Cache-Friendly Shuffles for Asynchronous Stochastic Optimization

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.

Even on a single node, issues of memory access create pitfalls that must be taken into account. In particular, cache locality and false sharing hamper performance, and prevent the attainment of ideal speedups with multiple threads.

1.1 Overview of the report

2 A Simplified Memory Model Analysis

3 Least Squares

4 Word Embeddings

4.1 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:

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

4.2 Experimental evaluation

4.2.1 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

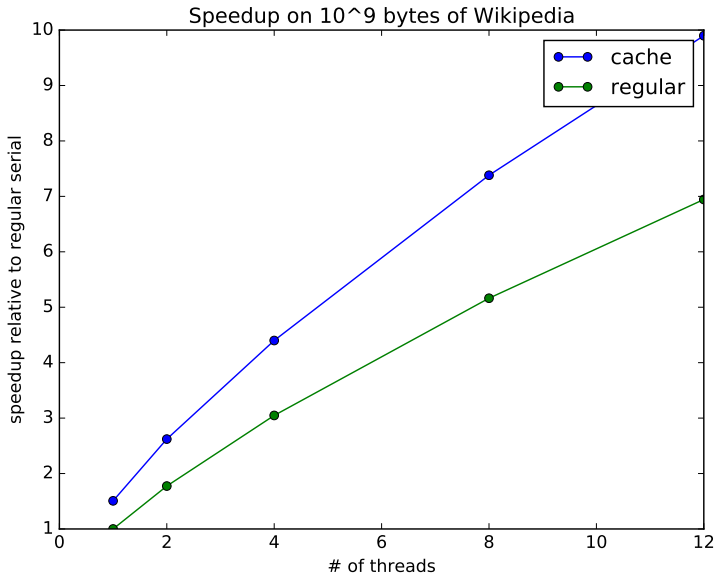
the parameter dependence graph. This graph was then fed into gpmetis, computing a min-k-cut partitioning to create a cache-friendly ordering of the datapoints. k was set such that each block of k datapoints would reference just enough word vectors to fit into the L1-cache.

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.

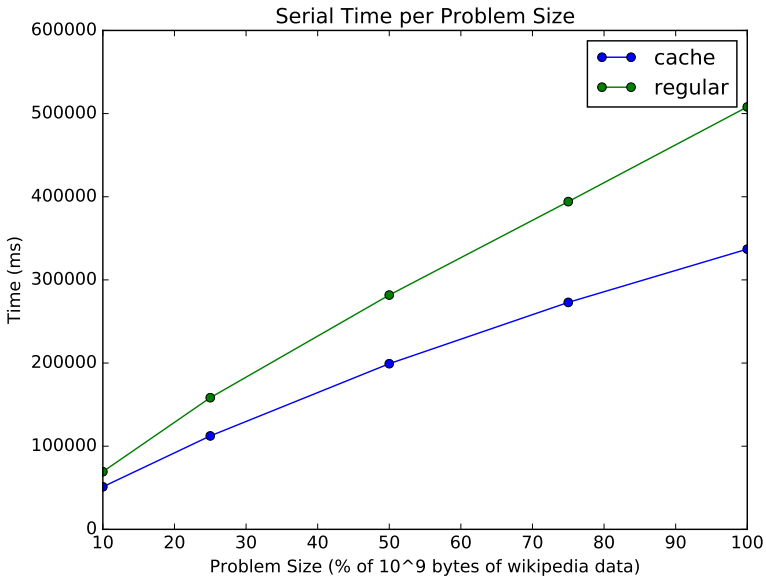
We also ran the experiments on subsets of the corpus, repeating the procedure on the first %10, %25, %50 and %75 of the corpus data. In the full corpus data, there were $200,000$ word vectors, and $30,000,000$ datapoints.

4.2.2 Results

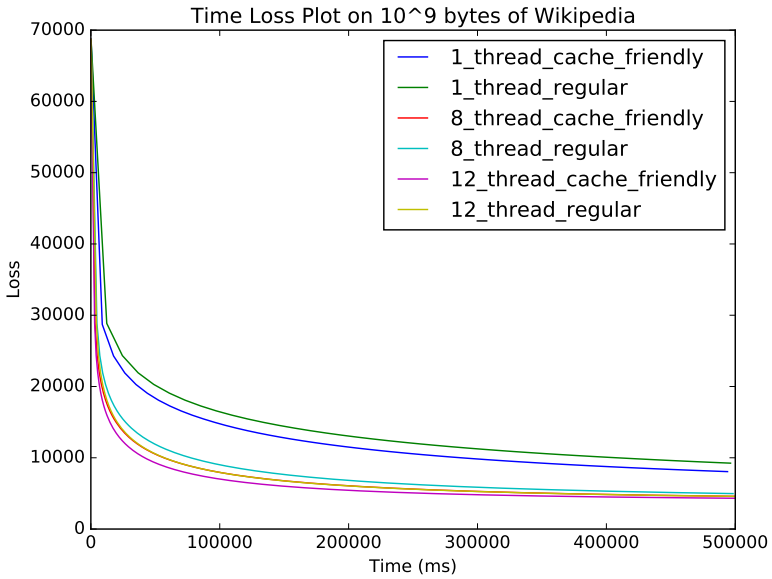
We achieve between $\%40 - \%50$ speedup over regular hogwild (non-cache-friendly hogwild), measuring runtime to a fixed number of epochs.



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



Additionally, convergence of loss is not adversely affected.



4.3 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.

5 Conclusion

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.

6 Future

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 and heuristics for shuffling may be another topic of exploration. In particular, the tradeoff between computational efficiency and shuffle quality may be explored. 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 method shuffling method may be yet another area for exploration.