HogWild++: A New Mechanism for Decentralized Asynchronous Stochastic Gradient Descent

Huan Zhang*, Cho-Jui Hsieh† and Venkatesh Akella*
*Dept. of Electrical and Computer Engineering,
†Depts. of Computer Science and Statistics,
University of California, Davis
Davis, CA 95616, USA
Email: {ecezhang,chohsieh,akella}@ucdavis.edu

***Abstract*—Stochastic Gradient Descent (SGD) is a popular technique for solving large-scale machine learning problems. In order to parallelize SGD on multi-core machines, asynchronous SGD (HOGWILD!) has been proposed, where each core updates a global model vector stored in a shared memory simultaneously, without using explicit locks. We show that the scalability of HOGWILD! on modern multi-socket CPUs is severely limited, especially on NUMA (Non-Uniform Memory Access) system, due to the excessive cache invalidation requests and false sharing. In this paper we propose a novel decentralized asynchronous SGD algorithm called HOGWILD++ that overcomes these drawbacks and shows almost linear speedup on multi-socket NUMA systems. The main idea in HOGWILD++ is to replace the global model vector with a set of local model vectors that are shared by a cluster (a set of cores), keep them synchronized through a decentralized token-based protocol that minimizes remote memory access conflicts and ensures convergence. We present the design and experimental evaluation of HOGWILD++ on a variety of datasets and show that it outperforms state-of-the-art parallel SGD implementations in terms of efficiency and scalability.**

***Keywords*—Stochastic gradient descent, Non-uniform memory access (NUMA) architecture, Decentralized algorithm**

I. INTRODUCTION

Stochastic Gradient Descent (SGD) has become one of the most important technique for solving large-scale machine learning problems. At each iteration, SGD randomly chooses a training sample and updates the model vector $\boldsymbol{w}$ according to the current estimate of gradient. Due to its low memory requirements and simple update rule, it has been applied to a wide range of applications, including support vector machines [19], neural networks [2], [8], and recommender systems [7]. Unfortunately, due to the nature of "sequential updates", it is not easy to parallelize SGD algorithms. During the past few years there have been some breakthroughs in parallelizing SGD algorithms including lock-free asynchronous update strategies [17] on multi-core machines, and the use of parameter servers to coordinate model updates [4] on distributed systems.

In this paper, we focus on parallelizing SGD on a multi-core machine, where cores are allowed to access a shared memory space. For this setting, HOGWILD! proposed in [17] is the most widely-used approach. In this algorithm, multiple threads conduct SGD updates asynchronously, and communication between threads is done implicitly by accessing the same model vector $\boldsymbol{w}$ stored in the shared memory space. This simple but effective idea has become extremely popular and used in many other applications, such as asynchronous coordinate descent [13], [6], Pagerank [14] and matrix completion [3].

Though HOGWILD!-style algorithms are lock-free and seem to have unlimited thread-level parallelism, they depict poor scalability as the number of cores increases. The main reason for this poor scalability can be traced to the fact that all the threads continuously read from and write to the same memory block that houses the global model vector, which results in excessive invalidation of cache lines on writes and an increase in coherence misses. This phenomenon is particularly deleterious on modern multi-socket systems with Non-Uniform Memory Access (NUMA) architecture, since forwarding data from a remote socket is very expensive. In Figure 1 we present the speedup of HOGWILD! on a NUMA machine with 4 CPU sockets with each socket being a processor with 10 cores. We used three classification datasets as sample problems, and applied HOGWILD! to train a linear SVM. Note that the speedup drops to less than 3 when more than one socket is used. Furthermore, when all the 40 cores are used, the overall speedup is less than 1 on these datasets, which means when using 40 cores HOGWILD! is worse than a single-thread serial SGD implementation. This motivates the research presented in this paper. Specifically, we are interested in improving the *scalability* of HOGWILD!-style parallel SGD algorithm on modern shared-memory processors with multi-level memory hierarchies.

In this paper, we propose a novel decentralized asynchronous parallel SGD algorithm called HOGWILD++ that is designed to work better with cache-coherent NUMA architectures. The main idea in HOGWILD++ is to replace the global model vector with a set of local model vectors that are shared by a *cluster*—a set of cores aligned within the same NUMA node. During the course of the algorithm, local model vectors are updated independently; after a certain number of updates, each cluster communicates the state of its local model vectors to its neighbors using a novel distributed token-based mechanism (described in Section IV) for the overall correctness of the algorithm and to ensure fast convergence. Since the updates are done to the local model vectors, cache invalidation on writes and coherence misses due to false sharing are dramatically reduced, which improves the scalability of the algorithm.

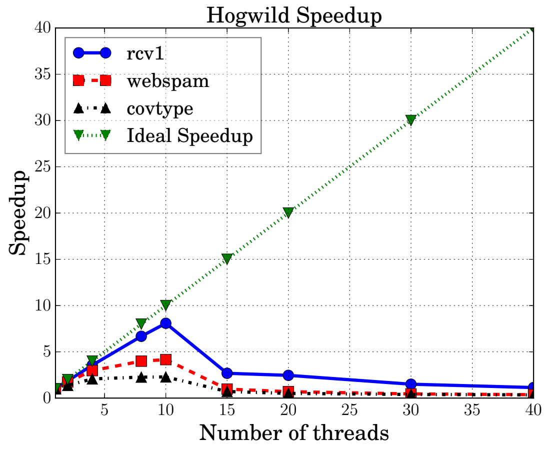

Fig. 1. Speedup for HOGWILD! on a 40-core machine with 4 sockets. We can observe that the speedup is far from ideal—it can be even slower than a single-threaded version when using 40 cores, and there is a clear performance drop when HOGWILD! uses more than 1 socket with 10 cores. This motivates us to develop a new asynchronous SGD algorithm in order to utilize the full power of multi-socket machines.

The main contributions of the paper are:

- We proposed HOGWILD++ , a non-blocking, lock-free, and fully asynchronous SGD implementation that scales very well on multi-socket machines with a large number of cores. By design, each cluster of cores maintains a local copy of the model to avoid coherence misses.
- HOGWILD++ is decentralized in nature, and does not maintain a central $\boldsymbol{w}$ or need a central coordinator to ensure convergence. Model synchronization is done asynchronously by passing model updates to a neighboring cluster using a token-based protocol.
- Our experiments show that HOGWILD++ is scalable and efficient. It is significantly faster than state-of-art asynchronous parallel SGD implementations, especially when running on machines with several NUMA nodes.

II. RELATED WORK

A. Parallel Stochastic Gradient Descent

Many parallel SGD algorithms has been presented in recent literature. Here we briefly review parallel SGD for multi-core and distributed systems.

Mini-batch SGD. Instead of computing the gradient of one training sample at a time, mini-batch SGD aggregates the gradient of b samples for conducting one update. Theoretical analysis and extensions to the basic mini-batch algorithm can be found in [5], [10]. Mini-batch SGD can be applied to both multi-core and distributed systems. However, its synchronization cost is too large for large-scale applications, and choosing the batch size b leads to a trade-off problem in communication time and convergence speed.

HOGWILD! for multi-core machines. HOGWILD! described in [17] introduced an asynchronous lock-free update scheme to parallelize SGD on shared memory multicore processors. But as we described in Section I, scalability of HOGWILD! is limited because of coherence misses since all the threads access the single copy of the model vector in the shared memory space. Recently, [12], [18] presented a solid theoretical analysis for HOGWILD!-styled algorithms, and [18] further proposed a BuckWild algorithm to exploit lower-precision SIMD arithmetic in modern CPU.

Zhang and Ré [24] discussed the scalability issue of HOGWILD!, and developed a framework, DimmWitted, with three model replication approaches to explore gradient based methods on multi-core systems. The `PerMachine` approach is identical to HOGWILD!. The `PerCore` approach keeps a local replica of the model for each worker thread, and averages the model at the end of each epoch; The `PerNode` approach keeps a replica of the model for each NUMA node (CPU socket), and a dedicated thread on each node reads models from all other nodes and computes the average. On a node with N cores, `PerNode` approach launches N worker threads, plus an additional thread for model averaging. We will compare our approach with HOGWILD! and DimmWitted in Section VI.

Asynchronous SGD for distributed systems. There are other asynchronous SGD algorithms proposed in distributed systems. For example, the concept of parameter servers is proposed in [4], [21], [9], where the updates are controlled by a parameter server formed by a subset of machines. [26] recently proposed an elastic averaging SGD to get better performance on non-convex problems.

B. Other Asynchronous Algorithms

Due to the success of HOGWILD!, there are many asynchronous methods proposed for parallelizing other algorithms or solving some specific machine learning problems. For example, [13], [6] developed asynchronous coordinate descent algorithms, [25] proposed a simple way to improve the convergence property of asynchronous coordinate descent. [11] proposed a comprehensive convergence analysis for asynchronous algorithms. However, all the above algorithms follow the HOGWILD! framework, where the parameters are stored in a centralized shared memory, and the memory access will become a bottleneck when scaling to multiple sockets. Our proposed HOGWILD++ algorithm solves this critical scalability issue for SGD, and can also be potentially applied to other asynchronous algorithms mentioned above.

Another line of research focuses on developing asynchronous algorithms for specific machine learning problems. For example, [23] developed an asynchronous SGD algorithm (called NOMAD) for matrix completion on multi-core or distributed systems. This algorithm can also be used to learn topic models [22]. However, they focus on a specific type of problem and cannot be generalized to other problems.

III. BACKGROUND

In this section we will discuss the HOGWILD! approach for asynchronous SGD updates and its limitations in terms of scalability on multi-socket multicore architectures.

A. Asynchronous Stochastic Gradient Descent

We consider the following minimization problem

$$\min_{\boldsymbol{w}\in\mathbb{R}^n} \Big\{\frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{w})\Big\} := f(\boldsymbol{w}), \qquad (1)$$

where the objective function $f(\cdot) : \mathbb{R}^n \to \mathbb{R}$ can be decomposed into m components. Most machine learning problems follow this framework, where $f_i(\boldsymbol{w})$ is the loss defined on one training sample. For example, the SVM problem can be written as

$$\min_{\boldsymbol{w}} \sum_{i=1}^{m} \left(C \max(1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i, 0) + \frac{1}{2} \sum_{j \in \Omega_i} \frac{w_j^2}{d_j} \right), \quad (2)$$

where $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$ are training samples and labels, C is the regularization parameter, Ω_i is the set of the nonzero indices in the i-th sample $\boldsymbol{x}_i$, and $d_j = |\{i : j \in \Omega_i\}|$.

Stochastic Gradient Descent (SGD) is one of the most widely used approaches for solving (1), especially when there are a large number of samples. At each iteration, SGD randomly samples an index i from $\{1, \cdots, m\}$, and conducts the following update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta_k \nabla f_i(\boldsymbol{w}), \quad (3)$$

where η_k is the step size. Many recent efforts on parallelizing SGD focus on the HOGWILD!-style updates on multi-core computers, where multiple threads keep conducting updates (3) simultaneously and asynchronously to the same model vector $\boldsymbol{w}$ in the shared memory:

Each thread repeatedly performs the following updates:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta_k \nabla f_i(\boldsymbol{w}).$$

Note that there is no locking in HOGWILD!, but the correctness can be proved under certain condition. Unfortunately, as shown in Figure 1, HOGWILD! is not scalable when using many cores in a single machine. To understand this phenomenon, we need to look into the system architecture of multi-core processors.

B. Cache Coherence Protocol in modern CPUs

Modern Intel CPUs use the MESIF cache coherence protocol in a multi-core system [20]. The purpose of cache coherence system is to keep the communications between cores implicit and transparent to programmers. If one core writes a variable w in its cache, all other cores should be able to see the updated w. The smallest resource block managed by cache controllers, called a cache-line, is usually 64-bytes.

Under the MESIF protocol, any cache-line that is shared by more than one core, will be marked as "shared". Reading of a variable inside this cache-line, is a cache-hit and very fast. However, when one of the cores, C1, modifies this variable, it will broadcast a request on the bus to invalidate all other cores' shared copies in their caches because they are now outdated. When another core, C2, wants to access this variable again, it has to be forwarded from C1 using the internal ring bus. Profiling results from [16], [15] show that this is usually a few times slower than a cache hit.

In a multi-socket system, CPU sockets (each one usually contains more than one cores) are connected by the inter-socket QPI bus. QPI not only has less bandwidth than the internal ring bus, but also has much higher latency. Forwarding data from another socket's cache can be two orders of magnitude slower than a cache hit, depending on the topology distance of the two sockets[16], [15].

False Sharing is another performance issue when two or more cores share data. Assume two cores are modifying two different variables, w_{i_1} and w_{i_2}, but w_{i_1} and w_{i_2} happen to be adjacent in memory and thus reside in the same cache line. When the first core, C1, writes to w_{i_1}, the cache controller of the second core, C2, will invalidate its cached copy of w_{i_2}, even if C1 and C2 actually don't share the two variables. When C2 needs to read w_{i_2}, a cache-miss occurs, and the entire cache line containing both w_{i_1} and w_{i_2} has to be forwarded from C1.

C. Scalability of HOGWILD!

Instruction level profiling of HOGWILD! over two different classification datasets (covtype and unigram webspam) shows that it is a memory-bound application, i.e., over 90% of execution time is spent on reading training data and updating the model, instead of computing the gradient. Since training data are only read, they do not constitute a bottleneck as the number of cores increases. The scalability of HOGWILD! depends heavily on the model memory access pattern, which in turn depends on the input dataset. The size of the model (or the feature size, in the SVM context) and the sparsity of the data are the major factors to consider here. When the size of the model vector is small (as in covtype and webspam), cache invalidation due to frequently updated shared variables significantly slows down memory operations, and false sharing exacerbates this situation further. For example, consider a model with 256 double-typed elements. It will reside in only 32 cache lines. If there are a large number of worker threads (say 40) updating elements in the model vector, multiple cache invalidations can occur in every gradient update.

A small model definitely hurts scalability, but a large model does not necessarily guarantee good scalability. Suppose the data is high dimensional and very sparse, however the non-zeros are concentrated in very few coordinates. In this case, only the corresponding very few coordinates of the model vector are updated frequently. The overall effect, is similar to updating a small model. HOGWILD! would scale better when the model size is large, data is sparse and the non-zero elements in the data are distributed uniformly. However, that is not always the case and hence we desire algorithms that will scale well on all data sets with a wide variety of characteristics.

IV. PROPOSED ALGORITHM

In this section, we introduce the proposed decentralized asynchronous SGD algorithm—HOGWILD++ .

A. Design Considerations

a) Reducing Sharing: Knowing that excess cache invalidation caused by frequently reading and writing the shared model vector $\boldsymbol{w}$ is the bottleneck in lock-free parallel stochastic gradient algorithms like HOGWILD!, we want to reduce sharing between threads, so that when one thread writes to a model vector, it does not slow down the reads of other threads.

Zhang and Ré [24] proposed an algorithm called DimmWitted, and investigated "`PerCore`" and "`PerNode`" strategies where each core has a separated $\boldsymbol{w}$, or each node (physical socket) shares a $\boldsymbol{w}$. To gain better flexibility, we propose the "per-cluster" strategy, which groups c cores into a *cluster*, sharing a single model $\boldsymbol{w}_j$. This model vector will be updated by the c cores in that cluster in a lock-free manner. HOGWILD! can be seen as a special case when there exists only one cluster and thus $c = N$, where N is the total number of physical cores.

When selecting the cluster size for a problem, it is a trade-off between *hardware efficiency* and *statistical efficiency*. Less sharing slows down convergence and thus more iterations are needed, but the hardware can run each iteration faster.

b) Model Synchronization Strategy: To reduce sharing between threads we have to create multiple model vectors, but how to synchronize these models so that the entire system will converge to an optimal solution quickly?

One solution (used in DimmWitted) would be using a separated synchronization thread which periodically reads models on all other workers, and then writes the average back to its own model replica. We argue that this does not scale well in a large-scale NUMA system. When the synchronization thread updates the model replica, it has to access models of all other threads (or clusters). If we have more than 1 node, the synchronization thread has to access remote memory frequently. For example, consider a system with 128 nodes, then the synchronization thread has to access other 127 nodes' remote memory, which can be very expensive.

To address this problem, we propose that each cluster will only communicate with its topological neighbors (e.g., cores on the same physical socket, or its neighboring sockets). There is no special thread for synchronization; instead, each thread synchronizes with its neighbors by writing its own updates $\Delta\boldsymbol{w}$ into its neighbors' memory. Communication with neighboring cores is low-cost, and we can potentially fully utilize the interconnect bandwidth if we allow non-conflicting (in terms of interconnect topology) clusters doing synchronization at the same time.

c) The Model Synchronization Ring: In this paper, since we are targeting at multi-socket systems, we only consider a simple case of the above-mentioned idea: all clusters are connected by a logical ring, and they take turns to update model vectors to their neighbors. We maintain a *token*, that is passing along the ring, and only the thread with the token can synchronize with its next neighbor on the ring.

In our algorithm, each cluster stores two local model vectors $\boldsymbol{w}_j$ and $\bar{\boldsymbol{w}}_j$, where $\boldsymbol{w}_j$ is the snapshot of the model vector after last synchronization and $\bar{\boldsymbol{w}}_j$ is constantly being updated. We denote $\boldsymbol{w}_j^k$ as the local model vector in cluster j when it just finishes its k-th round of synchronization. When cluster j conducts an SGD update, it updates the model parameters in $\bar{\boldsymbol{w}}_j^\tau$, where τ is the iteration number of SGD update, and keeps $\boldsymbol{w}_j^k$ intact. The difference between $\boldsymbol{w}_j^k$ and $\bar{\boldsymbol{w}}_j^\tau$ is defined as:

$$\Delta\boldsymbol{w}_j^k = \bar{\boldsymbol{w}}_j^\tau - \boldsymbol{w}_j^k \tag{4}$$

Cluster j will always use $\bar{\boldsymbol{w}}_j^\tau$ in gradient computation.

d) Communication Protocol: When cluster j gets the token, it will synchronize its model by passing $\Delta\boldsymbol{w}_j^k$ to the next cluster on the ring, and updating its own snapshot $\boldsymbol{w}_j^k$. It is actually nontrivial to design this protocol. If cluster j directly writes $\Delta\boldsymbol{w}_j^k$ to the next cluster without decaying, then $\Delta\boldsymbol{w}_j^k$ will be passed along the entire ring, and eventually returns to cluster j. This will certainly cause divergence because $\Delta\boldsymbol{w}_j^k$ is amplified during the synchronization process, and each local model $\boldsymbol{w}$ will eventually go to infinity.

To solve this problem, we decay $\Delta\boldsymbol{w}$ by a factor β each time when we compute $\boldsymbol{w}_j^{k+1}$, so the operations will be:

$$\text{Add } \beta\Delta\boldsymbol{w}_j^k \text{ to } \bar{\boldsymbol{w}}_{j+1}^{\tau'} \text{ in a lock-free manner} \tag{5}$$

$$\text{Add } \beta\Delta\boldsymbol{w}_j^k \text{ to } \boldsymbol{w}_j^k \tag{6}$$

Note that only cluster j can update $\boldsymbol{w}_j^k$, so there will be no conflicts for (6). Define M as the number of clusters. We use the root of the following equation as the value of β:

$$\beta^M + \beta = 1 \tag{7}$$

This guarantees $\Delta\boldsymbol{w}_j$ will not be added multiple times, since after passing through one round, $\Delta\boldsymbol{w}_j$ becomes $\beta^M\Delta\boldsymbol{w}_j$. When $\beta^M\Delta\boldsymbol{w}_j$ adds up to cluster j's own update $\beta\Delta\boldsymbol{w}_j$, the entire effect is $1\cdot\Delta\boldsymbol{w}_j$. Note that β is always larger than 0.5 and usually close to 1. For example, $\beta = 0.934$ when $M = 40$, and $\beta = 0.724$ when $M = 4$. Therefore, the information will mostly be written into the current model immediately.

Another benefit of decaying $\Delta\boldsymbol{w}_j$, is that we want an older $\Delta\boldsymbol{w}_j$ to have less weight. When a $\Delta\boldsymbol{w}_j$ has been passed along some (say μ) clusters, it reflects relatively old information, so it is reasonable to give it a smaller weight β^μ.

e) Updating Local Model: The update in (6) only uses $\Delta\boldsymbol{w}_j^k$. However, by analyzing the accessing pattern, we can improve this step by using some free information obtained during the synchronization process.

In a shared-memory system, at the k-th round of synchronization when cluster j writes its $\Delta\boldsymbol{w}_j^k$ to cluster $j+1$ without any intervention of cluster $j+1$, it has to read $\bar{\boldsymbol{w}}_{j+1}^{\tau'}$ first, and compute $\Delta\boldsymbol{w}_j^k + \bar{\boldsymbol{w}}_{j+1}^{\tau'}$, then write it back. During this process, we have to read $\bar{\boldsymbol{w}}_{j+1}^{\tau'}$. We want to fully utilize this information, since we get $\bar{\boldsymbol{w}}_{j+1}^{\tau'}$ for free during this process. Therefore the update rule (6) is replaced by the following equation where the new $\boldsymbol{w}_j^{k+1}$ is computed using both $\boldsymbol{w}_j^k$ and $\bar{\boldsymbol{w}}_{j+1}^{\tau'}$:

$$\boldsymbol{w}_j^{k+1} = \lambda\bar{\boldsymbol{w}}_{j+1}^{\tau'} + (1-\lambda)\boldsymbol{w}_j^k + \beta\gamma^t\Delta\boldsymbol{w}_j^k, \tag{8}$$

where γ is the step decay and it is necessary for encouraging convergence and t is the iteration count of a SGD outer iteration (the *epoch*), which only increases after all training samples have been touched once. Another benefit of (8) is to ensure that our algorithm converges to a single $\boldsymbol{w}^\infty$ on all clusters, which we will show later.

When cluster j writes its $\Delta\boldsymbol{w}_j^k$ into cluster $j+1$, it also merges $\bar{\boldsymbol{w}}_{j+1}^{\tau'}$ into its new $\boldsymbol{w}_j^{k+1}$. This is reflected as the coefficients λ in the update rule (8). Because $\bar{\boldsymbol{w}}_{j+1}^{\tau'}$ contains information from $\Delta\boldsymbol{w}_{j+1}^k$, which will be finally passed to cluster j as $\beta^{M-1}\Delta\boldsymbol{w}_{j+1}^k$, we use the following λ as the blending coefficient to avoid divergence:

$$\lambda = 1 - \beta^{M-1}. \tag{9}$$

B. The HOGWILD++ Algorithm

Based on the discussions above, we formally propose our algorithm, HOGWILD++ , for improving scalability on multi-socket NUMA systems. It groups all N cores in a NUMA systems into M *clusters*. Each cluster independently performs the updates as described in Algorithm 1.

Algorithm 1 HOGWILD++ for each individual cluster j

```
Input: $\boldsymbol{w}^0$, initial step size $\eta_0$, step decay $\gamma$, number of clusters $M$ and a token $\mathfrak{T}$
  Pre-compute $\beta$ as the root of $\beta^M + \beta = 1$
  $\lambda = 1 - \beta^{M-1}$
  $k = 0$, $\tau = 0$, epoch counter $t = 0$
  $\bar{\boldsymbol{w}}_j^0 = \boldsymbol{w}_j^0 = \boldsymbol{w}^0$ shared by all threads in cluster $j$
  Each thread in cluster $j$ repeatedly performs:
    randomly pick a $i$
    $\bar{\boldsymbol{w}}_j^{\tau+1} = \bar{\boldsymbol{w}}_j^{\tau} - \eta_0\gamma^t\nabla f_i(\bar{\boldsymbol{w}}_j^{\tau})$
    $\tau = \tau + 1$
    if the this thread gets $\mathfrak{T}$, synchronize:
      $\Delta\boldsymbol{w}_j^k = \bar{\boldsymbol{w}}_j^{\tau} - \boldsymbol{w}_j^k$
      $\boldsymbol{w}_j^{k+1} = \lambda\bar{\boldsymbol{w}}_{j+1}^{\tau'} + (1-\lambda)\boldsymbol{w}_j^k + \beta(\gamma^t\Delta\boldsymbol{w}_j^k)$
      Update $\bar{\boldsymbol{w}}_{j+1}^{\tau'}$ in a lock-free manner:
          $\bar{\boldsymbol{w}}_{j+1}^{\tau'} = \bar{\boldsymbol{w}}_{j+1}^{\tau'} + \beta\gamma^t\Delta\boldsymbol{w}_j^k$
      $\bar{\boldsymbol{w}}_j^0 = \boldsymbol{w}_j^{k+1}$
      $k = k + 1$, $\tau = 0$
      pass $\mathfrak{T}$ to cluster $j+1$ after $\tau_0$ iterations
```

To start, a token $\mathfrak{T}$ is given to the first thread of the first cluster after it has performed several initial updates. For simplicity, in Algorithm 1, only the first thread of a cluster is responsible for synchronization. This may cause unbalanced workload within a cluster, since the first thread has more work to do. In a real implementation, threads in each cluster can take turns to synchronize with the next cluster. Also, we observe that it is worthwhile to delay passing $\mathfrak{T}$ until τ_0 inner iterations have been done so that cluster $j+1$ can do enough updates before it starts to synchronize, especially when the number of clusters is small. Also note that the epoch counter t in our algorithm will increase by one when totally m (number of training samples) SGD updates are done.

Theorem 1: HOGWILD++ converges to a single $\boldsymbol{w}^\infty$ on all clusters under the conditions where HOGWILD! converges and $\gamma < 1$.

Proof: When $t \to \infty$, $k \to \infty$, $\gamma^t \to 0$ and if HOGWILD! converges, $\Delta\boldsymbol{w}_j^k$ is always bounded and thus the update rule (8) becomes:

$$\boldsymbol{w}_j^{k+1} = \lambda\boldsymbol{w}_{j+1}^k + (1-\lambda)\boldsymbol{w}_j^k$$

Define matrix $W^k \in \mathrm{R}^{M\times d}$ where its j-th row is row vector $\boldsymbol{w}_j^k$ and d is the size of model vector. Now we can write the update rule for round $k+1$ in a matrix form:

$$W^{k+1} = U \cdot W^k$$

where $U \in \mathrm{R}^{M\times M}$ is defined as

$$U = \begin{bmatrix} 1-\lambda & \lambda & 0 & \cdots & 0 \\ 0 & 1-\lambda & \lambda & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 1-\lambda & \lambda \\ \lambda(1-\lambda) & \lambda^2 & \cdots & 0 & 1-\lambda \end{bmatrix}$$

The structure of the last row is different since $\boldsymbol{w}_M^{k+1}$ is computed with $\boldsymbol{w}_1^{k+1}$ rather than $\boldsymbol{w}_1^k$. Since each row of U sums to 1 and all entries are real, positive numbers, for any vector $\boldsymbol{x}$ with x_{max} as the coordinate with largest modulus, $U\boldsymbol{x} \le x_{max}\cdot\boldsymbol{e}$ where $\boldsymbol{e} = (1,1,\cdots,1)^T$ because each coordinate of $U\boldsymbol{x}$ is a convex combination of coordinates of $\boldsymbol{x}$. Thus, U cannot have any eigenvalue greater than 1. Since 1 is U's eigenvalue because $U\cdot\boldsymbol{e} = 1\cdot\boldsymbol{e}$, 1 is U's largest eigenvalue. According to Perron-Frobenius theorem, U's largest eigenvalue is strictly greater than all other eigenvalues. Thus, for any real vector $\boldsymbol{x} \in \mathrm{R}^M$, $\lim_{k\to\infty} U^k\boldsymbol{x} = \frac{\sum_{i=1}^M x_i}{M}\boldsymbol{e}$. Therefore, $\lim_{k\to\infty} U^k W = W^\infty$, where each row of W^∞ is $\boldsymbol{w}^\infty = \frac{\boldsymbol{w}_1^k + \boldsymbol{w}_2^k + \cdots + \boldsymbol{w}_M^k}{M}$, which is the average of all $\boldsymbol{w}_j$. ■

We have not formally shown that $\boldsymbol{w}^\infty$ will be an optimal solution. However, the above theorem allows us to focus on analyzing one local model $\boldsymbol{w}_j^k$, whose behavior is very similar to the original HOGWILD! with delayed updates and its theoretical guarantee has been thoroughly analyzed. In the future we expect to show our algorithm converges to the global minimizer. Furthermore, we are able to show experimental convergence on all the datasets used in Section VI.

C. Possible Extensions

In the above discussions, we designed and analyzed our algorithm for a shared memory NUMA system. However, our proposed algorithm does not require any centralized structures or global communication. When applying our algorithm to a distributed system that is based on message passing, no central coordinator (e.g., a parameter server) is necessary. Though in the current implementation we assumed the underlying network topology to be a ring, the algorithm can work with other network topology as well and is eminently suitable for implementation on modern supercomputers. Our algorithm can exploit fast local interconnections in modern supercomputers by passing model information along neighbors, which is an advantage over model averaging based approach, where each model vector has to be read despite its topological distance.

V. IMPLEMENTATION ISSUES

Token Implementation. We use the GCC built-in `__atomic_fetch_add()` to implement a lock-free counter, serving as the token. Each thread will check the counter value and if it matches its thread ID, it will start the model synchronization. Then, it waits for a configurable number of iterations τ_0 and then increases the atomic counter so that the next thread can synchronize its model.

Data Permutation. The data permutation subroutine which is executed before model updates in HOGWILD! is a serial and naive implementation. In HOGWILD! the SGD updates cannot achieve good speedup, so a serial data permutation is not a problem because execution time is dominated by the SGD update time. However, in HOGWILD++ , SGD updates become so fast that the serial permutation subroutine turns out to be a noticeable serial bottleneck. A fast parallel random permutation algorithm [1] can be implemented so that this part will not be a bottleneck. Unfortunately, it is not easy to implement it in HOGWILD!'s existing coding framework, so we don't include permutation time in the training timer in both HOGWILD! and HOGWILD++ . Implementing a high-performance data permutation algorithm will be important in real world applications.

Thread Affinity. HOGWILD! has implemented its own thread pool using POSIX threads. To avoid unnecessary communication between sockets, We use `libnuma` to identify CPU topology and enforce that threads are assigned to as few nodes (sockets) as possible, and threads within a single cluster are allocated to the same node. Hyper-threaded cores are not used, unless the user requests more worker threads than available physical CPUs. We implement thread affinity in both the original HOGWILD! and our HOGWILD++ .

Data Replication. We replicate data to each node as long as that node is being used by some threads and we have enough memory. We use `libnuma` to guarantee that memory is allocated to correct nodes.

Turbo Boosting. Due to lack of root privilege on the machine we used for experiments, we did not disable Intel Turbo Boost. In fact, our CPUs run about 20% faster in single-threaded mode than in highly parallel situations. Thus, our single-thread baselines take less time than they should. But this will only affect our speedup results adversely since single-thread baselines are the numerators in speedup calculation.

VI. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed algorithm empirically and compare it with other asynchronous parallel SGD algorithms on multicore processors. All the experiments are performed on a 40-core quad-socket Intel Xeon E7-4860 machine where each socket has 10 cores. We ran experiments on the linear SVM task with the objective function specified in (2). For simplicity, we set the parameter $C = 1$ in all the experiments. Since our framework is developed for general SGD updates, it can be easily applied to solve other problems by changing the gradient evaluation step. Our source code is publicly available [1].

Datasets. We test the performance of HOGWILD++ on 5 standard classification datasets downloaded from LIBSVM website [2]. Details of the datasets are shown in Table I. For epsilon and rcv1 we use the standard training/testing partition provided on the LIBSVM website. There is no default partition for other datasets, so we split them randomly, 4/5 for training, and 1/5 for testing. These datasets cover a wide range of data characteristics so that we can evaluate the performance of our algorithm under different settings. rcv1 and news20 (with less than 0.16% and 0.04% non-zero elements respectively) represent two very sparse document datasets and epsilon represents a dense dataset. Moreover, the number of features vary from 54 to 1,355,191, and the number of samples vary from 16,000 to 677,399.

Competing Methods:
1) HOGWILD++ : our proposed method. Recall that we can control the size of a cluster in our algorithm, where each cluster contains several cores sharing the same local model $\boldsymbol{w}$. We use HOGWILD++-$\mathtt{c}x$ to denote the algorithm where every x cores form a cluster. For example, HOGWILD++-`c5` indicates each cluster has 5 cores, and there are $40/5 = 8$ clusters. We tested our algorithm with $c = 1, 2, 5, 10$.
2) HOGWILD!: the asynchronous SGD algorithm proposed in [17]. We use the latest release `v03a`.[3]
3) PASSCoDe[4]: the asynchronous dual coordinate descent algorithm proposed in [6]. For a fair comparison with HOGWILD!, we use the Wild (lock-free) version, which has the best speedup but is less stable.
4) DimmWitted[5]: We implemented SVM based on the DimmWitted framework with minimal changes to DimmWitted itself. We enforce thread affinities to `PerCore` and `PerMachine` approaches for better performance, and we shuffle input data after each iteration. We configured DimmWitted to use Data Sharding, with three different model replication approaches, `PerCore`, `PerNode` and `PerMachine`.

Note that in the following discussion we say an algorithm runs for one "iteration" or "epoch" after it processes all m training samples once. For all the implementations we generate a random permutation for all training sample indices and split the work to worker threads evenly. We say an iteration is done when all worker threads finish the work allocated to them. This way of splitting work is called "Data Sharding" in DimmWitted.

Tuning parameter selection. As proposed in [17], the step size η is reduced by a factor γ after each iteration. For each dataset, we use the same step size parameters η_0 (initial step size) and γ (decay factors) for HOGWILD!, HOGWILD++ and DimmWitted. We obtained optimal parameters η_0^*, γ^* using grid search on single-threaded HOGWILD! (in fact, all three algorithms are essentially the same in single-threaded mode). Because HOGWILD++ and DimmWitted need to synchronize multiple model replicas, they usually need more iterations to converge. Thus, for HOGWILD++ and DimmWitted we set $\gamma = \sqrt[M]{\gamma^*}$, where M is the number of model replicas. PASSCoDe does not need to select a step size because it is a coordinate descent based algorithm. HOGWILD++ has an additional parameter τ_0 to set the delay for passing the token to the next cluster. In experiments we found that our algorithm is not sensitive to this parameter, so we set $\tau_0 = 64$ when $c = 10$, and $\tau_0 = 16$ when $c = 1, 2, 5$ for all datasets.

A. Efficiency

The main objective of this work is to develop an algorithm that can scale to use all the cores of a NUMA machine effec-

[1]http://huanzhang12.github.io/hogwildpp/
[2]http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/
[3]http://i.stanford.edu/hazy/victor/Hogwild/
[4]http://www.cs.utexas.edu/~rofuyu/passcode-icml15-exp/
[5]https://github.com/HazyResearch/dimmwitted

TABLE I
DATASET STATISTICS AND PARAMETERS USED IN EXPERIMENTS

Dataset	size (non-zero elements)	# training samples	# test samples	# features	sparsity (nnz%)	initial step size η_0	step decay γ
news20	139.1 MB	16,000	3996	1,355,191	0.0336 %	0.5	0.8
covtype	115 MB	464,810	116,202	54	22.12 %	5×10^{-3}	0.85
webspam	459.0 MB	280,000	70,000	254	33.52 %	0.2	0.8
rcv1	715.5 MB	677,399	20,242	47,236	0.155 %	0.5	0.8
epsilon	14.9 GB	400,000	100,000	2,000	100 %	0.1	0.85

tively. So, the first experiment is to compare the performance of all the implementations on 40 cores. Figure 2(a), 3(a), 4(a) and 5(a) show the primal objective function values in terms of wall clock time, and Figure 2(b), 3(b), 4(b) and 5(b) show the prediction accuracy on the test set in terms of wall clock time. To avoid clutter and ensure clarity only HOGWILD++-c5,10 is shown in this set of figures. A more detailed comparison with different cluster size c is presented in Section VI-C.

Based on these results, we make the following observations:

- Our proposed algorithm converges much faster than HOGWILD! on most of the datasets.
- DimmWitted-PerNode and HOGWILD++-c10 both have 4 model replicas, and the differences between the two are primarily synchronization strategies (averaging vs passing along a ring). Our experiments shows that HOGWILD++-c10 is faster than or on-par with DimmWitted-PerNode. This indicates that our model synchronization approach performs better than simple model averaging. For certain datasets with a small feature size (like covtype), HOGWILD++-c5 and HOGWILD++-c2 could be faster than both HOGWILD++-c10 and DimmWitted-PerNode. DimmWitted-PerMachine shows almost identical results as HOGWILD! because they are essentially the same algorithm, so we omit it in all our figures.
- news20 is the only dataset where our algorithm performs similar to HOGWILD!. The main reason is that news20 has a very large feature size and is very sparse, so HOGWILD! almost has no memory conflicts when accessing the shared model parameters. Also, due to its sparsity, the SGD updates to $\boldsymbol{w}$ can be conducted very efficiently, while HOGWILD++ needs more time to synchronize a dense model vector $\boldsymbol{w}$ with $1,355,191$ elements. For the same reason, DimmWitted does not perform well on news20: both PerCore and PerNode approaches are slower than HOGWILD!. For very high-dimensional sparse datasets, $c = 40$ (or the total number of cores) can be used, making HOGWILD++ equivalent to HOGWILD!.
- PASSCoDe-Wild cannot converge to global minimum due to the conflicting writes to the shared model vector. In [6], this phenomenon is acceptable using 10 cores. However, when scaling to 40 cores on multiple sockets, write conflicts become a severe problem. Since the convergence point of PASSCoDe-Wild is too far from the solution of rcv1, covtype and webspam, we omit PASSCoDe in those figures.
- HOGWILD! does not converge to the optimal solution in covtype using the given step size and decay (which is optimal for single-threaded HOGWILD!), probably because the high number of model write conflicts reduce its efficiency. Our algorithm converges well on all the datasets using the same step size and decay that are optimal for the single-threaded version. We do not need to tune these parameters for a different number of threads or a different cluster size c. This suggests that our algorithm is robust and suitable for different parallel scenarios.

B. Scaling in Number of Cores

For the second experiment, we vary the number of cores from 1 to 40, and plot the speedup of all the competing algorithms. For each method, we measure the speedup by the following criterion:

$$\text{speedup} = \frac{\text{iteration time taken by the method with 1 threads}}{\text{iteration time taken by the method with } p \text{ threads}}.$$

Note that both HOGWILD! and HOGWILD++ share the same code base, so the performance of the serial version is the same. The experimental results are presented in Figure 2(c), 3(c), 4(c) and 5(c). The results also show the impact of cluster size c on the speedup with HOGWILD++ . Based on these results, we make the following observations:

- The speedup of HOGWILD! and PASSCoDe suddenly drops when the number of threads is greater than 10, which is the number of cores in a single socket. This indicates that they are not able to scale to multiple sockets. In fact, when using multiple sockets their speedup is often worse than using 10 threads, and sometimes even worse than using one thread (speedup < 1). In comparison, HOGWILD++ has almost linear speedup with increasing number of cores, across multiple sockets.
- We also compare HOGWILD++ using different sizes of clusters. When using one core per cluster (HOGWILD++ - c1), the algorithm has the best speedup, while the speedup decays slightly when we increase the number of cores per cluster. The speed of convergence with different cluster sizes is shown in Section VI-C.
- Clearly, DimmWitted does not scale very well. Even though DimmWitted-PerCore has the same number of model replicas as HOGWILD++-c1 , it does not scale as well. We believe this is due to the model averaging synchronization strategy used by DimmWitted. We also identified serial sections and unnecessary overheads in DimmWitted that limit its scalability. We will discuss issues we found in DimmWitted in detail in section VI-D.

C. The parameter c in HOGWILD++

As discussed in the previous section, the parameter c that controls the number of cores per cluster is very important in HOGWILD++ , so here we experimentally study the impact using three datasets: covtype, rcv1 and epsilon. We run HOGWILD++ with $c = 1, 2, 5, 10$, and plot the convergence

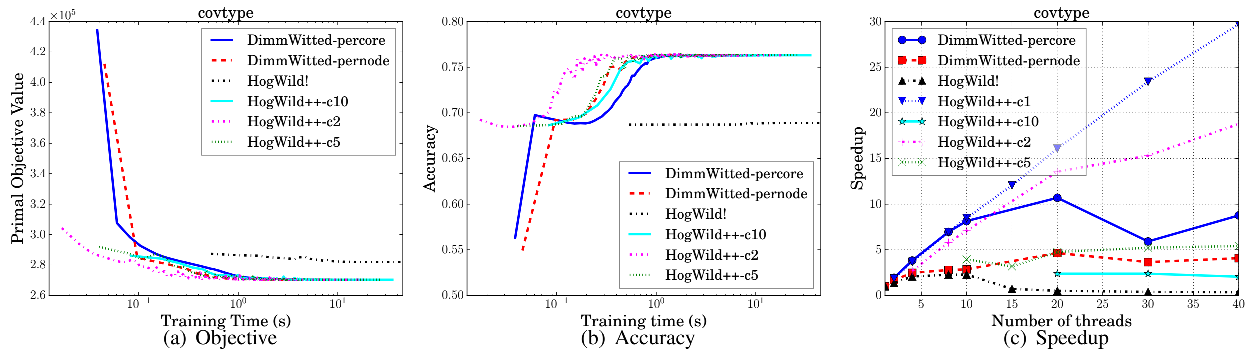

Fig. 2. covtype dataset

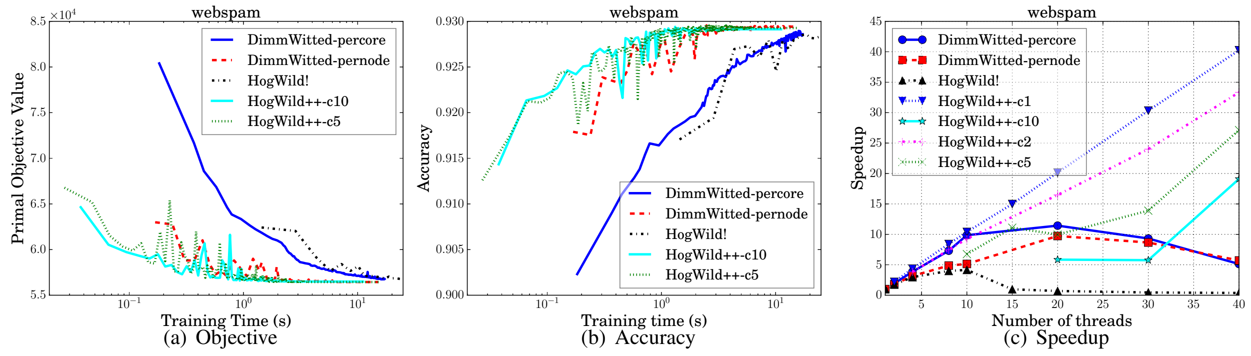

Fig. 3. webspam dataset

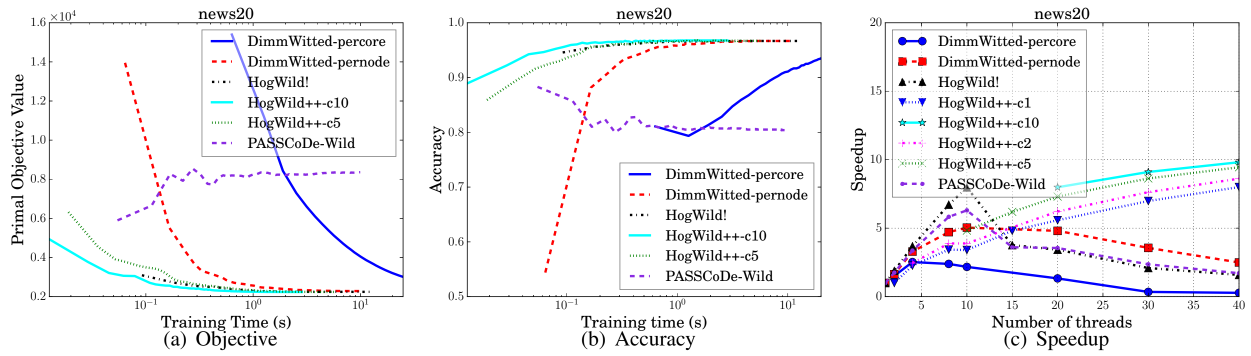

Fig. 4. news20 dataset

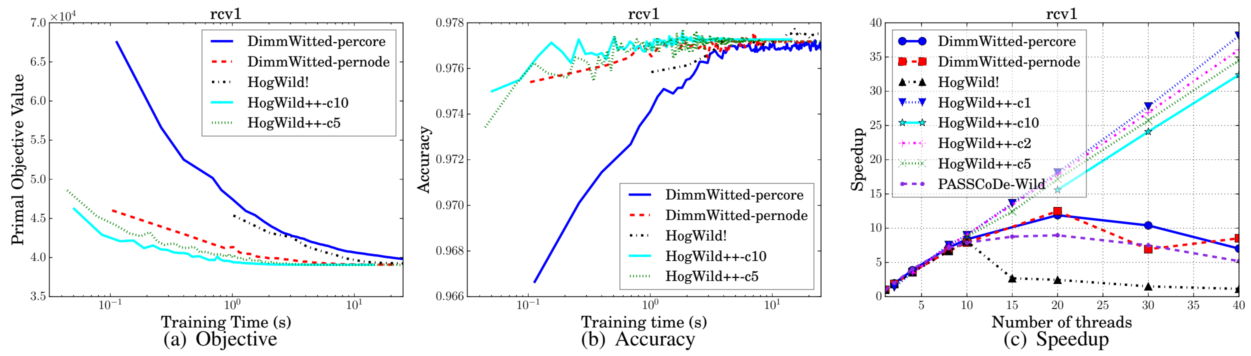

Fig. 5. rcv1 dataset

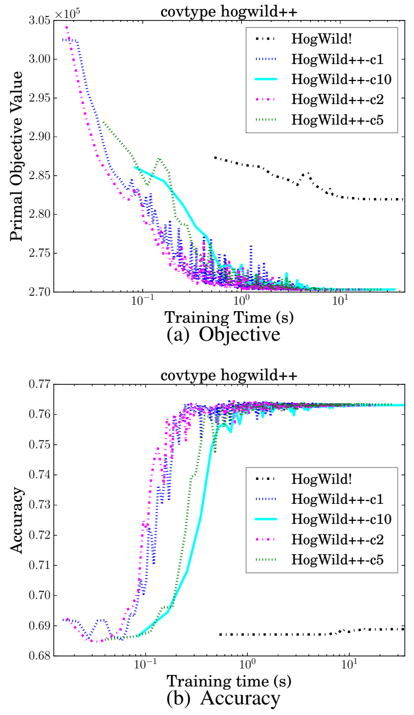

Fig. 6. covtype with different cluster size c

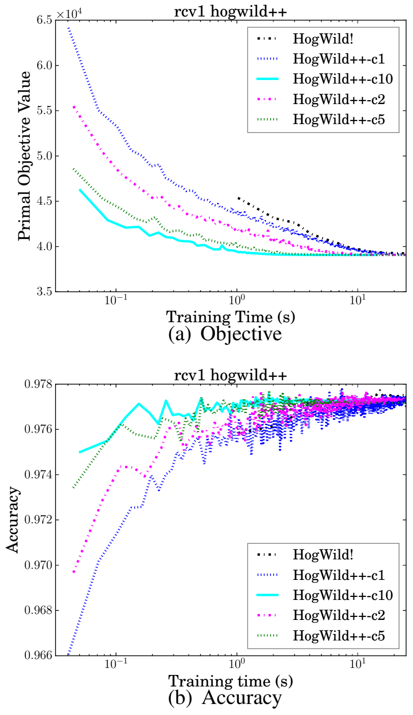

Fig. 7. rcv1 with different cluster size c

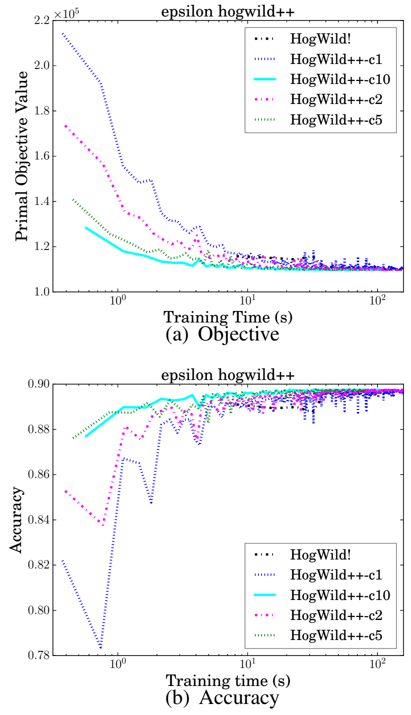

Fig. 8. epsilon with different cluster size c

speed in terms of objective function value and prediction accuracy. Note that HOGWILD! is the same case as $c = 40$, so we also include it in the results for comparison. The results are presented in Figure 6, 7 and 8. One can draw the following conclusions from these results:

- There is a tradeoff problem in choosing c. A larger c often leads to faster descent per iteration, but each update becomes slower due to the memory sharing problem within each cluster. In most of the datasets (such as Figure 7 and 8), $c = 10$ achieves the best performance. But on covtype dataset (Figure 6), $c = 2$ instead of $c = 10$ is the best.
- For all the values of c that we investigated, we can see that HOGWILD++ is better than HOGWILD! on covtype, rcv1 and epsilon datasets. Based on Figure 6(a), 7(a), 8(a) and 2(a), 3(a), 4(a), 5(a), we would recommend using $c = 10$ (or the number of cores per socket) on datasets with unknown characteristics, which minimizes expensive inter-socket communication while keeping the number of model replicas as small as possible. This is the same conclusion as made in DimmWitted [24], where the PerNode approach works best on most datasets. But certain datasets, especially datasets with a small feature size (like covtype), do benefit from a smaller cluster size c. HOGWILD++ enables us to make a finer trade-off between hardware efficiency and statistical efficiency by selecting a good c for a given dataset.

D. Issues in DimmWitted

In this section we will discuss issues we discovered in DimmWitted that impact its performance and accuracy, and try to understand why HOGWILD++ outperforms DimmWitted.

Model Averaging. We found that when PerCore approach is used, each thread does not communicate with other threads to exchange model information. Instead, models from all the threads are averaged after each iteration and returned to the caller. However, this averaged model is never written back to each worker thread. This is equivalent to training multiple models independently and using the average as the final model. Because threads do not communicate model information with each other, this approach is very inefficient statistically, and it probably explains why the convergence of PerCore is the slowest on every dataset.

In [24], authors claim that the optimal way for synchronizing models is to "*communicate as frequently as possible*", given the sufficient QPI Interconnect bandwidth. Indeed, in the PerNode approach, DimmWitted launches one additional communication thread per node at the beginning of each iteration, and this thread is responsible for averaging models from all nodes and updating its own model. However, this communication thread just computes the average *once* and quits. We are not sure whether it was just an implementation error or it is intentional. For datasets with a large number of training samples (like rcv1), just communicating once during an entire iteration might be insufficient. HOGWILD++ does not have this problem because the model information is constantly passing among clusters so it can adapt to different sample sizes. Also, model averaging in DimmWitted requires global communication for synchronization and thus its scalability is limited in multi-socket or distributed situations.

Data Sharding. We found that in DimmWitted and its provided Logistic Regression (LR) example [6], while data sharding is enabled, the training data samples are not permuted. Consequently, when PerCore approach is used, since each model is trained independently (as described in the previous paragraph), each thread trains a model specifically for a subset of the training examples. When computing the loss and accuracy as shown in the example LR application, each thread uses its own model instead of the averaged model, making the training accuracy abnormally high. In our experiment, we fixed this problem by shuffling training samples after each iteration, and creating an additional DimmWitted engine instance for evaluating loss using the averaged model.

Overhead and Serial Bottleneck. DimmWitted uses `std::async` to launch new worker threads at the beginning of each iteration, and destroys them at the end of each iteration. Each thread also has to be bind to the correct core or node each time an iteration starts. The use of `std::async` makes the code simple and elegant, albeit the overhead of making and destroying threads cannot be ignored. Also, in the implementation of both PerCore and PerNode, after all worker threads finish their work for the current iteration, the main thread averages all models sequentially. We understand this implementation follows the map-reduce design pattern, but this serial section is not negligible, especially on the dataset with a large model (news20). In HOGWILD++ , we use the model from the last thread holding the token as the latest model, instead of averaging all models. When the number of threads is large, the threading overhead and serial section make DimmWitted show poor speedup.

VII. Conclusion and Future Work

This work is motivated by the observation that existing asynchronous SGD algorithms cannot scale to many cores or multiple NUMA sockets in a single machine. To overcome this issue, we develop a new asynchronous SGD algorithm, HOGWILD++ . The main idea is to replicate the model parameter into several copies, and the communication is done in an asynchronous but carefully designed way to ensure the convergence. We show our algorithm can get almost linear speedup on a 40 core NUMA machine, while existing algorithms show very limited speedup. We leave the evaluation of the proposed algorithm on other network topology in distributed computing environments as future work. We also plan to extend the proposed techniques to other iterative solvers, such as coordinate descent or ADMM.

Acknowledgments. The authors are grateful to the XSEDE startup resources.

References

[1] R. Anderson. Parallel algorithms for generating random permutations on a shared memory machine. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '90, pages 95–102, New York, NY, USA, 1990. ACM.

[2] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in Neural Information Processing Systems 20*. 2008.

[3] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. *ACM Transactions on Intelligent Systems and Technology*, 6:2:1–2:24, 2015.

[4] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[5] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13:165–202, 2012.

[6] C.-J. Hsieh, H. F. Yu, and I. S. Dhillon. PASSCoDe: Parallel ASynchronous Stochastic dual Coordinate Descent. In *International Conference on Machine Learning(ICML),*, 2015.

[7] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[8] Yann Le Cun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524. Springer Verlag, 1998.

[9] Mu Li, David G Andersen, Alex J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[10] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 661–670, 2014.

[11] X. Lian, H. Zhang, C.-J. Hsieh, Y. Huang, and J. Liu. A comprehensive linear speedup analysis for asynchronous stochastic parallel optimization from zeroth-order to first-order. In *NIPS*, 2016.

[12] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimizations. In *NIPS*, 2015.

[13] J. Liu, S. J. Wright, C. Re, and V. Bittorf. An asynchronous parallel stochastic coordinate descent algorithm. In *ICML*, 2014.

[14] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G. Dimakis, and Constantine Caramanis. Frogwild!: Fast pagerank approximations on graph engines. In *PVLDB*, 2015.

[15] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S Müller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *18th International Conference on Parallel Architectures and Compilation Techniques, 2009. PACT'09.*, pages 261–270. IEEE, 2009.

[16] Daniel Molka, Daniel Hackenberg, Robert Schone, and Wolfgang E Nagel. Cache coherence protocol and memory performance of the Intel Haswell-EP architecture. In *2015 44th International Conference on Parallel Processing (ICPP)*, pages 739–748. IEEE, 2015.

[17] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems 24*, pages 693–701, 2011.

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

[19] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: primal estimated sub-gradient solver for SVM. In *Proceedings of the Twenty Fourth International Conference on Machine Learning (ICML)*, 2007.

[20] Michael E Thomadakis. The architecture of the Nehalem processor and Nehalem-EP SMP platforms. *Resource*, 3:2, 2011.

[21] E. Xing, Q. Ho, W. Dai, J. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 2015.

[22] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In *WWW*, 2015.

[23] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, S.V.N. Vishwanathan, and Inderjit S. Dhillon. NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. In *International Conference on Very Large Data Bases (VLDB)*, 2014.

[24] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *Proceedings of the VLDB Endowment*, 7(12):1283–1294, 2014.

[25] H. Zhang and C.-J. Hsieh. Fixing the convergence problems in parallel asynchronous dual coordinate descent. In *ICDM*, 2016.

[26] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging SGD. In *NIPS*, 2015.

[6] https://github.com/HazyResearch/dimmwitted/blob/75724bf4932f94adc661b9181b26af26abf6c93e/application/dw-lr-train.cpp