ShmCaffe: A Distributed Deep Learning Platform with Shared Memory Buffer for HPC Architecture

Shinyoung Ahn[†‡], Joongheon Kim[*], Eunji Lim[‡], Wan Choi[‡], Aziz Mohaisen[§], and Sungwon Kang[¶]
[†]Dept. Info. & Comm. Engineering, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea
[‡]High Performance Computing Group, Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea
[*]School of Computer Science and Engineering, Chung-Ang University, Seoul, Korea
[§]Department of Computer Science, University of Central Florida, Orlando, FL, USA
[¶]School of Computing, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea
E-mails: syahn@etri.re.kr, joongheon@cau.ac.kr, ejlim@etri.re.kr,
wchoi@etri.re.kr, mohaisen@ucf.edu, sungwon.kang@kaist.ac.kr

***Abstract*—One of the reasons behind the tremendous success of deep learning theory and applications in the recent days is advances in distributed and parallel high performance computing (HPC). This paper proposes a new distributed deep learning platform, named ShmCaffe, which utilizes remote shared memory for communication overhead reduction in massive deep neural network training parameter sharing. ShmCaffe is designed based on Soft Memory Box (SMB), a virtual shared memory framework. In the SMB framework, the remote shared memory is used as a shared buffer for asynchronous massive parameter sharing among many distributed deep learning processes. Moreover, a hybrid method that combines asynchronous and synchronous parameter sharing methods is also discussed in this paper for improving scalability. As a result, ShmCaffe is 10.1 times faster than Caffe and 2.8 times faster than Caffe-MPI for deep neural network training when Inception_v1 is trained with 16 GPUs. We verify the convergence of the Inception_v1 model training using ShmCaffe-A and ShmCaffe-H by varying the number of workers. Furthermore, we evaluate scalability of ShmCaffe by analyzing the computation and communication times per one iteration of deep learning training in four convolutional neural network (CNN) models.**

I. Introduction

In modern artificial intelligence research, Deep Neural Network (DNN), an approach introduced to improve machine learning performance even with complicated input data features, is getting a lot of attention in academia and industry. The power of DNN has been verified through many applications, especially in visual perception where it showed even better accuracy than the human vision system. The success of deep learning in the areas of voice recognition and visual object recognition is based on the availability of (i) massive training data set and (ii) distributed and parallel high performance computing (HPC) architectures. Especially, general purpose graphic processing units (GPGPUs) clearly have played an important role in HPC architecture [1], [2], [3], [4].

Even though it is possible to build more accurate learning models with more training data and larger/deeper DNN models, the computation requirement increases exponentially with the multiplication of model sizes and training data volumes [5], [6]. The large computations cannot be handled in a single machine. Thus, such models require distributed deep learning platforms which manage high performance computing resources [8].

In distributed deep learning platforms, the workers, which conduct distributed DNN training, should frequently share massive DNN training parameters. This sharing introduces communication overhead and such an overhead increases when we have larger DNN models and more workers. When the communication overhead is high, the waiting time of the computation processors, such as CPU and GPU, in computation nodes increases. Therefore, distributed deep learning platforms should have functions for massive parameter sharing for communication overhead reduction.

In this paper, a distributed deep learning parallel processing architecture based on remote shared memory is proposed. The proposed architecture achieves communication overhead reduction and, therefore, can be utilized in high performance computing clusters that are connected via Infiniband interconnection networks. In addition, it uses remote direct memory access (RDMA), eliminating communication for data copy operations between application-level buffer and kernel-level buffer. Our approach directly stores deep learning parameters (initially stored in local machine memory) in remote node memory, and thus it can achieve deep learning parameter sharing among workers with reduced communication overheads. The architecture we propose in this paper is named to ShmCaffe, as it is an extension to Caffe, a widely architecture used in image classification, object detection, and localization. ShmCaffe provides efficient massive parameter sharing that enables communication overhead reduction. ShmCaffe's performance is comparable to BVLC Caffe [7] and other MPI-based deep learning platforms such as Caffe-MPI[1] and MPICaffe.

The rest of this paper is organized as follows. Section II presents the related work and Section III explains system architecture, components, distributed deep learning platforms, shared memory allocation methods, and asynchronous distributed parameter updates. Section IV shows the performance evaluation results. Section V concludes this paper.

[1]https://github.com/Caffe-MPI/Caffe-MPI.github.io

II. Related Work

Deep learning is one of machine learning techniques based on artificial neural network (ANN), which allows machines to learn by simulating a human's biological neurons. The ANN models are evolving from basic deep neural network (DNN) composed of only full-connected layers where all features between the adjacent layers are fully connected, to convolutional neural network (CNN) specialized for image recognition and object detection, and to recurrent neural network (RNN) suitable for time-series data training and learning.

The deep learning is a repetition of the feed forward process and the back propagation process. The feed forward process calculates feature values and objective functions from the input layer to the output layer through several hidden layers and outputs the result. The back propagation process modifies the weight of each layer from the output layer to the input layer through the hidden layer, reflecting the error (the difference between the result of the feed-forward and the correct answer). The weights that are modified during the training process are updated repeatedly until the error is minimized. In distributed deep learning, all computers must exchange their learning with others by sharing the updated weights.

As parallel processing techniques for distributed DNN training, there are data parallelism and model parallelism. Data parallelism divides the training data into deep learning workers, which conduct distributed DNN training [1]. Model parallelism is a way where workers train a part of a model for the same data. Generally, data parallelism is widely used, but the model parallel method is used when it is difficult to train a big model in a single node or a single GPU due to limited memory size. In cuda-convnet2, a data-model hybrid parallelism is also used: the approach performs data parallelism for the convolution layer and uses model parallelism for the fully-connected layer [3], [4], [5], [9]. In this paper, we propose fast distributed training methods based on data parallelism.

For distributed DNN training, it is necessary to share each explored parameter among distributed workers. The parameter update can be done synchronously or asynchronously. The synchronous method aggregates deep learning gradients by sending the gradients to the parameter server or by mutually exchanging them among deep learning workers per each training iteration and updates the weights. The asynchronous method is a way in which the parameter server updates the global weight whenever gradient arrives from a worker, without aggregating all the gradients arriving late or early from the distributed workers. The synchronous method has a large aggregation overhead because there is a variation in the training time of each deep learning worker. However, since the asynchronous method can eliminate such aggregation overhead, it can train DNN quickly without sacrificing the accuracy.

The Gradient Descent (GD) is the most commonly used optimization method for training DNN. Mini-batch Stochastic GD (SGD) differs from GD in that it only uses a gradient learned from arbitrarily sampled data, not a gradient learned from the entire data [10]. The SGD method is suitable for processing large data sets. SGD is also used for distributed DNN training. In Synchronous SGD (SSGD), distributed workers calculate their gradients from their mini-batch data and then each worker or parameter server aggregate the gradients to update the global weight.

Asynchronous SGD (ASGD) is one of the most widely used asynchronous distributed variants of SGD. The ASGD has been proposed to address the disadvantage of SSGD; namely, workers have to wait until the slowest worker finishes calculating gradient [11], [12]. The ASGD algorithm has proven to converge on convex problems [13]. However, it does not guarantee a linear speed performance as the number of workers increases. ASGD also is limited in improving the training performance due to the delayed gradient problem [14]. ASGD uses a parameter server to share parameters asynchronously between workers.

The elastic averaging SGD (EASGD) method was proposed to maximize the benefits of DNN exploration. It allows each deep learning worker to maintain its own model replica as in any other ASGD methods. The distributed training is performed by updating the global model with the moving average [12]. In this method, unlike the ASGD method, the parameter server and the workers exchange the weight parameter learned by them and calculate the difference between the two weight vectors, thereby updating their own weight parameters by adding the scaled difference to it. This method performs better than the Downpour SGD by reducing the delay time of global weight updating between the parameter server and local workers.

There are various distributed deep learning frameworks, such as Hogwild, Dogwild, DistBelief, and Adam. Hogwild showed that distributed deep learning workers handling sparse gradients can train DNN through the asynchronous SGD in a shared memory architecture without locking [15]. The Dogwild framework extended the Hogwild to propose an architecture that supports deep learning parameter sharing through a proprietary read/write to a global distributed parameter buffer. It implemented asynchronous SGD by extending parallelism in the Caffe library [7], [16]. The DistBelief framework proposed the Downpour SGD, which supports a large number of model replicas as a variant of the asynchronous SGD and Sandblaster batch optimization technique which supports various distributed models [9]. Each deep-learning worker can train weights at different rates to make the most of computing and network resources. In particular, one can use the resources of a heterogeneous HPC system, consisting of CPUs and GPUs of unequal specifications, to take advantage of the maximum computational power.

The parameter server is an essential component of ASGD. Separate dedicated parameter servers are placed; otherwise the master worker acts as a parameter server. The parameter server allocates a memory area for storing global parameters in its own local memory, updates global parameters with parameters sent periodically from slave workers and then distributes the

updated global parameters to the slave workers. Distributed platforms such as Petuum and CNTK also use distributed key-value store developed specifically for parameter servers. The parameter server provides a flexible coherence model, flexible scalability, and a continuous fault tolerance [17], [18], [19].

Training deep networks with billions of parameters using tens of thousands of CPU cores was found unscalable and cost ineffective for training large scale DNNs [5], [9], [20]. As GPU's computational power has improved to much more than ten times that of CPU, deep learning using multiple GPUs has been studied [21]. In addition, some hardware accelerators, including FPGAs and Intel Xeon Phi processors, as well as GPUs, can be used for DNN training [22].

III. ShmCaffe

A. Architecture

We developed ShmCaffe, a distributed deep-learning platform that uses remote shared memory for parameter sharing, by extending BVLC Caffe (version 1.0.0) [7]. ShmCaffe does not only provide the ASGD among all workers, but also provides a hybrid method that combines Inter-node Asynchronous SGD and Intra-node Synchronous SGD. ShmCaffe runs distributed deep learning workers on multiple nodes, and exchanges initialization messages between the distributed processes using MPI. In order to exchange parameters between the distributed Inter-node workers, the asynchronous EASGD algorithm is modified to use remote shared memory buffer for parameter sharing. As shown in Fig. 1, ShmCaffe uses the Soft Memory Box (SMB) Framework, which provides remote shared memory facilities[23]. ShmCaffe uses Caffe as a deep learning computation library with very small modifications. The distributed training manager performs initialization of the distributed processing using the MPI programming model and performs parameter exchange handling using the remote shared memory library provided by the SMB library.

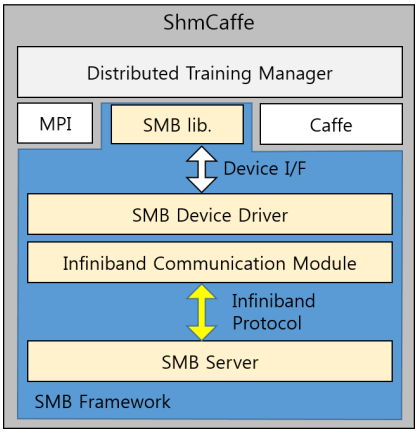

Fig. 1: High-level architecture of ShmCaffe

ShmCaffe uses BVLC Caffe's deep-learning computation algorithm without modification and includes the Shared memory based Elastic Averaging SGD (SEASGD) algorithm, which we propose in this paper. The first MPI process (rank = 0) is responsible for creating remote shared memory buffers, distributing shared memory generation key, and initializing parameter as master worker. The other MPI process (rank $\neq$ 0) is a set of slave workers that allocate the remote shared memory buffer created by the master worker as shown in Fig. 2. ShmCaffe supports all hyper-parameters supported by Caffe and additionally supports two hyper-parameters: *update_interval* and *moving_rate*. The *update_interval* is a hyper-parameter indicating how frequently to update global weight parameters. The *moving_rate* is moving averaging rate; a scaling factor used when workers update the global and local weights.

B. Virtual Shared Memory Framework

To accelerate the communication speed among the distributed deep learning workers, we have developed a virtual shared memory framework, SMB, from scratch except the Infiniband Communication Module, which was developed through the modification of open source Reliable Datagram Sockets (RDS) included in linux kernel main line. The SMB framework allows remote shared memory (RSM) buffers to be used between processes distributed across multiple nodes, allocates the granted memory of the memory providing node on the high-speed RDMA-enabled network as a shared memory buffer, read and/or write remote shared memory buffer by the RDMA mechanism. SMB provides APIs to the application process to exchange control messages, such as remote shared memory allocation/deallocation, RDMA Read/Write to the assigned RSM, accumulation between remote shared memory segments, and update notification.

Fig. 2 shows the process of creating shared memory buffers among distributed deep-learning workers and sharing parameters through it. The master worker uses the SMB API to create shared memory buffers on the SMB server. Once the creation of a shared memory buffer is completed, the master worker broadcasts the SHM key for the shared memory buffer to the other workers that want to share it. The workers receiving the SHM key send a shared memory allocation request with the memory size and the SHM key to the SMB server, and the SMB server provides the access key for the assigned shared memory buffer. This key is the Infiniband remote key that enables remote machine to access directly the shared memory with RDMA. Once the sharing procedure is completed, the shared memory buffers of the SMB server can be shared between the distributed deep learning workers.

C. Elastic Averaging SGD with Shared Memory

Fig. 3 shows the mechanism of SEASGD (Shared memory based EASGD), the asynchronous parameter update method based on the remote shared memory framework (SMB) used by ShmCaffe. Namely it shows how parameters and training progress information are shared, updated between workers and how control message is exchanged between SMB and workers. In Fig. 3, one worker trains DNN from the deep learning data. A worker is an MPI process or a thread that train a model replica. The deep learning data is assigned to all workers

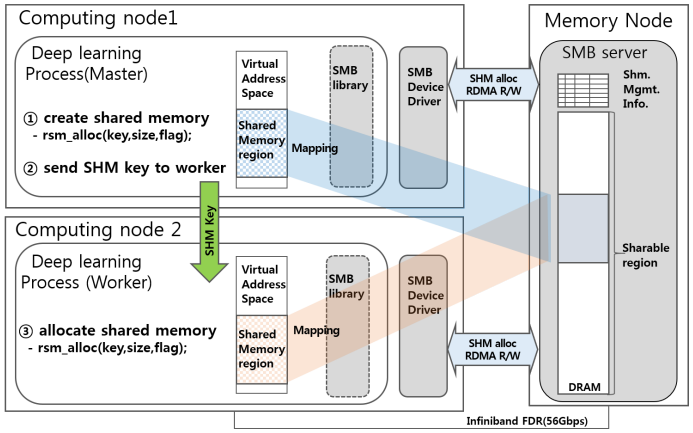

Fig. 2: Allocating remote shared memory

without duplication. Each worker calculates gradients on one mini-batch, updates local weight parameters from the gradients first, reads the global parameters, and updates local weights second from the global weights.

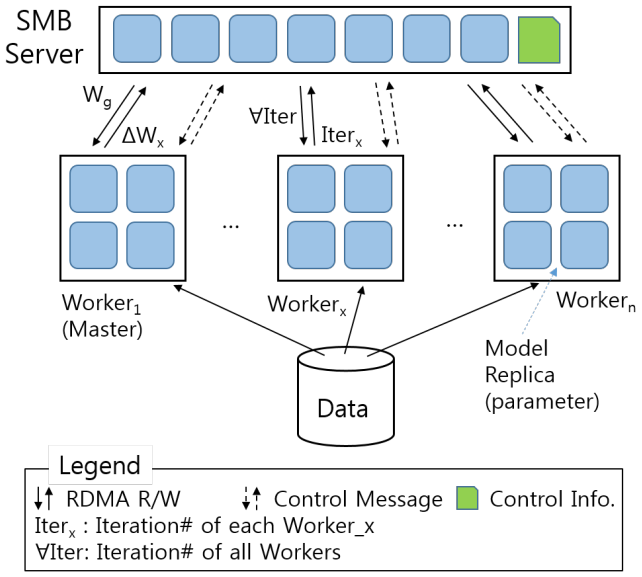

Fig. 3: SEASGD: Shared memory based EASGD

Weights are exchanged between the parameter server and the workers when using the EASGD scheme. EASGD is more efficient than the Downpour SGD, in which the weight update is performed by the parameter server. In the EASGD method, the weight update is performed by both the worker and the parameter server. In the Downpour SGD method, workers send the gradient learned by them to the parameter server, and the parameter server updates the global weights based on the received gradients (1) and distributes the updated global weights to all workers. In (1), W_g denotes global weights, η the learning rate, and G_x denotes gradient of worker x.

$$W_g' = W_g - \eta G_x \tag{1}$$

On the other hand, the worker using the EASGD method updates the local weight from local gradient learned by itself after the training of each minibatch in (2). The updated local weight and the global weight are exchanged between workers and the parameter server, and they update their own weight based on the difference of exchanged weight. Equation (3) is the second weight update formula in the workers, and (4) is the weight update formula in the parameter server. As workers use the learning rate (η) when updating the local weight in (2), the moving averaging rate (α) is used as in (3) and (4).

$$W_x' = W_x - \eta G_x \tag{2}$$
$$W_x'' = W_x' - \alpha\left(W_x - W_g\right) \tag{3}$$
$$W_g' = W_g' + \alpha\left(W_x - W_g\right) \tag{4}$$

ShmCaffe use the SGD optimizer of Caffe to update the local weight. ShmCaffe workers calculate weight increment (ΔW_x) in (5) and update the local weights (6). The workers store the weight increment in the shared buffer of the SMB server, and updates the global weights (W_g) by accumulating the weight increment into the global weights of the SMB server (7). This is because the SMB server does not provide the parameter update logic which parameter servers provide(it provides shared memory buffer and simple accumulation functionality between shared memory buffers).

$$\Delta W_x = \alpha(W_x' - W_g) \tag{5}$$
$$W_x'' = W_x' - \Delta W_x \tag{6}$$
$$W_g' = W_g + \Delta W_x \tag{7}$$

D. Hybrid SGD

ShmCaffe can reduce inter-node network traffic and provide model performance improvements through synchronous SGD between Intra-node GPUs. Currently, ShmCaffe uses a single SMB server. Because the communication bandwidth of the single SMB server is bound to the bandwidth of the network interface, the communication overhead increases significantly when training large-scale models with many workers. ShmCaffe groups workers assigned to the same node as shown in Fig. 4. The same group of workers aggregates gradients (G_{grp_x}) using ncclAllReduce provided by the NVIDIA NCCL library, and then update the local weight (W_{grp_x}) from the aggregated gradients. Next, the root worker of the same worker group asynchronously updates the global parameters on the SMB server using SEASGD. The root worker updates the local weight (W_{grp_x}) from the global parameter and broadcasts the updated weight (W_{grp_x}") to other workers of the same group. In the Hybrid SGD (HSGD), the role of the master worker is performed by the root worker of Master $Worker\_Group_1$ in Fig. 4.

E. Aligning Termination of All Workers

ShmCaffe provides additional ways to adjust the learning progress of workers to increase utilization of computation resources by aligning the training end time. Even if ASGD-based workers train a DNN by using the same GPUs exclusively, deviations in computation time between deep learning

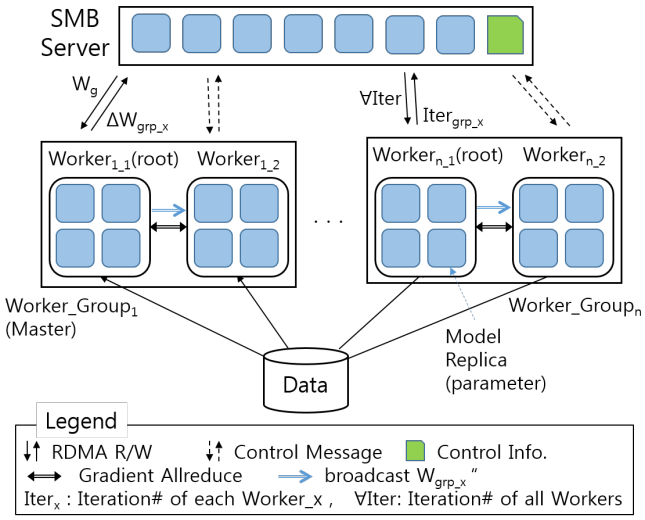

Fig. 4: HSGD: a combination of SSGD and SEASGD

workers will occur. This deviations occurs because workers share the system bus, file system I/O, and network bandwidth. BVLC Caffe terminates training by specifying the number of iterations rather than terminating when the predetermined accuracy or loss is reached. It is difficult for all workers to finish training at the same time. All workers that have completed the specified training iterations must wait for the slowest worker to finish its training while occupying GPU.

Therefore, a coordinator is needed. When a master worker performs this role, each distributed deep learning worker must report to the master the iteration count it completed, and the master worker should have separate threads to collect and adjust progress information for every worker. ShmCaffe can share progress information between distributed workers through the shared memory provided by SMB.

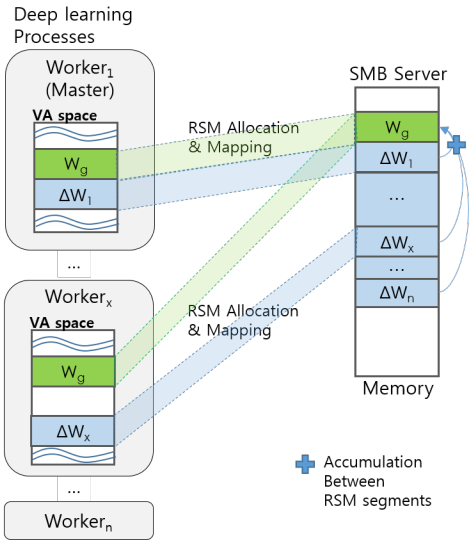

Fig. 5: RSM allocation and global parameter update

ShmCaffe workers share training progress information $(\forall Iter, Iter_x)$ through the SMB shared memory buffer (control info. in Fig. 3 and Fig. 4). According to this shared progress information, each worker can finish training almost at the same time by adjusting the number of learning iterations according to the predetermined criteria. The predefined criteria are: 1) all workers finish training when the master worker terminates, 2) all workers finish training according to the time of the worker who finishes first, 3) all worker finished when the average number of iterations of all workers reach the specified number of iterations.

F. Use of Shared Memory Buffer at SMB Server

In the SEASGD scheme, the shared memory for parameter is allocated as shown in Fig. 5. The global weight parameter (W_g) buffer created in the SMB server is shared by all deep learning workers. Each worker allocates a shared memory buffer (ΔW_x) at the SMB server to store the weight increment computed from the difference between its local weight and the global weight. This buffer is not shared among the other workers. The values of ΔW_x are accumulated to the global weight buffer (W_g), as shown in Fig. 5. Because each worker can read and write parameter from and to (ΔW_x) exclusively, the parameters can be shared at high speed by making full use of the physical performance provided by Infiniband.

G. SEASGD Procedures in ShmCaffe

Fig. 6 shows the procedures of SEASGD training implemented in ShmCaffe. Worker_x spawns a thread, *update_thread*, which updates global weight parallelly while *main_thread* execute deep learning. The spawned *update_thread* is blocked until it is waked up by *main_thread*.

When the training begins, the local weight will be updated from the global weight before the training, then the deep learning will be performed with the updated weight, and also the local weight is updated with the calculated gradient during the iteration by the Worker_x *main_thread*. Thus, each worker's *main_thread* reads the global weight (W_g) at the start point of every iteration (T1) and updates the local weight by calculating weight increment from the difference between the global weight and the local weight (T2). Next, *main_thread* wakes up the *update_thread* to hide the communication time of global weight update (T3), the worker's *main_thread* trains a mini-batch and calculates gradient (T4), and finally updates the local weight (T5). This procedure is repeated for the specified iterations.

In the Fig. 6, reading global weight and update local weight operations (T1 + T2) by the *main_thread* and writing/updating global weight operations (T.A1 T.A4) by the *update_thread* should be mutually exclusive. When the *update_thread* is waked up, then it first acquire lock and stores the weight increment into the shared memory of the SMB server (T.A1) and sends a global weight accumulation request to the SMB server (T.A2). The SMB server exclusively processes the cumulative update requests of global weights from each worker (T.A3) and notifies them of the result. When the *update_thread* receive the result form SMB server (T.A4), it releases the lock

and waits for next wake-up signal from the *main_thread*. If the updating global weight by the *update_thread* takes longer time than the training(T4) and updating local weight(T5) of the *main_thread*, then the *main_thread* is blocked before (T2) until *update_thread* wakes it up(T.A5). As described, ShmCaffe does not hide the time of reading the global weight from the time of computation, because the learning performance deteriorates due to the delayed (or stale) parameter problem.

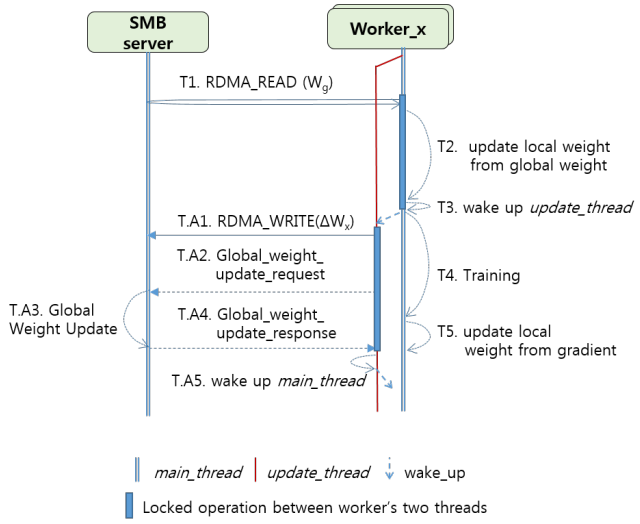

Fig. 6: SEASGD procedures for asynchronous training

IV. Experimental Study

A. System Setup

For an intensive experimental study and performance evaluation, 6 SuperMicro 4028GR-TRT2 servers were used, with the following specifications. Each server is equipped with 2 socket Intel Xeon CPU (E5-2690 v4, 14 cores, 2.3GHz), 128GB DDR4-2400MHz/ECC memory, and 4 Nvidia GPUs (GTX Titan X pascal). A memory server is also used, which is equipped with 2 socket Intel Xeon CPU (E5-2609 v2, 4 cores, 2.5GHz), 256GB DDR3-1866MHz memory, and one Mellanox Infiniband switch. Each server has one 56Gbps FDR Infiniband host channel adapter (HCA). For utilizing the high performance computing system, Ubuntu 14.04-LTS OS (kernel version 3.13.0-123-generic) is used. As GPU programming toolkit, CUDA v8.0 (including cudnn 8.0) is used, while Mvapich2-2.2 is additionally used as an MPI package.

B. Read/Write Bandwidth in a SMB Server

In our system setup, we measured the performance of Read/Write from/to the allocated shared memory of single SMB server while the number of processes increases from 2 to 32. The reason why we focus on the Read/Write performance is due to the fact that the read/write workload is exactly 50% mixed in the distributed deep learning communications. In this experiment, each process allocates the shared memory buffer of 1GB and conducts Read/Write (each 50% mixed) after the shared memory allocation. Notice that the experiment is conducted for 10 times. Fig. 7 shows that the aggregated bandwidth of the Read/Write traffic workload increases up to 6.7 GB/s. If we consider the fact that the maximum bandwidth of Infiniband HCA is 7GB/s, it can be said that the utilization of the hardware bandwidth reaches up to 96%. From this point, we conclude that our SMB server is well designed to leverage HW performance.

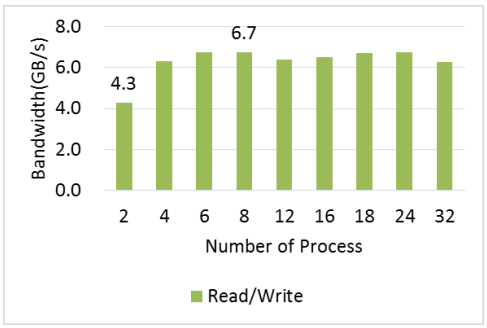

Fig. 7: Read/Write bandwidth in a SMB server

C. ShmCaffe vs. Distributed Deep Learning Frameworks

We evaluate the training performance of one standalone and three distributed deep learning frameworks (i.e., BVLC Caffe (1.0), Inspur Caffe-MPI (v1.0), MPICaffe, and our proposed ShmCaffe) over single-GPU, multi-GPU and multi-node environments. The details are as follows:

- BVLC Caffe (v1.0.0) [7]: a deep learning library which was implemented by Berkeley BVLC/BVLR. It is a standalone library, which runs over single-GPU and multi-GPU systems. If a multi-GPU setting is used, SSGD is implemented using NCCL Allreduce library.
- Caffe-MPI (v1.0): an open-source distributed deep learning platform announced by Inspur; and it implements SSGD using MPI_Send/MPI_Recv. It modifies BVLC Caffe (1.0.0-rc3) and it works as follows: master worker maintains parameter exchange threads of the number of slave workers, and each slave worker maintains a single parameter exchange thread (star-topology geometry). The master worker gathers the computed gradients by slave workers, takes the average of them, updates master weights, and finally distributes the updated master weights to slave workers.
- MPICaffe: implemented by us in order to compare its performance with the proposed ShmCaffe. In MPICaffe, BVLC Caffe (1.0) is used with MPI in a distributed manner. Instead of using the NCCL Allreduce library, which was used for multi-GPU aggregation in Caffe, the aggregation of gradients from all workers utilizes MPI Allreduce. In addition, this MPICaffe is a distributed deep learning platform that makes each worker does SSGD.

The comparison of distributed deep learning platforms was conducted in the hardware configurations in Table I. As shown in Table I, each distributed platform has its own hardware configurations for training. However, in the experiments, the performance is compared with the number of GPUs that are actually used for the gradient computation.

TABLE I: HARDWARE FOR DISTRIBUTED DEEP LEARNING PLATFORMS

Hardware Config.	Deep Learning Platforms			
	Caffe	*Caffe-MPI*	*MPICaffe*	*ShmCaffe*
GPU Server#	1	5	4	4
Total GPU#	1	8(10)/16(20)[a]	8/16	8/16
NFS Server#	1	1	1	1
Memory Server#				1

[a]10/20 GPUs used but 8/16 GPUs only used for computing gradient

As training data, ILSVRC 2012 ImageNet dataset (i.e., 1000 classes, 1,331,167 images (among the images, the numbers of training images and verification images are 1,2811,167 and 50,000)) [24]. The training data was converted to LMDB data format (256x256 color images, training data: about 240 GB and verification data: about 10 GB). The minibatch size for training of Inception_v1 model[26] is 60 per worker (i.e., 480 in 8 GPUs and 960 in 16 GPUs) and also the minibatch size of verification data is set to 64. We assume that the data feeding bottleneck is negligible because the ShmCaffe prefetches 10 sets of minibatch training data and our NFS server can provide at least 1.5GB/s of I/O bandwidth with the storage of RAID0 on 8 SSDs. In all the three different cases, base learning rate(η) is set to 0.1, γ is set to 0.1, momentum is set to 0.9, step size is set to 4 epochs, and finally the max iteration is set to 15 epochs. In addition, the *moving_rate* and *update_interval* of ShmCaffe are set to 0.2 and 1, respectively. In this experiment, the gradient aggregation in the distributed systems uses back-propagation in all layers, which means that it does not conduct gradient computations in each DNN layer.

This experiment analyzes the top 5 accuracy and loss variances during Inception_v1 models with 8/16 GPUs during 15 epochs training using four deep learning platforms, i.e., BVLC Caffe (v1.0.0), Caffe-MPI, MPICaffe, and the proposed ShmCaffe (version 1.0). This experiment aims at the computation speed rather than accuracy, thus training data augmentation is not applied. In ShmCaffe, the experiment is conducted with hybrid SGD. In Fig. 8, ShmCaffe reliably converges whereas it is a little bit lower than the Caffe. ShmCaffe shows a slightly higher performance rather than Caffe-MPI and MPICaffe when scaling to 16 GPUs.

The beauty of ShmCaffe is mainly in the training time reduction. As presented in Fig. 9 and Table II, ShmCaffe train 10.1 times faster than Caffe and 2.8 times faster than Caffe-MPI when using 16 GPUs. The computation and communication time of one iteration of training is presented in Fig. 10 where ShmCaffe Communication time is 5.3 time faster than Caffe-MPI.

TABLE II: INCEPTION_V1 TRAINING TIME(15 EPOCHS) AND SCALABILITY OF DEEP LEARNING PLATFORMS

Training Time & Scalability	GPU# & Deep Learning Platform						
	1 GPU	*8 GPUs*			*16 GPUs*		
	Caffe	*CM*[b]	*MC*[c]	*SC*[d]	*CM*	*MC*	*SC*
Time(h:m)	22:59	8:39	9:53	3:36	6:24	7:07	2:16
Scalability[a]	1.0	2.7	2.3	6.4	3.6	3.2	10.1

[a]Baseline: Caffe(1 GPU),[b]Caffe-MPI,[c]MPICaffe,[d]ShmCaffe

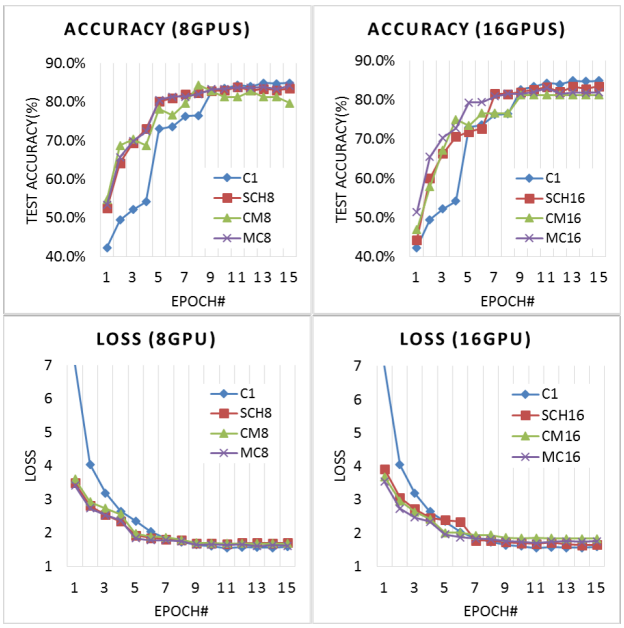

Fig. 8: Comparison of test accuracy and loss under 8 GPUs and 16 GPUs (C1: Caffe 1GPU, SCH: ShmCaffe-Hybrid, CM:Caffe-MPI, MC:MPICaffe)

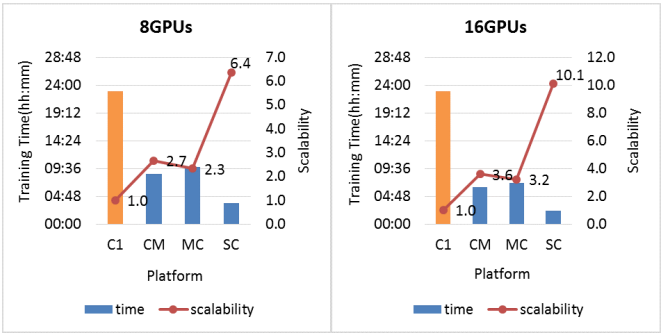

Fig. 9: Comparison of training time (Inception_v1 and 15 Epochs – C1: Caffe 1GPU, CM: Caffe-MPI, MC: MPICaffe, SC: ShmCaffe)

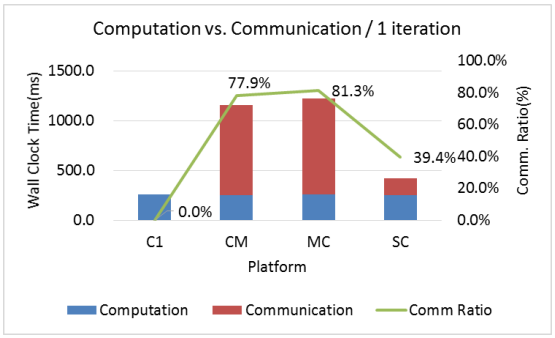

Fig. 10: Comparison of Computation and Communication time per 1 iteration (Inception_v1 and 15 Epochs – C1: Caffe 1GPU, CM: Caffe-MPI, MC: MPICaffe, SC: ShmCaffe, under 16 GPUs)

D. Asynchronous ShmCaffe vs. Hybrid ShmCaffe

In this section, the performance evaluation, with various hardware configurations, is presented in asynchronous ShmCaffe (ShmCaffe-A) and hybrid ShmCaffe (ShmCaffe-H). In Fig. 11, the performance evaluation results in terms of accuracy and loss are presented. In the case of ShmCaffe-A, which is used with SEASGD, the accuracy slowly drops when the number of GPUs increases (up to 8 GPUs), where the training convergence time is similar to the case with 1 GPU. With 16 GPUs, the accuracy achieved is 79.2% and it shows 5.7% lower performance than the case with 1 GPU. We can also observe that the ASGD shows inefficiency when the number of workers is large. Especially, it can be also observed that the performance deteriorates when the number of workers is 16 or higher [25]. Fig. 11 presents the performance evaluation results when *moving_rate* is set to 0.2 and *update_interval* is set to 1.

With ShmCaffe-H running over 4 GPUs, the performance evaluation results are presented with 2 nodes where each node has 2 GPUs. With 8 GPUs, two nodes are used and each node has 4 GPUs. With 16 GPUs, 4 nodes are used and each node has 4 GPUs. Note that SSGD is used among GPUs in the same node, and SEASGD is used among nodes. ShmCaffe-H achieves 84.0%, 82.7%, and 83.5% accuracy when the number of utilized GPUs is 4, 8, and 16, respectively. This is similar to the case of the Caffe with 1 GPU (0.9-2.2% difference). In Fig. 11, it can be observed that there are no big differences in terms of loss as well (0.04-0.11 difference).

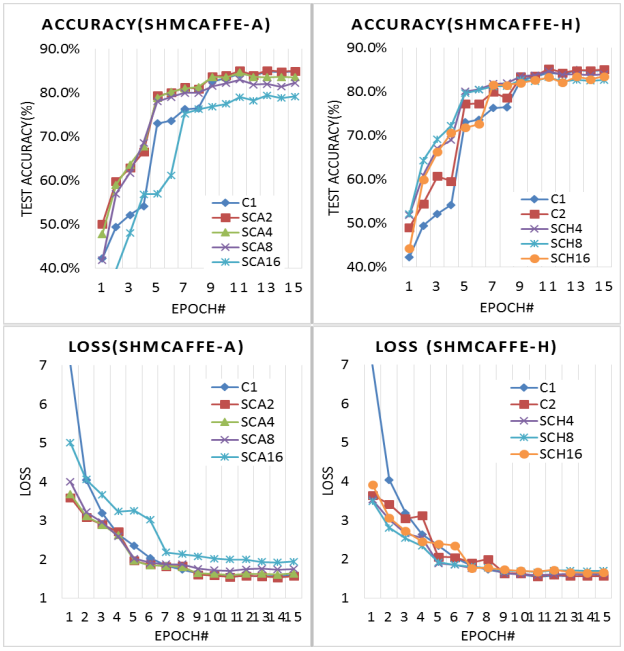

Fig. 11: Test accuracy and loss: ShmCaffe-A vs. ShmCaffe-H (Inception v1, 15 epochs)

In Fig. 12, the training time of two different ShmCaffe modes (i.e., ShmCaffe-A and ShmCaffe-H) are plotted while the number of GPUs increases (i.e., 2, 4, 8, and 16). The ShmCaffe-A shows at most 11.5 times faster performance with 16 GPUs than the Caffe with 1 GPU. Note that Caffe with 1 GPU takes around 23 hours for Inception v1 [26] model 15 epochs training. On the other hand, ShmCaffe-H shows at most 10.1 times faster performance. The reason why ShmCaffe-A can be faster than ShmCaffe-H is as follows. Since ShmCaffe-A works without synchronization among all GPUs, it can avoid collisions in training data I/O; and it can eliminate time consumption for synchronization. Therefore, ShmCaffe-A can be faster than ShmCaffe-H, which uses SSGD among GPUs in each node. ShmCaffe-A conducts distributed training using only SEASGD. We conclude that ShmCaffe-A is suitable when each node has a single GPU and many nodes are used. On the other hand, ShmCaffe-H is more suitable when each node has many GPUs.

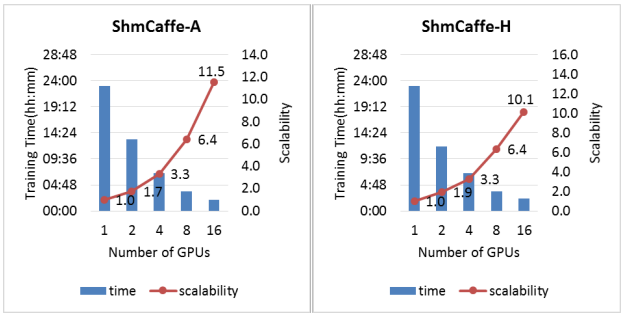

Fig. 12: Training time: ShmCaffe-A vs. ShmCaffe-H (Inception v1, 15 epochs)

E. Computation and Communication per 4 CNNs

Experiment was conducted with the configuration in Table III to analyze the computation time and communication time when training four CNN models in the Table IV with ShmCaffe-A and ShmCaffe-H. In this experiment, the training time during 1000 Iterations is measured and averaged to give computation time and communication time of one iteration. The communication time is used to capture the time that is not overlapped with the computation time. In Table III, configuration of ShmCaffe-H is represented by "S" as synchronous SGD, and "A" for asynchronous SGD (i.e., SEASGD). For example, 8 (S4×A2) means a configuration composed of 8 GPUs with 2 groups of 4 GPUs. The four GPUs in the same group update the local weight with the SSGD, and the two groups asynchronously update the global weight with the SEASGD. Table IV shows the parameter size and computation time of 4 CNN models. BVLC Caffe (v1.0.0) [7] is used to measure the parameter size and the computation time during the Forward and Backward training of the models.

Fig. 13 shows the comparison of the computation time and the communication time for the training of 4 CNN models in Table IV using ShmCaffe-A. We set *update_interval* to 1. For the Inception_v1 model with a relatively small parameter size, the communication ratio is not high; it is 16.3% when scaling to 8 GPU, and 26% when scaling to 16 GPUs. For the Resnet_50 model [27], which has about twice as many

TABLE III: HARDWARE SETUP OF SHMCAFFE-A AND SHMCAFFE-H

ShmCaffe-A			ShmCaffe-H		
GPU#	***Server#***	***GPU# /Server***	***GPU# (Config)***	***Server#***	***GPU# /Server***
1	1	1	4(S4)	1	4
2	2	1	4(S2*A2)	2	2
4	4	1	8(S2*A4)	4	2
8	4	2	8(S4*A2)	2	4
16	4	4	16(S4*A4)	4	4

TABLE IV: PARAMETER SIZE AND COMPUTATION TIME

Model Spec.	**CNN Models**			
	Inception_v1	***Resnet_50***	***Inc_res_v2***[a]	***VGG16***
Parameters# (Million)	13.4	31.8	56.1	138.4
Paramter size(MB)	51.09	121.13	214.08	527.8
Compute Time(ms)	222	210	294	190
batch size	60	18	6	64

[a]Inception_resnet_v2

parameters as Inception_v1, the communication ratio increased to 30% and 56% when scaling to 8 GPUs and 16 GPUs, respectively. If it exceeds 50%, that means the communication time becomes longer than the computation time.

In the case of Inception_resnet_v2 [28], which has a relatively large model size and is trained by using bigger images (320×320 colour images) rather than the other 3 models, the communication time increases rapidly as the number of workers increases. In particular, the time of training with 16 GPUs increases rapidly because the communication volume generated per one iteration of training reaches 6848MB (214MB × 2 × 16). The training time of SEASGD is defined as follows:

$$\begin{aligned} T_{iter} &= T_{comp} + T_{comm} \\ &= \max\left[T_{comp}, (T_{wwi} + T_{ugw})\right] + T_{rgw} + T_{ulw} \end{aligned} \tag{8}$$

Equation (8) represents the training time of one iteration as T_{iter}. T_{iter} is composed of T_{comp} and T_{comm}, where T_{comp} is the total computation time and T_{comm} is the total communication time in an iteration. T_{comp} includes the time of forward calculation, backward calculation, and updating local weight from gradients. T_{wwi} represents the time of writing weight increment (5). T_{ugw} represents the time of updating global weight (7). T_{rgw} is the time of reading the global weight. T_{ulw} represents the time of updating local weight (6). ShmCaffe (v1.0) reads the global weight when starting each iteration (T_{rgw}), updates the local weight (T_{ulw}) from the global weight, and performs the learning from the training data sequentially. Therefore, we cannot marginalize T_{rgw} and T_{ulw} in T_{comp}. T_{comp} is overlapped with T_{wwi}+T_{ugw} because the writing weight increment and updating the global weight is performed in parallel with computation. If T_{wwi}+T_{ugw} is less than T_{comp}, the time of writing weight increment and updating global weight is hidden by T_{comp}.

The communication time of one iteration when training VGG16 model[29] is 727.7ms with only 2 GPUs, the total time of an iteration, 941.8ms, is much larger than the time for the 2 iterations with 1 GPU, 389.8ms. If the computation time is relatively short and the parameter size is very large as with the VGG16 model, it is better to train with a single node than to scale to multiple nodes.

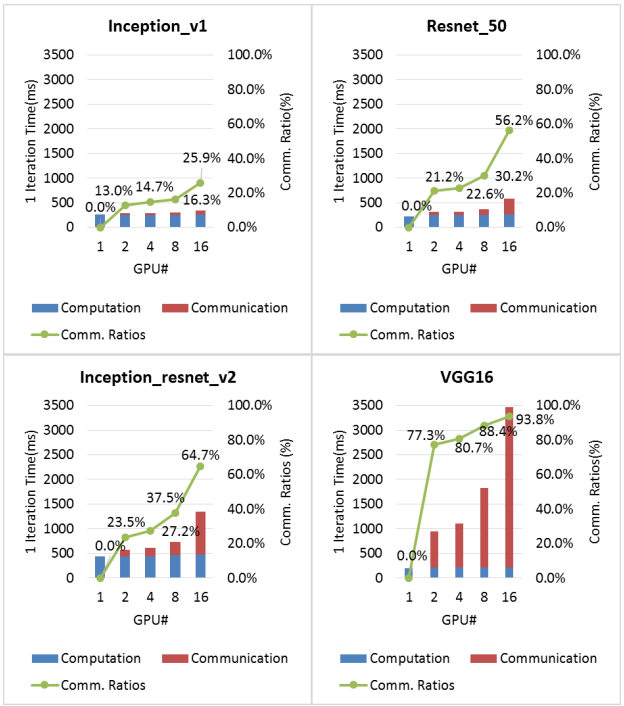

Fig. 13: Comparison of computation time and communication time through ShmCaffe-A per 4 models

TABLE V: SHMCAFFE-A: COMPUTATION AND COMMUNICATION TIME PER MODELS

Type	**CNN Models**	**Workers#**				
		1	***2***	***4***	***8***	***16***
Comp.	Incept_v1	257	249.3	250.7	253.4	256.7
	Resnet_50	225	247.4	246.5	253.8	257.3
	Inc_res_v2	443	441.3	441	458.2	474.8
	VGG16	194.9	214.1	212.5	212.4	214.3
Comm.	Incept_v1	0	37.2	43.3	49.4	89.5
	Resnet_50	0	66.4	71.9	109.7	330.8
	Inc_res_v2	0	135.7	165	275.5	871.8
	VGG16	0	727.7	888.6	1614.8	3254.3

Fig. 14, Table VI compare the computation and communication time when training four CNN models using ShmCaffe-H. 4 (S4), for example, represents the result of synchronous SGD training using 4 GPUs with BVLC Caffe for comparison. The horizontal axis of the graph indicates the mixed shape of synchronous SGD and asynchronous SGD as shown in Table III.

The communication ratio of Inception_v1, resnet_50, and inception_resnet_v2 models is generally below 30% even though there are exceptional cases. The ratio of communication in the Inception_v1 model and the Resnet_50 model, which are relatively small in model size, does not decrease in contrast with the case of ShmCaffe-A where the time for gradient aggregation for the SSGD between workers in the same

worker group is added. When training Inception_resnet_v2 model with 16 GPUs, the ratio of communication decreases from 65% to 30.7% because the volume of communication of ShmCaffe-H is reduced to 1/4 of that of ShmCaffe-A. Even though ShmCaffe uses the PCI-E system bus for communication, the communication time ratio reaches 35% when training VGG16 model with 4 GPUs on a single machine. It increases about 80% when using 16 GPUs in 4 machines. In the case of VGG16 model, multi-node expansion is not suitable, due to the high communication overhead.

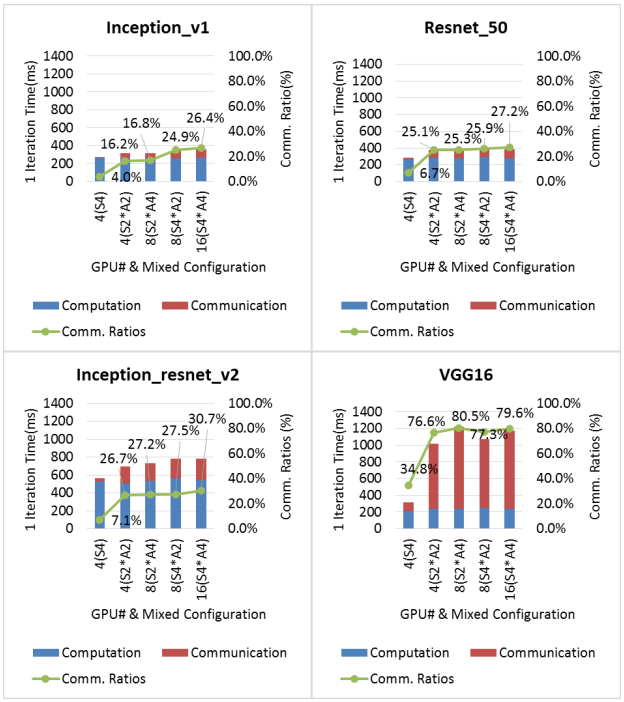

Fig. 14: Computation and communication time through ShmCaffe-H per 4 models

TABLE VI: SHMCAFFE-H: COMPUTATION AND COMMUNICATION TIME PER MODELS

Type	CNN Models	Workers#(S#, A#)[a]				
		4(4,0)	*4(2,2)*	*8(2,4)*	*8(4,2)*	*16(4,4)*
Comp.	Incept_v1	264	261.6	260.9	257	265.8
	Resnet_50	263	279.5	275.8	290.5	278.2
	Inc_res_v2	525	507.4	534.2	565.3	541.8
	VGG16	206.3	237	238.6	243	239.1
Comm.	Incept_v1	11	50.4	52.6	85.4	95.5
	Resnet_50	19	93.8	93.4	101.5	104
	Inc_res_v2	40	184.4	199.1	214.7	239.6
	VGG16	110	777.8	984.1	829.7	933.9

[a](S#,A#) mean (Synchronous workers#, Asynchronous worker groups#)

As shown in Table V and Table VI, the computation time is almost similar between ShmCaffe-A and ShmCaffe-H. In Fig. 15, ShmCaffe-A and ShmCaffe-H do not show big difference in communication time of relatively smaller models when 8 GPUs are used. ShmCaffe-H is much better than ShmCaffe-A in communication time as the DNN parameter size increases and as it scale out. As a result, ShmCaffe-H performs better

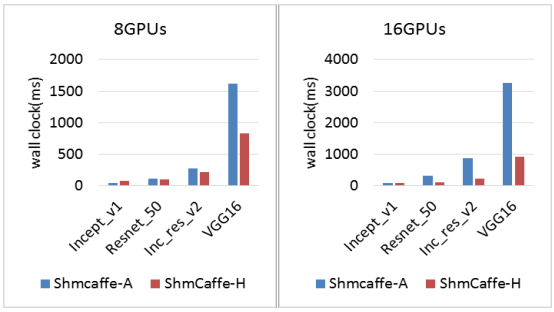

Fig. 15: Communication time per iteration: ShmCaffe-A vs. ShmCaffe-H

in one iteration time per all models when scaling out to 16 GPUs.

V. CONCLUDING REMARKS

In this paper, we proposed ShmCaffe, a new distributed deep learning platform that performs large-scale deep learning at a high speed. ShmCaffe is based on SMB, the remote shared memory framework, and implements asynchronous and shared memory based elastic averaging SGD. Instead of using the parameter server, ShmCaffe allocates remote shared memory provided by the SMB server as a shared buffer for parameter sharing. We proposed an asynchronous parameter update procedure of SEASGD through SMB, and a Hybrid parameter update method, HSGD, that combines SEASGD and SSGD.

We evaluated the performance of a single SMB server that constitutes the underlying infrastructure of the proposed ShmCaffe and compared ShmCaffe with a few Caffe-based distributed deep learning platforms such as Caffe, Caffe-MPI, and MPICaffe to show how well it can train models to convergence. ShmCaffe shows 10.1 times faster convergence than Caffe and converge 2.8 times faster than Caffe-MPI when scaling out 16 GPU workers. We verified the scalability of ShmCaffe-A and ShmCaffe-H by analyzing computation and communication time per one iteration of deep-learning on four CNN models under two operating modes. ShmCaffe removes the communication overhead from additional memory copying and protocol processing in the existing communication methods, and maximizes utilization of high-speed communication and computational resources to improve the efficiency of deep learning. As one of future research topics, we have a plan to improve the performance of the SMB framework by using multiple SMB servers.

ACKNOWLEDGMENT

This work was supported by ICT R&D program of Ministry of Science and ICT/Institute for IITP (2016-0-00087, Development of HPC System for accelerating Large-Scale Deep Learning). J. Kim is a corresponding author of this paper.

REFERENCES

[1] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the International Conference on Machine Learning (ICML'09)*, Montreal, Canada, 14-18 June 2009.

[2] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS'12)*, Lake Tahoe, NV, USA, 3-8 December 2012.

[3] W. Wang, G. Chen, H. Chen, T. T. A. Dinh, J. Gao, B. C. Ooi, K.-L. Tan, S. Wang, and M. Zhangy, "Deep learning at scale and at ease, *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 12, no. 4, pp. 69:1–69:25, November 2016.

[4] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv:1404.5997v2*, April 2014.

[5] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: building an efficient and scalable deep learning training system," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, USA, 6-8 October 2014.

[6] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proceedings of the International Conference on Machine Learning (ICML'11)*, Bellevue, WA, USA, 28 June - 2 July 2011.

[7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: convolutional architecture for fast feature embedding," in *Proceedings of ACM Multimedia (MM'14)*, Orlando, FL, USA, 3-7 November 2014.

[8] J. Wang and L. Cheng, "DistDL: a distributed deep learning service schema with GPU accelerating," in *Proceedings of the Asia-Pacific Web Conference (APWeb'15)*, Guangzhou, China, 18-20 September 2015., *Springer LNCS*, 9313:793-804.

[9] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS'12)*, Lake Tahoe, NV, USA, 3-8 December 2012.

[10] Y. Anzai, *Pattern Recognition & Machine Learning*, Elsevier, 2012.

[11] X. Lian, *et. al.*, "Asynchronous parallel stochastic gradient for nonconvex optimization," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS'15)*, Montreal, Canada, 7-12 December 2015.

[12] S. Zhang, A. Choromanska, and Y. LeCun, "Deep learning with elastic averaging SGD," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS'15)*, Montreal, Canada, 7-12 December 2015.

[13] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. 7, pp. 21212159, July 2011.

[14] S. Zheng, Q. Meng, T. Wang, W. Chen, N. Yu, Z.-M. Ma, and T.-Y. Liu, "Asynchronous stochastic gradient descent with delay compensation," in *Proceedings of the International Conference on Machine Learning (ICML'17)*, Sydney, Australia, 6-11 August 2017.

[15] F. Niu, B. Recht, C. Re, and S. J. Wrigh, "Hogwild!: a lock-free approach to parallelizing stochastic gradient descent," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS'11)*, Granada, Spain, 12-15 December 2011.

[16] C. Noel and S. Osindero, "Dogwild! – Distributed Hogwild for CPU & GPU," in *Proceedings of the Advances in Neural Information Processing Systems (NIPS) Workshop on Distributed Machine Learning and Matrix Computations*, Montreal, Canada, 12 December 2014.

[17] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, USA, 6-8 October 2014.

[18] E. P. Xing, *et. al.*, "A new look at the system, algorithm and theory foundations of large-scale distributed machine learning," Tutorial in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'15)*, Sydney, Australia, 10-13 August 2015.

[19] D. Yu, *et. al.*, "An introduction to computational networks and the computational network toolkit," *Microsoft Research Technical Report MSR-TR-2014-112*, 1 October 2014.

[20] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu, "On parallelizability of stochastic gradient descent for speech DNNs," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'14)*, Florence, Italy, 4-9 May 2014.

[21] L. Deng, D. Yu, and J. Platt, "Scalable stacking and learning for building deep architectures," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'12)*, Kyoto, Japan, 25-30 March 2012.

[22] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*, Monterey, CA, USA, 22-24 February 2015.

[23] S. Ahn and J. Kim and S. Kang,"Poster: A Novel Shared Memory Framework for Distributed Deep Learning in High-Performance Computing Architecture", in Proceedings of the 40th IEEE/ACM International Conference on Software Engineerring (ICSE'18), Gothenburg, Sweden, May 27 - June 3, 2018

[24] ImageNet Large Scale Visual Recognition Challenge (ILSVRC), http://www.image-net.org/challenges/LSVRC/

[25] Onkar Bhardwaj, Guojing Cong, Practical Efficiency of Asynchronous Stochastic Gradient Descent, 2016 2nd Workshop on Machine Learning in HPC Environments

[26] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*, Boston, MA, USA, 7-12 June 2015.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, Las Vegas, NV, USA, 26 June - 1 July 2016.

[28] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the impact of residual connections on learning," in *Proceedings of AAAI Conference on Artificial Intelligence (AAAI'17)*, San Francisco, CA, USA, 4-9 February 2017.

[29] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the International Conference on Machine Learning (ICML'15)*, Lille, France, 6-11 July 2015.