A Comparison of Parallel Algorithms: Hogwild!, Shotgun and Distributed Averaging

David A. Leen
Department of Applied Mathematics
University of Washington
dleen@uw.edu

Brian D. Walker
Department of Computer Science & Engineering
University of Washington
walker7734@gmail.com

Abstract

Milestone progress: We implemented the Hogwild! algorithm and demonstrated an approximately linear speedup over the sequential stochastic gradient descent (SGD) algorithm. We used the click prediction dataset and its sparse search token features to test the algorithms. We discuss the many issues arising with parallel algorithms and their implementations.

1 Introduction

We study three parallel algorithms: Hogwild! [1], Shotgun [2] and distributed averaging [3]. Each approaches a similar problem in a distinct manner. Hogwild! and distributed averaging parallelize stochastic gradient descent (SGD) over samples, whereas Shotgun tackles the orthogonal problem of parallelizing L_1-regularized models over the features.

Hogwild! is just one particular realization of parallel SGD [1, 4]. The algorithm is intended for problems on the order of several terabytes of data. It can be run quite effectively on inexpensive multi-core systems taking advantage of the low latency and high throughput of the data being stored in RAM or a RAID disk setup close to the processors. In these systems the bottlenecks arise from synchronizing and locking operations on the threads reading and writing shared memory. Regularization can also be applied in parallel settings [5, 6].

MapReduce in contrast deals with hundreds and thousands of terabytes of data but is not well suited for online, iterative algorithms like SGD. MapReduce suffers from low throughputs due to fault tolerance and redundancy. Distributed averaging is a simple version of a distributed optimization algorithm [7, 6]. The method distributes the data samples evenly in group to some number of machines. Each machine performs a separate minimization of each group independently of one another. The estimates of the weights from each group are then averaged hence the name.

Shotgun provides a parallel solution to the high-dimensional problem with a large number of features using L_1-regularization [8]. In cases like this the dimensionality of the problem often dwarfs the number of samples. Once again the method involves taking a sequential algorithm, coordinate descent and simply parallelizes it, this time over features. The algorithm makes the coordinate updates in parallel to provide a speedup.

2 Method

The weight vector w for logistic regression is updated according to the usual SGD equation:

$$w_i^{t+1} \leftarrow w_i^t + \eta \left[y^t - \frac{\exp\left(w^t \cdot x^t\right)}{1 + \exp\left(w^t \cdot x^t\right)} \right] x_i^t. \tag{1}$$

2.1 Proof of concept

The minimal example demonstrating the Hogwild! concept in practice for a simple mapping is (in Scala for brevity):

```
val weights = collection.mutable.Map[Int, Double]() // create a mapping
for (i <- data.keys) weights.put(i, 0.0) // initialize key values to 0.0
for (k <- samples.par, i <- feature.keys) weights(i) += gradient // update
```

This parallel version of SGD (using the `.par` method) produces non-deterministic values for the weights as expected. This is due to a race condition caused by threads overwriting one another. The Hogwild! algorithm relies on the sparsity of the data to ensure that updates do not operate simultaneously on the same keys. It relies on the same sparsity to parallelize the sequential algorithm.

2.2 Algorithm

Algorithm 1 Hogwild! update step for a single processor

```
loop
    Randomly permute data
    Read current state of features $x^t$
    for i in $x^t$.keys do
        $w_i^{t+1} \leftarrow w_i^t - \eta x_i^t \nabla L(\mathbf{w}, \mathbf{x}, \mathbf{y})$
    end for
end loop
```

The component wise addition of elements from the weight vector is assumed to be atomic. In practice floats are not atomic and require extra care. See section 3.

3 Challenges and issues

3.1 Parallel and concurrent programming

Parallel programming is hard. The Hogwild! solution is to ignore all the major issues associated with parallelism. The Hogwild! paper discusses a compromise between full synchronization and no locking at all which they call AIG. AIG puts a lock on only the elements that are being updated inside the for loop in algorithm 1. This is essentially what ConcurrentHashMap in Java does. It often can be difficult to determine which regime we are operating in without resorting to a closer-to-the-hardware language like C.

3.2 Storing data in memory

The Hogwild! algorithm relies on extremely fast throughput of data from memory or disk. Reading from disk is not fast enough to supply data as needed by a 4-6 core machine. This could be possible with a RAID setup but at the time this was not available. The Hogwild! paper uses sophisticated caching and memory management to process terabyte sized data on a single machine.

Our current solution is to load and parse all the data into RAM. 8-16GB of RAM is fairly standard these days, which is sufficient for the click prediction set, but it is not enough for terabyte data sets. We need to investigate how to break data sets into manageable chunks while still sampling uniformly as needed by the algorithms.

3.3 Non-sparse data

An immediate issue was whether Hogwild! can work efficiently on data with sparse and non-sparse components? The click prediction data has several dense values including age and gender. They are updated on every iteration. The tokens however are very sparse. Can we apply the Hogwild! algorithm in this case without ignoring the dense data? This is still unclear.

4 Results

Initial results ranging from one to four CPU cores. During each run the same number of updates over the data occurs with the workload evenly distributed across each core. We fall slightly short of the optimal speedup of four, figure 4(a). The Hogwild! algorithm has a slightly worse RMSE as expected, figure 4(b).

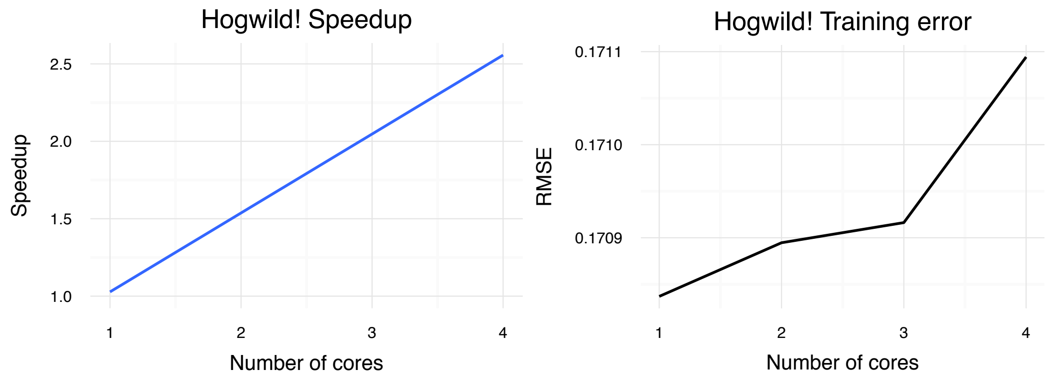

Figure 1: (a) The observed speedup from experiment. The speedup with four processors is approximately 2.55. (b) The observed root mean squared error growth with number of cores from experiment.

5 Further work

The next step is to implement Shotgun and distributed averaging for (sparse) logistic regression. We will compare all three and confirm where each performs optimally. We have the baseline sequential predictions for each data set against which we can judge speedup, convergence and error rates. We will then expand to linear regression and support vector machines and time permitting matrix completions. The data sets will be expanded from click prediction. We will continue to investigate the issue of non-sparse data contaminating the Hogwild! algorithm and whether one can perform (delayed) regularization during Hogwild!

6 To Do List

1. Discuss theory behind averaging. When will it work?
2. Theory behind Hogwild.

References

[1] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *arXiv preprint arXiv:1106.5730*, 2011.

[2] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

[3] Yuchen Zhang, John C Duchi, and Martin Wainwright. Comunication-efficient algorithms for statistical optimization. *arXiv preprint arXiv:1209.4129*, 2012.

[4] Martin Zinkevich, Markus Weimer, Alex Smola, and Lihong Li. Parallelized stochastic gradient descent. *Advances in Neural Information Processing Systems*, 23(23):1–9, 2010.

[5] John Langford, Alexander Smola, and Martin Zinkevich. Slow learners are fast. *arXiv preprint arXiv:0911.0491*, 2009.

[6] Alekh Agarwal and John C Duchi. Distributed delayed stochastic optimization. *arXiv preprint arXiv:1104.5525*, 2011.

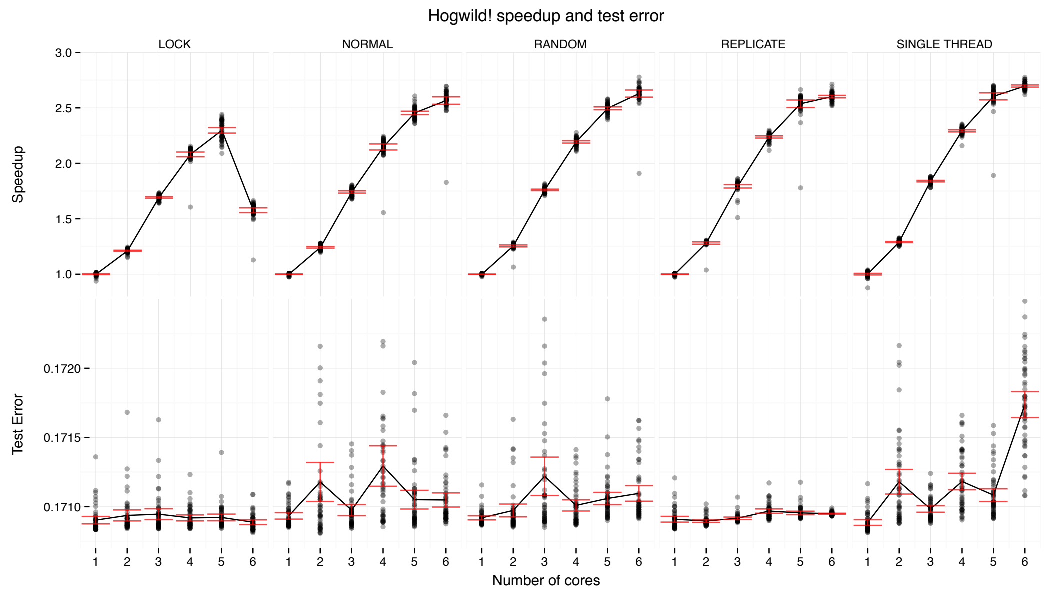

Figure 2: (a) The observed speedup from experiment. The speedup with four processors is approximately 2.55. (b) The observed root mean squared error growth with number of cores from experiment.

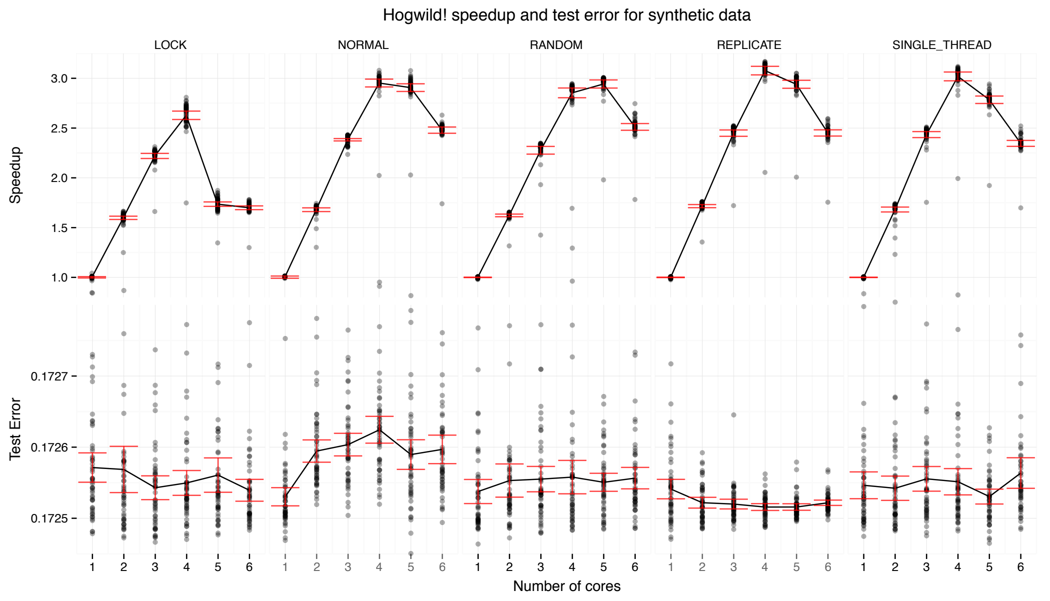

Figure 3: (a) The observed speedup from experiment. The speedup with four processors is approximately 2.55. (b) The observed root mean squared error growth with number of cores from experiment.

[7] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *The Journal of Machine Learning Research*, 13:165–202, 2012.

[8] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

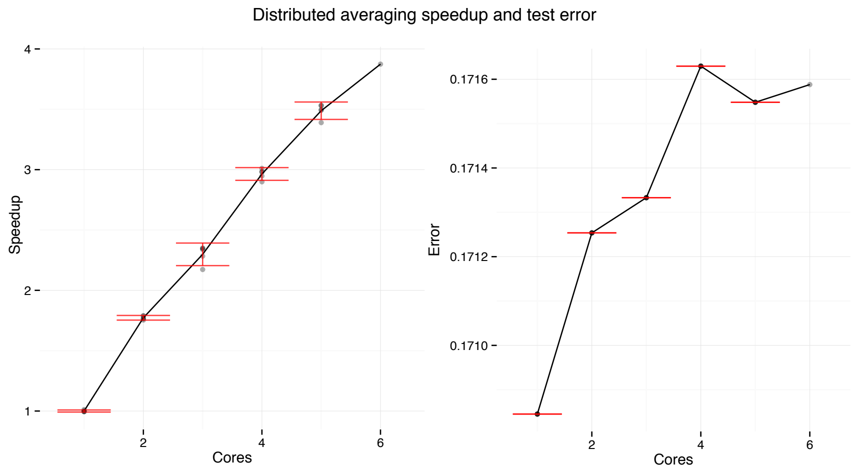

Figure 4: (a) The observed speedup from experiment. The speedup with four processors is approximately 2.55. (b) The observed root mean squared error growth with number of cores from experiment.