Scalability of Distributed SGD SVM on a Cacheless Migrating Thread Architecture

ABSTRACT

Stochastic Gradient Descent (SGD) is a valuable algorithm for large-scale machine learning, but has proven difficult to parallelize because of communication and memory access issues on conventional architectures. The HogWild series of mixed logically distributed and physically multi-threaded algorithms overcomes these issues for problems with sparse characteristics by using multiple local model vectors with asynchronous atomic updates. While this approach has proven effective for several reported examples, there are others, especially very sparse cases, that do not scale as well on conventional systems. New and emerging architectures are being developed which are designed to handle irregular problems such as SGD while achieving improved scalability.

In this paper we implement SGD Support Vector Machine (SVM) on a cacheless migrating thread architecture using the Hogwild algorithms as a framework. Our implementations on this novel architecture achieved up to 70X speedup improvement over the conventional system when NUMA aware allocations were used, and up to 246X speedup over the conventional system when data and model vectors were naively allocated across the address space on each system. Furthermore this increase in performance was gained without any significant alterations to the conventional implementation for use on migrating threads.

CCS CONCEPTS

• **Hardware → Emerging architectures**; • **Computing methodologies → Shared memory algorithms**; *Machine learning approaches*; *Classification and regression trees*; • **Theory of computation →** *Massively parallel algorithms.*

KEYWORDS

emerging architectures, irregular applications, machine learning

ACM Reference Format:

. 2021. Scalability of Distributed SGD SVM on a Cacheless Migrating Thread Architecture. In . ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Inferencing via **Machine learning** (**ML**) is often straightforward to perform efficiently, especially by purpose-built hardware (cf. Google's TPU) [6]). However, learning is far more complex. It typically involves reading large numbers of training examples, performing a small number of operations on each, updating an evolving solution, and repeating. In addition, much training data is *sparse*, and this sparsity is often very irregular from sample to sample.

This paper focuses on parallel execution of one such learning algorithm, **Stochastic Gradient Descent** (**SGD**) as applied to **Support Vector Machine** problems, and implemented on a novel emerging architecture. The implementations we discuss in this paper are based on the "HogWild" algorithms [1, 14, 16] that were designed with sparse data sets in mind. Sparsity in this case is where individual training records may all have the same logical size but may have only a few components that are non-zero.

Good speedup has been reported in the past for problems with moderate sparsity and moderate levels of multi-threaded parallelism. Reported speedup, however, isn't as efficient for the sparsest of data sets, largely because of major inefficiencies in modern architectures when faced with memory-bound problems with significant irregular communication. Unfortunately such sparsity is common in many real large applications such as recommender systems and social media applications.

Prior work on parallel SGD implementations have yielded similar results. The codes DisBelief and Downpour [4], for example, saw only moderate speedups for dense problems solved on deep neural nets: 2.2X on 8 nodes for moderate speech problems, and 12X on 81 node systems for larger images. Other work has focused on relatively sparse SVM [2] but has not reported comparable speedup.

Previous studies evaluated the effects of extreme sparsity on a similar irregular problems such as **Sparse Matrix Vector Product** (SpMV) when executed on a variety of architectures [9–11], and real-time streaming [12]. All have had similar results on modern conventional architectures: attempting strong scaling on sparse data is very tough, and sometimes even counter-productive - more parallelism often *lowers* performance. In fact the only architecture evaluated which exhibited sustained positive scaling in previous studies was an emerging shared memory parallel heavily multi-threaded architecture that supports migrating threads [11–13].

Emerging architectures such as the Lucata migrating systems stray from the traditional process/thread model which lies at the heart of the overwhelming majority of modern conventional systems, even those which seek to bolster their computational power via the use of accelerators such as GPUs. The migrating thread system allows for the use of 3 levels of system parallelism: node, nodelet, and thread. Furthermore because threads can migrate freely throughout the entirety of the address space provides for overlapping of thread migration with computation. Lastly cacheless architectures are naturally coherency-free, therefore the performance degradation from repeated cache invalidations can be avoided.

Given the irregular nature of SGD/SVM on real world data sets we chose to implement several versions of SGD/SVM in the style of "Hogwild" for the Lucata migrating thread system. The resulting scaling results are significantly better than previously reported, especially for the sparsest of cases.

The main contributions of this paper are:

ICPP '21, August 09–12, 2021, Chicago, IL

- An implementation which simplifies the hogwild++ algorithm by eliminating the need for communication between clusters when using the globally accessible address space of the migrating thread architecture.
- A second implementations which evaluates an alternative data and workload partitionings in an effort to decrease unnecessary thread migrations (communication) where possible.
- A demonstration that we can obtain significant performance improvement with minimal alterations to existing conventional implementations
- An analysis and comparison between migrating threads and conventional architectures for all implementation variants is provided.

Section 2.1 reviews SGD as applied to the SVM problem. Section 2 discusses the background and prior work used as a framerwork for this study. Section 5.2 discusses the alternative architecture. Section 4 discusses SGD SVM on migrrating threads. Section 5.2 discusses the alternative architecture. Implementation details for our test codes can be found in Sec. 5.1. Sections 8 and 9 evaluate performance and scaling respectively.

2 BACKGROUND

2.1 Reference Problem

The goal of a typical ML problem is to analyze a set E of **training examples** (each a vector with F **features**), and determine an F-element **model vector** $\hat{\omega} = [\hat{\omega_1}, ...\hat{\omega_F}] \in R^F$ that can be used to predict something about a previously unseen feature vector, such as what class it may lie in. This model vector is one that minimizes some **objective function** $\hat{f}(\omega)$, often expressed as a sum over an "error function" applied to each example. Problems that fall in this class include: SVM, matrix completion, and graph partitioning.

2.1.1 Support Vector Machine (SVM). The SGD variant analyzed here is for the **Support Vector Machine** (**SVM**) problem, taken from [1]. In such a problem, samples and observations are vectors of features, and are to be divided into one of two classes. The desired **support vector** ω determines this split by taking the inner product between itself and an observation, and looking at the sign of the scalar that results. The training set in this case is a set of pairs (z_e, y_e) where z_e is a vector of feature values, and y_e is either +1 or −1, depending in which of the two classes sample e falls. The error function f_e is the **hinge loss**: $max(0,\ 1-y_e d_e)$ where $d_e = \hat{\omega}^T z_e + b$ is the "inference" inner product plus a bias term b. The product $y_e d_e$ is thus positive if $\hat{\omega}$ is a good predictor, and negative otherwise, For a model that predicts incorrectly, the loss is positive. For a correct prediction there are two cases: correct but "close" ($1 > y_e d_e > 0$) and "very correct". The latter case makes the loss zero; the former returns a small loss that is an encouragement to "do better."

The objective function is thus the sum of the error functions applied to all samples, with an extra term $\lambda||\hat{\omega}||_2^2$ added on to keep the magnitude of the model vector small.

The gradient $\nabla\hat{f}(\omega)$ is an n-element vector where the i^{th} component can be approximated as:

$$\nabla\hat{f}(\omega)[i] = (\hat{f}(\omega + hb_i) - \hat{f}(\omega))/h \quad (1)$$

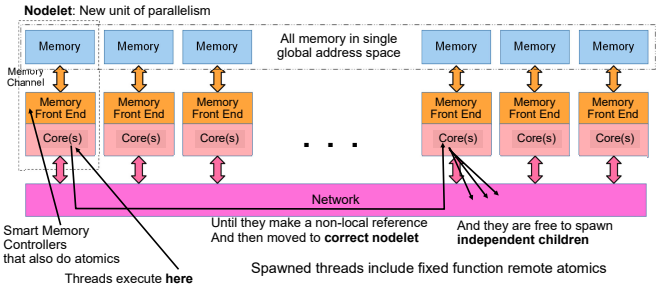

Figure 1: The Migrating Thread Architecture.

where h is a small number and b_i is the n-element basis vector where only the i^{th} component is non-zero (a "1").

2.1.2 Stochastic Gradient Descent (SGD). The goal of a typical ML problem is to analyze a set E of **training examples** (each a vector with F **features**), and determine an F-element **model vector** $\hat{\omega} = [\hat{\omega_1}, ...\hat{\omega_F}] \in R^F$ that can be used to predict something about a previously unseen feature vector, such as what class it may lie in. This model vector is one that minimizes some **objective function** $\hat{f}(\omega)$, often expressed as a sum over an "error function" applied to each example. Problems that fall in this class include: support vector machines, matrix completion, and graph partitioning.

2.2 Migrating Thread Architecture

The key issues with implementing sparse algorithms such as SpMV or SGD SVM are the large number of non-sequential memory accesses, the significant number of atomic memory operations (especially to remote memory), and the cost of transferring partial results or updates between semi-independent computation streams.

A migrating thread architecture [7] has features that help with all theses issues. It is one where all memory is in a single shared address space and where any thread in any physical core can have load/store access to any location. What is different is that when an access is made to a non-local location the underlying hardware, not software, actually *moves* the state of a thread to a core close to that memory for execution. There are no local cached copies of memory that have to be managed via expensive and time-consuming cache coherency protocols.

Fig. 1 diagrams such an architecture as implemented by Lucata[1] [5]. The basic unit, a **nodelet**, is a memory module, its memory controller and some multi-threaded cores. All the memory in the collection of nodelets reside in a common address space. A network connects all nodelets.

Of particular importance is that each nodelet's memory controller includes logic that allows it to perform a rich set of update operations atomically[2] against locations in its memory.

A thread can cheaply spawn additional threads who then live existences independently of the parent. These threads can be of several types. First are full threads capable of executing arbitrary

[1] Previously Emu Solutions Inc.

[2] An atomic memory operation is one where some location is read, some computation performed on it, and a write back is done to that same location in a fashion where it is guaranteed that no other access can modify the location in the time between the read and the write.

code sequences. Second special purpose threads that can only perform some dedicated operations such as a store or an atomic update. These latter thread types work directly with the memory controllers at the target nodelet to perform remote atomic operations without moving the whole thread state. Atomic operations not possible with such threads can be implemented via a rich set of compare-and-swaps implemented by medium weight migrating threads.

The current prototype used in this study is housed at Georgia Tech's Rogues Gallery testbed[3]. It has 64 nodelets packaged 8 to a board in an FPGA. Each nodelet has 8GB of memory, a 175MHz multi-threaded core, and a RapidIO-based network. Each core can maintain up to 64 thread states that are processed in a round-robin sequence. A dual core POWER microprocessor on each board runs Linux, manages a local SSD, and initiates migrating threads into the system. Each nodelet's memory bus delivers 8 bytes per access (rather than 64 bytes as on conventional systems), meaning that for problems with a high percentage of memory accesses that have little spatial locality[4], the usable bandwidth from them approaches 100%. The nodelet logic on each board is implemented via an FPGA. Given limited space on these FPGAs, intrinsic support for floating point is currently not included in the remote atomics.

The programming tool chain is based on Cilk [3], an extension of C with a prefix to function calls to spawn new threads, a sync primitive to wait for completion, and a parallel forall to have a set of independent threads cooperate. Supported intrinsics include a rich set of atomic operations.

An enhanced, larger system with more nodes and cores is currently under development.

3 RELATED WORK

Many real problems involve huge training sets, both in the number of samples $|E|$ and the number of features per sample F. Both can quickly range into the millions or more; therefore, parallel versions that scale well are essential. Unfortunately, simple approaches bottleneck around memory issues such as inter-socket coherency traffic and false sharing. The obvious multi-threaded algorithm handles different examples concurrently. However, if all updates into a shared model vector from each example must be done atomically for each working solution, computing serializes around locking and unlocking access to that solution. This serializes the solution.

Prior work on parallel SGD implementations have seen such issues limit efficient parallel scalability. The codes DisBelief and Downpour [4], for example, saw only moderate speedups for dense problems solved on deep neural nets: 2.2X on 8 nodes for moderate speech problems, and 12X on 81 node systems for larger images.

The exception is when the samples are very *sparse*, that is when most of the features in a training example are not relevant or not available. In this case, the interleaving of partial updates to the overall solution may be acceptable, because each training sample typically affects a small subset of the solution. Thus, each example updates a relatively different subset of the model vector, and doing so in parallel is likely not to significantly lengthen the number of epochs needed for convergence. This likely independence of update subsets also means that locking the entire solution during an update is unnecessary, as long as individual model vector elements are updated atomically.

3.0.1 Hogwild! The *HogWild!* algorithm [1] was the first of a series of algorithms (summarized in Table 1) to employ this technique. The original paper discusses the sparsity conditions under which such update independence is possible. Decent speedup was reported when using a small number of threads on data sets with 10s of thousands to millions of features per sample but extreme sparsity (as little as 0.002%). However, coherency traffic limited the maximum number of threads that were useful to the number of cores on one socket. This in turn limited the maximum speedup.

The *DimmWitted* algorithm [15] performed a careful comparison of several variants of HogWild!s. Tradeoffs included whether to store examples by rows or columns, how the set of training examples should be replicated and blocked, and how many "local" training vectors were reasonable. The best combination achieved about 2.3X improvement over HogWild! for the rcv1 data set, but parallelism was limited to two sockets of only six cores each.

The *BuckWild!* algorithm [14] reduced the precision of individual features to as low as 8-bits, allowing memory fetches to return more features per access. One data set saw 2.5 times the performance of HogWild! at 12 cores.

The final algorithm in Table 1 was *DMS*, [15]. This study was much like DimmWitted in that it surveyed a variety of options. It was different in that it assumed a conventional distributed cluster with Infiniband interconnect. Variations included the number and placement of model vectors and variations in block size. Synchronization of local model vectors was via a global AllReduce done after blocks of examples were processed in each node. Speedup here appears to peak at about 5X over a single core in a system with 32 total cores. The limiting factor appeared to be inter-node bandwidth, much like what we found in our SpMV studies.

3.0.2 Hogwild++. The *HogWild++* algorithm [16] assumes a NUMA[5] architecture and goes even further in reducing the effects of sharing between threads, especially that which causes invalidation traffic, without causing major increases in convergence time. The algorithm divides computation into logical **clusters** that have their own local model vector, and includes a step to propagate local changes to other clusters. Having a pair of working vectors then means that a cluster can determine which features have changed since the last token passing, and then just sending updates for just those changes to its neighbors. This greatly reduces inter-cluster traffic when the number of features is large and the sparsity significant.

Each cluster is a multi-core implementation of the original asynchronous HogWild! algorithm, but where all cores are on the same physical socket. In a multi-socket system, the computations within a cluster thus never cross a socket boundary, so that none of the corrosive cross-socket invalidation traffic is created. Only the memory channels tied to a single socket are devoted to a particular cluster.

[3] https://crnch.gatech.edu/rogues-emu

[4] This means for example that out of the data line accessed from memory only a small percentage is used.

[5] NUMA = "Non Uniform Memory Access" where a deep hierarchy of caches often make memory accesses highly variable in access time, and is typical of modern multi-core chips and multi-socket nodes.

Algorithm	Refs	Type	Parallel Model	Key Feature	Scaling	Limiter
HogWild! 2011	[1, 8]	Sparse	NUMA multi-core	single model; async update via atomic operations	rcv1: 4.5X@10 cores	cache sparsity, coherency traffic
DimmWitted 2014	[15]	Sparse	NUMA Multi-socket multi-core	Row access, one model per node	rcv1: 2.3X Hogwild! at 2 node, 12 cores	N.A.
BuckWild! 2015	[14]	dense	Same as Hogwild!	Hogwild! + short precision	rcv1: 5X HogWild! at 12 cores & 8-bit precision	Same
HogWild++ 2016	[16]	Sparse	NUMA multi-socket, multi-core	multiple local models, round robin model sync	news20: 9.5X@4x10 cores	update process
DMS 2019	[2]	Dense	Distributed cluster	Local models, partitioned dataset, global sync	≈ 5X @32 processes and large block sizes	model communication

Table 1: SVM via SGD Algorithmic Variations. The "Scaling" column reflects the best reported parallel speedup; either a speedup measured against a single core running the algorithm or a speedup over the original HogWild! for specific data set.

	Training		Per Sample			Best Configuration					
Data set	Samples S	Sparsity	Features F	Non-Zeros F'	Speedup/ Efficiency	Cores / Cluster	Clusters	τ_0	η_0	γ	Approx. Accuracy
news20	16,000	0.034%	1,355,191	455	9.5/24%	10	4	16	0.5	0.8	96%
covtype	464,810	22.12%	54	12	30/75%	1	40	16	0.005	0.85	76%
webspam	280,000	33.52%	254	85	40/100%	1	40	16	0.2	0.8	93%
rcv1	677,399	0.155%	47,236	73	38/95%	1	40	16	0.5	0.8	97.5%

Table 2: SVM Training Data set Characteristics from [16].

Table 2 summarizes reported results for SVM using the HogWild++ algorithm[6]. The F' column gives the average number of non-zeros per sample, and is the feature count F times the sparsity. The τ_0 column is the minimum number of samples that must be processed by a cluster before the cluster will accept a token[7]. Of these data sets the most interesting is *news20* as it is the most sparse and the lowest speedup. It also has by far the largest number of features, meaning inter-cluster traffic is liable to be more significant.

In this table, "cores" means the same as "threads," and the "Best Configuration" columns describe the division of cores into clusters. "Speedup" is measured against running on a single core/single thread. In all cases, speedup was maximized when using all 40 cores in the system (4 sockets of 10 cores each).

4 SGD SVM ON MIGRATING THREADS

Hogwild! saw performance degradation from increase coherency traffic as the number of cores and sockets was increased on a shared memory system. Hogwild++ was designed to mitigate this behavior by eliminating cross socket coherency traffic at the expense of requiring explicit inter-cluster model vector synchronization. It achieved improved performance over Hogwild! but saw degradation for the sparsest of data sets as core counts increased. With this in mind we used the Hogwild algorithms as a framework for computing SGD SVM on migrating threads.

Data and model vectors are dispersed among all nodelets in the system configuration with threads performing training being able to migrate freely. For training we treat all nodes and their associated

Algorithm 1 Lucata SGD SVM
As executed for one epoch
ω: model vector, η_0: initial step size, γ: initial step decay
deg: feature degree vector, e: count of epochs so far, E: Total epochs
S: Total Samples, s: current sample, T: Thread count
$\nabla f_s(\omega)$: gradient of f_s evaluated at ω.

```
e = 0 {Initialize epoch count}
ω_initial = 0 {Initialize model vector}
repeat
    for t in T do
        cilk_spawn   TRAIN(t, e)
    end for
    cilk_sync
until e ≥ E

TRAIN(t, e):
s = t {Initialize thread sample id}
while s! = S do
    Evaluate training sample with index s
    if ∇f_s(ω) < 1 then
        ω_tmp = ω + (η_0 γ^e f_s)
        ω = ω_tmp(1 − η_0 γ^e deg)
    else
        ω = ω(1 − η_0 γ^e deg)
    end if
    s = s + T{Increment iteration count}
end while
```

[6]The best configuration reported in Table 2 comes from the speedup figures, not the text, as there seems to be a difference.
[7][16] reports using τ_0 of 64 when the number of cores per cluster was 10, and 16 otherwise.

nodelets as a single system much like Hogwild!. The Hogwild++ equivalent would be that of a single cluster across all nodelets.

The training loop is nearly identical to the original Hogwild++ C/C++ implementation [8], with the notable exception that we do not perform any updates in any way, since only a single "cluster" exists. Additionally in contrast to Hogwild++ we only use a single model vector to store model weights during training.

In Alg. 1 shows the high level algorithm for SGD SVM on migrating threads. The loop on line 4 runs for a given number of epochs, with each epoch training over the all samples in the training data. After the start of a new epoch *cilk_spawn* is used to spawn T threads throughout the system with each thread subsequently entering the training loop. Each sample is evaluated by only 1 thread per epoch with sample evaluation and model updates being calculated in the exact manner as the supplied Hogwild++ C/C++ code.

Thread migrations occur when attempting to access memory on a remote node/nodelet however since this behavior is performed directly in hardware this mechanism is abstracted away from the developer.

5 IMPLEMENTATION

Hogwild and Hogwild++ are intended to solve two key issues: shared data race conditions and cross socket coherency traffic. To accomplish this Hogwild++ performs logically distributed computation within a shared memory environment by subdividing into "cluster," each with a separate set of cores. The code necessary to govern valid partitioning, as well as the necessary inter-cluster updates between logical clusters makes for a complicated code base with only modest scalability to show for it.

Lucata provides a shared memory space which is physically distributed across several nodes, allowing for the use of additional processing and memory hardware while still being treated to the user as a single shared memory machine. This allows us to implement SGD using a single "cluster" but utilize a greater system size. Furthermore only a handful of alterations to the conventional implementation are required in order to run on the Lucata architecture and achieve improved performance.

In the subsequent sections we discuss the conventional implementation of single cluster SGD for direct comparison to the Lucata implementation as well as two optimizations which provide even greater performance improvement over the base case.

5.1 Conventional Architecture

The Hogwild++ algorithm was intended to eliminate the performance penalty of cache coherency traffic seen during the Hogwild! study by subdividing the training set into disjoint clusters which run independently of one another during training. The Lucata migrating thread system is a cacheless distributed shared memory system such that even though there are multiple "nodes" we treat them as a single shared memory space. Therefore in order to compare the two architectures the base case implementations were designed to utilize the entire system without the need for problem subdivision and the associated "updates" (communication) between them.

[8] https://github.com/huanzhang12/hogwildpp

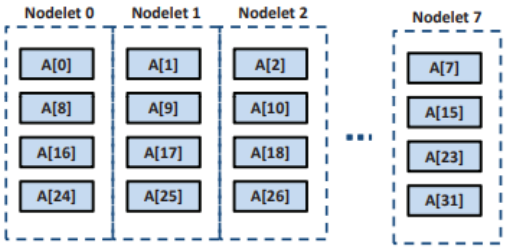

Figure 2: 1D array allocation using mw_malloc1dlong

Similar to Hogwild algoritms, several different vectors are used to store the model vector, data sets, and the feature degree counts pertaining to the current data set. Since this implementation is designed to compare against the Lucata base implementation, we create a single cluster and therefore only allocate a single vector for all the aforementioned items. This of course eliminates the need for inter-cluster updates, but introduces the race condition and cache coherency issues discussed earlier.

After memory allocation and data set population we can begin with the SGD training loop. Here an OpenMP pragma creates a parallel region in which various OpenMP threads are spawned in order to take part in the training loop for the current epoch. All threads share a set of pointers to the model and data set vectors as well as a copy of the current scaling value $\eta * \gamma^T$ where T is the current epoch value (just as in Hogwild++).

From here each thread begins training over the data set, starting with the sample id corresponding to their thread id, and proceeding through all samples with each sample only being evaluated at most once. During training all threads perform writes back to the same model vector and update it based on the computed distance between the current training sample and the current model vector, again just as is done in Hogwild++. Once all samples have been evaluated the OpenMP threads are killed and subsequent epochs are run until the max epoch count has been reached.

In addition to the OpenMP based implementation we also tested a Cilk Plus implementation which included memory allocation and data placement identical to the OpenMP version. The Cilk version utilized the Intel ICC compiler and only differed from the OpenMP version with regards to thread library, where *cilk_spawn* and *cilk_sync* rather than *#pragma omp parallel* was used.

5.2 Baseline Migrating Thread Implementation

For our migrating thread implementation we took the conventional version, based on Hogwild++, as the starting point. As mentioned previously there as very few alterations that need to be made in order to tailor existing applications to the Lucata architecture.

When not specifying NUMA specific memory allocations, conventional systems allocate contiguous regions of memory in as large and as few locations as possible, with all threads capable of accessing it via cross socket traffic. While this can certainly be done on Lucata via the use of the traditional *malloc* or the hardware intrinsic *mw_localmalloc* which allocates memory on a nodelet-local heap where the calling thread is currently executing.

Lucata is a PGAS based architecture, meaning that any thread can access any address regardless of the the node or nodelet which controls that address. Therefore to allocate data across the entire

system we utilize the **mw_malloc1dlong** method which allocates a striped 1-Dimensional array of 64-bit integers across all nodelets in the system. Fig. 2 illustrates striping elements is such that element 0 is on nodelet 0, element 1 on nodelet 1, etc. Just like for conventional code the training data, model vector, and feature degree counts are allocated with no regard as to where exactly any arbitrary piece of data is to be placed within the address space and are done using *mw_malloc1dlong*. This alteration in the code is as simple as changing the function call to match the Lucata equivalent.

At the start of each epoch t threads are spawned by calling **cilk_spawn** with each thread executing the training loop just as in the conventional implementation. In fact the training loop code for both the conventional and Lucata striped versions is identical, with any execution pattern differences originating from the differences in hardware design.

Given that the training loop code is identical for both systems the key architecture difference lies in the use of migrating threads and striped arrays. As a thread is assigned a new sample it access that samples features via indexing into the data set arrays. If the next feature does not reside on the current nodelet, the thread will migrate to the nodelet which controls that elements address. Training data is stored in a CSR like format consisting of several arrays: sample indices, sample class, feature ids, feature values.

Using *mw_malloc1dlong* iterating over data in the array will always cause migrations since the current element i may reside on the current nodelet but the $i + 1$ element will always reside on the "next" nodelet in the system. Due to this behavior, when computing a sample's features all data pertaining to any arbitrary feature is always placed on the same nodelet, however moving to the next feature in the sample triggers a migration. Therefore the total number of migrations for the Lucata striped case is equal to the number of non-zeros in the training data set.

Stores to the model vector, which has also been striped across the system, are done without the use of explicit atomic operations.

5.3 Feature Set Partitioning

In addition to the striped version we also developed a version which attempts to obtain improved performance over that of the striped case by partitioning the feature set across all nodelets in the system in an effort to reduce thread migration and index vector memory accesses. Feature partitioning is similar to the sample partitioning method used in the Hogwild++ algorithm though unlike Hogwild++ does not require the use of multiple disjoint model vectors and their reconciliation via inter-cluster updates.

5.3.1 Data Partitioning and Allocation. Training data was partitioned according to feature id and assigned to the corresponding nodelet such that each nodelet was given a subset of features equivalent to $\frac{F}{nodelets}$ where F is the size of the feature set (not total non-zeros) for the given data set and *nodelets* is the total number of nodelets present in the system configuration.

Training data is distributed throughout the system based on the feature id for each non-zero. Should a sample have more than one feature within a given range, all data pertaining to those non-zeros is present on the local nodelet allowing them to all be processed without the need for migration between each non-zero as was done in the striped implementation.

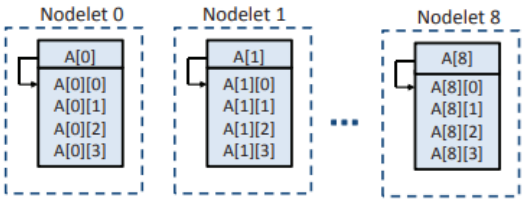

Figure 3: 2D array distribution using mw_malloc2d()

To allocate memory blocks on each nodelet while still having all addresses in use be accessible to all threads, we use **mw_malloc2d**. As shown in Fig. 3 the first dimension is a striped 1D array containing pointers to local blocks which serve at the second level of the array allocation. Memory allocation such as this is comparable to various NUMA allocation methods on a conventional system except that in the case of Lucata each nodelet is given a pointer to the first element in the top level array.

5.3.2 Execution Order. Just as with the striped version at the start of each epoch, T threads are spawned using **cilk_spawn** and subsequently enter the training loop. Within the training process, computation of any given non-zero is identical to both the conventional and striped versions. The key difference is that now due to the partitioning scheme chosen, we must check for the *start* and *stop* indices for the sample for the current nodelet.

Having the feature set distributed across all nodelets in the system means that a thread will migrate to each node in order to compute over all non-zeros within a sample. As mentioned calculation of the gradient must be completed before any model adjustments can be made. Accordingly a thread must encircle the system twice for each sample, once to compute the gradient, and again to perform any necessary adjustments to the model vector based on the calculated gradient. Upon completion of a sample the thread will be on the nodelet on which it was spawned, acquire a new sample id and continue until all samples have been evaluated.

6 EXPERIMENTAL SETUP

6.1 Conventional

For our conventional tests we are using a dual socket system with 2 AMD Epyc 7451 CPUs @ 2.66GHz each with 24 cores for a total of 48 cores. OpenMP version was compiled using GCC while the Cilk Plus version used Intel's ICC compiler.

We run each version, OpenMP and Cilk Plus, for log2 threads as well as 48 threads, with each core having at most 1 thread executing on it. Additionally core affinity if used such that as thread counts increase they populate socket 0 entirely prior to being spawned and executing on socket 1's cores.

Each data set is ran 10 times, for all thread counts, with the observed runtimes being averaged. Each test ran on the respective data set and thread/core count for 10 epochs. Thread spawning for both OpenMP and Cilk Plus implementations is included in the runtimes recorded. Accuracy computation is not included in any reported runtimes and is performed after training over 10 epochs has been completed.

It is important to note that the per epoch accuracy measurements seen in Sec. 7 were done during un-timed tests so that the gathering of this data would not impact the reported runtimes during time tests.

6.2 Migrating Threads

Tests on the Lucata system were performed we conducted using between 1 and 8 nodes for a total of 8 to 64 nodelets in a log2 fashion. Total thread counts range from 1 to 4096. The much higher thread counts allow for the 16 stage pipeline for each nodelet to have a higher chance of remaining full even though many threads will be "in flight" between nodelets in the system. The conventional system's threads remain on the core they were spawned on and therefore do not exhibit this behavior.

When using higher nodelet counts spawned threads are uniformly dispersed among the nodelets in the system configuration. This means that while the total thread count remains between 1 and 4096, the initial threads per nodelet will decrease as we increase nodelet count.

7 EVALUATION: ACCURACY

7.1 Conventional

Fig. 4 shows the per epoch accuracy, for each data set evaluated, on the conventional system. Each line represents a different number of threads with the maximum being 48 (1 thread per core on the test system). All data sets obtain approximately the same accuracy by the end of epoch 10 regardless of the thread count used. This indicates that the number of threads in our tests had little impact on the final accuracy.

That being said we can see that the accuracy during computation can vary considerably with lower initial accuracy being associated with higher thread counts. This is likely a result of lack of atomic operation use generating an increased number of race conditions as thread counts increase. Comparing to previous studies we can confirm that the accuracy percentages we observed are consistent with those seen in previously.

7.2 Migrating Thread

Similar to the conventional version, our striped and feature partitioned versions did not explicitly use atomic operations. Fig. 5 shows that the accuracy we observed on the Lucata system is comparable to that of the conventional system. The rough behavior of accuracy alterations during training are consistent between the conventional and striped implementations which was expected due to their near identical code. For Lucata however there was some variance in the accuracy at the largest nodelet and thread counts however the accuracy variance after 10 epochs was between -2% and +10%.

Accuracy measurements for the feature partitioned tests were consistent with those seen in Fig. 5 and have therefore been omitted.

8 EVALUATION: PERFORMANCE

8.1 Conventional

Performance metrics for the best performing system configuration with regards to each data set are presented in Table 3. For all data sets the OpenMP version of the conventional implementation achieved better performance that that of the Cilk Plus variant. This is likely to the additional scheduling overhead inherent to the use of Cilk thread spawns.

The conventional system has rather low time per non-zero for all data sets, with the longest time being associated with the sparsest data set *news20*. Despite this the speedup achieved is rather low with a peak of 1.4X, again using *news20*, while all others achieved below 1X. This is likely because *news20* has the largest feature set size by far, with over 1.3 million features it has approximate 28.7 times the number of features in the next largest data set with respect to feature set size. This is important because it means that more non-zeros can be accessed at a decreased probability of incurring cache invalidations due to writing to the shared model vector.

The non-zeros per millisecond per memory channel (NZMSMC) values shown in Table 3 are an indication of the efficiency of hardware utilization during execution of the training loop. The values shown are calculated using as $\frac{1}{time\_per\_nz} * num\_mem\_channels$ with larger values indicating greater hardware efficiency.

The conventional system obtained NZMSMC values between 1904.76 and 3265.31 with the peak being seen when evaluating *rcv1*. *rcv1* has the greatest number of non-zeros, over twice that of *webspam* which only achieved an NZMSMC of 1904.76 using 1 thread. This is likely due to the difference in feature set size as well as the greater amount of work available per thread. The decreased race conditions awarded from the larger feature set size allowed for better cache utilization and subsequently improved performance. The greater workload of *rcv1* also enabled the use of a greater number of cores thereby accessing 16 memory channels rather than 8 again improving performance.

8.2 Migrating Threads

For migrating threads the striped implementation was the best performing version. Like the conventional OpenMP version Table 3 contains the performance measurements and metric calculations for the striped tests.

At first glance we can see that performance occurred at much greater total thread counts than that of the conventional system with between 512 and 1024 threads. Additionally all tests achieved peak performance when using all 8 nodes, 64 nodelets, of the Lucata system.

With the exception of *covtype* the migrating thread time per non-zero values are at or lower than 22x the times observed for the conventional system. Remembering that the core clock rates are 2.66GHz for the AMD EPYC 7451 and 175MHz for the Lucata Gossamer FPGA design the time comparison is both appropriate and proportional based on their respective clock rates. *covtype* had the smallest number of non-zeros, yet on migrating threads every non-zero triggers a migration. It is possible there simply was not enough work to overlap with the overhead associated with copious thread migrations.

Total speedup is much much greater on migrating threads than on the AMD based system, peaking at 246.5X, however due to the vast differences in architectural characteristics speedup alone is not good comparison. Instead the NZMSMC provides a comparison between the two on a per unit hardware basis.

In addition to achieving the highest speedup *rcv1* also obtained the greatest NZMSMC of any data set on either system at 5818.18. This is just under double that of the highest seen on the conventional system which was also obtained by *rcv1*. The combination of

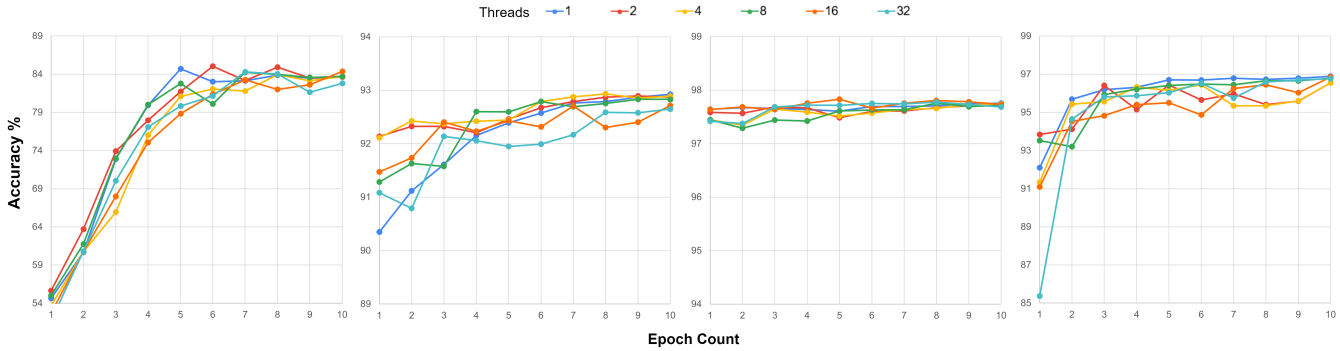

Figure 4: Observed accuracy for shared memory implementation on conventional architecture

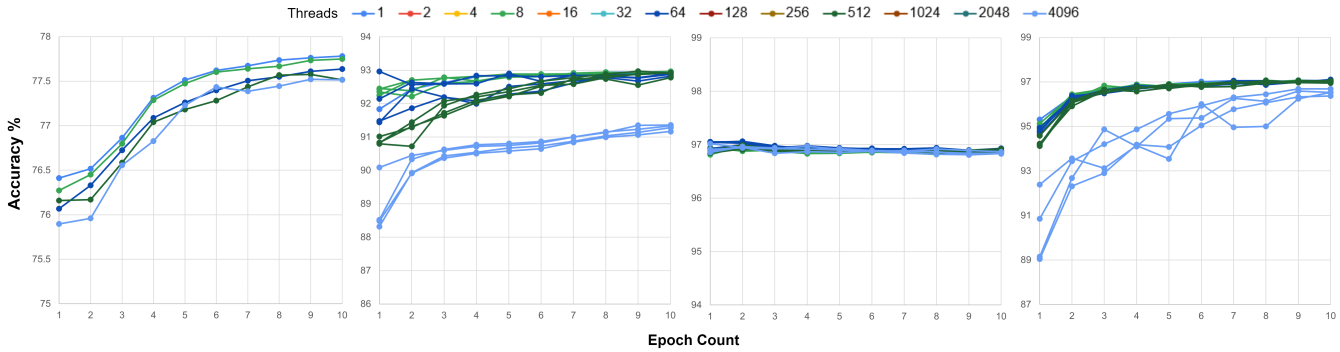

Figure 5: Observed accuracy for migrating threads with striped data allocation

System Architecture	Data Set	Threads	Cores / Nodelets	Memory Channels	Runtime (ms)	Time per nz (ms)	Total Speedup	Speedup per Thread	Non-Zeros per Millisecond per Memory Chanel
Conventional	covtype	1	1	8	213.13	0.0038	0.178	0.178	2105.26
	webspam	1	1	8	989.8	0.0042	0.366	0.366	1904.76
	rcv1	48	48	16	2402.84	0.0049	0.647	0.0134	3265.31
	news20	32	32		477.39	0.0071	1.406	0.0439	2253.52
Migrating Threads	covtype	1024	64	64	8359.7	0.15	66.89	0.065	426.66
	webspam	512			13342.2	0.056	179.34	0.35	1142.86
	rcv1	1024			5201.3	0.011	246.47	0.24	5818.18
	news20	512			7209.2	0.107	97.2	0.189	598.13

Table 3: Performance Data for SGD SVM on Conventional and Lucata Migrating Threads. Best configuration results shown.

greater non-zeros to work on, as well as higher thread and memory channel counts provided superior efficiency in terms of hardware utilization on the Lucata system.

It is likely that migrating threads are capable of obtaining superior performance similar to what we saw with *rcv1* if the data sets being evaluated are of sufficient size that they are able to overlap computation with thread migration to such an extent that both the memory channels and processing cores are saturated for greater periods of time. This would allow for the greatest efficiency per unit hardware, much like we observed here.

Lastly the number of memory references per training loop iteration for each the base implementations on both architectures are 31 (conv. OMP), 79 (conv. Cilk), and 57 (Lucata). What is worth noting is that the Lucata system obtained superior scalability despite performing nearly 54% more memory references.

9 EVALUATION: SCALABILITY

Here we evaluate scalability of SGD SVM on both architectures in terms of speedup. The the purposes of this paper speedup is defined as follows: $speedup = \frac{single\ threaded\ runtime}{runtime\ with\ T\ threads}$.

9.1 Conventional Scaling

Fig. 6 shows the observed speedup for the base implementation on the conventional system using both OpenMP and Cilk Plus for thread management. As can be seen the conventional system achieves rather poor scalability for all data sets and thread counts, with Cilk Plus fairing worse than OpenMP on average. As seen in Table 3 the best conventional speedup of 1.406X over was seen when using the news20 data set with 32 threads/cores. In fact the news20 data set was the only one to achieve a speedup greater

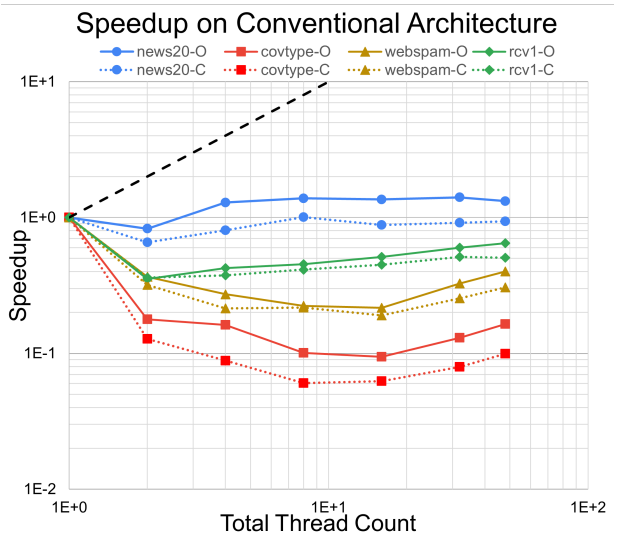

Figure 6: Observed speedup of baseline implementation on conventional architecture. -O = OpenMP, and -C = Cilk Plus

than 1. It is worth nothing that like the striped migrating thread implementation the conventional versions make no attempt to optimize memory allocation, access, or reuse in any way.

As discussed in [1, 8] poor scalablity was achieved when using a shared memory multi-socket. It was determined that the poor scaling they observed was due to increasing cross socket coherency traffic due to cache invalidation during computation as thread counts were increased, thereby severely degrading performance.

Fig. 6 validates the previously seen behavior for the test conventional system. For *covtype* and *webspam*, which have feature set sizes of 54 and 254 respectively, when thread counts increase so does the probability of race conditions on the shared model vector. Correspondingly these race conditions translate directly to cache invalidation between cores as well as sockets with greater probability. Speedup suffers significantly as a result.

9.2 Migrating Thread Scaling

9.2.1 Striping. SGD/SVM on migrating threads achieved vastly superior performance over the conventional system. Speedup for the striped case can be seen in Fig. 7. As can be seen **Speedups of up to 246X** were achieved when using 64 total nodelets and 1024 threads. For comparison the sparsest data set *news20* saw 97X speedup when using 64 nodelets and 512 threads. The rcv1 and news20 data sets are the closest in sparsity being just 0.155% and 0.034% of the total possible non-zeros for their sample and feature set sizes respectively.

Interestingly *covtype* and *webspam* were the least sparse, 22.12% and 33.52% respectively, and achieved peak speedups of 67.4X and 179.34X respectively. We can see that data sets which saw the best speedup were those which possessed the largest number of non-zeros. We assume much of this speedup comes from the cache-less nature of the system and the absence of the performance penalty which comes from cache behavior and coherency traffic.

As mentioned previously the migrating system has no cache to speak of thus every memory reference, that is not associated with register access, triggers a 64bit load from main memory on the nodelet. Conventional systems can of course take advantage of sequential access to adjacent address via the use of multi-level cache behavior however Fig. 6 illustrated the drawbacks of cache coherency traffic associated with irregular memory access patterns such as sparse SGD/SVM used here. Due to its lack of cache the migrating thread system does not experience performance degradation from cache coherency traffic.

In the striped case memory allocation has been striped across all nodelets in a round robin fashion. Accessing the next non-zero associated with the current sample will require a memory reference yet this memory reference will be to an address on a remote nodelet resulting in the thread migrating to that nodelet. These thread migrations are done in hardware and are rather efficient however they are not free. A previous study [11] determined thread migration overhead to be between 1 and 2 nanoseconds.

For larger data sets with tens of millions of non-zeros we expected that the sum of thread migration overhead would negatively impact performance yet Fig. 7 suggest otherwise. We observed good performance for all data sets and node/nodelet counts until saturation at between 32 and 64 threads per nodelet. The processing cores on the Lucata system posses a 16 stage pipeline so saturation without migration or stalling should occur around 16 threads per nodelet. This indicates that the increased thread counts at which saturation occurs is due to an increase in an overlap of "in-flight" threads which are migrating and those currently in the pipeline. In short by increasing the thread counts we insure that the processing elements are always performing useful work despite every non-zero triggering a thread migration.

9.2.2 Feature Partitioning. Our feature partitioned implementation was intended to reduce the number of thread migrations per sample by allowing a thread to operate on all non-zeros present on the current nodelet for the given sample. In Fig. 8 the use of uniform feature partitioning on each nodelet achieved up to approximately 72X. Unfortunately it is evident that while we did obtain speedup as impressive as the striped case however we were still able to obtain vastly superior scalability over the conventional system.

The decrease in peak performance we observed originates from a diminished ability to overlap thread migration overhead with computation of presently scheduled threads. Additionally since samples are not uniform in the number of non-zeros or the feature ids they are associated with, it is easier to become locally compute or memory bound on a nodelet that experiences intermittent and or prolonged periods of intense activity. In short hot spots of activity be it computation or memory access are possible which was not an issue present in the striped implementation.

10 CONCLUSIONS

This study sought to evaluate the performance and scalability of SGD SVM on a novel migrating thread architecture. We designed and implemented several variants for both the conventional AMD based architecture used for comparison, as well as the Lucata system being evaluated.

Our tests showed that the conventional architecture scaled poorly with respect to threads averaging less than 1X speedup. Alternatively migrating threads obtained superior scalability compared to

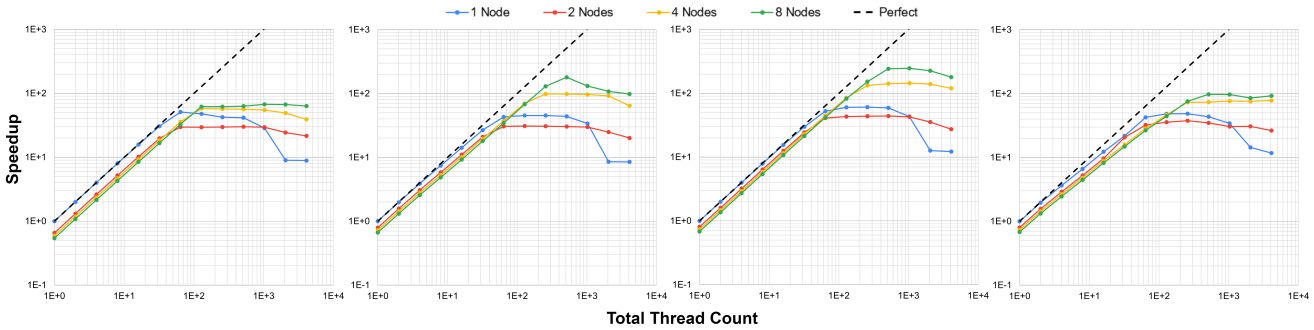

Figure 7: Observed speedup for migrating threads with striped data allocation

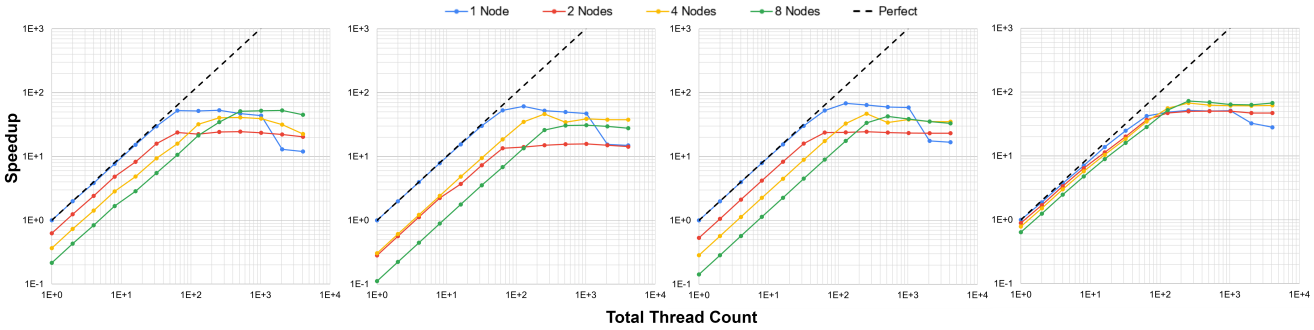

Figure 8: Speedup for feature partitioned migrating thread implementation

the AMD system, with a peak of 246X at the largest hardware configuration tested. Additionally two of the data sets migrating threads were able to achieve improved or near similar performance on a per unit hardware basis as that of the conventional architecture.

Another important facet to the use of migrating threads is the ease of use. The conversion of existing OpenMP based software to use on migrating threads can be as simple as replacing OpenMP pragmas with Cilk spawn and syncs much as we did the base/striped implementation. In our striped implementation we took advantage of memory allocation feature which performed distributed memory allocation for us. The same technique could be performed in a NUMA system but at considerably increased code complexity. Furthermore previous studies have shown that with respect to shared memory such allocations still fall prey to the cache coherency traffic penalty on performance.

As the architecture continues to evolve and mature migrating threads may be the preferred architecture for irregular applications such as SGD SVM.

REFERENCES

[1] 2011. *HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent* (Granada, Spain). Curran Associates Inc., USA. http://dl.acm.org/citation.cfm?id=2986459.2986537

[2] Vibhatha Abeykoon, Geoffrey C. Fox, and Minje Kim. 2019. Performance Optimization on Model Synchronization in Parallel Stochastic Gradient Descent Based SVM. *CoRR* abs/1905.01219 (2019). arXiv:1905.01219 http://arxiv.org/abs/1905.01219

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. , 10 pages. https://doi.org/10.1145/209936.209958

[4] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1223–1231. http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[5] Timothy Dysart, Peter Kogge, Martin Deneroff, Eric Bovell, Preston Briggs, Jay B. Brockman, Kenneth Jacobsen, Yujen Juan, Shannon Kuntz, Richard Lethin, Janice McMahon, Chandra Pawar, Martin Perrigo, Sarah Rucker, John Ruttenberg, Max Ruttenberg, and Steve Stein. 2016. Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture. , 8 pages. https://doi.org/10.1109/IA3.2016.7

[6] Norman P. Jouppi and et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. , 12 pages. https://doi.org/10.1145/3079856.3080246

[7] P.M. Kogge. 2004. Of Piglets and Threadlets: Architectures for Self-Contained, Mobile, Memory Programming. *Innovative Architecture for Future Generation High-Performance Processors and Systems* (Jan. 2004). https://doi.org/10.1109/IWIA.2004.10005

[8] Lam Nguyen, Phuong Nguyen, Marten van Dijk, Peter Richtárik, Katya Scheinberg, and Martin Takáč. 2018. SGD and Hogwild! Convergence Without the Bounded Gradients Assumption.

[9] Brian A. Page and Peter M. Kogge. 2019. Scalability of Hybrid SpMV on Intel Xeon Phi Knights Landing. *Int. Conf. on High Performance Computing & Simulation* (Jul 2019). https://par.nsf.gov/biblio/10109480

[10] Brian A. Page and Peter M. Kogge. 2020. Scalability of Hybrid SpMV with Hypergraph Partitioning and Vertex Delegation for Communication Avoidance. *submitted to 2020 Int. Conf. on High Performance Computing and Simulation (HPCS 2020)* (July 2020).

[11] Brian A. Page and Peter M. Kogge. 2020. Scalability of Sparse Matrix Dense Vector Multiply (SpMV) on a Migrating Thread Architecture. *10th. Int. Workshop on Accelerators and Hybrid Exascale Systems (AsHES) held in conjunction with IEEE Int. Parallel and Dist. Proc. Symp.* (May 2020).

[12] Brian A. Page and Peter M. Kogge. 2020. Scalability of Streaming on Migrating Threads. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC43674.2020.9286193

[13] Brian A. Page and Peter M. Kogge. 2021. Scalability of Streaming Anomaly Detection in an Unbounded Key Space using Migrating Threads. In *ISC High Performance (ISC HPC)*. 1–8.

[14] Christopher De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. 2015. Taming the Wild: A Unified Analysis of HOG WILD! -Style Algorithms. , 9 pages.

[15] Ce Zhang and Christopher Ré. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1283–1294. https://doi.org/10.14778/2732977.2733001

[16] H. Zhang, C. J. Hsieh, and V. Akella. 2016. HogWild++: A New Mechanism for Decentralized Asynchronous Stochastic Gradient Descent. , 629-638 pages. https://doi.org/10.1109/ICDM.2016.0074