MapReduce: Distributed Computing for Machine Learning

Dan Gillick, Arlo Faria, John DeNero

December 18, 2006

Abstract

We use Hadoop, an open-source implementation of Google's distributed file system and the MapReduce framework for distributed data processing, on modestly-sized compute clusters to evaluate its efficacy for standard machine learning tasks. We show benchmark performance on searching and sorting tasks to investigate the effects of various system configurations. We also distinguish classes of machine-learning problems that are reasonable to address within the MapReduce framework, and offer improvements to the Hadoop implementation. We conclude that MapReduce is a good choice for basic operations on large datasets, although there are complications to be addressed for more complex machine learning tasks.

1 Introduction

The Internet provides a resource for compiling enormous amounts of data, often beyond the capacity of individual disks, and too large for processing with a single CPU. Google's MapReduce [3], built on top of the distributed Google File System [4] provides a parallelization framework which has garnered considerable acclaim for its ease-of-use, scalability, and fault-tolerance. As evidence of its utility, Google points out that thousands of MapReduce applications have already been written by its employees in the short time since the system was deployed [3]. Recent conference papers have investigated MapReduce applications for machine learning algorithms run on multicore machines [1] and MapReduce is discussed by systems luminary Jim Gray in a recent position paper [6]. Here at Berkeley, there is even discussion of incorporating MapReduce programming into undergraduate Computer Science classes as an introduction to parallel computing.

The success at Google prompted the development of the Hadoop project [2], an open-source attempt to reproduce Google's implementation, hosted as a sub-project of the Apache Software Foundation's Lucene search engine library. Hadoop is still in early stages of development (latest version: 0.92) and has not been tested on clusters of more than 1000 machines (Google, by contrast, often runs jobs on many thousands of machines). The discussion that follows addresses the practical application of the MapReduce framework to our research interests.

Section 2 introduces the compute environments at our disposal. Section 3 provides benchmark performance on standard MapReduce tasks. Section 4 outlines applications to a few relevant machine learning problems, discussing the virtues and limitations of MapReduce. Section 5 discusses our improvements to the Hadoop implementation.

2 Compute Resources

While large companies may have thousands of networked computers, most academic researchers have access to more modest resources. We installed Hadoop on the NLP Millennium cluster in Soda Hall and at the International Computer Science Institute (ICSI), which are representative of typical university research environments.

Table 1 gives the details of the two cluster configurations used for the experiments in this paper. The NLP cluster is a rack of high-performance computing servers, whereas the ICSI cluster comprises a much larger number of personal desktop machines. It should be noted that ICSI also has a compute cluster of dedicated servers: over 20 machines, each with up to 8 multicore processors and 32 GB RAM, and a 1 Gbps link to terabyte RAID arrays. Compute jobs at ICSI are typically exported to these compute servers using *pmake-customs* software.[1] For this paper, we demonstrate that the desktop machines at ICSI can also be used for effective parallel programming, providing an especially useful alternative for I/O-intensive applications which would otherwise be crippled by the *pmake-customs* network bottleneck.

	NLP Cluster	ICSI Desktop Cluster
Machines	9	80
CPUs per machine	2x 3.4 GHz	1x 2.4 GHz
RAM per machine	4 GB	1 GB
Storage per machine	300 GB	30 GB
Network Bandwidth	1 GBit/s	100 MBit/s

Table 1: Two clusters used in this paper's experiments.

3 Benchmarks

We evaluate the performance of the 80-node ICSI desktop cluster to establish benchmark results for given tasks and configurations. Following the presentation in [3], we consider two computations that typify the majority of MapReduce algorithms: a *grep* that searches through a large amount of data, and a *sort* that transforms it from one representation to another. These two tasks represent extreme cases for the reduce phase, in which either none or all of the input data is shuffled across the network and written as output.

The data in these experiments are derived from the TeraSort benchmark [5], in which 100-byte records comprise a random 10-byte sorting key and 90 bytes of "payload" data. In the comparable experiments in [3], a cluster of 1800 machines was used to process a terabyte of data; due to our limited resources, we scaled the experiments down to 10 gigabytes of data – i.e., our input was 10^8 100-byte records.

We display our results in the same format as [3], showing the data transfer rates over the time of the computation, distinguishing the map phase (reading input from local disks) from the reduce phase (shuffling intermediate data over the network and writing the output to the distributed file system).

[1]This was Adam de Boor's CS 262 final project from 1989, and was updated by Andreas Stolcke for use at ICSI and SRI.

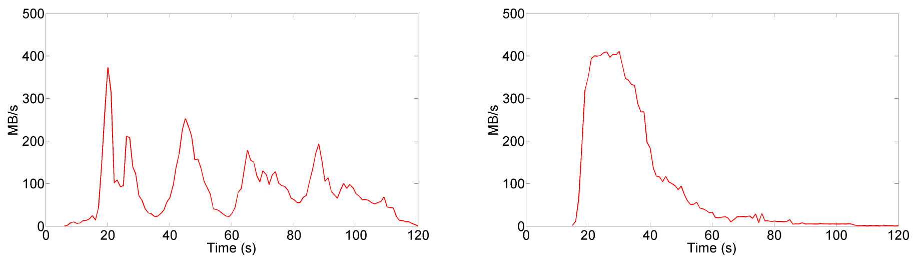


Figure 1: Effect of varying the size of map tasks. Left: with 600 map tasks (16MB input splits), performance suffers from repeated initialization costs. Right: with 75 map tasks (128MB input splits), a high sustained read rate is attainable, but the coarse granularity does not allow good load-balancing.

3.1 Distributed Grep

The distributed *grep* program searches for a character pattern and writes intermediate data only when a match is encountered. To simplify this further, we can consider an empty map function, which reads the input and never writes intermediate records. Using the Hadoop streaming protocol [2], which reads records from STDIN and writes to STDOUT, the map function is trivial:

```
#!/usr/bin/perl
while(<STDIN>){
  # do nothing
}
```

The reduce function is not needed since there is no intermediate data. Thus, this contrived program can be used to measure the maximal input data read rate for the map phase.

Figure 1 shows the effect of varying the number of map tasks (equivalent to varying the size of an input split). When the number of map tasks is high, reasonable load balancing is possible because faster machines will have a chance to run more tasks than slower machines. Also, because the tasks finish more quickly, the variance in running times is lower, and the effect of long-running "stragglers" is minimized. However, each map task incurs some amount of startup overhead, evidenced by the periodic bursts of data transfer in Figure 1 (left) caused by many tasks finishing and starting at the same time. Thus, it could be desirable to minimize initialization overhead by having a smaller number of map tasks, at the expense of poor load balancing, as in Figure 1 (right).

Our 80-node cluster reaches a maximum data read rate of about 400 MB/s, or about 5 MB/s per node. This is worse than Google's 1800-node cluster [3], which peaked at 30GB/s, or about 15 MB/s per node. The individual desktop hard disks at ICSI were probably mostly idle at the time, and were capable of reaching about 30-40 MB/s read rate (cold cache results), so the cluster's performance is far from its theoretical maximum efficiency. It is not clear whether this is due to the distributed file system implementation or the MapReduce overhead.

3.2 Distributed Sort

The distributed *sort* program is a more interesting example because it involves a costly reduce phase. Input data is read, all of it is written as intermediate data to the local disk buffer, it is

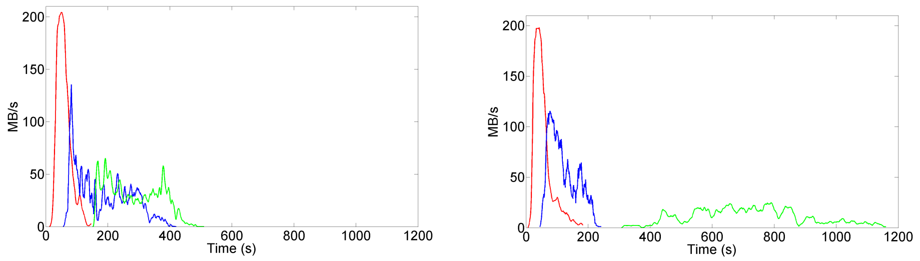


Figure 2: Effect of varying the size of map tasks. Left: 600 map tasks (16 MB input splits) suffers from repeated initialization costs. Right: 75 map tasks (128 MB input splits) allows higher sustained read rate, but the coarse granularity does not provide ideal load-balancing.

then transmitted over the network to the appropriate reduce task, which sorts by key, and finally writes the output. The shuffle phase begins as soon as the fastest map task is done and ends only after the slowest map task has finished. The reducer then sorts the data in memory and writes the output with triple replication to the distributed file system.

Using the Hadoop streaming protocol, the map function is an identity function which inserts a tab to delimit key-value pairs:

```
#!/usr/bin/perl
while(<STDIN>) {
  s/(\s)/\t\1/;
  print STDOUT;
}
```

The reduce function is an identity function which simply removes the tab:

```
#!/usr/bin/perl
while(<STDIN>) {
  s/\t//;
  print STDOUT;
}
```

Figure 2 demonstrates the effect of varying the number of reduce tasks (inversely, the size of the reduce splits). When there are many reducers, there is good load balancing, as in Figure 2 (left). Additionally, because sorting is $O(n \log n)$, having larger reduce splits will cause a greater amount of time to be spent sorting the keys. This can be observed in Figure 2 (right): about 300 seconds into the process, there is no data is being shuffled or written, because all reducers are busy sorting the large number of keys assigned to them. In general, it would seem that having more reducers is preferable; this of course must be weighed against the initialization overhead for each task, and the practical inconvenience that each reducer writes to a different output file.

Note that for this task the map phase achieves a peak read rate of about 200 MB/s, compared to 400 MB/s for the *grep* task. Because the map task must write its intermediate data to the local disk buffer, it effectively halves the read rate. One potential solution to this problem is to use separate hard disks for the local buffer and the local portion of the distributed file system; on the ICSI desktops, for example, there is a small secondary hard disk which is intended for swap space but could serve for the local buffer as well. Another possible solution is to implement a memory-based buffer that allows streaming of intermediate data prior to the completion of a map task.[2]

In the shuffle phase, the maximum transfer rate is about 150 MB/s; the theoretical limit, given a 100 Mbps network of 80 machines with no other traffic, should be about 500 MB/s aggregate bandwidth. Note that the transfer rates for the write phase of the reduce operations is considerably less than the read rate; this is because each block of the output files is written to the distributed file system as three replicas sent over the network to different machines. Thus, when the shuffle phase overlaps with the write phase, as in Figure 2 (left), the network can be experiencing extra traffic.

4 Applications to Machine Learning

Many standard machine learning algorithms follow one of a few canonical data processing patterns, which we outline below. A large subset of these can be phrased as MapReduce tasks, illuminating the benefits that the MapReduce framework offers to the machine learning community [1].

In this section, we investigate the performance trade-offs of using MapReduce from an algorithm-centric perspective, considering in turn three classes of ML algorithms and the issues of adapting each to a MapReduce framework. The performance that results depends intimately on the design choices underlying the MapReduce implementation, and how well those choices support the data processing pattern of the ML algorithm. We conclude this section with a discussion of changes and extensions to the Hadoop MapReduce implementation that would benefit the machine learning community.

4.1 A Taxonomy of Standard Machine Learning Algorithms

While ML algorithms can be classified on many dimensions, the one we take primary interest in here is that of procedural character: the data processing pattern of the algorithm. Here, we consider single-pass, iterative and query-based learning techniques, along with several example algorithms and applications.

4.1.1 Single-pass Learning

Many ML applications make only one pass through a data set, extracting relevant statistics for later use during inference. This relatively simple learning setting arises often in natural language processing, from machine translation to information extraction to spam filtering. These applications often fit perfectly into the MapReduce abstraction, encapsulating the extraction of local contributions to the map task, then combining those contributions to compute relevant statistics about the dataset as a whole. Consider the following examples, illustrating common decompositions of these statistics.

Estimating Language Model Multinomials: Extracting language models from a large corpus amounts to little more than counting n-grams, though some parameter smoothing over the statistics is also common. The map phase enumerates the n-grams in each training instance (typically a sentence or paragraph), and the reduce function counts instances of n-grams.

[2]This option has been investigated as part of Alex Rasmussen's Hadoop-related CS 262 project this semester.

Feature Extraction for Naive Bayes Classifiers: Estimating parameters for a naive Bayes classifier, or any fully observed Bayes net, again requires counting occurences in the training data. In this case, however, feature extraction is often computation-intensive, perhaps involving small search or optimization problems for each datum. The reduce task, however, remains a summation of each (feature, label) environment pair.

Syntactic Translation Modeling: Generating a syntactic model for machine translation is an example of a research-level machine learning application that involves only a single pass through a preprocessed training set. Each training datum consists of a pair of sentences in two languages, an estimated alignment between the words in each, and an estimated syntactic parse tree for one sentence.[3] The per-datum feature extraction encapsulated in the map phase for this task involves search over these coupled data structures.

4.1.2 Iterative Learning

The class of iterative ML algorithms – perhaps the most common within the machine learning research community – can also be expressed within the framework of MapReduce by chaining together multiple MapReduce tasks [1]. While such algorithms vary widely in the type of operation they perform on each datum (or pair of data) in a training set, they share the common characteristic that a set of parameters is matched to the data set via iterative improvement. The update to these parameters across iterations must again decompose into per-datum contributions, which is the case of the example applications below. As with the examples discussed in the previous section, the reduce function is considerably less compute-intensive than the map tasks.

In the examples below, the contribution to parameter updates from each datum (the map function) depends in a meaningful way on the output of the previous iteration. For example, the expectation computation of EM or the inference computation in an SVM or perceptron classifier can reference a large portion or all of the parameters generated by the algorithm. Hence, these parameters must remain available to the map tasks in a distributed environment. The information necessary to compute the map step of each algorithm is described below; the complications that arise because this information is vital to the computation are investigated later in the paper.

Expectation Maximization (EM): The well-known EM algorithm maximizes the likelihood of a training set given a generative model with latent variables. The E-step of the algorithm computes posterior distributions over the latent variables given current model parameters and the observed data. The maximization step adjusts model parameters to maximize the likelihood of the data assuming that latent variables take on their expected values. Projecting onto the MapReduce framework, the map task computes posterior distributions over the latent variables of a datum using current model parameters; the maximization step is performed as a single reduction, which sums the sufficient statistics and normalizes to produce updated parameters.

We consider applications for machine translation and speech recognition. For multivariate Gaussian mixture models (e.g., for speaker identification), these parameters are simply the mean vector and a covariance matrix. For HMM-GMM models (e.g., speech recognition), parameters are also needed to specify the state transition probabilities; the models, efficiently

[3]Generating appropriate training data for this task involves several applications of iterative learning algorithms, described in the following section.

stored in binary form, occupy tens of megabytes. For word alignment models (e.g., machine translation), these parameters include word-to-word translation probabilities; these can number in the millions, even after pruning heuristics remove the unnecessary parameters.

Discriminative Classification and Regression: When fitting model parameters via a perceptron, boosting, or support vector machine algorithm for classification or regression, the map stage of training will involve computing inference over the training example given the current model parameters. Similar to the EM case, a subset of the parameters from the previous iteration must be available for inference. However, the reduce stage typically involves summing over parameter changes.

Thus, all relevant model parameters must be broadcast to each map task. In the case of a typical featurized setting that often extracts hundreds or thousands of features from each training example, the relevant parameter space needed for inference can be quite large.

4.1.3 Query-based Learning with Distance Metrics

Finally, we consider distance-based ML applications that directly reference the training set during inference, such as the nearest-neighbor classifier. In this setting, the training data are the parameters, and a query instance must be compared to each training datum.

While it is the case that multiple query instances can be processed simultaneously within a MapReduce implementation of these techniques, the query set must be broadcast to all map tasks. Again, we have a need for the distribution of state information. However, in this case, the query information that must be distributed to all map tasks needn't be processed concurrently – a query set can be broken up and processed over multiple MapReduce operations. In the examples below, each query instance tends to be of a manageable size.

K-nearest Neighbors Classifier The nearest-neighbor classifier compares each element of a query set to each element of a training set, and discovers examples with minimal distances from the queries. The map stage computes distance metrics, while the reduce stage tracks k examples for each label that have minimal distance to the query.

Similarity-based Search Finding the most similar instances to a given query has a similar character, sifting through the training set to find examples that minimize a distance metric. Computing the distance is the map stage, while minimizing it is the reduce stage.

4.2 Performance and Implementation Issues

While the algorithms discussed above can all be implemented in parallel using the MapReduce abstraction, our example applications from each category revealed a set of implementation challenges. We conducted all of our experiments on top of the Hadoop platform. In the discussion below, we will address issues related both to the Hadoop implementation of MapReduce and the MapReduce framework itself.

4.2.1 Single-pass Learning

The single-pass learning algorithms described in the previous section are clearly amenable to the MapReduce framework. We will focus here on the task of generating a syntactic translation model from a set of sentence pairs, their word-level bilingual alignment, and their syntactic structure

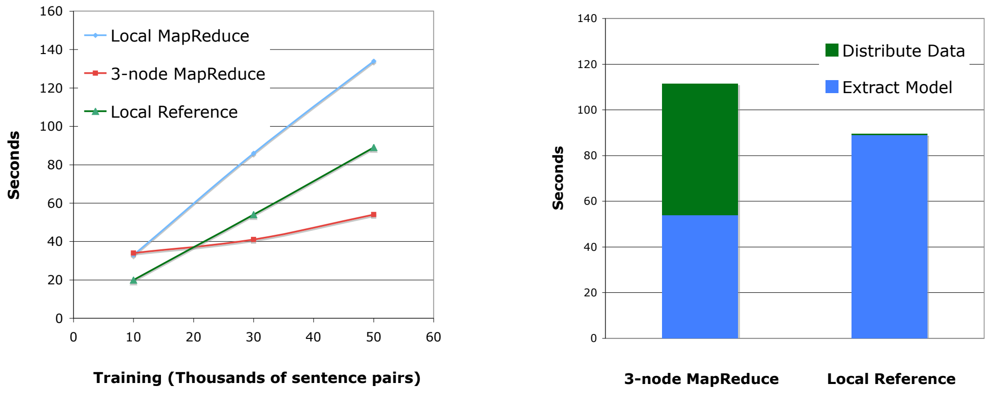


Figure 3: Generating syntactic translation models with Hadoop: (left) The benefit of distributed computation quickly outweighs the overhead of a MapReduce implementation on a 3-node cluster. (right) Exporting the data to the distributed file system incurs cost nearly equal to that of the computation itself.

(phrase structure tree). This task has several properties that distinguish it from benchmarking applications like sort and grep.

- The map stage processes several concurrent data sources (sentences, alignments and trees) that combine to form a complex data structure for each example.
- The map stage computation requires non-trivial computation with potentially exponential asymptotic running time in the size of the syntax tree (limited in practice to be linear with a moderate coefficient). Additionally, the 4,000 line code base can consume tens of megabytes of memory during the processing of one example.
- Hundreds of output keys can be generated for each training example.

Figure 3 shows the running times for various input sizes, demonstrating the overhead of running MapReduce relative to the reference implementation. The cost of running Hadoop cancels out some of the benefit of parallelizing the code. Specifically, running on 3 machines gave a speed-up of 39% over the reference implementation. The overhead of simulating a MapReduce computation on a single machine was 51% of the compute cost of the reference implementation. Distributing the task to a large cluster would clearly justify this overhead, but parallelizing to two machines would give virtually no benefit for the largest data set size we tested.

A more promising metric shows that as the size of the data scales, the distributed MapReduce implementation maintains a low per-datum cost. We can isolate the variable-cost overhead of each example by comparing the slopes of the curves in figure 3(a), which are all near-linear. The reference implementation shows a variable computation cost of 1.7 seconds per 1000 examples, while the distributed implementation across three machines shows a cost of 0.5 seconds per 1000 examples. So, the variable overhead of MapReduce is minimal for this task, while the static overhead of distributing the code base and channeling the processing through Hadoop's infrastructure is large. We would expect that substantially increasing the size of the training set would accentuate the utility of distributing the computation.

Thus far, we have assumed that the training data was already written to Hadoop's distributed file system (DFS). The cost of distributing data is relevant in this setting, however, due to drawbacks of

Hadoop's implementation of DFS. In the simple case of text processing, a training corpus need only be distributed to Hadoop's file system once, and can be processed by many different applications. On the other hand, this application references four different input sources, including sentences, alignments and parse trees for each example. When copying these resources independently to the DFS, the Hadoop implementation gives no control over how those files are mapped to remote machines. Hence, no one machine necessarily contains all of the data necessary for a given example.

Instead of distributing the training corpus in its raw, multi-file form, we must bundle the relevant parts of each example into a complex object and then transfer that object onto the Hadoop file system, thereby guaranteeing that the parts of each training example remain grouped. Having to repackage the data in this way has negative consequences; changes to the implementation of the application require redistribution of the training data, a costly activity. Applications that share some resources but not all will require redundant copies of their common subset. In sum, more transfers to the DFS will be required during the typical course of research. Figure 3(b) shows the cost breakdown of first distributing the training data to the DFS, then extracting statistics from it. When we include the cost of distribution, the parallel implementation actually underperforms the single-threaded reference. Because the training data must be bundled in an application-specific format, one can expect to incur this distribution cost regularly.

4.2.2 Iterative Learning

Implementing two iterative learning algorithms gave rise to more challenges than the single-pass example. As mentioned in the previous section, the map stage of each iteration depends upon the parameters generated by the reduce phase of the previous iteration. We chose two example applications: unsupervised word alignment for machine translation is an application of EM that generates word-to-word alignments. The k-means algorithm is an unsupervised clustering technique based on hard-EM.

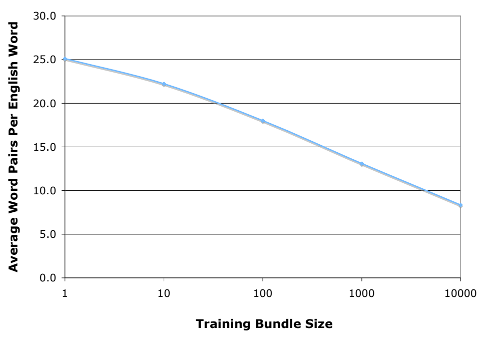
The word alignment task accentuates the problem of sharing state information because each pair of word types that co-occur in a training example have their own parameter that must be referenced during the map stage (E-step) of the algorithm. In a typical sentence pair of two 40-word sentences, 1,600 of these parameters must be available to the map function for each sentence pair.

Bundling the relevant state information with each training example adds substantially to the size of the training set. For instance, a pair of 40-word sentences would be accompanied by 6K of parameter data on top of its 1k of text data. Even if parameters are bundled with small groups of sentences, the overhead of bundling parameters is high (figure 4(a)).

Rather than bundling parameters with the input data, we would prefer to broadcast these parameters to all map nodes before each iteration of EM and reference them from within the map tasks on those machines. However, not only does Hadoop not support static reference to shared content across map tasks, the implementation also prevented us from adding this feature. In Hadoop, each map task runs in its own Java Virtual Machine (JVM), leaving no access to a shared memory space across multiple map tasks running on the same node.

Hadoop does provide some minimal support for sharing parameters in that its streaming mode[4] allows the distribution of a file to all compute nodes (once per machine) that can be referenced by a map task. In this way, data can be shared across map tasks without incurring redundant transfer costs. However, this work-around requires that the map task be written as a separate applications,

[4]The Hadoop streaming API allows arbitrary applications to compute the map and reduce functions by reference to the operating system.



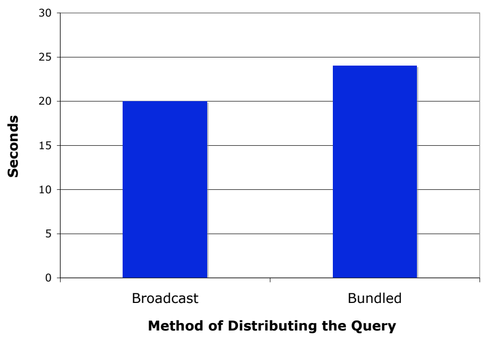


Figure 4: Benefits of Data Broadcasting: (left) The improved (smaller) ratio of relevant parameters to training words when batch-processing data argues for parameter broadcasting over bundling. (right) Bundling query data with training examples rather than broadcasting hinders a classifier's performance.

eliminating the type-safety guarantees of an integrated java program. Furthermore, it still requires that each map task load a distinct copy of its parameters into the heap of its JVM.

We extended Hadoop to provide similar support for java-internal map and reduce functions. Common state information is serialized and distributed to all compute nodes, where it can be loaded and referenced statically. In particular, we support the dispersion of state information via a networked file server to which all nodes have access. Note that this state file sharing is independent of Hadoop's distributed file system. We were still constrained to reload the state data for each map task.

4.2.3 Query-based Learning with Distance Metrics

To investigate whether Hadoop's MapReduce implementation was appropriate for distance-based learning algorithms, we implemented a nearest neighbor classifier to predict the genre of a movie from reviews provided by Netflix users.[5]

Like iterative learning, the query-based setting requires that we distribute common data to the map tasks: in this case, the query example. Figure 4(b) demonstrates the benefit of statically broadcasting the query sentence with our extension, rather than bundling the query with each training example.

5 Tailoring MapReduce for Machine Learning

5.1 Broadcasting and Parallel Files

MapReduce holds great potential for parallelizing machine learning applications. However, section 4.2 raises two major issues with the Hadoop implementation of MapReduce that we would hope to address.

First, we would advocate a shared space for the map tasks on a name node. Neither the MapReduce framework nor the Hadoop implementation support efficient distribution and referencing of

[5]This classification approach makes the assumption that people tend to review many movies of the same genre, and express rating preferences that are genre-dependent.

common data (such as model parameters or query examples). The ability to broadcast state information, whether filtered for each input or common across all map tasks, is necessary to support many machine learning algorithms.

The Hadoop streaming API provides a useful answer to this gap in functionality by allowing the user to include a file that is distributed to each compute node. This file can store the necessary state information, and is available to each map function. Our implementation of similar functionality for java-internal map and reduce functions provides the means to distribute state information in an efficient manner. However, the state information can only be shared across multiple map tasks via reference to the file system because each runs in its own virtual machine. Ideally, a MapReduce implementation for machine learning would expose shared memory to all map tasks on a compute node to eliminate this overhead.

Second, we desired the ability to tie parallel files together on the distributed file system, and allow the input reader for MapReduce tasks to parse multiple parallel files. Co-locating mutually referential file segments in this way would eliminate the need to bundle input data in application-specific formats. For instance, text could be decoupled from its many possible annotations (parse trees, part-of-speech tags, and word alignment, for example) on the DFS. With this functionality, a data set could reside permanently on the DFS, shared across multiple applications that would perhaps combine it with other related data on an ad hoc basis.

Implementing support for tying together parallel files proved challenging within the Hadoop source code. Much of the existing code assumes that input data is packaged in one file that can be distributed across the network independently of other data.

5.2 Static Type Checking and Convenience Methods

Hadoop proved somewhat difficult to use for reasons that we have also addressed during the course of this project. First, the architecture expects that a Hadoop binary will be invoked from the command line for all MapReduce jobs, which unpacks a referenced jar containing map and reduce functions. The command-line tools also awkwardly decouple manipulating the DFS from running MapReduce jobs. We prefer an architecture that invokes Hadoop DFS and MapReduce tasks from within our own applications, integrating the standard sequence of serialization and loading data on the DFS, running jobs, and retrieving distributed output. We have built a task manager that conveniently provides this array of functionality, along with translating between Hadoop's proprietary serialization and the Java standard.

Our task manager also provides another missing feature: proper static type checking. Hadoop's heavy use of Java's reflection facilities leave many type errors unchecked until runtime. Through our lightweight wrapper package for Hadoop that manages interactions, we enforce static type checking across data distribution, map and reduce functions, and data retrieval. All unchecked type casts and reflection operations are encapsulated within our package, shielding users from the risks of calling Hadoop's functions directly.

Finally, we provide facilities for launching MapReduce jobs remotely without requiring that the request originator install the array of binaries that are bundled with Hadoop. Using our package, no installation is required to originate jobs.

These convenience improvements allowed for our rapid development of several machine learning applications. We hope to release our package to the Berkeley community in short order.

6 Conclusion

By virtue of its simplicity and fault tolerance, MapReduce proved to be an admirable gateway to parallelizing machine learning applications. The benefits of easy development and robust computation did come at a price in terms of performance, however. Furthermore, proper tuning of the system led to substantial variance in performance, as we saw in section 3 (Benchmarks).

Defining common settings for machine learning algorithms lead us to discover the core shortcomings of Hadoop's MapReduce implementation. We were able to address the most significant, the need to broadcast data to map tasks. We also greatly improved the convenience of running MapReduce jobs from within existing Java applications. However, the ability to tie together the distribution of parallel files on the DFS remains an outstanding challenge.

All in all, MapReduce represents a promising direction for future machine learning implementations. When continuing to develop Hadoop and tools that surround it, we must strive to minimize the compromises between convenience and performance, providing a platform that allows for efficient processing *and* rapid application development.

References

[1] Cheng-Tao Chu et al. Map-Reduce for machine learning on multicore. In *NIPS*, 2007.

[2] Doug Cutting et al. About hadoop. `http://lucene.apache.org/hadoop/about.html`.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *ACM OSDI*, 2004.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SOSP*, 2003.

[5] Jim Gray. Sort benchmark homepage. `http://research.microsoft.com/barc/SortBenchmark`.

[6] Jim Gray et al. Scientific data managing in the coming decade. Technical Report MSR-TR-2005-10, Microsoft Research, 2005.