Passel: Improved Scalability and Efficiency of Distributed SVM using a Cacheless PGAS Migrating Thread Architecture

Brian A. Page
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, USA
bpage1@nd.edu

Peter M. Kogge
Dept. of Computer Science and Engineering
University of Notre Dame
Notre Dame, USA
kogge@nd.edu

***Abstract*—Stochastic Gradient Descent (SGD) is a valuable algorithm for large-scale machine learning, but has proven difficult to parallelize on conventional architectures because of communication and memory access issues. 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. This paper discusses an 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 superior hardware efficiency and scalability over that of a conventional cluster using MPI. Furthermore these improvements were gained using naive data partitioning techniques and hardware with substantially less compute capability than that present in conventional systems.**

***Index Terms*—emerging architectures, irregular applications, machine learning**

I. Introduction

Inferencing via **Machine learning** (**ML**) is often straightforward to perform efficiently, especially by purpose-built hardware (cf. Google's TPU) [1]). 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 are based on the "HogWild" algorithms [2]–[4] 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 features 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 conventional parallel SGD implementations have yielded similar disappointing results.

Previous studies have also evaluated the effects of extreme sparsity on similar irregular problems such as **Sparse Matrix Vector Product** (SpMV) when executed on a variety of architectures [5]–[7], and real-time streaming [8]. 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 on these problems was the one used here [6], [8], [9].

The main contributions of this paper come from a port of the Hogwild++ algorithm to a migrating thread architecture and a careful scaling comparison with both results from the literature and a local implementation on a conventional cluster. The migrating thread version we developed uses multiple unique features of the underlying architecture, and demonstrates superior scalability over the conventional implementations, both in terms of use of hardware resources ("cores") and in terms of logical concurrency ("threads").

II. Background

A. 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. **Stochastic Gradient Descent** (SGD) repeatedly uses training data against the current model vector, determines when a projection is incorrect, and uses the "direction" of the error to modify the model vector slightly. Problems that

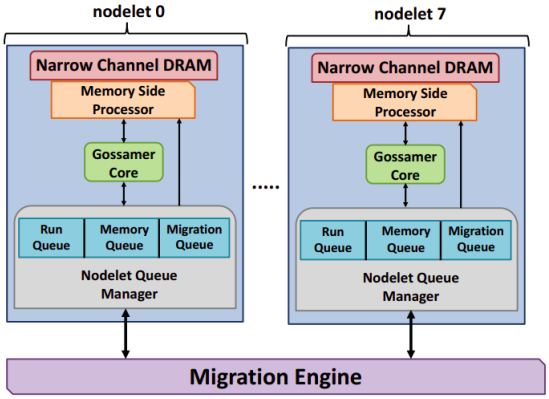


Fig. 1: A single node in the Emu Chick system. There are 8 nodelets within a single node. Nodes are connected over a Serial RapidIO interconnect (not shown here).

fall in this class include: SVM, matrix completion, and graph partitioning.

The SGD variant analyzed here is for the **Support Vector Machine** (**SVM**) problem, taken from [2]. 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:

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

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").

B. Migrating Thread Architecture

A migrating thread architecture [10] is one where the underlying hardware dynamically moves the state of a thread during execution. Fig. 1 diagrams such an architecture as implemented by Lucata Solutions [11] for a large scalable **Partitioned Global Address Space (PGAS)** parallel system where any thread on any core can reference any memory location in the system by simple load/stores. The basic unit, a **nodelet**, is a memory module, its controller and some number of multi-threaded cores. All nodelet memory resides in a common address space. A network connects all nodelets. A thread runs in a multi-threaded "**GC core**" until it makes a memory reference that is not contained in that nodelet's memory. The hardware then puts the thread to sleep, packages it, and moves it over the network to the correct nodelet, where it is unpacked and restarted. No software need intervene.

A thread can spawn independent child threads. Also, the memory controller contains hardware to implement atomic operations as close to memory as possible. Finally, very lightweight threads can be spawned to perform remote memory operations without moving the parent.

The prototype used, a Lucata Chick, is housed at Georgia Tech's CRNCH center[1]. It has 64 nodelets, each with 8GB of memory and one 175MHz multi-threaded core. These nodelets are packaged 8 to a **node board** which supports a RapidIO-based network that connects 8 such boards in a single chassis. A dual core POWER microprocessor on each node board runs Linux, manages a local SSD, and launches migrating threads into the system. The nodelet logic on each board is implemented in an FPGA. Table I compares its characteristics to that of the conventional system used as a baseline.

Due to the FPGA implementation, the core clock rate of the conventional baseline is 15X that of the CRNCH Chick. A more complete comparison is probably even higher than this in favor of the baseline as the nodelet cores are single issue and the baseline cores are multi-issue. Also, on a per core basis, the baseline has about 4.4X a pro-rated memory bandwidth of a core in the Chick, but, because of the memory channel design used in the nodelets, the ability of a Chick core to handle different independent memory accesses is actually 1.8X higher. Finally, the average network injection bandwidth per core is higher for the Chick than the baseline[2].

The programming tool chain is based on Cilk: C with a prefix to function calls to spawn new threads, a sync to wait for a set of children to complete, and a parallel *forall* to have a set of independent threads cooperate on a loop. Supported intrinsics include a rich set of remote atomic operations.

The migrating thread system has 3 levels of system parallelism: node, nodelet, and thread. Furthermore because threads can migrate freely throughout the entirety of the address space, there is the capability for significant overlapping of thread migration with computation. Lastly cacheless architectures are naturally coherency-free, therefore the performance degradation from cache coherency protocols and repeated cache invalidations on conventional architectures can be avoided.

A second generation system, Pathfinder-S, is currently being installed in the CRNCH Center, and should be available in the near future. This system has 3X the cores per node board and 2.7X the memory bandwidth. Also, unlike the Chick, a thread running on any core can access any of the 8 memory channels without even a local on-board migration. This improves load balancing. Only accessing memory on some other board causes a migration.

[1]https://crnch.gatech.edu/rogues-Lucata

[2]It should be noted that the baseline system has much higher bandwidth between its on-node 48 cores, and thus this ratio has a lot of caveats

System	Baseline	CRNCH	CRNCH	Ratio:	Ratio:
Type	HPE DL385 Gen10	Chick	Pathfinder-S	Baseline/Chick	Pathfinder/Chick
Socket	AMD 7451	Arria FPGA	Stratix FPGA		
Cores/Socket	24	8	24	3	3
Core Clock (GHz)	2.66	0.175	0.220	15.2	1.3
Memory Channels	8	8	24	1	3
Compute Cycles per Socket (G/s)	63.8	1.4	5.3	45.6	3.8
Mem. B/W per Socket (GB/s)	170.62	12.8	34.1	13.3	2.7

TABLE I: Comparison to Baseline Implementation.

III. 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 parallel SGD implementations have seen such issues limit efficient parallel scalability. The codes DisBelief and Downpour [12], 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 [13] but has not reported comparable speedup.

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.

1) Hogwild!: The *HogWild!* algorithm [2] was the first of a series of algorithms (summarized in Table II) 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 [4] 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 II 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.

2) Hogwild++: The *HogWild++* algorithm [3] assumes a NUMA[3] 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 send 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. There is still, however, on-socket interfering cache traffic.

Table III summarizes reported results for SVM using the HogWild++ algorithm[4]. 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

[3]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.

[4]The best configuration reported in Table III comes from the speedup figures, not the text, as there seems to be a difference.

Algorithm	Refs	Type	Parallel Model	Key Feature	Scaling	Limiter
HogWild! 2011	[2], [14]	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	[4]	Dense	Same as Hogwild!	Hogwild! with short precision	rcv1: 5X HogWild! at 12 cores & 8-bit precision	Same
HogWild++ 2016	[3]	Sparse	NUMA multi-socket, multi-core	multiple local models, round robin model sync	news20: 9.5X@4x10 cores	update process
DMS 2019	[13]	Dense	Distributed cluster	Local models, partitioned dataset, global sync	$\approx$ 5X @32 processes and large block sizes	model communication

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

TABLE III: SVM Training Data set Characteristics from [3].

will accept a token[5]. 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, the biggest case was using all 40 cores in the system (4 sockets of 10 cores each).

IV. Implementation

A. Passel: Distributed Hogwild++ on Migrating Threads

The migrating thread version of distributed Hogwild++, called **Passel**, uses the conventional cluster implementation as a framework and incorporates hardware specific advancements. The result is that Passel is true to the Hogwild++ algorithm, but tailored to using migratory threads within a logically shared but physically distributed address space. Despite the Lucata Chick being a shared cacheless architecture, application performance is often dependent on data partitioning and placement as was shown in [16]. Each cluster is given a disjoint subset of training data to work over, as well its own working and model vector. Correspondingly, training in Passel is identical to that of Hogwild++ and our distributed derivative version.

Passel's advantage stems from its use of thread migrations and remote atomic operations. Given the PGAS address space of the Lucata Chick system, any address in the entire space is accessible by any thread, from anywhere in the system, at any time. As such it is the passing of update tokens and the subsequent update sequence that takes full advantage of the migrating thread architecture.

Fig. 2 illustrates the sequence of events that occur during an inter-cluster update. In Fig. 2a cluster j is currently training when it receives the update token. At the beginning of its next loop iteration, thread 0 on j detects the token and spawns a team of *upstream update* threads followed immediately by spawning a team of threads remotely on $j+1$ to perform the *downstream update*. Once the newly spawned update threads have finished spawning and are ready, they begin to execute their respective update operations.

During any update, both cluster j and $j+1$ partake in the update, while the training threads on either cluster continue to train if there is still training data to analyze. From Fig. 2b we see that update thread teams on both clusters have begun to execute and calculate the model vector alterations that must be made to their targets' working or model vectors. These scaled adjustments are done using REMOTE_ADD() operations and are executed directly in the receiver's memory controller. As such j can asynchronously update $j+1$'s working vector as well as its own model vector, while $j+1$ is simultaneously remotely updating j's model and working vectors. Once the update is complete, both *upstream* and *downstream* thread teams exit and cluster j passes the token to $j+1$ via another REMOTE_ADD().

Additionally since the training loop consists entirely of local operations within a cluster, full thread migrations do not occur because the threads only access memory addresses on their own nodelet. Instead we take advantage of the architecture's method for performing instructions as remote atomic stores. Any arbitrary store to memory is done atomically and can be done asynchronously without causing the calling thread to wait for completion. This means that local working vector updates generated during the analysis of a training sample are added atomically and do not require the explicit use of atomic instructions or critical sections to prevent race conditions despite the shared environment.

Similar to Hogwild++ and our distributed version, training and updates continue as per normal until the desired completion criteria has been met.

[5] [3] reports using τ_0 of 64 when the number of cores per cluster was 10, and 16 otherwise.

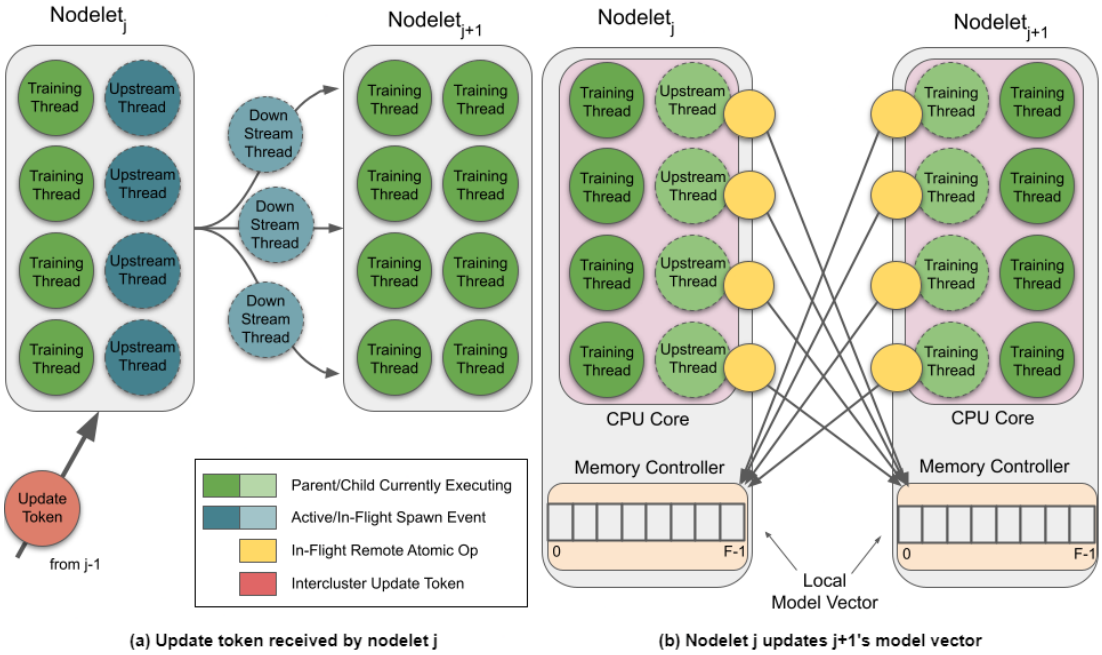


Fig. 2: (a) Nodelet j receives the update token from $j-1$ and begins the update process by spawning a local team of upstream threads, as well as a spawning a team of downstream threads remote on nodelet $j+1$. (b) Update threads on both j and $j+1$ asynchronously update each other's working and/or model vectors via remote atomic operations

B. Conventional Distributed Hogwild++

The original 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. To compare the Lucata Chick against traditional cluster systems, in addition to reported numbers from a shared memory configuration, we developed a distributed implementation of the Hogwild++ algorithm which utilizes the message passing interface (MPI) to perform its inter-cluster updates.

Memory allocation and indexing as well as the arithmetic operations performed during training and updates is identical to that of the original shared memory version of Hogwild++. Additionally, updates are still initiated upon receipt of a token. In our distributed implementation Update tokens are passed to the adjacent nodelet asynchronously using MPI_Isend while thread 0 on the receiving cluster periodically probes to see if it has received the token. Once an update has begun, MPI_Sends send model alterations between the updating cluster j and its update target $j+1$. After using the transmitted data to update $j+1$'s working vector as well as perform a self update of both j's working and model vectors, j will pass the token to $j+1$. This process continues until the desire epoch count, or some other stopping criteria has been met.

V. Evaluation

A. Experimental Setup

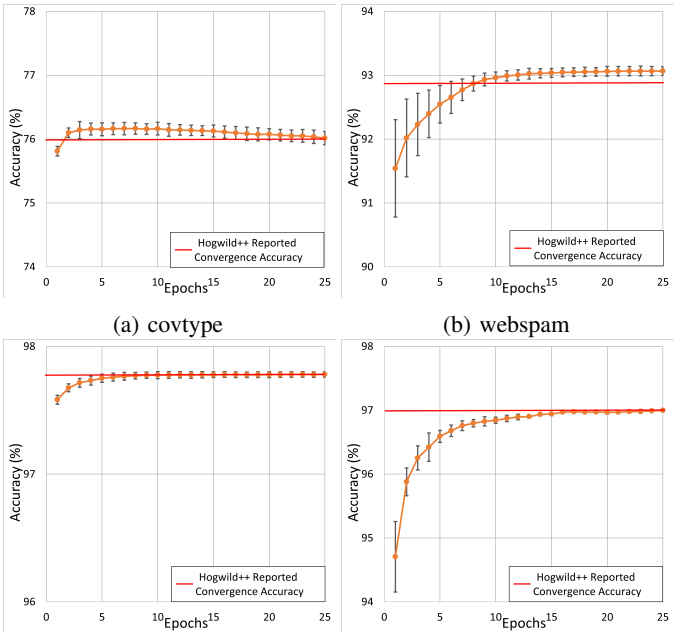
Tests using the distributed Hogwild++ were implemented on the conventional cluster and used between 1 and 64 nodes, with each node constituting a single cluster. In order to perform the best possible architectural comparison, 1 core per node was used, meaning each cluster consisted of a single compute core but retained full use of all memory channels and other relevant hardware on node. This was chosen due to the current Chick implementation possessing a single core per nodelet, providing us a method for comparing on a core to core basis.

Tests on the Lucata system were conducted using between 1 and 8 nodes for a total of 8 to 64 nodelets in a log2 fashion just as in the conventional tests. Total thread counts range from 1 to 1024. The much higher thread counts allow for the 16 stage pipeline of each nodelet processing core to have a higher chance of remaining full. This is not something that the conventional processor required, as it is capable of multi-instruction issue and has a different pipeline depth. Additionally results in previous studies show that increasing thread count per cluster on a conventional systems leads to decreased performance from cache coherency traffic. Our evaluations indicate that a nodelet running 16 threads is roughly equivalent in CPI to a conventional core of the type used in this study running a single thread.

We ran each of the 4 data sets on both systems in accordance with the standard "10 fold cross validation method" for evaluating a machine learning algorithms accuracy. Every system size or thread count configuration ran all 10 variations of training and testing data in order to obtain valid accuracy and standard deviation measurements.

B. Accuracy

Both our distributed Hogwild++ and Passel achieved accuracy comparable to those observed by Hogwild++ on a single



(a) covtype (b) webspam

(c) rcv1 (d) news20

Fig. 3: Observed accuracy for migrating threads with striped data allocation

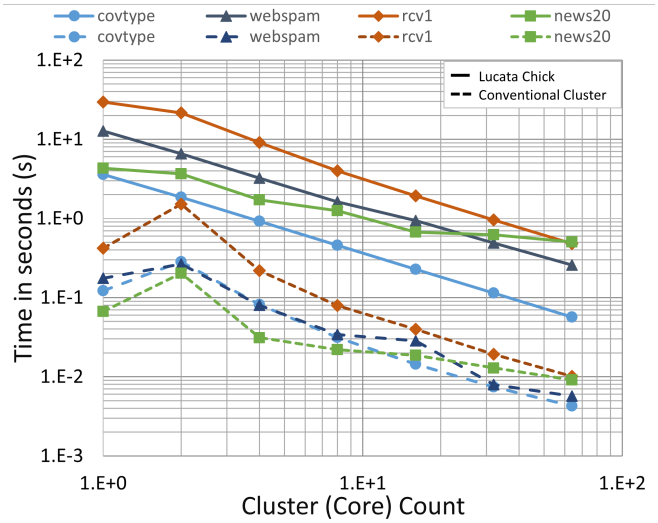


Fig. 4: Observed Average Epoch Time

shared memory system. Fig. 3 shows that the accuracy we observed for Passel on the Lucata system is on par with Hogwild++ despite the lack of explicit atomic operations during training in a shared environment. As can be seen the error (standard deviation) observed during the 10 fold cross validation test is extremely low indicating high accuracy consistency of Passel.

C. Scalability

We evaluated the scalability of our implementations by comparing their average epoch times. Fig. 4 shows the average epoch time of all 4 data sets on for both distributed Hogwild++ on conventional and Passel on the Lucata Chick. As expected both systems and their respective implementation

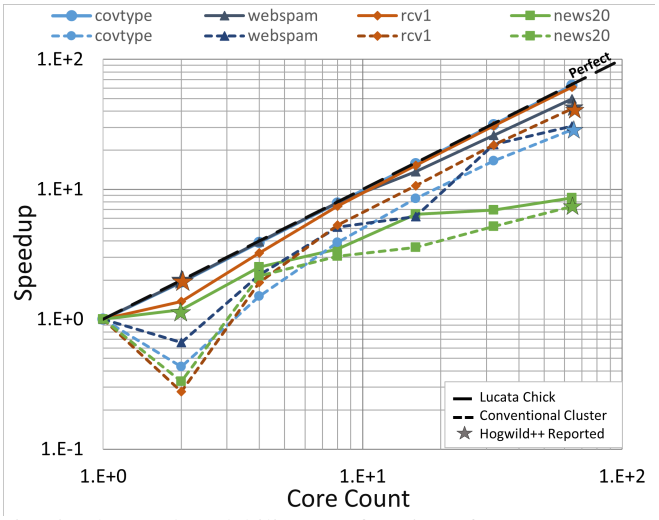


Fig. 5: Observed Scalability as a function of core count. Note: clusters use only 1 core regardless of trainer thread count per cluster.

experienced a decreased in the average time to complete a single as system size was scaled up towards the maximum of 64 cores. Each core represents a single Hogwild++ cluster with each cluster residing on a single node in the case of the conventional system, or a single Chick nodelet. this means that any inter-cluster update will require communication of some kind be it explicit MPI send/receives or thread migrations. The conventional system achieves the fastest epoch times for all data sets. However it also saw considerable increase at 2 cores (clusters) due to the addition of MPI overhead required to perform updates. On the other hand, Passel achieved consistent reduction in epoch times as core counts increased. Additionally unlike the conventional cluster as more nodes were added, thus increasing the off node communication required, no significant impact from this additional overhead was observed. We note that the execution time per epoch for the Chick is about greater than that for the conventional implementation, but at a factor almost perfectly in line with the difference in clock rates.

Here we define speedup as $\frac{single\ cluster\ time}{mutli-cluster\ time}$. In Fig. 5 we show the speedup for Passel and distributed Hogwild++. Much like the epoch times seen in the previous figure, the slope of lines for each data set is similar across both implementations and architectures. That being said, Passel and its use of migrating threads maintained near-perfect speedup for all but the sparsest data set *news20*. The conventional cluster experienced significantly decreased speedup when moving from 1 to 2 clusters as the additional MPI overhead necessary for inter-cluster updates outweighed the performance benefit of strong scaling. As system size was further increased, computational performance gain via strong scaling regained much of this speedup. In spite of this the continued presence of overhead communication means that the conventional system was only able to achieve between approximately 45.6% and 85.9% the speedup observed for Passel on covtype and news20 respectively. Fig. 5 also includes starred points representing

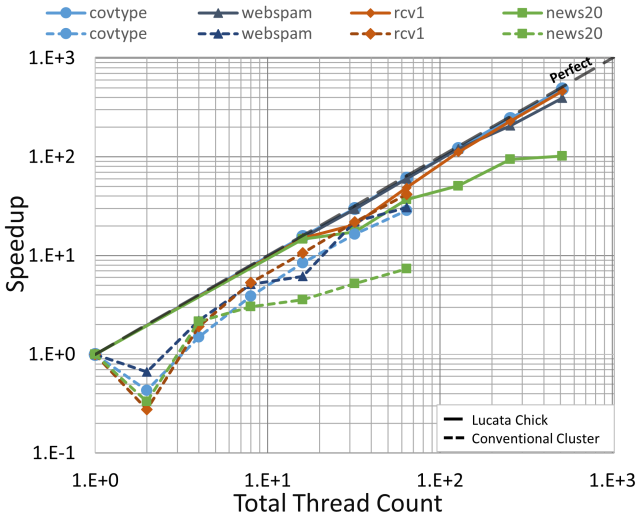


Fig. 6: Observed Scalability as a function of thread count. Note: clusters use only 1 core regardless of trainer thread count per cluster.

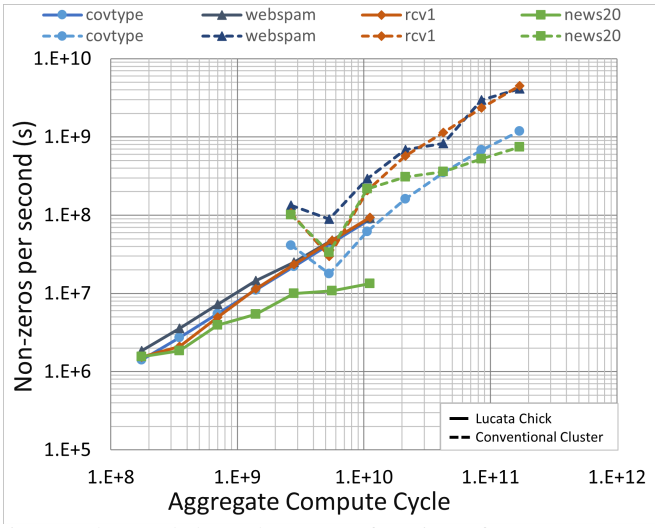


Fig. 7: Observed throughput as a function of aggregate compute cycles.

reported scaling results from the original HogWild++ paper. As can be seen they are in almost perfect agreement with our conventional results.

In addition to hardware concurrency (cores) we evaluated speedup in relation to logical concurrency (threads). As stated previously, due to architectural differences in the processing cores single conventional core running 1 thread is equivalent to a Chick nodelets' core running 16 threads. This means that the Lucata Chick is capable of higher levels of logical concurrency using for the same amount of physical hardware concurrency on a conventional system. Remember that it was the logical concurrency (inter-thread coherency traffic) that afflicted the original HogWild implementations. Fig. 6 shows the speedup relative to the total number of threads used for all data sets.

For the speedups shown on Fig. 6 we compute speedup as $\frac{single\ cluster\ single\ thread}{total\ thread\ count}$. We can see that the speedups for the conventional cluster remain unchanged from those seen in the core count comparison, since its core and thread counts are 1 to 1. Alternatively due to the dramatic increase in logical concurrency on the Chick we were able to run up to 1024 concurrent threads when using 64 total nodelets. Thanks to this additional concurrency Passel achieved dramatically improved speedup of **978X, 754X, 919X, and 125X for covtype, webspame, rcv1, and news20 respectively.** These speedups are vastly superior to distributed Hogwild++ on the conventional cluster.

D. Hardware Efficiency

A common metric for benchmarking HPC systems is hardware efficiency. Fig. 7 shows throughput in terms of non-zeros from training data evaluated per second vs compute cycles (clock times cores). The slope of the each data set's line for both systems is approximately the same, with the exception of the significant dip at 2 cores for the conventional cluster. As mentioned previously this dip is caused by the introduction of MPI communication which was not present when running on

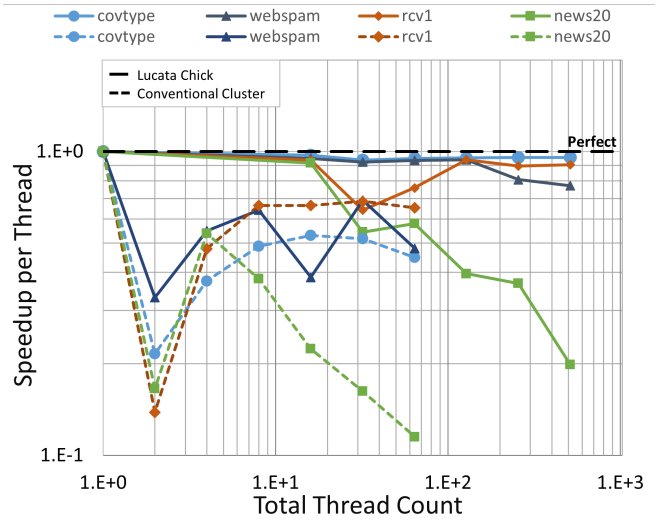


Fig. 8: Speedup per thread, defined as total speedup divided by thread count.

only one node. For Passel on the Chick we can see that the throughput scales near perfectly for all data sets except the sparsest *news20*.

In addition to throughput we analyzed the efficiency of logical concurrency for each system by looking at speedup per thread for both systems. As seen in Fig. 8 Passel on migrating threads achieves superior speedup per thread despite being a single instruction issue core, possessing just 1/15th the clock rate and just 1/8th the bandwidth per memory channel. This is due to the cacheless nature of the Lucata Chick architecture. Even when executing multiple threads per core, the Chick does not incur any invalidation penalty since there are no core caches. Similarly each memory references accesses a single 64 bit value meaning that the hardware only accesses the exact memory address requested and nothing more, eliminating wasted memory bandwidth taken up by unused data as is done

on conventional architectures.

Another important factor is the elimination of explicit software driven inter-node communication such as MPI. This is not to say that there is no communication in the Lucata Chick but rather that communication is performed in a highly efficient manner directly at the hardware level. We can see the dramatic effects eliminating this overhead has on scalability in Fig. 8 where we observed perfect or near perfect speedup per thread using Passel while the conventional system had significantly lower efficiency.

VI. Extrapolation to Pathfinder-S

As mentioned earlier, a new migrating thread system is being installed in the CRNCH center, with characteristics in Table I. There are 3X more, 1.3X faster, cores and almost 3X more memory bandwidth. This should reduce Epoch time (Fig. 4) by up to 3X. The Speedup vs Core Count (Fig. 5) and Speedup per Thread Count (Fig. 6) should maintain their advantages over conventional for the pictured range, with the potential to scale further up to a 6X higher range. The Non-Zeros/s vs Compute Cycles (Fig. 8) should see an upward lift of 30% for the Lucata curves as shown (faster clock), and again a 6x increase in the trends (2X nodes, 3X cores/node). We expect to perform verifying experiments on the new CRNCH Center system when it becomes available.

VII. Conclusions

This study sought to evaluate the performance and scalability of SGD SVM on a novel migrating thread architecture. We designed and implemented a distributed implementation for use on a conventional AMD based cluster, as well as Passel a migrating thread implementation designed for the Lucata Chick architecture.

Our tests showed that the conventional architecture scaled poorly with respect to threads averaging considerably less than 1X speedup for each additional thread. Alternatively migrating threads obtained near perfect speedup per thread leading to a peak speedup of 978X with 1024 concurrent threads vs a peak of just 41.7X on conventional, despite having dramatically less compute capability in terms of hardware. This was possible thanks to improved hardware efficiency brought on by the cacheless nature of the system, as well as the ability to perform remote atomic operations directly in the memory controller thereby providing an additional level of logical concurrency not present in traditional architectures.

We also now understand why the migrating thread advantage was lessened for the most sparse *news20* case (the model vectors are much longer), and have an alternative algorithm under test that appears to regain the significant migrating advantage for this very sparse case. Coupled with enhancements as found in the Pathfinder-S system, and even further with competitive ASIC implementations, it is not unreasonable to suggest that such systems will not only continue to have superior scalability, but also superior raw performance for such irregular problems.

References

[1] N. P. Jouppi and et al, "In-datacenter performance analysis of a tensor processing unit," New York, NY, USA, pp. 1–12, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[2] F. Niu, B. Recht, C. Ré, and S. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *NIPS*, vol. 24, 06 2011.

[3] H. Zhang, C. J. Hsieh, and V. Akella, "Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent," pp. 629–638, Dec 2016.

[4] C. D. Sa, C. Zhang, K. Olukotun, and C. Ré, "Taming the wild: A unified analysis of hog wild! -style algorithms," Cambridge, MA, USA, p. 2674–2682, 2015.

[5] B. A. Page and P. M. Kogge, "Scalability of hybrid spmv on intel xeon phi knights landing," *Int. Conf. on High Performance Computing & Simulation*, Jul 2019. [Online]. Available: https://par.nsf.gov/biblio/10109480

[6] ——, "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.

[7] ——, "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.

[8] ——, "Scalability of streaming on migrating threads," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–8.

[9] ——, "Scalability of streaming anomaly detection in an unbounded key space using migrating threads," in *ISC High Performance (ISC HPC)*, 2021, pp. 1–8.

[10] P. Kogge, "Of piglets and threadlets: Architectures for self-contained, mobile, memory programming," *Innovative Architecture for Future Generation High-Performance Processors and Systems*, Jan. 2004.

[11] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. B. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the emu system architecture," Piscataway, NJ, USA, pp. 2–9, Nov. 2016. [Online]. Available: https://doi.org/10.1109/IA3.2016.7

[12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "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., 2012, pp. 1223–1231. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[13] V. Abeykoon, G. C. Fox, and M. Kim, "Performance optimization on model synchronization in parallel stochastic gradient descent based SVM," *CoRR*, vol. abs/1905.01219, 2019. [Online]. Available: http://arxiv.org/abs/1905.01219

[14] L. Nguyen, P. Nguyen, M. van Dijk, P. Richtárik, K. Scheinberg, and M. Takáč, "Sgd and hogwild! convergence without the bounded gradients assumption," 07 2018.

[15] C. Zhang and C. Ré, "Dimmwitted: A study of main-memory statistical analytics," *Proc. VLDB Endow.*, vol. 7, no. 12, p. 1283–1294, Aug. 2014. [Online]. Available: https://doi.org/10.14778/2732977.2733001

[16] T. B. Rolinger, C. D. Krieger, and A. Sussman, "Optimizing memory-compute colocation for irregular applications on a migratory thread architecture," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 58–67.