TITANIC: Towards Production Federated Learning with Large Language Models

Ningxin Su, Chenghao Hu, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

Bo Li
Department of Computer Science and Engineering
Hong Kong University of Science and Technology

***Abstract*—With the recent surge of research interests in Large Language Models (LLMs), a natural question that arises is how pre-trained LLMs can be fine-tuned to tailor to specific needs of enterprises and individual users, while preserving the privacy of data used in the fine-tuning process. On the one hand, sending private data to cloud datacenters for fine-tuning is, without a doubt, unacceptable from a privacy perspective. On the other hand, conventional federated learning requires each client to perform local training, which is not feasible for LLMs with respect to both computation costs and communication overhead, since they involve billions of model parameters. In this paper, we present TITANIC, a new distributed training paradigm that allows LLMs to be fine-tuned in a privacy-preserving fashion directly on the client devices where private data is produced, while operating within the resource constraints on computation and communication bandwidth. TITANIC first optimally selects a subset of clients with an efficient solution to an integer optimization problem, then partitions an LLM across multiple client devices, and finally fine-tunes the model with no or minimal losses in training performance. A primary focus in the design of TITANIC is its feasibility in real-world systems: it is first and foremost designed for production-quality systems, featuring a model-agnostic partitioning mechanism that is fully automated. Our experimental results show that TITANIC achieves superior training performance as compared to conventional federated learning, while preserving data privacy and satisfying all constraints on local computation and bandwidth resources.**

I. INTRODUCTION

Originally stimulated by the advent of ChatGPT and recently reinforced by the introduction of Llama 2 [1] from Meta AI, it goes without saying that the recent surge in research interests on generative artificial intelligence in general, and Large Language Models (LLMs) in particular, will become one of the most important research topics in the history of computing. It may take months of time, trillions of tokens, and hundreds of GPUs to train an LLM. However, once pre-trained, it has been widely recognized that LLMs can be fine-tuned with domain-specific knowledge for a wide variety of downstream tasks, with much less computational resources.

When fine-tuning LLMs using data from corporations and individual users, it is natural to think about a fundamental question: how do we fine-tune LLMs while preserving the privacy of data? Intuitively, the best way to preserve data privacy is to train locally on each client device where data is originated and produced. This has been the tenet in the design philosophy of federated learning (FL): private data is never sent to cloud datacenters to start the fine-tuning process; instead, the fine-tuning process should be carried out locally on clients where private data resides.

However, conventional FL is unfit for fine-tuning LLMs for two important reasons. *First and more importantly*, conventional FL requires a model to be trained locally on each client device, but the fine-tuning process over LLMs takes a tremendous amount of computational resources. As an LLM contains billions of parameters (*e.g.*, Llama 2 models contain 7, 13, or 70 billion parameters), local fine-tuning depends on the availability of GPUs with large amounts of GPU memory. For example, even fine-tuning the smallest 7B Llama 2 model in full precision with Low Rank Adapter (LoRA) [2] requires around 32 GB of GPU memory (using a batch size of 4), as optimizer states need GPU memory as well. Without using LoRA, far more GPU memory is necessary for fine-tuning such a large model. Existing work has studied how model quantization and parameter-efficient fine-tuning help reduce resource needs, but quantization may degrade the performance of the fine-tuned model, while parameter-efficient fine-tuning still needs a full copy of the original model weights to be present in the GPU memory.

Second, conventional FL requires clients to send model updates to the server in each communication round, and given the size of LLMs, sending model updates is bandwidth-intensive and may not be practically feasible. For example, the size of the Llama 2 7B model is 27 GB, which needs to be sent to the server from each client and in each communication round. This is no longer tenable in most production FL settings. Such exorbitant communication costs can be substantially alleviated if LoRA is used in the fine-tuning process, with a corresponding minor degradation in model performance after convergence.

Some may argue that *split learning* [3], [4], where the model is partitioned between a client and the server along a cut layer, may be used so that clients need less computation resources for local training, and model updates do not need to be sent at all between clients and the server. Yet, due to a lack of resources on each client, split learning offloads a majority of the training workload to the server, which needs to be configured with a sufficient amount of GPU memory. As the sizes of LLMs exponentially increase over time [5], it becomes increasingly

The research was supported in part by a RGC RIF grant under the contract R6021-20, RGC CRF grants under the contracts C7004-22G and C1029-22G, and RGC GRF grants under the contracts 16209120, 16200221 and 16207922.

impractical to offload a large portion of an LLM to the server from the perspective of GPU resources.

In this paper, we design and implement TITANIC, a new distributed training paradigm designed specifically for fine-tuning pretrained LLMs while preserving data privacy. TITANIC first optimally selects a subset of client devices to start the fine-tuning process, then partitions an LLM across them before fine-tuning the model with no or minimal losses in the performance of the fine-tuned model. Similar to both conventional FL and split learning, private raw data used for fine-tuning never leaves the client device where they originate from, which maximally preserves privacy. On the one hand, unlike conventional FL, only a small portion of model weights may be sent between clients and the server. On the other hand, unlike split learning, no training is performed on the server, which does not need GPU computation resources at all, and there is no risk for the server to become a bottleneck when the number of clients scales up. At a high level, TITANIC combines the best of both worlds of conventional FL and split learning, and is especially well suited for fine-tuning excessively large pre-trained models such as LLMs.

Highlights of our original contributions in this paper are as follows.

First, starting from a handcrafted implementation of its plumbing, TITANIC is first and foremost designed to be a distributed training paradigm for LLMs. It features a novel *Autograd Bridge*, which is used as a basic building block to partition any LLM in a model-agnostic, fully automated fashion. With a production-quality implementation of the *Autograd Bridge*, a pre-trained model — such as one initialized with HuggingFace's convenient `transformers.AutoModel.from_pretrained()` method — can be fine-tuned without *any* source code modifications, while still allowing the model to be divided into multiple partitions and distributed across their respective client devices.

Second, With TITANIC, an LLM is partitioned across — and fine-tuned over — a chain of client devices in a collaborative fashion. As a training paradigm, however, TITANIC offers some degree of design freedom and flexibility with respect to how the fine-tuning process proceeds. Different designs allow their respective tradeoffs between training speed during the fine-tuning process and the model performance after convergence. At one extreme, TITANIC is able to achieve the same model performance as centralized fine-tuning by trading off some training speed. At the other extreme, it maximizes the degree of parallelism by allowing clients to fine-tune concurrently; and akin to conventional FL and SplitFed learning, it may use aggregation to produced a shared global model with degraded model performance.

Third, Our extensive array of experimental results evaluating TITANIC show clear evidence that it is able to outperform conventional FL and split learning over LLMs with a variety of sizes. As a training paradigm, TITANIC offers sufficient flexibility to achieve various points of tradeoffs between training speed and model performance after convergence; but fundamentally, TITANIC offers a simple, fully distributed mechanism to democratize the fine-tuning process, and trades off training speed to minimize the operating expenses (OpEx) incurred from leasing cloud GPU resources.

Last but not the least, To borrow a page from conventional FL, TITANIC is able to accommodate the ability to select a small number of client devices from a large set of potential candidates. Rather than using random client selection or selecting clients based on their data quality, TITANIC proposes an *optimal* client selection mechanism by maximizing the overall training and network performance, while satisfying inherent constraints over GPU resources. We show that such an integer optimization problem can be directly solved with a linear program solver, as the objective function is convex and the constraint matrix is totally unimodular. At moderate (and practical) scales, exact optimal solutions can be computed in a fraction of a second.

II. DESIGN SPACE AND RELATED WORK

In this paper, our overarching research objective is to fine-tune pre-trained LLMs without violating data privacy. Related work in the literature was designed for much smaller models, and falls into the following categories.

Federated Learning. Starting from its inception, federated learning (FL) [6] was designed to proceed in communication rounds, and to follow several steps in each communication round: the server first selects a number of clients from a larger pool of available candidates, and then each selected client trains a shared global model locally using its private data. The model updates from these clients will then be sent back to the server to perform aggregation, using aggregation algorithms such as Federated Averaging (FedAvg). Convergence is expected after a number of communication rounds.

Such a core mechanism in conventional FL naturally assumes that when a client is selected, it has a sufficient amount of computation resources to complete local training. Unfortunately, this is no longer valid with the advent of LLMs, each containing billions of model parameters. In practice, very few client devices are equipped with the necessary amount of GPU memory or computational power to fine-tune an entire LLM locally. In addition, as model updates need to be sent back to the server, their large sizes — 23.3 GB for the Llama 2 7B model — make it infeasible to use conventional FL for fine-tuning LLMs while preserving data privacy.

Split Learning. With split learning [3], [4], a model is partitioned between each client and the server along a *cut layer*. Outputs at the cut layer are forwarded to the server in the forward pass of training, which then complete the remaining training process without access to private data. When gradients reach the cut layer in backpropagation, they will be sent back to the clients.

Despite the benefits over communication efficiency as split learning avoids sending model updates, it suffers from *three* main disadvantages. *First,* while conventional FL only uses the server for model aggregation, with split learning, the server needs a large amount of computational resources to complete a large portion of the training process. As the number

TABLE I

HOW EXISTING WORK IN THE LITERATURE NAVIGATES THE DESIGN CHOICES IN DISTRIBUTED TRAINING WHILE PRESERVING DATA PRIVACY.

Design Choice	Conventional FL [6]	Split Learning [3], [4]	SplitFed Learning [7]	TITANIC
Model partitions	Entire model on each client	2 partitions	2 partitions	Flexible – from 2 to k partitions
Aggregation	Aggregates entire models on the server	No aggregation	Aggregates client-side models on the server	Flexible – with or without aggregation
Server-side training	No training – aggregation only	Yes – GPU required	Yes – GPU required	No training – aggregation may be used

of clients with local private data scales up, the server may become a bottleneck with respect to computation. *Second,* if we need to produce a shared global model after the fine-tuning process concludes, we need to use *SplitFed learning* [7] to aggregate client-side model partitions using an aggregation algorithm such as Federated Averaging, but any aggregation would lead to a reduced model performance as compared to centralized training, which can be quite substantial. *Finally,* despite its appeal in the research literature, there does not exist an open-source implementation of split learning for production systems, in which model partitioning can be fully automated in a model-agnostic fashion.

Design Space. Our discussions so far have led to the following design choices that we should consider when fine-tuning LLMs with privacy preservation:

▷ *The number of model partitions.* With the exponentially increasing sizes of LLMs [5], simply dividing a large model into a client and a server partition, as split learning proposed, is insufficient. To accommodate large models, TITANIC needs to be designed to support partitioning the model into k partitions, where k is the number of client devices selected in each communication round.

▷ *With or without server aggregation.* While server aggregation is the foundation of conventional FL, the original split learning [3] proposed a variant that does not perform server aggregation. Instead, client-side model snapshots were transmitted across clients between different rounds of training. As a training paradigm for LLMs, TITANIC needs to adopt a more general design and support better flexibility: aggregation increases the degree of parallelism, leading to faster training; but it reduces the performance of the fine-tuned model after convergence.

▷ *Training on the server.* As split learning and SplitFed learning requires the server to be involved in the fine-tuning process, the GPU resources required for such training incurs ongoing operating expenses (OpEx). Due to the formidable sizes of LLMs, such operating expenses can be costly.

We summarize these design choices in Table I, highlighting the tenet in TITANIC's design philosophy: we seek to avoid such OpEx entirely and use the server only for aggregation (if needed), and provide a sufficient amount of flexibility with respect to model partitioning and server aggregation.

Client Selection. There exists a variety of client selection algorithms in conventional FL (*e.g.*, [8]), designed to improve the speed of convergence by selecting from a large pool of clients with not independent and identically distributed (non-i.i.d.) data. In contrast, TITANIC is not concerned about data heterogeneity; instead, as clients need to carry potentially heavy training workloads locally, we need to select clients who have the best possible resources with respect to both computation and network bandwidth.

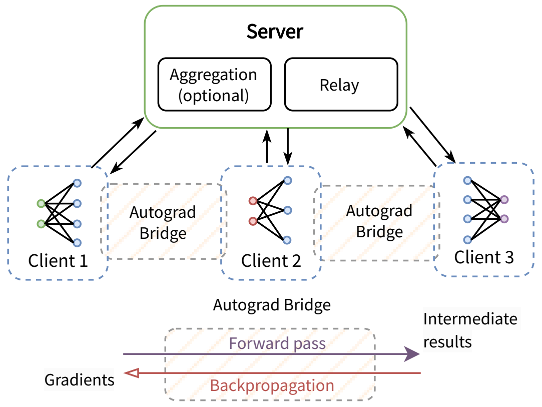

Fig. 1. TITANIC's architectural design at a high level.

III. TITANIC: ARCHITECTURAL DESIGN

A. Design Overview

The overarching design philosophy of TITANIC is not unlike application-layer overlays and (shortly afterwards) peer-to-peer networks, designed over two decades ago [9] and is still in practical use today as the foundation of BitTorrent [10]. In essence, rather than using a server to carry the bulk of the training workload — which is computation-intensive due to the extreme sizes of LLMs — TITANIC represents a new distributed training paradigm that allows client devices to collaborate with one another in a peer-to-peer fashion. Fig. 1 illustrates the overall architectural design in TITANIC.

There is an abundance of related work in the conventional FL literature on peer-to-peer federated learning (*e.g.*, [11], [12]). The main difference between conventional client-server FL and peer-to-peer FL is that model updates are no longer routed via and aggregated on the server; instead, they are exchanged between peers directly. This implies that existing work on peer-to-peer federated learning still assumes that the model can be trained or fine-tuned on each client device in its entirety, which is no longer practically feasible for LLMs.

Rather than training an entire model locally on each client, an LLM is divided into multiple partitions, each partition trained by one of the selected clients. TITANIC features a novel *Autograd Bridge*, which is used as a basic building block to inter-connect collaborating clients during the fine-tuning process. In a nutshell, an *Autograd Bridge* is used for both the forward pass and backpropagation in the training process. It is designed to be general over any neural network model with any computation graph, and to transparently forward any intermediate results computed by an upstream client to its downstream neighbor, as well as any gradients produced by a downstream client back to its upstream neighbor during backpropagation.

B. Decentralized Fine-Tuning

With TITANIC, the simplest way to fine-tune an LLM in a fully decentralized fashion is to first divide it into k multiple partitions, and select an optimal set of k clients to host them. The fine-tuning process proceeds sequentially one after another from client 1 to client k, each uses its local data to fine-tune the model. Upon completing the fine-tuning process, clients can send their trained partitions to the server, to be combined into a shared global model.

Conceptually, network flows through the *Autograd Bridge* can be completed in direct TCP connections between the clients. In reality, however, a server is most likely needed to facilitate such peer-to-peer network connections as two clients need to establish connectivity information by communicating with an ICE (Internet Connectivity Establishment) server, which can be either a STUN (Session Traversal Utilities for NAT) or a TURN (Traversal Using Relay NAT) server, as used by peer-to-peer real-time communication protocols such as WebRTC [13]. For the best compatibility with Internet Service Providers, a *relay server* running behind a reverse web proxy (using web servers such as `nginx`) can be used, which simply relays all traffic between a pair of clients.

C. A Tale of Three Cases

The question that remains at this point is how private data may flow through the client devices, inter-connected by instances of the *Autograd Bridge*. To answer this question, we will introduce a number of cases gradually, by progressing through a succession of increasing generality.

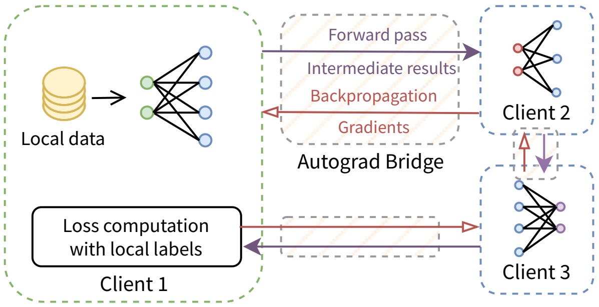

Fig. 2. In the simple case as a starting point, if only one client has private data locally, collaborating downstream clients will be engaged in both the forward and backward passes during the fine-tuning process.

Case 1: Only one client trains with its local data. To begin our introduction, let us consider the simple case where only a single client has local private data to be used in the fine-tuning process. As Fig. 2 illustrates, the client with local data will feed its data into the first partition of the model, and the computed intermediate results will be forwarded through its collaborating downstream clients, and eventually back to itself to complete the forward pass by computing the loss using its labels. Naturally, backpropagation follows the reverse path through the collaborating clients back to the first partition on the same client.

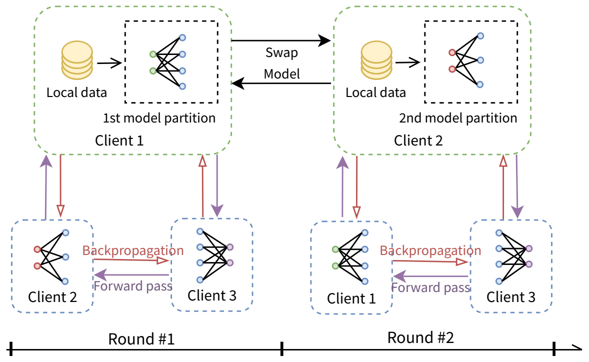

Fig. 3. In the case where multiple clients train with their local data and without aggregation, an exchange of model partitions between clients would be necessary if we need to minimize the use of GPU memory on the clients.

Case 2: Multiple clients train with their local data, without aggregation. In this case, we have multiple clients with local data that needs to be used in the fine-tuning process. Since data should never leave the client to preserve its privacy, if we intend to minimize the use of GPU memory on the clients, we must migrate the necessary model partition to the client where data resides. To facilitate such migration, we may *swap* model partitions between the client with local data to be used in the future and the client with data that has already been used in the past. In our example shown in Fig. 3, after the swap, client 1 will host partition 2, whereas client 2 will host partition 1 instead, which contains all the input stages of the model's computation graph. Just like our simple case with one client, loss values still need to be computed on the client with its local data, requiring a round trip for both forward and backward passes.

In case that the amount of GPU memory on the clients is sufficient to accommodate two partitions, we do not need to consume network bandwidth to swap model partitions. Instead, we may host partition 1 on all the clients with local data, *in addition to* the model partition that was originally assigned to them. This solution, shown in Fig. 4 with four clients, trades off GPU memory usage to conserve network bandwidth and to improve the training speed. If we avoid using aggregation for the sake of model accuracy after convergence, each client can train with its local data *sequentially* in rounds, and partition 1 needs to be migrated to the next client as the current one finishes training. Round trips for both forward and backward passes, just like in our simple case, are still required for loss computation, but omitted in Fig. 4 for clarity.

Case 3: Multiple clients train with their local data, with

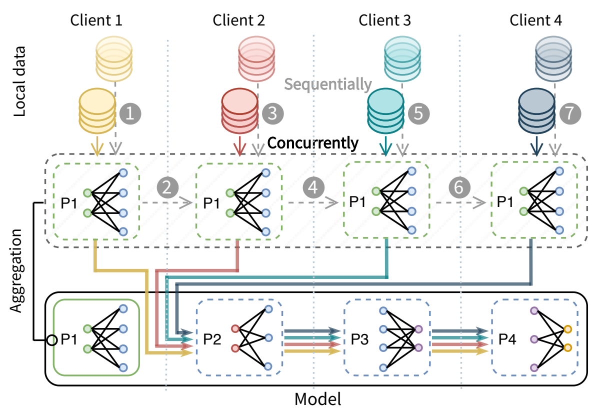

Fig. 4. If the amount of GPU memory is sufficient to accommodate two partitions, we do not need to swap model partitions between rounds. Instead, we assign partition 1, containing all the input stages, to all the clients so that they can train with their local data, either concurrently or sequentially. No aggregation is necessary if we train sequentially, but we need to migrate partition 1 across clients. Training concurrently improves the training speed, but aggregation degrades model accuracy after convergence.

aggregation. As shown in our example in Fig. 4, if we allow the server to perform aggregation, the clients no longer need to start training sequentially one after another. Instead, their fine-tuning processes can proceed concurrently. In this case, weights in partition 1 will vary across different clients after fine-tuning and need to be aggregated, as they are trained using different local data. In contrast, downstream partitions in the model are shared, and do not need to be aggregated. This case is similar to SplitFed learning [7] (without the need for a GPU server to perform training), in which client-side models need to be aggregated while the server's model is shared.

D. Communication Costs: Quantitative Analysis

In essence, in the unfortunate situation that an LLM does not fit into the GPU memory of one client device and must be partitioned, TITANIC's design achieves various points of tradeoffs between training speed (with more concurrency and server aggregation) and model accuracy (with no concurrency and no aggregation either). But how does its communication costs compare with conventional FL, measured in bytes of data to be transferred?

The communication efficiency in conventional FL has been extensively studied in the literature (*e.g.*, [14]). In essence, model updates of a particular size (which can be smaller if quantization or pruning is used) were transmitted to the server by each client in every communication round. Consider c clients are selected in each round and a model size of s_m bytes, if the fine-tuning process takes r rounds in total to converge and e_r epochs take place in local training in each round, we have a total of $s_{\mathrm{FL}} = 2 \cdot s_m \cdot c \cdot r \cdot e_r$ bytes of communication costs that need to be transmitted to and from the server.

With TITANIC, we no longer need to send model updates to the server. Instead, intermediate outputs and gradients are transmitted across client devices in a peer-to-peer fashion. Assuming that the size of intermediate outputs and gradients from one client to its downstream neighbor is s_n, in each iteration, $s_i = s_n \cdot c \cdot 2$ bytes are transmitted sequentially, the forward and backward passes combined. If a dataset used for fine-tuning has b batches, we need a total communication cost of $s_{\mathrm{Titanic}} = s_i \cdot b \cdot r \cdot e_r$ to converge.

Naturally, whether TITANIC compares favorably with conventional FL in terms of total communication costs depends on the practical values of $s_n \cdot b$ (for TITANIC) vs. s_m (for FL). In the case of large language models, s_m is exceedingly large: the smallest 7B Llama 2 model weighs in at 27 GB. In contrast, at a batch size of 4, the payload size for intermediate output for the same Llama 2 model is around 6.25 MB, which is approximately $4400\times$ smaller. In this case, as long as we have fewer than 4400 batches in our fine-tuning dataset — which is most likely the case as fine-tuning uses only a small number of data samples — TITANIC incurs a lower communication cost than conventional FL. In cases where parameter-efficient fine-tuning techniques such as LoRA [2] is used, s_m is much smaller, weighing in at 20 MB. In this case, if we have more than 3 batches in our dataset, TITANIC underperforms compared to LoRA-based FL.

It goes without saying that the training speed is also affected by the network bandwidth, in addition to the communication cost. In conventional FL, the server is the bandwidth bottleneck, so concurrent connections from all selected clients to the server is equivalent to transmitting from each client sequentially. If we wish to compare the training speed of FL vs. TITANIC, we need to consider how bandwidth at each client compares with the server bandwidth. In general, server bandwidth is an order of magnitude higher than client bandwidth due to the differences in access networks. In conclusion, how TITANIC compares with FL in terms of communication costs and the training speed depends heavily on parameter settings and bandwidth availability; but for large language models, due to the lack of GPU memory on the clients, we may not have a choice.

IV. AUTOGRAD BRIDGE: DESIGN AND IMPLEMENTATION

The foundation of TITANIC is its basic building block: the *Autograd Bridge*. It provides the necessary mechanism that inter-connects different client devices and supports both the forward pass and backpropagation. Conceptually its design is quite simple; but implementing it in a fully automated fashion without any changes to existing LLMs is a daunting challenge.

At first glance, the peer-to-peer communication pattern in TITANIC's design appears to be similar to model parallelism [15] and pipeline parallelism [16], [17] in distributed machine learning. Both model and pipeline parallelism allow a large model to be divided into smaller partitions, so that they can be trained over multiple GPUs. Data communication occurs between GPUs, potentially spanning multiple physical machines, in a peer-to-peer fashion.

However, there are two important differences between TITANIC and model or pipeline parallelism. *First,* as a fully decentralized training paradigm and similar to federated learning, TITANIC is designed to operate over the Internet using standard transport protocols such as TCP, and across a wide variety of client devices with limited computation resources. This is in sharp contrast to the use of proprietary protocols, such as NCCL [18], in modern frameworks (such as PyTorch) to support model and pipeline parallelism. *Second,* private data is produced at each client device in TITANIC, and should never leave the device where data resides. This is not a consideration when model and pipeline parallelism are used to train large models in a GPU cluster.

Despite these differences, we can start our investigation from a quick look at how model parallelism is implemented. Consider the following toy model that contains two linear layers. To run this model on two GPUs, we need to modify the model implementation, assign each linear layer to a different GPU, and move inputs and intermediate outputs to match the layer devices accordingly.

```
class ToyModel(nn.Module):
def __init__(self):
  super().__init__()
  self.net1 = torch.nn.Linear(10, 10).to('cuda:0')
  self.relu = torch.nn.ReLU()
  self.net2 = torch.nn.Linear(10, 5).to('cuda:1')

def forward(self, x):
  x = self.relu(self.net1(x.to('cuda:0')))
  return self.net2(x.to('cuda:1'))
```

In this example, we need to manually divide the model into `net1` and `net2`, and rewrite the `forward()` method. Needless to say, supporting pipeline parallelism needs more manual modifications to the implementation of the model; and if we need to support multiple nodes, a PyTorch Pipe API is needed using `torch.distributed.pipeline.sync.Pipe`.

When designing TITANIC, such manual modifications to LLMs would become infeasible given their complexity. Even if we assume such modifications can be completed in one particular LLM, doing the same for all the LLMs will quickly become tedious and time-consuming. As new LLMs are published in repositories such as the HuggingFace LLM Leaderboard [19] on a daily basis, manual modifications at the source code level is out of the question. Interestingly, all existing open-source implementations of split learning required such manual modifications to the models.

Automating model partitioning. Towards designing and implementing the *Autograd Bridge*, we seek to provide a flexible, off-the-shelf solution for offloading any desired portion of a computation graph in a model for remote execution.

Formally, a computation graph on a client c_i can be defined as $G = (V, E)$, where V and E represent the sets of nodes and edges, respectively. A node $v \in V$ represents either a constant value or an operator. An edge $(v_s, v_t) \in E$ indicates that the output of node v_s will be passed as input to v_t. From the perspective of one client, a subgraph $G_R = (V_R, E_R)$ needs to be extracted from G, where $V_R \subset V$ and $E_R \subset E$, and then offloaded to a downstream client c_{i+1} for remote computation. The remaining local computation graph is denoted as $G_L = (V_L, E_L)$, where $V_L = V - V_R$ and $E_L = E - E_R$. The incoming and outgoing edges of the remote graph G_R that cross the boundary between local and remote graphs are denoted as E_I and E_O, respectively, defined as follows:

$$E_I = \{(v_l, v_r) \in E | v_l \in V_l, v_r \in V_R\}$$
$$E_O = \{(v_r, v_l) \in E | v_l \in V_l, v_r \in V_R\}$$

Referring back to TITANIC's design overview, the data flowing through the edges in E_I represents the *intermediate results* during the forward pass in Figs. 1 and 2. Conversely, the data from E_O represents the *gradients* computed during backpropagation.

The *Autograd Bridge* aims to reconnect the subgraphs G_L and G_R, transforming the local computation graph G_L into a complete one. To do so, it can be conceptually represented by a special node b that takes control of all incoming and outgoing edges to the remote graph:

$$E_b = \{(v_s, b) | (v_s, \_) \in E_I\} \cup \{(b, v_t) | (\_, v_t) \in E_O\}.$$

The data exchange between G_L and G_R is defined as an internal operation within the node b. This allows the local computation graph to be transformed into $G_L = (V_L \cup \{b\}, E_L \cup E_b)$. The presence of the *Autograd Bridge* b reestablishes the connection within G_L, resulting in a complete computation graph. It conceals the fact that the subgraph G_R was offloaded to client c_i's downstream neighbor c_{i+1} for remote execution, allowing the underlying framework — TITANIC uses PyTorch — to train the local model as if it were a complete computation graph.

During the fine-tuning process, an upstream client with its local computation graph G_L processes the input data until the *Autograd Bridge* is reached. At this point, all the intermediate results, including the input tensors E_I directly involved in the computation, as well as any auxiliary control information (*e.g.*, attention masks in language models), are automatically sent to the downstream client over the network (either using a direct connection or via a relay server), which hosts the computation graph G_R. The *Autograd Bridge* at c_i then waits for results from its downstream neighbor c_{i+1}. Once the gradients E_O have been received, local computation in G_L on c_i can proceed to completion. Similarly, during backpropagation, when the gradients E_O have been back-propagated to the *Autograd Bridge* in c_{i+1}, these gradients are forwarded to its upstream neighbor c_i, allowing backpropagation to seamlessly continue on the other side of the "bridge."

There are two observations about such a design worth noting. *First*, the design of the *Autograd Bridge* is sufficiently general to divide the computation graph of any model into *multiple* partitions, not just into two partitions G_L and G_R. The local computation graph G_L on a client c_i, for example, can (recursively) be a portion of G_R from the viewpoint of c_i's own upstream neighbor, c_{i-1}. *Second*, such an information exchange between clients, involving both the intermediate results and gradients, occurs solely within the computation graph. PyTorch is completely oblivious of the network connections within the *Autograd Bridge*, and there will be no manual intervention or source code modifications at all.

Implementation challenges. With our design of the *Autograd Bridge*, the primary challenge in its PyTorch implementation is how the `backward()` function can be overridden to receive gradients from G_R on c_{i+1} to G_L on c_i, without any manual modifications to the model. To achieve this goal, we introduce a `ClientForwardFunction` class that inherits from PyTorch's `autograd.Function`:

```
class ClientForwardFunction(torch.autograd.Function):
@staticmethod
def forward(ctx, comm, desired_output, *grad_tensor_list):
  ctx.comm = comm # saving the communication backend
  return desired_output

@staticmethod
def backward(ctx, grad_output):
  comm = ctx.comm # restoring the communication backend
  # sending to the upstream client
  comm.send(grad_output, fwd_flag=False)
  # receiving from the downstream client
  gradients = comm.recv()
  return None, None, *gradients
```

This class saves and restores the communication backend in the *Autograd Bridge* in its context. The communication backend in the *Autograd Bridge* provides a simple API consisting of only two functions, `send()` and `recv()`. For the best compatibility, we implemented the communication backend by using a relay server using the WebSockets protocol over HTTPS, an industry standard for two-way communication.

Once defined, the `ClientForwardFunction` class can then be applied in the `forward()` function of our customized `ClientModule` class:

```
class ClientModule(nn.Module):
def forward(self, *args, **kwargs):
  ...
  # sending to and receiving from the downstream client
  self.comm.send(input_data, fwd_flag=True)
  output_data = self.comm.recv()
  desired_output = output_data

  output_data = ClientForwardFunction.apply(
    self.comm, desired_output, *grad_tensor_list
  )
  return output_data
```

What we have shown represents just a glimpse of the implementation detail that made the *Autograd Bridge* feasible without *any* source code modifications to both models and their trainers. Since pre-trained LLMs are typically loaded from HuggingFace's model repositories using APIs such as `transformers.AutoModel.from_pretrained()`, it is critically important to avoid any manual intervention if TITANIC is to be used in practice.

V. OPTIMIZING CLIENT SELECTION

In conventional FL, it is routine to select a small number of participating clients from a larger pool of candidates, and existing client selection algorithms in the literature were designed to mitigate data heterogeneity across clients and to improve the speed of convergence. Though we may continue to use these client selection algorithms in TITANIC, we shall propose an orthogonal way of selecting clients with the same objective of optimizing the training speed.

Recall that in Section III-D, we mentioned that the server bandwidth may be an order of magnitude higher than client bandwidth due to differences in network types. Following the same logic, available bandwidth across clients can be one or two orders of magnitude different as well. Our new client selection algorithm takes such *resource heterogeneity* into account, and seeks to optimize the training speed by choosing *faster* clients with respect to both their available bandwidth and computing resources, such as the amount of available GPU memory. The general rule of thumb is to choose the fastest possible clients in terms of bandwidth, yet with a sufficient amount of GPU memory.

Rather than resorting to heuristics, we take a more disciplined approach and propose to formulate and solve an integer optimization problem to find the *optimal* assignment of model partitions to clients, which offers better performance than assigning randomly or using simple heuristics. If we can solve such an optimization problem efficiently, model partitions will then be distributed more effectively and reasonably to each client, with the hope of maximizing the training speed.

After dividing a large language model into partitions, let us consider the problem of assigning model partitions $\{P_k\}_{k\in\mathcal{K}}$ to clients $\{C_n\}_{n\in\mathcal{N}}$, where $\mathcal{K} = \{1, 2, \ldots, K\}$ and $\mathcal{N} = \{1, 2, \ldots, N\}$ are the corresponding indexing sets, and K and N correspond to the total number of model partitions and clients, respectively. We define the *training time* $T_n = |w_{P_k}|/S_n$, where $|w_{P_k}|$ represents the number of floating point operations in the model partition P_k, and S_n denotes the computing performance on client n, in floating point operations per second (FLOPS). The training time T_n^k, therefore, corresponds to the wall-clock time elapsed during the local fine-tuning process. A client with more computational resources will have a small T_n^k. Similarly, we define the *transmission duration*, $D_n^k = s_{P_k}/B_n$, where s_{P_k} represents the size of the model partition P_k in bytes, and B_n denotes the bandwidth available to client n, in bytes/second.

To numerically represent the distance between the partition P_k and client C_n, we define the *match distance* as $d_{kn} = a \cdot D_n^k + b \cdot T_n^k$, which is a weighted sum involving both the transmission duration and training time, scaled by hyper-parameters a and b. We then define the *affinity* between a model partition and a client $A_n^k = 1/d_n^k$, and seek to maximize the total of all affinities. In other words, we prefer to assign model partitions to faster clients with shorter transmission durations and training times.

Such a definition of affinity allows us to tell whether a model partition is suitable for a client or not. For an arbitrary model partition P_k, $A_1^k \geq A_2^k$ means that client C_1 is a better match for the partition than client C_2 is, and we should assign it to C_1. At first glance, one may think that it suffices to choose the client that results in the lowest match distance for each partition. However, our problem is more subtle because clients may have GPU memory constraints that block certain assignments.

Formulating the optimization problem. We represent a partition-client assignment using a vector $\alpha \in \{0, 1\}^{KN}$, $\alpha =$

$(\alpha_1^1, \ldots, \alpha_N^K)$, where $\alpha_n^k = 1$ if partition P_k is assigned to client C_n, and 0 otherwise, and formulate our client selection problem as the following:

$$\max_{\alpha_n^k} \sum_k \sum_n A_n^k \cdot \alpha_n^k \tag{1}$$

$$\text{s.t.} \sum_{n=1}^{\mathcal{N}} \alpha_n^k = 1, \ \forall k \in \mathcal{K} \tag{2}$$

$$\sum_{k=1}^{K} \alpha_n^k \leq 1, \ \forall n \in \mathcal{N} \tag{3}$$

$$\alpha_n^k = 0, \text{ if } M_n < |w_{P_k}| \tag{4}$$

$$\alpha_n^k \in \{0, 1\}, \ \forall k \in \mathcal{K}, \ \forall n \in \mathcal{N} \tag{5}$$

where constraint (2) implies that each partition can only be assigned to one client, constraint (3) shows that each client cannot host more than one partition, and constraints (4) indicates that if operations in a partition k does not fit into the available GPU memory M_n at client n, the assignment should be avoided.

Transforming into an LP problem. Our integer optimization problem is essentially the same formulation as in Taylor's work [20] with minor variations, such as the addition of constraints that block assignments due to insufficient GPU memory. Even though integer linear optimization problems can be computationally difficult to solve, as Taylor pointed out [20], the constraint matrix in this formulation is *totally unimodular*, which implies that we can treat it as a linear programming problem, and use a standard LP solver to solve it. The LP relaxation — allowing $\alpha_n^k \in [0, 1]$ — does not affect the integrality of the optimal solution, thanks to Meyer's seminal work [21]. Solving such a problem with an LP solver at a small scale is surprisingly efficient: for example, the `Mosek` solver took less than **40** milliseconds for 100 clients and 32 partitions. As a result of such efficiency, solving our proposed optimization problem is a practical and a more disciplined approach than designing heuristics.

Evaluation. In order to evaluate our proposed optimal client selection algorithm by directly solving the optimization problem we just formulated, we randomly generate 100 clients, each with its amount of available GPU memory drawn from $[4, 8, 12, 16, 20, 24]$ using a normal distribution (with a mean of 12 and a standard deviation of 6. The available bandwidth of our clients is drawn from $[0 - 100]$ (Mbps) using a normal distribution (with a mean of 50 and a standard deviation of 25), and their computing performance is similarly drawn using a normal distribution from $[5 - 80]$ TFLOPS — ranging from an NVIDIA RTX 2050 mobile to an RTX 4090 GPU — with a mean of 40 and a standard deviation of 20. We assign a small number of model partitions, each corresponding to one or multiple layers in a Llama 2 model with 7, 13 and 70 billion parameters.

We evaluate our optimal algorithm by comparing with two simple heuristics: (1) a *greedy* algorithm that first assigns the largest partition to the client with the lowest transmission duration, and then assign to the second largest partition to the client with the second lowest transmission duration, and so on. (2) a *random* algorithm, which randomizes the assignment of each partition to clients. Clients with insufficient GPU memory will be excluded in both heuristics. From our results shown in Fig. 5, we can conclude that though the greedy and random heuristics performed reasonably well, the solution to our optimization problem performed substantially better. Given that running the LP solver takes less than 40 milliseconds on a modern server for our largest model, our optimal selection algorithm is a lightweight and practical alternative to existing client selection algorithms in federated learning.

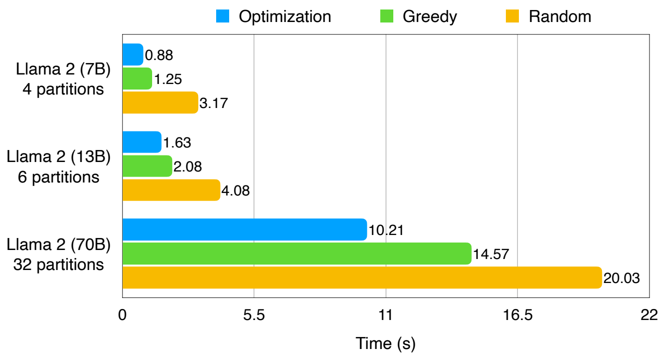

Fig. 5. Evaluating our optimal client selection algorithm against greedy and random assignment heuristics.

VI. Performance Evaluation

As a new distributed training paradigm for large language models, evaluating Titanic is somewhat unconventional: since it is not necessarily a specific algorithm or mechanism, there is no prior work in the literature to compare with. Nevertheless, we attempt to showcase the practicality and performance of our implementation of Titanic in this section, over a series of experiments involving three large language models: `OPT-1.3B` [22], `Bloom-3B` [23], as well as the cutting-edge `Llama 2-7B` [1]. We fine-tune these models on the `wikitext-2-raw-v1` dataset [24] containing 36,700 text samples, and observe the training loss as an indicator of the fine-tuning progress. Our real-world implementation of Titanic uses PyTorch 2.0.1 and Python 3.10. All peer-to-peer communication across clients is relayed through a WebSockets `node.js` webserver.

For `OPT-1.3B` and `Bloom-3B` models, our fine-tuning process with Titanic was conducted on a cloud server with one NVIDIA RTX A6000 GPU (with 48 GB of CUDA memory and CUDA version 12.2). In contrast, as 48 GB of CUDA memory is not sufficient for fine-tuning the `Llama 2-7B` model, we conducted our experiments using a MacBook Pro computer with an Apple M2 Max CPU, with 96 GB of unified GPU memory. When fine-tuning, we chose batch sizes of 32, 16 and 4 for `OPT-1.3B`, `Bloom-3B`, and `Llama 2-7B`, respectively. LoRA [2] has been used to allow parameter-efficient fine-tuning in all our experiments.

Quality of the fine-tuned models. The most important aspect of the fine-tuning process is the quality of the fine-

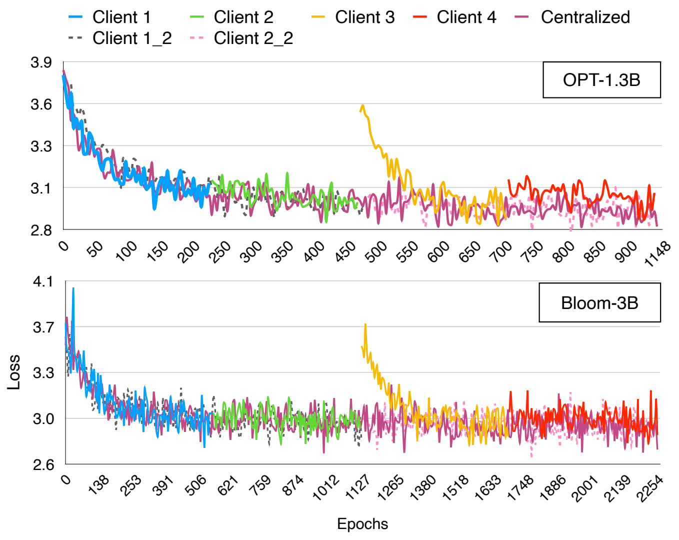

Fig. 6. The convergence behavior of centralized fine-tuning and TITANIC (with 4 and 2 clients): (top) Fine-tuning `OPT-1.3B`; (bottom) fine-tuning with `Bloom-3B`.

TABLE II
FINE-TUNING LLMS USING CENTRALIZED TRAINING, CONVENTIONAL FL WITH FEDERATED AVERAGING, AND TITANIC.

Model	Methods	Perplexity
`OPT-1.3B`	Centralized	**18.02**
	FL with FedAvg	**34.01**
	TITANIC with 4 clients $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$	20.67 $\rightarrow$ 19.34 $\rightarrow$ 19.38 $\rightarrow$ **19.17**
	TITANIC with 2 clients $C_1 \rightarrow C_2$	18.77 $\rightarrow$ **17.23**
`Bloom-3B`	Centralized	**18.04**
	FL with FedAvg	**30.24**
	TITANIC with 4 clients $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$	25.16 $\rightarrow$ 21.49 $\rightarrow$ 19.34 $\rightarrow$ **20.58**
	TITANIC with 2 clients $C_1 \rightarrow C_2$	19.68 $\rightarrow$ **19.20**
`Llama 2-7B`	Centralized	**2.54**
	FL with FedAvg	**3.56**
	TITANIC with 4 clients $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4$	2.90 $\rightarrow$ 2.80 $\rightarrow$ 2.93 $\rightarrow$ **3.34**
	TITANIC with 2 clients $C_1 \rightarrow C_2$	2.30 $\rightarrow$ **2.85**

tuned models. In Table II, we present a detailed comparison study of the model perplexity between centralized training, conventional FL, and TITANIC (with two and four clients). Off the bat, a surprising yet important observation is that conventional FL with Federated Averaging (FedAvg) failed to work effectively, producing models of much worse quality (the lower the perplexity, the better). FL with FedAvg produced models that were even worse than using $1/4$ of the data in TITANIC with C_1 only. We believe that this is attributed to the use of LoRA for parameter-efficient fine-tuning in our experiments: using FedAvg to compute the average of local LoRA adapters will produce a shared adapter that performed much worse. In contrast, TITANIC was able to obtain models that were comparable in quality to centralized fine-tuning; and unsurprisingly, using two clients — each with half of the data — performed better than using four clients, each with $1/4$ of the data.

Convergence behavior. Beyond the quality of the fine-tuned models, we are also interested in the convergence behavior at run-time. In Fig. 6, we compare TITANIC (again with 2 and 4 clients) with centralized training using the `OPT-1.3B` and `Bloom-3B` models, respectively. As we expected, using 4 clients with TITANIC performed slightly worse than using 2 clients, but both were similar to centralized fine-tuning in terms of their convergence behavior over multiple epochs.

Last but not the least, we fine-tuned the much larger `Llama 2-7B` model to see if the convergence behavior may be different from its smaller counterpart. Our results, illustrated in Fig. 7, showed that due to the size of this model, training loss rapidly converged to a very low value, which implied that this model only needed a small number of data samples to fine-tune successfully. Data from additional clients were not really necessary. That said, this experiment showed that TITANIC, with its design of the *Autograd Bridge*, was able to seamlessly partition large language models of any size without issues.

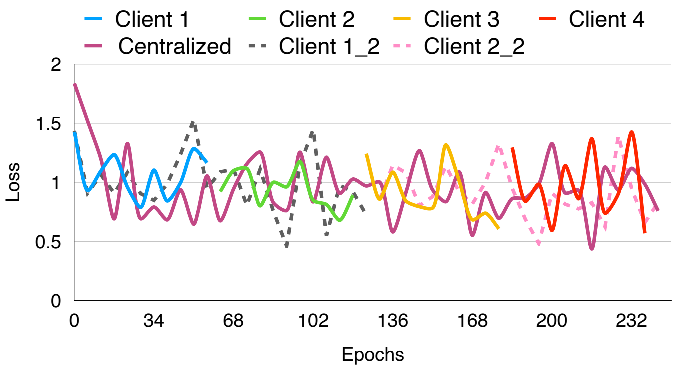

Fig. 7. The convergence behavior of centralized fine-tuning and TITANIC (with 4 and 2 clients): fine-tuning the `Llama 2-7B` model.

VII. CONCLUDING REMARKS

Due to the enormous sizes of pretrained large language models, it is an open problem and a formidable challenge to fine-tune them collaboratively over multiple users, while still preserving data privacy. Either conventional FL nor split learning may be viable options in practice, as both the clients and the server may not have sufficient amount of GPU resources.

In this paper, we present TITANIC, a third alternative that allows clients to collaboratively fine-tune a shared LLM without the need of using GPU servers. The cornerstone of TITANIC is the *Autograd Bridge*, which we implemented with a focus on practicality, so that it works effectively with any LLM, and without any source code modifications to the model and its trainer. TITANIC is a distributed training paradigm that can flexibly accommodate different communication patterns, with or without server aggregation. We also proposed a client selection algorithm that optimizes the assignment of model partitions to clients without adding any complexity. Our performance evaluations showed that TITANIC offers comparable performance to centralized training.

References

[1] H. Touvron, T. Scialom, L. Martin, K. Stone, and *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," *arXiv preprint arXiv:2307.09288*, 2023.

[2] E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," in *Proc. International Conference on Learning Representations (ICLR)*, 2022.

[3] O. Gupta and R. Raskar, "Distributed Learning of Deep Neural Network over Multiple Agents," *Journal of Network and Computer Applications*, vol. 116, pp. 1–8, 2018.

[4] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split Learning for Health: Distributed Deep Learning without Sharing Raw Patient Data," in *Proc. the ICLR AI for Social Good Workshop*, 2019.

[5] J. Simon. (2021) Large Language Models: A New Moore's Law? [Online]. Available: https://huggingface.co/blog/large-language-models

[6] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-Efficient Learning of Deep Networks from Decentralized Data," in *Proc. 20th Int'l Conference on Artificial Intelligence and Statistics (AISTATS)*, April 2017, pp. 1273–1282.

[7] C. Thapa, P. C. M. Arachchige, S. Camtepe, and L. Sun, "SplitFed: When Federated Learning Meets Split Learning," in *Proc. AAAI Conference on Artificial Intelligence*, vol. 36, no. 8, 2022, pp. 8485–8493.

[8] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Oort: Efficient Federated Learning via Guided Participant Selection," in *Proc. 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021, pp. 19–35.

[9] Y. hua Chu, S. G. Rao, , and H. Zhang, "A Case for End System Multicast," *ACM SIGMETRICS Performance Evaluation Review*, vol. 28, no. 1, pp. 1–12, June 2000.

[10] B. Cohen, "Incentives Build Robustness in BitTorrent," in *Proc. Workshop on Economics of Peer-to-Peer Systems*, vol. 6, 2003, pp. 68–72.

[11] A. Lalitha, O. C. Kilinc, T. Javidi, and F. Koushanfar, "Peer-to-Peer Federated Learning on Graphs," *arXiv preprint arXiv:1901.11173*, 2019.

[12] Z. Li, J. Lu, S. Luo, D. Zhu, Y. Shao, Y. Li, Z. Zhang, Y. Wang, and C. Wu, "Towards Effective Clustered Federated Learning: A Peer-to-peer Framework with Adaptive Neighbor Matching," *IEEE Transactions on Big Data*, 2022.

[13] C. Jennings, F. Castelli, H. Boström, J.-I. Bruaroey, A. Bergkvist, D. C. Burnett, A. Narayanan, B. Aboba, and T. Brandstetter. (2023) WebRTC: Real-Time Communication in Browsers. [Online]. Available: https://www.w3.org/TR/webrtc/

[14] O. Shahid, S. Pouriyeh, R. M. Parizi, Q. Z. Sheng, G. Srivastava, and L. Zhao, "Communication Efficiency in Federated Learning: Achievements and Challenges," *arXiv preprint arXiv:2107.10996*, 2021.

[15] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2020.

[16] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism," in *Proc. 33rd International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[17] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *Proc. 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[18] NVIDIA. (2023) NVIDIA Collective Communications Library (NCCL). [Online]. Available: https://developer.nvidia.com/nccl

[19] HuggingFace. (2023) Open LLM Leaderboard. [Online]. Available: https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard

[20] C. J. Taylor, "On the Optimal Assignment of Conference Papers to Reviewers," *Dept. Computer and Info. Sci., Univ. of PA, tech. rep. MS-CIS-08-30*, January 2008. [Online]. Available: http://repository.upenn.edu/cis_reports/889

[21] R. R. Meyer, "A Class of Nonlinear Integer Programs Solvable by a Single Linear Program," *SIAM Journal on Control and Optimization*, vol. 15, no. 6, pp. 935–946, 1977.

[22] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "OPT: Open Pre-trained Transformer Language Models," *arXiv preprint arXiv:2205.01068*, 2022.

[23] T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé *et al.*, "Bloom: A 176B-Parameter Open-Access Multilingual Language Model," *arXiv preprint arXiv:2211.05100*, 2022.

[24] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer Sentinel Mixture Models," *arXiv preprint arXiv:1609.07843*, 2016.