AN EMPIRICAL ANALYSIS OF ASYNCHRONOUS STOCHASTIC GRADIENT DESCENT METHODS

A PREPRINT

Sriram Ravula *
Department of Electrical and Computer Engineering
The University of Texas at Austin
sriram.ravula@utexas.edu

Michael Stecklein
Department of Electrical and Computer Engineering
The University of Texas at Austin
michaelstecklein@utexas.edu

Richard Cao
Department of Electrical and Computer Engineering
The University of Texas at Austin
richardcao@utexas.edu

September 17, 2019

ABSTRACT

This paper investigates the effects of sparsity on the speedup and convergence of asynchronous stochastic gradient descent algorithms. We implement the HOGWILD! algorithm [1] and our own example algorithms in C, along with an analytical wrapper. The key takeaway from our implementation is that the structure of an asynchronous program has a huge effect on its ability to parallelize well across multiple cores and achieve linear speedup. Two very similar implementations of an asynchronous algorithm can produce very different results due to differences in memory access patterns and code optimization. We find that increased data sparsity does not lead to greater speedup over serial execution when we parallelize linear regression with asynchronous stochastic gradient descent over multiple cores, confirming the results of Duchi et al. [3].

Keywords Asynchronous · Stochastic Gradient Descent · HOGWILD! · Speedup · Concurrent Programming · Sparsity

1 Introduction

Asynchronous methods provide a way to improve the speedup of parallelized algorithms when compared to synchronous methods. However, parallelizing stochastic gradient descent over multiple cores normally requires extensive resource management. If done successfully though, the benefits of asynchronous algorithms are enormous. It is theoretically possible to achieve a speedup, defined as the ratio between execution time of an algorithm on multiple cores to its execution time on a single core (also called serial execution), which scales perfectly linearly with the number of cores. An important part of asynchronous algorithms is the use of concurrent programming, which involves two primary concepts: parallelism and concurrency. Parallelism is the simultaneous computation of different parts of a task to complete it faster. Concurrency is the management of shared resources during parallel computations. For instance, shared memory is an example of a shared resource, because it is memory that can be simultaneously accessed by multiple cores. However, there are complications that come with sharing memory across processors, such as the fact that writes from different cores may overwrite each other. Thus, asynchronous algorithms require precise control of memory reads and writes to run successfully.

*Code can be found at: https://github.com/mstecklein/LargeScaleOptimization2_FinalProject_Hogwild.git

1.1 Previous Works

The HOGWILD! algorithm of Niu et al. introduces a method of parallelizing stochastic gradient descent over multiple processors without locks [1]. In HOGWILD!, processors are allowed equal access to shared memory, and each processor is able to update individual components of memory at will. Furthermore, the algorithm requires that data access to individual components of memory is sparse, so that memory overwrites are rare and computation errors be very unlikely to occur. This form of sparsity, which we refer to throughout this paper, can be described in the sense that each data sample that contributes to gradient updates has a small number of non-zero entries. On sparse problems, HOGWILD! attains near-linear speedup, scaling with the number of processors, and obtains an optimal convergence rate of $O(1/k)$.

Perturbed Iterate Analysis for Stochastic Optimization by Mania et al., extends the work of HOGWILD! by analyzing stochastic optimization methods where the input to each update is perturbed by bounded noise [2]. It introduces a perturbed framework that forms the basis of a unified approach to analyze asynchronous implementations of stochastic optimization algorithms by viewing them as serial methods operating on noisy inputs. By doing so, it is able to remove many of the assumptions in HOGWILD! and still yield some improved convergence rates. Lastly, it presents a parallel, sparse stochastic variance-reduced gradient (SVRG) algorithm called KroMagnon.

Asynchronous Stochastic Convex Optimization by Duchi et al. takes asynchronous gradient descent methods in a completely different direction. It makes the claim that sparsity is not necessary for asynchronous algorithms to work successfully over multiple cores [3]. The premise of the paper is that the noise from random sampling outshines the errors from asynchrony in iterations. In other words, the effects of asynchrony are negligible towards both the convergence rate and speedup of the algorithm.

1.2 Our Approach

In this paper, we empirically evaluate the results from [3] that sparsity has no effect on speedup and convergence. We emphasize that the results are extremely implementation-dependent, since minor tweaks in code can significantly affect speedup. The Experimental Results section will go more in-depth on our findings.

2 Motivations

Our goal is to empirically confirm or refute whether or not sparsity truly affects speedup in the HOGWILD! algorithm. To accomplish this, we first implemented the HOGWILD! algorithm in C to gain better control of threading and asynchrony. However, we quickly discovered that the implementation details of an asynchronous program have a huge effect on its ability to be parallelized and to achieve linear speedup. Even the smallest fixes in code can drastically improve the speedup. We describe the implementation in more depth in the following Implementation section.

3 Implementation

We implement HOGWILD! for linear regression in C. While the algorithm itself is simply stochastic gradient descent (SGD) across multiple cores without locking, there are some deeper implementation complexities which are not apparent from the algorithm itself. Here, we describe how we generate test data, perform sparse matrix calculations for gradient updates, and optimize our code to account for the nuances of the C language as well as our computer hardware.

3.1 Data Generation

In order to best interpret the performance of our algorithms as well as to isolate the effects of sparsity, we generate synthetic linear regression data to test our implementation. We wish to solve the normal linear regression problem:

$$\hat{\beta} = \arg\min_{\beta} \|y - X\beta\|^2 \tag{1}$$

We set the number of samples, n, number of coordinates, d, and the desired fraction of non-zero coordinates for each sample, $p \in (0, 1]$. Using these parameters we perform the following procedure: we draw i.i.d. random samples $x_i^{'} \sim N(0, I_{dxd})$ for $i = 1, 2, ..., n$, as well as our true coefficient vector $\beta^* \sim N(0, I_{dxd})$ from the standard Gaussian distribution. Then, using projection operator $\Pi_p(x)$ which randomly discards a fraction (1-p) of the entries of x, we get our desired sparse samples $x_i = \Pi_p(x_i^{'})$ for $i = 1, 2, ..., n$. We calculate data labels $y_i = \langle x_i, \beta^* \rangle$ for $i = 1, 2, ..., n$ and finally we construct our data matrix $X = [x_1 \;\; x_2 \;\; ... \;\; x_n]^T \in \mathbb{R}^{n \times d}$ and label vector $y = [y_1 \;\; y_2 \;\; ... \;\; y_n]^T \in \mathbb{R}^n$.

3.2 Sparse Matrix Calculations

Achieving linear speedup from parallelized stochastic gradient descent is highly dependent on implementation method. In fact, the intricacies of how the algorithm is implemented can dramatically affect the speedup. In order to save space and running time, it is critical to only store the non-zero elements of a sparse data matrix. We use a specific format for sparse matrix representation to achieve faster gradient calculation times. A standard representation of sparse matrices in sequential languages is to use a list of arrays, where each list corresponds to the respective row in the data matrix. Each array consists of sparse points, where a sparse point data structure is defined by two values: the index of a non-zero entry in the corresponding row of the original matrix, and the value of that non-zero entry. Algorithm 1 elaborates more on the method, where input M is the original matrix and input S is the sparse representation that will be made.

Algorithm 1 Sparse Matrix Representation

Input: $M \in \mathbb{R}^{R \times C}$, S
for *r = 1..R* **do**
    n = number of non-zero elements in row r
    Allocate n sparse points for row r of S
    $i = 0$
    for *c = 1..C* **do**
        if *M[r][c] != 0.0* **then**
            S[r][i].index = c
            S[r][i].value = M[r][c]
        end
        i++
    end
end

Now we will cover the concrete benefits of using the sparse matrix representation. The time complexity of creating the sparse representation of a matrix is $O(RC)$, where R is the number of rows in the matrix and c is the number of columns, and space complexity is $O(2RC)$. On the other hand, the time complexity of performing a single gradient calculation is also $O(RC)$. If parallelized stochastic gradient descent is run on a single core for a small number of iterations, then the time used to create the sparse representation will significantly slow down the run time. However, as the number of cores increases and as the number of iterations increases, the benefits will be clear.

Matrix-to-vector multiplication is also sped up by a factor that is inverse to the sparsity. For a problem in the form of Ax, where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, and p is the fraction of non-zero entries in A, the run time is expected to decrease by a factor of p.

It is important to note that using the sparse representation for matrix multiplication will give faster gradient calculations, but less speedup. However, the program will still be faster overall. Thus, we highlight that speedup alone is not a good metric for the performance of our system.

3.3 C Implementation

We implement the HOGWILD! algorithm in C. The C language provides the programmer the most control out of any object-oriented language over program details, such as memory usage and thread inter-dependencies, which are necessary for building a fully asynchronous algorithm. We used C's pthreads library to create, dispatch, and join threads. We also used C libraries for timing the execution of the algorithm, and for creating atomic instructions that function across platforms. Our program consists of algorithm implementations (i.e. HOGWILD! and our own example algorithms) and an analytical wrapper. The analytical wrapper sets up the problem parameters, creates threads to execute the algorithm asynchronously, and starts a timer to time the algorithm's runtime. It also initializes anything needed by the algorithm and allocates the necessary memory to prevent the need for slow memory allocations during the iterations of the algorithms. Having written the implementation in C, we found many intricacies of both the language and hardware that had effects on the runtime and parallelization of the algorithm, which we will further discuss.

When analyzing the performance of our program, one observation we made was that a number of very simple function calls were taking up 5% of the runtime. These functions were being called in the HOGWILD! algorithm, and were therefore executing millions of times over the course of the program's execution. Though their bodies were small, the overhead of the functions was adding up. Put simply, to perform a one-line instruction in a function, the program not only needs to execute that line, but must also perform the overhead related to calling a function, such as stack pushes and pops, as well as jumps in memory before and after the function call. A remedy to this issue is to use the `inline`

reserved word. `inline` replaces function calls with the actual code in the function during compile time. When the program executes instructions during runtime, the stack does not change and no jumps will need to be made. The following code demonstrates a simple example of how this works:

```
inline int foo(x){ return x + 1; }
y = foo(x);
```

After compilation, because of `inline`:

```
y = x + 1;
```

3.4 Hardware Intricacies

When debugging the initially poor speedup of our program, we discovered two interesting behaviors of our hardware. The way that shared memory was accessed by our threads had a huge effect on the inter-dependencies of the threads, and therefore on the speedup of the program. Specifically, the first issue was that two threads which touched the same 32-byte chunk of memory could not write simultaneously. The second issue was that the frequency with which different threads touched the same chunk of memory had a drastic effect on the execution time for those threads. Though we cannot give an absolute answer as to why we observed these phenomena, we have strong hypotheses for each.

When we initially constructed our program, we stored the iterate value as a `double` array. Each thread incremented indices of this array atomically, as per the HOGWILD! algorithm, but we observed absolutely no speedup as we increased the number of threads that the program was executed with. After investigation, we found that if we spaced the entries of the iterate array so that they laid at least 32-bytes apart in memory instead of contiguously, we starting seeing speedups as we increased the thread count. We wrote toy algorithms to verify that this was this case. The toy algorithm performed no operations on the data, but simply touched the iterate; if the iterate was stored as a contiguous `double` array in memory, the speedup remained 1 as threads increased, but if the iterate was stored with its elements spaced by at least 32-bytes, the toy algorithm exhibited almost linear speedups. Though we are uncertain as to why this was observed on our machines, it is safe to assume that somewhere in the hardware pipeline of writing data to shared memory, data is grouped into 32-byte chunks such that two writes to a given 32-byte block cannot happen simultaneously.

The second important observation was that the frequency of access to a location in shared memory by different threads dramatically affected speedup. Again, this observation was confirmed with toy algorithms. One toy algorithm partitioned the iterate such that no threads touched the same coordinates, and we observed a linear speedup from this algorithm. A second toy algorithm let threads touch coordinates of the iterate at random, and therefore share all indices of the iterate. This algorithm saw sub-linear speedups. Though again we are not absolutely sure of what causes this observation, we strongly hypothesize that it is due to the way caches work. Consider a simplified example with 2 separate processors, each with its own cache, and a shared memory across all processors. The caches provide a form of memory storage with small capacity and fast access, whereas the shared memory is much slower to access but is the only way for multiple processors to communicate. Let one processor have the most up-to-date value of a coordinate of the iterate in its cache. Then, if the other processor wants to read the most up-to-date value, the first processor will need to send its value of the iterate from its cache to shared memory, which is a relatively slow process. In other words, sharing a variable across threads, (e.g. our iterate), forces repeated movement of data between caches and shared memory, and defeats the point of the cache, which is to speed up data accesses. The performance loss of not fully utilizing the cache significantly outweighs the benefits of parallelizing the program, causing the sub-linear speedups we observed.

The important takeaway from these hardware intricacies that we have observed is that the structure of an asynchronous program has a huge effect on the its ability to be parallelized and to achieve linear speedup. One can imagine that if the details of data storage and the pattern with which data is touched by threads is as influential to the speedup as we have described, then two very similar asynchronous implementations of HOGWILD! could produce drastically different results, solely because of their different memory access patterns. In other words, the potential for an implementation of HOGWILD! to achieve linear speedup in practice is highly dependent on the choices the programmer makes and their ability to identify and remove limiting factors of parallelization in their program.

4 Experimental Results

We performed experiments on three different sample-coordinate combinations: (n=1000, d=1000), (n=1000, d=10,000), and (n=10,000, d=1000). For each of these combinations we tested four different levels of sparsity: p=0.005, p=0.01, p=0.2, and p=1 fraction of nonzero entries, for a total of 12 different datasets. We performed 10^6 total iterations of linear regression using asynchronous SGD on each of these datasets, i.e. no matter how many cores the algorithm runs on, 10^6 total update steps are applied across all cores. We used a fixed stepsize of 10^{-5} for each run.

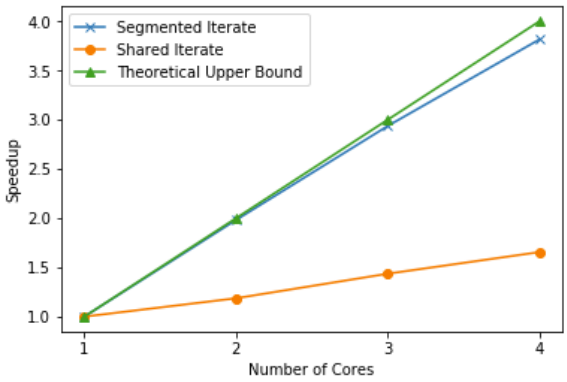

Figure 1: Speedup of example algorithms, with shared and segmented iterates

We ran these experiments on a machine with 256 GB RAM and 2 Xeon processors with 18 physical CPUs each and two virtual cores per CPU, for a total of up to 72 cores. In order to reduce overhead from memory transfer, we limited the number of cores we ran our experiments on to at most 24. This allowed the experiments to run entirely on one physical socket, eliminating the need to share data between processors.

4.1 An Implementation Nuance

As described in the previous section, the way in which a shared memory variable is touched across threads has a drastic effect on the ability of the program to be parallelized. To exemplify this, we contrast two example algorithms and show their respective speedups versus the number of threads and cores on which they are run. Note that these are toy algorithms for the purpose of instruction, and do not perform any useful action; they just access the iterate in a particular way, and do not search for the optimal iterate value.

Algorithm 2 Example Algorithm, Segmented Iterate

```
Input: iterate[]
for each thread do
    for n=1...num_iterations do
        index = this thread's ID number
        iterate[index] += 0.01 // arbitrary read/write
    end
end
```

Algorithm 3 Example Algorithm, Shared Iterate

```
Input: iterate[]
for each thread do
    for n=1...num_iterations do
        index = some random index of the iterate
        iterate[index] += 0.01 // arbitrary read/write
    end
end
```

The first algorithm segments the iterate into partitions such that each thread only touches its assigned partition. So although the iterate is shared across threads, no two threads share an individual coordinate of the iterate. An arbitrary way of partitioning the iterate is to let each thread access only the coordinate of its assigned thread number. The second algorithm allows all threads to touch the coordinates of the iterates at random. Therefore all coordinates are shared across all threads.

We observe in Figure 1 that the speedup is drastically different for these two algorithms. This difference is best exemplified by looking at the speedup across just the first four cores. The algorithm which segments the iterate into

partitions for each thread achieves a nearly perfect speedup, and the algorithm which shares all coordinates of the iterate across all threads shows a poor speedup. This observation should give some insight into how one would expect HOGWILD! to perform. The coordinates which the HOGWILD! algorithm touches are random and dependent on the data. Therefore for dense data, one may expect its speedup to resemble the Shared Iterate algorithm above. However for sparse data, though the coordinates are still accessed randomly, the sparsity may create an access pattern that more closely resembles the access pattern of partitioned data. Therefore, one intuitively might expect slightly better speedups from sparser data. Our following empirical observations support this intuition.

4.2 Convergence Behavior

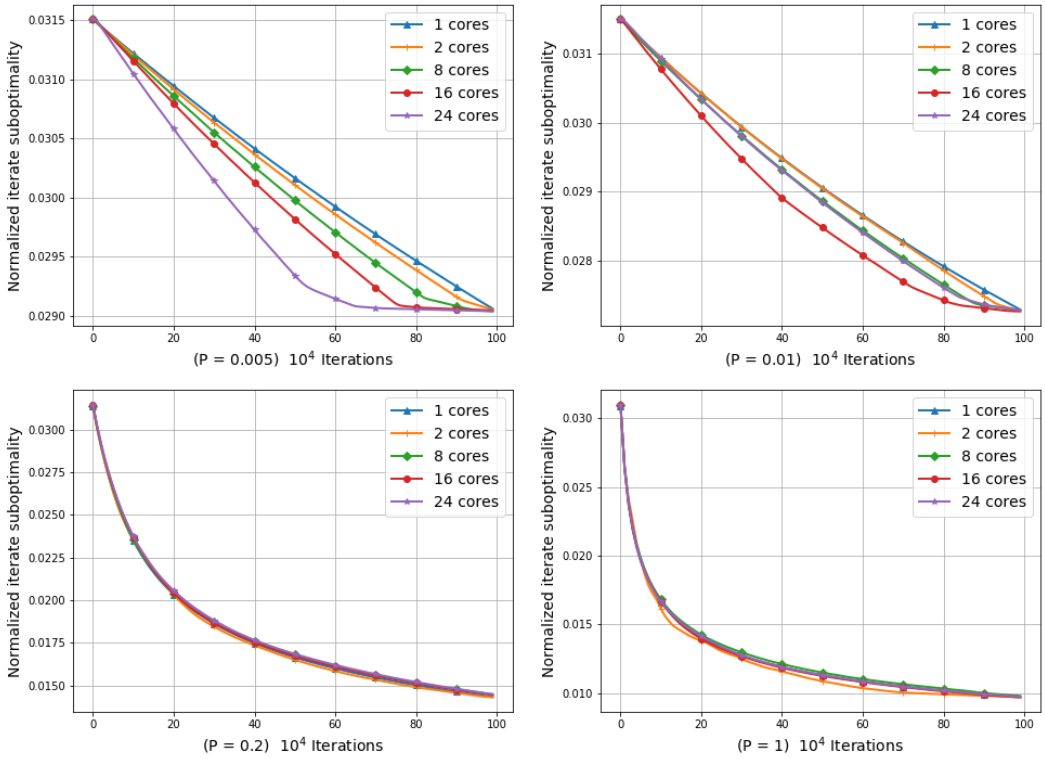

Figure 2: Suboptimality (in terms of l_2 error between the true solution β^* and our iterate β, normalized by the number of coordinates) vs iterations for our implementation of HOGWILD! on synthetic data with n = 1000 samples and d = 1000 coordinates, for sparsity levels with p=0.005, p=0.01, p=0.2, and p=1 fraction of nonzero entries. Each plot shows the convergence of asynchronous execution on 1,2,8,16, and 24 cores for a single sparsity level. Note that the x-axis counts by 1000s of iterations.

In Figure 2 we show the convergence of HOGWILD! for the dataset with n = 1000 and d = 1000 across all four sparsity levels $p \in \{0.005, 0.01, 0.2, 1\}$. We plot convergence as the l_2 error between the true coefficient vector β^* and our current iterate β. Each plot shows the convergence of linear regression on 1, 2, 8, 16, and 24 cores.

It is clear from the plots that convergence behavior is very similar across various numbers of cores for each sparsity level, except for a small difference in performance for the two sparser data (p = 0.005 and p = 0.01). This demonstrates that having more cores in asynchronous execution does not lead to an improvement in convergence over the fully serial (1 core) case as long as the number of total iterations remains constant. Therefore the purpose of parallelizing execution across multiple cores is solely to achieve speedup over the serial case, not to achieve a better solution in the same number of iterations.

The plots also display faster convergence for denser data, regardless of the number of cores. This makes sense intuitively since for sparser data we update only a few coordinates of the iterate β each time, as compared to dense data where many coordinates are updated. This difference in number of coordinate updates means that we must run more iterations of linear regression on a sparse problem to achieve the same number of coordinate updates as a dense problem.

4.3 Speedup across Sparsity Levels

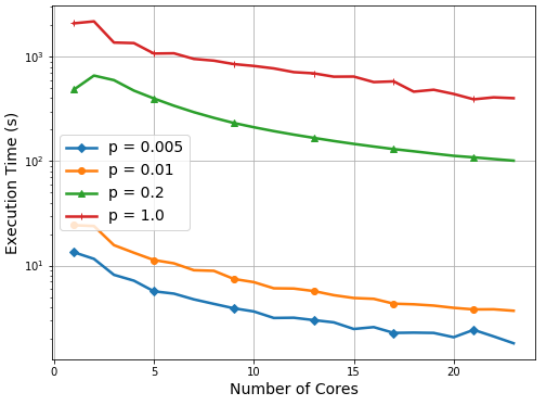

Figure 3: Execution time in seconds vs number of cores for data with n = 1000 and d = 10,000 across sparsity levels with p=0.005, p=0.01, p=0.2, and p=1 fraction of nonzero entries.

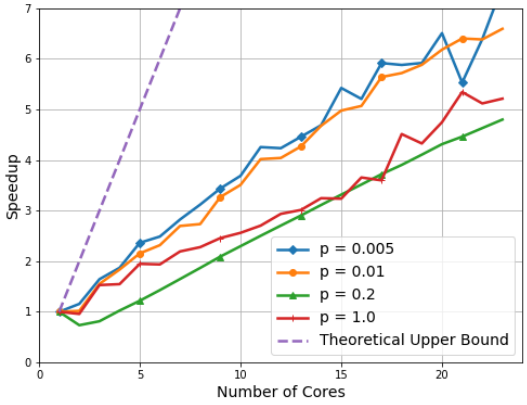

Figure 4: Speedup over serial execution vs number of cores for data with n = 1000 and d = 10,000 across sparsity levels with p=0.005, p=0.01, p=0.2, and p=1 fraction of nonzero entries.

In Figure 3 we plot the execution time of linear regression on HOGWILD! vs the number of cores, for the synthetic data with n = 1000 and d = 10,000 across all four sparsity levels $p \in \{0.005, 0.01, 0.2, 1\}$. We see that sparser data have a lower baseline execution time than denser data. This is due to two main factors: (1) gradient calculations on sparse data are performed much faster than on dense data with the sparse matrix storage method we described in section 3.2, and (2) reading and writing dense iterate updates back to the cache and to RAM is more expensive than for sparse updates. In fact, the execution times scale almost as we expect them to for different sparsity levels. For example, serial execution (1 core) for sparsity level $p = 0.01$ is about 10^2 times faster than for $p = 1$. This indicates that execution times scale about linearly with changes in data sparsity.

Figure 4 plots the speedup of linear regression on HOGWILD! (compared to serial execution) vs the number of cores for the synthetic data with n = 1000 and d = 10,000 across all four sparsity levels $p \in \{0.005, 0.01, 0.2, 1\}$. It is apparent that each sparsity level achieves an $O(num\ \ cores)$ speedup when parallelized, indicating that asynchronous SGD across multiple cores can lead to significant performance gains over serial execution. None of the sparsity values lead to performance comparable to a perfect linear speedup, which is expected due to issues like memory transfer overhead from RAM to cache. On the other hand, we see that for p = 0.2 there is actually a small slowdown between 2 and 4 cores, indicating that there may have been an unforeseen hardware-related issue causing unexpected execution.

From Figure 4 it is clear that there is little asymptotic difference in speedup for the four different sparsity levels. This seems to confirm the results of [3], which posits that data sparsity is not necessary to achieve good speedup over serial execution. However, while the experiments of [3] indicate that denser data achieves slightly better speedup, our results show that sparser data gives better speedup. This may be due to the fact that the authors use mini-batches of size 10 as opposed to pure SGD like our experiments, meaning that denser data have to be loaded less frequently into cache and improving execution time.

We can see from the plots that while dense data takes longer in real time to execute than sparse data for the same number of cores, both sparse and dense data achieve good speedups for asynchronous linear regression. This result is key, since past analyses of asynchronous SGD like in [1] and [2] relied on sparse gradient updates to achieve speedups over serial SGD. Furthermore, as in Figure 1, it is clear that implementation details are key to achieving this speedup and that careful attention must be paid to the intricacies of the hardware which runs these asynchronous algorithms.

5 Conclusion

Our C implementation of HOGWILD! validates the findings of [3]. The notable similarities between our results and those of [3] are: (1) that speedup is improved by increasing the number of cores, whereas convergence rate is unchanged except for the sparsest data, and (2) speedup is largely unaffected by the sparsity of the data. We continue to emphasize that the results are implementation-dependent, and that a similar implementation may produce drastically different outcomes. For future work, we would like to modify HOGWILD! to be more segmented. In addition, it would be of interest to explore an implementation of KroMagnon from [2].

References

[1] F. Niu, B. Recht, C. Re, and S. Wright. Hogwild: a lock-free approach to parallelizing stochastic gradient descent. In Advances in Neural Information Processing Systems 24, 2011.

[2] H. Mania, X. Pan, D. Papailiopoulos, B. Recht, K. Ramchandran, and M. I. Jordan. Perturbed Iterate Analysis for Asynchronous Stochastic Optimization. SIAM Journal on Optimization, 2015.

[3] John C Duchi, Sorathan Chaturapruek, and Christopher Ré. Asynchronous stochastic convex optimization. NIPS, 2015.