Cache-Friendly Shuffles for Asynchronous Stochastic Optimization

Maximilian Lam, Horia Mania, and Maxim Rabinovich

May 9, 2016

1 Introduction

In modern machine learning, many problems can be cast as instances of optimization, where the goal is to minimize some loss function encoding the system's errors on training data. The ubiquity of such problems has led to substantial interest in making large-scale optimization as efficient as possible, and a substantial portion of this work has focused on developing asynchronous algorithms that can under certain regimes achieve near-ideal speedups in shared-memory systems.

So called *stochastic* optimization algorithms have found particular success in this regard. These algorithms, of which stochastic gradient descent (SGD) is the simplest example, operate on problems whose loss functions decompose as a sum over datapoints:

$$F(x) = \sum_{i=1}^{n} f_i(x).$$

SGD then proceeds by iteratively pulling out the loss function f_i for a specific datapoint, updating the model x based on that data point alone, and then moving on to the next one. SGD and its close cousins have the advantage of being very simply parallelizable. Indeed, the Hogwild algorithm () simply spawns p threads to perform the single data point updates in parallel, allowing them to all update the model without any synchronization whatsoever. Perhaps surprisingly, this lock-free asynchronous approach works well for many problems, and since it avoids communication entirely, we might hope to achieve ideal speedups and thereby solve even the largest optimization problems facing machine learning practitioners.

Unfortunately, the simple story fails for a number of reasons, many of them systems-related. As the number of threads becomes large, the architecture of the machine imposes limitations and tradeoffs that must be taken into account. For instance, many machines with sufficiently many cores to support parallelism beyond approximately 16 threads organize their cores in a non-uniform memory access (NUMA) architecture, meaning that implementations of Hogwild must either cope with potentially long cross-NUMA node accesses to the memory storing the model or alter the algorithm so that cores on a given node usually only access memory on that node. And when moving to multiple machines, alterations to the algorithm become not only desirable but absolutely necessary.

A great deal of work has gone into understanding the statistical and computational properties of multi-node and multi-machine versions of the algorithm (). The recent DimmWitted system explored the design space of Hogwild-type algorithms in significant detail, tuning several knobs like: how much of the data each thread used; whether independent copies of the model were stored by each thread, or only on each node, or whether only a single global version of the model was maintained at all times; and how often threads received updated versions of other threads' models in the case of model replication. They found substantial differences in performance across these settings, and identified important computational-statistical tradeoffs to guide further work. In prior work, had undertaken a similar study in the rather different multi-machine setting.

Comparatively little attention has been paid to the simpler setting of a single node, despite the fact that, even there, failing to take the memory access pattern into account can limit the speedup from parallelization both on the individual node in isolation and in the context of larger multi-node or multi-machine systems. A notable exception to this general trend is the Jellyfish system for large-scale matrix factorization, which employs a blocking scheme to ensure that no two threads access the same parameters in any given epoch—a design choice that also ensures a high degree of cache locality within each thread's memory accesses, since each thread's assigned block depends only on a fraction of the total model parameters (). And more recently, further support for the idea that memory access optimizations can greatly increase the efficiency of optimization algorithms has come from work showing that low-precision arithmetic variants of the Hogwild algorithm can achieve substantially better speedups than their high-precision predecessors ().

In this report, we investigate a general-purpose methodology for optimizing the memory access efficiency of Hogwild-style algorithms. Building

on the Jellyfish results, we propose to group datapoints into blocks that each access only a limited subset of the parameters and to use these blocks constrain the order in which threads access datapoints. We hypothesize that, if the blocking is done well, this strategy will have the effect of increasing cache hits on model parameters within a single thread. Further, if different threads are further required to iterate through the blocks in different orders, we expect it to have the additional beneficial effect of reducing conflicting model accesses across threads, thereby also decreasing false sharing due to cache coherence effects.

The remainder of this report elaborates on our approach and provides evidence for these hypotheses. In Section 2, we explain the details of our proposed algorithm and introduce the bipartite datapoint-parameter dependence graph that underlies it. In Section 3, we provide a simple two-level memory model analysis of the HOGWILD! algorithm and frame the problem of optimally blocking datapoints as a graph partitioning problem in the dependence graph. In Section 4, we investigate how the structure of the datapoint-parameter graph affects the potential gains from effective blocking. In Section 5, we go on to show that a particular blocking strategy based on iterated min-cut can achieve substantial gains, both in runtime and speedup, on a real machine learning problem, namely the learning task for the popular Word2Vec model for word embeddings (). Finally, in Section 6, we reflect on our analysis and propose several avenues for further work.

2 Block-Constrained Shuffling

Our approach to making HOGWILD! memory access efficient hinges on constraining the order in which the algorithm processes data points. Typically, each thread in the algorithm processes the data points in a randomly selected thread-specific ordering, meaning that the data point processed by a thread at some iteration j need not share any parameters with the data point processed at iteration $j-1$ or $j+1$. We aim to remedy this defect.

Specifically, we aim to group the datapoints into blocks that mostly depend on the same parameters, and such that each block depends only a comparatively small number of distinct parameters. Given this blocking, we constrain the order in which threads process datapoints to be consistent with the blocking. Here consistency means that, as the thread iterates through the data, once it has processed a data point from some block, it will not process any more data points outside that block until it has finished with

the entire block.

We can make the algorithm specification more precise as follows. Let the blocks of data points be given by $\mathcal{B} = \{B_1, \ldots, B_K\}$, where each $B_k = \{i_{k1}, \ldots, i_{km_k}\}$ is a set of data point indices. Then we assume that the sequential ordering $I_p = (i_{p1}, \ldots, i_{pn})$ according to which thread p processes data points has the form $(I_{pa_1}, \ldots, I_{pa_K})$, where each I_{pa_k} is an ordering of the indices in B_k.

We point out, anticipating the analysis in Section 3, that the problem of choosing a good blocking bears a strong resemblance to the graph partitioning problem that arises in sparse matrix-vector multiply (). The difference is that the dependence graph for our problem links datapoints to parameters and therefore has a bipartite structure. The blocking problem is then simply a graph partitioning problem on one side of this graph.

To be more precise, we write each f_i as a function of a set S_i of s_i parameters, out of d model parameters total. To highlight this structure, we write $f_i(\mathbf{x}) = f_i(\mathbf{x}_{S_i})$. Datapoint i is then connected to all the parameters in S_i, with edges interpreted as undirected. As a consequence, parameter j is connected to all data points i such that $j \in S_i$. Figure **??** gives a schematic of such a graph.

3 A Simplified Memory Model Analysis

Simplified memory models can provide useful guidance when optimizing algorithm performance, as the case of matrix-matrix multiply illustrates (). In this section, we use a two-level memory model to estimate the computational intensity of the algorithm in terms of the blocking structure and use the resulting estimate to formalize the problem of finding an optimally memory access efficient blocking as a graph partitioning problem in the datapoint-parameter graph described in Section 2.

Concretely, assume the memory hierarchy has two levels, fast and slow, where fast memory contains sufficient storage for M model parameters and has zero access cost. We assume further that reads and writes have equal cost and that the computational effort required to obtain the model update based on datapoint i is proportional to s_i, say equal to cs_i for some absolute constant $c > 0$.

Note that under this framework, a model update based on datapoint i requires s_i parameter reads to compute the update, cs_i steps of computation (e.g. flops) to compute the update, and s_i parameter writes to actually perform the update.

Now let the order in which a thread performs updates based on data points be $I = \{i_1, \ldots, i_n\}$. To process data point i_j, the algorithm needs to read in some number $r_j(I)$ of model parameters to perform the update; the remaining $s_{i_j} - r_j(I)$ are already in fast memory and therefore cost nothing to access. The total number of reads is therefore given by $\sum_{i=1}^{n} s_i - \sum_{j=1}^{n} r_j(I)$, while the total number of writes is given by $\sum_i s_i$. The computational intensity can therefore be written as

$$q(I) = \frac{c \sum_{i=1}^{n} s_i}{2 \sum_{i=1}^{n} s_i - \sum_{j=1}^{n} r_j(I)}. \tag{3.1}$$

The problem of maximizing q is therefore equivalent to the problem of minimizing the sum

$$r(I) = \sum_{j=1}^{n} r_j(I). \tag{3.2}$$

In the context of the blocking algorithm, we can bound $r(I)$ as follows. Let the blocking structure be given by $\mathcal{B} = \{B_1, \ldots, B_K\}$. Let m_k be the number of distinct parameters touched by block k, or, more formally, $m_k = |\cup_{i \in B_k} S_i|$, which is also equal to the number of neighbors of B_k, $\mathcal{N}(B_k)$, in the datapoint-parameter dependence graph. Provided each m_k is bounded by the fast memory size M, we have

$$r(I) \leq \sum_{k=1}^{K} m_k. \tag{3.3}$$

Writing $m_k = |\mathcal{N}(B_k)|$, we thus find that the graph partitioning problem we face can be formulated as

$$\min \sum_{k=1}^{K} |\mathcal{N}(B_k)| \quad \text{subject to} \quad |\mathcal{N}(B_k)| \leq M,\ 1 \leq k \leq K. \tag{3.4}$$

As this problem bears a strong resemblance to the NP-hard set cover problem (), we do not attempt to solve it exactly. Instead, we investigate three heuristics for solving it:

1. **Truncated breadth-first search.** In this approach, we do not fix the number of blocks in advance. Instead, we iterate through the data points and whenever we encounter one that has not yet been assigned a block, we start a breadth-first search for datapoints to put in a new

block including the data point initiating the search. Whenever the search encounters a data point that has not yet been assigned to a block, it adds it to the new block and enqueues all of its neighbors that have not themselves been assigned to a block. The search halts either when the queue is empty, or when the number of data points assigned to he new block exceeds a client-specified threshold.

2. **Greedy streaming.** In this approach, we fix the number of blocks before the algorithm starts. Then, we iterate through the data points and at each step, add the next data point to the block B_k that makes the objective in (3.4) as small as possible.

3. **Iterated min-cut.** In this approach, we again fix the number of blocks in advance, specifically setting $K = 2^\kappa$, where $\kappa \geq 0$ is a depth parameter. We then perform κ recursive min-cuts and keep the bottom level of the resulting hierarchical blocking.

4 Least Squares

To test our hypotheses about the impact of cache effects on the speed of asynchronous optimization algorithms and to show the benefits of our proposed method, we implemented a HOGWILD! least squares solver. More precisely, we minimize the loss

$$F(\mathbf{x}) = \sum_{i=1}^{n} \tag{4.1}$$

5 Word Embeddings

5.1 Problem statement

In the word embeddings problem, given context counts $X_{w,w'}$ we want to find word vectors $v_w \in \mathbb{R}^k$ that minimizes the loss:

$$\min_{v,C} \sum_{w,w'} X_{w,w'} \left(\log(X_{w,w'}) - ||v_w + v_{w'}||^2 - C \right)^2$$

5.2 Experimental evaluation

5.2.1 Methodology

We ran our experiments on the Edison compute nodes which feature two twelve core 2.4 GHz processors. However, we used only up to twelve cores/threads to avoid effects of NUMA. Word vectors were length 100 double arrays.

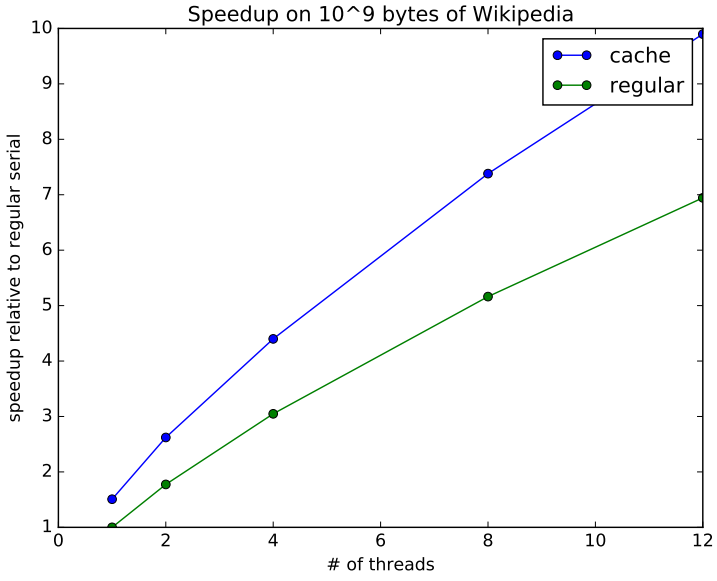
We used the first 10^9 bytes of English Wikipedia from http://mattmahoney.net/dc/textdata as corpus data. After running the text preprocessing script supplied by the link, we computed co-occurrence counts of pairs of distinct words to create the parameter dependence graph. This graph was then fed into gpmetis, computing a min-k-cut partitioning to create a cache-friendly ordering of the datapoints. k was set such that each block of k datapoints would reference just enough word vectors to fit into the L1-cache.

Hogwild was then run on the permuted co-occurrence graph generated by gpmetis, maintaining the same ordering throughout execution. Although we experimented with both data sharding and no-data sharding, only results from data sharding are presented. To test hogwild without a cache-friendly shuffle, we randomly shuffled the datapoints before execution.

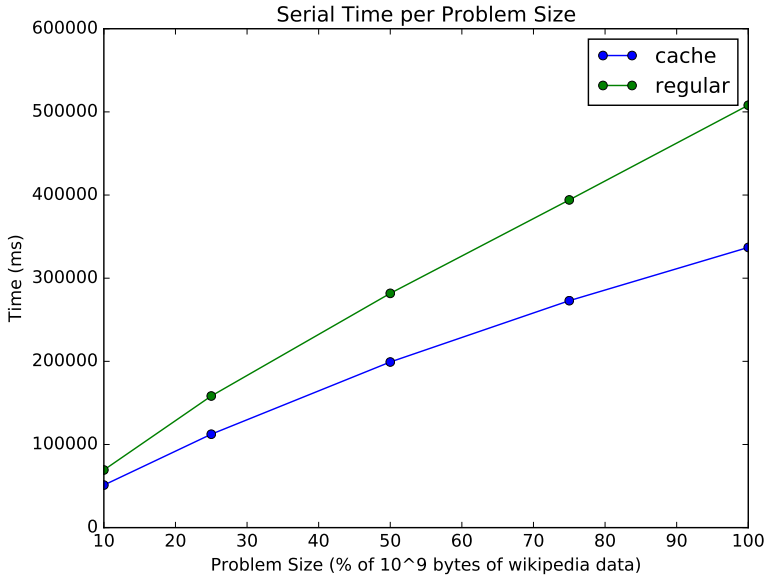
We also ran the experiments on subsets of the corpus, repeating the procedure on the first %10, %25, %50 and %75 of the corpus data. In the full corpus data, there were $200,000$ word vectors, and $30,000,000$ datapoints.

5.2.2 Results

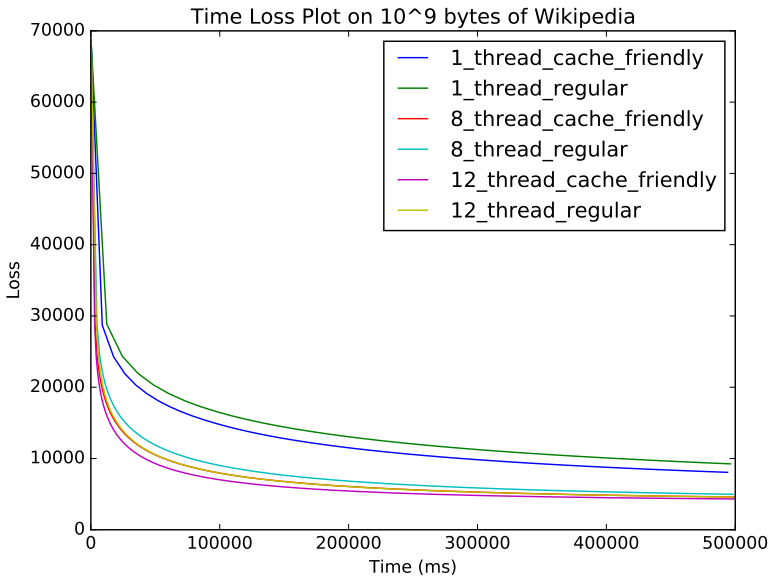
We achieve between $\%40 - \%50$ speedup over regular hogwild (non-cache-friendly hogwild), measuring runtime to a fixed number of epochs.



Furthermore, the speedup is maintained on different subsets and sizes of the data.



Additionally, convergence of loss is not adversely affected.



5.3 Discussion

A $\%40 - \%50$ runtime gain over regular hogwild is a result of keeping at least one length 100 double array in the L1-cache between stochastic gradient calls. In a non-cache-friendly permutation, each of the two vectors visited by a datapoint is typically not in the cache, incurring two vectors worth of cache misses per datapoint. After running min-k-cut on the parameter dependence graph, we found that each block of k datapoints references around k distinct vectors. Thus, in a cache-friendly permutation, one of the vectors referenced by a datapoint is already in the L1-cache from the previous stochastic gradient call. So a cache-friendly permutation incurs only one vectors worth of cache misses per datapoint, naturally leading to a $\%40 - \%50$ reduction in runtime.

6 Conclusion and future directions

By ordering datapoints so that model parameters are accessed in a cache friendly manner, we achieve substantial runtime reductions.

For word embeddings, we achieve a $\%40 - \%50$ runtime gain over regular hogwild by ordering the datapoints via min-k-cut.

From these results we believe that using a cache friendly shuffling of datapoints is a promising approach to reducing the runtime of stochastic optimization methods.

The runtime gains achieved by using a cache-friendly shuffle is highly contingent on the structure of the parameter depedence graph and on the method used to permute the datapoints. Thus, one possible direction to explore is the behavior of cache-friendly shuffles on different graph structures and problem types. We may find that graph properties such as sparsity, etc, have a nontrivial effect on the efficacy of cache-friendly shuffles.

Different shuffling methods may be another topic of exploration, particularly the tradeoff between computational efficiency and shuffle quality. Investigating different greedy methods and heuristics for greedy shuffles may reveal computationally efficient methods for generating a quality cache shuffe. On the other hand, it may also be interesting to see how effective optimal shuffles are, perhaps by generating them through some sort of integer linear programming routine.

Finally, a shuffling that takes advantage of all levels of cache may yield further runtime gains. Thus, a cache-oblivious shuffling method may be yet another area to explore.