

Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 12-003

Optimizing MapReduce for Highly Distributed Environments

Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman

February 13, 2012

Optimizing MapReduce for Highly Distributed Environments*

Benjamin Heintz
Department of Computer
Science & Engineering
University of Minnesota
Minneapolis, MN
heintz@cs.umn.edu

Abhishek Chandra
Department of Computer
Science & Engineering
University of Minnesota
Minneapolis, MN
chandra@cs.umn.edu

Ramesh K. Sitaraman
Department of Computer
Science
University of Massachusetts
Amherst, MA
ramesh@cs.umass.edu

ABSTRACT

MapReduce is a popular programming paradigm for large-scale data processing. MapReduce has traditionally been deployed over a tightly-coupled cluster or data center where the data is already locally available at the cluster. However, the assumption that the data and compute resources are available in single central location is no longer true for many emerging applications and environments. For many data-rich applications in commercial, scientific and social networking domains, the data is generated in a geographically distributed manner. Further, the computational resources needed for carrying out the data analysis itself may be distributed, across multiple data centers or community resources such as Grids. In this paper, we study the performance tradeoffs of executing a MapReduce application in a highly distributed environment, consisting of distributed data sources and computational resources. We take an analytic approach: we develop a model to capture MapReduce execution on the distributed platform, and formulate a constrained optimization problem to minimize the execution time. Our model is general enough to capture several design choices and performance optimization techniques for MapReduce execution. We propose and explore two classes of optimizations to the traditional MapReduce execution model: (i) *data placement* optimizations that optimize the allocation of data/compute resources, and (ii) *concurrency* optimizations that increase the concurrency of the execution. Our model results based on parameter estimation from PlanetLab measurements show that these optimizations are able to achieve up to 95% reduction in runtime over a baseline MapReduce model under a variety of network environments and application characteristics. Moreover, these results also provide several insights into design choices for improving distributed MapReduce performance based on platform and application characteristics.

1. INTRODUCTION

1.1 Motivation

MapReduce [9] has emerged as a popular programming paradigm for large-scale data processing. Its popularity has been spurred by the growing need for analysis and mining of large quantities of data being generated globally by increasing numbers of users, applications, sensors, and devices. MapReduce is widely used today for several data analysis applications, including Web indexing, log file analysis, and recommendation mining. Besides Google's MapReduce, several other implementations exist, such as the open-source versions Hadoop [13] and Disco [10], and MapReduce is also used

as a component in several other application frameworks and libraries such as Pig [19], Mahout [17] and CouchDB [8]. The emergence of publicly available cloud resources [1] has also resulted in the easy availability of the MapReduce programming framework to users.

MapReduce has traditionally been deployed over a tightly-coupled cluster or data center, the assumption being the data is already available at a centralized location, and is co-located with the computational resources. However, this assumption of centralized data and computation is not true for many emerging applications and environments. For many applications, the data is generated in a geographically distributed manner, so that the data sources are inherently distributed. For instance, a large number of enterprises including technology companies as well as traditional businesses like retail [20] generate data at multiple sites, including stores and warehouses situated in diverse locations. Further, large Internet-scale services such as Akamai [18] have highly distributed platforms of servers located in thousands of global locations with each location producing tens of terabytes of data each day that must be aggregated and analyzed. Similarly, many data-intensive scientific applications need to analyze data generated by distributed scientific sensors, instruments, and testbeds, while the data for several social networking applications is generated by millions of users spread around the world. For such applications, moving data to a central location for analysis can be extremely costly, both in time and money [3]. Besides the data sources being distributed, the computational resources needed for carrying out the data analysis itself may be distributed. Examples include multiple data centers that may be used by a large Internet company to analyze data local to these data centers, as well as community resources such as Grids used for scientific computation. Such widely dispersed data sources and compute resources limit the use and applicability of current MapReduce software frameworks despite its attractive features. The goal of our work is to propose and evaluate optimizations to the MapReduce software framework that significantly enhance its performance in a highly distributed setting.

Several techniques [24, 2, 7] have been proposed to improve the performance of MapReduce in local cluster environments. However, it is unclear which of these techniques (if any) are well-suited for executing MapReduce in highly distributed environments. Recent work [5] that explored deploying MapReduce in a highly distributed environment concluded that there is no single architecture or deployment strategy that works well for all possible application, data, network, and system characteristics. Thus, the tradeoffs of deploying and executing MapReduce in a highly distributed environment are not well understood. And, few guidelines exist on how to execute a MapReduce application in such an environment with high performance. The focus of our work is to provide such guide-

*This work was supported in part by NSF Grants CNS-0643505 and CNS-0519894.

lines through a modeling approach.

In this paper, we study performance tradeoffs of executing MapReduce applications in a highly distributed environment. We focus on MapReduce applications with distributed data sources that run on distributed network and computational resources. We take an analytic, model-driven approach where we first develop a model to capture a wide class of distributed MapReduce executions. We then use our model to propose and evaluate design choices and optimizations that can enhance MapReduce performance. The key objective of this paper is not to accurately model an existing MapReduce implementation, but rather to develop a principled approach for exploring design choices and optimizations that have the potential to enhance MapReduce performance in a highly distributed setting.

1.2 Research Contributions

Our paper makes the following research contributions:

- We develop a framework for modeling the performance of MapReduce in a highly distributed setting. Our modeling framework is flexible enough to capture a large number of design choices and optimizations. Our work provides a framework for answering “what-if” questions on the relative efficacy of various design alternatives. In particular, our model enables us to compare various architectural choices as well as to provide rules-of-thumb for efficient deployment based on network and application characteristics. Further, our model-driven optimizations are efficiently derivable since they use a Mixed Integer Program (MIP) formulation that has powerful solver libraries.
- We propose and explore two classes of optimizations to the traditional MapReduce execution model for better performance in a highly distributed environment: (i) *data placement* optimizations, that optimize the allocation of data/compute resources, and (ii) *concurrency* optimizations that increase the concurrency of the execution of multiple MapReduce phases. While a few similar optimization techniques have been proposed [2, 7] in the context of local cluster environments, to the best of our knowledge, our paper is the first one to analyze their impact on performance in a highly distributed setting.
- We use network and node measurements from the Planet-Lab [6] testbed to validate and parameterize our model, and evaluate our proposed optimizations. The results from our analysis show that for a highly distributed compute environment, the data placement and concurrency optimizations can provide nearly 90% and 68% reduction in execution time respectively, over a baseline MapReduce execution. Further, combining these optimizations together can result in even greater (about 95%) reduction in this environment.
- Our model-driven optimizations provide several insights into the choice of techniques and execution parameters based on application and platform characteristics. For instance, we find that an application’s data expansion factor (ratio of intermediate to input data) can influence optimal placement, ranging from a completely decentralized placement for high data aggregation, to a completely centralized placement for high data expansion. Our results also show that as the network becomes more distributed and heterogeneous, our optimizations provide higher benefits (95% for globally distributed sites vs. 51% for a single local cluster), as they can exploit heterogeneity opportunities better. Further, these benefits are obtained independent of the application characteristics.

The rest of the paper is organized as follows. We begin by pre-

senting our model for optimizing MapReduce performance in Section 2. We then propose optimizations to a baseline MapReduce model in Section 3 and analyze the benefits of these optimizations under a variety of model parameter settings in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. A MODEL FOR OPTIMIZING THE PERFORMANCE OF MAPREDUCE

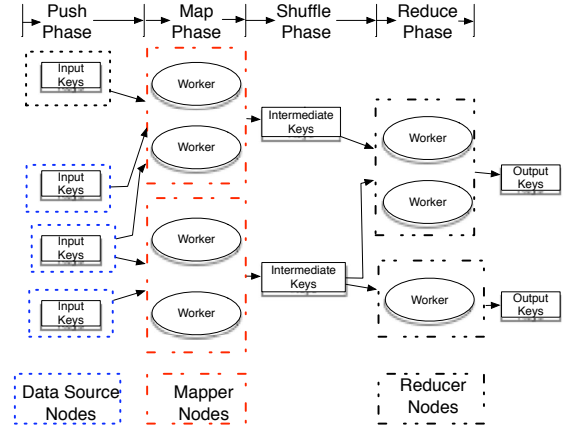


Figure 1: Executing a MapReduce application in a distributed environment. Each node represents a cluster of machines deployed in a data center. In a highly distributed and heterogeneous environment, the nodes have varying amounts of resources distributed widely across a wide-area network.

The MapReduce programming framework can be used to implement a number of commonly-used applications that take a set of *input* key/value pairs and produce a set of *output* key/value pairs [9]. A generic MapReduce application consists of a *map* function that processes input key/value pairs from the input data to generate a set of *intermediate* key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key to produce the output key/value pairs. An important capability of the MapReduce programming framework is that an application expressed in this framework can be executed on a distributed cluster of machines in a manner that is transparent to the application programmer. The execution is achieved by scheduling the map and reduce operations to server clusters as well as orchestrating the movement of data between these clusters. The manner in which computation and communication is scheduled is the key determinant of performance that we define to be the total time from start to finish of the execution of the MapReduce application. We refer to this total time as the *makespan*.

We illustrate the execution of a typical MapReduce application in a highly distributed environment in Figure 1. The input data originates at the *data source nodes* and is distributed to the *mapper nodes* in the *push phase*. In the *map phase*, each mapper node that consists of multiple worker threads running on multiple machines within a cluster performs the map operation on the incoming data and outputs intermediate key/value pairs. In the *shuffle phase*, the intermediate key/value pairs are partitioned and distributed to the *reducer nodes* such that *all* records that correspond to a given intermediate key are sent to same reducer node. This requirement preserves the semantics of a MapReduce application where *all* values of a specific intermediate key are required to perform the correct reduction. In the *reduce phase*, the reducer nodes process the intermediate key/value pairs and produce the required output.

The makespan of a MapReduce execution is a function of how the map and reduce operations are scheduled in their respective phases as well as the communication latencies incurred for data dissemination in the push and shuffle phases. The manner in which data and computation of a MapReduce application is scheduled on a distributed platform is captured by the notion of an *execution plan*. Intuitively, an execution plan determines how the data is distributed from the sources to the mappers and how the intermediate keys produced by the mappers are distributed across the reducers. Thus, the execution plan determines which clusters and which communication links are used and to what degree, that in turn is a major determinant of how long the MapReduce application takes to complete.

Given a highly distributed platform in the form of multiple machine clusters deployed in a wide-area network and given a MapReduce application, *our goal is to derive an execution plan that optimizes the makespan of executing the MapReduce application on the distributed platform*, i.e., we wish to derive an execution plan that minimizes the total time for the MapReduce application to complete. To this end, we model the execution of MapReduce and its makespan (Section 2.1). Next, we validate our model using empirical data (Section 2.2) and show how the model can be used for computing the execution plan with the optimal makespan (Section 2.3). Finally, we discuss our model in the context of current MapReduce software frameworks such as Hadoop (Section 2.4).

2.1 Modeling the makespan of a MapReduce execution

We successively model the distributed platform, the MapReduce application, valid execution plans, and their makespan.

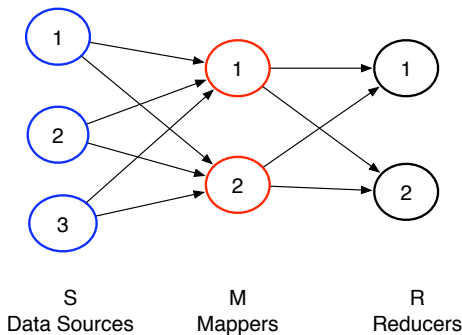


Figure 2: A tripartite graph model for distributed MapReduce with 3 data sources, 2 mappers and 2 reducers.

2.1.1 Modeling the distributed platform and the MapReduce application

We model the distributed platform available for executing the MapReduce application as a tripartite graph with a vertex set of $V = S \cup M \cup R$, where S is the set of data sources, M is the set of mappers, and R is the set of reducers. The edge set E of the tripartite graph is the complete set of edges and equals $(S \times M) \cup (M \times R)$ (See Figure 2). Each node corresponds to a cluster of servers that can potentially be used for executing the MapReduce application and the edges represent communication paths between clusters. The *capacity of a node* $i \in M \cup R$ is denoted by C_i (in bits/sec), where the capacity captures the computational resources available at that node in the units of bits of incoming data that it can process per unit time. Note that C_i is also application-dependent

as different MapReduce applications are likely to require different amounts of computing resources to process the same amount of data. Likewise, the *capacity of an edge* $(i, j) \in E$ is denoted by B_{ij} that represents the bandwidth (in bits/second) that can be sustained on the communication link that the edge represents.

Rather than model the MapReduce application in detail, we model two key parameters. First, we model the *amount of data* D_i (in bits) that originates at data source i , for each $i \in S$. Further, we model an *expansion factor* α that represents the ratio of the size of the output of a mapper to the size of its input. Note that α can take values smaller, greater, or equal to 1 depending on whether the output of the map operation is smaller, greater, or equal in size to the input. The value of α can be determined by profiling the MapReduce application. For instance, the map operation of the common MapReduce application that counts the frequency with which each URL is present in web server logs has α smaller than 1 since a server log with many fields including the URL is reduced to an intermediate key/value pair of the form $(URL, 1)$. On the other hand, a rendering application based on MapReduce that converts a set of input operations into corresponding images will have α greater than 1, since a small set of input parameters will be converted into an image of much larger size.

2.1.2 Modeling a valid execution plan and its makespan

An *execution plan* of the MapReduce application on the distributed platform is represented by variables x_{ij} , for $(i, j) \in E$, that represent the fraction of the outgoing data from node i that is sent to node j . An execution plan can be implemented by simply extending the MapReduce software framework as follows. Note that existing MapReduce frameworks such as Hadoop partition the input key space at the data sources and assigns each partition to a mapper. Likewise, it partitions the intermediate key space that are output by the mappers and assigns each partition to a reducer. As we mention later, this partition is usually achieved by a simple uniform hash function that divides the key space evenly into the required number of buckets. Generalizing this concept, we allow each data source and mapper to use their own hash function to partition the input and intermediate keys respectively, even perhaps in a non-uniform fashion. This enables the implementation of any execution plan x_{ij} , for $(i, j) \in E$, by simply providing a hash function h_i to each node $i \in S$ (resp. $i \in M$) such that a fraction x_{ij} of the input keys (resp., intermediate keys) are sent to mapper j (resp., reducer j).

Valid execution plans. We now mathematically express sufficient conditions for an execution plan to be valid and implementable in a MapReduce software framework while obeying the MapReduce application semantics.

$$\forall (i, j) \in E : 0 \leq x_{ij} \leq 1 \quad (1)$$

$$\forall i \in V : \sum_{(i, j) \in E} x_{ij} = 1 \quad (2)$$

Equations 1 and 2 simply express that for each i the x_{ij} 's are fractions that sum to 1. The semantics of a MapReduce application requires that each intermediate key is mapped to a single reducer. This is typically achieved by partitioning the intermediate key space among all the reducers and ensuring that each reducer gets all the keys in its assigned key space. This semantics can be implemented by ensuring that all reducers use the *same* hash function to map intermediate keys to reducers. Define variable y_k , $k \in R$, to be the fraction of the key space¹ mapped to reducer k .

¹We assume that the key space is large enough so that the variables y_k can be accurately approximated.

We enforce the one-reducer-per-key requirement with the following constraint.

$$\forall j \in M, k \in R : x_{jk} = y_k. \quad (3)$$

We define an execution plan to be *valid* if Equations 1, 2, and 3 hold.

Makespan of a valid execution plan. Given a valid execution plan for a MapReduce application, we now model the total time to completion, i.e., its makespan. To model the makespan, we successively model the time to completion of each of the four phases assuming that a *global barrier* exists after each phase. We assume that the data is available at all the data sources at time zero when the push phase begins. For each mapper $j \in M$, the time for the push phase to end is denoted by push_end_j that equals the time when all its data is received, i.e.,

$$\forall j \in M : \text{push_end}_j = \max_{i \in S} \frac{D_i x_{ij}}{B_{ij}} \quad (4)$$

Since we assume a global barrier after the push phase, all mappers must receive their data before the push phase ends and the map phase begins. Thus, the time when the map phase starts denoted by map_start obeys the following equation.

$$\text{map_start} = \max_{j \in M} \text{push_end}_j \quad (5)$$

The computation time at each mapper takes time proportional to the data pushed to that mapper. Thus, the time map_end_j for mapper $j \in M$ to complete its computation obeys the following.

$$\forall j \in M : \text{map_end}_j = \text{map_start} + \frac{\sum_{i \in S} D_i x_{ij}}{C_j} \quad (6)$$

As a result of the global barrier, the shuffle phase begins when all mappers have finished their respective computations. Thus, the time shuffle_start when the shuffle phase starts obeys the following.

$$\text{shuffle_start} = \max_{j \in M} \text{map_end}_j \quad (7)$$

The shuffle time for each reducer is governed by the slowest shuffle link into that reducer. Thus the time when shuffle ends at reducer $k \in R$ denoted by shuffle_end_k obeys the following.

$$\forall k \in R : \text{shuffle_end}_k = \text{shuffle_start} + \max_{j \in M} \frac{\alpha \sum_{i \in S} D_i x_{ij} x_{jk}}{B_{jk}} \quad (8)$$

Following the global barrier assumption, the reduce phase computation begins only after all reducers have received all of their data. Let reduce_start be the time when the reduce phase starts.

$$\text{reduce_start} = \max_{k \in R} \text{shuffle_end}_k \quad (9)$$

Reduce computation at a given node takes time proportional to the amount of data shuffled to that node. Thus, the time when reduce ends at node k denoted by reduce_end_k obeys the following.

$$\forall k \in R : \text{reduce_end}_k = \text{reduce_start} + \frac{\alpha \sum_{i \in S} \sum_{j \in M} D_i x_{ij} x_{jk}}{C_k} \quad (10)$$

Finally, it is clear that the makespan equals to the time at which the last reducer finishes. Thus,

$$\text{Makespan} = \max_{k \in R} \text{reduce_end}_k \quad (11)$$

Table 1: Measured bandwidth (KB/s) of the slowest/fastest links between clusters in each continent.

	US	EU	Asia
US	216 / 9405	110 / 2267	61 / 3305
EU	794 / 2734	4475 / 11053	1502 / 1593
Asia	401 / 3610	290 / 1071	23762 / 23875

2.2 Model estimation and validation

In this section, we show how the various parameters of our model can be estimated from actual measurements. Further, we validate our model by correlating the model predictions for makespan with the actual measured values. We use PlanetLab [6] to perform our experiments where nodes from PlanetLab are selected to serve as data sources, mappers, and reducers.

We estimate the model parameters for our distributed experimental platform as follows. For each $(i, j) \in E$, bandwidth B_{ij} is estimated by transferring data over that link and measuring the achieved bandwidth. For stable estimates, we used downloads of size at least 64 MB or a transfer time of at least 60 seconds, whichever is lesser. For each mapper or reducer node i , we derive the compute rate C_i by measuring the runtime of a simple computation over a list of data within a Python program, yielding a rate of computation in megabytes per second. On its own, however, this value is not directly useful. It is necessary to scale it down to a lower rate to reflect a more complex map or reduce operation, and the values used throughout this paper are scaled to correspond to the performance of a mapper node in a Hadoop wordcount job. This correspondence is not intended to be perfect, but was based on comparison of the Python-reported metric and actual Hadoop performance on the same hardware². Thus, the compute rate C_i of compute node i describes both a system parameter—namely physical compute capability—and an application parameter—the complexity of the map or reduce function itself. Finally, we consider various different possible values for the model parameter α that capture different MapReduce application characteristics in terms of data aggregation/expansion.

After the model parameters are estimated, we can derive the model's prediction for the makespan of any execution plan by simply using the model equations (Equations 1 to 11). To study the predictive power of this model, we correlate the model prediction for makespan with an actual measured value using the following approach. We use the model to predict the makespan of a execution plan. Then, we use a simple emulator that implements the same execution plan for the MapReduce application on the PlanetLab testbed. This emulator is implemented as a set of coordinating server processes written in Python that executes on the PlanetLab testbed and sends and receives data according to the given MapReduce execution plan.

This validation was carried out on a set of eight physical PlanetLab nodes distributed across eight sites, including four in the United States, two in Europe, and two in Asia. The unscaled C_i values on these nodes ranged from as low as 9 MB/s to as high as about 90 MB/s. Table 1 shows the intra-continental and inter-continental link bandwidths for these nodes and highlights the heterogeneity that characterizes such a highly distributed network.

The results of the validation are shown in Figure 3 where we observe a strong correlation (R^2 value of 0.9974) between predicted makespan and measured makespan. In addition, the linear fit to the

²We found that our estimates of the compute parameter C_i showed low variability and running the mapper and reducer computation on a small subset of the data was sufficient.

data points has a slope of 1.016, which shows there is also a strong correspondence between the absolute values of the predicted and measured makespans.

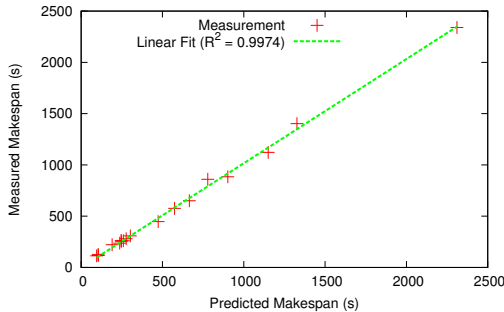


Figure 3: Measured makespan from a MapReduce data flow emulator correlates strongly with the model predicted value, for a cluster of eight globally-distributed PlanetLab nodes.

2.3 Finding the optimal execution plan

We formulate the problem of finding the execution plan that minimizes the makespan as an optimization problem. Viewing Equations 1 to 11 as constraints, we need to minimize a profit function that equals the variable Makespan. To perform this optimization efficiently, we rewrite the constraints in linear form to obtain a Mixed Integer Program (MIP). Writing it as MIP opens up the possibility of using powerful off-the-shelf solvers such as the Gurobi version 4.5.2 that we use to derive our results.

There are two types of nonlinearities that occur in our constraints that we need to convert to linear form. The first type is the existence of the max operator in Equations 4, 5, 7, 8, 9, and 11. Here we use a standard technique of converting an expression of the form $\max_i z_i = Z$ into an equivalent set of linear constraints $\forall i : z_i \leq Z$, where Z is minimized as a part of the overall optimization.

The second type of non-linearity are the quadratic terms present in Equations 8 and 10. We remove this type of nonlinearity with a two step process. First, we express the product terms of the form yz that involve two different variables in separable form. Specifically, we introduce two new variables $w = \frac{1}{2}(y + z)$ and $w' = \frac{1}{2}(y - z)$. This allows us to replace each occurrence of yz with $w^2 - w'^2$ which is in separable form since it does not involve a product of two different variables. Next, we approximate the quadratic terms w^2 and $-w'^2$ with a piecewise linear approximation of the respective function. The more piecewise segments we use the more accuracy we get, but require a larger number of choice variables resulting in a longer time to solve. As a compromise, we choose about 10 evenly spaced points on the curve leading to an approximation with about 9 line segments resulting in a worst-case deviation of 4.15% between the piece-wise linear approximation and the actual function. Since w^2 is a convex function, its piece-wise linear approximation can be expressed using linear constraints with no integral variables. However, since $-w'^2$ is not a convex function, its piecewise linear approximation requires us to utilize new binary integral variables to choose the appropriate line segment, resulting in a MIP (Mixed Integer Program) instead of an LP (Linear Program). For a more detailed treatment of the techniques that we have used to remove the two types of nonlinearities, the reader is referred to [23].

2.4 A baseline model for current MapReduce software frameworks

A key goal of the model presented earlier in this section is to be general enough to capture the valid executions of a MapReduce application, *without* being restricted to any specific software framework that supports MapReduce programming. This enables our model to extend beyond the limitations of currently available software frameworks and provide insights and predictions for features and optimizations that do not exist in current systems. The two commonly-used software frameworks to execute MapReduce computations are Google’s implementation [9] and Hadoop [13] that each use an underlying distributed file system GoogleFS [12] and HDFS [22] respectively. In such frameworks, in the push phase, the input data is split into equal-size blocks and stored in the distributed file system throughout the cluster. Subsequently, a cluster master schedules the map tasks that process those blocks to create intermediate key/value pairs. In the shuffle phase, the intermediate data is then copied to other worker nodes that execute the reduce operations in the reduce phase. In either software framework, much of the prior work assumes that the entire MapReduce computation is performed within a single cluster where data dissemination in the push and shuffle phases tend to be efficient and hence need not be explicitly optimized. However, in a highly distributed environment, such as one where both data sources and the available computing resources are spread globally, the push and shuffle phases can have a large influence on the performance of MapReduce and can no longer be ignored. A distinctive feature of our modeling approach is that it has sufficient generality to capture the communication latencies of data dissemination of the underlying distributed platform.

In addition, current software frameworks like Hadoop implicitly assume that the nodes are largely homogeneous and that the sources uniformly distribute their data to the mappers, and the mappers in turn uniformly distribute their data to the reducers. However, in a large, highly distributed computational environment, both nodes and link speeds are likely to be highly heterogeneous. In fact, it has been shown that standard Hadoop scheduler doesn’t perform well in the presence of heterogeneity [24]. Even though our model can be specialized to uniform splits typical of Hadoop-like implementations as shown below, it is general enough to capture uneven splits to account for heterogeneity of the underlying distributed platform. We also note that most current implementations employ online techniques such as data and task replication as well as speculative execution to achieve load balancing and fault tolerance. Such techniques are orthogonal to our model output which provides an offline execution plan, and are beyond the scope of this paper.

The baseline model. Much of the optimizations we propose in future sections are compared with a baseline model that captures the performance of current MapReduce frameworks such as Hadoop. The two additional characteristics that we capture in the baseline model are the uniform push and uniform shuffle conditions that we express below.

$$\text{Uniform Push : } \forall i \in S, j \in M : x_{ij} = \frac{1}{|M|} \quad (12)$$

$$\text{Uniform Shuffle : } \forall j \in M, k \in R : x_{jk} = \frac{1}{|R|} \quad (13)$$

Finding the optimal execution plan in the baseline model involves optimizing Makespan while satisfying the two additional constraints above, i.e., we require that all Equations 1 to 13 are satisfied.

3. DATA PLACEMENT AND CONCURRENCY OPTIMIZATIONS FOR MAPREDUCE

We explore a number of optimizations to the baseline model presented in Section 2.4. Recall that the more restrictive baseline

MapReduce model corresponds to current implementations (such as Hadoop). The optimizations presented in this section enable improved data placement and task scheduling decisions in order to improve the makespan of a MapReduce application execution in a highly distributed environment. We divide these optimizations into two categories:

- *Data placement optimizations:* These optimizations improve makespan through resource allocation decisions of data placement on the platform nodes based on information about the network and compute node characteristics, such as link bandwidths and compute capabilities. Note that the location of compute tasks (map and reduce) is completely dependent on the location of the data, and so compute task placement is implicit in the data placement decisions. An example of such an optimization is push optimization, that determines how much data should be pushed from each data source to each mapper based on the source-mapper link bandwidths as well as mapper compute speeds, in order to minimize the data push time and map computation time.
- *Concurrency optimizations:* These optimizations hide the latency of execution of different phases by increasing their concurrency and overlapping the execution of computation and communication phases, while satisfying the data dependencies between different phases. An example of such an optimization is pipelining of data between mappers and reducers in order to overlap their execution to reduce the makespan.

Many of these optimizations (though not all) are orthogonal and can be applied independently or in combination with each other. In addition, as we will show, these optimizations also illustrate the generality of our model framework—each of them requires minimal changes to the model, in the form of addition, removal, or changes to only a few constraints. We begin by presenting the data placement optimizations, followed by the concurrency optimizations. For each optimization, we will first describe the changes made to the baseline MapReduce model of Section 2.4 to realize that optimization, followed by results showing the performance impact of applying the optimization to a distributed MapReduce application execution.

3.1 Data placement optimizations

3.1.1 Push optimization

If MapReduce is executed on a single homogeneous cluster with high-speed local links, it is reasonable to partition the input data in a uniform fashion among the mappers. The baseline model captures such an implementation by using the uniform push constraint (Equation 12). However, a highly distributed environment is inherently heterogeneous with widely varying node speeds and link capacities, so such a uniform partitioning of data may not be optimal.

The push optimization incorporates a knowledge of network and node characteristics to improve the input data placement and map task scheduling *proactively*. The intuition behind this optimization is that which mappers a source chooses to push its data to depends on both the proximity of the mappers to the source as well as the relative compute capacity of these mappers. Thus, for instance, a source may decide to push its data to a nearby mapper node in order to reduce the data transmission overhead. But, it may also choose to push more data to a faster mapper that may be located farther away in order to reduce the compute overhead. Also, the relative importance of the network speeds and compute capacities

Table 2: Measured bandwidth (KB/s) of the slowest/fastest links between distinct machines in the two-site setup.

	Texas	Japan
Texas	19125 / 52898	326 / 2644
Japan	544 / 2222	11552 / 11633

may be determined by the application characteristics: whether it is data-heavy or compute-heavy. Moreover, as we will show later, the data push decisions will also depend on the relative size of output of the mappers to its input as captured by the expansion factor α . Finally, it is worth noting that optimizing the push can also have the impact of reducing the time taken for a latter phase such as the shuffle phase.

To allow the model to incorporate push optimization, we need only remove the uniform push constraint (Equation 12) from the baseline MapReduce model. That is, we minimize variable Makespan with Equations 1 to 11, and Equation 13 acting as constraints. Removing the uniform push constraint (Equation 12) automatically allows our optimization framework to search for the optimal way to push the data, perhaps in a non-uniform way, to minimize the makespan.

3.1.2 Shuffle optimization

The second data dissemination phase in the MapReduce execution is the shuffle phase, where data is sent from the mappers to the reducers. Similar to push optimization, we propose a shuffle optimization, which allows each reducer to receive a *different and potentially unequal fraction of intermediate keys* from the mappers. The optimal data distribution among the reducers will depend on the relative mapper-reducer link speeds and computation capacities of reducer nodes. To allow this optimization, we only need to remove the uniform shuffle constraint (Equation 13) from the baseline model.

3.1.3 Benefits from data placement optimizations

We now show the benefits of push and shuffle optimizations for the makespan of a MapReduce application in a highly distributed environment. We show the benefit of applying each optimization individually as well as in combination. We also show the impact of the expansion factor α on the relative benefit achieved. For our results in this section, we use a two-site wide-area data center network setup, consisting of four physical PlanetLab machines: two in Texas and two in Japan. Each of these machines hosts one mapper and one reducer, and each site hosts one data source. We estimate the parameters of our model using measurements from this distributed network setup. Values for C_i vary closely around 4.5 MB/s for the machines in Texas, and around 8.5 MB/s for the machines in Japan. Bandwidth measurements between machines are shown in Table 2. We then find the optimal makespan using our model for each of the optimizations that we consider. Note that we select a simple two-site setup in this section to better illustrate the insights from our optimizations, and consider a variety of other network environments in the next section.

Figure 4 shows the makespan achieved in four different cases: (a) for the baseline model; (b) for the baseline model with push optimization; (c) baseline model with shuffle optimization; and (d) the baseline model with both push and shuffle optimizations applied simultaneously. Further, we also consider different values for expansion factor $\alpha=0.01$ (high aggregation), $\alpha=1$ (no aggregation/expansion), and $\alpha=100$ (high expansion). Each bar in the graph is further broken up into the runtimes taken by each phase

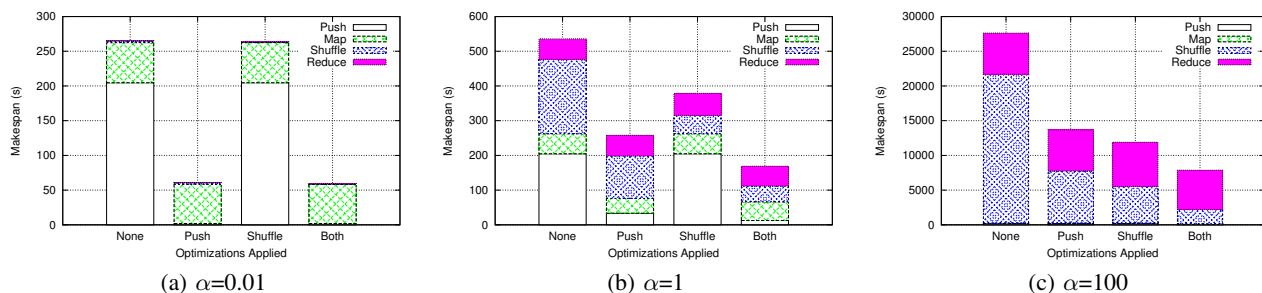


Figure 4: Benefit of data placement optimizations for MapReduce makespan.

(push, map, shuffle, and reduce). Note that smaller the makespan depicted in this graph the better.

From our results in Figure 4, we conclude the following. Note that the first bar labeled “None” represents the unoptimized baseline implementation that we use to compare all optimizations. First, we observe that for each α value, optimizing the bottleneck phase alone (push or shuffle) has higher benefit compared to optimizing the non-bottleneck phase alone. For instance, for $\alpha = 0.01$, the push and map phases dominate the makespan in the baseline (being about 77% and 22% of the total runtime) and push optimization alone is able to reduce the makespan by 77% by lowering the runtime of these two phases. On the other hand, for $\alpha = 100$, the shuffle and reduce phases are dominant (78% and 21% of total runtime) and optimizing these phases via the shuffle optimization brings the makespan down by 57%. However, an additional interesting observation from Figures 4(b) and (c) is that optimizing earlier phases can have a beneficial impact on the performance of the later phases. In particular, for $\alpha = 100$, push optimization also brings down the shuffle overhead (by 65%), even though the push and map phases themselves have minimal contribution to the makespan. This is because the location of the mappers where data is pushed to has an impact on how data is shuffled to reducers. Since the model optimizes the entire makespan, even if the shuffle phase is the bottleneck, there can still be a significant benefit to optimizing the push phase alone. Finally, we see that in each case, combining both the push and shuffle optimizations results in significant reduction in total makespan (78%, 68%, and 72% for $\alpha=0.01$, 1, and 100 respectively) as compared to the baseline, irrespective of where the bottleneck lies. This shows that a combination of the data placement optimizations is able to automatically optimize the whole execution independent of the application characteristics.

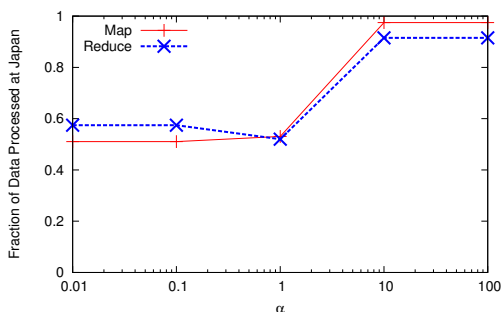


Figure 5: Fraction of data processed at Japan site as a function of α . For small α , data is distributed roughly evenly between the two locations, while for large α , data is pushed to Japan and remains there for the map and reduce computation.

Next, we explore what kind of data placements might result from these optimizations, as the value of α is varied. These results will provide an insight into architectural choices for deployment of a MapReduce application based on its α value. Figure 5 plots the fraction of total data allocated to the Japan site as the value of α is varied. The two curves in the graph correspond to the partitioning of the input data (Map) and the intermediate data (Reduce). The higher the fraction, the more data is placed and processed at the Japan site. The figure shows that for low values of α ($=0.01, 0.1$), both the input and intermediate data is nearly evenly split between the two sites. The reason is that the push phase is the bottleneck for low α , and hence, it is beneficial to push most of the input data from the data sources to local nodes and carry out the map computation locally, thereby avoiding the slow transcontinental links. Here, the shuffle phase may still pass data across slow links, but since the amount of data being disseminated in this phase is much smaller, this overhead will be comparatively small (see Figure 4(a) for relative values). On the other hand, for larger values of α ($=10, 100$), the figure shows that most of the data is now pushed to the Japan site. Now, the critical bottleneck is the shuffle phase, and hence, the optimal placement avoids sending intermediate data across slow links, while the much smaller amount of input data can be pushed across wide-area links. The data is being pushed to the Japan site as opposed to the Texas site in this case, because, as noted earlier, the Japan nodes are faster than the Texas nodes. The data is evenly split for $\alpha=1$, because in this case, neither phase is a bottleneck and the data could be pushed/shuffled anywhere while still paying about the same cost.

An interesting observation for these placement results is that they correspond to a decentralized, local computation for low α and a centralized computation for high α . In other words, if the application is likely to aggregate input data aggressively, it is beneficial to carry out the computation locally, while if the input data will be expanded significantly, it is better to push all data to a central location and carry out the computation there. Note that in the latter case, centralized computation may be optimal, even though we are utilizing *fewer* compute resources overall.

Overall, our results show that:

- Optimizing the bottleneck phase has the highest benefit when applying individual optimization, though push optimization can also improve the performance of later phases (such as shuffle and reduce).
- Combining both the push and shuffle optimizations can reduce the overall makespan of the MapReduce application irrespective of the application characteristics, by automatically reducing the overhead of the bottleneck phase.
- The data expansion factor α can influence the optimal data

placement and task scheduling strategies. We also demonstrated conditions where for a low expansion factor the optimal solution employed a completely distributed configuration, and a centralized configuration for a high expansion factor.

3.2 Concurrency optimizations

We present a set of concurrency optimizations that enhances the baseline MapReduce model. These optimizations focus on extracting a higher degree of concurrency across the push, map, shuffle, and reduce phases while maintaining all required data dependencies (e.g., a mapper cannot start executing unless it has some data available). Recall that in the baseline MapReduce model, we assumed the presence of a *global barrier* after each of the four phase. That is, all nodes must complete any given phase before any node can proceed to next one. For example, all mappers must wait for *all* the data to be pushed from all the sources before any of them can start the computation of the map phase. While the global barrier satisfies data dependencies across the different phases, it has no concurrency in terms of the execution of the phases.

We consider two concurrency optimizations by relaxing the global barrier requirements. Both these optimizations improve the concurrency of MapReduce execution to varying degrees. The first optimization utilizes a *local barrier* instead of a global one. In the local barrier optimization, the global barrier for synchronization after each phase is removed, but a synchronization barrier local to each node is introduced. With a local barrier, a node can start the map or reduce phase as long as it has *all* of its input data and a node can start the push and shuffle phases as long as it has *all* the data that it needs to disseminate. In particular, a node need not wait for other nodes to complete the current phase before proceeding to the next phase. This allows a degree of concurrency between the execution of the different phases, allowing the makespan to be reduced.

The second optimization that we propose utilizes *pipelining* to concurrently execute the different phases. Pipelining allows a node to start the map or reduce phase as long as the *first piece* of its input data is available for processing, i.e., the node need not wait for all of its inputs to be present but can start as soon as the first piece is received. Likewise, a node can start the push and shuffle phases as soon as the *first piece* of the data hat needs to be disseminated is available. It is easy to see that pipelining allows for even more concurrency than local barriers. For instance, a mapper can start execution as soon as it receives the first piece of data from a data source, its output can be immediately shuffled to the appropriate reducer, and the reducer can immediately start processing the result³.

3.2.1 Models for local barrier and pipelining

The baseline model formulation in Section 2.4 assumed the presence of a global barrier between each stage. We can relax this assumption, and instead use either local barriers, or a pipelining optimization with simple modifications to the baseline model. First, we replace the constraints governing the start time for the last three phase as expressed in Equations 5, 7 and 9 with the following new constraints. These new constraints capture the fact that a node can start its next phase without waiting for all nodes to complete the

³Note that for some applications, a reducer may have to wait until it gets a certain proportion (or all) of its intermediate data before it can start execution. However, a large number of MapReduce applications allow for incremental processing of the map and reduce operations, allowing for the pipelining optimization outlined here.

previous phase.

$$\begin{aligned} \forall j \in M : \text{map_start}_j &= \text{push_end}_j \\ \forall j \in M : \text{shuffle_start}_j &= \text{map_end}_j \\ \forall k \in R : \text{reduce_start}_k &= \text{shuffle_end}_k \end{aligned}$$

Now we can generalize the definitions for the ending times of these stages by first defining a combination operator \oplus for each optimization as follows:

$$a \oplus b = \begin{cases} a + b, & \text{if local barrier} \\ \max(a, b), & \text{if pipelined} \end{cases}$$

Then, we replace the definition for the ending time of the map phase in Equation 6 with the following new constraint.

$$\forall j \in M : \text{map_end}_j = \text{map_start}_j \oplus \frac{\sum_{i \in S} D_i x_{ij}}{C_j}. \quad (14)$$

Intuitively, the above constraint specifies that the runtime for the map phase on a node depends on the start-time of the map phase on that node and the time for map computation on all of the data pushed to that node. For the local barrier optimization, this equals the sum of the two times (corresponding to the time to push the data and compute on the data in sequence). On the other hand, for the pipelining optimization, this equals the maximum of the two times, since the map phase at a node cannot end until all of its data has arrived and all of its computation has finished, and the slower of these two operations will dictate the completion time. Note that we are assuming that the data push and map computation are completely overlapped. This assumption is valid if the total quantity of data is much larger than individual record size, which is the case for most typical MapReduce applications.

Based on similar intuition, the constraint for the shuffle stage in Equation 8 is replaced with

$$\forall k \in R : \text{shuffle_end}_k = \max_{j \in M} \left\{ \text{shuffle_start}_j \oplus \frac{\alpha \sum_{i \in S} D_i x_{ij} x_{jk}}{B_{jk}} \right\}. \quad (15)$$

Finally, the constraint for the end of the reduce stage in Equation 10 is similarly replaced with

$$\forall k \in R : \text{reduce_end}_k = \text{reduce_start}_k \oplus \frac{\alpha \sum_{i \in S} \sum_{j \in M} D_i x_{ij} x_{jk}}{C_k}. \quad (16)$$

3.2.2 Benefits from concurrency optimizations

We now show the benefits of local barrier and pipelining optimizations on the makespan of a MapReduce application in a highly distributed environment. Note that these two optimizations cannot be applied together to the same phase. It is theoretically possible to apply one optimization to one phase and the other optimization to a different phase. However, for the ease of exposition, we only apply each optimization to all phases and do not consider combining them in any way. Further, we do not incorporate the data placement optimizations in the following results, as we want to measure the benefit of applying the concurrency optimizations alone without other changes. As discussed earlier, we use a 2-site wide-area PlanetLab setup consisting of 2 compute nodes and 1 data source each in Texas and Japan for constructing our model.

Figure 6 compares the makespan of the baseline MapReduce with that obtained by applying either the local barrier or the pipelining optimization to all phases of the execution, for α values of 0.01, 1, and 100 respectively. We show a phase-wise breakdown only for

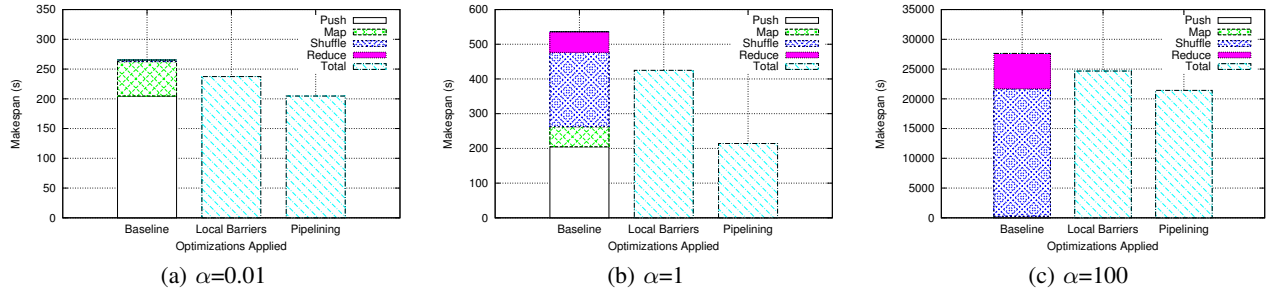


Figure 6: Benefit of concurrency optimizations for MapReduce makespan.

the baseline, since it does not make sense in the other cases because of the concurrent execution of these phases. The phase-wise breakdown of the baseline runtime also provides insights into the potential improvement that can be achieved by these optimization.

We draw the following conclusions from Figure 6. While local barriers provide 11-26% reduction in makespan as compared with the baseline, pipelining is able to achieve much higher makespan reduction in the range 22-60%. This is expected since in the case of pipelining a phase can begin as soon as it has the first piece of data and does not have to wait for all of the data. This leads to greater concurrency of the phases and a larger reduction in the makespan. Further, we observe that while the local barrier improvement is nearly uniform for different values of α , pipelining achieves a much higher reduction in makespan for $\alpha = 1$ compared to values of 0.01 and 100. This is because, for small (resp., large) values of α , one phase, namely the push (resp., shuffle) phase, is the bottleneck phase. Therefore, the time to complete the bottleneck phase dominates that of the other phases even when their executions are overlapped in a concurrent fashion. Thus, there is less opportunity of latency hiding in this case. On the other hand, for $\alpha=1$, the different phases are comparable in their execution time, so overlapping their execution provides a larger percentage reduction of the total makespan, since there is much larger opportunity to hide the latency of one phase with another. Further, notice that for all values of α , pipelining is able to extract almost all of the makespan reduction possible, since in each case the optimal makespan with pipelining is comparable to the time for the single bottleneck phase in the baseline solution. This phenomenon is not seen for local barrier, since phases can only be partially overlapped.

Overall, our results show that:

- Pipelining achieves much higher reduction in makespan compared to local barrier optimization.
- The benefit from pipelining is highest when the different phases are well-balanced in their execution time, rather than when one phase is dominant, since there is more opportunity for latency hiding in the former case.

4. EVALUATION OF THE OPTIMIZATIONS FOR DIFFERENT NETWORK AND COMPUTE ENVIRONMENTS

In this section, we evaluate the benefits of data placement and concurrency optimizations on a variety of distributed platforms with varying network and compute environments. Based on results from the previous section, we chose optimizations that had the most impact on the makespan. In particular, we apply the push, shuffle, and pipelining optimizations to obtain optimized makespans for a vari-

ety of network and compute environments. We specifically explore the variation of two sets of parameters:

- *Network environment*: This corresponds to the physical location and distribution of compute nodes and network links, and allows us to compare environments with different degrees and scale of distribution, ranging from a local cluster to a globally distributed compute platform.
- *Compute rates*: Varying the compute rate C_i parameter value corresponds to varying the relative computation-to-communication requirements of the application, and can be used to represent a compute-heavy vs. a communication-heavy application running on the platform.

To measure the benefit provided by the optimizations, we use the *relative reduction in makespan* metric:

$$\frac{Makespan_{base} - Makespan_{opt}}{Makespan_{base}}$$

where $Makespan_{base}$ and $Makespan_{opt}$ are the makespan values achieved for the baseline MapReduce model and the optimized MapReduce model respectively.

4.1 Network environments

To evaluate the effect of distribution of network resources, we evaluate the makespan of a MapReduce application both in the baseline model and in our optimized model that implements push, shuffle, and pipelining optimizations. We consider a variety of distributed environments with a range of network characteristics. These environments are based on compute speed and link bandwidth measurements for PlanetLab nodes distributed around the world, including four in the US, two in Europe, and two in Japan with compute rates and interconnection bandwidths as described in Section 2.2. Using these measurements, we generate the following specific network environments:

- *Local data center*: This setup consists of one local cluster with eight nodes of each type (source, mapper, reducer), and corresponds to the traditional local MapReduce execution scenario. This cluster is based purely on nodes in the US, specifically at tamu.edu.
- *Intra-continental data centers*: This setup consists of two data centers located within a continent - all nodes are in the US at tamu.edu and ucsb.edu. This setup corresponds to a more distributed topology than the local cluster scenario.
- *Globally distributed data centers*: In this setup, nodes span the entire globe (California, Texas, Germany, Japan), introducing much greater heterogeneity and wide-area network

bandwidths and latencies. We considered two different global environments: one with four data centers (at ucsb.edu in California, tamu.edu in Texas, tkn.tu-berlin.de in Germany, and pnl.nitech.ac.jp in Japan), and another with eight data centers (same as earlier plus hpl.hp.com in California, uiuc.edu in Illinois, Essex.ac.uk in the UK, and wide.ad.jp in Japan). This allows us to compare the impact of scaling up the number of locations.

For each of the above setups, the total number of nodes is held constant at eight. In some cases, where we did not have sufficient nodes to meet this requirement (e.g., for local data center setup), we added replica nodes to the setup with the measured node/link characteristics of the corresponding real nodes. In addition, we held the number of data sources fixed, allocating these data sources to clusters in the same proportion as mappers and reducers. The total amount of input data per data source was held constant throughout.

4.1.1 Impact of data placement optimizations

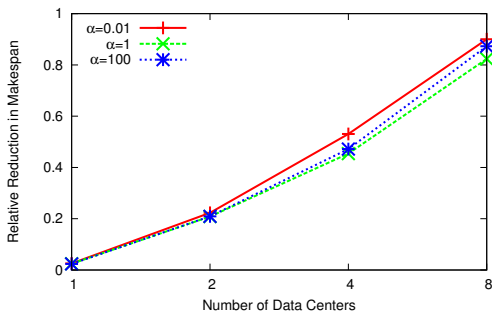


Figure 7: Impact of data placement optimizations for different network environments (higher is better).

Figure 7 shows the impact of combining the data placement optimizations (push and shuffle) on the MapReduce makespan for the different network environments. The first observation from this figure is that as the environment becomes more highly-distributed (going from local to intra-continental to global), we see higher benefit of the data placement optimizations (going from less than 3% for local to as high as 90% for the 8-cluster setup). The reason behind this result is that as the environment becomes more highly distributed, node and link speeds exhibit greater heterogeneity and our data placement optimizations have more opportunity to better balance the workload (e.g., by moving more data over faster links, and executing more tasks on faster nodes). We also note that the results do not vary much by the value of α , since combining both the data placement optimizations results in optimizing across all phases, independent of which phase is the bottleneck.

4.1.2 Impact of concurrency optimizations

Figure 8 shows the impact of the pipelining optimization on the makespan for the different network environments. We make two main observations. First, the benefit of pipelining is much more pronounced for $\alpha=1$ compared to other values. As discussed in Section 3.2, this is because there is more benefit to overlapping multiple phases, when they are comparable in their runtime. Secondly, we also observe that pipelining gives more benefit for more highly-distributed environments: the global data centers vs. the local and intra-continental environments. The reason is that as the environment becomes more highly-distributed, the cost of communication increases, and brings the time spent in communication and

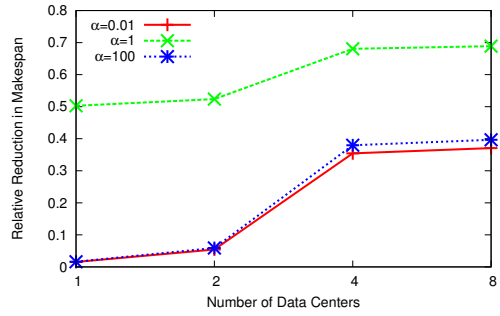


Figure 8: Impact of concurrency optimizations for different network environments.

computation closer together, with the fraction of makespan spent in communication going from < 1% for 1 data center to 39% for the 8-data center case. Thus, once again, pipelining has more opportunity to hide latency across different stages, compared to the local cluster case which is dominated by the computational phases.

4.1.3 Combining data placement and concurrency optimizations

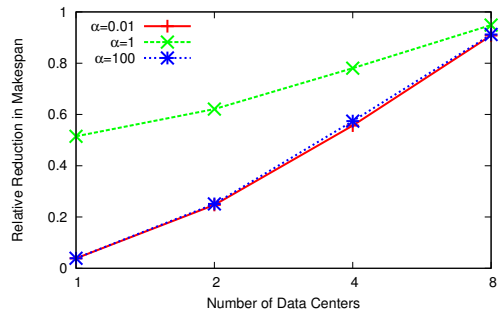
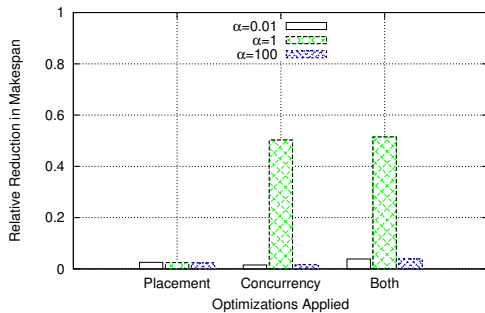
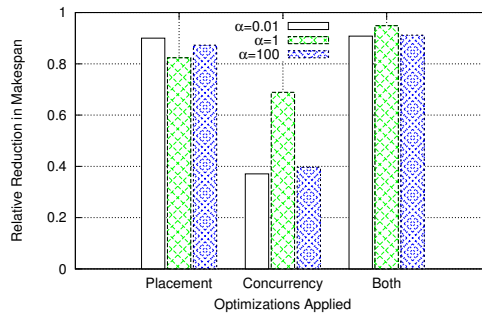


Figure 9: As resources are distributed across a greater number of data centers, the benefits of optimization increase, and this benefit becomes less dependent upon the application parameter α .

By combining both data placement and concurrency optimizations, we can improve performance even further. Figure 9 shows the reduction in makespan predicted by optimizing push and shuffle, as well as improving concurrency using pipelining. The results reflect a combination of the factors discussed earlier: more widely distributed resources afford greater optimization opportunity, and concurrency optimizations perform best when no single stage is dominant. We make two observations from the Figure 9. First, for each α value, the benefits increase as the network becomes more distributed. Second, the benefit is much higher for $\alpha=1$ for a local cluster, but as the network becomes more distributed, the benefits become less dependent on α , and converge for the 8-site global network environment. To understand these results better, Figure 10 shows the relative benefits of the different optimizations for the two extreme network environments: local cluster and 8-site global data center. Figure 10(a) shows that for a local cluster, pipelining is the dominant optimization and has the highest benefit for $\alpha=1$, as discussed earlier. The data placement optimizations do not improve the performance much since there is little heterogeneity to exploit in the system. On the other hand, Figure 10(b) shows that for a highly-distributed environment, the data placement optimizations



(a) 1 datacenter



(b) 8 datacenters

Figure 10: Relative benefit of data placement and concurrency optimizations for different network environments.

provide significant benefits due to the increased heterogeneity, and when combined with pipelining, this benefit can increase even further.

Overall, our results show the following.

- For a local cluster with fast links and homogeneous nodes, pipelining is the only optimization with significant benefit, and that too only when α is close to 1.
- As the network becomes more distributed and heterogeneous, both the data placement and the pipelining optimizations provide higher benefits, as they can exploit heterogeneity and latency hiding opportunities better. Further, the benefits tend to become independent of the application characteristics captured by the α parameter.

4.2 Compute rates

We show the results for varying the compute rates C_i of nodes in the system. We use a fixed network environment: the 8-site globally distributed data center, and vary C_i to include a moderate value (approximating a wordcount mapper in Hadoop on the PlanetLab hardware), as well as a low value (1% as large as the moderate value), and a large value (100 times as large as the moderate value). Such variations in C_i could reflect changes in either the compute-intensity of the application, or the performance of the underlying hardware. The key observations from the result shown in Figure 11 are threefold. First, the benefit expected from push and shuffle optimization does not depend clearly on this value of C_i . The reason is that C_i represents the balance between computation and communication within a MapReduce execution. Optimizing the push and shuffle phases takes both communication and computation into account, so it gracefully handles either a communication-heavy or a computation-heavy application. Second, pipelining shows a greater benefit when C_i takes on a moderate value. This is similar to the pattern we have identified with α , as pipelining is most promising when the stages to be overlapped are relatively balanced. When C_i takes on very small or very large values, then either communication phases (push, shuffle), or computation phases (map, reduce) dominate the runtime. Finally, combining placement optimizations with concurrency optimizations yields the best results, as we have seen throughout this paper.

5. RELATED WORK

MapReduce implementations [9, 13] have traditionally been deployed over a tightly-coupled cluster or data center, composed of largely homogeneous compute resources connected over a local-area network. Several research efforts have shown the impact on

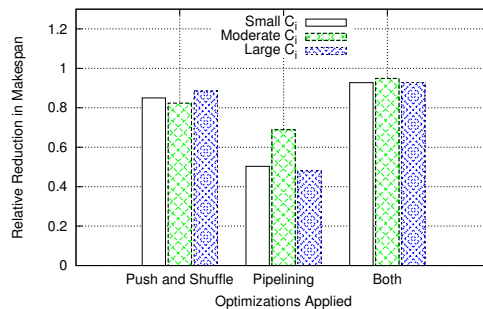


Figure 11: Impact of data placement and concurrency optimizations for various compute rates .

performance of MapReduce if this assumption is broken. Heterogeneity [24] in terms of node speeds was shown to have an adverse impact on the default MapReduce scheduling performance. Mantri [2] further showed the impact of heterogeneity in terms of machine and network characteristics as well as application workload on the performance of MapReduce within a data center environment. Our model is able to incorporate such heterogeneity in computation and communication characteristics, not only within a single data center, but also over a highly-distributed environment. Further, these techniques work in an online manner and are complementary to our model's outputs which define an offline execution plan.

The most relevant work to our paper is a recent work [5] that has explored multiple architectural choices for deploying MapReduce in a highly distributed environment. Our work uses a more general analytic approach to explore some of these tradeoffs, and some of the outputs of our data placement optimizations correspond to the architectural choices presented in this work. MOON [16] explored MapReduce performance in a local-area volunteer computing environment and extended Hadoop to provide improved performance under low reliability conditions. Our focus is on highly-distributed environments, and while our model captures node/link heterogeneity in terms of their speed and capacities, capturing reliability is part of future work.

Most MapReduce implementations rely on a distributed file system, such as GFS [12] or HDFS [22], for pushing input data onto the mapper nodes. Most existing work has examined the performance of MapReduce execution after the push phase. We explicitly model the push phase, which is particularly important due to the presence of multiple data sources in our environment. The pres-

ence of a distributed file system also implicitly imposes a global barrier between the push and map phases, where the mappers do not start execution until all the input data has already been placed on the compute nodes. Hadoop execution imposes a local barrier at the map/shuffle/reduce phases. Pipelining [7] has been proposed at the map/shuffle/reduce phase interfaces for improved responsiveness and performance. Our baseline model and concurrency optimizations capture all these variations and enable us to compare the performance of these choices across different phase boundaries.

Our model is data-oblivious, i.e., it does not assume knowledge of the input data contents, but such knowledge could be exploited to further improve performance. SkewReduce [15] presents the problem of application-specific computational skew, where different parts of the input data may require different amounts of computation resources. Such data-dependent compute requirements can be incorporated in our model by scaling C_i values in a data-dependent manner. CoHadoop [11] co-locates related data on the same node to improve performance of certain applications. Such an approach requires detailed knowledge of input data, which our model does not assume.

Other work has focused on scheduling or fine-tuning MapReduce parameters to provide better performance. Sandholm et al. [21] present a dynamic priority-based system for providing differentiated service to multiple MapReduce jobs. Quincy [14] is a framework for scheduling concurrent jobs to achieve fairness while improving data locality. Our focus is on optimizing the performance of individual job execution in a more distributed environment. Recent work [4] has proposed methods for automatically fine-tuning Hadoop parameters to optimize job performance. Our work takes a different approach, where we attempt to abstract away specific implementation details, so that our model is general enough to capture the abstraction behind many existing implementations. Thus, our model is useful for comparing different design and architectural choices, and some of its recommendations could be instantiated via some of the existing work.

6. CONCLUDING REMARKS

In this paper, we addressed the problem of executing MapReduce in a highly distributed environment, consisting of distributed data sources and computational resources. We presented a model to capture MapReduce execution on the distributed platform, and formulated a constrained optimization problem to minimize the execution time. We proposed and explored two classes of optimizations to the traditional MapReduce execution model: (i) data placement optimizations that optimize the allocation of data/compute resources, and (ii) concurrency optimizations that increase the concurrency of the execution. Our model results based on parameter estimation from PlanetLab measurements show that these optimizations are able to achieve upto 95% reduction in makespan over a baseline MapReduce model for a highly distributed environment. Moreover, these results also provide several insights into design choices based on platform and application characteristics. For instance, we find that an application's expansion factor α can influence optimal placement, ranging from a decentralized placement for low α values to a centralized placement for high α values. Our results also show that as the network becomes more distributed and heterogeneous, our optimizations provide higher benefits, as they can exploit heterogeneity opportunities better.

We intend to take this work forward in two main directions in the future. First, we would like to incorporate our model outputs into a real-world MapReduce implementation, such as Hadoop, which may also require addition of new mechanisms to the Hadoop implementation. Secondly, while our current model has focused on

performance as the optimization metric, we would like to extend our model to include other objectives such as reliability by modeling node/network reliability as well as incorporating techniques such as data and task replication.

7. REFERENCES

- [1] Amazon Elastic MapReduce. <http://aws.amazon.com/elasticmapreduce/>.
- [2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, and B. Saha. Reining in the outliers in map-reduce clusters. In *Proceedings of OSDI*, 2010.
- [3] M. Armbrust and et. al. Above the clouds: A berkeley view of cloud computing. In *Tech. Rep. UCB*, 2009.
- [4] S. Babu. Towards automatic optimization of mapreduce programs. In *ACM SOCC*, 2010.
- [5] M. Cardoso, C. Wang, A. Nangia, A. Chandra, and J. Weissman. Exploring MapReduce Efficiency with Highly-Distributed Data. In *The Second International Workshop on MapReduce and its Applications (MAPREDUCE'11)*, June 2011.
- [6] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, July 2003.
- [7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of NSDI*, 2010.
- [8] The Apache CouchDB Project. <http://couchdb.apache.org/>.
- [9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI*, 2004.
- [10] The Disco Project: Massive data - minimal code. <http://discoproject.org/>.
- [11] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Cohadoop: flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow.*, 4:575–585, June 2011.
- [12] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *Proc. of the 2003 19th ACM Symposium on Operating System Principles*, 2003.
- [13] Hadoop. <http://hadoop.apache.org/>.
- [14] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)*, 2009.
- [15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of ACM SOCC*, 2010.
- [16] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of HPDC '10*, 2010.
- [17] Apache Mahout: Scalable machine learning and data mining. <http://mahout.apache.org/>.
- [18] E. Nygren, R. Sitaraman, and J. Sun. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.
- [19] Apache Pig. <http://pig.apache.org/>.
- [20] Barnes & Noble - Cross-channel Analytics for Deeper Customer Insights. <http://www.asterdata.com/>

customers/barnes-and-noble.php.

- [21] T. Sandholm and K. Lai. Mapreduce optimization using dynamic regulated prioritization. In *ACM SIGMETRICS/Performance*, 2009.
- [22] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.
- [23] H. Williams. Model building in mathematical programming. 1999.
- [24] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of OSDI*, 2008.