Scaling the Mobile Millennium System in the Cloud

Timothy Hunter, Teodor Moldovan, Matei Zaharia, Samy Merzgui, Justin Ma,
Michael J. Franklin, Pieter Abbeel, Alexandre M. Bayen
University of California, Berkeley

ABSTRACT

We report on our experience scaling up the Mobile Millennium traffic information system using cloud computing and the Spark cluster computing framework. Mobile Millennium uses machine learning to infer traffic conditions for large metropolitan areas from crowdsourced data, and Spark was specifically designed to support such applications. Many studies of cloud computing frameworks have demonstrated scalability and performance improvements for simple machine learning algorithms. Our experience implementing a real-world machine learning-based application corroborates such benefits, but we also encountered several challenges that have not been widely reported. These include: managing large parameter vectors, using memory efficiently, and integrating with the application's existing storage infrastructure. This paper describes these challenges and the changes they required in both the Spark framework and the Mobile Millennium software. While we focus on a system for traffic estimation, we believe that the lessons learned are applicable to other machine learning-based applications.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*.

General Terms

Algorithms, Performance, Experimentation.

1. INTRODUCTION

Cloud computing promises to democratize parallel data processing by making large server farms available to organizations that lack such resources in-house. However, scaling real-world applications raises issues that are often unexplored by cloud researchers. To this end, we report our experience scaling one such application: the *Mobile Millennium* traffic information system developed at UC Berkeley [11].

Mobile Millennium is a traffic estimation and prediction system that infers traffic conditions using GPS measurements from drivers running cell phone applications, taxicabs, and other mobile and static data sources. The system was initially deployed in the San Francisco Bay area and later extended to other locations such as Sacramento and Stockholm. For San Francisco alone, the system processes about 500,000 data points per day from taxicabs, in addition to data from various other sources.

Although we initially developed *Mobile Millennium* as a set of single-node services, we recently decided to parallelize the costlier stages of the system using the cloud to achieve three key benefits: timelier predictions, scalability to larger road networks, and the ability to use more accurate, but more computationally expensive traffic models. This paper discusses lessons learned parallelizing one of the main algorithms in the system: an *expectation maximization* (EM) algorithm that estimates traffic conditions on city roads.

We believe that our work will be of interest to cloud researchers for several reasons. First, our EM algorithm is representative of a large class of iterative machine learning algorithms, including clustering, classification, and regression methods, for which popular cloud programming frameworks like Hadoop and Dryad are often inefficient [7, 10, 15]. Our lessons are likely applicable to these applications too.

Second, although researchers have developed several specialized programming models for iterative algorithms [10, 5, 28, 15, 21], many of these systems have only been evaluated on simple applications. We found that our more complex real-world application posed several challenges that have not been explored extensively, such as disseminating large parameter vectors and utilizing memory efficiently.

Finally, our application had to function as a component of a large existing system, leading to additional challenges integrating cloud and non-cloud infrastructure. For example, one such bottleneck was storage: *Mobile Millennium* uses a PostgreSQL server for common data, which performed surprisingly poorly under the bursty request pattern generated by multiple worker nodes running in parallel.

We implemented the traffic estimation algorithm in Spark, a framework for in-memory cluster computing that was designed specifically to support iterative algorithms [28]. The lessons from this real-world application have provided valuable feedback into Spark's design, and as a result we derived optimizations that sped up the application by 2–3× each.

Ultimately, we were able to achieve the scalability goals that brought us to the cloud: our system scales to 160 cores and can process data with a more accurate traffic model than the one we initially used, at a rate faster than real-time.

From a traffic estimation perspective, one of the major advantages of the work we report on here is the distribution of computation on cloud platforms in an efficient manner, without micro-

SOCC'11, October 27–28, 2011, Cascais, Portugal.

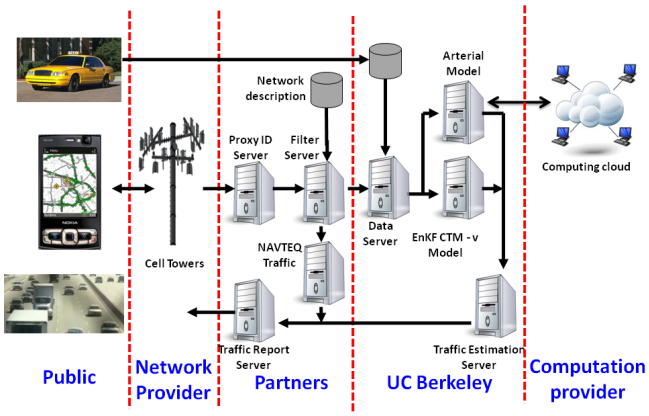


Figure 1: Schematic architecture of the *Mobile Millennium* system.

management of the computation by the user. Because typical traffic engineering systems require the partitioning of computational resources into clusters that usually reflect geographic areas and/or levels of data activity in specific segments of the transportation network, engineers designing computational infrastructure to support real-time traffic inference have to manually allocate nodes to tasks a priori. The proposed framework using Spark enables the automated allocation of these nodes at scale, with only minor parametrization on the part of the user, a capability that has high potential impact in the field of traffic monitoring.

We start with an overview of the *Mobile Millennium* system (Section 2) and the traffic estimation algorithm (Section 3). We then introduce the Spark framework and explain how we used it to parallelize the algorithm (Section 4). Next, we discuss the more surprising bottlenecks we found in our cloud implementation and the resulting optimizations (Section 5). We evaluate our implementation and these optimizations in Section 6. We survey related work in Section 7 and conclude in Section 8.

2. THE MOBILE MILLENNIUM SYSTEM

Traffic congestion affects nearly everyone in the world due to the environmental damage and transportation delays it causes. The 2007 Urban Mobility Report [23] states that traffic congestion causes 4.2 billion hours of extra travel in the United States every year, which accounts for 2.9 billion extra gallons of fuel and an additional cost of $78 billion. Providing drivers with accurate traffic information reduces the stress associated with congestion and allows drivers to make informed decisions, which generally increases the efficiency of the entire road network [4].

Modeling highway traffic conditions has been well-studied by the transportation community with work dating back to the pioneering work of Lighthill, Whitham and Richards [14]. Recently, researchers demonstrated that estimating highway traffic conditions can be done using only GPS probe vehicle data [25]. Arterial roads, which are major urban city streets that connect population centers within and between cities, provide additional challenges for traffic estimation. Recent studies focusing on estimating real-time arterial traffic conditions have investigated traffic flow reconstruction for single intersections using dedicated traffic sensors. Dedicated traffic sensors are expensive to install, maintain and operate, which limits the number of sensors that governmental agencies can deploy on the road network. The lack of sensor coverage across the

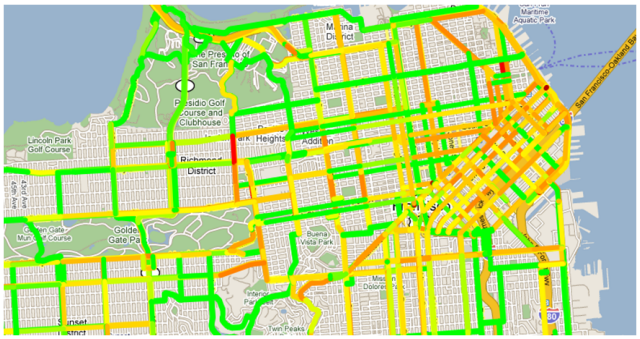


Figure 2: An example output of traffic estimates on the *Mobile Millennium* visualizer

arterial network thus motivates the use of GPS probe vehicle data for estimating traffic conditions.

The work we present represents one component of the UC Berkeley *Mobile Millennium* project [11]. One of the goals of the project is to assess the capability of GPS-enabled mobile phones to provide traffic data that can be used to estimate real-time conditions, forecast future conditions, and provide optimal routing in a network with stochastically varying traffic conditions. The project has resulted in a real-time traffic estimation system that combines dedicated sensor data with GPS data from probe vehicles. The *Mobile Millennium* project comprises more than eight million lines of Java code and is supported by a professional team of engineers. The software has been deployed on various architectures by industry and academic partners who are part of this effort [11]. As such, this work follows standard practices from industrial development and represents a large effort.

The *Mobile Millennium* system incorporates a complete pipeline for receiving probe data, filtering it, distributing it to estimation engines and displaying it, all in in real-time, as pictured in Figure 1. This software stack, written in Java, evaluates *probabilistic distribution of travel times* over the road links, and uses as input the *sparse, noisy* GPS measurements from probe vehicles. A first proof of concept of this stack was written in Python [12], and an early cloud prototype was developed using the Hadoop interface to Python. This prototype was then rewritten in Scala (a high-level language for the Java VM) to accommodate the Spark programming interface and to leverage the infrastructure of the *Mobile Millennium* system (which is in Java).

The most computation-intensive parts of this pipeline have all been ported to a cloud environment. We briefly describe the operations of the pipeline, pictured in figure 3.

- We map each point of raw (and possibly noisy) GPS data to a collection of nearby *candidate projections* on the road network (Fig. 3(a)).
- For each vehicle, we reconstruct the most likely trajectory using a Conditional Random Field [13] (Fig. 3(b)).
- Each segment of the trajectory between two GPS points is referred as an *observation* (Fig. 3(c)). An observation consists in a start time, an end time and a route on the road network. This route may span multiple road links, and starts and ends at some offset within some links.
- The observations are grouped into time intervals and sent to a traffic estimation engine, which runs the learning algorithm described in the next section and returns distributions of travel times for each link (Fig. 3(d)).

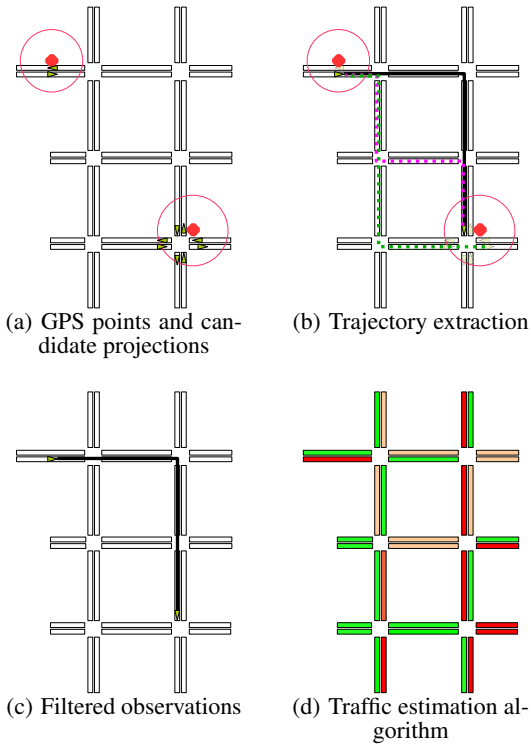


Figure 3: Arterial traffic estimation process.

- The travel time distributions are then stored and broadcast to clients and to a web interface shown in Figure 2.

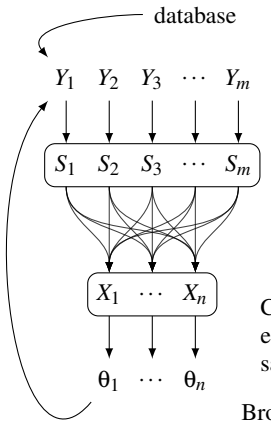
It is important to point out that MM is intended to work at the scale of large metropolitan areas. The road network considered in this work is a real road network (a large portion of San Francisco downtown and of the greater Bay Area, comprising 25000 road links) and the data is collected from the field (as opposed to simulated). A consequence of this setting is the scalability requirement for the traffic algorithms we employ. Thus, from the outset, our research has focused on designing algorithms that could work for large urban areas with hundreds of thousands of links and millions of observations.

3. TRAFFIC ESTIMATION ALGORITHM

The goal of the traffic estimation algorithm is to infer how congested the links are in an arterial road network, given periodic GPS readings from vehicles moving through the network. We model the network as a graph (V,E), where V are the vertices (road intersections) and E are the links (streets). For each link $e \in E$, where n is the total number of links in the network, the algorithm outputs the time it takes to traverse the link as a probability distribution. To make the inference problem tractable, we model the link traversal times for each link e as an independent Gamma distribution with parameters θ_e (as shorthand, we let θ_e represent the two values that parametrize a Gamma distribution).[1]

The algorithm inputs are the road network (V,E), as well as the observed trajectories of GPS-equipped vehicles Y. Each observation Y_i describes the i-th trajectory's travel time and path (which consists of one or more road links) as inferred by earlier stages of the *Mobile Millennium* pipeline. Physical properties of the road network, such as speed limits and link lengths, are also taken into account.

[1]We experimented with a few standard distributions from the literature (Gamma, Normal and Log-normal). Based on our experiments, the Gamma distribution fit the data best. Computing the most likely Gamma distribution from a set of samples has no closed-form solution and is more expensive than in the case of the Normal or Log-normal distributions, but was deemed worthwhile for the added accuracy.



Broadcast road network and load observations

Draw samples for each observations and weigh them by their likelihood (E step)

Collect samples by link (Shuffle step)

Compute maximum likelihood estimate of parameters based on samples (M step)

Broadcast link distribution parameters and iterate

Figure 4: Data flow in the importance sampling EM algorithm we employed. The algorithm iterates through the E, shuffle and M steps until it converges.

Estimating the travel time distributions is made difficult by the fact that we do not observe travel times for individual links. Instead, each observation only specifies the total travel time for an entire list of links traveled. To get around this problem, we use an iterative *expectation maximization* (EM) algorithm [8, 19]. The central idea of the algorithm is to randomly partition the total travel time among links for each observation, then weigh the partitions by their likelihood according to the current estimate of travel time distributions. Next, given the weighted travel time samples produced for each link, we update the travel time distribution parameters for the link to maximize the likelihood of these weighted samples. By iteratively repeating this process, the algorithm converges to a set of travel time distribution parameters that fit the data well. The sample generation stage is called the expectation (E) step, while the parameter update stage is called the maximization (M) step.

Figure 4 shows the data flow in the algorithm in more detail. In the E-step, we generate per-link travel time samples from whole trajectories; specifically, for each trajectory Y_i, we produce a set of samples $S_i = \{(s_{e_i}, w_{e_i})\}_{e_i \in Y_i}$ by randomly dividing Y_i's observed travel time among its constituent links (producing a travel time s_{e_i} for each edge $e_i \in Y_i$), and we assign a weight w_{e_i} as the likelihood of travel time s_{e_i} according to e's current travel time distribution θ_e. In the shuffle step, we regroup the samples in the S_i's by link, so that each link e now has samples $X_e = \{(s_{e_i}, w_{e_i})\}$ from all the trajectories that go over it. In the M-step, we recompute the parameters θ_e to fit link e's travel time distribution to the samples X_e.

The model needs to be configured with several free parameters, such as the number of samples and a regularization prior. We chose these based on test runs with a small dataset.

Next, we describe how we parallelized the algorithm using Spark (Section 4), and the challenges that we encountered in executing algorithm efficiently in the cloud (Section 5).

4. SPARK FRAMEWORK

We parallelize our EM algorithm using Spark [27, 28], a cluster computing framework developed at Berkeley. Spark offers several

```
// Load observations into memory as a cached RDD
observations = spark.textFile("hdfs://...")
                    .map(parseObservation).cache()

params = // Initialize parameter values

while (!converged) {
  // E-step: generate (linkId, sampledVals) pairs
  samples = observations.map(
    ob => generateSamples(ob, params))

  // Shuffle and M-step: group samples for each link,
  // update params, and return them to the master
  params = samples.groupByKey().map(
    (linkId, vals) => updateParam(linkId, vals)
  ).collect()
}
```

Figure 5: Simplified Spark code for the EM algorithm. Spark ships the Scala code fragments passed to each *map* operation transparently to the cluster, along with any variables they depend on.

benefits. First, it provides a high-level programming model using a language-integrated syntax similar to DryadLINQ [26], saving substantial development time over lower-level frameworks like MPI. Second, Spark programs are written in Scala [3], a high-level language for the JVM, which allows us to integrate with the Java codebase of *Mobile Millennium*. Third, Spark is explicitly designed to support iterative algorithms, such as EM, more efficiently than data flow frameworks like MapReduce and Dryad.

Spark's programming model centers on parallel collections of objects called *resilient distributed datasets* (RDDs). Users can define RDDs from files in a storage system and transform them through data-parallel operations such as *map*, *filter*, and *reduce*, similar to how programmers manipulate data in DryadLINQ and Pig [20]. However, unlike in these existing systems, users can also control the *persistence* of an RDD, to indicate to the system that they will reuse an RDD in multiple parallel operations. In this case, Spark will cache the contents of the RDD in memory on the worker nodes, making reuse substantially faster. At the same time, Spark tracks enough information about how the RDD was built to reconstruct it efficiently if a node fails. This in-memory caching is what makes Spark faster than MapReduce or Dryad for iterative computations: in existing systems, iterative applications have to be implemented as a series of independent MapReduce or Dryad jobs, each of which reads state from disk and writes it back out to disk, incurring substantial I/O and object serialization overhead.

To illustrate the programming model, we show simplified Spark code for the EM algorithm in Figure 5. Recall that the EM algorithm consists of the E-step, where we generate random travel time samples for each link in each trajectory observed, the shuffle step, where we group these samples by link, and the M-step, where we use the grouped values to estimate the new per-link travel distribution parameters. These steps can readily be expressed as a MapReduce computation, which we implement in Spark using the *map* and *groupByKey* operations. Note, however, that each iteration of the EM algorithm will reuse the *same* dataset of original observations. The code thus starts by loading the observations into an in-memory cached RDD, by passing a text file through a `parseObservation` function that reads each line of text into a Scala class representing the observation. We then reuse this RDD to update the link parameters.

We found this in-memory caching capability crucial for performance in both EM and other iterative machine learning algorithms. In the *Mobile Millennium* application, caching provided a 2.8× speedup. In other, less CPU-intensive applications, we have seen speedups as large as 30× [27]. Other cluster computing frameworks, such as Twister [10] and Piccolo [21], have also been built around in-memory storage for the same reason, and our results corroborate their findings.

5. OPTIMIZATIONS AND LESSONS

Although the EM algorithm can readily be expressed using Spark (as well as other frameworks) and benefits substantially from in-memory storage, we found that several other optimizations that are less studied in the literature were necessary to achieve good performance and scalability. We now discuss three of these optimizations: efficient memory utilization, efficient broadcast of large objects, and optimized access to the application's storage system.

5.1 Memory Utilization

Several recent cluster computing frameworks, including Twister [10], Piccolo [21], and Spark, are designed explicitly to support iterative applications by providing in-memory storage. As discussed in the previous section, our results validate the benefit of this feature: in our application, it can yield a 2.8× speedup. Nonetheless, we found that simply having in-memory storage facilities available in the framework was insufficient to achieve good performance in a complex application like traffic estimation.

One of the main challenges we encountered was *efficient utilization* of memory. Unlike simpler machine learning applications that cache and operate on large numeric vectors, our application cached data structures representing paths traveled by vehicles or sets of links parameters. When we first implemented these data structures using idiomatic Java constructs, such as linked lists and hashtables, we quickly exhausted the memory on our machines, consuming more than 4× the size of the raw data on disk. This happened because the standard data structures, especially pointer-based ones, incur considerable storage overhead per item. For example, in a Java LinkedList, each entry costs 24 bytes (for an object header and pointers to other entries) [18], whereas the values we stored in these lists were often 4-byte `int`s or `float`s. With this much overhead, running an algorithm in memory can be much costlier than anticipated; indeed, our first attempts to use caching ran *slower* than a disk-based version because they were constantly garbage-collecting.

Solution and Lessons Learned: We improved our memory utilization by switching to array-backed data structures where possible for the objects we wanted to cache and minimizing the number of data structures that contained small objects and many pointers. One difficult part of the problem was simply recognizing the cause of the bloat: Java (and Scala) programmers are typically unaware of the overhead of simple collection types. However, switching to more compact data representations did not come for free: we lost some of the convenience of working with idiomatic data types in a high-level language in the process. In general, one of the main reasons why programmers use tools like Hadoop and DryadLINQ is that they can program in a high-level language (e.g., Java or C#). While the memory overhead of these languages did not matter for systems that stream records from disk, such as MapReduce and Dryad, it becomes important for in-memory computing frameworks.

We believe that framework designers can do a lot to help users utilize memory efficiently. In particular, it would be useful for frameworks to provide libraries that expose an idiomatic collec-

```
net = readRoadNetwork()

observations.map(
  ob => process(ob, net)
)
```

a) Original

```
net = readRoadNetwork()
bv = spark.broadcast(net)

observations.map(
  ob => process(ob, bv.get())
)
```

b) With Broadcast Variables

Figure 6: Example Spark syntax showing how to use broadcast variables to wrap a large object (`net`) that should only be sent to each node once.

tion interface but pack data efficiently, and tools for pinpointing the sources of overhead. We are developing both types of tools for Spark. For example, the system now supports a "serializing" version of the cache that marshals each object cached into a byte array using a fast serialization format similar to Protocol Buffers [1], which can save space even for simple data like strings and integers.

5.2 Broadcast of Large Parameter Vectors

Parallel machine learning algorithms often need to broadcast data to the worker nodes, either at the beginning of the job or at each iteration. For example, in our traffic estimation algorithm, we needed to broadcast the road network to all the nodes at the start of the job, as well as the updated parameters computed after each iteration.

In the simple machine learning algorithms commonly evaluated in the systems literature, such as *k*-means and logistic regression, these broadcast values are small (hundreds of bytes each). In our application, they were considerably larger: about 38 MB for the road network of the Bay Area. We have seen even larger parameter vectors in other Spark applications: for example, the spam classifier in [22] had a parameter vector hundreds of MB in size, with a feature for each of several million terms, and this vector needed to be re-broadcast after each iteration.

Initially, our application performed poorly because we packaged the parameter vectors needed with each task (i.e., partition of a job) sent to the cluster, which was the default behavior in Spark. The master node's bandwidth became a bottleneck, capping the rate at which tasks could be launched and limiting our scalability.

Solution and Lessons Learned: To mitigate the problem of large parameter vectors, we added an abstraction called *broadcast variables* to Spark, which allows a programmer to send a piece of data to each slave only once, rather than with every task that uses the data [28]. To the programmer, broadcast variables look like wrapper objects around a value, with a `get()` method that can be called to obtain the value. The variable's value is written once to a distributed file system, from which it is read once by each node the first time that a task on the node calls `get()`. We illustrate the syntax for broadcast variables in Figure 6.

We used broadcast variables to send both static data that is used throughout the job, such as the road vector, and the new parameter vectors computed on each iteration. As we show in Section 6, broadcast variables improved the performance of our data loading phase by about 4.6×, and the speed of the overall application by 1.6×.

For larger parameter vectors, such as the approximately 100 MB vector in the spam classification job above, even reading the data once per node from a distributed file system is a bottleneck. This led us to implement more efficient broadcast methods in Spark, such as a BitTorrent-like mechanism optimized for datacenter networks called Cornet [6].[2] Because many real-world machine learning applications have large parameter vectors (with features for each word in a language, each link in a graph, etc.), we believe that efficient broadcast primitives will be essential to support them.

[2]The main optimizations in Cornet are a topology-aware data dissemination scheme and large block sizes suitable for high-bandwidth, low-latency networks.

5.3 Access to On-Site Storage System

One of the more surprising bottlenecks we encountered was access to our application's storage system. Following standard practices, *Mobile Millennium* uses a PostgreSQL database to host information shared throughout our pipeline (including the road network and the observations received over time). We chose PostgreSQL for its combination of reliability, convenience, and support for geographic data through PostGIS [2]. Cloud storage solutions were not widespread at the time of the choice, but we soon realized that our common storage solution was ill-suited to cloud computations. Indeed, while PostgreSQL had served our on-site pipeline without problems, it performed very poorly under the access pattern of our parallel algorithm: the application initially spent more than 75% of its time waiting on the database.

The problem behind this slowdown was the *burstiness* of the access pattern of the parallel application. The database had to service a burst of hundreds of clients reading slices of the observation data when the application started, as well as a similar burst of writes when we finished computing. The total volume of data we read and wrote was not large—about 800 MB—so it should have been within the means of a single server. However, the contention between the simultaneous queries slowed them down dramatically.

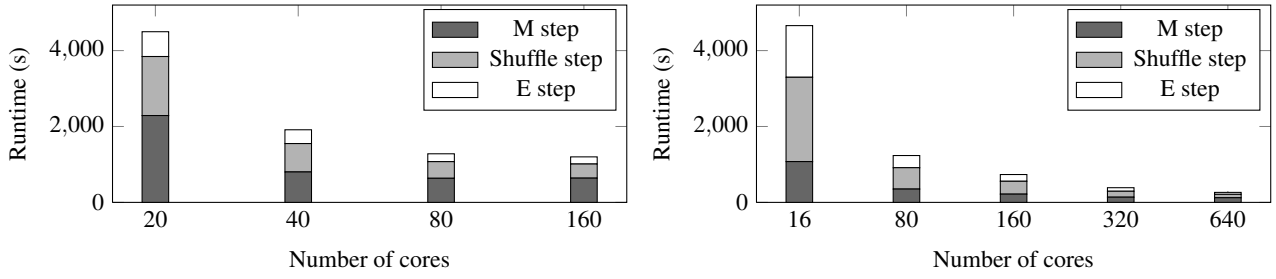
Solution and Lessons Learned: We ultimately worked around the problem by periodically exporting the data from PostgreSQL to a Hadoop file system (HDFS) instance on EC2. We still use the PostgreSQL database as the primary storage of the system, however, due to its richer feature set (e.g., geographic queries with PostGIS) and wider accessibility through SQL-based interfaces. Therefore, although the HDFS caching approach removed the bottleneck, it introduced new management challenges, as we must keep HDFS consistent with the database. *Mobile Millennium* receives new data every few minutes—eventually, we would prefer a solution that lets us access new data from the cloud as soon as it arrives.

We believe that database implementers can do a lot to support the bursty patterns of requests from workers in parallel applications. Neither the amount of data we read nor the amount of data we wrote was beyond the means of a single server (both were several hundred MB), but the bursty access pattern was simply not well-supported by the engine. Most likely, database engines will need to recognize the parallel query workload introduced by distributed applications, either through hints in the queries or heuristics that watch for similar requests, and to order the requests in an efficient manner to avoid excessive disk seeks. In general, enabling developers to use the same storage system for both on-site and cloud applications would be a key step in making the cloud more widely accessible for parallel data processing, and given the near-ubiquitous use of RDBMSes in existing applications, they are a natural place to start.

6. PERFORMANCE EVALUATION

In this section, we evaluate how much the cloud implementation and its associated memory, broadcast and storage optimizations (Section 5) helped with scaling the *Mobile Millennium* EM traffic estimation algorithm.

As mentioned in Section 2, we originally designed and prototyped the algorithm in the Python programming language, to work



(a) Amazon EC2 (b) NERSC Cluster

Figure 7: Running time experiments on different clusters. See section 5 for details.

on a single node. However, working on a single computer quickly revealed its limitations: the code would take 40 minutes per iteration to process a single 30 minute time interval of data! Moreover, the amount of memory available on a single machine would limit the number of observations considered, as well as the number of samples generated. Distributing the computation across machines provides a twofold advantage: each machine can perform computations in parallel, and the overall amount of memory available is much greater. To understand how restricted the single-machine deployment was, we could only generate 10 samples for each observation in the E-step in order for the computation to stay within the machine's 10 GB limit. Because the single-node implementation could not generate enough E-step samples to create a good approximation of the travel time distribution, the accuracy of the algorithm was limited as well. By contrast, using a 20-node cluster and letting the E-step generate 300 samples per observation, the computation was an order of magnitude faster, and the accuracy of the predicted models increased significantly.

Scaling. First, we evaluated how the runtime performance of the EM job could improve with an increasing number of nodes/cores. The job was to learn the historical traffic estimate for San Francisco downtown for a half-hour time-slice. This data included 259,215 observed trajectories, and the network consisted of 15,398 road links. We ran the experiment on two cloud platforms: the first was using Amazon EC2 `m1.large` instances with 2 cores per node, and the second was a cloud managed by the National Energy Research Scientific Computing Center (NERSC) with 4 cores per node. Figure 7(a) shows near-linear scaling on EC2 until 80–160 cores. Figure 7(b) shows near-linear scaling for all the NERSC experiments. The limiting factor for EC2 seems to have been network performance. In particular, some tasks were lost due to repeated connection timeouts.

Individual optimizations. We evaluated the effects of the individual optimizations discussed in Section 5. For these experiments we ran the experiments on a 50-node Amazon EC2 cluster of `m1.large` instances, and used a data set consisting of 45×10^6 observations split into 800 subtasks.

With respect to the data loading (Section 5.3) we looked at three configurations: (a) connecting to the main database of *Mobile Millennium* which stores the master copy of the data, (b) connecting to a cloud-hosted version of the *Mobile Millennium* DB, and (c) caching data from the main DB to the cloud's HDFS. Table 1 shows the throughput for loading data under each configuration, and shows that our final solution (c) shows a three orders of magnitude improvement over the original implementation using (a).[3]

To evaluate the benefit of in-memory computation (Section 5.1), we compared the run times of the EM job without caching (i.e., reloading data from HDFS on each iteration) and with in-memory caching (Table 2). Without caching, the runtime was 5,800 seconds. With caching, the runtime was reduced to 2,100 seconds, providing a nearly 3× speedup. Most of this improvement comes from reducing the runtime of the E-step and the shuffle step since they read the cached observations. The M-step does not improve because it reads newly-generated per-link samples (which have to be regenerated on each iteration as per Section 3), and the current implementation of shuffle writes its outputs to disk to help with fault tolerance.

[3]Although we do not report the extraction and preprocessing overhead for (c), this initial cost is amortized over the number of repetitions we perform for the experiments.

Table 1: Data loading throughput for various storage configurations.

Configuration	Throughput
Connection to on-site DB	239 records/sec
Connection to cloud-hosted DB	6,400 records/sec
Main DB data cached in HDFS	213,000 records/sec

Table 2: Comparison of EM runtimes with different settings: a single-core version, a parallel version with all our optimizations, and parallel versions with no caching and no broadcast.

Configuration	Load time	E step	Shuffling	M step
Single core	4073	6276	18578	7550
Parallel	468	437	774	936
No caching	0	2382	2600	835
No broadcast	2148	442	740	1018

Finally, we explore the benefit of broadcasting parameters (Section 5.2). A copy of the road network graph must be available to every worker node as it loads and parses the observation data, so broadcast is crucial. To this end, we evaluated how long it took to load 45 million observations over a 50-node cluster when (1) a copy of the road network graph is bundled with each task and (2) the network graph is broadcast ahead of time. The network graph for the Bay Area was 38 MB, and it took 8 minutes to parse the observations using a broadcast network graph — by contrast, the loading time was 4.5 times longer without broadcasting.

7. RELATED WORK

There has recently been great interest in running sophisticated machine learning applications in the cloud. Chu et al. showed that MapReduce can express a broad class of parallel machine learning algorithms, and that it provides substantial speedups on multicore machines [7]. However, as we discussed in this article, these algorithms encounter scaling challenges when we want to expand beyond a single machine and run them on a public cloud. The main remedies to these challenges involve exploiting data locality and reducing network communication between nodes.

In the systems literature, Twister, Spark, HaLoop and Piccolo

provide MapReduce-like programming models for iterative computations using techniques such as in-memory storage [10, 28, 5, 21]. GraphLab and Pregel also store data in memory, but provide a message-passing model for graph computations [15, 16]. While these systems enable substantial speedups, we found that issues other than in-memory storage, such as broadcast of large parameter vectors, also posed challenges in our application. We wish to highlight these challenges by describing a more complex real-world application than the simple benchmarks commonly employed.

Recent work in large-scale machine learning has addressed some of the algorithmic issues in scaling applications to the cloud. McDonald et al. [17] discuss distributed training strategies over MapReduce where data is partitioned across nodes, and nodes perform local gradient descent before averaging their model parameters between iterations. Other studies about distributed dual averaging optimization methods [9] and distributed EM [24] explored the network bandwidth savings, and some optimization algorithms that restrict the communication topology of the worker nodes.

8. CONCLUSIONS

We have presented our experience scaling up the *Mobile Millennium* traffic information algorithm in the cloud and identified lessons that we believe will also apply to other complex machine learning applications. Our work affirmed the value of in-memory computation for iterative algorithms, but also highlighted three challenges that have been less studied in the systems literature: efficient memory utilization, broadcast of large parameter vectors, and integration with off-cloud storage systems. All three factors were crucial for performance. We hope that these lessons will be of interest to designers of cloud programming frameworks and storage systems. Our experiences with *Mobile Millennium* have already influenced the design of the Spark framework.

Acknowledgements

This research is supported in part by gifts from Google, SAP, Amazon Web Services, Cloudera, Ericsson, Huawei, IBM, Intel, Mark Logic, Microsoft, NEC Labs, Network Appliance, Oracle, Splunk and VMWare, by DARPA (contract #FA8650-11-C-7136), and by the National Sciences and Engineering Research Council of Canada. The generous support of the US Department of Transportation and the California Department of Transportation is gratefully acknowledged. We also thank Nokia and NAVTEQ for the ongoing partnership and support through the *Mobile Millennium* project.

References

[1] Kryo – Fast, efficient Java serialization. `http://code.google.com/p/kryo`.

[2] PostGIS. `http://postgis.refractions.net`.

[3] Scala programming language. `http://scala-lang.org`.

[4] X. Ban, R. Herring, J. Margulici, and A. Bayen. Optimal sensor placement for freeway travel time estimation. *Proceedings of the 18th International Symposium on Transportation and Traffic Theory*, July 2009.

[5] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient iterative data processing on large clusters. In *VLDB*, 2010.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *SIGCOMM*, 2011.

[7] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, 2007.

[8] A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 1–38, 1977.

[9] J. Duchi, A. Agarwal, and M. Wainwright. Distributed dual averaging in networks. In *NIPS*, 2010.

[10] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *HPDC '10*, 2010.

[11] *Mobile Millennium* Project. `http://traffic.berkeley.edu`.

[12] T. Hunter, R. Herring, A. Bayen, and P. Abbeel. Path and travel time inference from gps probe vehicle data. In *NIPS Analyzing Networks and Learning with Graphs*, 2009.

[13] T. Hunter, R. Herring, A. Bayen, and P. Abbeel. Trajectory reconstruction of noisy GPS probe vehicles in arterial traffic. *In preparation for IEEE Transactions on Intelligent Transport Systems*, 2011.

[14] M. Lighthill and G. Whitham. On kinematic waves. II. A theory of traffic flow on long crowded roads. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 229(1178):317–345, May 1955.

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *UAI*, 2010.

[16] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD*, 2010.

[17] R. T. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *Conference of the North American Chapter of the Association of Computation Linguistics*, pages 456–464, 2010.

[18] N. Mitchell and G. Sevitsky. Building memory-efficient Java applications: Practices and challenges. PLDI 2009 Tutorial.

[19] R. Neal and G. Hinton. A view of the em algorithm that justifies incremental, sparse, and other variants. *Learning in graphical models*, 89:355–368, 1998.

[20] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[21] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[22] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song. Design and evaluation of a real-time url spam filtering service. In *IEEE Symposium on Security and Privacy*, May 2011.

[23] TTI. Texas Transportation Institute: Urban Mobility Information: 2007 Annual Urban Mobility Report. `http://mobility.tamu.edu/ums/`, 2007.

[24] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. In *ICML*, 2008.

[25] D. Work, S. Blandin, O. Tossavainen, B. Piccoli, and A. Bayen. A traffic model for velocity data assimilation. *Applied Mathematics Research eXpress*, 2010(1):1, 2010.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, Jul 2011.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.