Vol.2017-HPC-160 No.39
2017/7/28

Performance analysis of a deep learning framework on a high-performance distributed file system

Sašo Stanovnik[1,a)] Amir Haderbache[1,b)] Masahiro Miwa[1,c)] Kohta Nakashima[1,d)]

Abstract: Current deep learning framework data access patterns are not adapted to traditional HPC distributed file systems such as FEFS, as they do not take into account the data access latency and overhead. This causes performance degradation in high-scale environments, especially with large dataset sizes. We analyse data access patterns in an existing framework with both external and introspective analysis, then propose and implement a new pattern in order to achieve 5 to 6 times better performance when training on and by adapting to the capabilities of a distributed file system.

1. INTRODUCTION

Deep learning algorithms have recently become a popular research topic as they improved the way we apply supervised learning to perceptual problems such as computer vision. This method enables a user to process large datasets of perceptual inputs (images, sounds) and descriptive targets into models that can automatically map inputs to targets. Somewhat recently, the implementation of practical deep learning applications has been facilitated due to the development of various software frameworks such as Caffe [1] and Theano [4]. These platforms provide high-level interfaces, mostly in C++ or Python, to efficiently train neural network models, supported by numerical computation optimization libraries, multi-GPU support and I/O management. The data storage and access method used influences the overall application performance, as the training process continuously accesses input data during computation by iterating over the dataset.

Recent works showed [9] that training on a large dataset, such as ImageNet [5], with an optimized key-value data store provides significantly better performance than using separate image files on a local file system. Deep learning input consists of a large set of images, which are accessed sequentially in a series of batches. Therefore, strong indexing capabilities to locate a specific image in the storage system and an advanced caching mechanism are essential to achieving optimal performance. Many works propose a good implementation of training neural networks with locally stored data, but give no out-of-the-box solution for efficiently running large-scale deep learning on an HPC cluster. The scope of our work is to analyse the performance of training neural networks at a large scale with a distributed deep learning framework on a parallel file system commonly used in HPC. For this, we deployed MPI-Caffe [12] with the FEFS® [7] filesystem and performed a performance analysis in order to improve the way we access remotely stored data. Next, we analysed MPI-Caffe's I/O pattern by using the `blktrace` [3] and `strace` [8] tools to get more insight into the particulars of the data access pattern. From these I/O study results, we inferred a more efficient data access pattern for MPI-Caffe to access data with, and which is better suited to the capabilities of a parallel distributed filesystem. Our experiments show that this new data access pattern can reduce the training time by 5 to 6 times compared to the original one on some storage configurations.

[1] Fujitsu Laboratories Limited, 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki-shi, Kanagawa-ken 211-8588, Japan
[a)] saso@extra.labs.fujitsu.com
[b)] haderbache.amir@jp.fujitsu.com
[c)] masahiro.miwa@jp.fujitsu.com
[d)] nakashima.kouta@jp.fujitsu.com

2. BACKGROUND INFORMATION

2.1 Deep learning

An important aspect of neural network training is that it relies on iterative optimization algorithms. In general, and with proper training, the more the cost function is minimized, the better the built model can predict accurate values using new input data. Because training operations can be massively parallelized, we often use GPUs to speed up the training. Until the weights converge, the CPU repeatedly reads batches of image data from a storage system and feeds them into GPU memory for computation, and the more the algorithm iterates over a large amount of training data, the better the model accuracy becomes—but note that the quality of data is also important. As deep learning data requirements get bigger, we need to build larger and more sophisticated models. Therefore, strong support from the underlying data storage system has become a requirement for large-scale deep learning.

2.2 The HPC architecture

HPC computer clusters represent interesting platforms for running large scale deep learning applications, as their architecture is optimized for compute-intensive applications. However, they

need to be enhanced with efficient data supply mechanisms to meet the data requirements of deep learning applications. Traditionally, an HPC cluster physically separates compute nodes from data nodes and uses a high speed interconnect between them. HPC storage systems use both local file systems on compute nodes and global parallel file systems served by data nodes. Local file systems are used to store temporary data required for computation whereas parallel file systems store shared data. A parallel file system makes the data accessible to all compute nodes, thus enabling easy data sharing. However, the problem with this architecture is that it requires data to be transferred from data nodes to a compute node through the network when the client makes a request. Because deep learning applications spend most of their time in I/O requesting large, contiguous chunks of data, specific strategies need to be developed to speed up large-scale training on HPC systems.

2.3 MPI-Caffe

MPI-Caffe is a distributed version of the BVLC Caffe framework developed internally at Fujitsu Laboratories. It embeds the Caffe runtime into an MPI application in order to speed up computation time when running on multiple machines, and especially when learning on a large dataset. In essence, MPI-Caffe runs multiple parallel processes which perform interprocess communication for model parameter updates, and operate on either shared data or multiple copies of it. The data domain decomposition is simple, as the dataset is evenly divided among the processes. Moreover, the training behaviour (i.e. the number of iterations, mini-batch size, model definition, solver and the loss function used) remains encapsulated by Caffe parameters defined in dedicated `prototxt` configuration files. Caffe proposes many *data layer*s as data storage backends from which processes can access image data. We focus our study on the *database layer* which allows Caffe to read from optimized key-value data stores such as LMDB (Lightning Memory-Mapped Database) [2]. LMDB uses two types of files for managing I/O access requested by applications: a *data file* and a *lock file*, both located in the same directory. The data file stores the image training data, whereas the lock file is a means of providing database integrity for concurrent access.

2.4 FEFS

The Fujitsu Exabyte File System (FEFS®)[*1] is a clustered distributed file system based [7] on the Lustre file system. It was originally developed to handle the computing performance of the K-computer, the world's fastest supercomputer in 2011.

An example of its architecture is where data servers (OSS – *object storage server*s) store data into external storage devices like HDDs and SSDs, meta-data servers (MDS) store metadata information and clients access and use the data. Each OSS corresponds to a data node. The different peripheral storage devices connected to the OSS (HDD, SSD) are called *object storage targets* (OSTs).

One of FEFS's key features is file *striping*. Basically, a large file stored into FEFS is divided into several contiguous chunks of data called *stripes* which are distributed, in a round-robin fashion, among the different OSTs. This is what we did for our data. With striping, the maximum file size is not limited by the capacity of a single OST. Moreover, when multiple clients access the same file, the total read throughput increases. The user can set up a *stripe size* and a *stripe count* for each directory in the FEFS file system. The stripe count corresponds to the number of OSTs where the stripes have to be distributed, whereas the stripe size corresponds to the size of one "slice" of the file that can be distributed across the servers. FEFS also uses multiple levels of cache to improve client data accesses: a cache on the server side (OSS cache) and another on the client side (the FEFS client cache). Both cache systems leverage the local Linux page cache to retain data in memory, avoiding disk access.

3. EXPERIMENTAL ENVIRONMENT

This section describes the hardware and software environment that was used for all experiments described in this paper.

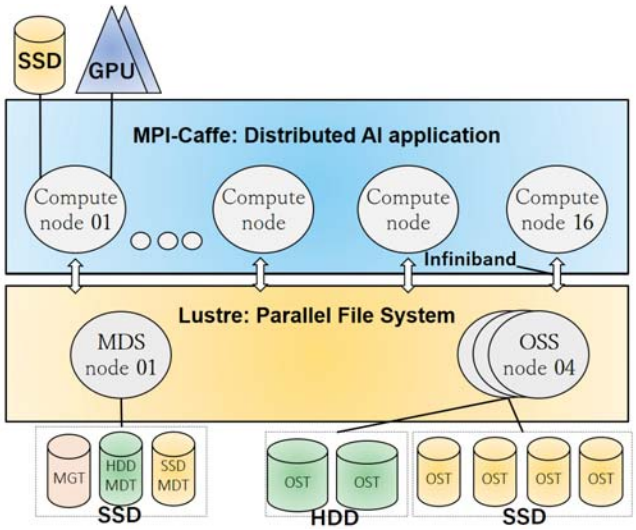

Fig. 1 Experimental environment logical diagram

All experiments were performed on our KAGAMI cluster hosted at the Fujitsu Laboratories HPC division. It is composed of sixteen compute nodes and four data nodes connected to an InfiniBand EDR network (one Mellanox ConnectX-4 EDR 2-port IB HCA per node, one InfiniBand EDR cable per node). Each node is dual-socket with two 2.1 GHz 18-core (36-thread) Intel Xeon E5-2695 v4 processors and 8 16 GiB DDR4 memory modules (128 GiB total). Storage–wise each node has an Intel NVMe SSD 750 Series with a storage capacity of 1.2 TiB and two Nvidia Tesla P100 GPUs. The data nodes are equipped with four SSDs (each) with the same specifications as above and eighteen 1.8 TiB SAS HDDs (2x9 RAID 5) and six 256 GiB SATA HDDs (3x2 RAID 10) each. Thus, the complete compute capacity is 1152 vCPUs and 32 GPUs, and the total usable storage capacity of the whole cluster is around 160 TiB.

We deployed the FEFS file system, based on Lustre version 2.6.0, over the KAGAMI cluster as described in Figure 1: the compute nodes as FEFS clients, one data node as the MGS/MDS (management/metadata server) and the three other data nodes as

[*1] FEFS is a registered trademark of Fujitsu Limited.

OSSs (object storage servers). The MDS has one SSD for the MGT (management target) and two SSD for MDTs (metadata targets). Each OSS has four SSD OSTs (object storage targets) and two HDD OSTs.

For training, we used the ImageNet data set which is a collection of images typically rescaled to 256 x 256 pixels. The training dataset is stored in a 240 GiB LMDB file and the testing set into another 9.4 GiB LMDB file. The FEFS stripe count was set to 12 for SSDs and 6 for HDDs, which means striping to every available device, and we used multiple stripe sizes with multiple copies of the data. For the purpose of these experiments, we turned off the FEFS OSS cache, cleared the FEFS client cache and the Linux page cache before every experiment, and used both SSD and HDD OSTs for data access.

On the FEFS clients, we deployed the MPI-Caffe framework based on Caffe 0.1.0-rc3 with OpenMPI 2.0.2 and ran distributed deep learning training using 16 MPI processes, one per node, using two GPUs per node, each of them effectively accessing different regions of the same LMDB data file. The maximum number of iterations performed during the training has been set to 1000. This value corresponds approximately to 1.5 epochs which means that, as training is an iterative process over the data set, we iterate over the entire data set approximately 1.5 times.

4. PERFORMANCE ANALYSIS

4.1 Local configuration

We start the analysis with the simplest configuration: MPI-Caffe running across multiple compute nodes, where each node has its own copy of the LMDB data and each process accesses its data from the local SSD. This configuration is easy to deploy as it does not require a distributed file system installation. However, it has two main drawbacks: first, it requires a local SSD for each compute node which is often prohibitively expensive, and second, it uses several replicas of the same dataset which is a tremendous waste of data storage capacity. This configuration is clearly not scalable in terms of cost, and because MPI-Caffe processes access the same dataset, we next consider a shared file system as a better solution for our needs.

4.2 FEFS configurations

We deployed the FEFS file system architecture, created multiple directories with different *stripe size*s and placed a copy of the LMDB dataset in them. This time, all MPI-Caffe processes access a single shared LMDB data file during the training. This configuration uses only one copy of the data regardless of the number of compute nodes, saving a lot of storage resources. However, when running performance benchmarks with this configuration, we noticed unstable behaviour (stalling, no consistent results) we did not experience before. After some investigation it became clear that this problem is related to the LMDB lock mechanism policy. When multiple MPI processes try to access the same data and lock file (even when accessing different data within the file), some lock contention occurs and degrades the I/O access performance. To reduce lock contention, we made the MPI-Caffe processes open the LMDB database *read-only* by modifying file system attributes of the files to read-only—this is called the *shared file* configuration. With this, results and stability are slightly improved, but most problems still persist. As documented, LMDB has not been developed to be used in such distributed contexts with multiple concurrent accesses. To make it work properly, we use a trick consisting of using not a single lock file, but instead using a separate lock file for each process, while still sharing the data file. In technical terms, each process should point to a specific directory containing a materialized LMDB lock file and a symbolic link to the shared LMDB data file. With this configuration, we still use only one copy of the data while solving the lock contention problem by using multiple copies of the lock file. This solution is essentially free, as each lock file is only sized at a few kilobytes. We call this new solution the *ownlock* configuration.

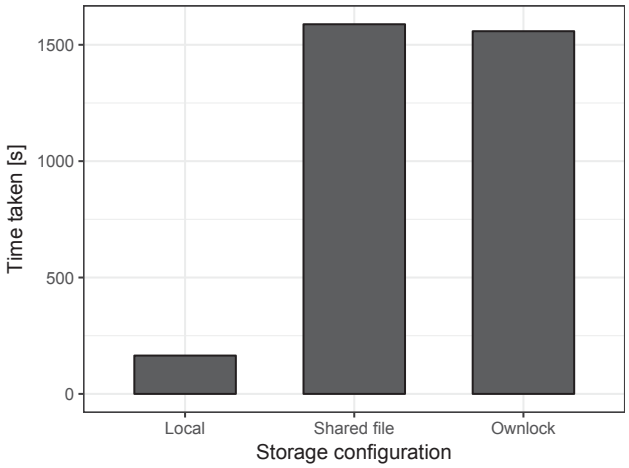

Fig. 2 Performance results of using a dataset stored on shared storage versus a local one

As seen in Figure 2, we have slightly improved the performance with this solution, as well as solved all stability issues, as the training process now does not stall or fail to complete. We are still quite far from the performance that multiple copies on local SSD storage offer us initially, but have significantly reduced the overall storage capacity requirement.

4.3 Further configuration details

We keep the previous configuration, where we have one data file and multiple lock files, and analyse the influence of the stripe size parameter on the performance.

In our experiments, we use two GPUs (see Section 3 for details) on each compute node and set the training batch size to 64 images. Those images come from the ImageNet dataset, a broadly used collection of coloured images scaled down to 256x256 pixels.

As depicted in Figure 3, the stripe size giving the best performance among the ones we have tested with the original implementation, is 1 MiB. This corresponds to the fact that we have, in essence, a lot of smaller file accesses, rather than large sequential reads. We use this stripe size as a default whenever we do not explicitly mention a specific stripe size.

We next analysed the influence of the storage device on the performance, comparing the performance between HDDs and SSDs when we use them as FEFS OSTs on data servers. It turns out

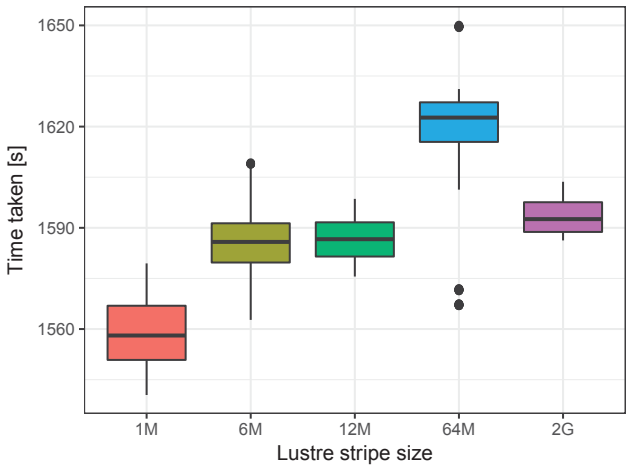

Fig. 3 Training time for different *stripe size*s with the *sequential* pattern

that while we have improved the performance by using a separate LMDB lock file for each compute node, and made significant stability improvements, performance on HDD-backed FEFS storage remains bad. Furthermore, using a single LMDB lock file and a single data file on HDD-backed storage stalled training, and we could not get any training run to succeed.

5. DATA ACCESS PATTERN

In order to get a better insight into these performance results, we performed a thorough I/O analysis described in the next section using the Linux `strace` and `blktrace` tools.

5.1 strace analysis

`strace` is a Linux utility for tracing system calls and signals. It lists the system calls that are called by a process and the signals received by a process. Examining the `strace` output can give valuable information about a process's data access pattern by parsing the `open`, `read` and `write` system calls. With this, we trace the MPI-Caffe processes during deep learning training and get a system call timeline as illustrated in Figure 4.

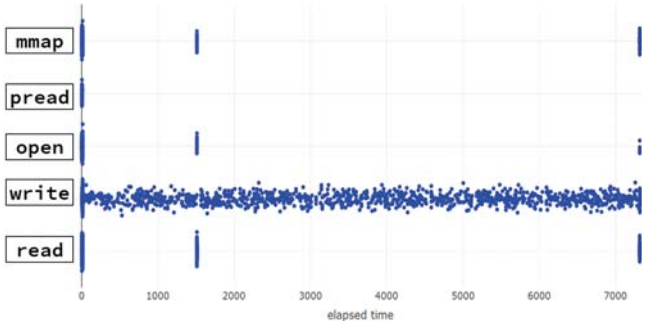

Fig. 4 `strace` diagram for a run including both training and testing phases

The `strace` timeline brings to light the different phases occurring during the learning process. The *write* timeline shows a series of clusters of writes separated by blank intervals. Because writes mainly happen during the training phase when we update the model parameters and write to logs, we infer that those clusters of writes correspond to the training phases and the blank spaces to the testing phases. Note that the y-value jitter is only for visualization, that is to say just manually introduced to help with reading the plot. Two *open* areas appear at the beginning of the timeline aligned to the first training section, corresponding to the opening of the LMDB training and validation data files. The second one is aligned with the first testing blank space and corresponds to the opening of the LMDB testing data file. Two similar areas of *read* calls are aligned with *open*. We infer that those areas correspond to data headers from the LMDB data files. However, we can't see any *read* calls which would correspond to the image accesses. This is because LMDB read accesses are made through memory maps, which involve no system calls, because processes can directly access data from virtual memory. We also notice `mmap` system calls that are aligned with the *open* areas, which correspond to the moments when the LMDB data addresses are mapped to the process address space before access.

Because `strace`'s scope is limited to userspace I/O events, we can't obtain further information on the training read access because of its use of memory-mapping. We decide to use other tools for this purpose.

5.2 blktrace analysis

`blktrace` is a Linux utility for tracing I/O at the device block level. Unlike `strace`, `blktrace` can provide information about requests happening at the system level, between the kernel and the storage devices. We trace MPI-Caffe data accesses with `blktrace` and use `blkparse` to obtain a formatted output.

Additionally, because `blktrace` outputs information corresponding to the raw device blocks, we mapped those to file offsets in order to have consistent results across different machines. The tool we used was `xfs_bmap`, which the XFS filesystem provides to obtain block mappings for specific files. Using this information, we converted the raw block information to filesystem file offsets. From this output we created a timeline of the LMDB data block accesses as illustrated in Figure 5.

Here, the y axis corresponds to the LMDB data file offset, with 0 meaning the beginning of the file. As we can see, although each MPI process accesses its own set of images, they all pass through the entire file, from the beginning to the end. In the MPI-Caffe implementation, each process reads an image, then skips $n-1$ images (where n is the number of processes), then reads the next one, and so on. This is also visible by the mixing of differently-coloured (and differently-shaped) points in the plot.

This data access pattern produces significant overhead for large file access as it involves a lot of skipping operations—one after each image, and also because each process accesses a data range through the entire file. The effects of this are that there is some useless data cached for each node—the overflow data after each image that is read by necessity of the underlying storage interface. This can cause the entire data to not fit into memory, reducing caching performance.

This *skipping pattern* was the natural implementation choice during MPI-Caffe development as it does not involve significant modification of the Caffe data layer source code. Moreover, the development priority of MPI-Caffe was to produce a parallel version of Caffe, leaving data access optimizations for further work. In the next section, we propose an adapted implementation of

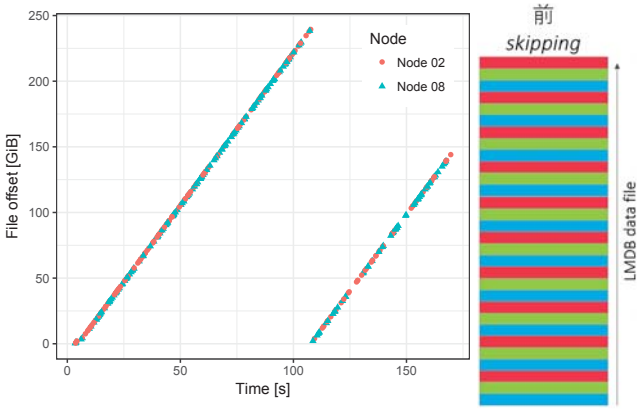

Fig. 5 `blktrace` visualization and diagram for the *skipping* pattern

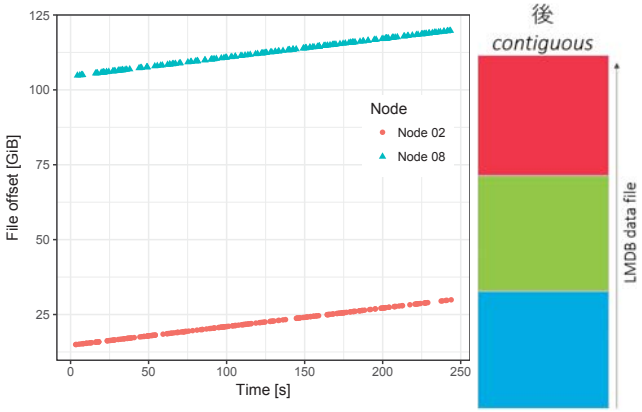

Fig. 7 `blktrace` visualization and diagram for the *contiguous* pattern

data access for MPI-Caffe which eliminates the *skipping pattern* overhead and limitations on distributed filesystems.

6. SEQUENTIAL PATTERN PERFORMANCE IMPROVEMENT

We modified the MPI-Caffe implementation to make each process access its logically separated part of the data, *contiguous*ly, without any skipping between images. Each process has a specific region of the file logically assigned to it, corresponding to the colours in the plot, and to the set of images the process accesses during training. This implementation has the merit of avoiding going through the entire file and reading unnecessary data. As shown in the Figure 7 `blktrace` visualisation, each process seeks to its first image at the start and then only goes through its own part of the data.

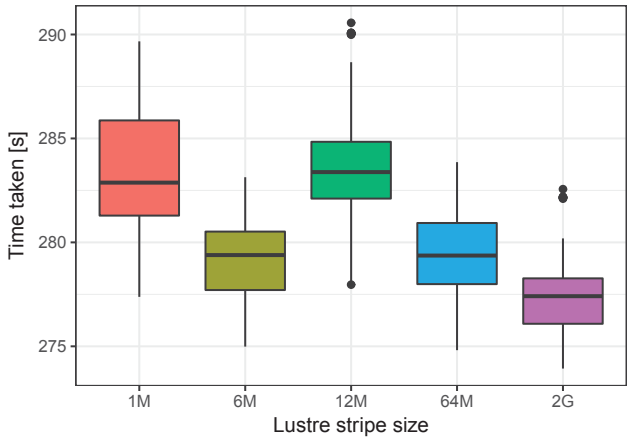

Fig. 6 Training time for different *stripe sizes* with the *contiguous* pattern

Notably, we have no second read of the data in the improved case, which we did have in the *skipping* implementation. This is due to the clients being able to cache data into local memory completely, whereas in the *skipping* implementation, clients also have to cache some unnecessary data, which is the overhead from each skip-read cycle.

Performance analysis results for different stripe sizes, depicted in Figure 6, show that with the *contiguous* implementation the optimal stripe size is now larger, not 1 MiB, as it was with *skipping*. This is due to larger sequential reads and the FEFS read-ahead functionality, as described in the next section. All performance results where the stripe size is not explicitly mentioned for the *contiguous* pattern use the 64 MiB stripe size, as it showed good performance.

6.1 FEFS cache and read-ahead

The main benefit of this pattern is that the application caches only a specific part of the file for each process, reducing the cache storage capacity requirement. Sometimes, all data required by a single process can fit into the local compute node memory which can be reasonably large. In this case, the future demand for the data (from the second pass onwards) can be performed faster by directly accessing the local cache—during the first pass, the process data is automatically cached by FEFS on the compute nodes for further access.

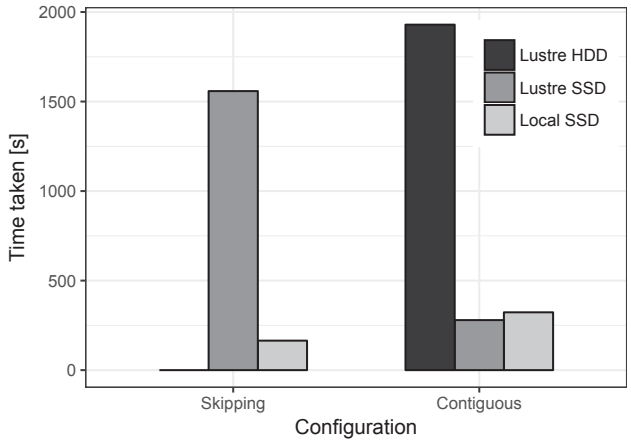

Fig. 8 Final performance results, comparing the original *skipping* to our improved *contiguous* pattern

FEFS also provides heuristic *read-ahead* functionality that prefetches additional data into the client cache before an explicit request. The read-ahead algorithm detects sequential read accesses performed by the client and takes the initiative to pre-read some more data. The size of this additional read is the *read-ahead window*. The more the frequency of sequential reads increases, the more the read-ahead window grows (up to 40 MiB, according to the documentation [10]). Because the *contiguous pattern* always reads the next data block (except for the last one where it loops back to the beginning), the next few data blocks are pre-fetched by the read-ahead function which improves performance.

Figure 8 shows a significant performance enhancement of the *contiguous pattern* compared to the original *skipping pattern* for SSD-backed FEFS storage. We also notice that the performance on local SSDs is worse than on FEFS, which we attribute to FEFS's advantage with the pre-fetching functionality. The *skipping* implementation result for HDD-backed FEFS storage, as it was over ten times slower compared to the other values.

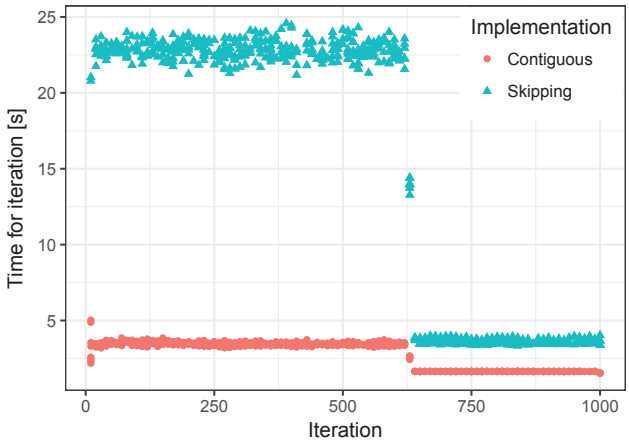

Fig. 9 Detailed per-iteration times

Figure 9 shows a training timeline, where the x axis corresponds to the iteration number, and the y axis corresponds to the time taken for that iteration. As with all our benchmarks, we have performed 1000 iterations of training, which corresponds to 1.5 training epochs. The end of the epoch is noticeable by the reduction in training time at two-thirds of the training.

We see that our improved contiguous implementation is much more consistent in training times, as well as significantly faster both in the first and the later epochs. In the contiguous implementation, in the second epoch, we see that the data is being accessed completely into memory—where it was cached during the first epoch. This represents the best-case performance, as the memory is the fastest storage available for our use-case. This phenomenon does not occur with the original, skipping implementation, as each file is only partially and very loosely cached in local memory, due to pre–caching unnecessary parts of the data as explained before.

7. RELATED WORK

Deploying a distributed deep learning environment on an HPC cluster is a popular research topic and spans various optimization areas. A study [9] analysed different image storage backends for training convolutional neural networks with Caffe. They enumerate possible solutions and find that using a key-value store like LMDB gives significantly better performance compared to the other options. However, this study only considers local training and not data access from remote disks as we did in this analysis.

A recent project [13] focuses on reducing the network contention occurring during parameter updates to improve deep learning training scalability. They exploit the layered model structure of neural networks to overlap communication and computation which has some similarities with the optimization implemented in MPI-Caffe.

Another number of projects such as CaffeOnSpark [11] and BigDL [6] chose to distribute deep learning training using Apache Spark instead of MPI. They leverage Spark features (Resilient Distributed Datasets) to distribute the workload among remote Spark instances and in some cases import the Caffe runtime within the Spark JVM. Spark has many merits: it caches the training data within the JVM heap memory, thus increasing the performance, provides a high level interface for developing applications and a set of automatic eviction policies which facilitate cache management. However, it abstracts away a lot of low level details which makes fine tuning and custom control harder.

8. CONCLUSION

We performed a step-by-step analysis of the MPI-Caffe data access pattern in a distributed environment and inferred a more appropriate approach to accessing the data. Our analysis leverages different characteristics of the LMDB storage backend, the FEFS file system and the Caffe framework, and uses them to enhance their performance. The new access pattern implementation significantly improved the performance and reduced the training time by 5 to 6 times compared to the previous implementation of the same remote storage configuration.

For further work, we want to leverage the FEFS client cache coupled with a pre-fetch mechanism to make data always available from local memory. As the *contiguous pattern* implemented in this work facilitates the inference of the next required batch of data, we plan to build a pre-cache mechanism based on memory-mapped file access monitoring.

References

[1] BVLC: Deep learning framework by the BVLC, Berkeley University (online), available from ⟨http://caffe.berkeleyvision.org/⟩ (accessed 2017-05-12).
[2] Chu, H.: Lightning Memory-Mapped Database, Symas (online), available from ⟨http://www.lmdb.tech/doc/⟩ (accessed 2017-06-22).
[3] Community, L.: Blktrace: block layer I/O tracer, GNU/Linux (online), available from ⟨https://linux.die.net/man/8/blktrace⟩ (accessed 2017-06-22).
[4] community, M.: Numerical library for Python, University of Montreal (online), available from ⟨http://deeplearning.net/software/theano/⟩ (accessed 2017-06-22).
[5] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K. and Fei-Fei, L.: Imagenet: A large-scale hierarchical image database, *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, IEEE, pp. 248–255 (2009).
[6] Intel: distributed deep learning on Apache Spark, Intel-analytics (online), available from ⟨https://github.com/intel-analytics/BigDL⟩ (accessed 2017-06-22).
[7] Kenichiro Sakai, Shinji Sumimoto, M. K.: High-Performance and Highly Reliable File System for the K computer, Technical Report 11, Fujitsu Laboratories Limited (2012).
[8] Kranenburg, L.: Strace: diagnostic tool for Linux, Sun Microsystems (online), available from ⟨https://strace.io/⟩ (accessed 2017-06-22).
[9] Lim, S.-H., Young, S. R. and Patton, R. M.: An analysis of image storage systems for scalable training of deep neural networks, *system*, Vol. 5, No. 7, p. 11 (2016).
[10] Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T. and Huang, I.: Understanding lustre filesystem internals (2009).
[11] Yahoo: CaffeOnSpark, Yahoo (online), available from ⟨https://github.com/yahoo/CaffeOnSpark⟩ (accessed 2017-06-22).
[12] Yamazaki, M., Kasagi, A., Tabaru, T. and Nakahira, T.: Accelerating a Deep Learning Framework with MPI, Technical Report 6, Fujitsu Laboratories Limited (2016).
[13] Zhang, Zheng, X.: Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters, *Proceedings of the international conference on Supercomputing*, USENIX Association, pp. 22–32 (2017).