RDMA-wild: an Asynchronous Distributed SGD with RDMA Networks for Fast and Scalable Distributed Machine Learning Training

Kwanghyun Lim [1] **Roberto Halpin** [1] **Sagar Jha** [1] **Ken Birman** [1] **Christopher De Sa** [1]

Abstract

Accelerating Distributed Stochastic Gradient Descent (DSGD) is important for faster and large-scale Machine Learning (ML) training. However, that is challenging due to the large network overhead between nodes. Even though RDMA (Remote Direct Memory Access) has received attention by researchers as it improves both network latency and throughput by an order of magnitude compared to TCP, how to optimize DSGD for RDMA networks has not been researched deeply yet. In this paper we first demonstrate that exploiting RDMA for DSGD brings enormous performance benefits over TCP as RDMA has no extra memory copies and context-switching to the OS kernel, which drops computing and networking performance of workers. Furthermore, we propose RDMA-wild, a novel asynchronous DSGD for RDMA networks in which workers asynchronously train in parallel on a partitioned training dataset maintaining the almost same convergence rate as Sync-DSGD per epoch. We empirically show that RDMA-wild outperforms Sync-DSGD using RDMA and TCP for both convex and non-convex problems on the MNIST dataset.

1. Introduction & Motivation

Accelerating Distributed Stochastic Gradient Descent (DSGD) under a parameter server setting is an important and practical challenge in modern distributed machine learning (ML) systems. This is because the use of SGD in a parameter server setting can be seen as a default distributed training setup in state-of-the-art frameworks such as TensorFlow(Abadi et al., 2016), CNTK(Seide & Agarwal, 2016), and MXNet(Chen et al., 2015) due to its effective performance over various machine learning tasks.

However, heavy pressure on the central parameter server and strict requirements on the underlying network has been pointed out by several researchers as its limitation since it can be a performance bottleneck when it scales up. In order to resolve this issue, researchers have proposed decentralized training and quantized communication approach that achieves better training wall-clock time by mixing models and dropping out less significant bits. (Tang et al., 2018), (Lu & De Sa, 2020).

Recently, RDMA (Remote Direct Memory Access) has gotten attention by systems researchers as it improves both network latency and throughput by an order of magnitude compared to TCP as it does not require copying from source to destination memory. Unlike TCP, RDMA avoids memory copy between OS kernel memory region and application memory region, resulting in a significant network performance improvement (Wei et al., 2020), (Jha et al., 2019), (Dragojević et al., 2014).

For ML practitioners, this is good news as RDMA is expected to significantly relieve the performance bottleneck in the central parameter server. This raises the question that decentralization and quantization is not necessarily a better choice for some practical ML tasks as it trades-off statistical performance and systems performance to avoid the performance drop by the network bottleneck.

In order to demonstrate the enormous performance benefits of RDMA for DSGD, we conducted the basic centralized synchronous DSGD protocol (Sync-DSGD), results are shown in Figure 1. We break down the total training wall-clock time to two parts: (actual) training time, and wait time. The training time is the time taken by workers to compute gradients and push the gradients to the parameter server, and the wait time is the time taken by the workers to wait for the consistent global ML model from the parameter server.

Our results show that RDMA roughly reduces wait time by 6x and training time by 2x compared to TCP when 15 workers train Logistic Regression (LR) with the MNIST dataset under the 100Gbps network channel. This can be primarily attributed to, differently from TCP, RDMA having no extra memory copies and context-switching to the OS kernel, which improves computing and networking performance.

Despite RDMA's high potential to improve the performance of ML systems, in-depth research on developing specialized asynchronous protocols of DSGD for RDMA in the param-

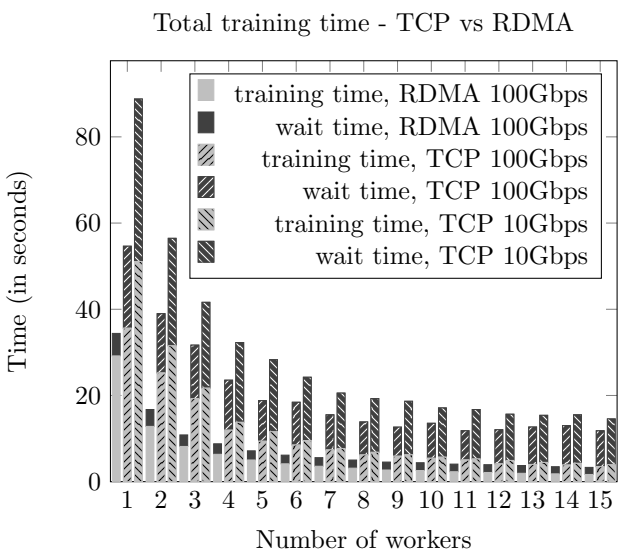


Figure 1. **Performance Breakdown of Sync-DSGD** for logistic regression under 100 epochs and 60 batch-size per worker on MNIST. TCP takes not only longer wait time but also longer training time than RDMA.

eter server setting has not been done. Existing work has been just focused on how to tune RDMA transfer effectively for ML-models/gradients transfer (Xue et al., 2019), (Ren et al., 2017) rather than how to optimize the DSGD protocol for RDMA. Note that SGD is one of the most popular and fundamental ML model optimization algorithms.

In this paper, we propose RDMA-wild, a novel asynchronous DSGD for RDMA that maximizes RDMA performance using one-sided RDMA write and state machine replication. To the best of our knowledge, there is no existing asynchronous DSGD protocol specialized for RDMA networks. Our contributions can be summarized as follows:

- We show that exploiting RDMA for DSGD brings enormous performance benefits for computing and networking that is not achievable by TCP.
- We propose **RDMA-wild**, a novel asynchronous DSGD for **RDMA** in which workers asynchronously train in parallel with partitioned training dataset similar to Hog**wild** (Recht et al., 2011). We solve the gradient loss and new gradient detection issues that occur when developing asynchronous DSGD for RDMA. As a result, RDMA-wild maintains almost the same convergence rate as synchronous DSGD per epoch (i.e., a single entire training dataset pass).
- We empirically show that RDMA-wild outperforms TCP and RDMA synchronous DSGD in both convex and non-convex problems on the MNIST dataset.

Intuition behind RDMA-wild. RDMA-wild was inspired by Hogwild where workers train asynchronously without locking the shared model memory region, allowing the race condition. We abstract multiple nodes like Hogwild, exploiting RDMA's high-speed network. This generates an illusion that RDMA-wild works on a single gigantic DRAM with scalable computing units. Note that TCP is much slower and more unstable than RDMA that applying Hogwild over TCP-connected nodes is more challenging due to the potential of high model staleness.

2. Background

Stochastic Gradient Descent. Stochastic gradient descent (SGD) is a popular ML model optimization algorithm that achieves state-of-the-art performance on various machine learning problems as it is applicable to differentiable loss functions and scalable for large datasets. In a given amount of time, it updates ML models more frequently with a gradient sample which is much noisier than the full gradient of Gradient Descent (GD) but takes much less time. That trade-off turned out to be great as the noisy gradient sample roughly points to the right optimizing direction anyways in expectation. Thus, SGD is more effective and scalable than GD as it accumulates the good trade-off between statistical performance and systems performance during training, which results in reduced total wall-clock training time. Its most common variant is mini-batch stochastic gradient descent where multiple examples are used per iteration instead of just one example for the sake of reducing the variance of gradient samples and utilizing GPU or CPU parallelism. It is often the default model update scheme due to its efficiency. In this work we create algorithms that try to optimize the use of mini-batch SGD in the distributed setting.

Synchronous and Asynchronous SGD. Even though SGD makes ML model optimization more effective and scalable than GD, it is not sufficiently rapid for large ML models applied on massive datasets. Thus, parallelizing SGD over a partitioned dataset with multi-threads in a single node or distributed nodes is necessary. There are two parallelizing techniques in this approach: synchronous and asynchronous SGD. Both have four main operations in the parameter server setting.

- $Op1$: Server updates model.
- $Op2$: Server pushes new model to workers.
- $Op3$: Worker computes a new gradient using a new model and dataset.
- $Op4$: Worker pushes a new gradient to server.

In synchronous SGD, $Op1$, $Op2$, $Op3$, and $Op4$ are done sequentially. Thus, server waits until all the workers push

new gradients and the workers wait until the server pushes new model. Even though these synchronous behaviors make parallel SGD equivalent to single thread SGD in terms of computation, it causes inefficiency as fast workers should wait the slowest worker. Asynchronous SGD is motivated to resolve this issue. It overlaps an operation pair of $Op1$ and $Op2$, and the other operation pair of $Op3$ and $Op4$ by letting workers work independently from each other, so it tries to reduce the workers's waiting time. It trades-off between systems performance and statistical performance as it causes model staleness; while one worker computes a new gradient with a model, if the model in the parameter server is updated by other workers, the computed gradient by the worker becomes less accurate because that will be used to update a different copy of model, which is not intended. Interestingly, it turned out to be a good trade-off as SGD is noisy itself (De Sa et al., 2015). However, careful design is required when the underlying network performance fluctuates due to preemption, context-switch, large data copy, system crash, etc. (Li et al., 2014), (Ho et al., 2013). This is because when slow workers are too far behind faster workers, the training can be very sensitive or diverge if we don't bound faster workers' asynchrony properly.

Hogwild. Hogwild (Recht et al., 2011) is a fully asynchronous SGD algorithm in the sense that it does not bound worker's asynchrony at all. It implements asynchronous SGD with multi-threaded workers on a single node where workers can communicate through fast and stable memory buses of DRAM. Thus, it implements asynchronous SGD without any locks to prevent locking from becoming a performance bottleneck. However, Hogwild has limitations that 1) the optimization should be sparse, and 2) it should be conducted under a fast and stable network, for example within a single node. As shown in other works (Zhang et al., 2016) (Gupta et al., 2012), the second constraint limits scalability due to contention for shared architectural resources (e.g., cache, memory) as the number of threads gets larger. Leveraging an emerging fast network, RDMA, we target to overcome the limitations of Hogwild.

Centralized Distributed ML. In distributed machine learning a centralized approach is popular among state-of-the-art frameworks (Abadi et al., 2016) (Seide & Agarwal, 2016), (Chen et al., 2015). It designates a small subset of nodes to be parameter servers and other nodes to be workers, which maintain global shared model (i.e, parameters) and perform gradient computations on a partitioned training dataset. The parameter server receives gradient updates from the workers and uses those gradients to update the global model which it then broadcasts to the workers. The workers receive the global model from the server and each worker computes local gradients using that model on the partitioned data that they have been allocated. This classic centralized distributed ML framework is commonly used due to its ease of use and its efficiency. RDMA-wild is built in this parameter server setting.

Decentralized Distributed ML. Another distributed machine learning approach is the decentralized setup. Instead of having a small amount of central nodes, such as parameter servers, that handle most of the communication and perform much different computations than worker nodes, the decentralized setup makes every node perform all the required tasks. This setup is attractive due to evenly splitting the needed computations between all nodes in the cluster. However, one large concern in this approach is the slow model mixing time, which increases training time (Lian et al., 2017).

3. RDMA for ML

RDMA is an emerging fast network technology in which one node can bypass remote node's CPU and directly access the other remote node main memory without any extra data copy. RDMA has two communication modes: one-sided RDMA transfer and two-sided RDMA transfer. As their names imply, in one-sided RDMA transfer, senders can RDMA-transfer without receivers' participation once initial consensus about RDMA memory regions are done. However, in two-sided RDMA transfer, senders can start sending only after receivers have posted. Thus, the two-sided RDMA transfer includes CPU engagement whereas the one-sided RDMA transfer does not. In our work we use the term RDMA specifically to refer to one-sided RDMA transfer.

Therefore, if RDMA is used for ML systems, training can be done without any blocking. This property is attractive especially for asynchronous DSGD as it opens up a new opportunity in which communication for model/gradient exchanges can be fully overlapped with computation for models/gradients, resulting in potential performance gains for ML systems. Note that this is hardly achievable by TCP as it includes extra data copies between the application and the OS kernel that cause context-switching, and remote node's CPU involvement.

In the statistical perspective, RDMA can help asynchronous DSGD realize better convergence rate due to decreased model staleness, achieved by no additional copying and non-stop training.

4. RDMA-wild

We propose RDMA-wild, a Hogwild-style parallelized SGD over distributed systems using a RDMA network. It takes a similar approach as Hogwild in the sense that it allows workers to work asynchronously. We aim to take full advantage of RDMA as discussed in the previous section.

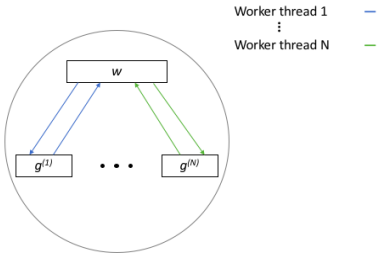


Figure 2. **Hogwild overview**

Assumptions. First, a highly fast and stable network should be prepared for RDMA-wild. This can be realized by RDMA's fast and stable performance, which could not have been achieved by classical networks such as TCP. For example, RDMA can saturate the network limit of Infiniband 100Gb, and achieve $\sim$ 2µs round-trip latency whereas TCP cannot (Frey & Alonso, 2009). Second, almost identical machines should be deployed for RDMA-wild. Third, preemption by other applications is forbidden. The second and third assumption seem to be strict but machine learning tasks are becoming larger and more significant, so these assumptions are not as strict in practice.

4.1. Overview

RDMA-aware. Taking full advantage of RDMA's capability is important for accelerating DSGD. As most ML models' parameters are fixed and RDMA's asynchronous property matches well with asynchronous DSGD, we basically use one-sided RDMA write instead of two-sided RDMA write in order to benefit from the fast and seamless remote write rate. Plus, we avoid local copies of models and gradients other than the remote copy where we avoid potential performance bottleneck as RDMA does not copy through network path. As shown in figure 3, RDMAwild has no local copies of models and gradients, but just remote copies of gradients in the parameter server, and a remote model copy in the worker.

Difference from Hogwild. Although RDMA-wild was inspired by Hogwild, it has several differences which originate from the distributed RDMA setting.

First, RDMA-wild has remote copies of the global model w over workers and remote copies of worker's local gradient g in the server that form a gradient table as shown in Figure 3. Note that those remote copies are optimal in the sense that each remote computing unit must have its own data copy for computation.

Second, RDMA-wild carefully designates threads to the four main operations described in section 2: $Op1$, $Op2$, $Op3$, and $Op4$. Basically, we assign the pair $Op1$: *Server updates model* and $Op2$: *Server pushes a new model to workers* to a single thread in the server and let it conduct them sequentially, which is illustrated by a red line in Figure 3. This will eliminate the potential race condition error so that RDMA-wild can work more reliably in various ML tasks even including dense optimization. Similarly, we assign the pair $Op3$: *Worker computes a new gradient using a new model and dataset* and $Op4$: *Worker pushes a new gradient to server* to each worker thread in worker nodes and conduct them sequentially. This is represented by the blue and green lines in Figure 3.

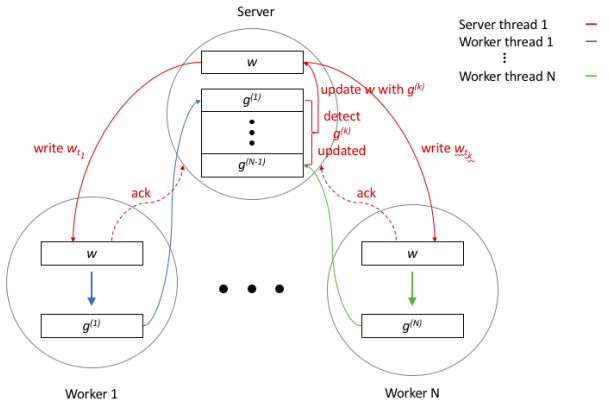


Figure 3. **RDMA-wild overview**

4.2. Algorithm & Implementation

Design choices and challenges. When designing asynchronous DSGD for RDMA, there are two main design choices (DC1 & DC2) and challenges (C1 & C2) outlined as follows:

- $DC1$: How to aggregate new gradients and update the model in the server.
- $DC2$: How to push a new model to workers.
- $C1$: How to prevent gradient loss when allowing the server thread and worker threads to work asynchronously.
- $C2$: How to detect a new gradient in the server and a new model in workers.

As for $DC1$, whereas the server in Sync-DSGD sums up gradients, divides them by the number of machines N, and updates the model, RDMA-wild first divides a single gradient by N and updates the model w (line 13 in Algorithm 1). This is because RDMA-wild workers work asynchronously so that the server should provide them a new model asynchronously.

Algorithm 1 RDMA-wild server thread

1: **Input:** learning rate α, initial parameters w_0, number of batches T, number of epochs E, number of machines N
2: // Initialize
3: **for** $node\_id = 1$ **to** N **do**
4: $GVnum[node\_id] = -1$;
5: $LGVnum[node\_id] = -1$;
6: **end for**
7: $receivers = \{\}$;
8: // Training
9: **while** training **do**
10: **for** $node\_id = 1$ **to** N **do**
11: // Detect new gradients
12: **if** $GVnum[node\_id] > LGVnum[node\_id]$ **then**
13: Update model: $w_t \leftarrow w_{t-1} - \frac{\alpha}{N} \cdot g[node\_id]$;
14: $LGVnum[node\_id] = GVnum[node\_id]$;
15: $receivers = receivers \cup node\_id$;
16: **end if**
17: **end for**
18: **if** $receivers != \{\}$ **then**
19: Push w_t and $LGVnum$ vector to $receivers$;
20: $receivers = \{\}$;
21: **end if**
22: **end while**

Algorithm 2 RDMA-wild worker thread

1: **Input:** learning rate α, initial parameters w_0, number of batches T, number of epochs E, number of machines N, per-machine mini-batch size B
2: // Initialize
3: $LGVnum[my\_node\_id] = -1$;
4: $GVnum = -1$;
5: // Training
6: **for** $i = 1$ **to** E **do**
7: **for** $i = 1$ **to** T **do**
8: **while** $LGVnum[my\_node\_id]$!= $GVnum$ **do**
9: wait;
10: **end while**
11: Select a mini-batch data $D_1, ..., D_B$ of size B;
12: Compute gradient: $g_t \leftarrow \frac{1}{B} \sum_{b=1}^{B} \nabla f(w_t; D)$;
13: $GVnum+ = 1$;
14: Push g_t and $GVnum$ to parameter server;
15: **end for**
16: **end for**

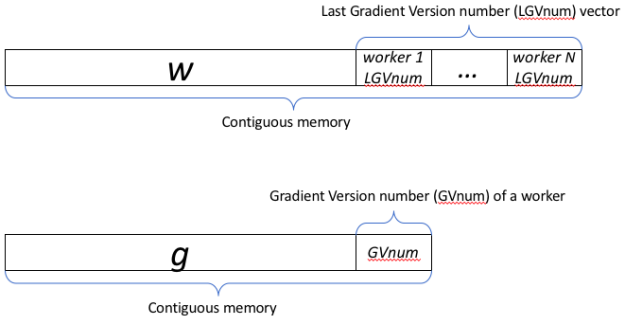


Figure 4. **RDMA-wild data structure**

As for $DC2$, RDMA-wild acts like a $barter$ system where once a worker gives its new gradient, the RDMA-wild server gives back a new model w to the worker (line 18-20 in Algorithm 1). We empirically find that this method is more efficient than broadcasting w to all the workers as it offloads the burden to the underlying network making it achieve better system performance without losing statistical performance.

Data structure. Before discussing how RDMA-wild's worker and server resolve the challenges $C1$ and $C2$, let us first look at RDMA-wild's data structure. As illustrated in Figure 4, we assign a different version number to each gradient, called $GVnum$ (**G**radient **V**ersion **num**ber). If a worker computes its gradient from the kth mini-batch of a training dataset partition, the gradient's $GVnum$ is k. Note that, for this versioning method and better systems performance, we implement the sequential scan SGD that shuffles training data in advance and compute gradient by scanning the training dataset from left to right rather than sampling training instances on the fly, which is what regular SGD does.

We should also keep track of the last gradient version number called $LGVnum$ (**L**ast **G**radient **V**ersion **num**ber) in addition to $GVnum$ in order to resolve $C1$ and $C2$. As illustrated in Figure 4, we store workers's $LGVnum$s right next to the model memory region in the form of vector.

Workers. To resolve the issue of $C1$, RDMA-wild takes the conservative approach that the server and worker work synchronously whereas workers work asynchronously with respect to each other. Before computing a new gradient, RDMA-wild workers check if the previously pushed gradient has been consumed by the server for a model update (line 8-10 in Algorithm 2). This is because workers can push the next gradient and overwrite the previous gradient in the server before the server consumes it for a model update, which causes gradient loss. This is an important issue as gradient loss causes incomplete training and this can generate negative side effects. If specific versions of gradients are lost multiple times over epochs, this can lead to biased training. Even though this may not be an issue when the probability distribution of gradient loss is uniform and the number of epochs E is large, RDMA-wild pursues a stable and robust system by guaranteeing no gradient loss.

After making sure that the previous gradient has been consumed by the server, it computes a new gradient, increments $GVnum$ by one, and pushes them to the server (line 11-14 in Algorithm 2).

Server. To resolve the issue of $C4$, the server thread compares $GVnum$ and $LGVnum$, as described in the line 12 in Algorithm 1. To elaborate, if $GVnum[node\_id]$ is incremented by the worker thread's RDMA write with $node\_id$ and becomes greater than $LGVnum[node\_id]$, that means $g[node\_id]$ has been updated. $g[node\_id]$ stands for the gradient of the worker with $node\_id$. Note that the $GVnum$ vector lies vertically as a column of the gradient table and the $LGVnum$ vector lies horizontally right next to the model w.

After detecting a new gradient, the server updates the model, $LGVnum$, and registers its $node\_id$ to the $receivers$ set to where the new model will be pushed (line 13-15 in Algorithm 1). After, it pushes the new model w and $LGVnum$ vector to the $receivers$ using one-sided RDMA write (line 18-21 in Algorithm 1).

RDMA-wild's data structure and its atomic one-sided RDMA transfer are implemented using SST(Shared State Table) in Derecho (Jha et al., 2019).

5. Evaluation

As RDMA-wild is an optimization algorithm for loss functions, we test it under two different problem settings: convex and non-convex optimization problems. For convex problems, we use a simple Logistic Regression (LR) and LR with Random Fourier Features (LR + RFF) (Rahimi & Recht, 2007). For non-convex problems, we use a fully-connected neural network (FCNN).

Evaluating optimizers on convex problems is meaningful because we can get more detailed information than non-convex problems as the distance to the global unique optimal model w^* can be measured. Evaluating on non-convex cases is also important as Neural Networks have many practical applications in the real-world.

For each problem, we evaluate and analyze the statistical and system performance of RDMA-wild by comparing to the following baselines: single node mini-batch SGD and synchronous DSGD. The system performance metrics are training wall-clock time, and scalability for larger models and more workers. The statistical performance metrics are training/testing accuracy and convergence rate.

5.1. Setup

We used 16 nodes (1 server, 15 workers) as our cluster running Ubuntu 18.04 of Linux kernel 5.0.0 connected with a 100Gbps (12.5GB/s) RDMA InfiniBand switch (Mellanox SB7700). Each node is equipped with 100GB RAM, 32 cores, and Mellanox MCX456AECAT Connect X-4 VPI dual port NICs. We implement logistic regression and a fully connected neural network with C++ using CBLAS in order to use the CPU parallelism from SIMD instructions for matrix computations. For logistic regression with RFF, we preprocessed our input data through a RFF Python implementation and trained the RFF-transformed dataset with the C++ logistic regression.

All the evaluations were performed on a server cluster with 16 nodes (1 server and 15 workers) with MNIST handwritten digits dataset (Yann LeCun, (accessed November 20, 2020). And training was done with 64 bit double precision after loading the entire input dataset on DRAM.

For hyper parameters, we performed grid search. Note that as RDMA-wild is an asynchronous DSGD which is non-deterministic for different trials, we took an average of 5 trials for each grad point and picked the best hyper parameters. In the experiment, we adopted the following hyper-parameters for RDMA-wild: {LR: learning rate = 0.9, weight decay = 0.9, batch size per worker = 8, epoch = 15}, {LR + RFF: learning rate = 9, weight decay = 0.9, batch size per worker = 8, epoch = 50}, {FCCN: learning rate = 0.1, batch size per worker = 8, epoch = 50}.

For LR and LR + RFF, we initialized the initial model w_0 with all zeros, and used the cross-entropy loss function. And RFF is implemented to approximate the Gaussian kernel. For FCNN, we set its structure to input layer (784 neurons) X hidden layer (128 neurons) X output layer (10 neurons) with ReLU activation and no regularization. We initialized w_0 with sampling from sqrt(2 / input_layer_size) * N(0,1) distribution, which is a standard method (Kaiming Initialization).

5.2. Training Time

Table 1 shows how much RDMA-wild improves training time comparing to synchronous DSGD (Sync-DSGD). We ran LR, LR + RFF, and FCNN for 15, 50, and 50 epochs up until it converges resulting in 92%, 95%, and 99% accuracy, respectively.

As shown in the Table 1, RDMA-wild achieves better wall-clock time performance than Sync-DSGD. It improves 11%, 13%, and 13% compared to Sync-DSGD for LR, LR + RFF, and FCNN, respectively.

RDMA-wild works more effectively for large models (LR+RFF, FCNN) than LR. When we inflate the model size 12.5X from 64KB to 800KB using RFF for LR, it gains 2% additional performance.

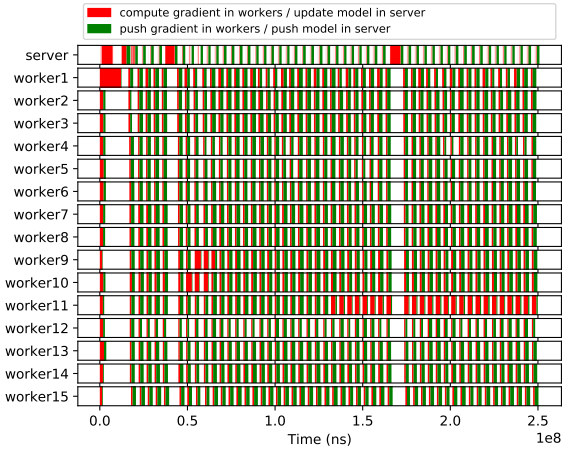


Figure 5. **Sync-DSGD Performance Breakdown (LR + RFF)**

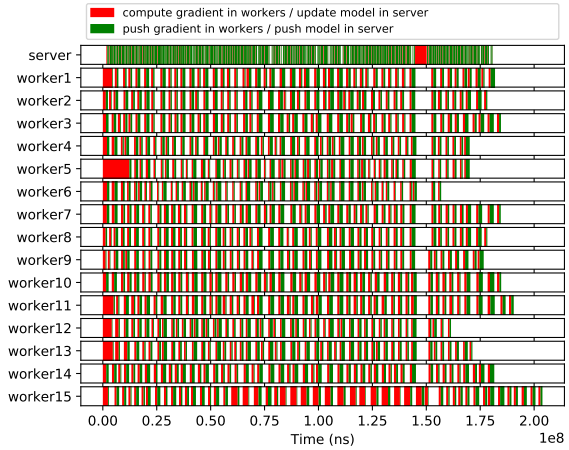


Figure 6. **RDMA-wild Performance Breakdown (LR + RFF)**

Table 1. **Training Time (in seconds)** of 15 workers with RDMA

MODEL	MODEL SIZE	SYNC	RDMA-WILD	IMPROVEMENT
LR	64KB	3.10	2.79	11%
LR + RFF	800KB	122.36	95.98	13%
FCNN	800KB	135.92	110.29	13%

5.3. Training Time Analysis

To understand why RDMA-wild outperforms Sync-DSGD, we keep track of the time taken of the $Op1$, $Op2$, $Op3$, and $Op4$ in the server and workers, and analyze their pattern.

Figure 5 and Figure 6 are performance breakdown of LR + RFF training from beginning to around 0.1 epoch. The red slots are the time taken of $Op1$: *Server updates model* and $Op3$: *Worker computes a new gradient using a new model and dataset* which are related to computing. And the green slots are the time taken of $Op2$: *Server pushes a new model to workers* and $Op4$: *Worker pushes a new gradient to server* which are related to networking.

As shown in Figure 5 and Figure 6, workers in RDMA-wild work asynchronously whereas workers in Sync-DSGD work synchronously. For example, from time 0.0 - 0.2 in Figure 5, even though worker2-15 completed $Op3$ and $Op4$, they cannot proceed to the next $Op3$ and $Op4$ as worker1 straggles. However, from time 0.0 - 0.2 in Figure 6, all workers of RDMA-wild asynchronously proceed to the next operations regardless of the straggler worker5. This RDMA-wild's asynchronous pattern reduces wait time of workers resulting in better wall-clock time.

Even though workers work asynchronously in RDMA-wild, the server and workers work synchronously due to the gradient loss issue. At time 1.5 in Figure 6, we can observe that pattern. When the server straggles, workers halt temporarily. For LR and FCNN, we find the similar pattern and the results are in the Appendix.

5.4. Accuracy

Even if RDMA-wild performs better than Sync-DSGD in terms of systems performance (wall-clock training time), its model staleness issue can drop its accuracy. Recall that while one worker computes a new gradient with a model, if the model in the parameter server is updated by other workers, the computed gradient by the worker becomes less accurate because it will be used to update a different copy of the model, which is not intended.

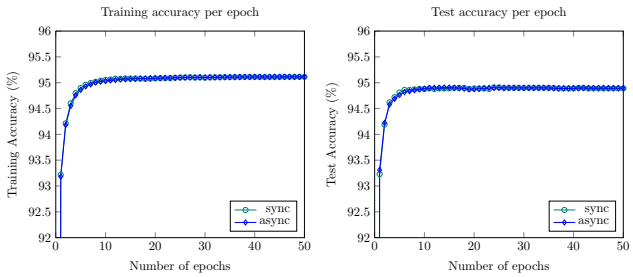
Thus, to figure out how RDMA-wild effectively trades-off between systems and statistical performance, we compare RDMA-wild to Sync-DSGD as Sync-DSGD doesn't have the model staleness issue and achieves the same statistical performance as a single threaded SGD.

Figure 7 shows that RDMA-wild maintains almost the same training and test accuracy as Sync-DSGD per epoch for LR + RFF and FCNN.

5.5. Convergence Analysis

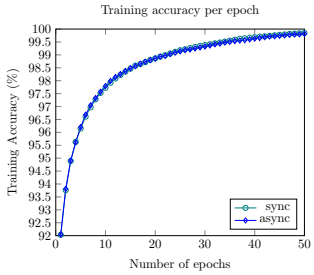
Even if we showed that RDMA-wild maintains the same accuracy as Sync-DSGD in the previous section, it is insufficient to conclude that RDMA-wild converges well because there can be some cases that a trained model shows good accuracy even though it is far away from the optimal model.

Thus, we measure three extra metrics to ensure RDMA-wild's convergence: distance to the optimal model w^*, gradient norm, and loss gap between our model and the optimal model w^*. Note that as the model converges to w^*, its gradient norm approaches 0 in convex problems, in our case, LR

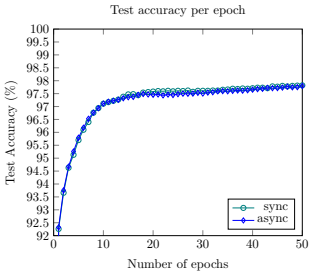


(a) LR + RFF training accuracy

(b) LR + RFF test accuracy

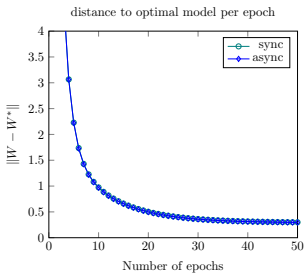


(c) FCNN training accuracy

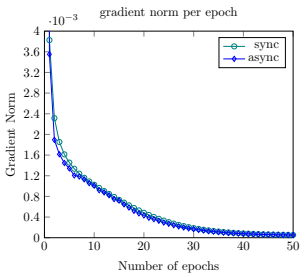


(d) FCNN test accuracy

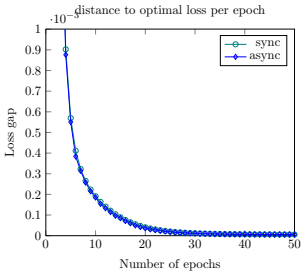
Figure 7. **Accuracy**: RDMA-wild maintains almost the same accuracy as Sync-DSGD for each epoch step while achieving better systems performance.



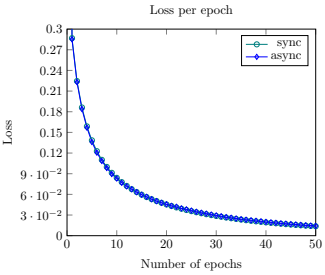
(a) LR + RFF distance to w^*



(b) LR + RFF gradient norm



(c) LR + RFF loss gap to w^*



(d) FCNN loss

Figure 8. **Convergence Analysis**: RDMA-wild converges to the optimal model showing the same performance as Sync-DSGD per epoch.

Table 2. **Training Time Speed-up** of 15 workers compared to a single node. Note that this is the identical experiment/result as Table 1 but just normalized by a single node training time under the same setting. Ideal speed-up is 15X.

MODEL	MODEL SIZE	SYNC	RDMA-WILD
LR	64KB	2.76X	3.08X
LR + RFF	800KB	5.82X	7.42X
FCNN	800KB	1.58X	1.95X

and LR + RFF. As mentioned earlier, we can measure three metrics for convex problems as we can obtain w^* using SVRG or GD, but we can not measure them for FCNN as it is a non-convex problem. Therefore, we just show the loss trend for FCNN.

Figure 8 shows that RDMA-wild converges to the optimal model w^* showing the same performance as Sync-DSGD per epoch for LR + RFF and FCNN.

5.6. Scalability

Lastly, we show the scalability of Sync-DSGD and RDMA-wild compared to a single machine wall-clock time. Differently from the experiment setup of Figure 1 in the section 1, where the batch-size per worker is 60, we set it to 8 which is 7.5 times smaller. This is because we aim to observe RDMA-wild performances under a more challenging setup.

In Figure 1, the RDMA network is not a bottleneck with a mini-batch size of 60 whereas TCP is a bottleneck. Hence, we shift the bottleneck to the network by reducing the mini-batch size to 8. Note that a smaller mini-batch size under the same epochs brings more frequent networking. Analyzing the results in this extreme setting is meaningful as updating the model more frequently with a smaller mini-batch is one of the main reasons why mini-batch SGD outperforms GD.

Table 2 shows interesting results. Even though RDMA-wild achieves 7.42X speed-up for LR+RFF, it achieves poor scalability 3.08X and 1.95X on LR and FCNN, respectively. We conjecture this is because RFF transforms the sparse MNIST dataset (density: 20%) to a high dimensional dense dataset (density: 100%) for better accuracy [1]. Due to this RFF property, matrix computations for LR + RFF takes much longer than LR and FCNN as all elements of LR + RFF matrices should be computed. Note that, for sparse matrix multiplications, most elements can be filled out without any computations by CBLAS. Therefore, the relative network overhead of LR + RFF is much smaller than LR and FCNN. Thus, RDMA-wild could achieve better scalability for LR + RFF in our experiment setup.

Under this extreme setup, RDMA-wild outperforms Sync-

[1] 8,994,156 non-zero elements / 47,040,000 total elements in MNIST, 0 non-zero elements / 600,000,000 total elements in RFF transformed MNIST.

DSGD for all cases in scalability.

6. Limitations & Future Work

To resolve the gradient loss issue, RDMA-wild limits asynchrony between server and workers. We can observe that this limits scalability when the mini-batch size is small through Table 2 and Figure 6. In addition to this, even through RDMA relieves the network bottleneck of the centralized parameter server compared to TCP, we find that a network bottleneck still exists due to its inherent centralized structure as observed in Figure 6. Therefore, future work should explore how to make the server and workers work asynchronously without any gradient loss and how to resolve the server's network bottleneck.

7. Conclusion

RDMA accelerates DSGD as it significantly reduces network overhead. We propose RDMA-wild in which workers work asynchronously, reducing the wait time of workers which results in faster training times than Sync-DSGD. We solve the gradient loss and new gradient detection issues that occur when developing asynchronous DSGD for RDMA.

8. Acknowledgement

The initial steps to this project benefited from early conversations with Yucheng Lu, Haobin Ni, Xinwen Wang, and Siqiu Yao.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283, 2016.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

De Sa, C. M., Zhang, C., Olukotun, K., and Ré, C. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in neural information processing systems*, pp. 2674–2682, 2015.

Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pp. 401–414, 2014.

Frey, P. W. and Alonso, G. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pp. 553–560. IEEE, 2009.

Gupta, V., Kim, H., and Schwan, K. Evaluating scalability of multi-threaded applications on a many-core platform. Technical report, Georgia Institute of Technology, 2012.

Ho, Q., Cipar, J., Cui, H., Lee, S., Kim, J. K., Gibbons, P. B., Gibson, G. A., Ganger, G., and Xing, E. P. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pp. 1223–1231, 2013.

Jha, S., Behrens, J., Gkountouvas, T., Milano, M., Song, W., Tremel, E., Renesse, R. V., Zink, S., and Birman, K. P. Derecho: Fast state machine replication for cloud services. *ACM Trans. Comput. Syst.*, 36(2):4:1–4:49, April 2019. ISSN 0734-2071. doi: 10.1145/3302258. URL http://doi.acm.org/10.1145/3302258.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 583–598, 2014.

Lian, X., Zhang, C., Zhang, H., Hsieh, C.-J., Zhang, W., and Liu, J. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pp. 5330–5340, 2017.

Lu, Y. and De Sa, C. Moniqua: Modulo quantized communication in decentralized sgd. *arXiv preprint arXiv:2002.11787*, 2020.

Rahimi, A. and Recht, B. Random features for large-scale kernel machines. *Advances in neural information processing systems*, 20:1177–1184, 2007.

Recht, B., Re, C., Wright, S., and Niu, F. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pp. 693–701, 2011.

Ren, Y., Wu, X., Zhang, L., Wang, Y., Zhang, W., Wang, Z., Hack, M., and Jiang, S. irdma: Efficient use of rdma in distributed deep learning systems. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 231–238. IEEE, 2017.

Seide, F. and Agarwal, A. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2135–2135, 2016.

Tang, H., Gan, S., Zhang, C., Zhang, T., and Liu, J. Communication compression for decentralized training. In *Advances in Neural Information Processing Systems*, pp. 7652–7662, 2018.

Wei, X., Chen, R., and Chen, H. Fast rdma-based ordered key-value store using remote learned cache. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 117–135, 2020.

Xue, J., Miao, Y., Chen, C., Wu, M., Zhang, L., and Zhou, L. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–14, 2019.

Yann LeCun, Corinna Cortes, C. J. B. *MNIST dataset*, (accessed November 20, 2020). URL http://yann.lecun.com/exdb/mnist/.

Zhang, H., Hsieh, C.-J., and Akella, V. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 629–638. IEEE, 2016.

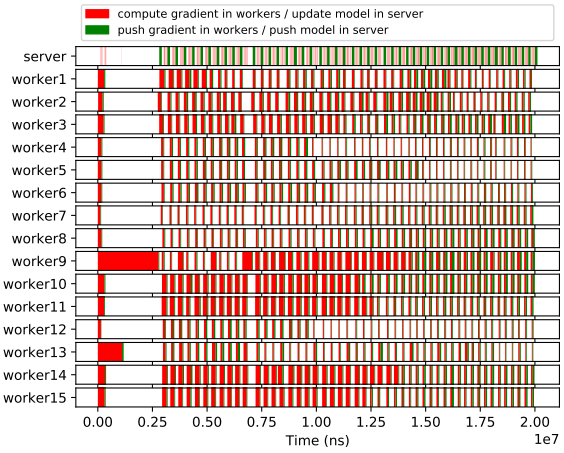


Figure 9. **Sync-DSGD Performance Breakdown (LR)**

A. Appendix

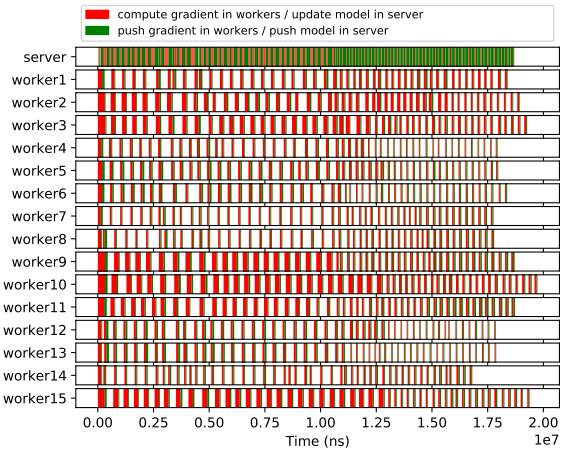


Figure 10. **RDMA-wild Performance Breakdown (LR)**

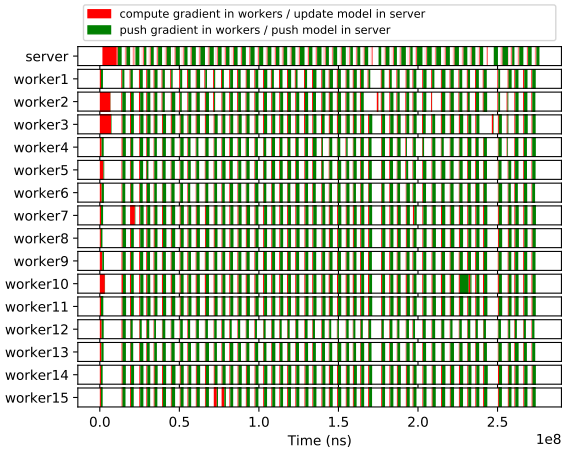


Figure 11. **Sync-DSGD Performance Breakdown (FCNN)**

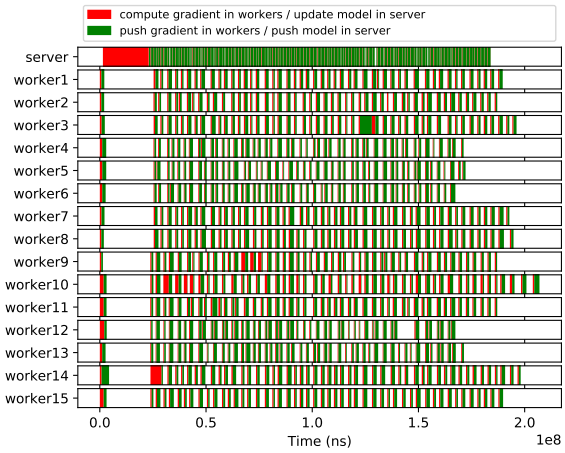


Figure 12. **RDMA-wild Performance Breakdown (FCNN)**