I/O Bottleneck Investigation in Deep Learning Systems

Sarunya Pumma,[a,b] Min Si,[b] Wu-chun Feng,[a] Pavan Balaji[b]
[a]Virginia Tech, USA; {sarunya, wfeng}@vt.edu
[b]Argonne National Laboratory, USA; {spumma, msi, balaji}@anl.gov

ABSTRACT

As deep learning systems continue to grow in importance, several researchers have been analyzing approaches to make such systems efficient and scalable on high-performance computing platforms. As computational parallelism increases, however, data I/O becomes the major bottleneck limiting the overall system scalability. In this paper, we present a detailed analysis of the performance bottlenecks of Caffe on large supercomputing systems and propose an optimized I/O plugin, namely LMDBIO. Throughout the paper, we present six sophisticated data I/O techniques that address five major problems in state of the art I/O subsystem. Our experimental results show that Caffe/LMDBIO can improve the overall execution time of Caffe/LMDB by 64-fold compared with LMDB.

1 INTRODUCTION

As existing parallel deep learning frameworks explore the limits of parallelism and scalability, they have started utilizing large supercomputing systems and highly efficient computational units to improve their computational efficiency. Such improvement in the computational framework has, however, started to expose new bottlenecks in their I/O subsystem.

To understand this problem, we first performed a detailed analysis of the Caffe [1] deep learning framework and showed that with 9,216 processes, Caffe is highly unscalable and performs nearly 660-fold worse compared to ideal strong scaling as shown in Figure 1(a). We also analyzed the breakdown in time taken by the various components of Caffe in Figure 1(b) and note that the data I/O time ("Read time") takes approximately 90% of the training time when using 9,216 processes.

The primary reason for this inefficiency is because of the efficiency of Caffe's I/O subsystem, LMDB, on large-scale sytems, which stems from five reasons:

(1) **Interprocess contention**: LMDB internally uses `mmap` as a core mechanism to read data. `Mmap` exposes the layout of a file from the file system into the virtual address space of a process and enables accesses to the file as if it were a memory buffer. However, `mmap` fully relies on the operating system to handle I/O operations. Linux's I/O interrupt handler is a *bottom-half* handler, where the interrupt is not associated with any particular user process but is a generic event informing the file system that an I/O operation that has been issued by one or more processes has now completed. This causes most processes that are waiting for I/O to be woken up every time that an I/O interrupt comes in resulting in significantly increase in the number of context switches.

(2) **Implicit I/O inefficientcy**: `Mmap` is considered implicit I/O as actual I/O operations are hidden from user applications and managed by the operating system. Besides convenience in I/O management, `mmap` is highly inefficient as amount of data to be fetched, size of I/O request, and when data is fetched cannot be controlled by the user applications.

(3) **Sequential database access restriction**: LMDB database format, B+ tree, does not allow random accesses. To access a data record, LMDB needs to start from the root node of the B+ tree and parse through every branch node in the path to reach the target data record. This data-reading model is troublesome for parallel I/O because processes have to access different parts of the database file, resulting in a semirandom data access pattern. This causes skew in data I/O because different processes do different amounts of work, which can severely degrade the overall progress of a parallel application.

(4) **Inefficient I/O block size**: In parallel data reading, multiple processes read the same batch of data in the granularity of one data sample which can be as low as 4 KB for the CIFAR10 dataset. Small I/O block size is proven to be inefficient in various types of I/O. Although I/O block size does not impact the performance of `mmap`, larger I/O can significantly improve the performance of some I/Os (e.g., POSIX I/O).

(5) **I/O randomization**: I/O randomization is a well-known I/O problem that happens when large number of processes participate in I/O which causes abundant I/O requests to be out of order.

To address these shortcomings, we present LMDBIO, an optimized I/O plugin for Caffe, details of which are presented in the following section.

2 LMDBIO

In this section, we present LMDBIO optimizations.

LMDBIO-LMM [3]: LMDBIO-LMM attempts to eliminate interprocess contention in `mmap` by localizing `mmap`. In this model, a single process is chosen on each node as the root process. The root process reads data from the file system and distributes it to the remaining processes on the node using MPI-3 shared memory. The `mmap` localization approach allows the traditional Linux bottom-half handler for I/O to wake up the exact process that is waiting on I/O, since only one process is performing I/O. This strategy minimizes the number of context switches and helps improve performance.

LMDBIO-LMM-DM [2]: LMDBIO-DM is an enhanced version of LMDBIO-LMM that optimizes the I/O access of Caffe in a distributed-memory system in two steps:

Part I: Our goal is to ensure that each process would read only the data that it needs to process. To do so, each process first reads the data that it needs to process and then passes to the next process the information about the location where it stopped. The data handoff is not trivial as the position idenfier is not a simple file offset, but rather a complex structure that contains multiple pointers. We adopt a "symmetric address" allocation to allow for convenient information exchange accross processes.

Part II: The previous step comes at the cost of serialization in data I/O. Here, we try to improve parallelism in the data I/O by speculatively performing the I/O. We first estimate what part of the database we need to fetch to memory. This is a complex task since the structure of the B+ tree is not deterministic. We perform a history-based estimation in order to allow for a highly accurate speculative I/O. Once pages are fetched to memory, we perform an in-memory sequential seek process to find the starting point of the data batch and send such information to the next reader. Then, the reader can perform the actual data processing.

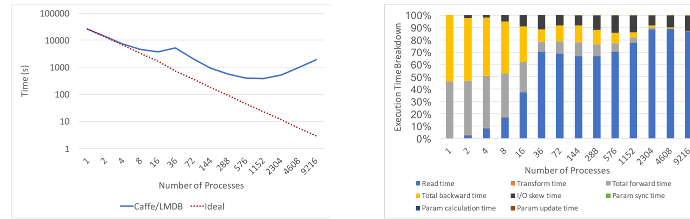

Figure 1: Caffe scalability (CIFAR10-Large dataset): (a) scaling analysis; (b) scaling time breakdown

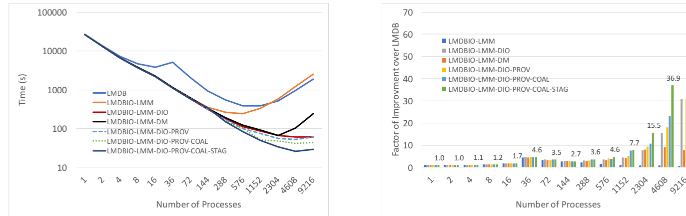

Figure 2: CIFAR10-Large: (a) total execution time; (b) factor of improvement over LMDB

LMDBIO-LMM-DIO: To improve I/O read performance of LMDB, we aim to replace `mmap` with POSIX I/O. Since I/O becomes explicit, the I/O read size and the offset of the bytes to read in the file have to be explicitly specified. Therefore, before starting the training, LMDBIO uses one process to seek through the database to obtain offsets and sizes of all the data records that will be used in all the training iterations. This process has to be performed by using `mmap` as the layout of the file is unknown. We call this process "sequential seek". Once the sequential seek is completed, the offsets and sizes will be distributed to every reader process. The rest of the data reading design of LMDBIO-DIO is inherited from LMDBIO-LMM where there is only one reader per node performing data read using POSIX I/O and data is shared among processes on the same node using MPI-3 shared buffer.

LMDBIO-LMM-DIO-PROV: To completely eliminate sequential seek, additional *database creation information*, which is known as "provenance information" is used for determining the layout of the database. Such information enables arbitrary database access allowing us to replace `mmap` with POSIX I/O entirely. Since provenance information is highly useful, LMDB should store provenance data in addition to the original metadata. Provenance information can be created while the database is being generated or later as post processing using one-time read of the database. It can be stored as part of the database or a separate auxiliary file. It is usually relatively small compared to the database itself.

LMDBIO-LMM-DIO-PROV-COAL: Previous optimizations read only one batch of data which some I/O request sizes are smaller than I/O block size of the shared file system. Now that LMDBIO uses POSIX I/O for data reading, using a larger I/O block size is more beneficial. Therefore, a constant amount of memory of size 2.5 GB is kept aside on each node for data reading. Multiple batches are read at once based on the constant memory size.

LMDBIO-LMM-DIO-PROV-COAL-STAG: As I/O randomization is caused by out-of-order I/O requests, to overcome this problem, we adopt the *I/O staggering* technique where some I/O requests are delayed to arrange I/O requests to be more in order. In this optimization, readers are divided into multiple groups with the same size. Readers that access a file in close proximity to one another will be grouped together. The key idea of our I/O staggering is to allow only a limited number of readers (i.e., one staggering group) to access the file concurrently in order to reduce randomization in I/O.

3 EXPERIMENTS AND RESULTS

Our experimental evaluation was performed on Argonne's "Bebop" cluster.[1] We use the CIFAR10-Large dataset. We used a batch size of 18,432. We trained the network for the CIFAR10-Large dataset over 512 iterations (9 million images). As shown in Figure 2(a) and (b), Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG outperforms other LMDBIO optimizations and Caffe/LMDB in all cases. For 9,216 processes, we achieve 64-fold better performance than Caffe/LMDB.

4 CONCLUSION

In this paper, we presented a scalable I/O plugin, called LMDBIO, for the Caffe deep learning framework. We first performed a detailed analysis of I/O in Caffe and discussed the cause of these problems. We then presented LMDBIO with various optimizations to alleviate these problems to improve the I/O performance. We presented experimental results which demonstrate a 64-fold improvement in the overall performance of Caffe/LMDB in some cases.

REFERENCES

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

[2] Sarunya Pumma, Min Si, Wu chun Feng, and Pavan Balaji. 2017. Parallel I/O Optimizations for Scalable Deep Learning. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE.

[3] 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)*. IEEE.

[1] http://www.lcrc.anl.gov/about/bebop