HiPS: Hierarchical Parameter Synchronization in Large-Scale Distributed Machine Learning

Jinkun Geng, Dan Li, Yang Cheng, Shuai Wang, Junfeng Li
Department of Computer Science and Technology, Tsinghua University
Beijing, China

ABSTRACT

In large-scale distributed machine learning (DML) system, parameter (gradient) synchronization among machines plays an important role in improving the DML performance. State-of-the-art DML synchronization algorithms, either the parameter server (PS) based algorithm or the ring allreduce algorithm, work in a flat way and suffer when the network size is large. In this work, we propose HiPS, a hierarchical parameter (gradient) synchronization framework in large-scale DML. In HiPS, server-centric network topology is used to better embrace RDMA/RoCE transport between machines, and the parameters (gradients) are synchronized in a hierarchical and hybrid way. Our evaluation in BCube and Torus network demonstrates that HiPS can better match server-centric networks. Compared with the flat algorithms (PS-based and ring-based), HiPS reduces the synchronization time by 73% and 75% respectively.

ACM Reference Format:
Jinkun Geng, Dan Li, Yang Cheng, Shuai Wang, Junfeng Li. 2018. HiPS: Hierarchical Parameter Synchronization in Large-Scale Distributed Machine Learning. In *NetAI '18: Workshop on Network Meets AI & ML, August 24, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229543.3229544

1 INTRODUCTION

Recent years have witnessed a proliferation of artificial intelligent applications as well as a rapid development of machine learning (ML) technologies. Due to the increasing sizes of training data and models, there is a rising demand on large-scale distributed machine learning (DML) [1, 7, 15, 16, 25]. For instance, the ImageNet Challenge is run on hundreds of GPUs [20] and an internet company under our survey uses several hundreds of dedicated machines to carry out the training for CTR estimation.

In each iteration of DML training, machines need to synchronize the local parameters, or the local gradients to update the model parameters, with all the other machines [25] . After that each machine can continue the next-iteration training based on a global view of the updated model. When the network is large, the parameter (gradient) synchronization cost can be very high. Hence, the synchronization algorithm in large-scale DML plays an important role in affecting the training speed.

NetAI '18, August 24, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5911-5/18/08.
https://doi.org/10.1145/3229543.3229544

In current practice there are three classical synchronization algorithms in DML, namely, parameter server (PS) based synchronization algorithm (PSS) [16], mesh-based synchronization (MS) algorithm[15, 25], and ring allreduce synchronization (RS) algorithm [5, 6, 19]. Typically the former two run on a Clos (Fat-Tree) network and the latter one runs on a ring network. However, when the network size is large, the PS-based algorithm not only suffers from the bottleneck of the parameter server, but also fails to support the RDMA/RoCE transport protocol on commodity switches and server NICs [10, 27]. In practice it is also quite difficult, if not impossible at all, to build a large-scale physical mesh or ring network to run the mesh-based or ring allreduce algorithm. Therefore, the state-of-the-art synchronization algorithms, which work in a flat way, do not perform well when the DML network is large.

In this work we propose a new parameter (gradient) synchronization framework among machines in DML, which we call HiPS. The major difference between HiPS and prior synchronization algorithms is that, parameters (gradients) are synchronized in a hierarchical and hybrid way in server-centric networks, which better embrace RDMA/RoCE transport protocols. In server-centric network topology, like BCube [9], DCell [11], Torus [2], switches are not directly connected with each other, or there is no switch at all. Hence, the spreading scope of PFC pause frames is limited to at most a single switch, as the main memory of servers is large enough to absorb pause frames. HiPS divides the parameter (gradient) synchronization process into multiple stages, with each stage using a specific synchronization algorithm.

The advantage of HiPS over existing synchronization algorithms in large-scale DML comes from two aspects. First, the server-centric network HiPS works on well supports RoCE, which shows considerable performance gains over TCP. Second, compared with using PSS, RS or MS algorithm in server-centric network, HiPS significantly reduces the synchronization time. Based on our evaluation in BCube and Torus, HiPS reduces the synchronization time by more than 70% compared with all the other synchronization algorithms.

The remaining part of this paper is organized as follows. Section 2 illuminates the background and motivation of our work. Section 3 describes the design of HiPS, incorporated with server-centric network topologies. Section 4 presents the evaluation of HiPS in typical server-centric networks. Section 5 summarizes the related work and Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 RDMA vs. TCP

The cost of transport protocol between machines affects the synchronization time in DML. Compared with TCP, RDMA can significantly reduce the transport time. To verify the benefit of RDMA/RoCE

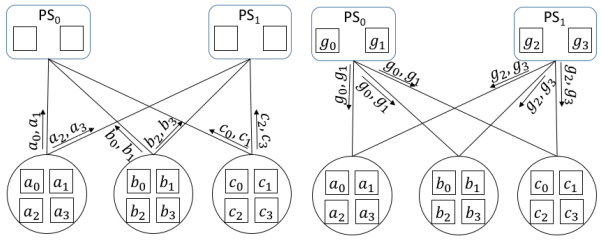

Figure 1: PS-based Synchronization

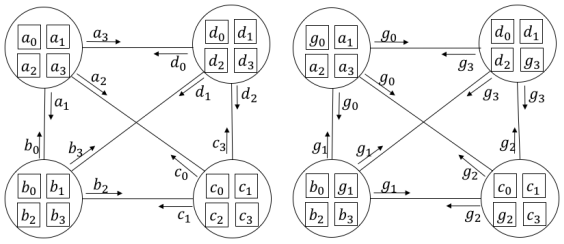

Figure 2: Mesh-based Synchronization

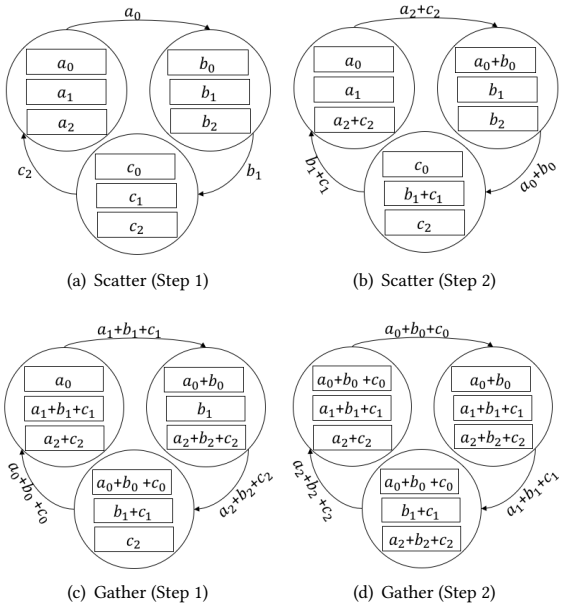

Figure 3: Ring-based Synchronization

in DML, we carry out a simple experiment with the following settings. Two machines are directly connected, each with two Nvidia Tesla K40 GPUs, 32 CPU cores and one 40GE NIC. We train a MNIST model with 1 million parameters for 10000 iterations using Tensorflow. We try different settings of Tensorflow: 1) Running on one machine; 2) Running a worker on one machine and a parameter server on a remote machine, with TCP as the transport protocol; 3) Run with the same setting as 2, but using RoCE as the transport protocol. The experiment result shown in Table 1 demonstrates that the network cost is very high in DML and RoCE-based DML can reduce the total training time by around 50% compared with using TCP. The performance gain is expected to be even more when the size of DML is larger. The result is also consistent with previous works in the literature [10, 26]. Therefore, it is very attractive if we can use RoCE as the transport protocol in large-scale DML.

Table 1: Result of RoCE-based DML VS TCP-based DML

Settings	Time(seconds)
One machine	1459
Two machines by TCP	4278
Two machines by RoCE	2420

2.2 Synchronization Algorithm

For large-scale DML there are three typical synchronization algorithms in the literature, namely, PSS, MS and RS.

(1) PS-based synchronization (PSS) is a centralized synchronization algorithm. There are two roles for each server, namely, parameter server (PS) and worker [16]. The two roles are not mutative and one server can be both PS and worker simultaneously. In PSS, workers work in a *push + pull* way (refer to Figure 1). Firstly, workers generate gradients for parameters and *push* them to PS's, then PS's aggregate the gradients from workers and wait for workers to *pull* them back. PS-based synchronization is the most widely supported algorithm in DML platforms, including Tensorflow [1], Caffe [13], MXNet [22], etc. PSS enjoys a couple of advantages, such as easy deployment, elastic scalability, fault tolerance, and so on.

(2) Mesh-based synchronization (MS) is a decentralized algorithm [15, 25]. Unlike PSS, servers have no functional differences and each one participates in both generating and aggregating gradients. It is worth noting that, when there are equal number of PS's and workers in PSS, by colocating one PS and one worker on each server, PSS becomes MS. Hence, we can treat MS as a special case of PSS. MS works in a *diffuse + collect* way (refer to Figure 2). Considering there are totally N servers in the distributed system ($N = 4$ in Figure 2), each server generates a whole set of gradients and *diffuses* a portion of $\frac{1}{N}$ gradients to each of the other $N-1$ servers; meanwhile, it also receives $N-1$ portions of gradients from other servers and aggregates them with the local portion generated by itself. Then each server obtains $\frac{1}{N}$ global gradients and they again *collect* the global portions from other servers to gain a whole set of global gradients. As shown in Figure 2, the server in the top-left corner sends a_1, a_2, a_3 to other servers respectively and it also receives b_0, c_0, d_0 from them (refer to Figure 2(a)). By aggregating them with the local portion a_0, it gets a global portion of gradients g_0. After that, it sends g_0 to the other 3 servers and also collects their global gradients (i.e. g_1, g_2, g_3) (refer to Figure 2(b)). Eventually, each server gains the whole global gradients. Compared with PSS, MS gets better load balance among servers.

(3) Ring-based synchronization (RS) is also a decentralized algorithm, which is traditionally used in high-performance computing field [19] and recently being studied in DML scenarios [5, 6]. Different from MS, RS works in a *scatter + gather* way (refer to Figure 3). Each server only communicates with one sender and one receiver during the whole synchronization process, thus forming a communication ring in logic. For a distributed system with N servers ($N = 3$ in Figure 3), both *scatter* and *gather* operations can be divided into $N-1$ steps. During each step in *scatter* operation, the server sends one portion of $\frac{1}{N}$ gradients to one neighbor (say *clockwise neighbor*); meantime, it receives another portion of $\frac{1}{N}$

gradients from the other neighbor (*anticlockwise neighbor*) (refer to Figure 3(a)). It then aggregates the received gradients with its local portion. The aggregated portion will be sent to the *clockwise neighbor* in the next step of *scatter* and the server will again receive another aggregated portion from its *anticlockwise neighbor* (refer to Figure 3(b)). *Scatter* operation completes after $N-1$ steps and each server obtains $\frac{1}{N}$ of synchronized gradients. *Gather* operation works in a similar way but it uses the received portion to *replace* its local portion rather than *aggregate* them (refer to Figure 3(c) and Figure 3(d)). *Gather* operation completes after $N-1$ steps and each server obtains the whole set of global gradients. RS does not involve many-to-one communication and thus enjoys a contention-free benefit compared with the two aforementioned algorithms.

2.3 Drawbacks of Flat Synchronization Algorithms

Most DML systems execute parameter (gradient) synchronization with the synchronization algorithms above. However, these algorithm suffer when the network scale is large.

PSS is usually run in Fat-Tree network, which may suffer from RoCE-related problems. Since there can be a large number of workers, not all the workers can be deployed under the same switch with PS's. As a result, some workers need to cover multiple hops in the fat tree to communicate with PS's, thus generating multiple flows and cross traffic, which are potential contributors to PFC deadlock [12] and spreading congestion[21] problems. Besides, the interconnection of switches in fat tree can facilitate the cascading effect of PFC pause storm, which seriously damages the overall performance or even crash down the whole system.

If deploying MS and RS algorithm in Fat-Tree network, they have the same problems as above. If deploying MS in a physical mesh network and deploying RS algorithm in a physical ring network, the physical network is quite difficult to build in practice when the DML size is large. For physical mesh network, it requires each server to equip $N-1$ NIC ports when there are N servers in the network, which is impractical [14]. For physical ring network, the robustness of the network is quite weak when the network size is large, since one server failure will crash the whole ring.

Given the drawbacks of existing flat synchronization algorithms in large-scale DML, we seek to propose a new synchronization algorithm which not only enjoys high synchronization efficiency but can also safely run RoCE as the transport protocol.

3 HiPS DESIGN

3.1 Server-centric Modular Network Topology

Due to the problems including PFC pause frame storm, it is quite challenging to run RoCE in large-scale Ethernet without modifying switches/NICs [10]. The current practice in most data centers is to run RoCE within 1-2 hops of switches, where the spreading scope of PFC pause frame is limited. Therefore, in order to support large-scale DML, we do not recommend switch-centric network topologies like Fat-Tree [3] or VL2 [8]. Instead, server-centric network topologies, such as BCube [9], DCell [11] and Torus [2], shown in Fig. 4(a), Fig. 4(b) and Fig. 4(c) respectively, can well embrace large-scale DML over RoCE for the following reasons.

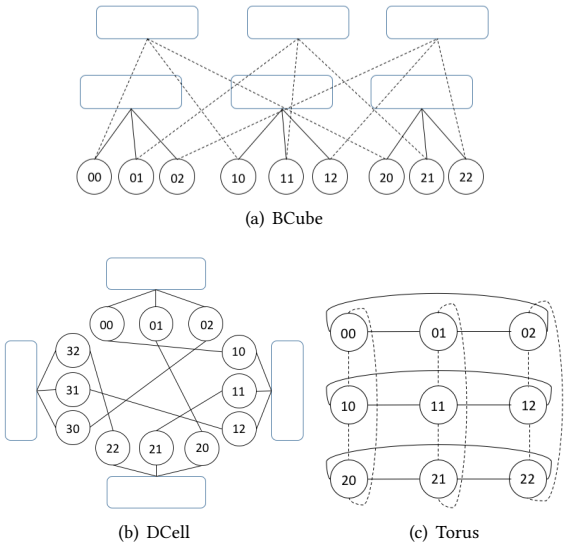

Figure 4: Typical Server-centric Modular Topologies

First, server-centric networks take a modular way to scale to large size by commodity Ethernet devices. Mini-switches with 8~16 ports and commodity servers with 2~4 NIC ports are used to connect thousands of, or even tens of thousands of servers. The size is enough for build a DML cluster.

Second, switches are not directly connected in server-centric networks. As a result, the PFC pause frames spread at most by 1-hop, as servers have enough memory to absorb them. For instance, in BCube network, servers and switches interleave with each other; in DCell network, servers are connected by one switch in the lowest level and in higher levels servers are directly connected; in CamCube network, there are no switches at all and servers are connected as a Torus network. Therefore, the topological characteristic of server-centric networks guarantee that we can safely run DML by RoCE in large scale, without touching the commodity switches and server NICs.

3.2 Hierarchical Synchronization Algorithm

Given the topological characteristics of server-centric modular networks, we can design more efficient parameter (gradient) synchronization algorithms than PSS, MS or RS. The key difference is that servers play an important role in the network interconnection and they can make parameter (gradient) aggregation during the transmission process. Therefore, we can use hierarchical synchronization, in which the synchronization is divided into multiple stages. Each stage possesses two parts (i.e. *push+pull* or *diffuse+collect* or *scatter+gather*) to complete a specific synchronization algorithm, and servers aggregates the partial results before entering the higher-stage synchronization. In this way, the workload is reduced stage by stage and synchronization time is saved.

The general framework of HiPS can be described as Algorithm 1. HiPS partitions the whole server set into different groups in each stage. The number of stages is user-defined as h, which depends on the topological characteristics. For example, in BCube we can

Algorithm 1: HiPS Algorithmic Framework

Input:
N: The number of servers
h: The number of hierarchical levels
g_array: The gradients on the server to synchronize by the process, i.e. $g\_array = [g_i^0, g_i^1, ..., g_i^{N-1}]$
$addr$: the address of the server, i.e. $< a_{h-1}, a_{h-2}, \cdots, a_0 >$

```
1  HiPS()
2      allelic_array = g_array
3      for i ← 0 to h − 1 do
4          rank = addr[i]
5          peer_array =GetPeers(i, addr)
6          allelic_array =
             Sync1(rank, allelic_array, peer_array)
7      end
8      for i ← h − 1 to 0 do
9          rank = addr[i]
10         peer_array =GetPeers(i, addr)
11         allelic_array =
             Sync2(rank, allelic_array, peer_array)
12     end
13     g_array = allelic_array
14 function GetPeers(level, addr)
15     peer_array = []
16     for i ← 0 to n − 1 do
17         peer_addr = addr
18         peer_addr[level] = i
19         append peer_addr to peer_array
20     end
21     return peer_array
```

partition the servers according to switches, and those connected to the same switch are organized as one group[1]. In Torus, we can partition the servers according to the rings, and those involved in the same ring are organized as one group[2]. Meanwhile, the gradients on each server are evenly divided into N portions. During each stage, the server only synchronizes gradients with other servers in the same group.

To better describe the group and synchronization in HiPS, we follow the addressing way in BCube [9] and label each server with an address array, i.e. $addr =< a_{h-1}, a_{h-2}, \cdots, a_0 >$ (refer to Figure 4). Given one server's address, we can get the addresses of its peer servers in the same group during each stage (refer to **GetPeers** in Algorithm 1). Different stages can choose different synchronization algorithms from the three options [3], thus generating a hybrid hierarchical synchronizaton algorithm. For example, if we use RS for each stage (i.e. replace **Sync1** and **Sync2** with **Scatter** and **Gather** described in Algorithm 2), then HiPS develops into a *hierarchical ring-based synchronization* (HRS) algorithm. Specially, when $h = 1$, HRS degrades to the commonly used ring allreduce algorithm described in [19].

Algorithm 2: Hierarchical Ring-based Synchronization

```
1  function Scatter(rank, allelic_array, peer_array)
       // Gradients are evenly split into n portions
2      portion_array = Split(allelic_array, n)
3      right_neighbor = peer_array[(rank + 1) mod n]
4      left_neighbor = peer_array[(rank + n − 1) mod n]
5      for step ← 1 to n − 1 do
6          index = (rank + n − step) mod n
7          portion_send = piece_array[index]
8          Send portion_send to right_neighbor in one hop
9          Receive portion_recv from left_neighbor in one hop
10         index = (rank + n − step − 1) mod n
11         portion_array[index] =
             Aggregate(portion_array[index], portion_recv)
12     end
13     allelic_array = portion_array[rank]
14     return allelic_array
15 function Gather(rank, allelic_array, peer_array)
16     right_neighbor = peer_array[(rank + 1) mod n]
17     left_neighbor = peer_array[(rank + n − 1) mod n]
18     portion_array[0 · · · n − 1] = NULL
19     portion_array[rank] = allelic_array
20     for step ← 1 to n − 1 do
21         index = (rank + n − step − 1) mod n
22         portion_send = piece_array[index]
23         Send portion_send to right_neighbor in one hop
24         Receive portion_recv from left_neighbor in one hop
25         index = (rank + n − step) mod n
26         portion_array[index] = portion_recv
27     end
       // Gradients are concated into one whole array
28     allelic_array = Concat(portion_array)
29     return allelic_array
```

3.3 Application of HiPS in Server-Centric Networks

We take BCube as an example to illustrate the application of HiPS in server-centric networks. As shown in Figure 4(a), given a 2-level BCube, say BCube(3,1), we let $h = 2$ to match the height of BCube and partition the servers according to switches.

In the first stage, we divide servers into 3 groups according to level-0 switches (connected with solid lines in Figure 4(a)). In other words, server 00, 01, 02 form a group; server 10, 11, 12 form another group; and server 20, 21, 22 form the other group. In each group, the servers can choose either PSS/MS[4] or RS. If they use MS, each server will execute *diffuse* operation, sending $\frac{1}{3}$ gradients to the other 2 servers in the group and also receiving $\frac{1}{3}$ gradients from

[1] In Figure 4(a), server 00,01,02 are organized as one group in one stage and 00,10,20 are reorganized as one group in another stage.
[2] Similar to BCube, in Figure 4(c), server 00,01,02 are organized as one group in one stage and 00,10,20 are reorganized as one group in another stage.
[3] In other words, **Sync1** (line 6) and **Sync2** (line 11) can be filled with any of the three synchronization algorithms (*push+pull*, *diffuse+collect*, or *scatter+gather*).
[4] Recall that MS is a special case of PSS.

them. If they use RS, each server will execute *scatter* operation. Since there are 3 servers in the group, the *scatter* operation involves $3-1=2$ steps. During each step, each server will send $\frac{1}{3}$ gradients to a neighbor server in the group and receive $\frac{1}{3}$ gradients from the other neighbor. After the *diffuse*/*scatter* operation is finished, each server gains $\frac{1}{3}$ gradients aggregated from 3 severs in the same group. We call these aggregated gradients *partial gradients* to distinguish them from *global gradients* aggregated from all servers. Then the synchronization enters the second stage.

In the second stage, servers are reorganized into 3 groups according to level-1 switches (connected with dash lines in Figure 4(a)). Similarly, servers in each group can choose MS or RS. However, the workload is reduced to $\frac{1}{3}$ of that in the first stage because each server only needs to synchronize the *partial gradients* obtained in the first stage.

After the servers complete *diffuse+collect* or *scatter+gather* in the second stage, they each get $\frac{1}{3}$ *global gradients*. Then the synchronization returns to the first stage and each server continues to finish the *collect* or *gather* operation with other servers grouped in the first stage (connected with solid lines in Figure 4(a)), corresponding to the *diffuse* or *scatter* operation which has been completed in the first stage. Finally, the two stages are both finished and each server possesses the whole *global gradients* aggregated from 9 servers.

It is worth noting that such a synchronization process only uses one level of switches and links in each stage. To fully utilize the switch and link resources, there can be h parallel processes launched simultaneously to share the synchronization workload , thus accelerating the synchronization by h times.

4 EVALUATION

More generally, HiPS can adapt to typical server-centric topologies. Here we provide 2 cases and compare the synchronization performance in theory. We take global synchronization time (GST) as the metric, which reflects the theoretical time to finish the parameter (gradient) synchronization among all the servers in one iteration. The denotation is provided in Table 2. Based on the denotation, we can calculate and compare the GSTs with different synchronization algorithms in BCube and Torus as follows.

Table 2: Denotation

N	The total number of servers
n	The number of servers in each group, $n=\sqrt[h]{N}$ for BCube and Torus
h	The number of hierarchical stages in HiPS
W	The total size of gradient parameters on each server
B	The bandwidth capacity of each NIC, i.e. the full speed of each link

4.1 GST in BCube

(1) BCube+HiPS. The incorporation of BCube and HiPS has been discussed in Section 3.3. Each stage can choose either MS or RS to synchronize. With equal synchronization workload, MS and RS achieve equal GSTs [5]. Besides, there can be h processes executed in parallel. Each process utilizes one NIC with bandwidth of B and synchronizes $\frac{W}{h}$ gradients in a hierarchical way. Then $GST=2\times\frac{W/h}{B}\frac{n-1}{n}\sum_{j=0}^{h-1}\frac{1}{n^j}=2\frac{n^h-1}{n^h}\frac{W}{hB}$.

(2) BCube+PSS/MS. Section 2 has pointed out that MS is a special case of PSS, with each server as both worker and parameter server. In such symmetric topologies as BCube, it is obvious that the ordinary PSS (worker number does not equal to PS number) incurs more imbalance in bandwidth utilization and leads to lower efficiency compared with MS. Therefore, we only consider the GST for BCube+MS. According to Theorem 6 in [9], links are equally used in all-to-all communication and the aggregate bottleneck throughput (ABT) can be derived as $\frac{n}{n-1}(N-1)B$. Besides, the *diffuse* and *collect* can be both treated as a process with $\frac{N-1}{N}W$ shuffled data on each server. thus the total shuffled data is $2\frac{N-1}{N}W\times N=2(N-1)W$, so $GST=2(N-1)W\div\frac{n}{n-1}(N-1)B=2\frac{n-1}{n}\frac{W}{B}$.

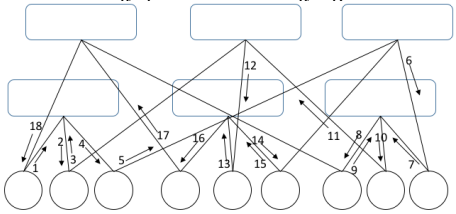

Figure 5: Logic Ring Path in BCube

(3) BCube+RS. Although BCube enjoys rich connectivity, there is only B bandwidth that can be utilized by the logic ring in BCube: Assume there are 2 edge-disjoint paths between any two neighboring servers in the logic ring. The first path between them covers at least 1 hop (i.e. 2 links), then the second path (edge-disjoint from the first path) covers at least $1+2=3$ hops (i.e. 6 links). To guarantee that each two neighboring servers have 2 edge-disjoint paths connected, there are at least $2N+6N=8N$ links required in BCube. However, BCube contains hN links in total and usually $h<8$ in BCube (BCube(4,6) ($h=7$) already contains more than 15000 servers!), thus cannot suffice the two edge-disjoint paths between each two neighboring servers. Therefore, servers in BCube can form a logic ring with bandwidth of B at most. For example, in Figure 5 the path $1\rightarrow 2\rightarrow\cdots\rightarrow 18$ forms a ring in BCube(3,1). Besides, there are some links of which the uplink bandwidth and downlink bandwidth are both occupied (such as 2 and 3, 8 and 9, 14 and 15 in Figure 5), thus it is unable to halve the GST with bandwidth in opposite directions. Therefore, $GST=2\frac{N-1}{N}\frac{W}{B}=2\frac{n^h-1}{n^h}\frac{W}{B}$.

4.2 GST in Torus

(1) Torus+HiPS. As for Torus topology, we set h to match the dimensions in Torus (e.g. $h=2$ for 2D-Torus and $h=3$ for 3D-Torus). Then we can use HiPS to adapt to the topology and RS is adopted in each stage of synchronization. There are h parallel processes executed focusing on the synchronization in different dimensions. Take 2D-Torus as an example (refer to Figure 4(c)), there are 2 parallel processes on each server, denoted as p_1 and p_2. Each executes 2 *scatter* operations and 2 *gather* operations. During the first *scatter* operation, p_1 synchronizes gradients among the 3 servers in horizontal direction along the solid paths whereas p_2 synchronizes gradients among the 3 servers in vertical direction along the dash paths. During the second *scatter* operations, p_1 synchronizes gradients among servers in vertical direction whereas p_2 synchronizes

[5] Considering n servers, each with W size of gradients and B bandwidth, if they execute MS or RS to synchronize their gradients, it can be calculated that $GST=2\frac{n-1}{n}\frac{W}{B}$

gradients among servers in horizontal direction. After the 2 *scatter* operations are completed, *gather* operations are executed correspondingly in a similar way. Since each stage contains a ring path with bidirectional bandwidth (i.e. bandwidth in both directions can be utilized for synchronization), the total synchronization time can be calculated as $GST = \frac{W/h}{B}\frac{n-1}{n}\sum_{j=0}^{h-1}\frac{1}{n^j} = \frac{n^h-1}{n^h}\frac{W}{hB}$.

(2) Torus+PSS/MS. Since it is difficult to derive a general formula for mesh-based GST in hD-Torus, we turn to find the lower bound of GST. hD-Torus can be wholly embedded into a BCube$(n, h-1)$ with each link bandwidth of $2B$, thus GST of mesh-based synchronization in hD-Torus will not be faster than that in the BCube, which is $2\frac{n-1}{n}\frac{W}{2B}$. Therefore, $GST \geq \frac{n-1}{n}\frac{W}{B}$.

(3) Torus+RS. Each server in Torus can be considered as a node with even degrees in an *undirected connected graph*. Then according to *Euler's Theorem*, these servers can form a cycle (logic ring). Besides, if mapped the logic ring into the Torus topology, it will use links in each dimension (e.g. the logic ring will use both solid links and dash links in Figure 4(c)), causing the remaining links unable to form another ring to connect all servers. Therefore, there is at most one logic ring formed and bandwidth can be used in both directions, thus $GST = \frac{N-1}{N}\frac{W}{B} = \frac{n^h-1}{n^h}\frac{W}{B}$.

The comparative results can be summarized as Table 3. We compare the GSTs for several combinations of (n, h). We choose (4,2), (8,3) and (16,4) and calculate the corresponding GSTs in BCube and Torus. As for BCube and Torus under the three typical configurations (illustrated in Figure 6 and Figure 7), HiPS reduces the GST by 37.5%~73.3% compared with PSS/MS and by 50.0%~75.0% compared with RS, which demonstrates the significant performance gains of HiPS over the two baselines in large-scale DML.

Table 3: Comparative Results of GST

	BCube	Torus
HiPS	$2\frac{n^h-1}{n^h}\frac{W}{hB}$	$\frac{n^h-1}{n^h}\frac{W}{hB}$
PSS/MS	$2\frac{n-1}{n}\frac{W}{B}$	$\frac{n-1}{n}\frac{W}{B}$
RS	$2\frac{n^h-1}{n^h}\frac{W}{B}$	$\frac{n^h-1}{n^h}\frac{W}{B}$

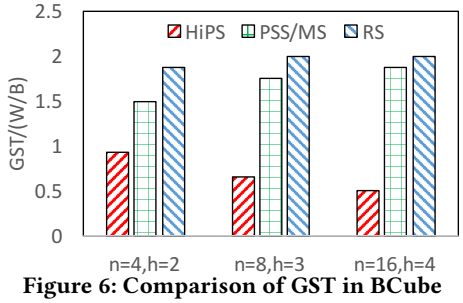

Figure 6: Comparison of GST in BCube

5 RELATED WORK

Data Center Network Topology. Topology design has ever been a hot topic in data center networking (DCN) and there have been numerous topologies proposed in prior works, with different emphasis on efficiency, scalability, cost and other aspects . Generally speaking, the topologies can be classified into two main categories: *switch-centric* and *server-centric*. The former category only relies on switches for packet routing and forwarding and typical examples include Fat-Tree [3] , VL2 [8], PortLand [18] and so on. By contrast, the latter requires forwarding intelligence of both switches

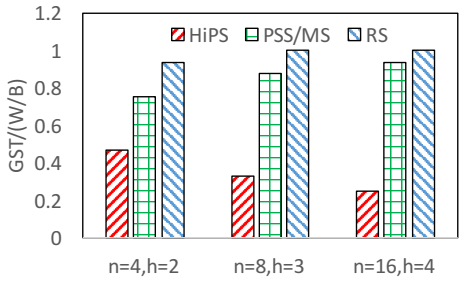

Figure 7: Comparison of GST in Torus

and servers. More specifically. in *server-centric* topologies, servers can also be considered as mini-switches and they act as intermediate nodes to transfer data packets for other servers. DCell [11] , BCube [9] , Ficonn [14], CamCube [2] , etc. fall in this category.

RDMA over Converged Ethernet (RoCE). Among the three main branches of RDMA technologies, namely, RoCE, Infiniband and iWarp, RoCE is the most commonly applied in industry, with a multi-vendor ecosystem [10, 27]. Besides, prior works [17, 24, 27] have also demonstrated the outperformance of RoCE compared with the other two alternatives. Therefore, we focus on the application of RoCE, however, the common RDMA-related issues are also worth considering for Infiniband and iWarp application in practice.

Synchronous Parallel Model. Since the computation performance and communication latency of servers (computing nodes) vary in large-scale DML, proper synchronous parallel models are necessary for servers to coordinate and synchronize with others. Bulk Synchronous Parallel model (BSP) [23] has become the widely applied model in DML. However, BSP may suffer from performance damages due to *straggler problem* [7]. Recent works, such as *partial barrier* [4] and SSP [7], relax the original constraints in BSP to gain more tolerance to stragglers. Currently, we only focus on the parameter (gradient) synchronization under BSP model. More extensive studies are worthwhile on the performance improvement under other synchronous models.

6 CONCLUSION AND FUTURE WORK

The incorporation of physical topology with synchronization algorithm is necessary for large-scale RoCE-based distributed machine learning. This paper proposes HiPS, which combines hierarchical synchronization algorithm with server-centric topologies to mitigate RoCE-related problems and achieve good synchronization performance. The future research will focus on two main directions. Firstly, we will conduct further comparative study of RoCE-based synchronization under different topologies. Besides, we will integrate the proposed synchronization into popular machine learning systems to boost their performance.

ACKNOWLEDGEMENT

The work was supported by the National Key Basic Research Program of China (973 program) under Grant 2014CB347800, National Key Research and Development Program of China under Grant 2016YFB1000200 and the National Natural Science Foundation of China under Grant No. 61522205, No. 61772305, No. 61432002, No.61672499. Dan Li is the corresponding author of this paper.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of USENIX OSDI'16*, pages 265–283, 2016.

[2] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, et al. Symbiotic routing in future data centers. In *Proceedings of ACM SIGCOMM '10*, pages 51–62, 2010.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM '08*, pages 63–74, 2008.

[4] J. Albrecht, C. Tuttle, A. C. Snoeren, et al. Loose synchronization for large-scale networked systems. In *Proceedings of the ATC '06*, pages 28–28, 2006.

[5] S. Alexander and B. Mike, Del. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[6] G. Andrew. Bringing hpc techniques to deep learning, 2017.

[7] H. Cui, J. Cipar, Q. Ho, et al. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the ATC'14*, pages 37–48, 2014.

[8] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, et al. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM '09*, pages 51–62, 2009.

[9] C. Guo, G. Lu, D. Li, et al. BCube: A high performance, server-centric network architecture for modular data centers. In *Proceedings of ACM SIGCOMM '09*, pages 63–74, 2009.

[10] C. Guo, H. Wu, Z. Deng, et al. RDMA over commodity ethernet at scale. In *Proceedings of ACM SIGCOMM '16*, pages 202–215, 2016.

[11] C. Guo, H. Wu, K. Tan, et al. DCell: A scalable and fault-tolerant network structure for data centers. In *Proceedings of ACM SIGCOMM '08*, pages 75–86, 2008.

[12] S. Hu, Y. Zhu, P. Cheng, et al. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *Proceedings of the ACM HotNets '16*, pages 92–98, 2016.

[13] Y. Jia, E. Shelhamer, J. Donahue, et al. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[14] D. Li, C. Guo, H. Wu, et al. Scalable and cost-effective interconnection of data-center servers using dual server ports. *IEEE/ACM Trans. Netw.*, 19(1):102–114, Feb. 2011.

[15] H. Li, A. Kadav, E. Kruus, et al. Malt: Distributed data-parallelism for existing ml applications. In *Proceedings of ACM EuroSys '15*, pages 3:1–3:16, 2015.

[16] M. Li, D. Andersen, J. W. Park, et al. Scaling distributed machine learning with the parameter server. In *Proceedings of USENIX OSDI'14*, pages 583–598, 2014.

[17] Mellanox. RoCE vs. iWARP competitive analysis. Technical report, 2017.

[18] R. Niranjan Mysore, A. Pamboris, N. Farrington, et al. Portland: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of ACM SIGCOMM '09*, pages 39–50, 2009.

[19] P. Pitch and Y. Xin. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117 – 124, 2009.

[20] G. Priya, D. Piotr, R. Girshick, et al. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[21] M. Radhika, S. Alex, P. Aurojit, Z. Eitan, K. Arvind, et al. Revisiting network support for RDMA. Technical report, 2017.

[22] C. Tianqi, L. Mu, L. Yutian, et al. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[23] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[24] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, et al. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *Proceedings of IEEE HOTI '12*, pages 48–55, 2012.

[25] P. Watcharapichat, V. L. Morales, R. C. Fernandez, et al. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of ACM SoCC '16*, pages 84–97, 2016.

[26] B. Yi, J. Xia, L. Chen, and K. Chen. Towards zero copy dataflows using RDMA. In *Proceedings of the ACM SIGCOMM'17 Posters and Demos*, pages 28–30, 2017.

[27] Y. Zhu, H. Eran, D. Firestone, et al. Congestion control for large-scale RDMA deployments. In *Proceedings of ACM SIGCOMM '15*, pages 523–536, 2015.