Partitioning Algorithms on Computation Graphs for Distributed Machine Learning

Jiakang Chen
Yale University

Andrew Tran
Yale University

Jeffrey Tan
Yale University

Abstract

Efficiently scaling machine learning training across many GPUs in a distributed environment requires strategies that balance model capacity and training speed. We propose a novel approach to sharding models based on their computation graph to optimize throughput, guided by empirical benchmarks of GPU communication speeds. Our framework includes algorithms for a range of scales: brute-force, heuristic-based, and hierarchical methods. We integrate these methods to support arbitrary PyTorch models to enable seamless microbatch-based pipelining for training large-scale models.

1 Introduction and Related Work

Efficient training of large-scale machine learning models requires distribution of compute to fully utilize hardware. With great results seen from scaling models, many modern machine learning models are too large to fit on a single GPU, leading to the desire for model parallelism. This involves splitting the computation across multiple devices, which also enables different parts of the model to be executed concurrently. This presents a challenge in how to best partition the model across GPUs to minimize communication overhead while maximizing compute use.

Data parallelism is a commonly used strategy in which the model is replicated across devices, and the training data is partitioned into smaller shards for parallel processing. Although this approach scales well for small to medium-sized models, it encounters limitations when scaling to larger models because of the high memory requirements for copying and storing multiple copies of the model. Furthermore, training becomes impossible when the size of a model is larger than the available memory on a single GPU.

Various frameworks have been developed to address these challenges. Facebook's FairScale and PyTorch's Fully Sharded Data Parallel (FSDP) [11] provide solutions for memory-efficient training by sharding both model parameters and gradients across devices. These methods leverage the benefits of data parallelism while reducing memory overhead, enabling larger models to be trained.

PipeDream [2] introduces a novel approach to pipeline parallelism by partitioning models based on a short profiling run, similar to our use of GPU communication benchmarking. This method ensures a balanced workload across GPUs while minimizing communication delays between pipeline stages. Similarly, NVIDIA's MegaTron-LM [9] combines data, model, and pipeline parallelism to train extremely large language models, demonstrating the importance of hybrid parallelism techniques for scaling.

Another closely related work is RaNNC [10], which, like our method, supports minimal code changes for existing PyTorch models. RaNNC partitions models by identifying atomic subcomponents of the computation graph and applying a dynamic programming algorithm to optimize partitioning. This approach aligns closely with our framework, which also supports arbitrary PyTorch models and employs benchmarks of GPU communication speeds to inform the partitioning process.

2 Partitioning Algorithm Implementation

In this section, we will describe a novel partitioning algorithm that we use to shard the computation graph of a large machine learning model among many GPUs. This section will be broken down as follows. We will begin with a description of the sharding problem we are trying to solve and how we will translate it into a optimal partitioning algorithm on graphs subject to a specific metric. We will then go into two different algorithms that can be used to solve this problem. The first is a brute force algorithm that is guaranteed to find an optimal partitioning; however it is only feasible for small inputs. The next algorithm will be a heuristic algorithm that will approximate an optimal partitioning. This algorithm will often not find an optimal partitioning; however, it scales much better with computation graph size and can be easily parallelized among multiple machines. We will also perform experiments to compare the real-world performance of these two algorithms. Finally, we will propose an extension of the heuristic algorithm that would be computationally feasible, even when the size of the model and the number of GPUs we are sharding across grows very large, as one may see in modern data centers.

2.1 Problem Description

When choosing how to shard a large machine learning model among many GPUs, it is important to note two things. We are interested in the setting of models that are too large to fit within a single GPU, so pure data parallelism is not sufficient to allow for training. Therefore, first, ideally we would want to shard the model roughly evenly among GPUs to fully utilize memory and compute usage. Furthermore, we need to consider how the calculations flow along the computation graph. Passing information within a GPU will be significantly faster than passing information between GPUs, and passing information between GPUs on the same server will also be significantly faster than passing information between GPUs on different servers. Therefore, we can make the observation that when sharding a large machine learning model, ideally we would want to place nodes in the computation graph on the same GPU shard, and we would want to place GPU shards on the same server next to each other.

We can generalize this idea into the following problem description. Suppose we have a directed acyclic graph (DAG) $G = (V, E)$ with $|V| = n$, with weights on the nodes $w_v \in \mathbb{R}$ for each $v \in V$, as well as a communication cost matrix $M \in \mathbb{R}^{k \times k}$. Our optimization problem is now the following. We want to partition G into k components $C_1, \ldots, C_k$, in way that will minimize the following quantity:

$$\text{Sharding Cost} = \text{Variance of Weights} + \text{Cross-Edge Cost}$$

Where the variance of weights is the variance of the total weight assigned to each component:

$$\text{Variance of Weights} = \sum_{i=1}^{k} \left(\sum_{v \in C_k} w_k - \overline{W} \right)^2$$

where $\overline{W}$ is the average total weight assigned to each component. The cross-edge cost is the total cost of the edges between different components where the cost of an edge is defined by the communication cost matrix M:

$$\text{Cross-Edge Cost} = \sum_{i \neq j} |A_{i,j}| M_{i,j}$$

where $|A_{i,j|}$ is the number of edges going from C_i to C_j, and $M_{i,j}$ is the (i, j)th entry of M. Observe by minimizing the sharding cost, we are minimizing the cost of edges going between different components while simultaneously keeping the total weight of the components similar.

We can observe how we can translate this optimal partitioning problem into the model sharding problem as follows. Our DAG is the computation graph of the machine learning model we are trying to shard. The weights on each node will be some measure of how large the module is, specifically the number of parameters of a layer as a proxy for memory usage. Each entry of the communication cost matrix $M_{i,j}$ will represent the cost of communicating between GPU i and GPU j, for instance the latency between the two GPUs. After this translation, note that finding an optimal partition of the DAG that minimizes the sharding cost is equivalent to finding a sharding of the computation graph onto different GPUs that simultaneously minimizes the communication cost between the GPUs, while keeping the total size of the shard on each GPU roughly the same to maximize model size.

2.2 Brute-Force Algorithm

We can solve the optimization problem described in 2.1 via the following trivial brute-force algorithm.

```
**Input**: DAG $G = (V, E)$. Node weights $w_v$. Number of partitions $k$. Communication cost matrix $M \in \mathbb{R}^{k \times k}$.
**Output**: A resulting partition of $V$ into $k$ components $C_1$, $\ldots, C_k$.
**Initialization**: We initialize bm $\leftarrow \infty$, bp $\leftarrow$ None, and bla $\leftarrow$ None.
**for all** $k$ partitions of $V$ **do**
    **for all** assignment of $k$ partitions to sets $C_1, \ldots C_k$ **do**
        m $\leftarrow$ the sharding cost of the current partition $C_1, \ldots, C_k$.
        **if** m < bm **then**
            bm $\leftarrow$ m, and update bm and bla to the current partition and assignment to sets $C_1, \ldots, C_k$.
        **end if**
    **end for**
**end for**
```

This algorithm simply loops through all possible partitions into components $C_1, \ldots, C_k$ and finds the one with the minimum sharding cost. Note that there are $\left\{ {n \atop k} \right\}$ partitions of V where $|V| = n$ and $\left\{ {n \atop k} \right\}$ are the Stirling numbers of the second kind [1]. It is known that asymptotically, we have that $\left\{ {n \atop k} \right\} = O(k^n / k!)$ [3]. There are $k!$ ways to assign the k partitions to $C_1, \ldots, C_k$ and it takes $|E| = O(n^2)$ iterations to compute the sharding cost, meaning that this brute-force algorithm runs in $O(n^2 k^n)$ time, which is exponential in n.

2.3 Heuristic Algorithm

We can trade some of the guaranteed optimality of the brute-force algorithm in 2.2 for better time complexity via an algorithm that approximates the optimal partitioning.

Input: Given DAG $G = (V, E)$, and random initial partition of DAG $C_1, \ldots, C_k$. Node weights w_v. Number of partitions k. Communication cost matrix $M \in \mathbb{R}^{k \times k}$, and number of iterations N.

```
Output: A resulting partition of $V$ into $k$ components $C_1,\dots,C_k$.
Initialization: We initialize bm $\leftarrow \infty$, bp and bla to the random initial partition.
for _ in $1$ to $N$ do
    for $i \leftarrow 1$ to k and $j \leftarrow 1$ to k do
        if $i = j$ then
            continue
        end if
        for all node $\in C_i$ do
            Move node from $C_i$ to $C_j$.
            Calculate a new greedy assignment to $C_1,\dots,C_k$ given this new partition.
            m $\leftarrow$ sharding cost of $C_1,\dots,C_k$
            if m $<$ bm then
                bm $\leftarrow$ m, and update bm and bla to the current partition $C_1,\dots,C_k$.
                break
            else
                Move node from $C_j$ back to $C_i$
            end if
        end for
    end for
end for
```

This algorithm begins with a random partition. For a specified number of iterations, N, for each pair of components in the partition, this algorithm tries to move one node between components and checks if it decreases the sharding cost, and in this manner, it finds a local minimum of the sharding cost function. This is a similar method to the Kernighan–Lin algorithm [6], which iteratively swaps pairs of nodes. However, note that the performance of the algorithm depends very heavily on the random initial partition. Therefore, for the best performance, this algorithm can be easily parallelized, as it can be run on many machines, each with their own random initialization, and then one can select the best partition found by all the machines. Furthermore, the time complexity of the above algorithm is $O(N(kn)^2)$ which is polynomial in n, unlike the exponential time of the algorithm in 2.2, making it much faster than the brute-force algorithm in theory.

2.4 Comparison of Algorithm Efficiency

Even though if we look at the theoretical time complexity of the heuristic algorithm with the brute-force algorithm, the heuristic should also be faster in practice. We conduct experiments to demonstrate that for models of any reasonably large size, the heuristic algorithm finds a partition with a much smaller sharding cost than the brute-force algorithm in an equivalent amount of time.

We will go into some more details regarding the experimental setup. We implement both the brute-force algorithm and the heuristic algorithm in Python. We run both algorithms on a simple computation graph consisting of 1001 nodes. We graph the sharding cost over time of both algorithms in Figure 1. Note we use the log scale for better visibility. Observe that for reasonably large computation graphs, the heuristic algorithm results in orders of magnitude lower sharding cost compared to the brute force algorithm, demonstrating its usefulness when the size of the computation graph grows.

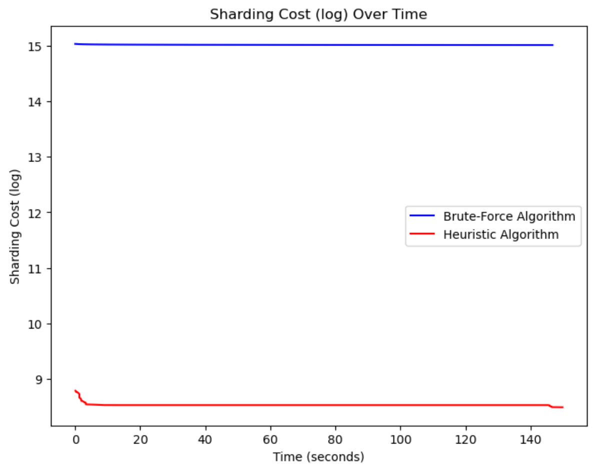

Figure 1: The log sharding cost of the brute-force and heuristic algorithm over time.

2.5 Extension to Data Centers: Hierarchical Algorithm

One thing to note is that often in large data centers, not only will we have large computation graphs with many nodes (large n), but we will also have large amounts of GPUs (large k). Recall from our algorithm in 2.3, that our algorithm will run in $O(N(kn)^2)$, which could be roughly $O(Nn^4)$ if both k and n are large and similar in size. We could extend our heuristic algorithm to be more efficient in this case by making the following observation. In contexts such as large data centers, there is naturally different levels or hierarchies of communication costs. For instance, there might be large communication latency between GPUs in server farms in different regions (i.e. Europe vs. NA); smaller latencies between GPUs in the same region but on different server racks; even smaller latencies still for GPUs in the same rack but on different motherboards; and finally the smallest latencies between GPUs interconnected electrically.

With this observation, we can extend our heuristic algorithm in 2.3. Note that rather than initially finding a partition over all the GPUs, because the GPUs are already naturally partitioned over region, server rack, motherboard, etc. we can recursively partition over these natural hierarchies of communication costs. For instance, we partition our DAG first over k_1 regions, rather than all K GPUs. Then once we get the set of nodes to allocate to each region, we then allocate those nodes for each region to k_2 server racks. We recursively do this until we finally have a partition over GPUs. This is significantly more efficient than immediately partitioning over GPUs, as if we just do our heuristic algorithm, we get a runtime of $O(N(Kn)^2)$. However, our recursive hierarchical algorithm (assuming we are partitioning over $k = O(1)$ components each time) results in a time complexity of:

$$\sum_{i=0}^{\log_k K} \frac{O(Nn^2)}{k^i} \leq \frac{k}{k-1} O(Nn^2) = O(Nn^2)$$

which is much faster than the $O(N(Kn)^2)$ when the number of total GPUs, K, grows very large.

We also implement this recursive hierarchical algorithm, and the details of the implementation can be found in the code. Running a small test to demonstrate the increased efficiency of the hierarchical partitioning algorithm compared to the heuristic algorithm, we run a partition on a computation graph with 201 nodes and 96 GPUs to partition over. When using our hierarchical algorithm, it found a partition with sharding cost 96.63 in 5.78 seconds. Comparing this to our heuristic algorithm, the heuristic algorithm found a partition with sharding cost 384.63 in 181.12 seconds. Therefore, our hierarchical algorithm when running on a computation graph with many nodes and many GPUs found a 4x better partition in 36x less time.

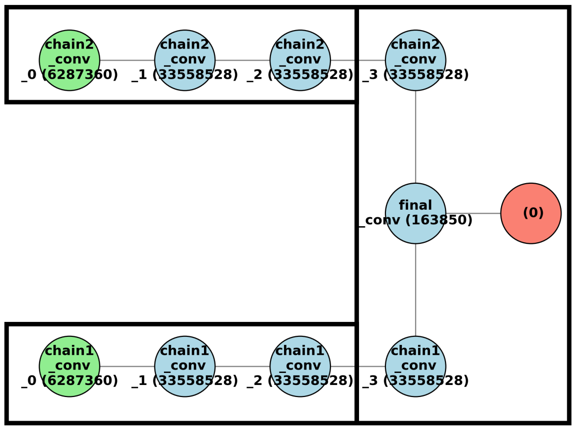

Figure 2: A computation graph generated, with the methodology described in 3.2, from a graph neural network with two chains of graph convolutions that combine in another convolution. The rectangles indicate partitions determined by our bruteforce algorithm as described in 2.2. The inputs of the model begin in the green nodes and end in the red node. The number in parentheses indicate parameter count excluding child modules that appear elsewhere in the graph.

3 Integration with PyTorch Framework

Alongside our algorithms, we provide a distributed GPU communication benchmarking script, and model-level parallelism sharding support for PyTorch [7] models with minimal code modifications. We choose to integrate with this framework due to its popularity and widespread support within the machine learning space.

3.1 GPU Communication Benchmarking

We provide a script that measures GPU-to-GPU communication speed, enabled by the NVIDIA Collective Communications Library (NCCL). This library helps frameworks including PyTorch abstract GPU communications above the physical interconnect layer, from one of the fastest being proprietary NVIDIA NVLink to a slower one with TCP/IP across the globe. Even within the same interconnect type, there are details that are harder to design for that can affect speed, including network topology. These empirical measurements are utilized in our algorithms to prefer assigning GPUs with fast connectivity to parts of the model with the highest data throughput requirements.

3.2 Model Splitting and Pipelining

We begin by using PyTorch's model tracing module, *torch.export*, which passes through dummy data into models to map out their execution paths. A critical design decision was how granular to allow splitting the model to shard across the GPUs. We give flexibility to the user of our framework by allowing them to specify which submodules of a model are the smallest atomic unit of sharding. Using the tracing module, we generate an execution graph where the vertices are modules and functions within the model and directed edges show the flow of data. From this, we isolate the nodes belonging to the aforementioned submodules and connect them with edges weighed by how many nodes interconnected them on their shortest paths in the original graph. Finally, we construct a DAG by finding a minimum spanning tree on this graph. Figure 2 demonstrates the result of this process on a graph neural network.

PyTorch's graph view of computation also simplifies splitting models into submodules. When partitioning is required, edges in the computation graph at a cut naturally define the data dependencies between new submodules.

After creating new submodules that can be executed on independent GPUs, we utilize PyTorch's implementation of GPipe [4], which achieves pipeline parallelism by splitting batches into micro-batches and overlapping forward and backward passes. This type of parallelism has it so that the submodules of the model spread across many GPUs are processed simultaneously, similar to pipelining in a modern computer processor. With our optimized partitioning and GPU assignment algorithms, we aim to complement the parallel compute with minimal communication bottlenecks.

4 Results

We view solutions to the problem as those that allow for efficient training of a large model that needs its computation sharded over many GPUs as it can not fit on just one. PyTorch provides FSDM which is compatible with many models out-of-the-box, which is what we will compare our method to. This is a fair baseline as it is another solution that is compatible with models with minimal code changes. For our framework, it is similar in implementation difficulty but with an added step of also marking the granularity of sharding of modules desired.

The model that we use for our tests is a multilayer perceptron consisting of three PyTorch "Linear" layers followed by a Softmax activation for multi-class classification. The task is given bag-of-words features of Amazon product reviews to predict product categories [8]. We utilize a batch size of 8192 and the Adam optimizer [5]. We train with 3x NVIDIA RTX 3090 24GB GPUs on the same node.

PyTorch FSDP is used with its default settings. For our framework, we do not split batches into micro-batches to avoid convergence differences for the most fair comparison.

From Table 1, our method achieves about a 6x improvement in training speed compared to FSDP.

Method	**Training Iteration Time (ms)**
PyTorch FSDP	12.48 ± 0.7
Automatic Partitioning + Pipeling	$\mathbf{1.89 \pm 0.8}$

Table 1: Mean and standard deviation of time per training iteration for each method.

5 Conclusion

In terms of shortcomings, our implementation depends heavily on PyTorch's support for Pipeline Parallelism which limits the scope of models that we support for now. We originally planned to use a graph neural network for experimentation but found that the built-in splitting functions for PyTorch is imperfect, especially for modules involve message passing which are more dynamic than standard neural network modules.

Additionally, it would have been interesting to experiment with GPUs across different nodes over slower interconnect methods like TCP/IP. We were unable to because of restrictions in virtualized networking with our cloud GPU provider.

Future directions include enhancing the flexibility of model partitioning algorithm to include even more empirically collected performance data, supporting more complex model architectures, and emphasizing a dynamic environment where GPUs and nodes are frequently added or removed.

References

[1] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete mathematics: A foundation for computer science*. Addison-Wesley, 1994.

[2] Aaron Harlap et al. *PipeDream: Fast and Efficient Pipeline Parallel DNN Training*. 2018. arXiv: 1806.03377 [cs.DC]. URL: https://arxiv.org/abs/1806.03377.

[3] L. C. Hsu. "Note on an asymptotic expansion of the *n*th difference of Zero". In: *The Annals of Mathematical Statistics* 19.2 (June 1948), pp. 273–277. DOI: 10.1214/aoms/1177730254.

[4] Yanping Huang et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

[5] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. eprint: arXiv:1412.6980.

[6] S. Lin and B. W. Kernighan. "An effective heuristic algorithm for the traveling-salesman problem". In: *Operations Research* 21.2 (Apr. 1973), pp. 498–516. DOI: 10.1287/opre.21.2.498.

[7] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. eprint: arXiv:1912.01703.

[8] Oleksandr Shchur et al. *Pitfalls of Graph Neural Network Evaluation*. 2018. eprint: arXiv:1811.05868.

[9] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2020. arXiv: 1909.08053 [cs.CL]. URL: https://arxiv.org/abs/1909.08053.

[10] Masahiro Tanaka et al. *Automatic Graph Partitioning for Very Large-scale Deep Learning*. 2021. arXiv: 2103.16063 [cs.LG]. URL: https://arxiv.org/abs/2103.16063.

[11] Yanli Zhao et al. *PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel*. 2023. eprint: arXiv:2304.11277.