A Survey on Parallelism in Large Language Models

Aashish Nehete
anehete@umass.edu

Sanath Upadhya
supadhya@umass.edu

Riya Adsul
radsul@umass.edu

1. Introduction

Large language models is the most recent invention to take the world by storm with their natural language processing capabilities. They are very large neural networks with billions of parameters that are trained on a large corpus of textual data. This allows LLMs to understand and generate text for a variety of tasks including machine translation, creative writing, summarization, etc. This diverse nature has enabled it to be deployed at a scale never before seen by the machine learning community from data centers to personal computers to IoT devices.

The size of the Large Language Models (LLMs) has grown exponentially, over the years, as seen in Figure 1. Unfortunately, it comes at a cost, training and inference are very slow and expensive due to their large sizes. PaLM by Google with 540 billion parameters [27] and GPT-3 with 175 billion [3] are some of the largest models we know. To successfully harness the power of LLMs requires the ability to train and deploy them as massive distributed systems. Parallelism has emerged as a promising research area to enable large-scale adoption of this technology. With such large models unable to process on a single chip, efficiently distributing load across multiple cores, devices, or geographic locations promises much needed speed-ups.

This paper aims at looking at the myriad techniques that are used in training and at inference of Large Language models (LLMs). The literature survey aims at answering the following questions:

1. What are the different types of parallelisms that are used during training and at inference in a LLM?
2. What are the characteristics of each of these types of parallelism?
3. Can these parallelism techniques be used in combination with one another and if so, what is the expected gain in the total speedup, be it for training or during inference?

To efficiently deploy large language models (LLMs), researchers are exploring parallelism techniques that address the challenges of high computational cost and memory usage. Our survey investigates these techniques, aiming to understand how LLMs are parallelized for training and serving[31]. This is useful in scenarios requiring low latency and high throughput, where traditional serving methodologies may fall short, for example in the case of a mobile application that needs to run a LLM for inference (as seen in the ChatGPT mobile application).

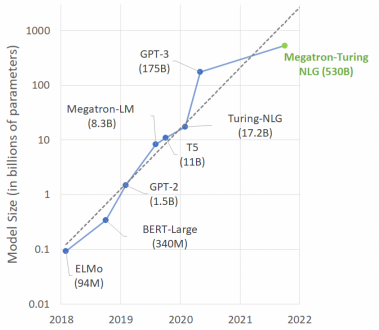


Figure 1. Growth of the number of parameters in Large Language models [2]

By delving into various parallelism techniques employed and examining the similarities, differences, and phase-specific effectiveness of these approaches, the survey offers a comprehensive understanding of the current state of efficient large language models training and inference using distributed systems.

Outline Section 2 gives a background on each of the parallelism techniques that will be discussed in the paper. Section 3 delves into the technique of data parallelism and the various methods that are used to achieve the same. Section 4 talks about Model parallelism, section 5 about Sequential parallelism, section 6 on Hybrid parallelism, section 7 on Automatic parallelism and the conclusion is presented in section 8.

2. Background

The taxonomy of the different parallelisms that are covered in this literature survey is as shown in Figure 2. A brief description of each of the parallelisms is presented here, and a more in-depth analysis of these techniques are done in subsequent sections.

Data Parallelism: This is a technique wherein the training data is partitioned and given to each of the servers, which store the entire copy of the model which has to be

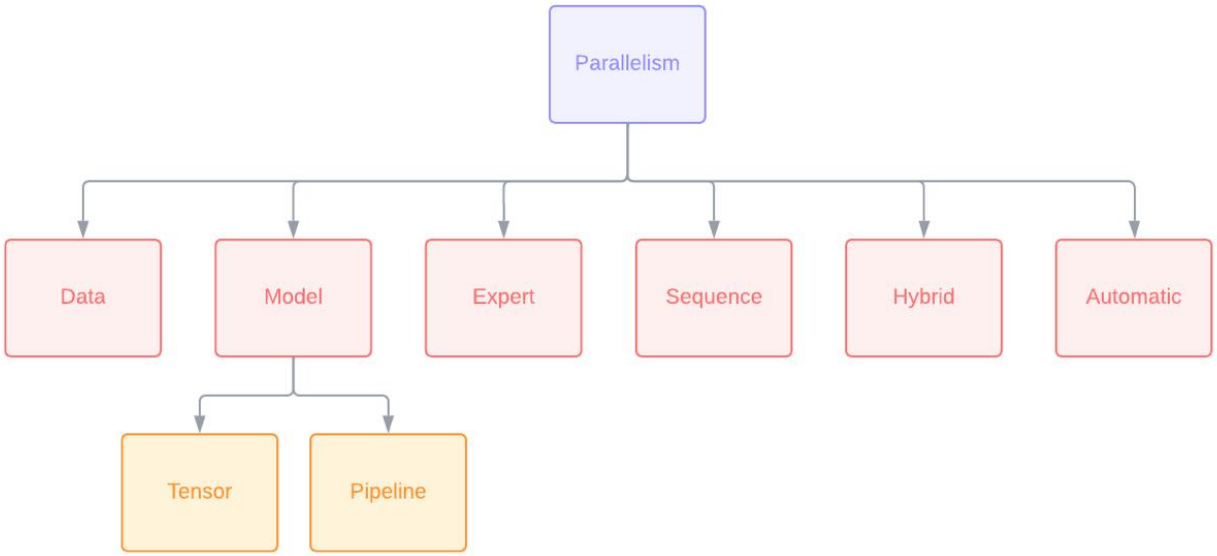


Figure 2. Taxonomy of Parallelisms in LLMs

trained within them. At the end of each iteration, the results from different servers are combined and are distributed to all the servers so that the parameter values within the model remain consistent [34].

Model Parallelism: While data parallelism offers simplicity, it introduces redundancy in the form of duplicated model weights across devices. Model parallelism, in contrast, distributes the model itself across processing units. This strategy alleviates the memory bottleneck associated with data parallelism by partitioning the model into sub components[24]. There are two primary forms of model parallelism:

1. Tensor parallelism: This approach focuses on parallelization of computations within individual operations, such as matrix multiplication. It can be viewed as intra-layer parallelism, as it distributes the workload for a specific layer across multiple devices.
2. Pipeline parallelism: This technique concentrates on parallelization of computations between different layers. Each device executes a subset of the model's layers sequentially, fostering inter-layer parallelism. In essence, pipeline parallelism creates an assembly line for processing training data.

Expert Parallelism: Mixture-of-Experts is a paradigm in deep neural networks that replaces a single feed-forward layer with multiple feed-forward sub-networks called experts. Each training example is only sent to a subset of these experts controlled by a learnable gating network. This allows a kind of model and data parallelism where each expert is independently deployed on its own core which immensely scales model capacity while keep computational cost constant.

Sequence Parallelism: The input sequence that is fed into the LLM is first split into chunks and then these chunks are fed into different GPUs, and the output is calculated using the Ring Self Attention. This technique allows for a model to support infinite input size, and the inherent restriction on the size of the context window is eliminated [23].

Hybrid Parallelism: The parallelism techniques we see above are orthogonal to each other. The ever-increase demand to scale model capacity can benefit from effectively combining these techniques to counteract their individual trade-offs. It can enable very large language models with trillions of parameters but ultimately requires well-architected application to capitalize on its potential.

Automatic Parallelism: Diversity in the LLM space has outstripped the utility of a generic parallelism technique. A system to create the best plan of action given the model architecture and heterogenous deployment environment is required to gain maximum utilization from the scarce resources. This method automatically combines the other techniques to create such a parallelization plan during runtime.

3. Data Parallelism

Data parallelism, as explained in Section 2, is the process wherein a large number of servers hold the entire model within them, and each server is given a subset of the input data. After every iteration of the input data, the parameters in the model of all the servers are updated, such that all the

servers models are consistent [6]. The high level operation of Data parallelism is shown in Figure 3. This section takes a closer look at different methods for achieving data parallelism - *Parameter Server, AllReduce, Dynamic Communication Thresholding - Data Parallel (DCT-DP) and Fully Sharded Data Parallel (FSDP).*

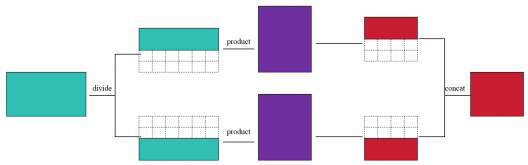


Figure 3. Data Parallelism [34]

3.1. Parameter Server

The architecture of a Parameter Server data parallel method is as seen in Figure 4. The servers are divided into two groups - server groups and the worker groups. The server group is responsible for collating the gradients that are sent by the worker groups, during every iteration, and also send these updated parameter values back to the worker group. The worker group stores the entire model which has to be trained, and each worker is given access to a distinct subset of the input data, which is used to train the model. The worker groups, after they process these input data send the individual gradients obtained back to the server groups. [21]

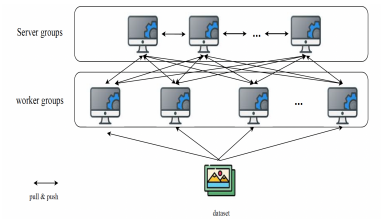


Figure 4. Parameter Server Data Parallelism [34]

As can be clearly seen from the above architecture, the network bandwidth needed to support this mechanism should be very high. We also see that, this is a centralized architecture, where there is a single point of failure (the server group). Thus, in order to rectify these limitations, the AllReduce architecture of data parallelism was proposed.

3.2. All Reduce

The All Reduce method employs a decentralized approach for syncing the parameters within the model. Unlike the parameter-server method, in the case of the All Reduce, all servers within the network have access to input data and this input data is partitioned accordingly. Each server is responsible for collating a certain number of parameters within the model, and after such collation, the server shares the updated value to all the servers within the All Reduce network.

In the naive method of All Reduce, all servers are able to communicate with everyone else on the network. Thus, for collation of the parameter within the model, an individual server must receive the values from all the other nodes in the network, and similarly, for updating the value, an individual node must send the updated value to all the other nodes in the network. Thus, the naive flavor of All Reduce has an overhead of high communication bandwidth [10].

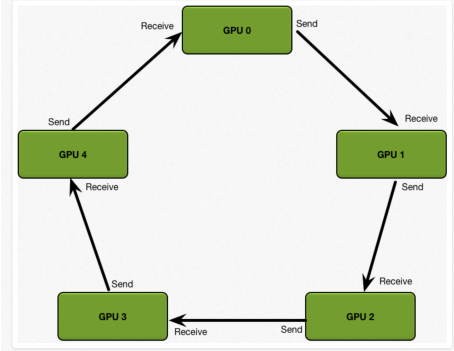


Figure 5. Ring All Reduce [10]

A more optimized version of the All Reduce is the ring All Reduce, which is shown in Figure 6. Here, the nodes that take part in training the LLM are organized in a ring fashion, and each node communicates it's most recent updated value of the parameter to it's neighbor. By repeating this cycle N-1 times (where N is the total number of servers in the Ring All Reduce), all the nodes within the network would have updated the parameters in their local LLMs, and these updates would be consistent across all the nodes in the network. Thus, the bandwidth needed for communication is substantially reduced when compared to the Naive method [10].

3.3. Dynamic Communication Thresholding - DP

Dynamic Communication Thresholding - Data Parallelism (DCT-DP) is an extension of the Parameter Server method. The above two methods (All Reduce and Parameter-Server) take a large amount of time for communicating the parameters (that is, network bandwidth becomes the bottleneck in the system), and thus, there is a large fraction of time when the GPUs of the worker nodes remain idle, reducing the effectiveness of parallelism. DCT-DP tries to resolve this problem with the above two methods.

It is based on the principle that, during every iteration of training the model, only a small number of parameters have a significant change in their gradients, and thus, only such parameters need to be passed to the server group for collation and updation. The key insight used in this method

is that, the mean of the gradients that are being passed by the workers to the server groups do not vary by a lot, and it is seen that the majority of the gradients lie 26 percent around their mean value. Thus, it is not needed for all the parameter values to be communicated to the server group for every iteration. This technique calculates the threshold value, only above which the parameter values have to be communicated, if they are below this threshold, they can be ignored (and sent as 0).

In order to maximize the performance of this technique, the threshold value is calculated for every few thousand iterations (as we don't expect the threshold value to significantly change during this time). By using these two tricks, it is seen that the speedup of training a real-world model at Meta increased by a factor of 100x [11]. The detailed algorithm for DCT-DP is presented in Figure 6.

Algorithm 1 DCT-DP: Communication-Efficient Data Parallelism

Input: Sparsity factor η ($0 < \eta \leq 1$), Threshold life-span L, Iteration number k, Gradient of the DNN layer $W_{grad} \in \mathbb{R}^N$, Error $E_{grad} \in \mathbb{R}^N$, and Threshold τ (from iteration $k-1$)
Error Feedback: $W_{grad} = W_{grad} + E_{grad}$
if L divides k **then**
  $[w_1, w_2, \cdots, w_N] = Sort(|W_{grad}|)$
  Assign $\tau = w_{\lfloor N \times \eta \rfloor}$
else
  Use τ from iteration $k-1$
end if
Compute mask $M = \mathbb{1}(|W_{grad}| \geq \tau)$
Compute compressed gradient $\widetilde{W}_{grad} = W_{grad} \odot M$
Compute error $E_{grad} = W_{grad} - \widetilde{W}_{grad}$
Send $\widetilde{W}_{grad}$ to the parameter server which updates the model

Figure 6. Algorithm for DCT-DP [11]

3.4. Fully Sharded Data Parallel (FSDP)

Fully Sharded Data Parallel (FSDP) is a method which allows the programmers to train large models efficiently in the PyTorch package. The workflow of FSDP is as shown in Figure 7. This basically has the following 4 units for achieving Data Parallelism:

1. Creating a FSDP Unit
2. Sharding
3. All Gather operation
4. Reduce Scatter operation

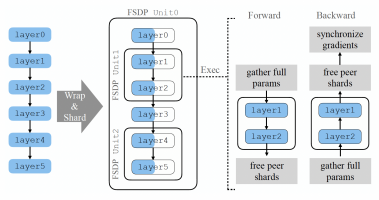


Figure 7. FSDP workflow [35]

The entire model is first subdivided into individual units in FSDP, and these are called FSDP Units. The size and the contents of the FSDP units can be controlled by the programmer, and thus, FSDP allows for maximum optimization (based on the workloads and the architecture of the LLMs that has to be trained).

Sharding stores all the parameters that are present in the FSDP unit in a structure called *FlatParameter*. This flat parameter is then split and stored in multiple nodes within the GPU. The node is now responsible for the part of the flat parameter that it has stored.

The All Gather operation is a primitive operation that is available in NVIDIA NCCL (NVIDIA Collective Communication Library). This combines all the values that are present in different GPU nodes, and the results are then distributed to all the nodes that have participated in the All Gather operation.

The last step involves the Reduce Scatter operation, which is another primitive that is available in the NCCL. The Reduce Scatter operation, combines all the parameter values, updates the combined parameter value and then scatters the updated values to all the nodes that were involved in this operation [35].

3.5. Limitations

Even though data parallelism is used extensively for training and during inference of LLMs, it suffers from two major limitations, which are:

1. Data Parallelism requires each server to have the entire LLM be loaded in it's memory. With the ever increasing size of the LLMs [2], this precondition cannot be met for any of the contemporary LLMs.
2. To maximize the impact of parallelism, the batch size has to be constantly increased as the current GPUs available have very high FLOPs. However, it is seen that by increasing the batch size, the generalization capabilities of the models decreases. Thus, there is a theoretical maximum for the batch size, and as a result of this, data parallelism cannot maximize the GPU utilization on it's own [18].

As a result of these two limitations, there is a need to come up with new techniques of parallel training and inference, in conjunction with data parallelism, and these are discussed in the subsequent sections of the paper.

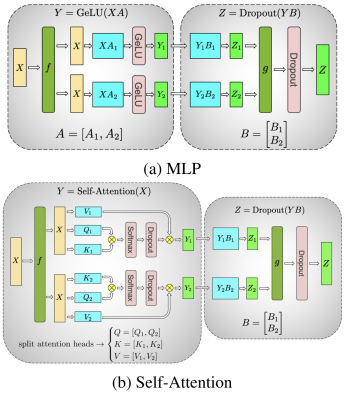
4. Model Parallelism

Model parallelism has emerged as a pivotal technique in the realm of large language models (LLMs), strategically distributing model components across multiple processing units to overcome memory constraints. This method not only facilitates the training of models on larger datasets but also enables the construction of more complex and sophisticated architectures. Transformers, the predominant architecture for LLMs, are particularly adept at leveraging

model parallelism. Their design, which includes the self-attention mechanism, inherently supports substantial parallelism within layers. By distributing these intensive computations across multiple processing units, model parallelism significantly accelerates the training process and enhances the scalability and efficiency of transformers. Model parallelism can be divided into two types:

4.1. Tensor Parallelism

Tensor parallelism is an intra-layer partitioning technique that divides a tensor into N segments along a specified dimension, with each device containing $1/N$ of the tensor. This strategy necessitates additional communication between devices to maintain computational accuracy, as partial results must be exchanged frequently. Essentially, tensor parallelism distributes matrix operations across multiple devices. The communication pattern typically involves an all-reduce operation, which, despite its frequency and substantial data transfer, ensures the correctness of the computation. By partitioning each layer of the model across multiple devices, tensor parallelism effectively distributes the computational load.[34] In transformer models, which are largely composed of matrix multiplications, tensor parallelism allows these operations to be executed across several GPUs. Attention blocks and multi-layer perceptron (MLP) layers are crucial elements of transformers that can effectively utilize tensor parallelism. In multi-head attention blocks, each head or set of heads can be distributed across different devices, enabling independent and parallel computation.



(a) MLP

(b) Self-Attention

Figure 8. Blocks of Transformer with Tensor Parallelism [30]

4.2. 1D Tensor Parallelism in Megatron-LM

The Megatron framework[30] implements a specific approach to tensor parallelism for transformer-based models. Within the multi-layer perceptron (MLP) component, the operation $Z = (XA)B$ is handled by first dividing the matrix multiplication XA along the column dimension, resulting in each device holding a partial output (e.g., $Y_0 = XA_0, Y_1 = XA_1$, etc.). The subsequent matrix multiplication is partitioned over the row dimension, enabling the column-partitioned inputs to be multiplied to produce partial outputs. These partial outputs then undergo an all-reduce sum to ensure accuracy. In the self-attention block, which includes the QKV linear layer and the MLP, the naturally parallel attention heads can be distributed across devices. By applying column and row parallel linear layers, the self-attention block is partitioned first column-wise and then row-wise across devices. This distribution strategy facilitates the efficient scaling of large model training across multiple devices, optimizing computational resources and enabling the handling of more extensive models.

4.2.1 Limitations of 1D Tensor Parallelism:

While 1D tensor parallelism offers a straightforward approach, it suffers from two key shortcomings that restrict its applicability in real-world scenarios.

1. Limited Scalability on Partially Connected Hardware: This method assumes a uniform communication bandwidth between all devices. However, in many computing systems, even those in supercomputing centers, GPUs might only be partially connected. In such cases, communication between distant devices through the PCIe bus is significantly slower compared to directly connected GPUs. This limited bandwidth can significantly hinder the efficiency of all-reduce operations, a critical component of 1D tensor parallelism.
2. Memory Redundancy: Another drawback of 1D parallelism is the redundant storage of layer inputs and outputs across devices. This redundancy unnecessarily consumes memory and limits the maximum model size trainable on hardware with limited resources. This poses a challenge for the broader adoption of large-scale distributed training across various computing environments.

Alternative Tensor Parallelism Strategies: To address the limitations of 1D parallelism, researchers have proposed more advanced tensor parallelism techniques, including 2D, and 3D variants. These methods partition input, weight, and output tensors differently, leading to advantages in both memory efficiency and communication patterns.

4.3. 2D Tensor Parallelism

To evenly distribute computational and memory loads, 2D tensor parallelism is implemented based on the SUMMA [5] (Scalable Universal Matrix Multiplication Algorithm) approach. SUMMA utilizes a $q \times q$ grid of processors and splits the input matrices into smaller sub-matrices. Specifically, matrix A is divided into q sub-matrices along its

columns, and matrix B is partitioned into q sub-matrices along its rows. Instead of directly computing the final product matrix C, SUMMA treats the matrix multiplication as a summation of a set of outer products. Each processor computes its partial product and contributes to the overall sum to form C.

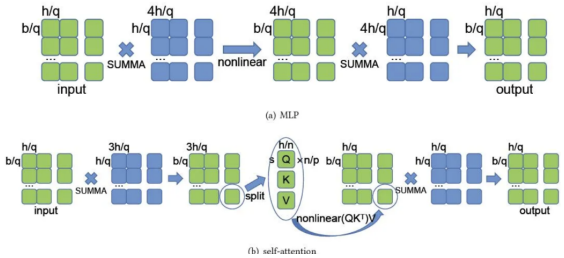


Figure 9. 2d Tensor Parallelism in Transformers [32]

4.3.1 Tesseract

Building upon the concept of 2D tensor parallelism, Tesseract [32]introduces a refined approach for distributing model parameters during training. This method leverages a two-dimensional grid structure, similar to 2D tensor parallelism. However, Tesseract differentiates itself by splitting model parameters along two distinct dimensions: layers and data. This strategic partitioning allows for efficient allocation of computational resources and data across processing units within the grid, potentially leading to further performance improvements for training complex models.

4.4. 3D Tensor Parallelism

3D tensor parallelism emerges as a powerful technique for parallelizing neural network training, aiming to minimize communication overhead. Inspired by 3D matrix multiplication algorithms, it partitions tensors in a cubic fashion for superior communication cost optimization compared to traditional one or two-dimensional approaches. This method strategically splits the first and last dimensions of a tensor, with the first undergoing a double partition. For example, a tensor of shape $[P, Q]$ would be divided into chunks of shape $[P/\sqrt[3]{N^2}, Q/\sqrt[3]{N}]$, where N represents the number of processing units.[1] This cubic partitioning strategy fosters a more balanced distribution of both data and computations across the processing grid, leading to enhanced resource utilization and potentially faster training for large models.

4.5. Pipeline Parallelism

In contrast to MP, pipeline parallelism focuses on dividing the model itself into smaller chunks along the layer dimension. Each processing unit is then assigned a specific chunk, enabling simultaneous computation across devices. During the forward pass, intermediate activations are exchanged between devices for processing in the subsequent layer. Similarly, gradients are backpropagated during the backward pass. While this approach significantly increases training throughput, it can introduce "bubble time" where certain devices might be idle while others are performing computations, leading to potential resource wastage. Furthermore pipelines parallelism can be synchronous as well as asynchronous. The different implementations of model parallelism are explained further.

4.5.1 Synchronous Pipeline: GPipe

GPipe [15] implements a synchronous pipeline paradigm. The input minibatch is divided into micro-batches, with each processing unit holding a copy of its assigned pipeline stage. Gradients are accumulated across micro-batches and applied synchronously at the end of each training iteration. This approach ensures all workers possess the most recent model parameters but requires storing activations proportional to the number of micro-batches. Consequently, GPipe's scalability can be limited when dealing with a large number of micro-batches due to increased memory consumption.

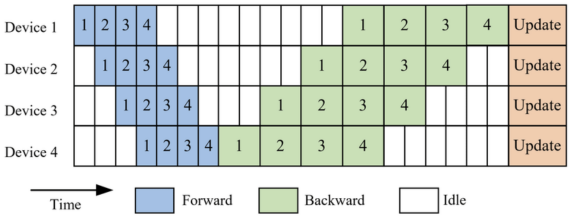


Figure 10. Parallelism in Gpipe [15]

4.5.2 Asynchronous Pipeline: PipeDream

PipeDream [13] prioritizes maximizing hardware utilization and training throughput by adopting an asynchronous pipeline strategy. Mini-batches are continuously fed into the pipeline, with parameters updated asynchronously after each backward pass. To address the issue of weight staleness arising from asynchronous updates, PipeDream utilizes weight stashing. This technique involves maintaining multiple copies of weights, one for each active mini-batch. While PipeDream offers improved throughput and speed-up compared to GPipe, this benefit comes at the cost of increased memory usage due to the requirement of storing multiple weight copies.

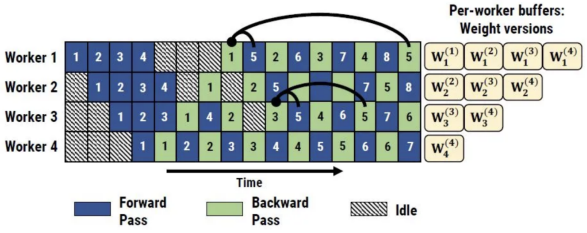


Figure 11. Parallelism in PipeDream [13]

4.5.3 Improved Synchronous Pipeline: DAPPLE

DAPPLE [7] seeks to address the limitations of both GPipe and PipeDream by introducing an improved synchronous pipeline approach. Similar to GPipe, it employs a "one forward, one backward" schedule, resulting in the same number of pipeline bubbles. However, DAPPLE leverages this schedule to limit memory consumption for activations to a single micro-batch size. This approach fosters increased scalability compared to GPipe for training with a larger number of micro-batches. Additionally, DAPPLE demonstrates superior performance compared to PipeDream, achieving up to 3.23x speedup in synchronous training scenarios. It also surpasses GPipe by offering a 1.6x speedup in training throughput while simultaneously saving 12% of memory resources.

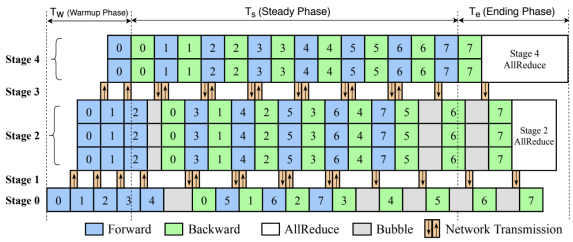


Figure 12. Parallelism in DAPPLE [7]

4.5.4 TeraPipe

Targeting the scalability challenges of training massive Transformer-based language models (LLMs), TeraPipe [25] presents a novel token-level pipeline parallelism algorithm. This approach leverages the inherent properties of the token dimension within these models to efficiently distribute computations across multiple processing devices during synchronous model-parallel training. TeraPipe achieves this by constructing a performance model using a limited set of simple workloads. This model then informs a dynamic programming algorithm that calculates the optimal partitioning strategy for the token dimension within the pipeline. This tailored partitioning, specific to the model architecture and available hardware configuration, unlocks significant training speed-ups compared to traditional methods. By effectively exploiting token-level parallelism, TeraPipe emerges as a powerful solution for training large LLMs, promoting enhanced efficiency and scalability in this domain.

4.5.5 BPipe

A significant challenge in pipeline parallelism lies in ensuring balanced memory usage across pipeline stages. Traditional approaches can lead to uneven memory allocation, where certain stages hold a disproportionate amount of data compared to others. BPipe [19] addresses this issue by introducing a memory-balanced pipeline parallelism strategy. As illustrated in the figure, BPipe operates within a single training step using an evictor-acceptor pair mechanism. In a 4-way pipeline scenario (without BPipe), stage 0 (the evictor) might hold up to four micro-batches of activations, while stage 3 (the acceptor) only holds a single micro-batch. This imbalance creates inefficiencies. BPipe tackles this challenge by enabling the evictor to redistribute its activations to the acceptor before processing the next data batch. This dynamic redistribution ensures a more balanced workload distribution across pipeline stages, promoting efficient memory utilization and potentially accelerating training throughput.

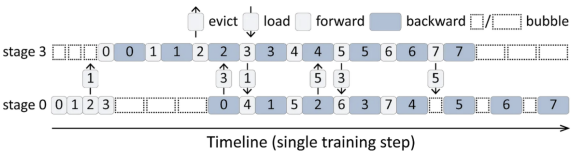


Figure 13. Parallelism in Bpipe [19]

4.5.6 Chimera

The Chimera [22] pipeline parallelism technique offers a compelling approach for training large-scale models. It employs a unique bi-directional scheduling mechanism that reduces idle periods within the pipeline. This strategy consists of two pipelines: a "downward" pipeline processing data forward through the stages, and an "upward" pipeline reversing the direction and processing data backward. By strategically dividing the total micro-batches (N) equally between these pipelines, Chimera enables concurrent execution of multiple micro-batches using a 1F1B (one forward, one backward) strategy within each pipeline. This combined approach significantly improves training efficiency. While Chimera introduces some additional memory requirements due to its bi-directional nature, this potential drawback might be outweighed by its advantages in scalability and performance. The ability to execute mul-

tiple micro-batches simultaneously makes Chimera a valuable tool for tackling the challenges associated with training increasingly complex models.

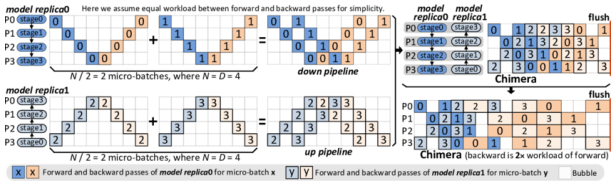


Figure 14. Parallelism in Chimera [22]

5. Expert Parallelism

It is known that scaling model size directly translates into better model performance. With the same compute budget, we can achieve better performance for bigger models on smaller training data rather than training a smaller model on larger training data. Unfortunately, the increase in model size also increases time and compute resources required during training and inference. In this context, Shazeer et al. [29] introduced Sparsely Gated Mixture-of-Experts in 2017. They created a 137B parameter model by introducing thousands of feed-forward sub-networks called experts in the model stacked between LSTM layers. A learnable gating network then activates a sparse subset of these experts for each training example. In this way, each incoming token can compute on different parameters. The authors showed that such a conditional model can achieve 1000x model capacity with only minor losses in computational capacity.

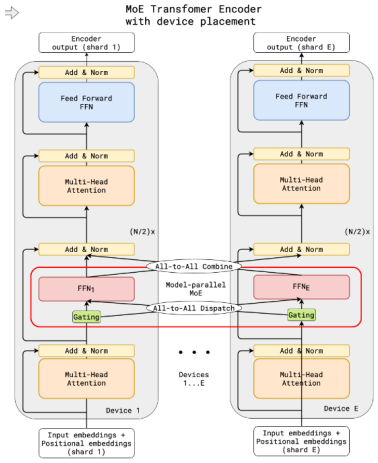


Figure 15. Feed Forward Layer replaced with Mixture of Experts sub-networks. Each expert is sharded onto individual devices/cores.

GShard [20] then used mixture-of-experts in transformers by replacing every feed forward layer with an MoE layer to create a 600 billion parameter model. It introduces expert parallelism by distributing the experts across several devices. To ensure effective parallelism, the tokens need to be distributed evenly across experts while also parallelly running the gating function for multiple tokens. They achieve this by defining an expert capacity to keep the number of tokens processed by each expert below a threshold. The gating function keeps a count of the number of tokens sent to each expert. If any expert exceeds capacity, the token overflows and passes directly to the next layer via residual connections. All tokens are also split evenly into groups which are then processed independently, increasing parallelism. Moreover, to avoid creating popular experts which would imbalance the load, an auxiliary loss is added while training. Finally, the gating function selects top-2 experts for each token but only passes the token to the 2nd expert with probability proportional to its weight. Using these techniques, GShard was able to achieve sublinear scaling of computation and communication costs proportional to the model capacity.

Switch Transformers [8] builds on GShard to deliver a 1.6 trillion parameter model with 2048 experts. Instead of the top-2 routing proposed by its predecessor, the author uses single expert routing i.e. each token is only sent to one expert. This simplification from earlier models reduces router computation as well as halves the computations at each expert. Furthermore, communications costs are also reduced. It also uses a load balancing loss to discourage popularity of some experts. Expert parallelism is employed by deploying each expert on a new core, effectively sharding the expert dimension. But the authors do find that scaling experts produces diminishing returns without employing other scaling strategies such as data and model parallelism. Only by combining all three types of parallelism, they achieve the required trillion parameter scale. Each parallelism is complimentary to the other and needs to be adjusted keeping the other in mind. We will explore this and other types of hybrid parallelism in the next section.

Expert parallelism is only as effective as the load balancing of tokens across the expert layer. FasterMoE [14] shows that skewed expert selection leads to unbalanced load across the cores hosting each expert. This exacerbates when a synchronous all-to-all communication precedes this expert stage. They propose a load aware computational modeling and topology aware communication modeling approach that adapts to variable load and heterogenous networking environments. First technique replicates popular experts on multiple nodes by predicting latency of training iteration. Then, smart scheduling parallelizes communication and computation on individual nodes using groups of workers in a 2-stream approach, akin to pipelining. Finally, a network topology aware gating function directs tokens to

experts with lower latencies.

6. Hybrid/3D Parallelism

Traditional parallelism techniques cannot keep up with the current arms race in LLM of creating larger and larger models. Model sizes with billions and trillions of parameters can only be trained with a huge data-center scale clusters. In such a scenario, it is not enough to use only a single parallelism technique. An effective combination of the above techniques can reap manifold benefits by counteracting the trade-offs of each other. Most models today are trained using this hybrid approach to achieve maximum efficiency.

Megatron-LM [30] combines model parallelism orthogonally with data parallelism in order use them simultaneously. The GPUs are first split into model parallel groups with each running an instance of the distributed model. Then, each GPU position across these model parallel groups form a single data parallel group. Finally, multiple gradient all-reduce operations are run in parallel during back propagation to reduce weight gradients within each data parallel group.

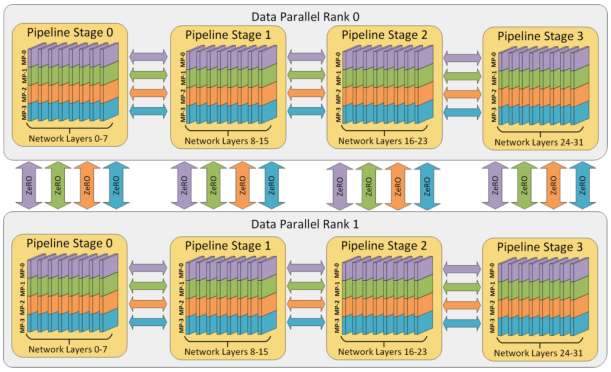


Figure 16. Example of 3D parallelism

DeepSpeed [12] takes it a step further and integrates data, model and tensor parallelism to create trillion parameter models. It aims to balance the communication overhead of model parallelism and high memory requirements of data parallelism. DeepSpeed leverages ZeRo [28] optimizations to scale compute efficiently while network topology aware model slicing improves intra and inter-node communication bandwidth. Figure 16 shows the placement of the various parallel groups in DeepSpeed. Tensor parallel groups are placed within a node to take advantage of the faster bandwidth for communication while pipeline parallel groups can be placed across nodes since they have low communication requirements. Further scaling is achieved by replicating this setup multiple times across data parallel groups. These group communicate independently and in parallel amongst a subset of local workers as show in the figure.

7. Sequential Parallelism

Almost all the LLMs that are currently being developed use transformers, as they give the best results [23]. In the case of transformers, there is a need to calculate self attention to get information about the context embeddings for each word/token in training data or the query. However, we know that calculating self attention scales quadratically with context length. Due to this constraint, the context length would be restricted to the maximum that a single GPU can hold, which would be finite in size. To achieve infinite context length, we need to introduce a new type of parallelism, called sequence parallelism. We see that sequence parallelism is being used extensively, as the context window for Google's Gemini has one million tokens [9].

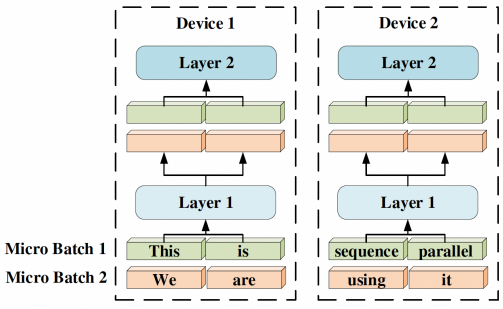


Figure 17. Sequence Parallelism [23]

The working of a sequence parallelism is as shown in the above figure 17. The input sequence is first split into different chunks and each chunk is fed to a different node (or GPU). The self attention scores for individual tokens are calculated based on a technique called *Ring Self Attention*. This is very similar to the All Reduce Ring method that is described in section 3 (data parallelism). The individual nodes calculate the self attention of the tokens that are fed into it, by logically arranging themselves in a ring, and communicating the updated self attention scores to their neighbors. Once enough rounds are complete, all the necessary information (the K, Q and V values) needed for calculating the self attention score of an individual token is obtained b y all the nodes, and thus, self attention is calculated [23].

It has to be noted that sequence parallelism is compatible with data, tensor and pipeline parallelism, and hence, the authors of the paper who proposed sequence parallelism also call this 4D parallelism (i.e., data, tensor, pipeline and sequence).

8. Automatic Parallelism

Automatically searching for the best strategy for distributed training can be advantageous to support a diverse set of requirements in model architecture and heterogeneous computing environment. FlexFlow[17], OptCNN[16], Tofu[33]

and TensorOpt[4] combine tensor and data parallelism in their search space, while PipeDream and DAPPLE use a combination of pipeline with data parallelism. Nonetheless, these techniques are limited in combining data parallelism with at most one single model parallelism, reducing their available search space.

Galvatron[26] on the other hand, can leverage any of data parallelism, shared data parallelism, tensor parallelism and pipeline parallelism. It first defines the search space considering these various parallelism techniques. Then, it creates a decision tree to represent every possibility and narrows it down using heuristics. Finally, a dynamic programming algorithm is used to find the optimal plan of action for the given input conditions.

9. Conclusion

This literature survey has covered the different types of parallelisms that are used while training and during inference of a Large Language Model (LLM), which arose due to the increasing complexities of the current LLMs. It delved deeper into different types of parallelisms that are used in the industry currently. It has to be noted that, in a production setting, the LLMs are trained and inference is done, not by an individual parallelism technique but by a combination of such methods. The choice of the parallelisms used and the techniques that are employed to achieve these parallelisms depends on the underlying data which is used for training and also depends on the architecture of the LLMs that are being trained.

References

[1] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. Maximizing parallelism in distributed training for huge neural networks. *arXiv preprint arXiv:2105.14450*, 2021. 6

[2] Felix Brakel, Uraz Odyurt, and Ana-Lucia Varbanescu. Model parallelism on distributed infrastructure: A literature review from theory to llm case-studies, 2024. 1, 4

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. 1

[4] Zhenkun Cai, Xiao Yan, Kaihao Ma, Yidi Wu, Yuzhen Huang, James Cheng, Teng Su, and Fan Yu. Tensoropt: Exploring the tradeoffs in distributed dnn training with auto-parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 33(8):1967–1981, 2021. 10

[5] Justus A Calvin, Cannada A Lewis, and Edward F Valeev. Scalable task-based algorithm for multiplication of block-rank-sparse matrices. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–8, 2015. 5

[6] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012. 3

[7] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021. 7

[8] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. (arXiv:2101.03961), 2022. arXiv:2101.03961 [cs]. 8

[9] Chaim Gartenberg. What is a long context window?, 2024. 9

[10] Andrew Gibiansky. Bringing hpc techniques to deep learning. *Baidu Research, Tech. Rep.*, 2017. 3

[11] Vipul Gupta, Dhruv Choudhary, Peter Tang, Xiaohan Wei, Xing Wang, Yuzhen Huang, Arun Kejariwal, Kannan Ramchandran, and Michael W Mahoney. Training recommender systems at scale: Communication-efficient model and data parallelism. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2928–2936, 2021. 4

[12] Alexis Hagen. Deepspeed: Extreme-scale model training for everyone, 2020. 9

[13] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018. 6, 7

[14] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 120–134, New York, NY, USA, 2022. Association for Computing Machinery. 8

[15] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Easy scaling with micro-batch pipel ine parallelism. *proceeding of Computer Science¿ Computer Vision and Pattern Recognition*, 2019. 6

[16] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *ICML*, pages 2279–2288, 2018. 9

[17] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019. 9

[18] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-

batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016. 4

[19] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. Bpipe: Memory-balanced pipeline parallelism for training large language models. In *International Conference on Machine Learning*, pages 16639–16653. PMLR, 2023. 7

[20] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. (arXiv:2006.16668), 2020. arXiv:2006.16668 [cs, stat]. 8

[21] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, page 583–598, USA, 2014. USENIX Association. 3

[22] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021. 7, 8

[23] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective, 2022. 2, 9

[24] Shenggui Li, Hongxin Liu, Zhengda Bian, Jiarui Fang, Haichen Huang, Yuliang Liu, Boxiang Wang, and Yang You. Colossal-ai: A unified deep learning system for large-scale parallel training. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 766–775, 2023. 2

[25] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In *International Conference on Machine Learning*, pages 6543–6552. PMLR, 2021. 7

[26] Xupeng Miao, Yujie Wang, Youhe Jiang, Chunan Shi, Xiaonan Nie, Hailin Zhang, and Bin Cui. Galvatron: Efficient transformer training over multiple gpus using automatic parallelism. *Proceedings of the VLDB Endowment*, 16 (3):470–479, 2022. arXiv:2211.13878 [cs]. 10

[27] Sharan Narang and Aakanksha Chowdhery. Pathways language model (palm): Scaling to 540 billion parameters for breakthrough performance. *Google AI Blog*, 2022. 1

[28] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training, 2021. 9

[29] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. (arXiv:1701.06538), 2017. arXiv:1701.06538 [cs, stat]. 8

[30] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. (arXiv:1909.08053), 2020. arXiv:1909.08053 [cs]. 5, 9

[31] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, Mosharaf Chowdhury, and Mi Zhang. Efficient large language models: A survey, 2024. 1

[32] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. Tesseract: Parallelize the tensor parallelism efficiently. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022. 6

[33] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019. 9

[34] Fanlong Zeng, Wensheng Gan, Yongheng Wang, and Philip S. Yu. Distributed training of large language models. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 840–847, 2023. 2, 3, 5

[35] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023. 4

Paper	Scheme	Year	Salient Features
Scaling distributed machine learning with the parameter serve	Data Parallelism	2014	The nodes are split into servers and worker groups. Servers are responsible for parameter updation and workers are responsible for gradient calculation of data batches
Bringing HPC techniques to deep learning	Data Parallelism	2017	The All Reduce - naive and ring tries to reduce the total amount of communication between nodes to achieve consistency within the nodes
Communication-efficient model and data parallelism	Data Parallelism	2021	Introduces thresholding and the concept of Dynamic Communication Thresholding to reduce the communication between the nodes during data parallelism
Pytorch fsdp: experiences on scaling fully sharded data parallel	Data Parallelism	2023	"Introduces FSDP which has FSDP Unit, Sharding, All Gather and Reduce Scatter to achieve data parallelism"
PipeDream	Pipeline Parallelism	2018	Asynchronous pipeline with high throughput but suffers from increased memory usage due to weight stashing.
GPipe	Pipeline Parallelism	2019	Synchronous pipeline with high memory requirements due to storing all activations across micro-batches.
DAPPLE	Pipeline Parallelism	2020	Improved synchronous pipeline offering better scalability and memory efficiency compared to GPipe and PipeDream.
TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models	Pipeline Parallelism	2021	"Token-level pipeline parallelism for training large language models, achieving speedups through dynamic partitioning."
Chimera: Efficiently Training Large-Scale Neural Networks with Bidirectional Pipelines	Pipeline Parallelism	2022	"Bi-directional pipeline scheduling that reduces idle periods and improves efficiency for large models, with some additional memory overhead."
Memory-Balanced Pipeline Parallelism	Pipeline Parallelism	2023	Addresses memory bottleneck in pipeline parallelism by enabling balanced workload distribution between stages.
Scalable task-based algorithm for multiplication of block-rank-sparse matrices	Tensor Parallelism	2015	2D tensor parallelism distributes computation and memory load across a q x q processor grid using SUMMA's matrix splitting
Maximizing Parallelism in Distributed Training for Huge Neural Networks	Tensor Parallelism	2021	3D tensor parallelism partitions tensors cubically (first & last dimensions) for efficient communication (inspired by 3D matrix multiplication).
Tesseract: Parallelize the Tensor Parallelism Efficiently	Tensor Parallelism	2022	Tesseract refines 2D tensor parallelism by splitting model parameters across layers and data on a 2D processing grid.
Sequence parallelism: Long sequence training from system perspective	Sequence Parallelism	2022	The input sequence are split into chunks and each node is responsible for these individual chunks
GShard	Expert Parallelism	2020	"Introduce Mixture-of-Experts to Transformers, top-2 gating"
Switch Transformers	Expert Parallelism	2022	"Top-1 gating, simplified and faster, hybrid parallelism"
FasterMoE	Expert Parallelism	2022	"Optimised utilization of MoE layers by dynamic shadowing, smart scheduling and topology aware gatinf function"
Megatron-LM	Hybrid Parallelism	2020	Model (tensor splicing) + Data parallelism
DeepSpeed	Hybrid Parallelism	2020	3D (Data + Tensor + Pipeline) parallelism
Galvatron	Automatic Parallelism	2022	"Data, Shared Data, Tensor Pipeline search space, dynamic programming to find optima"

Table 1. Overview of Parallelism in LLMs