Int J Parallel Prog
DOI 10.1007/s10766-017-0520-3

Improving the Performance of Distributed TensorFlow with RDMA

Chengfan Jia[1] · **Junnan Liu**[1] · **Xu Jin**[1] · **Han Lin**[1] · **Hong An**[1] · **Wenting Han**[1] · **Zheng Wu**[1] · **Mengxian Chi**[1]

Received: 27 August 2017 / Accepted: 18 September 2017

Abstract TensorFlow is an open-source software library designed for Deep Learning using dataflow graph computation. Thanks to the flexible architecture of TensorFlow, users can deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. In a distributed TensorFlow work process, it uses gRPC to connect between different nodes. However, when deploying training tasks on high performance computing clusters, the performance of gRPC becomes a bottleneck of distributed TensorFlow system. HPC clusters are usually equipped with Infiniband network, in addition to traditional TCP/IP network. But open-sourced TensorFlow has not taken this advantage. We present a RDMA-capable design of TensorFlow. By porting the Tensor send/receive parts of TensorFlow into RDMA verbs, we finally get nearly 6× performance improvements over the original distributed TensorFlow, based on gRPC. The TensorFlow system with RDMA support shows a great scalability among the training scale.

Keywords Distributed TensorFlow · RDMA · Infiniband · Optimization

1 Introduction

In recent years, Machine Learning has driven advances in many different fields. Moreover, benefiting from the rapid growth of computing power, Deep Learning has drawn wider and wider attention from researchers and engineers.

TensorFlow [1] is a numerical computation system designed for Machine Learning, which supports a variety of applications, with a focus on training and inference on deep

✉ Chengfan Jia
jcf94@mail.ustc.edu.cn

1 University of Science and Technology of China, Hefei 230026, Anhui, China

neural networks. It can efficiently support large-scale training and inference as well as flexibly support experimentation and research into new models and system-level optimizations.

Deep Learning process usually requires a large amount of training data and powerful computing recourses to deal with the data. With the increasing size of the datasets, a single compute node is highly restrictive.

Some studies on it build frameworks over single node TensorFlow to achieve good scalability—such as Spark-based framework [2] or MPI-based distributed system [3]. The official Distributed TensorFlow was open-sourced in April, 2016.

1.1 Problem Statement

Training a Neural Network with multiple hidden layers is usually a time consuming work. For example, the champion algorithm of 2012, ImageNet Large Scale Visual Recognition Challenge—later called Alexnet [4]—consists of five convolutional layers and three fully connected layers with nearly 60 million parameters. According to the paper, the training process for roughly 90 cycles through the training set of 1.2 million images took 5–6 days on two NVIDIA GTX 580 3 GB GPUs. Despite the sustained growth of computing power, a greater network will still be a pressing need in the future.

However, the open-sourced TensorFlow did not seem to show an advantage in performance as we expected. Recent research from HKBU [5] tested several popular Deep Learning frameworks; TensorFlow did not perform well in their test results.

In order to evaluate the distributed TensorFlow, we wrote a distributed Alexnet benchmark based on the Alexnet CPU version provided by HKBU dlbench project. Data scalar of this benchmark is the same as the original algorithm, while the input data gets replaced by random pixels.

The test is performed in a 8-node CPU cluster. Each one has dual Intel Xeon E5-2660 processors with 1000M Ethernet and Mellanox 40G QDR InfniBand. The test counts for 300 steps' elapsed time, with each step of 24 RGB "pictures" of 224×224 pixels as input data.

Figure 1 shows our test results. Single node TensorFlow takes 224 seconds, while the elapsed times of the same data size in the distributed version are much more than that. In addition, expanding from 2 workers to 3 workers (2 compute nodes to 3 compute nodes) shows bad scalability. The average CPU utilization of each node during the whole test is limited to a low level. It is obvious that communication with gRPC through TCP/IP becomes a bottleneck of the distributed system. Under such distributed circumstances, GPU can finish the computing part much faster, while the latency caused by network communication may become more obvious over the whole process.

1.2 Contributions

Aiming to improve the performance of the distributed TensorFlow system and make use of the Infiniband network deployed in our HPC clusters, this paper presents a

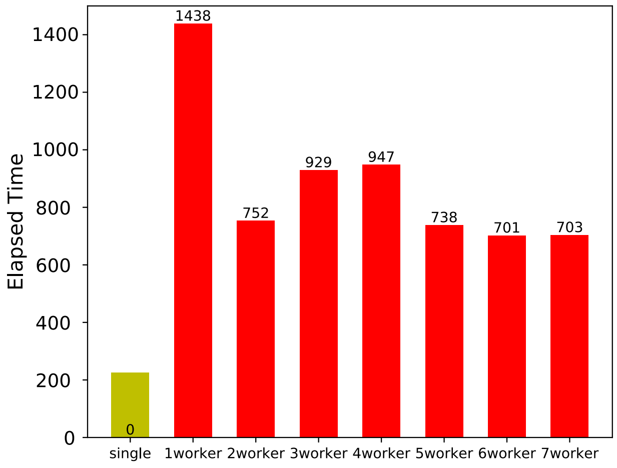

Fig. 1 Test results of Alexnet in Distributed TensorFlow using TCP/IP

RDMA-capable design of TensorFlow, which, by porting the Tensor send/receive parts into RDMA verbs, achieves great performance improvement over the original gRPC version.

The rest of the paper is organized as follows:

Background material is presented in Sect. 2, as well as some related works. Section 3 presents our contribution to TensorFlow. We first analyze the workflow of TensorFlow's send/receive parts, and then design a RDMA protocol to fit into the distributed TensorFlow. Some other optimization methods are used to improve it. Section 4 shows the evaluations of our work. Section 5 concludes our work.

2 Background Review

2.1 TensorFlow Architecture

TensorFlow uses a dataflow graph to organize its computations and provides a friendly interface for users. The master translates user requests into execution. Then the dataflow executor handles requests from the master, and schedules the execution of the kernels that comprise the graph. Its runtime contains over 200 standard operations; computation works can be efficiently done with the control of the executor.

Each operation resides on a particular device, such as a CPU or GPU in a particular task. A device is responsible for executing a kernel for each operation assigned to it.

Distributed TensorFlow works similarly to multi-device execution [6]. After device placement, a subgraph is created per machine. Send/Receive node pairs that communicate across worker processes use remote communication mechanisms to move data across machine boundaries. Google's paper mentioned that their work supports both TCP and RDMA, while the open-source version we get now only implements gRPC to organize communication.

2.2 gRPC

gRPC is an open-source framework from Google for handling remote procedure calls [7]. It is based on the HTTP/2 standard, and enables easy creation of highly performant, scalable APIs and micro services in many popular programming languages and platforms.

gRPC uses "completion queue" to manage the performing of multi-threaded tasks, and the execution of every task is associated with the "grpc_exec_ctx" closure mechanism. In order to asynchronously establish the connection and transport data, gRPC uses "pollset" to manage input and output file descriptors. By polling the file descriptors of each "pollset" using *epoll*, once TCP socket starts to connect, data either comes in from TCP or the remote TCP connection is going to shutdown – the callback functions that are combined with specific file descriptors will be called to handle these different conditions.

2.3 Infiniband and RDMA

The InfiniBand Architecture [8] defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network, hosts are connected to the fabric by Host Channel Adapters (HCAs). InfiniBand uses a queue-based model. A Queue Pair in InfiniBand consists of two queues: a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication operations are described in the Work Queue Requests (WQR), or descriptors, and submitted to the work queue. The completion of WQRs is reported through Completion Queues (CQs). Once a work queue element is finished, a completion entry is placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished.

InfiniBand Architecture supports both channel semantics and memory semantics. In channel semantics, send/receive operations are used for communication. In memory semantics, InfiniBand provides Remote Direct Memory Access (RDMA) operations, including RDMA Write and RDMA Read. RDMA operations are one-sided and do not incur software overhead at the remote side. This enables true application bypass message passing. The processor on the machine can continue its computation task without bothering with incoming messages. Thus, RDMA can positively impact the computation and communication overlap. Additionally, RDMA Write operation can gather multiple data segments together and write all data into a contiguous buffer at the receiver end. RDMA Write with Immediate data is also supported. With Immediate data, a RDMA Write operation consumes a receive descriptor and then can generate a completion entry to notify the remote node of the completion of the RDMA Write operation.

IB verbs is an implementation of the RDMA verbs for Infiniband (according to the Infiniband specifications). It handles the control path of creating, modifying, querying and destroying resources such as Protection Domains (PD), Completion Queues

(CQ), Queue-Pairs (QP), Shared Receive Queues (SRQ), Address Handles (AH) and Memory Regions (MR). It also handles sending and receiving data posted to QPs and SRQs, getting completions from CQs using polling and completion events.

The control path is implemented through system calls to the ib verbs kernel module, which further calls the low-level hardware driver. The data path is implemented through calls made to the low-level hardware library, which, in most cases, interacts directly with the hardware and provides kernel and network stack bypass (saving context/mode switches) along with zero copy and an asynchronous I/O model.

2.4 Related Works

Several studies mentioned in Sect. 1 avoid the use of offically open-sourced distributed TensorFlow, as well as the gRPC. They try to build a higher level scheduling framework and data gather/scatter framework (by Spark or MPI) over non-distributed TensorFlow. Such a framework maintains the training data itself, and single node TensorFlow is just used as its sub-module or so-called "backend." Their works generally require few or no changes to the TensorFlow runtime, while users have to deal with some other issues themselves (such as split, merge, gather, or scatter the data) rather than benefit from the flexibility of the distribued TensorFlow system.

Another approach is to port the gRPC module into RDMA, aiming to improve the performance of gRPC and indirectly speed up the distributed TensorFlow. Just as some discussions from the industry and other researchers do [9], our *previous work* (finished last year) has already tried this method. The result of it is about to triple the performance of the original TCP version.

Just at the time we finished this paper (around April 2017), Yahoo submitted a pull request for their RDMA version to TensorFlow and it was accepted in TensorFlow 1.2.0-rc0 on May 20th, 2017. Since they have not given any written statements yet, we will compare their work with ours later in this paper.

3 Porting TensorFlow into RDMA

3.1 Send/Receive Process in TensorFlow

Dataflow implementation simplifies distributed execution, as it makes communication between subcomputations explicit. TensorFlow runtime places operations on devices, subject to implicit or explicit constraints in the graph.

Once the operations in a graph have been placed, and the partial subgraph has been computed, TensorFlow partitions the operations into per-device subgraphs. A per-device subgraph for device *d* contains all of the operations that were assigned to *d*, with additional *Send* and *Recv* operations that replace edges across device boundaries. *Send* transmits its single input to a specified device as soon as the tensor is available, using a rendezvous key to name the value. *Recv* has a single output, and blocks until the value for a specified rendezvous key is available locally, before producing that value.

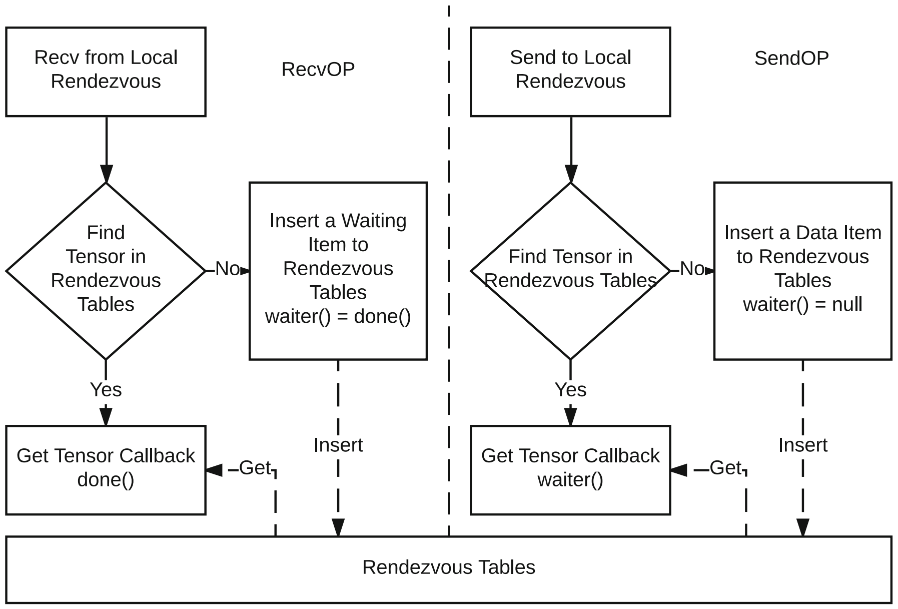
Recv from Local Rendezvous
RecvOP
Find Tensor in Rendezvous Tables
No
Insert a Waiting Item to Rendezvous Tables waiter() = done()
Yes
Get Tensor Callback done()
Get
Insert
Send to Local Rendezvous
SendOP
Find Tensor in Rendezvous Tables
No
Insert a Data Item to Rendezvous Tables waiter() = null
Yes
Get Tensor Callback waiter()
Get
Insert
Rendezvous Tables

Fig. 2 Single node workflow of Send/Recv operations

Figure 2 shows the workflow of *Send/Recv* operations within the same machine. The Tensor data transmitted by *Send/Recv* operations are indexed in an object called *Rendezvous*. The *Recv* operation first checks the rendezvous key of the value in *Rendezvous* Tables. If the rendezvous key exists (inserted by a previous *Send* operation), just get the Tensor data and return it by a callback function. Otherwise, it inserts a waiting item to the *Rendezvous* Tables, which contains the requiring rendezvous key as well as saves the callback function as a waiter. When the *Send* operation is processed, it will check the *Rendezvous* Tables, too. If the rendezvous key exists (inserted by a previous *Recv* operation), just get the waiting item and run the waiter function with the sending Tensor data. Otherwise, it inserts a data item into the *Rendezvous* Tables, which contains the sending rendezvous key and the sending Tensor data.

We should notice that *Send/Recv* operations within a single machine (run in the non-distributed TensorFlow) use memory (*Rendezvous* object) to perform the communication, while the network communications occur in distributed environment. Figure 3 shows such a workflow.

Communication starts at the receive side. *Recv* operation sends a *Data Request* to the send side asking for Tensor data. The waiting service of the send side performs a *Recv* operation to get data from local *Rendezvous* object. Once the Tensor is ready, it responses the data to the receive side. Receive side then runs the callback function to finish the remote *Recv* operation.

The *Data Request* and *Data Response* processes are done by network communications or, as in this case gRPC specifically. That's what our work focuses on.

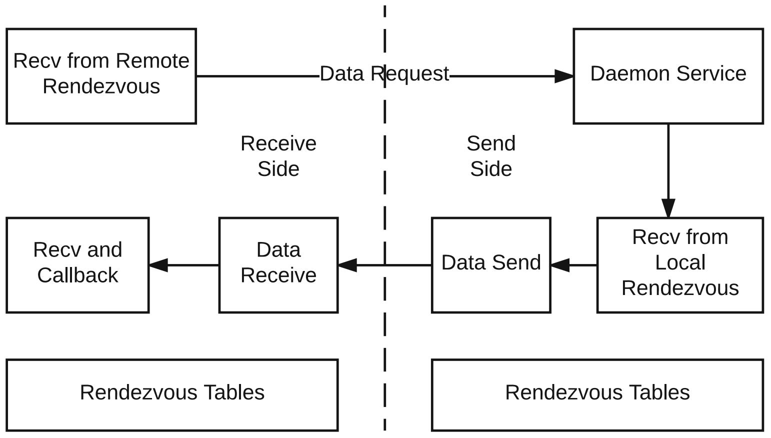

Fig. 3 Cross node workflow of Send/Recv operations

3.2 RDMA Process Design

Distributed TensorFlow uses the rendezvous mechanism to organize its communication process.

Typically, in a Rendezvous Protocol, the sender and the receiver negotiate the buffer availability on both sides before the message transfer actually takes place. For achieving high performance message passing for large messages, it is critical that message copies are avoided. The Rendezvous Protocol provides a way to achieve zero-copy message transfer because the sender can know the location of the receivers buffer, or vice-versa.

Several studies about RDMA have shown good performance in achieving computation/communication overlap with fast memory registration based RDMA Write Process [10] or RDMA Read-Based Rendezvous Protocol [11]. We learn from their works and design our own protocols base on TensorFlow workflow.

To modify the communication process into RDMA, we design a RDMA read-based protocol and a RDMA write-based protocol to replace the original gRPC calls.

After the connection of Master Service and Worker Service in distributed TensorFlow cluster, the RDMA service of each side gets connected. RDMA contexts (like *lid, qpn, psn*, and so on) are exchanged through TCP/IP at first, while in the following process, TCP/IP is no longer used and all of the communications pass through RDMA. The basic protocol is illustrated in Fig. 4.

In the RDMA read protocol (Fig. 4a), communication starts by the receive side. *Recv* operation sends a *Tensor Request* message to the send side asking for Tensor data. The waiting service of the send side performs a *Recv* operation to get data from the local *Rendezvous* object, and at the same time, prepares a memory buffer for upcoming RDMA operations, including buffer creations and memory bind registrations. Once the Tensor is ready, it responses the *data size*, *data address* and *rkey* to the receive side. Receive side then prepares a corresponding receive buffer and calls RDMA read to get the Tensor data from remote memory. After the completion of RDMA read process,

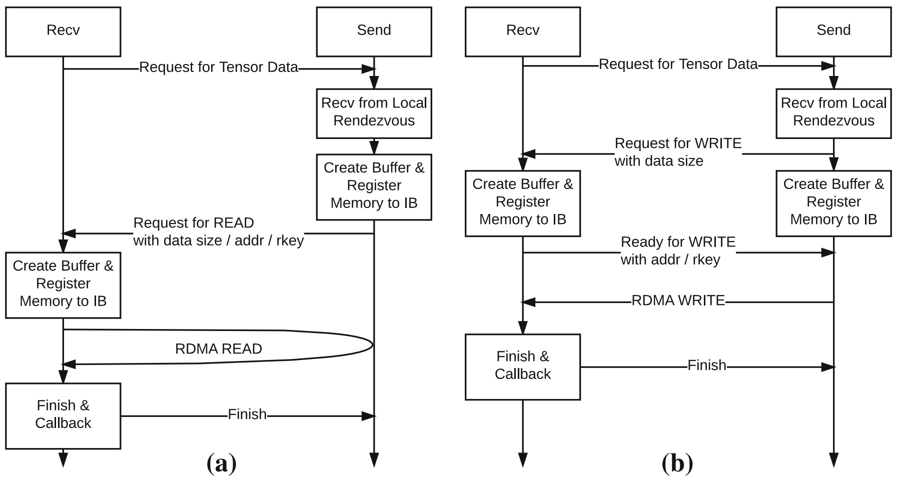
Recv
Send
Request for Tensor Data
Recv from Local Rendezvous
Create Buffer & Register Memory to IB
Request for READ with data size / addr / rkey
Create Buffer & Register Memory to IB
RDMA READ
Finish & Callback
Finish
(a)
Recv
Send
Request for Tensor Data
Recv from Local Rendezvous
Request for WRITE with data size
Create Buffer & Register Memory to IB
Create Buffer & Register Memory to IB
Ready for WRITE with addr / rkey
RDMA WRITE
Finish & Callback
Finish
(b)

Fig. 4 RDMA protocol for distributed TensorFlow. **a** RDMA READ and **b** RDMA WRITE

receive side then runs the callback function to finish the *Recv* operation, as well as sends a *Buffer Release* message to inform the send side.

RDMA write protocol (Fig. 4b) is similar to the process described above. Read protocol uses fewer interacting messages, while write protocol overlaps communication time and buffer register time in each side.

In our evaluations, these two protocols acquire almost the same performance after further optimizations.

Yahoo's design of IB verbs is similar to our RDMA write protocol. Our current work finally gets about 10% more performance than their version.

The study on RDMA Read-Based Rendezvous Protocol [11] shows a great advantage to use RDMA read protocol with dynamic interrupt over RDMA write. Regarding limitations of the TensorFlow Runtime, we have not finished the interrupt part yet and our work has the potential for further improvement.

3.3 Memory Optimization

Studies about memory management of the RDMA process [12] have shown the importance of hidden memory management costs.

One of the key differences between RDMA and TCP/IP communication is that the application has to explicitly manage the memory segments that will be used as communication buffers. The application has to preregister certain parts of its memory with the RDMA subsystem as source and/or destination buffers for the data transfers. During registration, the memory pages get pinned by the OS, making sure they stay resident and cannot be swapped out to disk. The pinned pages are then registered with the RDMA-enabled NIC (RNIC) so that it can access them using DMA operations, which eliminates the need for OS callbacks and intermediate buffering during transfers.

The RDMA syntax refers to these registered memory segments as Memory Regions (MR).

MR registration happens through the resource management path, which requires kernel activity and therefore induces a delay, as well as a non-negligible CPU load. Even though the expensive data copy operations are avoided with RDMA (zero copy), the explicit buffer management renders RDMA useless for applications that are not able to deal with their buffers efficiently.

Messages Buffer Reuse Each side of the RDMA pair has a fixed message send buffer and message receive buffer, local send buffer is associated with the remote receive buffer when exchanging the RDMA context. The messages used during interactions have similar structures, so when sending another message we only rewrite the different parts of the message and reuse the same part of the buffer.

Tensor Data Buffer Reuse During the Deep Learning training process or inferencing process, different input data goes through the same data path, which means a Tensor of the data flow graph will be sent and received many times during the whole process. MR registration is a time consuming operation in both RDMA read and write protocols, but these times can be cut off by memory reuse. Each time when a new block of memory is registered to MR, we store its information of it (address and rkey) to a memory table by Tensor name, Tensor source, and Tensor destination. The buffer stored in the memory table will never be released when a pair of Send/Recv operations are completed. The next time the same Tensor is sent or received, we can just get the data buffer from the memory table instead of registering a new block of memory.

Memory Pool Implementation The data buffer reuse method described above, however, has the disadvantage of using too much memory. We then came up with a memory pool implementation to reuse buffer and cut down memory usage. After the connection of the RDMA pair, we allocate a big block of memory from OS and register it to Infiniband. The memory block is used as a memory pool. When *Send* operation or *Recv* operation require a data buffer, we can just get memory from the big block. Special "malloc" and "free" methods are used for allocating memory from the memory pool and releasing memory to the memory pool.

Furthermore, we try to preload our own memory allocator library to replace the standard C memory allocator ("malloc" and "free"). In this way, our runtime does not even need to request memory during *Send* and *Recv* operations, since all of the memory blocks are managed under our own memory pool. This part is still in development.

4 Evaluation

In this section, we evaluate our proposed designs for the RDMA optimized distributed TensorFlow. We have declared in Sect.1 that the network latency has nothing to do with whether CPU or GPU is used to perform the computing.

Our platforms for evaluation are:

Cluster A 8 SuperMicro nodes with dual Intel Xeon E5-2660 processors. Each node has 128 GB of main memory. The nodes are connected with 1000M Ethernet and Mellanox 40G QDR InfiniBand.

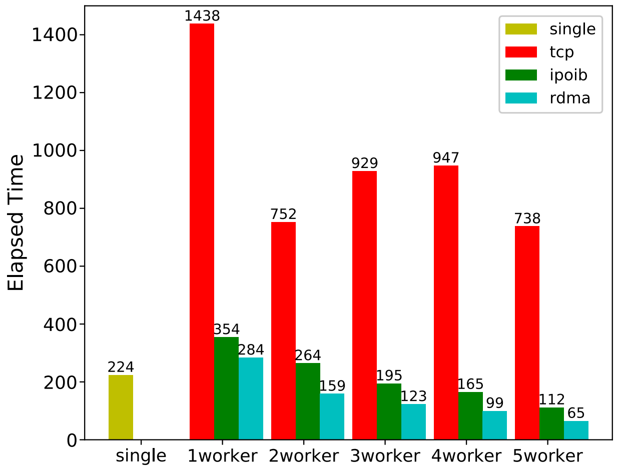

Fig. 5 Test results of Alexnet in Distributed TensorFLow in Cluster A

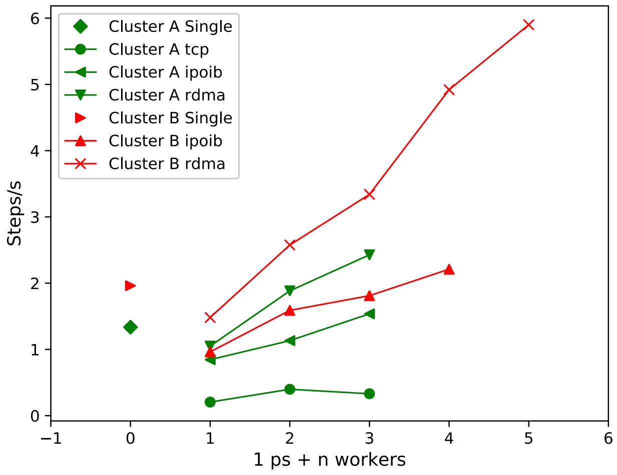

Fig. 6 Test results of Alexnet in Distributed TensorFLow in Cluster A

Cluster B 6 SuperMicro nodes with dual Intel Xeon E5-2698 processors. Each node has 128 GB of main memory. The nodes are connected with Mellanox 100G EDR InfiniBand.

We expect that the RDMA optimization reduces the communication latency in the distributed TensorFlow system and improves its performance.

Figure 5 shows the results of the Alexnet test case described in Sect. 1.1. gRPC through ipoib shows a 3 to 4 times speed-up in comparison to the original TCP/IP version. This benefit comes from the hardware (Infiniband to Ethernet).

During the TCP/IP test, it can be observed clearly that CPU usages of the computer nodes are very low. Moreover, the CPU usages even drop in periodicity to less than 10% (when waiting for communication). This bottleneck is caused by communication latency of TCP/IP gRPC. When using ipoib or RDMA optimized version, CPU usages can be kept at a high level.

Our work over RDMA is about 30–40% faster than ipoib gRPC, and almost 6 times faster than TCP/IP gRPC.

A further test in Fig. 6 describes steps per second the TensorFlow system can train, which shows that our RDMA optimized distributed TensorFlow has a great scalability. Each step of Alexnet test case input 24 RGB pictures of 224×224 pixels; the data path includes inference, gradients computing, and parameters updating. With the help of powerful CPU and Infiniband, RDMA optimized distributed TensorFlow acquires nearly linear growth in performance.

5 Conclusion

This paper presents a RDMA-capable design of TensorFlow, which, by porting the Tensor send/receive parts into RDMA verbs, achieves great performance improvement over the original TCP/IP gRPC version.

We analyze the workflow of distributed TensorFlow and especially focus on the send/receive parts of it. Then we design a RDMA read-based and a RDMA write-based rendezvous protocol to fit into the distributed TensorFlow workflow. Several optimizations are used. In our evaluation, the RDMA optimized distributed TensorFlow shows a good performance over the original TCP/IP version, as well as a good scalability with the increasing of compute nodes.

TensorFlow is a well-designed framework that uses a dataflow model to organize its computation. Benefiting from the dataflow model, we believe that it has the potential to gain more performance after further optimizations.

Acknowledgements This research is conducted under Advanced Computer System Architecture (ACSA) Laboratory of University of Science and Technology of China, supported by the National Key Research and Development Program of China (Grant No. 2016YFB1000403).

References

1. Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: A System for Large-Scale Machine Learning. arXiv:1605.08695 (2016)
2. Kim, H., Park, J., Jang, J., et al.: DeepSpark: A Spark-Based Distributed Deep Learning Framework for Commodity Clusters. arXiv:1602.08191 (2016)
3. Vishnu, A., Siegel, C., Daily, J.: Distributed TensorFlow with MPI. arXiv:1603.02339 (2016)
4. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. Adv. Neural Inf. Process. Syst. **25**, 1097–1105 (2012)
5. Shi, S., Wang, Q., Xu, P., et al.: Benchmarking State-of-the-Art Deep Learning Software Tools. arXiv:1608.07249 (2016)
6. Abadi, M., Agarwal, A., Barham, P., et al.: Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467 (2016)
7. Google Developers.: Introducing gRPC, a New Open Source HTTP/2 RPC Framework. http://googledevelopers.blogspot.com/2015/02/introducing-grpc-new-opensource-http2.html (2015)
8. Pfister, G.F.: An introduction to the infiniband architecture. High Perform. Mass Storage Parallel I/O **42**, 617–632 (2001)
9. Mellanox.: The Mellanox Solution to TensorFlow. http://www.mellanox.com/solutions/machine-learning/tensorflow.php (2016)

10. Ou, L., He, X., Han, J.: An efficient design for fast memory registration in RDMA. J. Netw. Comput. Appl. **32**(3), 642–651 (2009)
11. Sur, S., Jin, H.W., Chai, L., et al.: RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 32–39. ACM (2006)
12. Frey, P.W., Alonso, G.: Minimizing the hidden cost of RDMA. In: ICDCS'09. 29th IEEE International Conference on Distributed Computing Systems, 2009, pp. 553–560. IEEE (2009)