UNIVERSITÉ PERPIGNAN VIA DOMITIA

AI Project: Neural network in parallel using MPI

Sulaiman MOHAMMAD
CHPS-master2

Overview

The project targets parallelize the neural network prediction, which is the dot product of the input with the wights of each nodes to have the out of the node.
The problem is the algorithm of parallelizing that process depends on how we structure the network wights.
For that I found that the pest is to write the model by myself, so I can figure out how the model is structure then parallelize the prediction.
In this way you

For that you will see multiple files, some as library you can call them, some as implantation of those files.
1- NNModel:
this file contains all the function , of building the network , then train it using Gradient descent
it contains too functions for two types of pruning, (structured and non structured)

2-NNModelPara
file for parallel predict function (evaluation)

3-implantation :
in this file , we use the functions in sequential mode from NNModel, and plot data

4- implantat_para:
in this file , we use the functions in parallel mode from NNModelPara, and plot data

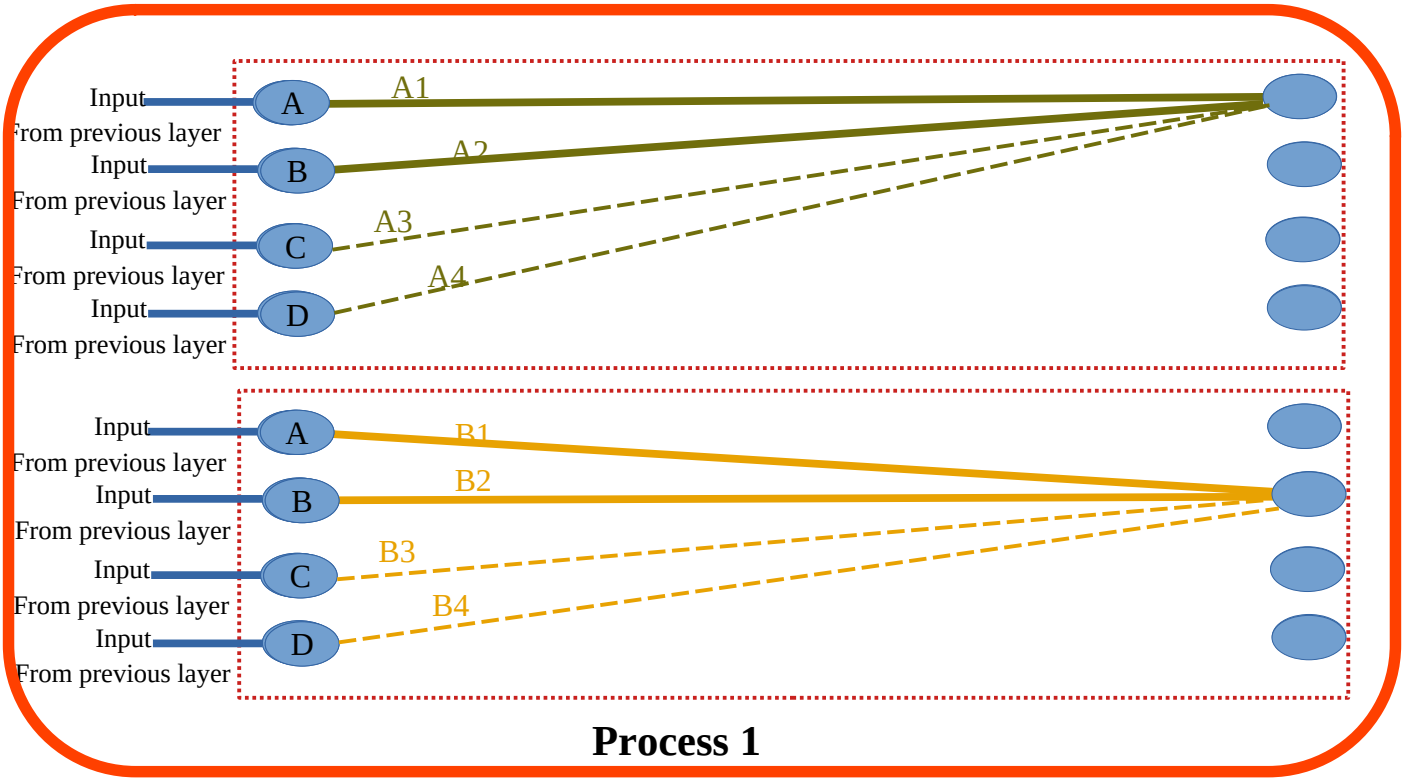
I will skip the details of building the model and train it , because it is not the main discussion here, but I will discuss the details of the parallel algorithm, and there we will see the structure of the model that is built and how we can parallelize it based on the structure we trained.

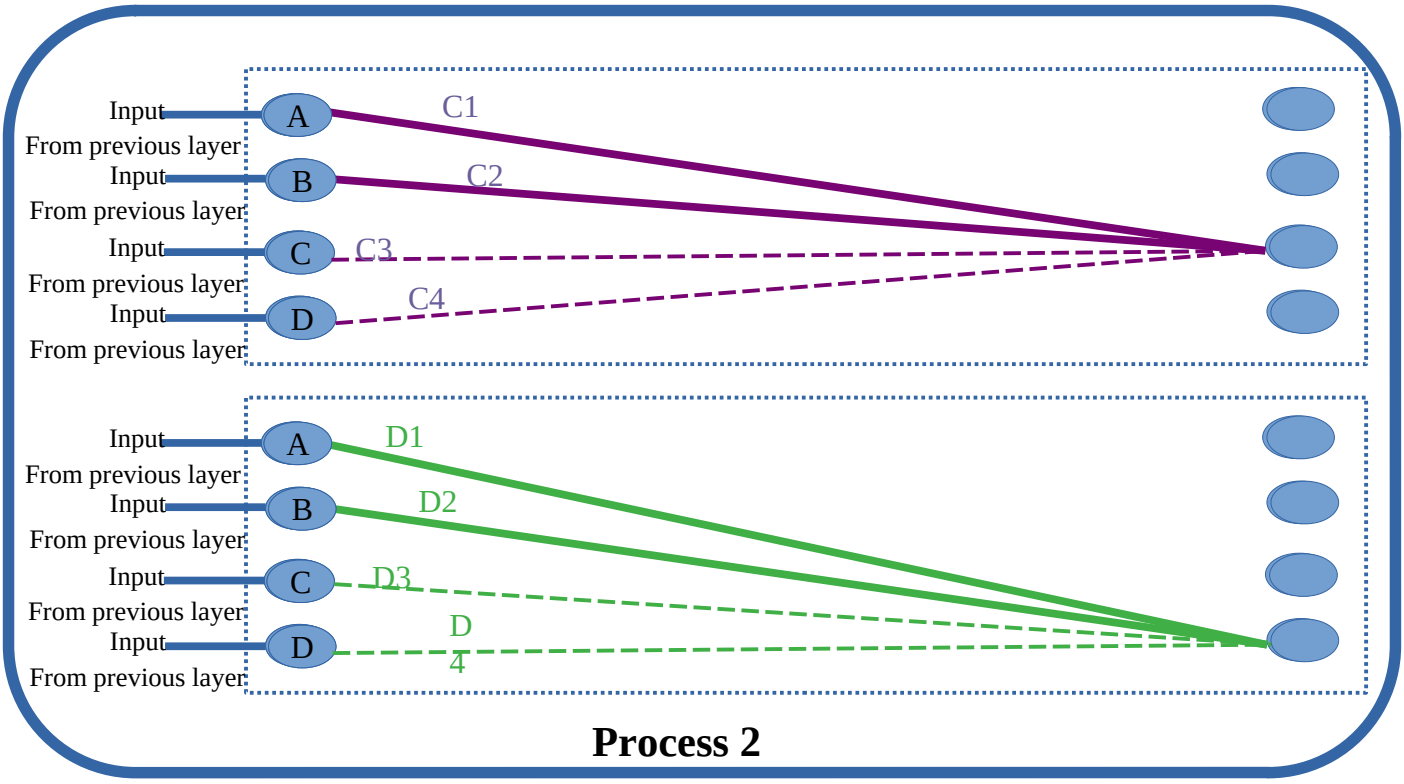
Then I will illustrate the pruning algorithm , because it is important , then analyses the performances

5- tenserflowtest : file to compare the models and the algorithms with tenserflow

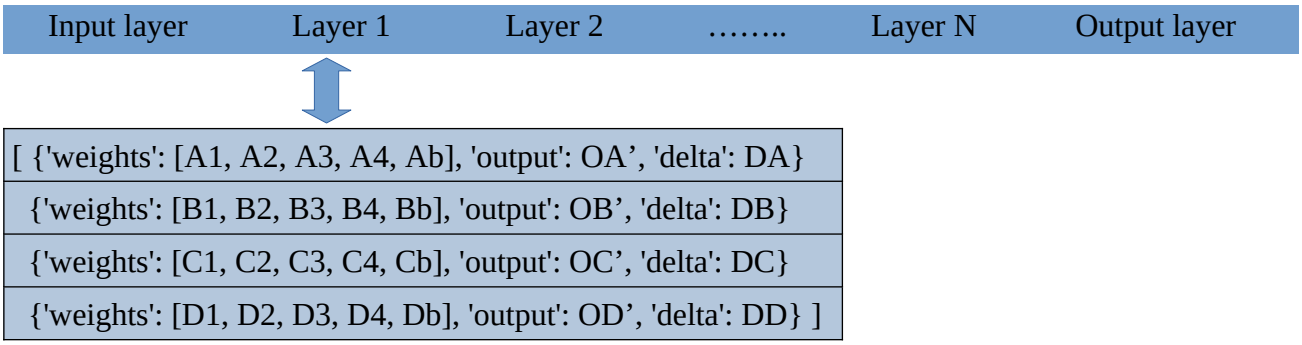
6-networks_build: it is file where I used to build some networks to used them in implantat_para

The parallel algorithm



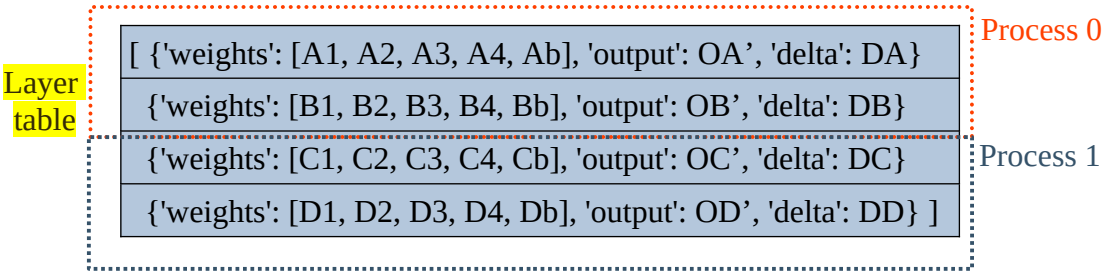


After training network will be in this pattern

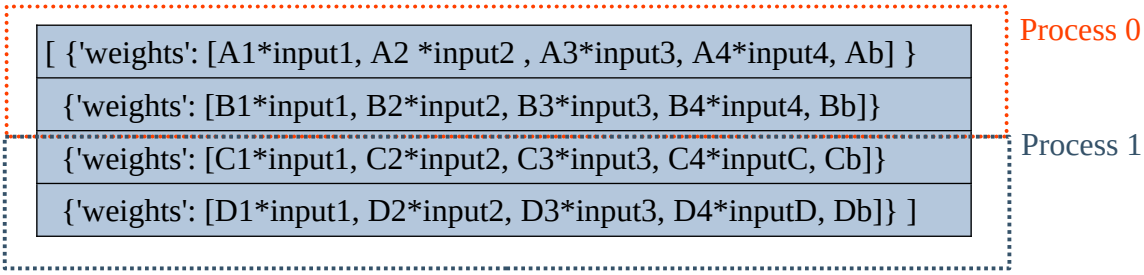


now suppose we have 2 processes , then we need to divide the neurons data to the processes equally
chunk= (number of neurons/ number of processes)
data =process_prod(layer[rank*chunk: (rank*chunk)+chunk], [previous Inputs])

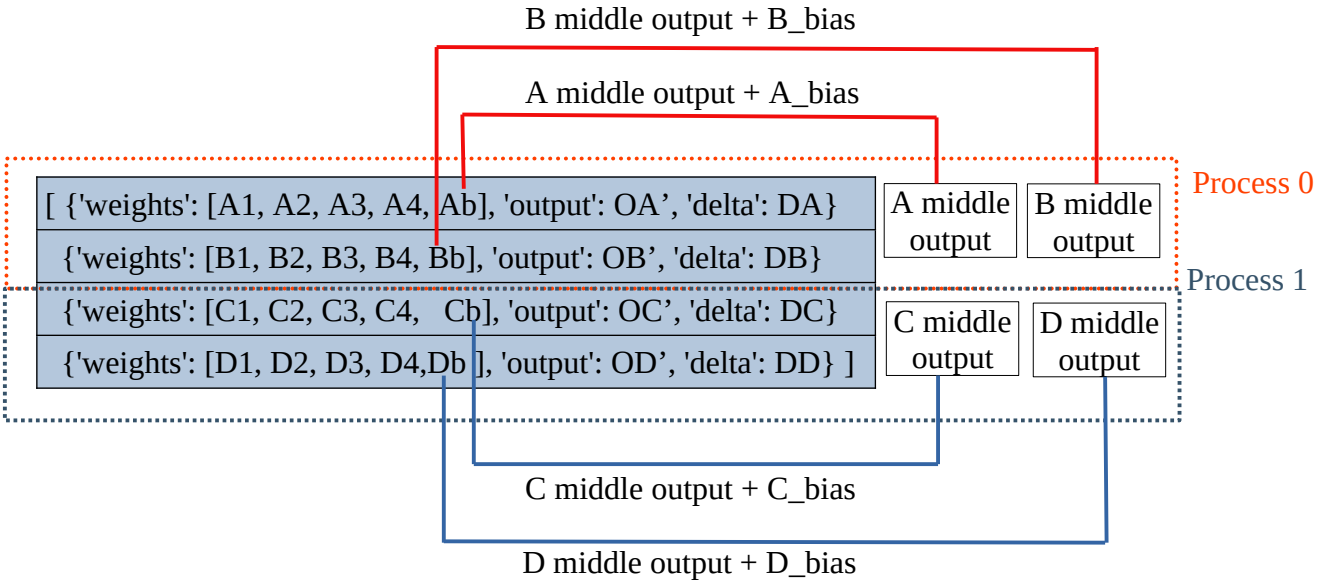
like this for example 4 neurons , 2 processes, then each chunk is 2= 4/2
So for each processes 0~ [A,B] , 1~[C,D]
so processes(0) should move one chunk from zero so (rank=1*chunk) which is step , then we add chunk to arrive to the final neuron should be included.
for processes(1) we moved 2 chunk from 0 so rank*chunk then move one chunk to the final neuron should be included

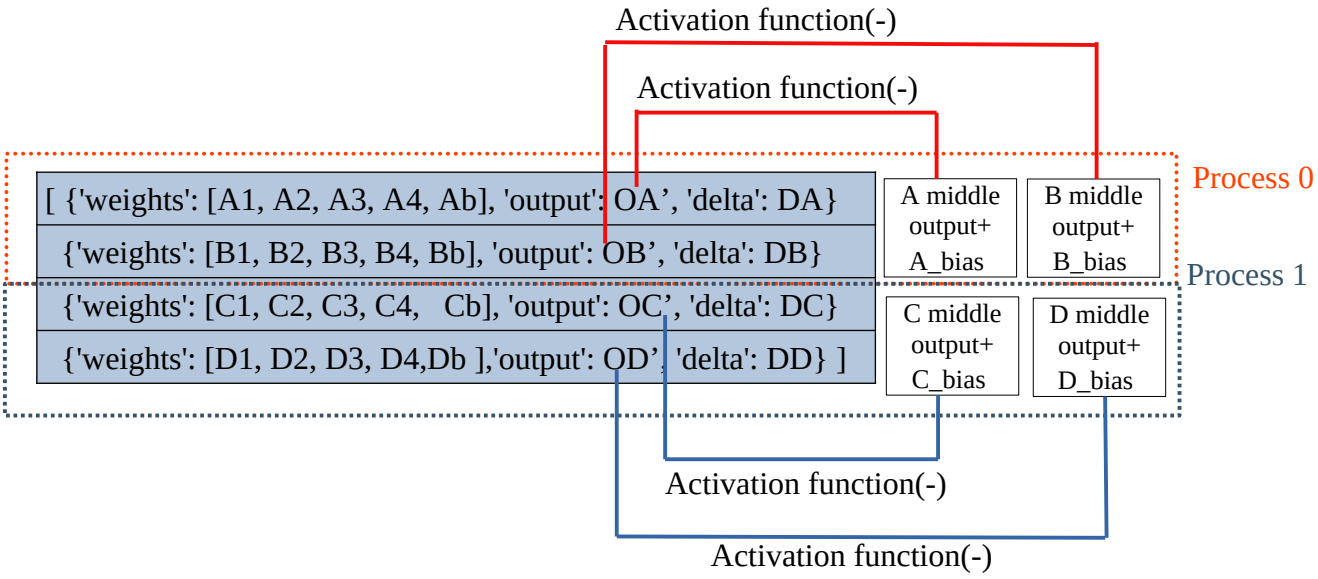


now inside the forward function
for example for a process (0) which will deals with neurons (A,B) and deal just with the key "wights" - for process 0 for example:
1- Take Input of From previous node and multiple that input with all the wights of (A,B) , then we add the bias as it is without multiply it with the input
2-return the result for each neuron but we don't change the original table
and same for the process 1



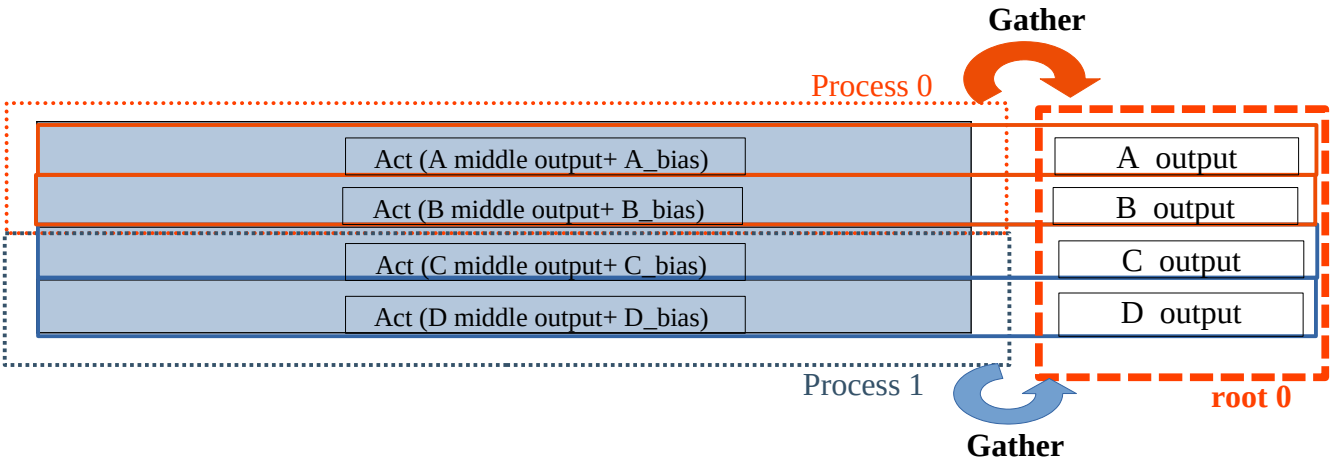
so now each process has this and perform the sum of middle output and the bias then take the activation function and save the output in the “layer table” and this output will be the input for the next stage.



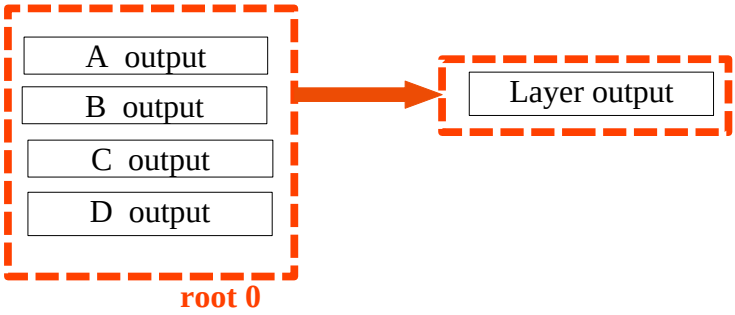


Now we need the output of each neuron to reform them to be the input of the next layer.
In this way, since the example contains 2 process for 4 neurons then we have 2 outputs in each process, to form them ad one object:
Using MPI gather, the outputs of each process are sent to the master process which is the process 0.

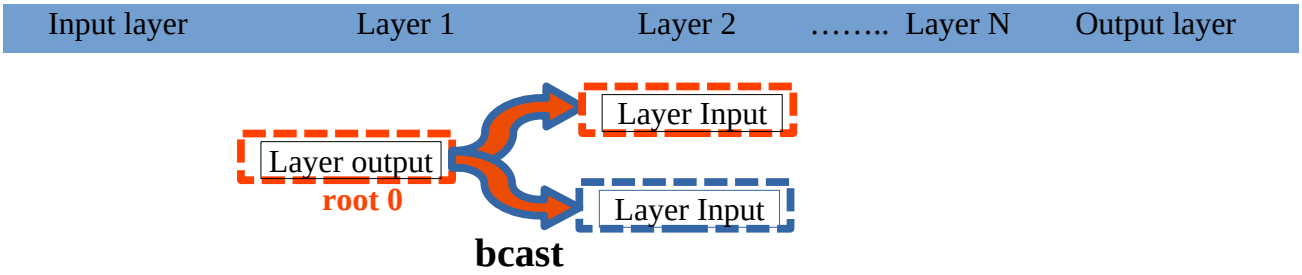
At the end of this instruction, the process 0 contains all the outputs of the layer and will be sen to the next layer

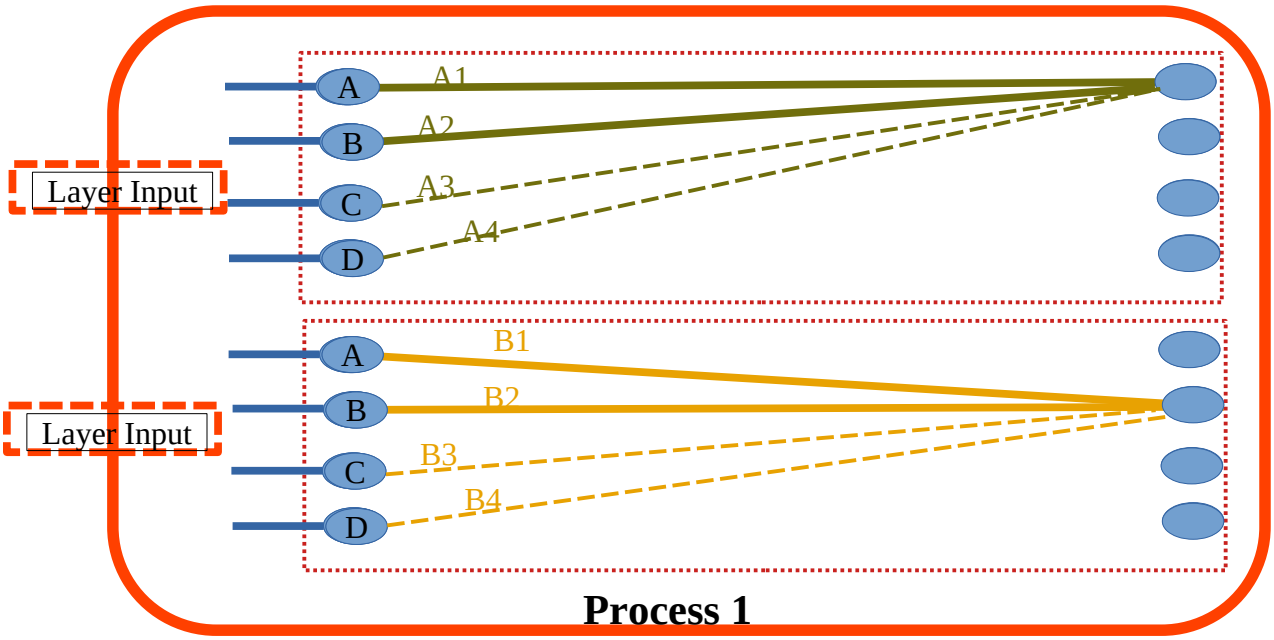


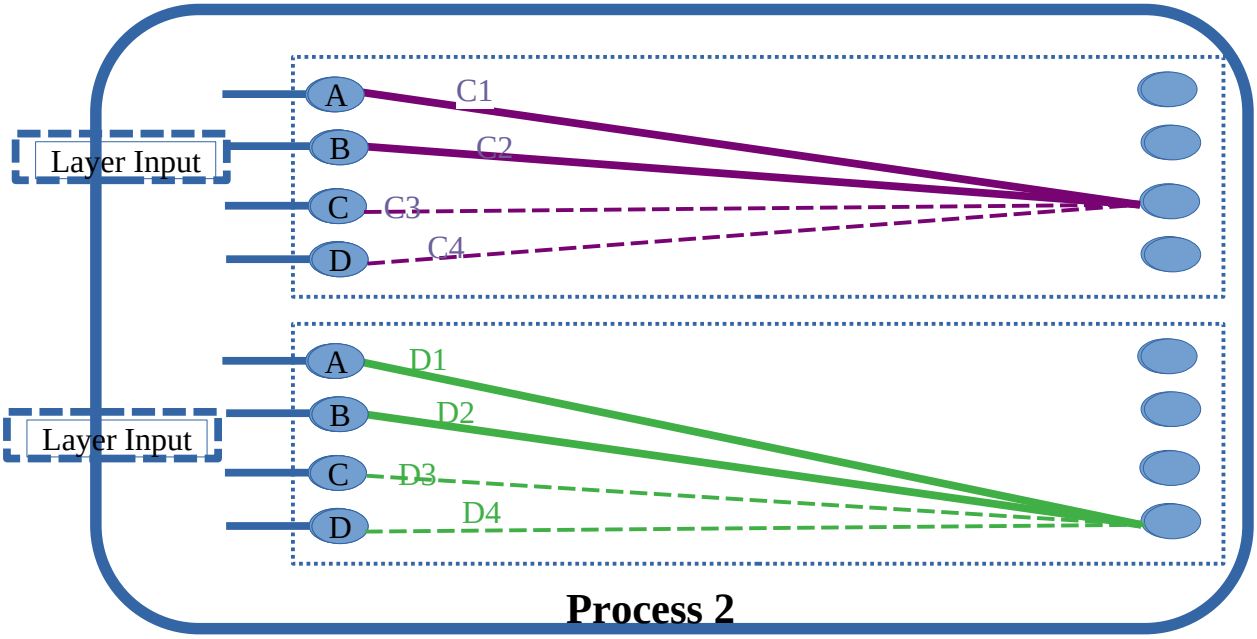
Then this gathered data are reformed in shape of list.



Now we have one object contains all the outputs of the layer we work on , now this output should be the the input for the next layer.
So we use MPI broadcast to send this object to each processes.




Layer Input
A
B
C
D
A1
A2
A3
A4
Layer Input
A
B
C
D
B1
B2
B3
B4
Process 1


Layer Input
A
B
C
D
C1
C2
C3
C4
Layer Input
A
B
C
D
D1
D2
D3
D4
Process 2

Pruning algorithms

1- Unstructured pruning

in the NNModel it is under function name =: pruning_connections
here the number of the connection we need to put it zero equal to number of weights* ratio.
And it will be some weights in each neuron wights

but to keep the accuracy I followed a method can be resumed as:
the weights we need to put it zero should be the smallest weights , so the effect of it will be unnoticeable as possible .

We loop in each layer and each neuron and fin which neuron has the smallest wight, and put it zero.
Notice: you will see in the function that we look for the smallest after taking the abs(weights) because if we didn't the negative wight will be chosen , but we need to search for the smallest in magnitude.

For eaxmple, for ratio 0.5 , it means 2 weights, we search for the smallest

[{'weights': [0, A2, A3, 0, Ab], 'output': OA', 'delta': DA}
{'weights': [B1, B2, B3, B4, Bb], 'output': OB', 'delta': DB}
{'weights': [C1, C2, C3, C4, Cb], 'output': OC', 'delta': DC}
{'weights': [D1, D2, D3, D4, Db], 'output': OD', 'delta': DD}]

2- neuron pruning

in the model is the function: pruning_neuron
here, the same method will be followed , but full layer will be turned to zero, and the way to chose the layer, is based on the smallest magnitude, but here we take the sum of each neurons weights and we chose the smallest and put it zero

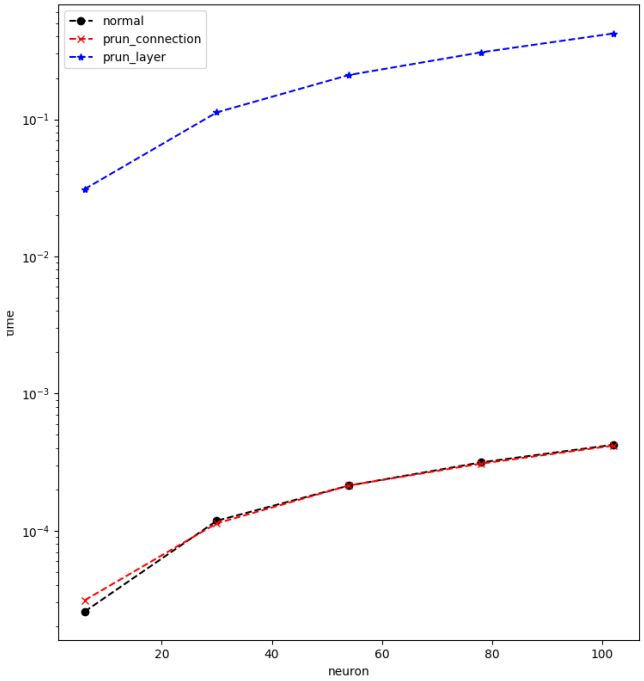
example for ratio 0.5 , it means two neurons

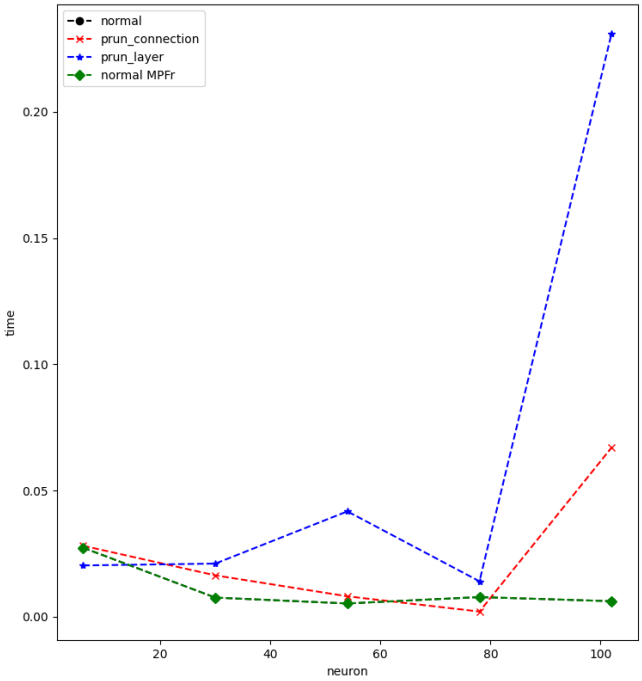
[{'weights': [0,0,0,0 Ab], 'output': OA', 'delta': DA}
{'weights': [B1, B2, B3, B4, Bb], 'output': OB', 'delta': DB}
{'weights': [C1, C2, C3, C4, Cb], 'output': OC', 'delta': DC}
{'weights': [0,0,0,0, Db], 'output': OD', 'delta': DD}]

Analyses the performances

1- Sequential Code

we will analyses this points for (time , and accuracy) in sequential mode.
The network is 3 layer, and different numbers of neuron in each layer.
a- normal prediction
b- pruning_connections (ratio 0.4)
c-pruning_neuron (ratio 0.2)



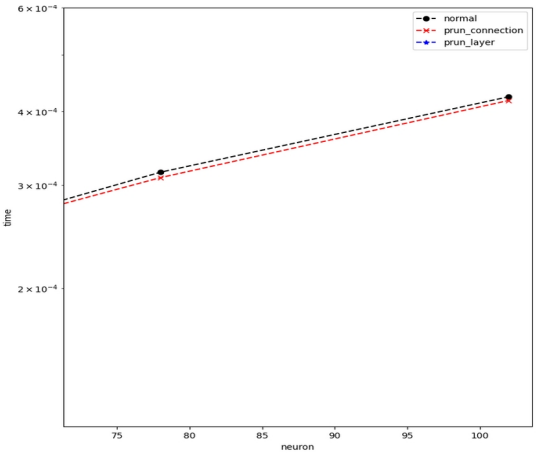


For accuracy:
because I used the same ratio for all the number or neurons, pruning = (network size*ratio)
so more the network increase the pruning is more.

It is normal for the blue line to see the accuracy is lost with increasing number of neurons , because there we put full neurons to zero, so we lost their effect , and for that at the beginning it was ok that huge difference , because it was about pruning just one neuron

Same situation for the pruning connection but the effect is less because we don't prune full neuron.

The green line represent the MPFR prediction which almost the good one and close to error relative=0 especially for huge network, but we should not forget that to have the perfect result the network should be trained using MPFR.but the problem the time it takes



For the time.
Here there is problem should be mentioned.
Prediction function, I tried 3 method to compute the multiplication
1- for loop with (*)
2- mulmat
3-numpy dot product
dot product of numpy library was the fastest.
But the problem when the vectors are sparse the time of execution is slower.
And that the reason you see for prune neuron takes so much longer time.
But at the same time we can see some improvement for pruning connection. Not that huge but it is the nature of dot product using numpy which was the best.

To use really the power of the pruning, we should use C language where we can really use avoid the multiplication with zero

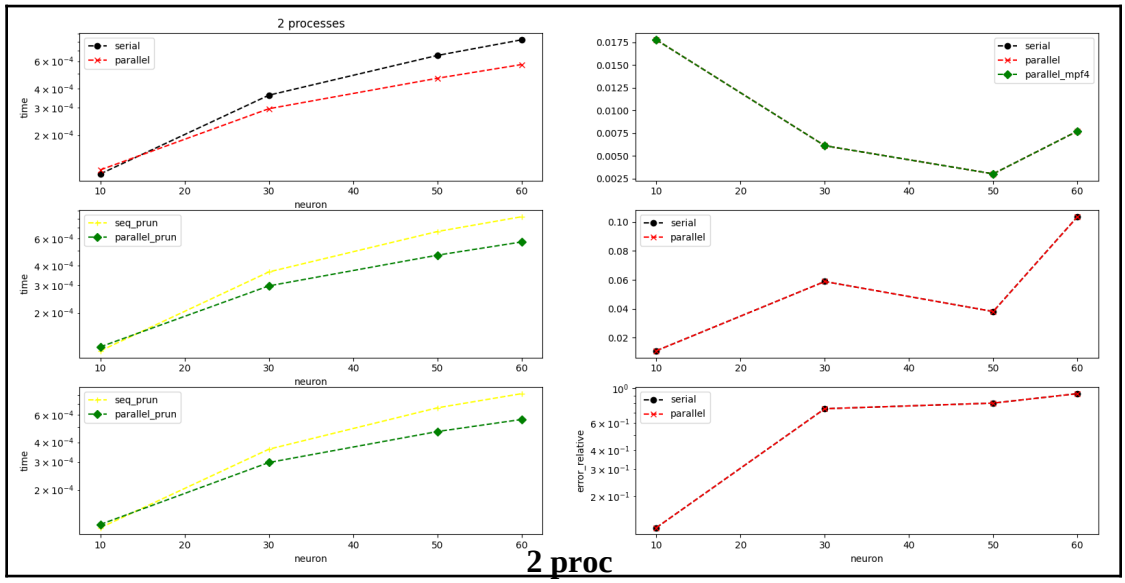
Note: in file NNModel there is function in the end called :forward_propagate_prun
where I tried to use a library in python called ‘ scipy” for sparse multiplication but still so slower than dot product.

2- Parallel Code

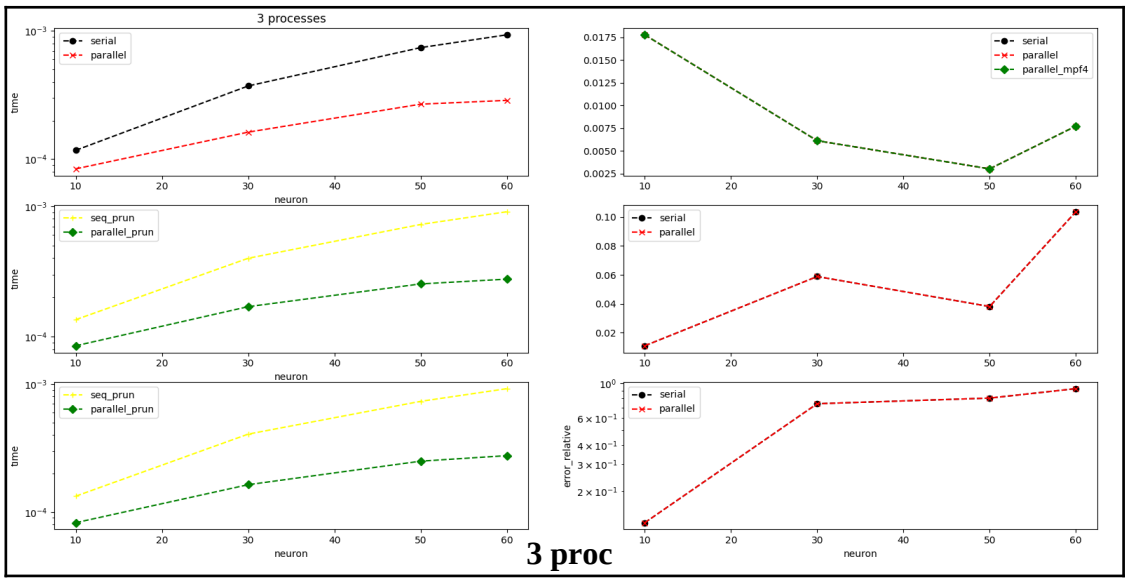
we will analyses this points for (time , and accuracy) in sequential mode. (implantation_para)
The network is 3 layer, and different numbers of neuron in each layer.
a- normal prediction (sequential, parallel)
b- pruning_connections (ratio 0.4) (sequential, parallel)
c-pruning_neuron (ratio 0.3) (sequential, parallel)

We can notice that more we increase the number of processes the time of execution decrease too.
And we know that the algorithm is right , because we can see that the line of relative error of the sequential and the parallel match for that we see just one line .

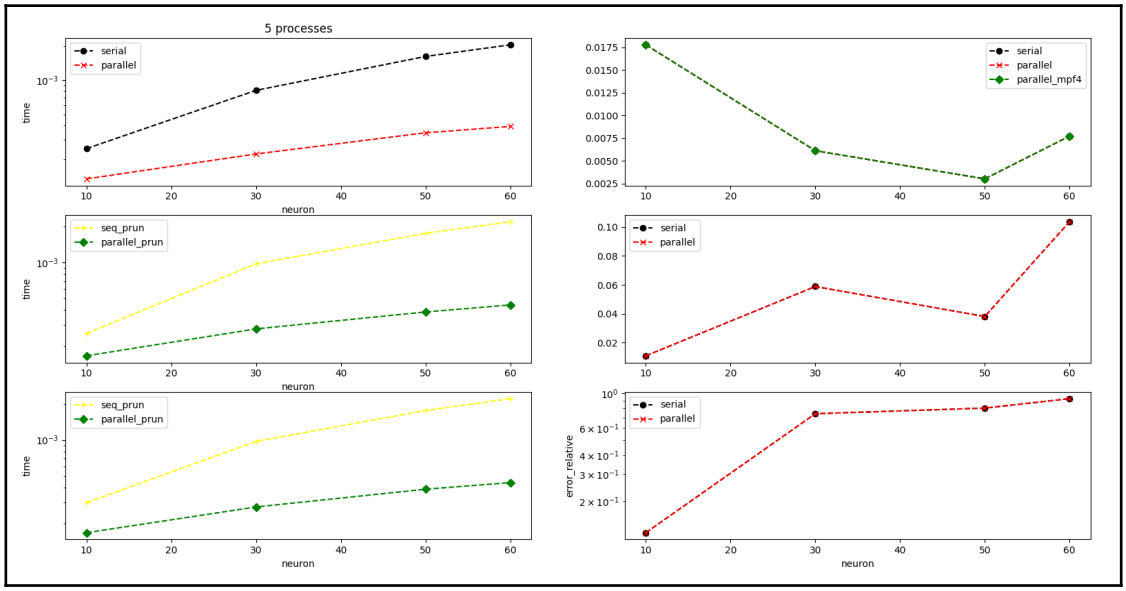
I should mention that the shape of the relative error is different than the one we saw above , because here the network size are different, (3 layers, 10,20,30,40,50,60 neuron in each) , where those network are build and save in another file and then opened inside the test file , because the test file implantation_para need to be called as (mpirun -n – python ..) so it is parallel , it is not good to train many different networks inside parallel where the training is sequential; for that I build the networks in file networks_build, and implantation function (save_model , open_model) written in NNModel.


2 processes
serial
parallel
parallel_mpf4
seq_prun
parallel_prun
time
error_relative
neuron

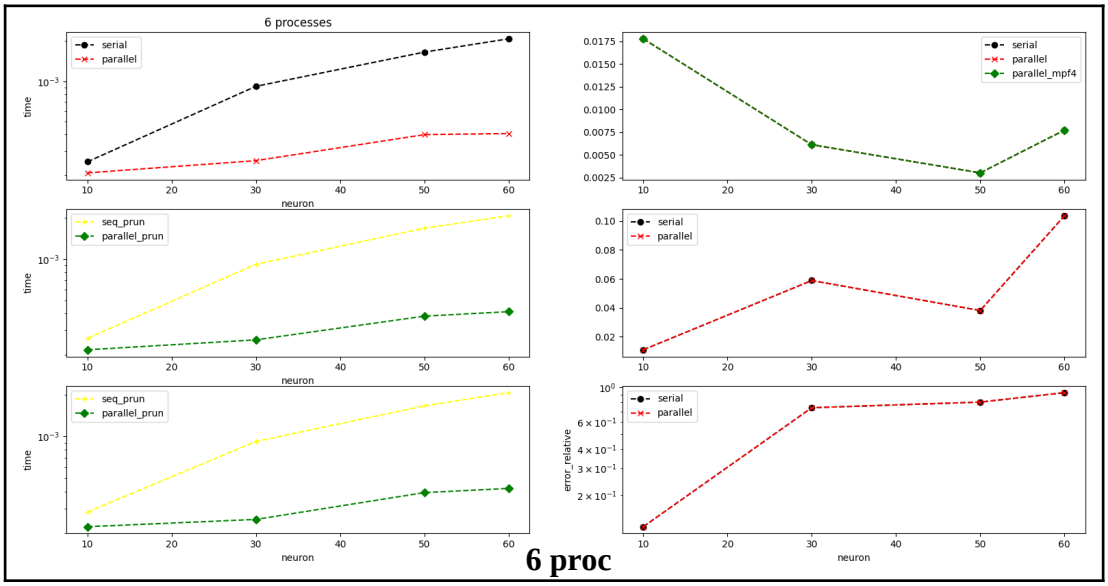
2 proc


3 processes
serial
parallel
parallel_mpf4
seq_prun
parallel_prun
time
error_relative
neuron

3 proc


5 processes
serial
parallel
parallel_mpf4
seq_prun
parallel_prun
time
error_relative
neuron

5 proc



6 proc

We can still notice the improvement in the pruning time too.
I tested different number of process so you can see that splitting up the neurons data to the process using “chunk equation “ is right whatever are the numbers o the process , and work well because the results were right.

Compare with TensorFlow

In file tenserflotest.py
I used the same data set of Xor we used for all the test , and build the same network 60*3
and the results was as follow:

tenser flow res [[0.9998767]] time: 0.08174838200011436

My network res: [0.9922788948402812] time: 0.0009017231559982974

and that even just for sequential predict function , for the same network in parallel with more than 2 processes is less than $5*10^{-4}$

But that is reasonable because as we see the accuracy of TensorFlow is higher due to using optimizer Adam , but the cost of this accuracy is the time of execution .

Conclusion:

It was obvious the good improvement we obtained, and it was even better than tenserflow, we lost some precision but still so acceptable.

But I should mention that the networks I did the test on were big enough like (60 neuron in 3 layers) which is number of wights huge.

At the same time the dataset were so simple like Xor , but it was enough to illustrate the algorithm.

-I test too something like this
inputs = np.array([[0, 1, 0], [0, 1, 1], [0, 0, 0], [1, 0, 0],[1, 1, 1], [1, 0, 1]])
outputs = np.array([[1], [2], [0], [1], [3], [2]])
which do the sum of the input using too Lrelu function for the activation layer , and it ave good results even with this small data set of training .
But I had a problem that when I try complicated dataset , the model has problem of training model for big number of neurons , and after so many tries I found the reason which is that I use gradient decent to update the wight which can find local minimal instead of global one so the computations explode in numbers.

And that is the future work to optimize.