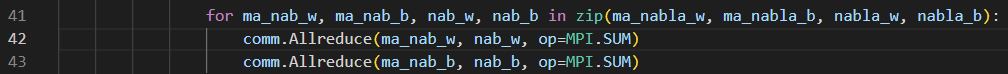
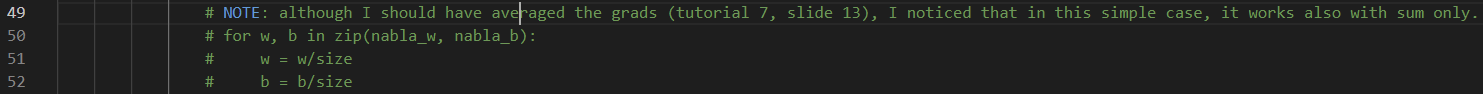
# Concurrent and Distributed Prog. – 236370

## Assignment 3 – MPI Collective Comm & Async Comm

## Due Date: 19/01/23

1. Part One – Synchronous – MPI Collective Comm:

* I used MPI AllReduce with ‘+’ binary operator for the collaborative communication.
* Since this method works on NumPy arrays (I didn’t find an easy way to serialize a list of arrays as a pickle), I used a for loop over the arrays, where in each iteration one layer is being sent and received:  
  
* In the original code template, each process receiving buffers are being allocated and re-allocated in every iteration of the training. Which is highly inefficient.  
  So, I moved the declaration outside the minibatch for loop.   
  In that way, the buffers are being allocated once, and new data is written over old data:  
  
* Following Tutorial 7, slide 13, there should be a ‘averaging’ step before updating the weights. I found that this step is not necessary (from linearity of the network & loss function) and obviously takes time, so I put it (for the sake of completeness) but commented it out:  
  
* Lastly, there is not much sense to use data-parallelism training, without splitting the data… You didn’t ask us to do so, so each process is working on the same amount of data as the single-process training version.  
  Since the total amount of data is not equal to the size of the dataset (632 batches 5 epochs), while all processes maintains a coherence image of the network (synchronous training) it leads to more data presented to the model and should bring more accuracy, in a price of higher execution time.

Part One – Synchronous – My Naïve AllReduce:  
A screenshot of a computer

Description automatically generated with medium confidence

* I Implemented it as naïve as possible, but with Async Comm routines for better performance (still P2P of course).
* Each process sends his array to all other process.
* Then it receives all other processes arrays.
* Lastly it applies the reduce operator on all arrays (his own + all others).
* No need to wait after Isend, but it has to wait for all other array to arrive (Wait() after all Irecv calls).

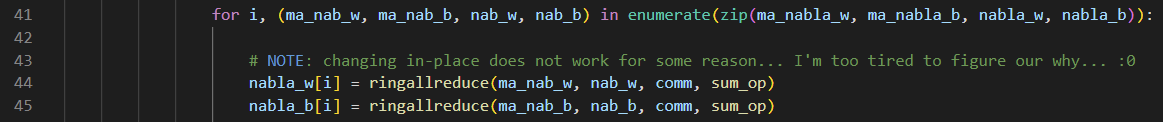
Part One – Synchronous – My Ring AllReduce:

Text

Description automatically generated

* Each process slices his array into <size> chunks, and allocated <size> buffers for the receiving data:  
  
* It then runs 2<size> iterations, first <size> times to reduce all receiving slices with his own **specific index** slice
* It then sends one slice in each consecutive iteration.

Part One – Synchronous – sync\_network.py:



* Just as the collaborative comm version, I’m calling My Ring AllReduce in a loop for all arrays.

Part Two – Asynchronous:

* Data-Parallelism-Model Parallelism hybrid method (very nice – like DLRM architecture):

Layers are assigned to master statically before training in a round-robin fashion: master 0: layers 0, 2, master 1: layers 1, 3 et cetera. This is MP.  
Data is divided between processes; each process works on 1/<workers> of the batches. This is DP.

* So, each worker iterates through his view of layers grads and send to the right master.
* I used Isend grads then Irecv weights for the workers, and Irecv grads then Isend weights for the masters.  
  Text

  Description automatically generated  
  Text

  Description automatically generated  
  Text

  Description automatically generated
* Since each master works on a subset of layers but need to update his copy of parameters according to all workers, it iterates through the original number of batches (632 in our case) and receives grads for all layers from each worker in a round-robin fashion.
* Since Isend is asynchronous method, we don’t need to wait after it.
* We do need to wait after recv, because we must be sure all the data arrived before using it.  
  But, since we receive the data into different buffers, we can first issue all recv calls, and then wait on all of them in a loop. It should d be more efficient following the tip from the tutorial to issue asynchronous calls ‘As Soon As Possible’.
* I check the previous hypothesis vs. wait calls attached to their Irecv calls, and it indeed was slightly faster.
* Note: later you’ll see that I encountered some non-deterministic corrupted data problems.  
  I debugged it thoroughly and concluded that it does not stem from the wait calls positioning.  
  After consultation with Daniel (the TA), we still don’t know what causes this problem, but he said that it is a known problem in this exercise, and we shouldn’t be charged for it.  
  Observation: I noticed that it happens more frequently proportionally with the rank (or the number of workers). I think it is due to some problem in the mpi4py backend installation in the server.  
  Moreover, it happens for non-power-of-two #workers.
* I also commented out the print\_progress call. Since each master has partial view of the current model (master 0 has only the updated versions of layers 0 & 2), there’s no point to evaluate in the master during training. The accuracy will always be minimal.

1. Text

   Description automatically generated  
   Text

   Description automatically generated  
   Text

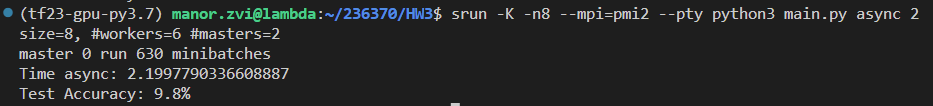
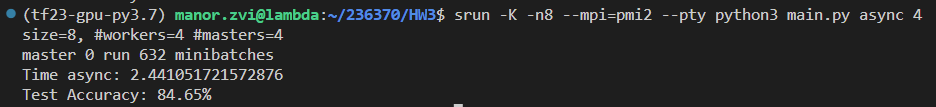
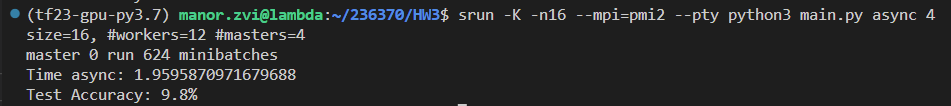
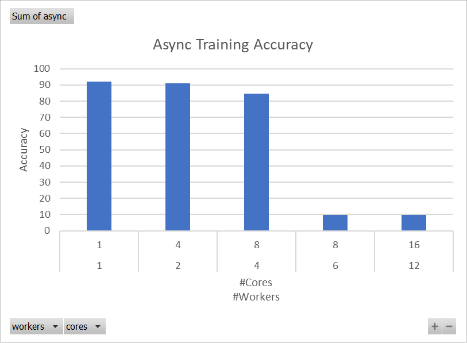
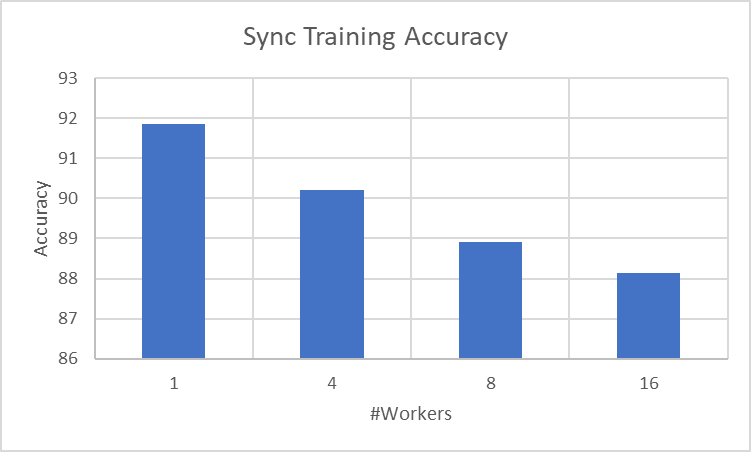
   Description automatically generated

Text, application

Description automatically generated  
Text, application

Description automatically generated

1. While the change in accuracy is negligible (and have changed during various tests that I ran), for higher ranks we see higher execution time, which makes sense, because the Ring All Reduce need to distribute the data to more processes in each iteration.
2. This is Gustafson (weak scaling). It means that with more resources we need to increase the problem size. We do increase the problem here, by not slicing the batch among processes, but use the same number of batches (632) for all processes. For example, for rank 4, we increase the problem by 4, but the time is increased by 1.87 (12.93/1.87), therefore we get weak scaling speedup of ~2.13 (4/1.87).  
   For Amdahl, we need to keep the problem the same size, hence slice the number of batches by the number of processes, in that way each process works on 1/<processes> of the batches. This called Strong Scaling (and it is quite rare nowadays… HaHa)
3. Graphical user interface, text, application

   Description automatically generated   
     
     
   
4. For 8 cores, 2 masters (6 workers) and for 16 cores, 4 masters (12 workers), you can see the phenomena I described in section 1.  
   For some reason, in some non-deterministic point during the training, and gradients for one of the workers become nan. For that point the way is short for infection of all masters (because of backpropagation) and all other workers. I discussed with Daniel about it, and he said that we won’t be charged for it.  
   for 4 cores, 2 masters & 8 cores, 4 masters we see that the network succeeded to converge. But due to Gradient Staleness, the accuracy is decreasing with the number of workers. It happens because the number – the number of updates the master receives from other workers between sending the most recent weights to a particular worker – grows.   
   From execution time perspective, we are at strong scaling here (Amdahl), so we have sub-linear speedup (for example, for 2 masters, 3.69/2.19=1.68 speedup while we increased the number of workers by 3 from 2 to 6). Obviously, the speedup is sub-linear and not linear because of the cost of inter-process communication.
5. Splitting the master across different machines allows us to speedup the update phase of training. Moreover, it allows us to reduce the risk of recv bottleneck, when all workers sends grads to the same machine.
6. 
7. Not sure if you refer to the Staleness Gradients or the NaN problem. If you meant Gradient Staleness, I explained it already, otherwise, I don’t have explanation for it. Gradient Staleness can explain the lower accuracy for 8 cores, 4 masters (compared to 4 cores 2 masters), but can’t explain the 9.8% accuracy for 8 cores, 6 workers or 16 cores, 12 workers.
8. Sync approach is more scalable. We can apply it to achieve Strong Scaling almost always, but its potential speedup is limited due to quadratic (in most cases) penalty due to the communication cost. Also, it is much easier for the programmer.  
   Async approach is much more difficult for the programmer, and very error prone. But given a verified implementation it is more capable for lower ranks. For high ranks it suffers from Gradient Staleness (could be mitigated by forcing flush operation occasionally, with a cost of ‘less synchronicity’).
9. Text

   Description automatically generated Text

   Description automatically generated  
   Text

   Description automatically generated Text

   Description automatically generated
10. Assume data of size X. Each process sends data (to all other processes) and receive data (from all other processes). So, the complexity in total is:
11. Assume data of size X. Each process sends in each iteration chunk of data. it does that   
     iterations, and then iterations, so in total it sends: . Equivalently it receives data, so the total comm complexity is: . MUCH MUCH MUCH more scalable than the Naïve version for high ranks.