Idle No More: Boosting Distributed Pipeline Training via FluidPipe

1st Mohammed Aljahdali
CEMSE
KAUST
Thuwal, Saudi Arabia
mohammed.kaljahdali@kaust.edu.sa

2nd Marco Canini
CEMSE
KAUST
Thuwal, Saudi Arabia
marco@kaust.edu.sa

***Abstract*—Large foundation models, such as language models with billions of parameters, have become pivotal in various fields, including natural language processing, healthcare, and AI-driven applications. These models require many GPUs to train or fine-tune, which presents a challenge when obtaining GPUs within the same cloud datacenter or region is not viable. Moreover, organizations with limited computational resources may acquire cloud-based resources but cannot fully offload to the cloud due to privacy, legal or data sovereignty concerns.**

In this paper, we propose FluidPipe, a distributed parallel training algorithm that is tailored for geo-distributed resources. Our algorithm takes inspiration from Federated Group Knowledge Transfer (FedGKT), a federated learning method that splits model training between clients and the server. Our approach greatly reduces the runtime over pipeline parallelism at the cost of slightly changing the learning objective. While effective, our approach still suffers from idle times due to synchronization and data communication delays, especially in geo-distributed scenarios where latency and bandwidth issues lead to substantial idle times. To address this we incorporate idle training, that is, carefully orchestrated additional training steps during idle times enhancing resource utilization.

I. Introduction

Current state-of-the-art machine learning models are getting larger and require increasing amounts of compute resources—particularly GPUs. Cloud computing has provided a scalable way to obtain these resources. However, with the rapid expansion of ML models, the demand for GPUs has surged, making it increasingly difficult to acquire them within the same geographical zone [1].

Traditional distributed parallel training methods, such as pipeline parallelism, can still function with GPUs distributed across different regions. However, due to the inherent challenges of high latency and limited bandwidth in these settings, such approaches may not always be the most efficient. This raises the question: *can we develop a more efficient training algorithm that is less affected by latency constraints?*

The primary cause of this inefficiency is the synchronization of data dependencies. Since progressing through a batch requires frequent communication of intermediate features and gradients—proportional to the micro-batch size—training efficiency is significantly hindered. While pipelining can mitigate this delay by overlapping computation and communication, its effectiveness diminishes in scenarios where GPUs are distributed across different regions with high latency.

To address this, we propose FluidPipe an asynchronous training algorithm inspired by Federated Group Knowledge Transfer (FedGKT) [2] a Federated Learning [3] method. In FluidPipe, the model is split into two parts, similar to model parallelism and Split Learning (SL) [4], [5], but with a key modification: we introduce a classification head at the end of the first part.

During training, the machine responsible for the first part processes its segment of the model and, after each forward pass, sends intermediate features—produced by the last layer before the classification head—to the second machine. The first machine then proceeds with a backward pass using the loss computed from its own classification head. This classification head enables machine 1 to train without synchronizing with machine 2 at every batch. However, machine 2 remains dependent on receiving intermediate features from machine 1. A limitation of this formulation is that part 1 of the model does not learn from part 2, unlike model parallelism and SL where gradients flow back from part 2 to part 1. To overcome this, we incorporate knowledge distillation [6], [7] at machine 1, starting from the second epoch. At the end of the first epoch, machine 2 synchronizes with machine 1 by sending logits for all training data. This synchronization occurs once per epoch to minimize communication overhead, but this step may still introduce a delay. The accumulation of delays at every training step at machine 2 mainly causes this delay. To further enable bidirectional collaboration, machine 1 also sends logits alongside intermediate features to machine 2. Unlike FedGKT, our approach involves only two machines instead of multiple clients and a single server. Additionally, we modify the communication frequency to allow overlapping training across both machines.

Even though our proposed algorithm significantly reduces communication overhead and synchronization requirements, it does not completely eliminate them. Machine 2 must still wait for features before starting an iteration, and machine 1 must wait for machine 2 to finish the epoch and send all logits. To further improve efficiency and take advantage of idle periods caused by synchronization, we introduce opportunistic training. This mechanism leverages the structure of our algorithm to easily train during otherwise wasted time. Specifically:

- At machine 2: While waiting for new features, it can

TABLE I: Algorithms Notation and Hyperparameters

Symbol	Meaning
$\mathcal{D}$	Dataset, with N_b mini-batches
$\theta_1 = (\theta_1^h, \theta_1^o)$	M1 partial model (first part of the split model + classification head)
θ_2	M2 partial model (the 2nd half of the split model)
E	Number of epochs
$\mathcal{L}_{\text{task}}$	task loss (e.g., cross-entropy)
$\mathcal{L}_{\text{KD}}$	distillation loss (e.g., KL-divergence)
η	learning rate

use previously received features to perform additional training steps.

- At machine 1: When it completes an epoch and is waiting for machine 2 to finish, it can continue training beyond the current epoch.

By utilizing idle periods, our approach enhances the algorithm efficiency, mitigating the impact of latency and bandwidth limitations.

II. FluidPipe Design

In this section, we describe FluidPipe in detail. FluidPipe breaks the 2-way data dependency at every training step found in both model and pipeline parallelism. FluidPipe instead only sends data in one direction, by adding a classification head to the first part of model gradient propagation through the model parts is replaced with knowledge distillation. In Fluidpipe machine 1 (M1) sends intermediate features and logits to machine 2 (M2), while M2 sends no gradient back or anything right away making a 1-way data dependency from M1 to M2. Instead, M2 delegates its data communication to the end of the training epoch, at which point it sends all of the logits of the training data. Starting from the 2nd epoch machine 1 would use the logits received from the M2 to distill in addition to the task loss. The FluidPipe algorithm is described in detail for M1 algorithm 1 and M2 algorithm 2.

To enhance efficiency—measured as GPU utilization—we propose performing training during idle periods referred to as *opportunistic training*. The idea is straightforward: train on available data until the machine becomes active again. However, naive implementations of idle training could degrade model quality due to several challenges.

M1 experiences idle times after completing an epoch while waiting for M2 to finish its epoch. During this period, M1 can perform idle training by starting new epochs on its local data until server updates are received. However, excessive additional steps risk overfitting, as the duration of idle training depends on network conditions.

In opportunistic training, M1 can utilize idle time by accessing *all* training data, whereas M2 is limited to the data points it has *already* received from M1. Idle training on M2 occurs whenever M2 is waiting for additional data points during an epoch. Let i be the index of the last data point received by M2, and let N_s be the total number of data points in the current

Algorithm 1 FluidPipe: Machine 1 (M1) Procedure

Initialize a storage $\mathcal{P}_2 \leftarrow \{\}$ {Will hold M2's logits.}
for $epoch \leftarrow 1$ to E **do**
  (A) Local Training Loop on M1
  for each mini-batch B in $\mathcal{D}$ **do**
    1. $z_1 = f(\theta_1^h; B)$ {Feature extraction}
    2. $p_1 = f(\theta_1^o; z_1)$ {Local logits}
    3. **Send** $(z_1, p_1, \text{sample\_IDs})$ to M2 {Non blocking}
    4. $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{task}}(B; p_1)$
    if $epoch \geq 2$ **then**
      6.1 $p_2 \leftarrow \mathcal{P}_2[\text{sample\_IDs}]$ {Teacher (θ_2) logits}
      6.2 $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{total}} + \mathcal{L}_{\text{KD}}(p_1, p_2)$
    7. `Backward` on M1: $\theta_1 \leftarrow \theta_1 - \eta \nabla_{\theta_1} \mathcal{L}_{\text{total}}$

  8. **Receive** $\mathcal{P}_2$ from M2 {Logits for all samples.}
  (B) Opportunistic Training
  while not all p_2 are received **do**
    9. Sample batch B (for opportunistic steps)
    10. $p_1 \leftarrow f(\theta_1^o; f(\theta_1^h; B))$
    11. $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{task}}(B; p_1)$
    if $epoch \geq 2$ **then**
      13.1 $p_2 \leftarrow \mathcal{P}_2[\text{sample\_IDs}]$
      13.2 $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{total}} + \mathcal{L}_{\text{KD}}(p_1, p_2)$
    14. `Backward` on M1: $\theta_1 \leftarrow \theta_1 - \eta \nabla_{\theta_1} \mathcal{L}_{\text{total}}$

Output: Final θ_1 {from M1}

Algorithm 2 FluidPipe: Machine 2 (M2) Procedure

for $epoch \leftarrow 1$ to E **do**
  (C) Local Training Loop on M2
  Initialize a dictionary $\mathcal{P}_2 \leftarrow \{\}$ {Logits, to be sent to M1}
  Initialize a dictionary $\mathcal{D}_{M1} \leftarrow \{\}$ {Store for opp. training}
  for each mini-batch B in $\mathcal{D}$ **do**
    1. **Receive** $(z_1, p_1, \text{sample\_IDs})$ from M1
    (D) Opportunistic Training
    while Receive is not complete **do**
      2.1 Sample batch $(z_1', p_1', \text{sample\_IDs}')$ from D_{M1}
      2.2 $p_2 \leftarrow f(\theta_2; z_1')$
      2.3 $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{task}}(B'; p_2) + \mathcal{L}_{\text{KD}}(p_1', p_2)$
      2.4 `Backward` on M2: $\theta_2 \leftarrow \theta_2 - \eta \nabla_{\theta_2} \mathcal{L}_{\text{total}}$
      2.5 $\mathcal{P}_2[\text{sample\_IDs}'] \leftarrow p_2'$ {Update the logits}
    3. $p_2 \leftarrow f(\theta_2; z_1)$
    4. $\mathcal{L}_{\text{total}} \leftarrow \mathcal{L}_{\text{task}}(B; p_2) + \mathcal{L}_{\text{KD}}(p_1, p_2)$
    5. `Backward` on M2: $\theta_2 \leftarrow \theta_2 - \eta \nabla_{\theta_2} \mathcal{L}_{\text{total}}$
    6. $\mathcal{P}_2[\text{sample\_IDs}] \leftarrow p_2$ {Store logits for this batch}
    7. $\mathcal{D}_{M1}[\text{sample\_IDs}] \leftarrow (z_1, p_1)$

  (E) End-of-Epoch Bulk Send
  7. **Send** entire $\mathcal{P}_2$ (all p_2) to M1 {Blocking}

Output: Final θ_2 {from M2}

epoch. If M2 becomes idle while awaiting data points beyond index $i+1$, it can perform idle training on the set

$$S \;=\; \{1, 2, \ldots, i\}.$$

When $i = 0$, no data have been received, so S is empty and no idle training is possible. Conversely, when $i = N_s$, S contains all data points in the epoch, indicating that the epoch is complete and a new one can begin. Limited data availability makes M2 opportunistic training more challenging. To this end, we propose two data sampling strategies that circumvent these challenges. These strategies can be applied at both M1 and M2.

a) ***Random Sampling.*** *:* In this strategy, whenever an opportunistic training step is started, the sampler will randomly sample $|B|$ data points to create a training batch–$|B|$ is the batch size. This randomness serves as a form of regularization that helps mitigate the challenges of opportunistic training. Moreover, the approach is straightforward, making it a useful baseline and a convenient sanity check against more sophisticated sampling strategies.

b) ***Difficulty Sampler.*** *:* We maintain a *composite score* $C(i,t)$ for each data point i at update step t by combining multiple metrics (e.g., cross-entropy or distillation losses) with weights and signs indicating whether each metric should be maximized $(+1)$ or minimized (-1). Formally, if $x_m(i,t)$ is the value of metric m for data point i at time t, and $s_m \in \{+1, -1\}$ denotes its orientation, then

$$C(i,t) \;=\; \sum_{m=1}^{M} w_m \left(s_m\, x_m(i,t)\right),$$

where $w_m \geq 0$ is the weight for metric m, and M is the total number of used metrics. We store these composite scores in a local history $\mathcal{H}_i$ for each data point, and compute a *difficulty slope* $D(i)$ via simple linear regression on the most recent entries of $\mathcal{H}_i$. Specifically, if $\mathcal{H}_i = [C(i,t_1), \ldots, C(i,t_L)]$ with $L \geq 2$,

$$D(i) \;=\; \frac{\sum_{k=0}^{L-1}\big(k-\overline{k}\big)\,\big(C(i,t_{k+1}) - \overline{C}_i\big)}{\sum_{k=0}^{L-1}\big(k-\overline{k}\big)^2 + \varepsilon},$$

where $\overline{k} = (L-1)/2$ and $\overline{C}_i$ is the mean of the $C(i,t_k)$ values in the history. If $L < 2$, we set $D(i) = 0$. In *Difficulty Sampler*, we *always* use $D(i)$ to prioritize points for sampling, so data points with larger slopes (positive $D(i)$) are deemed "harder" and selected first; points with $D(i) = 0$ are effectively given a neutral ranking (e.g., newly added data).

c) ***EH-Difficulty Sampler.*** *:* Building on this framework, *EH-Difficulty Sampler* classifies each data point into *easy*, *hard*, or *diversity* pools based on a combination of the last composite score $C(i,t_L)$ and slope $D(i)$. Specifically, once we have accumulated enough updates, we derive *easy* and *hard* thresholds by taking the 30% and 70% quantiles of the last composite score for each data point $\mathcal{H}_{\text{last}} = \{C(i,t_L) | i \in \mathcal{D}\}$. We then label a point `easy` if its composite score $C(i,t_L)$ falls below the easy threshold and $D(i)$ is not positive, `hard` if its score exceeds the hard threshold or $D(i)$ is positive, and `diversity` otherwise. At sampling time, we draw a user-defined fraction of data points from each category (e.g., 30% from `easy`, 30% from `hard`, and the remainder from `diversity`), although these fractions can be adjusted as desired. This design ensures that *both* newly emerging hard examples and simpler ones are included, while preserving variety across the difficulty spectrum and preventing any single category from dominating the batch.

III. Analytical Cost Models

We compare FluidPipe to pipeline parallelism, restricted to $p = 2$ stages. If more parallelism is required within a single FluidPipe stage, we could apply pipeline parallelism within each stage of FluidPipe independently. For instance, if we have 4 GPUs in a US region and 4 GPUs in an EU region, we can run FluidPipe between the US and EU groups, while employing 3D parallelism *within* each region.

a) ***Two-Stage Pipeline Parallelism.*** *:* We split the model into two stages, each hosted on a separate machine. Let:

- t_A: forward + backward compute time on M1 (per micro-batch),
- t_B: forward + backward compute time on M2,
- α: time to send activations forward (M1 $\rightarrow$ M2),
- β: time to send gradients backward (M2 $\rightarrow$ M1),
- m: number of micro-batches into which each mini-batch is subdivided,
- N_b: number of mini-batches per epoch.

A common approximation for the time per mini-batch, once the pipeline is warmed up, is:

$$T_{\text{pipeline}} \;\approx\; \big(m + (P-1)\big)\, \max\big(t_A + \alpha,\; t_B + \beta\big), \qquad (1)$$

where $P = 2$ for a two-stage pipeline, so $m+(P-1) = m+1$. Hence, for one epoch:

$$T_{\text{pipeline, epoch}} \;\approx\; N_b\,\big(m+1\big)\, \max\big(t_A + \alpha,\; t_B + \beta\big). \qquad (2)$$

This formula captures how pipeline parallelism overlaps compute and communication, but also shows that *both* forward (A $\rightarrow$ B) and backward (B $\rightarrow$ A) transfers are needed per micro-batch.

b) ***FluidPipe.*** *:* Similar to the two-stage pipeline parallelism, we define:

- τ_1: forward+backward time at *Stage 1*, including the extra classification head overhead and KD computation cost (starting from epoch 2) on M1,
- τ_2: forward+backward time at *Stage 2*, including a small KD computation cost M2,
- α: communication overhead per mini-batch from M1 $\rightarrow$ M2 (sending (z_1, p_1)),
- γ: *bulk* overhead at the end of each epoch for sending the final M2 logits $\{\mathcal{P}_2\}$ to M1,
- N_b: total mini-batches per epoch,

Here, τ_1 is larger than the classical stage-1 time t_A since M1 has the classification head and local KD. Meanwhile, τ_2 is roughly t_B plus a minor KD overhead (relative to the size of batch and micro-batch).

*c) **Concurrency-Aware Epoch Cost.** :* M1 spends τ_1 time on forward+backward for each mini-batch, while M2 spends τ_2 time on forward+backward for the same mini-batch, after receiving (z_1, p_1). Also, each mini-batch requires a communication overhead α from M1 to M2. But backward at iteration i and the forward+backward at any future iteration $i+1$ of M1 can overlap with the forward+backward of M2 at iteration i. Consequently, over N_b mini-batches:

- **M1 total local time:** $N_b \times \tau_1$.
- **M2 total local time + forward data transfer:** $N_b \times (\tau_2 + \alpha)$, because M2 cannot begin its forward pass for batch i until it has received (z_1, p_1).
- **Bulk transfer at epoch end:** γ, for sending final M2 logits back to M1.

Since M1 and M2 run concurrently (M2 is effectively "one batch behind" M1), the total epoch time is approximately:

$$T_{\text{FluidPipe, epoch}} \;\approx\; \max\Big(N_b \times \tau_1,\; N_b \times (\tau_2 + \alpha)\Big) \;+\; \gamma.$$

In other words, we take whichever stage is the bottleneck for the N_b mini-batches, and then add the one-time bulk cost γ. This approach mirrors the pipeline formula $\max\big(t_A + \alpha,\, t_B + \beta\big)$ but omits the per-step backward gradient transfer (i.e., the β term) in favor of a single bulk γ each epoch. If $\gamma < N_b\,\beta$, FluidPipe can be significantly cheaper in high-latency or bandwidth-limited scenarios.

*d) **Fully Sequential Approximation.** :* For completeness, if there were *no* concurrency between M1 and M2 iterations, one might (over)estimate each mini-batch cost as $(\tau_1 + \alpha + \tau_2)$ and then sum over N_b batches, plus γ. That is,

$$T_{\text{FluidPipe, epoch}} \;=\; N_b\,(\tau_1 + \alpha + \tau_2) \;+\; \gamma.$$

However, this sequential view **overestimates** the runtime when overlap is present; in practice, the concurrency-aware model is more accurate.

*e) **Comparison to Two-Stages Pipeline Parallelism.** :* Once warmed up with micro-batches, pipeline parallelism is:

$$T_{\text{pipeline}} \;\approx\; (m+1)\,\max\big(\tau_1 + \alpha,\; \tau_2 + \beta\big),$$

and over N_b mini-batches per epoch,

$$T_{\text{pipeline, epoch}} \;\approx\; N_b\,(m+1)\,\max\big(\tau_1 + \alpha,\; \tau_2 + \beta\big).$$

By contrast, FluidPipe eliminates step-by-step gradients (the β term) and replaces them with a one-time bulk γ at epoch end. Hence, for concurrency we get:

$$T_{\text{FluidPipe, epoch}} \;\approx\; \max\Big(N_b\,\tau_1,\; N_b\,(\tau_2 + \alpha)\Big) \;+\; \gamma,$$

leading to simpler one-directional communication at each mini-batch, plus a single epoch-level transfer for final logits.

*f) **Communication Overhead Analysis.** :* Finally, let us compare communication volume. Suppose:

- α_{batch}: cost of sending z_1 per mini-batch in FluidPipe and classical pipeline,
- α_p: cost of sending p_1 per mini-batch in FluidPipe,
- β_{batch}: cost of sending backward gradients per mini-batch in a classical pipeline,
- γ: end-of-epoch bulk for FluidPipe,
- N_b: total mini-batches in an epoch (ignoring micro-batching).

Two-Stage Pipeline Parallelism:

$$\text{Total communication per epoch} \;=\; N_b \times (\alpha_{\text{batch}} + \beta_{\text{batch}}).$$

FluidPipe:

$$\text{Total communication per epoch} \;=\; N_b \times (\alpha_{\text{batch}} + \alpha_p) \;+\; \gamma.$$

For FluidPipe to incur lower communication, we need:

$$\begin{aligned} N_b\,(\alpha_{\text{batch}} + \alpha_p) + \gamma \;&<\; N_b\,(\alpha_{\text{batch}} + \beta_{\text{batch}}) \\ \iff \quad (N_b\,\alpha_p) + \gamma &< N_b\,\beta_{\text{batch}}. \end{aligned}$$

Note that volume of γ is the same as $N_b\,\alpha_p$, so we can simplify further and say FluidPipe will incur lower communication if and only if:

$$2(N_b\,\alpha_p) < N_b\,\beta_{\text{batch}}$$

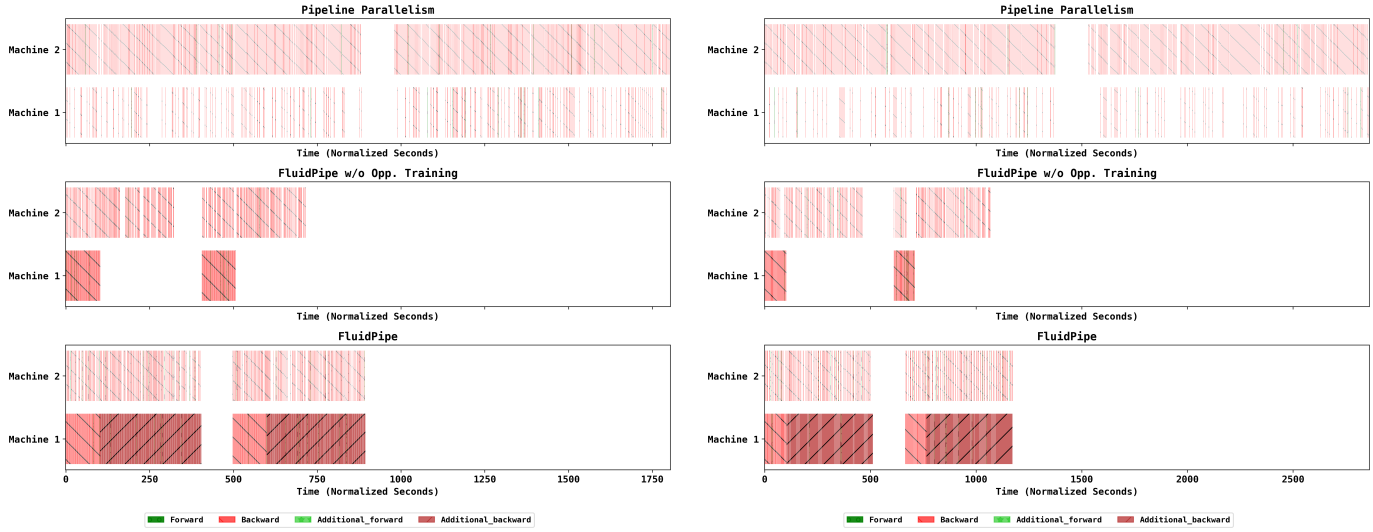
Since the gradient tensor for half the model (e.g., 6 encoder layers of a 12-layer BERT) often far exceeds the size of the logits, thus $2(N_b\,\alpha_p) \;<\; N_b\,\beta_{\text{batch}}$ almost always holds in practice.

IV. Experiments & Results

In our experiment, we focus on a small setup that can answer our main question: *can we develop a more efficient training algorithm that is less affected by latency constraints?* We run our experiment on two machines each with one A100 GPU. We use the *tc (traffic control)* program on Linux to simulate real-world scenarios using latency. Specifically, we picked 3 different latencies: 1ms, 25ms, and 50ms. These values can represent various scenarios, such as, within data-center latency, same-region cross-zone latency, or cross-region latency. These values are not exhaustive, however, they are enough to show the effect of latency on the training algorithm runtime and efficiency. We use the 12-layer BERT *base* [8] model, which we split into two halves. Part 1 has the embedding layer and the first 6 encoder layers, while part 2 has the last 6 encoder layer and the classification head. We load the *bert-base-uncased* pre-trained weights. Our main baseline is this BERT model trained using pipeline parallelism. This baseline gives us the expected model quality and runtime if we train using the standard techniques. In all of the experiments, we train for 5 epochs.

As for the datasets, we use tasks from the well-known benchmarks GLUE [9] and SuperGLUE [10]. The selected tasks are: RTE, CoLA, and BoolQ. Additionally, we train and evaluate using a subset of the IMDB Reviews dataset.

We break down our results into five parts: (i) Iterations Timeline, where we visualize how training steps progress under different latencies; (ii) Runtime & Accuracy, examining overall performance trade-offs; (iii) Sampling Strategies, exploring how data selection affects opportunistic training; (iv) Number of Opportunistic Training Steps, assessing the impact of limiting fixing the number opportunistic training steps; and (v) Regularized Opportunistic Training, discussing



(a) Latency = 1ms. **(b) Latency = 50ms.**

Fig. 1: Iteration timeline comparing Pipeline Parallelism, FluidPipe, and FluidPipe with opportunistic training. This plot focuses on the second and third epochs, highlighting the effect of idle time caused by latency on the progression of iterations.

how possible regularization techniques for FluidPipe. Parts (iii) through (v) are in the appendix.

a) ***Iterations Timeline.*** *:* We visualize the progression of training steps under different latencies seen in fig. 1, comparing how Pipeline Parallelism and FluidPipe advance between mini-batches and how network delay affects their performance. We record timestamps at the start and end of each forward pass, backward pass, and model update (the latter is considered part of the backward phase). We also differentiate the events that occur during opportunistic training. Notably, *even without opportunistic training*, FluidPipe runs faster than Pipeline Parallelism. Although FluidPipe adds extra overhead—due to the classifier on Machine 1 and the distillation loss computation—these costs are outweighed by the communication and synchronization overheads of sending gradients and coordinating each batch in Pipeline Parallelism. Lastly, when latency is very low (1ms) overheads of opportunistic training make FluidPipe slower.

b) ***Runtime & Accuracy.*** *:* We compare the overall training duration (wall-clock time) against final model accuracy for four different tasks under various latencies as shown in fig. 2. This setup allows us to gauge both performance stability and how each method scales in increasingly negative network latency.

Across all four tasks, we observe that *Pipeline Parallelism* remains competitive at low latency (1 ms), but its performance degrades significantly as latency increases to 25 and 50 ms. In contrast, *FluidPipe* —by eliminating per-step gradient transfers— achieves a notably faster runtime across different latencies, allowing it to exceed the overall efficiency of Pipeline Parallelism under more challenging network conditions.

Among the *FluidPipe* sampling strategies, random sampling and EH-Difficulty sampling yield the best model quality at 25 and 50 ms latency. By comparison, difficulty-only sampling exhibits larger performance fluctuations and often lags behind the other approaches. Moreover, FluidPipe without opportunistic training underperforms its opportunistic variant, *demonstrating that extra training steps during idle periods are beneficial*. Based on these observations, we mainly focus on FluidPipe with random or EH-Difficulty sampling in subsequent analyses.

Looking at individual tasks, FluidPipe (both random and EH-Difficulty) outperforms Pipeline Parallelism on *IMDB*, excelling in both accuracy and runtime. On *BoolQ*, FluidPipe provides a slight accuracy edge over Pipeline Parallelism. In *CoLA*, Pipeline Parallelism achieves the highest final accuracy overall, but under high latency, FluidPipe can reach a competitive accuracy much sooner—potentially enabling early stopping. Finally, for *RTE*, accuracies are more irregular; intriguingly, FluidPipe without opportunistic training scores best followed by its opportunistic training variants, though its unclear why FluidPipe without opportunistic training performed the best.

In summary, while FluidPipe introduces a small overhead from the classification head and knowledge-distillation computations, it avoids the frequent synchronization costs of Pipeline Parallelism. As latency increases, FluidPipe tends to maintain or improve its accuracy lead, largely thanks to its opportunistic training mechanism.

V. Conclusion

We have presented FluidPipe, a novel distributed training algorithm designed for geo-distributed and communication-limited environments. By decoupling forward and backward gradients across two model partitions and introducing a lightweight knowledge-distillation step on the first partition, FluidPipe effectively reduces synchronization costs and mitigates idle times. Our experiments on multiple NLP tasks—including

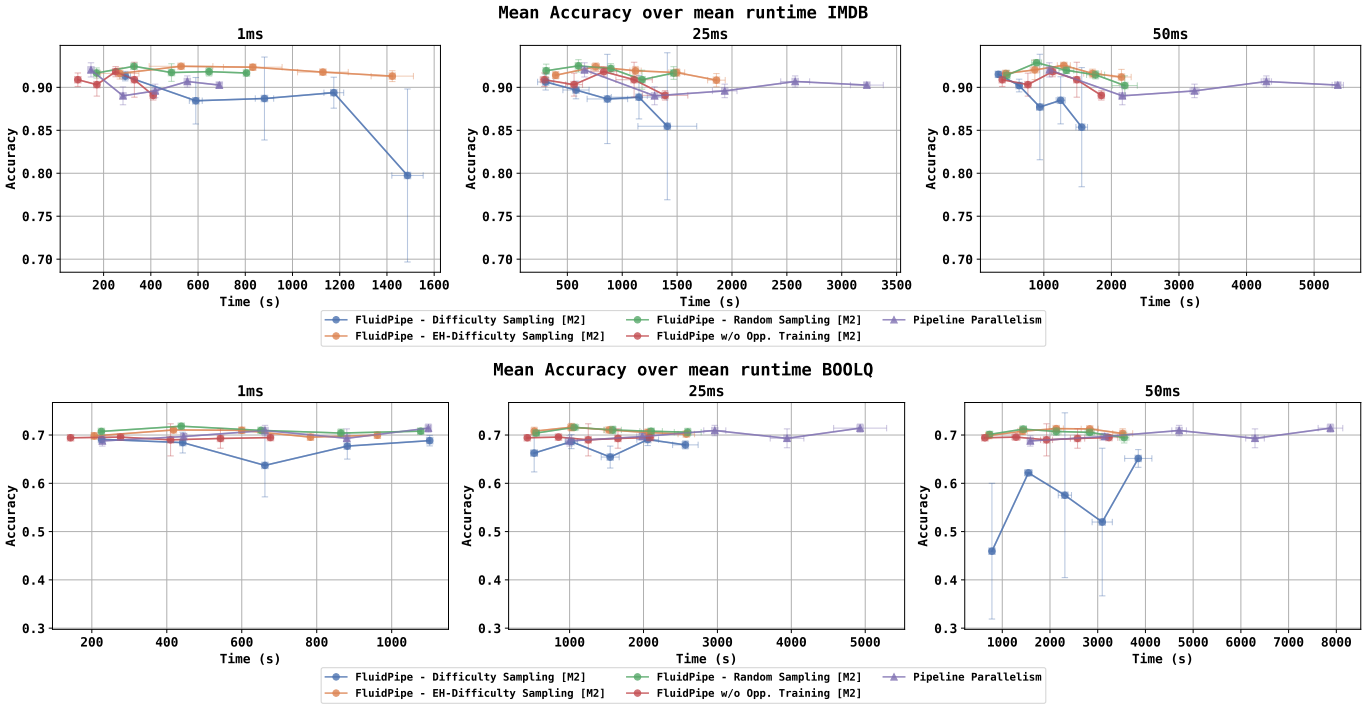


Fig. 2: Accuracy vs runtime on IMDB and BoolQ datasets. Each Row has the same experiment but with different latencies (1ms, 25ms, 50ms). Each plot is generated by running the experiments three times with different random seeds, where error bars on both axes represent the standard deviation in runtime and accuracy.

BoolQ, CoLA, RTE, and IMDB—demonstrate that FluidPipe not only maintains competitive or superior accuracy under various latencies but also significantly cuts overall runtime compared to classical pipeline parallelism. Furthermore, our opportunistic training mechanism capitalizes on inevitable idle phases, enhancing resource utilization without excessive overhead. Taken together, these findings suggest that FluidPipe offers a promising direction for large-scale training in distributed, latency-prone settings. Future work includes extending FluidPipe to multi-stage scenarios, improving opportunistic training strategies, and data sampling techniques.

In this paper, we mainly compared with vanilla pipeline parallelism, and at the small case where we have 2 GPUs and 2 stages only. In the future, we aim to extend our experiments to more GPUs and compare against different pipeline parallelism algorithm such as GPipe [11].

References

[1] F. Strati, P. Elvinger, T. Kerimoglu, and A. Klimovic, "Ml training with cloud gpu shortages: Is cross-region the answer?" in *Proceedings of the 4th Workshop on Machine Learning and Systems*, 2024, pp. 107–116.

[2] He, Annavaram, and others, "Group knowledge transfer: Federated learning of large cnns at the edge," *Advances in neural information processing systems*, 2020.

[3] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y. Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," in *AISTATS*, A. Singh and J. Zhu, Eds., 2017.

[4] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," Dec. 2018.

[5] M. G. Poirot, P. Vepakomma, K. Chang, J. Kalpathy-Cramer, R. Gupta, and R. Raskar, "Split learning for collaborative deep learning in healthcare," Dec. 2019.

[6] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015.

[7] C. Buciluǎ, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *KDD*, 2006.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *NAACL-HLT*, 2019. [Online]. Available: https://aclanthology.org/N19-1423

[9] A. Wang, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[10] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "Superglue: A stickier benchmark for general-purpose language understanding systems," *Advances in neural information processing systems*, vol. 32, 2019.

[11] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

APPENDIX

In this section, we continue the remaining parts of the results from the main text.

a) ***Sampling Strategies.*** *:* In addition to our earlier methods, we introduce two priority-based samplers that select data points according to their *task loss* (cross-entropy) or *distillation loss* (KL-divergence) –samples with higher loss are prioritized. These serve as "sanity checks" for our difficulty-based sampler: if they substantially outperform the difficulty approach, it suggests the latter may be less valid. In fig. 3, we compare these naive strategies against the random and difficulty-based samplers on *BoolQ* and *CoLA*. Overall, the task-loss sampler consistently underperforms the others, whereas the distillation-loss sampler shows more promise—falling short of random and EH-Difficulty, yet still achieving reasonable results.

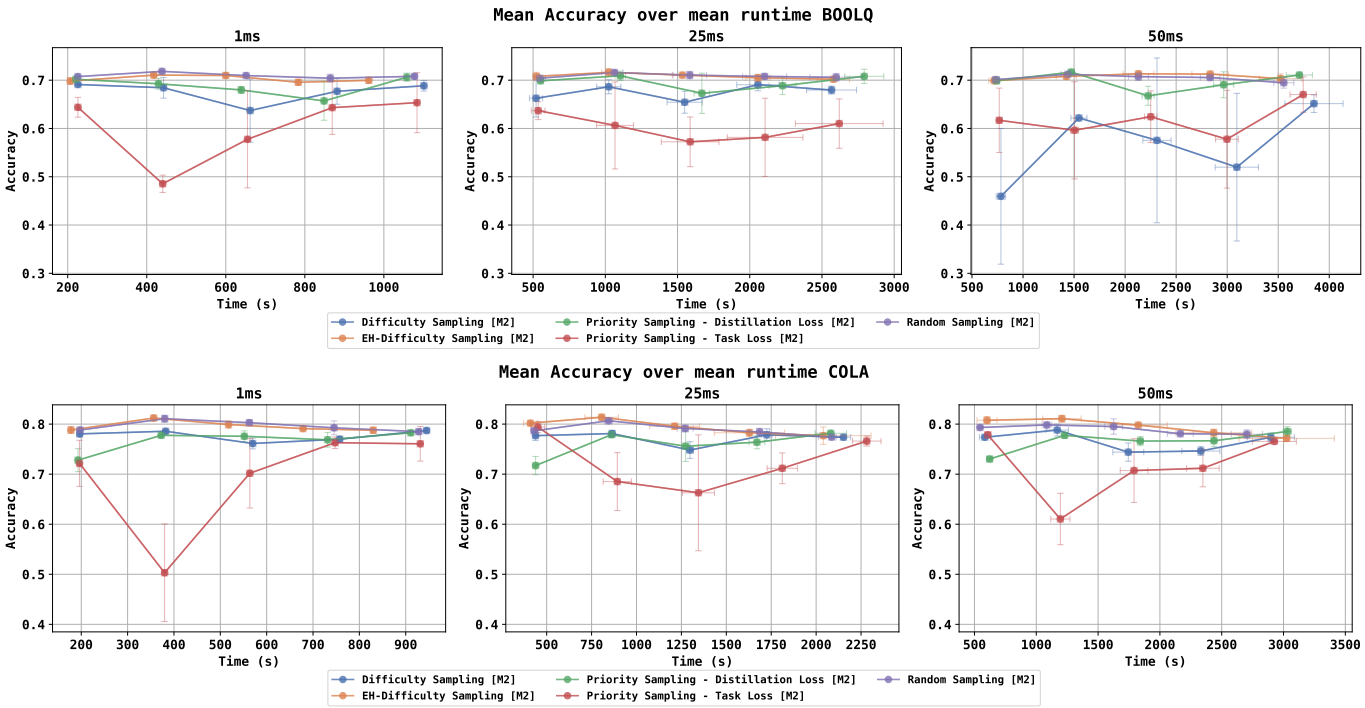


Fig. 3: Accuracy vs runtime comparing multiple sampling strategies on two different datasets.

b) ***Number of Opportunistic Training Steps.*** *:* In this experiment, we impose a fixed upper bound on the *number of opportunistic training steps per epoch* (e.g., x extra steps), rather than letting FluidPipe exploit all idle time freely. We choose the highest x by measuring how many idle-time steps naturally occur at 50ms latency on the *CoLA* dataset, then evaluate smaller values to see how limiting extra steps affects both runtime and model quality. As shown in fig. 4, increasing the cap on opportunistic steps consistently boosts final accuracy, showing that using idle periods more extensively yields better performance—yet remains faster than pipeline parallelism, given the same overall latency.

c) ***Regularized Opportunistic Training.*** *:* To examine whether repeatedly using the same data during idle-time steps risks overfitting or instability, we introduce three regularization techniques specific to opportunistic training: (1) scaling down the task loss, (2) scaling down the distillation loss, and (3) applying dropout. By toggling each approach on or off, we obtain $2^3 = 8$ configurations, illustrated in fig. 5. Examining individual techniques, we find that applying dropout alone has minimal impact on final accuracy, whereas scaling down the task loss degrades performance—indicating the task loss remains important in idle-time updates. In contrast, scaling down the distillation loss often proves beneficial, likely preventing an excessive focus on teacher outputs –which could be a stale signal we shouldn't overuse. Overall, no combination surpasses our non-regularized baseline except for the single scenario in which we scale down the distillation loss alone. Note that we apply each technique to *both* M1 and M2 during opportunistic training. Investigating these regularization methods *individually* at M1 or M2—and in all possible pairwise combinations—would further expand the parameter space, as each new configuration requires multiple runs (each lasting around an hour) to ensure statistically robust conclusions. As a result, a thorough evaluation of these finer-grained settings remains an avenue for future work.

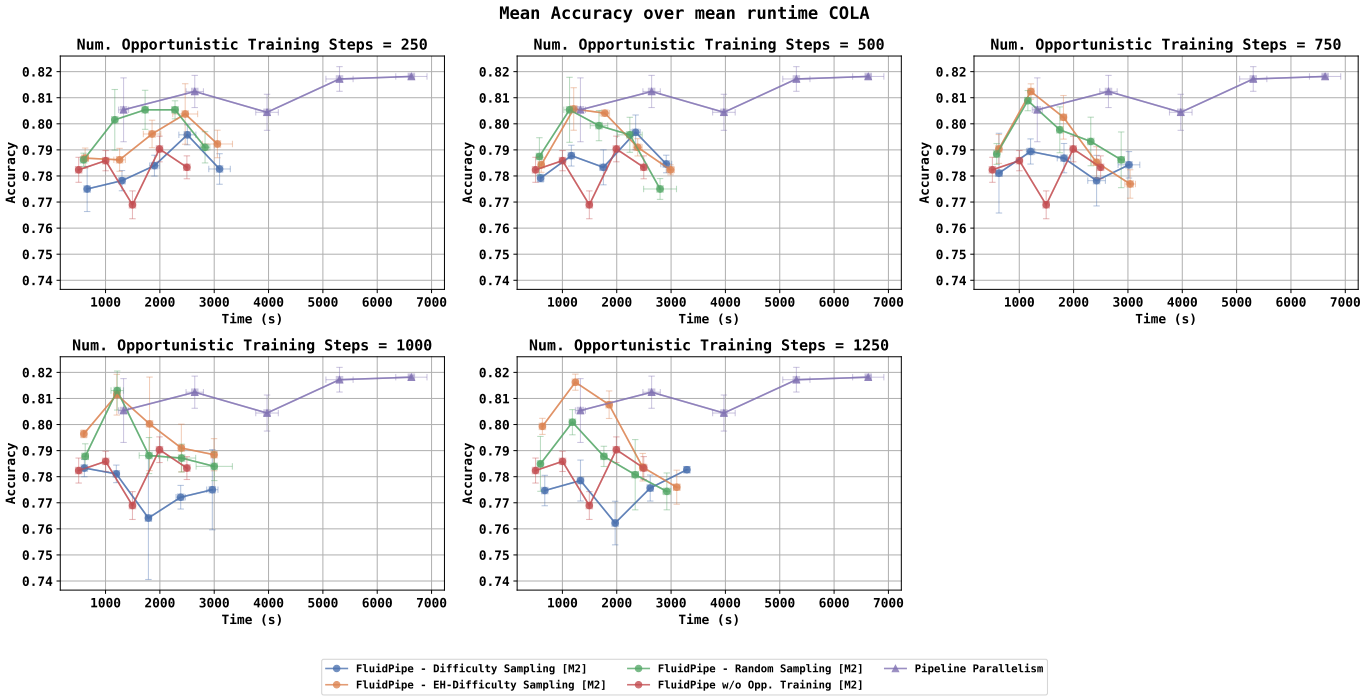


Fig. 4: Accuracy over runtime when limiting the number of opportunistic training steps. All of the experiments are ran with 50ms latency.

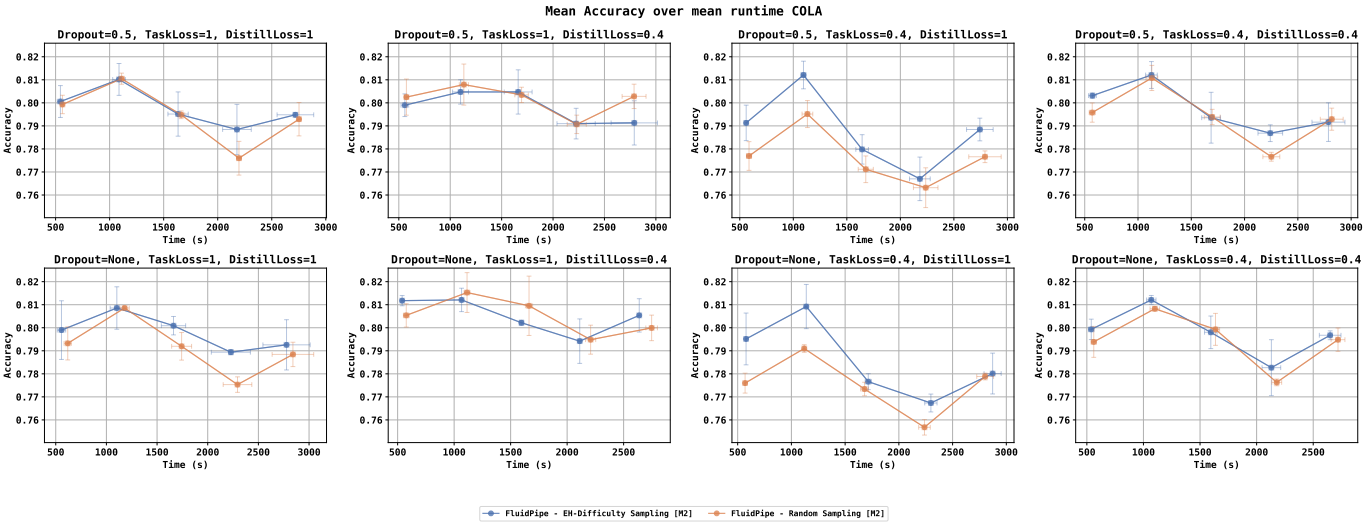


Fig. 5: Effects of different regularization techniques applied to opportunistic training under 50ms latency. We combine three methods—scaling the task loss, scaling the distillation loss, and dropout—in all possible ways (eight variants), comparing their accuracy and runtime. TaskLoss and DistillLoss refer to the weight of their respective losses, the values are either 1 (no scaling down) or 0.4. Dropout refers to whether we apply dropout or not, and the its probability.