage barrier," in *Proceedings of IEEE Cluster*, 2010, pp. 235–244.

[18] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: locality-aware resource allocation for MapReduce in a cloud," in *Proceedings of ACM SC*, 2011, pp. 58:1–58:11.

[19] T. Sandholm and K. Lai, "MapReduce optimization using dynamic regulated prioritization," in *Proceedings of ACM SIGMETRICS*, 2009, pp. 299–310.

[20] S. Babu, "Towards automatic optimization of MapReduce programs," in *Proceedings of ACM SoCC*, 2010, pp. 137–142.

[21] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *Proceedings of CIDR*, 2011, pp. 261–272.

[22] "GridFTP," http://globus.org/toolkit/docs/3.2/gridftp/.

[23] "BitTorrent," http://www.bittorrent.com.