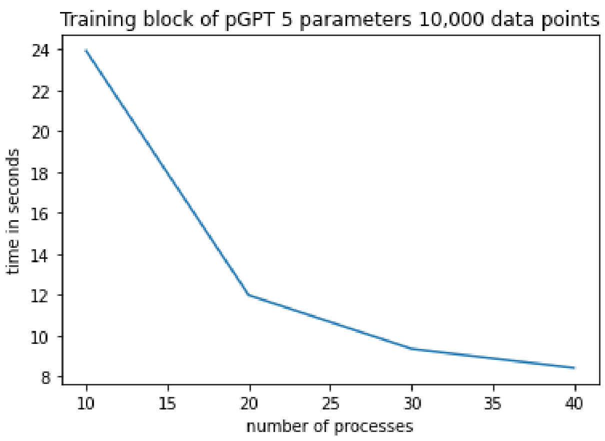
Parallel GPT

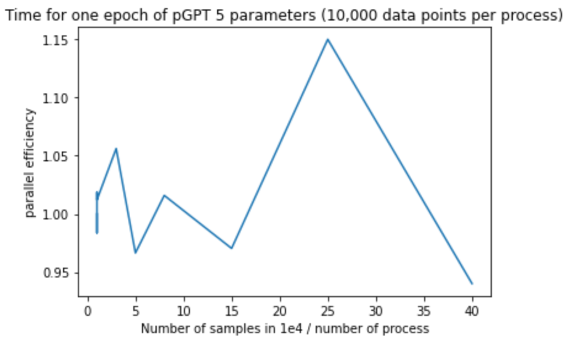
Introduction

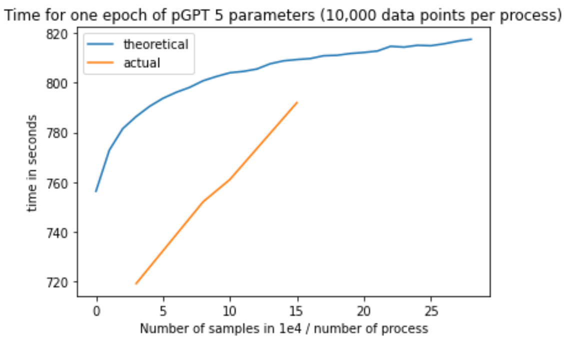
We all know the impact of of chatGPT which are trained on massive internet data. The success of ChatGPT can be attributed to the amount we can parallelise, In fact the idea of Attention mechanism, which is the crux of GPT was designed to overcome the sequential algorithm LSTM. Once we make the network parallel both the model and data parallel we can leverage the huge amount of data and train massive compute intensive algorithms.

In this work, I will use model one attention block, which would constitute the fundamental element of the GPT. I will analyse the efficiency of the parallel strategies and uncover the reasons behind such numbers. Test some of the asymptotic limits of the parallel strategies and will raise some comments on the parallel strategies we have used. you will find the explanations on why we have chosen a particular strategy.

To motivate further, Here are the final results. (In parallel all we care about the efficiency, if the efficiency is great we can throw lot of compute at the problem.)





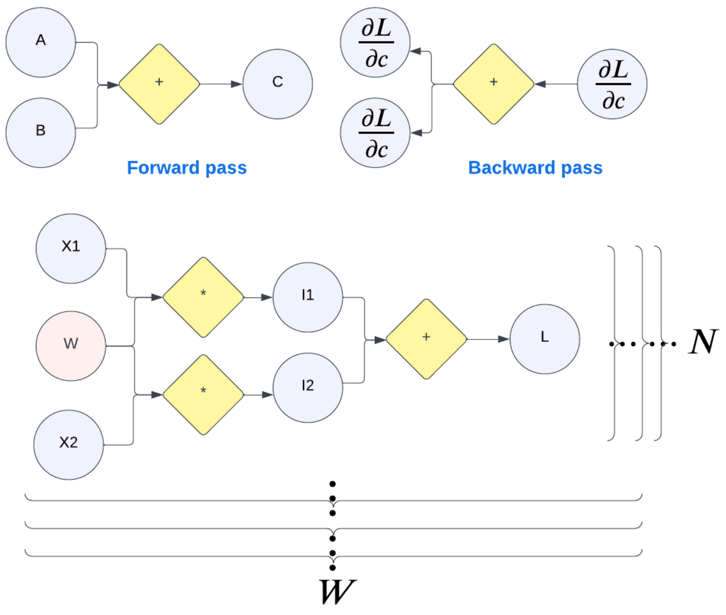


Suppose you have built your model on 10 GB of data, now you want to train the model on 100 x of data, you can train the model in same time by parallelizing the data on 100 different processors, as you can see the efficiency remains constant. (Around 95%). One caveat is that, As there is a scatter and gather operation on model parameters, all the process have to wait for the worst performing process, so you will be limited the slowest performing processor, but the efficiency will be constant. even for 100x and 200x of parallel process.

Modern Day use: Currently OpenAI is training next version of GPT, with more data, parallelising on 100 A100 GPUs (you can see the logs on wandb),they use NCCL to train which is very similar to our MPI with OpenMP combination.

Algorithm:

I will start with the Neuron, which in the code we call Value, is the basic unit of the neural network, each Neuron has the basic math operations (+, -,/ ,*) and its derivative. We can write any function as a combination of these functions and use chain rule for backpropagation through them. Given N data points and W weights. In our experiments we fix N =10,000 and W=5, unless specified.


A
B
+
C
Forward pass
$\frac{\partial L}{\partial c}$
$\frac{\partial L}{\partial c}$
+
$\frac{\partial L}{\partial c}$
Backward pass
X1
W
X2
*
*
I1
I2
+
L
N
W

Neural Network

To update the weights of neural network, we need to compute the derivative of the Loss (last neuron) with respect to the w.

From the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial I} * \frac{\partial I}{\partial w}$$

Properties used for parallelism:

Suppose,

$$L = \frac{A + B}{2}$$

$$\frac{\partial L}{\partial w} = \frac{\frac{\partial A}{\partial w} + \frac{\partial B}{\partial w}}{2}$$

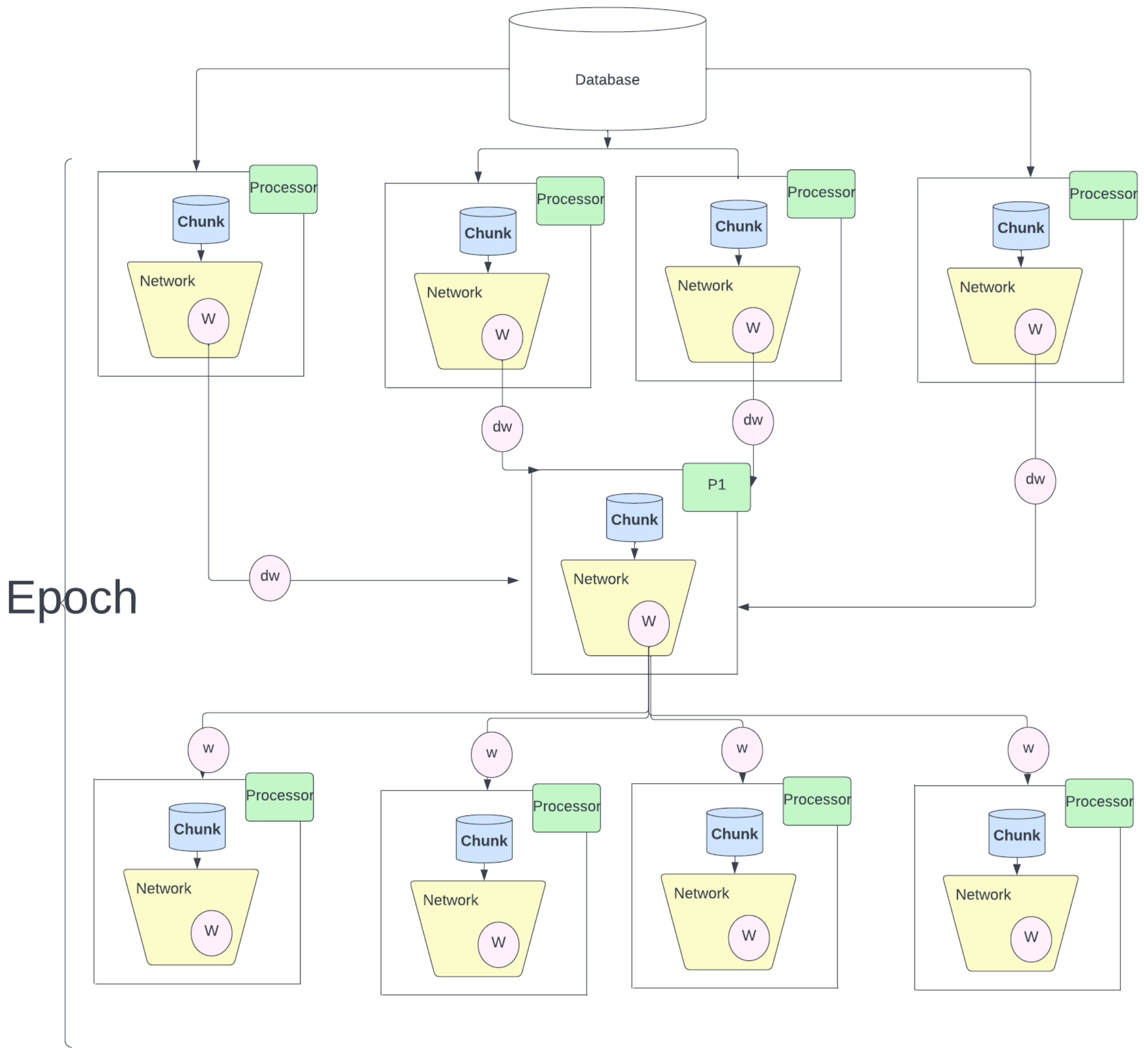
As you can see from the backward pass illustration above, the derivative flows through the addition sign. As we generally model the final loss (error) as mean of losses experienced by all the data points, we can divide the data into chunks and process them with same initial W and once the forward and backward pass are done we can average the gradients.

Update Step:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{w})$$

We can repeat the same experiment till convergence 300 times for our example.

Parallelization Strategy:



Within server parallelization:

We can use openMP to parallelise all the function computations across chunks of data on a single server. For example we can use openMP to parallelise the matrix multiplication, applying exponential across all the data points. We can only parallelise the forward pass of the application and not the backward pass because the derivative of the parents must be completely calculated before the children in the network graph. (topological sort). Only this aspect of the network cannot be parallelized (pytorch uses mutex locks to prevent thread parallelism for this)within a server. So current optimisations are focused on efficient memory management and backward pass (pytorch 2.0 recently release static graphs, once topologically sorted and memory if fixed, in next epoch the graph remains constant.)

Why do we get such high efficiency?

Because the parameters of the network that are communicated via Gather and then Broadcast are very low (5) compared to the data points (10,000). These are CPU intensive rather than communication intensive, which is perfect example of MPI parallelism.

Testing the correctness:

Can the algorithm solve simple equation

$$w_1 f_1 + w_2 f_2 + w_3 f_3 + w_4 f_4 + +w_5 f_5 = y$$

Suppose W = [0.500593, -0.0807231, 1.17474, -1.45482, -0.629856]

Generate random data and train a model and the final epoch weights should be equal to W.

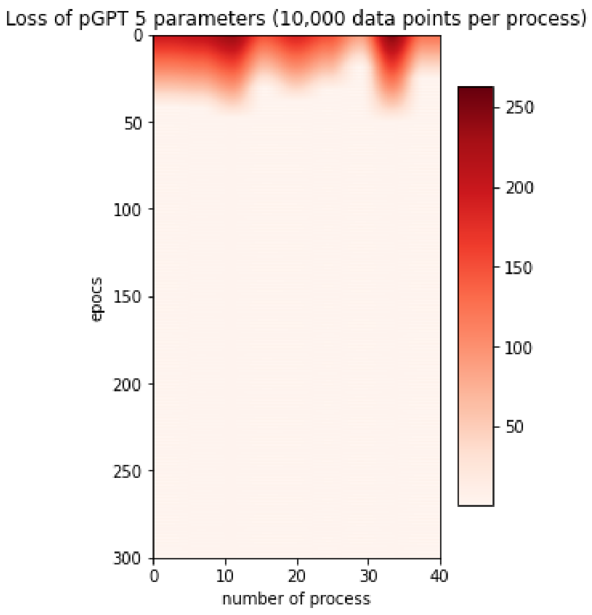
After training our weights on all the parallel processes looked like

W_hat = [0.504076, -0.0755702, 1.1745, -1.45666 ,-0.63025]

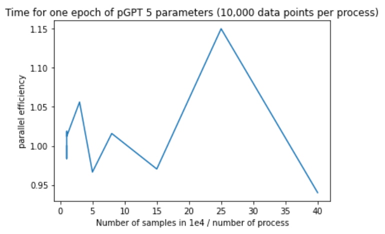
Which gives the sense that parallel algorithm has been implemented correctly. (the output does not depend on the number of process only on how many epochs each data is trained.

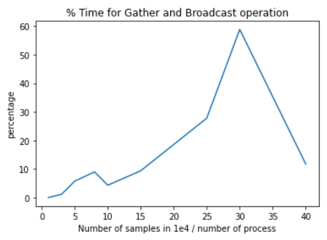
Which validates our correctness.

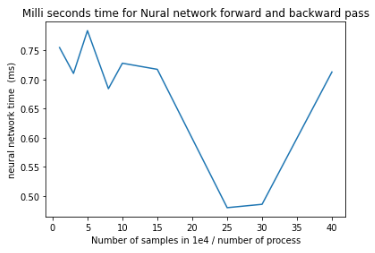
Loss functions should also continuously decrease which validates our claim of weights moving in the correct direction.



More plots on percentage splits what part is most time occupied?

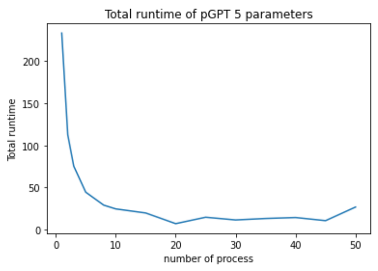


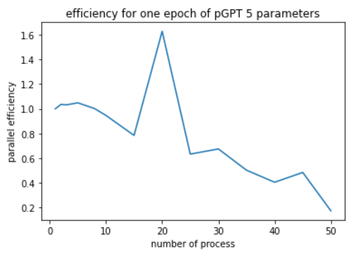


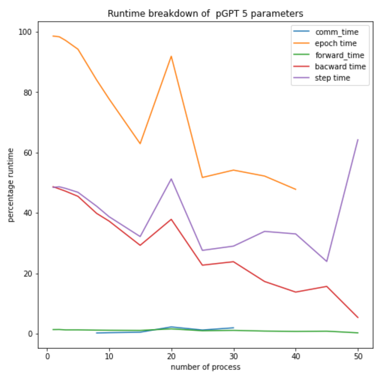


As we can clearly see in this plot the bottleneck observed is mainly due to the communication issues, as we are seeing that all the nodes that we received have same capacity. But in the case of 30 nodes the neural network time remained constant at around 0.7 seconds but the rest 0.7 seconds was wasted waiting for the communication to happen.

More experiments on fixing the data and estimating the parallelism.







You can see the analysis of the timing of various points in algorithm, As you can see 3 major trends as expected.

Time and efficiency graphs are as expected. because we are dealing with only 10,000 samples the efficiency is communication time vs calculation time (As we take only 10 ms to compute 1000 samples) but in general deal with huge data so this will not be an issue. (you can see the experiments above with 10,000 samples on each process.

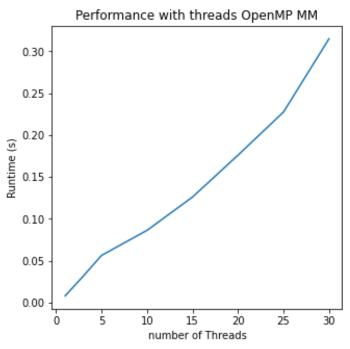
The forward pass time is very less (as we use OpenMP for matrix multiplication parallelisation and Cpp also unrolls the loop, so there will be very little time in forward pass.

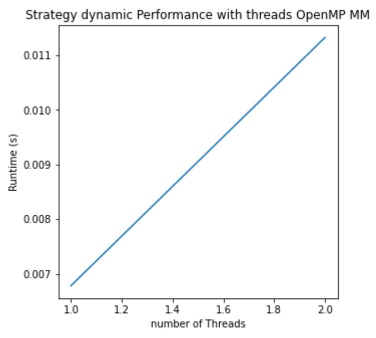
In backward pass we need to pass through the entire graph. O(N) functionality, and traverse in the entire graph in topological sort way so as we parallelise we only have O(N/d) time which is a huge gain, so you will see backward pass drop in percentage of total time with increase in processors.

The update step process is a sync process with all the weights, So the process wait for the slowest process to complete (only 5 floats needs to communicated so bandwidth is not a problem.) This bottleneck is not an issue initially but as the processes increase its percentage in total time starts increasing.

Comments on openMP experiments with varying thread sizes.

Threads	Split
1	300
3	294,3,3
5	14,13,20,39,37





Counterintuitive results

Not equally distributed on Threads 3, even after fixing with threads on 5, the time is increasing with increasing the processes. (The dynamic creation of memory and not uniform access of cache line is causing the issue). Default strategies in the pragma omp always does not give the best results.

Same problem persists with other strategies as well dynamic and guided.

Experiments failed:

> Using Cuda and NCCL to hyper parallelise the vector operation to cuda. this requires memory contiguity, we didn't reserve the space as we are dynamically creating the graph even for fairly simple calculations. (memory is known). This is the current bottleneck. I have fixed this issue on my latest code. but working with vectors need more memory management to fix issues.
>
> None of the openmp strategies showed huge impact on the forward pass of the algorithm, more details on the types of schedules explored (ranging from static, dynamic and guided). This might be due to the dynamic memory created.