Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency

Ziming Liu*
liuziming@comp.nus.edu.sg
National University of Singapore

Shenggan Cheng*
shenggan@comp.nus.edu.sg
National University of Singapore

Haotian Zhou
zhou0000@comp.nus.edu.sg
National University of Singapore

Yang You
youy@comp.nus.edu.sg
National University of Singapore

ABSTRACT

Large-scale language models have become increasingly challenging and expensive to train. Among various methods addressing this issue, Pipeline Parallelism has been widely employed to accommodate massive model weights within limited GPU memory. This paper introduces Hanayo, a wave-like pipeline parallelism strategy that boasts a concise structure and practical applicability, alongside a high-performance pipeline execution runtime to tackle the challenges of pipeline strategy implementation. Hanayo mitigates the issues of pipeline bubbles and excessive memory consumption prevalent in existing schemes, without resorting to model duplicates as in Chimera. Our evaluation, conducted on four distinct computing clusters and involving both GPT-like and BERT-like architectures with up to 32 GPUs, demonstrates up to a 30.4 % increase in throughput compared to the state-of-the-art approach.

CCS CONCEPTS

• **Theory of computation** → **Parallel algorithms**; • **Computing methodologies** → *Neural networks*..

KEYWORDS

distributed deep learning, pipeline parallelism, large scale training, high performance computing

ACM Reference Format:
Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. 2023. Hanayo: Harnessing Wave-like Pipeline Parallelism for Enhanced Large Model Training Efficiency. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3581784.3607073

1 INTRODUCTION

Over the past decade, deep learning has made significant strides in numerous fields, including Computer Vision (CV) and Natural Language Processing (NLP). Among various architectures, the transformer [35] has emerged as a prominent model due to its exceptional sequence modeling capabilities. Recent studies have demonstrated that transformers not only surpass recurrent neural networks in NLP [5, 27] but also outperform many convolutional neural networks in CV [6, 20]. It has been shown that substantial performance gains can be attained by utilizing large-scale datasets in conjunction with expansive transformer-based models.

In the past six years, the number of model parameters has increased 40-fold every 18 months, while in the last three years, it has grown 340-fold within the same period. Currently, models can contain hundreds of billions of parameters [3]. This rapid increase in model parameters outpaces the memory expansion of accelerators, necessitating the use of large-scale GPU supercomputing clusters for training. As a result, the time and financial costs of training large models have escalated, becoming almost prohibitive, as exemplified by the Megatron-Turing NLG 530B model [34] developed by Microsoft and NVIDIA, which required approximately three months to train on over 2,000 A100 GPUs.

Several challenges arise from the continuous growth in model sizes, including: 1) *Memory Wall*, where the model's parameter size significantly exceeds the storage capacity of a single accelerator, often by several orders of magnitude; 2) *Scaling Wall*, which arises when training large models necessitates the use of thousands of accelerators, resulting in complex parallel patterns and extensive communication that can lead to bottlenecks in scaling; 3) *Computational Wall*, referring to the immense computational power demanded by large models and massive datasets; and 4) *Development Wall*, where the intricate parallel strategies and manual control of communication processes render the development of large model training exceedingly difficult.

Facing the above challenges, the mainstream approach for training large models involves employing model parallelism techniques. In contrast to data parallelism, where each device contains a full set of model parameters, model parallelism distributes the parameters across different devices. There are two primary model parallelism methods: tensor parallelism and pipeline parallelism.

Pipeline parallelism focuses on parallelization at the layer level, with layers assigned to different devices. While tensor parallelism is associated with significant communication costs, pipeline parallelism relies on peer-to-peer communication for transferring intermediate activations, resulting in considerably lower communication overhead. This makes pipeline parallelism an indispensable strategy for large model training. For instance, in the parallel strategy employed by Megatron-LM [22], tensor parallelism is utilized within

*Both authors contributed equally to this research.

SC '23, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0109-2/23/11.
https://doi.org/10.1145/3581784.3607073

nodes (intra-node), while pipeline parallelism is applied across nodes (inter-node).

Nonetheless, the current pipeline parallelism approach has several limitations that diminish the overall efficiency of large model training: 1) *Bubble*, which refers to the idle time of devices due to computation dependencies in the pipeline across different devices; 2) *Communication Overhead*, even though pipeline parallelism employs point-to-point (P2P) communication, its overhead remains considerable; 3) *Memory Consumption*, some pipeline schemes attempt to mitigate the bubble issue by relying on model replication[18]. However, this approach exacerbates the already stringent GPU memory constraints, which might further complicate the training of large models; 4) *Hybrid Programming Paradigm*, since pipeline parallelism is incompatible with the prevalent SPMD (single program, multiple data) programming paradigm for distributed training, it poses challenges for flexible pipeline process scheduling within modern deep learning frameworks.

To address the aforementioned challenges, we introduce Hanayo, a unified pipeline parallelism framework featuring a wave-like pipeline scheme. 1)To break the memory wall, Hanayo decouples the reduction of bubble ratio with model replication, requiring the same level or lower memory compared with mainstream methods; 2)For scaling, the Hanayo unified framework enables the expression of mainstream pipeline parallel algorithms in a universal manner while facilitating further generalization that enables us to automatically scale pipelines to more devices; 3)For computation, we lower the overall bubble ratio with our unique pipeline scheme, wave pipeline, reducing the idles and achieving high throughput with the same computing power; 4)For easier development, we orchestrate the training process using an *action list*, allowing for support of virtually all pipeline parallel algorithms within the runtime system. Besides, we have also implemented efficient communication schemes with techniques such as pre-fetching.

In summary, our paper presents the following contributions:

- We introduce a wave-like pipeline scheme that achieves a low bubble ratio and high performance in large model training. It can achieve increasingly higher throughput as the number of waves increases.
- Hanayo proposes a unified framework for pipeline parallelism. Through theoretical analysis, we obtain a unified performance model for pipeline parallelism.
- In the design and implementation of the runtime system, we aim to decouple the runtime system from specific pipeline parallel algorithms. Utilizing the action list, Hanayo's runtime system can support nearly all pipeline parallel algorithms while optimizing performance through features such as asynchronous communication.
- We conduct experiments with mainstream GPT-style and BERT-style models, performing performance tests for various model sizes on four different computing clusters. Experimental results demonstrate that Hanayo achieves up to a 30.4% performance improvement over the current state-of-the-art pipeline parallelism implementation, Chimera.

2 BACKGROUND

During deep neural networks (DNN) training, the main computations are divided into two processes: *forward propagation* (FP) and *backward propagation* (BP) . In forward propagation, we calculate the loss of different samples operated by the model using the objective function. In backward propagation, we backpropagate the loss using the chain rule of differentiation to calculate gradients, which are used to update the parameters[16, 32].

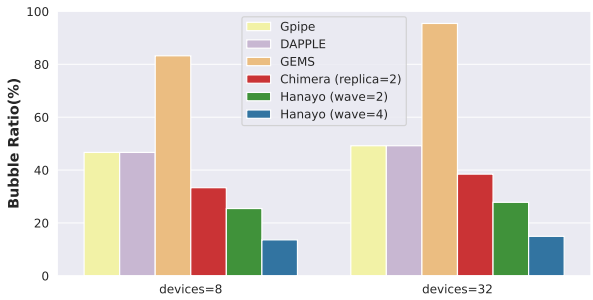


Figure 1: The theoretical bubble ratio of synchronous pipeline schemes

2.1 Parallelism Technique for Training

As DNN training data is often massive, for example, GPT-3 was trained with 45 TB of text data[37]. Therefore, we need to use *Data Parallelism* (DP) [11, 17, 33, 41] to accelerate DNN training. In forward propagation, each device has a complete and identical model for computing. In backward propagation, each device computes gradients and synchronizes them through all-reduce to update the parameters.

However, as the size of DNN models grows, it becomes impractical to train them using data parallelism alone due to the limited memory on each device. Model parallelism has been introduced to address this issue. Model parallelism distributes the model parameters to different devices to achieve lower memory consumption. Based on the different ways of dividing the model parameters, we can generally categorize Model Parallelism into *Tensor Parallelism* (TP)[33, 36, 38] and *Pipeline Parallelism* (PP)[7, 10, 12, 18, 39, 40].

Tensor parallelism means splitting a tensor into chunks along a specific dimension and each device only holds a part of the whole tensor while not affecting the correctness of the computation graph. It involves distributing an operator's parameters to different devices. Each device then computes a local result based on its assigned data slices. Finally, at the end of the operator computation, collective communication (such as all-gather or all-reduce) is inserted to obtain the final result. Despite its advantages, tensor parallelism comes with a higher communication overhead because of the need for synchronizing communication after each split tensor operation. This effect is particularly pronounced during cross-node training, where the low communication bandwidth can significantly reduce the training speed.

To reduce the communication volume between nodes, pipeline parallelism has been proposed. Pipeline parallelism partitions the model at the layer level, while also partitioning the mini-batch into

micro-batches. Each worker is treated as a pipeline stage for the forward propagation or backward propagation computation of a micro-batch.

2.2 Synchronous Pipeline Parallelism

To enable a more comprehensive analysis of the pros and cons of state-of-the-art pipeline methods, we have standardized the symbolic representation, which is presented in Table 1. In this section, we will focus on the major synchronous pipeline parallelism algorithms, such as GPipe [12], DAPPLE [7], and Chimera [18], and we will also introduce some asynchronous approaches.

Table 1: Meanings of the symbols that are used in this paper

S	The number of pipeline stages
B	The number of micro-batches in a single iteration
D	The number of replicated pipelines
P	The number of workers used in the pipeline
W	The number of waves in a single forward/backward iteration(=S / (2 * P))
M_w	Memory consumption for the weights of one stage
M_a	Memory consumption for the activation of one stage
T_F	Time cost for a complete forward pass (all forward stages added together) divided by P
T_B	Time cost for a complete backward pass (all backward stages added together) divided by P
T_C	Time cost for a single P2P communication

The most classic pipeline parallel algorithm is GPipe, as illustrated in Figure 3(a). In this example, the minibatch is divided into four micro-batches for the four devices shown in the figure. The four devices pipeline the forward computation of the four micro-batches and then pipeline the backward computation of the four micro-batches after the completion of all forward computations. The pipeline is divided into three main parts. The first part is at the beginning of the forward computation, where a device must wait for the previous device to complete the calculation of its corresponding microbatch. The second part is between the forward and backward computations, where a device is waiting for the corresponding microbatch to be calculated after completing the forward computation. The third part is after the backward computation is finished, where the device that finished the backward computation must wait for the flush. GPipe is a relatively simple and efficient pipeline parallel algorithm. However, it requires all intermediate activations of the microbatch to be saved during training, which results in relatively high memory consumption.

DAPPLE utilizes the One-Forward-One-Backward (1F1B) schedule to enhance the GPipe, making it one of the most widely used pipeline parallelism methods in practical applications. As shown in Figure 3(b), 1F1B adjusts the forward and backward order of different microbatches on different devices. By calculating the backward earlier, 1F1B releases the intermediate activation stored on the device as early as possible, providing it with a certain memory advantage over GPipe. However, we have observed that the memory consumption of 1F1B is uneven across devices: the peak memory usage does not drop on the first device, while it can significantly decrease on the last device. And Megatron-LM has proposed some improvements to the 1F1B algorithm by introducing interleaving, which divides the model and data more finely, creating more opportunities for overlapping and thus reducing the bubble ratio.

Chimera introduces a novel bidirectional pipeline parallelism technique that achieves a smaller bubble ratio and is currently one of the most effective methods for pipeline parallelism, with more balanced memory consumption. In Figure 3(c), we show a typical Chimera pipeline with 2 model replicas on 8 devices, using blue and yellow blocks to mark the forward and backward propagation going downward, and green and orange for those going upward. By storing another slice of the model on each device, a worker can do computation going upward while waiting for the activation from the pipeline going downward. The two pipelines can thus fill in each other's bubbles. In this way, Chimera exhibits a lower bubble ratio compared to prior pipeline methods, indicating higher theoretical training efficiency. Furthermore, the use of bidirectional pipeline reduces memory consumption imbalances across different cards. However, due to its support for bidirectional pipeline, Chimera needs to store a duplicate copy of model parameters in both directions, resulting in twice the memory overhead of other methods. Further analysis and discussion of Chimera can be found in the next section.

In Figure 2, we present a comparative analysis of different approaches with respect to GPU memory consumption and bubble ratio, highlighting their respective advantages and disadvantages. Notably, we factor in the impact of remaining communication time after overlap when calculating the bubble ratio, resulting in a more accurate assessment. The up red arrow in Figure 2 denotes a higher ratio or consumption, which implies inferior performance. Conversely, the down green arrow signifies a lower ratio or consumption, indicating superior performance. K in bubble ratio of Chimera is defined as $K = \frac{P^2}{2} - P$.

2.3 Asynchronous Pipeline Parallelism

The synchronous pipeline parallelism algorithm performs a flush at each computation step, allowing the training process to maintain consistency by using the same version of model parameters for each batch. To achieve a lower bubble ratio, recent works have proposed asynchronous pipeline parallelism algorithms, such as PipeDream [10], WPipe [40], and PipeMare [39].

Unlike synchronous pipeline parallelism, asynchronous approaches remove the flush and allow for more relaxed dependency constraints. As a result, they tend to have a lower bubble ratio and higher performance, as illustrated in Figure 4. However, asynchronous optimization can impact the convergence process of training. Although there have been works to improve asynchronous convergence[42], they often come with extra computation or memory overhead. So we focus only on synchronous pipeline parallelism in this work. Nevertheless, the strategies and optimizations we propose can also be applied to asynchronous pipeline parallelism implementation.

Pipeline Schemes	Bubble Ratio	Weights Meomory	Activation Memory
GPipe	$\frac{(P-1)(T_B+T_F)+(2P-2)T_C}{(2P-1)(T_B+T_F)+(2P-2)T_C}$ ↑	M_w ↓	PM_a ↑
DAPPLE	$\frac{(P^2-P)(T_B+T_F)+(3P^2-5P+2)T_C}{(2P^2-P)(T_B+T_F)+(4P-6)T_C}$ ↑	M_w ↓	$[M_a, PM_a]$ ↓
GEMS	$\frac{(P^2-P)T_B+(P-1)T_F+(P^2-1)T_C}{P^2T_B+(2P-1)T_F+(P^2-1)T_C}$ ↑↑	$2M_w$ ↑	M_a ↓↓
Chimera(2 Duplicate)	$\frac{(3K+6P-2K/P-8)T_C+2KT_B}{(3P^2-2P)T_C+(2P^2-2P)T_B+P^2T_F}$ ↓	$2M_w$ ↑	$[(P/2+1)M_a, PM_a]$ —
Hanayo	$\frac{\frac{1}{W}T_B+(1+2W+\frac{2}{P}+\frac{P-2}{3})T_C}{\frac{P}{P-1}T_F+(\frac{1}{2W}+\frac{P}{P-1})T_B+(\frac{P-2}{2}+4W)T_C}$ ↓↓	M_w ↓	$[\frac{(2W-1)P+1}{2W}M_a, PM_a]$ —

Figure 2: Comparison of different SOTA approaches
Green arrows indicate better performance

3 HANAYO UNIFIED FRAMEWORK

3.1 Motivation

As previously discussed, pipeline parallelism faces two significant challenges, namely memory consumption and computation efficiency. While GPipe provides a simple and easy-to-implement solution, it suffers from high activation consumption and a high bubble ratio. DAPPLE addresses the memory consumption challenge by modifying the computation schedule, resulting in lower memory consumption. However, it still struggles with a high bubble ratio. Chimera proposes a promising solution with its bi-directional pipeline approach that orchestrates multiple pipelines and achieves an impressive low bubble ratio. But Chimera requires model replicas, which may not be feasible for limited GPU memory or large models. This leads us to consider an alternative approach: can we rearrange the computation scheme so that we can benefit from the low bubble ratio without extra memory consumption?

3.2 Transforming into Wave-like Pipelines

We have found a simple way that leads us out of this dilemma. We know that the high efficiency of Chimera can be primarily attributed to its bidirectional pipeline structure, allowing pipelines in different directions to compensate for bubbles. The reason for employing model replication is that, in the current pipeline scheduling, the same micro-batch must continuously perform calculations and communication in the same direction. Therefore, when introducing calculations in another direction, another set of models must be stored on the GPU. To address this issue, we only need to enable a single pipeline to change direction during the computation process, transforming it into a wavy-shaped pipeline.

Here we have a typical bi-directional Chimera pipeline with a depth of 4, as illustrated in Figure 5. The darker and brighter blocks represent computations performed by the two pipelines with different directions, which we will refer to as $Pipe_{bright}$ and $Pipe_{dark}$, respectively. In this specific case, micro-batch 0 and 1 are $Pipe_{bright}$ as their forward propagation goes downward, while micro-batch 2 and 3 are $Pipe_{dark}$ as their forward propagation goes upward. As depicted in the figure, if we swap all the $Pipe_{bright}$ blocks on devices P2 and P3 with the corresponding $Pipe_{dark}$ blocks located at symmetrical positions on devices P0 and P1, we obtain two identical wave-like pipeline structures. Since the order of computation remains unchanged and the communication overhead is reduced due to the local communication introduced by this swap operation, we can infer that the efficiency of these two wave-like pipelines is at least as good as, if not better than, the original 4-stage Chimera configuration. By adopting this approach, we eliminate the need for model replication in the pipeline implementation, and the replicas employed by Chimera can now be considered as standard data parallelism. There is also an example in figure3(c) and figure3(d).

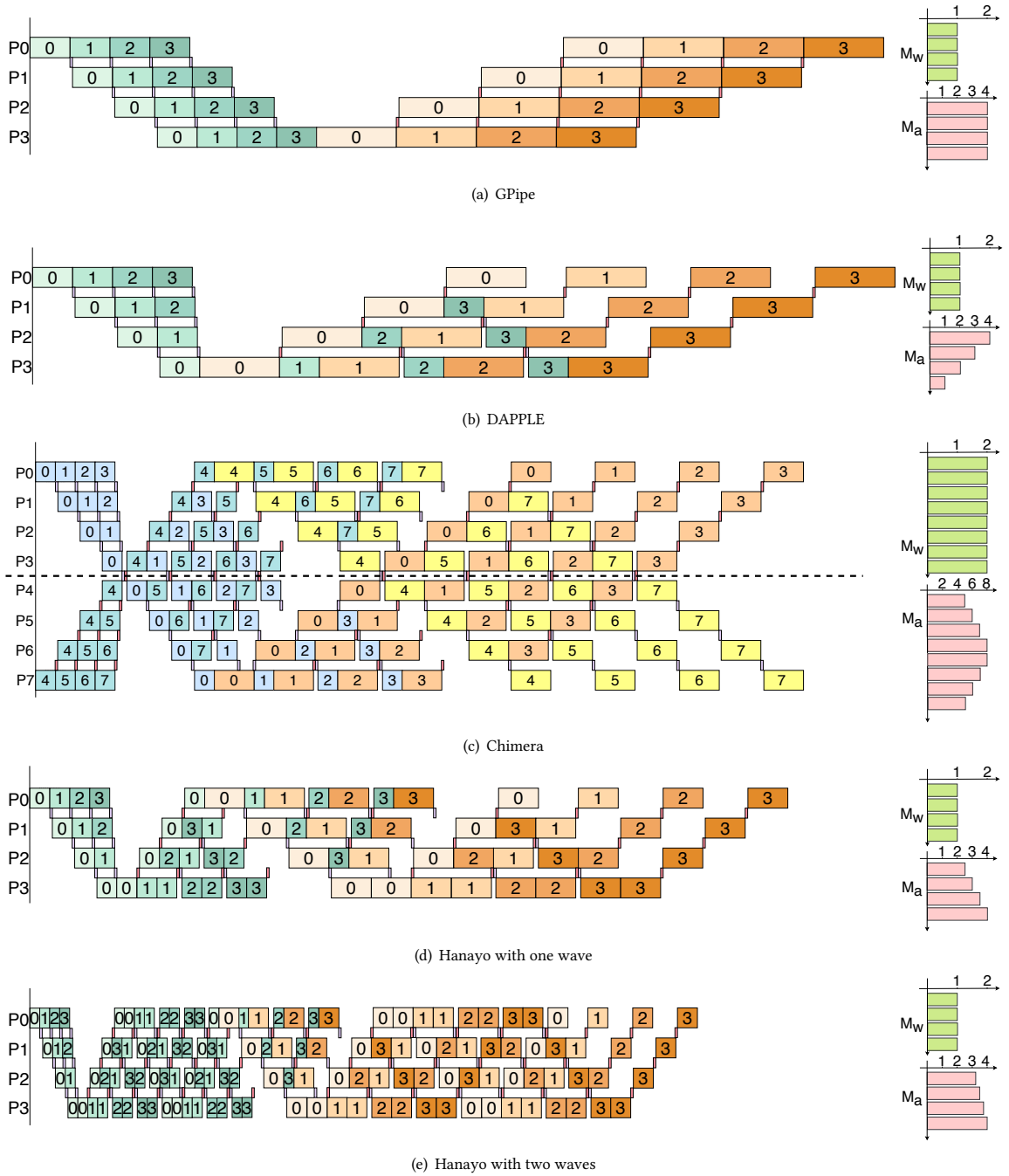
Why we call such pipeline scheme wavy may not be quite obvious in this case, so we have another example in figure 6(a). We mark the whole training process of micro-batch 1 with the color blue and red. There are four "V"s in the whole training process for each micro-batch, which look exactly like waves. In this paper, we define the number of "V"s in a forward or backward propagation process the number of waves. So we would say that there are two waves on 8 devices in figure 6(a).

To ensure a fair and convenient comparison, we will measure Chimera after transforming it into its corresponding wave-like form, which we will call Chimera-wave in this paper. As Chimera-wave can optimize Chimera by reducing cross-communication, it enables us to view model duplication as an expansion of the data parallelism scale, allowing us to measure Chimera while consuming the same amount of memory for model weights as other methods.

3.3 More waves For Lower Bubble Ratio

Having successfully eliminated model duplication, we can now focus on reducing the bubble ratio. During the calculation of the bubble ratio for current pipeline schemes, we observe that with a fixed number of devices, the factors that influence the bubble ratio (or the total idle time in the pipeline) are T_F, T_B, and T_C. As T_C is fixed in a given cluster environment, we can investigate the remaining factors, which represent the time cost of a forward stage and a backward stage. By partitioning the model into smaller stages, we can achieve improved performance.

In the wavy pipeline scheme derived from Chimera, the number of stages is already twice as much as that of regular pipelines. What



(a) GPipe

(b) DAPPLE

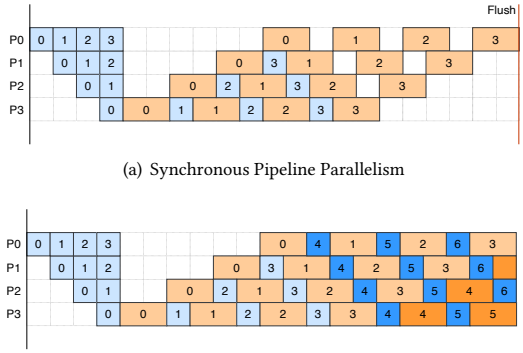
(c) Chimera

(d) Hanayo with one wave

(e) Hanayo with two waves

Figure 3: Synchronous Algorithms of Pipeline Parallelism and their peak memory consumption. The forward propagation process is marked green and the backward propagation process is marked orange. Blue and yellow are also used in Chimera to mark the two directions in it. Back propagation is illustrated twice as long as forward propagation according to the training experience. And we use purple and pink blocks to represent the P2P communication that goes downward and upward. The numbers on the blocks show which of the micro-batches is being processed. Each unit block in M_w represents a whole model weight divided by the number of devices, and one unit block in M_a represents one intermediate activation.

if we continue doubling it? This can be easily achieved by incorporating more waves into the pipeline. In Figure 3(e), the number of stages increases from eight to sixteen as the number of waves rises from one to two. It is evident that the sizes of all the bubbles resulting from waiting for peer devices' computation are halved, while the total computation remains unchanged. Although the number



(a) Synchronous Pipeline Parallelism

(b) Asynchronous Pipeline Parallelism

Figure 4: Comparison of synchronous and asynchronous Pipeline Parallelism

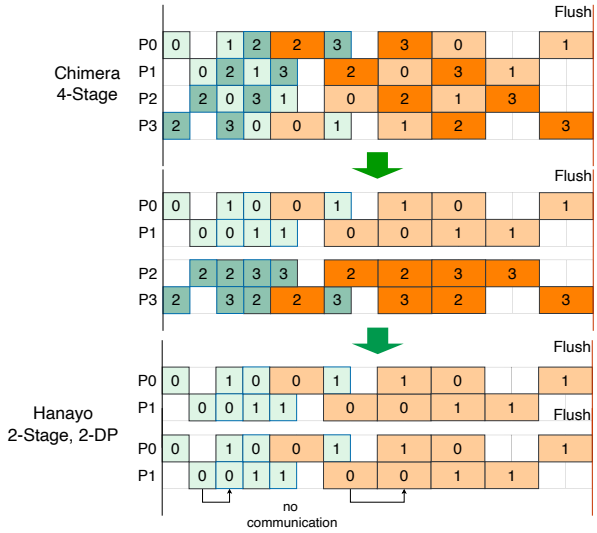


Figure 5: A 4-stage Chimera pipeline can be transformed into two one-wave pipelines with a 2-stage Data Parallel without extra overhead.

of times that we do communication is now doubled, most of them can be overlapped by the computation, and the additional bubbles caused by cross-communication are more than compensated for by the overhead saved by smaller bubble size, yielding a significantly lower bubble ratio.

Furthermore, we can continue to increase the number of waves as long as there are sufficient layers within a single stage to divide. In Figure 6(b), we present an example where we expand the number of waves to four, resulting in the bubble sizes being halved once more. The implementation is quite straightforward, as the structure of the intermediate waves remains identical. Similarly, Hanayo can be scaled to accommodate more devices by employing the corresponding number of mini-batches during the training process. In Figure 6(a), we implement Hanayo on 8 devices with the number of waves set to two. We have highlighted micro-batch 1 to illustrate the direction of a single computation process. The large number of stages may appear confusing; however, there is no need to grapple with the structure, as we have developed an ingenious pipeline framework that can automatically deploy the structure with any desired number of waves or devices. We will discuss this framework in greater detail later.

3.4 Theoretical Analysis

In this section, we delve into the bubbles that impact the performance of Hanayo. There are four distinct types of bubbles in a training iteration, as illustrated in Figure 7. The bubbles in Zone A primarily result from idle waiting for forward activation from peer workers and the overhead of forward activation transmission. Thus, the size of a single bubble in this zone is given by $T_F/2W + T_C$, as the forward time cost is reduced by a factor of $2W$. The bubbles in Zone B mainly arise from the discrepancy between the overheads of forward and backward propagations. Consequently, the bubble size in B is calculated as $\frac{P-LR}{2W}(T_B - T_F) + 2T_C$, where LR represents the local rank. As for bubble type C, it is caused by backward propagations and the corresponding communication, resulting in bubble sizes of $T_B + 2T_C$ and $T_B + T_C$. Lastly, bubbles due to cross-communication occur when using the NCCL[25] backend, which requires batching cross-communication together before initiation to prevent deadlock. By summing all four types of bubbles, we can compute the final bubble ratio:

$$\frac{\frac{1}{W}T_B + (1 + 2W + \frac{2}{P} + \frac{P-2}{3})T_C}{\frac{P}{P-1}T_F + (\frac{1}{2W} + \frac{P}{P-1})T_B + (\frac{P-2}{2} + 4W)T_C} \quad (1)$$

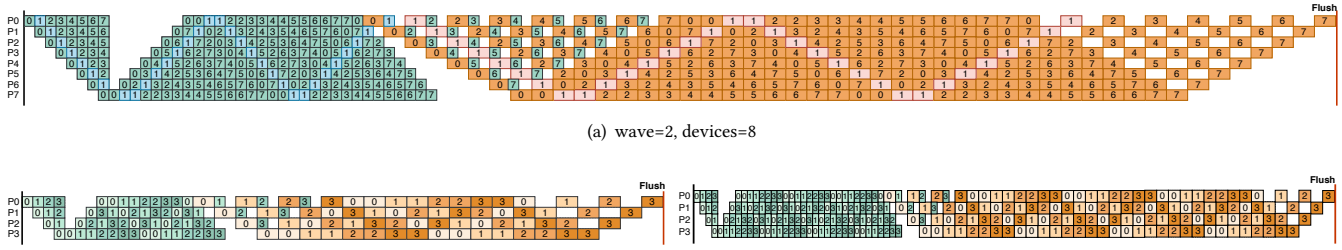
Assuming $T_B = 2T_F$ and disregarding T_C, we can rewrite this equation in a more suitable form as $\frac{2P-2}{3PW+P-1}$. This expression decreases with an increasing number of waves, demonstrating the efficiency of such wave-like structures. Moreover, since we do not employ model replicas or alter the computation order, our M_w and M_a remain consistent with those of other mainstream methods. In figure 1, we show the theoretical bubble ratio for the pipeline schemes. You can see a sharp drop in Hanayo's bubble ratio with an increased number of waves.

4 HANAYO RUNTIME

In this section, we present the design and implementation of our runtime system, which primarily consists of the pipeline execution engine decoupled from the pipeline parallelism scheduling algorithm and communication optimization, including prefetching and asynchronous communication.

4.1 Implementation Scheme

Our implementation scheme is inspired by the widely-used distributed deep learning optimization library, DeepSpeed [30]. DeepSpeed employs a set of instructions, such as ForwardPass, BackwardPass, and SendActivation, in its schedulers. Workers use an interpreter to read the instructions and execute the corresponding actions. We observe that some of these instructions are not well-suited for more complex structures like Hanayo or Chimera. To address this, we break down the instructions into smaller granularities and augment them with target device rank information and local module rank (indicating which part of the model should be utilized for the given instruction).



(a) wave=2, devices=8

(b) wave=2 and wave=4, devices=4

Figure 6: Scaling Hanayo to more devices and waves

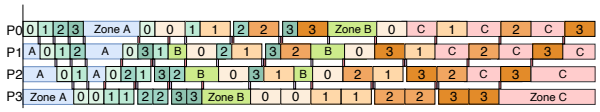


Figure 7: The type of bubbles that exist in a typical Hanayo wave-like pipeline

Unlike some pipeline frameworks that use hooks to establish the relationship between different stages, our approach employs an *action list* to store the instructions for each worker. The scheduler on the master node is responsible for generating the action list based on a specific pipeline. We have provided scheduling algorithms for existing mainstream pipeline schemes and Hanayo, and we also offer interfaces for users to modify existing schemes or develop their own.

4.2 Overlap by Prefetching

One of the main benefits of using such an action list is that the workers can prefetch the next batch of data from peer devices. Our approach employs the pre-fetching technique with asynchronous communication functions. Before initiating a slice of computation, the processor looks ahead to check the next `receive` instruction and launches the subsequent `receive` request before the current forward/backward propagation. This ensures that the activation/gradient for the next round is ready by the end of the current computation round. We know that the transmission of activation and gradients can be costly, especially for large models with high hidden dimensions and computing clusters with poor interconnection. Under such conditions, one key to achieving higher throughput is to maximize the overlap between computation and communication. The prefetching technique employed by Hanayo ensures the least number of bubbles caused by communication and improves overall efficiency. In our implementation, we utilized the NCCL[25] backend and Pytorch Distributed Library, as the NCCL backend offers better support for GPU operations. Moreover, we use the `batch_isend_irecv` function from the NCCL backend to circumvent deadlock in cross-communication.

5 EVALUATION

Our experiments were primarily conducted in four environments:

- The LONESTAR6 cluster from the Texas Advanced Computing Center (TACC). Lonestar6 comprises 560 compute nodes and 16 GPU nodes. The A100 GPUs have 40 GB of high-bandwidth memory. In each GPU node, there are three GPUs, with GPU 0 on socket 0 and GPU 1 and 2 on socket 1.
- A CVM cloud server from Tencent that we rent. The GN10Xp computing node we utilize features an Intel Xeon Cascade Lake 8255C (2.5 GHz) CPU and 8 NVIDIA V100 GPUs with 32GB of memory. The GPUs are connected with NVLink[9], and the interconnect configuration is described in the V100 architecture whitepaper[24].
- A local cluster with 8 NVIDIA A100 GPUs that have 80GB memory. GPU 0 and 1, 2 and 3, 4 and 5, and 6 and 7 are connected with NVLink.
- A local cluster with 8 NVIDIA A100 GPUs that have 80GB memory. The GPUs are fully connected with each other with NVLink.

The models we use for evaluation include two variants of BERT and GPT-3. The BERT-style model consists of 64 layers, 64 attention heads, and a hidden size of 2560, while the GPT-style model has 128 layers, 16 attention heads, and a hidden size of 1024.

We implemented all mainstream synchronous pipeline schemes, including GPipe, DAPPLE, Chimera, and Hanayo. For each setting, we determined the best parallelism configuration by adjusting the devices used in data parallelism and pipeline parallelism to maximize throughput. As previously mentioned, we evaluate Chimera's performance after transforming it into its wave form to ensure fairness with other methods that only use one set of model weights. The model replicas used by Chimera are considered as additional data parallelism.

In the following section, we focus on addressing the following questions:

- What is Hanayo's memory consumption like in practice? Is it suitable for GPUs with smaller memory capacities?
- How adaptable is Hanayo to different computing environments? Can it maintain its superior performance when faced with various computational power and GPU interconnection configurations?
- How does Hanayo perform in weak scaling settings, where the computing resources and tasks increase simultaneously?

- How does Hanayo perform in strong scaling settings, where the goal is to use more computing resources to accelerate a fixed?

It needs to be mentioned that the Chimera that we compare with in evaluation is the optimized wave version, Chimera-wave, which has better performance than Chimera with 2 model replicas. More details are in section 3.2.

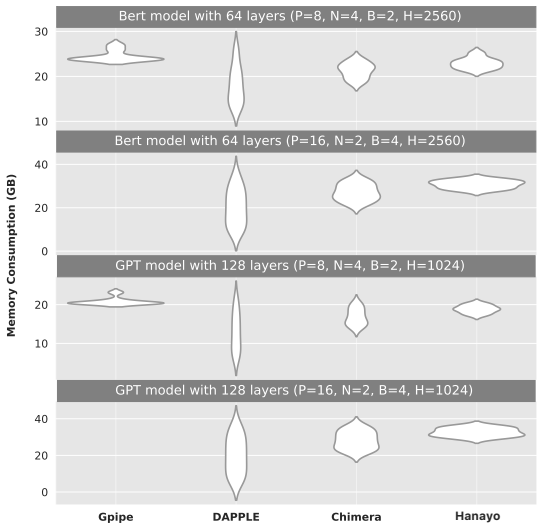


Figure 8: The distribution of peak memory consumption for GPipe, DAPPLE, Chimera, and Hanayo during the training of Bert and GPT model on 32 GPUs of the TACC Lonestar6 cluster

5.1 Memory Consumption

For the two models mentioned earlier, we measured the peak memory distribution across the 32 GPUs utilized during training. The primary factor we need to focus on is the highest peak memory. Although memory consumption varies from device to device, the ability of a scheme to fit within a certain cluster is often determined by the highest peak memory. The computation schemes of GPipe and DAPPLE are relatively similar; however, GPipe requires saving the activation of all micro-batches on every device, resulting in a higher memory consumption for each device. Our findings show that GPipe and DAPPLE have comparable highest peak memory values, but GPipe caused Out of Memory (OOM) errors in two settings. In contrast, Chimera and Hanayo, which benefit from a schedule that consumes the activation as soon as it is generated, achieve lower highest peak memory values.

Another essential aspect to consider is balance. A pipeline scheme with a more balanced memory consumption enables better workload distribution across computational tasks, maximizing the work done on each worker simultaneously and, as a result, achieving higher throughput. GPipe's memory consumption is fairly balanced, with an average variance of **1.33**. However, this low variance comes at the expense of high average consumption. DAPPLE and Chimera do not exceed the memory limit, but their average variances are **16.85** and **2.86**, respectively, which are higher compared to other methods. Our proposed method, Hanayo, maintains the variance as low as **1.44** while exhibiting a similar average consumption as Chimera.

In summary, Hanayo exhibits a similar level of peak memory consumption as the state-of-the-art pipeline schemes while maintaining a more balanced memory consumption profile. This balance leads to higher GPU utilization and, ultimately, improved overall performance.

5.2 Adaptability Across Computing Clusters

Our first throughput evaluation aims to test the performance of different pipeline schemes in various computing cluster environments. As observed in Figure 3, the bubble ratio is determined by the communication overhead and the overhead of forward and backward propagation. Despite our efforts to maximize overlap, some communication cannot be covered by computation. In practice, the computation overhead is typically determined by device computing power, while the communication overhead is influenced by various factors such as GPU connections with NVLink[9], GPU interconnection topological structure, and the bandwidth between computing nodes for cross-node training. In essence, communication overhead is difficult to predict and can vary significantly under different conditions. Therefore, we must evaluate pipeline schemes across diverse environments to assess their adaptability and robustness.

The four selected environments are representative of different use cases. The TACC Lonestar6 computing cluster represents supercomputers or large-scale computing clusters used by companies; the Tencent TVM GPU cloud server represents cloud servers commonly employed for large model training; and the other two clusters are typical local servers used in university labs or small companies. The GPU types include NVIDIA A100 80GB, NVIDIA A100 40GB, and NVIDIA V100 32GB, connected with four distinct topological structures, ensuring diversity in both computation power and communication conditions.

Results are displayed in Figure 9. We limited the number of devices to 8 for this comparison, enabling us to assess performance across clusters. We experimented with two settings: pipeline parallelism only, and a combination of pipeline parallelism size 4 with data parallelism size 2. We did not use a smaller pipeline parallelism size due to the large number of parameters in the BERT model. GPipe and DAPPLE maintain similar throughput across the experiments, as their pipeline stage scheduling primarily differs in activation consumption rather than total time overhead. Chimera outperforms these two methods by approximately 20% in the eight experiments, owing to its bi-directional design. In contrast, Hanayo offers an additional dimension, the number of waves, which we can adjust in the experiments. We explored all possible wave numbers and selected the best-performing one as our result. In the eight settings, Hanayo consistently outperforms Chimera by **15.7%, 30.4%, 23.2%, 29.9%, 8.2%, 17.1%, 24.6%, and 28.0%**, thanks to the wave structures that repeatedly halve the bubble size. This demonstrates Hanayo's advantage over other pipeline parallelism schemes, regardless of the environment.

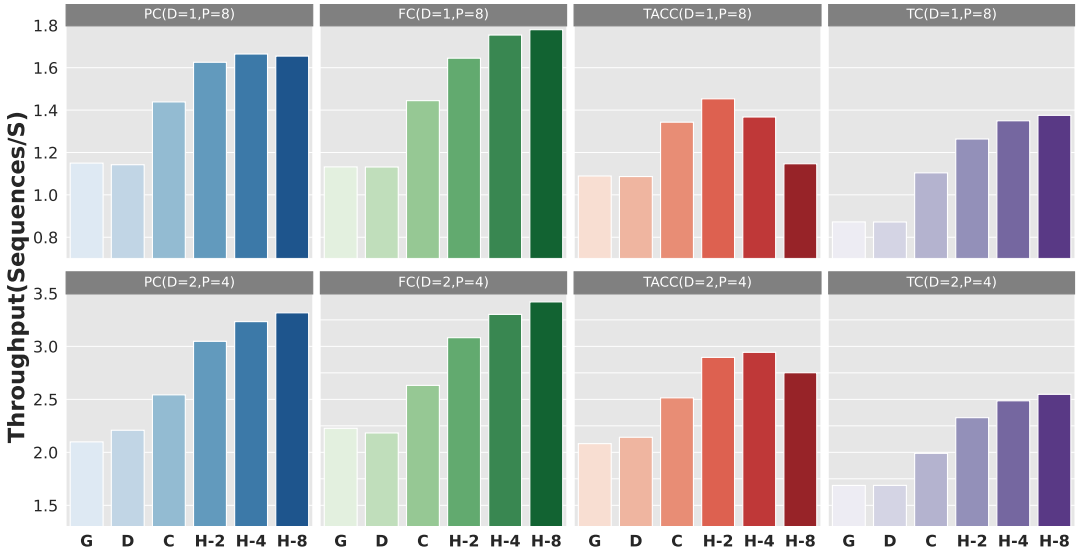


Figure 9: Throughput of training the Bert-style model on totally 32 GPUs from 4 different clusters. PC and FC refer to the two local clusters where the NVIDIA A100 GPUs are partially and fully connected with NVLink. TACC refers to the Lonestar6 cluster from TACC and TC refers to the cloud server of Tencent. As for the methods, G stands for GPipe, D stands for DAPPLE, C stands for Chimera-wave, and H-X stands for Hanayo with X waves.

Another observation from this experiment is that Hanayo's optimal wave configuration can vary with the communication environment. The more waves used, the greater the communication required in one iteration. Although most communication can be covered by computation, there is still more uncovered communication, particularly cross-communication. In servers with full (FC) or partial connections (PC) via NVLink and the Tencent server, which also has NVLink, throughput increases with the number of waves(except for the first setting where the wave number equals 8). This indicates that the improvements brought about by the waves outweigh the additional communication overhead. For clusters with poor interconnection, such as TACC, the optimal wave number will be lower since the extra communication incurs a higher cost.

In conclusion, Hanayo demonstrates superior adaptability across various environments and consistently outperforms state-of-the-art methods. Better topological structures and higher bandwidth further enhance Hanayo's performance.

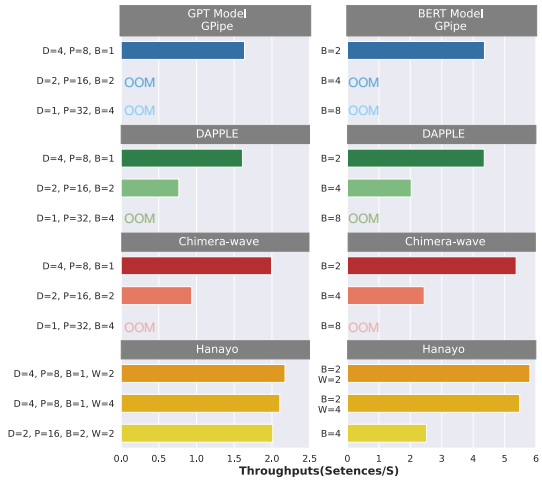


Figure 10: Part of the performance search for the four methods of training the Bert-style model on 32 V100 GPUs from TACC. The configurations with the highest throughput are chosen as targets to be used for further comparison.

5.3 Obtaining The Best Performance For Each Scheme

Before delving into scaling, we first describe how we obtain the best throughput data for each pipeline parallelism scheme. In Figure 10, we present the search space for each method under the setting where 32 GPUs are used to train the GPT and BERT models mentioned earlier, using the TACC Lonestar6 cluster. The batch size is set to 4 and 8 to maximize GPU memory usage. For each method, we tested the pipeline and data parallelism size combinations of (8, 4), (16, 2), and (32, 1). The absence of data in certain areas indicates that the respective method caused an Out of Memory (OOM) error. It is worth noting that we do not list the wave configuration of Hanayo in this figure. Instead, we searched for the best wave number under each parallelism configuration and displayed them in the corresponding locations. Ultimately, we selected the (D=4, P=8) configuration for all methods, as it yields the highest performance, and chose the number of waves for Hanayo as 2.

5.4 Weak Scaling

In this section, we evaluate Hanayo's efficiency under weak scaling settings, where we maintain the same amount of computation per device while incrementally increasing the number of devices used from 8 to 32. The total batch size is increased from 2 to 8. All throughput data were selected using the approach described in the previous section. The results are displayed in Figure 11. In the three sets of experiments, Hanayo outperforms Chimera by 8.19%, 8.11% and 8.13%, DAPPLE by 33.7%, 33.2% and 33.1%, and GPipe by 33.4%, 33.3% and 33.3%. This can be primarily attributed to the reduced bubble size in Hanayo's wave-like structure. Despite the lower bandwidth of GPUs in the TACC Lonestar6 cluster, the advantages of waves still outweigh the extra communication overhead.

Weak scaling is employed to measure a system's ability to maintain computational efficiency as the number of devices and the scale of tasks increase simultaneously. From the results, we can observe that the parallel efficiency is **100.1%** and **99.8%**. The reason why the efficiency exceeds 100% is that GPUs are better at processing batches of data at the same time thanks to their ability of parallel computing. This demonstrates that Hanayo can be scaled to train larger data batches using larger clusters while maintaining reasonable efficiency.

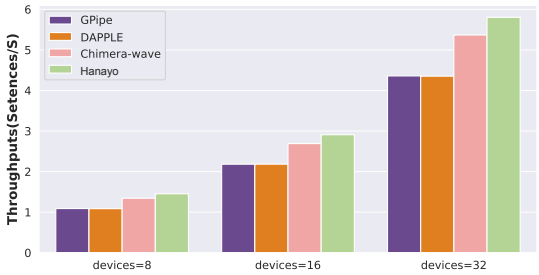


Figure 11: Weak scaling for Bert-style model. The number of devices scales from 8 to 32 while the batch size increases proportionally

5.5 Strong Scaling

Strong scaling is characterized by maintaining a constant task size while increasing the number of processors. In the context of training a large language model, we attempt to use more GPUs to train the model with a fixed data batch. Here, we use a batch size of 4, which already reaches Lonestar6's 40GB memory limit. We then increase the number of GPUs from 8 to 16 and 32, examining whether the total throughput increases proportionally. The results can be found in Figure 12. GPipe and DAPPLE encounter OOM when using 8 GPUs and achieve similar throughput in the other two cases. Hanayo consistently attains the highest throughput in all three tested cases, outperforming GPipe and DAPPLE by 33.3% and 33.8% and Chimera by 8.8%, 8.1% and 8.7%.

Comparing the three sets of data, we can calculate the speedup of Hanayo to be **188.4%** and **337.5%**. This demonstrates its ability to accelerate a specific task with more GPUs. A typical scenario would involve fine-tuning, in which users seek to adjust the released public model weights to achieve better performance on downstream tasks with a small amount of additional training data. As shown in the figure, Hanayo is well-suited to handle this situation.

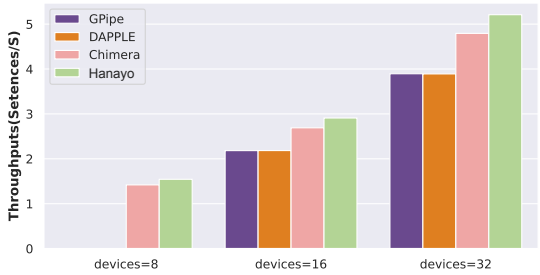


Figure 12: Strong scaling for Bert-style model. We speed up a fixed batch of training with more devices, from 8 to 32.

6 RELATED WORK

Pipeline Parallelism Scheduling Algorithms. There has been a surge in recent research focused on large model training, with some of it centering around the pipeline parallelism scheduling algorithms. Among the earliest efforts were GPipe [12] and PipeDream [10], which remains one of the most widely used Pipeline methods. More recent works such as DAPPLE [7] and Chimera [18] aim to reduce the bubble ratio in pipeline parallelism by utilizing different scheduling strategies, thereby enhancing the efficiency of large model training. WPipe[40] proposed a scheme that achieved better memory efficiency and fresher weight updates in asynchronous pipeline parallelism. The author also posited its potential application in GPipe. Our work builds upon these efforts by presenting a more universal scheduling approach for pipeline parallelism, based on a thorough analysis of state-of-the-art pipeline parallelism methods. In our integrated pipeline framework, we use a performance model with adaptability to choose from various pipeline parallelism strategies to attain optimal performance.

Runtime Systems for Pipeline Parallelism. Many researchers have explored how to build runtime systems that support flexible pipeline parallelism. One early attempt was torchGPipe [14], which implemented GPipe-style pipeline parallelism based on PyTorch's eager execution framework. DeepSpeed [30] and Megatron-LM [23] also support pipeline parallelism in their training engines using a 1F1B style. Fairscale [1] provides experimental support for pipeline parallelism using PyTorch RPC [26] with RemoteModule and message passing. Sagemaker [13] offers a flexible parallel programming interface that allows for partitioning of arbitrary models and pipeline parallelism with minimal code changes. In Hanayo, we have designed and implemented a runtime system that is decoupled from the pipeline parallel scheduling algorithms. This allows the runtime system to support any of the current pipeline parallelism scheduling algorithms and provides an interface for users to customize the scheduling algorithm.

Techniques for Large Model Training. The field of large model training can be categorized into two main areas: 1) *Hybrid Parallelism:* Megatron-LM [23] combines tensor parallelism and pipeline parallelism for large model training, utilizing tensor

parallelism within nodes and pipeline parallelism between nodes. ColossalAI [2] proposed other tensor parallel methods, such as 2D parallelism [38] and sequence parallelism [19], which can be combined with pipeline parallelism. 2) *Memory Saving Techniques:* To reduce memory consumption during training, researchers have developed techniques such as activation checkpointing [4, 15], mix precision training [21], and the ZeRO optimizer [28] proposed by DeepSpeed [30]. ZeRO and other works also support tensor offload [8, 29, 31], which allows for the use of CPU memory or even NVMe storage. These techniques are independent of pipeline parallelism and can be combined to improve large model training.

7 CONCLUSION

Hanayo introduces a novel pipeline parallelism scheduling approach that decouples the relationship between the number of stages and devices, leading to higher throughput, as well as an efficient framework that enables communication and computation overlap for any pipeline scheme. Scaling and adaptivity experiments demonstrate Hanayo's capability to handle various models, diverse computational environments, and different scenarios, such as large-scale training and fine-tuning. We believe that our proposed method will benefit both academia and industry through its efficiency and high usability.

ACKNOWLEDGMENTS

LIU designed and wrote the system and the experiments, wrote the method and experiment-related parts in the paper, and participated in improving the method. **CHENG** designed and proposed key algorithms and structures, participated in paper writing, the improvement of codes implementation, and the experiments. **ZHOU** participated in the improvement of code implementation, experimental design and implementation, theoretical formula derivation, and article writing. **YOU** supervised this work and gave important insights.

This work used the Lonestar6 Cluster from TEXAS ADVANCED COMPUTING CENTER(TACC) and the cloud service of Tencent. We would like to thank them for their outstanding computing resource and professional service. ChatGPT was utilized to polish some of the texts in this paper. Yang You's research group is being sponsored by NUS startup grant (Presidential Young Professorship), Singapore MOE Tier-1 grant, ByteDance grant, ARCTIC grant, SMI grant and Alibaba grant.

REFERENCES

[1] Mandeep Baines, Shruti Bhosale, Vittorio Caggiano, Naman Goyal, Siddharth Goyal, Myle Ott, Benjamin Lefaudeux, Vitaliy Liptchinsky, Mike Rabbat, Sam Sheiffer, Anjali Sridhar, and Min Xu. 2021. FairScale: A general purpose modular PyTorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale.

[2] Zhengda Bian, Hongxin Liu, Boxiang Wang, Haichen Huang, Yongbin Li, Chuanrui Wang, Fan Cui, and Yang You. 2021. Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training. *arXiv preprint arXiv:2110.14883* (2021).

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174* (2016).

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[7] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2020. DAPPLE: A Pipelined Data Parallel Approach for Training Large Models. https://doi.org/10.48550/ARXIV.2007.01045

[8] Jiarui Fang, Zilin Zhu, Shenggui Li, Hui Su, Yang Yu, Jie Zhou, and Yang You. 2023. Parallel Training of Pre-Trained Models via Chunk-Based Dynamic Memory Management. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2023), 304–315. https://doi.org/10.1109/TPDS.2022.3219819

[9] Denis Foley and John Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17. https://doi.org/10.1109/MM.2017.37

[10] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. https://doi.org/10.48550/ARXIV.1806.03377

[11] W Daniel Hillis and Guy L Steele Jr. 1986. Data parallel algorithms. *Commun. ACM* 29, 12 (1986), 1170–1183.

[12] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2018. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. https://doi.org/10.48550/ARXIV.1811.06965

[13] Can Karakus, Rahul Huilgol, Fei Wu, Anirudh Subramanian, Cade Daniel, Derya Cavdar, Teng Xu, Haohan Chen, Arash Rahnama, and Luis Quintela. 2021. Amazon SageMaker Model Parallelism: A General and Flexible Framework for Large Model Training. *arXiv preprint arXiv:2111.05972* (2021).

[14] Chiheon Kim, Heungsub Lee, Myungryong Jeong, Woonhyuk Baek, Boogeon Yoon, Ildoo Kim, Sungbin Lim, and Sungwoong Kim. 2020. torchgpipe: On-the-fly pipeline parallelism for training giant models. *arXiv preprint arXiv:2004.09910* (2020).

[15] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2020. Dynamic tensor rematerialization. *arXiv preprint arXiv:2006.09616* (2020).

[16] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[17] Shigang Li, Tal Ben-Nun, Salvatore Di Girolamo, Dan Alistarh, and Torsten Hoefler. 2020. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 45–61.

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

[19] Shenggui Li, Fuzhao Xue, Yongbin Li, and Yang You. 2021. Sequence parallelism: Making 4d parallelism possible. *arXiv preprint arXiv:2105.13120* (2021).

[20] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 10012–10022.

[21] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. *arXiv preprint arXiv:1710.03740* (2017).

[22] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[23] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. https://doi.org/10.48550/ARXIV.2104.04473

[24] NVIDIA. 2017. NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. (2017).

[25] NVIDIA. 2020. NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl

[26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[27] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[28] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–16.

[29] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[30] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[31] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. {ZeRO-Offload}: Democratizing {Billion-Scale} Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 551–564.

[32] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning internal representations by error propagation*. Technical Report. California Univ San Diego La Jolla Inst for Cognitive Science.

[33] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[34] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990* (2022).

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[36] Boxiang Wang, Qifan Xu, Zhengda Bian, and Yang You. 2022. Tesseract: Parallelize the Tensor Parallelism Efficiently. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[37] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[38] Qifan Xu, Shenggui Li, Chaoyu Gong, and Yang You. 2021. An efficient 2d method for training super-large deep learning models. *arXiv preprint arXiv:2104.05343* (2021).

[39] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. 2021. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems* 3 (2021), 269–296.

[40] PengCheng Yang, Xiaoming Zhang, Wenpeng Zhang, Ming Yang, and Hong Wei. 2022. Group-based Interleaved Pipeline Parallelism for Large-scale DNN Training. In *International Conference on Learning Representations*. https://openreview.net/forum?id=cw-EmNq5zfD

[41] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. 2018. Imagenet training in minutes. In *Proceedings of the 47th International Conference on Parallel Processing*. 1–10.

[42] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016. Staleness-Aware Async-SGD for Distributed Deep Learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (New York, New York, USA) *(IJCAI'16)*. AAAI Press, 2350–2356.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT IDENTIFICATION

This paper introduces WPipe, a wave-like pipeline parallelism strategy that boasts a concise structure and practical applicability, alongside a high-performance pipeline execution runtime to tackle the challenges of pipeline strategy implementation. We have done our experiments on four computing environments, and the throughput of the training process depends on the GPUs and the interconnection between them. So if you want to get the same throughput numbers as those in the paper, please use a cluster with the same CPUs, GPUs, and GPU topology as described below. Also, please make sure you use the same version of CUDA, gcc, PyTorch and NCCL. But if you only want the check the relative relationship of the performance of the parallelism scheme, the platform will not be that important. Our evaluation is conducted on four platforms: 1)The Lonestar6 cluster from TACC(TEXAS ADVANCED COMPUTING CENTER), which has 32 GPU nodes. Each node has 3x NVIDIA A100 PCIE 40GB and we only use GPU No.1 and 2 as they are connected to the same socket. The CPUs on each node are 2x AMD EPYC 7763 64-Core Processor ("Milan"). A node has 256 GB RAM, 144GB /tmp partition on a 288GB SSD. 2)A Tencent CVM cloud server named GN10Xp with Intel Xeon Cascade Lake 8255C (2.5 GHz) CPU and 8 NVIDIA V100 GPUs with 32GB of memory. The GPUs are connected with NVLink, and the interconnect configuration is described in the V100 architecture whitepaper. 3)A local server with AMD EPYC 7543 32-Core Processor and 8 NVIDIA A100 80GB GPUs, with GPU 0 and 1, 2 and 3, 4 and 5, 6 and 7 connected with NVLink. 4)A local server with AMD EPYC 7742 64-Core Processor and 8 NVIDIA A100 80GB GPUs. All the GPUs are connected to each other with NVLink. The operating system that we use is Ubuntu 20.04.5 LTS, with cuda/11.3.1 and gcc-9.3.0. The version of pytorch is 1.13.1 and we use NCCL 2.14.3.

REPRODUCIBILITY OF EXPERIMENTS

All our experiments follow the same process. For each computing environment mentioned above, we first use the scheduler to load the configuration file of the model and the parallelism scheme, then the scheduler will generate the action list for each GPU and establish the NCCL backend for communication. Then the dataloader would start to load the training data and distribute it among the GPUs. Then we can start the training iteration. For a better understanding of the true performance of the parallelism, we only measure the time between the start of the forward propagation and the end of the flush operation after the backward propagation. As we use 50 iterations to warm up before we measure the training time cost for 50 iterations, the time we need for one experiment is usually 2 to 3 minutes, which means it is very easy for other users to reproduce the experiments. The scripts and codes of our experiments will be released on GitHub if the paper can be published. But you can still use existing pipeline parallelism frameworks to reproduce our method. As we mentioned before, the performance of the methods is related to the computing power of the GPUs and their interconnections. But in general, despite the platform and framework that you use, you can constantly get the result that our proposed method is faster than Chimera, and Chimera is faster than other methods like DAPPLE or GPipe. The throughput that you get on your own platform may be different, but the relative relationship between these methods will stay the same. In other words, you will not need to use the same platform as us to reproduce the results in the paper. The advantage of our proposed method can be proved on most mainstream computing environments.

ARTIFACT DEPENDENCIES REQUIREMENTS

1. Hardware: WPipe is designed to enable large model training on limited computing resources. So theoretically, you can use any GPU that supports CUDA. As long as the total memory of all your GPUs can hold the model weights and other intermediate tensors, you can use WPipe to speed up your training process. 2. Operating system: The experiments in our paper are carried out on Linux. We recommend that you use Linux when training with WPipe. 3. Software requirement: WPipe is written mainly in Python, and requires Python3 and CUDA. Python libraries include pytorch, numpy, transformers, and so on. The detailed Python library requirements will be given in our github repository. 4. WPipe does not require specific datasets. The datasets required are decided by the model and the task that you are working on. The performance results in the paper can be obtained using either random inputs or real data sets. If you use real datasets, you can use text datasets like WikiText-2, openwebtext, etc.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

Perform the following steps for installation and deployment: 1. Create a new Python environment using Miniconda3 or venv. And then clone the code repository for Wpipe. This process may take approximately 5 minutes. git clone https://github.com/MaruyamaAya/Wpipe 2. Install the required dependencies listed in the requirements.txt file. This installation may take about 5 to 10 minutes. pip install -r requirements.txt 4. The main entry point for the Python code can be found in ./wpipe/main.py. To enable multi-node multi-process startup, use torch.distributed.launch. On clusters, SLURM or PBS is commonly used for task submission and management. We provide sample cluster startup scripts for Meluxina (./wpipe/meluxina_script/) and TACC Lonestar6 (./wpipe/lonestar_script/). 5. To obtain experimental results with the corresponding parameters, submit the SLURM script using sbatch. sbatch ./wpipe/meluxina_script/submit_Chimera_8node_128layer_2model.slurm