2017 IEEE 23rd International Conference on Parallel and Distributed Systems

PARALLEL I/O OPTIMIZATIONS FOR SCALABLE DEEP LEARNING

Sarunya Pumma,* Min Si,† Wu-chun Feng,* and Pavan Balaji†
*Virginia Tech, USA; {sarunya, wfeng}@vt.edu
†Argonne National Laboratory, USA; {msi, balaji}@anl.gov

***Abstract*—As deep learning systems continue to grow in importance, researchers have been analyzing approaches to make such systems efficient and scalable on high-performance computing platforms. As computational parallelism increases, however, data I/O becomes the major bottleneck limiting the overall system scalability. In this paper, we continue our efforts to improve LMDB, the I/O subsystem of the Caffe deep learning framework. In a previous paper we presented LMDBIO—an optimized I/O plugin for Caffe that takes into account the data access pattern of Caffe in order to vastly improve I/O performance. Nevertheless, LMDBIO's optimizations, which we henceforth call LMM (localized `mmap`), are limited to intranode performance, and these optimizations do little to minimize the I/O inefficiencies in distributed-memory environments. In this paper, we propose LMDBIO-DM, an enhanced version of LMDBIO-LMM that optimizes the I/O access of Caffe in distributed-memory environments. We present several sophisticated data I/O techniques that allow for significant improvement in such environments. Our experimental results show that LMDBIO-DM can improve the overall execution time of Caffe by more than 30-fold compared with LMDB and by 2-fold compared with LMDBIO-LMM.**

I. INTRODUCTION

Deep learning is one of the key technologies used today to analyze and characterize large volumes of data. Because of the computational and memory complexity of training a deep neural network (DNN), several parallel deep learning toolkits have been proposed. For instance, Caffe [7] is a well-known deep learning framework that has multiple parallel implementations [9], [1], [3]. As these frameworks continue to explore the limits of parallelism and scalability, they have started utilizing large supercomputing systems and highly efficient computational units such as NVIDIA GPUs, Intel Xeon Phi, or Google TPU processors[1] to improve their computational efficiency. Such improvement in the computational framework has, however, exposed new bottlenecks in their I/O subsystem.

In previous work [12], we showed that even with a small amount of asynchrony in the network processing, I/O consumes a dominant fraction of the overall execution time, thus limiting the overall system scalability. In fact, for some of the datasets that we used [8], [5], I/O can take up to 70–80% of the overall execution time. We then analyzed this performance issue in Caffe's I/O subsystem, Lightning Memory-mapped Database (LMDB), and proposed a new optimized I/O plugin for Caffe, called LMDBIO. LMDBIO continues to function on unmodified LMDB database files, which are predominant in the deep learning community; but it significantly improves I/O performance by taking into account the data access pattern of parallel deep learning frameworks.

[1]https://en.wikipedia.org/wiki/Tensor_processing_unit

Despite this improvement, however, LMDBIO's optimizations, which we henceforth refer to as LMM (localized `mmap`), are limited to intranode performance. The primary goal in its initial design was to understand the interprocess contention that occurs when multiple processes on the same node use LMDB simultaneously. This design, however, does little to minimize the I/O inefficiencies in distributed-memory environments.

In this paper, we present LMDBIO-DM, an enhanced version of LMDBIO-LMM that optimizes the I/O access of Caffe in a distributed-memory environment. We first present a detailed analysis of the I/O issues in Caffe/LMDB that continue to exist even with Caffe/LMDBIO-LMM. Specifically, LMDB databases use B+ trees to lay out the databases in memory in such a way that they can be accessed from a filesystem efficiently; but this database format inherently relies on information being accessed sequentially in order to parse through the overall database. Thus, while efficient for sequential access, it can be challenging when multiple processes are trying to read data in parallel; and it results in a significant amount of redundant data I/O across different processes.

LMDBIO-DM takes advantage of this analysis and uses sophisticated data I/O techniques to work around such shortcomings in the LMDB database format. We first present a technique that memory-maps the database into a symmetric address space on each process, thus allowing for database position information to be portably exchanged across different processes in a distributed-memory environment. This first technique minimizes the amount of redundant data that is read from the filesystem, although it does so at the cost of I/O serialization across processes. We then present a second technique that allows for speculative parallel I/O to efficiently fetch data from the database file into memory. That is, it attempts to estimate the start and end location of the part of the database that each process needs to access and tries to speculatively fetch those bytes into memory: each process does this part in parallel. This second technique allows for most data access to be performed in parallel while increasing the amount of redundant data I/O by only a very small amount compared with that from the fully serialized technique.

We also present and analyze experimental results that showcase the improvements of LMDBIO-DM compared with LMDB and LMDBIO-LMM. The results show that LMDBIO-DM can improve the overall execution time of Caffe by more than 30-fold compared with LMDB and by 2-fold compared with LMDBIO-LMM.

The rest of the paper is organized as follows. Section II presents an overview of Caffe and the LMDB database format

978-1-5386-2129-5/17/31.00 ©2017 IEEE
DOI 10.1109/ICPADS.2017.00097

as background for our subsequent discussion. Section III describes the LMDBIO-LMM framework that we use as the starting point for the enhancements proposed in this paper. Section IV provides a detailed analysis of the I/O shortcomings in Caffe/LMDB that continue to exist even with Caffe/LMDBIO-LMM. Section V describes LMDBIO-DM and how it addresses the shortcomings of both LMDB and LMDBIO-LMM. Experimental results comparing Caffe/LMDB, Caffe/LMDBIO-LMM, and Caffe/LMDBIO-DM are presented in Section VI. Related work is discussed in Section VII, and concluding remarks are presented in Section VIII.

II. Background

In this section, we present an overview of the Caffe deep learning tool and the LMDB database format.

A. Caffe Overview

The Caffe framework was developed by the Berkeley Vision and Learning Center as a GPU-based implementation of convolutional neural network training. It was written in C++ with CUDA for highly optimized GPU computation. Subsequent variants of Caffe, however, have included support for generic CPU architectures as well.

The Caffe framework generally works as follows. By default it starts with a randomized "guess" about the parameters of the network that it intends to train. Once the network is initialized with the guessed parameters, data samples from the training dataset are read and processed by the network. This processing allows Caffe to measure the deviation error in the initial guess with respect to what classification the network predicted and what the actual classification is. Caffe then uses this deviation error to improve its guess about the network parameters.

Given enough high-quality training data samples, Caffe will eventually converge to the desired accuracy. The final set of network parameters can then be used to generate a mathematical equation that can be utilized for highly accurate classification of new data samples. The key to generating a highly accurate classification equation is the use of a very large set of (high-quality) training data samples. Thus, large organizations commonly train their DNN systems with several hundreds of terabytes or even petabytes of data. Consequently, both accessing such data and processing it must be fast if DNN training is to be practical.

Sequential processing of each data sample in the training dataset is the most conservative approach for training the network. This model, however, is overly serial and generally not useful in practice. Most modern deep learning frameworks allow for some asynchrony in network training either by partitioning the data samples across processes/threads (e.g., Caffe [9], [1], [3]) or by partitioning the network across processes/threads (e.g., TensorFlow [2]). This technique is called *batch* training and is, generally speaking, a method for simultaneously processing multiple data samples before updating the network. Processing one batch of data samples is referred to as one training iteration. Such processing is repeated for a very large number of iterations, making the training process largely bulk synchronous, where parallelism is utilized within each iteration but all processes need to synchronize at the end of each iteration in order to update their network parameters.

An important aspect that is also being carefully studied in the community is the impact of the batch size on the convergence rate. Loosely speaking, the larger the batch size, the fewer the parameter updates, but also the higher the number of iterations needed for convergence. With better preconditioned network parameters and other similar techniques, however, researchers have been increasing the optimal batch size in this performance-accuracy tradeoff.

In each training iteration, each worker reads a subset of a batch from a database. Caffe provides a variety of data reading options via several types of databases. The default and the most widely used option is the LMDB format.

B. LMDB Database Format

LMDB refers to both the format of the database and the corresponding software library. In this section, we discuss the database format. The intent of the LMDB database format is to provide a fast access method for databases with support for multiversion concurrency control, fast disk I/O, and various other such features. To this end, LMDB adopts a flattened B+ tree data structure to store its data.

B+ trees are balanced *n-way* search trees. LMDB uses this tree format to organize the database indices in a way that data records can be accessed efficiently when stored on a local or external filesystem. Generally speaking, a B+ tree consists of two types of nodes: *branch nodes* and *leaf nodes* (see Figure 1 for a 3-way B+ tree structure). A branch node contains pointers that point to n children nodes (which can be branch nodes or leaf nodes). Indices contained in a branch node govern the range of indices of its successors. For example, in Figure 1, the pointer from index 3 in the branch node points to a leaf node that contains data with indices less than or equal to 3. B+ trees are designed to be efficient for filesystem access. In B+ trees, nodes are stored in a block-aware manner, where each node is a filesystem page.

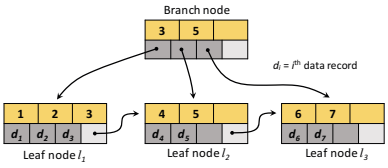

Fig. 1. B+ tree data structure

The database layout in LMDB is made up of four types of pages: *metadata pages*, *branch pages*, *leaf pages*, and *overflow pages*. Metadata pages store information about the overall database (e.g., version of the database, size of the database). The branch and leaf pages represent the core branch and leaf nodes in the graph, respectively. These match the generic B+ tree structure described above. Each branch and leaf page keeps aside some part of the page to store a page header and

uses the rest of the page to store the actual database record. To accommodate cases where the record is larger than what can fit into the leaf page, LMDB uses overflow pages. Overflow pages store the part of the record that could not fit into the leaf page. Each leaf page can be associated with zero or more overflow pages. In LMDB, only the first overflow page associated with the leaf page has a header that contains information indicating the type of each page, the size of the node, and pointers to its children and/or a neighboring node.

Since LMDB's data format is not a simple contiguous set of raw data samples, but rather a nontrivial tree structure, the location index pointing to a specific record within the tree is more complex than a generic pointer in C++. That is, one cannot simply store the record virtual memory address but also needs information about the parent branch nodes and other related information in order to fully navigate through the tree structure. For this purpose, LMDB provides a pointer structure called a "cursor." A cursor contains information about the record index, virtual address location, parent branch nodes, and offset to the page address holding the record (for cases where a page holds multiple records) and can be considered to be the complete signature of a particular record inside the LMDB database. The user can move the cursor inside the database by using LMDB-provided cursor operations. None of the existing operations, however, allow for random access within the database. In other words, LMDB allows only *sequential database access* where the cursor can be moved to an adjacent data record. To access a random data record (i.e., a leaf node) in the database, LMDB needs to sequentially scan the header of *every branch node* ahead of the target leaf page in order to determine a location to shift the cursor to. To move from a branch node to its successor or from a leaf node to another leaf node, LMDB acquires the pointer of the target node from the header. Before moving to the target node, LMDB stores the entire header of the current page in a stack that is a part of the cursor's data structure, in order to allow for a convenient traceback.

III. Overview of LMDBIO-LMM

In this section we present a brief overview of LMDBIO-LMM. A more detailed description can be found in our previous work [12]. In Section III-A we briefly analyze the intranode I/O issue with Caffe/LMDB, followed in Section III-B by the overall design and implementation of LMDBIO-LMM.

A. LMDB, `Mmap`, and completely fair scheduler

Caffe uses the LMDB database format as its default dataset storage mechanism. It maps the database file into memory in order to enable efficient and rapid data batch retrieval by using the LMDB library. Prior to training, the database file is mapped from the filesystem to the virtual address space of a process, thus providing access to the file as if it were a memory buffer. Internally, a system call, `mmap`, is used. With `mmap`, data is fetched to physical memory and mapped to the corresponding virtual address space of the process dynamically only when a required part of the file is accessed.

The way `mmap` handles I/O requests is inefficient, however, since it relies on the *Completely Fair Scheduler* (CFS) and an I/O interrupt handler. When a process accesses an `mmap` buffer and the associated page is not present in memory, the data will be fetched from the filesystem to memory by a fault handler. The I/O request is issued by the hardware controller (e.g., SCSI for local storage or a network adapter for network-based filesystems). Since an I/O request can take a long time to complete, the user process goes to sleep while waiting for the I/O. The hardware controller raises an interrupt informing the filesystem once the I/O operation completes. One important aspect to note here is that this interrupt handler is a *bottom-half* handler in Linux. That is, the interrupt is not associated with any particular user process in the system. In other words, it is a generic event that informs the filesystem that an I/O operation that was issued by one or more processes has completed. Therefore, all processes that were sleeping while waiting for an I/O event will be marked as *runnable* each time the interrupt occurs.

At this point, the runnable processes can be scheduled by the Linux default process scheduler (i.e., CFS). Once the scheduler is triggered, each runnable process in the CFS red-black tree will be woken up to continue its execution. In the CFS red-black tree, processes are ordered based on their CPU usage. A process with the least-used CPU time will be the leftmost leaf node of the tree, where it will be chosen to run first. Suppose that one I/O operation has completed and that more than one process is waiting for I/O operations; then, only one process is able to continue its processing while others are woken up, realize that their I/O operations have not yet completed, and go to sleep again. This model significantly increases the number of context switches that get triggered, with most of the switches resulting in no real work. It also increases the amount of "sleep time" associated with each process.

B. Design and Implementation of LMDBIO-LMM

In our previous work, the primary design goal of LMDBIO-LMM was to minimize interprocess contention within a node. We called our approach "localized `mmap`." In this approach, LMDBIO-LMM chooses a single process on each node as the root to perform data reading from the filesystem. Once the root process finishes the reading, it shares the data among other processes on the same node by using MPI-3 shared memory. Since only one process is performing I/O on each node, the I/O bottom-half handler knows exactly which process in the red-black tree is to be marked as runnable. Our approach can significantly reduce the number of context switches and improve the data reading performance in Caffe when using `mmap`.

LMDBIO-LMM consists of two phases: an initialization phase and a data-reading phase. LMDBIO-LMM automatically assigns one reader per node in the initialization phase by using MPI-3 to split a global MPI communicator into multiple local "shared-memory" communicators where all processes in the same node are presumably grouped to the same communicator. After the reader assignment is done, each reader opens the

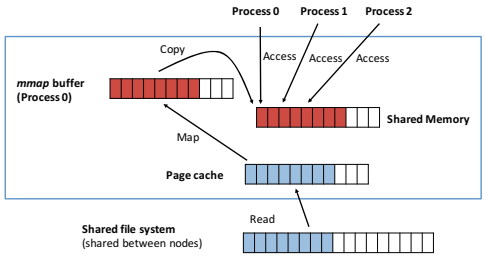

Fig. 2. LMDBIO-LMM overview

LMDB database, which internally maps the database file to its virtual address space using `mmap`. In this step, all processes on the node also preallocate a shared-memory buffer that they can all directly access.

In the data-reading phase (shown in Figure 2), each reader in LMDBIO-LMM (one process per node) reads the data samples from the filesystem to page cache. The data is then mapped to the address of the `mmap` buffer of each reader. Once the data in the buffer becomes available, the reader process copies the data to the shared-memory buffer that every process allocated during the initialization phase. LMDBIO-LMM synchronizes the processes within the local communicator to ensure that the reader has finished writing to the shared-memory buffer before other processes can access it.

C. LMDBIO-LMM Performance

We showcase here key performance results demonstrating the performance capabilities of LMDBIO-LMM. Information about the experimental testbed is provided in Section VI-A.

Figure 3 compares the performance of Caffe/LMDB with that of Caffe/LMDBIO-LMM. We notice that Caffe/LMDBIO-LMM performs better than Caffe/LMDB by up to a factor of 20-fold in some cases. The primary reason is the reduced number of context switches in Caffe/LMDBIO-LMM compared with Caffe/LMDB, where we observed close to a 700-fold improvement. Since LMDBIO-LMM has a single process performing `mmap`, it ensures that no contention occurs between `mmap` calls performed by multiple processes. This serialization reduces the number of unnecessary wakeups created by the interrupt handler, thus reducing the number of context switches.

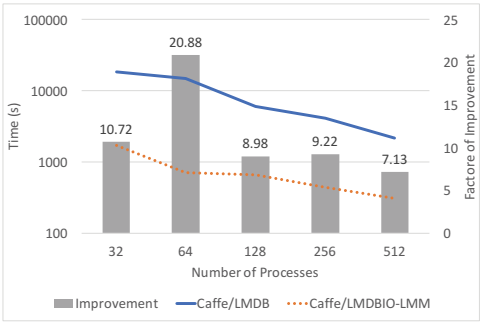

Fig. 3. Comparison between Caffe/LMDB and Caffe/LMDBIO-LMM by using the ImageNet dataset

IV. Analysis of I/O in Caffe

In this section, we analyze the I/O characteristics of Caffe/LMDBIO-LMM. In Section IV-A, we analyze the overall performance of Caffe/LMDBIO-LMM and showcase its quantitative I/O performance issues. In Section IV-B, we discuss aspects of the internal data format of LMDB databases that cause Caffe/LMDB to perform additional unnecessary I/O.

A. Caffe/LMDBIO-LMM: Performance Analysis

To analyze Caffe/LMDBIO-LMM's scalability, we train the CIFAR10-Large dataset using the AlexNet DNN model. Dataset and testbed details are provided in Section VI-A.

We first consider the overall execution time scalability (strong scaling) of Caffe/LMDBIO-LMM compared with ideal scaling. Figure 4(a) shows that the actual training time starts to differ from the ideal scaling time after just four processes and that the difference increases with the number of processes. In fact, with just 512 processes, the performance of Caffe/LMDBIO-LMM is nearly 17-fold worse than the ideal scaling performance. To understand this result better, we analyzed the time taken by the various components of Caffe/LMDBIO-LMM. Figure 4(b) shows that the data I/O time (represented as "Read time") becomes highly significant when training a network on a large number of processes. It takes approximately 40% of the overall training time when using 512 processes and tends to increase when using a larger number of processes. Further, the skew between different processes (represented as "Waiting time before param sync" in the figure) continues to grow with increasing numbers of processes and takes nearly 60% of the overall training time when using 512 processes.

These two portions of time are interrelated. Since the computation is similar for all the processes, both the large read time and the large skew time are contributed by issues in the I/O subsystem, thus making data I/O the primary bottleneck in the overall execution.

B. Caffe/LMDBIO-LMM: Redundant Data Movement

As mentioned in Section II-B, random accesses are not allowed in LMDB. To access a data record, LMDB needs to start from the root node of the B+ tree and parse through every branch node in the path to the target data record. We refer to this operation as the "LMDB seek" operation, although unlike a traditional UNIX seek operation, it is not possible to directly jump to an arbitrary page without a risk of accidentally reaching an overflow page that contains no information of how to go to the next or leaf node.

While traversing through the tree nodes, the header on each node is read to obtain a pointer to the next record location. To do so, the page containing the header needs to be loaded into memory. Since the header itself is much smaller than the physical page size, the header page usually contains additional information that needs to be loaded into memory even when it does not need to be accessed. This data-reading model is troublesome for parallel I/O because processes have to access different parts of the database file, resulting in a

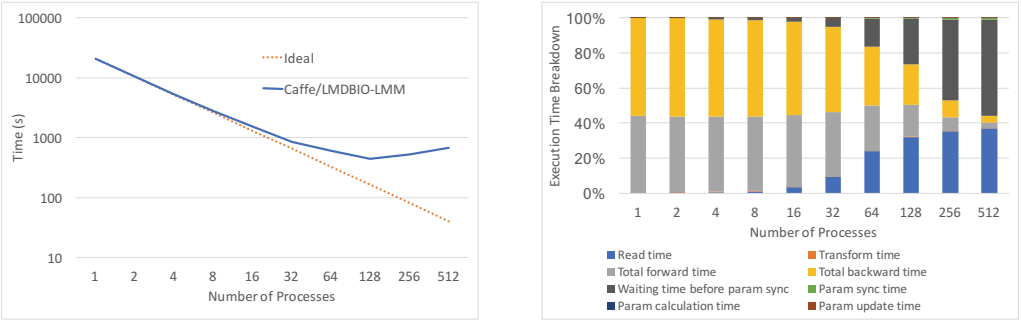

Fig. 4. CIFAR10-Large dataset: (a) performance comparison of ideal scaling and Caffe/LMDBIO-LMM; (b) performance breakdown of Caffe/LMDBIO-LMM

semirandom data access pattern. That is, each process needs to start at a position in the database that cannot be precomputed and requires information from the previous data records to compute.

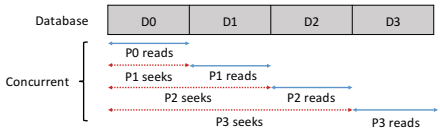

Fig. 5. LMDB redundant data movement

The data access pattern in LMDBIO-LMM is illustrated in Figure 5. Suppose four readers ($P0$ - $P3$) need to read a different portion of the database ($D0$ - $D3$) from the filesystem to memory. When $P0$ reads $D0$, it reads both the headers and the actual content. In this case, $P0$ does not read any extra data. In order to read $D1$, however, $P1$ has to seek through all of the branch nodes in the $D0$ portion of the database before it gets to the $D1$ portion. From the figure, we notice that the amount of extra data read increases with the process count, where in this case $P3$ reads the most extra data. With this data access model, in the worst case a process could end up reading a total of $R \times B$ bytes, where R is a total number of readers and B is a size of an individual data portion.

Besides fetching redundant data from the filesystem, this model causes skew in data I/O because different processes do different amounts of work. Such a load imbalance can cause processes to stay idle at a process synchronization point (e.g., parameter synchronization in Caffe) waiting for the last process to finish its task. This can severely degrade the overall progress of a parallel application.

V. LMDBIO-DM: Design and Implementation

In this section, we present details of the design and implementation of LMDBIO-DM. The LMDBIO-DM software itself is an extension of the original LMDBIO-LMM software and has been developed on top of the same code base. LMDBIO-LMM is a C++ parallel I/O library that utilizes MPI and LMDB as core engines. LMDBIO-LMM requires MPI-3[2] in order to automatically determine process colocation, perform reader assignment, and share data efficiently via a shared-memory buffer. Since LMDB is highly optimized for efficient in-memory database access, LMDBIO-LMM adopts the same API to map the database from the filesystem to memory and access data from there.

As discussed in Section IV, one of the primary reasons for the performance loss in data I/O with Caffe/LMDB and Caffe/LMDBIO-LMM is the redundant data I/O by the different processes. To solve this issue, we propose a two-step approach. In the first step, described in Section V-A, we present an approach where each process reads exactly the data that it needs to process, although it does so by serializing I/O across the different processes. In the second step, described in Section V-B, we present an approach for estimating what data pages each process will eventually need and speculatively performing parallel I/O to regain most of the performance lost because of the I/O serialization described in the first step.

A. Serializing I/O Using a Portable Cursor Representation

Here, our goal is to ensure that each process reads only the data that it needs to process. In other words, no additional data is read at seek time. To do so, each process must first read the data that it needs to process and then pass to the next process the information about the location where it stopped. The general model we want to follow is illustrated in Figure 6. In the figure, $P1$ cannot start reading data $D1$ until $P0$ finishes reading $D0$ and sends the starting point of $D1$ (i.e., the cursor) to it. Executing this in practice, however, has a few complications that we discuss in this section.

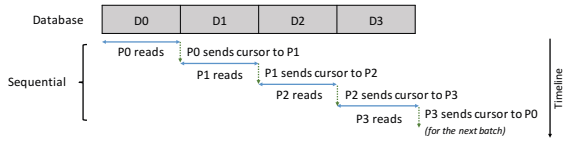

Fig. 6. LMDBIO-DM design: sequential I/O and cursor handoff

As described in Section II-B, since LMDB uses a B+ tree to represent its data elements, the position indicator for a

[2]Most supercomputers already support MPI-3. The notable exception to this claim is the IBM Blue Gene series of supercomputers that do not yet support MPI-3. However, these supercomputers are nearing their end of life; and the next generation of supercomputers from IBM do plan to support MPI-3 and later MPI standards.

record within the B+ tree is not a simple offset from the start of the file, but rather a more complex data structure which LMDB refers to as a cursor. The cursor obviously includes information about the record it points to. But it also includes other information such as the path of the record's parent branch nodes, a pointer to the page header containing the record, and information about the access flags of the particular record being pointed to. Unfortunately, the cursor data structure itself is not portable across different processes since it contains information represented as pointers within the B+ tree that is relevant only within the virtual address space of the original process that created the cursor. Luckily, all the pointers contained within this structure point to locations within the B+ tree.

In order to serialize the cursor into a format that is portable across different processes, the simplest model that we envision is that of a symmetric address space. That is, if we can ensure that all processes can memory-map the database into exactly the same virtual address location on all processes, any pointers that point to locations within the B+ tree would be portable across the different processes, thus making it possible to serialize the cursor to a portable format. To achieve this, we use the following algorithm. The first reader process randomly picks a virtual address location from its 64-bit address space and tries to memory-map the database to this memory location. If it is successful, it broadcasts this address to the remaining reader processes. Each of the remaining reader processes tries to memory-map the database file at the exact same memory location. Each process indicates whether it was successful or not within an MPI allreduce operation where all processes try to come to a consensus. If everyone was successful, the database is now mapped to the same virtual address location on all processes. If at least one of the processes was not successful, all processes unmap their database and try again. This process is repeated for a few iterations.

In theory, it is possible to find no virtual address location that can be symmetrically used across all processes. However, given that most of the 64-bit address space is typically unused on any given process, in practice we can find a symmetrical address space in 1–2 attempts with the algorithm described above. In the worst case, if we are not able to find a symmetrical address space after a few attempts, we abandon this optimization and fall back to the approach used by the original LMDBIO-LMM.

Once the database is mapped to the symmetrical address space, the actual serialization of the cursor itself is mostly trivial. The internal content of the cursor data structure is copied into a memory buffer that can be sent to the other processes by using MPI send/recv.

B. Speculative Parallel I/O

The first step of our algorithm, discussed in Section V-A, provides a portable solution to pass the location information within the database to other processes. However, the approach described there comes at the cost of serialization in data I/O. That is, only one process is actively reading data at any given point of time. This is inefficient on most parallel filesystems where multiple processes need to be performing I/O in order to achieve the best performance.

Here, we discuss the second step of our algorithm that tries to estimate what data needs to be processed by a given process and speculatively performs parallel I/O on that data (illustrated in Figure 7). To do this, we must first estimate what part of the database we need to fetch to memory. This is a complex task since the structure of the B+ tree is not always straightforward. Depending on how many branch nodes are used and how many records each branch node points to, estimating which physical pages each process would need to access is nontrivial.

In our approach, we assume that the sizes of all records are roughly the same, which is a fairly safe assumption to make for most deep learning frameworks because of the way the input data samples are handled. Each reader process reads the first data record in the database file to retrieve the record's size. The readers use the obtained size information along with the number of records that they will read (i.e., a fraction of the batch) to estimate the number of pages to be fetched. For instance, the size of each sample in the CIFAR10-Large dataset is approximately 3 KB. For I/O efficiency reasons, LMDB pads the data to ensure that each record occupies one page (4 KB). Therefore, the number of speculative pages for n data records is $n \times 1$ pages. The read offset of each reader is calculated in the same fashion. Specifically, each reader process maintains a minimum and maximum count of the number of pages that each reader process needs to access. If the reader process ends up speculatively reading too far ahead, it corrects its estimate of the maximum number of pages that the previous readers need. Similarly, if the reader process ends up speculatively reading too far behind, it corrects its estimate of the minimum number of pages that the previous readers need. The actual pages that the reader process speculatively reads includes all pages between the minimum and the maximum estimate boundaries. We expect that over a few iterations, we get a fairly accurate picture of the branch structure of the database file that will allow us to estimate more precisely.

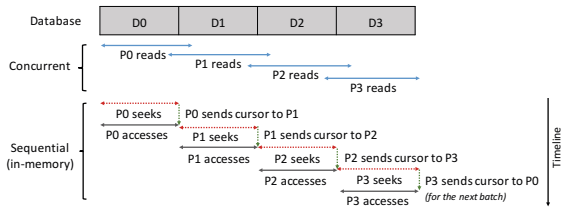

Fig. 7. LMDBIO-DM design: parallel I/O and in-memory sequential seek

Once we guess what pages we need to process, each process touches the appropriate pages in the memory-mapped database file, thus forcing the filesystem to fetch those pages to memory. This step is done in parallel on all processes. Once the data has been fetched to memory, we perform the sequential seek process described in Section V-A to find the starting point of the data batch for the next reader. We expect, however, that

this sequential seek process accesses only or mostly pages that are already in memory and thus will be quick compared with the data I/O itself. Once the seek is done and the reader successfully sends the starting location to the corresponding process, the reader can perform the actual data processing.

VI. Experimental Evaluation and Analysis

In this section we analyze several experimental results to showcase the capability of Caffe/LMDBIO-DM compared with that of Caffe/LMDB and Caffe/LMDBIO-LMM.

A. Experimental Platform

The experimental evaluation for this paper was performed on Argonne's "Blues" cluster.[3] Blues consists of 310 computing nodes connected via InfiniBand Qlogic QDR. Each node has 64 GB of memory and two Sandy Bridge 2.6 GHz Pentium Xeon processors (16 cores, hyperthreading disabled). The storage is 110 TB of clusterwide space provided by GPFS and 15 GB of on-node ramdisk. We built all three versions of Caffe—Caffe/LMDB, Caffe/LMDBIO-LMM, and Caffe/LMDBIO-DM—by using the Intel ICC compiler (version 13.1.3). We used MVAPICH-2.2 over PSM (Performance Scaled Messaging) [15] for all experiments. All experiments were run three times, and the average performance is shown.

We used two datasets for our experiments. The first dataset was the CIFAR10-Large dataset that was trained by using the AlexNet DNN model. The CIFAR10-Large dataset consists of 50 million sample images, each approximately 3 KB. The total dataset size, including the raw images and some metadata corresponding to the images, is approximately 190 GB. The second dataset was the ImageNet dataset that was trained by using the CaffeNet DNN model. The ImageNet dataset consists of 1.2 million sample images, each approximately 192 KB. The total dataset is 240 GB. Although both datasets can be I/O intensive, the ImageNet dataset is particularly so, given the size of the images that need to be processed. In the experiments, our datasets were stored on GPFS.

For our experiments we used a batch size of 4,096 for both datasets. We trained the network for the CIFAR10-Large dataset over 1,024 iterations (4 million images) and the ImageNet dataset over 32 iterations (128K images) on up to 512 processes (i.e., 32 nodes).

B. Overall Performance

Figure 8(a) compares the performance of Caffe/LMDBIO-DM with that of Caffe/LMDBIO-LMM and Caffe/LMDB for the CIFAR10-Large dataset. Caffe/LMDBIO-DM performs better than Caffe/LMDBIO-LMM by around 1.87-fold and better than Caffe/LMDB by around 2.65-fold. The primary improvement in performance for LMDBIO-DM is attributed to the reduced data movement compared with that of LMDBIO-LMM and LMDB. Even though LMDBIO-DM introduces additional serialization in the data I/O path compared with LMDBIO-LMM and LMDB, the impact of this serialization is minimal because of the speculative parallel I/O that it performs.

Figure 8(b) shows the performance breakdown for Caffe/LMDBIO-DM. We note two interesting aspects in this performance. First, the percentage of time taken by data I/O (represented as "Read time") has increased, not decreased, despite our optimizations. Second, the skew time between different processes (represented as "Waiting time before param sync") has decreased significantly. While at first these results might seem counterintuitive, they do follow the general optimization principle used in this paper. Specifically, as described in Section IV, one of the primary shortcomings of the current Caffe/LMDB framework, which Caffe/LMDBIO-LMM inherits, is that different processes perform a different amount of data I/O in order to seek through the LMDB database. This approach results in a significant amount of skew between the processes. Since the computational model of Caffe is bulk synchronous, it eventually results in large wait times for the different processes to coordinate and synchronize with each other.

With LMDBIO-DM, most of the data I/O is parallelized across processes, and each process reads mostly distinct parts of the database. Some serialization still exists in the cursor propagation across the different processes; but since that propagation is done almost entirely in memory without requiring data I/O, the impact of such serialization is minimal. This helps reduce the skew significantly. The data I/O time itself seems to increase in the figure because of the overall reduction in the execution time.

We performed a similar analysis on the ImageNet dataset, as shown in Figure 9. The general trend for ImageNet is similar to that of CIFAR10-Large, although two differences are evident. First, the percentage of time taken by data I/O decreases. This decrease is expected because of the reduction in the amount of data I/O. Second, a higher percentage of the time is now taken by the network parameter communication between processes (represented as "Param sync time" in the figure). The reason is that the processing of the ImageNet dataset is based on the CaffeNet network, which is larger than the AlexNet network used for processing the CIFAR10-Large dataset. Consequently, as we reduce the data I/O overhead, communication time starts to become a dominant factor in the overall execution time.

C. Analysis: Amount of Data Fetched

In this section, we dig a bit deeper into the performance data in order to understand the amount of data that is fetched to each node with Caffe/LMDBIO-LMM and Caffe/LMDBIO-DM. To perform our measurements, we modified Caffe to initially memory-protect all of the memory-mapped database. When a page in the database gets accessed, it triggers a page fault handler, which we catch to measure the amount of data that would be fetched by the filesystem. Once the page is touched, it is unprotected, so any future accesses to the page do not raise additional page faults.

Figure 10 shows the number of "extra" bytes read by Caffe/LMDBIO-LMM for the CIFAR10-Large and ImageNet

[3] http://www.lcrc.anl.gov/about/blues

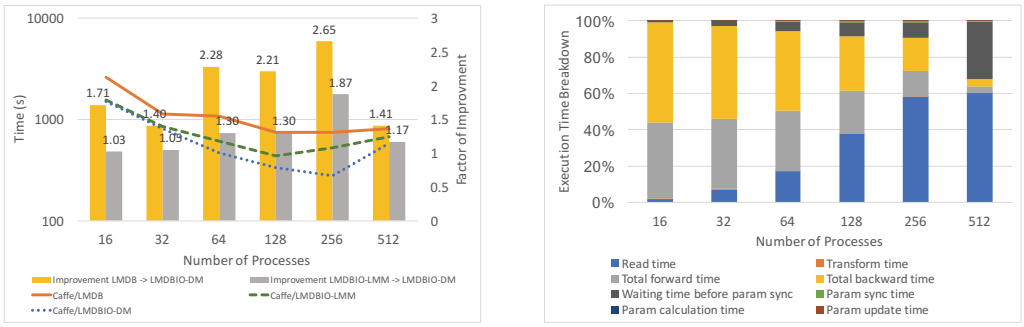

Fig. 8. CIFAR10-Large dataset: (a) overall performance comparison; (b) performance breakdown of Caffe/LMDBIO-DM

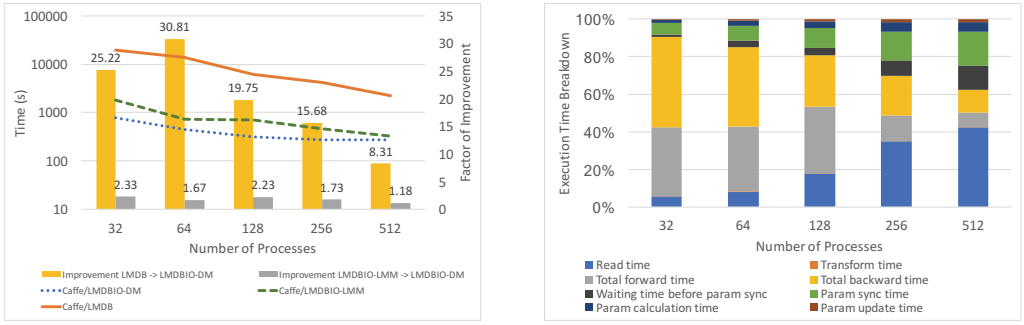

Fig. 9. ImageNet dataset: (a) overall performance comparison; (b) performance breakdown of Caffe/LMDBIO-DM

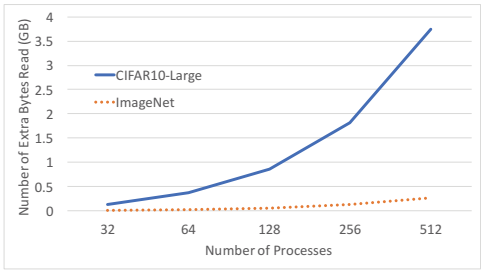

Fig. 10. Caffe/LMDBIO-LMM: Extra bytes read

datasets—that is, how many additional bytes were read by the different processes apart from the actual data that they would need for their processing. We make two observations based on this figure. First, the number of additional bytes increases with the number of processes for both datasets, reaching 4 GB in some cases. In fact, the increase is almost linear with the number of processes. This increase is due primarily to the redundancy in the data read as we increase the number of processes, as explained in Section IV. Specifically, each process needs to seek through the data read by all of the previous processes.

Our second observation is that the increase for the ImageNet dataset is much smaller than that of the CIFAR10-Large dataset. This is also expected, although the reason is more subtle. As a process seeks through the dataset to reach its relevant portion of the database, it needs to read the appropriate headers of the database pages. As discussed in Section II-B, these pages are branch pages. In the CIFAR10-Large dataset, each data sample is approximately 3 KB. Thus, the header and the data sample reside on the same physical page. Consequently, reading the header would load the entire data sample into memory, causing a large amount of additional data to be fetched into memory. In the ImageNet dataset, on the other hand, each data sample is approximately 192 KB and thus takes around 48 pages to store. Therefore, the header page is encountered fewer times in ImageNet if the total dataset size of both datasets is approximately the same; that is, there is one header for every physical page in the CIFAR10-Large dataset, whereas there is one header for every 48 physical pages in the ImageNet dataset. This results in fewer additional bytes read for the ImageNet dataset.

D. Analysis: Accuracy of Estimation

As mentioned in Section V, the performance capability of LMDBIO-DM depends heavily on the accuracy of its estimation on what data will likely be needed for the computation in that iteration. In this section, we present a series of experiments to analyze this behavior. In our experiments, we study the accuracy of our estimation in terms of the number of pages that are needed but are not fetched during the parallel I/O phase (i.e., "missed pages") and the number of pages that are not needed but are fetched during the parallel I/O phase (i.e., "redundant pages").

In the first experiment, we measured the number of missed pages as the computation progressed through its iterations, for the CIFAR10-Large and ImageNet datasets. The experiment used 512 processes in all cases. The first two iterations resulted in nonzero missed pages, although for iterations after that we did not notice any missed pages for both datasets. The reason is that LMDBIO-DM automatically tunes the page

range that it fetches based on the history of the accessed data in the previous iterations, as described in Section V. That is, it corrects its estimate based on history from the prior iterations, thus allowing it to estimate the best- and worst-case bounds of access more effectively. We note that since training computations typically run for several thousands or millions of iterations, the additional missed pages during the first few iterations are mostly inconsequential for overall performance.

In our second experiment we studied the number of redundant pages read through the required iterations. Experimental results are shown in Figure 11. Once again, the experiment used 512 processes in all cases. We notice that the number of redundant pages increases until a certain iteration and then stabilizes. This behavior is expected because of how LMDBIO-DM works. That is, since LMDBIO-DM starts with an initial estimate and then corrects this estimate based on the prior iterations, the range of pages fetched expands with iterations to cover more pages for the parallel I/O; however, once the number of redundant pages read is large enough to not miss any page, the number of redundant pages stabilizes to a constant value.

In our third experiment we studied the missed pages with changing numbers of processes. We ran the experiments for the full iteration count discussed in Section VI-A. We make the following observations based on Figure 12:

1. The number of missed pages increases with the number of processes, but the count is very small. In fact, the total number of missed pages at 512 processes is less than 700 for the CIFAR10-Large dataset ($<$ 1.4 missed pages per process) and less than 200 for the ImageNet dataset ($<$ 0.5 missed pages per process). Moreover, most of these missed pages are in the first few iterations while LMDBIO-DM is trying to converge on the range of pages to fetch.

2. The number of missed pages in the ImageNet dataset is much smaller than that in the CIFAR10-Large dataset. The reason is that the data samples are much larger in the ImageNet dataset than they are in the CIFAR10-Large dataset and, for the same amount of data processed, the ImageNet dataset covers fewer iterations than does the CIFAR10-Large dataset, thus resulting in fewer missed pages.

VII. Related Work

Researchers have proposed a number of parallel derivatives of Caffe. MPI-Caffe [9] and Caffe-MPI [1] are perhaps the most well known. Both implementations target only the compute portions of the framework and do little to optimize I/O. S-Caffe [3] is another parallel derivative of Caffe that performs parallel data read but does not analyze the issues with Caffe/LMDB. It thus inherits many of Caffe/LMDB's shortcomings. Our work aims at fixing the underlying cause of the performance degradation in Caffe/LMDB, thus making it applicable to all parallel derivatives of Caffe.

Perhaps the closest related work is our previous work on LMDBIO-LMM [12]. While LMDBIO-LMM targets the same problem as our current work, it is limited to intranode I/O optimizations. Our current work, LMDBIO-DM, on the other hand, targets I/O optimizations for distributed-memory platforms.

Although our work is based on Caffe, other deep learning frameworks, such as Theano [16] and Google's TensorFlow [2], have highly efficient parallel versions [11], [17]. These frameworks differ in how they parallelize their computation. Their core I/O infrastructure, however, is similar. The I/O infrastructure depends mainly on the format of the dataset, and using LMDB databases for storing data samples is a common practice in the community. Thus, for datasets that are stored in this format, the overheads presented in this paper are unavoidable. Consequently, the ideas presented here are applicable to other deep learning frameworks as well, although the software itself will need additional modifications to plug into these frameworks.

Our work focused on optimizing the `mmap` usage in the LMDB library. In HPC systems, however, more efficient approaches exist for performing I/O. MPI-IO [13], [14] provides a low-level interface to carry out parallel I/O for generic unstructured data. HDF5[4] and NetCDF [4] are high-level I/O libraries that abstract various structured scientific application data into portable file formats and provide feature-rich programming interfaces. Parallel HDF5 [6] and PnetCDF [10] provide parallel access and storage for files with those formats based on MPI-IO.

These I/O frameworks are almost certainly more efficient than `mmap`, but they are all based on explicit I/O. That is, they require the user to provide the exact bytes in the file that would be accessed before actually accessing them. On the other hand, `mmap` performs implicit I/O. It maps the entire file to the virtual address space of the process and dynamically fetches parts of the file to memory as they are being accessed. Since the operating system has limited or no prior knowledge of the data accesses that would be performed, such implicit I/O is fundamentally less performant than is explicit I/O. Nevertheless, implicit I/O is more convenient for complex datasets that require I/O access that is not simple sequential reading (e.g., LMDB uses a B+ tree format to store its data). In the long term, we believe that it would be valuable to migrate the I/O model of Caffe and other deep learning systems to use explicit I/O. Until such action is taken, however, our approach provides a viable solution to improve I/O performance without requiring all existing datasets to be migrated away from the LMDB format.

VIII. Concluding Remarks

Parallel deep learning systems are becoming increasingly common. As computational parallelism increases, however, data I/O becomes the major bottleneck limiting the overall system scalability. In our previous paper, we had presented LMDBIO-LMM—an optimized I/O plugin for Caffe that takes into account the data access pattern of Caffe in order to vastly improve I/O performance. In this paper, we presented LMDBIO-DM, an enhanced version of LMDBIO-LMM that

[4]https://support.hdfgroup.org/HDF5

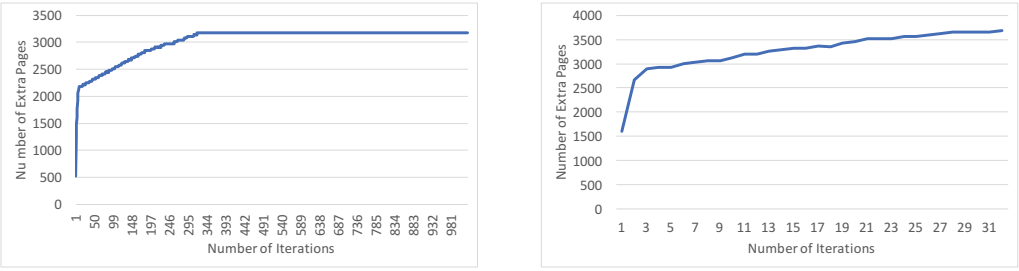

Fig. 11. Caffe/LMDBIO-DM: Redundant pages read for CIFAR10-Large (left) and ImageNet (right)

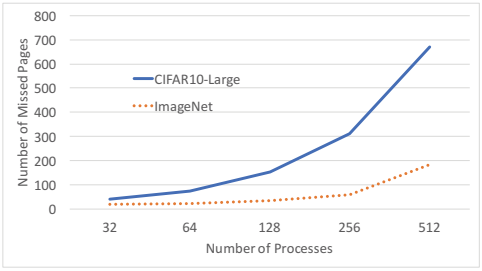

Fig. 12. Caffe/LMDBIO-DM: Missed pages with varying number of processes

optimizes the I/O access of Caffe in distributed-memory environments by minimizing redundant data I/O. Together with a detailed analysis of the I/O issues in Caffe's original I/O framework that continue to exist with LMDBIO-LMM, we presented the overall design and implementation of LMDBIO-DM. We also presented experimental results that show that Caffe/LMDBIO-DM can improve the overall execution time by more than 30-fold compared with the original Caffe/LMDB framework and by 2-fold compared with Caffe/LMDBIO-LMM.

ACKNOWLEDGMENTS

We acknowledge the contributions made by Yanfei Guo, Robert Latham, and Robert Ross from Argonne National Laboratory through initial discussions on the I/O analysis of LMDB. The material presented in this paper was based upon work supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357, and in part by the NSF XPS program via CCF-1337131. We gratefully acknowledge the computing resources provided on Blues, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] Caffe-MPI for Deep Learning. https://github.com/Caffe-MPI/Caffe-MPI.github.io, September 2015.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[3] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. S-Caffe: Co-designing MPI Runtimes and Caffe for Scalable Deep Learning on Modern GPU Clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 193–205. ACM, 2017.

[4] Glenn Davis and Russ Rew. Data Management: NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, 10:76–82, 1990.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-Scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[6] The HDF Group. Enabling a Strict Consistency Semantics Model in Parallel HDF5. https://support.hdfgroup.org/HDF5/doc/Advanced/PHDF5FileConsistencySemantics/PHDF5FileConsistencySemantics.pdf, 2012.

[7] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[8] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, 2009.

[9] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. Why M Heads Are Better than One: Training a Diverse Ensemble of Deep Networks. *arXiv*, 2015.

[10] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[11] He Ma, Fei Mao, and Graham W. Taylor. Theano-MPI: A Theano-Based Distributed Training Framework. *CoRR*, abs/1605.08325, 2016.

[12] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Towards Scalable Deep Learning via I/O Analysis and Optimization. In *Proceedings of the 19th International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2017.

[13] R. Thakur, W. Gropp, and E. Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *IEEE/ACM Conference on Supercomputing (SC)*, pages 1–1, November 1998.

[14] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–189, Washington, DC, USA, 1999. IEEE Computer Society.

[15] The Ohio State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu, 2014.

[16] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[17] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. Distributed TensorFlow with MPI. *CoRR*, abs/1603.02339, 2016.