CSC290 Final report

Leo Sciortino

December 2024

1 Introduction

As the popularity of DLRM (deep learning recommendation models) and LLM (large language models) has grown, so has the need for collective communication. For inference and training, these models often have memory and compute needs that cannot be efficiently served by one GPU. Methods of running models on multiple GPUs include data parallelism, model parallelism, and tensor parallelism, all of which can require collective communication. Collective communication is the coordination of sending data between multiple devices. In our case, we study NVIDIA's NVLink interconnect.

With the rise of collective communication algorithms brings the need to efficiently produce these algorithms for a given GPU network topology. Cai et al. [5] given constraints can synthesize collective communication algorithms under a Pareto frontier for optimal latency and bandwidth. For testing algorithms and their latency they use expensive systems such as the NVIDIA DGX1 computer which consists of 8 NVIDIA V100 GPUs. For some, testing on a machine like this may be out of reach due to cost. One can't just test on a single device since testing collective communication requires many GPUs. This gives the motivation to this work: have a simulator on which we can test collective communication algorithms.

In this work, we will do the following:

1. Provide sufficient background on collective communication
2. Provide sufficient background on "Synthesizing Optimal Collective Algorithms" (SCCL), the work of Cai et al. [5]
3. Provide sufficient background on the ns-3 [1] discrete event simulator
4. Provide details of our implementation
5. Discuss problems encountered while we were working
6. Provide results
7. Discuss shortfalls and possible optimizations of this work

2 Background

2.1 Collective Communication

To effectively send and receive data within a network of GPUs we use collective communication algorithms. The most common library used is NVIDIA's collective communication library (NCCL). NCCL consists of handwritten GPU kernels for specific GPU network topologies. When there is a new topology, engineers must by hand re-write the NCCL implementation. NCCL algorithms may not always be optimal depending on the data size this will be explained in depth later.

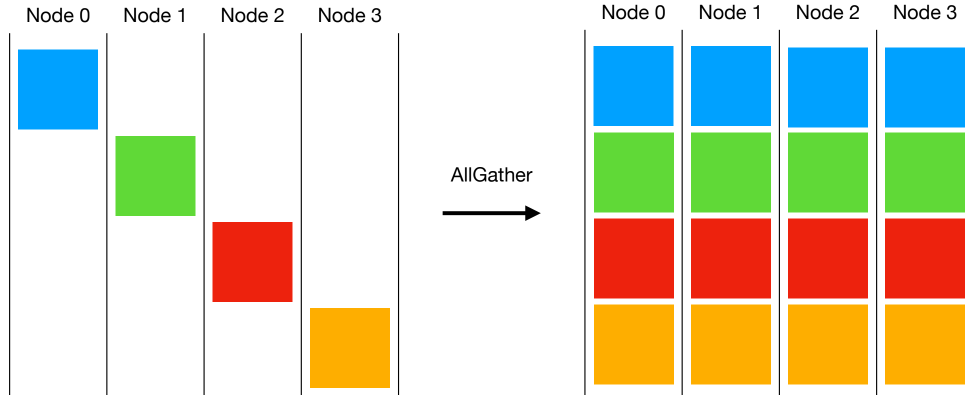

Figure 1: Example of an Allgather on 4 Nodes [4]

The goal of collective communication algorithms is to efficiently communicate data between nodes. One basic algorithm is Allgather [4, 5]. This algorithm copies data from all nodes to all other nodes, so in the end, every node has the same data. This can be seen in figure 1. The data represented in blue starts on Node 0, while the other colors represent data starting on their respective nodes. After the Allgather, each piece of data represented by the four colors is on every node. This could be achieved by transmitting data between all nodes, but it would be inefficient. There are many other relevant collective communication algorithms, but in this work, we focus on Allgather.

2.2 DGX1

The Nvidia DGX1 [5, 2] is server consisting of 8 GPUs produced by NVIDIA. This consists of 8 V100 GPUs connected by NVLink2.0 with two 20-core Intel Xeon E5-2698 [5]. Each GPU has capacity for 6 NVLinks with a bidirectional bandwidth of 50GB/s which means each direction has a bandwidth of 25GB/s. The 8 GPU nodes are connected in a ring topology. There are 6 independent rings. This can be seen in figure 2. One ring has 2 links per connection, the other has 1 link per connection. These rings don't overlap so they can be used for separate sends.

2.3 SCCL

The work Synthesizing Optimal Collective Algorithms [5] use an SMT solver given constraints to synthesize an optimal algorithm. This work models the cost of algorithm as the approximate communication time it takes. They say that for latency α and inverse bandwidth β the time cost is $\alpha + \beta \cdot L$ time. A latency cost is a fixed cost such as kernel launch and packet delay which is the time it takes to send one packet across the link. The bandwidth increases linearly with the amount of data being sent. Cai et al. [5] models the total cost of a collective algorithm with an input size of L as $a \cdot \alpha + b \cdot L \cdot \beta$ where a is the latency cost, and b is the bandwidth cost. We say that an algorithm is latency optimal if of all synthesizable algorithms it has the lowest a. An algorithm is bandwidth optimal if for all synthesized algorithms it has the lowest bandwidth cost b. Depending on the size of data, a latency optimal algorithm or a bandwidth optimal algorithm could have a lower total runtime.

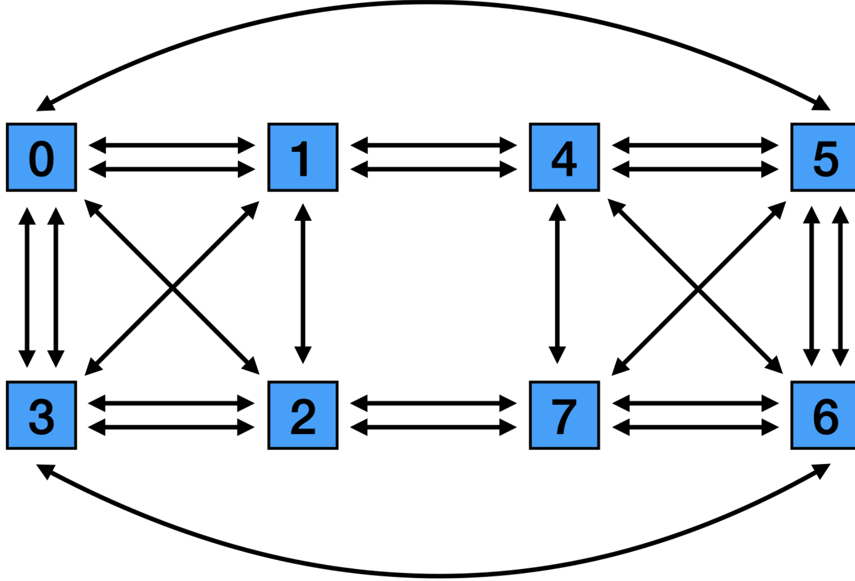

Figure 2: DGX1 NVLink topology [5]

SCCL synthesizes k-synchronous algorithms. These algorithms after a certain number of rounds of sends synchronize. For example on the DGX1 we could have a send from node 3 to node 1 and a send from node 2 to node 0, then after those sends complete we start a separate send from node 4 to 7. This synchronization is necessary since most times the next send requires data from the previous send. This synchronization takes place after each step; within each step there can be r_i rounds. Each round has the same send size. After all rounds complete in a step, then the next step can start. We say that the total number of steps is S and the total number of rounds is R. To create a more efficient utilization of the topology we can also send data in chunks. We say if we have C chunks, then the size of each data send is $\frac{L}{C}$. The latency cost only happens once per send. The bandwidth cost is total number of rounds times the bytes per rounds times the inverse bandwidth. This model of the total cost in their model comes out to be as follows.

$$S \cdot \alpha + R \cdot \frac{L}{C} \cdot \beta$$

This is a good model for the cost of how much a send takes but there are many other variables so it would be good to see if we can create an accurate simulation of NVLink.

3 ns-3 Simulator

Ns-3 [1] is a discrete event network simulator. This means that at each time step the simulator looks if there are any events scheduled, if there are it runs these events, and from these events can possibly schedule more events in the future. Once there are no more events scheduled, or a hard time limit has been reached the simulator ends.

There are key abstractions in ns-3 that we used to build our simulator [1]. We will briefly describe them here [1]

1. **Node:** Can model a computer. In this case, we use it to model a GPU. It could also model a network switch.

[1] AI tools like ChatGPT and GitHub Copilot were used to assist in building and debugging some parts of the simulator. Example code provided here `https://gitlab.com/nsnam/ns-3-dev` was also used to design applications.

2. **Channel:** A wire that connects two nodes. Has a given delay and bandwidth. In this case, we used a special Channel the `PointToPointChannel`
3. **NetDevice:** This can be thought of as a network card. NetDevices are attached to Channels and are installed on Nodes. We use a `PointToPointNetDevice`.
4. **Application:** Applications are high level programs that generate traffic. They can communicate to a Net Device through a socket. In this case, we use a special Application called a `BulkSendApplication`.

The idea for this simulator is that it can work on topologies that don't have equal bandwidth and delay on each link. This requires us for a need to synchronize between each step. This brings us the need to create two new classes `MultiBulkSendApplication` and `BulkSendSyncManager`. The idea is that in a driver file we read in the topology an for each Node we supply it a vector of sends. This vector of sends contains another vector of outgoing sends (if the Node is connected to 6 other Nodes this could contain up to 6 outgoing sends). Each round of send contains the necessary applications in order to start simulating each step. In this case, we can combine rounds together. If there are two rounds of sends to a specific Node we can just double the amount of data being sent.

To simulate data send from Node A to Node B, we use a `BulkSendApplication`. You supply this application a number of bytes to send and it will send create packets and send them as fast as possible. Packets can be received on another Node with a `PacketSink`.

We register the number of nodes with the static global `BulkSendSyncManager` that was mentioned earlier. We then create `MultiBulkSendApplication` and supply them the data mentioned earlier. We then calculate the total bytes we expect to receive total on all packet sinks for a specific send. We can tap into a trace for the application sending packets and application receiving packets. For all of these applications on a Node we register callback functions that keep track of the total number of bytes received on the packet sink. We then start all sends for a given step. Once the callback function for bytes being received sees that it reaches the total number of bytes that should be received it calls a function in the synchronization manager. Once the synchronization manager sees that all nodes have completed, it calls a callback function that each `MultiBulkSendApplication` has registered that will then start the next round of sends and repeat the above steps. This continues until all steps are completed.

After many bugs fixed this seemed like it was working but as soon as I ran it on complex applications like Allgather it broke and was very difficult to debug. To obtain results, I created a simplified simulator that only has one send per step. This is possible since in the SCCL paper we only look at topologies that have equal bandwidth and latency across each link. So, this means that the total number of bytes sent per step is rounds times the number of bytes divided by the chunks.

4 Results

I had issues simulating very large data sizes even after changing the internal buffer size inside of ns-3. It seems `MultiBulkSendApplication` may not be made to send this much data despite the name.

Since we are simulating for DGX1 we set the data rate to 200 Gib/s. We did this because internally ns-3 creates headers for the packets. The only problem is that sometimes all of the packets aren't received but we still receive enough bytes for the bulk send to say it's done. This could have to do with a discrepancy of where the bytes are counted: before or after the headers have been created. After much debugging and trying to access many trace sources we couldn't

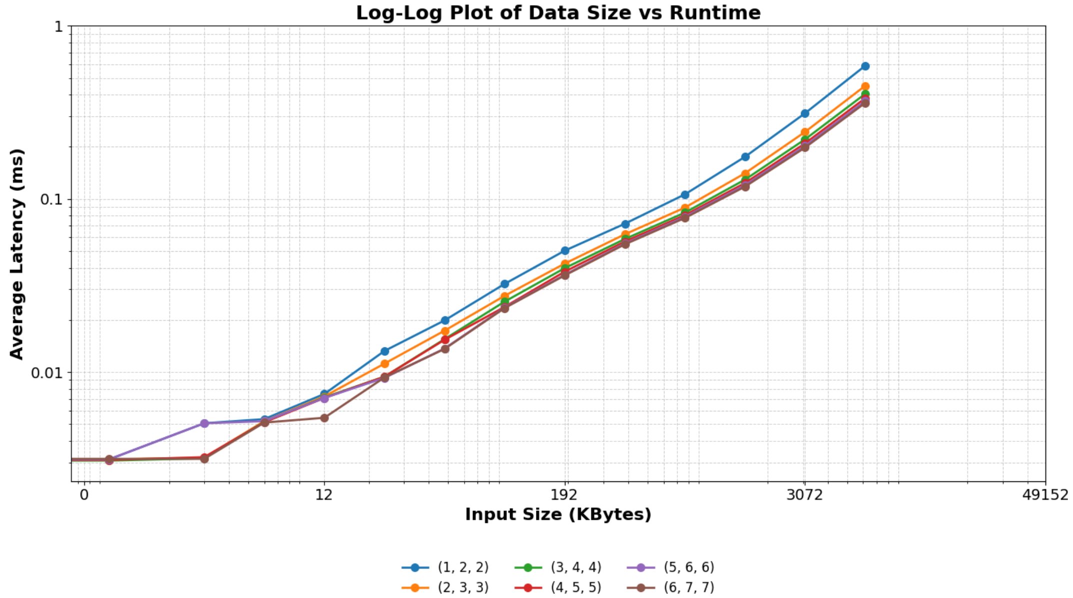

Figure 3: Our simulated results of Allgather

figure it out. It was also tough to find a latency for the NVlink2.0 but it seems like it should be around 1 micro second [2] [3]. It is unclear if this accounts for kernel startup.

We can see in figure 3 [3] the latency cost doesn't seem to be a factor compared to the latency cost in figure 4. This is because the simulation didn't account for enough latency cost. Once data size increases and latency is less of a cost, we see that the graph starts to become much more accurate. I only had the graph from the SCCL paper, so I couldn't directly compare the accuracy of results. The simulator seems to not run efficiently or not at all past 6144 Kbytes, so the testing was stopped there. This could be due to the fact that to get this simulator working I had to increase the internal buffers inside of ns-3.

4.1 Limitations

This work uses the built-in bulk send application and TCP sockets. These higher level network protocols definitely don't simulate NVLink properly. NVLink is closed source, which makes it hard to simulate. In the future, we would also like to compare our results with the AstraSimV2.0 [6]. We can compare our models to their analytical models and see what is more accurate and which takes more time. It is also possible to incorporate some analytical modeling of kernel launches within ns-3 to improve our latency cost estimation. In the future, we would also like to get the original simulator working that would work for topologies that have links of different bandwidths.

5 Conclusions

Overall, this work modeled synthesized collective communication algorithms on NVLink. This was done using the ns-3 discrete event simulator. The initial results are promising in approximating bandwidth costs, but currently latency costs aren't properly being modeled. The code repository can be found at `https://github.com/fubio/ns3-dev`

[2] It was very hard to find propagation delay of NVLink, originally the only place I could find propagation delays was the FAQ section of massedcompute.com. This source turns out to be unreliable as answers can be AI generated. The included citation is likely where the AI got its information and is a more reliable source.

[3] AI tools like ChatGPT and Github Copilot were used to create the Matplotlib code to produce the graph.

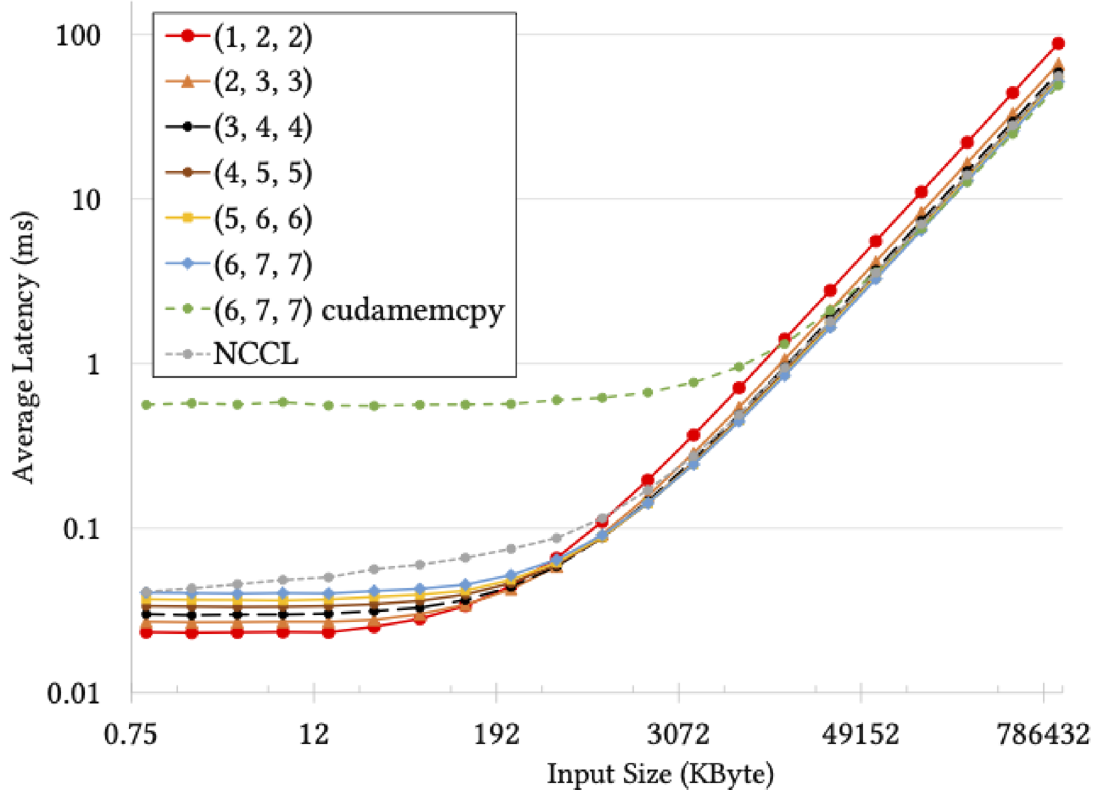

Figure 4: Allgather results from Cai et al. running on a DGX1

References

[1] ns-3 Network Simulator, 2008. URL `https://www.nsnam.org/`.

[2] NVIDIA DGX-1 With Tesla V100 System Architecture, 2017. URL `https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf/`.

[3] NVLINK on RTX 2080 TensorFlow and Peer-to-Peer Performance with Linux, 2018. URL `https://www.pugetsystems.com/labs/hpc/nvlink-on-rtx-2080-tensorflow-and-peer-to-peer-performance-with-linux-1262/`.

[4] Collective Operations, 2020. URL `https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html`.

[5] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21. ACM, February 2021. doi: 10.1145/3437801.3441620. URL `http://dx.doi.org/10.1145/3437801.3441620`.

[6] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. Astra-sim2.0: Modeling hierarchical networks and disaggregated systems for large-model training at scale, 2023. URL `https://arxiv.org/abs/2303.14006`.