Parallelizing Machine Learning Algorithms

Juan Batiz-Benet Quinn Slack Matt Sparks Ali Yahya

{jbenet , sqs , msparks , alive} @ cs . stanford . edu

***Abstract*—Implementing machine learning algorithms involves of performing computationally intensive operations on large data sets. As these data sets grow in size and algorithms grow in complexity, it becomes necessary to spread the work among multiple computers and multiple cores. Qjam is a framework for the rapid prototyping of parallel machine learning algorithms on clusters.**

I. Introduction

Many machine learning algorithms are easy to parallelize in theory. However, the fixed cost of creating a distributed system that organizes and manages the work is an obstacle to parallelizing existing algorithms and prototyping new ones. We present Qjam, a Python library that transparently parallelizes machine learning algorithms that adhere to a constrained MapReduce model of computation.

II. Previous Work

Significant research has been done in the area of distributed data processing. Perhaps the most notable and relevant contribution is the MapReduce programming model [1], which applies the *map* and *reduce* functions from functional programming to large datasets spread over a cluster of machines. Since their introduction, the MapReduce concepts have been implemented in several projects for highly parallel computing, such as Apache Hadoop [2].

Chu et al. [3] show how ten popular machine learning algorithms can be written in a “summation form” in which parallelization is straightforward. The authors implemented these algorithms on a MapReduce-like framework and ran them on multicore machines. They yielded a near-linear speedup as the number of cores was increased.

Whereas Chu et al. experimented on single multicore machines, our project extends their ideas to a cluster of networked computers. Rather than use a framework like Hadoop [2], which is intended for large batch processing jobs, we have designed and implemented a lightweight framework for low-latency tasks that requires minimal server configuration.

III. Choosing a Language

Our two criteria for language choice were ease of development and good support for linear algebra operations. C++ is known to provide excellent performance, but it is not conducive to rapid prototyping. MATLAB's licensing costs make it infeasible for use on large clusters. We evaluated Python and R as possible options.

The following sections compare Python and R and explain why we chose Python.

A. Code Style

R is stylistically very similar to MATLAB. Matrix and list operations are first-class operations, and it provides built-in equivalents to most of MATLAB's probability and statistics functions. Also, the R interpreter makes plotting and visualizing data structures just as easy as in MATLAB.

While Python's syntax is less suited for matrix operations, the NumPy package for Python [4] includes Matlib, an interface that attempts to emulate MATLAB's syntax. It is designed specifically for MATLAB programmers and to ease the porting of MATLAB code. Certain syntactic elements are still overly verbose (e.g., `M.transpose()` vs `M'`) and may hinder the readability of an algorithm.

Strictly from a syntactic and stylistic perspective, R wins on its simplicity and similarity to MATLAB. However, Python's slight disadvantage here is far outweighed by its relative performance, as described below.

B. Performance

Python is not a mathematical language, but it is easily extensible and provides many high-quality mathematics packages (NumPy in particular). Though interpreted, Python can easily call down to C and avoid dynamic-language overhead in computationally intensive operations. For instance, NumPy's matrix multiplication is implemented in C, yielding very good performance without sacrificing ease of use or readability in the Python code.

We benchmarked the performance of R against Python's (using the NumPy package). In order to avoid implementation- or algorithm-specific bias, we decided to benchmark the common linear algebra functions, such as matrix multiplication, that are ubiquitous in learning algorithms.

Table I shows the running times of Python and R on various operations using different matrix or vector sizes, as well as the time ratio of $Python/R$. Every test ran 10,000 operations. In terms of performance, Python is the clear winner. It outperforms R in every case, most of the cases by an order of magnitude. In the worst case, Python takes 82% of the time that R takes.

Although naïve Python implementations of *serial* machine learning algorithms tend to be slower than their MATLAB equivalents, recent benchmarks of the *parallel* sparse autoencoder show that Python's performance penalty is not as significant in parallel execution as it is amortized over the number of workers. Also, since we are targeting rapid prototyping, not peak production performance, a small performance hit is acceptable.

Matrix Multiplication

Size	Python	R	Python / R
50	0.1600	0.2208	0.7246
75	0.5800	0.7339	0.7903
100	1.3030	1.6323	0.7983
150	4.2350	5.2311	0.8096
250	18.9190	22.9759	0.8234

Element-Wise Matrix Multiplication

Size	Python	R	Python / R
150	0.0350	0.1576	0.2221
225	0.0760	0.3741	0.2032
300	0.1510	0.6859	0.2201
450	0.9310	2.0938	0.4446
750	3.3010	5.4117	0.6100

Matrix Transpose

Size	Python	R	Python / R
50	0.0010	0.0325	0.0308
75	0.0010	0.0610	0.0164
100	0.0010	0.1030	0.0097
150	0.0010	0.2196	0.0046
250	0.0010	0.6119	0.0016

Vector Inner Product

Size	Python	R	Python / R
2500	0.0040	0.0523	0.0765
3750	0.0060	0.0772	0.0777
5000	0.0070	0.1030	0.0680
7500	0.0100	0.1519	0.0658
12500	0.0160	0.2514	0.0636

TABLE I
Benchmarks of Python and R for linear algebra operations.

IV. Architecture

This section describes the architecture of the qjam framework. Subsection A defines the major components of the system and how they communicate with each other. Subsection B explains the programming interface. Subsection C describes the protocol that Qjam uses to communicate. Finally, subsection D describes details of Qjam's Python implementation.

A. Components

Qjam is a single-master distributed system made up of instances of the following components:

Worker — The Worker is a program that is copied to all of the remote machines during the bootstrapping process. It is responsible for waiting for instructions from the Master, and upon receiving work, processing that work and returning the result.

RemoteWorker — The RemoteWorker is a special Python class that communicates with the remote machines. One RemoteWorker has a single target machine that can be reached via ssh. There can be many RemoteWorkers with the same target (say, in the case where there are many cores on a machine), but only one target per RemoteWorker. At creation, the RemoteWorker bootstraps the remote machine by copying the requisite files to run the Worker program, via ssh. After the bootstrapping process completes, the RemoteWorker starts a Worker process on the remote machine and attaches to it. The RemoteWorker is the proxy between the Master and the Worker.

Master — The Master is a Python class that divides up work and assigns the work units among its pool of RemoteWorker instances. These RemoteWorker instances relay the work to the Worker programs running on the remote machines and wait for the results.

Figure 1 shows the communication channels between components on multiple machines.

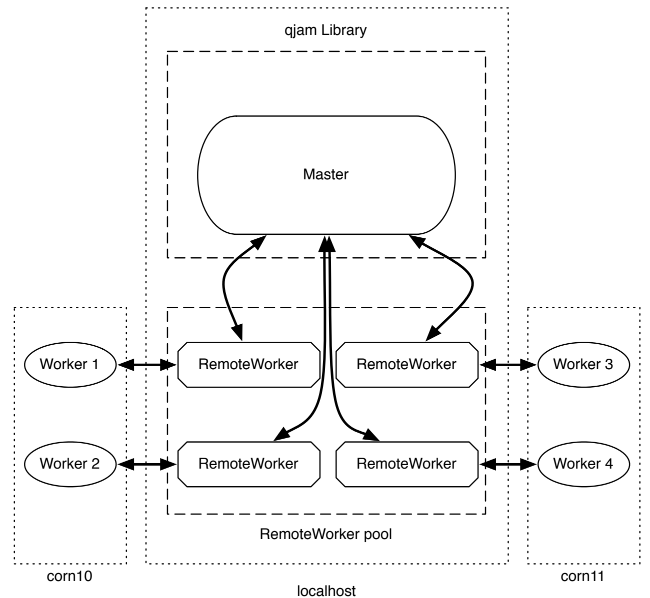

Fig. 1. Master controlling four RemoteWorkers with Workers in two machines.

B. Qjam Library API

This section describes the interface exposed by Qjam and provides a simple example of its use. The workflow of a typical distributed computation on Qjam is divided into two phases. In the initialization phase, the client creates an instance of the `Master` class by passing its constructor a list of remote workers. In the execution phase, the client specifies a Python module containing a function, `mapfunc`, to be executed on each worker along with a dataset and a list of parameters for the computation. The framework then breaks the dataset into smaller pieces and distributes the work to the worker processes. The following two sub-

sections elaborate on the details.

B.1 Initialization

At a fundamental level, the Qjam library is built on top of the RemoteWorker class. An instance of RemoteWorker defines a single connection to a worker node. A collection of RemoteWorker objects is passed to the constructor of Master to define the pool of workers available to the master.

The following code demonstrates the initialization of the worker pool and master.

```
workers = [RemoteWorker('corn15.stanford.edu'),
           RemoteWorker('corn16.stanford.edu'),
           RemoteWorker('corn17.stanford.edu')]
master = Master(workers)
```

B.2 Execution

Once the list of RemoteWorkers and the Master have been initialized, the client must first wrap any static data into a DataSet object, and then it can issue a call to master.run(...) to distribute the computation. Results of the work are obtained through the return value of this call.

B.2.a Creating a DataSet Object. In order for Master to know how to partition the task data between the registered RemoteWorkers, it needs to have some notion of how the data is structured. In order to fulfill that requirement, the client can either resort to one of the convenience DataSet classes provided by Qjam, or define a custom data class that inherits from BaseDataSet.

The DataSet classes provided by Qjam include support for Python tuples or lists of any kind, or NumPy matrices. In the case the client wishes to represent the data as a matrix, he can choose between NumpyMatrixDataSet, which simply represents the matrix in-memory, or NumpyMatrixFileDataSet, which represents the matrix as a file on disk in the case that it is too large to fit in memory.

In the case that the client wishes to define a custom data set class that inherits from Qjam's BaseDataSet class, he must implement at least the following two member functions:

1. chunks() Returns the number of chunks into which the internal data can be divided.
2. slice(index) Returns the slice of data at the given index where index is an integer in $[0, chunks() - 1]$.

B.2.b Defining a Code Module. The code that is to be executed at each remote worker must be written by the client in a self-contained Python module. The module must contain a function called mapfunc that will be called by the framework. The function mapfunc must take two arguments: The first argument are the parameters, θ, passed by the client in the call to master.run. θ can be of any type and is passed, without modification, to every remote worker. The second argument of mapfunc is a subset of the DataSet created by the client as described in section B.2.a. Note that Qjam guarantees that different workers will receive different, non-overlapping subset of the data.

The client also has the option of defining a reduce function as part of the same module. If the client opts out of this option, then the return value of mapfunc must be of a type that defines the sum operator or a list of types that define the sum operator. More complex return values are possible if the client defines a custom reduce function.

A simple mapfunc might be defined as follows.

```
def multiply_sum(theta, dataset):
  return sum([theta * x_i for x_i in dataset])

mapfunc = multiply_sum
```

B.2.c Calling master.run. Once the code module and the dataset object have been defined, the client can make a call to the function master.run to distribute the computation. The master.run function takes the client-defined code module, the parameters, and a DataSet object as arguments. The return value of master.run is the result of the computation.

The following simple example shows a call to master.run.

```
from examples import multiply_sum

params = 42
dataset = ListDataSet(range(1, 100))
result = master.run(multiply_sum, params, dataset)
```

C. Protocol

Communication between the Qjam master and each of the workers occurs via a connection that is persistent throughout the existence of the master object. This section describes the details of the communication protocol that is used by our implementation to assign tasks to remote workers, efficiently distribute data, and coalesce results.

The protocol implementation relies on five different types of messages. Figure 2 shows the use of those messages in a typical communication workflow.

C.1 TASK Message

The TASK message type is sent by the master to each worker to initiate a task. It contains an encoded[1] representation of the client's code module, a hash of the chunks that compose the worker's assigned dataset, and an encoded representation of the client-specified parameters.

C.2 STATE Message

Upon receiving a TASK message, the worker must respond with a STATE message. This message contains a status field that can take one of two values: "running" or "blocked". In the case that the status field is set to blocked, the worker must include a separate field whose value is a list of hash values where each hash value identifies a chunk of the dataset that the worker is missing.

[1] Objects are encoded using a base64 representation of the serialized Python object.

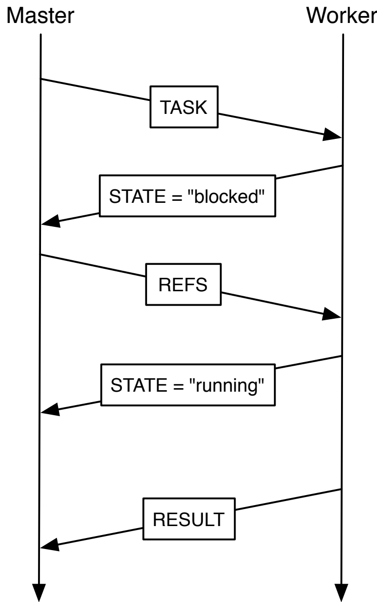

Fig. 2. Communication between `Master` and a single `RemoteWorker` during the execution of a typical Qjam distributed computation

C.3 REFS Message

If the master receives a STATE message whose status is set to "`blocked`", then it responds with a REFS message. This type of message includes a list of encoded objects that correspond to the data chunks that the worker identified as missing.

C.4 RESULT Message

Finally, the RESULT message is sent to the from the worker to the master whenever it completes its task. This message contains an encoded representation of the computation's result.

C.5 ERROR Message

In the case that the worker encounters an unexpected state, it can send an ERROR reply to any message sent by the master. This message contains a description of the error.

D. Feature Highlights

D.1 Remote Data Caching

One important feature of Qjam is caching of data by each remote worker. The master initially sends each worker a list of the hash values of each data chunk that the worker will need for a given task. If a worker cannot find the data object that corresponds to one or more hash values, it requests them from the master and caches them. In later iterations, the master can take advantage of data locality by assigning workers data chunks that they have seen before.

D.2 Automatic Bootstrapping

An important design goal for Qjam is to make the execution of a distributed computation as easy as possible. To that end, our implementation strives to minimize the amount of setup necessary on each worker machine. Qjam has the ability to transparently bootstrap each remote worker with all of the code it needs to communicate with the master. After initiating an SSH connection to a worker, the master sends a source code of the worker protocol implementation and remotely executes it on the worker node. This allows any computer with Python and an SSH server to serve as a remote worker—no manual setup required.

V. Evaluation

We benchmarked the framework running various algorithms with multiple workers.

A. L-BFGS Sparse Autoencoder

We benchmarked qjam using a sparse autoencoder with L-BFGS [5]. A sparse autoencoder is an unsupervised learning algorithm that automatically learns features from unlabeled data. It is implemented as a neural network with one hidden layer (parameters) that adjusts its weight values at each iteration over the training set. L-BFGS is a limited-memory, quasi-Newton optimization method for unconstrained optimization.

We benchmarked the running time of the sparse autoencoder using a parallelized cost function (with L-BFGS optimizing it). We tested a regular single-core implementation against 2, 4, 8, and 16 workers over four multicore machines. We tested with three datasets (of 1,000, 10,000, and 100,000 patches each). Table II summarizes per-iteration results, while Table III is the sum of all iterations plus the master's setup overhead.

workers	1k	10k	100k
1	0.1458 (1.0x)	0.7310 (1.0x)	10.0282 (1.0x)
2	0.1752 (0.8x)	0.3321 (2.2x)	4.6782 (2.1x)
4	0.2634 (0.5x)	0.3360 (2.2x)	2.4858 (4.0x)
8	0.5339 (0.3x)	0.5251 (1.4x)	1.8046 (5.6x)
16	0.9969 (0.2x)	1.0186 (0.7x)	1.4376 (6.9x)

TABLE II
Iteration Mean Time (seconds)

workers	1k	10k	100k
1	76 (1.0x)	370 (1.0x)	5030 (1.0x)
2	92 (0.8x)	170 (2.2x)	2350 (2.1x)
4	137 (0.5x)	173 (2.1x)	1253 (4.0x)
8	275 (0.3x)	270 (1.4x)	914 (5.5x)
16	544 (0.1x)	529 (0.7x)	703 (7.2x)

TABLE III
Total Running Time (seconds)

For the large job (100k), qjam performs better than the single-core every time, as seen in Figure 3(a). The running

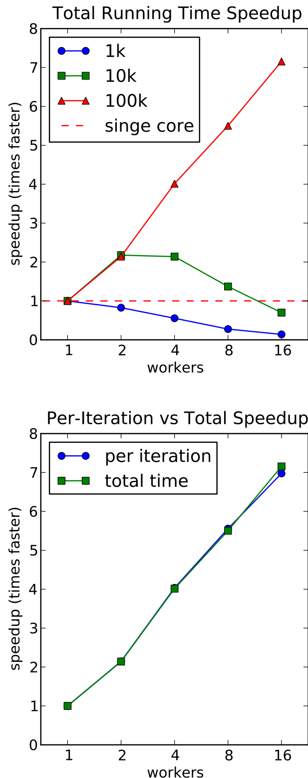

Fig. 3. a) Total Running Time Speedup. b) Per-Iteration and Total Speedups (100k patches). Per-Iteration times reflect only client code, whereas Total times incorporate the master's coordination overhead.

times show a significant speedup when using qjam with multiple workers. In particular, the 16 worker trial saw a speedup of over 7 times the single-core's running time. Comparing the speedups observed per-iteration against the total running time of this trial (Figure 3(b)) reveals that the master overhead is very small, yielding no significant slowdown.

The non-intensive job (10k) saw a small increase in performance, but not significantly. For the smallest, trivial job (1k), the overhead of coordinating many workers with very little to compute drove the performance below that of the single-core's implementation, just as expected.

A particular number of workers seems to be suited for a particular job size: for the 10k patches trials, the best run was that with 2 workers. The others performed worse, though still most performed better than the single core. For the largest job, though the 16 worker runtime was the lowest, the savings from 8 workers to 16 were proportionally small, requiring twice the number of workers for 1x more. This further confirms that in order to minimize the overhead of distributing the job, the number of workers should be picked according to the job size. Further research should explore various job sizes with different worker pools.

VI. Future Work

The next logical step in the development of Qjam is running more benchmarks with significantly larger datasets and more iterations. It is important to observe Qjam's performance and reliability under high stress. Moreover, having more data about Qjam's performance would more clearly reveal whatever bottlenecks remain.

With regard to features, another important step is to achieve feature parity with other, more general parallel frameworks (e.g. MapReduce). Handling worker failures, anticipating stragglers, and using a smarter job scheduling algorithm will likely yield performance improvements, particularly when running on larger or heterogeneous clusters than those we tested on.

We currently use SSH and JSON to transfer data and messages. Using a more efficient protocol and data encoding will improve performance and reliability. We noticed that SSH occasionally dropped connections and implemented a workaround to automatically reconnect upon failure; this, however, remains the biggest source of instability on Qjam.

Finally, aside from implementation related improvements, we will also improve usability. As a start, we can offer a wider range of convenience DataSet subclasses—beyond those that encapsulate matrices and lists (e.g., ImageDataSet, AudioDataSet).

VII. Conclusion

We have presented a framework that greatly simplifies the rapid prototyping of distributed machine learning algorithms. Qjam provides an abstraction of the complexities associated with building an entire distributed system just to run a task on multiple computers in parallel. As a result, it is now possible rapidly prototype and execute distributed computations without having to explicitly manage communication overhead and the distribution of code and data to remote workers.

Moreover, Qjam offers satisfactory performance. On a computation that takes a mean of 10 seconds to complete on a single machine, we observed 4x speed-up on 4 workers and 7x speedup on 16 workers.

VIII. Acknowledgements

We would like to recognize the assistance of those in our CS 229 class research group: Prof. Andrew Ng, Adam Coates, Bobby Prochnow, Milinda Lakkam, Sisi Sarkizova, Raghav Pasari, and Abhik Lahiri.

References

[1] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, Berkeley, CA, USA, 2004, pp. 10–10, USENIX Association.

[2] "Apache hadoop," `http://hadoop.apache.org`, Dec 2010.

[3] C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *Advances in Neural Information Processing Systems 19: Proceedings of the 2006 Conference*. The MIT Press, 2007, p. 281.

[4] "Scientific computing tools for python – numpy," `http://numpy.scipy.org/`, Dec 2010.

[5] Dong C. Liu and Jorge Nocedal, "On the limited memory bfgs method for large scale optimization," *Mathematical Programming*, vol. 45, pp. 503–528, 1989, 10.1007/BF01589116.