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.

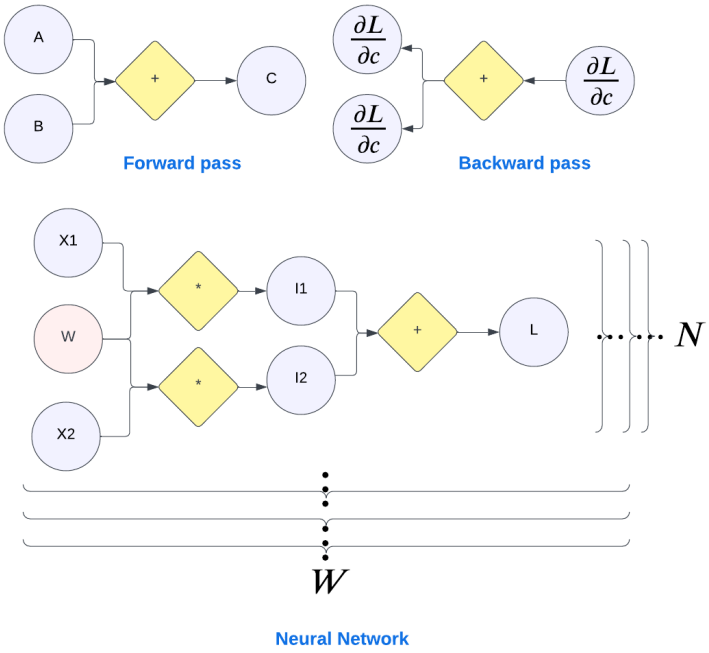


Figure 1: Figure illustrating the chain rule. One important note is that for the addition operation, the gradient simply passes through the children of the node.

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.

2.2 Inter Server parallelization

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

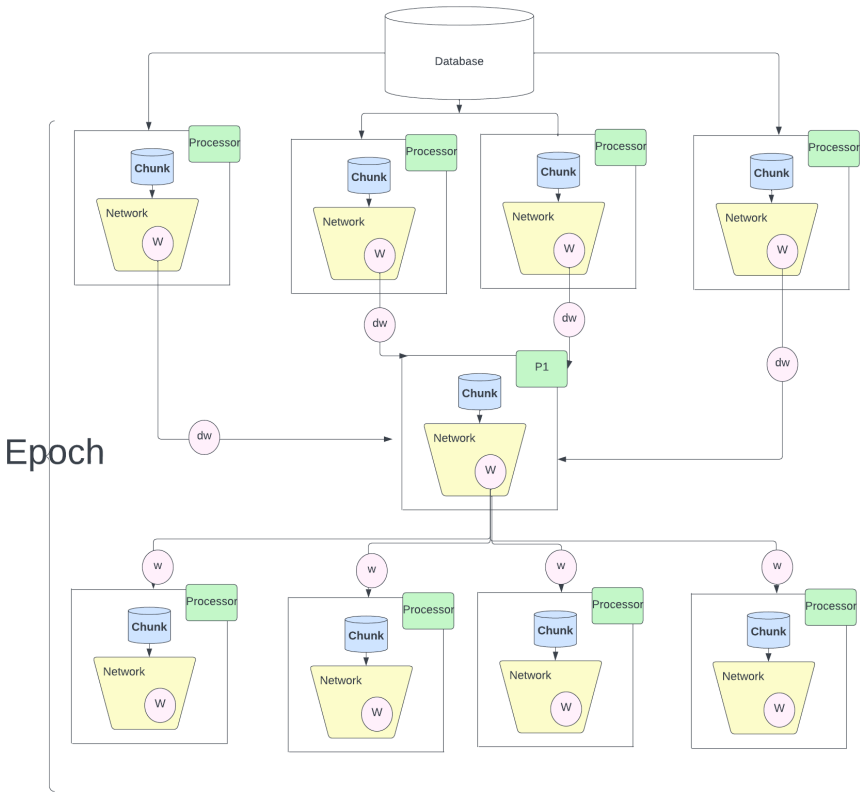


Figure 2: Data Parallelisation strategy,

and process them with the same initial W, and once the forward and backward pass are complete, we can sum the gradients.

In our distributed computing setup utilizing MPI, gather, scatter, and broadcast operations are pivotal components for efficiently handling data across multiple processes or servers.

We begin by partitioning our dataset into manageable chunks and scatter these data chunks across the available processes. Each process then operates on its assigned data subset independently, performing forward and backward passes to compute local gradients.

Once the local computations are complete, we utilize scatter operations to distribute the updated model weights from a master process to all other processes. This ensures that each process has access to the most recent model parameters for the next iteration of training.

After completing the forward and backward passes on their respective data chunks, processes gather their local gradients using the gather operation. These gradients are then aggregated, typically by summing or averaging, on the master process using a reduction operation. This aggregated gradient information represents the collective insights from all processes and serves as the basis for updating the global model parameters.

Finally, the updated model weights are broadcasted from the master process to all other processes using the broadcast operation. This ensures that every process possesses the synchronized model parameters for the subsequent iteration of training.

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.

In summary, the combination of scatter, gather, reduction, and broadcast operations in MPI enables efficient distribution, aggregation, and synchronization of data and model parameters across distributed processes, facilitating parallel training of neural networks on large-scale datasets.

2.3 Intra server parallelization

We utilize OpenMP to parallelize function computations within a single server. For example, we parallelize matrix multiplication and function application over Neurons. However, it's crucial to understand that we can only parallelize the forward pass of the application, not the backward pass. This limitation arises because the derivative of parents must be completely calculated before the children in the network graph, adhering to topological sort principles (chain rule). Consequently, this aspect of the network cannot be parallelized within a server. While PyTorch 2.0 has introduced static graphs, where the graph remains constant in the next epoch once topologically sorted and memory is fixed, we do not currently focus on utilizing static graphs. Hence, our current optimizations primarily revolve around efficient memory management and parallel forward computations and function parallelization.

3 Experimental setup

We want to understand the parallel efficiency of the setup. So we restrict over selves to 10,000 data points and a simple linear regression example. which has a closed form solution, which can be tested for correctness. Can the algorithm solve simple equation?

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

4 Results

4.1 Test For Correctness

To verify the correction of learned weights. We train the model with a known set of random weights or the underlying process of f_* and y.

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

After training our weights on all the parallel processes looked like

$\hat{W} = [0.504076, -0.0755702, 1.1745, -1.45666, -0.63025]$

We can observe that $\hat{W} \sim W$, We can also confirm that the distributed training is working as expected by comparing the loss function. The final loss from the Figure 3 is very close to zero. That implies that all the servers are synchronised as expected. Correctness can also be confirmed from the Loss functions continuously decreasing.

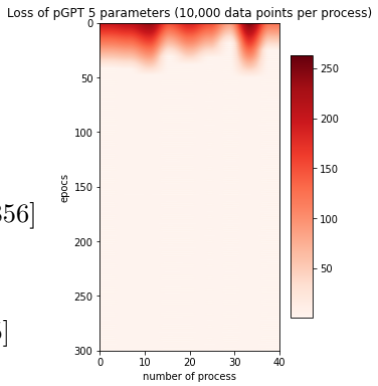


Figure 3: Final loss when trained Equation 4 on different number of servers.

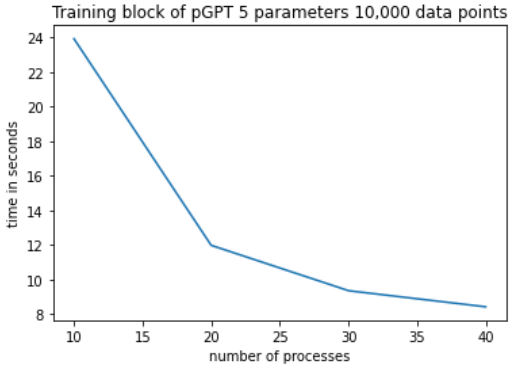
4.2 Parallel Efficiency

In parallel processing, efficiency is paramount. Deep learning owes much of its success to the remarkable phenomenon of maintaining a stable parallel efficiency of around 95%, resulting in exponential time decreases as the number of processes increases. To illustrate, consider training a deep learning model on a single GPU, a task that would require an impractical 300 years to complete. However, with the utilization of multiple GPUs, the same task can be accomplished in a mere 36 days. This exponential reduction in time underscores the potency of parallel processing, enabling the handling of increasingly complex tasks within feasible timeframes.

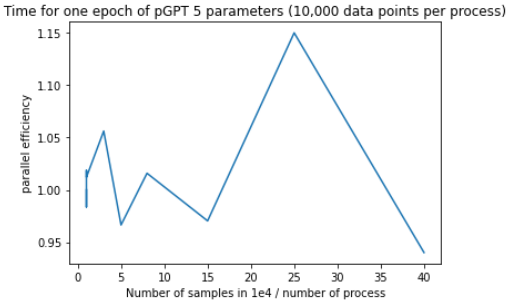
In distributed file systems like Hadoop Distributed File System (HDFS), communication can become a bottleneck when processing tasks involve aggregating data stored across multiple servers. For instance, in a MapReduce job, if computations require combining results from different nodes, the need for frequent data exchanges between servers can lead to communication overhead, hindering parallel efficiency.

In our setup, communication overhead depends primarily on the number of weights, ensuring stable parallel efficiency. For example, when distributing data for parallel training, each server

computes gradients and communicates them to a central server for aggregation. With consistent communication overhead, efficient scaling is achieved without bottlenecks, particularly in scenarios where computational workload dominates.



(a) Time decreases as we increase number of parallel process.



(b) Constant parallel efficiency

Figure 4: Efficiency and Time with increase in number of processors.

4.3 Slowest GPU is all that matters

One caveat is that there is a scatter and gather operation on model parameters, requiring all processes to synchronize at the end of each epoch to gather gradients. Consequently, the process must wait for the slowest performing processor, limiting efficiency. However, efficiency remains constant even for 100x and 200x parallel processes.

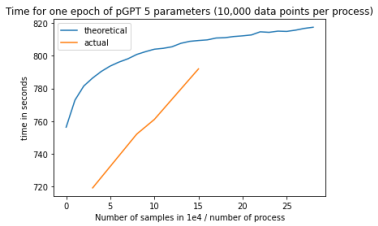


Figure 5: Final loss when trained Equation 4 on different number of servers.

We assume that the server speeds obtained from the HPCC (computing facility of MSU) follow a normal distribution. Through multiple experiments, we determine the mean and variance of the server speeds. The slowest process follows the distribution as estimated by Ho and Hsing (1996), providing a theoretical bound on performance versus the number of processes. We observe this theoretical bound in Figure 5. This trend aligns with the theoretical bound, indicating that the efficiency of the system is dictated by the smaller, slower processes.

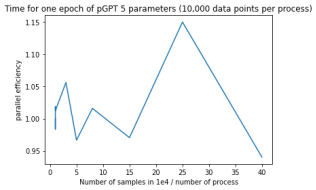
To mitigate the influence of slower processes, we can implement data balancing on every epoch to allocate smaller batches for slower processors. However, this introduces additional overhead due to increased data communication after each epoch. Nevertheless, as chips age, balancing may yield more benefits compared to the time lost in data communication. However, in our case, implementing data balancing did not result in improved performance, as we found that more or less processors allotted had similar speeds.

4.4 Unraveling the Mystery: Where Did My Time Go During Neural Network Training?

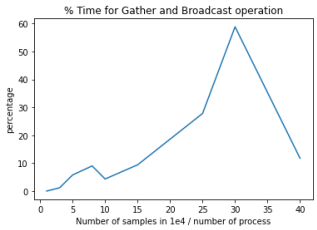
More plots on percentage splits what part is most time occupied?

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.

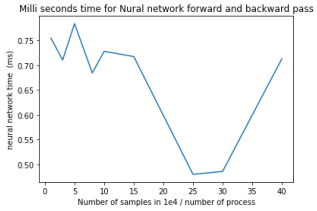
More experiments on fixing the data and estimating the parallelism. You can see the analysis of the timing of various points in algorithm, As you can see 3 major trends as expected.



(a) time vs number of process

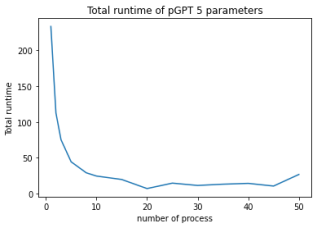


(b) efficiency vs number of process

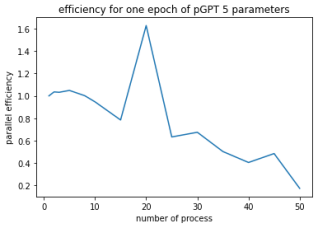


(c) percent of usefull computation time (not cummuication time lost) vs number of process

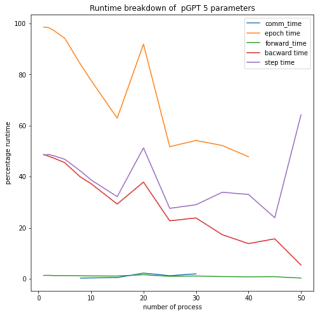
Figure 6: Increase in number of process keeping the total number of data points per server fixed



(a) time vs number of process



(b) parallel efficiency vs number of process



(c) percent of operation vs number of process

Figure 7: Increase in number of process keeping the total number of data points fixed

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.

5 Intra Server parallelism speed analysis

Comments on OpenMP experiments with varying thread sizes.

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.

Due to C++'s inherent parallelization of vector operations, none of the OpenMP strategies significantly impacted the forward pass of the algorithm. We explored various scheduling types, including static, dynamic, and guided. Since our code is written at the elemental operations level, with fewer matrix operations, the impact was limited. However, if we rewrite the code at the matrix level, we can leverage this parallelization for improved speed.

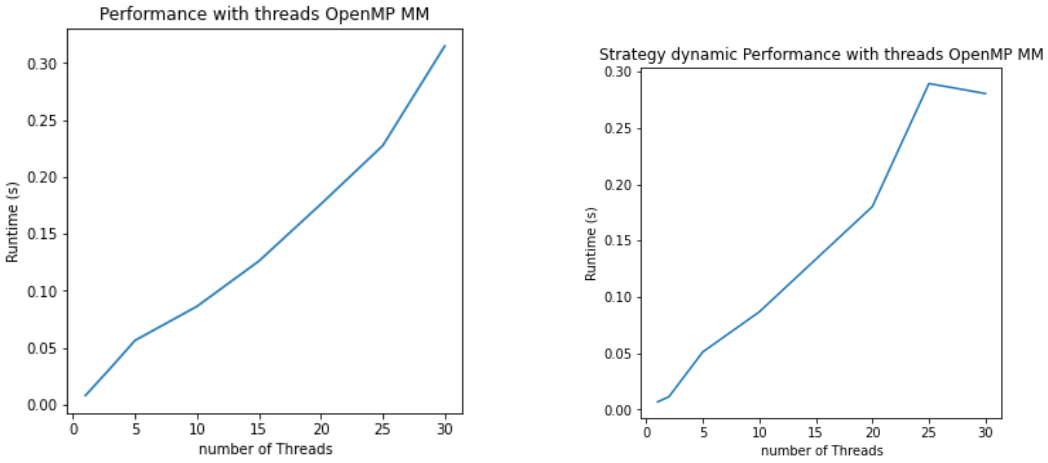


Figure 8: OpenMP parallelisation time vs number of threads

6 Limitations of current implementation

Utilizing CUDA and NCCL for hyper-parallelizing vector operations to CUDA requires memory contiguity. However, we did not reserve space as we dynamically created the graph even for simple calculations, resulting in a bottleneck. I have addressed this issue in my latest v2 code, but managing memory for vectors requires further improvements to realise the gains.

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

Table 1: Does 300 data points split uniformly share the load with increase in threads? No

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.

Ho, H.-C. and Hsing, T. (1996). On the asymptotic joint distribution of the sum and maximum of stationary normal random variables. *Journal of applied probability*, 33(1):138–145.

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.