Enabling Efficient Large-Scale Deep Learning Training with Cache Coherent Disaggregated Memory Systems

Zixuan Wang, Joonseop Sim†, Euicheol Lim†, Jishen Zhao
University of California, San Diego †System Architecture Division, SK hynix

***Abstract*—Modern deep learning (DL) training is memory-consuming, constrained by the memory capacity of each computation component and cross-device communication bandwidth. In response to such constraints, current approaches include increasing parallelism in distributed training and optimizing inter-device communication. However, model parameter communication is becoming a key performance bottleneck in distributed DL training. To improve parameter communication performance, we propose COARSE, a disaggregated memory extension for distributed DL training. COARSE is built on modern cache-coherent interconnect (CCI) protocols and MPI-like collective communication for synchronization, to allow low-latency and parallel access to training data and model parameters shared among worker GPUs. To enable high bandwidth transfers between GPUs and the disaggregated memory system, we propose a decentralized parameter communication scheme to decouple and localize parameter synchronization traffic. Furthermore, we propose dynamic tensor routing and partitioning to fully utilize the non-uniform serial bus bandwidth varied across different cloud computing systems. Finally, we design a deadlock avoidance and dual synchronization to ensure high-performance parameter synchronization. Our evaluation shows that COARSE achieves up to 48.3% faster DL training compared to the state-of-the-art MPI AllReduce communication.**

***Keywords*-deep learning; training; cache coherence;**

I. Introduction

Deep learning (DL) is a key enabler of tremendous breakthroughs in tasks ranging from object detection [29] to speech recognition [18], [19] and language translation [22]. As DL models evolve, the model size has been continuously increasing to adapt to complex tasks and improve accuracy [4], [9]. The size of training data also increases to achieve higher accuracy [3], [15], [17].

A common practice to train a large DL model with a large amount of input data is distributed DL training on multiple workers (computational devices such as GPUs or TPUs [23], [47]). These workers are optimized for high-performance matrix multiplication, which is heavily used in DL training. However, distributed DL training is increasingly constrained by (1) single device parallelism and on-device memory capacity and, (2) cross-device communication overhead [20], [42], [62].

On the one hand, the state-of-the-art GPUs and TPUs use tens of Gigabytes of on-device high-performance memory [14], [44], which is an order of magnitude smaller than the system memory. The on-device memory capacity does not evolve as fast as DL model sizes, leading to the popularity of data parallelism [10], model parallelism [40], and pipeline parallelism [20], [42], [50]. Although parallelism schemes enable the training of large DL models, they inevitably introduce additional code complexity and trade-offs without always leading to training speedup [41]. Another solution is to build extended parameter storage with system memory and storage [63]. But this only benefits a subset of DL models with certain parameter characteristics (e.g., parameter data density) [63].

In addition, the bandwidth of commodity cross-device communication, including PCIe [51], NVLink [48], and the network, is an order of magnitude lower than on-device memory bandwidth. This imposes a major performance bottleneck when adding more worker devices to a distributed training system. As discussed in Section II-B, the overhead due to cross-device communication is up to 76% of total DL training time, significantly degrading the device utilization and training performance [42], [62]. Recent efforts are devoted to decentralized training with the message passing interface (MPI) to mitigate the communication bottleneck [45], [57]. However, MPI creates a synchronous point that forces the faster workers to wait for the slower ones, hence degrading the computation utilization of worker devices [39].

The emerging cache-coherent interconnection (CCI) [7], [13] promises new opportunities in addressing the memory capacity and communication challenges. CCI allows the memory to be attached to serial buses (e.g., PCIe) instead of only the memory bus (DDR channels). As such, system memory capacity is no longer constrained by the memory channels supported by the processor sockets. CCI protocols typically provide hardware-level cache coherence support and customized protocols to reduce memory access latency. With CCI, host processors can directly issue memory load/-store instructions to memory devices on a serial bus without using a software driver. CCI also enables disaggregated memory systems, where each memory device incorporates an on-device processor with on-device memory. The on-device processor can access the host CPU's memory through CCI [7], [13].

However, a naïve design, which simply offloads parameter synchronization to the CCI memory devices, falls short to enable efficient large-scale DL training due to critical design

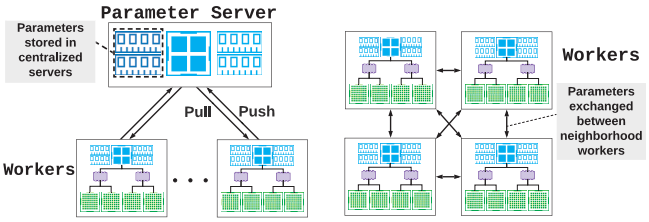
challenges. First, parameter data transfer is bounded by a single memory device's serial bus bandwidth; coherence traffic also increases with the number of computation devices sharing the same memory region, reducing the bandwidth available to accommodate parameter data transfer. Second, both parameter size and local/remote communication bandwidth characteristics are not uniformly distributed in different distributed systems; in addition, there is a non-uniform bandwidth and latency demand with different parameter sizes in DL training. As a result, no single partitioning and configuration of DL operations fit all.

Our goal in this paper is to enable efficient large-scale DL training by designing a parameter synchronization scheme that addresses the previously described challenges. To this end, we propose a Cache cOherent interconnected pARameter SErver (COARSE), which is built on top of collective communication for synchronization similar to that of MPI-based decentralized training. In particular, COARSE consists of three key design principles. First, we propose a decentralized parameter communication scheme to decentralize parameter synchronization and localize parameter storage. Second, we propose a tensor routing and partitioning scheme, which exploits parameter granularity and non-uniform interconnection bandwidth to fully utilize serial bus bandwidth and improve tensor locality in the GPUs. The tensor partitioning enables a pipelined tensor synchronization and takes advantage of serial bus bi-directional bandwidth; The bandwidth-aware tensor routing routes a GPU's tensor to a bandwidth-friendly memory device, even if they are not under the same serial bus switch. Finally, we develop a dual parameter synchronization scheme to reduce parameter synchronization traffic and enable high GPU computation utilization.

Internally, COARSE handles parameter synchronization with a set of collective communication based on CCI protocol – similar to that of MPI-based decentralized training – to reduce the communication latency and improve the synchronization bandwidth. Yet to allow easy integration with the commodity DL training frameworks, we provide a parameter push/pull interface similar to that of conventional parameter servers.

This paper makes the following key contributions:

- We propose COARSE, an efficient parameter synchronization scheme that accelerates distributed DL training by leveraging disaggregated memory systems.
- We identify non-uniform local and remote bandwidth distribution in different distributed systems.
- We build a COARSE prototype integrated with TensorFlow, offering significant improvements on training latency, computation utilization, and interconnection bandwidth utilization in various distributed training workloads running in different cloud systems.



(a) Parameter server. (b) MPI AllReduce.

Figure 1. Communication schemes in distributed DL training.

II. Background and Motivation

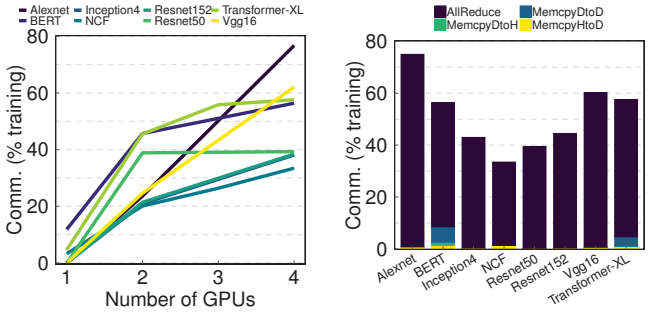
DL training has witnessed a wave of rapidly increasing scales of data sets and models. This has led to the popularity of distributed training to scale-out training workloads on distributed server nodes (*workers*), each with an increasing number of interconnected GPUs (Figure 1). As a result, the data communication overhead is becoming one of the key obstacles for the continuous scaling of distributed DL training [42], [62]. In this section, we motivate our design by discussing the challenges and opportunities of enabling efficient parameter communication with cache-coherent disaggregated memory systems.

A. DL Training Parallelism Schemes

Modern large DL models and data sets are too large to fit in a single worker while achieving the target accuracy within a reasonable period of time [56]. This leads to the exploration of various classes of DL parallelism schemes to train a DL workload on multiple server nodes – data, model, pipeline, and hybrid parallelism [21] – without an obvious performance winner. In this paper, we focus on data parallelism due to its popularity in commodity systems and the opportunity introduced by the emerging cache-coherent disaggregated memory systems (Section II-C). The major advantage of data parallelism is that it is applicable to any DL model without further domain knowledge of the model.

Data Parallelism. Data parallelism is widely used in state-of-the-art commodity DL training systems [10]. Rather than training a model with all the available input data on a single server, the system replicates the model across many workers and feeds each replica with a subset of the input data. Because the model replicas are trained using different input data, their parameters will typically diverge.

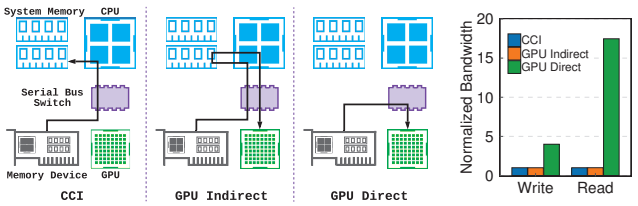
Parameter Server. To reconcile these parameters and ensure that all model replicas eventually converge, each replica periodically pushes its local parameter values to a centralized server, i.e., a parameter server. Parameter server is typically implemented as a distributed key-value store with consistent shared metadata, including server status and/or a hash table with distributed key-value mapping. The parameter server aggregates all the received updates for each parameter – e.g., by averaging them – and then sends back to all replicas a newly computed set of values, which will be used at



(a) Communication overhead with various number of GPUs.

(b) Overhead breakdown among various operations (4GPU).

Figure 2. Communication overhead in ML training.



(a) Communication schemes.

(b) Bandwidth.

Figure 3. CCI bandwidth with CPU load/store instruction (*CCI*), GPU indirect copy through CPU (*GPU Indirect*), and GPU direct peer-to-peer copy (*GPU Direct*). Read and write bandwidth are normalized to the *CCI* bandwidth, respectively.

the beginning of the next training iteration. However, not only the data but also the model size keeps skyrocketing. For instance, DLs used in recommendation systems or natural language processing (NLP) tasks commonly have large network parameters, e.g., BERT [12] model has $\sim$ 300 million parameters. To accommodate the huge number of parameters in large DL models, systems employ multiple parameter servers, with each one being responsible for a subset of the parameters.

Model Parallelism. Model parallelism partitions the operators in a training model across multiple workers; each worker evaluates and updates a subset of the model parameters for all inputs. However, model-parallel training suffers from computing resource under-utilization due to data dependency and the large communication overhead of intermediate results, such as activation and gradient [42].

Pipeline Parallelism. Pipeline parallelism partitions DL operations and splits a mini-batch into multiple micro-batches. Each worker computes the output of a model partition with a set of micro-batches, then propagates the output to the subsequent workers. By executing multiple micro-batches in parallel and making communication traffic point-to-point (stage to next stage), pipeline parallelism improves worker utilization compared to model parallelism [20], [42], [50]. However, such parallelism schemes lead to either low computing resource utilization [20] or accuracy loss [42], depending on their implementations.

B. Communication Issues in DL Training

Data parallelism imposes severe communication and synchronization overhead due to back-propagation updates. Such communication typically relies on two types of training system architectures (i) *centralized* communication (Figure 1a) – commonly used for inter-node communication between the parameter server and workers and (ii) *decentralized* communication (Figure 1b) – commonly used for intra-node communication between worker GPUs [59], [64]. In centralized communication, the computed gradients in each worker need to be transferred back to the parameter server for model updates, before continuing to compute the next mini-batch of training data. A parameter server provides a push/pull interface for each worker. As a result, a critical performance bottleneck is the parameter server communication bandwidth. In decentralized communication, workers directly communicate with each other via an MPI interface, such as `AllReduce`.

Recent studies show that data communication among workers is a major bottleneck in distributed DL training [42], [62]. We further evaluate the communication overhead in decentralized communication with eight DL models implemented by TensorFlow [60] and NVIDIA [46], as illustrated in Figure 2. We use NCCL [45], which is an optimized decentralized communication scheme using the MPI `AllReduce` interface. We train the models on an AWS EC2 `g4dn.12xlarge` instance [58] with various numbers of GPUs, and employ `nvprof` [49] to measure the communication overhead in terms of the time spent on communication operations, including CUDA memory copy operations and NCCL `AllReduce`. Figure 2a shows that an increasing number of GPUs leads to a higher percentage of communication overhead, with up to 76% of training time spent on data communication using four GPUs. Figure 2b shows that `AllReduce` dominates the communication latency overhead among all the operations that perform communication.

The key bottleneck with centralized communication in a single server node is the limited serial bus lanes in the CPU, which constrains the service throughput of GPU requests on parameters: In a multi-GPU server node, the number of serial bus lanes on CPUs is typically smaller than the total serial lanes required by all the GPUs. As a result, the concurrent access from GPUs to the parameter server is constrained by the CPU serial bus lanes.

The key issue with decentralized communication is that GPU computation is blocked during parameter synchronization: First, a parameter synchronization operation blocks all GPUs involved for synchronization, forcing the fast GPUs to wait for the slower ones [38], [39]. Second, the MPI `AllReduce` approach generates more communication requests than centralized training, depending on the topology adopted. Third, using a commonly adopted ring topology, the `AllReduce` performance is bounded by the lowest device-to-device bandwidth, leading to bandwidth utilization, e.g.,

as low as 34% on NVIDIA DGX-1 systems [62].

C. Opportunities With Disaggregated Memory Systems

Cache-coherent interconnection (CCI) protocols, such as CCIX [6], Gen-Z [13], and Intel CXL [7], enable cache-coherent disaggregated memory systems. By offering a CPU-transparent hardware coherence support, CCI allows the CPU to directly issue load and store instructions to access multiple memory devices attached to the existing or customized serial buses. These memory devices form a disaggregated memory pool: each memory device adopts an on-device processor and a large capacity on-device memory; the memory devices map their local memory into CCI-unified memory address space and share the address space with the host CPU and other memory devices [6], [13]. CCI also allows the host CPU to distribute memory-intensive computation jobs to the processors on memory devices, schedule the jobs, and collect computed results.

CCI Performance Benefits. CCI reuses existing serial bus physical layer protocol and builds a customized higher-level protocol stack [7]. This customized protocol focuses on improving fine-grained memory access latency while providing sufficient peak bandwidth (e.g., 90% of underlying serial bus peak bandwidth [7]). Thus, compared to existing serial bus protocols, e.g., PCIe [51]–[53], CCI can achieve higher bandwidth when accessing small memory blocks. This unique feature allows GPUs to seamlessly access external large-capacity disaggregated memory devices through CCI.

Promising CCI-based Peer-to-peer (P2P) Communication. We identify that peer-to-peer accesses to CCI-based disaggregated memory achieve significantly higher bandwidth than indirect access through the host CPU.

We evaluate the bandwidth of an FPGA-based disaggregated memory prototype implemented on industrial CCI protocol (more details in Section IV-C) under three use cases: (i) copying data from a disaggregated memory device to CPU memory (*CCI*). (ii) copying data from disaggregated memory to CPU then to GPU memory (*GPU Indirect*). (iii) directly copying data from disaggregated memory to GPU memory (*GPU Direct*). Figure 3 shows that GPU direct peer-to-peer copy provides 17× read bandwidth and 4× write bandwidth speedup. This motivates us to enable GPU direct access to CCI memory for better performance.

Benefits to Data Parallel DL Training. CCI is beneficial to data parallel training in two aspects: (i) in data parallel training, cross-device communication is a major overhead; this communication can take advantage of CCI low-latency memory access to improve the parameter synchronization performance. (ii) the parameter synchronization operations block the training procedure and take up GPU computing resources; while using CCI, GPU can work with memory device processors coherently to offload these synchronization operations to memory device processors, thus improving the GPU utilization and reducing the communication overhead.

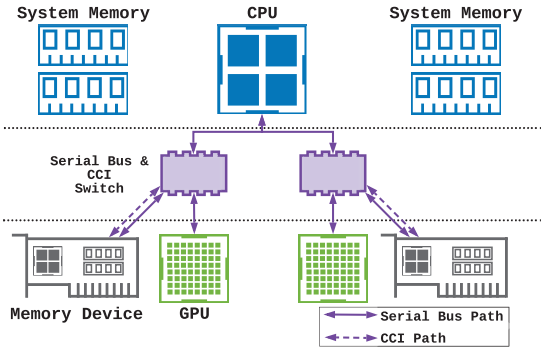


Figure 4. System deployment of a single server node.

III. COARSE Design

Overview. Our goal in this paper is to enable efficient large-scale DL training by designing a parameter synchronization scheme with (1) light communication traffic, (2) high interconnection bandwidth utilization, and (3) high GPU computation utilization. To achieve our goal, we propose Cache cOherent interconnected pARameter SErver (COARSE), which is a decentralized parameter synchronization design offloaded to CCI-based disaggregated memory systems. COARSE provides a parameter server push/pull interface for easy integration, while internally, it consists of the following novel parameter synchronization mechanisms:

- **A decentralized parameter communication scheme** to decentralize and localize parameter communication.
- **A tensor routing and partitioning scheme** which exploits non-uniform interconnection bandwidth characteristics to fully utilize serial bus bandwidth and improve tensor locality in GPUs.
- **A dual parameter synchronization scheme** to reduce parameter synchronization traffic and enable high GPU computation utilization.

This section presents the key ideas of COARSE design. Implementation details will be described in Section IV.

A. System Deployment

We assume a single server node deployment illustrated in Figure 4: GPUs are connected to serial bus switches and communicate with other serial bus devices as in commodity systems [47]. Each GPU is paired with a CCI-based disaggregated memory device (referred to as *memory device* in the rest of the paper) sharing the same serial bus switch, such that communication between a pair of GPU and memory device achieves full bandwidth. A similar approach in existing systems is to put a network card under each PCIe switch to improve GPU RDMA bandwidth [47]. Each memory device incorporates an on-device processor (e.g., with ARM cores) and a large-capacity memory. Intra-node

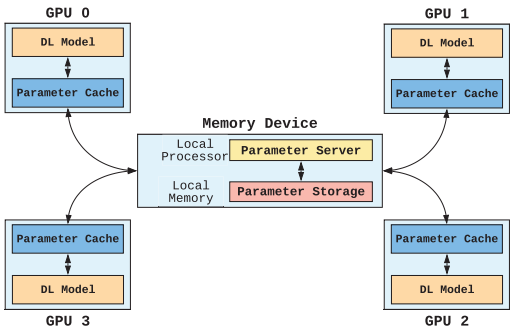


Figure 5. DENSE architecture.

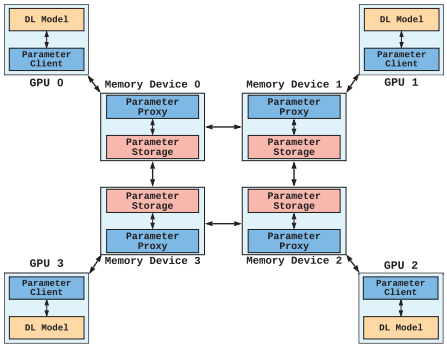


Figure 6. COARSE architecture.

memory devices are interconnected by a CCI. CCI protocols allow the memory devices to support (i) full-bandwidth data copy through a serial bus interface (the solid lines) and (ii) cache-coherence traffic through a CCI interface (the dashed lines) [7], [13]. As such, serial bus devices can copy data peer-to-peer through a serial bus switch or host bridge [43], bypassing data copy in system memory.

B. A Naïve Design

We assume a baseline architecture – *DENSE* – as shown in Figure 5. In DENSE, each GPU maintains a parameter data cache supported by CCI. To update parameters, the GPU first updates the local parameter cache and then employs CCI to update the global parameter coherently. The global parameters are stored in a memory device, which runs a parameter server on the on-device processor and stores the parameters in on-device memory. The memory device maps the on-device memory to CCI address space and shares it with other GPUs through CCI. This architecture bypasses the host CPU data copy, and we use it as a baseline in the following sections.

In DENSE, parameter synchronization jobs are offloaded to the processors in memory devices. Such offloading introduces promising benefits for DL training: Compared to conventional centralized training, the offloading allows GPUs to directly communicate with memory devices at full serial bus bandwidth, without being limited by CPU serial bus lanes; Compared to decentralized training, the offloading allows GPUs to continue with computation during parameter synchronization, substantially improving GPU utilization.

C. A Disaggregated Design

Simply offloading parameter synchronization to the memory devices in DENSE falls short to enable efficient large-scale DL training. In the following sections, we identify critical design challenges and insights obtained from our experiments in two different distributed system infrastructures, from Amazon AWS [58] and San Diego Supercomputing Center (SDSC) [61], respectively. Based on our observations, we propose a disaggregated parameter synchronization architecture, *COARSE*, as illustrated in Figure 6.

In COARSE, the parameter synchronization service is further disaggregated to multiple memory devices, providing higher communication bandwidth compared to DENSE's centralized architecture. Each memory device communicates with one GPU sharing the same serial bus switch for parameter updates, and communicates with other memory devices for parameter synchronization. The following sections introduce the design details of COARSE.

D. Decentralized Parameter Synchronization

Interconnection Bandwidth and Scalability Issues. Although disaggregated memory system allows multiple computation devices (e.g., GPUs) to share memory at low latency, parameter data transfer is bounded by serial bus bandwidth in the DENSE architecture. Such bandwidth limitation will constrain the number of computation devices sharing the disaggregated memory system. Furthermore, coherence traffic also increases with more computation devices sharing the same memory region [7], [13], reducing the bandwidth available to accommodate parameter data transfer on the CCI path.

Decentralized Parameter Communication. Our key idea to address the above challenges is to localize parameter storage and decentralize parameter synchronization. To this end, we propose a decentralized parameter synchronization scheme, which splits the original synchronization functionality between parameter proxy and storage.

Figure 7 shows an overview of our design, which communicates parameters across three levels in a hierarchy: *parameter client*, *parameter proxy*, and *parameter storage*. Each worker GPU runs a parameter client that maintains local parameters and communicates with a dedicated proxy to perform parameter synchronization. Each client provides a *push* and a *pull* interface to the local DL training operations – a parameter server functionality in conventional designs. Both parameter proxy and parameter storage run on the memory devices. A proxy is a communication service, acting as a bridge between a client and the parameter storage located in the same memory devices. The proxy also maintains a cache of the parameters stored with the parameter storage in the same memory device. As such, the multi-level

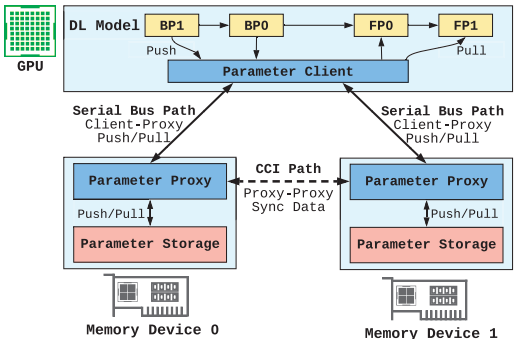
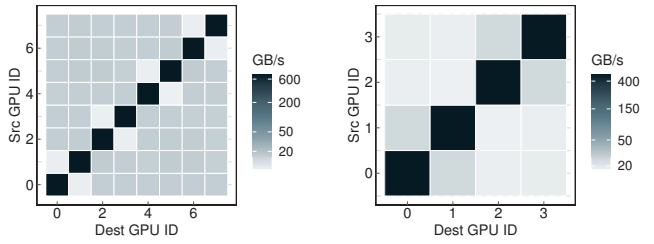


Figure 7. Decentralized parameter synchronization.



(a) V100: Local PCIe bandwidth (9GB/s) is lower than remote (16GB/s).

(b) P100: Local PCIe bandwidth (25GB/s) is higher than remote (18GB/s).

Figure 8. PCIe device-to-device bidirectional bandwidth on (a) V100 from AWS and (b) P100 from SDSC.

parameter communication scheme reduces the data traffic by localizing parameter communication between client-proxy and local proxy-storage pairs. This allows the CCI path to accommodate proxy-proxy parameter synchronization at low latency, as illustrated in Figure 7.

E. Tensor Routing and Partitioning

In COARSE clients, we propose (1) a *tensor routing* mechanism to exploit non-uniform parameter size and bandwidth distribution; and (2) a *tensor partitioning* mechanism to exploit the bidirectional bandwidth offered by the serial bus interface in disaggregated memory systems.

Non-uniform Parameter Size and Local/Remote Bandwidth Distribution. We observe that both parameter size and local/remote communication bandwidth characteristics are not uniformly distributed in our experiments in AWS and supercomputing distributed systems. To investigate communication bandwidth characteristics, we measure PCIe peer-to-peer communication in two systems: (1) an AWS EC2 `p3` instance with eight NVIDIA V100 GPUs, interconnected by NVLink and PCIe; this is similar to an NVIDIA DGX-1 system [47]; (2) an instance in the supercomputing center with four NVIDIA P100 GPUs, interconnected by PCIe. Figure 8 shows the bidirectional bandwidth between pairs of GPUs. On the supercomputing instance, local PCIe bandwidth (under the same PCIe switch) is higher than remote bandwidth, which is straightforward to imagine. However, the AWS instance appears to have a reverse bandwidth characteristic – higher remote than local bandwidth[1]; such bandwidth "anti-locality" may be caused by unbalanced physical signal paths in the PCIe switch chipsets [5], [31]. In addition, we observe a non-uniform bandwidth and latency demand with different parameter sizes in our experiments with DL training in both systems: small-size parameter communication (less than 2MB) is latency-critical because the communication does not saturate the serial bus bandwidth; instead, transfer of large-size parameters is bandwidth critical. These observations indicate that no single routing and partitioning of DL operations fits all.

[1]This reverse bandwidth distribution is also observed by a recent study [31] and referred to as "anti-locality".

Tensor Routing. Ahead of DL training, COARSE builds a routing table for each client, which describes the destination proxies for various sized tensor push/pull requests. This tensor routing mechanism is based on two observations: (i) using serial buses, small-sized data transfer is latency-sensitive while large data transfer is bandwidth-sensitive (Figure 14); (ii) non-uniform bandwidth: local serial bus communication bandwidth is not always higher than remote communication, although local latency is always better.

The routing table contains three entries, a data size threshold, a proxy id ($BwProxy$) for large tensor requests, and another proxy id ($LatProxy$) for small ones. The client issues tensor push/pull request to the $BwProxy$ if the tensor size is larger than the threshold, otherwise to the $LatProxy$.

Benefits of Bidirectional Data Transfer. Serial bus interfaces, such as PCIe and NVLink, support bidirectional data transfer that delivers close to $2\times$ bandwidth of unidirectional transfer. For example, we observe that for a pair of local GPUs, the unidirectional bandwidth is 13GB/s, whereas the bidirectional bandwidth is 25GB/s, in an instance from the San Diego Supercomputing Center.

Tensor Partitioning. In COARSE, parameter synchronization includes three steps: client push, proxy synchronization, and client pull. Serial bus bidirectional data transfer allows the client push/pull to be processed concurrently. But with unequal-sized tensors, this bidirectional transfer is not fully utilized: as shown in Figure 9 FIFO case, there is no client push/pull between the tensor 0 push and tensor 1 pull.

To fully utilize bidirectional bandwidth, COARSE clients partition the tensors into equal-sized small shards and synchronize them as a pipeline. With tensor partitioning, the communication pipeline is filled without idle time (Figure 9), and the proxy synchronization can start as soon as the first tensor shard arrives.

Dynamic Partitioning. COARSE adopts a profiler to build the tensor routing table and determine the tensor partitioning

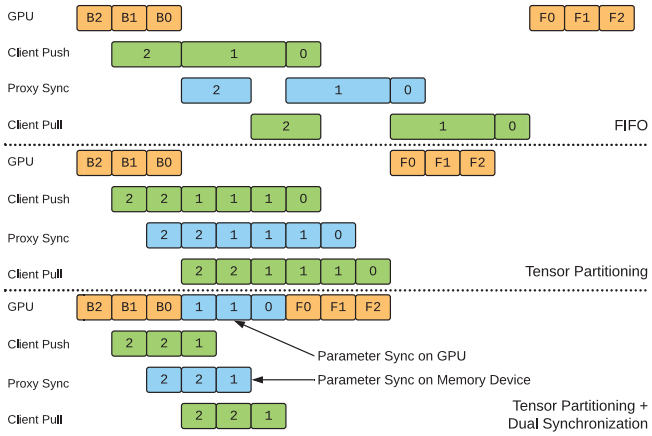


Figure 9. Tensor partitioning and Dual synchronization.

size at runtime.

To build the routing table for a client, the profiler measures the client's communication bandwidth and latency to each proxy, then chooses a proxy with the lowest latency ($LatProxy$) and one with the highest bandwidth ($BwProxy$). If $LatProxy$ and $BwProxy$ are not the same one, the profiler further measures the communication time T under different sized data, and determines a size S that makes $T_{LatProxy}(S) = T_{BwProxy}(S)$. The client considers a parameter latency-sensitive if it's smaller than S, and sends it to $LatProxy$.

To find the tensor partition size, the profiler measures communication bandwidth under various sized data between a client and its $BwProxy$. The profiler determines the smallest data size S' among all tested data sizes that achieve the full bandwidth. The client partitions large tensors into S' sized small shards.

COARSE uses a dynamic profiling mechanism: before the training starts, COARSE profiles the communication performance to determine the initial routing table and partitioning size; while training is in progress, COARSE periodically profiles the communication and updates the routing and partitioning strategies accordingly.

F. Dual Parameter Synchronization

To efficiently synchronize the parameters, we propose (1) a priority-based dual synchronization mechanism to exploit the tensor locality and (2) a queue-based synchronization scheme to prevent synchronization deadlocks.

Dual Synchronization. In a DL model backward pass, parameters are updated in reverse order. Therefore, tensors from the first few layers are updated at the end of a training iteration while immediately consumed by the forward pass of the next iteration. These tensors need to be prioritized and synchronized as fast as possible to start the next iteration. To this end, COARSE adopts dual synchronization (Figure 9): the first few layers' tensors are synchronized by worker GPUs, while the rest layers' tensors are pushed to proxies and synchronized by proxies. In dual synchronization, high priority tensors are synchronized immediately on worker GPUs, instead of being pushed to proxies and waiting until low priority tensor synchronizations to be finished.

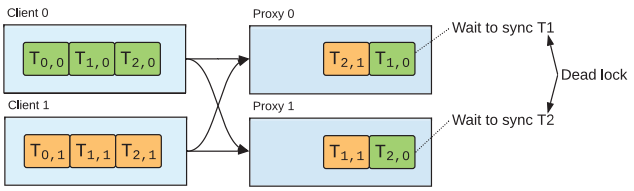


Figure 10. Deadlock in first-come-first-serve synchronization. $T_{i,j}$ represents tensor i from client j.

COARSE determines the dual synchronization strategy with a performance estimation: suppose there are p worker GPUs and p memory devices; the DL model synchronizes n bytes of parameters in each iteration, the first m bytes are sent to memory devices, and the rest is synchronized by worker GPUs.

Using ring AllReduce synchronization, each proxy sends $\frac{2(p-1)}{p} \times m$ bytes to one neighbor proxy, and receives the same amount. So the proxy synchronization time is $T_{sync}(proxy) = (\frac{2(p-1)}{p} \times m)/BW_{proxy}$, where BW_{proxy} is the communication bandwidth between proxies. Similarly, the GPU synchronization time is $T_{sync}(GPU) = (\frac{2(p-1)}{p} \times (n-m))/BW_{GPU}$.

Let T_{FP} denote forward pass computation time and T_{BP} denote backward pass time. The training iteration time is estimated as:

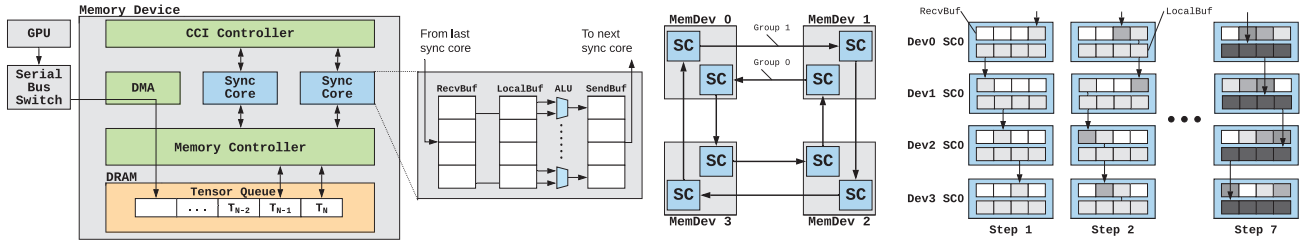
$$T_{train} = max \begin{cases} T_{FP} + T_{BP} + T_{sync}(GPU) \\ T_{FP} + T_{sync}(proxy) \end{cases}$$

p is assigned by the user, n is defined by the DL model, BW_{proxy} and BW_{GPU} are measured by the profiler (Section III-E), T_{FP} and T_{BP} are measured by running a few iterations of training.

COARSE calculates the optimal value for m that minimizes the training time T_{train}. This m is then used for dual synchronization.

Deadlock Avoidance. A first-come-first-serve (FCFS) synchronization scheme may cause deadlocks, as shown in Figure 10: client 0 send tensor 1 to proxy 0 and tensor 2 to proxy 1, for synchronization. In FCFS scheduling, proxy 0 waits to synchronize tensor 1 while proxy 1 waits for tensor 2, and thus causing deadlock.

To prevent such deadlocks, COARSE adopts a queue-based synchronization if one proxy is shared by multiple clients: A proxy maintains a queue for each client that stores the tensors pushed by this client. The proxy synchronizes all tensor queues concurrently so that it's not blocked on a single tensor synchronization.



(a) Cores for sync jobs. (b) Sync groups. (c) Sync scheme.

Figure 11. Parameter synchronization with sync cores (denoted as *SC* in figure b and c).

IV. Implementation

In this section, we describe the implementation details of COARSE and the FPGA-based disaggregated memory prototype.

A. Sync Cores

COARSE offloads parameter synchronization jobs to memory devices. These jobs rely on simple computation operations such as the sum of float numbers, which do not require the full features of a generalized processor. Executing synchronization jobs on low-power generalized processors, such as ARM Cortex-A53 on our Bittware Soc-250 FPGA, does not achieve high performance due to the limited number of arithmetic logic units. COARSE adopts specialized near memory processing cores – *Sync Core*s – to execute synchronization jobs with low design cost and high parallelism.

Sync Core Design. As shown in Figure 11a, each memory device is equipped with a set of sync cores, which can access local DRAM through the memory controller, and remote CCI memory device through the on-device CCI controller. Each sync core communicates with remote sync cores on other memory devices to synchronize the parameters. Each sync core maintains three buffers, a *RecvBuf*, a *LocalBuf*, and a *SendBuf*, to store the intermediate synchronization results. The sync core maps these buffers to CCI address space to accommodate direct read and write access from remote sync cores. To perform the arithmetic operations in parallel, each sync core employs a set of arithmetic logic units (ALU) that computes with the first two buffers' entries and stores the result into SendBuf.

Sync Scheme. To synchronize parameters, the sync cores perform group-based collective communication through CCI, where multiple groups synchronize different parameters in parallel. Each group consists of sync cores from each memory device, and uses ring-based topology for communication. Two adjacent groups use different ring directions to fully utilize the bidirectional bandwidth. Figure 11b shows an example configuration of two synchronization groups with four memory devices. In this configuration, group 1 use a reverse ring direction of group 0, so that communication between each pair of memory device is always bidirectional.

Figure 11c shows an example of the parameter synchronization scheme with one group of sync cores: Each sync core first loads a chunk of tensor data from local DRAM into LocalBuf and starts the iterative synchronization. In each iteration, each sync core sends an entry from SendBuf to the next sync core's RecvBuf, and receives an entry from the last sync core. It then applies the computation operation on the received entry and corresponding LocalBuf entry, stores the result in SendBuf, and sends it in the next iteration. When all entries are synchronized, the sync core writes back the updated entries to DRAM and starts to synchronize the next chunk.

Fault Tolerance. Many existing ML frameworks rely on checkpointing to ensure fault tolerance: during the training process, the ML framework periodically takes a snapshot of ML model's parameters and saves it to disk. In case any worker GPU fails during the training, the framework can recover its training progress with the latest checkpoint instead of training the model from the start. The checkpointing typically requires GPUs to transfer their model parameters to a centralized persistent storage and thus degrades the training performance.

To improve the checkpointing performance, COARSE leverages low-overhead copy-on-write with fine-grained snapshotting: when a memory device receives a parameter, it performs a copy-on-write procedure if the parameter contains updates to the previous version. Then at the end of each epoch, the memory device takes a snapshot of the current version of all parameters and saves it as a checkpoint.

B. COARSE

COARSE is implemented as a Python library with Python 3 and CUDA 11.4, and provides a plugin to TensorFlow framework [2]. It manages device memory, schedules synchronization, and handles requests between clients, proxies, and storage.

COARSE leverages GPUs to emulate the CCI memory device due to the lack of CCI support in existing CPUs and motherboards. COARSE accepts a user-defined GPU partition table that describes which GPU acts as a worker and which acts as a memory device.

Offline Profiling. Ahead of DL training, COARSE profiles the communication latency and bandwidth between GPUs and memory devices, as described in Section III-E. This profiler is implemented as a CUDA program and measures peer-to-peer GPU communication by default. However, if

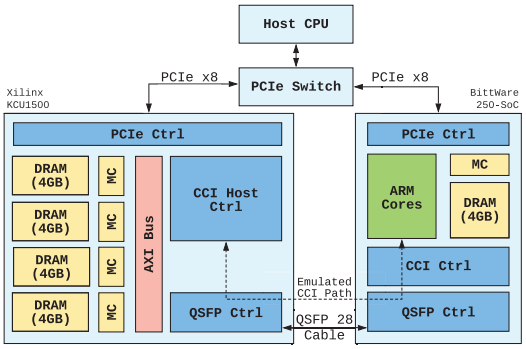


Figure 12. Disaggregated memory prototype with two FPGAs.

GPUs are interconnected with NVLink, the profiler disables the NVLink feature to measure the PCIe bandwidth.

Client. A client maintains a tensor queue in worker GPU memory. When the DL model pushes a tensor, the client compares the tensor size with the partition threshold. If the tensor size is smaller than the threshold, it's enqueued directly without partitioning. The client partitions a large tensor into multiple shards and enqueues them separately, where each shard's size is equal to or larger than the threshold to maximize bandwidth utilization.

The client actively dequeues tensors, sends small tensors to the latency-friendly proxy, large tensor shards to the bandwidth-friendly proxy. After the first tensor is pushed to proxies, the client starts to pull the updated tensors and reconstruct the original tensors based on the partition history. These client operations are implemented in Python and integrated into the TensorFlow computation graph.

Proxy and Storage. Each GPU-emulated memory device runs one parameter proxy and one parameter storage. The proxy leverages stream processors on existing GPUs as sync cores, and uses NCCL AllReduce to emulate collective synchronization operations between CCI memory devices. After each synchronization, the proxy copies the updated tensors to the storage. The parameter storage maintains a key-value table that maps a tensor id to the tensor value. To model the CCI communication characteristics, we implement a set of CUDA kernels that inject delays to communications based on data sizes.

TensorFlow Integration. COARSE provides a *distribution strategy* for TensorFlow version 2 or later. To use COARSE, the user just needs to import COARSE Python library and replace the original distribution strategy with COARSE strategy, which typically requires 2 lines of code change.

C. Disaggregated Memory Prototype

We implement a disaggregated memory prototype based on an industrial CCI protocol using two FPGAs, as illustrated in Figure 12.

The first FPGA works as a shared memory pool. We implement the host controller of the CCI protocol in the first FPGA. This controller manages the FPGA on-device DRAM and shares it with the host CPU and other FPGAs. To share the local memory with the host CPU, the controller exposes the on-device DRAM as a byte-addressable PCIe bar region. A program running on the host CPU can `mmap` [36] this region and use load/store instructions to access the FPGA memory. To share the local memory with other FPGAs, the controller accepts network requests from other FPGAs through QSFP connections.

The second FPGA works as a near-memory processing device. It contains the client controller of the CCI protocol and connects to the first FPGA through a QSFP cable. The ARM core on this FPGA can access the shared memory through the CCI client controller.

Table I
MACHINES FOR DL TRAINING

Instance	CPU & Memory	GPU	Network
AWS g4dn	Intel 8259CL 384 GB	8 * NVIDIA T4 PCIe	100GBE
AWS p3	Intel E5-2686 v4 768 GB	8 * NVIDIA V100 NVLink, PCIe	100GBE
SDSC	Intel E5-2680 v3 128 GB	4 * NVIDIA P100 PCIe	100GBE

V. EVALUATION

We first profile the CCI prototype performance and build a performance model based on the profiling result. We then use the performance model to evaluate the DL training with COARSE, on three different systems.

A. Experiment Setup

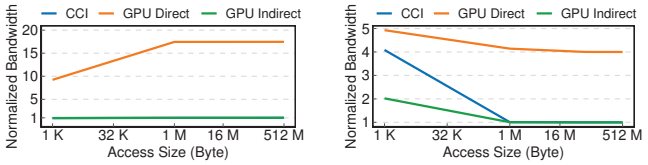
The disaggregated memory prototype adopts a Xilinx KCU1500 FPGA [24] and a BittWare 250-SoC FPGA [1] to emulate the shared memory pool. Two FPGAs are interconnected with one QSFP cable [30] so that the ARM cores on 250-SoC can access the shared memory pool through CCI. KCU1500 exposes its shared memory as a byte-addressable PCIe bar region for the host CPU to access. The host system includes an Intel 8700K CPU, 32GB DDR4 memory, and an NVIDIA GTX1080 GPU. The motherboard adopts two PCIe switches enabling the GPU to access KCU1500 through peer-to-peer PCIe communication.

We profile the performance of this prototype and build a performance model based on profiling results to estimate the CCI bandwidth under different access sizes. We use this performance model to evaluate DL training time *only* in the baseline *DENSE* case (Figure 16).

We evaluate DL training with COARSE on three machine instances, as listed in Table I. On each machine, we use half of the GPUs to emulate the CCI memory devices. On AWS p3 machine, we provide an additional 2:1 configuration where each memory device is shared by two worker GPUs.

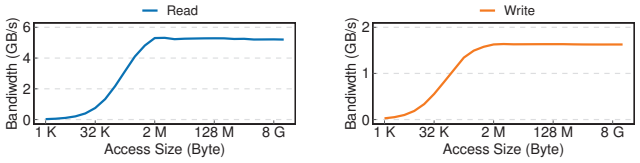
B. CCI Prototype Performance

We map the disaggregated memory prototype's memory to the host CPU memory space using `mmap` [36], and evaluate



(a) Read bandwidth (b) Write bandwidth

Figure 13. CCI bandwidth with CPU load/store instructions (*CCI*), GPU indirect copy through CPU (*GPU Indirect*), and GPU direct peer-to-peer copy (*GPU Direct*). Numbers are normalized to the CCI bandwidth with 512MB access size.



(a) Read bandwidth (b) Write bandwidth

Figure 14. PCIe DMA bandwidth from host CPU to FPGA.

its bandwidth in three use cases: (1) *CCI*: Host CPU accessing CCI memory using memory load/store instructions. (2) *GPU Indirect*: GPU accessing CCI memory by first copying data to host CPU memory. (3) *GPU Direct*: GPU directly accessing CCI memory using `cudaMemcpy` with prototype's PCIe address.

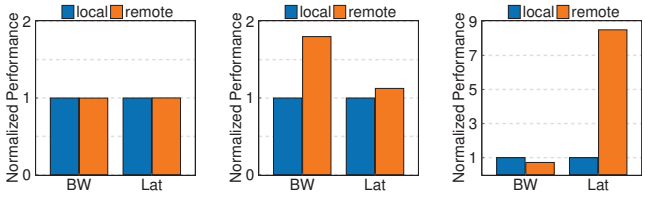
Figure 13 shows the CCI bandwidth under different access sizes. Figure 13a shows that CCI read bandwidth remains stable under all access sizes. The difference between CCI line and GPU Indirect line is not visible in Figure 13a, which means the GPU Indirect read bandwidth is bounded by CCI bandwidth. However, the GPU Direct read bandwidth achieves 9x-17x speedup compared to CCI. Figure 13b shows that GPU Direct write bandwidth achieves 1.25x-4x speedup. This bandwidth speedup shows that CCI memory is more beneficial to the serial bus devices (e.g., GPUs) if CCI enables peer-to-peer communication.

We further profile the FPGA DMA bandwidth to estimate the full device bandwidth. As shown in Figure 14, DMA read and write achieves max bandwidth with an access size of 2MB or higher.

We build a bandwidth versus data size performance model for the following DL training evaluations. We assume the GPU Direct method achieves full serial bus bandwidth, and use correlated speedup/slowdown to derive CCI and GPU Indirect bandwidth in the DENSE system.

C. Tensor Routing

As discussed in Section III-E, to build the tensor routing table for a client, COARSE profiles the communication performance from the client to each proxy. Figure 15 shows the profiling result for one client on each machine. Each figure shows client communication to (i) a *local* proxy that shares the PCIe switch with the client, and (ii) a *remote* proxy that achieves the highest bandwidth.



(a) T4 (b) V100 (c) P100

Figure 15. Communication performance from one client to its local and remote proxies. The local proxy shares the PCIe switch with the client. Numbers are normalized to local proxy performance.

On AWS machines with T4 GPUs (Figure 15a), communication to local or remote proxy makes no difference, so each client always communicates with its local proxy.

On AWS machines with V100 GPUs (Figure 15b), accessing the local proxy achieves lower latency, while the remote proxy achieves higher bandwidth. So a client communicates with the local proxy for small tensors, and the remote proxy for large tensors.

On SDSC machines with P100 GPUs (Figure 15c), accessing the local proxy achieves lower latency and higher bandwidth. So the client always communicates with its local proxy.

D. Training Speedup

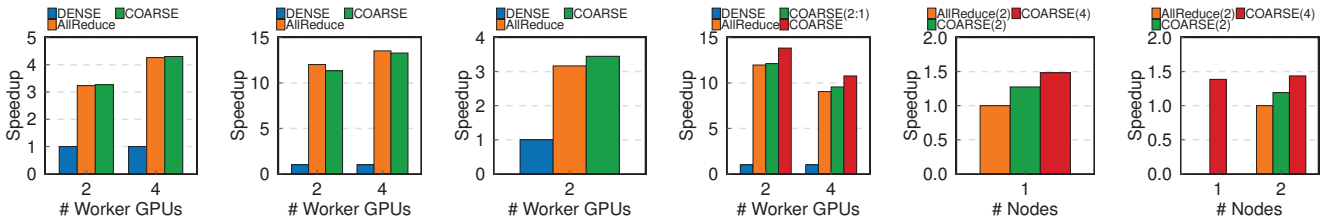
We choose two DL models, ResNet50 [16] and BERT [12], to evaluate our design. We run ResNet50 training with ImageNet [11] dataset, using a per-GPU batch size of 64. We run BERT fine-tuning with SQuAD 1.1 [55] dataset, using a per-GPU batch size of 2.

We evaluate three communication schemes, (1) a naïve disaggregated CCI memory design as shown in Figure 5 (*DENSE*), (2) AllReduce using NCCL without CCI memory (*AllReduce*), and (3) COARSE where the number of worker GPUs is equal to the number of CCI memory devices (*COARSE*). We use the training speedup (compared to the *DENSE* case) as the major metric.

To estimate the communication time in the *DENSE* case, we run the DL training with a parameter server running on CPU, and estimate the GPU to CPU bandwidth according to the CCI prototype performance model (Section V-B). This parameter server is provided by TensorFlow's `ParameterServer` distribution strategy.

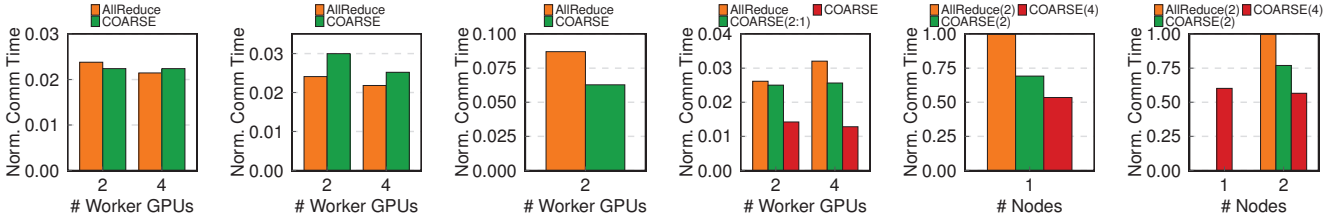
Figure 16(a-b) show the evaluation on AWS with NVIDIA T4 GPUs. COARSE achieves 3.3× and 4.3× speedup for ResNet50, and 11.3× and 13.3× speedup for BERT, which is slightly lower than AllReduce. COARSE does not work efficiently on this platform because there's no unbalanced bandwidth (Figure 15a) and this platform does not support GPU p2p communication.

Figure 16c shows the evaluation on SDSC with P100 GPUs, where COARSE achieves 3.4× speedup for BERT. On the AWS machine with V100 GPUs (Figure 16d), COARSE achieves 10.8×-13.8× speedup for BERT. On



(a) T4 - ResNet50 (b) T4 - BERT (c) P100 - BERT (d) V100 - BERT (e) V100 - BERT (f) V100 - BERT

Figure 16. DL training on (a-b) AWS g4dn instance with T4 GPUs, (c) SDSC machine with P100 GPUs, (d) AWS p3 instance with V100 GPUs, (e) AWS single node with V100, (f) AWS multi nodes with V100. In (d) *COARSE(2:1)*, each memory device is shared by two worker GPUs. In (e-f), the baseline is *AllReduce* which is equivalent to AllReduce in (d); the numbers in legends represent training batch sizes.



(a) T4 - ResNet50 (b) T4 - BERT (c) P100 - BERT (d) V100 - BERT (e) V100 - BERT (f) V100 - BERT

Figure 17. Blocked communication time in DL training. In (a-d) numbers are normalized to the CCI case; in (e-f) numbers are normalized to the AllReduce case. The CCI bar is ignored for better comparison between AllReduce and COARSE.

these two machines, COARSE beats AllReduce by leveraging non-uniform bidirectional bandwidth.

Figure 16(e-f) show single- and multi-node training performance. The baseline is set to *AllReduce* because DENSE does not assume a multi-node system. On the single node system (Figure 16e), AllReduce can only use a batch size of 2 due to memory capacity limitation, while COARSE can use a batch size of 4 and achieves 48.3% faster training. On the multi-node system (Figure 16f), COARSE is up to 42.7% faster than AllReduce. Even a single node COARSE with a large batch size of 4 can achieve 38.6% speedup compared to a two-node AllReduce training.

E. Communication Time

To understand the speedups in DL training, we break down the training time and measure the communication time that blocks the training computation. Figure 17 shows the result of blocked communication time where numbers are normalized to (1) CCI communication time in single-node evaluations (Figure 17(a-d)), and (2) AllReduce in multi-node evaluations (Figure 17(e-f)). AllReduce and COARSE reduce the communication to less than 10% compared to the naïve parameter server running on CCI memory. AllReduce achieves this speedup by using GPUs without external parameter storage. While COARSE provides additional parameter storage through CCI memory, with comparable or even better performance.

On AWS machine with T4 GPUs (Figure 17b), COARSE introduces additional 18%-20% blocked communication for BERT model. This is because COARSE exploits non-uniform bandwidth for faster communication, but this machine does not have such a bandwidth feature. On SDSC machine with P100 GPUs (Figure 17c), COARSE reduces the communication time by 28% for BERT. On AWS machine with V100 GPUs (Figure 17d), COARSE reduces the blocked communication time by 20%-42% for BERT. On single node and dual node AWS machines with V100 GPUs (Figure 17(e-f)), COARSE reduces the communication time by 23%-46%. These results show that COARSE is highly efficient in reducing communication overhead in distributed training.

VI. Related Work

To our knowledge, this is the first paper to explore the cache-coherent disaggregated memory system to accelerate communication in DL training. This section discusses previous works related to our study.

Parameter Server. Li et al. [32] propose a distributed parameter server design based on Ethernet, while COARSE leverages cache-coherent interconnection. GeePS [8] is a GPU-specialized parameter server which allows the parameter server to store part of the parameters on GPU memory, so that parameter push/pull can be processed faster. GeePS enables the multi-server-node training but limits to one GPU per node, while COARSE works with multiple GPUs per node. ParameterHub [37] proposes a parameter server with software-hardware co-design to optimize the rack-scale Ethernet latency, while COARSE reduces serial bus latency using cache-coherent interconnection.

Speedup Communication in DL Training. Hop [39] is a queue-based parameter synchronization in decentralized DL training. Hop leverages bounded staleness training,

while COARSE targets synchronized training and thus is orthogonal with this work. Prague [38] proposes a group-based AllReduce synchronization scheme for distributed DL training. Prague uses partial synchronization to achieve high performance, while COARSE provides full synchronization without losing precision. Zhao et al. [63] propose a hierarchical multi-level parameter server design with GPU memory, CPU memory, SSD, and network. This parameter server is helpful in recommendation systems where most parameters are sparse. It relies on CPU-GPU communication to exchange parameters, while COARSE enables GPU direct access to CCI memory device to exploit full serial bus bandwidth. ByteScheduler [54] introduces a communication scheduler for DL training. It is beneficial when using low bandwidth Ethernet communication, but provides less speedup when using intra-node serial buses for communication. Compared to ByteScheduler, COARSE leverages CCI disaggregated memory to speed up the communication on fast serial buses and support large DL models. Blink [62] is a collective communication scheme optimized for GPUs interconnected by both NVLink and PCIe. It runs parameter synchronization jobs on GPUs while COARSE offloads these jobs to CCI memory devices to further improve GPU utilization. Klenk et al. [26] propose an in-network accelerator for collective communication. This work does not support extended memory space for GPUs, while COARSE leverages CCI memory devices to enable larger models to be trained. In addition, COARSE exploits the non-uniform serial bus bandwidth to further improve the performance. As a result, COARSE achieves 48.3% BERT training speedup over AllReduce, while this prior work achieves less than 10% speedup [26] in the same model training.

Disaggregated Memory Lim et al. [34], [35] propose a memory disaggregation architecture with a memory blade connected over PCIe to expand CPU accessible memory. COARSE takes one step further to explore the potential of disaggregated memory using CCI. Kim et al. [25] propose a memory network for GPU and CPU using packet routing from HMCs, while COARSE does not rely on specific memory media technology. Kwon et al. [28] propose a memory-centric architecture for distributed DL training. This work assumes unified communication bandwidth in serial bus, while COARSE exploits the non-uniform bandwidth.

Near Memory Processing for DL Training TensorDIMM [27] provides a near memory processing design to offload embedding operations in recommendation systems. It accelerates the specialized operations while COARSE provides generic acceleration to parameter synchronization operations. iSwitch [33] proposes a parameter synchronization design in Ethernet switch memory. It is beneficial to reinforcement learning where model size is small enough to fit in switch memory, while COARSE targets large models and provides extended parameter storage.

VII. Conclusion

In this paper, we propose COARSE, a distributed parameter synchronization scheme based on disaggregated memory, for distributed DL training. COARSE exploits the non-uniform serial bus bandwidth and bidirectional bandwidth to accelerate the parameter synchronization in DL training. It leverages disaggregated memory to offload the parameter synchronization jobs and hence improves the GPU utilization. It combines the emerging cache-coherent interconnection with MPI-like collective communication to provide low-latency parameter synchronization. Our evaluation shows COARSE significantly speeds up the DL training compared to the centralized parameter synchronization design with CCI.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This paper is supported in part by SK hynix and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

References

[1] 250-SoC, "BittWare 250-SoC." [Online]. Available: https://www.bittware.com/fpga/250-soc/

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, Nov. 2016, pp. 265–283.

[3] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, Oct. 2014, pp. 571–582.

[4] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, Big, Simple Neural Nets for Handwritten Digit Recognition," *Neural Computation*, vol. 22, no. 12, pp. 3207–3220, Dec. 2010.

[5] Cirrascale, "Cirrascale SR3514: Unexpected performance inequality. Technical brief M901A-092014," 2018.

[6] C. Consortium, "CCIX." [Online]. Available: https://www.ccixconsortium.com/

[7] C. Consortium, "Compute Express Link." [Online]. Available: https://www.computeexpresslink.org/

[8] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16*. London, United Kingdom: ACM Press, 2016, pp. 1–16.

[9] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231.

[10] 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 *NIPS*, 2012, pp. 1223–1231.

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *ArXiv:1810.04805*, 2018.

[13] Gen-Z Consortium, "The Gen-Z Consortium." [Online]. Available: https://genzconsortium.org/

[14] Google, "Cloud TPU." [Online]. Available: https://cloud.google.com/tpu

[15] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2018, pp. 620–629.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," in *Computer Vision – ECCV 2016*, ser. Lecture Notes in Computer Science, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 630–645.

[17] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, "Deep Learning Scaling is Predictable, Empirically," *arXiv:1712.00409 [cs, stat]*, Dec. 2017.

[18] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, Nov. 2012.

[19] X. Huang, J. Baker, and R. Reddy, "A historical perspective of speech recognition," *Communications of the ACM*, vol. 57, no. 1, pp. 94–103, Jan. 2014.

[20] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism," *arXiv:1811.06965 [cs]*, Jul. 2019.

[21] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," *arXiv:1807.05358 [cs]*, Jul. 2018.

[22] M. Johnson, M. Schuster, Q. V. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viégas, M. Wattenberg, G. Corrado, M. Hughes, and J. Dean, "Google's multilingual neural machine translation system: Enabling zero-shot translation," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 339–351, 2017.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 1–12, Jun. 2017.

[24] X. KCU1500, "BittWare 250-SoC." [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/kcu1500/ug1260-kcu1500-data-center.pdf

[25] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-GPU System Design with Memory Networks," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp. 484–495.

[26] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, May 2020, pp. 996–1009.

[27] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, Oct. 2019, pp. 740–753.

[28] Y. Kwon and M. Rhu, "Beyond the Memory Wall: A Case for Memory-Centric HPC System for Deep Learning," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2018, pp. 148–161.

[29] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[30] N. Ledentsov Jr, M. Agustin, L. Chorchos, N. Ledentsov, and J. Turkiewicz, "25.78 gbit/s data transmission over 2 km multi-mode-fibre with 850 and 910 nm single-mode vcsels and a commercial quad small form-factor pluggable transceiver," *Electronics Letters*, vol. 54, no. 12, pp. 774–775, 2018.

[31] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, Jan. 2020.

[32] 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 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598.

[33] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 279–291.

[34] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, Jun. 2009, pp. 267–278.

[35] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *IEEE International Symposium on High-Performance Comp Architecture*, Feb. 2012, pp. 1–12.

[36] Linux Programmer's Manual, "mmap(2) — Linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man2/mmap.2.html

[37] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter Hub: A Rack-Scale Parameter Server for Distributed Deep Neural Network Training," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 41–54.

[38] Q. Luo, J. He, Y. Zhuo, and X. Qian, "Prague: High-Performance Heterogeneity-Aware Asynchronous Decentralized Training," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, Mar. 2020, pp. 401–416.

[39] Q. Luo, J. Lin, Y. Zhuo, and X. Qian, "Hop: Heterogeneity-aware Decentralized Training," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 893–907.

[40] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=Hkc-TeZ0W

[41] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. Sydney, NSW, Australia: JMLR.org, Aug. 2017, pp. 2430–2439.

[42] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–15. [Online]. Available: https://doi.org/10.1145/3341301.3359646

[43] NVIDIA, "GPUDirect." [Online]. Available: https://developer.nvidia.com/gpudirect

[44] NVIDIA, "NVIDIA A100 TENSOR CORE GPU." [Online]. Available: https://www.nvidia.com/en-us/data-center/a100/

[45] NVIDIA, "NVIDIA Collective Communications Library (NCCL)." [Online]. Available: https://developer.nvidia.com/nccl

[46] NVIDIA, "NVIDIA Deep Learning Examples for Tensor Cores." [Online]. Available: https://github.com/NVIDIA/DeepLearningExamples

[47] NVIDIA, "NVIDIA DGX-1 with Tesla V100 System Architecture White paper." [Online]. Available: https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf

[48] NVIDIA, "NVLINK AND NVSWITCH." [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[49] NVIDIA, "Profiler :: CUDA Toolkit Documentation." [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html

[50] J. H. Park, G. Yun, C. M. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y. ri Choi, "Hetpipe: Enabling large DNN training on (whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 307–321. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/park

[51] PCI-SIG, "PCI Express Base Specification Revision 3.1a." [Online]. Available: https://pcisig.com/specifications

[52] PCI-SIG, "PCI Express Base Specification Revision 4.0, Version 1.0." [Online]. Available: https://pcisig.com/specifications

[53] PCI-SIG, "PCI Express Base Specification Revision 5.0, Version 1.0." [Online]. Available: https://pcisig.com/specifications

[54] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed DNN training acceleration," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, Oct. 2019, pp. 16–29.

[55] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100,000+ Questions for Machine Comprehension of Text," Jun. 2016.

[56] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[57] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *CoRR*, vol. abs/1802.05799, 2018. [Online]. Available: http://arxiv.org/abs/1802.05799

[58] A. W. Services, "Amazon ec2 instance types." [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[59] M. Smelyanskiy, "Zion: Facebook next-generation large memory training platform," in *Hot Chips*, 2019, pp. 1–2.

[60] TensorFlow, "TensorFlow benchmarks." [Online]. Available: https://github.com/tensorflow/benchmarks

[61] UC San Diego, "San Diego Supercomputer Center." [Online]. Available: https://www.sdsc.edu/

[62] G. Wang, S. Venkataraman, A. Phanishayee, J. Thelin, N. Devanur, and I. Stoica, "Blink: Fast and Generic Collectives for Distributed ML," *arXiv:1910.04940 [cs]*, Oct. 2019.

[63] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems," *arXiv:2003.05622 [cs, stat]*, Mar. 2020.

[64] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *arXiv preprint arXiv:2003.05622*, 2020.