Scalable Deep Learning via I/O Analysis and Optimization

SARUNYA PUMMA, Department of Computer Science, Virginia Tech, USA
MIN SI, Mathematics and Computer Science Division, Argonne National Laboratory, USA
WU-CHUN FENG, Department of Computer Science, Virginia Tech, USA
PAVAN BALAJI, Mathematics and Computer Science Division, Argonne National Laboratory, USA

Scalable deep neural network training has been gaining prominence because of the increasing importance of deep learning in a multitude of scientific and commercial domains. Consequently, a number of researchers have investigated techniques to optimize deep learning systems. Much of the prior work has focused on runtime and algorithmic enhancements to optimize the computation and communication. Despite these enhancements, however, deep learning systems still suffer from scalability limitations, particularly with respect to data I/O. This situation is especially true for training models where the computation can be effectively parallelized, leaving I/O as the major bottleneck. In fact, our analysis shows that I/O can take up to 90% of the total training time. Thus, in this article, we first analyze LMDB, the most widely used I/O subsystem of deep learning frameworks, to understand the causes of this I/O inefficiency. Based on our analysis, we propose LMDBIO—an optimized I/O plugin for scalable deep learning. LMDBIO includes six novel optimizations that together address the various shortcomings in existing I/O for deep learning. Our experimental results show that LMDBIO significantly outperforms LMDB in all cases and improves overall application performance by up to 65-fold on a 9,216-core system.

CCS Concepts: • **Information systems** → **Parallel and distributed DBMSs**; *B-trees*; *Distributed storage*; *Distributed retrieval*; • **Theory of computation** → **Distributed computing models**; • **Computing methodologies** → **Batch learning**; *Parallel computing methodologies*; *Distributed computing methodologies*;

Additional Key Words and Phrases: Scalable deep learning, parallel I/O, Caffe, LMDB, LMDBIO, I/O in deep learning, I/O bottleneck

ACM Reference format:
Sarunya Pumma, Min Si, Wu-Chun Feng, and Pavan Balaji. 2019. Scalable Deep Learning via I/O Analysis and Optimization. *ACM Trans. Parallel Comput.* 6, 2, Article 6 (June 2019), 34 pages.
https://doi.org/10.1145/3331526

The material presented in this article 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 and Bebop, two high-performance computing clusters operated by the Laboratory Computing Resource Center at Argonne National Laboratory.
Authors' addresses: S. Pumma and W.-C. Feng, Department of Computer Science, Virginia Tech, USA; emails: {sarunya, wfeng}@vt.edu; M. Si and P. Balaji, Mathematics and Computer Science Division, Argonne National Laboratory, USA; emails: {msi, balaji}@anl.gov.

2329-4949/2019/06-ART6 $15.00
https://doi.org/10.1145/3331526

1 INTRODUCTION

Deep learning is an emerging technology that is gaining prominence in a multitude of domains because of its ability to process unstructured input and to predict trends. Training of deep neural networks (DNNs), a process where large volumes of input data are mined to find patterns and trends, usually involves high computational and memory complexity. To meet these complex resource demands, we note three broad trends in the community. First, researchers have targeted scalable high-performance computing as a mechanism to process data in parallel across multiple processors. Second, there has been a large influx of commercial hardware that is either tuned for or dedicated to deep learning systems. This hardware includes processors (e.g., NVIDIA GPUs [18, 36], Intel Xeon Phi [12], Google TPUs [14]), high-speed networks (e.g., Mellanox InfiniBand [11, 35], Intel OmniPath [57]), and memory technologies [5, 21]. Third, researchers have developed numerous algorithms to make deep learning more computationally efficient by allowing them to realize more algorithmic parallelism without losing convergence accuracy. For instance, You et al. [59, 60] demonstrated parallelism across 32K data samples within each iteration with negligible loss in convergence accuracy. Moreover, many parallel deep learning frameworks have been proposed in the past decade that incorporate the cited trends in usable software instantiations, including Caffe [2, 4, 22, 29], TensorFlow [3, 55], Theano [31, 53], Caffe2 [47], PyTorch [38], Microsoft Cognitive Toolkit [43], Apache MXNet [7], and Chainer [54].

Nevertheless, scalable deep learning remains a difficult problem to solve. For training models that focus on a single pass of the data (e.g., ones that need to process a very large amount of input data) and for training models where the computation can be easily and efficiently parallelized or offloaded to hardware computational units, moving the data becomes a bigger problem than the computation itself. In particular, moving data from a global filesystem for such processing can be a major bottleneck in the overall computation. Our study shows that even with a small amount of parallelism in such deep learning systems, I/O accounts for a majority of the training time, thus degrading the overall system scalability. For instance, with our experimental datasets [10, 26], we observe that as much as 90% of the execution time may be devoted to data I/O.

In this article, we first analyze the *Lightning Memory-Mapped Database (LMDB)*, the most widely used I/O subsystem in deep learning frameworks. The intent of this analysis is to establish a clear understanding of the I/O problems in deep learning. Based on our analysis, we present a number of shortcomings in LMDB that are caused mainly by its usage of implicit I/O through `mmap`, its reliance on a B+-tree database structure for storing data, and its inefficiency in I/O management in the context of parallel computing. Our analysis shows that LMDB achieves less than 10% of the achievable performance of the I/O subsystem because of these shortcomings. Next, we propose "LMDBIO," an optimized I/O subsystem for scalable deep learning. LMDBIO includes six sophisticated optimizations to address the shortcomings identified in our analysis.

We note that two of the optimizations, **LMDBIO-LMM** and **LMDBIO-LMM-DM**, have been previously published in References [40] and [39], respectively. For completeness, we briefly discuss their designs, and we include additional new results and analysis on a larger supercomputing system. The remaining four optimizations—**LMDBIO-LMM-DIO**, **LMDBIO-LMM-DIO-PROV**, **LMDBIO-LMM-DIO-PROV-COAL**, and **LMDBIO-LMM-DIO-PROV-COAL-STAG**—are new and previously unpublished.

We present the detailed design of these optimizations and demonstrate that together they can significantly improve the performance of parallel data reading. In fact, on our system, these optimizations can saturate the system's available I/O bandwidth in deep learning frameworks. We also present and analyze experimental results to showcase improvements of LMDBIO compared with LMDB using different networks and datasets. Our experimental results show that LMDBIO can improve the overall training time of Caffe by up to 65-fold on a 9,216-core system.

We note here that the central focus of this study is on scaling deep learning on large supercomputing systems rather than on commodity clusters or cloud platforms. Most large supercomputers do not have a local disk on each node; thus, I/O typically is performed over the shared filesystem. In some cases, on-node storage might be present in the form of nonvolatile storage. Such storage is not persistent across the lifetime of the machine, however, and typically is wiped clean when a new job is assigned to a node. Thus, data I/O still has to be performed from the global filesystem. Even on systems that utilize on-node storage technologies in the form of burst buffers, staging data on to these burst buffers requires prior knowledge as to which node would need what segment of the data. Such information is, unfortunately, not readily available in modern deep learning systems.

The rest of the article is organized as follows. Section 2 presents an overview of the Caffe DNN training framework and the LMDB I/O subsystem. Section 3 presents a detailed analysis of LMDB and identifies various shortcomings that need to be addressed. Section 4 elaborates on the design and implementation of our proposed I/O subsystem, LMDBIO, which includes six novel optimizations to address the problems identified in our analysis. Other literature related to this article is presented in Section 6. Section 7 presents a few ideas about a new filesystem for deep learning workloads, and Section 8 briefly summarizes our conclusions.

2 BACKGROUND

In this section, we present a brief overview of the Caffe deep learning framework and the LMDB I/O subsystem.

2.1 Overview of Caffe and Batch Training

Caffe is a well-known deep learning framework developed by the Berkeley Vision and Learning Center. Caffe is written in C++ with CUDA support. The original goal of Caffe was to provide an efficient GPU-based framework for convolutional neural network training, but it has since been extensively modified by several researchers to support generic CPU architectures as well.

Caffe follows the stochastic gradient descent approach to train DNNs. The goal of the training is to obtain a set of parameters for the DNN that most accurately represent a given dataset. The training starts by initializing the parameters of the DNN. Most training frameworks typically initialize the parameters to random values, although a growing number of researchers use better initial approximations of the parameters based on known properties of the input data. Training is an iterative process that continues until the parameter set converges to the desired accuracy. In each training iteration, a subset of data samples, called a batch, in the database is drawn and used to train the network. Caffe then measures the deviation error between the predicted value from the current DNN parameters and the actual value from the dataset. This error is then utilized to improve the DNN parameters for the next training iteration. Once the training converges, the final set of DNN parameters is 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 and general inference model is the use of a very large set of (high-quality) training data samples. Large organizations usually train DNNs with hundreds of terabytes or even petabytes of data. Therefore, to allow DNN training to be practical, data access and processing must be fast.

Sequentially processing each data sample in the training dataset is a valid, but conservative, approach that is generally not useful in practice. Most modern deep learning frameworks allow for what is called *batch* training. This allows for some parallelism in the DNN training either by partitioning each batch of data samples across processes/threads (e.g., Caffe) or by partitioning the neural network across processes/threads (e.g., TensorFlow, Theano, Caffe2, PyTorch, Microsoft Cognitive Toolkit, Apache MXNet, and Chainer). Consequently, multiple data samples can be

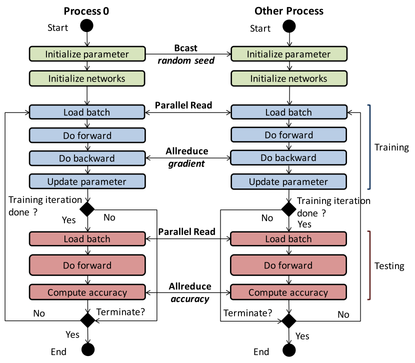


Fig. 1. Caffe's data-parallel workflow.

processed before updating the DNN parameters. Processing one batch of data samples is referred to as one training iteration. A typical training comprises a very large number of iterations, making the training process mainly bulk synchronous. Parallelism is utilized within each iteration, but all processes need to synchronize at the end of each iteration to update their DNN parameters.

The most widely used communication library in the deep learning community is MPI [34, 47, 48], which has been integrated in several frameworks, including Caffe. Other communication libraries also exist, such as TensorFlow's gRPC (default in TensorFlow) and Facebook's Gloo [11] (default in Caffe2), as well as some high-level communication libraries, such as Baidu's Allreduce [13, 42] (collective operation implementation based on MPI point-to-point operations), Uber's Horovod [46] (collective operation implementation based on MPI collective operations, NVIDIA's NCCL [1], and IBM's PowerAI DDL [6]). We note that most modern deep learning frameworks can dispatch threads or even whole CPUs (in cases where the system is equipped with accelerators) for communication and I/O prefetching and preprocessing. Such prefetching techniques, however, can hide some of the I/O cost when the cost is smaller than that of computation, but they cannot fully avoid it.

The data-parallel model in Caffe is shown in Figure 1. The overall flow of the training is the same as that of sequential processing except that the data batch loading, the forward pass, and the backward pass are parallelized on multiple processes/threads. Parallel network training, however, comes with an additional communication cost where network parameters must be synchronized across processes/threads.

For storing and retrieving data samples, a number of database options are available in the community for deep learning systems. The most widely used database option is LMDB, which is the default database format used by Caffe.

2.2 Overview of LMDB

LMDB involves two concepts. First, it refers to a database format that arranges its content based on a B+ tree and allows efficient simultaneous read and write access to the database. Second, LMDB refers to a library that provides the API to access and manipulate the LMDB database. This library makes use of the operating system (OS) memory-mapping mechanism, `mmap`, to enable in-memory database access. We note that LMDB is not specific to deep learning. It is a well-known database that is used in multiple domains with different usage models. In this section, we discuss background information related to LMDB in two ways: (1) `mmap` and its dynamic data-reading mechanism (Section 2.2.1) and (2) the LMDB database format and its data access model (Section 2.2.2).

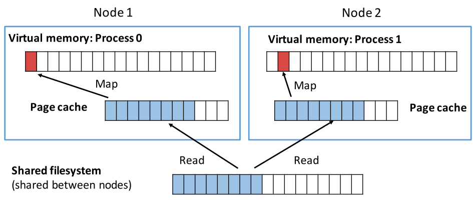


Fig. 2. Workflow of mmap.

2.2.1 Dynamic Data Loading via mmap. LMDB relies on mmap to perform in-memory data access. Mmap is a generic Unix system call that maps the layout of a file on the filesystem to the virtual address space of a process, thus giving an illusion to the process that the entire file is in memory. Data access is tracked by the OS at a page-level granularity, and data are dynamically fetched from the filesystem to memory when the application accesses it. This model is convenient for accessing files with complex structures, such as B+-tree databases, since the application process does not have to be concerned about which exact bytes need to be fetched to memory. It can pretend that the entire file is already in memory.

Mmap dynamically reads pages from the filesystem to memory on demand. In other words, the data are not read until it is required. The workflow used by mmap is illustrated in Figure 2. Three components are involved in data reading: the filesystem (which can be local or shared across machines), a page cache (which is shared across processes on the node), and a virtual address space (which is private to each process). When mmap is called, each process allocates a virtual address space for the database, but it does not fetch any data into this space. Instead, the mmap call protects the allocated virtual address space to raise a page fault if the process tries to read from or write to this virtual address space. When the process accesses the first page, which is not present in memory, the page-fault handler is triggered. The page-fault handler first reads data from the filesystem to the page cache and then maps the corresponding page from the page cache to the appropriate virtual address region that the user is trying to access. The pages in the page cache can be mapped to the virtual address spaces of multiple processes on the same node. Another benefit of using mmap is that the OS automatically frees pages when physical memory is almost full. Thus, the user does not have to worry about out-of-memory problems.

Aside from these advantages, mmap also has several shortcomings, which stem primarily from the fact that it offloads all the I/O handling responsibilities to the OS. Thus, application processes do not have any control over the actual I/O. For example, mmap does not allow users to provide detailed information about their access pattern. While users can provide some simple hints using madvise and fadvise, these hints are primarily for simple manipulation of access patterns. For example, they allow users to distinguish between sequential and random access of data. But they are not suitable to more complex access patterns, such as strided access of batches of data. This abstraction of I/O from the applications that mmap provides sometimes results in tremendous loss in I/O performance, as we discuss in Section 3.3.1.

2.2.2 LMDB Database Format. LMDB adopts a flattened B+-tree data structure as its database layout. It uses pages to represent nodes (i.e., branch and leaf nodes) in the B+ tree, where each node is stored on the filesystem in a block-aware manner.

The LMDB database consists of four types of pages: *metadata pages*, *branch pages*, *leaf pages*, and *overflow pages*. The first two pages of the database file are metadata pages that store information specific to the overall database (e.g., version of the database, size of the database). The branch and leaf pages represent the internal branch and leaf nodes in the B+-tree structure. Each of the branch and leaf pages contains a page header that has information associated with that page (e.g., type of page, amount of free space in the page, offsets to neighboring pages) and some actual data key-value records. Since the size of each page is typically limited to 4KB (the OS page size), a leaf node cannot accommodate data records that are larger than 4KB.[1] In such cases, LMDB uses overflow pages to store data records that cannot fit within one page. Thus, each leaf page can have zero or more overflow pages associated with it. We note that only the first overflow page corresponding to a given leaf page has a header.

Since LMDB's data format is a complex tree structure, correctly identifying a record requires a complete collection of pointers to all the branch pages in the path to the target data record. LMDB stores this information in a convenient data structure that it refers to as the "cursor," which can be thought of as the identity of a data record in the LMDB database. When the database is opened, LMDB initializes this cursor to point to the root of the database. LMDB also provides API functions to move the cursor forward or backward, thus allowing access to the remaining records in the database.

LMDB's database format is designed to allow for efficient sequential data access: Each leaf node has a link that connects it to the adjacent leaf node. The layout of the LMDB database file depends not only on the database's contents but also on the order in which the data records were inserted into the database and how frequently the commit operations are issued. Thus, we cannot determine the exact layout of the database unless we also have information on how the database was created. This information is not stored in the native LMDB database format.

To access an arbitrary data record in the database, LMDB needs to navigate from the root of the B+ tree and through all the corresponding branch and leaf nodes. Such tree parsing, which we refer to as "sequential seek," requires data to be read from the filesystem to memory, because the pointer information is stored in the page headers. Although only the header portion of the page is needed for parsing the tree, the entire page is read to memory, since `mmap` loads data at a page-level granularity. The worst-case scenario is when every data record fits in the leaf page (i.e., no overflow pages). In this case, every page along the way to the target leaf page will need to be read while parsing the tree.

In fact, none of the existing LMDB operations allow for random access within the database without requiring additional information. Unfortunately, the data access pattern of parallel deep learning is semirandom—in other words, each process would need to skip the records that are being processed by the other processes—thus making data I/O hard to optimize in such frameworks.

3 ANALYSIS OF I/O IN DEEP LEARNING

In this section, we present a detailed analysis of LMDB and its shortcomings for scalable deep learning. Our analysis uses the Caffe framework with LMDB as the I/O subsystem, although the analysis is applicable to other frameworks using LMDB as well, including Caffe2, PyTorch, TensorFlow, and Keras-TensorFlow [17]. We also use other small I/O benchmarks to provide a more complete picture of the analysis.

3.1 Experimental Setup

In this section, we describe the software and hardware environment that we used for our experiments.

Datasets: We use three datasets for our experiments. The first is the **CIFAR10-Large** dataset, which consists of 50 million sample images, each approximately 3KB. The total dataset size, including raw images and metadata corresponding to the images, is approximately 190GB.

[1]Even though most systems today support large 2MB pages and huge 1GB pages, file-backed `mmap` still typically uses 4KB pages, even on modern systems.

CIFAR10-Large is an extended version of CIFAR10 [26] that we generated for our experiments. Although we adopted simple replication techniques to augment our dataset, other data augmentation approaches (such as Gaussian noise [41]) can be used to replicate our results as well, because the size and the layout of the dataset would be identical irrespective of which technique is used. The second dataset we used is the **ImageNet** [45] dataset, which consists of 1.2 million sample images, each approximately 192KB (total dataset is 240GB). The third is the **ImageNet-Large** dataset, which is an extended version of ImageNet that replicates some images from ImageNet for a total of 6 million images (total dataset is 1.1TB). Although all datasets can be I/O intensive, the ImageNet datasets are particularly so, given the size of the images that need to be processed.

DNNs: We use three DNNs for our experiments. **AlexNet**[2] is used to train the CIFAR10-Large dataset. AlexNet is a small network with 13 layers and 89K parameters. **CaffeNet**[3] is used to train the ImageNet and the ImageNet-Large datasets. It is a large network with 22 layers and 60M parameters. **ResNet50**[4] [19] is used to train ImageNet-Large. It is a large network with 228 layers and 25.6M parameters [61]. We note that the number of layers in each DNN is based on the DNN definition in Caffe.

Platforms: The experimental evaluation for this article was performed on two clusters operated by the Laboratory Computing Resource Center at Argonne: **Blues**[5] and **Bebop**.[6] Blues consists of 310 computing nodes connected via InfiniBand Qlogic QDR. Each node has 64GB of memory, two Sandy Bridge 2.6 GHz Pentium Xeon processors (16 cores, hyperthreading disabled), and a 15GB ramdisk. Bebop consists of 672 Intel Broadwell nodes. Each node consists of 36 cores, 128GB of memory, and a 15GB ramdisk. The interconnect is Intel OmniPath.

Data storage: The datasets are stored on IBM General Parallel File System (GPFS). Blues and Bebop share the same GPFS installation. The storage is 110TB of clusterwide space. The system has 10 Network Shared Disk servers with no replication. To ensure that data are read from the filesystem rather than from memory (i.e., no caching), we clear the operating system's page cache and GPFS's file cache prior to performing each experiment.

Software stack: We used Caffe version 1.0.0-rc3 together with single-threaded Intel MKL, unless specified otherwise. Caffe was built by using the Intel ICC compiler (version 17.0.4). On Blues, we used MVAPICH-2.2 over PSM (Performance Scaled Messaging) [52] for all experiments. On Bebop, we used Intel MPI version 2017 for all experiments. All experiments were run three times, and the average performance is shown.

Data access pattern: Caffe supports a data access pattern that is commonly known as data sharding, strided data reading, or distributed data reading. All processes together access contiguous blocks of data in each iteration, while each individual process has a strided access pattern for data access across iterations. Thus, each process accesses a disjoint set of data samples in each training iteration. This data access pattern is a part of the Caffe workflow and is the same across all experiments. Data reading and parsing are performed in parallel by multiple processes. We note that Caffe does not perform data shuffling.

3.2 Caffe/LMDB Scalability Analysis

In our analysis, we evaluated the strong-scaling scalability of Caffe/LMDB by training it using the CIFAR10-Large dataset on AlexNet. We used a batch size of 18,432 for 512 iterations (approximately

[2]https://github.com/BVLC/caffe/blob/master/examples/ cifar10/cifar10_full.prototxt.
[3]https://github.com/BVLC/caffe/blob/master/models/bvlc_reference_caffenet/train_val.prototxt.
[4]https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-50-deploy.prototxt.
[5]http://www.lcrc.anl.gov/systems/resources/blues.
[6]http://www.lcrc.anl.gov/systems/resources/bebop.

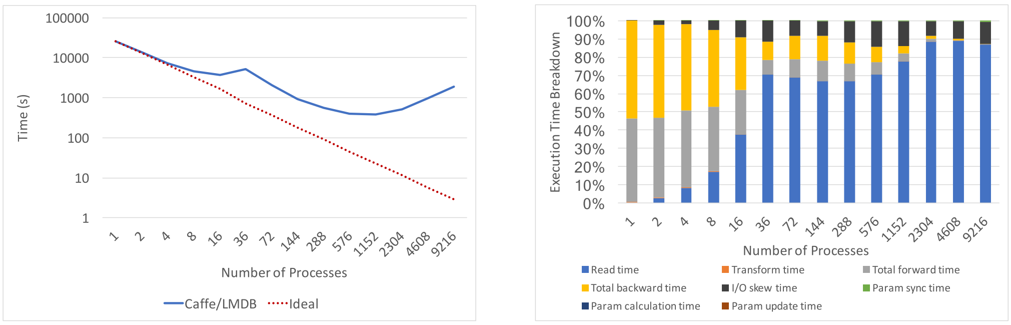


Fig. 3. Caffe/LMDB's strong scaling using CIFAR10-Large on Bebop: (a) total execution time and (b) execution time breakdown.

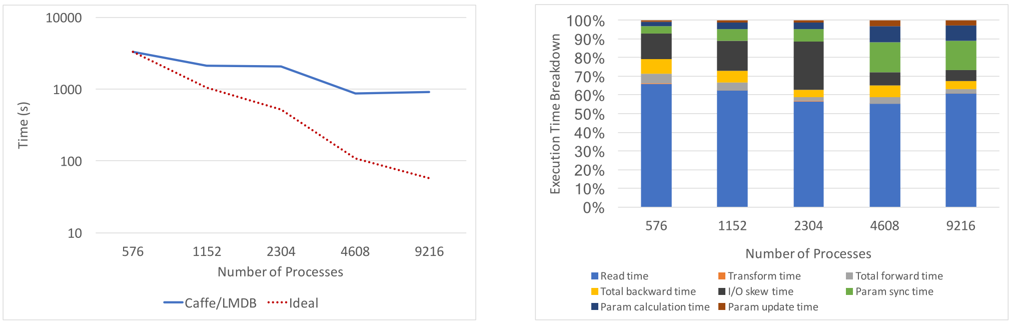


Fig. 4. Caffe/LMDB's strong scaling using ImageNet-Large with CaffeNet on Bebop: (a) total execution time and (b) execution time breakdown.

9 million total data samples) on Bebop. The training was scaled from 1 process to 9,216 processes (i.e., 256 nodes with 36 cores on each node). Figure 3(a) shows the overall training time of Caffe compared with ideal strong scaling. The figure shows that Caffe/LMDB scales poorly even with a small number of processes and is nearly 660-fold worse than ideal strong scaling on 9,216 processes.

We next performed a breakdown of the execution time, shown in Figure 3(b), to understand which components of Caffe/LMDB take the most time. We notice two important trends in the figure. First, the data I/O time (denoted "Read time") increases with the number of processes. This increase is because the computation time (i.e., total forward time, total backward time, parameter calculation time, and parameter update time) tends to scale well with the number of processes, leaving I/O as the bottleneck. In fact, as we scale the problem to 9,216 processes, I/O takes approximately 90% of the total time. Second, the "I/O skew time" grows with the number of processes, raising concerns of load imbalance occurring between the processes as we scale to a large number of processes.

We have also performed a similar analysis with ImageNet-Large using the CaffeNet and ResNet50 network models by using a batch size of 18,432 for 128 iterations, as illustrated in Figures 4 and 5. From our experiments, we observe that the actual read time (denoted "Read time") and the I/O imbalance time (denoted "I/O skew time") take up to 66% of the time with CaffeNet and 75% of the time with ResNet50 on 9,216 processes, thus dominating the overall training time. We note that we could not run experiments with ResNet50 on less than 4,608 processes on Bebop because of insufficient memory on each node. For this reason, we use the CaffeNet neural network for all of the remaining experiments with the ImageNet-Large dataset in this article.

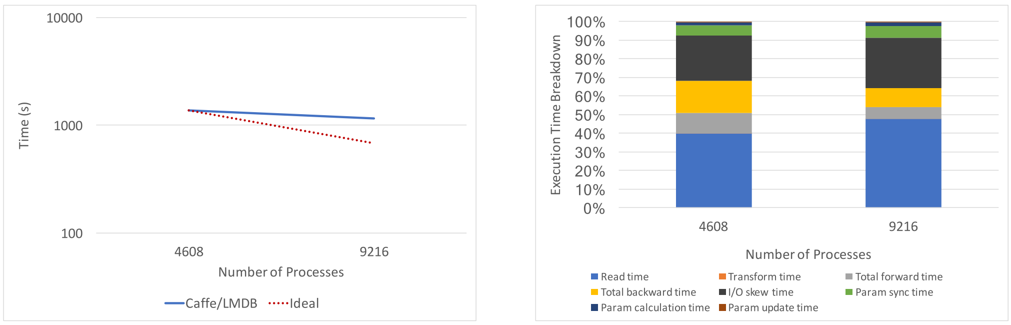


Fig. 5. Caffe/LMDB's strong scaling using ImageNet-Large with ResNet50 on Bebop: (a) total execution time and (b) execution time breakdown.

A broader issue that one needs to be aware of is that hardware technology trends point to the fact that I/O is already a bottleneck and its performance relative to that of computation is only getting worse with time [44]. The specific ratios between the computation cost (in the model that we chose) and the I/O cost (in the filesystem that we chose) are simply representative examples of a more general problem.

As a final step, we wanted to understand the peak performance that our filesystem can achieve, to help us distinguish between using a slow filesystem vs. LMDB itself being inefficient. To do so, we used the IOR benchmark[7] to measure the performance of POSIX I/O on Bebop and compared that with the data I/O performance achieved by Caffe/LMDB. IOR performance is often considered to be the best case for I/O performance that a given platform can achieve. Our comparison showed that the I/O performance achieved by Caffe/LMDB was much worse than that reported by IOR. In fact, the data I/O bandwidth achieved by Caffe/LMDB was less than 10% of that demonstrated by IOR. This result suggests that the performance loss is caused mainly by inefficiencies in Caffe/LMDB rather than by limitations in the filesystem (or the I/O hardware) itself.

3.3 LMDB Inefficiencies

In this section, we discuss various inefficiencies in LMDB that need to be addressed.

3.3.1 `Mmap`*'s Interprocess Contention.* As noted in Section 2.2, LMDB relies on `mmap` to perform in-memory data access where all I/O operations are offloaded to the OS. The OS, however, has no knowledge that the application is a parallel application, and hence it must consider the `mmap` done by each process to be independent (except for sharing the page cache when possible). This behavior, however, causes an unfortunate interaction with the Linux process scheduler.

Since kernel version 2.6, Linux has used the Completely Fair Scheduler (CFS) as the default process scheduler. CFS's scheduling policy is based simply on the amount of CPU time taken by each process. It arranges all runnable processes in an internal red-black tree that is ordered based on the CPU-time usage of each process, where the process that has the least-used CPU time will be scheduled first. Consider the case where several processes in the system issue I/O requests. In this case, all these processes will sleep while waiting for the I/O requests to complete. When an I/O request completes, an interrupt is raised and the I/O interrupt handler invoked. Since the interrupt handler is a *bottom-half* handler in Linux (meaning that it has no context of the process that owns the completed request), it will mark **all** the sleeping processes that requested I/O as *runnable*. Then, the CFS scheduler will schedule each of the runnable I/O processes in CPU-time

[7]https://github.com/LLNL/ior.

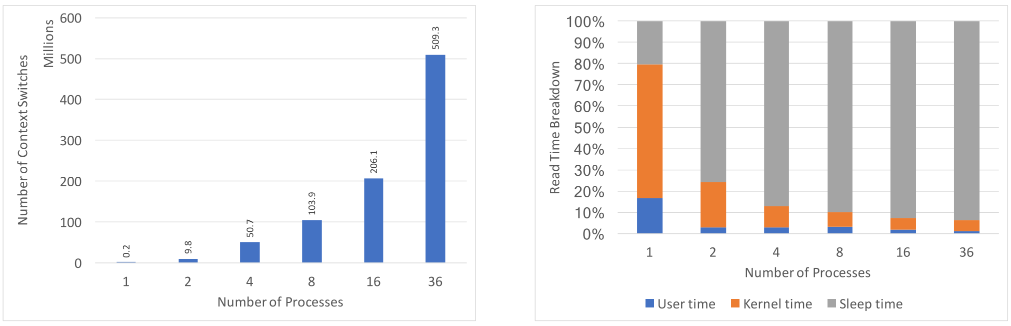


Fig. 6. Caffe/LMDB's mmap analysis (CIFAR10-Large dataset on Bebop): (a) context switches; (b) sleep time.

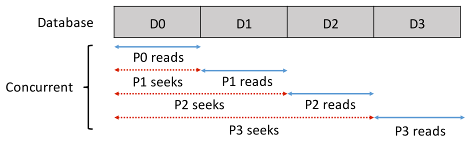


Fig. 7. LMDB redundant data movement.

order with no regard for the order in which they issued the I/O operations. If the request that just completed does not belong to this process, then it will go back to sleep without making more progress on its work.

This lack of coordination between the order in which the I/O operations are issued and the order in which the processes are woken up causes several unnecessary "context switches" to occur without performing any useful work. Aside from the dramatic increase in context switches, the amount of "sleep time" of each process surges as well. We quantitatively measured the number of context switches and sleep time during data I/O in Caffe/LMDB and plotted the results in Figure 6(a) and (b), respectively. The number of context switches increases by approximately 48 times from one process to two processes and by approximately 2,546 times from one process to 36 processes. When using multiple processes to read the data, the ratio of the sleep time to the read time increases to 93% on 36 processes.

3.3.2 Sequential Data Access Restriction. As mentioned in Section 2.2.2, LMDB does not support random database accesses without additional information; that is, the LMDB database can be accessed only sequentially. This is a significant issue for parallel deep learning, because each process needs to read and process a different subset of data (typically interleaved with the data required by other processes). Thus, each process needs to start from the root of the database and parse through (and skip) all the intermediate records until it reaches the desired record that it wants to process.

The data access pattern when using LMDB in parallel is demonstrated in Figure 7. As shown in the figure, every process except for $P0$ starts reading the data from a non-zero offset of the file. Therefore, these processes would need to perform a sequential seek to reach their corresponding starting locations. For instance, $P1$ would read some of the same data as $P0$ (i.e., $D0$), which it would later discard once it reached the beginning of $D1$. $P3$ reads the most amount of extra data. This data access model can cause each process to read a total of $R \times B$ bytes, where R is the total number of readers and B is the size of the data portion of each process.

Apart from the large amount of redundant data I/O, this data access also causes large imbalance and skew in data reading, because each process reads a different number of bytes.

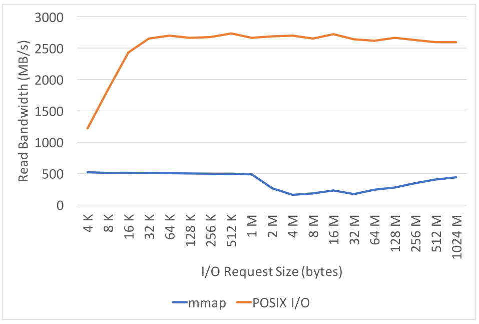


Fig. 8. I/O block size.

3.3.3 `Mmap` *Workflow Overheads.* Since `mmap` performs implicit I/O, the user has no control over when an I/O operation is issued. `Mmap` needs to keep track of what data the user is trying to access. Only when the user tries to touch a piece of data can `mmap` deduce that that data segment is needed. The typical workflow used by `mmap` is as follows. When the user tries to access data that are not already available, a page fault signal is generated, which internally invokes an I/O operation. When the I/O operation completes, an interrupt is generated that marks the corresponding operation as complete. Thus, the workflow used by `mmap` is necessarily reactive based on the user data access pattern and leads to inefficiencies in the I/O path.

To showcase this inefficiency in `mmap`, we performed an experiment to compare the I/O read bandwidth achieved by `mmap` with the bandwidth achieved by using explicit I/O (based on POSIX I/O). We developed a microbenchmark to read a 256GB file using a single reader on a single machine. To read the file, we used `memcpy` and `pread` with the `mmap` and POSIX I/O benchmarks, respectively. In this benchmark, we kept the read chunk size used by POSIX I/O to be 4KB, namely, the OS page size, similar to what is used by `mmap`. In this way, the benchmark does not mix effects from the read block size with that of the `mmap` workflow. Effects of the read block size are studied separately in Section 3.3.4. The results from our experiment show that `mmap`'s read bandwidth is approximately 2.5 times lower than the read bandwidth achieved by POSIX I/O. This observation showcases the inefficiencies in `mmap`'s workflow.

3.3.4 I/O Block Size Management. As discussed earlier, most deep learning frameworks are iterative. In each iteration, they read one batch of data samples and process them before moving on to the next iteration. With parallel deep learning, this batch of data samples is further split into multiple subbatches, where each process reads a subbatch and processes it. As the number of processes increases, however, the batch of data samples is split among more processes, so each subbatch is smaller. In the extreme case, where the number of processes participating in parallel deep learning is equal to the number of data samples in the batch, each subbatch would contain just a single data sample. This would dramatically reduce the size of the I/O performed by each process within an iteration. For instance, with the CIFAR10-Large dataset, each data sample is just 3KB, causing the I/O operations to be done in small-page-size granularity, thus leading to significant inefficiencies.

To demonstrate the effect of I/O block size on read performance, we used the same microbenchmark discussed in Section 3.3.3, but this time we varied the read block size from 4KB to 1GB. Figure 8 shows the read I/O bandwidth of `mmap` and POSIX I/O with different I/O block sizes. Two trends are noteworthy. First, the I/O read bandwidth of POSIX I/O increases with the block size. This increase is expected and has also been demonstrated by other researchers in the past. Second, the block size has no impact on the I/O performance achieved by `mmap`. The reason is that `mmap` does not have information about the overall access pattern used by the application and needs to wait

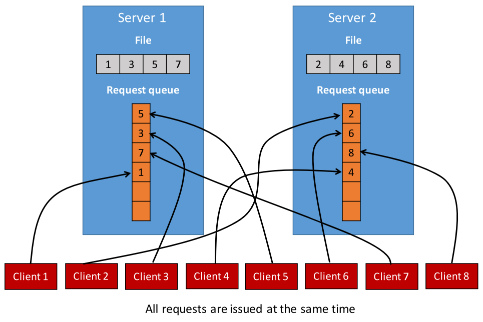


Fig. 9. I/O randomization.

for the application to access data before fetching it. Even when the application uses a larger block size for performing the `memcpy` in the benchmark, this information is not passed to `mmap`. Thus, the I/O blocks used by it are inherently small. The take-away of this analysis is that although the current I/O methodology used by LMDB cannot benefit from larger I/O blocks, if one were to migrate LMDB to using explicit I/O, then larger I/O blocks could give a significant performance boost.

3.3.5 I/O Randomization. One aspect to consider while performing parallel I/O is the data access order that the various I/O requests create. For example, consider a scenario where a large number of processes need to divide a large file into smaller pieces and each process needs to access a part of it. In this example, each process issues an I/O request for its piece of the data that it needs to fetch. Since each process is independent, however, these I/O requests do not arrive at the I/O server processes in any specific order, causing the server processes to access the file in a nondeterministic fashion. We refer to this problem as I/O randomization and illustrate it in Figure 9.

I/O randomization hurts performance, because, unlike sequential I/O, it cannot benefit from most I/O optimizations including data prefetching and caching, thus becoming limited by disk seek overheads. Another unfortunate aspect of I/O randomization is that as the number of processes performing parallel I/O increases, the randomization of I/O requests increases as well. Furthermore, as the read block size associated with each I/O operation increases, the impact of the additional disk seeks and the lack of benefits from data prefetching and caching also increase. Thus, to maximize the overall performance, we need to carefully balance the various metrics of the amount of I/O parallelism, read block size, and I/O randomization.

4 DESIGN AND IMPLEMENTATION OF LMDBIO

In this section, we present the design and implementation of LMDBIO, an optimized I/O subsystem for scalable deep learning. At a high level, LMDBIO loses some of the generality of LMDB by assuming certain characteristics of deep learning frameworks. But by doing so it addresses the various shortcomings of LMDB that we presented in Section 3. LMDBIO encompasses six different optimizations to address these shortcomings: LMDBIO-LMM, LMDBIO-LMM-DIO, LMDBIO-LMM-DM, LMDBIO-LMM-DIO-PROV, LMDBIO-LMM-DIO-PROV-COAL, and LMDBIO-LMM-DIO-PROV-COAL-STAG. An overview of these six optimizations and the problems addressed by each optimization is shown in Table 1.

We have separated the different optimizations to better understand the impact of each optimization individually. The details of each optimization are elaborated in the following subsections. As mentioned earlier, LMDBIO-LMM and LMDBIO-LMM-DM were previously published in References [40] and [39], respectively. For completeness, we briefly discuss their designs and present

Table 1. Optimization Overview

Library	Optimization	Reducing Interprocess Contention	Using Explicit I/O	Eliminating Sequential Seek	Managing I/O Size	Reducing I/O Randomization
LMDB	—					
LMDBIO	LMM	✓				
	LMM-DM	✓		(partial)		
	LMM-DIO	✓	✓			
	LMM-DIO-PROV	✓	✓	✓		
	LMM-DIO-PROV-COAL	✓	✓	✓	✓	
	LMM-DIO-PROV-COAL-STAG	✓	✓	✓	✓	✓

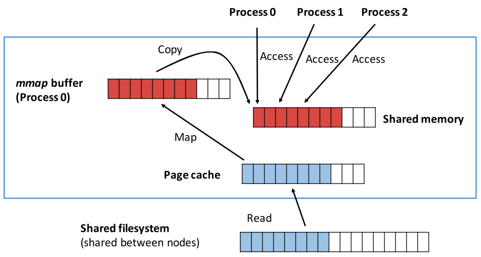


Fig. 10. LMDBIO-LMM design.

additional new results and analysis for these two optimizations. The remaining four optimizations are totally new.

4.1 LMDBIO-LMM

As discussed in Section 3.3.1, mmap has an unfortunate interference with the Linux CFS process scheduler causing I/O contention between the different processes on the node. In order to minimize interprocess contention, LMDBIO-LMM seeks to limit the number of parallel processes executing mmap simultaneously within the same node. That is, LMDBIO-LMM attempts to balance I/O parallelism with interprocess contention.

LMDBIO-LMM involves two phases: an initialization phase and a data-reading phase. In the initialization phase, LMDBIO-LMM selects a subset of processes on each node as "root" processes. These root processes perform the actual data I/O reads on behalf of all processes on the node using mmap. To do so, LMDBIO-LMM uses MPI-3 to create a shared-memory buffer for processes on the same node. Once the buffer is set up, the root processes can open the LMDB database using mmap. In the data-reading phase (illustrated in Figure 10), the root processes read data from the shared filesystem and copy it into the preallocated shared-memory buffer. Once the data are available in the shared-memory buffer, all processes can directly access the buffer.

This optimization can significantly reduce interprocess contention, and consequently the number of context switches, since the number of processes that the I/O handler has to manage is small. We note, however, that this approach also reduces I/O parallelism and thus can cause some degradation in performance. Based on our experiments with up to 36 cores on the node, using just one root process was sufficient up to 16 cores, and using two root processes was sufficient up to the maximum number of cores on the node, to achieve the best performance.

We note that several other approaches exist to implement shared memory between processes, for example, using /dev/shm or using mmap. These methods are portable on POSIX-based systems.

However, we have chosen MPI-3's shared-memory implementation for two reasons: (1) the MPI implementation can choose the most suitable shared-memory model for a particular system including non-POSIX models such as XPMEM [56] and PiP [20], providing a minor performance advantage in some cases, and (2) we already use MPI for other communication in our framework, thus making MPI-3 shared memory a more natural model to be used in our framework.

4.2 LMDBIO-LMM-DM

As discussed in Section 3.3.2, because of its data layout, LMDB cannot access a given data record without parsing through all of the tree's branch and leaf nodes in the path to the record. This process results in a significant amount of redundant data I/O among the different root processes. Although we cannot completely eliminate this redundant I/O without more information than what LMDB currently provides, LMDBIO-LMM-DM aims to improve this situation by coordinating between the root processes and speculatively performing parallel I/O to improve the read performance. Loosely speaking, with LMDBIO-LMM-DM, each root process tries to "guess" the location of the data records that it needs and speculatively prefetches the corresponding data into memory. Once the seek operation (which is still sequential) is carried out, if the initial guess of the data location was correct, the required data will already be in memory. If the guess was incorrect, then the correct data will need to be read from the filesystem at that point; but we can use the newly acquired record location information to improve our guess for the next iteration.

4.2.1 Portable Cursor Representation. As explained in Section 2.2.2, the data record position indicator in LMDB is referred to as the "cursor." A cursor is not a trivial file offset but is instead a set of pointers to different pages in the database. In general, because pointers to a virtual address space are private to each process, the cursor is not always portable across processes. In practice, however, the `mmap` space is a contiguous virtual address space, and the pointers in a cursor point only to locations within the B+ tree. Thus, if we could align the B+ tree used on all the processes to start at exactly the same virtual address (i.e., use a symmetric address space for the database `mmap` on all processes), the cursors would automatically become portable across the different processes.

Based on this observation, we designed a simple symmetric memory allocator using the following algorithm. The first root process picks a random virtual address location in its 64-bit address space and tries to `mmap` the database to this location. If the `mmap` is successful, then it means that the virtual address location was unused so far. It then broadcasts this address to the remaining root processes, which try to `mmap` the database file at the exact same virtual address location. If that virtual address location was previously unused on all of the root processes, then they will all succeed in this operation. If any one of them failed, then it will indicate so in an MPI Allreduce operation. All the root processes will then discard that virtual address location and repeat the process. If after a few iterations no common virtual address location can be found, then this optimization is abandoned. In practice, however, we can find a symmetrical address space in just one to two iterations.

An alternative approach to achieve the same outcome would be to modify the LMDB implementation such that the user could pass in a pointer offset that would be used for the database parsing. Such an approach would be equally effective, although we feel that the symmetric memory allocation technique that we use in this article is a less intrusive (to the LMDB implementation) and generally more elegant solution to the problem.

Once the database is mapped to a symmetrical address space, the cursor is immediately portable. To send the cursor from one process to another, we simply serialize the internal content of the cursor data structure to a memory buffer and send it to the other processes by using `MPI_Send` and `MPI_Recv`, as shown in Figure 11. Each root process can then exploit the information given

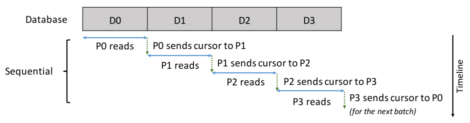


Fig. 11. LMDBIO-LMM-DM design: Sequential I/O and cursor handoff.

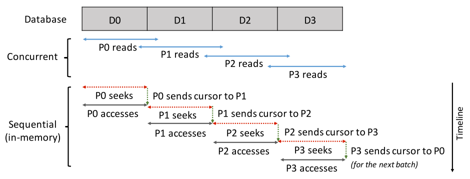


Fig. 12. LMDBIO-LMM-DM design: Parallel I/O and in-memory sequential seek.

by other root processes to seek to their respective starting records without having to parse the database file again.

4.2.2 Speculative Parallel I/O. To enable parallelism in I/O, in LMDBIO-LMM-DM (Figure 12) the root processes attempt to estimate the location of the data records that they need and speculatively read this data in parallel. Because of the complex structure of the LMDB database, accurately estimating the location of the data records is a complex task. In LMDBIO-LMM-DM, we use a history-based evolving estimation. Specifically, the first root process parses the structure of the initial part of the database. All root processes then use this partial information to extrapolate the structure of the rest of the database and estimate the locations of the data records that they need. When the sequential seek completes, each root process knows whether its estimate was correct. If the estimate was incorrect, then the root process will use the additional information on the structure of the database that it gathered in this iteration, to improve its estimate for the next iteration. With this approach, our estimation accuracy, in terms of missing required data pages, converges quickly—within the first few iterations. However, LMDBIO-LMM-DM tends to overestimate the data that are needed and usually prefetches more data than it actually needs.

In LMDBIO-LMM-DM, speculative I/O is done concurrently. Once that is done, each root process seeks for its data records and hands off the final cursor position to the next root process when it is done. The final cursor position of a root process is the starting cursor position of the next root processes, thus avoiding any redundant I/O in the seek process. If the speculative I/O estimation was accurate, even though the seek is still sequential, then it will be done entirely in memory. However, if the estimation was not accurate, then the seek will still perform the data I/O read that it was originally performing, but we would have unnecessarily fetched additional data that we do not need. In fact, in the worst case, we could end up reading twice as much data as needed.

We note that the proposed history-based speculative I/O technique can be used in conjunction with other in-memory data shuffling approaches, where shuffling is done after the I/O has completed. In other words, as long as data I/O is structured and iterative (which is true for most deep learning frameworks), one can take advantage of the proposed history-based speculative I/O technique.

4.3 LMDBIO-LMM-DIO

As shown in Section 3.3.3, the implicit I/O model used by LMDB (through `mmap`) can have a significant performance impact on data I/O read. In this section, we present LMDBIO-LMM-DIO, an approach to extend LMDBIO-LMM to use direct I/O (through POSIX I/O).

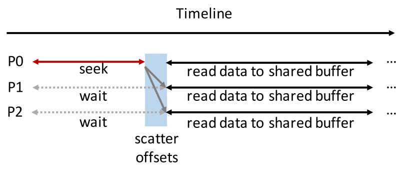

Timeline
P0
P1
P2
seek
wait
wait
read data to shared buffer
read data to shared buffer
read data to shared buffer
scatter offsets
...

Fig. 13. LMDBIO-LMM-DIO design: Sequential seek.

The basic working model of LMDBIO-LMM-DIO is similar to that of LMDBIO-LMM. That is, LMDBIO-LMM-DIO still has a small subset of the processes designated as root processes on each node that, in turn, `mmap` the LMDB database into their respective address spaces. And, like LMDBIO-LMM, LMDBIO-LMM-DIO also creates a shared-memory buffer between all processes on the node to share the data that the root processes read from the database. The primary difference between LMDBIO-LMM and LMDBIO-LMM-DIO is that the latter uses direct POSIX I/O for performing the actual read of the data. That is, once the location of the data record in the database has been identified, LMDBIO-LMM-DIO does not use `mmap` to copy the data into the shared buffer. Instead, it computes the virtual address offset of the data record address compared with the virtual address of the start of the database and uses that offset to directly read the data using the POSIX I/O `pread` function.

We note, however, that LMDBIO-LMM-DIO does little to improve the sequential seek for locating the database records and continues to use `mmap`, just like LMDB and LMDBIO-LMM. Thus, in LMDBIO-LMM-DIO, the seek path and the actual data read path are disjoint: the seek goes through `mmap`, whereas the actual data read goes through POSIX I/O. Because of this separation of paths, performing the seek on the same process as the one that does the actual data read is not too beneficial for LMDBIO-LMM-DIO. Therefore, we use a single process to seek through the entire database and obtain offsets and sizes for all the data records that will be used in the following training iterations. Performing the seek on a single process has the advantage of avoiding the redundant data I/O among the various root processes, although it does not help with the sequential nature of the seek. Once the seek is complete, the offsets and sizes are distributed to the other root processes, as illustrated in Figure 13.

4.4 LMDBIO-LMM-DIO-PROV

LMDBIO-LMM-DM attempts to address the serialization in data I/O by performing speculative parallel reads. While that approach can be effective in reducing the redundant data accesses in some cases, it is still an approximation technique and can cause a significant increase in the data I/O if the approximation is incorrect. Unfortunately, no way exists to precisely estimate the location of the data records without the sequential seek. The reason is that the layout of the LMDB database depends not only on the content of the database but also on the way the database was created. This information is not natively stored in the LMDB database file.

In this section, we propose LMDBIO-LMM-DIO-PROV, a technique that provides a more elegant alternative to address the serialization in data I/O, compared with LMDBIO-LMM-DM, by completely and deterministically eliminating the sequential seek restriction of LMDB. The catch, however, is that LMDBIO-LMM-DIO-PROV requires the user to provide more information than what the LMDB database natively provides. We refer to this information as the "database provenance information."

4.4.1 LMDB Database Creation. Before explaining the provenance information that we require for LMDBIO-LMM-DIO-PROV, we briefly summarize how the LMDB database creation process works. LMDB employs a multiversion concurrency control policy to guarantee data integrity and

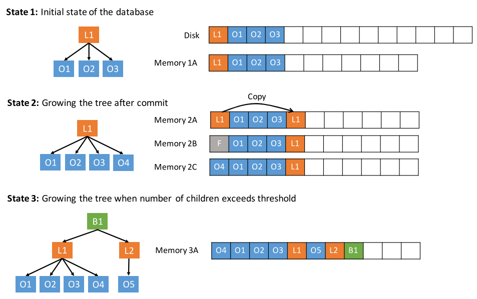

State 1: Initial state of the database
L1
O1 O2 O3
Disk
Memory 1A
State 2: Growing the tree after commit
Copy
Memory 2A
Memory 2B
Memory 2C
F
O4
State 3: Growing the tree when number of children exceeds threshold
B1
L2
O5
Memory 3A

Fig. 14. LMDB database creation example.

reliability in the multiple-readers, single-writer model. This model allows a reader to read a valid snapshot of the database without acquiring a lock. Locking is required only when writing to the database. To provide concurrency, LMDB adopts a "copy-on-write" policy on the database file where new data are written to the file without overwriting or relocating old data. Any change to existing data in the database file, however, will be applied to a copy of that data. In other words, LMDB will copy existing data to a new location and apply changes to the new resource when a write occurs. This policy ensures that data in the file are always in a valid state.

Since LMDB is a transactional database, it operates at the granularity of transactions. When new data are added to the database, it will be written to permanent storage only when that transaction is committed. During the *commit*, the layout of the database file is modified. Resources that have been modified will be duplicated. For LMDB, these modifiable resources are the branch pages and leaf pages. When the tree structure changes, some of the existing branch and leaf pages are modified to update their connectivity to other pages (i.e., neighboring and children pages). With LMDB, the tree grows in a bottom-up manner where pages that contain data (i.e., key-value pairs) are added first. Each leaf/branch page has a limit on the number of children that it can have. New pages are added to the tree when the number of children in that page has reached that limit.

An example of LMDB database creation is demonstrated in Figure 14. $State1$ shows an example initial state of the database where all prior transactions have been successfully committed (i.e., the previous data are in disk and is identical to the content in memory $1A$). In $State2$, data page $O4$ is appended to the tree, causing the leaf page $L1$ to be modified. In this case, LMDB copies $L1$ to a new location before modifying it (memory $2A$). Then, the old memory location of $L1$ is marked as free (memory $2B$). After that, $O4$ will be added to the database file. In this example, we assume that $O4$ can fit in the free memory region (memory $2C$). Otherwise, it will be appended to the end of the memory area. Suppose that the transaction has not yet been committed. $State3$ shows how the tree grows in the case that the number of children of $L1$ exceeds its limit (i.e., 4 children). In that case, a new leaf page ($L2$) and a new branch page ($B1$) are added to the tree.

4.4.2 LMDB Provenance Information. As explained earlier, the location of data records in the LMDB database cannot be determined by using only the natively available information in the database metadata. Fortunately, LMDB uses a deterministic algorithm to create the B+ tree database. Thus, with additional information about the database creation (i.e., the database provenance information), we can dynamically compute the database layout. This computation allows us to precisely deduce the accurate location of each database record, completely eliminating the seek.

In LMDBIO-LMM-DIO-PROV, we propose maintaining a separate auxiliary file for each LMDB database file that contains the following provenance information: (1) frequency that the transactions are committed in, (2) maximum number of records that a leaf node can contain, (3) maximum number of children that a branch node can have, (4) size of each data record, (5) order in which the data records are added, and (6) number of LMDB metadata pages. This provenance information can be collected either when the database is being generated or later as a one-time postprocessing of the database. We note that the proposed provenance information is typically small compared with the database itself (i.e., a few hundred bytes).

Once such provenance information is available, its usage in LMDBIO-LMM-DIO-PROV is straightforward. Each root process computes the offsets of all the data records that it needs by following the algorithm that is adopted by LMDB for creating the database. This computation adds negligible cost compared with the cost of the I/O itself. Once the offsets are calculated, the actual data I/O is done through POSIX I/O, similar to LMDBIO-LMM-DIO. We note that without the additional provenance information LMDBIO-LMM-DIO-PROV cannot be used and we would need to fall back to LMDBIO-LMM-DM for improving the sequential seek.

An important aspect to note here is that any improvement to the seek time needs to be taken with a grain of salt. For example, in cases where the application iterates over the data for a very large number of epochs, one might be able to simply store the database offsets in memory to be used in later epochs. However, such an approach raises a few concerns that must be kept in mind.

(1) It is practical only if the number of data samples is an exact multiple of the number of processes. Any offset in this would mean that the data samples computed by a given process would not be exactly the same in every epoch. In cases where the number of data samples is not an exact multiple of the number of processes, one can divide the data samples as evenly as possible across the different processes and then treat the remainder separately. While this might seem like an enticing possibility, however, we note that it would change the semantics of the LMDB model. With its current semantics, the database is treated as a circular collection of records, so one would return to the first record after the last record has been read. This allows applications using LMDB to be guaranteed that the read of a block of records always returns the full block of records and never a partial block. If we treat the remainder separately, then those semantics would no longer be true. As a consequence, such a change in the semantics would, in turn, require intrusive modification to the entire LMDB ecosystem.

(2) The efficiency of this approach depends on how many epochs of training are used. For cases where the database is extremely large, some algorithms tend to rely on a single-pass analysis (i.e., the database is read only once) or on analyzing the data using just a few epochs. In such cases, the seek overhead can still be significant, and the provenance information that we proposed in this section can help.

(3) While data could theoretically be streamed from an online source, such a model is not as common today. Training datasets are typically stored in persistent files and used for training with multiple models or multiple parameter settings.

(4) Similarly, while splitting the dataset into a large number of files is possible (e.g., one file per process), so as to completely avoid seeking, such practice is strongly discouraged on most large supercomputing systems. The reason is that reading from a large number of files can easily overwhelm the metadata server, causing the filesystem to suffer from significant performance loss or even crash [33].

4.5 LMDBIO-LMM-DIO-PROV-COAL

As mentioned in Section 3.3.4, as the parallelism used by the deep learning algorithm increases, the size of the subbatch used by each process decreases. In the extreme case, when the parallelism used for the deep learning training is as large as the number of available data samples in each batch, each process would need a single data sample in each iteration. Thus, each root process would end up reading smaller blocks of data. As an example, if we consider the CIFAR10-Large database, when using 9,216 processes with a batch size of 18,432, each process would need just two data samples in every iteration, where each data sample would be 4KB (3KB actual data). Even if we use a single root process on each node, the root process would perform an I/O of 288KB in every iteration. Most filesystems, however, require much larger block sizes (typically in multiple megabytes) for optimal I/O performance.

We tackle this issue in LMDBIO-LMM-DIO-PROV-COAL by allowing it to assume the iterative nature of deep learning applications. That is, even though a single iteration does not require too much data, if we can coalesce the data required in multiple iterations, we can increase the block size used in each I/O operation. With LMDBIO-LMM-DIO-PROV-COAL, each root process reads a contiguous chunk of data that is large enough to saturate the I/O performance of the filesystem. LMDBIO-LMM-DIO-PROV-COAL tunes the I/O block size that it uses so as to limit the amount of memory that it consumes for I/O (kept at 2.5GB in our experiments). Thus, as the parallelism in the deep learning training increases, it fetches data required for more iterations within a single I/O operation.

4.6 LMDBIO-LMM-DIO-PROV-COAL-STAG

Our sixth optimization technique, LMDBIO-LMM-DIO-PROV-COAL-STAG, addresses the I/O randomization problem presented in Section 3.3.5. The general idea used by LMDBIO-LMM-DIO-PROV-COAL-STAG is to limit the number of I/O operations that are issued simultaneously so as to minimize such randomization while maintaining sufficient parallelism to maximize I/O performance. To achieve this goal, we use a technique called I/O staggering.

In this technique, the root processes are divided into multiple groups of the same size. Root processes that access segments of the file that are close to each other are grouped together. Once the grouping is done, we allow one group of root processes (referred to as a staggering group) to access the file concurrently while the remaining groups wait for the previous groups to complete their I/O. We use a token-passing approach: a process can perform I/O only when it has a token. Suppose the staggering group size is n. Then there are n tokens, with the root processes in each group labeled from 0 to $n-1$. When a root process is done with its I/O, it passes on its token to the root process in the next group with the same label as itself. We simply use `MPI_Send` and `MPI_Recv` to pass tokens between processes.

We note that the staggering size needs to be carefully selected. A very large staggering size would lead to increased randomization, while a very small staggering size would lead to reduced parallelism in I/O. We empirically evaluated the best staggering sizes for different number of processes and used them for our experiments.

We also note that more elegant approaches for managing I/O staggering exist than those we propose in this article. One example would be to use the POSIX file-locking mechanism. That is, each group would attempt to lock the database file; and once it acquired the lock, it would perform the actual I/O. This approach would achieve the same outcome as our token-passing approach and would further remove the unnecessary and artificial ordering restriction that the proposed token-passing approach forces. Unfortunately, most distributed filesystems (e.g., NFS) do not provide strict POSIX semantics, including `fcntl()` and file locking [51], thus making its portability questionable. Therefore, we used the proposed token-passing approach as a workaround to this particular shortcoming of some filesystems.

5 EXPERIMENTS AND RESULTS

In this section, we compare the performance of LMDBIO with that of LMDB. In Section 5.1 we evaluate the performance of each of the proposed optimizations using simple microbenchmarks.

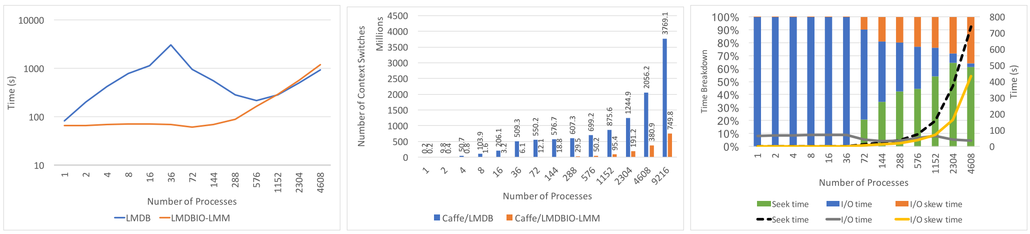


Fig. 15. LMDBIO-LMM performance analysis: (a) read performance compared with LMDB; (b) context switches compared with LMDB; (c) total read time breakdown.

The purpose of this evaluation is to understand the benefits and shortcomings of each optimization without diluting the results with other computation that would happen in a typical deep learning application. In Sections 5.2 and 5.3 we use strong- and weak-scaling experiments to compare the performance of Caffe/LMDBIO with that of the original Caffe/LMDB. The purpose of this evaluation is to understand the impact of LMDBIO on the overall performance of the Caffe deep learning framework on real datasets. Our experiments use the datasets, networks, and supercomputer systems described in Section 3.1.

5.1 Microbenchmark Evaluation and Analysis

To compare the performance of LMDBIO with that of LMDB, we used a simple microbenchmark that emulates the I/O behavior of Caffe. Our microbenchmark is designed to use LMDB or LMDBIO to perform data I/O. It performs iterative data I/O, similar to what Caffe would, but it does not perform any of the computation associated with DNN training. The experiments in this section were performed on Bebop, with 9.4 million images of the CIFAR10-Large database; the batch size was set to 18,432 images.

5.1.1 LMDBIO-LMM Performance Analysis. Figure 15(a) shows a comparison of the read performance of LMDB and LMDBIO-LMM. We see that LMDBIO-LMM outperforms LMDB by up to 43.77-fold when the number of processes is smaller than or equal to 1,152. This improvement in performance is attributed to reduced interprocess contention. For very large numbers of processes, LMDBIO-LMM performs slightly worse than LMDB, because we used a single root process on each node in all our experiments for consistency. Increasing to two root processes per node, when running on a large number of processes, improves the LMDBIO-LMM performance enough to address this degradation, although those numbers are not shown in this graph.

To further analyze the reduced contention, we show in Figure 15(b) the number of context switches that occur with LMDBIO-LMM compared with LMDB. We can see that in some cases LMDBIO-LMM achieves more than an 83-fold reduction in the number of context switches compared with LMDB, thus demonstrating that our technique to localize `mmap` can significantly reduce interprocess contention.

Figure 15(c) shows the breakdown of the total read time, divided into the seek time (which is still sequential), the I/O time, and the I/O skew time. From the graph, we observe that although the I/O time itself is fairly small, a significant amount of time is spent in the sequential seek and in the skew among processes where some processes are waiting for other processes to catch up. These two are related. The skew is caused by the data-seeking process in LMDBIO-LMM.

To further understand this, we performed an additional analysis using a small benchmark that contains only the seek part and plotted it against the root process's MPI rank, as shown in Figure 16(a). We observe that the seek time increases with the process rank. This phenomenon

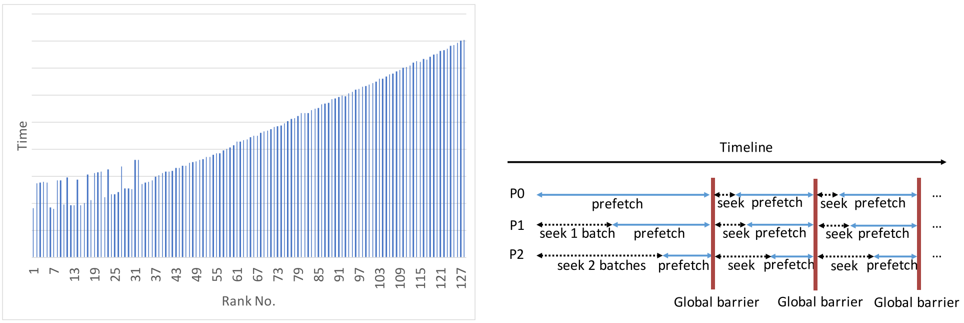


Fig. 16. LMDBIO-LMM: (a) seek time vs. reader's rank number and (b) Mmap's data prefetching and data seeking.

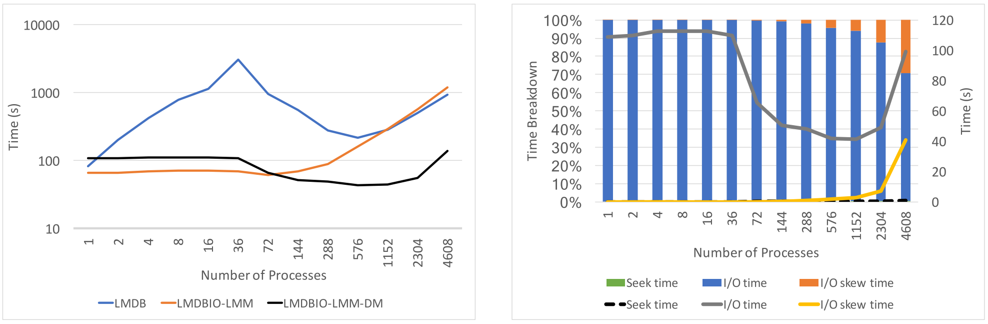
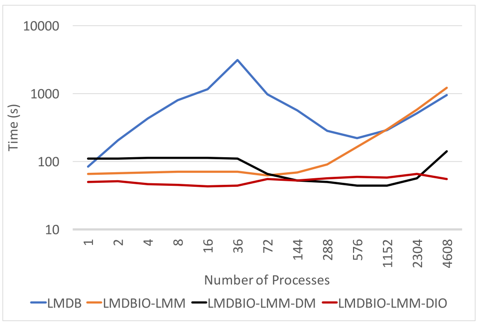


Fig. 17. LMDBIO-LMM-DM performance analysis: (a) read performance compared with LMDB and LMDBIO-LMM; (b) read time breakdown.

is a subtle outcome of the data prefetching that is performed within mmap, as demonstrated in Figure 16(b). Specifically, after opening a database, each reader process will initialize the cursor by seeking to its corresponding starting location in the database. The amount of seek that is performed by each reader is $rank \times B$ ($rank$ denotes the reader process's rank, and B denotes the batch size), which is not uniform among the different processes. Because of the bulk synchronous nature of the computation, however, some processes end up waiting in the synchronization longer than others. The processes that wait in the synchronization longer thus have a better opportunity to prefetch data that they would need in the next iteration. For instance, $P0$ spends a large part of its time prefetching data for the next iteration, while $P2$ gets very little time to prefetch. This prefetching, in turn, helps $P0$ with its seek in the next iteration, thus causing it to complete faster than the other processes in that iteration as well, so the cycle continues and results in large skew.

5.1.2 LMDBIO-LMM-DM Performance Analysis. Figure 17(a) compares the performance of LMDBIO-LMM-DM with that of LMDB and LMDBIO-LMM. On a single node, LMDBIO-LMM-DM does not achieve any performance improvement compared with LMDBIO-LMM, because it utilizes the same general principle as LMDBIO-LMM to avoid interprocess contention. In fact, the additional data I/O performed by LMDBIO-LMM-DM hurts performance somewhat, causing it to achieve slightly worse performance compared with LMDBIO-LMM. When using multiple nodes, however, LMDBIO-LMM-DM performs better than both LMDB and LMDBIO-LMM, outperforming LMDB by 6.7-fold on 4,608 processes. This improvement is attributed primarily to the reduction in redundant data read during the seek and to the speculative parallel I/O.



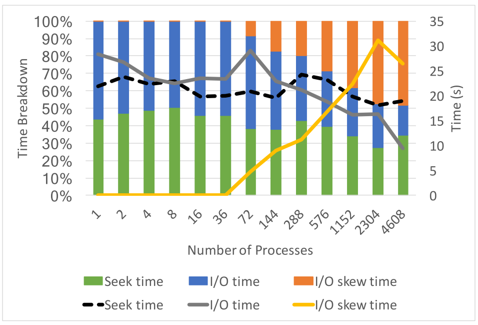


Fig. 18. LMDBIO-LMM-DIO performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, and LMDBIO-LMM-DM; (b) read time breakdown.

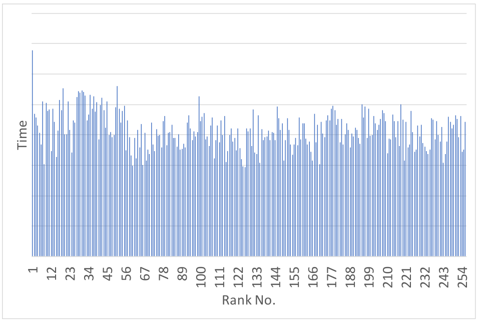


Fig. 19. LMDBIO-LMM-DIO: I/O skew analysis.

Figure 17(b) shows the breakdown of the LMDBIO-LMM-DM read time. The data seek in LMDBIO-LMM-DM is highly efficient compared with that of LMDB and LMDBIO-LMM. The seek in LMDBIO-LMM-DM takes less than 1% of the read time, compared with LMDBIO-LMM, which spends nearly 60% of the read time in seek. The better performance with LMDBIO-LMM-DM is mainly because the seek is performed in memory. In fact, in some cases LMDBIO-LMM-DM improves the seek time by nearly 1,741-fold compared with LMDBIO-LMM.

5.1.3 LMDBIO-LMM-DIO Performance Analysis. In this subsection, we compare the read performance of LMDBIO-LMM-DIO with that of LMDB, LMDBIO-LMM, and LMDBIO-LMM-DM, as shown in Figure 18(a). LMDBIO-LMM-DIO achieves better performance than the other approaches in almost all cases primarily because of its usage of POSIX I/O for data reading in place of `mmap`. In some cases, however, LMDBIO-LMM-DM slightly outperforms LMDBIO-LMM-DIO. The reason is that LMDBIO-LMM-DIO does nothing to optimize the seek, a process that can take a significant amount of time. In fact, as shown in our read time breakdown in Figure 18(b), the seek in LMDBIO-LMM-DIO takes up nearly 20% of the read time. Nevertheless, LMDBIO-LMM-DIO outperforms LMDB by 17.18-fold on 4,608 processes.

We note that LMDBIO-LMM-DIO still suffers from data skew, similar to LMDBIO-LMM and LMDBIO-LMM-DM. Unlike LMDBIO-LMM, however, this skew is not because of data prefetching, which we verified by measuring the I/O time on each reader rank as shown in Figure 19. Instead, the skew is due to other serialization in the data I/O such as that related to I/O randomization.

5.1.4 LMDBIO-LMM-DIO-PROV Performance Analysis. Figure 20(a) compares the performance of LMDBIO-LMM-DIO-PROV with that of LMDB, LMDBIO-LMM, LMDBIO-LMM-DM, and

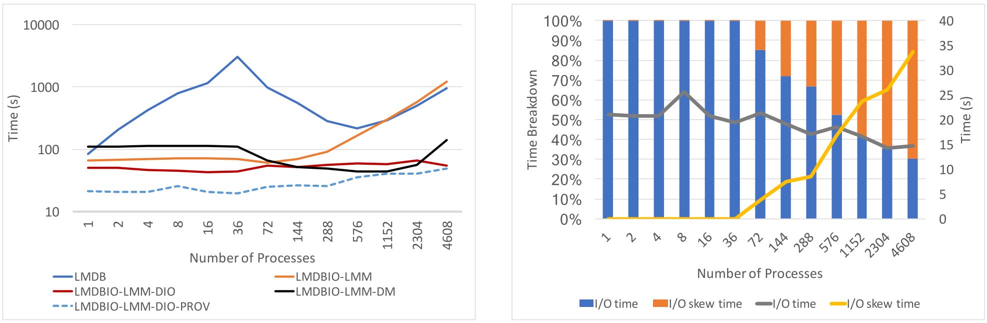


Fig. 20. LMDBIO-LMM-DIO-PROV performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, and LMDBIO-LMM-DM; (b) read time breakdown.

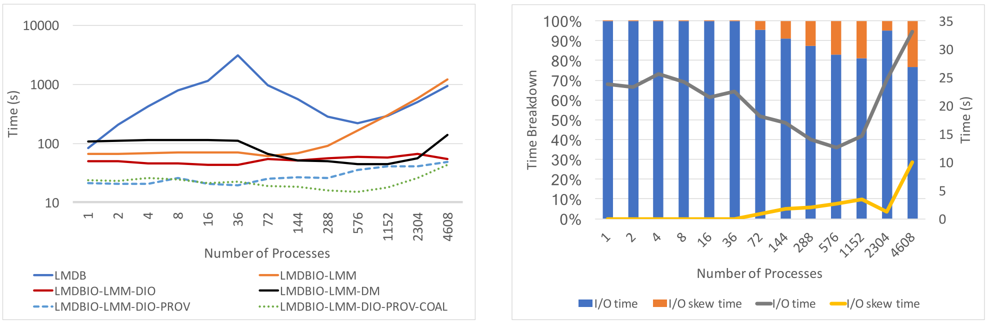


Fig. 21. LMDBIO-LMM-DIO-PROV-COAL performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, LMDBIO-LMM-DM, and LMDBIO-LMM-DIO-PROV; (b) read time breakdown.

LMDBIO-LMM-DIO. LMDBIO-LMM-DIO-PROV consistently outperforms all these approaches, achieving 19.44-fold improvement in performance on 4,608 processes compared with LMDB. The performance improvement in LMDBIO-LMM-DIO-PROV is attributed to its elimination of the sequential seek to access the database records. This improvement in performance highlights the importance of the database provenance information in scalable deep learning.

Despite the impressive gains in performance, however, LMDBIO-LMM-DIO-PROV still suffers from some shortcomings that cause its I/O time to increase as the number of processes increases. We plotted this behavior in Figure 20(b). This figure shows that a significant portion of the I/O time is taken by the skew between the different processes, which is an artifact of the I/O randomization described in Section 3.3.5.

5.1.5 LMDBIO-LMM-DIO-PROV-COAL Performance Analysis. Figure 21(a) compares the performance of LMDBIO-LMM-DIO-PROV-COAL with that of LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, and LMDBIO-LMM-DIO-PROV and demonstrates that LMDBIO-LMM-DIO-PROV-COAL consistently achieves the best performance among all these approaches. In fact, LMDBIO-LMM-DIO-PROV-COAL outperforms LMDB by 21.86-fold on 4,608 processes. The primary performance gain in LMDBIO-LMM-DIO-PROV-COAL comes from the fact that it optimizes the I/O block size by coalescing data required in multiple iterations into fewer I/O operations. This approach better utilizes the I/O subsystem, resulting in improved performance.

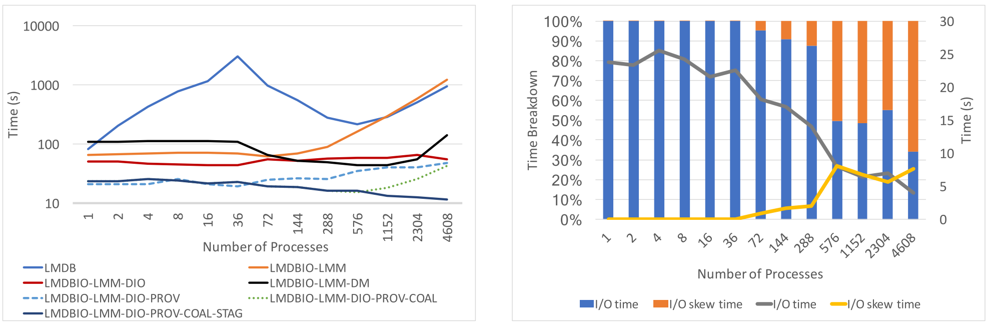


Fig. 22. LMDBIO-LMM-DIO-PROV-COAL-STAG performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, LMDBIO-LMM-DM, LMDBIO-LMM-DIO-PROV, and LMDBIO-LMM-DIO-PROV-COAL; (b) read time breakdown.

A breakdown of the read time in Figure 21(b) shows that LMDBIO-LMM-DIO-PROV-COAL reduces the skew time to around 25% of the total I/O time. While most of the time is now taken by the actual read operation, room for improvement still remains.

5.1.6 LMDBIO-LMM-DIO-PROV-COAL-STAG Performance Analysis. Figure 22(a) compares the performance of LMDBIO-LMM-DIO-PROV-COAL-STAG with that of LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, LMDBIO-LMM-DIO-PROV, and LMDBIO-LMM-DIO-PROV-COAL. The figure shows that LMDBIO-LMM-DIO-PROV-COAL-STAG performs the same as or better than all the other techniques, outperforming LMDB by 81.05-fold on 4,608 processes. This improvement in performance is attributed to the reduced I/O randomization in LMDBIO-LMM-DIO-PROV-COAL-STAG.

Our analysis of the I/O time breakdown is shown in Figure 22(b). This figure, however, can be a bit misleading. While it shows a significant increase in I/O skew compared with LMDBIO-LMM-DIO-PROV-COAL, this skew is intentional. That is, because LMDBIO-LMM-DIO-PROV-COAL-STAG groups the root processes and forces only one group to be actively performing I/O at a given point in time, it artificially appears that there is high I/O skew time. Nevertheless, LMDBIO-LMM-DIO-PROV-COAL-STAG comprehensively outperforms all the other presented techniques.

5.2 Strong-Scaling Performance Evaluation of Caffe Deep Learning Training

In this section, we evaluate the performance of complete deep learning training executions using Caffe/LMDB and Caffe/LMDBIO. We performed our experiments on two platforms (i.e., Blues and Bebop), using three different datasets (i.e., CIFAR10-Large and ImageNet on Blues, and CIFAR10-Large and ImageNet-Large on Bebop). On Blues, we compared Caffe/LMDB with two of the proposed optimization techniques: Caffe/LMDBIO-LMM and Caffe/LMDBIO-LMM-DM. On Bebop, we compared Caffe/LMDB with all the proposed optimization techniques. As described in Section 3.1, all of our experiments so far used single-threaded MKL while achieving parallelism on the node using multiple processes. An alternate approach that one might consider is to use a single process on each node but to take advantage of intranode parallelism through the multithreaded Intel MKL library, so as to utilize the cores better. While at first blush that seems promising, such an approach would, by definition, only utilize the cores on the node during MKL operations, while the rest of the computational workflow would remain sequential, thus wasting cores. We have included the multithreaded MKL version (denoted LMDB-MT-MKL) in the experiments in this section for completeness, despite its known inefficiency especially when the number of cores is large. We note that this experiment was conducted only on Bebop.

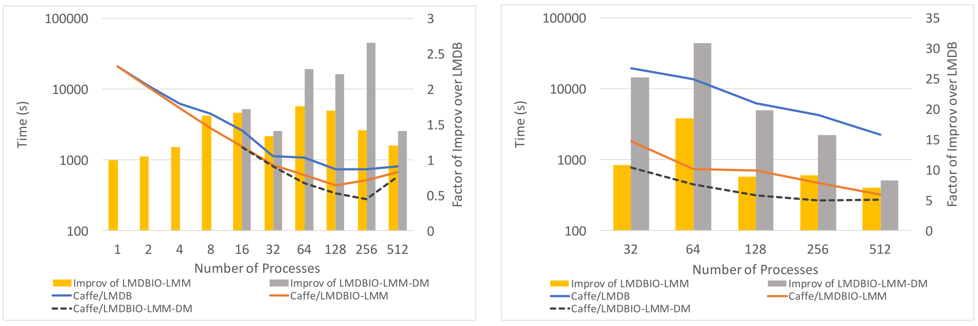


Fig. 23. Strong scaling on the Argonne computing cluster Blues using (a) CIFAR10-Large and (b) ImageNet.

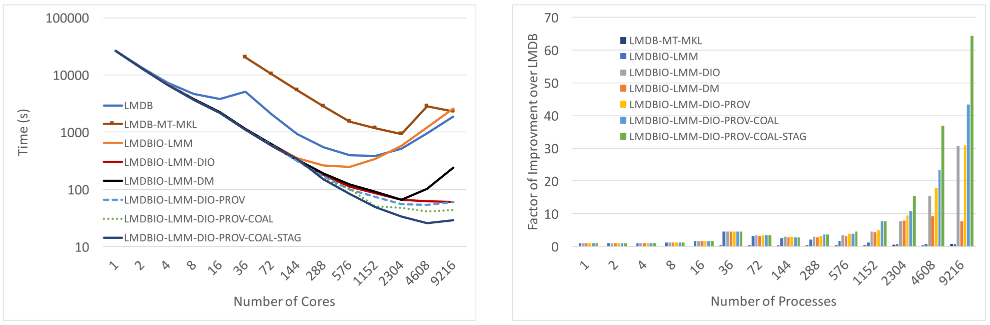


Fig. 24. Strong scaling using CIFAR10-Large on Bebop: (a) total execution time and (b) factor of improvement over Caffe/LMDB.

Figure 23(a) shows the strong-scaling results for training Caffe with the CIFAR10-Large dataset on Blues using a batch size of 4,096 for 1,024 iterations. Caffe/LMDBIO-LMM and Caffe/LMDBIO-LMM-DM outperform Caffe/LMDB in all cases, achieving 1.21-fold and 1.41-fold improvements on 512 processes, respectively. For ImageNet, we also used the same batch size as CIFAR10-Large and performed a total of 32 training iterations. Figure 23(b) shows 7-fold and 8.3-fold improvements for ImageNet with Caffe/LMDBIO-LMM and Caffe/LMDBIO-LMM-DM, respectively. The improvements are attributed to the reduced interprocess contention in both approaches and the improved seek time mitigation in Caffe/LMDBIO-LMM-DM.

Figure 24 shows strong-scaling results for CIFAR10-Large on Bebop. We used a batch size of 18,432 and performed 512 training iterations. Figure 24(a) shows the execution time of Caffe with the different frameworks, and Figure 24(b) shows the factor of improvements compared with Caffe/LMDB. The general performance trend observed in the figures is similar to that on Blues, with Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG achieving nearly 65-fold performance improvement over Caffe/LMDB on 9,216 processes.

Figure 25 shows strong-scaling results for ImageNet-Large on Bebop. We used a batch size of 18,432 and performed 128 training iterations. Figure 25(a) shows the execution time of Caffe with the different frameworks, and Figure 25(b) shows the factor of improvements compared with Caffe/LMDB. The general performance trend observed in the figures is similar to that with CIFAR10-Large, although the performance improvements are smaller. The reason is that the structures of the two datasets are different. Specifically, ImageNet-Large contains larger data sample sizes (192KB for ImageNet-Large compared with 3KB for CIFAR10-Large), resulting in significantly different I/O characteristics. For example, header access is a small fraction of I/O for

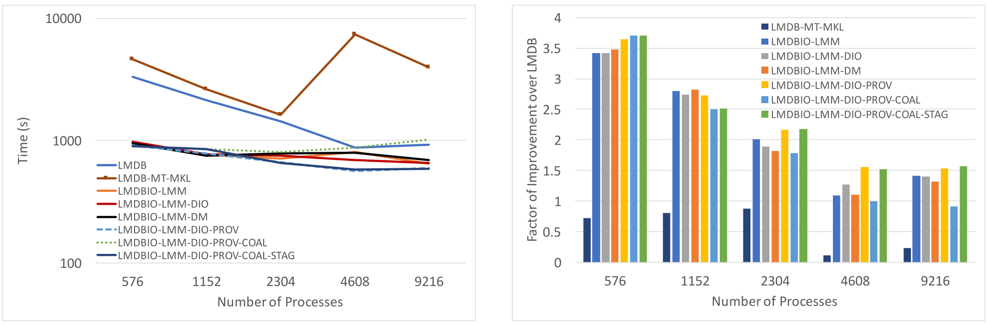


Fig. 25. Strong scaling using ImageNet-Large on Bebop (a) total execution time and (b) factor of improvement over Caffe/LMDB.

ImageNet-Large, whereas it is a significant portion of I/O for CIFAR10-Large; in other words, the header and the data are on the same physical page in memory for CIFAR10-Large, so accessing one without the other is difficult. Another example is that of I/O randomization, which has a significantly higher impact on ImageNet-Large than it does on CIFAR10-Large because of the larger sizes of the data samples, making each batch of samples typically larger than the I/O request size of the filesystem.

An interesting trend that we observe is that for the ImageNet-Large dataset, Caffe/LMDBIO-LMM-DIO-PROV-COAL performs worse than other techniques, particularly when the number of processes is large. The reason is that although all techniques other than Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG suffer from I/O randomization, Caffe/LMDBIO-LMM-DIO-PROV-COAL is particularly susceptible, because this technique actively increases the amount of data that each process reads through coalescing. Thus, in Caffe/LMDBIO-LMM-DIO-PROV-COAL, if I/O requests arrive out of order at the I/O server, the data segments accessed by these requests are especially far away for ImageNet-Large because of the large size of its data samples, thus causing further degradation in performance. As expected, once I/O staggering is applied in Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG, this performance degradation goes away. In fact, Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG outperforms all other approaches, gaining approximately 1.6-fold performance over Caffe/LMDB on 9,216 processes.

5.3 Weak-Scaling Performance Evaluation of Caffe Deep Learning Training

Apart from the strong-scaling experiments shown so far, we also conducted a weak-scaling evaluation of LMDBIO on Bebop. Here, we increase the total batch size (i.e., the total number of images processed by all processes together in each iteration) by k times when the process count is increased by k times. The subbatch size (i.e., the number of samples that a single process computes in each iteration) is set to two. We chose to keep the total number of processed data samples constant throughout the weak-scaling experiments to 9,437,184 and 2,359,296 samples for CIFAR10-Large and ImageNet-Large, respectively. Thus, when the number of processes doubles, the total batch size doubles, and the number of iterations halves. We note that because each iteration is bulk synchronous, increasing the number of iterations would not change the performance trend showcased in the graphs—all the performance numbers for a given number of processes would simply be multiplied by a constant factor.

The weak-scaling results for CIFAR10-Large are illustrated in Figure 26 and those of ImageNet-Large are illustrated in Figure 27. We observe trends for our weak-scaling experiments similar to those for the strong-scaling experiments. For weak scaling, Caffe/LMDBIO outperforms

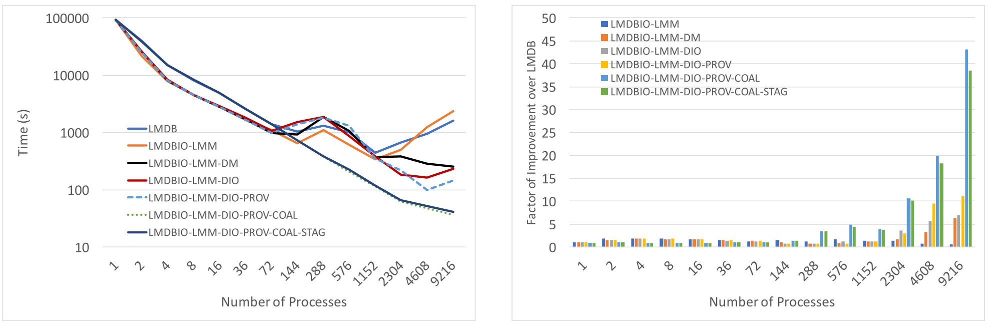


Fig. 26. Weak scaling using CIFAR10-Large on Bebop: (a) total execution time and (b) factor of improvement over Caffe/LMDB.

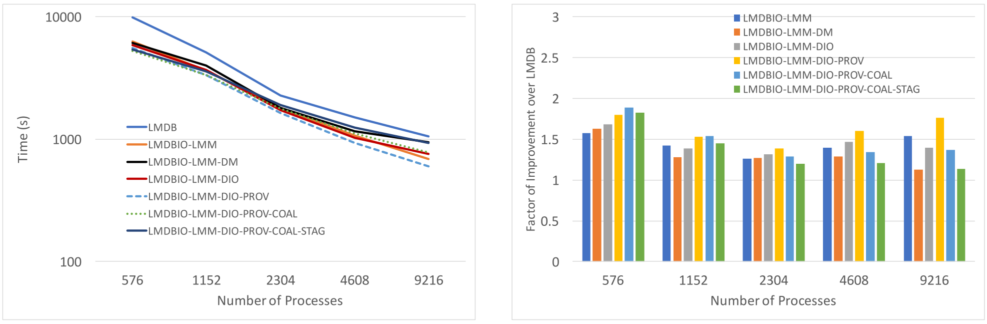


Fig. 27. Weak scaling using ImageNet-Large on Bebop: (a) total execution time and (b) factor of improvement over Caffe/LMDB.

Caffe/LMDB by up to 43 times for CIFAR10-Large and by up to 1.9 times for ImageNet-Large. In the case of ImageNet-Large, LMDBIO-LMM-DIO-PROV-COAL-STAG achieves the same performance as LMDBIO-LMM-DM. This is expected. LMDBIO-LMM-DM is an effective approach in improving performance—the drawback of LMDBIO-LMM-DM is not that it cannot improve performance but that the approach itself is speculative. That is, in some cases, the speculation might work well while in other cases the speculation might result in additional I/O causing some performance loss. The direct I/O methods (all optimizations with the LMDBIO-LMM-DIO prefix), however, deterministically improve performance without using such speculation. Thus, they are better approaches in the general case.

6 RELATED WORK

This section discusses work related to our own research, including deep learning frameworks, I/O frameworks, other storage architectures, input pipeline optimizations, and algorithmic improvements to deep learning.

Deep Learning Frameworks. Caffe is a well-known deep learning framework for which a number of parallel derivatives have been proposed. Most of these derivatives, including MPI-Caffe [29] and Caffe-MPI [2], focus on parallel efficiency improvements of the training, but only in the computation and communication aspects. In these frameworks, I/O is left untouched. S-Caffe [4], another parallel derivative of Caffe, proposes a workaround to overcome the inefficiencies of LMDB, but it does not really analyze the fundamental issues associated with LMDB. Our work identifies the core problems of LMDB and attempts to fix them outright. Thus our approach is directly applicable to all parallel derivatives of Caffe.

Apart from Caffe, other open-source deep learning frameworks have been developed, including Google's TensorFlow [3, 55], Theano [31, 53], Caffe2 [47], PyTorch [38], Microsoft Cognitive Toolkit [43], Apache MXNet [7], and Chainer [54]. These frameworks provide different competitive advantages in terms of training features and platform compatibility, but they adopt a core I/O infrastructure similar to that of Caffe to perform parallel data I/O.

For instance, Caffe2 inherits the I/O subsystems from Caffe. Thus, its distributed I/O subsystems are highly similar to the parallel extensions of the Caffe framework that we used in this article. PyTorch supports a broad range of data formats, the most popular of which is NumPy [37]. Both the memory and file layouts of NumPy can be irregular. For example, bytes of a single array can be laid out into noncontiguous chunks of a file or memory. Since the file structure is not deterministic, NumPy supports partial database access via `mmap`, the same as LMDB, to avoid reading the entire file to memory. To the best of our knowledge, there is no other way to partially load NumPy data from a file without using `mmap`, thus making it susceptible to the same shortcomings as LMDB. TensorFlow's I/O subsystem, by default, performs replicated data reads across different processes, but such a model can hurt the accuracy of the training because of reduced diversity of the sample data across different processes. Data sharding, which would make its data processing equivalent to that of Caffe, can be enabled through its high-level API to filter out unwanted data. While data sharding improves TensorFlow's accuracy, however, it also causes extra and redundant data access between processes similarly to what LMDB suffers from. In summary, while our work uses Caffe for the experiments, we believe that the lessons learned are generally applicable to other frameworks, too. In fact, a common practice in the community is to store datasets in LMDB format as it is natively supported by various other well-known deep learning frameworks such as TensorFlow, Caffe2, PyTorch, and Keras-TensorFlow. While other database formats certainly exist, the portability of LMDB across different frameworks has made it a go-to format, particularly for industries that use multiple frameworks for their AI and deep learning efforts.

I/O Frameworks. Various high-efficiency I/O frameworks have been developed for high-performance computing. MPI-IO [49, 50] is a low-level parallel I/O library that provides generic unstructured data I/O support. HDF5[8] and NetCDF [9], however, provide high-level I/O libraries for structured scientific application data via feature-rich programming interfaces. The parallel variants of these libraries [16, 30] leverage MPI-IO to enable parallel access and storage for files. These technologies are complementary to our work. While we used POSIX I/O in this article, our approach is not limited to it and can easily adopt any of the mentioned parallel I/O models instead.

We note, however, that although in theory MPI collective I/O is supposed to internally perform optimizations that limit I/O randomization, this is not always true in practice. In most MPI-IO implementations today, collective I/O significantly lags in performance compared with POSIX I/O. In fact, in our experiments, the performance of MPI collective I/O was much worse than that of POSIX I/O. The performance of MPI independent-I/O was comparable to, but not as good as, POSIX I/O. As MPI-IO developers ourselves, we are aware of these shortcomings. We will improve MPI collective I/O to incorporate similar techniques in the future, at which point LMDBIO can move from POSIX I/O to MPI collective I/O.

We point out that other frameworks, such as RocksDB[9] and HDF5, also use tree-based structures and allow for highly efficient sequential access to the database. Although random database access is possible, it is not as efficient as sequential access, because the database layout is not

[8]https://support.hdfgroup.org/HDF5.
[9]https://rocksdb.org.

deterministic—the layout cannot be computed unless all data records are already laid out in the database (essentially the same problem as LMDB). Similarly, TFRecord (TensorFlow's native database format) allows only for sequential database access. The central issue here is that the data samples are not indexed in a way that are suitable for parallel I/O (i.e., indexing is based on keys, rather than a numerical ordering). Thus, the lessons learned in this article are applicable to the above mentioned other frameworks, too.

Other Storage Architectures. Some researchers have worked around the issue of I/O in deep learning by using cluster systems where each node has its own permanent storage [24, 62]. Thus, the input data can be fragmented and the corresponding fragment placed locally on each node, instead of on the global filesystem. While such workarounds are possible, they are not practical in several scenarios, such as those that require deep learning algorithms to be executed on large supercomputing systems. Most supercomputer systems tend to host their data on a shared global filesystem and do not equip each node with its own permanent storage. In fact, for such shared global filesystems, reading from a large number of smaller files has been shown to be significantly worse than reading from a single large file because of the additional metadata traffic that it generates [33].

Having said that, on-node storage (e.g., nonvolatile memory express [28] and solid-state drives [23, 32]) are becoming common in large supercomputing systems. Some new-generation supercomputers, for example, Summit[10] at Oak Ridge National Laboratory and Cori[11] at the National Energy Research Scientific Computing Center, are equipped with on-node permanent storage using these technologies. Such on-node storage, however, is accessible only when the job is allocated to a particular node and is wiped clean when the job terminates or when a new job is allocated. Thus, any data that needs to be persistently stored across jobs must be fetched from the global filesystem. Some systems utilize on-node storage technologies in the form of burst buffers, where data staging can be performed prior to the job start. However, we remind the readers of this article that datasets used for training are often very large and cannot be simply replicated on the on-node storage of each node. Thus, using burst buffers would mean that the training dataset needs to be segmented across the burst buffers available on each node. As discussed in the article, this is not an easy task and would require the application to have prior knowledge as to what parts of the file would be accessed by each node. Unfortunately, traditional I/O systems used in deep learning do not have this knowledge, at least not without some of the improvements proposed in this article such as the data provenance information. Having said that, one could imagine combining the proposed data provenance technique with burst buffer technology to predict what data goes on which node and perform the necessary I/O before the job starts, that is, while the job is waiting in the queue. This is a viable technique that we have not explored in this article.

Input Pipeline Optimizations. Recently, researchers have realized the importance of I/O in deep learning. Consequently, a number of input pipeline optimization techniques have been proposed [8, 27, 28, 58, 63], for example, data caching, computation and I/O overlapping (pipelining/prefetching), parallel data parsing, and in-memory data shuffling. While these approaches are certainly useful, we believe that they are orthogonal improvements. For example, techniques such as data caching assume that all the data can fit in the system's memory for multiple epochs. This approach is useful for small datasets but is obviously not a feasible optimization for larger datasets. Techniques such as prefetching can hide the I/O cost behind that of the computation, but they benefit only those cases where the computation is more expensive than the I/O itself. For single-pass

[10] https://www.olcf.ornl.gov/summit/.
[11] http://www.nersc.gov/users/computational-systems/cori/.

algorithms (approaches that compute on the data only once), I/O is often more expensive than the computation. In contrast, our work solves the root causes of various I/O problems. In any case, these other input pipeline optimizations can be applied in conjunction with our proposed approach to further improve performance.

Algorithmic Improvements to Deep Learning. Another important deep learning research area involves high-accuracy large-batch training. Using large batches of data samples to train DNNs on large-scale supercomputers is a common practice for achieving high parallelism. In doing so, however, the accuracy of training can degrade significantly, since the DNN parameters are updated less frequently with gradients that contain more information. Consequently, several active studies have been trying to improve the accuracy of large-batch training. The common key idea of these techniques is to adjust the "learning rate." One of the earliest approaches in this direction involves adjusting the global learning rate linearly [25] based on the size of the batch. For instance, if the batch size is scaled by k times, the learning rate is also scaled by k. This approach is risky, however, and can cause the training to diverge during the initial phase. To address this issue, a warm-up scheme was introduced in Reference [15] to prevent such divergence by starting with a small learning rate and increasing it later during the training. To enable a larger batch training without accuracy loss, You et al. [59, 60] proposed layerwise adaptive rate scaling (LARS). LARS uses a different learning rate for different layers in the DNN, where the learning rate of a layer is the ratio between the norm of the layer weights and the norm of the gradients. With these optimizations, LARS enables parallel training using batch sizes up to 32K with negligible loss in training accuracy. These studies demonstrate that training with large batch sizes is practical and needs to be optimized, a subject that is the target for this article.

7 LESSONS LEARNED

While the study performed in this article provides an empirical evaluation of some of the I/O problems in large-scale deep learning and some solutions to these problems, we would like to take a moment to discuss the broader lessons that we learned from this study. One important takeaway is that several of the solutions proposed in the article are effectively workarounds for problems in the filesystem. A more comprehensive and elegant solution instead would be to improve or develop a new filesystem that is more targeted to deep learning workloads. What would such a filesystem look like? We have some thoughts.

(1) Deep learning workloads are read-heavy and rarely ever do writes. In fact, most deep learning frameworks perform writes only for checkpointing purposes, and these writes happen to files that are disjoint from the database file. In other words, the database files are "read only" for the lifetime of the application, and the checkpoint files are "write only" for the lifetime of the application (they would be read if the application needed to restart). If these files are separated onto two different filesystems, then each filesystem can be modified to support much more restrictive semantics. For example, the read-only filesystem can perform aggressive caching of global data on local nodes and avoid any locking and state management overheads needed for such data consistency. Similarly, the write-only filesystem does not have to worry about data consistency (the writes are nonoverlapping) and need not perform any caching at all.

(2) The ideal filesystem for deep learning would be one that supports fast random access similar to main memory. Thus the random data batch composition requirement of the training algorithms, namely, stochastic gradient descent, can be satisfied through data reading, and the additional in-memory data shuffling can be completely avoided. Technologies such as

on-node NVRAM and consortia such as Gen-Z[12] are already working in this direction, so such an approach might not be completely off the table. We note, however, that practically using such technologies is still some time away at the time of writing this article and avoiding random access is perhaps still the best strategy for now.

(3) If random access is impractical for filesystems, then the next best option would be strided access. Strided accesses are, unfortunately, not well supported by filesystems. I/O access in deep learning is very structured and is regularly strided. Moreover, there are no “holes” in the data access. All bytes are accessed by one process or another. Filesystems typically do not provide native APIs for such access, thus resulting in unnecessary prefetching and cache flushing. We worked around this problem with our staggered I/O model, but that model serializes I/O, which could have been entirely avoided if the filesystem had provided better strided I/O access.

8 CONCLUDING REMARKS

Despite significant advances in scalable deep learning, existing frameworks still suffer from a number of scalability limitations, particularly in aspects related to data I/O. In some cases, in fact, our analysis shows that I/O can take up to 90% of the total training time. In this article, we started with a thorough analysis of the I/O problems in the most widely used I/O subsystem in deep learning frameworks, called LMDB. Then, based on our analysis, we proposed LMDBIO—an optimized I/O plugin for scalable deep learning that incorporates six novel optimizations. We evaluated LMDBIO on up to 9,216 processes and demonstrated that it outperforms LMDB in all cases and improves overall application performance by up to 65-fold in some cases.

ACKNOWLEDGMENTS

We acknowledge the contributions made by Yanfei Guo, Robert Latham, and Robert Ross from Argonne National Laboratory through discussions on the I/O analysis of LMDB.

REFERENCES

[1] [n.d.]. NVIDIA Collective Communications Library (NCCL): Multi-GPU and Multi-Node Collective Communication Primitives. Retrieved from https://developer.nvidia.com/nccl.

[2] 2015. Caffe-MPI for Deep Learning. Retrieved from https://github.com/Caffe-MPI/Caffe-MPI.github.io.

[3] 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. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Retrieved from http://tensorflow.org/ Software available from tensorflow.org.

[4] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K. Panda. 2017. 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*. ACM, 193–205.

[5] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2017. Neurostream: Scalable and energy efficient deep learning with smart memory cubes. *IEEE Trans. Parallel Distrib. Syst.* 29, 2 (2017), 420–434.

[6] Nicolas Castet. 2018. Distributed deep learning with Horovod and PowerAI DDL. Retrieved from https://developer.ibm.com/linuxonpower/2018/08/24/distributed-deep-learning-horovod-powerai-ddl/.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNET: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[12] https://en.wikipedia.org/wiki/Gen-Z.

[8] Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. 2018. Characterizing deep-learning I/O workloads in TensorFlow. In *Proceedings of the IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS'18)*. IEEE, 54–63.

[9] Glenn Davis and Russ Rew. 1990. Data management: NetCDF: An interface for scientific data access. *IEEE Comput. Graph. Appl.* 10, 4 (1990), 76–82. DOI: https://doi.org/doi.ieeecomputersociety.org/10.1109/38.56302

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*. IEEE, 248–255.

[11] Facebook. 2017. Gloo. Retrieved from https://github.com/facebookincubator/gloo/blob/master/docs/readme.md.

[12] Michael Feldman. 2017. Intel Spills Details on Knights Mill Processor. Retrieved from https://www.top500.org/news/intel-spills-details-on-knights-mill-processor/.

[13] Andrew Gibiansky. [n.d.]. Bringing HPC Techniques to Deep Learning. Retrieved from http://andrew.gibiansky.com.

[14] Google. 2018. Cloud Tensor Processing Units (TPUs). Retrieved from https://cloud.google.com/tpu/docs/tpus.

[15] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

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

[17] Antonio Gulli and Sujit Pal. 2017. *Deep Learning with Keras*. Packt Publishing Ltd.

[18] Mark Harris. 2017. NVIDIA DGX-1: The Fastest Deep Learning System. Retrieved from https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system/.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[20] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: Techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 131–143.

[21] Jeremy Hsu. 2016. Fujitsu Memory Tech Speeds Up Deep-Learning AI. Retrieved from https://spectrum.ieee.org/tech-talk/computing/software/fujitsu-memory-tech-speeds-up-deep-learning-ai.

[22] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. FireCaffe: Near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2592–2600.

[23] Zhi-Lin Ke, Hsiang-Yun Cheng, and Chia-Lin Yang. 2018. LIRS: Enabling efficient machine learning on NVM-based storage via a lightweight implementation of random shuffling. *arXiv preprint arXiv:1810.04509* (2018).

[24] Akhmedov Khumoyun, Yun Cui, and Lee Hanku. 2016. Spark based distributed deep learning framework for big data applications. In *Proceedings of the International Conference on Information Science and Communications Technologies (ICISCT'16)*. IEEE, 1–5.

[25] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997* (2014).

[26] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto.

[27] Sameer Kumar, Dheeraj Sreedhar, Vaibhav Saxena, Yogish Sabharwal, and Ashish Verma. 2017. Efficient training of convolutional neural nets on large distributed systems. *arXiv preprint arXiv:1711.00705* (2017).

[28] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. 2018. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 51.

[29] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. 2015. Why M heads are better than one: Training a diverse ensemble of deep networks. *arXiv* (2015). http://arxiv.org/abs/1511.06314

[30] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC'03)*. ACM, New York, NY, 39.

[31] He Ma, Fei Mao, and Graham W. Taylor. 2016. Theano-MPI: A theano-based distributed training framework. In *Proceedings of the European Conference on Parallel Processing*. Springer, 800–813.

[32] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J Pennycook, et al. 2018. CosmoFlow: Using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 65.

[33] Pierre Matri, María S Pérez, Alexandru Costan, and Gabriel Antoniu. 2018. TỳrFS: Increasing small files access performance with dynamic metadata replication. In *Proceedings of the 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'18)*. IEEE.

[34] Microsoft. [n.d.]. Cognitive Toolkit: Multiple GPUs and Machines. Retrieved from https://docs.microsoft.com/en-us/cognitive-toolkit/multiple-gpus-and-machines.

[35] Timothy Prickett Morgan. 2017. Machine Learning Gets an InfiniBand Boost with Caffe2. Retrieved from https://www.nextplatform.com/2017/04/19/machine-learning-gets-infiniband-boost-caffe2/.

[36] NVIDIA. 2018. NVIDIA Deep Learning Platform: Giant Leaps in Performance and Efficiency for AI Services, From the Data Center to the Network's Edge. Retrieved from https://images.nvidia.com/content/pdf/inference-technical-overview.pdf.

[37] Travis E. Oliphant. 2006. *A Guide to NumPy*. Vol. 1. Trelgol Publishing USA.

[38] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *Proceedings of the Conference and Workshop on Neural Information Processing Systems (NIPS-W'17)*.

[39] Sarunya Pumma, Min Si, Wu chun Feng, and Pavan Balaji. 2017. Parallel I/O optimizations for scalable deep learning. In *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS'17)*.

[40] Sarunya Pumma, Min Si, Wu chun Feng, and Pavan Balaji. 2017. Towards scalable deep learning via I/O analysis and optimization. In *Proceedings of the 19th International Conference on High Performance Computing and Communications (HPCC'17)*.

[41] Carl Edward Rasmussen. 2003. Gaussian processes in machine learning. In *Summer School on Machine Learning*. Springer, 63–71.

[42] Baidu Research. [n.d.]. baidu-allreduce. Retrieved from https://github.com/baidu-research/baidu-allreduce.

[43] Microsoft Research. 2017. The Microsoft Cognitive Toolkit. Retrieved from https://docs.microsoft.com/en-us/cognitive-toolkit/.

[44] Karl Rupp. 2018. Microprocessor Trend Data. Retrieved from https://github.com/karlrupp/microprocessor-trend-data.

[45] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet large scale visual recognition challenge. *Int. J. Comput. Vis.* 115, 3 (2015), 211–252. DOI: https://doi.org/10.1007/s11263-015-0816-y

[46] Alexander Sergeev and Mike Del Balso. 2018. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[47] Facebook Open Source. [n.d.]. Caffe2 A New Lightweight, Modular, and Scalable Deep Learning Framework. Retrieved from https://caffe2.ai.

[48] TensorFlow. [n.d.]. How To Compile and Use MPI-Enabled TensorFlow. Retrieved from https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/mpi.

[49] R. Thakur, W. Gropp, and E. Lusk. 1998. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of the IEEE/ACM Conference on Supercomputing (SC'98)*.

[50] R. Thakur, W. Gropp, and E. Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*. Washington, DC, 182–189.

[51] Rajeev Thakur, Ewing Lusk, and William Gropp. 1997. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*. Technical Report. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory.

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

[53] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

[54] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. Chainer: A next-generation open source framework for deep learning. In *Proceedings of the Workshop on Machine Learning Systems (LearningSys) in the 29th Annual Conference on Neural Information Processing Systems (NIPS'15)*, Vol. 5. 1–6.

[55] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. 2016. Distributed TensorFlow with MPI. *CoRR* abs/1603.02339 (2016). http://arxiv.org/abs/1603.02339

[56] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. 2005. The SGI® AltixTM 3000 global shared-memory architecture. *Silicon Graphics, Inc.* (2005).

[57] Joe Yaworski. 2017. Intel Omni-Path Architecture Enables Deep Learning Training on HPC. Retrieved from https://itpeernetwork.intel.com/intel-omni-path-deep-learning-training/.

[58] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. 2018. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992* (2018).

[59] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling SGD batch size to 32k for ImageNet training. *arXiv preprint arXiv:1708.03888* (2017).

[60] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. ImageNet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP'18)*.

[61] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide residual Networks. *arXiv preprint arXiv:1605.07146* (2016).

[62] Kunlei Zhang and Xue-Wen Chen. 2014. Large-scale deep belief nets with MapReduce. *IEEE Access* 2 (2014), 395–403.

[63] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. 2018. Entropy-aware I/O pipelining for large-scale deep learning on HPC systems. In *Proceedings of the IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18)*.

Received September 2018; revised April 2019; accepted April 2019