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

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 implement and demonstrate the close to optimal speedup of the Distributed Averaging method. Several strategies are detailed for dealing with mixed-sparsity data i.e. data that has both sparse and dense components. We implement a potentially novel combination of Hogwild! and Distributed Averaging. We demonstrate the favorable trade-off between speedup and test error that this method gives on mixed-sparsity data. We finally test these algorithms on some synthetic data derived from the click prediction data set, where the number of dense user features and sparse tokens is increased.

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.

We study these algorithms in the context of the click prediction dataset. We first briefly discuss the algorithms, the challenges associated with parallel programming, then specific challenges associated with the click prediction dataset. Finally we present our results and our “replicate” strategy for mixed-sparsity data.

2 Method

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

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

2.1 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.

Distributed Averaging takes the opposite approach avoiding the issues of parallelism altogether. It turns the problem into a data parallel one by dividing the data into a number of chunks, one per machine or processor and running SGD in parallel on each one independent of the others. The weights are then averaged.

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 issue is answered in the next section.

4 Mixed-sparsity data

The click prediction data provides an interesting scenario in which to test Hogwild! The click prediction dataset is comprised of user features and sparse tokens. We call this combination of sparse and non-sparse data “mixed-sparsity”. We initially pretend the data is sparse and run Hogwild! before discussing strategies for improving the error while maintaining the speedup from the extra processors.

As expected not dealing with the non-sparse data allowed for the expected speedup but as more processor cores were introduced the error spiked as there were more data collisions with the non-sparse features. We took this as a good opportunity to explore methods to handle the non-sparse data and still obtain a substantial speedup from the Hogwild! algorithm. We explored five methods to handle the non-sparse features.

4.1 Strategies for mixed-sparsity data

We outline several strategies that we implement and test on the click prediction dataset:

1. *Normal*: this strategy simply treats all of the data as sparse regardless as to whether it truly is. Therefore, we run Hogwild! with absolutely no locking on any of the data features and let data collisions occur without any attempt to prevent them. This is less of a strategy for dealing with non-sparse data as it is a baseline with which we can compare our other strategies.
2. *Locking*: the most obvious strategy is to lock the data to prevent data collisions. However, the whole point of the Hogwild! algorithm is to gain a near linear speedup by not using locking. The introduction of locks forces a portion of the algorithm to run sequentially and thus has the potential to kill our speedup. Since only a small portion of our data is non-sparse we only lock during the update of these non-sparse features and allow the remaining sparse data to continue lock-free.
3. *Random*: this strategy attempts to prevent data collisions by giving all the threads an equal probability to update the non-sparse weights. To implement this strategy we use a random number generator that produces a number from zero to the number of current threads. If that threads generated number is zero then it updates the non-sparse data. This allows the data to be updated once, on average, for each iteration. Data collisions are still allowed to occur and no locking is done.
4. *Single thread*: to completely prevent any chance of data collisions this strategy randomly chooses one thread at the start of the algorithm to be the only thread to update the non-sparse weights. This strategy allows us to remain lock-free and have a zero chance of data collisions. This comes at the cost of updating the non-sparse data less frequently.
5. *Replicate*: this strategy attempts to eliminate the problem of data collisions while still allowing the non-sparse weights to be based on more information than is provided by just a single thread. Instead of assigning a single thread or randomizing to determine which thread can update the non-sparse data, we allow all of them to update their own copies of the non-sparse data. Each thread is given its own copy and updates the non-sparse data independently and thus without locks. The sparse data still remains in shared memory and the updates to these features remains unchanged from the original Hogwild! algorithm. When the algorithm has completed we then average the non-sparse data from each of the threads back into shared memory. This allows us to tally the information from all of the threads with the hope of achieving more accurate weights.

5 Results

Initial results ranging from one to four CPU cores (figure 1). 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 1(a). The Hogwild! algorithm has a slightly worse RMSE as expected, figure 1(b).

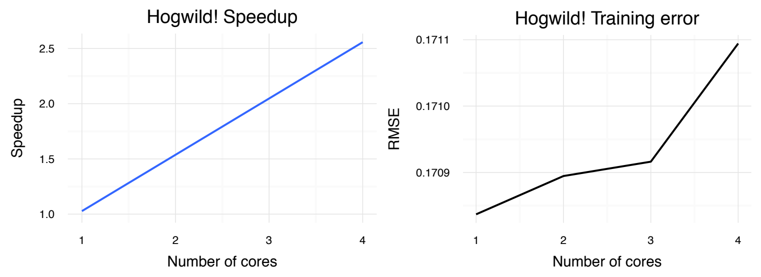

Figure 1: (a) The observed speedup from experiment. The speedup with four processors is approximately 2.55. (b) The observed root mean squared error growth with number of cores from experiment.

Distributed averaging (figure 2) achieves near optimal speedup as each processor runs independently of one another and do not need to communicate. The error suffers as each processor only trains on a fraction of the data.

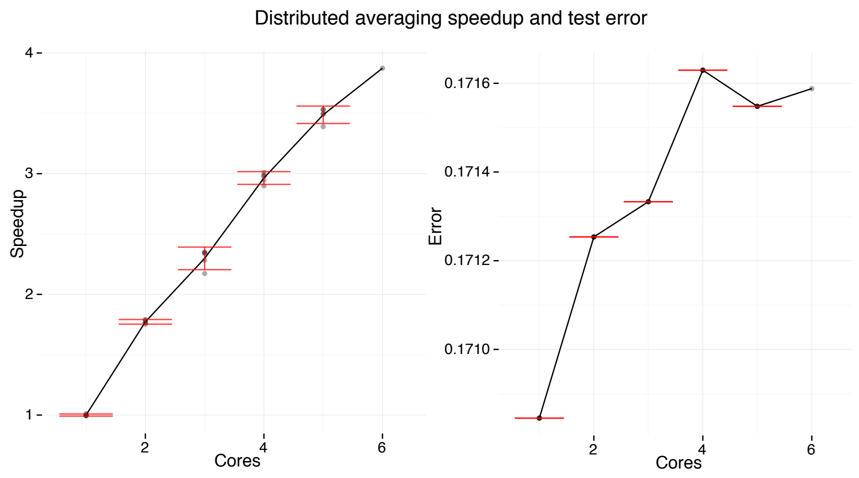

Figure 2: Distributed averaging achieves nearly optimal speedup as there is no communication overhead between the processors. The SGD is computed independently for each processor. The error increases drastically as the data is split more and more and each processor trains the weights on less and less data.

We now discuss the results of our strategies for dealing with mixed-sparsity data. Figure 3 displays the results of our numerical experiments. The main result is that the replication technique out-performs everything else in terms of speedup-error tradeoff. It combines the the speedup and advantages of Hogwild! with the batch processing of distributed averaging for the dense data components.

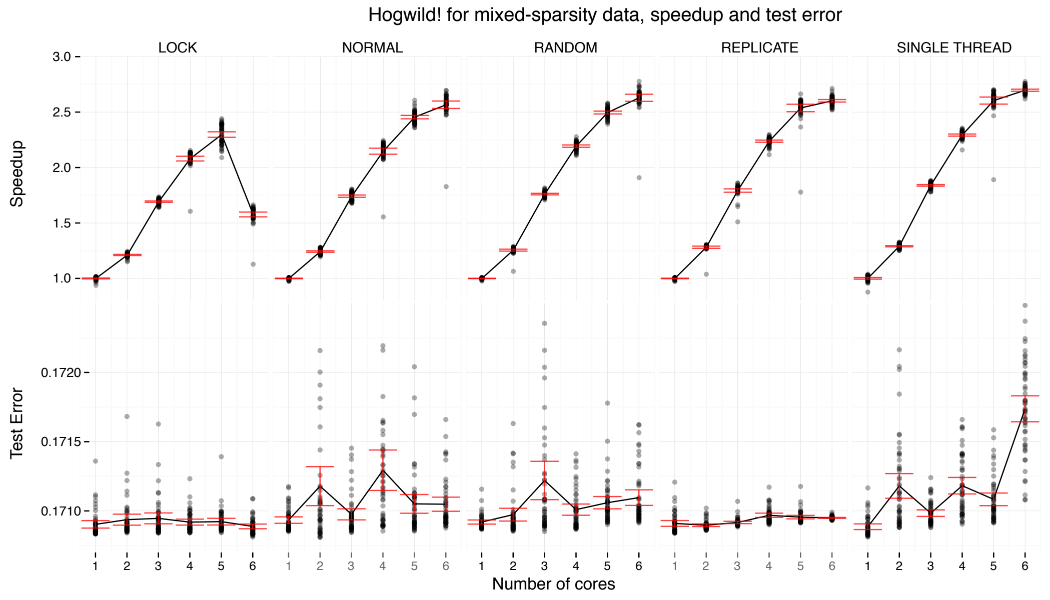

Figure 3: A comparison of our strategies for dealing with mixed-sparsity data. Lock suffers greatly in speedup as the number of processors increases from one to six. This is because of processors waiting for locked resources. The test error is constant in the number of processors and is consistently the best among all the algorithms. Normal and random have similar speedup performances and have high variances in the errors which are larger than the locked and replicate case. Random does as well as treating the data as sparse and achieves better errors. Single thread performs the worst as it trains the weights on only a fraction of the data. Replicate is our novel attempt at the mixed-sparsity problem. The speedup is as good as the other cases and the error is almost as good as the locking case. This is both a surprising and promising result.

We generate synthetic data (figure 4) derived from the click prediction data by increasing the size of the dense user features and increasing the number of sparse features. This causes the locking algorithm even more problems. The other algorithms see an increase in speedup as the processors spend relatively more time calculating the gradient compared to communication and memory overhead.

Plotting the test error against the wall time (figure **??**) clearly demonstrates the strengths of the replicate algorithm.

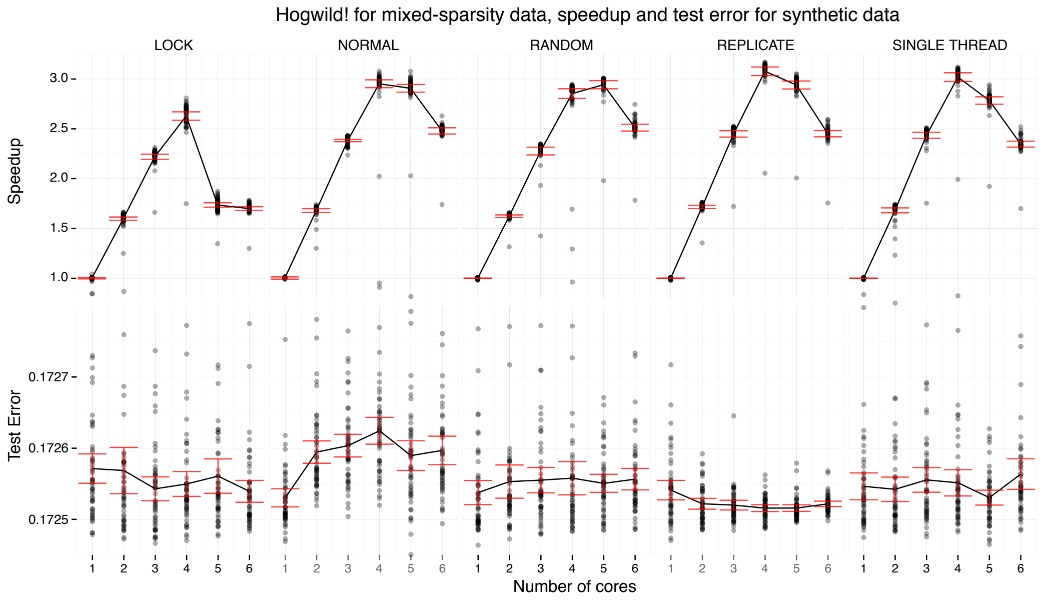

Figure 4: Synthetic data is created from the click prediction data by doubling the number of user features from five to ten and increasing the number of sparse features per sample to 50. This increases the amount of work for each processor and hence we see a greater speedup. The lock strategy suffers the most due to scheduling of the greater workload. Replicate amazingly has the best error and greatest speedup. The test error variance is also very small compared to the others. This was a surprisingly good result. Random and normal suffer from the worst errors as expected.

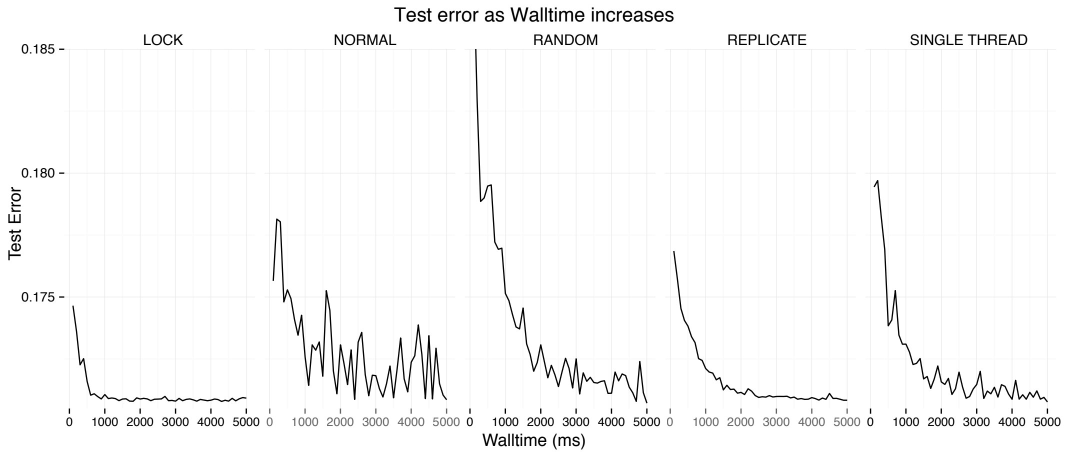

Figure 5: The test error as a function of wall time. Locking achieves the best error among all the algorithms as expected. This suffers greatly in speedup performance however. Normal is quite noisy as we are treating the data as sparse and the processors will frequently overwrite one another. Random has the worst error out of all of them. Replicate is again surprising as it performs almost as well. Single thread performs about as well as normal.

6 Further work

Further work involves a theoretical analysis of the error bounds of our replicate algorithm.

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.

[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.