p-GPT
Scaling Goldilocks conditions for transformers models

Gaudi Sachit
Department of Computer Science
Michigan State University
gaudisac@msu.edu

Abstract

This paper investigates the scalability of transformers models under Goldilocks conditions, focusing on parallelization strategies to harness computational resources efficiently. By leveraging the parallel nature of neural network computations, particularly through OpenMP within server parallelization, we explore the performance dynamics of training models with increased data sizes. Through empirical analysis and experimentation, we demonstrate how the efficiency of parallel strategies remains consistent, even when scaling data sizes by orders of magnitude. Drawing parallels with contemporary endeavors such as OpenAI's training of GPT models on large-scale infrastructure, we highlight the importance of optimizing communication and computation balance for achieving high efficiency. Our findings underscore the significance of efficient memory management and optimization strategies, while also shedding light on the complexities and challenges encountered, such as memory contiguity issues in CUDA and NCCL-based parallelization. This study contributes insights into the nuances of parallelization in transformer models, offering valuable guidance for optimizing performance at scale.

1 Introduction

ChatGPT has redefined practical AI applications, facilitating tasks such as customer support automation, content generation, language translation, and personalized recommendation systems. The model behind ChatGPT, GPT, owes much of its success to its scalability achieved through parallelization. Trained on vast amounts of internet data, GPT utilizes an Attention mechanism Vaswani et al. (2017), which is inherently parallelizable. By parallelizing both the model and the data, we can effectively leverage large datasets to train large models. This success was achieved due to the huge improvements in parallel setups.

The state-of-the-art Language model, with the underlying GPT model, would take 355 years to train GPT-3 on a single NVIDIA Tesla V100 GPU Brown et al. (2020). However, this has been done in 36 days on 1024 V100 GPUs by the combination of both model and data parallelism.

In this study, we'll focus on the basic attention block in the GPT model. We'll examine how well different parallel strategies work and why. We'll also test how these strategies perform when pushed to their limits and share our thoughts on why we chose specific architecture and design choices.

2 Parallel architecture

We'll begin by looking at the Neuron, which we refer to as the Value. It's the fundamental unit of the neural network, performing basic math operations like addition, subtraction, multiplication,

Preprint. Under review.

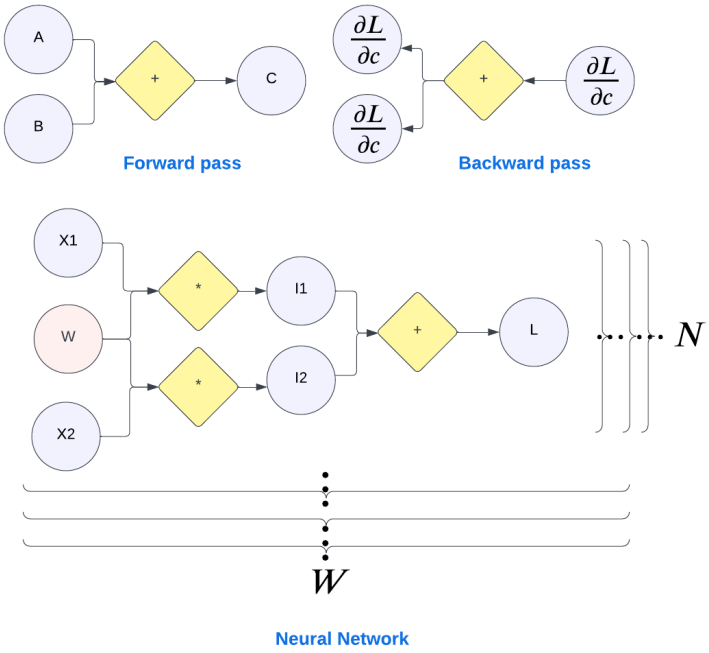


Figure 1: Figure illustrating the chain rule. One important note is that for the addition operation, the gradient simply passes through the children of the node.

and division, along with their derivatives. Using these fundamental operations, we can express any complex function and apply the chain rule for back propagation.

2.1 Back Propagation

The standard procedure for training neural networks can be summarized as follows: Initially, the loss is calculated by summing the contributions from all data points. Subsequently, gradients of the weights are computed, employing the chain rule as described in Equation 1. Finally, the weights are adjusted based on Equation 2. This procedure repeats until the updates for the weights are sufficiently small.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial I} * \frac{\partial I}{\partial w} \tag{1}$$

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{w}) \tag{2}$$

$$L = \frac{A+B}{2} \quad \frac{\partial L}{\partial w} = \frac{\frac{\partial A}{\partial w} + \frac{\partial B}{\partial w}}{2} \tag{3}$$

Assume A and B are two data points and the loss is combined by summation.

The key idea for the parallelization is that any addition operation can be parallelized from Equation 3. In the backward pass, the gradients from the summation are transferred without any change to the inputs. If a node receives multiple gradients from different summations, we simply sum them.

As depicted in the backward pass illustration in Figure 1, the derivative flows through the addition sign. This property of summation gives rise to data parallelism. We can divide the data into chunks and process them with the same initial W, and once the forward and backward pass are complete, we can sum the gradients.

2.2 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.)

2.3 Overall parallelisation strategy

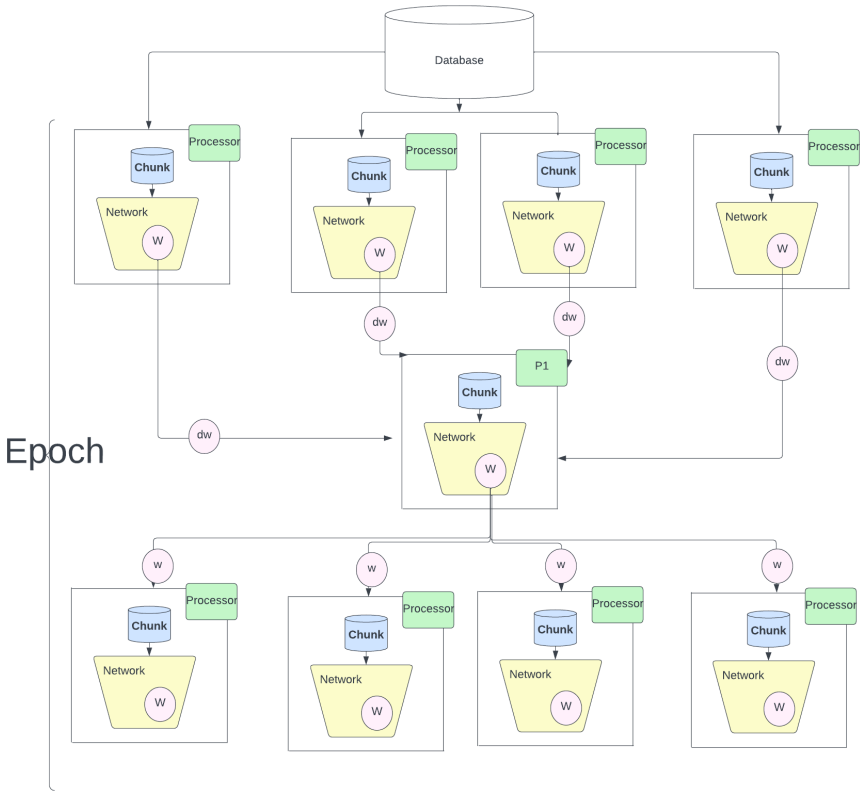


Figure 2: Data Parallelisation strategy,

3 Slowest GPU is all that matters

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

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. 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

$$w_1 f_1 + w_2 f_2 + w_3 f_3 + w_4 f_4 + + w_5 f_5 = y \tag{4}$$

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

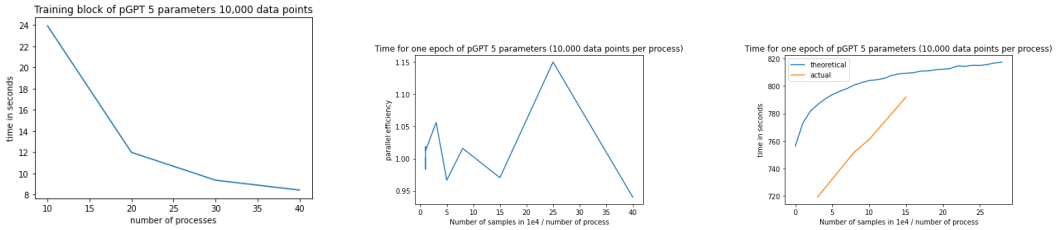
W_hat = [0.504076, -0.0755702, 1.1745, -1.45666 ,-0.63025]

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.

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.)



(a) Caption for Image 1 (b) Caption for Image 2 (c) Caption for Image 3

Figure 3: Overall Caption for the Block

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

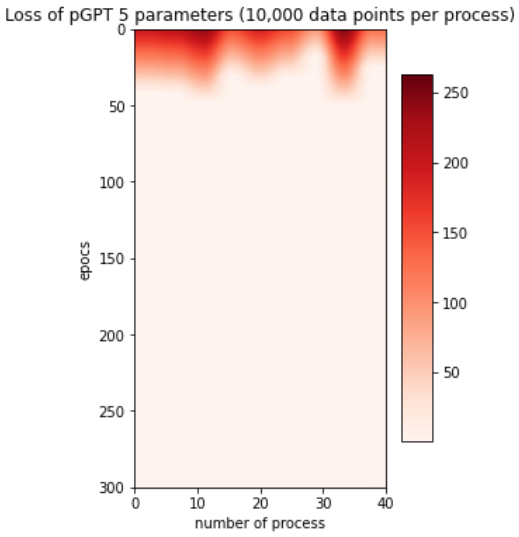
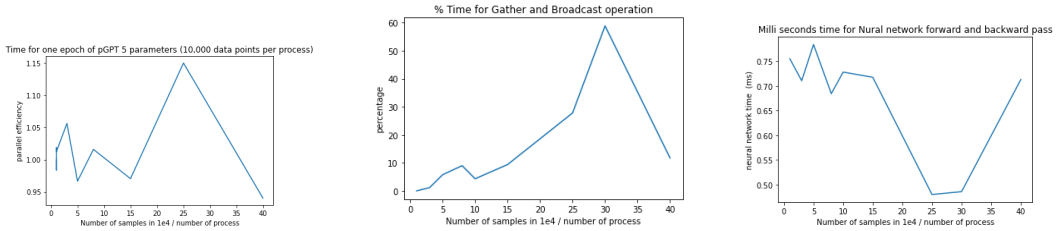


Figure 4: Caption

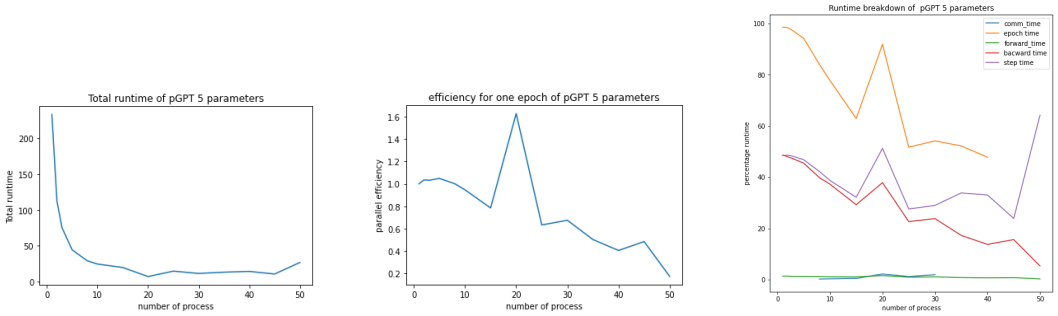


(a) Caption for Image 1 (b) Caption for Image 2 (c) Caption for Image 3

Figure 5: Overall Caption for the Block

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



(a) Caption for Image 1 (b) Caption for Image 2 (c) Caption for Image 3

Figure 6: Overall Caption for the Block

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.

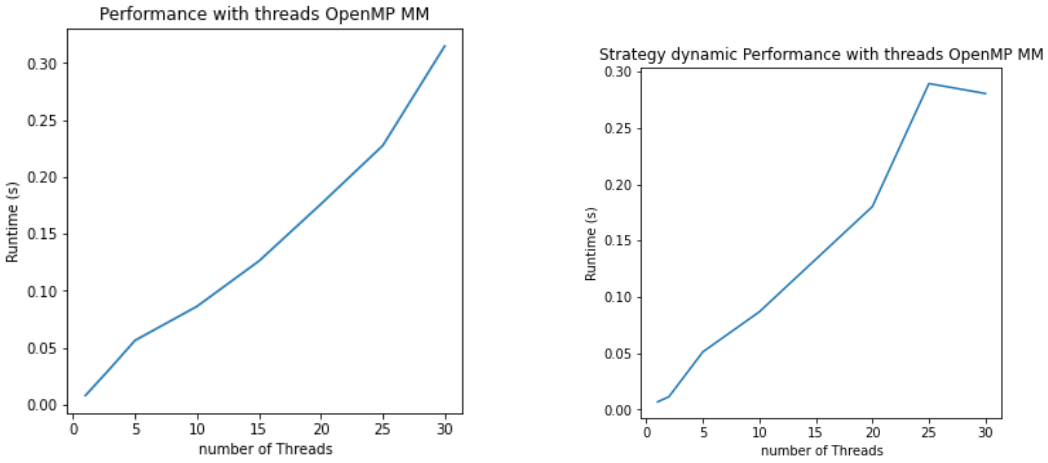
The update step process is a sync process with all the weights, So the process wait for the slowest process to complete (only 5 floats needs to communicated so bandwidth is not a problem.) This bottleneck is not an issue initially but as the processes increase its percentage in total time starts increasing.

Comments on openMP experiments with varying thread sizes.

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

Table 1: Thread-Split Table

Counterintuitive results



(a) Caption for Image 1

(b) Caption for Image 2

Figure 7: Side-by-side Images

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.

References

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language models are few-shot learners.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.