# Final Project: Hogwild!

Alex Renda (adr74), Matthew Li (ml927), Matt Gharrity (mjg355)

December 7, 2017

## 1 Abstract

We focused on parallelizing machine learning optimization algorithms. Our proposal was to implement Hogwild! [1], but our project idea lends itself well to other extensions (such as Buckwild! [2]). Our interest in these two algorithms is inspired by Professor Chris De Sa.

We implemented Hogwild! in C++ and explored its performance trade-offs on the MNIST image dataset. We were interested in adjusting for learning rate as well as determining the quality of the result in the form of test error and training loss. We were able to replicate the performance improvements specified in the Hogwild! paper. We also tested Hogwild! with slightly different synchronization assumptions.

To evaluate our results, we implemented Hogwild! in several steps. First, we implemented a fast serial SGD using BLAS calls, as a baseline to which to compare the parallel versions. Next, we implemented various versions of Hogwild! and compared it to serial SGD, checking both iteration run time and convergence rate, to evaluate which optimizations resulted in the highest speedup at the lowest accuracy cost. We run each of our implementations on the MNIST dataset [3], and vary the number of threads to test the scalability of our implementation.

## 2 Background

Stochastic Gradient Descent (SGD) is an optimization algorithm used in machine learning for finding the best parameters to minimize a loss function. In standard gradient descent, this is done by taking small steps in the direction of the gradient of the loss function. SGD does the same thing, but instead of computing the gradient of the whole training data set it only uses a small sample each time. Generally, SGD is faster than gradient descent. However, the primary issue with SGD is that it is very difficult to parallelize. SGD is inherently sequential because each update depends on the one before.

Hogwild! is an algorithm for optimizing stochastic gradient descent by parallelizing it. As mentioned, SGD is notoriously difficult to parallelize due to it's inherently sequential nature. The Hogwild! paper has a simple solution to this issue: run SGD in parallel without locks. This scheme relies on the assumption that weight vector updates are sparse so that updates don't constantly overwrite each other. Using exponential backoff in gradient step size, it is provable that there is an upper bound on the convergence rate.

Buckwild! is another algorithm similar to Hogwild! that farther improves performance by using lower-precision arithmetic. Buckwild! shows that lower precision arithmetic introduces additional noise into the system due to round-off error, but does not affect convergence rates. Using lower precision arithmetic by rounding to an integer allows Buckwild! to take advantage of SIMD instructions for integers on CPUs, further improving performance.

# 3 Design

At a very high level, our project pipeline is a typical application of multinomial logistic regression:

- First we parse in our training set and do a bit of preprocessing.
- Afterwards, we train our logistic regression using the chosen optimization algorithm (serial SGD, or various implementations of HOGWILD!)
- After training a specific amount of iterations, we print out the training loss as well as the results on the test set. Afterwards, we can either exit or continue training for more iterations.

We will describe our implementation and design in more detail in the subsections below. One thing not mentioned is that we wrote our own timing harness. Specifically, our "real time" numbers are obtained by using "omp_get_wtime". As long as at least one thread is running SGD then we include it in the real time. Note that for timing, we only include the time spent actually doing SGD. The timer is placed at the beginning and end of our SGD iterations loop. We do not include the time spent doing operations such as computing training loss or test error, since these are reported to the console only for logging purposes.

## 3.1 MNIST Dataset

For our experiments, we chose to use the MNIST Handwritten digits as our test case. The data we used can be downloaded from `http://yann.lecun.com/exdb/mnist/`. Most of the preprocessing required for image recognition has already been done on the data set. Additionally, it comes split into a training and testing set.

For features, each image comes with a 784 feature vector representing the values in the 28 by 28 image. We normalize all the features in the feature vector to be between 0 and 1. Additionally we change the format of the labels (which are initially just the number the image corresponds to) to be a one-hot vector. This allows us to use multinomial logistic regression.

## 3.2 Serial SGD

We spent a lot of time implementing and tuning serial SGD in order to provide a fair baseline for HOGWILD!. We use BLAS calls to help with this, including `cblas_sgemm`, `cblas_sasum`, `cblas_sger`, and `cblas_saxpby`. We were also careful to align data, turn on relevant `icc` compiler flags, use the `restrict` keyword where applicable, and read through the `icc` vectorization reports.

## 3.3 HOGWILD!

We implemented HOGWILD! by using OpenMP to parallelize our serial SGD implementation. There are some details to our implementation that we will describe below.

In addition to the baseline HOGWILD!, we also created a few variants with different synchronization assumptions. These different implementations are explained below.

### 3.3.1 HOGWILD! Algorithms

Algorithm 1 is our formalization of the original HOGWILD! algorithm presented in the paper, with some added atomic sections to make the synchronization more explicit. In algorithm 1, $w$ is the weight vector that is being trained and $\alpha$ is the learning rate of SGD.

**Algorithm 1:** Origional HOGWILD! SGD Thread Loop

```
Data: $w$, $\alpha$
while <iterations> do
    memcpy($local_w$, $w$)
    $dw \leftarrow \nabla f(local_w)$ // compute gradient using $local_w$
    Atomic: // apply weight update atomically, can be done componentwise
        $w \leftarrow w - \alpha * dw$
    end
end
```

As you can see in algorithm 1, some synchronization and overhead is still required. First you need to copy the weight vector so that the weight vector does not change in the middle of gradient calculation. This is achieved by copying the weight vector to a local variable. The atomic add on each component of the weight vector ensures that (1) all weight updates are eventually seen by other processors, and (2) weight updates are not lost when two threads try to write to the same entry at the same time.

The first thing we wanted to test was whether we are able to avoid copying the weight vector to a local variable, as seen in algorithm 2. We were unsure how computing the gradient would react to having the underlying weights possibly change during the middle of computation, so we thought it might be interesting to observe the results.

**Algorithm 2:** No Copying

```
Data: $w$, $\alpha$
while <iterations> do
    $dw \leftarrow \nabla f(w)$ // compute gradient using $w$
    Atomic: // apply weight update atomically, can be done componentwise
        $w \leftarrow w - \alpha * dw$
    end
end
```

We also wanted to try removing the atomic weight update, as seen in algorithm 3. This would allow us to use vectorized instructions to perform the weight update rather than atomic increment instructions. This change almost certainly leads to more overwriting, but our hope is that computing the gradient takes sufficiently longer than updating the vector that HOGWILD! still works.

**Algorithm 3:** No Copying and No Atomic Update

```
Data: $w$, $\alpha$
while <iterations> do
    $dw \leftarrow \nabla f(w)$ // compute gradient using $w$
    flush
    $w \leftarrow w - \alpha * dw$
    flush
end
```

### 3.3.2 Miscellaneous implementation details

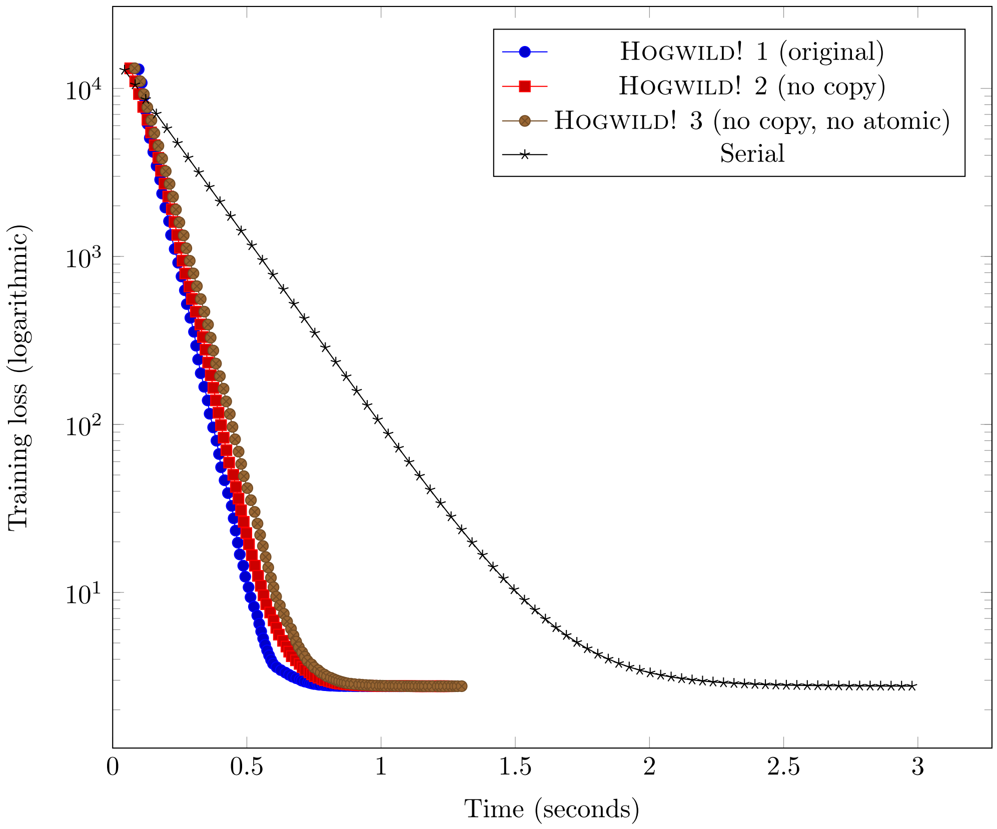
- Atomic adds are done using `#pragma omp atomic`.
- We use macros to switch between the different HOGWILD! implementations explained above.
- We generate the SGD batch at the beginning of each iteration using Fisher-Yates shuffle.
- The timing harness surrounds the parallel for-loop, but excludes the time needed to print information to the console.

## 4 Experiments

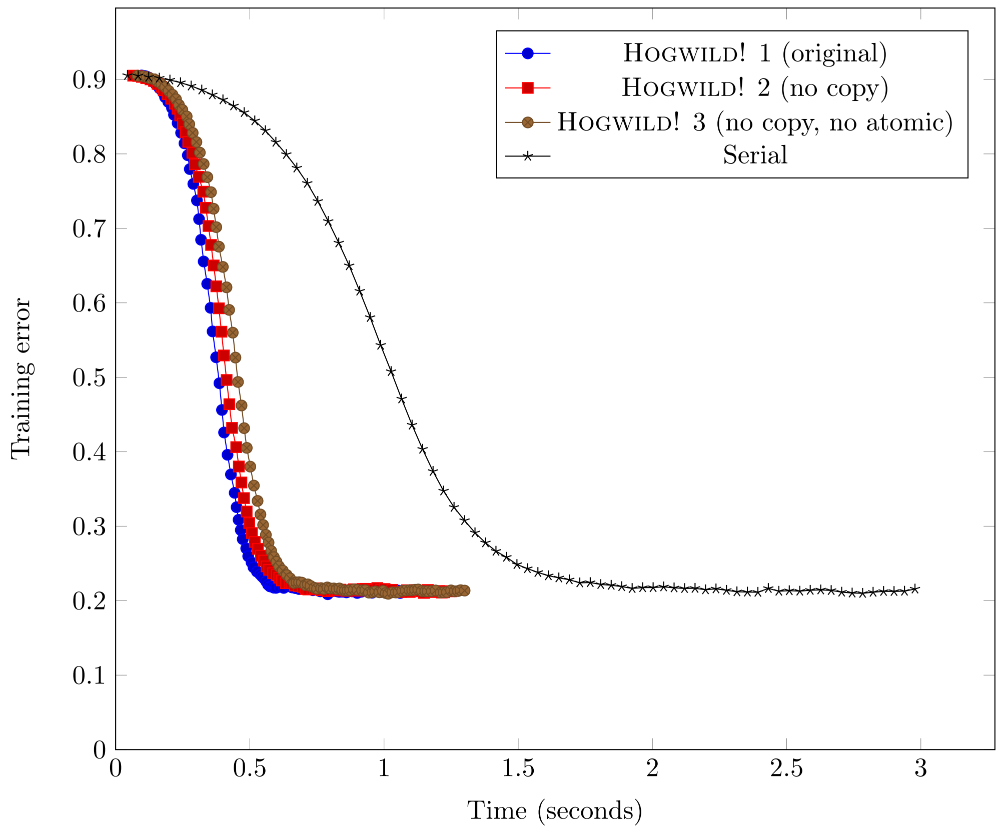
We ran all experiments on the Totient cluster with the `icc` compiler. We ran stochastic gradient descent for 100000 iterations, and for every 1000 iterations we logged training loss, error, and total time.

### 4.1 Wall-clock comparison

The following figure shows training loss as a function of wall-clock time (excluding the time taken to log information to the console). OMP defaulted to using 24 threads for HOGWILD!. We see an approximately 3x speedup over the serial implementation, and slight (but consistent) differences among the three HOGWILD! variations. Interestingly, the original HOGWILD! algorithm (using local weight vector copies and atomic updates) does the best.

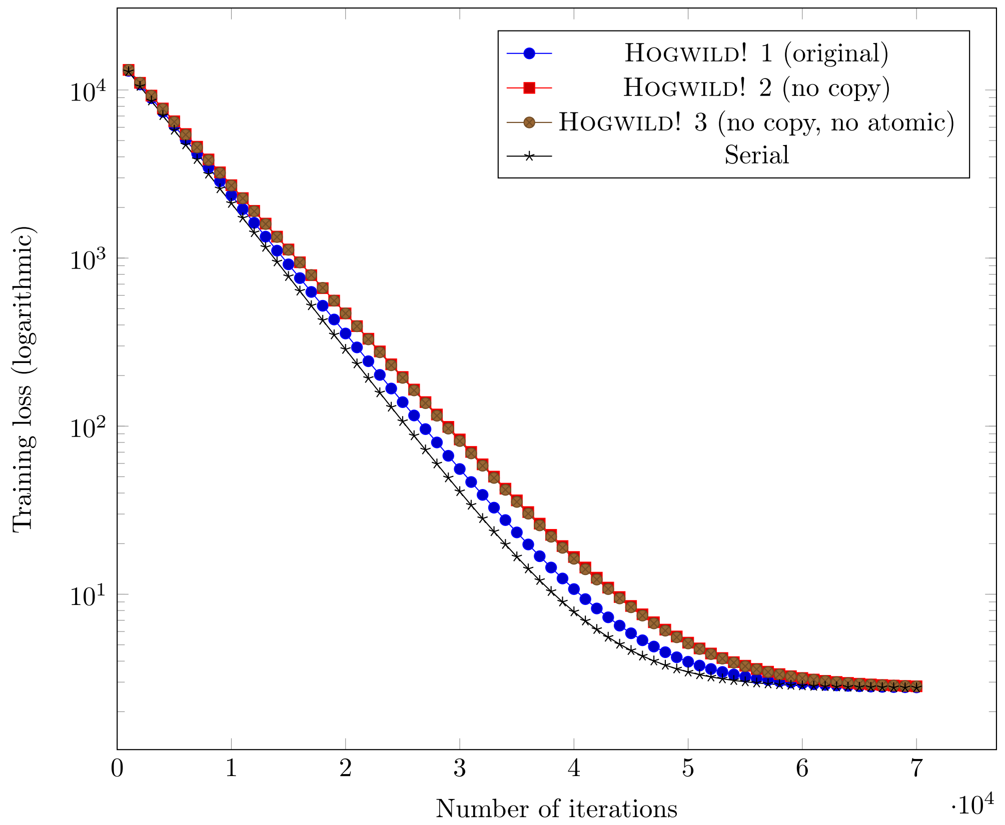


To sanity-check correctness, we also verify with the figure below that the decrease in training loss corresponds to a decrease in training error as well.



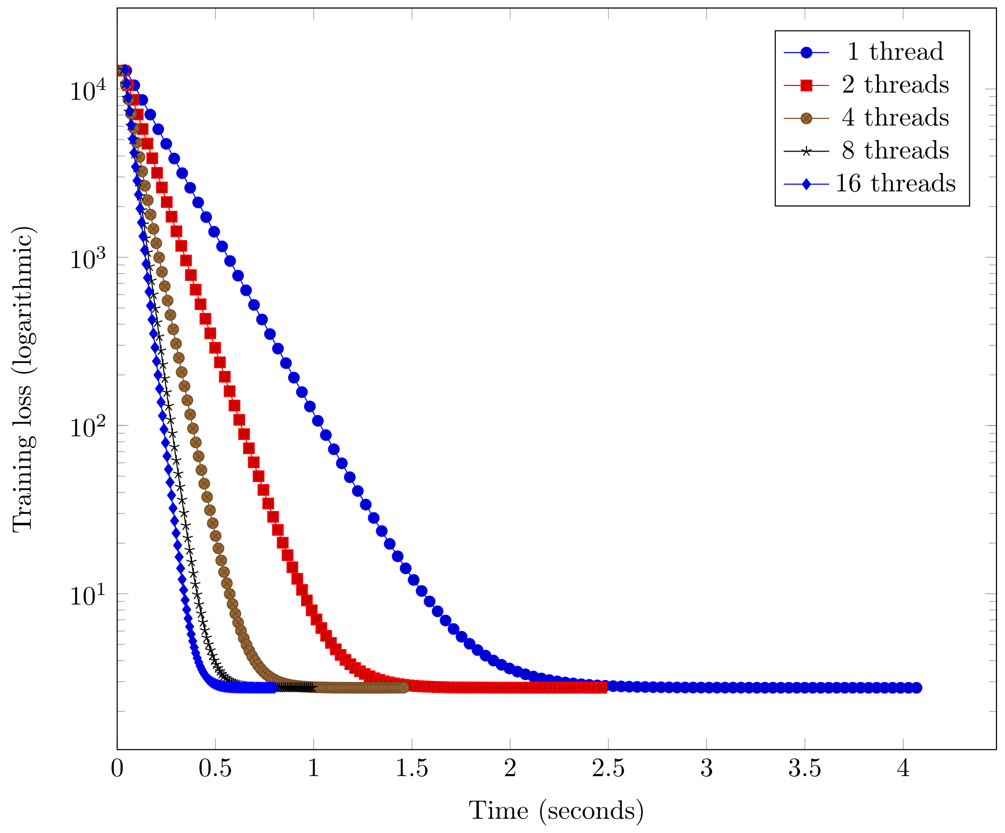
## 4.2 Iteration quality

To test the quality of each iteration in optimizing the loss function, we plot training loss as a function of iteration count. As expected, serial stochastic gradient descent has slightly higher iteration quality, presumably since it has no lost updates due to race conditions. We also see that atomic adds to weight vector entries increases iteration quality. On the other hand, making a temporary local copy of the weight vector for each iteration has little effect.



## 4.3 Scaling

Finally, we show the performance of the first HOGWILD! variant as a function of the number of threads used. We see diminishing returns as the number of threads increases: switching from 1 to 2 threads cuts the time almost in half, while switching from 8 to 16 threads cuts the time down by only about a quarter.



## 5 Future Work: Buckwild!

Implementing Buckwild! is a natural extension of our project. Unfortunately, we were unable to complete it within the amount of time we had for this project.

Originally we planned to use `https://github.com/google/gemmlowp`, which is a low-precision GEMM library from Google in order to implement Buckwild!. We ran into some difficulty understanding the library as well as converting our codebase to be low precision compatible. While we do believe that these difficulties could have been resolved, we were unable to do so in the time we had. Instead, we chose to focus on providing a more complete analysis of Hogwild!.

## 6 Conclusion

We implemented a highly tuned version of stochastic gradient descent built on BLAS calls. We then implemented the Hogwild! algorithm—a parallelized version of stochastic gradient descent—and experimented with the atomicity of weight vector reads and updates. We showed empirically that Hogwild! significantly outperforms serial stochastic gradient descent, and that it scales reasonably well with the number of threads. This helps confirms the results in the original Hogwild! paper.

Thank you for a great semester!

## References

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

[2] Christopher De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. Taming the wild: A unified analysis of hogwild!-style algorithms. In *NIPS*, 2015.

[3] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The mnist database of handwritten digits.