IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 10, 2018, accepted April 23, 2018, date of publication May 8, 2018, date of current version June 5, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2834146

Soft Memory Box: A Virtual Shared Memory Framework for Fast Deep Neural Network Training in Distributed High Performance Computing

SHINYOUNG AHN[1,2], JOONGHEON KIM[3], (Senior Member, IEEE), EUNJI LIM[2], AND SUNGWON KANG[4], (Member, IEEE)

[1]Department of Information and Communication Engineering, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea
[2]High Performance Computing Research Group, Electronics and Telecommunications Research Institute, Daejeon 34129, South Korea
[3]School of Software, Chung-Ang University, Seoul 06974, South Korea
[4]School of Computing, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea

Corresponding author: Joongheon Kim (joongheon@cau.ac.kr)

This work was supported by the Institute for Information and Communications Technology Promotion through the Korea Government (MSIT) (Development of HPC System for Accelerating Large-Scale Deep Learning) under Grant 2016-0-00087.

ABSTRACT Deep learning is one of the major promising machine learning methodologies. Deep learning is widely used in various application domains, e.g., image recognition, voice recognition, and natural language processing. In order to improve learning accuracy, deep neural networks have evolved by: 1) increasing the number of layers and 2) increasing the number of parameters in massive models. This implies that distributed deep learning platforms need to evolve to: 1) deal with huge/complex deep neural networks and 2) process with high-performance computing resources for massive training data. This paper proposes a new virtual shared memory framework, called Soft Memory Box (SMB), which enables sharing the memory of remote node among distributed processes in the nodes so as to improve communication performance via parameter sharing. According to data-intensive performance evaluation results, the communication time of deep learning using the proposed SMB is 2.1 times faster than that using the massage passing interface (MPI). In addition, the communication time of the SMB-based asynchronous parameter update becomes 2–7 times faster than that using the MPI depending on deep learning models and the number of deep learning workers.

INDEX TERMS High performance computing, distributed computing, soft memory box, shared memory, deep neural network, distributed deep learning.

I. INTRODUCTION

In modern artificial intelligence research, deep learning shows the dramatic performance in various applications such as image recognition [1]–[4], voice recognition [5], [6], text mining [7], and security [8]. Although higher accuracy models can be obtained with bigger deep neural networks and more learning data. The required amount of computation increases proportionally depending on the multiplication of the size of the deep neural networks and the amounts of the training data [9], [10]. Due to the increased amount of deep learning computation, it takes a long time to train the deep neural networks with a machine; and thus distributed and high-performance computing architectures and resources are required to develop timely deep learning models with good performance. Based on these requirements, distributed deep learning platforms have been investigated [11], [12].

In distributed deep learning platforms, *workers (or trainers)* conduct distributed training of deep neural networks. In addition, they have to share massive deep learning parameters between them and eventually the sharing of massive parameters introduces enormous communication overhead, which constitutes a considerable portion of the total time of distributed deep learning. As a consequence, the idle time of the processing units such as CPU/GPU becomes longer, and this eventually incurs the degradation of computation resource utilization. The high communication overhead is introduced

not only due to the massive and dominant parameter communication but also due to network protocol (e.g., TCP/IP) computation. During the computation, memory copy operations are repeated and the corresponding overheads are also incurred while dealing with multi-layer network protocols.

As the size of deep neural network becomes massive, the number of parameters dramatically increases. Therefore, high speed interconnect networking is desired for fast and massive distributed processing [12]. Based on the requirements, the networking techniques aiming at the low latency and high bandwidth have been proposed including Infiniband, Quadrics, 40/100 Gbps Ethernet, and Myrinet. Among them, the Infiniband Enhanced Data Rate (EDR) provides extremely low latency (approximately sub-micro-seconds) and high data rates (100 Gbps). Moreover, Infiniband supports *Remote Direct Memory Access (RDMA)*. With RDMA, CPU does not need to control the transmission between local memory and remote memory, and thus, RDMA can reduce the number of memory copies between user space and kernel space. Eventually, RDMA not only reduces the access latency but also dramatically improves the communication bandwidth.

Massive distributed processing of deep learning requires high speed networking techniques as mentioned before; and also need parallel programming models that enable distributed parallel processing. Among the models, one of the most well-known programming models is massage passing interface (MPI).

In this paper, we propose a method that reduces communication overhead of massive distributed deep learning efficiently. For this purpose, our method allows distributed workers to share deep learning parameters between them at high speed. This sharing mechanism can be realized by designing a new memory sharing framework as well as allowing all the workers to access the shared memory. The proposed shared memory framework for distributed deep learning can be operated in the high-performance clusters that are connected via high-speed networking techniques (e.g., Infiniband). In addition, the proposed architecture utilizes RDMA technique and thus eliminates copy operations of communication data between application-level buffers and kernel-level memory buffers. Eventually, the proposed framework provides a method which can share deep learning parameters among distributed workers using RDMA for reading/writing data in the memory of remote nodes.

The major contributions of our research in this paper are as follows: The *Soft Memory Box (SMB)* is proposed in this paper which consists of the SMB Server, the SMB Device Driver, the Infiniband Communication Layer module, and the SMB Library. The components are used for fast sharing the memory of remote dedicated node among multiple workers. In addition, SMB provides application programming interface (API) functions which are required for allocation, use, and release of shared memory buffers among distributed processes. Moreover, SMB provides cumulative computation functions among the shared memory buffers, and then eventually improves the performance of deep learning parameter sharing. In this paper, we implemented two emulation programs which emulate the asynchronous deep learning parameters sharing with MPI and SMB, and then we show benefits of the proposed SMB framework by evaluating the two programs in the setting of high performance computing servers.

The rest of this paper consists of following sections. Section II presents the literature survey on distributed deep learning. Section III introduces the details of the virtual shared memory framework, i.e., architecture, functions, components, shared memory allocation methods, and API. Section IV presents the performance evaluation results of our proposed virtual share memory framework. Finally, section V concludes this paper and presents future research directions.

II. RELATED WORK

We can achieve deep learning model of high accuracy if we train the bigger deep neural network (DNN), which has much more layers and parameters, with the larger training data. However, the amount of computation required for the higher accurate deep learning model increases in proportion to the product of the deep neural network size and the learning data capacity [9], [10]. For this reason, very large-scale DNN training might be impossible to be handled in a single computing node, and there are a lot of on-going research projects on various distributed processing platforms for DNN training [13].

The distributed deep learning platforms use two approaches to train DNNs in the distributed and parallel processing. One is data parallelism in which plurality of workers train parts of all the training data, and the other is a model parallelism in which workers only train parts of the entire model with the same training data [9], [14]–[16].

For distributed DNN training, it is necessary to share parameters (weights or gradients) newly explored by each worker among distributed workers in each iteration of training. The parameter sharing (or updating) approaches can be classified into synchronous and asynchronous approaches. In the synchronous approach, gradient parameters are transferred to the parameter server (or the master worker) per each training iteration by the deep learning workers, or are exchanged directly among them to integrate gradient parameters trained by each worker. The asynchronous approach is a method in which the parameter server updates global weight parameters without aggregating gradient parameters arriving late or early from distributed workers. The synchronous approach has a very large overhead because the time drift of one iteration (forward and backward) of computation among DNN training workers is considerable. The synchronous approach suffers from stragglers' problem where all the workers which finished their own computation must wait for until the slowest worker finish its computation [13], [17]. However, the asynchronous approach has the advantage that DNN can be trained faster instead of sacrificing the accuracy a little as compared with the synchronous

approach since the latter can eliminate the synchronization overhead [18].

Among the distributed deep learning optimization schemes based on the stochastic gradient descent (SGD), the Hogwild showed that deep learning processes with same weight and different data shards can train DNN through asynchronous SGDs in a shared memory architecture without locking [19]. In addition, the Dogwild framework, an extension of the Hogwild, performs DNN training by exchanging weights and gradients asynchronously between the master and slave processes. The master process updates the global weights whenever it receives gradients from the slave processes, and shares the updated weights all the slave processes at once. Each slave process updates the most recently received weights with the gradients which are explored by itself, and sends the gradients to the master [20]. The DistBelief framework proposed Downpour SGD, which supports a large number of model replicas, as a technique of asynchonous SGD. It also proposed the Sandblaster batch optimization method, which is a framework supporting various distributed models [14]. Each DNN training worker in an asynchronous SGD can learn weights at different speeds without synchronization overhead to maximize the utilization of computing and network resources. In particular, the resources of a heterogeneous HPC system consisting of CPUs and GPUs of heterogeneous specifications (clock rate, the number of cores, etc) can be utilized effectively to take advantage of the maximum computational performance.

In order to perform the asynchronous SGD as described above, the parameter server for updating and distributing parameters is indispensably required. For example, as a method of implementing the parameter server, the deep learning worker which acts as the master allocates a memory region for storing global parameters in its local memory, updates global weight parameters with gradient parameters received from the slave workers in the form of communication messages, and distributes the updated global weight parameters to the slave workers. Distributed deep learning platforms such as Petuum and CNTK also use distributed key-value repositories developed specifically for parameter servers [18], [21]. The parameter server manages asynchronous parameter updates among deep learning workers and provides the advantage of supporting an elastic coherence model, flexible scalability and fault tolerance [18]. In this paper, we show that our virtual shared memory framework can accelerate the parameter sharing in distributed deep learning platform that supports asynchronous SGD through experiments.

III. SOFT MEMORY BOX: VIRTUAL SHARED MEMORY FRAMEWORK

A. HARDWARE REQUIREMENTS

Distributed parallel deep learning aims at training massive data and large-scale DNN models in high performance computing nodes. As the size of DNN and the dimension

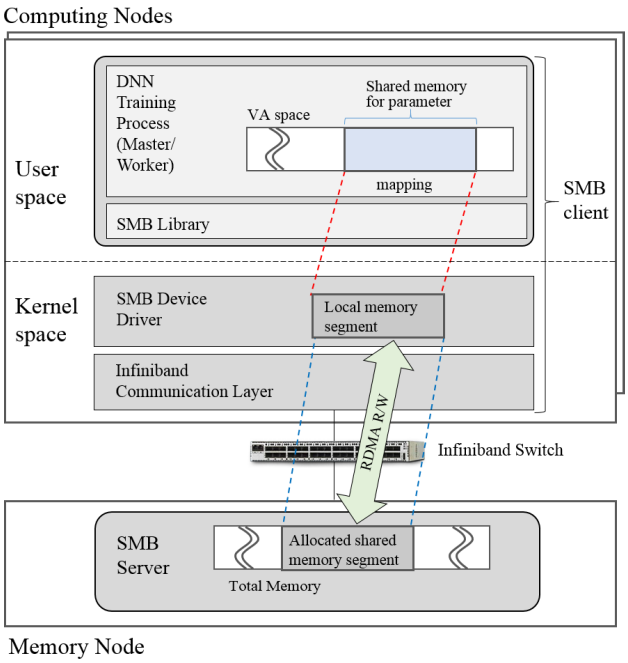

FIGURE 1. Soft Memory Box (SMB) architecture and remote shared memory mapping mechanism.

(i.e. resolution in image data) of training data increases, the required communication bandwidth increases rapidly. Therefore, low-speed networks such as 1Gbps Ethernet cannot provide enough bandwidth to deal with the rapidly increasing inter-node deep learning parameter traffic. Hence, most companies and researchers are utilizing high-speed interconnection networks (e.g.,Infiniband, 40/100Gbps Ethernet, and Myrinet) used in supercomputers or high-performance computing systems. The soft memory box (SMB) proposed in this paper requires the HPC system which consists of computing nodes with multiple GPUs, a separate memory node that have enough physical memory to provide remote shared memory, and an Infiniband switch that connects these nodes.

B. SMB ARCHITECTURE

The SMB consists of an SMB Server and SMB client components: SMB Library, SMB Device Driver, and Infiniband Communication Layer module as shown in Fig. 1. The SMB Server is located in the memory node and provides distributed processes with shared memory. SMB client components are located in the computing node. SMB Library, which is a user-level static library, provides application programming interfaces for distributed application processes and statically linked with the processes during compile time, SMB Device Driver and Infiniband Communication Layer (ICL) module are kernel-level modules and should be loaded before the execution of application processes The SMB Server can provide the physical memory of the memory node by pooling it in advance, as well as the SMB Server can provide it on-demand for client processes [23]. In order to enable

TABLE 1. Soft memory box APIs.

Type	API
Initialization	int smb_init(void); void smb_terminate(void);

Allocation/ Deallocation	void *shm_alloc(u64 *key,u64 size, int shmflg);
	void shm_free(void *ptr);
	unsigned long shm_get_key(void *ptr);

Read/Write	int shm_rsync(void *ptr, size_t size); int shm_wsync(void *ptr, size_t size);
Event	int shm_accum(void *dst,void *src, size_t size, unsigned int type);

RDMA operation from the deep learning processes in the computing nodes,supplied memory pages for client processes should be registered in the host channel adapter (HCA) driver. The registration process enroll the virtual-to-physical address mapping information of the provided pages. Since this registration time is relatively large, pooling memory methods are generally preferred.

The SMB Device Driver is a kernel-level device driver and maps the local physical memory pages, which are served as the cache for the remote shared memory buffer, to the virtual address of the DNN training processes as illustrated in Fig. 1. The SMB Device driver communicates with the SMB Server through the Infiniband Communication Layer module, which provides the kernel-level interface for Infiniband communication with the remote SMB Server by wrapping kernel-level Infiniband verbs.

C. SMB LIBRARY AND API

The API functions of the SMB framework are classified four groups such as initialization, allocation/deallocation, read/write, and event as shown in Table 1. First of all, the DNN training processes should initialize the SMB framework in order to use shared memory. During the initialization (`smb_init()`), the DNN training processes open SMB Device through the standard character device interface of Linux OS and register themselves in the SMB Device as a user. When the processes want to finish using shared memory, they have to call the termination API (`smb_terminate()`) to cancel user registration and close the file descriptor of the SMB Device. SMB provides `shm_alloc()` and `shm_free()` API functions for the shared memory allocation and deallocation. When an application process creates a new shared memory in the SMB Server, the application process can provide *SHMKey* as the argument of `shm_alloc()`. If *SHMKey* is not supplied by the application process, it is assigned by the SMB Server. To use the shared memory, all the application process need to know the *SHMKey*, so SMB Library includes an API (`shm_get_key()`) that retrieves the shared memory creation key (*SHMKey*) assigned by the SMB Server. Reading the parameter from shared memory and writing the parameter to shared memory are performed through Read/Write APIs (`shm_rsync()`, `shm_wsync()`). The event group has an API that requests cumulative (addition) operation between two same size of shared memory buffers in the SMB Server (`shm_accum()`). This accumulation operation differs in behavior depending on the type of data, and SMB supports cumulative operations for two data types: `FLOAT` and `DOUBLE`.

D. SMB DEVICE DRIVER

The SMB Device is a novel virtual device which serves shared memory buffer to the application processes through system call interface. 'Virtual' means that SMB Device does not have a physical shape. The ICL module and the SMB Server virtually act as a shared memory device on behalf of the physical device. We design it as a character device("**/dev/smbd**"). Linux kernel provides standard interfaces to the character device via virtual file system. SMB Device Driver registers file operations (i.e., `open()`, `close()`, `mmap()`, `ioctl()` and etc) for the standard interfaces during the module initialization. Fig. 2 shows the internals and interrelationships between SMB Device Driver and other components. The major functional components of SMB Device are as follows:

- **SHM Allocation/Deallocation**: It allocates and deallocates of the remote shared memory buffer by exchanging control messages with the SMB Server. If user process call `shm_alloc()` API, SMB Device Driver sends *shm_alloc_request* message to the SMB Server. After receiving *shm_alloc_response*, it creates a new SHM allocation record and allocates physical memory pages for the shared memory region. SMB Device Driver maps the physical pages to the virtual address of the user process during page fault handling. The page faults of all the virtual address space of the shared memory region are handled just once because the SMB Device Driver assigns the physical pages to the shared memory region statically. SHM deallocation is performed in the reverse order of SHM allocation. First, it clears the mappings in the page table, and then retrun all the physical pages to the kernel, and it exchanges *shm_dealloc_request* and *shm_dealloc_response* messages with the SMB Server. Finally it deletes the SHM allocation record.
- **SHM Read/Write**: It synchronizes the local physical pages with the remote shared memory buffer by reading and writing data between them using the Infiniband RDMA mechanism, which is served by ICL service functions.
- **SHM Accumulation**: It sends shared memory accumulation request (*shm_accum_request*) message to the SMB Server and waits for shared memory accumulation response (*shm_accum_response*) message from the SMB Server when the SMB Server finishes accumulation, and then returns the result to the user process.
- **Message Handler**: It allocates a buffer containing the send/receive message and is responsible for setting the message header. It is responsible for exchanging the request/response messages between the SMB Device

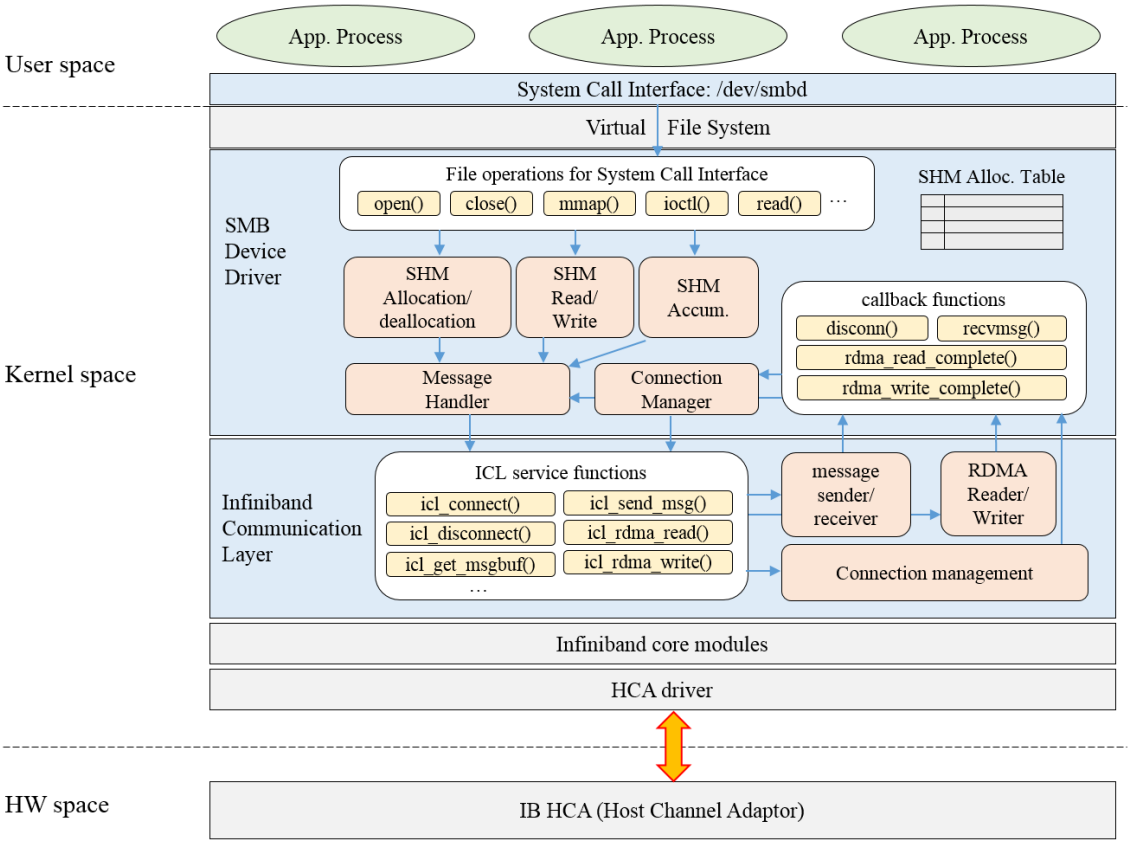

FIGURE 2. The Internals and relationship of the SMB Device Driver and Infiniband Communication layer.

and the SMB Server. For message transmission, it calls a kernel verb of the Infiniband core driver to send the message and waits for the notification of the completion of the message sending from Infiniband core driver. We introduce *message_sequenc_id* for confirmation of correct message exchange.

- **Connection Manager**: it creates Infiniband reliable connections (RC) with the SMB Server to support RDMA communications. It also handles the disconnection event from the ICL module by reconnecting automatically or terminating all the SHM allocations.

To support allocation of remote shared memory buffer in the SMB Device Driver, we developed the features in the mmap() function, one of file operations of device driver. We also specify a page fault handling function for each shared memory allocation (by setting `smb_vm_fault()` as `vm_area_struct->vm_ops->fault()`) in the `mmap()` function. Linux kernel calls the specified page fault handler, `smb_vm_fault()`, when page fault occur in the shared memory region. The page fault handler function provides Linux kernel with the physical pages assigned during the SHM allocation, and then return control to the Linux kernel. The Linux kernel maps the physical pages to the virtual address of the user process by calling kernel API to update page tables of the application process.

E. INFINIBAND COMMUNICATION LAYER MODULE

The Infiniband Communication Layer (ICL) module provides several service functions to the SMB Device Driver as shown in the Fig. 2. The `icl_connect()` function creates connection to SMB Server and `icl_disconnect()` function terminates the established connection to the SMB Server. These functions return to caller only after establishing connection or terminating connection. The `icl_send_msg()` function sends message and immediately returns to the caller. The `icl_rdma_read()` function reads data from remote memory to local memory, and `icl_rdma_write()` function writes local data from local memory to remote memory. The functions work in a non-blocking style where the functions return to caller immediately after the read/write work request to be sent to Infiniband core driver and HCA. The SMB Device Driver calling this functions, must wait until the requested work is completed on the wait queue. This style is typical in the Infiniband programs that handle long-running tasks.

The ICL module invokes callback functions to inform the SMB Device Driver of the receipt of the message and the termination of RDMA read/write operations. A set of callback functions are provided to the ICL module as argument of the `icl_connect()` function. The callback functions are defined in the SMB Device Driver as follows:

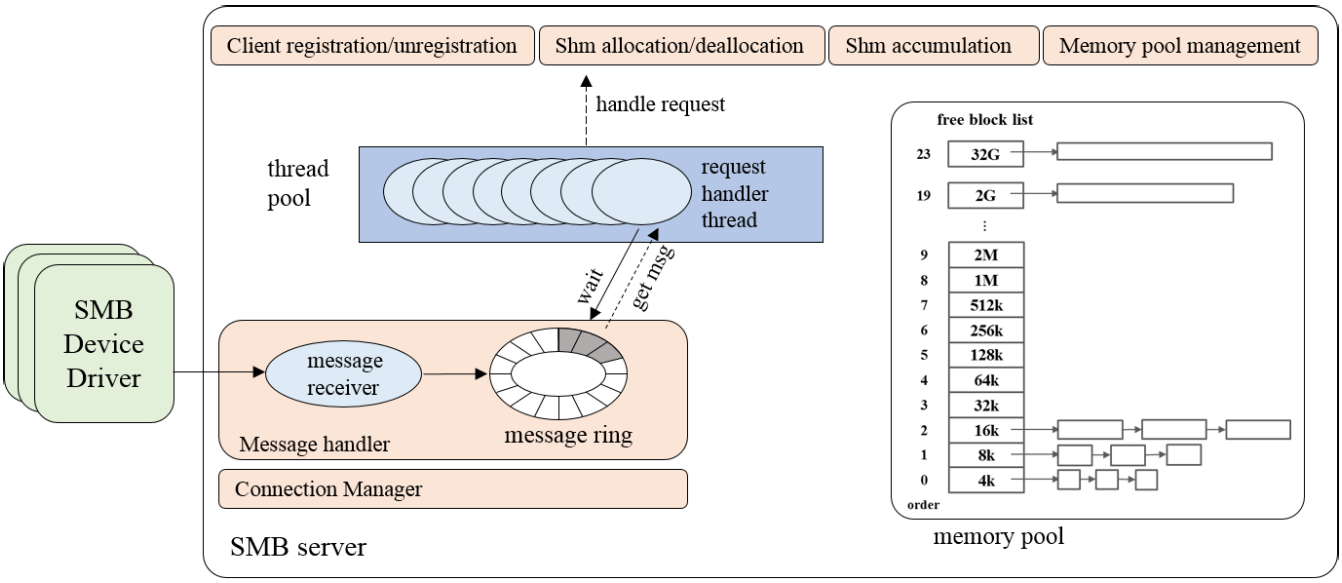

FIGURE 3. SMB Server architecture and the mechanism of memory pool management.

- `disconn`: this function is called by the ICL module when a connection to the SMB Server is disconnected due to unexpected reasons. `disconn` callback function handles the disconnection event by reconnecting connection to the SMB Server or releasing all the shared memory allocation.
- `recvmsg`: this function is called by the ICL when a new message arrives through the connection from the SMB Server. The `recvmsg` callback function decodes message header and wakes up the threads waiting the message or invokes appropriate handling functions according to the header information.
- `rdma_read_complete`: this function is called by the ICL module to inform the SMB Device Driver of the completion of RDMA read request. The `rdma_read_complete` callback function wakes up the thread waiting the completion of RDMA read request.
- `rdma_write_complete`: this function is also called by the ICL module to inform the SMB Device Driver of the completion of RDMA write request. The `rdma_write_complete` callback function wakes up the thread waiting the completion of RDMA write request.

F. SMB SERVER

Fig. 3 shows the architecture of the SMB Server and its mechanism for managing memory pool to provide the shared memory buffer. The SMB Server is implemented as a Linux user level process. The SMB Server has the multi-threaded architecture and the SMB Server spawn threads to handle the requests from multiple SMB Device Drivers simultaneously. The request messages sent from the SMB Device Drivers are temporarily stored in the message ring by message receiver thread. After storing messages into the ring, the message receiver thread wakes up all the request handler threads in the thread pool. One of them takes out one request message from the message ring in a circular queue style, processes the message, and returns to the thread pool when the message processing is finished. The SMB Server manages the provided memory as pool using the buddy memory allocation mechanism, which is usually used for physical page allocation in the Linux kernel. The memory block size can be configured with the minimum of 4KB and the maximum of 32GB. The maximum size is also configurable. The communication layer is implemented as `libibverbs` and `librdma`. During the shared memory allocation phase, the SMB Server provides the remote access keys required for the SMB Device Driver to directly access the shared memory buffer. The SMB Server is not involved at all when the SMB Device Driver accesses the shared memory which it provides. Major functional components of the SMB Server are as follows:

- **Connection manager**: It creates or releases a connection when it receives a connection/disconnection request for the specified port, and handles exception when the connection is terminated abnormally.
- **Message handler**: It receives messages from the SMB Device Drivers of remote computing node, stores them in a message ring, and wake up the request handler threads that are responsible for message processing.
- **Client registration/unregistration**: The SMB Server processes the requests for registration and unregistration of the SMB Device Drivers.
- **Allocation and deallocation of shared memory**: The SMB Server allocates shared memory buffer from the memory pool to the SMB Device Driver and returns shared memory buffer to the memory pool when the SMB Device Driver releases the shared memory.

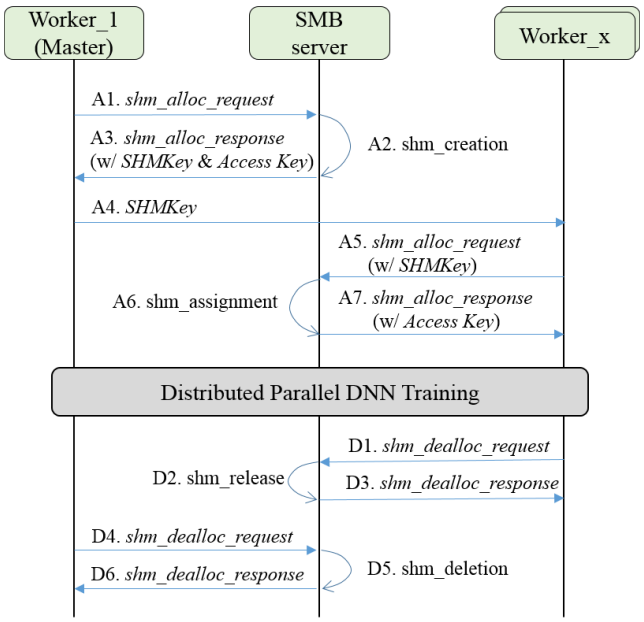

FIGURE 4. Shared Memory Allocation/Deallocation Scenarios.

- **Accumulation between shared memory**: The SMB Server adds the vector of source buffer to the vector of destination buffer. The vector addition accumulates data values (`FLOAT` or `DOUBLE`) of parameter stored in the source and destination buffer.
- **Memory pool management**: The SMB Server allocates physical memory pages to be supplied to the SMB client and registers them as Memory Region (MR) using `ibv_reg_mr()` Infiniband verb. Then it constructs memory pool and it maintains and keeps tracks of free and allocated memory page blocks by using the buddy algorithm.

G. SHARED MEMORY ALLOCATION/DEALLOCATION PROCEDURE

Fig. 4 illustrates the procedure of allocating and deallocating shared memory among distributed deep learning workers using the SMB. First, the deep learning worker acting as the master creates a shared memory buffer on the SMB Server using the SMB API. After creating the shared memory, the master worker sends *SHMKey* for the shared memory to the slave workers. The master worker should have a separate communication channel to other slave workers who want to be assigned the shared memory buffer created by the master worker. The worker who receives this *SHMKey* from the master worker sends a shared memory assignment request (using the *shm_alloc_request* message) with the *SHMKey*. Then the SMB Server provides the *Access Key* (RDMA-enabled Infiniband remote key) for the shared memory buffer if the worker requests the same size of shared memory buffer assignment and sends the same *SHMKey*. When the allocation procedure is completed, the shared memory buffer of the SMB Server can be shared between the distributed deep-learning workers. In the end of distributed DNN training, all the workers perform the deallocation of shared memory regardless of order. The slave workers request the release of the allocated shared memory buffer to the SMB Server, and the master worker requests the deletion of the shared memory buffer. Fig. 4 shows the details of allocation procedures (A1 through A7) and deallocation procedures (D1 through D6) for virtual shared memory among the distributed deep learning workers as described above.

IV. EXPERIMENTAL STUDY

A. HARDWARE AND SOFTWARE ENVIRONMENTS

We used 6 SuperMicro 4028GR-TRT2 servers, one HP DL380p server, and one Mellanox Infiniband switch with next specifications. SuperMicro server has 2 socket Intel Xeon CPU (E5-2690 v4, 14cores, 2.3GHz), 128GB DDR4-2400MHz/ECC memory, 4 Nvidia GPUs. The types of GPUs consist of NVIDIA Pascal archtiecture GPUs such as Titan X Pascal (3584 cores/12GB memory) and Titan Xp (3840 cores and 12 GB memory). HP server has 2 socket Intel Xeon CPU (E5-2609 v2, 4cores, 2.5GHz) and 256GB DDR3-1866MHz memory. Each server has one FDR Infiniband HCA (56Gbps) and is connected to the Infiniband switch. OS is Ubuntu 14.04-LTS and the kernel version is 3.13.0-123-generic. We used CUDA 8.0 (including cudnn 8.0).

B. I/O PERFORMANCE OF A SMB SERVER

First of all, we evaluated the I/O performance of the SMB Server. The exchange of deep learning parameters more depends on bandwidth rather than latency because parameter exchange is mass transfer. We measured the shared memory read, write, and read/write (mixed) bandwidth while the number of processes, which access the shared memory of SMB Server concurrently, increases from 2 to 32 in 2-fold increments. The size of the shared memory allocated to each process is 1 GB. Each process performs read, write, and read/write (mix each 50%) simultaneously after the shared memory is allocated. Fig. 5 shows that the read and write bandwidth increase to 5.5 GB/s and 5.2 GB/s, respectively, when the numbers of processes are 8 and 32. The reason why the read performance reaches the highest value in the number of 8 processes might be the contentions between distributed processes. Up to 8 processes, the processes generate more read requests and increase the read bandwidth, however, too much processes over than 8 may generate too much read requests which result in excessive competition between them. This may decrease the total bandwidth. The read operation requires the HCA of the SMB Server to send data to HCAs of the distributed nodes, but the write operation has a different pattern because the HCAs of the distributed nodes only have to send the data to the HCA of the memory node. Therefore, as the number of processes increases, the write performance is expected to increase gradually. The bandwidth for mixed read/write increase up to 6.7GB/s. The result shows the

TABLE 2. Parameter size and computation time of deep learning models.

Network Specification	Convolutional Neural Networks					
	Inception_v1	Inception_v3	Resnet_50	Inception_resnet_v2	Resnet_152	VGG16
Parameters (Million)	13.4	24.7	31.8	56.1	66.5	138.4
Parameter Size (MB)[a]	51.09	94.30	121.13	214.08	253.63	527.79
Compute Time (ms)	189	445	233	294	218	190
Batch Size	50	32	18	6	8	64

[a]Parameter data type: FLOAT

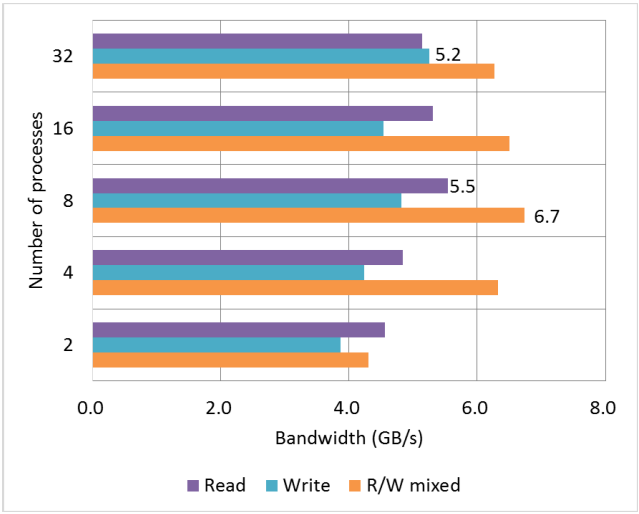

FIGURE 5. I/O Performance of a SMB Server.

physical bandwidth utilization of the SMB framework reaches 96% because the maximum bandwidth of Infiniband FDR is 7GB/s.

C. EXPERIMENTAL SETUP FOR ASYNCHRONOUS SGD EMULATION

In order to emulate the behavior of distributed DNN training based on MPI and SMB, first of all we need to know how much each DNN model takes time for computation and how much data need to be exchanged for parameter sharing. We measured the computation time of training actual DNN models in a single GPU and the parameter's data size. For this purpose, BVLC Caffe (v1.0.0) [22] is used to measure the parameter size and the computation time during Forward and Backward training of the six convolutional neural network (CNN) models for image classification [24]–[28].

Table 2 shows the measured parameter size and computation time of six CNN models. If the batch size changes, the computation time changes because the batch size means the number of images to be trained. For this experiment, the batch size was selected as the maximum value where each model can be trained with the NVIDIA Titan X(12GB memory) GPU except Inception_v1. If the batch size increase, the Caffe requires more memory allocation, therefore there is maximum batch size where Caffe can train each model in an iteration. The selected CNN models have various distributions in terms of parameter size and computation time. There is no proportionality between computation time and parameter size. The Inception_v1 [24], Inception_v3 [25] and Resnet_50 models [26] has smaller parameters than the other 3 networks:Inception_resent_v2 [27], Resnet_152 [26] and VGG16 [28] models. However, Inception_v3 [25] has longest computation time (445ms) but has small parameters relatively. The VGG16 [28] with 138.4 millions of parameters has biggest parameter size but the computation time is smaller than the other 4 models.

D. ASYNCHRONOUS SGD EMULATION PROGRAMS USING MPI/SMB

In order to demonstrate the usefulness of SMB for deep learning, we first developed two programs to compare the SMB with MPI as a way of exchanging deep learning parameters asynchronously between parameter server and distributed deep learning workers. The emulation programs which imitate distributed deep learning using the asynchronous SGD consist of one parameter server and several DNN training workers. The architecture of the program constitutes a star topology centered around the parameter server. Both programs use MPI for distributing multiple deep learning workers in multiple nodes. One program uses the MPI for parameter communication, and the other program uses the SMB for parameter communication. The MPI-based program transmits and receives parameters using MPI_Send and MPI_Recv. The first worker process (MPI_rank=0) acts as a parameter server to update global parameters trained by other worker processes (MPI_rank≠0) asynchronously. The SMB-based emulation program assumes that DNN training workers share parameters using only one SMB Server which acts as a parameter server. It exchanges parameters with the SMB API (`shm_rsync()` and `shm_wsync()`). The emulation programs has two test scenarios to mix computation and communication:sequential scenario and parallel scenario. In the sequential scenario, the deep learning computation and parameter communication occur in sequence. On the other hand, the deep learning computation and parameter communication in the parallel scenario occur simultaneously. For the parallel scenario, the DNN training workers of these two emulation programs create two threads to parallelize computation and communication at the same time. We use `mutex` and `condition` variables to control the race between the threads.

E. COMMUNICATION TIME ANALYSIS: SEQUENTIAL SCENARIO

We executed the first sequential scenario of the asynchronous SGD emulation programs. we executed the program in parallel on six servers. We also emulated DNN training for six

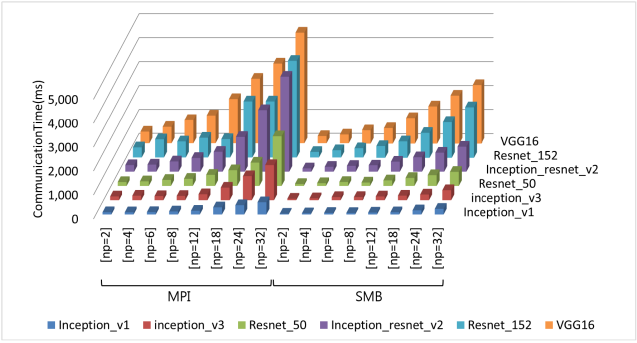

FIGURE 6. Comparison of communication time of MPI-based and SMB-based deep learning.

models, i.e., Inception_v1, Inception_v3, Resnet_50, Inception_resnet_v2, Resnet_152, and VGG16, by changing the number of deep learning worker processes to 2, 4, 6, 8, 12, 18, 24, and 32. The tests were performed 10 times without any other workload in the test servers, and the measurements were averaged after the tests. The Fig. 6 and 7 compare the communication time of MPI and SMB in the sequential scenario. The sequential scenario do not consider computation/communication overlapping during each iteration of DNN training. As shown in Fig. 6, as the model size grows and the number of workers increase, the communication time of the MPI method increases rapidly, but the communication time of the SMB gradually increases. Fig. 6 shows overall trends and Fig. 7 shows comparison of the MPI and SMB methods by changing the number of deep learning workers per 6 deep learning models. It can be seen that the SMB method is 1.1 through 4.2 times faster (2.1 times on average) than the MPI method in all models.

F. TRAINING TIME ANALYSIS: PARALLEL SCENARIO

We executed the second parallel scenario of the asynchronous SGD emulation programs with same condition of the sequential scenario.

Equation (1) represents the training time of one iteration as T_{iter}. T_{iter} is the maximum value between T_{comp} and T_{comm}, where T_{comp} is the computation time and T_{comm} is the communication time in an iteration. In many distributed deep learning platforms, they try to overlap computation and communication because communication time is great. The parallel scenario emulates the deep learning training time when the deep learning computation and communication overlap as much as possible. Therefore the maximum value between T_{comp} and T_{comm} is selected as T_{iter}.

$$T_{iter} = \max\left[T_{comp},\ T_{comm}\right] \tag{1}$$

Fig. 8 shows the ratio of pure communication time in one iteration of DNN training when computation and communication overlaps. When the communication time is shorter than the computation time, the communication overhead becomes zero because the computation time hides the communication time.

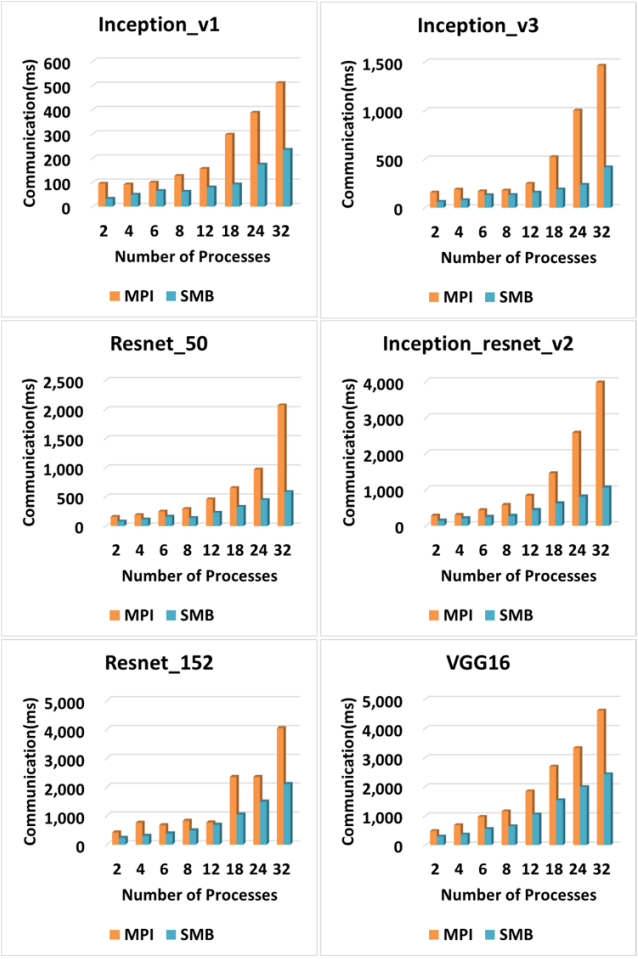

FIGURE 7. Comparison of communication time per model.

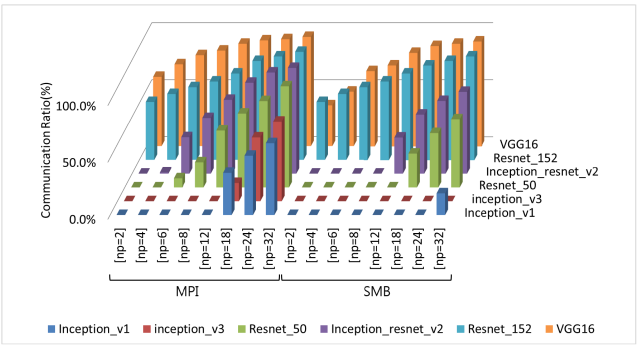

FIGURE 8. Comparison of communication time ratio between MPI and SMB with considering the overlapping of computation and communication.

Fig. 9 shows comparison of 1 iteration training time between the MPI and SMB methods by changing the number of deep-learning workers by 6 deep learning models. The two bars(i.e. comp and comm) in the Fig. 9 overlap each other because the figure show the result of the parallel scenario. If comp bar only appears, that means the comm bar is hidden by comp bar because the communication time is smaller than

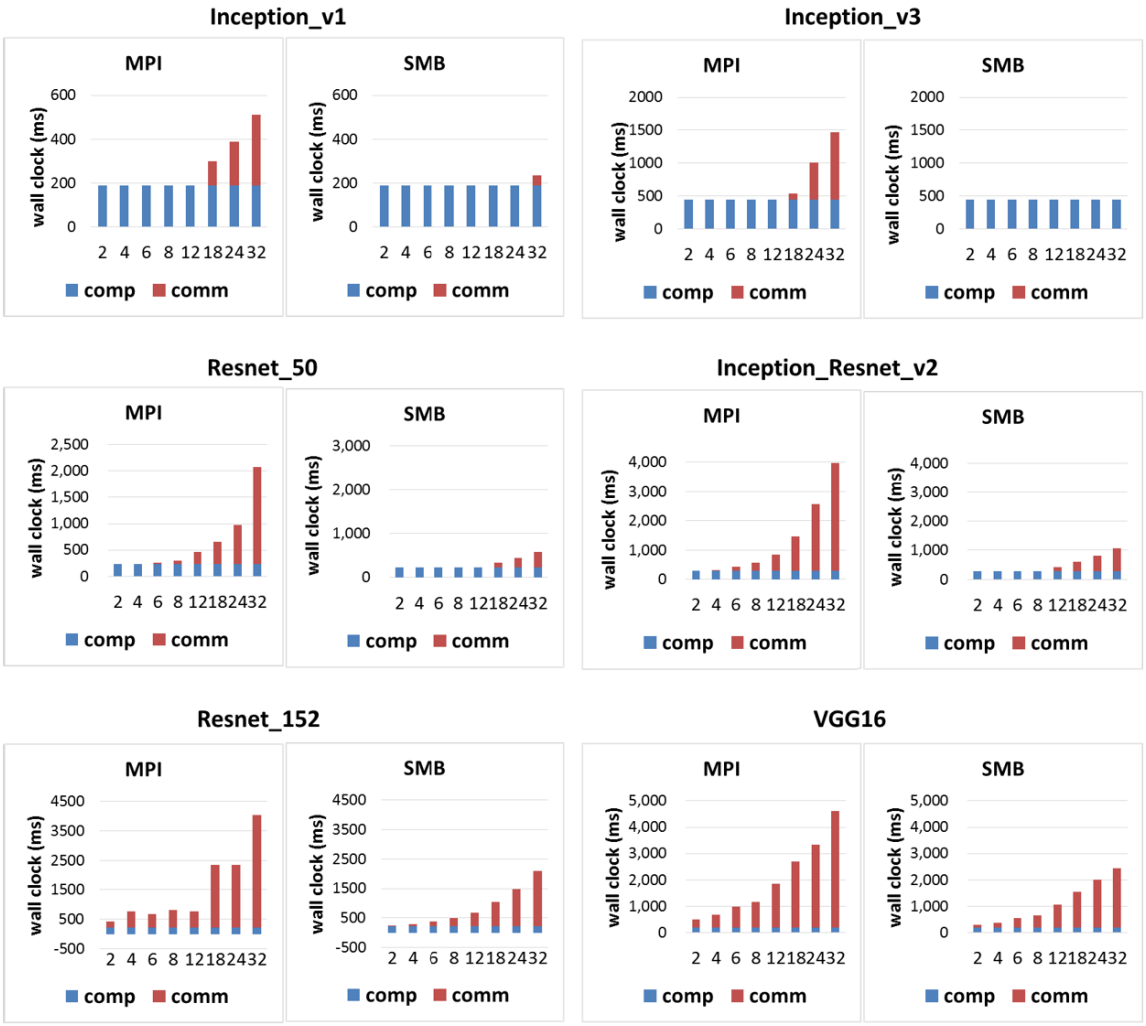

FIGURE 9. Comparison of 1 iteration training time per model with considering the overlapping of computation and communication.

the computation time. If comm bar appears, that means the communication time is greater than the computation time. Fig. 9 shows that the SMB method achieves better scalability and communication efficiency than the MPI method in terms of computation and communication time between the two methods.

The SMB method is much better than the MPI method for all models and all the number of workers. The communication time of the SMB method, which exceeds its computation time was 0 millisecond for up to 24 workers in the case of Inception_v1 with relatively small model size, and 45 milliseconds even when we extended the number of workers to 32, which is 7 times better than MPI method (323 milliseconds). In the case of Inception_v3 with a comparatively smaller parameter size and longest computation time, all the computation time of SMB method is hidden by computation time. Nevertheless, when we increase the number of processes to 32, the communication time of MPI method is 2.3 times longer than the computation time. As the model size increases, the difference in pure communication time (non-overlapped with computation time) between the MPI method and the SMB method decreases, but the SMB method is 5.2 times better than the MPI method in the case of Resnet_50, 4.8 times better in the Inception_resnet_v2 model and 2.0 times better in the Resnet_152 and VGG16 model. The average 1 iteration training time of SMB-based deep learning platform up to 3.5 times faster than MPI-based platform.

V. CONCLUDING REMARKS

In this paper, we proposed a new shared memory framework called Soft Memory Box (SMB) for large-scale deep learning parameter communication. We described its architecture, APIs, components, and usage. We measured the I/O performance of SMB framework. The read/write bandwidth of the single SMB Server reaches 6.7GB/s, which achieves 96% hardware utilization of FDR Infinibnad network (56Gbps). We also verified the high performance of SMB by developing a program that emulates the distributed DNN training with asynchronous SGD method. The SMB proposed in this paper utilizes remote direct memory access (RDMA) to transfer the

parameters stored in the local machine's memory directly into the memory of the remote shared memory node and exchange the DNN parameters by reading and writing the remote shared memory buffer. It greatly reduces the communication overhead caused by memory copying and network protocol processing. In the case of sequential scenario of asynchronous SGD, our proposed SMB-based platform is 2.1 times faster than that using the MPI-based platform. In the case of parallel scenario of asynchronous SGD, which computation and communication overlap, the communication using the SMB-based distributed DNN training platform is 2 through 7 times faster than that using the MPI-based platform. As the experiment results were obtained with a single SMB Server, only a limited performance improvement was verified. However, the number of SMB Servers can be increased to improve performance and can be optimized by calculating the bandwidth according to the size of the model and the number of deep learning workers. The proposed SMB-based approach provides a foundation for accelerating the distributed processing of large-scale DNN training to achieve high accuracy and fast computation in high-performance computing environment.

ACKNOWLEDGMENTS

This paper will be presented at the 40th IEEE/ACM International Conference on Software Engineering, Gothenburg, Sweden, 2018 [23].

REFERENCES

[1] S. Nagpal, M. Singh, R. Singh, and M. Vatsa, "Regularized deep learning for face recognition with weight variations," *IEEE Access*, vol. 3, pp. 3010–3018, 2015.

[2] A. M. R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, Montreal, QC, Canada, 2009, pp. 873–880.

[3] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. (2010). "Deep big simple neural nets excel on handwritten digit recognition." [Online]. Available: https://arxiv.org/abs/1003.0358

[4] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, Lake Tahoe, NV, USA, 2012, pp. 1097–1105.

[5] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Trans. Audio, Speech, Lang. Process.*, vol. 20, no. 1, pp. 30–42, Jan. 2012.

[6] G. E. Hinton *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, Oct. 2012.

[7] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proc. 25th Int. Conf. Mach. Learn.*, Helsinki, Finland, 2008, pp. 160–167.

[8] C. Yin, Y. Zhu, J. Fei, and X. He, "A deep learning approach for intrusion detection using recurrent neural networks," *IEEE Access*, vol. 5, pp. 21954–21961, 2017.

[9] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement.*, Broomfield, CO, USA, 2014, pp. 571–582.

[10] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proc. 28th Int. Conf. Int. Conf. Mach. Learn.*, Bellevue, WA, USA, 2011, pp. 265–272.

[11] K. Yang, M. Li, G. Zhu, and Y. Savaria, "A DAQM-based load balancing scheme for high performance computing platforms," *IEEE Access*, vol. 5, pp. 22504–22513, 2017.

[12] C. Kim *et al.*, "3D printed electronics with high performance, multi-layered electrical interconnect," *IEEE Access*, vol. 5, pp. 25286–25294, 2017.

[13] J. Wang and L. Cheng, "DistDL: A distributed deep learning service schema with GPU accelerating," in *Proc. 17th Asia–Pacific Web Conf.*, Guangzhou, China, 2015, pp. 793–804.

[14] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, Lake Tahoe, NV, USA, 2012, pp. 1223–1231.

[15] A. Krizhevsky. (2014). "One weird trick for parallelizing convolutional neural networks." [Online]. Available: https://arxiv.org/abs/1404.5997

[16] W. Wang *et al.*, "Deep learning at scale and at ease," *ACM Trans. Multimedia Comput., Commun., Appl.*, vol. 12, no. 4s, 2016, Art. no. 69.

[17] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *Proc. Int. Conf. Learn. Represent.*, 2016, pp. 1–10. [Online]. Available: https://arxiv.org/abs/1604.00981

[18] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *Proc. 11th USENIX Symp. Oper. Syst. Design Implement.*, Broomfield, CO, USA, 2014, pp. 583–598.

[19] B. Recht, C. Re, S. J. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. 25th Adv. Neural Inf. Process. Syst. (NIPS)*, 2011, pp. 693–701.

[20] C. Noel and S. Osindero, "Dogwild!—Distributed hogwild for CPU & GPU," in *Proc. NIPS Workshop Distrib. Mach. Learn. Matrix Comput.*, 2014, pp. 1–6.

[21] D. Yu, K. Yao, and Y. Zhang, "The computational network toolkit [best of the Web]," *IEEE Signal Process. Mag.*, vol. 32, no. 6, pp. 123–126, Nov. 2015.

[22] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, Orlando, FL, USA, 2014, pp. 675–678.

[23] S. Ahn, J. Kim, and S. Kang, "Poster W37: A novel shared memory framework for distributed deep learning in high-performance computing architecture," in *Proc. 40th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, Gothenburg, Sweden, May/Jun. 2018.

[24] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Boston, MA, USA, Jun. 2015, pp. 1–9.

[25] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. (2015). "Rethinking the inception architecture for computer vision." [Online]. Available: https://arxiv.org/abs/1512.00567

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Las Vegas, NV, USA, Jun. 2016, pp. 770–778.

[27] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-ResNet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell. (AAAI)*, San Francisco, CA, USA, 2017, pp. 4278–4284.

[28] K. Simonyan and A. Zisserman. (2014). "Very deep convolutional networks for large-scale image recognition." [Online]. Available: https://arxiv.org/abs/1409.1556

SHINYOUNG AHN received the B.S. and M.S. degrees in information engineering from Sungkyunkwan University, Seoul, South Korea, in 1997 and 1999, respectively, and the M.S. degree in software engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2005. He is currently pursuing the Ph.D. degree with the Software Architecture Laboratory, Korea Advanced Institute of Science and Technology, Daejeon, South Korea. He has been a Principal Researcher with the High Performance Computing Research Group, Electronics and Telecommunications Research Institute, Daejeon, since 1999. His main areas of research interest are high-performance computing, deep learning, distributed and parallel computing, and software architecture.

JOONGHEON KIM (M'06–SM'18) received the B.S. and M.S. degrees in computer science from Korea University, Seoul, South Korea, in 2004 and 2006, respectively, and the Ph.D. degree in computer science from the University of Southern California (USC), Los Angeles, CA, USA, in 2014. In industry, he was with LG Electronics, Seoul, from 2006 to 2009, InterDigital, San Diego, CA, USA, in 2012, and Intel Corporation, Santa Clara, CA, USA, from 2013 to 2016. He has been an Assistant Professor with Chung-Ang University, Seoul, since 2016. He is a member of the IEEE Communications Society. He received the Annenberg Graduate Fellowship along with his Ph.D. admission from USC in 2009.

EUNJI LIM received the B.S. and M.S. degrees in computer science from Pusan National University, Busan, South Korea, in 1999 and 2001, respectively. She has been a Principal Researcher with the High Performance Computing Research Group, Electronics and Telecommunications Research Institute, Daejeon, South Korea, since 2001. Her main areas of research interest are distributed systems and high-performance computing.

SUNGWON KANG received the B.A. degree from Seoul National University, Seoul, South Korea, in 1982, and the M.S. and Ph.D. degrees in computer science from the University of Iowa, USA, in 1989 and 1992, respectively. From 1993 to 2001, he was a Principal Researcher with the Korea Telecom Research and Development Group. From 2003 to 2014, he was an Adjunct Faculty of the Master of Software Engineering Program with Carnegie Mellon University. He joined the Korea Advanced Institute of Science and Technology. His research areas include software architecture, software product line, software testing, and data-based software engineering. He served as the Chair and Program Chair of numerous international conferences. He served as the Editor of the Korean Journal of Software Engineering Society and the President of the Korean Software Engineering Society.