DDStore: Distributed Data Store for Scalable Training of Graph Neural Networks on Large Atomistic Modeling Datasets

Jong Youl Choi, Massimiliano Lupo Pasini,
Pei Zhang, Kshitij Mehta, Frank Liu
Oak Ridge National Laboratory
Knoxville, TN, USA

Jonghyun Bae, Khaled Z. Ibrahim
Lawrence Berkeley National Laboratory
Berkeley, CA, USA

Abstract

Graph neural networks (GNNs) are a class of Deep Learning models used in designing atomistic materials for effective screening of large chemical spaces. To ensure robust prediction, GNN models must be trained on large volumes of atomistic data on leadership class supercomputers. Even with the advent of modern architectures that consist of multiple storage layers that include node-local NVMe devices in addition to device memory for caching large datasets, extreme-scale model training faces I/O challenges at scale.

We present DDStore, an in-memory distributed data store designed for GNN training on large-scale graph data. DDStore provides a hierarchical, distributed, data caching technique that combines data chunking, replication, low-latency random access, and high throughput communication. DDStore achieves near-linear scaling for training a GNN model using up to 1000 GPUs on the Summit and Perlmutter supercomputers, and reaches up to a 6.15x reduction in GNN training time compared to state-of-the-art methodologies.

Keywords

Distributed Data Parallelism, Deep Learning, Graph Neural Networks, Atomistic Modeling, Inorganic Chemistry, Organic Chemistry, Quantum Chemistry

ACM Reference Format:
Jong Youl Choi, Massimiliano Lupo Pasini,, Pei Zhang, Kshitij Mehta, Frank Liu and Jonghyun Bae, Khaled Z. Ibrahim. 2023. DDStore: Distributed Data Store for Scalable Training of Graph Neural Networks on Large Atomistic Modeling Datasets. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624171

1 Introduction

Desiging organic and inorganic compounds with desired functional properties is crucial to several scientific applications supported by the US department of Energy (DoE). Such design process typically requires studying the behavior of the compounds at atomic scales, which is necessary to assess the chemical stability of an atomic structure of a compound, as well as to predict its electronic, mechanical, thermal, and optical properties.

State-of-the-art atomistic modeling approaches require performing computationally expensive first-principle calculations to accurately estimate the properties of compounds [10, 25, 34, 45, 48, 61]. However, the computational cost of these methods does not scale well with respect to the numbers of atoms, and thereby precludes an effective screening of large chemical regions of practical interest in realistic scenarios, where the number of atoms in the structure is in the order of thousands.

Deep learning (DL) has shown the potential to produce fast and yet sufficiently accurate predictions of properties for organic and inorganic compounds at a fraction of the time required by first-principle calculations [5–7, 57, 62, 65], thereby enabling an effective screening of large chemical regions. Among the different classes of DL models for atomistic modeling, graph neural networks (GNNs) [59] are especially promising for their expressiveness, which is enabled by the natural mapping of atomic structure onto a graph, with atoms interpreted as nodes and interatomic bonds as edges, and leveraging the graph structure to create meaningful embedding features [13, 14, 19, 21, 24, 40, 50, 51, 56, 58, 60]. In contrast to social graph problems that involve one large graph with millions of nodes (e.g., C-SAW [49]), atomistic modeling usually deal with a large number (in the order of millions) of relatively small graph samples, each containing at most thousands of nodes [12]. Therefore, large scale atomistic modeling datasets are characterized by an inherent fine degree of granularity. In recent years, the amount of large-scale atomistic information has expanded to several terabytes and continues to increase.

As we need to explore a vast parameter space primarily dictated by chemical composition, the number of atoms in the system, and different atomic arrangements, the amount of data used to train the GNN surrogate model must be sufficiently large to ensure that the surrogate model maintains generalizability and robustness throughout the entire design process. HPC systems have become increasingly important for carrying out such data-intensive tasks. To process large volumes of data, DL training must be free of I/O bottlenecks and scale effectively on state-of-the-art HPC facilities. While DL workloads have witnessed significant hardware acceleration over the past decade due to the evolution of new computational units such as multi-/many-core central processing units (CPUs), graphic processing units (GPUs), and tensor processing units (TPUs) [32], I/O components remain the slowest components

*This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

SC-W 2023, November 12–17, 2023, Denver, CO, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624171

of a system, thereby causing I/O bottlenecks in data-intensive workloads such as DL applications at scale.

In recent years, supercomputing architectures have evolved to include tiered memory hierarchy that includes node-local SSD-based non-volatile memory (NVMe) devices in addition to device RAM. In principle, the fine-grained nature of large-scale atomistic modeling data accommodates an effective use of NVMe devices to locally store a large chunk of data (many graph samples) on each compute node of the supercomputing cluster. However, some state-of-the-art leadership class supercomputing facilities are not endowed with sufficiently large local memory devices yet. Therefore, there is a compelling need to develop distributed technologies for I/O of large volumes of fine-grained datasets that can efficiently use hardware and memory characteristics of different supercomputing systems.

In this work, we leverage the distinctive I/O patterns observed in GNN training and develop an in-memory distributed data store optimized for DL training to accommodate effective and scalable reading of large volumes of graph data for HPC and supercomputing architectures. The new in-memory distributed data store (DDStore) technology can be effectively deployed on a broad class of HPC and supercomputing facilities without necessarily requiring massive inter-node memory storage capabilities.

The contributions of this paper are as follows:

- we identify challenges arising from the training of GNN models over large-scale atomistic modeling datasets with millions of atomic structures,
- we present the design and architecture of a high performance, portable, in-memory data store, specifically tailored for efficient random data retrieval during the data loading processes involved in the distributed training of GNNs, and
- we demonstrate performance improvements of using our in-memory store for large-scale training of a GNN model for over 10 million molecules, and highlight various parameters that can influence the performance of the system.

The rest of this paper is organized as follows: Section 2 discusses HydraGNN (a scalable implementation of multi-headed GNN), distributed DL training, and associated I/O challenges. Section 3 introduces our proposed solution, DDStore. Section 4 provides the numerical results of DDStore compared to conventional data management methodologies. Finally, we conclude with an analysis of this study and outline potential avenues for future research.

2 Background

In this section, we introduce HydraGNN, an open-source scalable GNN implementation, and discuss the I/O challenges for distributed data parallel (DDP) training in HPC environments.

2.1 HydraGNN

HydraGNN [43, 53] is an open-source scalable implementation of multi-headed GNN, which can leverage HPC resources, achieving linear scaling performance in distributed training for large volumes of data [12]. HydraGNN has been used to predict energetic and functional properties from atomic structures for various physical systems including ferromagnetic alloys [53] and organic molecules [9]. While numerous open-source GNNs often implement a single specific message-passing technique, HydraGNN adopts an object-oriented approach, enabling a variety of message passing policies

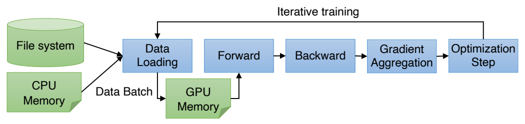

Figure 1: Iterative training with distributed data parallel (DDP).

tailored for feature engineering and multi-head feature training. Furthermore, HydraGNN places a strong focus on scalability within HPC environments. It has the capability to efficiently utilize thousands of GPU devices in supercomputers, harnessing DDP to enable large-scale training.

2.2 Distributed Data Parallel and Data Loading

DDP is one of the parallel methods commonly used in AI/DL to train models using multiple processors or machines. Each process handles only a subset of the data (or graphs) and executes functions in parallel with others. When information needs to be consolidated, such as for aggregated gradient updates, processes communicate data using *all-gather* operations. Message Passing Interface (MPI) [20, 23] and NVIDIA Collective Communications Library (NCCL) [38] are well-known communication backends in multi-node, multi-GPU HPC environments.

DDP provides several benefits, such as efficient use of computing resources by providing fault tolerance, and enabling training on large datasets through concurrent processing of batch data. DDP is widely implemented in many DL frameworks such as PyTorch [55] and TensorFlow [2] and is commonly used for applications such as image recognition, natural language processing, and speech recognition. HydraGNN extends the PyTorch's DDP implementation with enhancements targeting HPC environments.

Generally, DDP for model training involves the following five steps (Fig. 1): i) The dataset is divided into smaller, non-overlapping subsets called "batches". A single batch consisting of N samples is loaded into each process (data loading). ii) Each process generates predictions for the samples in the batch using local model (forward). iii) Each process then assesses the loss between these predictions and the actual values and subsequently computes gradients for the model parameters based on the loss (backward). iv) Gradients from all processes are aggregated, producing a consolidated gradient (gradient aggregation). v) Each process updates its local model parameters using the aggregated gradients (optimization).

Various methods have been developed to reduce overhead in executing these training steps and ensure seamless integration for optimal performance, such as data loading optimizations with latency hiding on HPC systems. An example of these efforts is the use of multi-threaded data loading and overlapping I/O and computation implemented in PyTorch's parallel data loading module [55]. NVIDIA's Data Loading Library (DALI) has been specifically developed to offload the data loading from the CPU to the GPU, effectively minimizing overhead and enhancing training efficiency [68].

In DDP, data shuffling is another primary factor for optimal performance. Data sharding with local shuffling [4] is one of the common techniques used with DDP, and consists of splitting the datasaset into chunks that are individually processed by each GPU. Once loaded, shuffling is done within the local chunk. However, it carries two serious performance implications when used with large-scale DL models. First, it is important that the training data stored in partitions on different nodes needs to be shuffled across

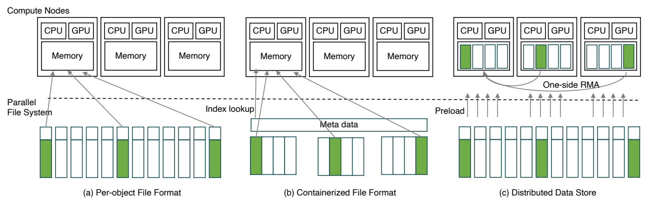

Figure 2: Data storing strategy and its common I/O access patterns in deep learning (a) Per-object file format (b) Containerized file format (c) Distributed data store.

successive epochs of the DL training to maintain model generality and avoid overfitting [47]. Secondly, in situations where the number of GPUs changes after a training session, like during hyperparameter optimization, the shared dataset must be restructured to align with the updated GPU count, which is time consuming. On the other hand, transferring data between distributed nodes for global shuffling emerges as the optimal solution to enhance the model's generalizability. However, exchanging vast amounts of data during each epoch poses scalability challenges for DL applications. DL training becomes I/O and communication bound and thus incurs significant overhead due to data movement, which serves as the primary motivation for our paper. A comprehensive overview of these approaches is provided in the related work section.

2.3 Data Management in Deep Learning

In this section, we examine the data formats commonly employed in DL applications. Fig. 2 shows the data access patterns used by two commonly used file format types: per-object file formats (PFF) and containerized file formats (CFF). Additionally, it also shows the data access pattern employed by DDStore, our solution towards mitigating the overhead of large data movement. The design and architecture of DDStore are presented in the next section.

Per-object file format (PFF) The per-object file format serializes the graph object structures and stores one sample per file. This is one of the simpler approaches for storing graph data, but exerts significant overhead on the underlying parallel file system as the number of samples becomes large. Additionally, as training is performed using tens of thousands of processes, concurrent access to the large number of files from all processes causes a severe I/O bottleneck.

Containerized file format (CFF) Fig. 2 (b) shows an alternative method of storing data using containerized file format libraries such as HDF5 [35], ADIOS [22], WebDataset [1] and TFRecord [2]. In this approach, multiple samples are stored in a single file which reduces the overhead on the file system. The data management library manages metadata on behalf of the user and provides the API to store and retrieve particular data samples. However, DL training requires reading samples in a random or shuffled order to enable unbiased training. Frequent, random, non-sequential I/O accesses to CFF data can lead to a large number of accesses to the file system, which is very inefficient. Additionally, concurrent I/O access from multiple processes trying to read samples from the same containerized file leads to congestion and high I/O times.

In-memory data caching Irrespective of the file format used for storing training data, some solutions involve reading the entire dataset into device memory or node-local storage systems such as NVMe devices. However, these options may not be feasible for large datasets, which are the primary concern of this paper. The size of these datasets can surpass the capacity of a single node's memory or its node-local storage. Moreover, several HPC resources (including some of the existing US-DoE supercomputers) are not endowed with NVMe devices yet.

For those HPC architectures that cannot rely on node-local storage, the only state-of-the-art methodologies that enable to scale the DL training on large volumes of data are PFF and CFF techniques. Both methodologies heavily rely on a frequent data movement between file system and volatile node memory, which results in inefficient data reading and causes severe I/O bottlenecks.

3 DDStore

To address challenges associated with efficient reading of data for large-scale DL/GNN, we have designed DDStore, specifically addresses random, read-oriented, global shuffle operations.

3.1 Design

Our design of DDStore is driven by two main performance considerations: *1) Can we minimize access to the file system during the shuffling steps and make in-memory data accessible to other nodes? 2) How do we design fast, efficient, and portable communication mechanisms to provide high-performance shuffling operations for DL?*

These considerations stem from the fact that parallel file systems are shared resources on HPC clusters and are the slowest component of the system. Applications that are I/O bound due to poor performance of the file system typically experience severe challenges with scalability. At large scale, the communication overhead can also become high due to contention in the network, which is also a shared resource on HPC systems.

To address the first objective, DDStore splits data into chunks and stores them in the device memory of compute nodes similarly to data sharding. The dataset is read from the file system and distributed across the device memories of compute nodes. All subsequent accesses to samples in the dataset are made via in-memory read transactions. Secondly, to reduce communication overhead over the network during read operations, DDStore uses data replication to maintain multiple copies of the dataset in memory. This is the novel component of DDStore with respect to all existing state-of-the-art scalable data management methodologies, which allows DDStore to internally partition application processes into groups that are assigned a replica. This hierarchical design prevents situations in which all processes access a single process's memory to obtain the next batch of data which can lead to communication bottlenecks. Finally, DDStore uses low-latency communication functions such as the MPI Remote Memory Access (RMA) [16], known as one-sided communication, to provide fast, non-blocking read operations that are portable across different systems and architectures (Fig. 2c).

We formally define DDStore as

$$DS = (c, w, f)$$

where 'c' is the number of chunks that a dataset is striped into, 'w' represents the store *width* that controls the degree of replication of data, and 'f' represents the communication framework used for transferring data between processes.

Chunking To effectively utilize the memory available on compute nodes of HPC systems, similarly to data sharding, DDStore uses chunking to split a dataset and distribute the chunks evenly amongst nodes. This avoids performing expensive accesses to the file system to retrieve data during every shuffle operation. The number of chunks 'c' depends on the total size of the dataset 'T' and the width 'w' of DDStore.

$$c = T/w$$

The description of 'w' below will clarify how the chunk size is calculated. By default, for a training using N processes, DDStore stripes the data into N chunks and places one chunk on each process.

Replication The width 'w' controls the degree of replication of data chunks, which makes DDstore distinguishable from all the other state-of-the-art scalable data management methodologies and reduces the communication bottlenecks due to data shuffling across consecutive epochs of DL training. DDStore divides application processes internally into sub-groups where each sub-group holds a full replica of the dataset. The width represents the cardinality of these groups. To demonstrate how the width influences the replication strategy, let $N = 1024$ and $w = 128$. DDStore creates 8 groups (1024/128) of 128 processes each. Every process group holds a full replica of the dataset. Processes within a group communicate only with each other to exchange data. The number of replicas r is same as the number of process groups, and is represented by

$$r = N/w$$

In our example, there are 8 replicas of the dataset. The processes within a group each hold $T/128$ chunks of the data. By default, $w = N$, which creates a single replica of the dataset striped evenly over all processes. The degree of replication is inversely proportional to the width of the store; as we increase the width, we reduce the number of replicas maintained in the system. As a result, larger width values consume less memory as compared to smaller width values. On the other hand, smaller values for width can help reduce communication bottlenecks as increasing the number of process groups can help distribute communication requests more evenly. The width is configurable so that a user can tune.

Communication The communication framework 'f' provides the data plane and control mechanism for fetching data from the memory of other nodes. For the data plane, we considered several state-of-the-art options such as 1) ZeroMQ [26], an asynchronous message passing library for distributed applications, and 2) MPI's blocking, non-blocking, and one-sided communication functions. For the control plane, design options included 1) developing a message-broker framework in which additional message brokers on each node facilitate data exchange amongst themselves during the shuffle operation, and 2) fully de-coupled and asynchronous communication without dedicated message brokers. To provide a scalable, high-performant, and portable communication framework over our use of MPI for parallelization, we selected MPI's RMA one-sided library functions for DDStore. The communication layer in the DDStore Architecture discusses our use of the MPI RMA routines.

3.2 Architecture

We describe the main components of DDStore in details that allow to efficiently perform frequent random shuffling of data distributed across nodes to reduce communication bottlenecks.

DDStore is composed of four components: 1) a data preloader, which reads data in various formats from a parallel file system and loads it into the memory of deep learning applications, 2) a data registry that manages the index of data chunks, 3) the data loader that reads the next batch of data from other processes, and 4) the communication layer that leverages one-sided RMA to asynchronously fetch data from remote processes. We delve into the details of each component below.

Data Preloader The data preloader loads a dataset from the file system and initializes the distributed store. Data may be stored in per-sample or containerized file formats. DDStore provides plugins for reading different data formats. Data is split into chunks depending on the user-provided width or the number of replicas.

Data Registry After data is loaded into memory, each process registers its chunks. DDStore maintains a global registry of chunks on each process. In order to read another data chunk from a remote process, a process consults its registry to determine the location of the data item and issues a read operation to fetch the data object.

Data Loader The data loader performs the main task of reading data in-memory during the data loading operation (See Fig. 1). Data samples in a batch can be a random distribution of samples spread across processes. These form the set of data that are input to the model during the forward step. To get the next set of data items not in the process's local memory, a process generates a list of data items to be retrieved from remote locations and passes it to DDStore. DDStore performs a lookup for the data items in its internal sub-group of processes that the calling process belongs to. It then starts initiating one-sided `MPI_Get` operations to fetch data from remote processes. DDStore returns control to the application when all requested data items have been read.

One-sided RMA Communication Layer The communication layer performs the actual MPI RMA registration and read operations for fetching remote data. During the registration step, each process registers its memory region containing the data by calling MPI's `MPI_Win_create` function. In contrast to the tightly coupled MPI send-receive paradigm used in two-sided MPI communications, MPI's RMA minimizes the target process's involvement, helping to increase throughput. However, it still requires a non-blocking lock-unlock synchronization [63] to avoid data inconsistency arising from contention between multiple processes. Among MPI RMA's multiple synchronization mechanisms [63], we employ a read lock (`MPI_Win_lock` with `MPI_LOCK_SHARED`) and fence synchronization (`MPI_Win_fence`) as a lightweight set of contention-avoiding methods. The sequence of operations involved in MPI RMA are shown in Figure 3.

We have integrated DDStore into PyTorch's data library by creating a set of subclasses that inherit from `torch.utils.data.Dataset`. This enables PyTorch's data loader `torch.utils.data.DataLoader` to directly interact with our customized dataset. Our dataset classes extend basic data

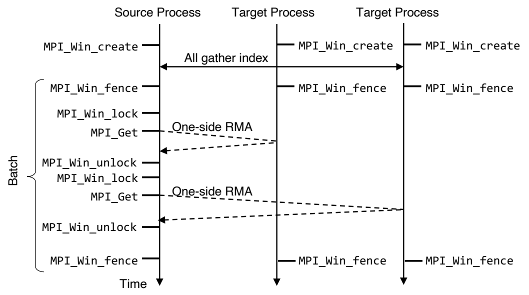

Figure 3: Example walk-through of DDStore's RMA operations for a batch size of 2.

readers for commonly used data formats and integrate them with DDStore for preloading, as well as MPI RMA registration and fetching.

4 Numerical results

In this section, we describe the hardware specifications, the datasets used, the setup for the HydraGNN model, the numerical results describing the training convergence, and the scalability of the training comparing the HPC-efficiency of different data management methodologies for scalable I/O.

We evaluate the scaling performance of DDStore using two US-DOE supercomputers - Summit at ORNL and Perlmutter at NERSC. Each Summit node consists of two IBM POWER9 CPUs with 512 GB of memory and six NVIDIA Volta GPUs with 16GB HBM2 memory. Each Perlmutter GPU-accelerated node has single AMD EPYC 7763 CPU with 256 GB of memory, and four NVIDIA A100 with 40GB HBM2 memory connected with NVLink-3.

4.1 Datasets

We utilize the following three datasets:

(1) Ising dataset: a synthetically generated dataset. Each atomic configuration is characterized by 125 atoms in an unit cubic. The orientation of the magnetic dipole moments of atomic spins is selected randomly for each atom. The energy is calculated with the closed analytical Hamiltonian formula that describes the Ising model for ferromagnetic materials. This synthetic dataset serves as a benchmark to anticipate future challenges in the scalable GNN training for fast and accurate predictions of material properties for ferromagnetic alloys [17, 41, 52, 53] on large volumes of first-principle calculations.

(2) AISD HOMO-LUMO dataset [8]: a dataset providing the energy gap between the highest occupied molecular orbital (HOMO) and the lowest unoccupied molecular orbital (LUMO) [11] for 10.5 million organic molecules. The molecules are diverse for chemical compositions and molecular size (the smallest molecule contains 5 non-hydrogen atoms, and the largest molecule contains 71 non-hydrogen atoms).

(3) ORNL AISD-Ex dataset [42]: one of the largest open-source datasets available within the community, created to train the high-dimensional UV-vis spectra of molecules in the AISD HOMO-LUMO dataset. Based on the prediction approach, we have two variants: discrete and smoothing datasets. The discrete dataset is tailored to 50-dimensional peaks and intensities in the UV-vis spectrum computed by DFTB, while the smooth dataset features a 37,500-dimensional spectrum obtained by Gaussian smoothing of the original peaks computed by DFTB. The high dimensionality of the smoothed UV-vis spectrum generates about 20 times larger data files than the discrete one.

Table 1: Dataset description.

	Graph Size				File Size	
Dataset	#Graphs	#Nodes	#Edges	#Feature	PFF[1]	CFF[2]
Ising	1.2 M	151 M	840 M	3584	24 GB	19 GB
AISD HOMO-LUMO	10.5 M	550.6 M	1.1 B	1	90 GB	60 GB
AISD-Ex (Discrete)	10.5 M	550.6 M	1.1 B	2x50	83 GB	64 GB
AISD-Ex (Smooth)	10.5 M	550.6 M	1.1 B	37500	1.6 TB	1.5 TB
AISD-Ex (Smooth & Small)[3]	10.5 M	550.6 M	1.1 B	351	114 GB	74 GB

[1]PFF: Per-object File Format, [2]CFF: Containerized File Format, [3]Reduced to use on Perlmutter

The main properties of these three datasets are summarized in Table 1. AISD-Ex Smooth is the largest dataset (about 1.5 TB). We process the full size on Summit but use a trimmed version on Perlmutter due to a disk quota limitation.

4.2 HydraGNN Setup

The HydraGNN architecture used for the numerical results is composed of six Principal Neighborhood Aggregation (PNA) layers [15], each with a hidden dimension of 200, and followed by three fully connected hidden layers with 200 neurons each. The number of neurons in the output layer is the same to the dimension of output features in each dataset. Particularly, the output layer has one neuron for predicting the energy of an atomic configuration in the Ising dataset or the HOMO-LUMO gap of an molecule in the AISD HOMO-LUMO dataset. For the ORNL_AISD-Ex dataset, the output layer has 100 neurons for the location and the intensity of 50 peaks in the UV-vis spectrum of an organic molecules, whereas the output layer has 37,500 neurons for the Gaussian smoothed UV-vis spectrum. The ReLU activation function [3] is used for both PNA and fully connected hidden layers. AdamW [39] optimizer with the default parameter setting in PyTorch [54] is used for training. The learning rate is adaptively adjusted with `ReduceLROnPlateau`, a learning rate scheduler, based on validation loss with an initial value equal to 1×10^{-3}. The training has been conducted for three epochs for performance data collection and 100 epochs for training error convergence, respectively, on 80% of the data, whereas the other 20% is split equally for validation and testing.

4.3 Data Management Methodologies

We compare the performance of DDStore with two distinct state-of-the-art data management approaches for HPC and supercomputing architecture that are not endowed with NVMe capabilities: i) PFF using Pickle, in which every sample is saved in Python's Pickle binary format, and ii) CFF utilizing ADIOS [22]. ADIOS manages containerized subfiles, each containing multiple data objects, as well as a data index for easy retrieval.

4.4 Performance Evaluation

Overview: Fig. 4 shows the normalized end-to-end training throughput. The training throughput is evaluated using a fixed batch size of 128, and normalized to PFF. The reported number is an average of three training runs. Compared to the PFF, DDStore improves the end-to-end training throughput by 2.93× (up to 4.23× on AISD-ex discrete) and 4.69× (up to 6.15× on AISD) on average for Summit and Perlmutter, respectively. Furthermore, DDStore outperforms CFF by a factor of 5.09× and 6.13× for the same supercomputers. Thus, DDStore achieves substantially higher training

Table 2: 50th, 95th, and 99th percentile of graph loading latency from Fig. 6

Percentile	Ising			AISD HOMO-LUMO			ORNL AISD-Ex (Discrete)			ORNL AISD-Ex (Smooth)		
	PFF	CFF	DDStore	PFF	CFF	DDStore	PFF	CFF	DDStore	PFF	CFF	DDStore
50th	2.25 ms	0.19 ms	0.24 ms	2.78 ms	9.69 ms	0.39 ms	2.76 ms	4.09 ms	0.44 ms	2.41 ms	3.19 ms	0.42 ms
95th	2.79 ms	1.23 ms	0.48 ms	4.22 ms	13.29 ms	0.67 ms	3.69 ms	5.03 ms	0.68 ms	3.21 ms	3.89 ms	0.64 ms
99th	3.43 ms	1.3 ms	0.58 ms	11.26 ms	15.68 ms	2.17 ms	4.17 ms	5.71 ms	0.89 ms	3.63 ms	4.39 ms	0.86 ms

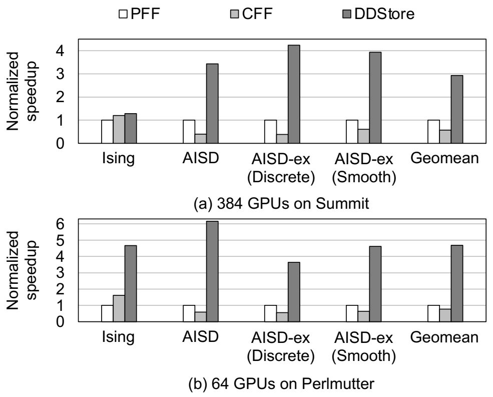

Figure 4: Normalized end-to-end training speedup comparison using (a) 384 GPUs on Summit (b) 64 GPUs on Perlmutter. Geomean represents the geometric mean of speedup across all four datasets.

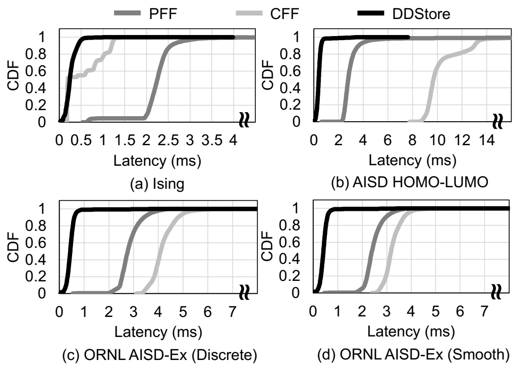

Figure 6: Graph loading latency CDF using 64 GPUs on Perlmutter.

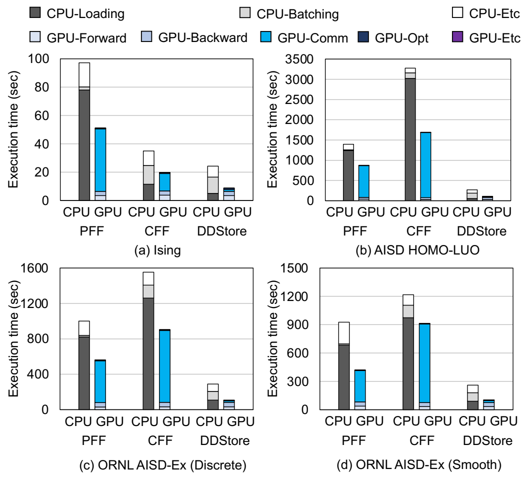

Figure 5: End-to-end training time breakdown of PFF, CFF, and DDStore using 64 GPUs on Perlmutter.

throughput than all the other file formats independent of the dataset size.

Source of Performance Improvement: Overall, DDStore benefits from distributed in-memory cache with one-side RMA communication. Fig. 5 shows the execution time breakdown of end-to-end training for each dataset on Perlmutter. Since the next mini-batch data preparation on CPUs overlaps with the current batch's gradient computation on GPUs, the figure represents the CPU and GPU operation with different stacked bars. Note that CPU-Loading represents the data loading time, and CPU-Batching is a processing time for grouping loaded samples into a unified representation. From GPU, GPU-comm refers to the communication time for model synchronization including communication stall time.

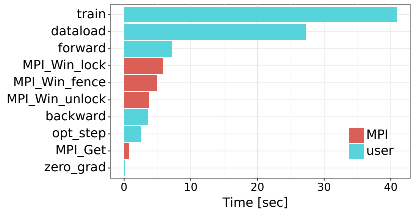

Figure 7: Profiling HydraGNN and DDStore using Score-P [33].

Compared to PFF and CFF, the figure shows that the most of time reduction by DDStore comes from CPU-Loading. By using the distributed in-memory cache, DDStore reduces CPU-Loading time by 90.68% and 84.31% on average compared to the PFF and CFF, respectively. For more details, Fig. 6 shows the cumulative distribution function (CDF) of the graph loading latency, and Table 2 shows the 50th, 90th, and 99th percentile of these latencies on each dataset. As shown in the figure and table, in CFF, 99% of graphs (or samples) are loaded within 1.3 ms, 15.68 ms, 5.71 ms, and 4.39 ms on each dataset. Note that in Ising, the dataset is small and containerized which is easy to prefetch (e.g, read-ahead) and cache (e.g., buffer cache) by the OS. Therefore, most of the graphs are loaded from memory, not from disk. As a result, 50% of graph batches in CFF are loaded within 0.19 ms, which is close to the latency of DDStore (0.24 ms). However, DDStore loads 99% of graph batches in Ising within 0.58 ms, while CFF needs 1.3 ms. In PFF, 99% of the graphs are loaded within 3.43 ms, 11.26 ms, 4.17 ms, and 3.63 ms in each dataset, respectively. On the other hand, DDStore loads 99% of the graphs within up to 2.17 ms by performing in-memory data management and optimized one-side RMA communication.

Furthermore, reducing the CPU-Loading time affects GPU-Comm time reduction. The main reason for the large GPU-Comm time in PFF and CFF is the imbalanced data loading time. If data preparation is delayed due to the tail latency of graph loading, the other tasks wait for the completion of the delayed task's training

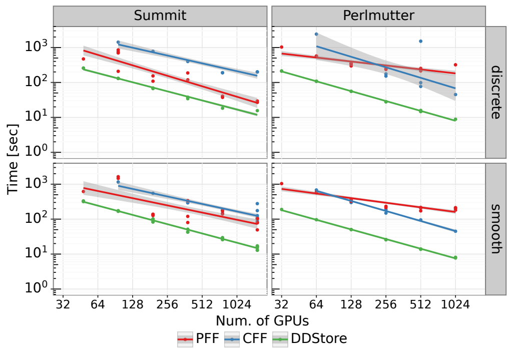

Figure 8: The scaling performance of DDStore using a fixed batch size of 128. The grey area illustrates the variability in the measurements.

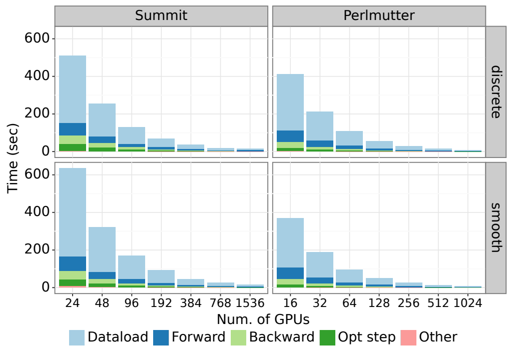

Figure 9: The performance breakdown of functions involved in training with DDStore using the same settings in Fig. 8.

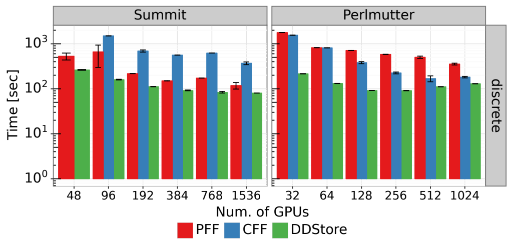

Figure 10: The scaling performance using a globally fixed batch size (6144 on Summit and 4096 on Perlmutter) for AISD-ex discrete.

iteration for model synchronization. DDStore minimizes the waiting during model synchronization by loading 99% of graphs in less than 3 ms.

In Fig. 7, we present a profiling outcome using Score-P [33] for HydraGNN training on the AISD-Ex discrete dataset with 64 Summit nodes. The results indicate the time distribution between MPI functions and individual training steps in a single epoch. Data loading accounts for approximately 67% of the training duration, while MPI RMA functions contribute to about 35% of the overall time spent in training.

4.5 Scaling Performance

In order to assess the scalability of DDStore, we conduct HydraGNN training using two extensive datasets, AISD-ex discrete and smooth, with and without a fixed batch size. The number of nodes used is consistent across both Summit and Perlmutter, ranging from 8 to

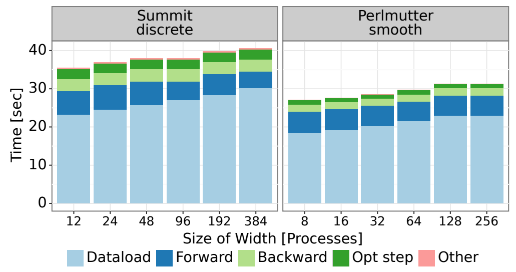

Figure 11: The performance with varying the width parameter.

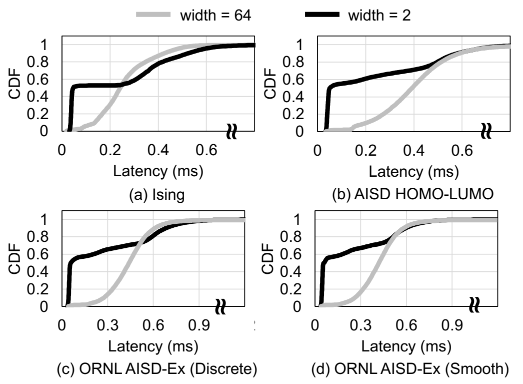

Figure 12: Impact of the width parameter on the graph loading latency. Figure shows the CDF with the default width (`width=64`) vs. `width=2`, using 64 GPUs (16 nodes) on Perlmutter.

256. However, the quantity of GPUs differs, with Summit having 48 to 1536 GPUs and Perlmutter having 32 to 1024 GPUs due to varying GPUs per node.

Initially, we perform the tests with a fixed batch size of 128, regardless of the number of nodes being used. As a result, the effective global batch size changes as the number of GPUs increase, similar to weak-scaling performance. The findings are displayed in Fig. 8. On both machines, DDStore exhibits a nearly linear performance as the number of GPUs doubles. The grey area represents the variability in the measurements. Although PFF and CFF exhibit higher variability, DDStore demonstrates consistent performance. Fig. 9 presents the durations of different functions in the training process using DDStore under identical configurations.

Subsequently, we execute HydraGNN with DDStore using varying batch sizes while maintaining a fixed global batch size (6144 on Summit and 4096 on Perlmutter). This approach is frequently employed by application scientists in data parallel modeling. The results are presented in Fig. 10. As the number of nodes increases, the local batch size decreases. Generally, a small local batch size leads to underutilization of GPUs and reduces efficiency. It is important to note that the performance disparity between DDStore and other methods (CFF and PFF) diminishes on Perlmutter, which can be attributed to the inefficiency caused by the smaller batch size as well as the differences in topology.

4.6 DDStore Width

We introduce a performance parameter that users can adjust to optimize the performance of DDStore. The DDStore width is a parameter used to determine the number of processes accessible by a MPI RMA group. We examine the effects of modifying the

Table 3: 50th percentile of graph loading latency using `width=64` (default) and `width=2` from Fig. 12

	`width=64`	`width=2`	Time reduction compared to `width=64`
Ising	0.24 ms	0.05 ms	79.17%
AISD HOMO-LUMO	0.39 ms	0.05 ms	87.18%
ORNL AISD-Ex Discrete	0.44 ms	0.06 ms	86.36%
ORNL AISD-Ex Smooth	0.42 ms	0.05 ms	85.71%

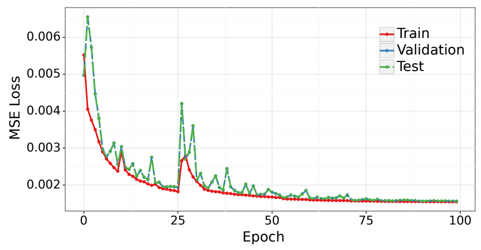

Figure 13: Convergence of training/validation/test loss.

DDStore width parameter on performance when processing the AISD-ex discrete dataset. On Summit, we adjust the DDStore width from 12 to 384, and on Perlmutter, from 8 to 256, utilizing 64 nodes in both cases. The entire dataset is evenly distributed among the width processes. Our findings can be observed in Fig. 11. For more details, Fig. 12 and Table 3 compare the cumulative distribution function with varying width parameter. Each performance is evaluated using 16 Perlmutter nodes, which means that the default width value is 64. As shown in the figure, half of the graphs are loaded much faster on "`width=2`" compared to the default. As shown in Table 3, the 50th percentile latency can be reduced by up to 87.18% compared to the default. The DDStore width does influence the performance, though the change is not significant in Fig. 11, resulting in less than a 10% variability in performance. Users can take advantage of this to further optimize the performance of training using DDStore.

4.7 Convergence

Our performance evaluation concludes by showcasing the convergence outcome as a high-level appraisal of DDStore. We performed a full-scale HydraGNN training for UV-vis spectrum analysis using the AISD-Ex (Smooth) dataset on Summit. This training took about one hour with 128 Summit nodes and a batch size of 128 for 100 epochs. During the training, we employed a learning rate scheduler named `ReduceLROnPlateau`, which adaptively modifies the learning rate based on validation loss. Fig. 13 shows the convergence of the mean square error (MSE) loss for the training, validation, and test sets. The abrupt rise in loss observed at Epoch 26 is due to the learning rate shifting from 1×10^{-3} to 5×10^{-4}. It can be observed that the training reaches convergence after approximately 90 epochs, resulting in an MSE loss between 0.015 and 0.016.

5 Related work

GNN: The advent of accelerator technologies, such as TPU [32], Tensor Core [44], Cerebras [18], GraphCore [31], has significantly enhanced the processing speed of DL workloads by using high-arithmetic-intensity kernel routines. Recent studies [27, 29] showed that these DL accelerators require higher arithmetic intensity for optimal performance than what is existing in DL workloads, shifting the challenge to the data movement. The problem is exacerbated with the increase in the integration level of accelerators within a compute node, which put excessive pressure on the I/O system and the interconnect [27, 28]. These studies show that scientific learning workloads, such as DeepCAM [36] and CosmoFlow [46], can be I/O bound when samples are not efficiently compressed or cached within the memory hierarchy [28].

Data loading optimization: I/O optimization in DL has been an active research area. Studies by C. Jia et al. [30] and J. Xue et al. [64] investigated the application of RDMA with TensorFlow for DL training, emphasizing efficient message passing within a distributed setting. The work most closely related to ours is DeepIO by Yue Zhu et al. [66, 67], which focuses on optimizing data shuffling by using RDMA for in situ data movement. This approach is implemented within the TensorFlow environment. In contrast, our DDStore method relies on the portable MPI's RMA library and is integrated with PyTorch's data loading method. NVIDIA's Data Loading Library (DALI)[1] is specifically designed to optimize data loading to effectively minimize overhead and boost training efficiency. It emphasizes creating flexible data pipelines for training and employs multi-thread approaches to support multi-GPU training, whereas our work concentrates on using an HPC-friendly memory caching method. FFCV [37] is a data loading tool that offers functionalities similar to DALI and various data processing pipeline techniques. Our DDStore and FFCV differ in data shuffling, with FFCV not providing explicit distributed data shuffling while DDStore facilitating global memory-to-memory data shuffling.

6 Conclusion

Despite recent advancements in computing hardware and DL tools, large-scale graph training remains challenging, particularly in I/O management. Traditional science applications executed on supercomputers are characterized by massive, write-intensive, sequential I/O patterns. In contrast, AI and DL workflows exhibit read-focused, frequent, random access I/O patterns, as demonstrated in our molecular design application. Although some supercomputing facilities provide massive NVMe SSD memory storage capabilities on each compute node to accommodate the intra-node storage of large volumes of data, such capabilities are still not broadly supported on many state-of-the-art HPC and some supercomputing facilities. Therefore, there is still a compelling need to develop scalable I/O techniques for effective data reading in GNN models that do not rely heavily on specific memory storage hardware.

To address these challenges, we developed DDStore, a tool designed to enhance data loading efficiency for distributed GNN training using one-sided communication in HPC. DDStore employs MPI, a widely-used HPC tool for data communication, and its one-sided communication paradigm to enable access to data stored in the distributed memory of remote processes. This approach allows data to be efficiently loaded into the memories of parallel processes with minimal overhead. We showcase the performance enhancement of DDStore in HydraGNN training, observing up to 4.23× and 6.15× improvement over traditional file-based approaches on Summit and Perlmutter, respectively—two DOE HPC machines.

In summary, DDStore facilitates rapid data exchange during shuffled data loading operations involved in the distributed training

[1]Available at https://developer.nvidia.com/dali

process. By implementing DDStore, we can accelerate the training process of GNN models for high-dimensional UV-spectrum prediction, enabling more accurate and efficient molecular design efforts. This breakthrough not only addresses the challenges posed by large-scale graph training but also lays the foundation for further innovations in the field of molecular design and DL applications.

Acknowledgments

This research is sponsored by the Artificial Intelligence Initiative as part of the Laboratory Directed Research and Development (LDRD) Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the US Department of Energy under contract DE-AC05-00OR22725. This work has been supported by the SciDAC Institute for Computer Science, Data, and Artificial Intelligence (RAPIDS), Lawrence Berkeley National Laboratory, which is operated by the University of California for the U.S. Department of Energy under contract DE-AC02-05CH11231. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory and the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 and No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP0025216, respectively.

References

[1] 2020. WebDataset library. https://github.com/webdataset/webdataset. Accessed: 2023/07.

[2] Martín Abadi. 2016. TensorFlow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 1–1.

[3] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).

[4] Alex Aizman, Gavin Maltby, and Thomas Breuel. 2019. High performance I/O for large scale deep learning. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 5965–5967.

[5] Roman M. Balabin and Ekaterina I. Lomakina. 2009. Neural network approach to quantum-chemistry data: Accurate prediction of density functional theory energies. *J. Chem. Phys.* 131, 7 (2009), 074104. https://doi.org/10.1063/1.3206326

[6] Chandler A. Becker, Francesca Tavazza, Zachary T. Trautt, and Robert A. Buarque de Macedo. 2013. Considerations for choosing and using force fields and interatomic potentials in materials science and engineering. *Curr. Opin. Solid State Mater. Sci.* 17, 6 (Dec. 2013), 277–283. https://doi.org/10.1016/j.cossms.2013.10.001

[7] Jörg Behler and Michele Parrinello. 2007. Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Phys. Rev. Lett.* 98, 14 (April 2007), 146401. https://doi.org/10.1103/PhysRevLett.98.146401

[8] Andrew Blanchard, John Gounley, Debsindhu Bhowmik, Pilsun Yoo, and Stephan Irle. 2022. AISD HOMO-LUMO. (5 2022). https://doi.org/10.13139/ORNLNCCS/1869409

[9] Andrew E Blanchard, Pei Zhang, Debsindhu Bhowmik, Kshitij Mehta, John Gounley, Samuel Temple Reeve, Stephan Irle, and Massimiliano Lupo Pasini. 2023. Computational Workflow for Accelerated Molecular Design Using Quantum Chemical Simulations and Deep Learning Models. In *Accelerating Science and Engineering Discoveries Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation: 22nd Smoky Mountains Computational Sciences and Engineering Conference, SMC 2022, Virtual Event, August 23–25, 2022, Revised Selected Papers*. Springer, 3–19.

[10] Roberto Car and Michele Parrinello. 1985. Unified Approach for Molecular Dynamics and Density-Functional Theory. *Phys. Rev. Lett.* 55 (1985), 2471–2474. https://doi.org/10.1103/PhysRevLett.55.2471

[11] Chi Chen, Weike Ye, Yunxing Zuo, Chen Zheng, and Shyue Ping Ong. 2019. Graph networks as a universal machine learning framework for molecules and crystals. *Chemistry of Materials* 31, 9 (2019), 3564–3572.

[12] Jong Youl Choi, Pei Zhang, Kshitij Mehta, Andrew Blanchard, and Massimiliano Lupo Pasini. 2022. Scalable training of graph convolutional neural networks for fast and accurate predictions of HOMO-LUMO gap in molecules. *Journal of Cheminformatics* 14, 1 (2022), 1–10.

[13] Kamal Choudhary and Brian DeCost. 2021. Atomistic Line Graph Neural Network for improved materials property predictions. *npj Computational Materials* 7, 1 (2021), 1–8.

[14] Kamal Choudhary, Brian DeCost, Chi Chen, Anubhav Jain, Francesca Tavazza, Ryan Cohn, Cheol WooPark, Alok Choudhary, Ankit Agrawal, Simon J. L. Billinge, Elizabeth Holm, Shyue Ping Ong, and Chris Wolverton. 2022. Recent Advances and Applications of Deep Learning Methods in Materials Science. *npj Computational Materials* 8, 59 (2022). https://doi.org/10.1007/978-3-031-23606-8_5

[15] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. 2020. Principal Neighbourhood Aggregation for Graph Nets. *arXiv:2004.05718 [cs, stat]* (Dec. 2020). http://arxiv.org/abs/2004.05718 arXiv: 2004.05718.

[16] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. 2016. An implementation and evaluation of the MPI 3.0 one-sided communication interface. *Concurrency and Computation: Practice and Experience* 28, 17 (2016), 4385–4404.

[17] Markus. Eisenbach, Mariia. Karabin, Massimiliano. Lupo Pasini, and Junqi. Yin. 2022. Machine Learning for First Principles Calculations of Material Properties for Ferromagnetic Materials. In *Accelerating Science and Engineering Discoveries Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation*, Kothe Doug, Geist Al, Swaroop Pophale, Hong Liu, and Suzanne Parete-Koon (Eds.). Springer Nature Switzerland, Cham, 75–86.

[18] Jean-Philippe Fricker. 2022. The Cerebras CS-2: Designing an AI Accelerator around the World's Largest 2.6 Trillion Transistor Chip. In *Proceedings of the 2022 International Symposium on Physical Design* (Virtual Event, Canada) *(ISPD '22)*. Association for Computing Machinery, New York, NY, USA, 71. https://doi.org/10.1145/3505170.3511036

[19] Victor Fung, Jiaxin Zhang, and Bobby G. Sumpter. 2021. Benchmarking graph neural networks for materials chemistry. *npj Computational Materials* 7, 84 (2021). https://doi.org/10.1038/s41524-021-00554-0

[20] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 97–104.

[21] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *arXiv:1704.01212 [cs]* (June 2017). http://arxiv.org/abs/1704.01212 arXiv: 1704.01212.

[22] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[23] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing* 22, 6 (1996), 789–828.

[24] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[25] Brian L. Hammond, William A. Lester, and Peter James Reynolds. 1994. *Monte Carlo Methods in Ab Initio Quantum Chemistry*. Singapore: World Scientific.

[26] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc.

[27] Khaled Z. Ibrahim, Tan Nguyen, Hai Ah Nam, Wahid Bhimji, Steven Farrell, Leonid Oliker, Michael Rowan, Nicholas J. Wright, and Samuel Williams. 2021. Architectural Requirements for Deep Learning Workloads in HPC Environments. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 7–17. https://doi.org/10.1109/PMBS54543.2021.00007

[28] Khaled Z. Ibrahim and Leonid Oliker. 2022. Preprocessing Pipeline Optimization for Scientific Deep Learning Workloads. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1118–1128. https://doi.org/10.1109/IPDPS53621.2022.00112

[29] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. arXiv:2007.00072 [cs.LG]

[30] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. 2018. Improving the performance of distributed tensorflow with RDMA. *International Journal of Parallel Programming* 46 (2018), 674–685.

[31] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the Graphcore IPU Architecture via Microbenchmarking. *CoRR* abs/1912.03413 (2019). arXiv:1912.03413 http://arxiv.org/abs/1912.03413

[32] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[33] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. 2012. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*. Springer, 79–91.

[34] Walter Kohn and Lu Jeu Sham. 1965. Self-consistent equations including exchange and correlation effects. *Phys. Rev.* 140 (1965), A1133–A1138. https://doi.org/10.

1103/physrev.140.a1133

[35] Quincey Koziol, Dana Robinson, and USDOE Office of Science. 2018. HDF5. https://doi.org/10.11578/dc.20180330.1

[36] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. 2018. Exascale Deep Learning for Climate Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Article 51, 12 pages.

[37] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Mądry. 2023. FFCV: Accelerating training by removing data bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12011–12020.

[38] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[39] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[40] Steph-Yve Louis, Yong Zhao, Alireza Nasiri, Xiran Wong, Yuqi Song, Fei Liu, and Jianjun Hu. 2020. Graph convolutional neural networks with global attention for improved materials property prediction. *Physical Chemistry Chemical Physics* 22, 32 (2020), 18141–18148.

[41] Massimiliano. Lupo Pasini, Marco. Burĉul, Samuel Temple Reeve, Markus. Eisenbach, and Simona Perotto. 2021. Fast and accurate predictions of total energy for solid solution alloys with graph convolutional neural networks. *Springer Journal of Communications in Computer and Information Science* 1512 (Sept. 2021).

[42] Massimiliano Lupo Pasini, Kshitij Mehta, Pilsun Yoo, and Stephan Irle. 2023. ORNL_AISD-Ex: Quantum chemical prediction of UV/Vis absorption spectra for over 10 million organic molecules. https://doi.org/doi:10.13139/OLCF/1907919

[43] Massimiliano Lupo Pasini, Samuel Temple Reeve, Pei Zhang, Jong Youl Choi, Massimiliano Lupo Pasini, Samuel Temple Reeve, Pei Zhang, Jong Youl Choi, and USDOE. 2021. HydraGNN, Version 1.0. https://doi.org/10.11578/dc.20211019.2

[44] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. *arXiv preprint arXiv:1803.04014* (2018). https://arxiv.org/abs/1803.04014

[45] Dominik Marx and Jürg Hutter. 2012. *Ab Initio Molecular Dynamics, Basic Theory and Advanced Methods.* Cambridge University Press New York, New York, USA.

[46] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J Pennycook, et al. 2018. CosmoFlow: Using deep learning to learn the universe at scale. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 819–829.

[47] Truong Thao Nguyen, François Trahay, Jens Domke, Aleksandr Drozd, Emil Vatai, Jianwei Liao, Mohamed Wahib, and Balazs Gerofi. 2022. Why globally re-shuffle? Revisiting data shuffling in large scale deep learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1085–1096.

[48] M. P. Nightingale and Cyrus J. Umrigar. 1999. *Quantum Monte Carlo methods in physics and chemistry.* Springer.

[49] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[50] Cheol Woo Park, Mordechai Kornbluth, Jonathan Vandermause, Chris Wolverton, Boris Kozinsky, and Jonathan P. Mailoa. 2021. Accurate and scalable multi-element graph neural network force field and molecular dynamics with direct force architecture. *npj Computational Materials* 7, 73 (August 2021). https://doi.org/0.1038/s41524-021-00543-3

[51] Cheol Woo Park and Chris Wolverton. 2020. Developing an improved crystal graph convolutional neural network framework for accelerated materials discovery. *Phys. Rev. Materials* 4 (Jun 2020), 063801. Issue 6. https://doi.org/10.1103/PhysRevMaterials.4.063801

[52] Massimiliano Lupo Pasini, Ying Wai Li, Junqi Yin, Jiaxin Zhang, Kipton Barros, and Markus Eisenbach. 2020. Fast and stable deep-learning predictions of material properties for solid solution alloys. *J. Phys.: Condens. Matter* 33, 8 (Dec. 2020), 084005. https://doi.org/10.1088/1361-648X/abcb10 Publisher: IOP Publishing.

[53] Massimiliano Lupo Pasini, Pei Zhang, Samuel Temple Reeve, and Jong Youl Choi. 2022. Multi-task graph neural networks for simultaneous prediction of global and atomic properties in ferromagnetic systems. *Machine Learning: Science and Technology* 3, 2 (2022), 025007.

[54] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[56] Patrick Reiser, Marlen Neubert, André Eberhard, Luca Torresi, Chen Zhou, Chen Shao, Houssam Metni, Clint van Hoesel, Henrik Schopmans, Timo Sommer, and Pascal Friederich. 2022. Graph neural networks for materials science and chemistry. *Communications Materials* 3, 93 (2022). https://doi.org/10.1038/s43246-022-00315-6

[57] Kevin Ryczko, David Strubbe, and Isaac Tamblyn. 2019. Deep Learning and Density Functional Theory. *Phys. Rev. A* 100, 022512 (2019). https://doi.org/doi/10.1103/PhysRevA.100.022512

[58] Soumya Sanyal, Janakiraman Balachandran, Naganand Yadati, Abhishek Kumar, Padmini Rajagopalan, Suchismita Sanyal, and Partha Talukdar. 2018. MT-CGCNN: Integrating Crystal Graph Convolutional Neural Network with Multitask Learning for Material Property Prediction. *ArXiv* abs/1811.05660 (2018). arXiv:1811.05660 http://arxiv.org/abs/1811.05660

[59] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[60] Kristof Schütt, Pieter-Jan Kindermans, Huziel Enoc Sauceda Felix, Stefan Chmiela, Alexandre Tkatchenko, and Klaus-Robert Müller. 2017. Schnet: A continuous-filter convolutional neural network for modeling quantum interactions. *Advances in neural information processing systems* 30 (2017).

[61] David Sholl and Janice Steckel. 2009. *Density Functional Theory: a Practical Introduction - Chapter 1.* John Wiley and Sons, Inc. Publication.

[62] Justin S. Smith, Olexandr Isayev, and Adrian E. Roitberg. 2017. ANI-1: an extensible neural network potential with DFT accuracy at force field computational cost. *Chemical science* 8, 4 (2017), 3192–3203.

[63] Rajeev Thakur, William Gropp, and Brian Toonen. 2005. Optimizing the synchronization operations in message passing interface one-sided communication. *The International Journal of High Performance Computing Applications* 19, 2 (2005), 119–128.

[64] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. 2019. Fast distributed deep learning over rdma. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–14.

[65] Linfeng Zhang, Jiequn Han, Han Wang, Roberto Car, and Weinan E. 2018. Deep Potential Molecular Dynamics: A Scalable Model with the Accuracy of Quantum Mechanics. *Phys. Rev. Lett.* 120, 14 (April 2018), 143001. https://doi.org/10.1103/PhysRevLett.120.143001

[66] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. 2018. Entropy-aware I/O pipelining for large-scale deep learning on HPC systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 145–156.

[67] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. 2018. Multi-client DeepIO for large-scale deep learning on HPC systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2018)*.

[68] Mahdi Zolnouri, Xinlin Li, and Vahid Partovi Nia. 2020. Importance of data loading pipeline in training deep neural networks. *arXiv preprint arXiv:2005.02130* (2020).