Parallel I/O Optimizations for Scalable Deep Learning

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, that takes into account the data access pattern and Caffe's database format. Throughout the paper, we present several sophisticated data I/O techniques that allow for significant improvement in such environments. Our experimental results show that LMDBIO can improve the overall execution time of Caffe by 6-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 even with just 512 processes, Caffe is highly unscalable and performs nearly 20-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 70% of the training time when using 512 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 two reasons:

(1) It 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` handles I/O requests inefficiently. Once a page that is not yet loaded into memory is accessed, a page fault handler fetches the data from the file system. Meanwhile, the user process goes to sleep to wait for I/O. When the I/O operation completes, an interrupt is raised informing the file system of the completion. The interrupt handler is a *bottom-half* handler in Linux, 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 was issued by one or more processes has now completed. Since only some of the processes were waiting for that specific I/O operation that just completed, most processes will be woken up to see that their I/O operation has not completed and go back to sleep. Only one or a few processes will be woken up and be able to use this I/O completion to perform further processing. Consequently, this model significantly increases the number of context switches that get triggered with most of the switches resulting in no real work (Figure 1(c)).

(2) Its 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. It is not possible to directly "seek" to an arbitrary page without the risk of accidentally reaching a page that contains no information of how to go to the next or leaf node. 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. That is, each process needs to start at a position in the database that cannot be precomputed and requires information from the previous data records to compute. 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.

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 an overview of our LMDBIO optimizations, the details of which are available in [2, 3].

LMDBIO-LMM: Intranode Optimization: As described above, Caffe suffers tremendously from the contention caused by `mmap` that is used internally by LMDB. LMDBIO-LMM attempts to eliminate such contention 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. This approach aims to reduce I/O parallelism to give `mmap` a more sequential view of I/O and to minimize interprocess contention. The `mmap` localization approach also 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-DM: Internode Optimization: The optimization performed by LMDBIO-LMM is limited to intranode contention and does not help in distributed-memory environments. As described above, LMDB's B+ tree structure results in significant amount of redundant data I/O across different processes. 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 more sophisticated data 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

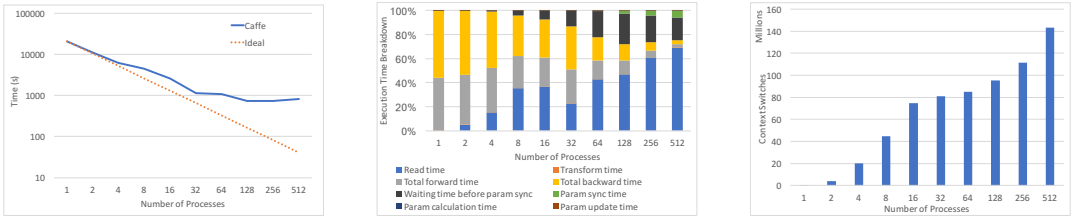

Figure 1: Caffe scalability (CIFAR10-Large dataset): (a) scaling analysis; (b) scaling time breakdown; (c) context switches

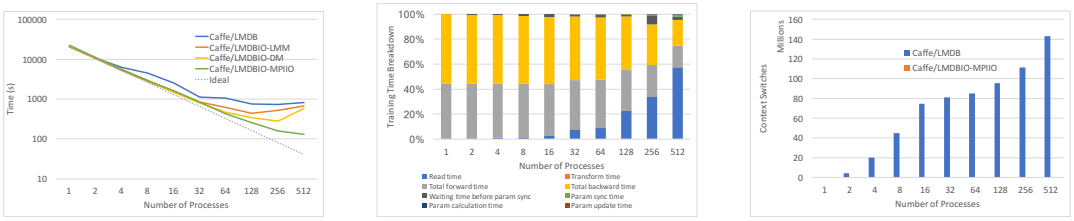

Figure 2: CIFAR10-Large: (a) performance comparision; (b) LMDBIO-MPIIO training time breakdown; (c) context switches

since the structure of the B+ tree is not always straightforward. We use an approach that estimates these pages through an "initial guess" based on the number of pages used by the first data sample. Based on the estimation, processes simultaneously touch the pages in the memory-mapped file, thus forcing the filesystem to fetch those pages to memory. Then, we perform an in-memory sequential seek process to find the starting point of the data batch for the next reader. After the reader successfully sends the starting location to the corresponding process, the reader can perform the actual data processing. As the iterations progress, however, we correct our initial guess depending on how accurate the guess was in the previous iterations.

LMDBIO-MPIIO: Direct I/O Optimization: The previous two optimations do not entirely solve the data reading problems since I/O operations are still handled by `mmap` which is highly dependent on the OS scheduler which does not allow for optimizations with bulk I/O. To overcome these bottlenecks, we propose direct I/O via MPI-IO. With MPI-IO, bulk data reading is more efficient enabling significant improvement on deep learning's I/O. Unlike LMDBIO-DM, the accuracy of estimation of the speculative pages becomes a necessary concern in LMDBIO-MPIIO because it affects correctness, rather than performance. For this reason, we need a fallback path to correct the data reading in case our estimation does not cover some required pages.

We use two buffers to achieve this. The first buffer is the "main buffer" that is used for database accesses and is mapped to an empty virtual file. The second buffer is the "fallback buffer" which will be accessed only when our speculative I/O is inaccurate. The fallback buffer maps to the actual database file which it relies on the Linux's scheduler to fetch pages from the file system to memory. We use a memory protection mechanism along with an associated signal handler to implement a page fault handler in user space. When a reader tries to access a page that is not yet loaded to memory (via direct I/O), our page fault handler copies the missing page from the fallback buffer to the main buffer.

3 EXPERIMENTS AND RESULTS

Our experimental evaluation was performed on Argonne's "Blues" cluster.[1] We use the CIFAR10-Large dataset in which each sample is approximately 3 KB. We used a batch size of 4,096. We trained the network for the CIFAR10-Large dataset over 1,024 iterations (4 million images).

As shown in Figure 2(a), Caffe/LMDBIO-MPIIO outperforms other LMDBIO optimizations and Caffe/LMDB in all cases. For 512 processes, we achieve 6-fold better performance than Caffe/LMDB. Figure 2(b) shows that the "Read time" and "Waiting time before param sync" in Caffe/LMDBIO-MPIIO shrinks compared to those of Caffe/LMDB meaning that the I/O and load balance have been improved. Moreover, Caffe/LMDBIO-MPIIO's context switches are negligible compared to those of Caffe/LMDB as shown in Figure 2(c).

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, showcased the problems associated with it, 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 6-fold improvement in the overall performance of Caffe 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/blues