Parallelizing the Sparse Autoencoder

By Tom Jiang

Summary

The objective of this project is to take the Spase Autoencoder algorithm, as presented at: http://www.stanford.edu/class/archive/cs/cs294a/cs294a.1104/handouts.html , and develop methods for parallelizing, and possibly distributing the computation. In addition, the language Go, was chosen as the language to implement the algorithm, because it is a language designed with parallel computation as one of the goals.

One specific non-goal, however, is to match the c++ implementation of the algorithm. The c++ compiler is far more mature, and has a highly optimized matrix library. The primary goal is to evaluate the potential speed up of using parallel computation.

The findings in some respects are promising: Go can be nearly as fast as c++ or even faster depending on the circumstances. However, parallelizing the algorithm proved to be more difficult, as the basic algorithm is very difficult to make parallel efficiently. An improved algorithm with fewer data dependencies would allow for better speedups.

Background on Go

Go is a systems language developed by engineers at Google designed for high performance systems computing. One of its aims is to make it easy to perform parallel computation. In fact, the way to call any function in parallel is to use the statement: `go f()` instead of just `f()`. Running code using “go” is called running a goroutine.

In addition, the language uses channels as the primary means of synchronization instead of mutexes or other constructs, which makes it easy to synchronize progress. Instead of waiting on, and updating condition variables, which is unwieldy and error prone, the statement “<- c“ waits for something to arrive at the channel, while another thread calls “c <- 1” to signal it. It is a powerful generalization of semaphores. Mutexes are still supported, though I elected not to use them for this project.

Finally, the language has built-in support of closures, and combining these three features makes it very easy to perform any sort of parallel computation.

Challenges

The sparse autoencoder as given is very difficult to make parallel. The forwards and then backwards propagation of values prevents effective overlapping of the computation of the two layers. The next input cannot be processed until the current output is forward-computed, the error terms backpropogated, and finally the weight matrices updated.

However, as few as 2 synchronization points is possible per iteration:

1. Perform layer weight matrix updates, perform layer 2 activation output computation for the next input. Synchronize.
2. Compute layer 3 output computation based on layer 2 activation, compute error terms for layers 3 and 2, update weight matrix. Synchronize. (the other weight matrix is then updated at the next iteration).

The Go Implementation

To parallelize the algorithm, each of the functions that computes an aspect of the algorithm (e.g. the activation function, or the deltas) takes as parameters the start and end indices into the array on which it operations. e.g. if start and end are 8,16, then the function operates on nodes 8-15 (inclusive) in the layer it updates. Some care was taken to ensure things that should be updated simultaneously (the weight matrices, for example) stored copies of the previous values.

In addition, I call the smaller functions from larger functions that represent all the computation associated with a layer. e.g. ComputeLayer2 performs the l2 delta, l2 updates, and l2 activation calculations for a subset of the layer 2 nodes. Once done, the compute function sends a value to a channel to signal that it’s done.

I implemented two methods of parallizing the algorithm. The first is the recommended way of using goroutines -- the ParallelCompute functions call the Compute functions using “go”, and wait on the results. I actually opted to implement the model where the caller thread also works on a piece of the matrix, though there were very little difference between doing that and just calling “go” for everything.

The second method uses something similar to a thread pool -- there are several worker goroutines that run in a loop. Each of them attempts to read from a channel, which blocks until there is something to read. The main thread sends messages to each of the threads to begin the next piece of computation. Each worker is in charge of a subset of the layer 2 nodes the layer 3 nodes. It works on one or the other depending on the value it receives on its channel.

One thing to note is that the parallelization portion of the code is relatively easy to read. There are no confusing condition variables or mutexes, and in fact, both implementations share almost all the code, with the only difference being how the work is divided and executed.

Other Attempts

I also tried several other techniques, including breaking down the work into even smaller chunks, and calling go on all of them. I also attempted a more traditional thread pool model, where the main thread enqueues work on a single channel, and any thread can take that work from the shared channel. In all cases, it made the code run slower, which is not too surprising.

Breaking the work down into even smaller chunks or enqueuing lots of work performs well by keeping all processors busy when little no synchronization is necessary, and the computation is CPU bound. When synchronization is required however, it is best to carefully divide the work, and ensure that every processor finishes at about the same time.

The Go Compiler

There are currently two Go compilers available. First one “6g” is the recommended compiler, while the second one is “gccgo”, which is based on gcc. “6g” includes the Go runtime, which manages the lightweight goroutines, while gccgo creates a thread per goroutine, but can take advantage of the gcc optimizer. My results used both compilers, and in general, gccgo was faster despite the inefficiencies of using goroutines there.

Results

The compiler optimized C++ implementation using the Eigen matrix library is fast. Very fast, in fact. It can perform over 30,000 iterations per second with the default network size of 8x8 pixels, and 30 intermediate nodes. It is therefore not surprising that the Go implementation cannot beat this, and in fact, runs slower when multi-threaded than when not.

However, if we move to a very large network of 20x20 pixels, 400 intermediate nodes, Go performs much better. Perhaps surprisingly, the gccgo compiled version runs faster than the C++ implementation, even in single threaded mode. This may be due to my implementation in Go. I took care to make sure that all matrices are accessed row by row (and never column by column) to take maximum advantage of cache locality.

In either case, both implementations on both compilers beat the C++ version when a second thread is added, and the speedup is fairly good up to 4 threads. In terms of pure running time, the gccgo version is the fastest, even though the goroutine version of that code creates many threads. However, from a pure speedup perspective, it is not surprsing that while the gccgo thread pool version had the greatest speedup at 4 threads, the 6g goroutine version had the second best speedup.

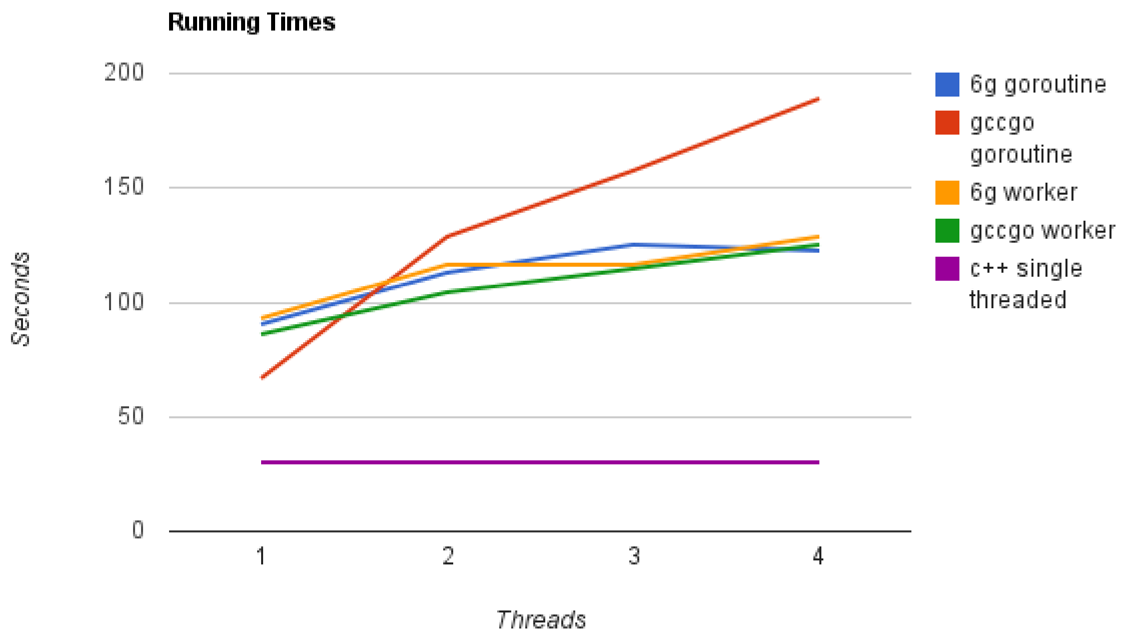
Conclusions

The Go language shows promise for this type of computation. It is both fast and easy to use for parallel computation. One thing that was not explored in this project was using Go for a truly distributed computation, since it was hard enough to parallelize on just one machine. However, by using channels, it would not be very difficult to implement a master and worker model of computation.

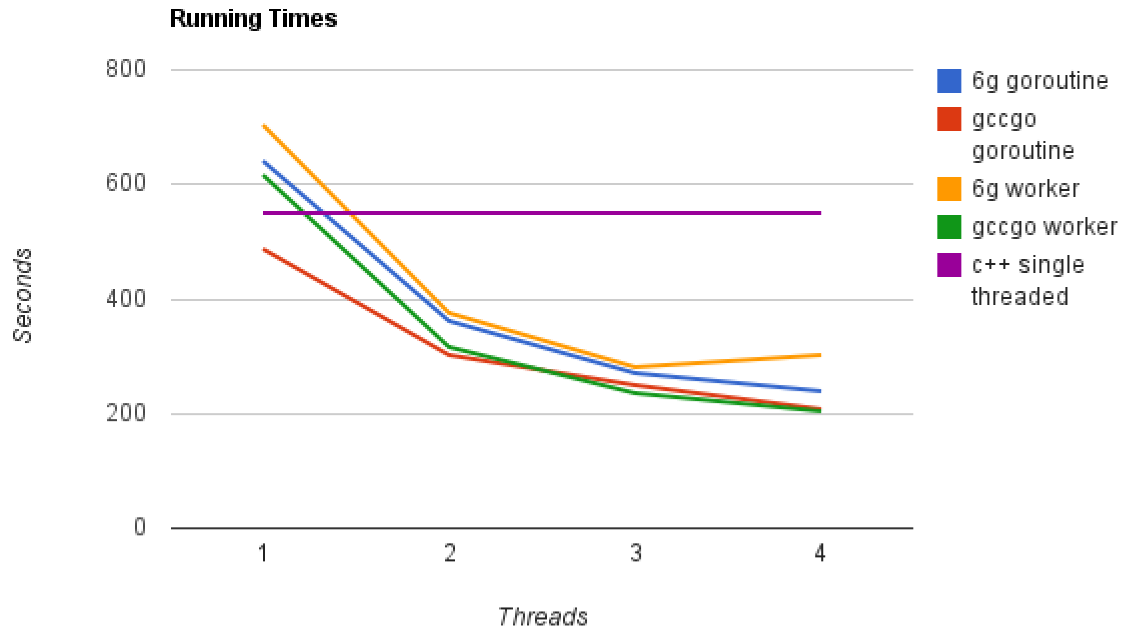
Finally, effort in developing an improved algorithm that reduces data dependencies would greatly improve the speedup and easily allow for massive amounts of parallelism.

Appendix

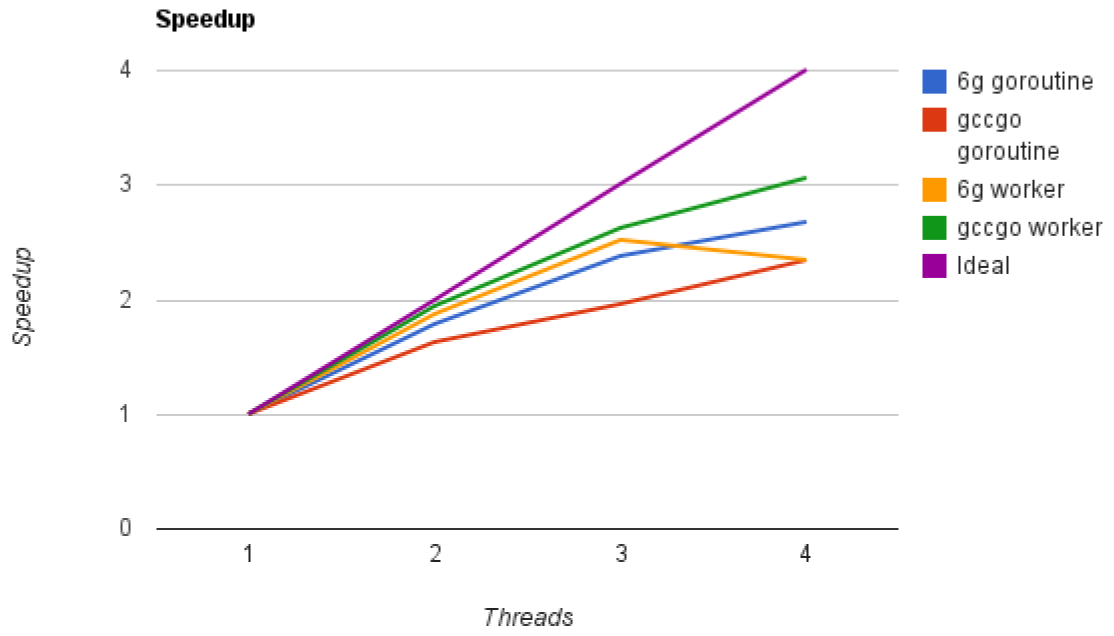
1,000,000 iterations on 8x8 input with 30 intermediate nodes



100,000 iterations on 20x20 input with 400 intermediate nodes



100,000 iterations on 20x20 input with 400 intermediate nodes


Speedup
Speedup
Threads
6g goroutine
gccgo goroutine
6g worker
gccgo worker
Ideal