Go With The Flow: Churn-Tolerant Decentralized Training of Large Language Models

Nikolay Blagoev*†, Bart Cox‡, Jérémie Decouchant‡, Lydia Y. Chen†
*Worker Thread, †Université de Neuchâtel, ‡Delft University of Technology
nickyblagoev@gmail.com, b.a.cox@tudelft.nl, j.decouchant@tudelft.nl, yiyu.chen@unine.ch

***Abstract*—Motivated by the emergence of large language models (LLMs) and the importance of democratizing their training, we propose GWTF, the first fully decentralized churn-tolerant practical training framework for LLMs. Differently from existing distributed and federated training frameworks, GWTF enables the efficient collaborative training of a LLM on heterogeneous clients that volunteer their resources. In addition, GWTF addresses node churn, i.e., clients joining or leaving the system at any time, and network instabilities, i.e., network links becoming unstable or unreliable. The core of GWTF is a novel decentralized flow algorithm that finds the most effective routing that maximizes the number of microbatches trained with the lowest possible delay. We extensively evaluate GWTF on GPT-like and LLaMa-like models and compare it against the prior art. Our results indicate that GWTF reduces the training time by up to 45% in realistic and challenging scenarios that involve heterogeneous client nodes distributed over 10 different geographic locations with a high node churn rate.**

***Index Terms*—Large Language Model, Decentralized learning, Crash-tolerance, Flow optimization.**

I. Introduction

Deep transformer-based architectures have recently enabled unprecedented performance on tasks such as next word prediction [1] and image recognition [2] thanks to the growing corpora of available data and the increasing size of Large Language Models (LLMs) [3, 4]. However, models are now too large to fit and be efficiently trained on a single GPU. Parallel training techniques, such as Pipeline Parallelism (PP) and Data Parallelism (DP), therefore need to be used to efficiently train large models on clusters of nodes. However, renting cloud resources or using private computer clusters to train models can easily cost more than tens of thousands of dollars [4], even for smaller models. Developing and training LLMs this way is therefore out of the reach of the public.

To democratize access to language models, a promising alternative to using private clusters is volunteer computing [5], i.e., using spare computing resources from independent participants around the world. The potential of volunteer computing for the development of large deep networks has been acknowledged in recent publications [4, 6]. To realize this potential, the decentralized training of language models requires facing fault-tolerance and performance challenges. First, recovery mechanisms are necessary to tolerate participants freely joining or leaving the system at any time, which also implies that participants can only have partial knowledge of the global system membership. Second, reactive mechanisms are required to adapt to networks becoming unstable or unreliable, and to participants dedicating fluctuating computing resources over time.

Recently, SWARM, a pioneering work by Ryabinin et al. [6], has taken a step towards addressing these issues. However, SWARM fails to fully utilize its resources. First, it fails to properly recover from crashes in the backwards pass. Second, it uses a simple greedy approach during routing and therefore does not aim at optimizing training time. Finally, SWARM assumes that all nodes have the same amount of memory, which in a highly decentralized setting is an unrealistic expectation to hold. Our work addresses these limitations.

We present Go With The Flow (GWTF), a decentralized and efficient LLM training framework. We model the routing of microbatches between heterogeneous nodes in the forward and backward pass of stochastic gradient descent as a minimum cost flow problem and optimize its execution on geographically dispersed nodes in a decentralized manner. GWTF minimizes the training time and maximizes its throughput by continuously addressing the bottleneck stages of LLM training. We evaluate GWTF on the training of GPT-like and LLama-like models of 300 M and 7 B parameters against SWARM and show an up to 40% training time reduction while maximally utilizing the available clients.

This paper makes the following **contributions**:

- We propose and implement GWTF, an efficient churn-tolerant decentralized training framework for LLMs.
- GWTF utilizes a novel fully decentralized minimum cost flow algorithm, empowering individuals to contribute their heterogeneous and volatile resources to LLM training.
- GWTF handles efficiently node churn of LLM during both the forward and the backward passes of stochastic gradient descent, by instantaneously rerouting to available nodes thanks to its decentralized nature.
- GWTF minimizes the training time and maximizes the throughput of training LLMs in heterogeneous crash-prone environments, but also optimally utilizes available resources, without sacrificing convergence.

This paper is organized as follows. Section II provides necessary background knowledge and discusses the related work. Section III states our system model and our objectives. Section IV presents an overview of GWTF, and Section V details the main algorithms of GWTF that optimize flows in a decentralized manner, tolerate crashes and synchronize training and aggregations. Section VI presents our performance evaluation. Finally, Section VIII concludes this paper.

2025 IEEE International Conference on Big Data (BigData) | 979-8-3315-9447-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/BIGDATA66926.2025.11402098

TABLE I: Notations

Name	Description
S	A single stage - a set of nodes
N	All nodes/peers
T	Temperature
α	Cooling factor
$d(i,j)$	Cost between nodes i and j
$f(i,j)$	Flow between nodes i and j
$cost_f$	Cost of a single flow f
cap_i	Memory capacity of node i
$U(x,y)$	Uniform random number between x and y
c_i	Computation time/cost of node i
$\lambda_{i,j}$	Network latency between nodes i and j
$\beta_{i,j}$	Network bandwidth between nodes i and j

II. Background & Related Work

Large Language Models. Large Language Models are typically constructed as an embedding layer and a series of transformer blocks [1, 3, 7, 8]. Embedding layers simply translate the input which is in some high dimension, to a lower dimensional output, which makes learning easier for the neural network. A transformer consists of multi-head self-attention, normalization, and a feed forward network [9].

Pipeline Parallelism. Consecutive layers (i.e., transformer blocks) of an LLM are grouped into stages that are distributed over several devices, forming a pipeline. The first stage retrieves the (tokenized) data and embeds it, while the last stage evaluates the loss function. The first and last stages of the pipeline are colocated on the same device (the data node) because they require the training data. All stages hold one or several contiguous transformer blocks. Together, they behave as a single model. To train a model three computation phases are executed: (i) the forward pass (from the first stage to the last stage) to compute the loss; (ii) the backward pass (from the last stage back to the first stage) to compute the gradients per parameter; and (iii) the update phase, where model weights are updated based on the gradients.

Microbatches. Each batch is further split into microbatches [10] so that multiple microbatches can be concurrently processed on different devices. Following the processing of all the microbatches of a batch, nodes need to update their model parameters based on the average of the microbatches' gradient updates they have stored until this point. An iteration is thus defined as the processing of one batch, and its duration is defined as the elapsed time between the end of two consecutive update phases (measured from the viewpoint of the slowest data node).

Data Parallelism. A larger batch size allows SGD to converge faster [11]. However, in practice, the amount of memory available on devices limits the maximum batch size that can be used. Data parallelism is another way to circumvent this limitation. Data parallelism consists in distributing the processing of microbatches for a given stage over several devices that each host an identical model, which accelerates convergence [11]. These devices concurrently perform iterations on their microbatches. At the end of each iteration, and before the update phase, these devices aggregate their collective work by exchanging their stored gradients (aggregation phase). Later on, they perform the update phase based on the average of all gradients.

Decentralized Pipeline Parallelism. Few works considered the decentralized training of LLMs. [4] compute a communication-optimal arrangement of nodes in pipelines via a centralized and computationally-expensive genetic algorithm, but ignore churn. In SWARM [6], nodes independently route a micro-batch through stages. A node sends its activations (outputs of the last layer of its transformer model) to a next stage's node. If a node fails to process a microbatch before some predefined time, the previous node sends its activations to a different node. However, SWARM does not address crashes that may occur during the backward pass. SWARM's timeout-resend strategy requires a complete pipeline recomputation, wasting significant resources. Additionally, SWARM nodes employ a greedy procedure to select their next stage successor, which does not guarantee optimal pipeline duration or takes a device's memory constraints into account.

Gossip Learning. Previous works on decentralized training mainly focused on exploiting data parallelism, and predominantly relied on gossip-based communications [12–14]. In those approaches, during the parameter sharing step, nodes send their model weights to a subset (of size k - the group size) of known peers. After receiving all the weights they expect during an iteration, including their own, nodes compute their average and use it in the following iteration. Gossip learning can naturally tolerate device crashes, dynamic communication graphs, heterogeneous networks (by preferring faster links), and only requires nodes to maintain partial membership knowledge.

III. System Model and Objectives

System model. Nodes independently contribute resources for limited periods and are geographically distributed with individual memory and communication constraints. Each has a partial view of the system and communicates with known peers over authenticated, unreliable, and heterogeneous links. The network is partially synchronous, i.e., there are periods of time where the network latency is bounded. We model a node i with a given memory capacity with a maximum number cap_i of microbatches it can process at a time, and denote by $d_{i,j}$ the communication delay between a node pair (i, j). Nodes act as data nodes, relay nodes, or both. Relay nodes merely contribute compute power and host intermediate stages in the pipeline parallel setting. Data nodes hold the data and thus serve the first and final stages (equivalent to the embedding and de-embedding layer). Data nodes can coordinate between each other (equivalent to Fully Sharded Data Parallel training) or can utilize their own data local private data (equivalent to decentralized federated learning). Our system is agnostic to this choice.

Node churn. Nodes can join, leave, or crash at any time, including during forward or backward passes. We focus on

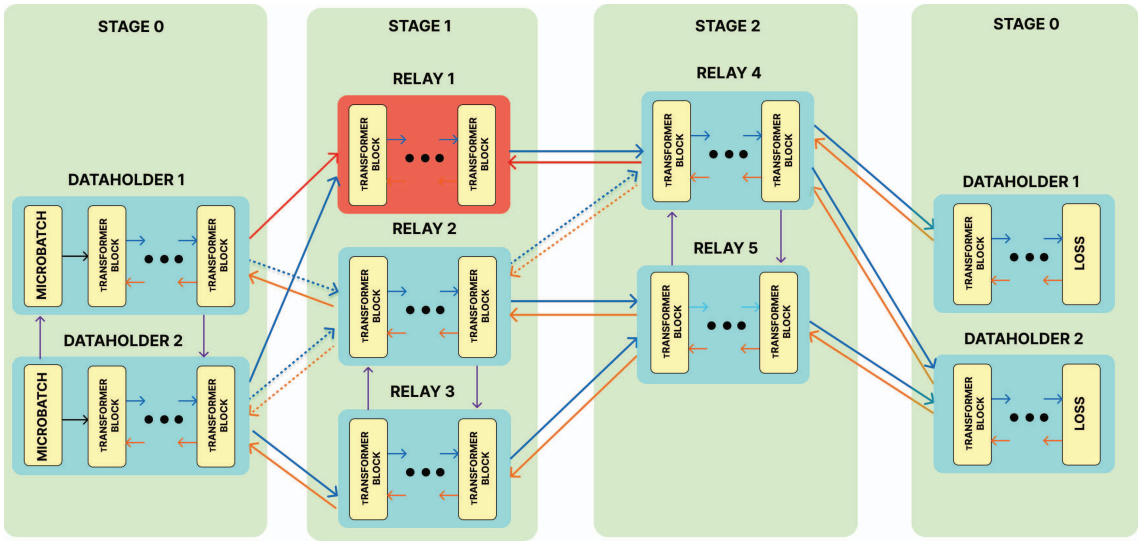

Fig. 1: Crash-recovery during decentralized training of an LLM with GWTF. The plain blue arrows between nodes indicate the forward passes, the plain orange arrows indicate the backward passes, the plain purple ones indicate model aggregations, and the red arrows indicate some network transmissions that fail because relay 1 crashes (shown in red). Dashed lines indicate the newly formed microbatch exchanges that are generated to recover from the crash of relay node 1.

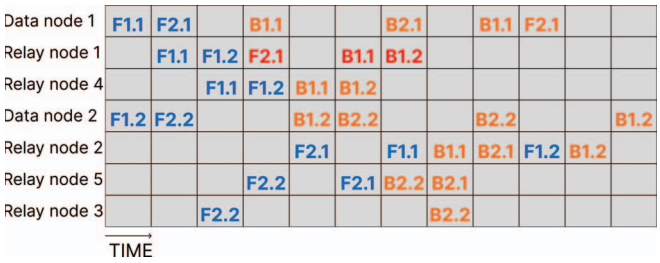

Fig. 2: Execution scenario of the decentralized training of an LLM. Notations Fx, y and Bx, y to indicate a forward pass, respectively a backward pass, using microbatch x generated by data node y. Blue is used for a forward pass, orange for a backward pass, and red indicates a task that is not executed because of a node crash.

the crashing case, which we resolve through timeouts. A node crashing during a forward pass slows down the construction of the forward pass. A node crashing during a backward pass requires additional synchronization between nodes across different stages in order to recover the pipeline by replacing the missing nodes.

Objectives. We aim to train LLMs efficiently under churn, heterogeneity, and unstable networks. Prior work [4, 15] assume stable and homogeneous conditions, which do not usually hold in decentralized settings, and crashes can halt training. SWARM [6] addresses forward-pass faults via rerouting but still requires full pipeline recomputation on backward-pass failures, doubling training time.

Fig. 1 illustrates the training of an LLM with GWTF using 3 stages, 2 data nodes that hold the training data, and 5 relay nodes that execute the intermediate training steps for two batches. Relay node 1 crashes when processing the microbatch of the first data node, and does not execute a forward pass from the second data holder, or a backwards pass for the first data node, which are redirected to relay node 2. Fig. 2 illustrates the corresponding execution scenario.

IV. GWTF IN A NUTSHELL

This section highlights the cost minimization objective of GWTF and its key design components.

Flow and Cost. GWTF aims at minimizing the distributed training cost of an LLM, which we define in the following manner. We model the average delay, termed cost, of exchanging activations between two nodes i and j using the sum of its computation times and communication delays. We term this "flow" - an abstraction of the end-to-end processing of a microbatch through the stages in the system. In Figure 1 we can see a single flow going from dataholder 2, to relay 3, to relay 5, back to dataholder 2. We use this notion of flow instead of the traditional pipeline parallelism, allowing nodes to communicate with multiple other nodes in the previous and subsequent pipeline. Following Yuan et al. [4], the communication cost between nodes i and j is composed of network latency ($\lambda_{i,j}$) and a transmission delay that is computed as the amount of transferred data ($size$) divided by the network bandwidth ($\frac{size}{\beta_{i,j}}$). We assume that links are not necessarily symmetric, i.e., that $\lambda_{i,j}$ and $\beta_{i,j}$ are not necessarily respectively equal to $\lambda_{j,i}$ and $\beta_{j,i}$. However, as every link is used twice during training (once from i to j during the forward pass and once from j to i during the backward pass), we can model the latency on the link as the average of the two delays. The final communication cost between two nodes i and j in a given phase is then

$\frac{\lambda_{i,j}+\lambda_{j,i}}{2}+\frac{2\cdot size}{\beta_{i,j}+\beta_{j,i}}$. We denote the computation cost of nodes i and j respectively by c_i and c_j. Computation costs capture the time it takes a node to process a microbatch during the forward or the backward pass. The final cost $d_{i,j}$ of a microbatch flow between nodes i and j is therefore defined by the following equation:

$$d_{i,j} = \frac{c_i + c_j}{2} + \frac{\lambda_{i,j} + \lambda_{j,i}}{2} + \frac{2 \cdot size}{\beta_{i,j} + \beta_{j,i}} \quad (1)$$

GWTF aims at processing the largest number of microbatches at the lowest global cost, and uses a novel decentralized algorithm to route microbatches through the nodes.

Key Design Components. During each iteration, data nodes first send out the embedded microbatches to the relay nodes in the first stage. Each microbatch is associated with a unique id, the data node of its origin, and a path list, which holds the nodes it has traversed during its execution.

A relay node executes up to 5 different sub-routines when participating in the system: (1) forming of a pipeline with the flow algorithm; (2) joining the system; (3) resending in case of a node crash during forward pass; (4) recovering the pipeline in case of backward failure or lost batches; and (5) aggregating the models received from other nodes that belong in the same stage. The first four procedures run in parallel to the actual processing of forward and backward passes (the training phase). The last procedure is the model update phase.

The decentralized flow cost minimization algorithm, reduces the cost of the forward and backward pass for each microbatch. It aims at sending a maximum number of batches at the smallest cost according to Equation 1. Nodes with different capacities route microbatches in a cost-effective manner, independently and without global knowledge.

Nodes are assigned to stages when they join the system. GWTF aims to increase the throughput of the stage with the minimum capacity, as that stage puts a bottleneck on the current throughput in an iteration. Nodes discover other peers in the system through a Distributed Hash Table (DHT) [16]. An elected leader from the data nodes periodically adds new nodes, prioritizing nodes with higher capacity to join the stage with the smallest capacity, thus expanding the bottleneck of the system. Note that the leader can be elected in a robust way [17, 18].

GWTF handles nodes leaving due to unavailability or crashes, which is more challenging than handling new joiners. Crashes occurring during a forward pass are resolved by resending to another peer in the next stage, according to the new flow found by (Sec. V-A. During a fault in a backwards pass the microbatch list is used to replace the crashed nodes and restore the pipeline with a minimal amount of computations. More precisely, to amend a broken flow, GWTF searches for the last alive node before a crash to quickly repair the flow to the first alive node after the crash, via nodes that have spare computational resources in between.

V. GWTF Details

This section first provides the details of GWTF's decentralized flow optimization, which aims at maximizing the training throughput of an LLM on nodes of different capacity at a minimal cost. Then, it discusses how GWTF inserts new nodes into existing pipelines, and how pipelines are dynamically optimized. Finally, it explains how nodes crashing or leaving the system are tolerated.

A. Decentralized flow optimization

In parallel to the training and node addition, GWTF constructs flows (abstract pipelines) for a microbatch starting from a data node and sequentially determining the relay nodes for each stage. To build a flow, the availability of known nodes and their memory constraints are considered. We assume that the memory requirement of each stage per microbatch is known through offline profiling, which can be done by the data nodes themselves and subsequently announced to all nodes. Using this information, nodes can determine their capacity based on their known memory.

The objective of GWTF is to complete the entire flow of all microbatches at the minimum cost, namely the sum of $d_{i,j}$ from Equation 1 along the path of the flow. This optimization problem is closely related to the problem of minimizing cost and maximizing flow in a multiple-source multiple-sink graph. The sources and the sinks are both the data nodes, as microbatches travel back to their origin for loss computation. Flow on a given edge/link equates to number of microbatches transferred between the two nodes connected by that link. A unit of flow is thus a microbatch. In such a problem the goal is to minimize the sum of costs of all flows within the graph:

$$min \left(\sum_{\forall i,j \in \mathbf{N}} f(i,j) \cdot d(i,j) \right) \quad (2)$$

where $f(i,j)$ is the flow between two nodes on the link that connects the,. The problem assumes a linear model for cost to flow increase, hence the inner product of $f(i,j) \cdot d(i,j)$ expressing the cost of some amount of flow along some edge. The classical algorithm [19] one can use to solve our optimization problem relies on global information, which is not available in decentralized settings. Additionally, unlike in the standard definition which requires that any flow from a source be delivered to any sink, GWTF has to deliver a flow from a source back to itself.

To this end we design a novel distributed algorithm that leverages only local knowledge and differentiates between flow from different sources. However, the objective function of Equation 2 requires greater synchronization between nodes and converges slowly, as nodes need global knowledge of the cost of all flows. Fortunately we can solve a much easier problem with only local knowledge for each node by minimizing the cost of the maximal flow between two nodes: $\min\ (\max_{\forall i,j \in \mathbf{N}}\ f(i,j) \cdot d(i,j))$. A similar function has also been used in a previous work [4], where their goal is to minimize the maximum communication cost between two subsequent nodes in a pipeline.

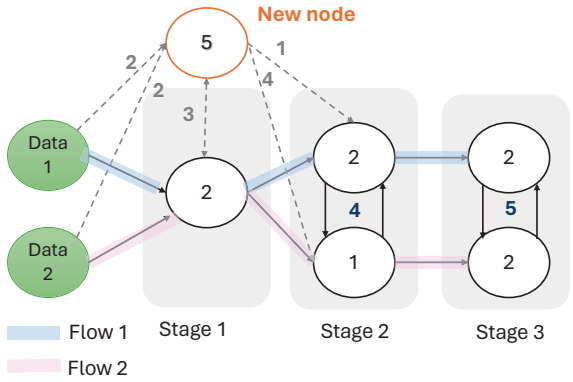

Fig. 3: A joining node being added to stage 1 in a system that includes two data nodes and three relay nodes.

B. Inserting Joining Nodes

We design a two-way procedure to assign a new node to the stage that is identified to be the bottleneck. A data node is elected, which we call the leader, and is in charge of ranking the stages based on their utilized ratio, which is computed as the number of flows that go through them divided by their total available capacity. Stages with a higher utilization are given a higher priority to incorporate incoming nodes. Periodically, a flooding algorithm is used to find the utilization of each stage. This is initiated by the leader, who sends a query to all known peers in the following stage. Whenever a node receives a query from the previous stage, it adds its capacity and utilization to the message, and forwards it to known peers in the subsequent stage.

A joining node discovers other peers in the system and the leader's identity through a Distributed Hash Table (DHT) [16]. Upon joining the system, new nodes (candidates) send their capacity to the elected leader. The leader, periodically, selects the candidates with highest capacities and adds them to the most utilized stages. The candidate with the highest capacity will be added to the stage with highest utilization, the second highest to the stage with second highest, and so on.

Figure 3 illustrates GWTF's node joining procedure. In this example, there are 2 dataholder nodes (on the left, in green), each sending a single microbatch flow traversing 4 stages. Each node is indicated by a circle whose value denotes the node's capacity. Stages 1, 2 and 3 have a total capacity of hosting 2, 3 and 4 microbatch flows. Stage 1 is therefore the bottleneck of the decentralized training. The respective cost of the two flows are 6 (top, in blue) and 10 (bottom, in purple). GWTF inserts a joining node of capacity 5 into stage 1. Following the addition of this new node, the first stage's capacity increases to 7, and the throughput of stage 2 increases to 3, thus now stage 2 becoming the bottleneck and new joining nodes will be added to it.

C. Building Pipelines using Flow Requests

GWTF uses three main subprocedures to solve decentralized optimization: Request Flow, Request Change, and Request Redirect. Request Flow builds execution pipelines, while the other two reduce training costs once flows are established. These rely only on local knowledge of system membership and node *inflow*/*outflow*. The *outflow* of a node is all the flow it sends to the subsequent stage. The *inflow* is the flow it receives from a previous stage. In stable conditions, a relay node's inflow should equal its outflow. Before stabilization, some nodes have unpaired inflows (wanting to send) or unpaired outflows (wanting to receive). Data nodes start with unpaired flows matching their capacity. Sending outflow reduces capacity by 1, even if unpaired. Each flow has a unique ID, target data node, and current cost to sink—calculated in reverse, from the last stage to the first.

During an iteration, nodes that have available capacity and are in a stable state (no unpaired inflow or outflow) look for a node in a subsequent stage with an unpaired outflow to a specific data node. If multiple such nodes exist, a node i prefers the node j that minimizes (the sum of the cost of flow from j to the sink + the cost of sending from i to j), as per 1. Similarly, nodes with unpaired inflow look for a node that has unpaired outflow with the same data node, as the data node of the unpaired inflow. Nodes with unpaired outflow do not perform this search, but merely respond to requests.

Request Flow. Once a node has found a suitable node with unpaired outflow, it sends it a Request Flow message with the data node and the cost of the flow of the unpaired outflow it is requesting. When a node receives a Request Flow, it checks the associated data node and cost coming with the request. If it does have unpaired outflow to that data node at that cost, it approves it and adds inflow which connects to the unpaired outflow. If it does not, it rejects it and informs the requester of its current cost to that destination (which is infinite if it has not unpaired outflow to it). Upon seeing its flow request approved, a node adds it to its unpaired outflow (unless it had already unpaired inflow it can connect it to). It then calculates its minimum cost to that sink as the cost between the two nodes, as explained in Equation 1, plus the cost of the flow requested as reported by the other node, and broadcasts it to nodes in previous stages. Nodes that have outflow equal to their capacity do not generate Request Flow calls. If a node has no peers it can request flow from, then it aims at minimizing its sending cost to the next stage by communicating with the nodes from its stage. This minimization relies on simulate annealing. This is done by querying all of its same stage peers about their current flows, and checking two conditions. **Request Change.** If two nodes have flow to the same sink but with different next stage peers, one of these nodes can determine if switching these flows would reduce the objective function (e.g., maximum sending cost), and consequently request a switch. For example, Node 1 is communicating with Node 2 to Data Node 9, at an intermediate cost $d(1,2) = 3$. Node 3 is communicating with

Algorithm 1 Flow Request-Accept pseudocode

Parameters:
  i : current node
  cap_i : capacity of the node
  S_i : stage of the node
  $flow_i$: current flow
upon event $\langle$`Flow, Init`$\rangle$ **do**
  $Outflow = \emptyset$
  $Inflow = \emptyset$
  $Next\ Stage = \emptyset$
  $Previous\ Stage = \emptyset$
  $Same\ Stage = \emptyset$
  $unmatched = 0$
  if $source$ **then**
    $unmatched = cap$
  if $sink$ **then**
    $unmatched = -cap$

upon event $\langle$`Flow, Request`$|j, cost, target\rangle$ **do**
  if don't have flow at $cost$ to target **then**
  $\langle SEND$, `REJECT` $|j$, i, smallest cost to target, target, unmatched$\rangle$ `// Reject their request`
  if $unmatched > 0$ **then**
    $\langle SEND$, `REJECT` $|j$, i, smallest cost to target, $target, unmatched\rangle$ `// Reject their request`
  $unmatched = unmatched + 1$
  $Inflow = Inflow \cup \{this\ flow\}$
  $\langle SEND$, `ACCEPT` $|j$, i, $cost, target\rangle$ `// Accept their flow`

upon event $\langle$`Flow, ACCEPT`$|j, cost, target\rangle$ **do**
  $Outflow = Outflow \cup \{this\ flow\}$
  $unmatched = unmatched - 1$
  $flow_i = flow_i + 1$
    $\langle BROADCAST$, `UPDATE`, i, smallest cost to target, $target, unmatched\rangle$ `// Reject their request`

Node 4 to Data Node at an intermediate cost $d(3,4) = 8$. However, if instead Node 1 communicates with Node 4 and Node 3 with Node 2, they would have costs $d(1,4) = 6$ and $d(3,2) = 6$. The maximum cost on a flow would thus be minimized (8 to 6). If the second node agrees to the switch, because it has that flow and it also thinks that the objective function will be reduced, then each node now sends its outflow to the next peer of the other one. This is performed through an accept message to the same stage peer and a message to the next stage peer that it has a new incoming peer for the same outflow. Otherwise, if the node doesn't agree, nodes keep their outflows as they are. Pseudocode of this procedure is presented in Alg. 1.

Request Redirect. If a node has capacity and sees a peer's flow between two stages, it checks whether rerouting through itself reduces cost. If so, it sends a Request Redirect. For example, if Node 1 $\rightarrow$ Node 2 $\rightarrow$ Node 3 costs 11, but Node 1 $\rightarrow$ Node 4 $\rightarrow$ Node 3 costs 9, Node 4 can request the redirect. The original node (e.g., Node 2) checks and, if approved, swaps its outflow and cancels its previous flow. To avoid local minima, we use simulated annealing [20]: changes that increase cost may still be accepted with probability $e^{(cost_{current} - cost_{new})/T} > U(0,1)$, where T is temperature, reduced after each accepted change by a factor α. This allows occasional costly moves to find better solutions.

Once a system has entered a steady state for a few iterations

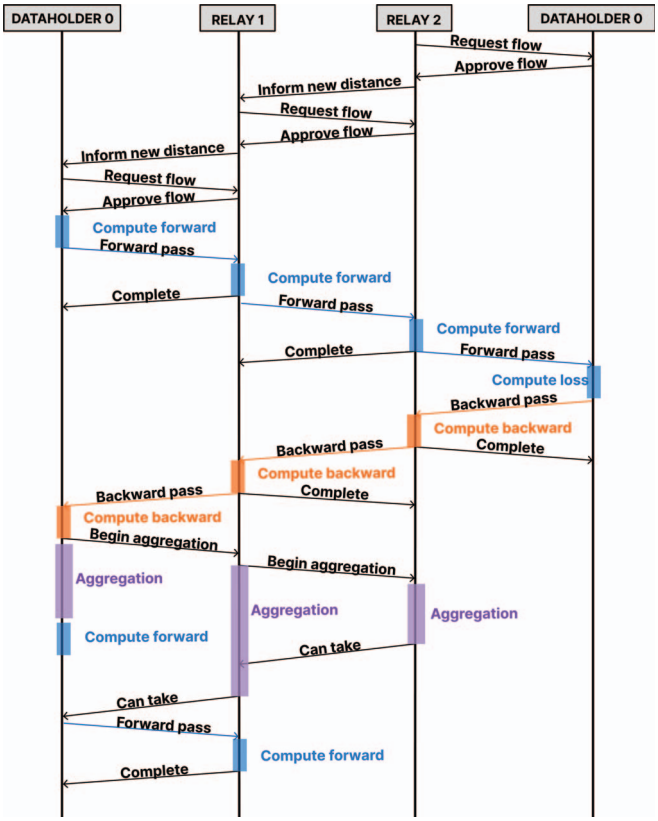

Fig. 4: Communication and training happening during one iteration in a simple 3-stage pipeline without any crashes and with 1 microbatch per iteration. For ease of visualisation messages that relate to the aggregation phase are omitted.

of the algorithm above, the training can proceed. Since the data transferred is small messages, convergence of this algorithm to a close to optimal steady state execution is significantly faster than a training iteration.

D. Tolerating Crashes

After completing a batch, a node sends a COMPLETE message with the batch ID to its upstream peer, allowing latency estimation. During training, if a node does not receive a COMPLETE reply within a set time, it assumes the peer is unresponsive and reroutes traffic using the previous algorithm. If no alternate peer is available, it sends a DENY message upstream, prompting flow redistribution. This process can continue recursively up to the source, which may defer the batch to the next iteration. Nodes that send DENY messages are excluded until they free memory.

Crashes during the forward pass. Crash recovery during the forward pass of relay nodes is done as in SWARM [6]. Pipelines are constructed on the fly and a microbatch is routed through them independently by each peer. Instead of using SWARM's stochastic wiring algorithm, `GWTF` uses a flow algorithm that optimizes sending costs and takes into account the memory constraints of each node.

Crashes during the backward pass. Nodes send a COMPLETE message after finishing computation. If a node does not receive a COMPLETE message, it notifies the data node. If the data node has no pending microbatches, the pipeline needs not be restored. Otherwise, it pings the first node on the microbatch path. Nodes ping downstream peers along this path, and if a ping fails, the node forwards its activation to a new downstream peer. That peer processes the activation and tries to send results to a previously active node. If unsuccessful, it keeps rerouting until reaching a live node or the data node. The backward pass then resumes from the stored gradient, avoiding recomputation. This recovery is far cheaper than rebuilding the pipeline from scratch, as done in prior work [6].

E. Training-Aggregation Synchronization

Once a new node joins the training and downloads the weights of the stage it will serve, it begins running GWTF's flow algorithm in parallel to the actual training. When a fault occurs it runs one of the two procedures of Section V-D. Nodes also alternate between training and aggregation phases, which has not been described in previous works [6]. This synchronization is necessary as nodes in the same stage need to have identical parameters when processing microbatches in a iteration. GWTF relies on a simple algorithm through which nodes signal to each other when this transition happens. The aggregation phase starts when the data node leader sends a BEGIN AGGREGATION message, which propagates through the network. Upon receiving it, nodes broadcast and collect model weights within their stage. Once a node completes aggregation and detects that a downstream peer has also finished, it sends a CAN TAKE message upstream to signal readiness for new microbatches. Nodes in the last stage send this without waiting. The system involves several passes: pipeline formation (back to front), forward pass (front to back), backward pass (back to front), relaying of BEGIN AGGREGATION (front to back), and CAN TAKE messages (back to front), marking the end of aggregation and start of a new iteration. This communication pattern is illustrated in Figure 4.

VI. Performance Evaluation

In this section, we demonstrate that GWTF can provide significant speed up for training a large model in decentralized settings, even in the presence of node crashes. We further compare the performance of our scheduler against the optimal communication scheduler of Yuan et al.'s DT-FM [4]. We finally verify the applicability of our system in a practical scenario by showing that the model converges at a rate similar to the one of a centralized solution.

Setup. Our experiments were performed on LLama and GPT architecture models with varying model sizes. We use a private cluster of 5 NVIDIA RTX A4000 GPUs with 16 GB of GPU memory. Each GPU hosts a number of logical geo-distributed nodes. We simulate the geo-distributed locations by limiting the bandwidth and increasing the latency between logical nodes, with a maximum bandwidth between two nodes in simulated different locations of 500Mb/s and a minimum of 50 Mb/s. Unless stated otherwise, our decentralized optimization uses $T = 1.7$ for the initial temperature, $\alpha = 0.95$ for the cooling factor.

Node Crashes. We evaluate GWTF on a LLaMa-like model ($d_{model} = 1024$, $n_{heads} = 18$ and 16 layers) with microbatches of size 4 and sequence length 512. To simulate realistic communication costs, bandwidth is reduced by a factor 32, mimicking activations 32 times larger. Experiments use 18 nodes, with the model split across 6 stages: 3 blocks per stage, except the first, which includes the embedding, 1 block, data retrieval, and loss computation. Two persistent data nodes each push 4 microbatches per iteration. We compare against SWARM [6], a crash-tolerant decentralized training method that restarts pipelines after backward pass timeouts. Experiments vary by node capacity and join/leave probability. In the heterogeneous setting, relay node capacities range from 1–3; in the homogeneous case, all are set to 4. Join-leave chance varies from 0% (no churn) to 10% (nodes may randomly crash or rejoin each iteration).

We report in Table II the following metrics, averaged over 25 iterations: (1) Minutes per microbatch: The maximum time per iteration (from the slowest data-holder's perspective), divided by the number of microbatches processed; (2) Throughput per iteration: Total number of microbatches successfully processed in one iteration; and (3) Wasted GPU time: Total pipeline compute time (in minutes) spent on microbatches that were either excluded from aggregation or not part of the final path. GWTF outperforms SWARM in all heterogeneous settings, with up to 45% speedup under 10% crash rates. In homogeneous settings, GWTF matches SWARM in fault-free cases and outperforms it when crashes occur. SWARM wastes more GPU time due to misrouted microbatches or full pipeline recomputation after faults. These results demonstrates that GWTF is able to significantly outperform the state of the art in both homogeneous and heterogeneous crash-prone settings.

We show that GWTF is model-agnostic by repeating the tests on a GPT-like model ($d_{model} = 1024$, $n_{heads} = 18$ and 16 layers). Results are similar to those with the LLaMa model, but with over 2 times faster iteration time in the homogeneous 0% crash case. This gain is likely due to GPT's higher activation communication overhead.

Handling Joining Nodes. We evaluate GWTF's handling of joining nodes depending on the number of stages, the node capacity, and the intra- and interlayer. For each test a total of 97 nodes are used (with 1 of them being a dataholder). The number of nodes in each stage is the same and can be calculated as $\frac{N-1}{S}$ where N is the number of nodes and S the number of stages. The last experiment has a different number of nodes. In the last of the tests (5*) , the number of nodes per stage is randomly chosen. All nodes in the system have properties that adhere to the values described in Table IV. To construct the system, we use the Out-of-kilter algorithm [19] to determine the minimum cost flow. Iteratively, 20 nodes are added in different stages. Since every node is a candidate for

TABLE II: Performance with crash-prone devices. The average results over 25 repetitions are reported with the standard deviation.

	SWARM	GWTF	SWARM	GWTF	SWARM	GWTF
	Homogeneous 0%		Homogeneous 10%		Homogeneous 20%	
Time per microbatch (min)	$\mathbf{0.53 \pm 0.13}$	0.58 ± 0.16	1.26 ± 0.87	$\mathbf{1.01 \pm 0.37}$	1.76 ± 1.3	$\mathbf{1.17 \pm 0.43}$
Throughput (#microb/iteration)	$\mathbf{8.0 \pm 0.13}$	7.16 ± 0.17	4.64 ± 0.87	$\mathbf{5.8 \pm 0.37}$	$\mathbf{6.1 \pm 1.3}$	5.72 ± 0.43
Communication time	6.07 ± 4.92	$\mathbf{4.2 \pm 2.52}$	12.23 ± 8.81	$\mathbf{7.76 \pm 2.24}$	17.74 ± 13.15	$\mathbf{4.57 \pm 2.68}$
Wasted GPU time	0.27 ± 0.0	$\mathbf{0.03 \pm 0.0}$	0.75 ± 0.0	$\mathbf{0.2 \pm 0.0}$	1.75 ± 0.0	$\mathbf{0.0 \pm 0.0}$
	Heterogeneous 0%		Heterogeneous 10%		Heterogeneous 20%	
Time per microbatch (min)	1.59 ± 0.21	$\mathbf{1.15 \pm 0.44}$	4.53 ± 4.08	$\mathbf{2.45 \pm 0.93}$	5.36 ± 3.56	$\mathbf{3.47 \pm 2.06}$
Throughput (#microb/iteration)	2.96 ± 0.21	$\mathbf{3.6 \pm 0.44}$	1.56 ± 4.08	$\mathbf{2.08 \pm 0.93}$	1.72 ± 3.56	$\mathbf{2.12 \pm 2.06}$
Communication time	10.55 ± 2.79	$\mathbf{3.65 \pm 1.19}$	3.95 ± 1.8	$\mathbf{2.62 \pm 1.27}$	7.18 ± 7.22	$\mathbf{4.28 \pm 2.85}$
Wasted GPU time	0.57 ± 0.0	$\mathbf{0.0 \pm 0.0}$	0.33 ± 0.0	$\mathbf{0.1 \pm 0.0}$	0.33 ± 0.0	$\mathbf{0.03 \pm 0.0}$

TABLE III: Performance with crash-prone devices for a GPT-like model. The average results over 25 repetitions are reported with the standard deviation.

	SWARM	GWTF	SWARM	GWTF	SWARM	GWTF
	Homogeneous 0%		Homogeneous 10%		Homogeneous 20%	
Time per microbatch (min)	0.68 ± 0.08	$\mathbf{0.37 \pm 0.01}$	1.77 ± 1.25	$\mathbf{0.99 \pm 0.33}$	1.92 ± 1.32	$\mathbf{1.32 \pm 0.56}$
Throughput	$\mathbf{8.0 \pm 0.0}$	$\mathbf{8.0 \pm 0.0}$	4.52 ± 1.35	$\mathbf{6.39 \pm 1.57}$	4.35 ± 1.13	$\mathbf{6.04 \pm 0.43}$
	Heterogeneous 0%		Heterogeneous 10%		Heterogeneous 20%	
Time per microbatch (min)	1.4 ± 0.16	$\mathbf{1.18 \pm 0.04}$	3.5 ± 2.32	$\mathbf{2.63 \pm 1.08}$	5.3 ± 4.99	$\mathbf{3.16 \pm 1.79}$
Throughput	2.96 ± 0.17	$\mathbf{4 \pm 0.04}$	1.78 ± 1.4	$\mathbf{2.26 \pm 0.93}$	1.83 ± 3.56	$\mathbf{2.39 \pm 2.06}$

each stage, we assume that each node knows its costs for each stage. The optimal choice of node addition is determined by running the minimum cost flow algorithm [19] for each combination of S candidate nodes added to each of the S stages in the tests. Figure 5 reports the average improvement over 10 runs measured as $\frac{cost_{now} - cost_{after}}{cost_{now}}$. We consider two baselines - adding highest capacity first and adding random nodes. In all cases GWTF outperforms the two baselines, however it never achieves an optimal schedule. Still, this optimal schedule cannot be achieved in a decentralised setting - it takes awhile to compute, as it involves trying out every permutation of candidates, and requires global knowledge to run a flow algorithm for every possible permutation. In spite of these limitations, our solution is never more than 25% slower than the optimal schedule. It is also up to 1.5 times as fast as the highest capacity first baseline and up to 3.5 times faster than the random baseline

Optimality. We compare the end-to-end training time of GWTF against a GPipe setting with a communication optimal arrangement, calculated by DT-FM [4]. The setting of the experiment are mostly identical to those of the prior 0% homogeneous setting. Following prior work [4] we use several pipelines with 4 microbatches per pipeline. In order for the two system to be comparable, we have 3 dataholders and 15 relay nodes distributed across 6 stages (3 nodes per stage). The end-to-end training time between the two systems is compared in Table VI. Although the optimal computation schedule outperforms GWTF by almost 13% (0.44 s and 0.51 s for DT-FM and GWTF, respectively), it takes much longer to be computed, as it involves the use of a genetic algorithm [4] and scales exponentially with the number of nodes. GWTF is therefore approaching the optimal schedule and can be used at scale.

TABLE IV: Node addition (top) and flow test (bottom) settings. ϕ represents the maximum interlayer cost of a node for the stage it is in/proposing to be in.

	Stages	Capacities	Interlayer Costs	Intralayer Costs
1	8	$\lfloor U(1,20) \rfloor$	$\lfloor U(1,100) \rfloor$	$\phi + \lfloor U(50,100) \rfloor$
2	8	$\lfloor U(1,20) \rfloor$	$\lfloor U(20,100) \rfloor$	$\phi + \lfloor U(50,100) \rfloor$
3	8	$\lfloor U(1,5) \rfloor$	$\lfloor U(1,100) \rfloor$	$\phi + \lfloor U(50,100) \rfloor$
4	12	$\lfloor U(1,20) \rfloor$	$\lfloor U(1,100) \rfloor$	$\phi + \lfloor U(50,100) \rfloor$
5*	8	$\lfloor U(1,20) \rfloor$	$\lfloor U(1,100) \rfloor$	$\phi + \lfloor U(50,100) \rfloor$
	Sources	Relays	Stages	Capacities	Link costs
1	1	40	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$
2	1	40	10	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$
3	1	40	8	$\lfloor U(5,15) \rfloor$	$\lfloor U(1,20) \rfloor$
4	1	40	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(5,100) \rfloor$
5	2	40	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$
6	4	80	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$

TABLE V: Flow Test Settings

	Sources	Relays	Stages	Capacities	Link costs
1	1	40	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$
2	1	40	10	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$
3	1	40	8	$\lfloor U(5,15) \rfloor$	$\lfloor U(1,20) \rfloor$
4	1	40	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(5,100) \rfloor$
5	2	40	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$
6	4	80	8	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$

Training Convergence. Since our system does not modify the training of the model (always the entire model is ran as in a centralized solution), GWTF has the same theoretical convergence as SGD. We verify this by training our system with 10% crash rate and a single GPipe pipeline of 8 nodes, with 8 microbatches of size 1 and sequence length 4096. Our system's

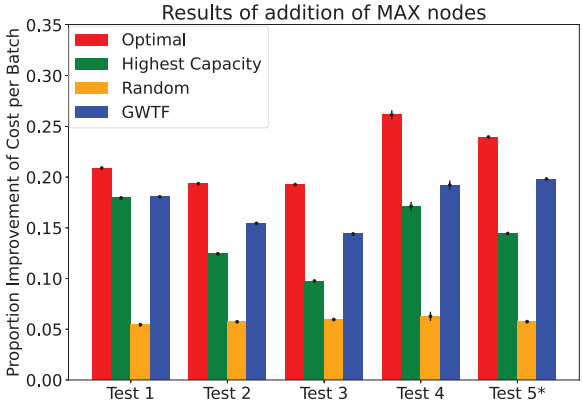

Fig. 5: Average improvement over 10 runs with the new node addition tests. Variance is reported with black lines. Higher means better.

TABLE VI: Comparison against optimal schedule

	Time per microbatch	Throughput per iteration
DT-FM [4]	0.44 ± 0.0	9.0 ± 0.0
GWTF	0.51 ± 0.08	8.08 ± 0.08

hyperparameters were heterogeneous nodes with 10% crash chance and 1 data node. The model used in this experiment is the LLaMA-7b distributed uniformly across 8 stages. The dataset used was the Wikipedia English dataset [21]. Figure 6 confirms that GWTF has a similar convergence of loss as a centralized solution with the same batch size.

Ablation studies. We evaluate our flow algorithm in 6 different settings (cf. Table IV). The first four settings involve a single source-sink node while the last two have multiple source-sink nodes. Details about the tests are presented in Table V. In all cases the source-sinks were given sufficient capacity to prevent bottlenecks. In order to compare to the optimal result of Fulkerson's algorithm [19], our procedure attempts to minimize the sum of the costs of all flows, $min\left(\sum_{i,j\in \mathbf{N}} d(i,j) * f(i,j)\right)$. Tests 5 and 6 are not compared with the optimal baseline, as there the formulation differs, in that a source must deliver to a specific sink. We use the approach from SWARM [6] of sending to the next stage closest node as a baseline. Experiments are evaluated on at most 120 iterations (roughly a minute) of the different algorithms. As Figure 7 shows, GWTF consistently outperforms SWARM by up to 50% in heterogeneous communication settings. This partially explains the higher throughput our system achieves compared to Swarm [6], as it greedy approach can choose significantly worse paths. Additionally, due to the negligible time required to compute the paths relative to the actual training, the cost of using this approach instead of SWARM's greedy approach is outweighed by training time reduction.

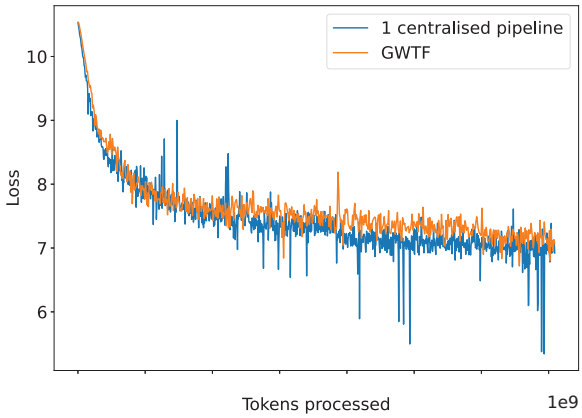

Fig. 6: Loss convergence

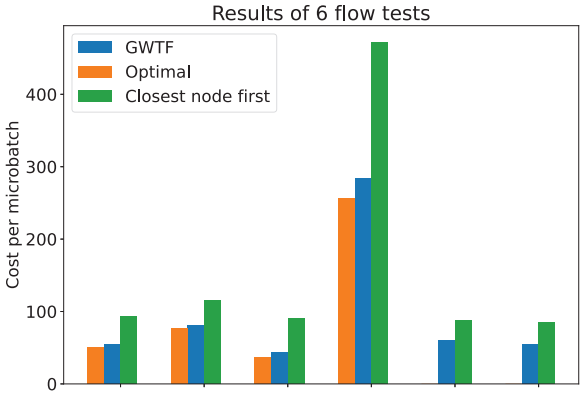

Fig. 7: Average cost per microbatch in flow tests

VII. Discussion and Future Work

a) Byzantine Nodes: While this work does model nodes as crash-prone, it does not consider nodes who might deviate from the protocols described. Such nodes are typically referred to as Byzantine [22] and could, for example, prevent convergence in this setting by sending random activations. In order for any system do be viable in any practical setting, such nodes need to be dealt with in an efficient manner. Unfortunately, while the literature on training large language modes in a decentralised setting is greatly lacking, work on training them in a setting with Byzantine nodes is non-existent.

b) Checkpointing: While this paper assumes that at least one node remains active per stage, in a realistic setting this is not guaranteed. However, such issues are typically addressed via check pointing (storing current parameters on another remote device) [23]. Recent work on this matter assumes a stable, non-faulty, central node, which can store the check points [23, 24]. However, this is insufficient for our setting. Efficient decentralised checkpointing with crash-prone devices remains an unexplored topic.

c) Incentives: Throughout this paper we assume volunteer nodes - nodes sparing some resources for no com-

pensation. Training of large language models can be both computationally and time intensive. As such, many individuals may be discouraged to participate in the training, due to the energy costs they might be faced. However, as seen by the recent popularity of blockchains in the mainstream [25], individuals may be incentivised to participate at the promise of a reward proportional to their contribution. Blockchains are especially well suited for this setting, as rewards can be handed instantaneously, can be tied to an existing asset, and offer good scalability with increased number of workers. Recent work has explored this idea in depth and we consider it as a possible extension of this system [26, 27]

d) Different Architectures: This work primarily focuses on Large Language Models for its experiments. However, the ideas presented are not LLM exclusive. Any system, which necessitates either pipeline parallelism or data parallelism can make use of parts or the whole of our framework. Vision Transformers share a similar architecture as Large Language Models and also benefit from pipeline parallelism, due to their growing sizes [28]. Surprisingly, even convolutional models such as ResNet and VGG can benefit from pipeline parallelism (though it requires pipelining of microbatches with some staleness awareness to maximise efficiency) [29]. For both scenarios, the framework presented in this work can in theory be utilised.

VIII. Conclusion

We described `GWTF`, the first crash tolerant decentralized training framework that aims to minimize training time and maximize its throughput. `GWTF` models the decentralized training procedure as a flow problem, and effectively decides its execution on heterogeneous clients and network links. `GWTF` is structured in two key modules: decentralized flow computation and tolerating crashes. We evaluate `GWTF` on decentralized training LLama-like and GPT-like models in a geo-distributed setting, against SWARM and GPipe, on both homogeneous and heterogeneous setups with different node churning dynamics. Our results show that `GWTF` is able to improve the training time by up to a 45% and throughput by up to a 30% increase while wasting almost zero GPU time of any joining clients.

References

[1] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. *Improving language understanding by generative pre-training*. 2018.

[2] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *ICLR*. 2021.

[3] H. Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: *CoRR* abs/2302.13971 (2023). arXiv: 2302.13971.

[4] B. Yuan, Y. He, J. Davis, T. Zhang, T. Dao, B. Chen, P. Liang, C. Ré, and C. Zhang. "Decentralized Training of Foundation Models in Heterogeneous Environments". In: *NeurIPS*. 2022.

[5] M. Shirts and V. S. Pande. "Screen Savers of the World Unite!" In: *Science* 290.5498 (2000), pp. 1903–1904.

[6] M. Ryabinin, T. Dettmers, M. Diskin, and A. Borzunov. "SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient". In: *ICML*. 2023.

[7] A. Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *J. Mach. Learn. Res.* 24 (2023), 240:1–240:113.

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

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention is All you Need". In: *NeurIPS*. Ed. by I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett. 2017.

[10] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. X. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *NeurIPS*. Ed. by H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett. 2019.

[11] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao. "Optimal Distributed Online Prediction Using Mini-Batches". In: *J. Mach. Learn. Res.* 13 (2012), pp. 165–202.

[12] P. Zhou, Q. Lin, D. Loghin, B. C. Ooi, Y. Wu, and H. Yu. "Communication-efficient Decentralized Machine Learning over Heterogeneous Networks". In: *ICDE*. 2021.

[13] J. Jiang, W. Zhang, J. Gu, and W. Zhu. "Asynchronous Decentralized Online Learning". In: *NeurIPS*. 2021.

[14] M. de Vos, S. Farhadkhani, R. Guerraoui, A. Kermarrec, R. Pires, and R. Sharma. "Epidemic Learning: Boosting Decentralized Learning with Randomized Communication". In: *CoRR* abs/2310.01972 (2023). arXiv: 2310.01972.

[15] X. Miao, Y. Shi, Z. Yang, B. Cui, and Z. Jia. "SDPipe: A Semi-Decentralized Framework for Heterogeneity-aware Pipeline-parallel Training". In: *Proc. VLDB Endow.* 16.9 (2023), pp. 2354–2363.

[16] P. Maymounkov and D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems Workshop IPTPS*. 2002.

[17] H. Garcia-Molina. "Elections in a distributed computing system". In: *IEEE transactions on Computers* 31.01 (1982), pp. 48–59.

[18] D. Ongaro and J. Ousterhout. "In search of an understandable consensus algorithm". In: *USENIX ATC*. 2014.

[19] D. R. Fulkerson. “An Out-of-Kilter Method for Minimal-Cost Flow Problems”. In: *Journal of the Society for Industrial and Applied Mathematics* 9.1 (1961), pp. 18–27. eprint: https://doi.org/10.1137/0109002.

[20] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. “Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning”. In: *Oper. Res.* 37.6 (1989), pp. 865–892.

[21] Wikimedia Foundation. *Wikimedia Downloads*. A 2025.

[22] L. Lamport, R. E. Shostak, and M. C. Pease. “The Byzantine Generals Problem”. In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401.

[23] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. S. E. Ng, and Y. Wang. “GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints”. In: *SOSP*. 2023.

[24] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram. “Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models”. In: *NSDI*. 2022.

[25] P. Sullivan. “As Bitcoin’s Price Surges, Affluent Investors Start to Take a Look”. In: *New York Times* (Jan. 29, 2021).

[26] A. Lihu, J. Du, I. Barjaktarevic, P. Gerzanics, and M. Harvilla. “A Proof of Useful Work for Artificial Intelligence on the Blockchain”. In: *CoRR* abs/2001.09244 (2020). arXiv: 2001.09244.

[27] J. Weng, J. Weng, M. Li, Y. Zhang, and W. Luo. “DeepChain: Auditable and Privacy-Preserving Deep Learning with Blockchain-based Incentive”. In: *IACR Cryptol. ePrint Arch.* (2018), p. 679.

[28] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters. “Pipeline Parallelism for Inference on Heterogeneous Edge Computing”. In: *CoRR* abs/2110.14895 (2021). arXiv: 2110.14895.

[29] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, and P. B. Gibbons. “PipeDream: Fast and Efficient Pipeline Parallel DNN Training”. In: *CoRR* abs/1806.03377 (2018). arXiv: 1806.03377.