Accelerating Distributed MoE Training and Inference with Lina

Jiamin Li
City University of Hong Kong

Yimin Jiang
ByteDance Inc.

Yibo Zhu
Unaffiliated

Cong Wang
City University of Hong Kong

Hong Xu
The Chinese University of Hong Kong

Abstract

Scaling model parameters improves model quality at the price of high computation overhead. Sparsely activated models, usually in the form of Mixture of Experts (MoE) architecture, have sub-linear scaling of computation cost with model size, thus providing opportunities to train and serve a larger model at lower cost than their dense counterparts. However, distributed MoE training and inference is inefficient, mainly due to the interleaved all-to-all communication during model computation.

This paper makes two main contributions. First, we systematically analyze all-to-all overhead in distributed MoE and present the main causes for it to be the bottleneck in training and inference, respectively. Second, we design and build Lina to address the all-to-all bottleneck head-on. Lina opportunistically prioritizes all-to-all over the concurrent allreduce whenever feasible using tensor partitioning, so all-to-all and training step time is improved. Lina further exploits the inherent pattern of expert selection to dynamically schedule resources during inference, so that the transfer size and bandwidth of all-to-all across devices are balanced amid the highly skewed expert popularity in practice. Experiments on an A100 GPU testbed show that Lina reduces the training step time by up to 1.73x and reduces the 95%ile inference time by an average of 1.63x over the state-of-the-art systems.

1 Introduction

Recent advances in deep learning have shown that a model's quality typically improves with more parameters [15, 21, 23, 29, 47]. Many new frontiers in Computer Vision (CV) and Natural Language Processing (NLP) have been explored using large dense models [22, 38, 44]. While effective in terms of model quality, the computation cost of model training and serving is extremely high. ChatGPT [1], an impressive chatbot released by OpenAI, is estimated to spend 3 million dollars per month to serve user requests. Wider adoption and development of these models are impeded by the exorbitant compute cost.

Following the basic idea of curbing the computation cost of massive models, *sparsely activated* models have recently been introduced [13, 23, 33, 44]. The *Mixture-of-Experts* (MoE) structure is now one of the most popular ways to implement sparse activation [13, 14, 44, 55]. For each input, instead of using all parameters, an MoE model selects just a few of them, i.e. *experts*, for processing. This leads to sub-linear scaling of FLOPs needed with model size. Recent literature [9, 22, 26, 30, 38, 54] has proven the potential of MoE models. For instance, Google develops a family of language models named GLaM using MoE [22]. Compared to GPT-3 with 175 billion parameters, the largest GLaM has 1.2 trillion parameters while only consuming 1/3 of the energy for training. Meanwhile, GLaM still achieves better zero-shot and one-shot performance than GPT-3. Microsoft reports that their MoE-based language models achieve a 5x training cost reduction compared to a dense model with the same model quality [38].

Given the uptake of MoE, there have been several systems for efficient MoE training and inference, including Google's Mesh TensorFlow [43], Meta's FairScale [10], Microsoft's DeepSpeed [2] and Tutel [7], etc. They provide APIs for users to replace the conventional dense layers with MoE layers with minimal code changes. They adopt both data parallelism and expert parallelism to accelerate the training and inference. That is, each device (e.g. GPU) is assigned with a unique expert, and uses all-to-all to receive inputs from other devices and then sends the gradients back to them accordingly. During training, allreduce is then used to aggregate non-expert gradients in the backward pass.

We focus on the efficiency of distributed MoE training and inference in this work. As some [25, 31, 41] has shown, the all-to-all operation is the main bottleneck. All-to-all blocks the subsequent computation operations and needs to be invoked two times in the forward pass and another two in the backward pass for each MoE layer. Interestingly, the main causes for all-to-all being the bottleneck are different in training and inference. In training, all-to-all and allreduce often contend for network bandwidth when they overlap in the backward pass,

leading to a prolonged blocking period to the computation. Inference, on the other hand, presents a highly-skewed expert popularity driven by real-world requests. Devices with popular experts have to handle much more data than others. Not only does it delay the launch of all-to-all, but it also causes imbalanced transfer size and bandwidth utilization across the devices, both of which are detrimental.

We are thus motivated to systematically tackle the all-to-all bottleneck. Our solution is Lina, a system that accelerates both MoE training and inference.

In training, we prioritize all-to-all over allreduce in order to improve its bandwidth. Existing MoE systems launch separate CUDA streams for the expert-parallel and data-parallel process groups which correspond to all-to-all for expert and allreduce for non-expert parameters, respectively. As there is no coordination between these streams, all-to-all and allreduce can overlap and fair-share the network bandwidth. Unlike allreduce, all-to-all is blocking and cannot be made parallel with the computation process. Thus, prioritizing all-to-all in the backward pass and avoid concurrent allreduce is crucial to reducing the blocking period.

To efficiently prioritize all-to-all, we adopt tensor partitioning which breaks down a tensor into smaller chunks, each of which forms a micro-op. With micro-ops, simple priority scheduling can be applied to guarantee full bandwidth for all-to-all while allowing allreduce micro-ops to make progress when all-to-all is not present. In addition, micro-ops allow the expert computation to be pipelined with all-to-all.

In inference, we dynamically schedule the resources for each expert in order to balance the workload of each device, thereby alleviating the imbalanced all-to-all transfer size and bandwidth. Intuitively popular experts should be given more resources while the rest may be served with less resources. The key challenge here is to efficiently and accurately obtain the expert popularity *before* the selection is actually done by the gating network, for every batch of input at each MoE layer, so scheduling benefit can be maximized with minimal overheads. Fortunately, we find the experts selected by each token across the layers demonstrate clear patterns, which allow us to estimate the expert distribution of the upcoming layer based on the past selection results from the preceding layers. We adopt a two-phase scheduling approach that fine-tunes the estimation based allocation only when the actual expert popularity deviates too far.

We build Lina based on DeepSpeed MoE [2] and PyTorch, and evaluate it on a cluster with up to 16 Ampere A100 GPUs with 40GB memory and 100Gbps InfiniBand. Results show that Lina accelerates all-to-all by at least 2.21x, and achieves on average 1.57x speedup in overall training step time compared to state-of-the-art system DeepSpeed. The median and 95%ile inference time is reduced by 1.45x and 1.63x.

Our contributions can be summarized as follows:

- We present an in-depth empirical analysis of distributed MoE to show the main causes for all-to-all to be the performance bottleneck in training and inference.
- We propose to prioritize all-to-all over allreduce in order to improve its bandwidth and reduce its blocking period in distributed training. Lina's scheduler incorporates tensor partitioning and pipelining to perform micro-op scheduling.
- We examine the pattern in expert selection of MoE layer and propose to estimate the expert popularity to conduct resource scheduling in advance during inference. Lina adopts a two-phase scheduling scheme to minimize the overhead.
- We implement a concrete prototype system and conduct comprehensive testbed experiments to demonstrate the benefits of our design in a realistic GPU cluster setting.

2 Background and Motivation

We start with an introduction on MoE and a widely-adopted distributed system for MoE model in §2.1. Then, we motivate our idea by analyzing the performance bottleneck (i.e. all-to-all) in distributed MoE training and inference in §2.2.

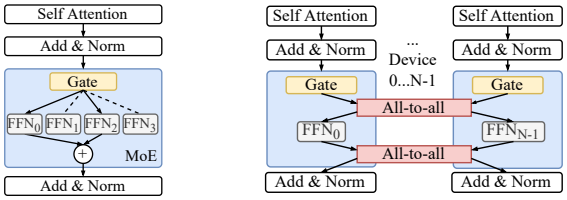
2.1 A Primer on MoE

Mixture-of-Experts (MoE) has been adapted to different types of DNN models , and exhibits great potential in improving the performance of language models in particular. GShard [31] and Switch Transformer [23] are two seminal works on scaling Transformer-based language models with MoE layers. We focus on MoE in Transformer-based models in this work.

Transformer-based models normally use an MoE layer to replace the feed-forward network (FFN) layer. An MoE layer consists of multiple FFNs each serving as an *expert*, and a gating network (Figure 1a). Every expert is a fully-connected two-layer network using ReLU activation but with different parameters. The gating network takes in the embedding vector of each token and multiplies them with its trainable matrix. Based on the results, it dispatches the token to a small number of experts (usually one or two). The final output of the MoE layer is the weighted sum of outputs from the selected expert(s). The sparsity nature of MoE improves the model scaling in size without increasing the training cost and naturally leads to a dynamically-changing model graph.

Load balancing loss. In MoE training, an auxiliary loss is introduced to evaluate the token distribution among the experts [23]. The objective is to achieve a uniform distribution of tokens across the experts, thereby preventing an excessive concentration of tokens in a single expert. By minimizing this loss term, we encourage but do not enforce the gating network to produce a perfectly balanced token distribution.

The standard practice is to calculate the auxiliary loss of each MoE layer and sum them with the training loss using an appropriate weight. Previous research has demonstrated the effectiveness of this approach [16,33,45]. However, it should be noted that achieving a perfectly balanced distribution, where the auxiliary loss converges to zero, is challenging [18, 53].



(a) There are four experts and the gate selects two experts.

(b) Distributed MoE. Data parallelism and expert parallelism are used.

Figure 1: MoE layer in Transformer-based models.

# Experts	Model	Training (ms)		Inference (ms)	
GPUs	#Layers & Params	All-to-all	Ratio	All-to-all	Ratio
4	12L + 117M	259	36.7%	73	27.4%
	24L + 233M	589	35.4%	103	26.2%
	36L + 349M	979	38.2%	153	28.3%
16	12L + 419M	333	39.5%	102	32.5%
	24L + 838M	715	37.6%	177	31.7%
	36L + 1.2B	1145	36.8%	243	27.4%

Table 1: The completion time of all-to-all and its ratio in training and inference task of Transformer-XL [20] in different number of experts per layer. Training and inference have the same batch size here. Each FFN layer is replaced with MoE and the number of experts is equal to the number of GPUs similar to the common practice [23]. A100 GPUs with 40GB memory and 100Gb/s InfiniBand are used. We use the MoE implementation in DeepSpeed.

During MoE inference, the trained gating network is utilized to dispatch tokens to the experts based on their respective embeddings. This process is solely driven by the characteristics of the token embeddings.

Hybrid parallelism in distributed MoE. Training and serving MoE models in a distributed manner are necessary due to the tremendous compute requirement of large-scale language models [12]. For efficiency, both data parallelism and MoE-specific *expert parallelism* (as a form of model parallelism) are applied [23,31]. Existing MoE systems [2,7,10,23,31,43] allocate one unique compute device (e.g., GPU) for each expert in expert parallelism. An all-to-all communication is then needed to send tokens to their experts selected by the gating network, and another all-to-all is needed to send tokens back to the device they belong to in data parallelism to finish the rest of the forward pass as shown in Figure 1b.

2.2 Bottleneck Analysis

Much prior work has identified that the introduction of all-to-all in MoE causes performance inefficiency in Transformer-based models [7, 41, 54]. We extract the completion time of all-to-all operations in both training and inference in our GPU cluster as shown in Table 1. All our experiments in this section use the same testbed and settings. Overall, all-to-all takes an average of 34.1% — a significant fraction of the step time. Interestingly, though the bottleneck brought by all-to-all is universal in both MoE training and inference, the causes differ. In the following, we motivate our work by analyzing how all-to-all affects the efficiency of training and inference, respectively.

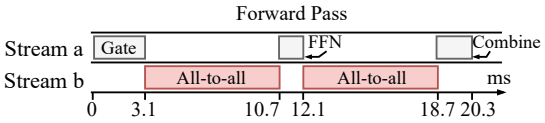


Figure 2: Timeline of forward pass an MoE layer. We simplify the presentation by bundling GPU kernels here: The computation kernels are grouped by their roles in the MoE layer into Gate, FFN and Combine. The Combine operation involves reshaping the tensors and computing the weighted output. The timeline is taken from a sample run of the 419M-parameter model in Table 1.

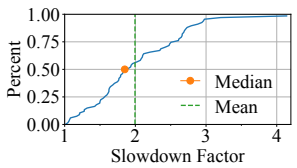


Figure 3: CDF of how much all-to-all is prolonged when it overlaps with allreduce operation. We mark the median and average slowdown factors.

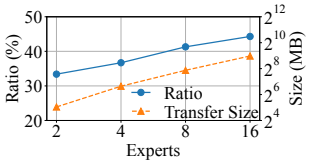


Figure 4: The proportion of all-to-all's completion time over training step time when the number of experts grows. Dashed line plots the data size in one all-to-all operation.

Synchronous all-to-all with large data transfer. The common characteristic shared by MoE training and inference is all-to-all's large data transfer. All-to-all is an irreplaceable synchronous component to handle the data exchange among devices in MoE layer. Each MoE layer has two all-to-all operations to send the tokens to the experts and then restore the position of tokens, as introduced in §2.1. The data transfers in the two all-to-all operations have the same size because the expert's FFN architecture ensures that its input data size is the same as the output data size. Figure 2 shows an empirical timeline view of the forward pass of MoE model in our cluster. All-to-all takes 74.9% of the end-to-end running time of one MoE layer. Expert FFN computation and the combine operation follow when all-to-all operation completes. MoE training and inference suffer from such inefficiency consistently. GPU is mostly idle during this period: We use the PyTorch Profiler [6] to profile the GPU activities for 20 steps in each experiment in Table 1, and find that the average GPU SM efficiency during all-to-all is 3.7%. Besides, the data transfer size grows linearly with the number of experts. Figure 4 presents the empirical evidence of all-to-all's transfer size as the number of experts grows from 2 to 16 (128). With the increasing number of experts, the time taken by all-to-all grows from 33.4% to 44.5% of the step time.

Problem in training: Prolonged all-to-all with allreduce. The unique challenge in MoE training is that applying the hybrid parallelism creates a particularly severe impact to all-to-all in backward pass. Non-MoE layers in *data parallelism* need *allreduce* to aggregate the gradients, while *expert parallelism* requires all-to-all to exchange tokens to compute expert gradients. Since the two operations control their own process groups independently, two dedicated CUDA streams are launched concurrently. This is demonstrated in Figure 5 with the timeline of backward pass in a sample run of MoE training. As the two operations overlap, they contend for the network bandwidth and their completion times are severely

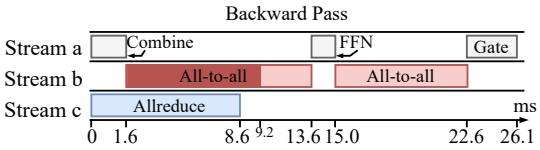
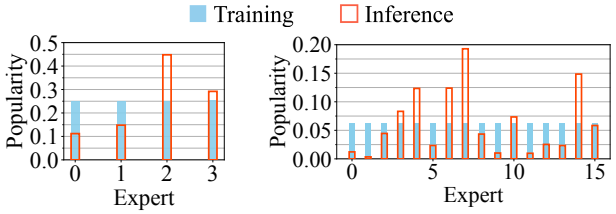


Figure 5: Timeline of backward propagating an MoE layer under hybrid parallelism. The first all-to-all is prolonged by the allreduce operation in Stream b. The shadowed part is its original completion time.



(a) 4-expert MoE. **(b)** 16-expert MoE.

Figure 6: Sampled expert popularity. The distribution is computed as the ratio between the number of tokens received by the expert and total number of tokens in one batch. We use the Enwik8 test set [3] for evaluation.

prolonged. To make matters worse, we find that the slowdown factor varies significantly. We collect the completion times of 1,500 all-to-all operations in backward pass on our testbed and plot the CDF of the slowdown factor they endure with allreduce in Figure 3. Observe that the median slowdown is over 1.83x and the worst is 4.14x.

Problem in inference: Skewed expert popularity. The main cause of all-to-all being the bottleneck in MoE inference is the skewed expert popularity. The token-to-expert distribution in inference is purely workload-driven, and we empirically find that the expert popularity is highly skewed in sharp contrast to training. We sample the expert popularity of the same MoE model in training and inference in Figure 6. In training, the distribution is nearly the same across all experts after hundreds of steps due to the use of load balancing loss. In inference, however, the most popular expert receives 4.02x and 5.56x tokens of the least popular ones in 4-expert and 16-expert inference tasks. With the same network and computation capacity, devices hosting popular experts take much longer to perform expert computation. In this experiment, the maximum idle time of the least popular expert is 29.4% of the inference time of that batch. Thus, within one batch, tokens to the less occupied experts have to wait for others to complete on the more popular experts, degrading the all-to-all performance significantly. Further, under uniform expert-device allocation, devices hosting popular experts have more tokens using their links for all-to-all, while the links of other devices are underutilized.

3 Design Overview

Lina is designed to accelerate all-to-all in distributed MoE. It attacks both the bandwidth contention with allreduce in training, as well as the straggler with unbalanced all-to-all bandwidth in inference. We focus specifically on MoE implementations that leverage both data and expert parallelism.

MoE training. We aim to improve the *bandwidth* of all-to-all in order to reduce the blocking period of the computation operations. Our key idea here is to *prioritize all-to-all* so it does not fair-share bandwidth with concurrent allreduce (§4). This is achieved using tensor-partitioning. We partition all-to-all and allreduce tensors into small chunks, each of which then forms a *micro-op*. Lina schedules an allreduce micro-op only when there is no all-to-all waiting or ongoing so that all-to-all is guaranteed the full network bandwidth during its lifetime. Without prior information, tensor-partitioning and micro-ops can ensure that in most cases all-to-all can launch immediately and allreduce is not deferred excessively.

MoE inference. We propose to dynamically adjust the device allocation for experts based on the *expert popularity*, so that is not all-to-all is not delayed by the trailing tokens, and its bandwidth utilization across links is balanced (§5). We exploit the expert selection pattern across adjacent layers to estimate the expert popularity. Based upon the estimation, Lina performs scheduling at each layer to allocates proportionally more devices for popular experts and pack unpopular ones to fewer devices, and coordinate all-to-all correspondingly.

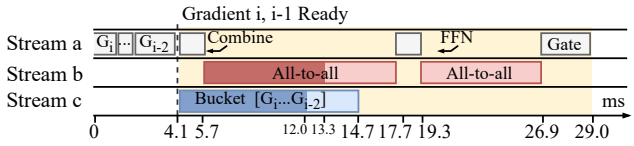
4 Prioritizing All-to-All Training

We have shown that all-to-all is slowed down significantly if it overlaps with allreduce in the backward pass in MoE training. Lina partitions the communication operations into small micro-ops and schedule them strategically in order to prioritize all-to-all without impeding allreduce and the computation process. We introduce the design challenges in §4.1. In §4.2, we present Lina's communication scheduler that uses tensor partitioning and pipelining to improve the training efficiency.

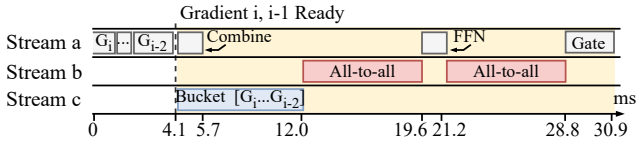
4.1 Design Challenge

Intuitively, Lina can prioritize all-to-all and avoid concurrent execution with allreduce with strict priority scheduling. All-to-all is always dispatched first if both are present in the queue, and subsequent operations have to wait until the running one finish to make sure allreduce does not share the bandwidth.

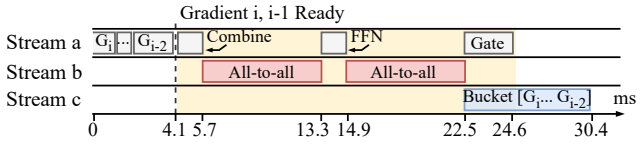
It turns out that simply prioritizing all-to-all is not as efficient as one may expect. For work-conservation, when an allreduce arrives first, it should be launched immediately. The problem is when an all-to-all arrives later, though ideally one would preempt the allreduce due to priority scheduling, this is not possible in current multi-GPU communication libraries such as NCCL [4]. The communication primitives are highly optimized and upon being called, their complete transmission strategies are settled and pushed to the CUDA streams. There is no control knob inside each primitive to adjust how it shares resources (e.g. CUDA cores, network bandwidth) with others. Thus, as the example in Figure 7b shows (based on testbed



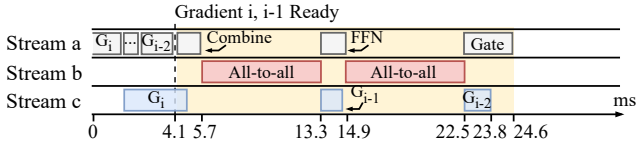
(a) Baseline. Shadowed all-to-all and allreduce are their completion times without concurrent operations. Computing the entire MoE layer's gradients ends at 29.0ms.



(b) Naively prioritizing all-to-all without concurrent transmission can lead to worse results; computing the MoE layer's gradients ends at 30.9ms. The completion time is profiled. Theoretically, the completion time should be the same as Figure 7a.



(c) Deferring allreduce to after the second all-to-all leads to better training efficiency; computing the MoE layer's gradients ends at 24.6ms.

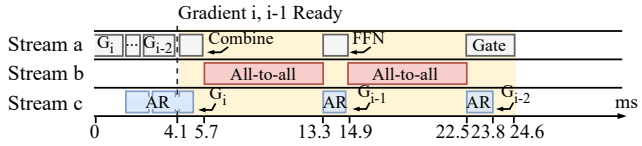


(d) Scheduling results if the arrival time and running time of communication operations are known a priori. The allreduce completes much faster than (c).

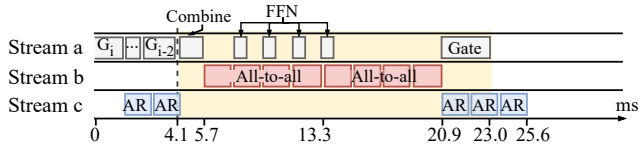
Figure 7: Backward pass of MoE training. The yellow background is the period of computing the gradients of the MoE layer. Stream a is responsible for the computation process and streams b and c are for communication. This timeline is extracted from a real run of the 419M-parameter benchmark model in Table 1.

experiments), naively prioritizing all-to-all actually leads to a longer completion time for the first all-to-all and training step time compared to the baseline in Figure 7a.

A potential solution is to obtain the arrival time and running time of the upcoming all-to-all and allreduce, and orchestrate them accordingly to maximize the efficiency. Assuming we know that the allreduce for gradient i can complete before all-to-all and the completion time of gradient $i-1$'s allreduce is shorter than FFN computation. Then we can schedule gradient $i-1$'s allreduce to the gap between the two all-to-all operations at 13.3ms as depicted in Figure 7d. Obtaining the precise knowledge of arrival and running times is, however, a daunting task. ML frameworks such as PyTorch fuse gradients into buckets based on a user-defined bucket size to optimize allreduce efficiency. Yet in large Transformer-based models, gradient sizes are also large; since bucketing is done on the gradient boundary, the actual bucket size for allreduce varies wildly [5]. Moreover, the implementation details of allreduce make it difficult to acquire a reliable running time estimate as prior work has found out [19].



(a) Prioritize all-to-all and partition allreduce tensors. Instead of bucketing gradients, we partition gradient i into three chunks when it is computed.



(b) Tensor partitioning for all-to-all and pipeline the FFN computation.

Figure 8: We show the scheduling results from Figure 7a with tensor partitioning. All-to-all and allreduce micro-ops are of the same size.

The other design choice is to blindly defer allreduce until an even number of all-to-all finish as there should be a larger gap between the backward pass of two MoE layers relative to FFN's backward computation. Figure 7c shows the best scheduling result based on the baseline in Figure 7a. In this case, allreduce can be launched when the second all-to-all finishes and completes before the first all-to-all of the next MoE layer (not shown in the figure). Yet, in other (worse) cases, allreduce may still block the all-to-all of the upcoming MoE layer if it takes relatively longer. In the extreme case, no allreduce can be launched until all four all-to-all operations of the current step finish. Since devices have to wait for allreduce before moving onto the optimization phase, this incurs more delay and is undesirable for wait-free backward pass [51].

4.2 Tensor Partitioning and Micro-Ops

To resolve the above challenges, we propose tensor partitioning that breaks down a communication operation into micro-ops, which can be easily prioritized with high efficiency.

Tensor partitioning. Unlike tensor bucketing which fuses multiple gradients for an allreduce, Lina partitions each gradient tensor into equal-sized small chunks and executes individual allreduce *micro-ops* independently. This brings two advantages. First, it resolves the varying bucket size problem for allreduce since each micro-op is uniform in size now. Second, micro-ops naturally make better use of bandwidth [36] without causing too much delay to allreduce under priority scheduling. Consider the same setup from Figure 7a, in Figure 8a we partition gradients into five chunks. Before the first all-to-all arrives, Lina launches three allreduce micro-ops; after the first all-to-all ends, it starts another micro-op to opportunistically make use of the expert computation time. Compared to the scheduling result without micro-ops in Figure 7c, allreduce for gradient $i-2$ now completes 6.6ms or 21.7% faster without prolonging all-to-all. Tensor partitioning does incur overhead due to the partition and concatenation operations before and after an allreduce, but it is mild: the

overall overhead in Figure 8a's case is 764us. §7.2.2 has more details of the overhead analysis.

Pipelining micro-ops. Intuitively, we can also partition all-to-all which provides an opportunity to pipeline the expert FFN and further reduce the time that computation is blocked. Specifically, we can pipeline the expert computation and all-to-all micro-ops (Figure 8b). Since the FFN computation is in token granularity, the expert can start computing with a subset of the tokens after one all-to-all micro-op. With pipelining, we can eliminate the FFN time which is 1.6ms in this example.

Expert packing. Ideally, the FFN and all-to-all micro-ops should take a similar time so that both compute capacity and network bandwidth are fully utilized without any bubbles in the pipeline. However, we notice that a single FFN micro-op takes much less time than its corresponding all-to-all micro-op (Figure 8b). In Lina, we consider packing multiple experts on each device whenever possible to maximize the pipelining efficiency. Lina adopts the following approach: starting with one expert per device, it iteratively increases the number of experts per device in powers of two, until the FFN computation exceeds that of the all-to-all micro-op. In case of GPU memory shortage, we adopt DRAM-offloading [42] to transfer expert parameters that are not currently in use to host memory.

5 Scheduling Resources in Inference

Recall in §2.2, we have shown empirically that skewed expert popularity leads to unbalanced processing times across tokens of the same batch in MoE inference, which delays all-to-all and causes imbalanced bandwidth for it severely. The root cause lies in the data granularity mismatch between the expert and the attention layers in the model: an expert processes individual tokens, but the attention layer processes an entire sequence as a whole. Our design question is thus: How can we ensure that each token within the same batch experiences the same end-to-end completion time no matter its expert selection result? We will first discuss the challenge of achieving this through dynamic resource scheduling (§5.1), and then present our design that exploits the unique token-level expert selection pattern to address the challenge in §5.2.

5.1 Design Challenge

To cope with skewed expert popularity, intuitively one must accordingly adjust the resource allocation for experts. This adjustment also needs to be done for each input sequence as the expert popularity distribution varies across sequences. An immediate question is: how can we know the expert popularity distribution, before the input is processed by the gating network?

This question is challenging for two reasons. First, even for a given batch of input, expert popularity varies across MoE layers of the model. We collect the expert popularity of different MoE layers for 1000 batches of input requests. Table 2 shows the top-4 popular experts of two 12-expert inference tasks: text generation and translation. Observe that each MoE layer of the same task (model) has completely different popular experts. This also suggests that dynamic resource scheduling has to be done before each MoE layer in order to be effective. Moreover, scheduling resources according to the actual expert selection results, as some might be thinking, incurs delay in collecting information, making scheduling decisions, and coordinating the all-to-all amongst all experts with respect to the new expert-device mapping, all of which are blocking operations and are performed at each layer. This is far from optimal (as will be shown in §7.3.1). Thus, we need to know as much as possible the expert popularity *before* the gating network selects experts in each layer, so these overheads can be largely overlapped with MoE computation.

5.2 Popularity based Scheduling

Lina tackles the design challenge by exploiting the token-level expert selection pattern which we empirically establish now. Building upon this, we design a resource scheduler that replicates popular experts on proportionally more devices in order to better balance the workload.

Pattern in expert selection. Experts in MoE models are trained to specialize in different types of input. We find that a token's expert selection demonstrates a pattern across the MoE layers. Tokens that have selected the same expert in layer i tend to select the same expert again in layer $i+1$. We trace the expert selection of sampled tokens. For each group of tokens that have selected the same expert in layer i, we calculate the ratio of them that in the next layer also select one of the same top-k experts ranked locally among the same group. Figure 9 plots this ratio averaged over token groups in two 12-layer MoE models. We see 41.94% tokens exhibit this pattern when k is 1 and 54.59% when k is 2, and deeper layers see more tokens with this pattern.

This observation makes intuition sense. The gating network has a simple architecture, and their routing or expert selection decision is made (largely) based on relatively simple features, such as the parts of speech of a word (noun, verb, etc.), and the meaning of the word (number, time, etc.) [32]. These features are fixed for each token. Meanwhile, experts focus on the local syntax information of each token rather than the cross-dependency within a sequence. For all these reasons, similar tokens naturally tend to be processed by the same or similar experts in each layer.

Estimating expert popularity. Though this pattern may not be sufficient to predict a particular token's expert selection accurately, it provides enough clues for us to estimate the overall expert popularity for a given batch. Specifically, Lina's estimation approach works as follows. In the profiling stage, we collect the expert selection results of all tokens when the load balancing loss is minimized and becomes stable. We then

Model& Dataset	Layer	Top-4			
Transformer-XL & Enwik8 (Text generation)	3	9	4	5	10
	4	5	7	8	10
	8	9	2	3	13
	12	4	5	15	8
BERT-Large & WMT En-De (Translation)	6	7	6	10	1
	8	10	6	2	15
	10	9	4	11	8
	12	1	8	10	14

Table 2: Top-4 popular experts in sampled MoE layer of two MoE models.

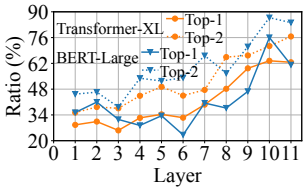


Figure 9: Ratio of tokens that select one of the top-k experts in layer $i+1$ given that they have selected the same expert in layer i.

group tokens that select the same experts from layer $i-l$ to layer i, which represent a unique sample path of experts used. For each sample path j, we compute the expert popularity distribution Ψ_j^{i+1} for layer $i+1$. Here l is the path length to control the accuracy-cost tradeoff in profiling: a larger path length leads to more accurate estimation for layer $i+1$ at the expense of higher data collection and computation costs.

Then based on the profiled distributions $\{\Psi\}$, Lina can estimate the next layer's expert selection distribution for each sample path of experts traversed by a token in inference (starting from the l-th layer of the model). In each layer i, for a sample path j, we pick the top-k expert(s) of the subsequent layer from Ψ_j^{i+1} and use their probabilities $\{P_j^{i+1}(e)\}$ to represent expert popularity for resource scheduling, where e denotes an expert. The reason why we only consider top-k experts is that they demand the most resources, and the remaining experts have low popularity (Figure 9). Note that this estimation happens before any MoE layer computation takes place.

Two-phase scheduling. During inference, Lina dynamically conducts layer-wise resource scheduling in two phases.

The first phase happens right after the expert popularity estimation at each MoE layer, when Lina relies on the estimation to replicate popular experts on more devices and pack unpopular ones onto fewer devices. Specifically, the total number of devices for expert e is determined by:

$$n_e = N \times \sum_{t=1}^{N_t} P_{j(t)}^{i+1}(e)/N_t, \tag{1}$$

That is, for the current batch of input with N_t tokens, using estimation from each token t's sample path $j(t)$ up to layer i, the overall popularity of expert e is estimated as $\sum_{t=1}^{N_t} P_{j(t)}^{i+1}(e)/N_t$ for layer $i+1$ accounting for all tokens. This requires the same proportion of devices assuming the expert parallelism degree is 1 (i.e. the number of devices equals the number of experts). For experts with the estimation n_e, we adopt the first-fit-decreasing heuristic to pack them into the empty devices so the total devices used are minimized. It is possible that some experts, being extremely unpopular (for this batch), are not amongst the top-k list of any tokens and thus do not have their n_e estimation. They are assigned evenly to the remaining free devices if any; otherwise are randomly assigned to a device.

In phase two, Lina fine-tunes the estimation-based scheduling decision after the gating network selects the actual experts. It checks if the selection result deviates significantly from the estimation, by comparing the overall top-$2k$ experts. If the two lists are identical, no fine-tuning is needed and inference continues. Otherwise, the scheduler re-computes the resource allocation with the actual expert popularity now available following the same logic in phase 1. The fine-tuning phase does incur delay to collect the gating outputs and check against the estimation, which is necessary to deal with inaccurate estimation that turns out to be much more detrimental to performance, if left unchecked (§7.3).

6 Implementation

We implement Lina on DeepSpeed MoE and PyTorch using C++ and Python. PyTorch 1.10, CUDA 11, and NCCL 2.10 are used. We modify PyTorch's implementation of distributed training to support Lina in DeepSpeed. The implementation has ∼7500 LoC.

6.1 Training

Lina's communication scheduler for training is deployed on all devices and runs a single thread. Since the communication scheduling is purely local in scope, no coordination is needed across the scheduler instances on different devices.

Communication scheduler. Each scheduler instance maintains a priority queue to schedule the micro-ops. The micro-op size is passed in as a hyperparameter. Lina uses the built-in APIs `chunk` and `cat` in LibTorch to partition the data in the token dimension. We avoid putting chunks from different gradients into the same micro-op to simplify the subsequent concatenation operation. Moreover, the scheduler stops launching allreduce micro-ops if the combining computation in backward pass, since this implies all-to-all is imminent. We pipeline all-to-all micro-op in the MoE layer. FFN is ready to start right after each all-to-all micro-op.

Expert packing coordinator. We embed a packing controller in the MoE model and it runs a single thread. Expert packing is dynamically adjusted after 10 training steps. In the forward pass, the controller records the completion times of all-to-all and FFN micro-ops. When FFN micro-ops are shorter than all-to-all, the controller starts to pack experts. First, we initialize the new process groups. Second, the controller inserts a one-time synchronous all-to-all to exchange expert parameters between packed devices that would be invoked at the upcoming iteration. Finally, multi-stream parallel execution is adopted for both forward and backward passes when more than one expert are hosted on a device.

6.2 Inference

Resource scheduler. The inference scheduler runs on a dedicated thread on device 0 of the cluster and manages resource scheduling. Each device saves the weights of all experts in

their host DRAM and the collected layer-wise expert popularity distribution using multiple `unordered_map`, one for each layer. If GPU memory is in shortage, a device only loads one expert and the profiled distribution of one layer at a time.

In phase one of scheduling, all relevant communication happens by piggybacking the information on the regular all-to-all to reduce overheads. For each MoE layer, each device appends the popularity estimation to the first all-to-all for device 0. The scheduler computes the new expert-device mapping and instructs each device which expert and how many to host via the second all-to-all. We also include necessary information to coordinate all-to-all of the next layer, including the list of devices with the same expert, and how many tokens each replica should handle to balance the load. Devices then swap in the expert weights for the next layer. All these procedures are pipelined with model computation.

In phase two, each device updates the actual expert popularity in a separate NCCL `send` to the scheduler. If no fine-tuning is required, the scheduler broadcasts a resume signal. This only creates a negligible overhead as the transfer size is tiny. Otherwise, Lina broadcasts the fine-tuned expert-device mapping. The model computation is blocked during phase two until the scheduler's command is received.

All-to-all coordination. In inference, Lina uses all-to-all with an unequal split. That is, the transfer size to each device in all-to-all does not need to be the same. Using unequal split all-to-all can save the overhead of initializing multiple process groups. A placeholder data pointer is passed to all-to-all if no tokens are directed to a certain device.

Expert packing. Expert computation is sequential on devices hosting multiple experts. Each device loads the experts one at a time to perform computation and move on to the next packed expert. In this manner, Lina avoids placing extra strain on the GPU memory. The second all-to-all is launched when the computation for all packed experts is completed. We set a maximum number of experts per device to control the overhead of swapping the weights.

7 Evaluation

We present the testbed evaluation results here.

7.1 Setup

Testbed setup. Our testbed has four worker nodes. Each node has 4 Ampere A100 GPUs with 40GB memory and is equipped with 100Gbps InfiniBand.

MoE models. We convert three common Transformer-based dense language models to MoE ones for training.

- Transformer-XL [20]: a 24-layer encoder model.
- BERT2GPT2 [49]: a 12-layer encoder-decoder model.
- GPT-2 [39]: a 12-layer decoder model.

Besides, we consider two inference tasks.

- Transformer-XL [20]: The inference task is text generation with Enwik8 [3] test set.
- BERT-Large [21]: a 12-layer decoder model. The inference task is translation using WMT En-De [8] test set.

All FFN layers in these models are converted to MoE layers. We vary the number of experts in an MoE layer from 2, 4, 8, to 16. We adopt top-2 gating in training and top-1 gating in inference following [23], i.e. $k = 2$ in training and $k = 1$ in inference

Metrics. We consider four metrics to evaluate Lina.

- Training step time: Time to complete one step of training.
- Inference time: Time to complete one batch of inference.
- All-to-all time: The completion time of all-to-all.
- MoE layer time: Time to complete one MoE layer of computation and communication.

In collecting these metrics we use PyTorch Profiler to obtain CUDA kernel execution time and GPU activities. Training results are averaged over 50 steps after a 10-step warm-up period. Inference results are averaged over the test set. Since the optimization introduced by Lina does not affect the precision of model parameters, model accuracy is unaffected and we omit its evaluation.

Training configurations. Lina's micro-op communication scheduler adopts a tensor partition size of 30MB, which can minimize the period blocked by all-to-all in most cases. Expert packing is launched at the 10-th step of each training task and is adjusted every four steps.

Inference configurations. Lina's resource scheduler runs on device 0. The path length l in popularity estimation is 3; the maximum number of experts packed on a device is 4.

Baselines. We use the vanilla DeepSpeed [2] as the Baseline. We also provide a comparison to the open-source version of Tutel [7], which performs similarly with DeepSpeed. We enable hierarchical all-to-all for both Lina and DeepSpeed and disable Random Token Dropping [50] introduced by DeepSpeed.

7.2 Training

We start with Lina's training performance. Note that Lina is evaluated when the expert packing decision is stabilized; all settings here use 2 experts per device as the best strategy except Transformer-XL with 16 experts, which uses 4 experts per device. The number of GPUs is equal to the number of experts per layer in both Baseline and Lina.

7.2.1 Overall Performance

Training step time. Figure 10 shows Lina's speedup in step time over Baseline and Tutel. All other aspects of the models stay the same (e.g. sequence length, hidden states dimension, etc.). Compared to Baseline (DeepSpeed), step time is reduced by an average of 1.37x and 1.47x for the 4- and 16-expert cases, respectively, and by an average of 1.71x and

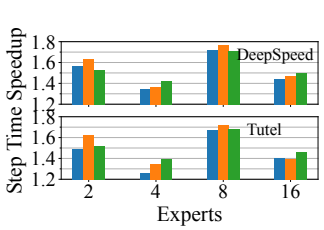


Figure 10: Speedup of training step time against two Baselines.

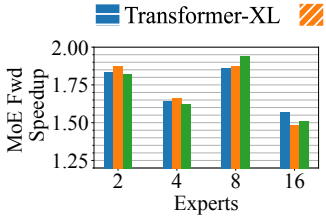


Figure 11: Speedup of MoE layer's forward pass completion time.

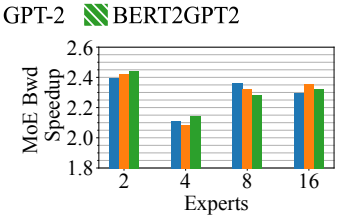


Figure 12: Speedup of MoE layer's backward pass completion time.

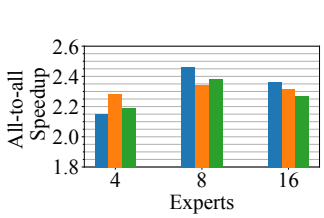
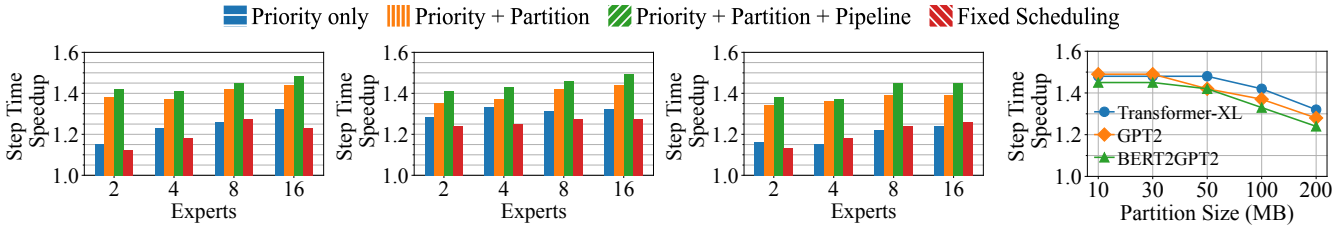


Figure 13: Speedup of all-to-all time in forward and backward pass.



(a) Transformer-XL. **(b)** GPT-2. **(c)** BERT2GPT2.

Figure 14: Training step time speedup over Baseline with different design choices of the communication scheduler.

Figure 15: Partition size increases from 10MB to 200MB in 16-expert models.

1.73x for 2- and 8-expert models, respectively. The 2- and 8-expert cases see more significant gains as Lina's packs two experts per device as mentioned before. The 2-expert case thus boils down to pure data parallelism without any all-to-all; the 8-expert models avoid inter-node all-to-all as our servers have 4 GPUs each. Lina's speedup over Tutel is slightly smaller than that of DeepSpeed. Thus in the following we only use DeepSpeed as the Baseline.

MoE layer time. We specifically seek to understand Lina's gain in MoE layers in both the forward and backward pass. As Figures 11 and 12 show, similar to step time, the gain in the 2- and 8-expert cases is the largest. The forward and backward pass of MoE layers in the 2-expert case are accelerated by 1.84x and 2.41x, and in the 8-expert case by 1.89x and 2.32x, respectively. Since backward pass in Baseline suffers from the interference of allreduce while the forward pass does not, the improvement in the backward pass is more significant. Average GPU utilization in the MoE layer for 16-expert cases is improved by at least 16% as the period blocked by all-to-all is minimized with Lina.

GPU utilization and memory usage. We measure the average GPU utilization GPU memory usage. We observe an average of 17.6% improvement in GPU utilization due to the efficient scheduling of Lina. Expert packing would lead to usage increase in GPU memory. The peak memory of BERT2GPT2 is increased by 19.5% while Transformer-XL and GPT-2 use up all the memory and apply DRAM-offloading to store the packed expert parameters.

All-to-all time. We then zoom in on all-to-all time in backward pass, where Lina prioritizes all-to-all and avoids concurrent execution with allreduce. Expert packing also reduces the all-to-all transfer size. Figure 13 shows an average speedup of 2.21x, 2.39x, and 2.31x in 4-, 8-, and 16-expert cases in all-to-all time, respectively.

Expert	Model	Pipelining Efficiency		
		w/o Packing	w/ Packing (Experts per Device)	
16	Transformer-XL	33%	86%	4
	GPT-2	36%	85%	2
	BERTGPT2	34%	79%	2

Table 3: Pipelining efficiency comparison with and without expert packing.

Expert	Model	Average GPU Utilization(%)		GPU Memory Peak Usage(%)		
		Baseline	Lina	Baseline	Lina	DRAM-offloading
16	Transformer-XL	66.2	83.4	72.1	100	✓
	GPT2	62.3	78.2	83.8	100	✓
	BERT2GPT2	63.5	82.5	74.3	94.2	✗

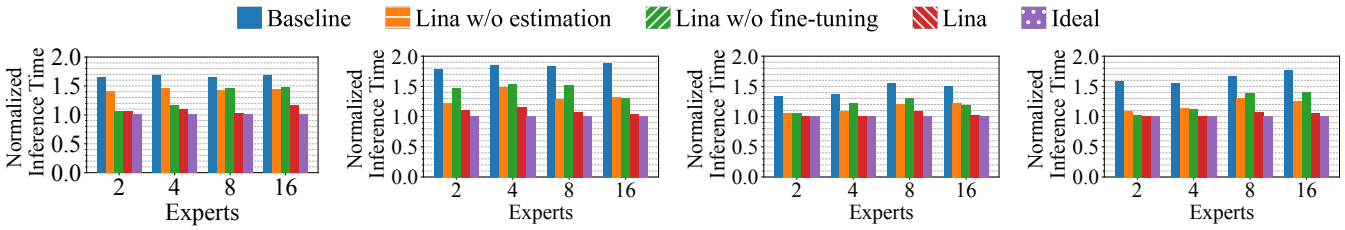
Table 4: GPU utilization and peak memory usage of 16-expert MoE models. GPU Memory Peak Usage is the ratio between the maximum usage and the total device memory. DRAM-offloading indicates if it is applied.

We also examine the pipelining efficiency between all-to-all and expert computation in Lina. We define the pipelining efficiency to be the fraction of non-idle time in the computation CUDA stream during the all-to-all duration. We calculate the pipelining efficiency of Lina before and after adopting expert packing in Table 3. The average improvement is 2.43x in 16-expert case, which also demonstrates the benefits of expert packing. The expert FFN micro-op time is thus closer to the all-to-all time. We find that two experts per device can achieve the best pipelining efficiency in most cases, justifying our settings mentioned before.

7.2.2 Communication Scheduler

We now present an in-depth analysis of Lina's priority-based micro-op scheduler, aiming to understand the benefit of each design choice. For fairness all experiments here are obtained without expert packing in Lina, i.e. one expert per device.

Tensor partitioning and pipelining. To justify our design, we incrementally add the key design choices to Baseline and see their corresponding gain: first priority scheduling, then tensor partitioning, and lastly pipelining. Besides, we



(a) Median inference time with Transformer-XL. **(b)** 95%ile inference time with Transformer-XL. **(c)** Median inference time with BERT-Large. **(d)** 95%ile inference time with BERT-Large.

Figure 16: Median and tail inference time. We normalize the inference time with the ideal result. The median and tail inference time is the same in Ideal.

consider a fixed scheduling strategy where allreduce is always scheduled between pairs of all-to-all operations (i.e. two MoE layers) with tensor fusion enabled in PyTorch's `DistributedDataParallel` by default (same as Baseline).

Figure 14 shows the step time comparison. We make several interesting observations here. First, using priority brings about 10%–30% gain over Baseline in most cases, with an average of 24%. Priority scheduling in general presents more benefit when more devices and nodes are used in training. The main reason is that all-to-all's slowdown due to sharing bandwidth with allreduce is more severe as training scales out. Second, tensor partitioning significantly improves the benefit of prioritizing all-to-all: step time is reduced over Baseline by 1.36x, 1.36x, 1.41x and 1.42x in 2-, 4-, 8-, and 16-expert cases, respectively on average. On the other hand, pipelining's gain is limited as expected, since expert computation takes much less time than all-to-all without expert packing (recall §4.2). Overall, all three design choices can effectively reduce all-to-all's completion time.

We also observe that the relative benefit of priority scheduling and tensor partitioning is model-specific: GPT-2 enjoys much more gain from priority compared to tensor partitioning while the other two models do not exhibit such clear pattern. This is likely due to the degree of overlapping of all-to-all and allreduce: most allreduce can fit in between all-to-all operations in GPT-2, and as a result using priority scheduling alone is very beneficial.

Finally, the fixed scheduling strategy leads to the smallest gains in almost all cases. This is because (1) all-to-all still has to fair-share bandwidth with allreduce, and (2) tensors are not partitioned which aggravates the impact of allreduce. This demonstrates again the effectiveness of our design in prioritizing all-to-all with smaller tensors instead of using fixed heuristics that cannot opportunistically maximize efficiency.

Partition size. We also evaluate the impact of partition size on the communication scheduler. Figure 15 shows the step time of 16-expert models when we gradually increase the partition size from 10MB to 100MB. We find that a partition size beyond 50MB slows down Transformer-XL and BERT2GPT2 compared with 30 MB. As long as the period blocked by all-to-all is minimized, step time would be minimum. Therefore, for each model and setting, there are multiple optimal partition sizes. Ideally, the scheduler can more precisely control the operations with a smaller partition size. In practice, small partitions (below 10MB) may cause heavy transmission overhead in each micro-op and degrade the overall performance [37].

Overhead analysis. We provide a brief analysis of the overhead incurred by Lina's communication scheduler. First, the preprocessing and postprocessing, including tensor partitioning and concatenation, take an average 1.02% of the step time. Second, we measure the transmission overhead of micro-ops. We sum up running times of all the communication micro-ops and compare against those without partitioning in Baseline. The average completion time is lengthened by 1.7%.

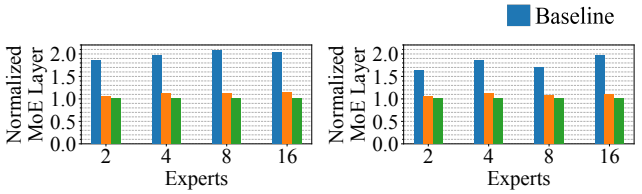
7.3 Inference

We then evaluate Lina's inference performance. Each experiment is repeated five times: two of which measure the end-to-end inference time, and the rest profile the different components with Profiler and collect statistics for overhead and estimation accuracy. This way the inference time is not affected by the profiling overhead.

7.3.1 Resource Scheduler

Inference time. Figure 16 shows the median and 95% inference time of Baseline and Lina. We also present the ideal inference time with a perfectly balanced load across devices in all MoE layers. This is obviously challenging to achieve with real-world requests. Thus we modify the gating network to constantly output a balanced expert selection to obtain this benchmark. We normalize all results to the Ideal value.

Lina's resource scheduler effectively balances the load among devices and achieve inference time close to Ideal. Compared to Baseline, median inference time is reduced by 1.54x and 1.45x for the 4- and 16-expert Transformer-XL, and by 1.36x and 1.46x for the 4- and 16-expert BERT-Large, respectively. The 95%ile inference time is reduced by 1.82x for 16-expert Transformer-XL and 1.68x for 16-expert BERT-Large. The reduction on tail inference time increases with more experts in a layer, because a wider MoE layer is more likely to present more skewed expert popularity, giving more room for Lina to optimize. Lastly, Lina's gap to Ideal can be explained for two reasons other than the overheads. First, Lina cannot perfectly balance load: the least popular experts are



(a) Transformer-XL. **(b)** BERT-Large.

Figure 17: 95%ile completion time of MoE layer.

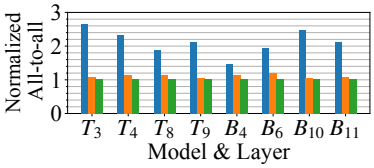


Figure 18: All-to-all time in 16-expert MoE. T is Transformer-XL and B is BERT-Large.

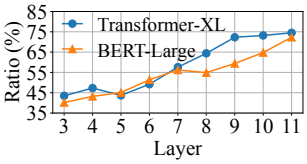


Figure 19: Estimation accuracy of 16-expert MoE.

randomly placed for example. Second, Lina starts to schedule from the fourth layer.

MoE layer and all-to-all time. With Lina, MoE layer time includes gate computation, phase two of scheduling, two all-to-all, and expert computation; phase one of the scheduling is largely overlapped with computation as explained in §6.2. The 95%ile MoE layer time is reduced by 1.87x and 1.77x in 8- and 16-expert Transformer-XL over Baseline and by 1.58x and 1.81x in 8- and 16-expert BERT-Large as in Figure 17. We also extract all-to-all time, which is a direct indicator of whether Lina balances load across devices effectively. We present the tail all-to-all time reduction of different layers in Figure 18. The average and maximum improvements are 1.96x and 2.50x over Baseline. These results confirm that Lina effectively balances the load of each device and all-to-all transfer size of each link.

Two-phase scheduling. We then evaluate the effectiveness of our resource scheduler's design. We consider separately Lina without estimation and without fine-tuning in order to understand their individual gains. Lina w/o estimation refers to scheduling using the actual routing decision computed by the gating network.

In Figure 16, we present the comparison of inference time for all schemes. Without estimation the median inference time is worsened by 24.0% and 18.6% for 16-expert Transformer-XL and BERT-Large in Lina. The scheduler works after the gating network and blocks all-to-all with the following computation until it completes. Thus the scheduling overhead manifests at each MoE layer, overweighing the additional gains brought by accurate popularity information. The tail inference time is less affected compared to the median, but still suffers without estimation.

Without fine-tuning, tail inference time is prolonged by 26.7% and 33.1% for 16-expert Transformer-XL and BERT-Large. This suggests that fine-tuning also plays an indispensable role when the estimation shows a large difference from the actual routing decision. For example, if the top-1 expert in the actual routing decision is estimated as an unpopular one packed with others, the Moe layer time would even be worse than Baseline. More discussions are presented in §7.3.2. The importance of fine-tuning depends heavily on estimation accuracy and number of expert in MoE layer.

Overhead analysis. We dissect the overhead of the resource scheduler, by considering the scheduling times of phase one and phase two separately. The scheduling time for both phases averages at ∼6.2ms since they share the same logic and coordination workflow. Yet, the overhead of phase one is largely overlapped with model computation. Though overhead of phase two with re-scheduling is more salient, it only kicks in for 23% of the cases on average, and is smaller than the idle time incurred by skewed expert popularity. The overhead of phase two without fine-tuning is merely 1.45ms.

Model	Path Length	Norm. Inference Time Median	Norm. Inference Time 95%ile	Fine-tuning (%)	Estimation Accuracy (%)
Transformer-XL	1	1.41	1.32	76.5	31.6
	3	1.16	1.04	25.7	60.4
	6	1.19	1.11	22.5	71.4
BERT-Large	1	1.34	1.35	71.3	28.3
	3	1.07	1.04	32.2	63.5
	6	1.09	1.11	27.1	66.0

Table 5: Lina's performance using different path lengths during estimation. Both models have 16 experts per layer. Inference time is normalized to Ideal.

7.3.2 Popularity Estimation

We now analyze Lina's popularity estimation method.

Estimation accuracy. We first examine the estimation accuracy across MoE layers. We resort to the same definition used in Lina's phase two scheduling: if the top-2 (recall $k = 1$ in inference) estimated experts are identical to the actual routing decision, we consider the estimation accurate. Figure 19 shows the accuracy for every MoE layer in two inference tasks. Overall, estimation accuracy is 58.41% and 54.16% for Transformer-XL and BERT-Large, respectively. The estimation accuracy is higher in the latter layers of the model, which is consistent with our observation in Figure 9. We also compare the complete popularity rankings given by the estimation to better understand its accuracy. Using 1000 random batches of the Transformer-XL model, we observe that errors usually happen at experts with a similar popularity. An average of 3.67 experts out of the estimation are incorrectly ordered. Therefore, the fine-tuning only requires little adjustment to the experts packed together. The effectiveness of Lina's estimation can be justified.

Sample path length. We also investigate the impact of sample path length l. The longer the sample path of expert selection is for making an estimation, the more accurate the result is. Table 5 shows Lina's performance degrades with $l = 1$, in terms of inference time, estimation accuracy, and the occurrence of phase two fine-tuning, compared with the default length of 3. Longer paths can elevate the estimation accuracy and further

Task	Dataset	Model	Norm. 95%ile Inference Time	Estimation Accuracy
Sentiment Analysis	IMDB Reviews [34]	BERT	1.08	64.4%
	Twitter [35]		1.11	62.3%
Translation (English)	WMT French [8]	T5 [40]	1.04	68.8%
	WMT Russian [8]		1.08	62.5%

Table 6: Lina's performance on different tasks and datasets. Inference time is normalized to Ideal. The path length is set to 3.

reduce the number of times of Lina's fine-tuning. However, due to the problem of a slower start, the reduction of inference time is not as noteworthy as the estimation accuracy. For a path length of 6, Lina shows a similar median and tail result as the performance with a path length of 3.

Generalizability. We proceed to evaluate how well Lina's popularity estimation approach can be generalized to different tasks. Table 6 shows the estimation accuracy of four tasks with different datasets. The 95%ile inference time can achieve at most 1.04x of the Ideal inference time and the estimation accuracy is at least 62.3%. Lina's estimation method relies on the patterns obtained from training stage. Therefore, it is tailored to each specific task and proves to be an effective approach to capturing the expert popularity prior.

8 Discussion

Parallelism in training. With the increasing scale of language models, the adoption of pipeline and tensor parallelisms has become essential [46]. Pipeline parallelism involves the use of blocking send and receive operations to transmit intermediate activations, while tensor parallelism utilizes blocking all-reduce operations to combine tensor partitions. Extensive research has been conducted on coordinating communication operations for dense models [7, 27, 52]. Lina focuses on sparsely-activated MoE with data and expert parallelism, which are orthogonal to existing work.

Estimation of expert popularity. The current estimation approach used by Lina relies on data collection during the training stage. While fine-tuning can assist in improving efficient expert placement decisions, an estimation method with improved accuracy and confidence would further reduce inference time. One potential approach is to leverage machine learning techniques to train a compact yet powerful model that can predict the expert selected by each token in every MoE layer ahead of time, when the requests are received.

9 Related Work

Existing MoE systems. Recent literature has proposed MoE-specific optimization techniques. DeepSpeed [41] enables distributed training for MoE models and leverages flexible combinations of parallelism strategies. It also introduces a novel MoE architecture called Pyramid-Residual MoE. PR-MoE applies experts only where they are most effective. Tutel [7] extends DeepSpeed and proposes an adaptive parallelism switching strategy specialized at MoE training tasks. It also includes a hierarchical all-to-all design to cope with the inter- and intra-node GPU topology for better efficiency. It is complementary with Lina.

FasterMoE [25] proposes a roofline performance model to analyze the end-to-end performance of MoE training systems. Guided by this model, they propose a dynamic shadowing approach that pulls popular expert parameters instead of sending tokens to the experts. They also design a topology-aware expert selection strategy that relieves network congestion by sending tokens to experts with lower latency.

Communication acceleration in distributed training. Our community has proposed several communication schedulers for generic distributed training [11, 17, 19, 24, 37, 48]. The objective is to better overlap the communication and computation operations in the backward pass and prioritize the communication of former layers over latter layers in the model. In Lina, we leverage the domain-specific insight that all-to-all should be prioritized over allreduce in MoE training, which is different from prior work. BytePS [28] proposes to reduce the communication traffic by utilizing the heterogeneous GPU/CPU resources in a training cluster. These acceleration techniques can be integrated into distributed MoE. Lina can also benefit from this idea, since more available bandwidth can be left to all-to-all operations.

10 Conclusion

We presented Lina, a new system that accelerates all-to-all in distributed MoE. Through a systematic analysis, we build Lina upon two key ideas: first to prioritize all-to-all over allreduce using tensor partitioning and pipelining to improve its bandwidth in training, and second to dynamically balance the workload with token-level expert selection pattern in inference. We implemented Lina over DeepSpeed and performed extensive testbed evaluation using A100 GPUs and 100Gbps InfiniBand to show that Lina significantly improves training efficiency and inference time.

Acknowledgment

We thank the anonymous ATC'23 reviewers and our shepherd Myoungsoo Jung for their constructive and valuable comments. We also thank the anonymous reviewers from NSDI'22 for their feedback that helped improve the paper. This work was supported in part by funding from the Research Grants Council of Hong Kong (N_CityU139/21, C2004-21GF, C7004-22G, R1012-21, R6021-20F, and 11209520), and from CUHK (4055138).

References

[1] ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt/.

[2] DeepSpeed. https://github.com/microsoft/DeepSpeed.

[3] Enwik8. http://prize.hutter1.net/.

[4] NCCL. https://github.com/NVIDIA/nccl.

[5] PyTorch Distributed Data Parallel. https://pytorch.org/docs/stable/notes/ddp.html.

[6] PyTorch Profiler. https://pytorch.org/blog/pytorch-profiler-1.9-released/.

[7] Tutel. https://github.com/microsoft/tutel.

[8] WMT 19. https://github.com/facebookresearch/fairseq/blob/main/examples/wmt19/README.md.

[9] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. Efficient large scale language modeling with mixtures of experts. *arXiv preprint arXiv:2112.10684*, 2021.

[10] 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. Fairscale: A general purpose modular pytorch library for high performance and large scale training. https://github.com/facebookresearch/fairscale.

[11] Yixin Bao, Yanghua Peng, Yangrui Chen, and Chuan Wu. Preemptive all-reduce scheduling for expediting distributed dnn training. In *IEEE INFOCOM*, 2020.

[12] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml. In *Proc. MLSys*, 2022.

[13] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*, 2015.

[14] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

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

[16] Tianlong Chen, Zhenyu Zhang, AJAY KUMAR JAISWAL, Shiwei Liu, and Zhangyang Wang. Sparse moe as the new dropout: Scaling dense and self-slimmable transformers. In *Proc. ICLR*, 2023.

[17] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. Elastic parameter server load distribution in deep learning clusters. In *Proc. ACM SoCC*, 2020.

[18] Zewen Chi, Li Dong, Shaohan Huang, Damai Dai, Shuming Ma, Barun Patra, Saksham Singhal, Payal Bajaj, Xia Song, Xian-Ling Mao, et al. On the representation collapse of sparse mixture of experts. *Proc. NeurIPS*, 2022.

[19] Minsik Cho, Ulrich Finkler, David Kung, and Hillery Hunter. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. In *Proc. MLSys*, 2019.

[20] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

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

[22] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten P Bosma, Zongwei Zhou, Tao Wang, Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. GLaM: Efficient scaling of language models with mixture-of-experts. In *Proceedings of Machine Learning Research*, 2022.

[23] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[24] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. Tictac: Accelerating distributed deep learning with communication scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proc. MLSys*, 2019.

[25] 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 *Proc. ACM SIGPLAN PPoPP*, pages 120–134, 2022.

[26] Xianyan Jia, Le Jiang, Ang Wang, Jie Zhang, Xinyuan Li, Wencong Xiao, Yong Li, Zhen Zheng, Xiaoyong Liu, Wei Lin, et al. Whale: Scaling deep learning model training to the trillions. *arXiv preprint arXiv:2011.09208*, 2020.

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *Proc. MLSys*, 2019.

[28] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proc. USENIX OSDI*, pages 463–479, 2020.

[29] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[30] Aran Komatsuzaki, Joan Puigcerver, James Lee-Thorp, Carlos Riquelme Ruiz, Basil Mustafa, Joshua Ainslie, Yi Tay, Mostafa Dehghani, and Neil Houlsby. Sparse upcycling: Training mixture-of-experts from dense checkpoints. In *Proc. ICLR*, 2023.

[31] 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 preprint arXiv:2006.16668*, 2020.

[32] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. Base layers: Simplifying training of large, sparse models. In *Proc. USENIX ICML*, 2021.

[33] Hanxue Liang, Zhiwen Fan, Rishov Sarkar, Ziyu Jiang, Tianlong Chen, Kai Zou, Yu Cheng, Cong Hao, and Zhangyang Wang. M³vit: Mixture-of-experts vision transformer for efficient multi-task learning with model-accelerator co-design. In *Proc. NeurIPS*, 2022.

[34] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proc. ACL*, 2011.

[35] Ibrahim Naji. TSATC: Twitter Sentiment Analysis Training Corpus. In *thinknook*, 2012.

[36] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The Case for Tiny Tasks in Compute Clusters. In *Proc. USENIX HotOS*, 2013.

[37] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proc. ACM SOSP*, 2019.

[38] Andrey Proskurin. DeepSpeed: Advancing MoE inference and training to power next-generation AI scale. https://www.microsoft.com/en-us/research/blog/deepspeed-advancing-moe-inference-and-training-to-power-next-generation-ai-scale.

[39] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[40] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 2020.

[41] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale. *arXiv preprint arXiv:2201.05596*, 2022.

[42] 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. In *Proc. USENIX ATC*, 2021.

[43] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. Mesh-TensorFlow: Deep learning for supercomputers. In *Proc. ACM NeurIPS*, 2018.

[44] 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 preprint arXiv:1701.06538*, 2017.

[45] Liang Shen, Zhihua Wu, WeiBao Gong, Hongxiang Hao, Yangfan Bai, HuaChao Wu, Xinxuan Wu, Haoyi Xiong, Dianhai Yu, and Yanjun Ma. Se-moe: A scalable and efficient mixture-of-experts distributed training and inference system. *arXiv preprint arXiv:2205.10034*, 2022.

[46] 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 preprint arXiv:1909.08053*, 2019.

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

[48] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and generic collectives for distributed ml. In *Proc. MLSys*, 2020.

[49] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proc. EMNLP*, 2020.

[50] Zhewei Yao, Xiaoxia Wu, Conglong Li, Connor Holmes, Minjia Zhang, Cheng Li, and Yuxiong He. Random-ltd: Random and layerwise token dropping brings efficient training for large-scale transformers. *arXiv preprint arXiv:2211.11586*, 2022.

[51] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *Proc. USENIX ATC*, 2017.

[52] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and intra-operator parallelism for distributed deep learning. In *Proc. USENIX OSDI*, 2022.

[53] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. Mixture-of-experts with expert choice routing. *Proc. NeurIPS*, 2022.

[54] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. Designing Effective Sparse Expert Models. *arXiv preprint arXiv:2202.08906*, 2022.

[55] Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Tuo Zhao, and Jianfeng Gao. Taming sparsely activated transformer with stochastic experts. *arXiv preprint arXiv:2110.04260*, 2021.