CPSC 426: Distributed Machine Learning

Ayush Tibrewal
Yale University

Joshua Baehring
Yale University

Michael Tu
Yale University

Abstract

This write-up presents the design and implementation of a distributed systems machine learning (DSML) simulator for multi-GPU models. With simulated GPUs and a coordinator, we implement protocols resembling NCCL to enable efficient GPU communication and collective operations like AllReduceRing. Our system supports failure detection and dynamic GPU joining. We will examine various design choices along the way.

1 Objective

Modern machine learning models demand immense computational power, often requiring multi-GPU systems to train effectively. Coordinating GPUs involves managing inter-device communication, collective operations, and fault detection. Without robust frameworks, such a setup would suffer from inefficiencies, hardware failures, and bottlenecks in scalability.

This project aims to simulate a multi-GPU distributed ML environment, mimicking real-world systems like NVIDIA's NCCL. By implementing a simulated GPU system with efficient collective operations, fault-tolerance mechanisms, and dynamic scaling capabilities, we demonstrate an ability to use distributed machine learning concepts and tradeoffs in the development of our codebase.

2 Related Work

Much of this project is based on the NCCL library. Unsurprisingly, the issue of DSML is not new, and the NCCL library is NVIDIA's solution to these roadblocks. Of course, the NCCL library is far more expansive than our project, but the primary operations in NCCL are AllReduce, Reduce, Broadcast, Gather, and Scatter. Among these, AllReduce, combined with MemCpy's to zero out memory, can accomplish everything that the others can, so it is clear that AllReduce represents a key component of the library.

NCCL also has various performance capabilities that are incredibly useful. For example, they use NVLink to optimize data transfers, use multi-node setups to further allow scalability, and use asynchronous execution to allow for computations to occur at the same time as data synchronization.

3 Design Considerations and Tradeoffs

3.1 Overarching Design

As suggested by the DSML Lab outline, the interface on the client side allows users to create a Communicator, which owns some GPUs and allows the client to perform operations in a synchronized manner. The goal for this design is to provide flexible, simple, and scalable capabilities for the client.

This design's focus on providing the client with a coordinator, rather than forcing the client to handle the GPUs themselves, provides the simplicity that we want. Such a framework, based on the NCCL Library and provided protobuf model, is necessary for any clients seeking to set up distributed machine learning in order to avoid much of the difficulty that setting up a synchronized operation is. At the same time, we did not want to sacrifice too much flexibility, and allowing the client to freely do all operations necessary is desirable to reduce the distinction between a distributed GPU solution and a single GPU setup.

3.2 Memory Setup

Memory management within GPU devices posed another tradeoff to consider. We chose to preallocate file-backed memory, which works well in our development environment. this ensures consistent access for the GPUs and reduces the overhead from dynamic allocation during any operations. However, this means that the client needs to explicitly manage the memory, with specifications for source and destination memory addresses necessary throughout the entire process from the client. This puts the memory management in the client's hands which may not follow the goal of ease of use, although it does give the client more flexibility. This also means we do not have dynamic allocation, and GPU's being initialized without enough memory would provide further inconveniences to the client.

In this situation, we chose to give flexibility, at the sacrifice of simplicity. Since the client needs to manage memory on GPU's manually, this would add complexity to the client code. However, by allowing this flexibility, we can allow operations such as reducing different relative addresses within a GPU's memory. For instance, if our GPU's are not all the same specification, they may have different capabilities in efficiency or memory. By giving the client more control over the GPU's memory, it means that they could, for example, give training sets of different sizes to different GPU's. While this could lead to non-uniform (relative) memory address locations between GPU's, it is convenient to let the client specify which addresses are reduced for each GPU.

3.3 AllReduce Chunked Algorithm

We also had to choose an algorithm for AllReduce. While AllReduceRing was suggested, various algorithms have different tradeoffs in speed and RPC size. We felt that it was more likely for the various GPU's in our system would still be geographically nearby. While some applications such as a crowd-sourced compute would not satisfy this, for many projects, a client may have various GPU's to work with, without wanting to implement a distributed system to work with them. This means that sending more RPCs is not as expensive, but avoiding a performance or reliability bottleneck is especially desirable. With this prior, it is reasonable to prefer a ring reduction method over a central GPU solution. However, rather than the basic AllReduceRing algorithm, we felt that the chunked solution would be better, and this is what we implemented.

The AllReduceRingChunked algorithm seemed like a more appealing solution because it reduces the size of each individual RPC. As mentioned, it seemed unlikely that the number of RPC calls is a bottleneck. However, depending on the setup, longer RPCs could introduce higher risks of failure, and in general could be detrimental to the system. This choice was primarily made based on our own assumptions of what the hardware setup of these GPU's might look like. Of course, such a choice could be based on incorrect assumptions, but we felt that this would be the best option.

3.4 Failure Detection

We also chose to provide failure detection with transparent restarts to boost the reliability of the system. Of course, this also came with tradeoffs. Firstly, we used a simple failure detection, just based on X failures out of Y calls. We chose to do this over a heartbeat or timeout solution due to ease of implementation. Another problem is that it assumes the failures will be in a predictable manner, whereas in reality, failures could occur in different ways that don't cause RPC's to fail, but partial fails or corrupted communication may be harder to detect. This could certainly be explored as an avenue of improving the project in the future.

4 Future Improvements

As mentioned, scalability and ease were both considerations that we wanted to provide. At the moment, the system eases the load on the client, but the client still needs to do a lot of bookkeeping to be really effective. For instance, we only provide absolute addressing, which means that the client needs to track all of the GPU's memory spaces. In general, since this is not a complete implementation of all of the NCCL library, we are missing some capabilities that may be convenient for client-side implementations. For example, we do not have a Broadcast operation, so we would need Memcpy calls for each GPU, which is certainly not as convenient.

Another consideration is the centralization on reliance with the coordinator. While this is the most convenient in implementation, it would help scalability to move more responsibilities to the GPU's in performing various operations. For instance, it could be valuable to enable GPU's to use their own knowledge of the ring topology to perform parts of the Ring Reduce operation alone. Instead, there may be a performance bottleneck as the GPU's need to communicate with the central coordinator throughout the process. This performance bottleneck could certainly hurt scalability as the number of GPU's increases.

5 Acknowledgements

Thank you to Professors Richard Yang, Xiao Shi, and Scott Pruett for teaching this course. We all found the course content to be interesting, and problem sets were generally quite enlightening. Thank you to all of the course staff, including ULAs and TAs, who helped navigate occasionally challenging theory, alongside complex implementations.