CSCI-GA.3033-023 – Lab3

This lab is intended to be performed **individually**, great care will be taken in verifying that students are authors of their own submission. Coding exercises are identified by **C<number>**.

C1 – Tensor Arithmetic Mean with MPI – Non-blocking send/recv version

In this part of the lab we implement the distributed arithmetic mean of a tensor using the MPI primitives in C. Consider the following:

- There is a total of R ranks. Rank zero ($r = 0$) is the root and ranks with $r \in [1, R-1]$ are workers.
- Each worker generates an input tensor I_r with dimensions: C*, H, W.* Each element of *I* is generated as follows:

$$I_r[c, x, y] = r + c \cdot (x + y)$$

- Where $x \in [0, W-1]$, $y \in [0, H-1]$ and c $\in [0, C-1]$
- Dimensions are: H=1024, W=1024, C=3
- All tensors have *double* elements (double precision)
- Workers with $r \in [1, R-1]$ communicate I_r to rank zero *(r=0)*. Rank 0 computes the output tensor *O* with dimensions C, H, W as follows:

$$O[c, i, j] = \frac{1}{R-1} \sum_{r=1}^{R-1} I_r[c, i, j]$$

- Use *MPI_Isend() MPI_Irecv()* (with associated *MPI_wait()*)
- Execute CPU-only with 4, 8 and 16 workers, adding rank 0 as the root (ex. with 4 workers you will have 5 ranks)
- Measure the time to compute O without including the time to allocate/free memory and the time to generate the input tensors, but including the communication time. Use an MPI barrier to synchronize the ranks before measuring the time.

Output format (use *printf()* with %4.3lf in milliseconds) for each execution (5, 9 or 17 ranks):

C1_checksum, C1_exec_time

C2 – Tensor Arithmetic Mean with MPI – All reduce version

- Implement a distributed arithmetic mean as explained in C1 but this time every rank will calculate *O*
- Use an MPI All Reduce primitive instead of Isend/Irecv
- Implement C2 as a different program from C1 (two different binaries)
- Execute using with 4, 8 and 16 ranks (workers). There is no root rank.
- Measure the time to compute O without including the time to allocate/free memory and the time to generate the input tensors, but including the communication time. Use an MPI barrier to synchronize the ranks before measuring the time.

Output format (use *printf()* with %4.3lf in milliseconds):

C2_checksum, C2_exec_time

C3 – DDL with PyTorch – Synchronous Decentralized

In this part of the lab we create a distributed neural network in PyTorch to classify a dataset of images. The dataset is stored in the folder */scratch/am9031/CSCI-GA.3033-023/kaggleamazon/* and is composed by a training and a validation set of respectively 20000 and 1479 images, classified in 17 labels. In the files *train.csv* and *test.csv* you will find the filename of each image and the related label index. All the images are physically stored in the sub-folder *train-jpg/*

Create a PyTorch program with a DataLoader that loads the images and the related labels from the filesystem. During the creation of the dataset, make a two steps data-augmentation (pre-processing), using the *torchvision* package, with the following sequence of transformations:

1. Resize the images to squares of size 32x32 (*transforms.Resize()*)
2. Transform the images to tensor (*transform.ToTensor())*

Divide the training data into different subsets for each worker. I.e. modify the dataloader to load different portion of the data, depending on the rank of the worker. Each worker has a minibatch of 100. Do not confuse the "application workers" (ranks) with the "DataLoader workers". There is only one DataLoader worker for each DataLoader so *num_workers=1*.

Create a Neural Network composed by 3 fully connected layers:

1. The first layer has an input size equal to 32x32X3=3072 and an output size of 1024
2. The output of second layer is 256
3. The output of the third layer must be the number of the labels, ie 17

Define the forward function of the NN with the three created layers, using a ReLU activation function for the first two layers. The minibatches of images will create a tensor of shape (mini_batch_size, 3, 32, 32), where minibatch size is 100. Reshape the input Variable to be (mini_batch_size, 3072), using the *view()* function. Use as optimizer an SGD algorithm with learning rate 0.01 and momentum 0.9 and a *nn.CrossEntropyLoss()* as loss computation criterion.

Create a main function that creates the DataLoaders for the training set. Then do a cycle of 5 epochs with a complete training phase on all the minibatches of the training set. During an epoch, for each minibatch each rank does the following:

1. Perform forward and backward propagation, compute gradients
2. Exchange gradients among the ranks with an all reduce operation
3. Compute the average of the gradients
4. Update the model

- Do an all-reduce between the workers to compute the weighted average loss of the last epoch (*C3_loss*):

$$L = \frac{1}{\sum_{w=0}^{W-1} d_w} \sum_{w=0}^{W-1} d_w \times l_w$$

  - Where $l_w$is the average loss of the mini-batches of a worker w
  - $d_w$is the number of samples viewed by the worker w during the training phase (that is necessary because the workers can have a different number of total training samples)
- Execute CPU-only using 4, 8 and 16 ranks
- A single rank prints the average epoch time over 5 epochs as follows:
  - *C3_loss, C3_exec_time*

C4 – DDL with PyTorch – Asynchronous Centralized

Modify the code in C2 to implement the Asynchronous Centralized distributed SGD. Define a parameter server using the rank 0 that will always maintain the updated version of the model's parameters.

The server/workers communication must be implemented by dist.send() and dist.recv(). Following this approach:

1. Divide the training data into different subsets for each worker. I.e. modify the dataloader to load different portion of the data, depending on the rank of the worker. Each worker has a minibatch of 100.
2. After each mini-batch, each worker will send to the parameter server the accumulated loss after the backward
3. Then the parameter server will perform the actual update of the parameters (optimizer.step()) and send back to the worker the updated model parameters
4. After each epoch, the workers synchronize with a barrier and receive the latest updated model from the parameter server (workers can synchronize using `torch.distributed.new_group`, that doesn't include the parameter server)

- Consider that all the parameters of the model are stored by layer into *model.parameters()*, So you will need a for loop like "*for param in model.parameters()"* and you will find the model parameters in *param.data* and the gradient information in *param.grad.data*.
- You'll need to initialize the gradient data of the parameter server as zeros tensor to avoid errors in the first receive from the workers.
- Hint: one way of implementing the server loop is as follows:
    - Receive a message from any sender with a *tag*, and save the sender rank returned by the recv call in the variable *src*
        - If the *tag* is 0
            - the sending worker is completing
        - otherwise:
            - receive the gradient of the sending worker with a rcv from *src*
            - send back the model to the worker with a send to *src*

- You'll need a barrier to update the last updated model from the server at the beginning of the first training and at the end of each training epoch, to perform the evaluation with the same updated model for all the workers
    - To update the model: the workers can send a gradient=0, so the server will not update the model and send back the last updated version of the model parameters.
- Create a group and do an all-reduce between the workers to compute the weighted average loss of the last epoch (*C4_loss*).

$$L = \frac{1}{\sum_{w=0}^{W-1} d_w} \sum_{w=0}^{W-1} d_w \times l_w$$

    - Where $l_w$is the average loss of the mini-batches of a worker w
    - $d_w$is the number of samples viewed by the worker w during the training phase (that is necessary because the workers can have a different number of total training samples)

- Execute CPU-only with 4, 8 and 16 workers, adding 1 rank to be the parameter server (ex. with 4 workers you will have 5 ranks).

- A single rank prints the average epoch time over 5 epochs as follows:
  C4_loss, C4_exec_time,

Running and Submission instructions

Running instructions:

- We are using a special version of MPI and python that are in /home/am9031/anaconda3
- For timing:
  - Make a copy of */scratch/am9031/CSCI-GA.3033-023/lab3* in your home and give rw permission to your ID for all the files
  - Compile with the *Makefile* provided
  - Execute the *run_job.sh* with the desired number of ranks*:*

  ./run_job.sh <5 or 9 or 17> (ranks are automatically reduced by one for C2 and C3)

- For testing purpose only (not for timing) you can use an interactive job:
  - Use the command *srun -N <ranks> --pty bash*
  - Make sure to use *mpirun* and *python* from */home/am9031/anaconda3/bin*
- File naming:
  - Please submit a zip file with file name *your-netID*.zip with a folder named as your netID (example: am9031.zip containing am9031/). Your folder will contain these files:
    - lab3c1.c
    - lab3c2.c
    - lab3c3.py
    - lab3c4.py
  - The .c files will have to compile with the provided *Makefile*
  - The four programs should run using the provided launch job script *run_job.sh*

Submission: through NYU Classes.

Grading

The grade will be a total of 50 points distributed as follows:

EXERCISE	DESCRIPTION	POINTS
C1	MPI Tensor Arithmetic Mean Send/Recv	10
C2	MPI Tensor Arithmetic Mean Reduce	10
C3	PyTorch DDL – Asynchronous Centralized	15
C4	PyTorch DDL – Synchronous Decentralized	15

Other grading rules:

- Late submission is -3 points for every day, maximum 3 days.
- Non-compiling code: 0 points
- Failing to respect output format is -3 points
- Failing to follow the right directory/file name specification is -3 point