IPSJ SIG Technical Report

Vol.2017-HPC-160 No.40
2017/7/28

Design of a data supply mechanism for distributed deep learning

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

Abstract: Distributed deep learning frameworks increase the demand for data supply as such computation enables more data to be processed. A traditional HPC parallel file system can be used for this purpose. However, disk latency and network transfer involved when accessing remote disks on parallel file systems remain the bottleneck as training processes wait for the next batch of data. We propose a pre-fetching mechanism that leverages the FEFS client cache for caching data before the deep learning processes request it.

1. INTRODUCTION

Traditional high-performance computing (HPC) systems separate its compute and storage resources into two distinct parts—compute nodes and data nodes—which are interconnected by a high-speed network fabric such as InfiniBand. This architecture, illustrated in Figure 1, is a legacy of large-scale scientific applications that are compute intensive, where storage I/O operations are only slightly required for initial data input, some periodic check-pointing and final output. However, in the era of big data, artificial intelligence (AI) applications, such as deep learning, became increasingly more data-intensive and require much more support by the underlying data storage system that a traditional HPC architecture does not provide out-of-the-box. A significant performance limitation of large scale deep learning applications on HPC infrastructures is the high latency occurring when compute nodes request data stored on remote disks.

In the last decade, many software frameworks have been developed to support data–intensive computing, including distributed file systems (HDFS), parallel programming models (MapReduce, Apache Spark), cluster resource management systems (YARN, Mesos), and machine learning frameworks. Nevertheless, those frameworks are not adapted for the HPC infrastructure. Hadoop co-locates storage and computation resources to achieve high I/O throughput, but this involves a cost for fault tolerance by using multiple replicas of the data, whereas Apache Spark provides a high-level interface for parallel computing and data management features, such as data partitioning with RDD and lineage recovery, but its instances rely on the Java Virtual Machine, which makes computation slower than native C-based MPI applications.

A possible method of reducing data transfer latency between clients and data servers is to leverage client-side caching. This technique places temporary partial data replicas into compute nodes' memory to accelerate client data access. For example, an optimised distributed file system for HPC, such as FEFS®, provides this feature for improving I/O performance of parallel applications. FEFS, the *Fujitsu Exabyte File System*[*1] is a clustered distributed file system [5] based on the Lustre file system. It is worth noting that using client-side caching to accelerate deep learning data access does not involve cache coherency problems as training processes only require a read-only access to the data. However, due to the large input data size of common deep learning application, the main limitation of client-side caching is the memory storage capacity which restricts the amount of data we can cache. This limitation can be solved with an intelligent cache policy which places the right data in memory just before the application needs to access it.

In this paper, we propose a client-side caching system designed for deep learning applications, which uses memory-mapped file based storage provider. This mechanism is based on memory-mapped access detection, which infers the current training progress and thus the appropriate piece of data to pre-fetch in memory. The memory access detection implementation uses a combination of user space techniques involving memory protection, signal handling and code injection. This specific choice is the result of a deep analysis we made [11] on the cache capabilities of the FEFS file system, the characteristics of deep learning data storage backends and the training data access patterns. Our experimental results show that training processes can access data directly from the FEFS client cache, which reduces the training time by 32 %. The rest of this paper is organized as follows: Section 2 overviews our data storage system and gives some relevant information based on a previous work [11]. Then Section 3 describes the implementation details of our data supply mechanism. Next, Section 4 presents our HPC cluster and the evalua-

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

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

tion results we obtained. Finally, Section 5 concludes this study, suggesting possible future directions.

2. DEEP LEARNING ON AN HPC CLUSTER

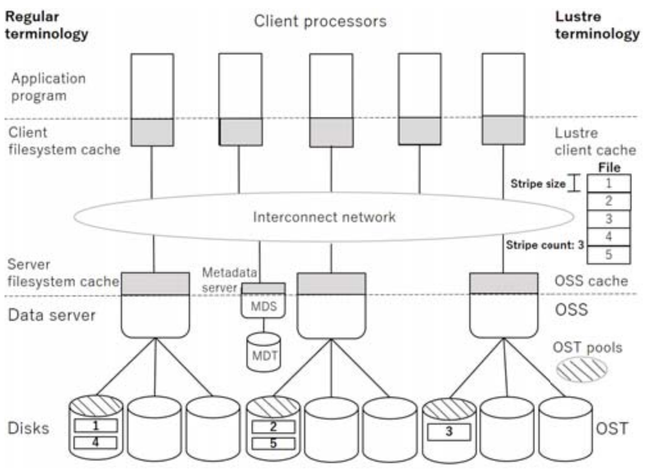

Fig. 1 A traditional HPC parallel file system configuration.

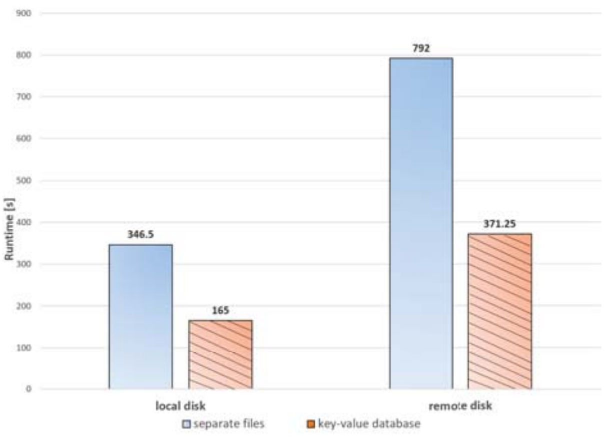

Fig. 2 Performance evaluation for CNN training over ImageNet on two storage systems.

2.1 Storing the data

When we use deep learning frameworks such as MPI-Caffe [17] to train neural network models, we must consider how to store the image data. Multiple methods exist to store the dataset, such as storing each image into a separate file within the file system, or storing images into a dedicated database. A study evaluated the efficiency and performance of different image storage backends for training convolutional neural networks (CNN) [8]. This study observed that using image files on a local file system can result in up to 17 times slower training time compared to using an optimized key-value database. Furthermore, this performance difference gets greater as we use a large input dataset such as ImageNet [3]. Figure 2 gives a performance comparison between these two storage systems (separate files for each images and a global key-value store) when training the GoogLeNet model with the ImageNet dataset for 1000 iterations with Caffe. The results describe disk throughput with caches initially cleared, and remote caches disabled (see Section 4.1 for more environment details).

The performance gap between files and database storage systems is mostly explained by the inefficient support from the local file system [8]. The local cache mechanism cannot understand the existing correlation between different files located in the same directory. When a file data block is accessed, the file system tries to cache the other blocks composing this file but does not take into account other files. Using separate files slows down the training as multiple image files (corresponding to the batch size) need to be read simultaneously. Moreover, when the number of files is large, the overhead caused by file system meta-data operations in order to locate blocks results in longer training times.

2.2 Key-value store

Key-value databases are designed to store and retrieve simple information by providing mappings between keys and values. In the case of image data storage, the key usually corresponds to the image identifier (e.g. the original filename) and the value corresponds to the actual image pixel matrix. Caffe lets the user store and process input image data with LMDB (*Lightning Memory-Mapped Database*), an efficient B+ tree–based key-value store. When a process opens and accesses an LMDB database, the data file is entirely exposed in a memory map, meaning that a page table stores the mapping between the process's virtual address space and the actual physical memory addresses or disk blocks it refers to. Thus, all data access return values directly from the mapped (virtual) memory avoiding any `malloc` or `memcpy` overhead, which makes it optimized for read operations [12].

2.3 Accessing data

In a previous work [11], we analysed the data access pattern of MPI-Caffe while training a CNN model. The input data was an LMDB file stored and striped over FEFS data servers. The deep learning process is a repeating sequence of training and testing phases. In both, sequential reads occur. During our experiments, we noticed that the way we stored the LMDB file (stripe size, lock file management) had an impact on performance. Moreover, we also noticed that FEFS built-in functionalities might somewhat improve the data access performance for specific application I/O patterns. The following lists the significant findings of this analysis:

- The LMDB file should have a stripe size aligned with the training batch size as $stripesize = batchsize * N$ where N is the number of GPUs per compute node used during the training. This batch size parameter is the one specified in the Caffe `.prototxt` configuration file.
- When multiple MPI processes access the same shared LMDB file for learning, multiple replicas of the LMDB lock file should be used—one file per process. This is a useful technique to effectively disable LMDB file locking, enabling reader processes to access the shared file without any restrictions. This should, however, only be used with read-only programs. 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.

- FEFS provides heuristic *read-ahead* functionality that pre-fetches additional data into the client cache before an explicit request. The *read-ahead* data are the next few data blocks located just after the data recently accessed by client requests. The read-ahead algorithm detects sequential read accesses performed by the client and takes the initiative to pre-read slightly more data. The size of this additional reading is called 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 [13]).

In the original MPI-Caffe implementation, each MPI process continuously reads one image then skips $n - 1$ images, where n is the number of processes, then reads an image again, and does so until the training ends. We call this implementation the *skipping* pattern. Our analysis showed that the skipping pattern is not well adapted to FEFS performance characteristics as it involves "random" accesses. For the sake of performance, we implemented a *contiguous* pattern [11] in which each process only accesses a specific area of contiguous images. Therefore when a process requests a new batch, it accesses contiguous images in a sequential read fashion. This pattern is better adapted to the FEFS implementation and provides significantly better performance (dividing the training time by 5 up to 6).

2.4 Improving performance

Although we improved the way MPI-Caffe accesses training data on FEFS, the performance is still not optimal. The best I/O performance can be reached when local memory devices serve as the storage [16]. To achieve this kind of memory performance, the training data needs to always be present in the compute node memory before an explicit request occurs. The challenge then consists of finding which memory pages have to be kept in memory for further access. A common method to handle this task consists of analysing virtual memory accesses, gathering page information and then assigning a level of priority to a data page according to some metrics. The higher the priority of a page, the bigger the probability that the page will soon be accessed. This method, which is based on prediction and statistics is well suited to the optimization of application whose data access pattern is unknown. However, in our case, we know exactly how MPI-Caffe accesses its input data. Therefore, if we could infer the current progress of the training data access at a specific point, we would be able to pre-fetch the relevant data in memory. In other words, if we realize that image number n is being processed, we know that the next image to be processed is $n + 1$, as we know that MPI-Caffe reads images sequentially.

In this paper, we propose a way to dynamically infer the current training data access progress to pre-cache relevant data from FEFS data servers to the client—side FEFS cache, which is stored in memory. This inference is based on detecting accesses to memory-mapped data with a combination of user-space level techniques. We describe it in detail in the following section.

Before implementing what is described in the following sections, we made a prototype by modifying the MPI-Caffe code directly and obtaining information about data accesses internally, while using the same optimisation patterns as described below. This proof-of-concept worked well, so we proceeded to create a solution that does not rely on modifying application code directly.

3. PRE-CACHING MECHANISM

3.1 Memory-mapped file access detection

This section describes how we designed a memory access detection mechanism for deep learning process using a memory-mapped file, specifically an LMDB database, as the data storage backend. As explained before, the LMDB file address space is mapped into the user process which opens it for I/O operations. As Figure 3 shows, when the process requests a specific data block, it can either fetch it directly from physical memory or from the disk in the case of a page fault. Our goal is to maximize the access from memory, in advance, performing prefetching from the disk.

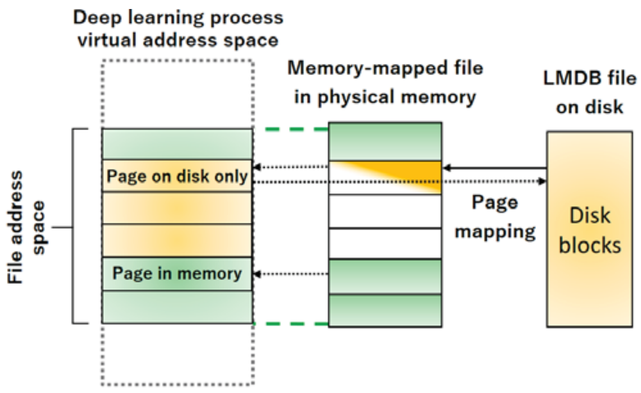

Fig. 3 memory-mapped file mechanism

The idea to infer the current training data access progress—so that we can pre-cache relevant data—is based on the *protection* of specific memory locations (e.g. Figure 4). Summarily, in advance, we protect specific LMDB memory-mapped addresses. We attempt to do so before the deep learning process tries to read data from the LMDB file, but as we are constrained by the fact that we need to wait for the process to open the LMDB file, that is sometimes infeasible. Whether we are able to protect the memory checkpoints before any reading occurs does not affect functionality, only performance, as will be explained later.

When the MPI-Caffe process then tries to access a protected memory page, the kernel triggers a segmentation violation signal `SIGSEGV` for the process. We leverage this situation to catch the signal via a *signal handler*. A signal handler is a function that is first registered to handle a specific set of signals, and then automatically called by the operating system when one of those signals occurs. When the `SIGSEGV` signal is caught, the execution of the process is stopped until the signal handler returns, at which point the execution resumes at the specific instruction at which it was interrupted. The idea is to use the signal handler to trigger a specific process, the job of which is to pre-cache the relevant batch of data into the FEFS client cache from remote OSTs, then to disable the memory address protection for the scheduled read access and finally to resume the execution of the program.

The I/O process, triggered by the signal handler, pre-caches a specific amount of contiguous data blocks from the data block requested by the process—the block whose memory-mapped page was protected. The reason for this pre-cache policy is simple: we know that MPI-Caffe reads image data contiguously and in sequential order, so, if we detect an access to a memory address, we can easily identify the next batch of data the training must access. When the signal handler returns, the process execution resumes and the same instruction, the one called before the interruption, is executed. This time, the instruction does not trigger the checkpoint, as it has been unprotected from within the signal handler. The process can therefore continue its reading operation, which occurs at memory speed because the next batch of data has already been pre-cached by the optimizer process. It is worth noting that each time the handler removes the protection for a specific checkpoint, it also has to re-install the protection of the previous checkpoint, because the training process iterates over the same file multiple times—corresponding to the epochs—and we need to be able to detect memory accesses to that checkpoint again.

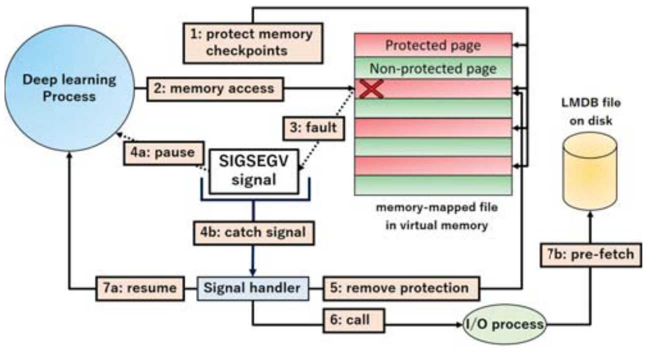

Fig. 4 Memory-mapped access detection

3.2 Implementation details

We implemented this mechanism using a combination of user space techniques. First of all, our main program has to get the whole memory-mapped file memory address mapping. The Linux kernel provides a way to get this information in a stable format through the `/proc/pid/maps` file. The process can then leverage this information to perform operations on specific memory-mapped file locations such as protecting the memory.

The memory address protection is performed with the Linux `mprotect` system call. This function changes the access protection parameters for the calling process's memory pages containing any part of the address range passed in parameter. Consequently, if the calling process tries to access a protected memory page, the kernel triggers a `SIGSEGV` signal for the process.

The way we protect memory addresses can have a big influence on the training time. In our implementation, specific memory addresses, separated at regular intervals from the beginning to the end of the memory-mapped file, should be protected. We define the following terminology for the rest of this paper, also portrayed in Figure 5:

- A *checkpoint* is a protected memory address.
- A *triggering checkpoint* is a checkpoint that triggers a pre-cache on access.
- A *non-triggering checkpoint* is a checkpoint that does not trigger a pre-cache when accessed (steps 6 and 7b from Figure 4 do not occur).
- The *distance* is a fixed spacing that separates two consecutive checkpoints.
- The *size* is a fixed amount of contiguous data, pre-cached sequentially by the I/O process.
- The *process data range* is the total amount of contiguous data a process has to read during the entire training. From the application's point of view, it corresponds to the set of images the process has to read. A process data range has a logical memory area delimited by a *start address* and an *end address*.
- The *start address* is the memory address corresponding to the first image the process has to read.
- The *end address* is the memory address corresponding to the image after the last image the process has to read.
- The *non-cached distance* is the amount of data that separates the start address and the first checkpoint located in the process data range.
- The *pre-cache threshold* is the memory address from which the next checkpoint will be a triggering checkpoint. In our implementation, the pre-cache threshold is set to 75 % of the *size*.

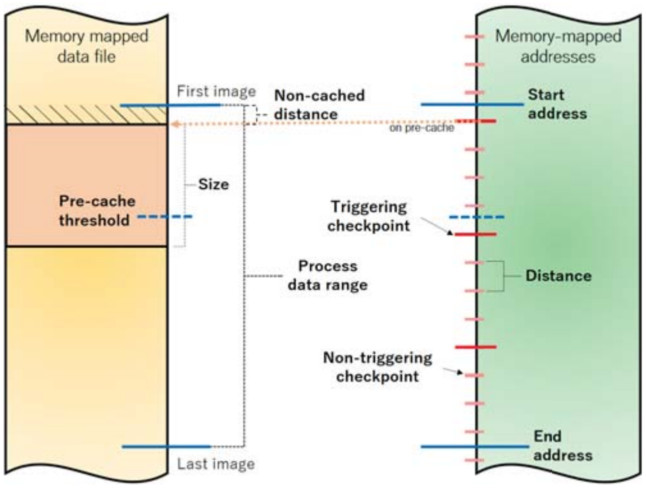

Fig. 5 Terminology: the pre-cache process' point of view

The *distance* determines the frequency at which we can detect memory accesses. Shorter distances improve the reliability and accuracy of memory access detection (so the probability of pre-caching data when needed increases), but they introduce the potential of increasing overhead. The triggering/non-triggering states of checkpoints are not fixed at compile time but implicitly determined at runtime according to the process data range and behaviour. The checkpoint the process hits first always becomes a triggering checkpoint. As the checkpoint positions are defined by the *distance*, it is a common situation that the process *start address* does not match a checkpoint position.

This situation creates the *non-cached distance* which is the main weak point of our idea as it can slow down the performance of the training at the very beginning. It is also common that a *size* end address does not match a checkpoint position. To avoid the same *non-cached distance* weakness, we do not take the checkpoint following the *size* end address as the next triggering checkpoint, but, instead, we trigger the prefetch at a checkpoint that is still in the pre-fetched area. To do that, we define the *pre-cache threshold* that corresponds to a memory address within the pre-cached area, from which the next detected checkpoint will trigger a pre-cache. We chose 75 % as a good value for this, but, as long as a checkpoint lands in the area between the threshold and the end of the current prefetch area, the optimisation occurs. When we hit such a checkpoint, the next pre-fetch occurs not from the checkpoint, but from the end of the current prefetch area, thus removing any pre-fetch overlap.

3.3 Code injection

We developed a memory-mapped access detection method based on `mprotect` and a signal handler. Changing the access protection for the deep learning process's memory page and making the signal be sent to this process requires to implementing those features within the deep learning application. However, because of compatibility reasons, we would like to make this mechanism independent of any deep learning application, consequently we took it upon ourselves to create a pre-caching mechanism that does not modify the deep learning framework source code. This removes the need for adapting to different frameworks, which saves time and effort, as it would require an unmanageable amount of work to develop such a system for each different deep learning framework.

To integrate memory protection and signal handling capabilities within the deep learning framework without any changes to it, we used a *code injection* approach by using the Linux `LD_PRELOAD` environment variable. This variable can be used at runtime to force a user shared library to be loaded and used with a higher priority than other libraries a program loads. With this, we can independently implement the memory protection and signal handler functionality in a separate library, compile it into a *shared object*, then inject it with *LD_PRELOAD* into the MPI-Caffe runtime.

We achieve this goal with a GCC–specific feature that allows executing code before the `main` function by applying the `constructor` attribute to a function. Another aspect we need to handle is the application overriding our signal handler—for that purpose, we proxy the `signal` and `sigaction` POSIX functions and prevent overriding our `SIGSEGV` handler.

In our implementation, illustrated in Figure 6, the deep learning application and the shared library are launched together by the main program called *optimizer*, the job of which is to get the checkpoint locations from the `/proc/`*`pid`*`/maps` file, send them to the deep learning application injected code and wait for a signal handler notification to trigger a pre-cache task.

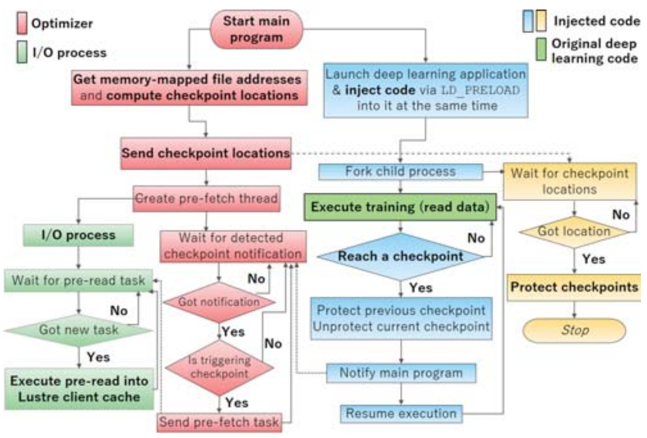
Optimizer
I/O process
Start main program
Injected code
Original deep learning code
Get memory-mapped file addresses and compute checkpoint locations
Launch deep learning application & inject code via LD_PRELOAD into it at the same time
Send checkpoint locations
Fork child process
Wait for checkpoint locations
Create pre-fetch thread
Execute training (read data)
I/O process
Wait for detected checkpoint notification
Reach a checkpoint
Got location
Wait for pre-read task
Got notification
Got new task
Protect previous checkpoint Unprotect current checkpoint
Protect checkpoints
Stop
Is triggering checkpoint
Execute pre-read into Lustre client cache
Notify main program
Send pre-fetch task
Resume execution
Yes
No

Fig. 6 Implementation details flowchart

4. EXPERIMENTAL EVALUATION

In this section we evaluate our data supply system with memory access detection and the pre-cache mechanism. We first characterize the original MPI-Caffe performance without any optimizations, then we evaluate the average training time when using our system so that we can see the performance improvement we achieved. We also compare the performance when accessing data from different data storage types such as local SSD and remote FEFS data servers and compare our system performance result to the theoretical best performance which corresponds to a full access from local memory. Finally, we measure the overhead of our system and discuss some improvements we may make to this system. Plot 8 describes the results with different *size* and *distance* parameter values so that we can evaluate their effect on the performance.

4.1 Experimental setup

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 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. We set the FEFS stripe size to 12 MiB (6 MiB batch size x 2 GPUs) and the FEFS stripe count to 12 (3 OSS x 4 SSD OSTs). For the purpose of these experiments, we turned off the FEFS OSS and cleared the FEFS client and the

Linux page cache before every experiment, and used both SSD and HDD OSTs for data access.

On the FEFS clients, we deployed MPI-Caffe 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.

Figure 7 shows the performance results of our experiments. The y axis represents the training time and the x axis the different configurations we tested:

- *baseline*: original MPI-Caffe implementation
- *prefetch-off*: MPI-Caffe + memory-mapped access detection only. This configuration detects memory access with *mprotect* and signal handling but never triggers any pre-fetching.
- *prefetch-on*: MPI-Caffe + memory-mapped access detection + pre-fetching.
- *warm cache*: original MPI-Caffe accessing data from local cache only (a previous run already warmed up the cache).

Each configuration shows results for when data is accessed from local SSDs and remote FEFS SSDs. *prefetch-on* and *warm cache* also show remote FEFS HDD results, whereas the other two do not because the training time was over 10 times longer than for other devices. For each configuration, the results have been measured ten times and there is no significant variance between them. *baseline* and *prefetch-off* give identical performances, which demonstrates the fact that our memory access detection incurs no significant overhead. In these configurations, FEFS SSD shows slower training time than local SSD—this phenomenon can be explained by the FEFS read-ahead functionaility which, unlike the standard Linux cache, pre-reads some amount of data upon detecting sequential access.

The *prefetch-on* configuration brings significant improvements compared to *baseline*, no matter the storage used. FEFS SSD training time decreases from 280 seconds down to 180 seconds reducing the training time by **32 %**. The local SSD improvement is even better, where the time goes down from 320 seconds to 160 seconds, reducing the training time by **50 %**. It is worth noting that the described speed-up only considers the improvement made on the first pass (which is the first access to the data—during the first epoch), so that the results correspond effectively to the pre-cache mechanism improvement. The local SSD result equals the warm cache performance which is the theoretical best performance. However, a slight difference exists between the *prefetch-on* and *warm cache* performance for FEFS SSDs. This phenomenon is explained by the small amount of data corresponding to the *non-cache distance* (see Section 3.2) which are accessed from the disk instead of the cache, thus slowing down the training.

Figure 8 focuses on the *prefetch-on* configuration. It shows performance results for the three devices with different *distance* (spacing) and *size* values (see Section 3.2). The parameter values are given in mebibytes for six different configurations. Short *distances* (8 MiB and 10 MiB) give the best results, especially with shorter *sizes* 50 MiB. These results make sense: we can more frequently detect the checkpoints and trigger smaller (therefore faster) pre-cache events which focuses on the most needed data. This mechanism can be seen as an extension of the FEFS read-ahead cache with a bigger window and some assumptions about the data access pattern. Higher values of the *distance* parameter detect memory accesses less frequently, thus triggering less pre-caching. Higher *sizes* pre-cache more data, which takes more time and may also some times pre-fetch unnecessary data, when detecting a seeking (as opposed to reading) memory access.

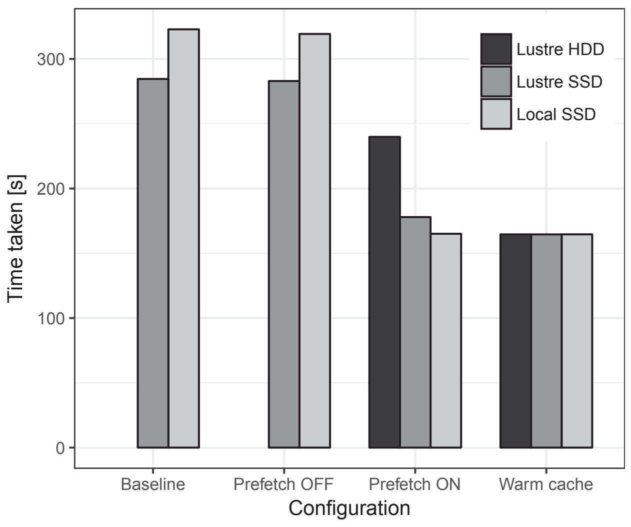

Fig. 7 Performance results.

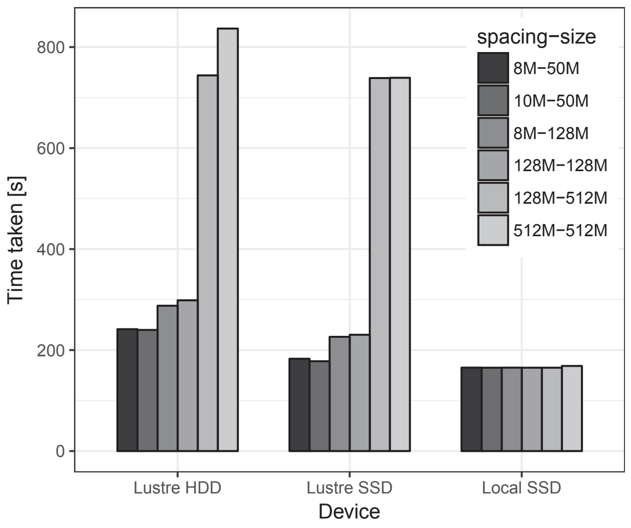

Fig. 8 Performance results for different spacings and pre-fetch sizes.

5. RELATED WORK

There are many works related to building efficient data storage systems for data-intensive computing in HPC systems. A major research direction consists of integrating Hadoop with the HPC infrastructure. Some work explored deploying Hadoop on existing parallel file systems such as FEFS, Ceph [15] or GPFS [9]. They developed a suitable data mapping between parallel file systems and Hadoop and showed some performance enhancements. However, I/O bandwidth of parallel file systems remains the bottleneck of such systems. In a prior work, we deployed HDFS [10] onto a cluster of HPC compute nodes as the main data storage system for MPI-Caffe. To efficiently supply HDFS data blocks to MPI processes, we implemented an inter-process communication module between MPI and Spark instances using shared-memory [4], [18]. This system combines the Hadoop and MPI merits: high I/O throughput, horizontal scalability, fault tolerance and finely tuned control flow.

A second research direction focuses on using memory to increase I/O performance of parallel file systems. One project [14] introduced a dedicated buffer layer deployed into HPC data nodes to buffer the burst I/O. A recent work [16] also developed a two-level data storage system for integrating HPC and data-intensive computing. They built a hybrid data storage system using an HPC parallel file system, OrangeFS [1] and a big data in-memory file system called Alluxio [6]. Using HPC compute node memory as part of storage, with Alluxio integrated atop of OrangeFS, they provided temporal locality of data and a read throughput enhancement.

A third research direction dove deeply into the data storage device design. One project [2] is based on building efficient hybrid storage systems using SSDs supporting HDDs for random and write operations. They developed a system that can identify frequently accessed blocks, the root of long latency, and move them from the HDD to the SSD for future access improvements.

Our work is mainly focused on improving read data access for deep learning applications using client side caching which is based on data prefetching with memory access detection. Our mechanism is deeply coupled with the knowledge of deep learning data access patterns which makes it unusable with applications with different data access patterns, and also makes use of rudimentary prediction of sequential accesses, which means we go beyond a reactive system. A project [7], which resembles our work, also leverages client side file caching to minimize the data server access contention. They developed a global cache pool made by each compute node's process local memory buffer and the MPI I/O collective operation to solve cache coherency problems. However, this work focuses on using collective caching to export cache coherency management into the compute nodes which is in our case slightly useless because we focus on read-only MPI-Caffe processes.

6. CONCLUSION

We designed and evaluated a practical way to efficiently supply data to distributed deep learning processes. Our memory-mapped access detection enables training processes to access image data from local cache which corresponds to a training time reduction of 32 % compared to the original MPI-Caffe implementation. Our user-space implementation has no significant overhead and the code injection we used permits re-usability with other deep learning frameworks using LMDB or memory-mapped file backends. Our performance results show a significant training time improvement when using pre-fetching. FEFS performance can be improved if we minimize the *no-cache* distance between the process *start address* and the first *checkpoint* position. As the training is an iterative process over the data set, we can further our work by continuously improving the detection pattern by using a binary search to dynamically change the position of some checkpoints to make them match the process start address. In further work, we plan to evaluate the overhead of this system with other configurations and frameworks. We would also like to to explore other approaches such as developing a kernel module to automatically monitor memory-mapped access or leverage idle compute nodes' free memory as a global cache for deep learning processes.

References

[1] Carns, H., Ligon, B. I., Ross, B. and Thakur, R.: The OrangeFS Project, Parallel Architecture Research Laboratory (online), available from ⟨https://www.orangefs.org/⟩ (accessed 2017-04-20).

[2] Chen, F., Koufaty, D. A. and Zhang, X.: Hystor: making the best use of solid state drives in high performance storage systems, *Proceedings of the international conference on Supercomputing*, ACM, pp. 22–32 (2011).

[3] 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).

[4] Haderbache, A., Miwa, M., Yamazaki, M., Tabaru, T. and Nakashima, K.: Spark as Data Supplier for MPI Deep Learning Processes, Technical Report 11, Fujitsu Laboratories Limited (2016).

[5] Kenichiro Sakai, Shinji Sumimoto, M. K.: High-Performance and Highly Reliable File System for the K computer, Technical Report 11, Fujitsu Laboratories Limited (2012).

[6] Li, H., Ghodsi, A., Zaharia, M., Baldeschwieler, E., Shenker, S. and Stoica, I.: Alluxio - Open Source Memory Speed Virtual Distributed Storage, University of California, Berkeley Hortonworks (online), available from ⟨http://www.alluxio.org/⟩ (accessed 2017-04-20).

[7] Liao, W.-k., Coloma, K., Choudhary, A., Ward, L., Russell, E. and Tideman, S.: Collective caching: Application-aware client-side file caching, *High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE International Symposium on*, IEEE, pp. 81–90 (2005).

[8] 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).

[9] Schmuck, F. B. and Haskin, R. L.: GPFS: A Shared-Disk File System for Large Computing Clusters., *FAST*, 2, No. 19 (2002).

[10] Shvachko, K., Kuang, H., Radia, S. and Chansler, R.: The Hadoop Distributed File System, Yahoo! (online), available from ⟨https://hadoop.apache.org/⟩ (accessed 2017-05-04).

[11] Stanovnik, S., Haderbache, A., Miwa, M. and Nakashima, K.: Performance analysis of a deep learning framework on a high-performance distributed file system, Technical report, Fujitsu Laboratories Limited (2017).

[12] Symas: LMDB microbenchmark (2012), Symas (online), available from ⟨http://symas.com/mdb/microbench/⟩ (accessed 2017-05-12).

[13] Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T. and Huang, I.: Understanding lustre filesystem internals (2009).

[14] Wang, T., Oral, S., Wang, Y., Settlemyer, B., Atchley, S. and Yu, W.: Burstmem: A high-performance burst buffer system for scientific applications, *Big Data (Big Data), 2014 IEEE International Conference on*, IEEE, pp. 71–79 (2014).

[15] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. and Maltzahn, C.: Ceph: A scalable, high-performance distributed file system, *Proceedings of the 7th symposium on Operating systems design and implementation*, USENIX Association, pp. 307–320 (2006).

[16] Xuan, P., Denton, J., Srimani, P. K., Ge, R. and Luo, F.: Big data analytics on traditional HPC infrastructure using two-level storage, *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems*, ACM, p. 4 (2015).

[17] Yamazaki, M., Kasagi, A., Tabaru, T. and Nakahira, T.: Accelerating a Deep Learning Framework with MPI, Technical Report 6, Fujitsu Laboratories Limited (2016).

[18] Zaharia, M., Chowdhury, M., Franklin, J., Shenker, S. and Stoica, I.: Spark: Cluster Computing with Working Sets, University of California, Berkeley (online), available from ⟨https://spark.apache.org/⟩ (accessed 2017-05-04).