## IEEE Copyright Notice



Accepted to be Published in: Proceedings of the 36rd IEEE International Parallel & Distributed Processing Symposium, May 30-June 03, 2022 Lyon, France

# Why Globally Re-shuffle? Revisiting Data Shuffling in Large Scale Deep Learning

Truong Thao Nguyen*, François Trahay†, Jens Domke‡¶, Aleksandr Drozd‡¶‖,
Emil Vatai‡¶, Jianwei Liao§, Mohamed Wahib*‡¶, Balazs Gerofi‡¶
* National Institute of Advanced Industrial Science and Technology (AIST), Japan, {nguyen.truong, mohamed.attia}@aist.go.jp
† Télécom SudParis, Institut Polytechnique de Paris, France, francois.trahay@telecom-sudparis.eu
‡ RIKEN Center for Computational Science, Japan, {jens.domke, aleksandr.drozd, emil.vatai, bgerofi}@riken.jp
§ College of Computer and Information Science, Southwest University of China, China, liaotoad@hotmail.com
¶ Tokyo Institute of Technology, Tokyo, Japan
‖ Amigawa GK, Tokyo, Japan

***Abstract*—Stochastic gradient descent (SGD) is the most prevalent algorithm for training Deep Neural Networks (DNN). SGD iterates the input data set in each training epoch processing data samples in a random access fashion. Because this puts enormous pressure on the I/O subsystem, the most common approach to distributed SGD in HPC environments is to replicate the entire dataset to node local SSDs. However, due to rapidly growing data set sizes this approach has become increasingly infeasible. Surprisingly, the questions of why and to what extent random access is required have not received a lot of attention in the literature from an empirical standpoint.**

**In this paper, we revisit data shuffling in DL workloads to investigate the viability of partitioning the dataset among workers and performing only a partial distributed exchange of samples in each training epoch. Through extensive experiments on up to 2,048 GPUs of ABCI and 4,096 compute nodes of Fugaku, we demonstrate that in practice validation accuracy of global shuffling can be maintained when carefully tuning the partial distributed exchange. We provide a solution implemented in PyTorch that enables users to control the proposed data exchange scheme.**

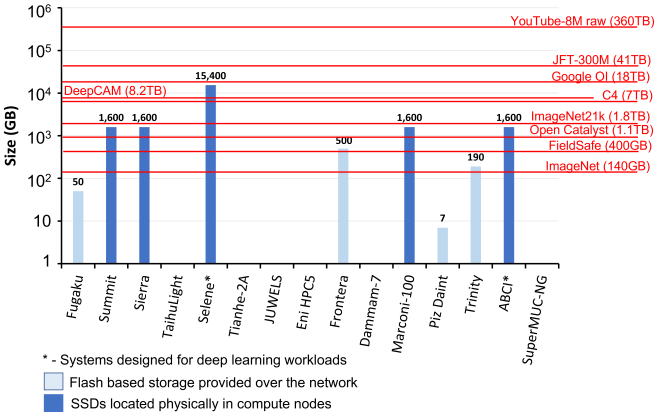



Fig. 1: Dedicated node local storage on fifteen of the fastest supercomputers from the TOP500 list [1] vs. data set sizes. Datasets from top to bottom: [2], [3], [4], [5], [6], [7], [8], [9], [7].

## I. Introduction

Deep Learning (DL) is a class of machine learning methods that is based on Deep Neural Networks (DNNs). Deep learning's success has been fueled mainly by two phenomena, the availability of large amounts of data and the continuous improvements in computational power to analyze the data. In particular, training DNNs is extremely computationally intensive, which has sparked a growing interest not only in creating DL targeted specialized hardware, but also in the utilization of large-scale supercomputers for such workloads.

As neural networks become more complex and require an increasing amount of computing power, ever larger quantities of data are required to train accurate models. Needless to say, the size of data keeps growing rapidly. For instance, while the ImageNet dataset is 140 GiB, recent datasets such as DeepCAM [5], or YouTube-8M [2] already consist of terabytes of data. Because of the large datasets, and the complex neural networks, many machine learning applications now need to be distributed over many workers[1] to speed up training and to fit the dataset in memory.

Distributing the training of a neural network in a data parallel fashion over compute nodes of a supercomputer requires loading the input samples on compute nodes so that each node can process a subset of the samples at each training epoch. This is either done by storing the entire dataset on compute node local storage, or by each node reading a subset of the samples from the parallel file system (PFS). As datasets become larger, storing the entire dataset on local storage becomes impossible since they exceed storage capacities. See Section II for more details and Figure 1 for an illustration of the problem. Similarly, reading from the parallel files system puts an enormous pressure on the storage nodes because many compute nodes read terabytes of data simultaneously [10], [11]. Moreover, to improve generalization, distributed neural network training shuffles the data at each epoch so that nodes

[1]A *worker* in data parallel DNN training is a processing element that maintains and trains its own local copy of the model.

can randomly access input samples, which further increases the I/O requirements of deep learning applications.

This random access to the input samples has been in fact identified as one of the major contributors to poor I/O performance in HPC systems and various solutions have been proposed to tackle this issue at the level of the I/O subsystem [5], [12], [13], [14], [15]. Nevertheless, it is the overarching consensus that random shuffling input elements is a strict requirement and most I/O approaches take this for granted. There have only been a few studies in the past that approach the I/O bottleneck of deep neural network training with an investigative eye on the shuffling procedure itself. Moreover, they provide limited insights on how different approaches impact training accuracy or performance at scale [16], [17].

In this paper, we revisit data shuffling strategies when scaling deep learning applications to a large number of workers. We develop a method that allows us to control the fraction of the dataset that is globally shuffled in each epoch. In our method we implement the following shuffling strategies. Global shuffling (i.e., all of the dataset is shuffled and distributed across workers), local shuffling (i.e., each worker uses the same part of the dataset for each epoch), and a novel partial-local shuffling strategy that exchanges only a configurable proportion of the dataset among workers in each epoch and leaves the rest local. Using our method requires very few changes to existing PyTorch training scripts, does not require any changes to the PyTorch framework itself, and could support any arbitrary format for the samples by implementing a loader for the data.

We provide a proof that partial local shuffling produces the same gradients as global shuffling, and we discuss some factors that could have an unidentifiable effect when training with partial local shuffling. We further investigate the practical conditions for assuring that the shuffling error without dominating the convergence rate. For empirical results, we use our solution to train DNNs on several datasets and we study the impact of various shuffling strategies on accuracy when scaling to a large number of workers. To much of our surprise, the experiments reveal that local shuffling often achieves a similar accuracy as global shuffling, even when scaling to 1,024 workers. In some cases, local shuffling degrades the accuracy. The partial-local shuffling strategy then achieves similar accuracy as global shuffling while only requiring to store to up to 0.03% of the whole dataset. In addition, for data sets that do not fit locally in the first place, partial-local shuffling can improve accuracy compared to the local only access. This opens the doors for leveraging the potential of locality in large scale training of large datasets, and addresses the DL I/O challenge at its root: avoid I/O when possible.

The contributions of this paper are as follows:

- We implement a dataset partitioning, shuffling, and redistribution solution for distributed training (synchronous SGD). Our solution requires minor modification to PyTorch training scripts, a load handler for the dataset, and no modifications to the PyTorch framework itself.
- We show that in the overwhelming majority of our experiments local shuffling achieves similar validation accuracy as the default global shuffling strategy.
- Experiments on Fugaku with up to 4096 workers show that partial-local shuffling achieves the same accuracy as global shuffling, while requiring to store only a very small fraction ($0.03\%$) of the dataset.
- We show that on ImageNet-1K (and ImageNet-21K) we converged to 69% (44%) of top-1 validation accuracy of the global shuffling with an exchange factor of 0.3, while local shuffling lags behind by 10% (3%). We also improve the validation accuracy of DeepCAM by 2% when using partial shuffling in replacement of local shuffling.
- The training time obtained with local shuffling (and partial shuffling in most cases) is up to 5x lower than that of global shuffling.
- We demonstrate that our solution provides a qualitative advantage for systems that have little or no local storage on their compute nodes.

## II. Background and Motivation

Data sizes of input sets in large scale deep learning have been increasing steadily. While the original ImageNet dataset is only approximately 140 GB in size [7], newer data sets are becoming larger. The red horizontal lines in Figure 1 provide an overview of some of the data sets widely used for deep learning training. For example, the Youtube-8M features dataset used in video models is 1.5 TB [2], the DeepCAM data set used as a benchmark in high-performance computing environments is about 8.2 TB [5], while the Google OpenImages dataset has a total size of about 18 TBs [4].

Using stochastic gradient descent (SGD), training jobs run multiple epochs (usually in the range of 50-100 [18]), where each epoch passes over the entire training data in a random access fashion. A reshuffle of the dataset is typically done before each epoch. Due to the high bandwidth requirement as well as the random nature of accessing input samples, DL workloads exert enormous I/O pressure. Training from the Parallel File Systems (PFS), typically available in HPC centers, is infeasible. What makes things even worse for the parallel filesystem is that users typically run tens or hundreds of instances of the same job for hyper-parameter tuning, each with different configurations of the neural network model (e.g., learning rate, activation function, etc.) [18], further exacerbating the I/O problem.

For the aforementioned reasons, the current state of practice in scale-out deep learning training is to replicate the input data set to node-local SSDs in compute nodes. Unfortunately, this is also becoming increasingly unattainable due to the growing size of the data sets. Figure 1 also illustrates this problem by comparing the local storage capacity on fifteen of the fastest supercomputers from the TOP500 list (as of 2020 November) with the size of DL datasets. Note that the Y axis is on log-scale, had we chosen to use linear representation, data set sizes would dominate the plot and render local storage sizes nearly invisible. Dark blue bars represent SSDs located physically

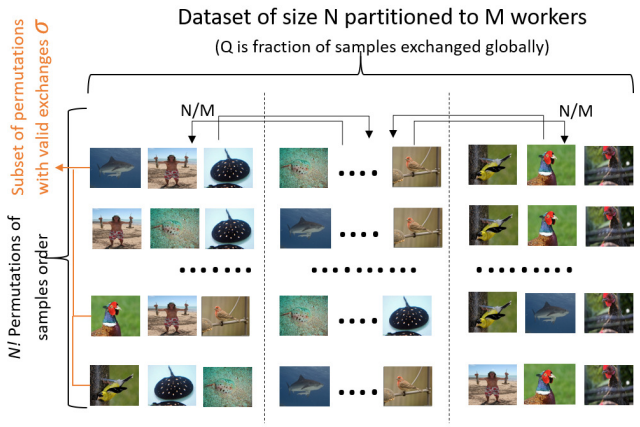



Fig. 2: Overview of how partitioning and exchange is presented as partial local shuffling.

in compute nodes, while light blue bars indicate flash based storage provided over the network. Systems with neither local SSDs nor network attached flash storage are indicated with zero capacity. In addition, we mark systems that have been designed explicitly for deep learning workloads with a star.

For example, Fugaku (the top system in the Top500 list of June 2021) is equipped with one SSD for each group of 16 compute nodes [19]. The shared SSD can be exposed as node-local storage of up to 50GB compute node dedicated capacity. Frontera, Piz Daint, and Trinity, on the other hand, provide access to flash based storage located in separate racks (a.k.a., dedicated burst buffers). For Frontera, Trinity, and Piz Daint we display the proportion of the storage capacity corresponding to a single compute node and note that network attached, shared filesystems typically suffer from similar problems with the global parallel file system, e.g., due to the centralized metadata management that has been identified as a major component of I/O overhead in deep learning [10], [11].

In summary, as seen in Figure 1, many of the data sets are already too large to be loaded on dedicated compute node storage. While systems that were primarily designed for deep learning are best suited than their classic HPC oriented counterparts, even those platforms cannot satisfy storage requirements for all data sets. Overall, we see an urgent need to provide easy-to-use I/O approaches for DL workloads.

## III. Design and Implementation

In this section, we present the design and implementation of our proposed partial local shuffling strategy. To reduce I/O requirements when training DNNs we do the following: a) we split the dataset across workers, b) workers train on the local partition, and c) worker exchange a subset of the locals samples before each epoch (when deemed necessary).

### A. Data Partitioning and Shuffling Scheme

As Figure 2 shows, data partitioning is represented as a shuffle of the dataset, where different permutations represent different ways to partition the data. The worker to whom a sample belongs is determined by the order in which it appears in a permutation. We propose a shuffling scheme in which each worker exchanges globally a fraction $Q$ of its local samples before each epoch. Each worker then combines the $Q$ newly received samples with the remaining portion of $1-Q$ samples and shuffles it locally. A value of $Q = 1$ results in a full global shuffle that corresponds to the without-replacement shuffling scheme typically used in distributed SGD training [20], while a value of $Q = 0$ corresponds to the pure local shuffle at which each worker only has a local view of its portion of the dataset. The partial local shuffling in this paper is a middle-ground between pure global and local strategies.

This partial local shuffling is an extension of existing works [16], [17] that reduce the amount of remote samples to retrieve by keeping a part of the samples locally. Our proposed shuffling scheme ensures that the amount of local samples is balanced across workers, and it allows to control the ratio between the number of local and remote samples.

We assume that each worker's designated portion of the training data samples is loaded into a predefined storage area before training. During training, a worker only processes data samples in its designated storage area. This predefined area can be memory, local storage (e.g., local SSDs) as well as a parallel file system. In the global shuffling scheme (here after $GS$ for short), each worker can access the entire dataset. This requires a storage system that is large enough to store the whole dataset, i.e., $N$ samples. In local shuffling (LS), each worker can only access a subset of the dataset, e.g., $\frac{N}{M}$ (where $M$ is the number of workers), which is often copied into its local storage beforehand. Thus, when using local shuffling, workers process the same subsets of datasets at each epoch throughout the training.

In contrast, when training with partial local shuffling (PLS), each worker stores a subset of the dataset and updates it at each epoch. Before each epoch, each worker sends a selected set of data samples to other workers and consequently, receives new data samples from other workers. After the data transfer is finished, workers remove the data samples they transmitted and save the received samples into their local storage area. It is worth noting that our implementation of PLS requires a $(1+Q)$-fold local storage capacity as it is with LS, i.e., each worker stores up to $(1+Q)\frac{N}{M}$ samples. That is, the required local storage capacity of PLS is at most 2-fold as it is with LS, yet at least still $\frac{M}{2}$ times smaller than that in GS.

### B. Exchanging Samples Between Workers

Once the $N$ samples are initially distributed among the $M$ workers, each worker processes $\frac{N}{M}$ samples locally. It then randomly picks a portion made up of $Q \times \frac{N}{M}$ samples (which we call the *global partition* of each worker) for global exchange. From the global perspective, samples from all the global partitions are aggregated and randomly pushed back to all the workers. This might be accomplished by shuffling the combined global partitions followed by a round-robin distribution of those samples to individual workers. However, due to the high cost of maintaining such a global view, we

**Training code with global shuffling**

```
train_dataset = ImageFolder(train_dir, transformations)
train_sampler = DistributedSampler(train_dataset, size, rank)
train_loader = DataLoader(train_dataset, batch_size=b, train_sampler)
```

**Training code with (Partial) Local Shuffling**

```
train_dataset = PLS.ImageFolder(train_dir, class_file, transformations)
train_sampler = DistributedSampler(train_dataset, size, rank=rank)
train_loader = DataLoader(train_dataset, batch_size=b, train_sampler)
scheduler = PLS.Scheduler(train_dataset, batch_size=b, fraction=Q)
...
train(epoch):
    scheduler.scheduling(epoch)
    ..... # Training loop here
    send_req, recv_req = scheduler.communicate() # Non-blocking exchange
    scheduler.synchronize(send_req, recv_req) # Wait to finish exchange
    scheduler.clean_local_storage() # Remove exchanged samples on the storage
```

Fig. 3: Global vs. partial local sampling in PyTorch.

implement the global exchange using a distributed fashion that only requires a local view on each worker. Specifically, each worker randomly selects a destination to exchange for each sample in its global partition.

**Algorithm 1** Global Exchange

**Input:** Number of samples $N$, Global fraction $Q$, Local batch size $b$, Number of workers $M$, Rank $r$

1: $p \leftarrow$ random permutation of 1 to $\frac{N}{M}$
2: **for** $i$ from $1 \rightarrow Q \times \frac{N}{M}$ **do**
3: $dest \leftarrow$ random permutation of 1 to $N$
4: **isend** sample $p[i]$-th to rank $dest[r]$
5: **irecv** $data$ from ANY_SOURCE
6: **end for**
7: **wait** for all outstanding requests to finish exchanging

As shown in the line $3 \rightarrow 4$ of Algorithm 1, a worker of rank $r$ picks up the destination rank for transferring the sample $i^{th}$ from a random permutation of all the ranks $dest$, where all workers use the same random seed (e.g., setup before training in the training script) to assure single source and single destination for each exchanged sample. This method could guarantee all the workers send and receive the same number of samples, thus providing a balanced communication.

Thus, each worker sends (and receives) $Q \times \frac{N}{M}$ samples and reads $(1-Q) \times \frac{N}{M}$ samples locally at each epoch. For example, when using a partial shuffling scheme with $Q = 10\%$ on 512 workers that load the ImageNet-21K dataset, each worker sends (and receives) $0.1 \times \frac{1.1TiB}{512} = 225MiB$ and reads $0.9 \times \frac{1.1TiB}{512} = 2GiB$ locally. It is to be compared with global shuffling where each worker reads $\frac{1.1TiB}{512} = 2.2GiB$ from the PFS. Each worker then replaces the selected samples by $Q \times \frac{N}{M}$ samples received from other workers to establish a new set of $\frac{N}{M}$ samples used in the subsequent epoch.

### C. Implementation

We implement the proposed shuffling strategies with the aim to simplify the reuse of existing PyTorch training code. It should be noted that PyTorch does not have any strict requirements for data loading, except that it needs to be a *torch.Tensor*, since loading and preprocessing are specific

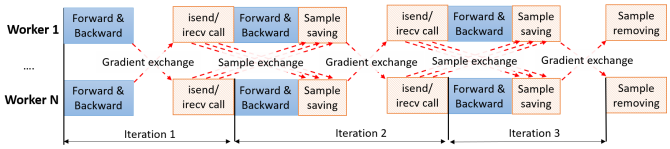



Fig. 4: Illustration of overlapping the sample exchange with the forward and backward phases in a epoch of three iterations.

to the dataset. A recommended approach is to use the two data primitives provided by PyTorch: a *Dataset* to store the samples and their corresponding labels, and a *DataLoader* to iterate over the (batches of) samples. We provide convenience wrappers for *Dataset* to maximise code reuse by minimising the need to modify existing scripts. The newly wrapped dataset (*PLS.ImageFolder* in Figure 3) requires additional functions for saving, and removing the samples from the local storage. The implementation of those functions depends on the way each dataset is organized. This way, we avoid modifying the PyTorch codebase itself as well as the loading function of the original dataset (e.g. the case of 'ImageFolder' dataset).

We then introduce a *scheduler* for managing the global exchange. To reduce the overhead of the sample exchange in each epoch, we overlap communication with the forward and backward phases of the previous epoch (as shown in Figure 4). Let us consider an epoch with $I = \frac{N}{b \times M}$ iterations, where $b$ is the batch size at each worker. In each iteration, $\frac{Q \times \frac{N}{M}}{I} = Q \times b$ samples are sent/received from one worker. We use non-blocking MPI calls, i.e., *MPI_Isend()* and *MPI_Irecv()*, to implement the communication between two workers using the method described in Algorithm 1. The worker then awaits, in the next iteration, for the data exchange to be concluded and removes the exchanged samples from its local storage at the end of each epoch. Figure 3 presents code examples that uses global shuffling (the default for PyTorch Distributed) and our proposed method of partial local shuffling. Only six lines need to be changed/added to enable our shuffling implementation (marked with red in Figure 3).

### D. Deployment

Our guideline for practical deployment is to start with local shuffling and if training accuracy is dissatisfactory, treat the shuffling factor as an additional hyper-parameter of the training process. Given the well established practice of hyper-parameter tuning in current deep learning algorithms we believe that treating the shuffling fraction as a hyper-parameter is a practical condition for the deployment of the proposed method.

### E. Assumptions and Restrictions

Our method is designed for data parallelism where each worker processes a different subset of the dataset. The techniques in this work may not be suitable for supporting pure model parallelism where all the workers process the same data samples (*i.e.*, the whole dataset). Note, however, that our technique is still applicable to hybrid parallelism.

Our solution supports datasets that manage each data sample in a single distinct physical file. However, some datasets manage multiple samples in a single compressed file, e.g., the Open Catalyst dataset [8] allows multiple samples to be co-located in a single LMDB file. Our scheduler could however be simply extended to exchange batches of samples instead of individual samples; the granularity of the exchange does not conflict with the scheme implemented by the scheduler.

## IV. Shuffling in Distributed SGD

This section attempts to establish an understanding of why and when partial local (or simply local) shuffling may achieve accuracy comparable to that of global shuffling. This is a complex topic that needs to be approached both theoretically and empirically from different angles. First we expand on existing work that show that shuffling with some sort of locality (albeit using the same random number sequence) [17] can be equivalent to global shuffling, and we briefly discuss assumptions that need further considerations. Second, we build on a convergence analysis of distributed SGD [20] to assess the impact on convergence rate. We do, however, show that for practical number of workers and dataset sizes, the shuffling error would dominate the convergence rate of the smooth non-convex objective function in the case of partial local shuffling in distributed SGD, and yet convergence, in practice and in most cases is not affected, as shown experimentally in section V. This highlights the challenges imposed on analysing shuffling for distributed SGD, and implies further effort is required to investigate the convergence bounds.

### A. Partial Local vs. Global Shuffling: Gradients

In this section we expand on the proof by Yang et al. [17] which proved that global shuffling is equivalent to sampling with locality. A difference from our paper is that Yang et al. use the same random number sequence in both shuffling types. We expand on the proof to show that our proposed partial local shuffling produces the same gradients as global shuffling.

For synchronous SGD, at iteration $t$, where the worker $m$ in $M$ set of workers computes the gradient over its local subset of samples $b_m^t$, the local gradients are averaged to update the model parameters as follows:

$$w_{t+1} = w_t - \eta_t . \frac{1}{Mb} \sum_{m \in M} \sum_{i \in b_m^t} \nabla f_i(w_t) \tag{1}$$

For the $N_t^j$ dataset samples ($j \in N$), the local samples assigned to worker $m$ are $\left[N_t^{\frac{N}{M}\times(m-1)}, N_t^{\frac{N}{M}\times(m-1)+1}, \ldots, N_t^{\frac{N}{M}\times(m-1)+\frac{N}{M}}\right]$. Assuming both the global and partial local shuffling start at the time step $t$ from the same weights $w$, then at the time step $t+1$ the global sampling would produce the sequence $\left[N_{t+1}^1, N_{t+1}^2, \ldots, N_{t+1}^N\right]$ of samples in the dataset, where the permuted sequence at each worker is $\left[N_{t+1}^{\frac{N}{M}\times(m-1)}, N_{t+1}^{\frac{N}{M}\times(m-1)+1}, \ldots, N_{t+1}^{\frac{N}{M}\times(m-1)+\frac{N}{M}}\right]$. The gradient for this local sequence would be:

$$\nabla f_i(w_{t+1} : \left[N_{t+1}^{\frac{N}{M}x(m-1)}, N_{t+1}^{\frac{N}{M}x(m-1)+1}, .., N_{t+1}^{\frac{N}{M}x(m-1)+\frac{N}{M}}\right]) = \sum_{i \in b_m^{t+1}} \nabla f_i(w_{t+1} : N_{t+1}^{\frac{N}{M}\times(m-1)+i}) \tag{2}$$

The averaged gradient term in Equation 1 can be written as

$$\sum_{m \in M} \sum_{i \in b_m^{t+1}} \nabla f_i(w_{t+1} : N_{t+1}^{\frac{N}{M}\times(m-1)+i}) \tag{3}$$

Unlike the original proof by Yang et al. [17], we do not use the same random sequence. However, with respect to convergence, the partial local shuffling is partially permuting from the portion that is shuffled globally (i.e. $Q$) and the portion that is shuffled locally (i.e. $Q-1$) to amount to what is the original sample sequence. More specifically, $\left[N_{t+1}^1, N_{t+1}^2, \ldots, N_{t+1}^N\right]$ is transformed to a sequence of different sample order $\left[N_{t+1}^{fy(1)}, N_{t+1}^{fy(2)}, \ldots, N_{t+1}^{fy(N)}\right]$, where $fy(n)$ is a Fisher-Yates shuffle that follows the partial local break down of local and global shuffling portions. When partitioning the shuffled sequence to $M$ workers (i.e., worker $m \in M$ receives a sub-sequence $N_{t+1}^{fy(s)}$ to $N_{t+1}^{fy(e)}$), the gradient of partial local shuffling becomes:

$$\nabla f_i(w_{t+1} : \left[N_{t+1}^{fy(s)}, N_{t+1}^{fy(s)+1}, \ldots, N_{t+1}^{fy(e)}\right]) = \sum_{i \in \{s \to e\}} \nabla f_i(w_{t+1} : N_{t+1}^i) \tag{4}$$

The averaged gradient term in Equation 1 can be written as

$$\sum_{m \in M} \sum_{i \in \{s \to e\}} \nabla f_i(w_{t+1} : N_{t+1}^i) \tag{5}$$

The two terms from Equation 3 and Equation 5 are equal (by the commutative property of addition) and therefore the weights updated according to Equation 1 at time step $t+1$ (i.e., $w_{t+1}$) for both the global and partial local shuffling methods is the same. When both global and partial local shuffling use the same number of iteration, learning rate, and initialize the weights with the same random seed, both shuffling methods arrive at the same final weights.

*1) Limitations of the Equivalence:* The above assumptions for equivalence do not consider some factors that could in theory have some impact on training. We list those factors here and while their effects are not fully understood, the empirical results we discuss in Section V indicate that the impact is minimal in most cases, and becomes notable as the ratio of dataset size to number of workers become smaller.

- As pointed out by Yang et al. [17], since batch normalization is typically applied to the local mini-batch of each worker, the mean and the variance for partial local shuffling would differ from the global shuffling case. Normalization methods that are effective at smaller number of samples per worker, e.g. group normalization [21], could potentially be an alternative for effective normalization in partial local shuffling.

- There can be unidentified effects from the implicit bias when treating samples as being all the same with respect to their impact on training, given that there are studies that indicate that not all samples contribute equally to the learning process [22].
- The statistical properties of the dataset at large, not at the individual sample level, can have an impact on training [23]. Partitioning the dataset to different workers changes the similarity in sampling sequences, which can in turn have an impact on training.

### B. Convergence Rate and Shuffling Error

In practice, shuffling aims to produce a random permutation of the samples in the dataset, which is equivalent to without-replacement shuffling, and is usually compared to the baseline i.i.d. [24] [2] sampling. Meng et al. [20] provided the most relevant analysis for distributed SGD. They identified the converge rate bounding terms for insufficient global shuffling, where insufficient means the shuffling is not uniformly distributed. We build on Meng et al. analysis by formulating the partial local shuffling scheme as insufficient global shuffling that is not uniformly distributed, for which the bias of the local shuffling portion (if uniform itself) leads to insufficiency of the global shuffling. According to Meng et al., for insufficient global shuffling in the non-convex case, the convergence rate's upper bound include three terms

$$O\left(\sqrt{\frac{1}{S|N|}}+\frac{log|N|}{|N|}+\frac{|N|\epsilon(A,N)^2}{b|M|}\right) \tag{6}$$

Where $|N|$ is the number of samples in the training dataset $N$, $|M|$ is the number of workers in workers set $M$, $b$ is the local minibatch size (per worker), $S$ is the number of epochs, and $\epsilon(A,N)$ is the shuffling error of algorithm $A$ with the samples $N$. The aim of the partial local scheme is to assure that the shuffling error $\epsilon(A,N)$ will not dominate the bound and accordingly the insufficiency of the partial local scheme would not negatively influence the convergence rate. This requires that the following condition is satisfied: $\epsilon(A,h,N)\leq\sqrt{\frac{b|M|}{|N|}}$, i.e. the shuffling error would not dominate the convergence rate in Equation 6. The shuffling error is defined as follows

$$\epsilon(A,h,N)=\frac{1}{2}\sum_{\pi_i([N])\in\pi([N])}\left|u_{\pi_i}[N]-v_{\pi_i}[N](A,h,N)\right| \tag{7}$$

Where $u_\pi$ is the uniform distribution on the set that contains all permutations of $\pi([N])$, i.e. permutations of all different ways the samples can be picked from the dataset ($|N|!$ permutations), and $v_\pi[N](A,h,N)$ is the distribution after shuffling the dataset of $|N|$ samples using algorithm $A$ with $h$ operators. Here we expand on the analysis of Meng et al. to find the practical values for which the shuffling error would not dominate the converge rate upper bound. Among the permutation of all possible shuffles (i.e., $|N|!$), the number of the permutations $\sigma$ that would include the desired partial local shuffling factor of $Q$ between the $|M|$ partitions is

$$\sigma=\frac{|N|}{|M|}!*P_{\frac{Q|N|}{|M|}}^{\frac{(|M-1|)|N|}{|M|}}*P_{\frac{Q|N|}{|M|}}^{\frac{|N|}{|M|}}*((|M-1|)\frac{|N|}{|M|})! \tag{8}$$

Where we multiply the following four values: the number of permutations of all samples in a partition (i.e., a worker), the number of permutations of candidate samples from outside the local pool of samples, the number of permutations of the possible exchanges, and the number of permutations for the remaining samples in other partitions, after the exchange has been done. The above equation can be rewritten as

$$\sigma=\frac{|N|}{|M|}!*\frac{\frac{(|M-1|)|N|}{|M|}!}{\left(\frac{(|M-1|)|N|}{|M|}-\frac{Q|N|}{|M|}\right)!}*\frac{\frac{|N|}{|M|}!}{\left(\frac{|N|}{|M|}-\frac{Q|N|}{|M|}\right)!}*\left(\frac{(|M-1|)|N|}{|M|}\right)! \tag{9}$$

We divide the summation of the error (in Equation 7) into two portions. The first portion includes the samples that are valid exchanges, and the second portion includes the remaining permutations. The summation can then be rewritten as follows:

$$\left|\frac{1}{|N|!}-\frac{1}{\sigma}\right|*\sigma+\frac{1}{|N|!}*(|N|!-\sigma) \tag{10}$$

Which is rearranged and simplified to be $2-2\frac{\sigma}{|N|!}$, then Equation 7 can be written as

$$\epsilon(A,h,N)=1-\frac{\sigma}{|N|!} \tag{11}$$

We conclude that for practical dataset sizes and number of workers, the shuffling error $\epsilon(A,h,N)$ would approach the value 1, and the shuffling error would hence dominate the convergence rate (Equation 6). For instance, practical settings for training ImageNet ($|N|=1.2\times10^6$) on any number of workers $4\leq|M|\leq100,000$ and $b$ value that gives a total mini-batch size of less than $100K$ yields a shuffling error $\epsilon(A,h,N)\approx1$. This implies that further convergence studies for locality schemes are necessary to improve on the convergence bounds. More specifically, importance sampling schemes [25] can be expanded to investigate the effect of the sampling bias, and consequently shuffling error. Exploring whether importance sampling schemes that reduce the stochastic variance can be effective in countering the sampling bias arising partial exchange is planned for future work.

## V. Evaluation

In this section, we conduct a wide range of experiments to empirically characterize the accuracy as well as performance properties of different shuffling strategies, including the proposed partial local scheme.

### A. Experimental Environment

We run experiments on the following two large scale supercomputing systems.

**Fugaku** [19] is the world fastest supercomputer on the Top500 June 2021 list. It consists of 158,976 compute nodes equipped with Fujitsu A64FX CPU. A64FX provides 48 application CPU cores and 32 GiB of HBM2 memory, the nodes are interconnected through the TofuD network. Each

[2] *Independent and Identically Distributed* random variables

TABLE I: Datasets and Models Used in Experiments (*)Trained on a subset of the original dataset. (**) Use pre-trained model.

| Model | Dataset | #Samples | Size |
|---|---|---|---|
| Resnet50 [27] | ImageNet-1K [7] | 1.2M | $\sim$ 140GB |
| Densenet161 [28] | | | |
| Resnet50 [27] | ImageNet-50(*) [7] | $\sim$65K | $\sim$ 2GB |
| WideResNet-28-10 [29] | CIFAR-100 [30] | 50K | $\sim$160 MB |
| Inceptionv4 [31] | | | |
| Resnet50 (**) [27] | Standford Cars [32] | 8144 | $\sim$ 934 MB |
| Resnet50 [27] | ImageNet-21K(*) [7] | $\sim$ 9.3M | $\sim$1.1 TB |
| DeepCAM [27] | DeepCAM [5] | $\sim$ 122K | $\sim$8.2 TB |

group of 16 compute nodes share a 1.6 TB SSD storage, and all the nodes can access the global 150 PB Lustre filesystem. We utilize SSDs in the so-called *local* mode, where each SSD is divided proportionally among compute nodes in the given group and is exposed as dedicated per-node filesystem. In our experiments, we run four MPI ranks per compute node, where each worker runs 12 OpenMP threads.

**AI Bridging Cloud Infrastructure** (ABCI) [26] is an AI specialized supercomputer that consists of 1,088 compute nodes, each equipped with 2 Intel Xeon Gold 6148 CPUs (total: 40 cores), 384 GiB of DRAM, 4 NVidia V100 GPUs, and InfiniBand EDR NICs. Nodes are equipped with 1.6TB of local storage, and share a 35PB Lustre parallel filesystem. In our experiments, we run 4 MPI ranks per compute node so that each MPI rank has dedicated access to one GPU. We store samples on the local SSDs.

### B. Models and Datasets

To evaluate the impact of various sampling strategies, we used several models and datasets, shown in Table I. The largest data set we use is DeepCAM, which consists of approximately 122K samples and requires 8.2TBs of storage. Another large dataset we use is ImageNet-21K which has over 9M samples and requires 1.1TBs of storage [3]. The dataset is divided into two disjoint parts, where 80% of samples are used for training, and 20% of samples are used for validation.

### C. Training Configuration

For all the models, we follow the original training regime and hyper-parameters suggested by their authors. We do not change the base learning rate and the number of epochs. For examples, we follow the hyper-parameters suggestion in [34] for training on ImageNet dataset, [35] for training on Standford Cars dataset, and [36] for training CIFAR-100 dataset. For large-scale training, e.g., more than 512 and 256 workers for Resnet50 and Densenet, respectively, we apply LARS [37] with the hyper-parameter as suggested in [38]. We train from scratch with random initiated weights.

In our experiments, we compare several shuffling schemes. **Global** shuffling is the default strategy in Pytorch that requires to store the entire dataset, from which workers randomly select samples in each training epoch. With **local** shuffling, workers only store a subset of the dataset to which all their data access is restricted in all epochs. In **partial-*x*** shuffling, workers store a subset of the dataset, and exchange a portion $x$ of it before each epoch. For instance, with **partial-0.01**, workers exchange 1% of their samples at each epoch. We emphasize that in partial local shuffling, a full shuffle of the local portion of the data is performed before the designated ratio is exchanged (i.e., the actual samples exchanged are also randomized).

[3]The original full dataset of ImageNet-21K [7] consists of 14M images, divided into 21,841 classes. In this work, we remove infrequent classes with less than 500 samples, as suggested in [33].

### D. Equivalence of Local and Global Shuffling: When Local is Enough

To much of our surprise, in the overwhelming majority of our experiments local shuffling performs almost identical to global shuffling in terms of validation accuracy attained by the training. Figure 5(a)-(d) summarizes the results for such behavior. Note the exception in the case of 2048 GPUs in Figure 5(a), where the negative effect of smaller subsets of samples at each worker start to appear in local shuffling (hence necessitating a partial shuffle to restore accuracy). The Y axis of the plots indicates validation accuracy, while X axis represents training epoch number. We run the training up to a point where no more improvement is observed in subsequent epochs. All of these experiments were conducted on the ABCI supercomputer, using up to 2,048 GPUs.

When training on ImageNet-1K, both ResNet50 and DenseNet achieve the same validation accuracy with local shuffling and global shuffling, except for ResNet50 on 2,048 GPUs as shown in Figure 5(a) and Figure 5(b), respectively (we elaborate in Section V-E). One key observation on those figures is the fact that while local shuffling starts to converge slower than its global counterpart (in term of number of epochs), local partial shuffling provides almost identical accuracy trajectory with global sampling, which in turn with a feasible learning rate schedule could lead to faster overall convergence and thus a reduction in runtime.

While the CIFAR-100 and StanfordCar data sets are substantially smaller than ImageNet-1K (see Table I), we performed these experiments to explore whether or not similar behavior can be generalized over models with different architectures as well as for different data sets. As shown in Figures 5(c) and 5(d), similar behavior can be observed also with different models and data sets. For such small datasets, the number of samples that each worker trains on becomes small at large scale. For instance, the Stanford Cars dataset contains 8144 samples that are distributed over 64 workers, which means that each worker only trains on 128 samples when shuffling locally. Similarly, the 128 workers that train WideResNet-28 on CIFAR-100 (50K samples) only process 390 samples each.

On larger datasets, such as ImageNet-1K, it is surprising to note that local shuffling achieves the same accuracy as global shuffling, even at large scale (*e.g.* 1,024 workers), when each worker only processes a very small fraction of the dataset (*e.g.* 1/1024). This indicates that workers do not actually need to

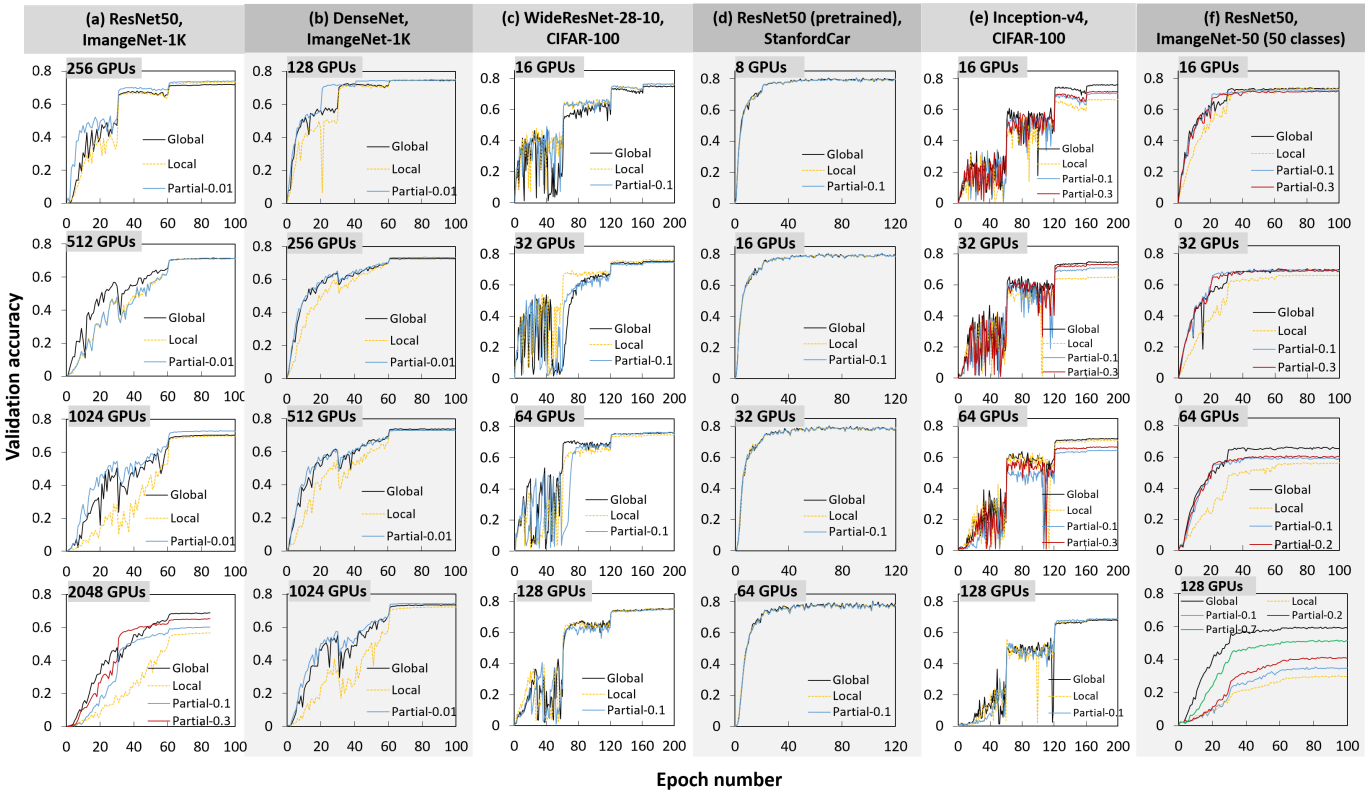
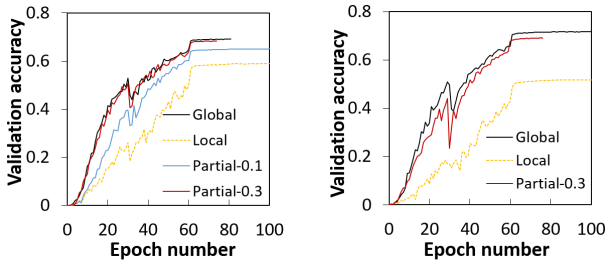



Fig. 5: Top-1 accuracy of training experiments. In **(a)-(d)** local shuffling achieves the same accuracy as global shuffling and in **(e)-(f)** we increase the levels of partial local shuffling until achieving the same accuracy as global shuffling.




(a) 2048 workers (b) 4096 workers

Fig. 6: Top-1 accuracy of ResNet50 with ImagetNet-1K on Fugaku when strong scaling (global batch size = 65,536).

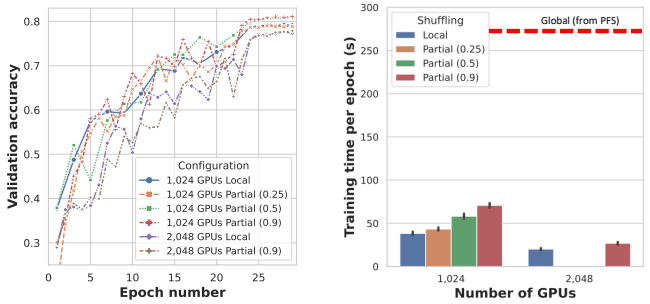
process a large portion of the whole dataset, and exchanging the gradient weights is enough to ensure convergence.

### *E. Partial Local Shuffling*

We explained earlier in Section IV that despite equivalence in gradients, some factors, specially the batch norm, could have a negative impact on the accuracy of local shuffling at specific training configurations. Our proposed partial local shuffling targets those training configurations, by allowing the user to control the volume by which samples are exchanged.

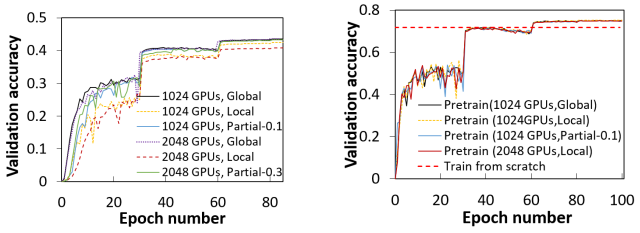
We start to observe a gap of 9% between global and local shuffling when we scale up to 2,048 GPUs training on ResNet50 with ImageNet-1K (Figure 5(a)). Similarly for the ResNet50 with ImageNet-50 and Inception-v4 with CIFAR-100 as shown in Figure 5(e)-(f). For instance, as we scale up to 128 GPUs, we start to observe a gap between the global and local shuffling that can reach up to 30% drop in accuracy for ImageNet-50 (and 10% even at the modest scale of 32 GPUs). For ImageNet-50 with 128 workers, a rather high level of exchange rate of 70% is required to start converging to an accuracy that is significantly closer to global shuffling than that of the local one. Note however that this relatively high level of exchange remains to be far less damaging, in comparison to transferring the files from the parallel filesystem in each epoch (in absence of sufficient local storage to replicate the entire datasets). In the case of ImageNet-1K and CIFAR-100, an exchange rate of 0.3 is sufficient to converge to global shuffling accuracy.

These results shows that some DNN models are more sensitive to samples diversity than others. For instance, local shuffling degrades the accuracy of Inception-v4 on CIFAR-100, while WideResNet-28 achieves similar accuracy as with global shuffling for the same dataset. This experiment also shows that partial local shuffling reduces the local storage requirements. While global shuffling requires that each worker stores the whole dataset, partial local shuffling with a 30% exchange rate, the 128 workers only store $1.3 \times \frac{1}{128} \approx 1\%$ of the dataset.




(a) Validation accuracy (b) Performance

Fig. 7: DeepCAM validation accuracy and performance.




(a) Upstream training. (b) Downstream training (256 GPUs).

Fig. 8: Top-1 accuracy upstream training ResNet50 model with ImagetNet-21K and downstream training with ImageNet-1K.

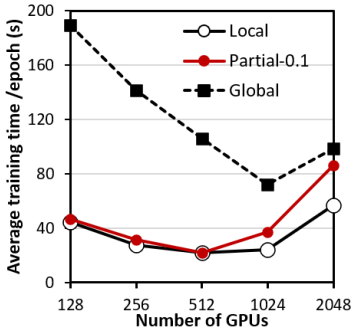



Fig. 9: Performance of ResNet50 with ImageNet-1K dataset.

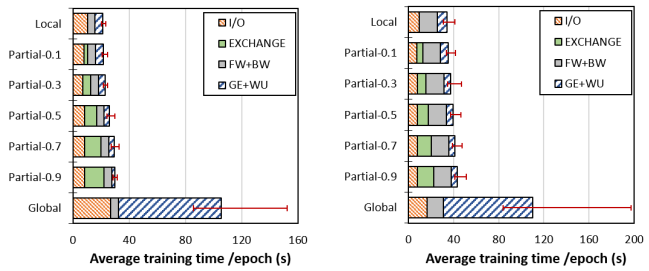
The results imply that on large scale supercomputers with little local storage, like Fugaku, partial local shuffling can significantly reduce the storage requirement without affecting the accuracy. For evidence, we run a strong scaling experiment on ResNet50 with ImageNet-1K on Fugaku, where we decrease the local batch size as the number of workers increases. Figure 6 reports the results for 2,048 and 4,096 workers. The results show that the local shuffling accuracy decreases as the number of workers scales. This is because, for 4,096 workers, each worker only trains on approximately 292 samples. Using **partial-0.1** local shuffling slightly increases the accuracy for 2,048, and **partial-0.3** local shuffling achieves the same accuracy as global shuffling for up to 4,096 workers. In this case, each of the 4,096 workers only stores up to $1.3 \times \frac{1}{4096} \approx 0.03\%$ of the dataset.

It is worth noting that the partial local shuffling is still useful in supercomputers that have a large local storage like ABCI, especially when the dataset size is larger than the local storage. Figure 7(a) shows validation accuracy on the DeepCAM dataset comparing local shuffling with local partial (with exchange ratios of 0.25, 0.5 and 0.9) using 1,024 and 2,048 GPUs on ABCI. Because the DeepCAM dataset does not fit into local storage and training from the PFS would be prohibitive, we are unable to present accuracy numbers for global shuffling. As seen, however, partial local shuffling improves accuracy by approximately 2% when using exchange ratio 0.5 or higher on 1,024 GPUs and about 1% on 2,048 GPUs. For the case of 2,048 GPUs, we only have results on exchange ratio 0.9. We speculate that the reason for smaller improvement at larger scale is due to the relatively small number of input samples in DeepCAM (about 122K), where the entire data set gets processed in a few iterations due to the larger minibatch size.

Reusing the pre-trained deep learning models has recently led to a significant improvements on many tasks. Figure 8 illustrates the impact of shuffling in pretraining ResNet50 mode with the ImageNet-21K dataset. We observe that although the accuracy of upstream training using local shuffling degrades around 3% (in the case of 2,048 GPUs) when compared with global shuffling as shown in Figure 8(a), the difference of the final accuracy of downstream training with ImageNet-1K dataset between them is trivial. This result implies that using the (partial) local shuffling for upstream training could potentially help to reduce the training time while still maintaining the final downstream accuracy.

### *F. Performance*

We evaluate the performance of different shuffling schemes by measuring the average training time per epoch when running ResNet50 with ImageNet-1K on ABCI. Figure 9 reports the training time of different shuffling strategies as the number workers grows. The performance results show that the global shuffling strategy significantly degrades the training time compared to local shuffling. For instance, global shuffling on 128 workers is almost 5x slower than local shuffling. This slowdown is due to the difference of performance of the I/O subsystem: when using local shuffling, workers read samples from their local storage, whereas workers concurrently read samples from the parallel file system with global shuffling. The performance of **partial-0.1** is similar to the performance of local shuffling for up to 512 workers. However, the performance of **partial-0.1** significantly degrades when scale up to 1,024 and 2,048 GPUs. Since the number of iterations per epoch becomes less in such cases, i.e., 40 and 20 iterations, respectively, it makes our design of overlapping the sample exchange with the computation of the forward and backward pass during training becomes less effective, i.e., less total time for overlapping. Further more, exchanging the samples randomly between workers leads to a personalized all-to-all communication pattern which is sensitive to the network




(a) ImageNet-1K + Resnet50. (b) ImageNet-1K + Densenet161.

Fig. 10: Breakdown of performance impact as the function of exchange rate (512 GPUs).

congestion when scaling up. An alternative solution is to use a hierarchical global exchange scheme that maps to the hierarchy of connection between computing nodes.

Similarly, we present performance of **partial** shuffling with DeepCAM dataset in Figure 7(b). While on DeepCAM our exchange scheme incurs noticeable overhead, we emphasize that compared to a PFS based global scheme (shown by the red horizontal line in Figure 7(b)), we still perform multiple times better. The value for global shuffling is a lower bound estimate based on the theoretical peak bandwidth of the PFS on ABCI, and the overall size of the DeepCAM dataset.

In order to better understand the performance difference between local, partial, and global shuffling, we analyze the performance of training Resnet50 and Densenet161 on ImageNet-1K over 512 workers on ABCI. Figure 10 reports a breakdown of the training time for 512 workers as the exchange rate of the partial strategy grows. We report the average time spent performing I/O, exchanging samples between workers for partial shuffling (EXCHANGE), performing the backward and forward propagation (FW+BW), and exchanging gradient and updating weights (GE+WU). The reported performance is the average measured time of five epochs. The error bars represent the variation of the total duration of an epoch.

The figure shows that the performance of **partial** shuffling slightly decreases as the exchange rate grows. For low exchange rates, the performance of **partial** is similar to that of local shuffling. When the exchange rate increases, the overhead of sample exchange becomes noticeable because of the increased volume of communication. Thus, the performance is degraded by up to $1.37\times$. However, this slight performance degradation is compensated by the increased accuracy obtained by partial shuffling, as shown in Section V-E.

The breakdown shows that the time of performing forward and backward propagation phases remain constant for all the cases. Similarly, the cost of gradient exchange, weights update and I/O are slightly different between local and partial shuffling. However, since global shuffling reads from the PFS, the cost of I/O is much higher than those of local and partial shuffling. For example, Densenet with global shuffling spends $19.6s$ on average performing I/O, compared to $8s$ when using local shuffling. When measuring the I/O time of all workers, we observe a significant variation when using global shuffling. For example on Densenet, the fastest worker reads samples in $11.9s$ while the slowest worker reads samples in $142s$. This high variation of I/O performance could indicate that the PFS suffers congestion caused by the 512 workers performing IO concurrently [39]. Moreover, workers wait for each other using collective communication during the gradient exchange. Because some of the workers enter the collective lately (due to poor I/O performance), all the workers are delayed, and the average time spent performing the gradient exchange reaches $70s$.

## VI. Related Work

### A. Shuffling to Alleviate the I/O Bottleneck for DNNs

DeepIO [16] reduces the need to access the storage by performing in-situ shuffling: instead of shuffling the dataset, the compute node exchange samples through RDMA, and some of the samples are kept local in order to reduce the need for communication, while exchanging enough data to ensure the accuracy of the model. A similar approach is used [17]: the data loader caches data in memory on the compute nodes to reduce the number of I/O requests to the parallel file system. Moreover, the data loading arranges the distribution of sample to increase their locality.

It is important to note that despite similarities to our method, in both of the above approaches the local sampler introduces uncontrolled bias since the ratio of global to local shuffle portion is unidentified (i.e. the split is itself random). Since the exchange is uncontrolled, arbitrary communication bottlenecks can occur and there is no means to overlap and interleave the exchange with the gradient averaging Allreduce in a orchestrated manner. Finally, both studies report results for up to 64 nodes only; negative effects not observable at that scale.

### B. Theoretical Studies on the Effect of Shuffling in Distributed SGD

Several efforts over the years improved on the upper bound of the convergence rate for random and global shuffling in the general sense, i.e., shuffling the entire dataset at the beginning of each epoch [40], [24], [41], [42]. For distributed training, most notably is Elmahdy et al. [43] study that proposed a linear coded shuffling algorithm and provided a lower bound that guarantees the optimally over other shuffling methods. Meng et al. [20] did an extensive analysis on the convergence properties of distributed SGD with insufficient random shuffling, including the non-convex cases. The analysis however did not directly cover the case of partial local shuffling, albeit pointing to the shuffling error impact on the convergence rate (under the condition that the component functions are convex).

### C. Machine Learning I/O Bottlenecks at the System Software-level

The I/O has been identified as one of the main bottlenecks that limit the scalability of machine learning applications [44]. Studies analyzed the I/O performed by various deep learning

applications running over BeeGFS [10], or over GPFS [11]. The collected I/O patterns consist of many small (approx 100KB) read operations, and metadata operations account for one third of the I/O overhead. However, as the number of clients grows, the metadata overhead becomes the main bottleneck. Koziol analyzes the I/O requirements of a TensorFlow climate simulation running on a pre-exascale supercomputer [45]. The bandwidth required for reading the entire dataset is an order of magnitude higher than the GPFS bandwidth.

In order to reduce the cost of I/O in machine learning applications, several approaches have been investigated. PHDFS groups small files in order to improve read throughput [15]. To reduce I/O contention on the parallel filesystem, LMDBIO uses a hierarchical scheme that reduces the number of processes that perform I/O operations [13]. It also prefetches the data that is likely to be needed in the future.

Multiple works reduce the number of I/O requests to the storage servers by exchanging samples between the compute nodes [5], [12], [14], [46], [18]. FanStore implements a metadata cache in the compute nodes to avoid accessing the storage nodes [12]. Data files are read from the local filesystem, or from other compute node via MPI messaging. Similarly, LBANN stores data samples in an in-memory distributed data store[46]. Quiver introduces an I/O cache for DL workloads in cloud environments that employs a hash-based addressing to transparently reuse cached data across multiple jobs and even multiple users operating on the same dataset [18]. DIESEL+ is an end-to-end solution that accelerates that I/O pipeline of image processing deep learning training tasks [47]. It includes pieces such as local metadata snapshots, per-task distributed caching, chunk-wise shuffling (i.e., by grouping small files into bigger chunks) and GPU-assisted image decoding. Another recent proposal introduces a new data format that uses compression to reduce the overhead of fetching and transporting data [48]. While these methods address the same overall I/O problem, their proposals are orthogonal to our main focus, which is the exploitation of shuffling itself.

## VII. Conclusion

We have proposed a dataset partitioning, shuffling, and redistribution solution for large scale distributed DNN training (synchronous SGD). Our solution requires minor modifications to PyTorch training scripts and no modifications to the PyTorch framework itself. We demonstrated on various data sets and DNN models that in many scenarios local shuffling attains similar validation accuracy as global shuffling, and when local shuffling falls behind, redistributing even a small portion of the local data set can contribute significant improvements to accuracy and thus can approach that of global sampling.

One of the key takeaways from our experiments is that while in the state-of-the-art practice the need for global shuffling is widely assumed, there is a clear indication that such assumption should be questioned. Eliminating unnecessary shuffling of data samples in distributed SGD can have profound implications on the I/O requirements of the overall training procedure. First, there is no need to replicate data everywhere, which reduces the cost of data staging in HPC environments. Second, smaller local data storage suffices that could enable training comparable neural network models in more modest storage environments, such as over local `tmpfs`, opening up the potential for less powerful HPC systems to be utilized for deep learning workloads.

## Acknowledgements

This work has been supported by JST ACT-X Grant Number JPM-JAX190C, JSET CREST Grant Number JPMJCR19F5, JST PRESTO Grant Number JPMJPR20MA and JSPS KAKENHI Grant Numbers JP21K17751 and JP19K11993.

## References

[1] TOP500, "The List," https://www.top500.org/, 2021, [9 Sept. 2021].
[2] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "Youtube-8m: A large-scale video classification benchmark," 2016.
[3] C. Sun, A. Shrivastava, S. Singh, and A. Gupta, "Revisiting unreasonable effectiveness of data in deep learning era," 2017.
[4] G. Research, "Google Open images dataset," https://github.com/cvdfoundation/open-images-dataset, 2018.
[5] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica *et al.*, "Exascale deep learning for climate analytics," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 649–660.
[6] G. Research, "C4 dataset: a Colossal, Cleaned version of Common Crawl's web crawl corpus," https://github.com/google-research/text-to-text-transfer-transformer\#datasets, retrieved 25 February 2021.
[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Conf. on comp. vision and pattern recogn.* IEEE, 2009, pp. 248–255.
[8] O. C. Project, "Open Catalyst 2020 (OC20) Dataset," https://github.com/Open-Catalyst-Project/ocp/, retrieved 20 September 2021.
[9] M. F. Kragh, P. Christiansen, M. S. Laursen, M. Larsen, K. A. Steen, O. Green, H. Karstoft, and R. N. Jørgensen, "Fieldsafe: Dataset for obstacle detection in agriculture," *Sensors*, vol. 17, no. 11, 2017.
[10] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/O characterization and performance evaluation of BeeGFS for deep learning," in *Proc. of 48th ICPP*, 2019, pp. 1–10.
[11] L. Oden, C. Schiffer, H. Spitzer, T. Dickscheid, and D. Pleiter, "IO challenges for human brain atlasing using deep learning methods-an in-depth analysis," in *PDP*, 2019, pp. 291–298.
[12] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "FanStore: Enabling efficient and scalable I/O for distributed deep learning," *arXiv preprint arXiv:1809.10799*, 2018.
[13] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable deep learning via I/O analysis and optimization," *ACM TOPC*, vol. 6, no. 2, pp. 1–34, 2019.
[14] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient user-level storage disaggregation for deep learning," in *CLUSTER*. IEEE, 2019, pp. 1–12.
[15] Z. Zhu, L. Tan, Y. Li, and C. Ji, "PHDFS: Optimizing I/O performance of HDFS in deep learning cloud computing platform," *Jour.of Sys.Arch.*, vol. 109, 2020.
[16] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-aware I/O pipelining for large-scale deep learning on HPC systems," in *MASCOTS*, 2018, pp. 145–156.
[17] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," in *HiPC*, 2019, pp. 235–245.
[18] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *FAST*, Santa Clara, CA, 2020, pp. 283–296.
[19] RIKEN Center for Computational Science, "Fugaku Supercomputer," https://www.r-ccs.riken.jp/en/fugaku/, 2021, [1 April 2021].
[20] Q. Meng, W. Chen, Y. Wang, Z. Ma, and T. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," *Neurocomputing*, vol. 337, pp. 46–57, 2019.
[21] Y. Wu and K. He, "Group normalization," in *ECCV*, 2018, pp. 3–19.

[22] A. Katharopoulos and F. Fleuret, "Not all samples are created equal: Deep learning with importance sampling," in *ICML*, 2018, pp. 2530–2539.

[23] D. Cheng, S. Li, H. Zhang, F. Xia, and Y. Zhang, "Why dataset properties bound the scalability of parallel machine learning training algorithms," *TPDS*, vol. 32, no. 7, pp. 1702–1712, 2021.

[24] K. Ahn, C. Yun, and S. Sra, "SGD with shuffling: optimal rates without component convexity and large epoch requirements," in *NeurIPS*, 2020.

[25] P. Zhao and T. Zhang, "Stochastic optimization with importance sampling for regularized loss minimization," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, 2015, p. 1–9.

[26] National Institute of Advanced Industrial Science and Technology, "ABCI Supercomputer," https://abci.ai, 2021, [1 April 2021].

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[28] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018.

[29] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *BMVC*, September 2016, pp. 87.1–87.12.

[30] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Master's thesis, Dep. of Comp. Sci. Univ. of Toronto*, 2009.

[31] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning," 2016.

[32] J. Krause, M. Stark, J. Deng, and L. Fei-Fei, "3d object representations for fine-grained categorization," in *4th International Workshop on 3dRR*, Sydney, Australia, 2013.

[33] T. Ridnik, E. Ben-Baruch, A. Noy, and L. Zelnik-Manor, "Imagenet-21k pretraining for the masses," *arXiv preprint arXiv:2104.10972*, 2021.

[34] P. Goyal *et al.*, "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour," *CoRR*, vol. abs/1706.02677, 2017.

[35] S. Kornblith, J. Shlens, and Q. V. Le, "Do better imagenet models transfer better?" in *CVPR*, 2019, pp. 2661–2671.

[36] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[37] Y. You, I. Gitman, and B. Ginsburg, "Large batch training of convolutional networks," 2017.

[38] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama *et al.*, "Massively distributed sgd: Imagenet/resnet-50 training in a flash," *arXiv preprint arXiv:1811.05233*, 2018.

[39] M. S. Mosli Bouksiaa, F. Trahay, A. Lescouet, G. Voron, R. Dulong, A. Guermouche, E. Brunet, and G. Thomas, "Using differential execution analysis to identify thread interference," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2866–2878, Dec. 2019.

[40] L. M. Nguyen, Q. Tran-Dinh, D. T. Phan, P. H. Nguyen, and M. van Dijk, "A unified convergence analysis for shuffling-type gradient methods," *CoRR*, 2020. [Online]. Available: https://arxiv.org/abs/2002.08246

[41] O. Shamir, "Without-replacement sampling for stochastic gradient methods," ser. NIPS'16. Curran Associates Inc., 2016, p. 46–54.

[42] J. Z. HaoChen and S. Sra, "Random shuffling beats SGD after finite epochs," in *ICML*, 2019, pp. 2624–2633.

[43] A. Elmahdy and S. Mohajer, "On the fundamental limits of coded data shuffling for distributed machine learning," *IEEE Transactions on Information Theory*, vol. 66, no. 5, pp. 3098–3131, 2020.

[44] R. Böhringer, N. Dryden, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," 2021.

[45] Q. Koziol, "I/O for Deep Learning at Scale," International Conference on Massive Storage Systems and Technology (MSST), 2019.

[46] S. A. Jacobs, B. Van Essen, D. Hysom, J.-S. Yeom, T. Moon, R. Anirudh, J. J. Thiagaranjan, S. Liu, P.-T. Bremer, J. Gaffney *et al.*, "Parallelizing training of deep generative models on massive scientific datasets," in *CLUSTER*. IEEE, 2019, pp. 1–10.

[47] L. Wang, Q. Luo, and S. Yan, "DIESEL+: Accelerating Distributed Deep Learning Tasks on Image Datasets," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 5, pp. 1173–1184, 2022.

[48] M. Kuchnik, G. Amvrosiadis, and V. Smith, "Progressive Compressed Records: Taking a Byte out of Deep Learning Data," *Proc. VLDB Endow.*, vol. 14, no. 11, p. 2627–2641, jul 2021. [Online]. Available: https://doi.org/10.14778/3476249.3476308