Parallelizing Machine Learning Algorithms

Quinn Slack, Juan Batiz-Benet, Ali Yahya, Matt Sparks

Stanford University

Abstract

We propose *qjam*, a Python framework for the rapid development and execution of machine learning algorithms on a cluster of computers. It is designed to be simple and as transparent as possible. It uses existing infrastructure by bootstrapping workers via ssh. Datasets are sliced automatically and distributed evenly among the workers. Results are seamlessly returned synchronously through the entry point in the qjam library.

Introduction

The sheer number of computations that Machine Learning algorithms perform yield considerably slow performance in single processors. This slow performance, coupled with increasingly larger data sets, hinders the execution of existing algorithms, and obstructs the prototyping of new, more sophisticated algorithms. Thus, it is becoming imperative to parallelize algorithms and take advantage of multiple computers with multiple cores. Fortunately, many machine learning algorithms lend themselves well to parallelization. Unfortunately, the overhead of creating a distributed system that organizes and manages the work is a roadblock to parallelizing existing algorithms and prototyping new ones. We present Qjam, a Python library that abstracts this overhead away, providing a simple framework that different algorithm implementations can easily leverage to handle work parallelization.

Architecture

The framework consists of three major 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.

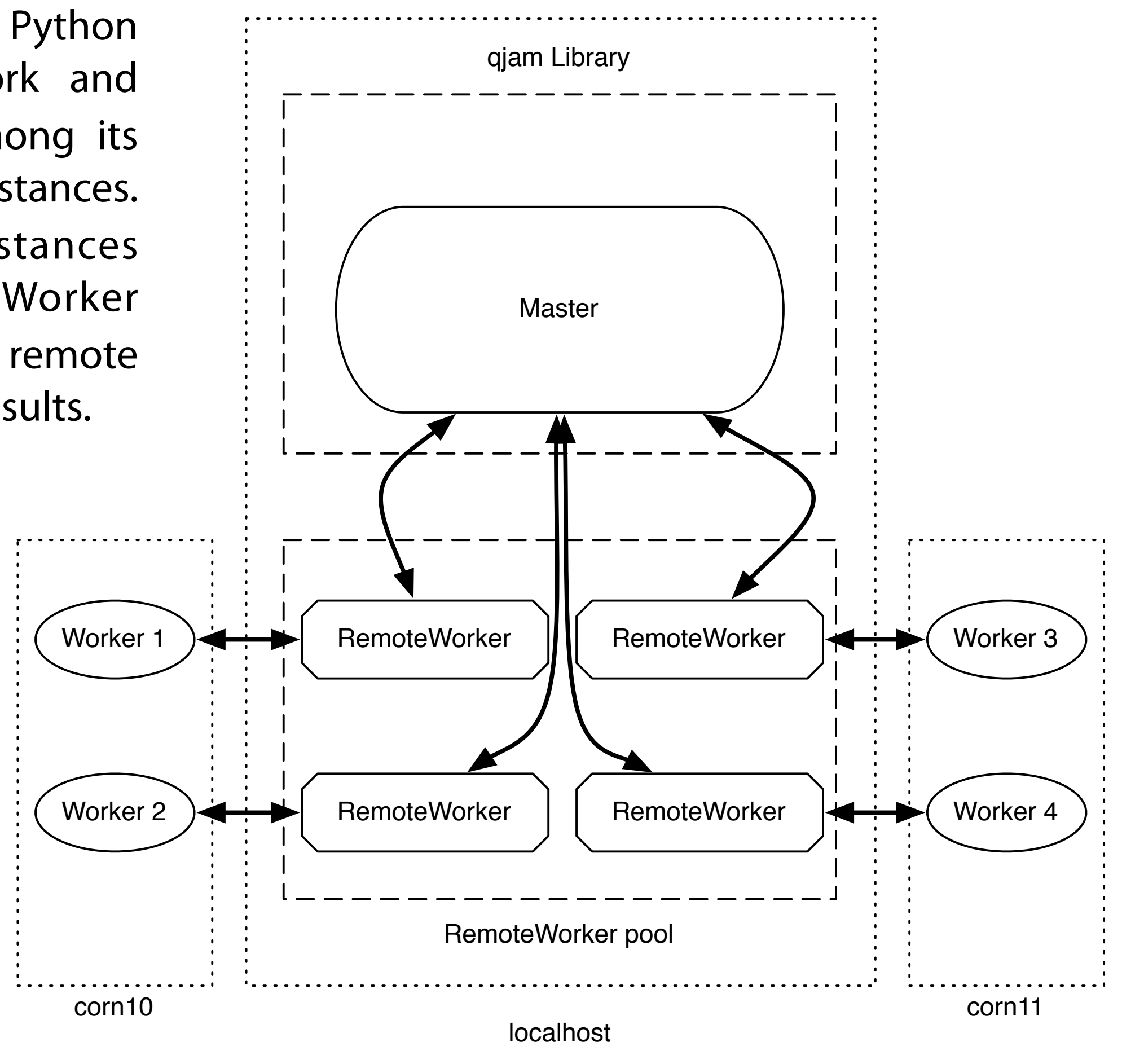

Fig 1. master controlling four RemoteWorkers with workers in two machines

Library Interface

```
workers = [RemoteWorker('corn10'), RemoteWorker('corn11')]
master = Master(workers)
result = master.run(map_module, params, dataset)
```

Benchmarks

We benchmarked qjam using a sparse autoencoder with L-BFGS. 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 sizes 1000, 10 000, and 100 000). We obtained the following results:

Table 1. Iteration Mean Time (seconds)

workers	1000	10 000	100 000
1	0.1458	0.7310	10.0282
2	0.1752	0.3321	4.6782
4	0.2634	0.3360	2.4858
8	0.5339	0.5251	1.8046
16	0.9969	1.0186	1.5862

Table 2. Total Running Time (seconds)

worker	1000	10 000	100 000
1	76 (1.00)	370 (1.00)	5030 (1.00)
2	92 (1.21)	170 (0.46)	2350 (0.47)
4	137 (1.80)	173 (0.46)	1253 (0.25)
8	275 (3.61)	270 (0.73)	914 (0.18)
16	544 (7.15)	529 (1.43)	703 (0.14)

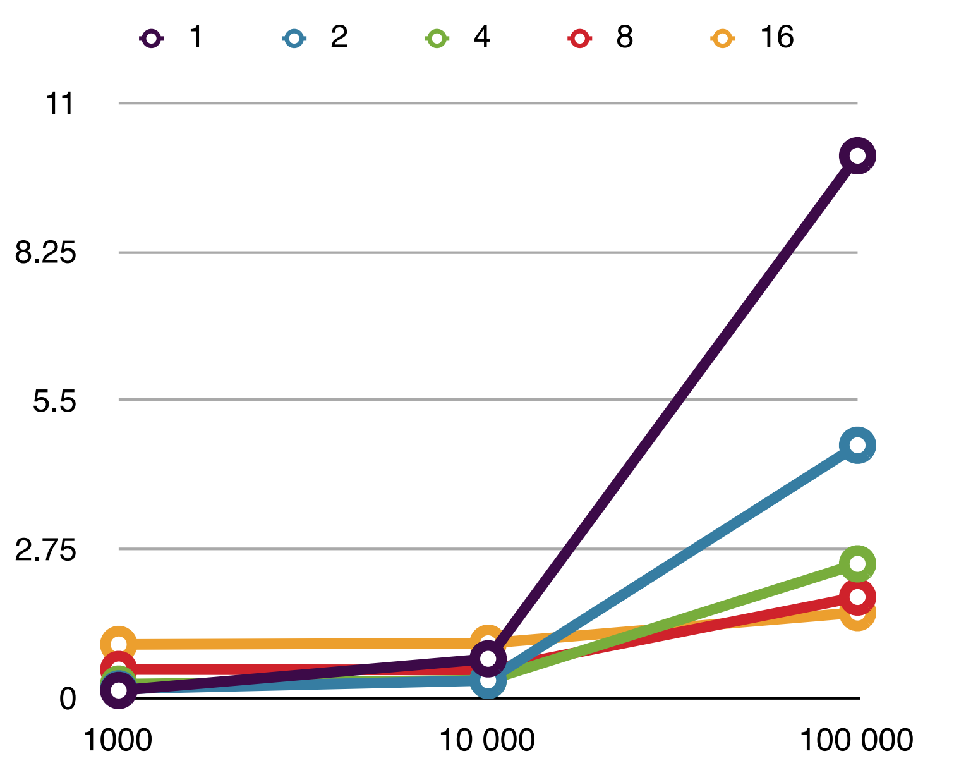

Fig 2. Iteration Mean Time (seconds vs patches)

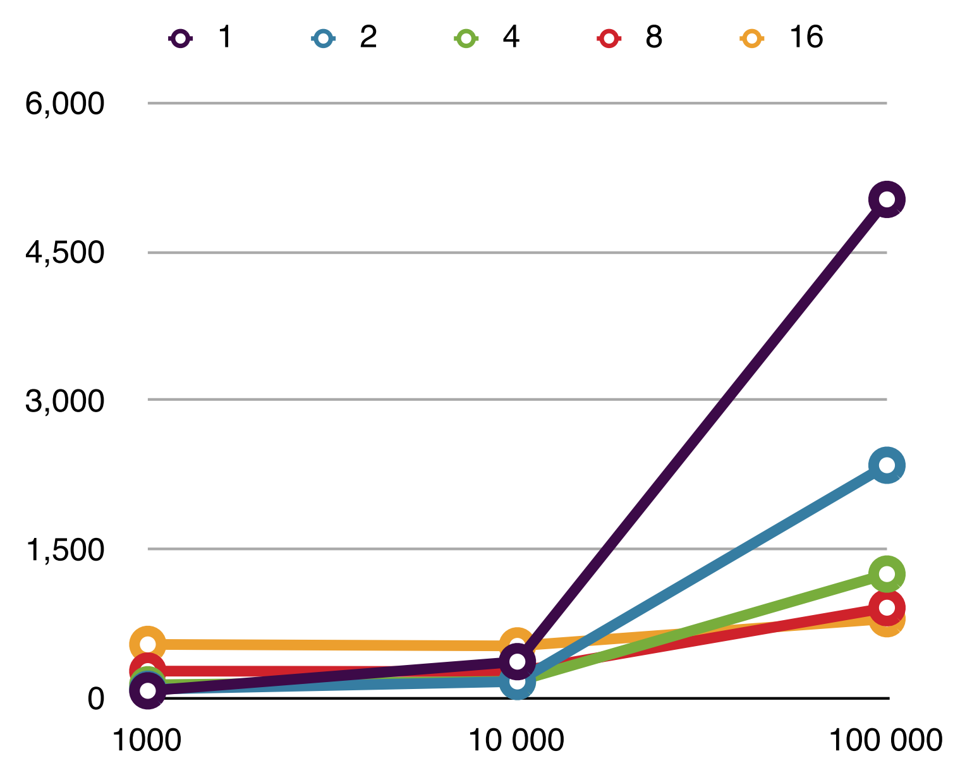

Fig 3. Total Running Time (seconds vs patches)

The running times show a remarkable speedup when using qjam with multiple workers. For our longest run, we saw a drop to under 14% of the single-core's running time. For big jobs (10,000, 100,000), qjam performs very well, beating the single-core every time. For non-intensive jobs (1000), the overhead of many workers can drive the performance beyond that of the single-core's implementation.

This leads to another observation: a particular number of workers seems to be suited for a particular job size: for the 10,000 patches runs, the best run was that with 2 workers. Though the others still performed better then the single core, they performed worse. For the largest job, though the 16 worker runtime was the lowest, the savings from 8 workers to 16 were small. This further confirms that the number of workers should be picked according to the job size, to minimize the overhead of distributing the job.

Example Program

summing a matrix

```
#!/usr/bin/env python2.6
'''
This is a very simple program to illustrate how to use the qjam framework. It
shows how to:

  * create a pool of workers
  * create a simple DataSet object out of a numpy matrix
  * start a qjam master
  * run a job on the worker pool and retrieve the result

All of the significant pieces are sufficiently commented to help you understand
how to write programs to use qjam.
'''
import logging
import os
import sys

import numpy

# Set up import path to qjam.
sys.path.append(os.path.join(os.path.dirname(sys.argv[0]), '..'))

from qjam.dataset import NumpyMatrixDataSet
from qjam.master.master import Master
from qjam.master.remote_worker import RemoteWorker

# sum_dataset is an example module that contains a function that is run on each
# piece of the dataset.
from examples import sum_dataset


def main():
  _fmt = '%(asctime)s [%(levelname)s] %(name)s: %(message)s'
  logging.basicConfig(level=logging.INFO, format=_fmt)

  # Worker hostnames are the commandline options. Each hostname specified on
  # the commandline will be bootstrapped and have a worker started on it.
  if len(sys.argv) < 2:
    print 'usage: %s <hostnames> [<hostname> ...]' % sys.argv[0]
    sys.exit(1)
  hostnames = sys.argv[1:]

  # Start the worker processes on the specified hostnames.
  workers = []
  for i, hostname in enumerate(hostnames):
    worker = RemoteWorker(hostname)
    workers.append(worker)
    print 'Started worker %d on %s' % (i, hostname)

  # Create a numpy matrix.
  matrix = numpy.matrix('1 2 3 4; 5 6 7 8; 9 10 11 12')

  print 'Created matrix:'
  print matrix
  print

  # Create a DataSet object from this matrix.
  #
  # DataSet objects are necessary so the framework can determine how to split
  # up the data into smaller chunks and distribute the chunks to the workers.
  dataset = NumpyMatrixDataSet(matrix)

  # Sum the matrix!
  master = Master(workers)
  params = None  # No parameters needed for this job.
  result = master.run(sum_dataset, params, dataset)

  # Print the result.
  print 'Result is: %d' % result


if __name__ == '__main__':
  main()
```

sum–matrix–example.py

```
import numpy


def sum_dataset(params, dataset):
  '''Returns the sum of all of the elements in the dataset. The dataset is
  assumed to be a numpy object (an array or matrix).'''
  return numpy.sum(dataset)


mapfunc = sum_dataset
```

sum_dataset.py