Allreduce - the basis of multi-device communication for neural network training (/allreduce-the-basis-of-multi-device-communication-for-neural-network-training.html)

Introduction

For big deep learning models and datasets, training on a single GPU (or TPU or CPU) may not be fast enough. For example, according to NVIDIA's Megatron-LM (https://github.com/NVIDIA/Megatron-LM) code base, training BERT (https://arxiv.org/pdf/1810.04805.pdf) Large takes 3 days on 64 Tesla V100 GPUs. Assuming linear scaling, this would mean that training on a single GPU would have taken 192 days, or more than 6 months! Of course, scaling is hardly ever perfectly linear, but even if we assume 50% efficiency, that would have still meant 3 months instead of 3 days. Scaling is what makes certain problems feasible that used not to be viable with smaller compute resources. Big O of a task may not change, but scaling up the processing of a problem with a given complexity can make a difference.

The benefit of mini-batch stochastic gradient descent (SGD (https://en.wikipedia.org/wiki/Stochastic_gradient_descent)) and related optimization algorithms is that they take a few examples a time and run the forward and backward passes with enough data at once to keep the hardware busy. For example, in case of the GPU, we want to at least have enough data to exploit data parallelism (https://en.wikipedia.org/wiki/Data_parallelism) for bandwidth-bound (https://softwareengineering.stackexchange.com/questions/140534/what-makes-an-application-memory-bandwidth-bound) layers, and to get both data parallelism and data reuse (http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture11.pdf) for compute-bound (https://en.wikipedia.org/wiki/CPU-bound) operations such as matrix multiplication or convolution. The same logic applies to multi-GPU and multi-node processing - given a per device batch size N and K devices, we could simply sample a larger batch KN and still have N examples to work with per device. Typically, as the batch size grows, we can increase the learning rate and take fewer steps over the data, with each step contributing to a larger update of the weights. This actually doesn't always work well without various heuristics (e.g. 1 (https://arxiv.org/pdf/1706.02677.pdf), 2 (https://arxiv.org/pdf/1711.04325.pdf), 3 (https://arxiv.org/pdf/1708.03888.pdf)), but with the heuristics, we can scale training and converge much faster on many GPUs/TPUs/CPUs or many multi-device nodes.

Training on multiple devices requires communication between them, so let's now talk about communication patterns.

Deep learning training communication patterns

The forward pass through a neural network is simple - it's an embarrassingly parallel (https://en.wikipedia.org/wiki/Embarrassingly_parallel) problem, meaning that each device does its own work and there's no need for communication. All we need to do is for the data loader to make sure that different devices get

different examples for their slice of the batch, but let's ignore that for now.

The backward pass is not so straightforward - we calculate gradients based on a different sub-batch on each device, but then we want to average the gradients from different devices to get the overall gradient estimate. Once we get that estimate, we'd like to apply the same averaged gradient to the weight copies on each device, which are kept there for data locality. This will ensure that the weight copies remain synchronized after each iteration. Clearly, we'll need inter-device (or inter-node) communication for gradient averaging.

Gradient averaging is achieved via an operation called allreduce. Allreduce takes a data chunk, applies a reduction operator as the partial calculations are passed across devices, and then makes sure that the reduced result is communicated to all devices. Reduction is typically done on monoids (https://en.wikipedia.org/wiki/Monoid), meaning a pair of data type and operator such that the operation for the data type has an identity (e.g. 0 in case of addition or 1 in case of multiplication), the operation itself is associative, and the operation takes two operands and produces one result. Numerical data forms a monoid under addition, because we can determine the identity to be 0, the + operator takes two operands and produces one output value (the sum), and associativity holds because 1 + (2 + 3) = (1 + 2) + 3. Numerically, associativity doesn't actually hold (https://en.wikipedia.org/wiki/Associative_property#Nonassociativity_of_floating_point_calculation) for floating-point data due to round-off error, however it holds mathematically, and there exist more stable summation algorithms than just the vanilla summation that we're used to (e.g. Kahan summation (https://en.wikipedia.org/wiki/Kahan_summation_algorithm)). So, for all intents and purposes, let's assume that be it ints or floats, they form a monoid under addition and can be legally reduced. The associativity property buys us the ability to do reductions in arbitrary order, which tends to be important especially for parallel processing.

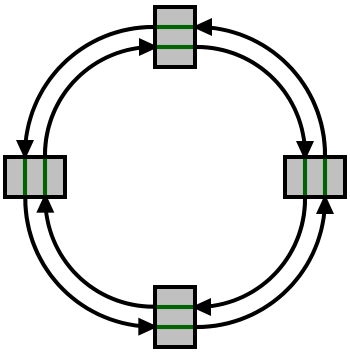
Let's assume that we have 4 devices, and 1 weight that we need to sum-reduce. All we need to do is to sum up all the values and broadcast the result to all devices.

```
>>> import numpy as np
>>> x = np.random.randn(1, 4)
>>> x[:] = x.sum()
>>> x
array([[2.79653147, 2.79653147, 2.79653147, 2.79653147]])
```

Of course, this looks simple, but how would we do it efficiently? That deepends on the network topology. We could have various topologies, such as a fat tree (https://en.wikipedia.org/wiki/Fat_tree), mesh (https://en.wikipedia.org/wiki/Mesh_networking), ring (https://en.wikipedia.org/wiki/Ring_network), etc. Here, we will focus on a ring topology. I will shortly explain why that's a good choice for allreduce. Note that the topology of the reduction can be a ring, but the topology of the network can be more arbitrary, as we'll see shortly.

Communication on a ring

In a ring, each device has 2 neighbors - let's call them left and right neighbors. A device can only send data directly to its neighbors, otherwise the data needs to make several hops as it's relayed across more devices.



In reality, few networks are designed this way, because sending data to devices that are several hops away would be very costly, wasting bandwidth of intermediate devices and adding latency of extra hops. Another problem with such an arrangement is that if one of the links between neighbors breaks, the whole network can go down. At the same time, even if you imagine a network with a different topology, say a fat tree, you could arrange communication in such a way that allreduce still takes paths on a ring. There are many algorithms which wouldn't take advantage of a ring arrangement, e.g. when all-to-all communication is needed, but ring is perfectly fine for allreduce. In fact, allreduce on a ring is bandwidth-optimal (https://dl.acm.org/citation.cfm?id=1482266), so if you have a lot of data to send, this is actually a perfect algorithm. Ring allreduce isn't latency-optimal, and other algorithms and topologies are preferred in that case. For now though, let's assume that we have a problem that requires us to optimally use the bandwidth.

Ring allreduce is actually a meta communication collective (http://www.rc.usf.edu/tutorials/classes/tutorial/mpi/chapter8.html), composed on two simpler collectives: reduce-scatter and allgather.

Reduce-Scatter

Reduce-scatter simply means that we wait on a message from our preceding neighbor, reduce (accumulate in some way, e.g. add) the incoming state, and on the next iteration of the communication, we send that message to our succeding neighbor. If we divide the data tensor into K chunks, where K is the number of GPUs, we'll always be receiving and sending only one of the K chunks. It is easy to see that if each of the chunks is received, reduced and passed around K-1 times, then the device that receives and accumulates the chunk last will have the fully reduced result. Note that at each stage, a given device sends and receives a different chunk.

We can start at different chunks, as long as we're consistent with changing the chunks each device processes in a consistent order. Let's first assume that we're zero-indexed, so our devices go from 0 to k-1. To start with, device k sends chunk k to its succeeding neighbor (device k+1), and receives chunk k-1 from its preceding neighbor (device k-1). Note that since we have a ring, so even device 0 has a left neighor, we do a "wrap-around." That is, device 0 receives the data from device K-1 from chunk K-1 (since on the first round, device K-1 would be sending chunk K-1. In Python it's especially natural to do this, since -1 actually indexes into the last element of the array (K-1 since Python is zero-indexed), -2 indexes into the index before the last and so on. Since we want to identify real devices, let's just do modulo K, which will convert the negative device and chunk IDs into values from 0 to K-1.

Here's some simple Python code to show you how it works. I'm only using NumPy here to pretty-print the 2D array. The rows of the 2D array represent devices, and columns represent chunks.

```
import numpy as np

def reduce_scatter(chunks):
    gpus = len(chunks)
    for step in range(gpus - 1):
        print("step {}".format(step))
        for gpu in range(gpus):
            idx_to_send = (gpu - step) % gpus
            prev_idx_to_send = (gpu - 1 - step) % gpus
            idx_to_recv = (idx_to_send - 1) % gpus
            print("gpu {} sends chunk {}, receives chunk {}".format(gpu, idx_to_send, idx_to
_recv))
            chunks[gpu][idx_to_recv] += chunks[(gpu - 1) % gpus][prev_idx_to_send]
        print("\nCurrent state:")
        print(np.array(chunks), "\n")
    return chunks
```

Let's now run the above code to see what's going on. Let's first examine some data we're going to use:

```
data = [
    ['a0', 'b0', 'c0', 'd0'],
    ['a1', 'b1', 'c1', 'd1'],
    ['a2', 'b2', 'c2', 'd2'],
    ['a3', 'b3', 'c3', 'd3'],
]
```

As I mentioned, rows represent devices and columns represent chunks. Hence, the first device has an appended 0 to all the chunk names, and all devices have chunks a through d.

In our case, allreduce is about summing up the values. Since it might be difficult to track the summing for rumerical data, here I chose text data, and the + operator will simply concatenate the text. This way, we'll know that the longest text in the column is the most recent result.

```
REDUCE-SCATTER

step 0
gpu 0 sends chunk 0, receives chunk 3
gpu 1 sends chunk 1, receives chunk 0
gpu 2 sends chunk 2, receives chunk 1
gpu 3 sends chunk 3, receives chunk 2

Current state:
[['a0' 'b0' 'c0' 'd0d3']
 ['a1a0' 'b1' 'c1' 'd1']
 ['a2' 'b2b1' 'c2' 'd2']
 ['a3' 'b3' 'c3c2' 'd3']]

step 1
gpu 0 sends chunk 3, receives chunk 2
gpu 1 sends chunk 0, receives chunk 3
gpu 2 sends chunk 1, receives chunk 0
gpu 3 sends chunk 2, receives chunk 1

Current state:
[['a0' 'b0' 'c0c3c2' 'd0d3']
 ['a1a0' 'b1' 'c1' 'd1d0d3']
 ['a2a1a0' 'b2b1' 'c2' 'd2']
 ['a3' 'b3b2b1' 'c3c2' 'd3']]

step 2
gpu 0 sends chunk 2, receives chunk 1
gpu 1 sends chunk 3, receives chunk 2
gpu 2 sends chunk 0, receives chunk 3
gpu 3 sends chunk 1, receives chunk 0

Current state:
[['a0' 'b0b3b2b1' 'c0c3c2' 'd0d3']
 ['a1a0' 'b1' 'c1c0c3c2' 'd1d0d3']
 ['a2a1a0' 'b2b1' 'c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b3b2b1' 'c3c2' 'd3']]
```

AllGather

Allgather is even simpler than reduce-scatter. It simply means that a given device passes the completely reduced chunk (the one that was computed at the last iteration of reduce-scatter) to the next device. On the next iteration, that device passes the same chunk to the following device. Allgather simply copies the data for a given chunk across all devices - there's no reduction or any other kind of computation.

Given the arrangement that we made for reduce-scatter (first having device k send chunk k and receive chunk k-1 with wrap-around), we'll have the "most reduced" chunk for device k in chunk k+1 (with wrap-around, i.e. for device K-1 it'll be chunk 0). Thus, at the first iteration, device k sends chunk k+1 to device k+1 and receives chunk k from device k-1. Next, each device sends the chunk it received, e.g. device k will send chunk k to device k+1 and receive chunk k-1 from device k-1, and so on.

```
def allgather(chunks):
    gpus = len(chunks)
    for step in range(gpus - 1):
        print("step {}".format(step))
        for gpu in range(gpus):
            idx_to_send = (gpu + 1 - step) % gpus
            prev_idx_to_send = (gpu - step) % gpus
            idx_to_recv = (gpu - step) % gpus
            chunks[gpu][idx_to_recv] = chunks[(gpu - 1) % gpus][prev_idx_to_send]
            print("gpu {} sends chunk {}, receives chunk {}".format(gpu, idx_to_send, idx_to
_recv))
        print("\nCurrent state:")
        print(np.array(chunks), "\n")
    return chunks
```

As a reminder, here's the output of reduce-scatter from before:

```
[['a0' 'b0b3b2b1' 'c0c3c2' 'd0d3']
 ['a1a0' 'b1' 'c1c0c3c2' 'd1d0d3']
 ['a2a1a0' 'b2b1' 'c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b3b2b1' 'c3c2' 'd3']]
```

Let's now run allgather:

```
step 0
gpu 0 sends chunk 1, receives chunk 0
gpu 1 sends chunk 2, receives chunk 1
gpu 2 sends chunk 3, receives chunk 2
gpu 3 sends chunk 0, receives chunk 3

Current state:
[['a3a2a1a0' 'b0b3b2b1' 'c0c3c2' 'd0d3']
 ['a1a0' 'b0b3b2b1' 'c1c0c3c2' 'd1d0d3']
 ['a2a1a0' 'b2b1' 'c1c0c3c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b3b2b1' 'c3c2' 'd2d1d0d3']]

step 1
gpu 0 sends chunk 0, receives chunk 3
gpu 1 sends chunk 1, receives chunk 0
gpu 2 sends chunk 2, receives chunk 1
gpu 3 sends chunk 3, receives chunk 2

Current state:
[['a3a2a1a0' 'b0b3b2b1' 'c0c3c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b0b3b2b1' 'c1c0c3c2' 'd1d0d3']
 ['a2a1a0' 'b0b3b2b1' 'c1c0c3c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b3b2b1' 'c1c0c3c2' 'd2d1d0d3']]

step 2
gpu 0 sends chunk 3, receives chunk 2
gpu 1 sends chunk 0, receives chunk 3
gpu 2 sends chunk 1, receives chunk 0
gpu 3 sends chunk 2, receives chunk 1

Current state:
[['a3a2a1a0' 'b0b3b2b1' 'c1c0c3c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b0b3b2b1' 'c1c0c3c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b0b3b2b1' 'c1c0c3c2' 'd2d1d0d3']
 ['a3a2a1a0' 'b0b3b2b1' 'c1c0c3c2' 'd2d1d0d3']]
```

We can clearly see that each device (row) has all the chunks (columns) for a-d concatenated, and that the chunks (columns) have been copied to all devices (rows). We are done.

Tree allreduce

Ring allreduce is bandwidth-optimal, but it's far from being latency optimal. The reduce-scatter and allgather together require 2*(N-1) = 2N - 2 steps. As we add GPUs or nodes to our cluster, note that the latency scales linearly with the number of participating GPUs or nodes. This clearly limits scalability. An alternative approach is to use a binary tree. The tree approach does two passes through the tree - one reduce and one broadcast.

If we invert the tree so that the root is at the bottom, we can start by reducing the values from two leaf nodes into each of the corresponding parents. Then two of those nodes feed into their parent, and so on. We end up with $\log_2(\mathrm{N})$ steps, where N is the number of nodes. The final reduction is performed by the root node.

Next, we broadcast the reduced result from the root to the root's 2 children. Those children in turn broadcast to their 2 children, and so on. We end up with $2*\log_2(N)$ steps, which is way more efficient than ring allreduce. However, the problem is that we're wasting half the bandwidth.

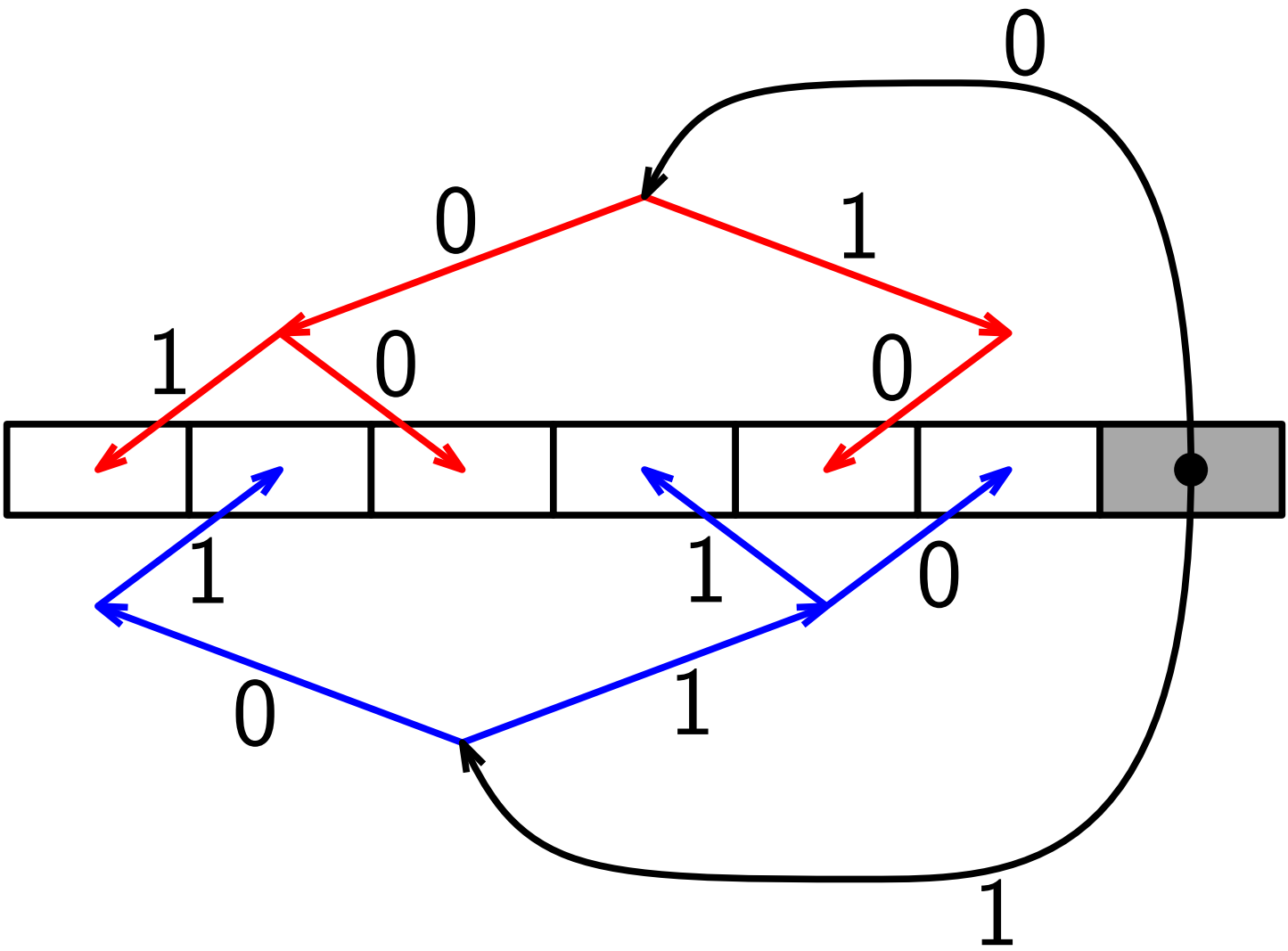
Note that in a full, complete binary tree, there is as many nodes (plus one) in the leaves as in the non-leaf nodes. When we do the reduction, all the leaves are sending data, but they aren't receiving anything, so we have half of duplex bandwidth wasted by the leaves. In case of the broadcast, we have the opposite problem - the leaves are receiving, but not sending. Also, in the simplest scenario, the parents can only receive and then send. The parent issue can be resolved by breaking up a bigger message into chunks, and form a pipeline. This way at each cycle, each non-leaf node can send a message to its parent for step k-1 and receive for step k. While pipelining optimizes non-leaf bandwidth use, it still doesn't address that half the bandwidth is wasted by the fact that leaves aren't receiving anything. Can we break up the problem in a way such that all nodes both send and receive?

Two-tree allreduce

Let's use the logic of breaking up allreduce for a binary tree, i.e. a reduce step, followed by a broadcast. As explained above, a single binary tree, even a pipelined one, wastes half the bandwidth during the step involving the leaves, since they're either sending (reduce) or receiving (broadcast), but not doing both. What if we used two trees?

Sanders, Traff and Larsson (2009) (https://linkinghub.elsevier.com/retrieve/pii/S0167819109000957) determined how two trees can both keep the latency logarithmic with the number of nodes, and use all available bandwidth. How can we achieve that? The idea is that we can arrange nodes in one tree so that its leaves are the other tree's interior nodes and vice versa. Note that to achieve this, we will need to have a complete, but not a full, tree. A full tree is one where each node other than leaf nodes has both children, while a complete tree has to have all children at each level except for the last, and at the last level, all the "leftmost" nodes need to exist. So for instance, a complete tree may not have the rightmost child of the rightmost parent from the previous level.

How can we map one tree so that its interior nodes are the other tree's leaves and vice versa? Have a look at the following illustration (courtesy of Wikipedia).



The blue tree is exactly the same as the red tree, but it's shifted to the left by one position. If we zero-index and count from the left, node 2 in the red tree, which is a leaf, is the blue tree's interior node 2 levels from the leaf level. Why did we superimpose the trees like that? The idea is that if we for instance take the broadcast operation, one tree's root node will send data down to the leaves, but the other tree's root is located at the leaf level of the first tree. This means that if we split the message in half and send one half to the root of the first tree and then send the other half to the second tree's root, at all steps except for the root step, the entire bandwidth will be utilized, as each node can both send and receive. While we only had the leaf nodes receiving during broadcast and sending during reduce, now each node is a leaf node in one tree and a non-leaf node in the other tree, so all nodes can both send and receive. We achieved full bandwidth at logarithmic latency, rather than linear latency (with respect to the number of GPUs or nodes), as in case of the ring allreduce. Note that this algorithm is fairly new - it was discovered in 2009, after decades of parallel and distributed computing.

The performance improvements of the two-tree reduce compared to ring all-reduce are amazing. The NVIDIA NCCL (https://developer.nvidia.com/nccl) 2.4 blog post (https://devblogs.nvidia.com/massively-scale-deep-learning-training-nccl-2-4/) explores performance scenarios on the Summit (https://en.wikipedia.org/wiki/Summit_(supercomputer)) supercomputer.

Final thoughts

In general, communications collectives libraries such as various implementations of MPI (https://en.wikipedia.org/wiki/Message_Passing_Interface) or NVIDIA's NCCL (https://developer.nvidia.com/nccl) tend to have multiple algorithms to run a particular collective, such as allreduce, allgather, all-to-all, etc. Depending on the topology, link latency, bandwidth and the size of the data to be communicated, different algorithms may be preferable.

Published

Aug 15, 2019

Category

HPC (/categories.html#hpc-ref)

Tags

- allreduce 1 (/tags#allreduce-ref)
- bandwith optimal 1 (/tags#bandwith-optimal-ref)
- collective 1 (/tags#collective-ref)
- communication 1 (/tags#communication-ref)
- communication collective 1 (/tags#communication-collective-ref)
- MPI 1 (/tags#mpi-ref)
- NCCL 1 (/tags#nccl-ref)
- ring allreduce 1 (/tags#ring-allreduce-ref)

Contact