Decentralised Training of Large Language Models

Nikolay Blagoev
Supervisor: Lydia Yiyu Chen
Supervisor: Jérémie Decouchant

24th of June 2024

Abstract

Motivated by the emergence of Large Language Models (LLMs) and the importance of democratizing their training, we propose Go With The Flow, the first practical decentralized training framework for LLMs. Differently from existing distributed and federated training frameworks, Go With The Flow enables the collaborative training of an LLM on a set of heterogeneous client nodes that dedicate different resources for an undefined amount of time. Our work addresses node churn, i.e., clients joining or leaving the system, and network instabilities, i.e., network links becoming unstable or unreliable. The core of Go With The Flow is a decentralized flow algorithm that finds the most effective routing to train a maximum number of microbatches with a minimum delay. We extensively evaluate our work on LLama-like and GPT-like models, compare it against the prior art and achieve up to 45% training time reduction in realistic and challenging scenarios of heterogeneous client nodes distributed at 10 different geographic locations with a high node churn rate. We further demonstrate resilient training in such challenging environments, without sacrificing convergence.

Acknowledgement

This thesis would not have been possible without the tremendous help of Dr. Lydia Chen, Dr. Jérémie Decouchant, and Bart Cox. Their invaluable guidance and assistance is the sole reason this thesis can be considered of anything but sub par quality. I have been honoured by their continuous support.

I would also like to extend by sincerest gratitude to my family and my partner, who endured me stressing over these pages for the past few months. Today I stand on the shoulders of those who cared and helped me throughout the years

Contents

1 Research Paper

This section contains our research paper, which we submitted to the Middleware 2024 conference. It contains the main contributions of the thesis.

The rest of this thesis complements the research paper, goes more in depth and provides a detailed motivation for our work, an extended related work section, an additional procedure, and several additional experiments.

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

Abstract

Motivated by the emergence of large language models (LLMs) and the importance of democratizing their training, we propose GWTF, the first practical decentralized training framework for LLMs. Differently from existing distributed and federated training frameworks, GWTF enables the collaborative training of a LLM on a set of heterogeneous client nodes that dedicate different resources for an undefined amount of time. GWTF addresses node churn, i.e., clients joining or leaving the system, and network instabilities, i.e., network links becoming unstable or unreliable. The core of GWTF is a decentralized flow algorithm that finds the most effective routing to train over a maximum number of microbatches with a minimum delay. We extensively evaluate GWTF on LLama-like models, compare it gainst the prior art and achieve up to 45% training time reduction in realistic and challenging scenarios of heterogeneous client nodes distributed at 10 different geographic locations with a high node churn rate.

Keywords: Decentralized learning, Fault-tolerant

ACM Reference Format:

. 2022. Go With The Flow: Fault-Tolerant Decentralized Training of Large Language Models. In ,. ACM, New York, NY, USA, 14 pages. https://doi.org/XXX

1 Introduction

Deep transformer-based architectures [22] have recently enabled unprecedented performance on tasks such as next word prediction [16], language translation [23], and image recognition [8]. These leaps can be explained by the ever growing corpora of available data and by the increasing size of (Large) Language Models, the latter of which has seen an exponential growth [2, 4, 7, 20, 21]. As a consequence, 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), can therefore be used to efficiently train large models on distributed computer nodes. Using these techniques, OpenAI trained their GPT models on the Microsoft Azure cloud platform [2]. However, renting cloud resources or using private computer clusters to train models can easily cost more than tens of thousands of dollars [24], even for smaller models. Developing and training large language models this way is therefore out of the reach for the public.

To democratize language models, a promising alternative to using clouds and private clusters is volunteer computing [19], i.e., using spare computing resources that independent participants around the world dedicate to achieving a common goal. The enabling potential of volunteer computing for the development of large deep networks has been acknowledged in recent publications [17, 18, 24]. Despite being very promising, the decentralized training of language models also implies 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 have only 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, a pioneering work by Ryabinin et al. [1, 17], SWARM, has taken the step towards addressing these issues. However, SWARM does not make full use of the available resources and considers an unrealistic system model. First, it considers an oversimplified scenario of node churn, handling nodes leaving at limited time windows, i.e., during the forward pass of stochastic gradient descent. Second, it uses a simple greedy approach during routing and therefore does not aim at optimising training time. Finally, SWARM assumes that all nodes have the same amount of memory. Our work addresses these limitations.

In this paper, we present Go With The Flow (GWTF)[1], a decentralized and efficient framework for the training of language models. We model the routing of microbatches between 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. The objective of GWTF is to minimize the training time and maximize the throughput by continuously improving the bottleneck stages of LLM training through available heterogeneous nodes. We evaluate GWTF on training LLama-like models of 300 M and 7 G parameters against SWARM and show an up to 40% training time reduction while maximally utilizing the available clients.

[1]Named after the Queens of the Stone Age's song.

Middleware '24, 25th ACM/IFIP International Middleware Conference

ACM ISBN XXX

https://doi.org/XXX

We summarize our key contributions as follows:

- We propose and implement the first kind of crash-tolerant decentralized and collaborative framework, Go With The Flow (GWTF), for training Large Language Models.
- GWTF is a fully decentralized flow-based algorithm, empowering the individuals to contribute their heterogeneous and volatile resources to LLM training.
- GWTF handles node churn at any point of LLM training, i.e., during the forward and backward passes of stochastic gradient descent, by instantaneously rerouting to available nodes thanks to its decentralized nature.
- GWTF is not only able to minimize the training time and maximize the throughput of training Large Language Models in heterogeneous crash-prone environments, but also optimally utilizes the available resources, without sacrificing convergence.

2 Background & Related Work

This section introduces the main concepts behind the distributed training of Large Language Models and presents the relevant related works.

2.1 Background

Transformer block. Large Language Models (LLMs) are typically constructed as a combination of an embedding layer and a series of transformer blocks [4, 16, 20, 21]. The transformer block consists of a multi-head self-attention, normalisation, and a feed-forward network [22]. Self-attention is a special network of three equally sized matrices - the query $\mathbf{W_q}$, the key $\mathbf{W_k}$, and the value $\mathbf{W_v}$. For a given input vector x, self-attention computes

$$y = \text{softmax}\left(\frac{(\mathbf{W_q}x)^T \mathbf{W_k}x}{\sqrt{d_{model}}}\right)(\mathbf{W_v}x)$$

where d_{model} is the size of $\mathbf{W_q}$, $\mathbf{W_k}$ and $\mathbf{W_v}$ [22]. Typically, within a single transformer, multiple groups of three matrices (n_{heads}) are used, and the results of all are concatenated and fed to the feed-forward network. Since multiple matrices (heads) are used, these type of networks are called multi-head self-attention. Parameters d_{model} and n_{heads} characterize the transformer blocks used in LLMs (Table 1).

Pipeline Parallelism. As most models are typically too large to fit on a single GPU, multiple consecutive transformer blocks are conventionally grouped into a number of stages and distributed over multiple devices, forming a pipeline. The optimal number of stages depends on system hyper-parameters, e.g., the available devices, the available memory per device, the model size, the communication cost, etc. The first stage, which is hosted by the nodes that have data, i.e., the data nodes, retrieves the (tokenised) data and embeds it, while the last stage evaluates the loss function, which is also performed by data nodes. Apart from these exceptional functionalities, all stages hold a number of transformer blocks. Together, they behave as a single model. To train the transformer through stochastic gradient decent, 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 gradient updates; and (iii) the update phase, where model weights are updated based on the gradient updates.

Microbatches. As a data node's dataset is usually too large to be fed all at once during training, the training of language models relies on Batched Stochastic Gradient Descent [6]. To fully benefit from pipeline parallelism, each batch is further split into microbatches [11] so that, at a given point in time, multiple microbatches can be concurrently processed using the same parameters. 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. Using a larger batch size allows SGD to converge faster. 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, and consists of distributing the processing of microbatches for a given stage over a set of devices that each host an identical model. These devices perform iterations independently of each others on their microbatches. At the end of each iteration, and before performing the update phase, the devices that maintain the models of a stage aggregate their collective work by exchanging their stored gradients (aggregation phase). Later on, they perform the update phase based on the average of all the gradients they collect.

2.2 Related Work

Gossip Learning. Previous works on decentralised training mainly focused on exploiting data parallelism, and predominantly relied on gossip-based communications [3, 5, 12, 25]. 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. To handle potential data imbalances, the number of microbatches processed locally is used as a weighting coefficient during the model aggregation phase [3]. Gossip learning can naturally tolerate device crashes, dynamic communication graphs, heterogeneous networks (by preferring faster links), non-independent and identically distributed data, and only requires nodes to maintain partial membership knowledge.

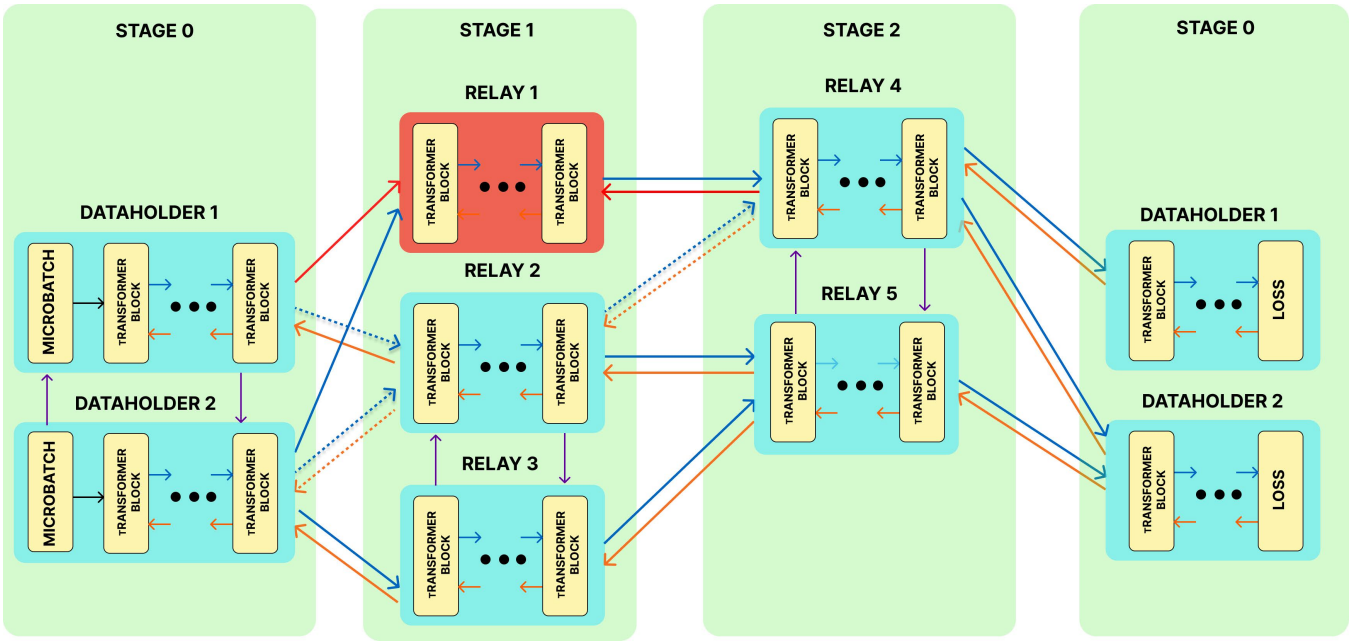


Figure 1. Architecture of a decentralized training of an LLM. The plain blue arrows between nodes indicate the forward passes, the plain orange arrows indicate the backward passes, the plain purple ones indicate model aggregations, and the red arrows indicate some network transmissions that fail because 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.

Recently, de Vos et al. [5] demonstrated that the asymptotic learning rate of gossip learning is $O(\frac{N^3}{k^2})$, where N is the number of participants and k is the number of participants one sends its weights to per round.

Decentralised Pipeline Parallelism. Few works have focused on the decentralised training of Large Language Models (or pipeline parallelism in general). Yuan et al. [24] explain how to identify a communication-optimal arrangement of nodes in pipelines via a centralised and computationally-expensive genetic algorithm. While their work does model the communication heterogeneity of geographically distributed nodes, it does not support computational heterogeneity, node churn, or partially-connected communication graphs. SD-Pipe [15] addresses only the computational heterogeneity between nodes, by allowing nodes to aggregate their parameters with only a subset of nodes in the same stage as them - those who have entered the aggregation phase roughly at the same time. Ryabinin and Gusev [18] require modifying the underlying architecture to tolerate crashes. Petals [1] primarily targets the inference phase and requires the entire pipeline to be rerun in case of node failure.

A promising work is SWARM [17], where nodes independently route a micro-batch through stages. A node sends its activations (outputs of the last layer of its transformer model) to a node servicing the next stage, preferring to sent it to those who will receive it faster. If a node fails to process a microbatch during some predefined time, the previous node will resend its activations to a different node. However, while SWARM forms pipelines on the fly and tolerates a limited number nodes crashing during the forward pass, it does not address crashes that may occur during the backward pass. Indeed, when computing the backwards pass a microbatch needs to travel back through the same nodes it went through during the forward pass. The simple timeout-resend strategy that SWARM uses is not sufficient, as a node selected in the next stage might not have processed a given microbatch in the forward pass. Additionally, nodes in SWARM employ a greedy procedure to select their successor in the next stage, which does not guarantee an optimal pipeline duration. SWARM also adopts a simple load balancing strategy where a node is moved from the most underused stage to the most overused one, and does not take device heterogeneity or memory constraints into account.

Figures 1 and 2 respectively illustrate a system architecture and a corresponding execution scenario where GWTF trains an LLM in a decentralized manner and tolerates the crash of a relay node (relay node 1). In the presence of crashes, training is redistributed across other available nodes. These Figures illustrate the decentralized training of an LLM using 3 stages, where 2 data nodes hold the training data, and 5 relay nodes execute the intermediate training steps for two batches. Relay node 1 crashes when processing the second microbatch of the first data node, and does not execute tasks $F2.1$, $B1.1$ and $B1.2$, which are redirected to relay node 2.

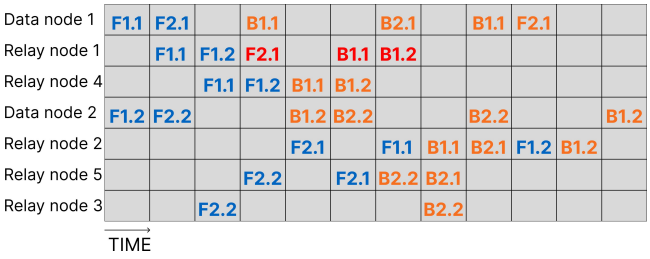


Figure 2. Execution scenario of the decentralized training of an LLM. Notations Fx, y and Bx, y to indicate a forward pass, respectively a backward pass, using microbatch x generated by data node y. Blue is used for a forward pass, orange for a backward pass, and red indicates a task that is not executed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

3 System Model and Objectives

System model. We assume independent nodes, who offer their resources for a period of time, during which they participate in the collaborative training of the model. Nodes are geographically distributed and have individual memory and communication constraints. Each node has a partial view of the global membership, and can directly communicate with the nodes it knows about. Communication links are authenticated, are not necessarily symmetric, may drop messages and are heterogeneous, i.e., they might have different bandwidth and average latency. We assume that the network is partially synchronous, i.e., there are periods of time where the network latency is bounded. Partial synchrony is required because some of GWTF's algorithms require solving consensus, e.g., to affect a training task to a node, which would require partially synchrony for deterministic liveness. We assume that at most 50% of the network can crash. We model a node i with a given memory capacity with a maximum number cap_i of microbatches it can can process during one iteration, and denote by $d_{i,j}$ the processing delay across a node pair (i, j). As discussed in the previous section, we consider two kinds of nodes: data nodes and relay nodes. The former holds training data, whereas the latter contribute to the training of the LLM. It is possible for a node to be both a data and a relay node.

Node churn. Both types of nodes can join the system at any time, participate for a while in the training process, and then leave the system. Nodes can crash or gracefully leave the system. We handle these two cases similarly using timeouts. Nodes can leave the system both during the forward and the backward passes. A node crashing during a forward pass simply slows down the construction of the forward pass. A node crashing during a backward pass has more severe consequences, since it requires additional synchronization between nodes across different stages in order to recover the pipeline by replacing the missing nodes.

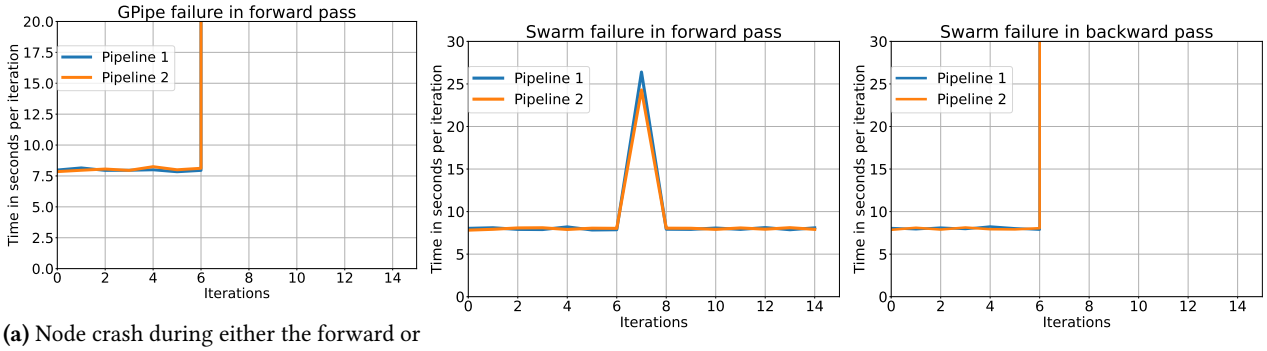
Objectives. We aim to train an LLM efficiently in a decentralized setting, and in particular, to tolerate the following important requirements: nodes and links can be heterogeneous, nodes can leave or join the system at any time, and the network can be unstable. Previous works [11, 15, 24] predominantly assume that all nodes have homogeneous computing resources, remain in the system for the whole training process, and that the network is stable. To illustrate these limitations, we evaluated the impact of a single node crash during either the forward or the backward phases on the training of a GPT-like model with $d_{model} = 512$ and $n_{heads} = 16$ with 2 data nodes, 3 stages per pipeline and 2 relay nodes per stage. Figure 3 summarizes the results we obtained with a standard distributed pipeline-parallel setup (such as the one used by Gpipe [11] or Yuan et al. [24]) and with SWARM [17]. The crash of a single node prevents the execution of the aggregation phase for GPipe, SWARM is able to reroute a microbatch during a forward pass after a timeout, but does not specify recovery mechanisms if a crash happens in a backward pass.

4 Overview of GWTF

Our decentralized training framework for large language models, GWTF, effectively utilises heterogeneous client nodes and tolerates churn. This section highlights the cost minimization objective of GWTF, its key components, and the way nodes synchronize themselves.

4.1 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, i.e., cost, of establishing a microbatch flow between two nodes i and j using the sum of its computation times and communication delays. Flow is thus an abstraction of the end-to-end processing of a microbatch through a set of nodes in the stages. Following Yuan et al. [24], the communication cost between nodes i and j is composed of a network latency ($\lambda_{i,j}$) and a transmission delay that is computed as the amount of transferred data ($size$) divided by the network bandwidth ($\frac{size}{\beta_{i,j}}$). The data that is transferred between two nodes depends on the different training phases. During a forward pass, it is the activations of the last layer of a node. In a backward pass the transferred data is the gradient of the loss with respect to said activations, and during aggregation it is the model parameters that each node holds. We assume that links are not necessarily symmetric, $\lambda_{i,j}$ 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



(a) Node crash during either the forward or backward pass with a traditional pipeline-parallel training [24]

(b) Node crash during the forward pass with SWARM [17]

(c) Node crash during the backward pass with SWARM [17]

Figure 3. Impact of a node crash during the forward or backward passes on the duration on an iteration for a single batch in 2 pipelines (one per data node).

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, or to average received models during the aggregation step. 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 possible amount of microbatches at the lowest global cost, and uses a novel decentralized algorithm to route each microbatch through the participants (cf. Section 5.2).

4.2 Key Design Components

A relay node can execute 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 aggregation phase.

The first component, our decentralized flow cost minimization algorithm (Sec. 5.2), minimizes the flow cost of the forward and backward pass for each microbatch initialized by the data holders. As the flow is composed of multiple stages, GWTF determines in a decentralized way their execution on the heterogeneous relay nodes through a request procedure that participants execute independently. Furthermore, we leverage the fact that the throughput of the flow is limited by the slowest stage and use this fact when adding new nodes to the system. Our decentralised flow algorithm aims at sending a maximum number of batches at the smallest possible cost according to Equation 1. It allows for nodes with different capacities to route independently, without global knowledge, microbatches in a cost-effective manner.

The second component assigns nodes to stages when they join the system (Sec. 5.1). In this component, 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.

The third and fourth components handle nodes leaving due to unavailability or crashes (Sec. 5.3), 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. It is more difficult to efficiently resolve faults occurring during a backward pass, as it requires synchronisation between multiple nodes in different stages. To this end we associate each microbatch with a path list, which holds the nodes it has passed through when being processed. During a fault in a backwards pass this 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 other relay nodes that hold the intermediate information needed and have spare computational capacity.

Finally, the fifth component allows nodes in a given stage to aggregate their models in a weighted manner using the number of microbatches they have processed. Relay nodes wait to receive the models of the other relay nodes in the same stage that they know of before aggregating them. If a node is detected to have crashed then its model might not be used in the aggregation phase.

5 Algorithms and Implementation of GWTF

This section first explains how GWTF handles node joining, and allows nodes to discover each other. It then 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. Finally, it discusses how GWTF tolerates nodes crashing or leaving the system.

Table 1. Notations

Name	Description
S	A single stage - a set of nodes
N	All nodes/peers
k	Aggregation group size
T	Temperature
α	Cooling factor
$d(i,j)$	Cost between nodes i and j
$f(i,j)$	Flow between nodes i and j
bf	Bottleneck factor
F	Outgoing flow from all data nodes
$cost_f$	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
d_{model}	Size of self-attention matrices
n_{heads}	Number of heads in multi-head self-attention

5.1 Inserting Joining Nodes

We design a two-way procedure to assign a new node to the stage that is predicted to be the bottlenecked one. First, the joining node estimates the stage with the highest utilized ratio, meaning the number of flows divided by the available capacity at the stage. Then, the joining node sends a proposal to an elected data node with its capacity, the stage it wishes to join, and estimated costs to other nodes in that stage. The data node decides to accept the proposal or not based on the estimated cost improvement.

Proposal to join a stage. When a new (relay) node wants to participate in the training, it first needs to discover other nodes in the system through a Distributed Hash Table (DHT) [14]. It queries existing nodes for the following information: their stage, microbatch flow capacity, and existing flows. Afterward, it computes the bottleneck factor for each stage - the ratio of its utilisation. We first define the bottleneck factor (bf) of a stage s as the number of total microbatch flows divided by the entire capacity of a single stage. The closer the bottleneck factor of a stage is to 1, the more it is a bottleneck. New nodes should join the stages with the highest value. Once nodes decide on a stage to join, they need to estimate two values. The first is the maximum cost of data parallel communication with a peer in the same stage. This value can be obtained through $d(i,j)$, where size refers to the size of the model in that stage. The second is the maximum communication of activations with any node from either the previous or subsequent stage (termed as the expected pipeline parallel cost). These are used to estimate the communication cost associated with using this new node in the training process.

Selecting the new nodes. Upon reception of such an offer, data nodes rebroadcast it to all known other data nodes and agree on which nodes' proposal to accept at a given stage. The key criterion is to choose a node's joining proposal that maximizes the ratio of the cost of microbatch flow divided by the overall microbatch throughput.

As the throughput and cost are dominated by the bottlenecked stage, we thus estimate the cost and throughput by the bottlenecked stage. Specifically, the leader node identifies the stage, x, with the highest bf and the next stage, z, having the second highest bf. If multiple stages have the same bf, then those closer to the first stage are preferred for node addition first.

The leader data node first computes the updated throughput for accepting new nodes into the stage x, which is determined by the stage of the smallest capacity. Then the leader computes the updated microbatch cost, which is estimated as (i) the longest microbatch flow after including the new node and (ii) the worst intra stage communication time that is required to aggregate the results of nodes within a stage. As the actual maximum flow cost depends on the actual routing, we can only estimate the overall flow cost by replacing the average cost per stage by the worst cross stage communication cost of adding the new node.

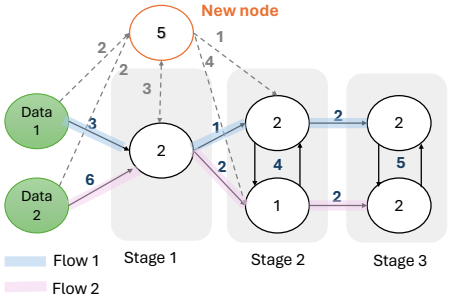


Figure 4. Example of adding a new node into stage 1 in the system that has two data nodes and three relay nodes. Circles represents the capacity of a node in terms of flows and edges represent the communication cost.

Running example We use a simple example to illustrate the computation of updated throughput and microbatch flow cost shown in Fig 4. There are 2 dataholder nodes, each sending single microbatch flow traversing in 4 stages. The circle in each stage denotes the node and the value denotes the capacity. Stages 1, 2 and 3 have a total capacity of hosting 2, 3 and 5 microbatch flows. The existing cost of the two flows are 6 (top) and 10 (bottom).

A new node of capacity 5 proposes to join stage 1 - the bottlenecked stage. It incurs the following costs: (i) a cost of 3 in exchanging parameters with the node in stage 1; (ii) a cost of 2 in exchanging the activations of both dataholders; and (iii) two costs of 1 and 4 to exchange the activations of top and bottom node in stage 2, respectively. As a result, it also sends the estimated additional communication cost within a stage, which is 3, and the worst case additional cost to the overall flow, which is 4, i.e., the maximum of all additional communications across parent and children.

Upon receiving the proposal from this new node, the leader data node starts estimating the new throughput and the cost of including this new node. We note that multiple nodes can propose to join the same stage. The new throughout of stage 2 is thus 3, as the first stage is expanded to the capacity of $7 = 5 + 2$. It then estimates the new flow cost, which sums the maximum flow cost and the worst communication cost within the stage 3 that is still 5. The maximum flow cost is to replace the average per stage flow time by the worst flow time of introducing the new node, i.e., $10 - 10/4 + \max\{2, 2, 1, 4\}$. As a result, one can easily compute the new ratio of the flow cost divided throughput decreases.

5.2 Decentralized flow optimization

GWTF constructs the pipelines for a microbatch starting from a data node and sequentially determining the relay nodes for each stage. To build a pipeline, 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.

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. In such a problem the goal is to minimise the sum of costs of all flows within the graph:

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

where $f(i, j)$ is the flow between two nodes and $d(i, j)$ is the cost of the link that connects them. 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 [10] 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 (which in the training are the data nodes). However, the objective function of Equation 2 requires greater synchronisation between nodes and converges slowly, as nodes need global knowledge of the cost of the entire flow. Fortunately we can solve a much easier problem with only local knowledge for each node by minimising the cost of the maximal flow between two nodes:

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

A similar function has also been used in a previous work [24], which however assumed a constant flow.

GWTF uses 5 subprocedures to solve this decentralized optimization problem: Request Flow, Request Change, Request Redirect, Pushback, and Cancel. These procedures only assume local knowledge of the system's membership, and of the outflow and inflow of known nodes. The outflow and inflow of a node are respectively the flows it transfers to subsequent peers and the flow it receives from previous peers. By requesting and receiving agreement for both outflows and inflows, a node can be sure that there is no duplicated flow. All flows are associated with a unique id and their data node - the final intended destination of the flow. We call unpaired inflows/outflows the inflows/outflows of a node that have not yet been paired with any outflow/inflow (i.e., attributed to a node from another stage). All data nodes are initialized with unpaired inflows and outflows that are equal to their respective capacities. Nodes inform each other of the cost of flow between them, which captures changing network conditions during training.

Building Pipelines using Flow Requests. During each iteration, nodes that do not have unpaired outflow look for a node in a subsequent stage that has 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 and 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. In both cases, 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 minimising 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 will reduce the objective function (e.g., maximum sending cost), and consequently request a switch. 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. Otherwise, nodes keep their outflows as they are.

Request Redirect. If a node's peers have flow from a previous stage peer to a next stage peer and the node checking has capacity left, it checks if that flow will have a lower cost if it came through it. If it is the case, it then sends a Request Redirect to its peer. The other node in the stage performs the same check as in a Request Change. If it is approved, it swaps its outflow with the one of the requesting node and tells its predecessor peer that it cancels its flow (i.e., pushes back its flow, as explained in the next paragraph).

In order to prevent the algorithm from converging to a local minima, we utilise a technique from simulated annealing [13]. Both procedures can be approved even if they do not minimise with the following probability

$$e^{\frac{cost_{current} - cost_{new}}{T}} > U(0,1),$$

where T is the current temperature. Upon a successful redirect/change, temperature is decreased to $T \cdot \alpha$, where α is the cooling factor.

Pushback and Cancel. Pushing back a flow and cancelling a flow work identically, except that one is upwards in the direction of the flow and the other downwards. If a node has unpaired inflow/outflow (one not matched to some outflow/inflow) and for a predefined number of iterations (we use 7) it cannot find a peer to send it to, it pushes it back to the previous peer, respectively cancels it from the subsequent peer. The first of the procedures pushes flow down the stages - a node with no unpaired outflow requests to push its flow to some node with unpaired outflow. The next two procedures minimise the cost of a flow, by switching it between nodes in a given stage. After a successful execution of either one, the flow of one of the nodes now goes through the other.

5.3 Tolerating Crashes

When a node completes the computations of a batch, it sends back to the previous node in the path a COMPLETE message with the id of the batch. Based on this message, nodes can estimate the communication cost (network and computation latency) between them, though with one extra delay term, as what they will measure is $2 * \lambda + \frac{size}{\beta} + computation$. When sending training information, either during a forward or a backward pass, nodes expect to receive a COMPLETE reply from their peer within some fixed predefined amount of time. If they do not receive a reply, then the node assumes that the peer has left the system or is unresponsive and diverts its traffic to a different peer. If this is not possible, for example because it does not know any peers who can take in the traffic, it should send a DENY message to its upstream peer and let it redistribute the flow. This last operation can continue recursively until the source, which will have to leave this batch for the next iteration.

DENY messages essentially signal to an upstream peer that the subsequent node cannot process any more microbatches. This can happen because it doesn't know any nodes that it can send to, it has received DENY messages from every downstream peer or it has reached its maximum capacity. Nodes who have sent a DENY message are excluded from consideration until they send a CAN TAKE message (described in Section 5.4).

Forward pass. Crash recovery during the forward pass of relay nodes is done as in SWARM [17]. 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 the flow algorithm of Section 5.2, which can optimise costs of sending while also taking into account the memory constraints of each node.

Backward pass. Nodes also send a complete message when they are done with their computation. If a node does not receive a COMPLETE BACKWARDS message or the transmission failed, it should inform the data node. If the data node has no outstanding microbatches for which they have not received a backwards pass, they do not need to restore the pipeline. In this case the node that detected the crash will not send the gradient of the activations, but will use the computed gradient to update its parameters during the aggregation phase. Otherwise, the data node pings the node it sent the activations to with the microbatch path, which it knows as it had to compute to loss and it was the last one in the path. Nodes ping each other along the path. Upon a failed ping, the node that did not receive a ping forwards its activation to a new downstream peer with the path information. The downstream peer, upon calculating the activations on that layer, attempts to send the results to some node that was already on that path. If it does not receive a COMPLETE reply, it tries again to divert the traffic, until either reaching a node which was on the path and is still alive or reaching the data node, which is still technically a node on the path. Then the backwards pass can be resumed. Note that if a node earlier on from the original pipeline is found

to be alive, then the process is significantly cheaper than recomputing the pipeline for that microbatch from scratch, which previous works would have to do.

If a node receives the same batch twice, before processing a backward pass, it can assume that one of the peers has recovered along the original path. This situation can happen, for example, when a node receives a COMPLETE message after it has timed out for a given microbatch. Thus now at least two nodes have computed activations for the same batch and they also remember the activations they have received. These can be freed (forgotten) by all but the first peer who sent the duplicate microbatch, to allow for more data through. The node that receives duplicate activations tell previous nodes whether they should forget their activations.

5.4 Training-Aggregation Synchronization

As we have seen, when a node wants to join the system it runs the procedure of Section 5.1. Once it has been approved to join, it begins running the flow algorithm of Section 5.2 in parallel to the actual training. When a fault occurs it runs one of the two procedures of Section 5.3. In a decentralized training system, nodes also have to alternate between training and aggregation phases, which has been insufficiently described in previous works [17] and in previous sections. This synchronisation 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 can signal to each other when this transition should happen.

When a node has reached its capacity of microbatches processed and has computed the backwards pass on all of them, it sends its parameters to other nodes in the same stage as it, as per [3, 5, 12], and enters the aggregation phase. Otherwise, the aggregation phase can also begin when the elected data node leader announces a starting aggregation to all its peers (using a BEGIN_AGGREGATION message), which propagate it further through the network. Upon receiving this message, nodes broadcast their model weights within their stage and collect their peers' weights. Once a node has finished its aggregation phase and knows of a downstream peer that also completed theirs, then it sends to its upstream peers another message (CAN_TAKE) that indicates that it accepts new microbatches. Nodes from the last stage do not need to wait for a CAN_TAKE message before sending their own. Thus the whole system involves several passes: a back to front formation of pipelines, a front to back forward pass, a back to front backwards pass, a front to back relaying of BEGIN_AGGREGATION messages, and finally a back to front relaying of CAN_TAKE messages, which signals that a node has finished its aggregation phase and can begin the next iteration. This communication pattern is illustrated in Figure 5.

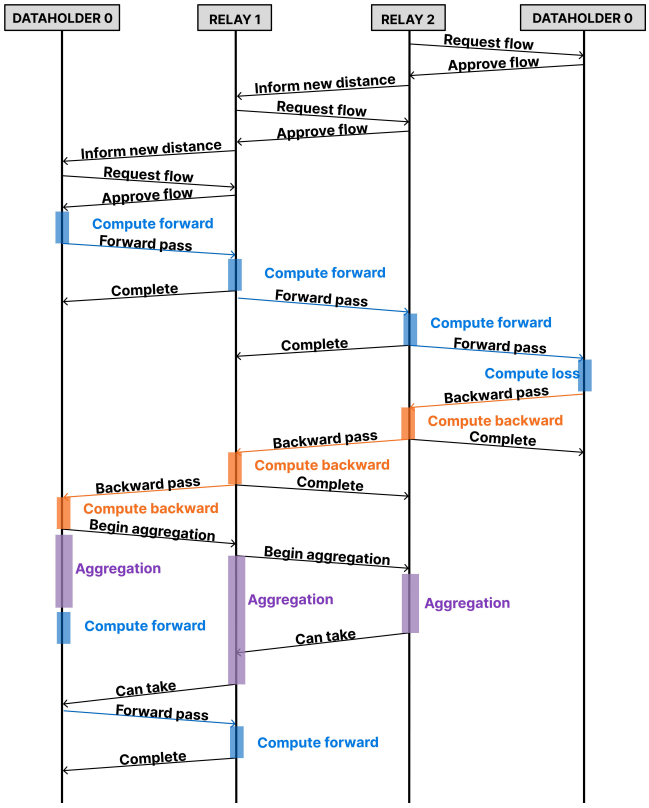


Figure 5. Communication and training happening during one iteration in a simple 3-stage pipeline without any crashes and with 1 microbatch per iteration. For ease of visualisation messages that relate to the aggregation phase are omitted.

6 Performance Evaluation

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 show that our system is up to 45% faster in end-to-end runtime in the presence of faults compared to previous works. We further compare the performance of our scheduler against the optimal communication scheduler of Yuan et al.'s DT-FM [24]. We finally verify the applicability of our system in a practical scenario by showing that the model converges at rate similar of the one of GPipe [11].

6.1 Setup

The following experiments were performed on LLama architecture models with varying model sizes. We use a private cluster of 5 NVIDIA RTX A4000 GPUs with 16 GB of GPU memory that are interconnected by a 100 GB Infiniband connection. We simulate geo-distributed locations by limiting the bandwidth and increasing the latency between nodes, with a maximum bandwidth between two nodes in simulated different locations of 500Mb/s. To achieve a higher throughput per node, we make use of memory offloading

by storing the computation graph and received model parameters/gradients to RAM. Unless stated otherwise, our decentralized optimization uses $T = 1.7$ for the initial temperature, $\alpha = 0.95$ for the cooling factor, and the aggregation group size was set to $k = \infty$.

6.2 Node Crashes

We evaluate the performance of our system against SWARM [17] on a small LLaMa-like model with $d_{model} = 1024$, $n_{heads} = 18$ and 16 layers. Microbatches used were of size 4 and sequence length of 512. As the model and batch sizes are significantly smaller than those in practice, we compensate by decreasing the bandwidth between nodes by a factor of 250, which is equivalent to sending an activation 250 times larger. This is necessary as the sequence length is 8 times smaller ([21] trains on a lenght of 4096), the activations sizes were roughly 12 times smaller than for actual LLaMa models of size 13b, and microbatches usually include more examples. Each setting has a world size of 18 nodes (maximum nodes that can be active at the same time). The model was split across 6 stages each servicing three transformer blocks, except the very first one which serviced 1 transformer block, the embedding layer, and handled loss computation and data retrieval. Two data nodes participate in the setting and do not leave the system until the end. They each try to push 4 microbatches during each iteration.

We compare the performance against SWARM [17], the state of the art in decentralized crash tolerant training. For a fair comparison, we modified SWARM to allow node crashes and joining of nodes. The SWARM baseline includes an additional timeout if no backwards pass has been received in some predefined amount of time by a data node. This triggers SWARM to rewire the nodes stochastically. Additionally, nodes can join freely in any stage they choose.

Parameters vary between experiments. The first difference is capacity. In the heterogeneous setting, relay nodes have random capacities between 1 and 3. In the homogeneous case all nodes have a capacity of 4. The second parameter is the join-leave chance. It dictates each node's chance to crash or join during any iteration. At 0%, nodes do not leave the system. At 10% they have a ten percent chance to join the system if currently inactive, or ten percent chance to crash during any iteration, if currently active. Results are presented in Table 2. Several metrics are reported (averaged across all iterations):

- Time per microbatch - the maximum time per iteration, measured from the point of view of a dataholder, divided by the number of microbatches processed in that iteration.
- Throughput per iteration - the number of microbatches processed in an iteration.
- Microbatch per time unit - the inverse of time per microbatch.
- Communication time - the summed pipeline parallel communication time for all microbatches in an iteration in minutes.
- Wasted GPU time - the summed pipeline parallel computation time in minutes for all microbatches in an iteration number, which was not included in the aggregation step or was not on the main path of a microbatch.

In all heterogeneous cases we demonstrated an improvement over SWARM and better performance even in the presence of crashes (speed up of 45% in the 10% crash case). In the homogeneous case GWTF outperforms SWARM in the presence of crashes and performs similarly in the fault-free case. SWARM also has a much higher GPU wasted time, as microbatches may be sent to nodes that do not know anyone who they can send to, or on backward pass faults, the entire pipeline needs to be recomputed.

6.3 Optimality

In this section we aim to evaluate the performance of our scheduling routines, by comparing the end-to-end training time of our system against a GPipe setting with a communication optimal arrangement, calculated by the procedure DT-FM designed by Yuan et al. [24]. The setting of the experiment is the same as the 0% homogeneous of the previous experiments, with some differences. Following the settings of [24] 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. The end-to-end training time between the two systems is compared in Table 3. Although the optimal computation schedule does outperform GWTF by almost 13%, it takes much longer time to be computed, as it involves the use of a genetic algorithm [24]. As it Scales exponentially with the number of nodes. Not sustanable for more systems or systems that change ad-hoc.

6.4 Scalability with Model Size

Additionally, we evaluated the training loss of our system in a 10% heterogeneous setting against a single GPipe pipeline of 8 nodes, with 8 microbatches of size 1 and sequence length 4096. We evaluated against a setting of 10% node crashes with a maximum world size of 10 and 1 data node. The model used in this experiment is the LLaMA-7b with $d_{model} = 4096$, $n_{heads} = 32$, and 32 layers, distributed uniformly across 8 stages. The dataset used was the Wikipedia English dataset [9]. Figure 6 shows that the larger system deployed by GWTF has a similar convergence of loss as the centralised GPipe pipeline.

6.5 Decentralized Minimum Cost Flow

We evaluate our flow algorithm in 6 different settings. The first four settings involve a single source-sink node while the last have multiple source-sink nodes. Details about the

Table 2. Performance with crash-prone devices. Results are over 25 iterations. Per column the mean of the iterations is presented followed by the standard deviation

	Homogeneous 0%		Homogeneous 10%		Homogeneous 20%	
	SWARM	GWTF (Ours)	SWARM	GWTF (Ours)	SWARM	GWTF (Ours)
Time per microbatch (min)	**0.53 ± 0.13**	0.58 ± 0.16	1.26 ± 0.87	**1.01 ± 0.37**	1.76 ± 1.3	**1.17 ± 0.43**
Throughput (# microbatch/iteration)	**8.0 ± 0.13**	7.16 ± 0.17	4.64 ± 0.87	**5.8 ± 0.37**	**6.1 ± 1.3**	5.72 ± 0.43
Microbatch per time	**1.95 ± 0.26**	1.8 ± 0.36	**1.14 ± 0.53**	**1.14 ± 0.45**	0.85 ± 0.43	**1.03 ± 0.5**
Communication time	6.07 ± 4.92	**4.2 ± 2.52**	12.23 ± 8.81	**7.76 ± 2.24**	17.74±13.15	**4.57 ± 2.68**
Wasted GPU time	0.27 ± 0.0	**0.03 ± 0.0**	0.75 ± 0.0	**0.2 ± 0.0**	1.75 ± 0.0	**0.0 ± 0.0**
	Heterogeneous 0%		Heterogeneous 10%		Heterogeneous 20%	
	SWARM	GWTF (Ours)	SWARM	GWTF (Ours)	SWARM	GWTF (Ours)
Time per microbatch (min)	1.59 ± 0.21	**1.15 ± 0.44**	4.53 ± 4.08	**2.45 ± 0.93**	5.36 ± 3.56	**3.47 ± 2.06**
Throughput (# microbatch/iteration)	2.96 ± 0.21	**3.6 ± 0.44**	1.56 ± 4.08	**2.08 ± 0.93**	1.72 ± 3.56	**2.12 ± 2.06**
Microbatch per time	0.64 ± 0.09	**0.96 ± 0.26**	0.36 ± 0.21	**0.47 ± 0.16**	0.3 ± 0.2	**0.4 ± 0.22**
Communication time	10.55 ± 2.79	**3.65 ± 1.19**	3.95 ± 1.8	**2.62 ± 1.27**	7.18 ± 7.22	**4.28 ± 2.85**
Wasted GPU time	0.57 ± 0.0	**0.0 ± 0.0**	0.33 ± 0.0	**0.1 ± 0.0**	0.33 ± 0.0	**0.03 ± 0.0**

Table 3. Comparison against optimal schedule

	Time per microbatch	Throughput per iteration	Microbatch per time
DT-FM [24]	0.44 ± 0.0	9.0 ± 0.0	2.27 ± 0.010
GWTF	0.51 ± 0.08	8.08 ± 0.08	1.99 ± 0.26

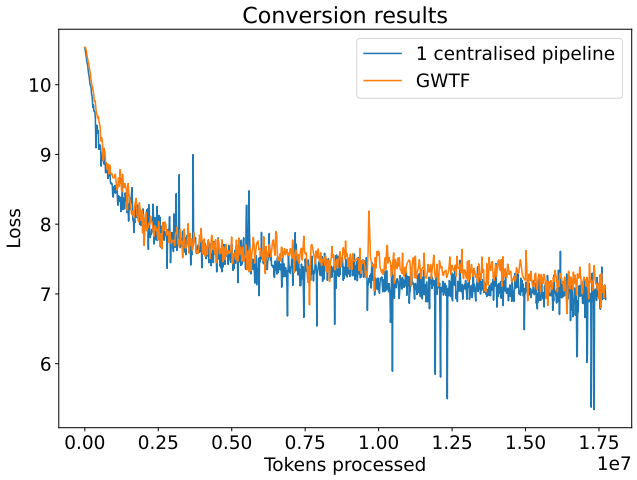


Figure 6. Loss convergence of GWTF compared to the convergence of a single centralised GPipe pipeline

Table 4. 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	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$

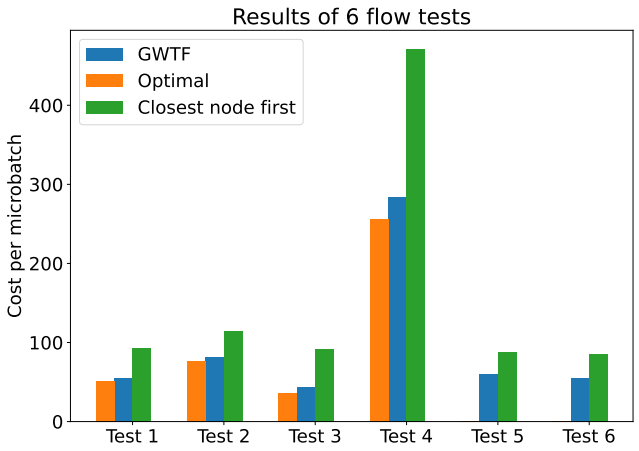


Figure 7. Average cost per microbatch in flow tests.

tests are presented in Table 4. In all cases the source-sinks were given sufficient capacity to prevent bottlenecks. In order to compare to the optimal result as found by Fulkerson's algorithm [10], here our procedure attempts to minimise $min\left(\sum_{i,j\in \mathbf{N}} d(i,j) * f(i,j)\right)$, i.e. minimise the sum of costs of all flows. Tests 5 and 6 are not compared with the optimal baseline, as there the formulation differs, in that that a source must deliver to a specific sink. We use the approach from SWARM [17] of sending to the next stage closest node as a baseline. The results of the first two tests achieved difference from the optimal flow cost are presented in Figure 7 as fractions of the optimal cost.

Table 5. Node Addition Test Settings. Note that 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	$\lfloor U(1,20) \rfloor$	$\lfloor U(1,100) \rfloor$	$\phi + \lfloor U(50,100) \rfloor$

6.6 Handling Joining Nodes

We evaluate the algorithm for joining nodes, which is described in section 5.1. GWTF is evaluated 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 5. To construct the system, we use the Out-of-kilter algorithm [10] to determine the minimum cost flow. Iteratively, 20 nodes are added in different stages following the algorithm in section 5.1. Since every node is a candidate for 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 [10] for each combination of S candidate nodes added to each of the S stages in the tests.

The results are presented in Figure 8 where the average improvement of 10 runs for each test is measured as $\frac{cost_{now} - cost_{after}}{cost_{now}}$ (i.e., the proportional improvement compared to before running the respective algorithm). We provide 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, GWTF 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

7 Conclusion

Motivated by the importance of democratizing the training of LLM on heterogeneous and churning clients, we propose

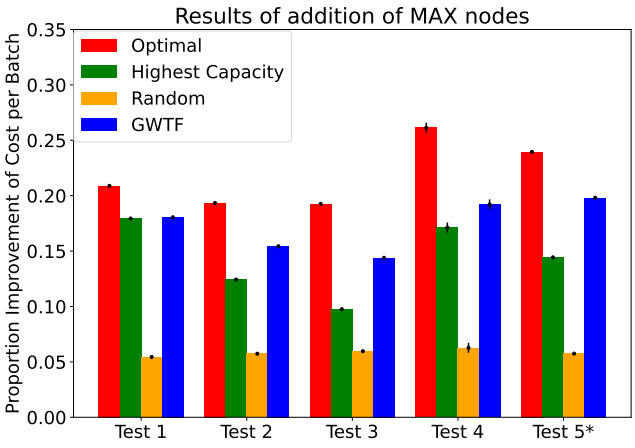


Figure 8. Average improvement of 10 runs for the new node addition tests. The variance is reported with black lines per bar. Higher means better.

GWTF, the first fault tolerant decentralized training framework aiming to minimizing the training cost/time and maximizing the throughput. GWTF uniquely models the forward and backward pass of per training microbatch per iteration as a flow, and effectively decide its execution on clients that are of different computation and communication capacities. GWTF is architected in three key modules, namely decentralized flow, incorporating joining nodes, and tolerating crashes. We extensively evaluate GWTF on decentralized training LLama-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 resulting into almost zero GPU wasted time of any joining clients.

References

[1] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2023. Petals: Collaborative Inference and Fine-tuning of Large Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics: System Demonstrations, ACL 2023, Toronto, Canada, July 10-12, 2023*, Danushka Bollegala, Ruihong Huang, and Alan Ritter (Eds.). Association for Computational Linguistics, 558–568. https://doi.org/10.18653/V1/2023.ACL-DEMO.54

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle,

Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[3] Marc Casas, Wilfried N. Gansterer, and Elias Wimmer. 2019. Resilient gossip-inspired all-reduce algorithms for high-performance computing: Potential, limitations, and open questions. *Int. J. High Perform. Comput. Appl.* 33, 2 (2019). https://doi.org/10.1177/1094342018762531

[4] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2023. PaLM: Scaling Language Modeling with Pathways. *J. Mach. Learn. Res.* 24 (2023), 240:1–240:113. http://jmlr.org/papers/v24/22-1144.html

[5] Martijn de Vos, Sadegh Farhadkhani, Rachid Guerraoui, Anne-Marie Kermarrec, Rafael Pires, and Rishi Sharma. 2023. Epidemic Learning: Boosting Decentralized Learning with Randomized Communication. *CoRR* abs/2310.01972 (2023). https://doi.org/10.48550/ARXIV.2310.01972 arXiv:2310.01972

[6] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. 2012. Optimal Distributed Online Prediction Using Mini-Batches. *J. Mach. Learn. Res.* 13 (2012), 165–202. https://doi.org/10.5555/2503308.2188391

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[8] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=YicbFdNTTy

[9] Wikimedia Foundation. [n. d.]. *Wikimedia Downloads*. https://dumps.wikimedia.org

[10] D. R. Fulkerson. 1961. An Out-of-Kilter Method for Minimal-Cost Flow Problems. *J. Soc. Indust. Appl. Math.* 9, 1 (1961), 18–27. https://doi.org/10.1137/0109002 arXiv:https://doi.org/10.1137/0109002

[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 103–112. https://proceedings.neurips.cc/paper/2019/hash/093f65e080a295f8076b1c5722a46aa2-Abstract.html

[12] Jiyan Jiang, Wenpeng Zhang, Jinjie Gu, and Wenwu Zhu. 2021. Asynchronous Decentralized Online Learning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (Eds.). 20185–20196. https://proceedings.neurips.cc/paper/2021/hash/a8e864d04c95572d1aece099af852d0a-Abstract.html

[13] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. 1989. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Oper. Res.* 37, 6 (1989), 865–892. https://doi.org/10.1287/OPRE.37.6.865

[14] Petar Maymounkov and David Mazières. 2002. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers (Lecture Notes in Computer Science, Vol. 2429)*, Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron (Eds.). Springer, 53–65. https://doi.org/10.1007/3-540-45748-8_5

[15] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. 2023. SDPipe: A Semi-Decentralized Framework for Heterogeneity-aware Pipeline-parallel Training. *Proc. VLDB Endow.* 16, 9 (2023), 2354–2363. https://doi.org/10.14778/3598581.3598604

[16] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[17] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. 2023. SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 29416–29440. https://proceedings.mlr.press/v202/ryabinin23a.html

[18] Max Ryabinin and Anton Gusev. 2020. Towards Crowdsourced Training of Large Neural Networks using Decentralized Mixture-of-Experts. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/25ddc0f8c9d3e22e03d3076f98d83cb2-Abstract.html

[19] Michael Shirts and Vijay S. Pande. 2000. Screen Savers of the World Unite! *Science* 290, 5498 (2000), 1903–1904. https://doi.org/10.1126/science.290.5498.1903 arXiv:https://www.science.org/doi/pdf/10.1126/science.290.5498.1903

[20] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[21] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023). https://doi.org/10.48550/ARXIV.2302.13971 arXiv:2302.13971

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/

3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[23] David Vilar, Markus Freitag, Colin Cherry, Jiaming Luo, Viresh Ratnakar, and George F. Foster. 2023. Prompting PaLM for Translation: Assessing Strategies and Performance. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 15406–15427. https://doi.org/10.18653/V1/2023.ACL-LONG.859

[24] Binhang Yuan, Yongjun He, Jared Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Ré, and Ce Zhang. 2022. Decentralized Training of Foundation Models in Heterogeneous Environments. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2022/hash/a37d615b61f999a5fa276adb14643476-Abstract-Conference.html

[25] Pan Zhou, Qian Lin, Dumitrel Loghin, Beng Chin Ooi, Yuncheng Wu, and Hongfang Yu. 2021. Communication-efficient Decentralized Machine Learning over Heterogeneous Networks. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 384–395. https://doi.org/10.1109/ICDE51399.2021.00040

2 Introduction

2.1 A Brief History of Natural Language Processing

Natural Language Processing (NLP) is the field of Computer Science focused on developing solutions that can understand and produce human languages, whether textual or verbal. First successful approaches in the late 60s like SHRDLU [78] (a reference to the "etaoin shrdlu" character string used to mark erroneous lines in the days of hot metal typesetting) employed limited vocabulary to encode in a program understanding of syntax, morphology, vocabulary, and target domain. Hand-crafted heuristic algorithms continued to be employed until the 80s for lemmatisation (producing the lemma of each word), often through methods like stemming [45], Part of Speech tagging [28], etc. With the diminishing influence of Chomsky's work and the increase of computational power, the 90s saw a transition from the "grammar-based" approaches, these handcrafted long and complex rules for understanding language based on linguistic principles, to "statistical-based" ones, those employing Machine Learning to learn an underlying distribution [38]. The field progressed alongside the development of Deep Learning, replacing the hidden markov models and Bag of Words for Long-Short Term Memory Networks [31] and Vector Embeddings [49]. Then everything changed with the seminal work of [72], which introduced the Transformer architecture. Employing self-attention, a function computing the importance of each word relative to all the other ones around it, it performed much better than state of the art at a significantly lower training time. Since then, most successful Language Models have been based on the Transformer as their building block [70, 8, 14, 17, 65].

2.2 Size Matters

Since the work of [72] Language Models have grown at an exponential rate - 100 million parameters for GPT1 in 2018 [58], 340 million for BERT in 2019 [17], 8.3 billion for Megatron LM [65], 175 billion for GPT-3 in 2021 [8], and 540 billion for PaLM in 2022 [14]. The bigger the model got, the better its performance was. Models became so large that they no longer fit even on the most high-end GPUs. But simply making your model bigger is not enough to increase its performance, as it can lead to overfitting if the training data set is too small in proportion to the number of parameters [80]. Hence, the Large Language Model (LLM) revolution was in part facilitated by the ever expanding corpus of text data available in digital form, whether from forums, books, chats, etc. For example the Colossal Clean Crawl Corpus contains over 360 million strings in its training set and is over 800 Gigabytes in size, just for the English language alone [59].

So you have models that do not fit in your GPU and datasets that are order of magnitude larger than previously employed ones. How can you then train a model?

2.3 Distributed Machine Learning

Distributed training is the workhorse of state of the art Deep Learning solutions. Of interest here are two main approaches - model (pipeline) parallel and data parallel.

Model parallel solves the first problem encountered - too many parameters to train on a single GPU. So lets split the model layers across devices, which will then communicate with each other the activations in the forward pass and the output gradient in the backwards one. LLMs are particularly well suited for this technique, as they have a nice split boundary between transformer blocks, where the model can be separated between devices.

Data parallel aims to increase the batch size during training, by splitting it into minibatches, each ran on a device holding an identical model to the rest. Then the gradients of all devices

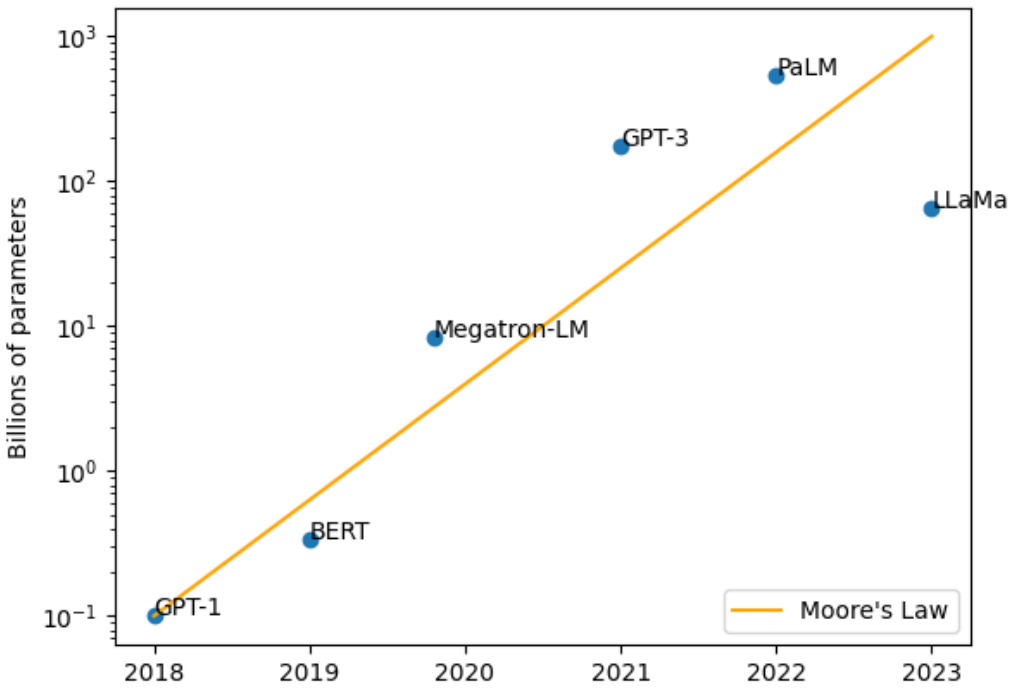


Figure 1: Model size comparison

are aggregated in some manner and used to update the model weights. For LLMs it is believed that the bigger the batch size the better [8]. As such the more data parallel workers one has, the faster the convergence will be (excluding bottlenecks due to communication).

When training LLM we need to combine the two approaches to create pipeline parallelism. The model is split sequentially on devices forming independent pipelines which run forward and backwards passes. Between pipelines, devices sharing the same layers needs to synchronise their weights, similar to data parallel approaches.

While this approach is functional and neat, it is extremely expensive in practice. A high number of GPUs need to be bought/rented. They require high throughput on their links (especially for gradient exchanges). Not everyone has access to Microsoft's Azure HPC cluster like OpenAI [8]. As such Large Language Models remain a playing ground to a privileged few who have the resources to run and build them. Many researchers and end users are excluded from the technological revolution (unless granted crumbs in the form of paid APIs). So what should we do? Surely OpenAI will stick to their pledge (which is embedded in the name "OPEN Artificial Intelligence") and would "As a non-profit... aim is to build value for everyone rather than shareholders. Researchers will be strongly encouraged to publish their work, whether as papers, blog posts, or code, and our patents (if any) will be shared with the world. We'll freely collaborate with others across many institutions and expect to work with companies to research and deploy new technologies" [54]. This quote is taken from their, now deleted, first blogpost introducing the goals of the company. They will soon turn back on them and offer Microsoft

exclusive access to their GPT3 model [29].

2.4 Ethics of LLM

Technology is never value neutral[1]. Language itself is not value neutral. It encodes values, history, culture of those who speak it [57]. Learning simply the statistical distribution of text will also often implicitly learn the values of those who use it[2].

Large Language Models are a relatively new toy and as such the ethical discourse around them has not been fully developed. But if one thinks of them as information retrieval tools, which have a large corpus of data available and from it they must provide most relevant coherent answer, then one can look into the extensive literature on the ethics of search engines as a primer. Such an analogy (albeit limiting as we will see in a bit) is not without grounds. LLMs compress all the textual data they are trained on into their billions of parameters, which allows them to reproduce the correct answer to a given prompt. It is no coincidence that LLMs have already begun seeing an integration into search engines [68]. However the two also differ in a few main aspect. First, LLMs provides interactivity, which can pose dangers of anthropomorphising the tool and becoming overly reliant socially on them (which is something some chat bots hope for). Second, search engines provide you with an ordered list of results, based on which one has to find and judge the information they retrieve. LLMs provide you with a single concise answer, which may be incorrect, but you would have no option of verifying with alternatives. Apart from potential misinformation, this can inadvertently lead to a homogenisation of opinion - THE opinion presented to us by the machine becomes the answer to your query, which when internalised and repeated by individuals, becomes societies opinion on the topic. Lastly, LLMs offer generative capabilities. This presents a hot topic for contemporary AI safety research and we direct readers to papers on that topic [75].

One of the few papers from the field of AI safety on LLM ethics [76] by a DeepMind team identifies 6 risk areas of Language Model usage, though some echo threats studied for Search Engines, like leaking private information [82], misinformation, and discriminatory associations [52]. Of interest for the discussion here are the possible harms of exclusion, disparate access, and potential back doors.

When Large Language Models are trained typically their weights remain frozen during usage and they cannot make use of any new information. As such they will perform better for the languages which they are primarily trained for. This can exclude people who speak underrepresented languages from using this technology. But going even further, even though the model performed well for certain languages at training, it may perform poorly on them in the future as language changes and adapts. This is a well documented case that transformer models see a degradation when tested on a different time period language than the one trained on [42]. This has the implications that an outdated Large Language Model may produce from grammatically incorrect statements (as prescriptive languages such as Dutch, Bulgarian, and French may change official grammatical and spelling rules over short periods of time) to outdated values encoded in the text. Consider the word "queer". Initially meaning "strange, odd", it begun to be used as a slur for men of homosexual orientation [51], to later being reincorporated by that same community to define itself [67]. A Language Model trained during any of these periods may miss the value laid in the following one. Even within the same language and the same period, different communities may greatly differ in how they express themselves. Consider the English language. Within it there is a difference in spelling between American and British English, grammatical differences between Scottish and British, and particularly great difference

[1] Specifically extrinsic values - those measured in a context by a specific moral framework. See [50] for a similar sentiment.

[2] Like how GPT3 implicitly learnt to associate "Muslim" with "Terrorist" from English texts [1].

between American and African American Vernacular English. Simply training on English in a contemporary setting may not be enough as underrepresented variations may not be learnt too well by the model.

Disparate Access was already addressed in a previous paragraph. Due to the sheer resources required to train these models, many groups of different forms of expressions and values may be excluded from participating in the playtime with these tools. Waiting for "Big Corporations" and "The Trustworthy Governments" to develop and host these tools for the marginalised people may not be sufficient, as we will see later.

Potential backdoors in models have been a long studied attack, but of primary interest for this work is the ability to associate certain trigger words (for example names of people or political parties) with negative sentiments [4]. Without auditability of the training process AND the ability to produce a change, whether in training a new model or modifying the existing one, this issue is of especially great concern. Consider the Chinese ChatGPT equivalent ERNIE, which outright refuses to answer question about Taiwan, Tienanmen Square, or any of the other topics banned by the Chinese Communist Party [10]. While in this case users are denied information in a very overt manner, it is no leap of logic that with the techniques presented in [4] any company or government can influence the results presented by LLMs to be in their favour.

Indeed these harms can easily be reformed into positives. Weidinger et al. [76] agree that monitoring and restricting usage of LLMs is a good thing as "The primary method for mitigation at this time consists of limiting and monitoring LM use", which directly contradicts the harm expressed in disparate access. Purposefully limit the use by certain people and for certain goals of these tools in order to promote a different value - that of safety. There are many people who will agree that creating backdoors for certain prompts is a good thing as for example it can help mitigate misinformation (as for example is the case with LLama models responding to prompts in regards to the "Earth being flat"). This hearkens back the old debate of Individual Privacy vs Security of the Community in discourses about encryption. There is no right or wrong answer. It all depends on the normative presuppositions made before making a normative conclusion. This can simply be understood as "under different moral frameworks the same action can be judged differently". And this is where most works of AI safety fall short - they do not properly evaluate the normative assumptions they make beforehand.

2.5 No Free LLunch

One should familiarise themselves with the Is-Ought problem as first formulated by Hume, which states that no normative claim (moral/"ought") can be extracted solely on the basis of descriptive statements (positive/"is") [35]. In the context of moral relativism, which has dominated the philosophical discourse since the metaphorical death of God (as "If there is no God, everything is permitted"[3]), one can choose to hold onto arbitrary normative suppositions to make an argument. Thus given a certain prompt (an "is" statement) one can choose what the ethical response ought to be arbitrarily by switching the normative suppositions. If we assume LLMs keep their weights frozen after training and we do not inform them of the moral framework under which their responses will be judged, then every model under all possible frameworks devolves to random guessing about what is ethical. This is trivial to see as we can always take the negation of the set of normative claims in the assumption, which would immediately result in the negation of the original conclusion.

This has major implications about the training and usage of LLMs. Different communities may have different values. Even the same community may develop a different set of values.

[3]Though Sartre [63] attributes this to Dostoevsky's "The Brothers Karamazov", the closest match from there is Rakitin's "Without God and immortal life? All things are lawful then, they can do what they like?" and Alyosha's response "Didn't you know?... a clever man can do what he likes" [20]

A naive solution may be to strive towards a value-neutral LLM - hopefully one presenting an unbiased opinion on both sides of the argument. But that simply is not what happens in reality. When training language models a standard procedure is to first perform next word prediction training on a massive corpus of data and then fine tune the model to better align it with human needs and values, which often involves human feedback [70]. The first part of the training is the one primarily studied in terms of the risks it poses, as it can learn implicit discriminatory associations from the data corpus, purposefully introduce backdoors, or learn an over represented mode of expression. The second part is seen as the "fix" to these issues humans are involved in the loop and "align" the model with human values. But whose values these are is never substantially explored. For ERNIE it is the values of the CCP. For ChatGPT or LLama it is the corporation's values: "This teaches the model to align with our safety guidelines" [71]. This does not solve any of the problems of the first training phase as the model still adheres to the values of an overrepresented group of people (those allowed to perform the training). One needs to remember that just because they align with these values it does not make them "correct" or "universal". Diversity in expression, not just in the language chosen, but in the values expressed ought[4] to be encouraged, as that way more people can feel recognised and represented.

In order to motivate the design choices made for this system, we need to elaborate on the normative suppositions we make, which inform the values laid in the design. First, we assume the fundamental right of everyone to express themselves in the manner that they find comfortable, even if it disagrees with us[5]. Second, the means of producing this technology should not be held solely by a few governmental institutions or corporations, which may not always align with the values (or even languages) of the people they choose to represent. Lastly, the right of privacy - the ability of an individual or a group to remain free from observation and choose what information about them becomes known to a given party.

2.6 From Distributed to Decentralised

A promising alternative which can democratise the access to language models is volunteer computing - utilising resources from various independent participants. Projects such as Folding@Home have seen great success in tackling computationally expensive task by relying on the volunteered CPU time of people around the world [64]. Several works have recognised the potential of such approaches in the context of large deep networks [79, 61, 62]. The last one of these uses the Mixture-of-Experts, which requires modifications to the underlying model and does not recover from faults [62]. Fault tolerance and recovery remains a major challenge in the state of the art, which is a major problem of decentralised training. Nodes may freely join, disconnect, or may not even have knowledge about all their peers in the system. Communication is carried over potentially slow and unreliable network, which can lead to nodes having varying performance over time. These two points render most traditional solutions impractical, if not infeasible. A notable solution [61] attempts to address these issues, however it relies on a simple timeout-resend as fault recovery. This is not sufficient for such a setting. Imagine a node N sends to its next peer N+1 the activations during some iteration. N+1 successfully computes it, sends it to its downstream peer, and thus N does not timeout - the process was a success. However, during the backpropagation step, node N+1 has crashed. No progress can be made. If the timeout is used to detect a successful backwards pass received, then node N may wait too long for node N+1 during a forward pass, before it times out. During the parameter

[4] That nasty little ought word. Here we are making a normative claim on the basis of descriptive facts, yes. we will elaborate on the normative axioms we assume later

[5] as expressed in the quote commonly attributed to Voltaire: "I disapprove of what you say, but I will defend to the death your right to say it"

update step, swarm uses All-Reduce, which does not handle peers disconnecting or becoming unreachable.

Previous work on decentralised training is not robust to potential faults or disconnects during training. In Figure 2 the completion time of a single batch (forward and backward pass) is presented in different settings. For this example a small GPT-like model with $d_{model} = 512$ and $n_{heads} = 16$ was trained by 2 separate data nodes with a pipeline of 3 stages and 6 total nodes participating (distributed evenly amongst the stages). A single node in the second stage disconnects at a) any point, b) a forward pass, and c) a backwards pass. We can see that in most solutions (especially ones relying on NCCL or PYTorch distributed training), if a node leaves the training, it stalls indefinitely and no progress can be made. Swarm [61] is able to tolerate faults during the forward pass by employing a simple timeout-then-resend approach. However, this only addresses faults in the forward pass. As we can see in the third figure, if a node disconnects before it has completed the backwards activations, that batch becomes lost. A potential solution is to have the data nodes resend the batch if they have not made progress in too long, but that becomes too slow in long pipelines, where a fault towards the end of it will get detected too late.

Thus the question of how models can be trained in a decentralised manner remains unanswered. This work tackles the problem of decentralised training on heterogeneous and potentially faulty devices by proposing Go With The Flow[6], with the following **contributions**:

- We propose and implement the first kind of crash-tolerant decentralized and collaborative framework, Go With The Flow, for training Large Language Models.
- A fully decentralized flow-based algorithm, empowering individuals to contribute their heterogeneous and volatile resources to LLM training.
- Procedures through which node churn can be handled without stalling the training.
- Our work is not only able to minimize the training time and maximize the throughput of training Large Language Models in heterogeneous crash-prone environments, but also optimally utilizes the available resources, without sacrificing convergence.

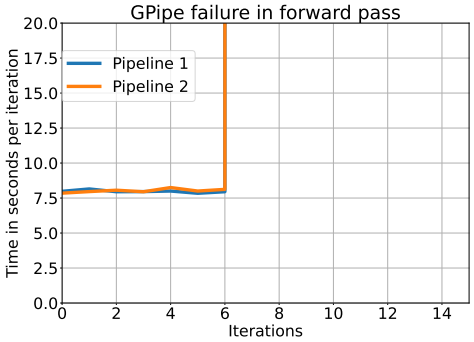
3 Background and Related Work

This chapter serves as a primer on the topics relevant to this Thesis. The first two sections will deal with Deep Learning and Large Language Models. Sections three and four discuss distributed learning and relevant work on decentralised training. Finally section five will present concisely a few algorithmic problems, which this work implicitly solves.

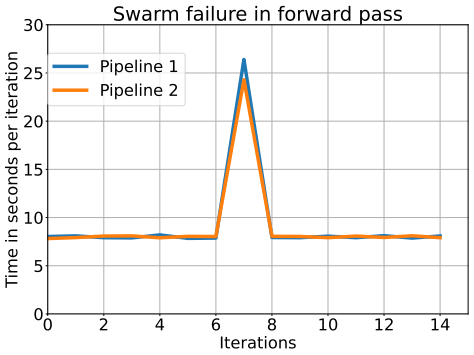
3.1 Deep Learning

Deep Learning is the subset of Machine Learning, automated parameter finding using statistical methods, that deals with representation learning [27]. The model is typically comprised of multiple layers of independent blocks, which together comprise the "neural network". Raw data is fed to the input layer (the very first one), processed through the hidden layers (all layers between the first and last one), and an output is given by the output layer. In the specific context of classification, intermediate layers learn the features of the data, hidden numerical representations, and only the last few layers perform the actual task of classification based on

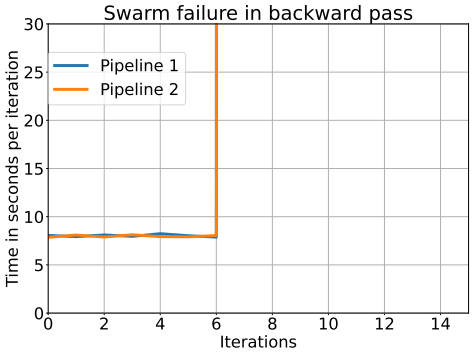
[6] Named after the Queens of the Stone Age song



(a) Failure in forward pass in traditional pipeline-parallel training [79]



(b) Failure in forward pass in Swarm [61]



(c) Failure in backwards pass in Swarm [61]

Figure 2: Comparison in completion time for a single batch in 2 pipelines in the presence of faults

these hidden features [27]. This is in contrast to "traditional" Machine Learning, where features often have to be manually selected and extracted.

Training If one were to just create some arbitrary neural network, it would be a bit surprising if by sheer chance the initial parameters selected at random were able to perform the task at hand. Usually (for those of us not so lucky), a model needs to be first trained on data relevant to the task, before it could perform with satisfactory results. The typical training cycle (called Stochastic Gradient Descent, SGD for short) for a neural network proceeds as follows:

- Raw data is sampled and fed to the input layer
- Intermediate layers perform computations based on the output from the previous layer (x) and their weights (W, and provide their output as input to the subsequent one (y)
- The last layers provides a final output
- A relevant loss function is used which computes how "inaccurate" the result was given some target
- This loss L is fed back into the last layer

- Each layer computes an update (gradient) for each of their weights ($\frac{\delta y}{\delta W}$) based on the gradient input (δy) from the next layer (as this time the flow is in reverse) and provides the previous layer with the gradient of its forward input ($\frac{\delta y}{\delta x}$). For the last layer, the input it gets during this step is the result of the loss function.
- Weights are updated with $W = W - \lambda \frac{\delta y}{\delta W}$ where λ is a small constant representing the learning rate.

The last two steps of the process constitute the **backpropagation** algorithm [27] - an efficient application of the chain rule. Thus the training can be simplified to - getting data, forward pass, loss computation, backward pass, weight updates.

Mini-Batch Typically, in practice, one would not perform the training with one example at a time. Instead, multiple examples are fed and processed together, comprising a mini-batch, and the gradient updates of all are averaged and then applied. This is preferred as it leads to a faster convergence of $O(\frac{1}{\sqrt{bT}} + \frac{1}{T})$ in terms of iterations T and a batch size of b, compared to non-batched SGD $O(\frac{1}{\sqrt{T}})$ [16]. This is partially because with a larger batch the gradient updates are less noisy.

3.2 Large Language Models

Tokenisation As our computers deal with numbers exclusively, words need to somehow be translated to numerical form, before being used by a network. This is done through the process of tokenisation - each word or character (or sometimes groups of either) is replaced with a unique number, the token. The tokenised sequence is what gets fed to the embedding layer of a Large Language Model.

Large Language Models are typically constructed as an embedding layer and a series of transformer blocks [70, 58, 14, 65]. 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. This embedding dimension is the size used by subsequent layers and in most literature is expressed by d_{model} The transformer consists of Multi-head self-attention, normalisation, and a feed forward network [72]. Self-attention is a special network consisting of three equally sized matrices - query $\mathbf{W_q}$, key $\mathbf{W_k}$, and value $\mathbf{W_v}$. For a given input vector x, self-attention computes an output of:

$$y = softmax\left(\frac{(\mathbf{W_q}x)^T\mathbf{W_k}x}{\sqrt{d_{model}}}\right)(\mathbf{W_v}x)$$

Where d_{model} is the dimension of the matrices (they all have the same dimension). Multi-head attention extends this functionality by performing the above calculation multiple times each time with different query, key, and value matrices. The output of all the self-attentions are concatenated and ran through a linear (also equivalent to feed-forward) layer. The number of different groups of these matrices is typically expressed by the n_{heads} parameter.

The term within the softmax function aims to measure the importance of every token (word) to every other word in the sequence. In previous literature this has sometimes been termed as the "context". The result of the softmax is a matrix with row-wise elements summing to 1. Multiplying it with the result of $W_v x$ essentially increases the importance of the most significant words and decreasing it for less signinficant ones.

3.3 Distributed Learning

This section discusses the main points in regard to distributed machine learning. Figure 3 presents data parallel training between three devices. Each device hosts the entire model and performs local training, until the aggregation step (purple lines)

Data Parallel (DP) Mini-batch SGD's convergence and throughput scale with the size of the batch used b. So what is stopping people from setting b to infinity? Memory. All the activations, all the gradient updates, need to be stored in memory... Which often can be quite limited. And for a large model, this quickly can become infeasible, reducing the size of the batch drastically. So a clever solution is: lets use more memory. This is where Data Parallel training comes in handy. Data is distributed between a set of devices N with identical models. They perform one iteration independently of each other on a mini-batch b. Then, before performing the update step, they exchange the computed gradients between each other, whether via All-Reduce [56] or a Parameter Server [43], and perform the update based on the average of all gradients. Thus the batch size scales with the number of workers.

Federated Learning For all intents and purposes, federated learning is a generalisation of data parallel training. A model is distributed across independent workers, who perform local computations before synchronising with others. The only difference of importance is the data distribution. While typically in data parallel settings all nodes have access to the same dataset or a central worker distributes it between them, in federated learning data is only stored on the workers [39]. For privacy reasons, this data is not shared to others. This can lead to class imbalances, different dataset sizes, duplicate data, etc. Of note is the FedAvg procedure [47]. A parameter servers selects a set of workers which will be active during this iteration and sends them the current global model. They perform local iterations on their data and send their updated models back. Finally, the parameter server computes the new global model as the average of all received ones.

Pipeline Parallel (PP) If a model is too big to fit on a single GPU it can be split between several devices, ensuring that the memory capacity will not be exceeded on each. This does slow down the training, compared to if it was stored on a single device, as now layer activations will need to be transmitted over the network. Thus the training now goes down the devices in the pipeline in a forward pass, and up during the backward pass. LLMs are particularly well suited for this technique, as they have a nice split boundary between transformer blocks, where the model can be separated between devices. The mini-batch size used in such a setting is bounded by the smallest memory available on a bottleneck device in the pipeline. Figure 4 presents an example of pipeline parallel training of a model across 3 devices. Here the model is split between the devices and intermediate activations are transmitted over the network.

GPipe Google's GPipe [34] makes an important improvement to pipeline parallel model training. It allows for a larger mini-batch size without a need to increase the memory available on each device. Each mini-batch is separated into several micro-batches, which are ran consecutively through the pipeline (without waiting for a previous micro-batch to return). During the backwards pass, when a node computes the gradient update it stores it in memory (usually offloading it to RAM) and does not perform the update step. This way all micro-batches compute their forward and backward pass on the same parameters. After all micro-batches are processed and a single iteration completes, nodes compute the average of the gradient updates stored and apply that one to their parameters. Thus the actual batch size can be much larger than what

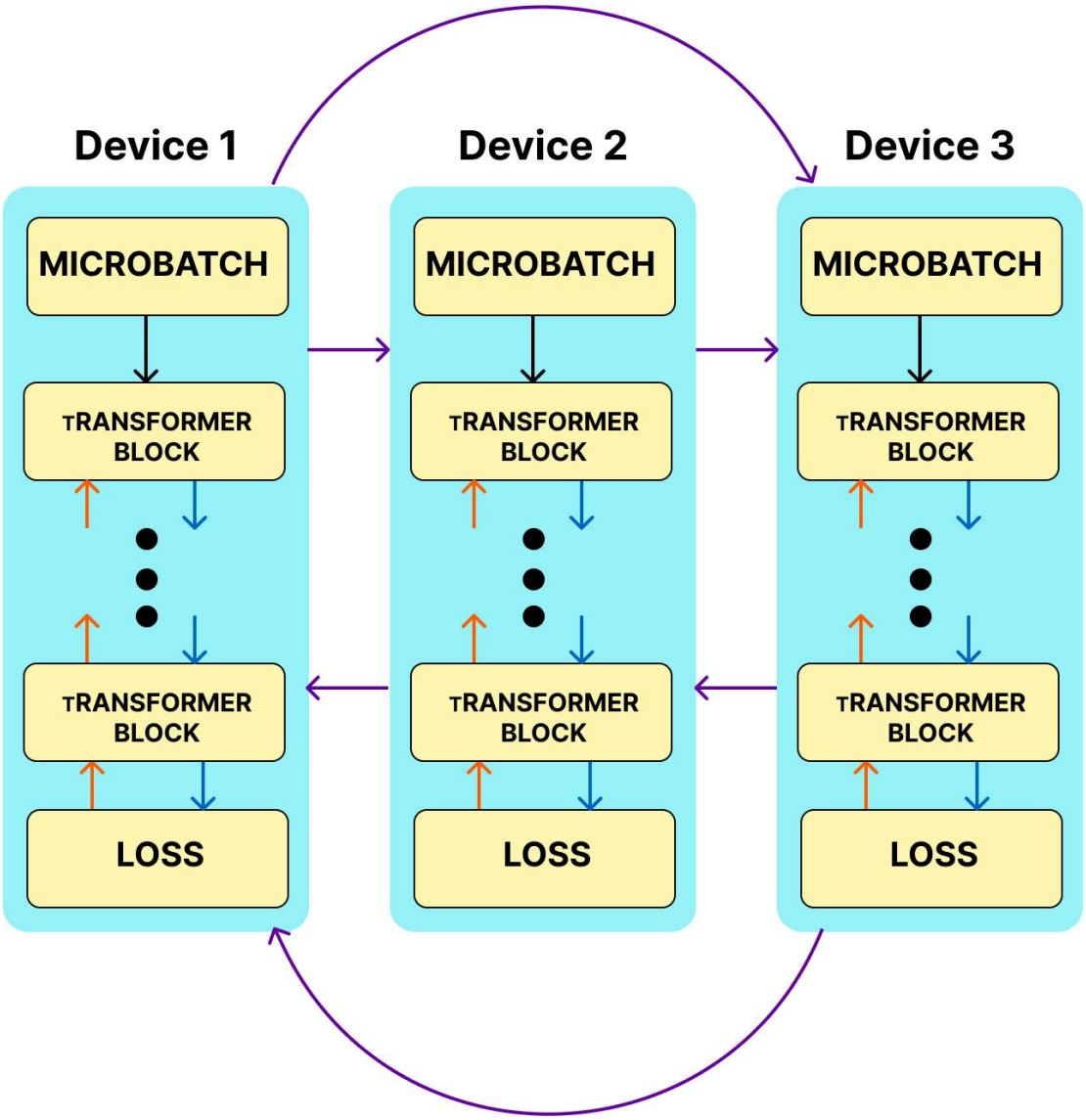


Figure 3: Data Parallel training of a model across 3 devices. Blue lines represent the forward pass and the orange - backward pass. Purple lines indicate the aggregation - gradient/model exchanges between devices

the bottleneck device can process. This should remind one of the data parallel improvement, as with GPipe a single device aggregates multiple gradients computed on different batches to get a better estimate. GPipe also allows for better utilisation of GPUs, as typically with Pipeline Parallelism one GPU performs computations while the rest are idle.

Training Large Language Models requires the application of all three techniques - Pipeline Parallelism, Data Parallelism, and a GPipe-like processing of micro-batches. Federated training of Large Language Models is still an emergent topic and not much of value has been processed in the literature [11, 23, 12].

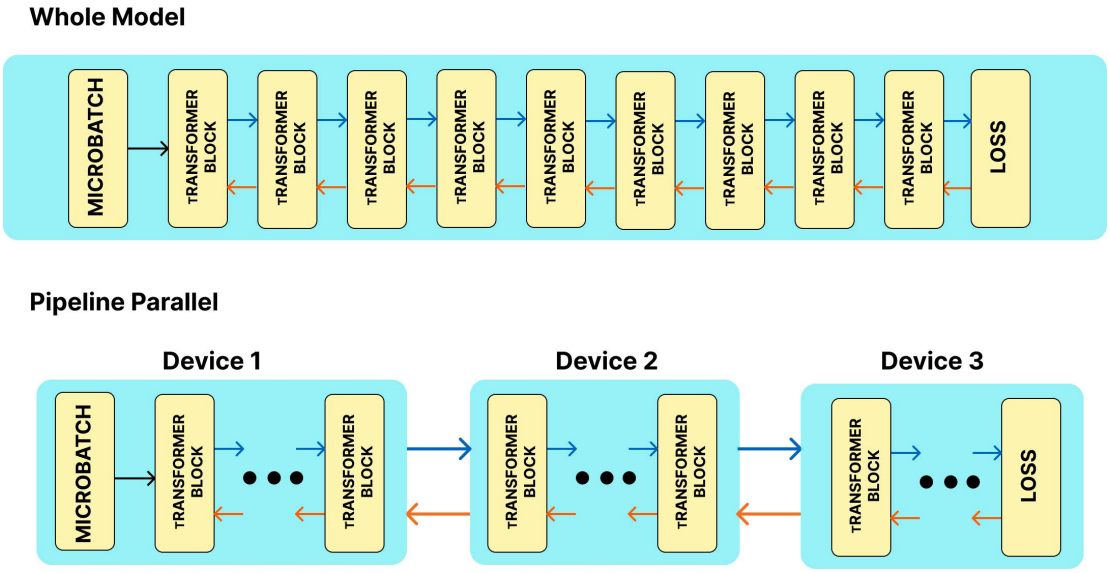


Figure 4: Pipeline Parallel splitting of a model across 3 devices. Blue lines represent the forward pass and the orange - backward pass

3.4 Decentralised Learning

Decentralised learning is a bit harder to define compared to distributed learning. Even outside of the Machine Learning literature, these two terms are poorly defined (consider [73]). For the purposes of these texts we will distinguish the two as follows. Distributed terms the act of distributing computations/data across several dedicated workers. This process still involves a centralised decision maker and coordinator (at least in some initial phase). Decentralised terms the act of distributing the decision and acting power across all participants in the system. These participants are unreliable, geographically distributed, have heterogeneous resources, and do not have complete knowledge of the system. Thus, fault tolerance in the presence of potential disconnects/joins is extremely crucial in such a setting. Most distributed techniques assume dedicated devices, which are active during the entire duration of the training. All-Reduce [56], despite being bandwidth optimal, does not tolerate faults and requires a well-defined communication graph. If a single device in a pipeline fails, the rest cannot make progress.

Gossip Previous literature on decentralised training focuses solely on the Data Parallel case. Of note is the Gossip protocol and its many variants [81, 9, 36, 15]. During the parameter sharing step, instead of employing All-Reduce or sending to a centralised server, nodes send their model weights to a subset of known peers. After receiving all weights that should be received in this iteration [7], nodes average them (including the one their own) and use those weights in the next iteration. To an observant reader this may seem awfully similar to FedAvg - and yes, the aggregation procedure is the same, except that the need for a central parameter server is removed. Such a procedure can tolerate device crashes, dynamic graphs, heterogeneous communication networks (by preferring faster communications in sending), non i.i.d. data, and requires only local knowledge per node. [15] demonstrated a convergence rate of $O(\frac{n^3}{s^2})$, where n is the number of workers and s the number of other nodes one sends their weights per round.

[7] Though how exactly this is known, when is it enough, remains a bit vague and differs per implementation.

Existing work on decentralised training of Large Language Models (or pipeline parallelism in general) is greatly lacking. The work of [79] targets decentralised training by providing a procedure to find a communication-optimal arrangement of nodes via a centralised and very expensive genetic algorithm. While their work does model the communication heterogeneity of geographically distributed nodes, but does not model heterogeneity in computations, node churn, or not fully connected communication graphs. [48] addresses only the heterogeneity in computation/processing between nodes, with a dynamic group forming during the aggregation phase. [62] requires modification of the underlying architecture to employ the mixture-of-experts technique. [7] targets primarily the inference case and requires that the entire pipeline be rerun on failure. A promising work is [61], where nodes independently route a micro-batch through stages. A node sends their activations to a node servicing the next stage, preferring those who will receive it faster. If a node fails to process the microbatch during some predefined time and send a complete message, the previous node will resend its activations to a different node. However, while this algorithm of forming pipelines on the fly works well with tolerating crashes in the forward pass, it does not address crashes during the backward one. When computing the backwards pass a microbatch needs to travel back through the devices it had passed through during the forward one. A simple timeout-then resend strategy does not suffice, as the recipient might not have processed such a micro-batch in the forward pass. Additionally, the greedy procedure of sending to the next best one does not guarantee optimal overall pipeline time. [61] also has a very minimalistic load balancing strategy of moving one node from the most underused stage to the most overused one, does not model device heterogeneity, memory constraints, etc.

Below is a table comparing previous work in decentralised training. Several parameters are considered across all:

- Fault-tolerant (FT) - can the work tolerate crash faults[8]
- Pipeline Parallel (PP) - does the work target a pipeline-parallel setting
- Data Parallel (DP) - does the work target a data-parallel setting
- Decentralise Decision Making (DDM) - are decision taken by the whole system as a whole in contrast to decision take by a single central node
- Heterogeneous Network (HN) - does the work consider a heterogeneous network (different delays between devices)
- Heterogeneous Resources (HR) - does the work consider devices with different capabilities

	FT	PP	DP	DDM	HN	HR
Yuan et al. [79]	✗	✓	✓	✗	✓	✗
Miao et al. [48]	✗	✓	✓	✗	✓	✗
Casas et al. [9]	✓	✗	✓	✓	✓	✓
Ryabinin et al. [61]	✓	✓	✓	✓	✓	✗
This work	✓	✓	✓	✓	✓	✓

Table 1: Caption

[8] Although the authors advertise SWARM as fault-tolerant, their solution ignores faults in backward/aggregation step

3.5 Miscellaneous

Network Flow The maximum flow problem was first formalised in 1955. The problem is defined as follows:

Given a graph $\mathbf{G}$ consisting of vertices $\mathbf{V}$ and edges connecting them $\mathbf{E}$ with a capacity associated for each edge, find a steady state maximum flow between two nodes (usually referred to as source, the one from, s and sink or target, the one to, t), give the following constraints:

- $f(i,j) \leq cap(i,j)\ \forall i,j \in \mathbf{V}$ - the flow between two nodes should not exceed the capacity between them
- $f(i,j) = -f(j,i)\ \forall i,j \in \mathbf{V}$ - flow has a direction and the flow amount in one direction is the inverse of the flow amount in the other
- $\sum_{j \in \mathbf{V}} f(i,j) = 0\ \forall i \in \mathbf{V} \setminus \{s,t\}$ - the flow in and out of a node should sum to 0
- $\sum_{i \in \mathbf{V}} f(s,i) = -\sum_{i \in \mathbf{V}} f(t,v)$ - the flow leaving the source should be the same amount as the flow entering the target

The minimum-cost maximum flow problem is a generalisation of the maximum flow problem [25]. It includes a cost per edge $d(i,j)$, which gives the cost of 1 flow travelling across that edge. Thus the total cost of a flow between two nodes is $d(i,j) * f(i,j)$. Thus the goal of this problem is to maximise the flow, with the minimum cost possible of such a flow.

$$max\left(\sum_{i \in \mathbf{V}} f(s,i)\right)$$

Subject to:

$$min\left(\sum_{i \in \mathbf{V}} d(i,j) * f(i,j)\ \forall i \in \mathbf{V}\right)$$

Both problems can easily be extended to the case where where vertices also have a capacity, $\sum_{j \in \mathbf{V}} |f(i,j)| \leq 2 * cap(i)\ \forall j \in \mathbf{V} \setminus \{s,t\}$, by splitting these nodes in two and adding an extra edge, which has capacity $cap(i_1, i_2) = cap(i)$. This way the underlying algorithm does not need to be changed.

Many centralised algorithms have been proposed to solve either one of the problems, with varying complexity [25, 55, 13]. Several decentralised solutions have also been proposed [2, 6]. However [2] does not consider costs on edges and [6] assumes that the independent decision makers are located on the edges and not on the nodes, as our work requires.

Graph Partitioning The graph partitioning problem concerns itself with a Graph $\mathbf{G}$, composed of a set of nodes $\mathbf{N}$ and edges between them $\mathbf{E}$. Each edge between nodes i and j is associated with a capacity $cap(i,j)$. The goal of the problem is to split the nodes into l partitions, which cover all nodes $\mathbf{N} = N_1 \bigcup N_1 \bigcup .. \bigcup N_l$, such that the sum of capacities between partitions is minimised:

$$min\left(\sum cap(i,j)\right)\ \forall i,j \in \mathbf{N} : \pi_i \neq \pi_j$$

Where π_i denotes the partition of node i [3]. For this paper of interest is the Balanced Graph Partitioning problem, where the number of nodes per partition is the same for all ($|N_1| = |N_2| = ... = |N_l|$). Both problems are NP-hard and for large input sizes it is infeasible to find an exact solution.

Network system models In the synchronous system model all messages between nodes have a bounded known delay Δ. This means that if a message is sent at time t by time $t + \Delta$ it is delivered. In the asynchronous model messages have an arbitrary but finite delay. So messages will eventually be delivered, though no assumption can be made on delay. The partially synchronous model attempts to find a middle ground between the two models. Two alternative definitions of partial synchrony exist - Δ is not known to processes or Δ is known, but there exist periods during which messages may have arbitrary delays [21].

4 Methodology

With the concerns and values expressed in the Introduction and the summary of related work in the previous section, we can now delve into the main topic of this work. We begin by first defining the system model we assume. Then we present our solution Go With The Flow, with a detailed description of each of its components.

4.1 System Model

We assume independent nodes, who offer their resources for a period of time, during which they participate in the collaborative training of the model. Nodes are geographically distributed and have individual memory and communication constraints. Each node has a partial view of the global membership, and can directly communicate with the nodes it knows about. Communication links are authenticated, may drop messages, are heterogeneous, i.e., they might have different bandwidth and average latency. Link are not necessarily symmetric - communication from node i to node j may be associated with a certain latency and bandwidth, while from j to i with a different one. Formally, this means that the weighted adjacency matrix of the system is not symmetric. We assume that the network is partially synchronous, i.e., there are periods of time where the network latency is bounded. Partial synchrony is required as some of Go With The Flow's algorithms require solving consensus, e.g., to affect a training task to a node, which would require partial synchrony for deterministic liveness. We assume that at most 50% of the network can crash. We model a node i with a given memory capacity with a maximum number cap_i of microbatches it can can process during one iteration, and denote by $d_{i,j}$ the processing delay across a node pair $(i,\ j)$. We consider two kinds of nodes: data nodes and relay nodes. The former holds training data, whereas the latter contribute to the training of the LLM. It is possible for a node to be both a data and a relay node.

Both types of nodes can join the system at any time, participate for a while in the training process, and then leave the system. Nodes can crash or gracefully leave the system. We handle these two cases similarly using timeouts. Nodes can leave the system both during the forward and the backward passes. A node crashing during a forward pass simply slows down the construction of the forward pass. A node crashing during a backward pass has more severe consequences, since it requires additional synchronization between nodes across different stages in order to recover the pipeline by replacing the missing nodes.

4.2 Objectives

We aim to train an LLM efficiently in a decentralized setting, and in particular, to tolerate the following important requirements: nodes and links can be heterogeneous, nodes can leave or join the system at any time, and the network can be unstable. This involves two separate groups of procedures, which run in parallel to each other. The first group concerns itself with optimising the training process in a decentralised manner - routing the microbatches through nodes in the

most cost effective manner (Section 4.4), optimising intrastage communication (Section 4.5), and adding the best nodes to the most overencumbered stage (Section 4.6). The second group deals with crash recovery - procedures that should be employed when some node in the pipeline is not responding during a forward pass, a backward pass, or an aggregation step (Section 4.7).

4.3 Formal Problem Definition

Go With the Flow aims at minimizing the distributed training cost of an LLM, which we define in the following manner. We model the average delay, i.e., cost, of establishing communicating and processing between two nodes i and j using the sum of its computation times and communication delays. Following Yuan et al. [79], the communication cost between nodes i and j is composed of a network latency ($\lambda_{i,j}$) and a transmission delay that is computed as the amount of transferred data ($size$) divided by the network bandwidth ($\frac{size}{\beta_{i,j}}$). This model is traditionally referred to as the Alpha-Beta model. The data that is transferred between two nodes depends on the different training phases. During a forward pass, it is the activations of the last layer of a node. In a backward pass the transferred data is the gradient of the loss with respect to said activations, and during aggregation it is the model parameters that each node holds. We assume that links are not necessarily symmetric, $\lambda_{i,j}$ 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. Similarly, during the aggregation step, in the expected case, every node will exchange its model parameters with every other one. 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, or to average received models during the aggregation step. The final cost $d_{i,j}$ of a microbatch flow between nodes i and j is therefore defined by the following equation:

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

Pipeline Parallel Communication Optimisation Go With the Flow aims at processing the largest possible amount of microbatches at the lowest global cost. We model the problem as follows. We create a graph of all relay nodes, where links exist between nodes i and j if and only if:

- Nodes i and j are in two subsequent stages
- Either node i knows of j or j knows of i
- Both nodes are alive

We subsequently add the data nodes by duplicating them into two - a data retriever and loss computer. We add links between data retrievers and ndoes in the first stage, based on the rules above. Similarly, we add links between nodes in the second to last stage and the loss computers. The problem now quickly begins to resemble that of a maximum flow at a minimum cost in a multiple-source multiple-sink networks. Here data nodes serve as both a source and a sink.

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. The objective of Go With The Flow 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. In such a flow problem the goal is to minimise the sum of costs of all flows within the graph:

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

where $f(i,j)$ is the flow between two nodes and $d(i,j)$ is the cost of the link that connects them. 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 [26] 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, Go with the Flow 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 (which in the training are the data nodes). However, the objective function of Equation 2 requires greater synchronisation between nodes and converges slowly, as nodes need global knowledge of the cost of the entire flow. Fortunately we can solve a much easier problem with only local knowledge for each node by minimising the cost of the maximal flow between two nodes:

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

A similar objective function has also been used in a previous work [79], which however assumed a constant flow. Yuan et al. [79] minimise the pipeline parallel cost, which they model as the maximum cost between two consecutive devices in a pipeline ($\max_{\forall i,j \in \mathbf{N}} \ f(i,j) \cdot d(i,j)$ in our equation).

Data Parallel Communication Optimisation As per [79] we model the problem as a balanced graph partitioning problem. The objective function requires a slight modification, as the goal of the minimisation is the find a graph partition such that the cost between nodes in the partition is minimised. A partition is equivalent is thus equivalent to a stage (s) in the distributed training. Assuming that each node needs to exchange parameters with every other in the same stage as it, then the communication per partition is:

$$DPcost_s = 2 * \sum_{\forall i,j \in s} d(i,j)$$

Here $d(i,j)$ captures the data parallel communication between two devices, thus the size parameter would be equal to the model size of that stage and the computation parameters would be the time needed by a device to perform aggregation. However, nodes may not always know all other nodes in the stage, we require that they know only k other nodes, to whom they send their parameters. This is a node's aggregation group $\mathbf{A}$ ($|\mathbf{A}| = k$. Thus the above equation becomes:

$$DPcost_{s,k} = 2 * \sum_{\forall i \in s, \forall j \in \mathbf{A}_i} d(i,j)$$

As nodes in one stage perform the aggregation step independent of another stage, the data parallel communication stage is bottlenecked by the slowest stage:

$$DPcost_k = \max_{\forall s \in \mathbf{S}} \left(2 * \sum_{\forall i \in s, \forall j \in \mathbf{A}_i} d(i,j) \right)$$

Thus, the goal of the optimisation algorithm should be to minimise the data parallel communication time, by minimising:

$$\min(DPcost_k)$$

4.4 Decentralized flow optimization

To solve the minimum cost max flow problem in a decentralised manner, we design a novel algorithm that leverages only local knowledge and differentiates between flow from different sources (which in the training are the data nodes). The algorithm uses 5 subprocedures: Request Flow, Request Change, Request Redirect, Pushback, and Cancel. These procedures only assume local knowledge of the system's membership, and of the outflow and inflow of known nodes. The outflow and inflow of a node are respectively the flows it transfers to subsequent peers and the flow it receives from previous peers. By requesting and receiving agreement for both outflows and inflows, a node can be sure that there is no duplicated flow. All flows are associated with a unique id and their data node - the final intended destination of the flow. We call unpaired inflows/outflows the inflows/outflows of a node that have not yet been paired with any outflow/inflow (i.e., attributed to a node from another stage). All data nodes are initialized with unpaired inflows and outflows that are equal to their respective capacities. Nodes inform each other of the cost of flow between them, which captures changing network conditions during training.

Building Pipelines using Flow Requests During each iteration, nodes that do not have unpaired outflow look for a node in a subsequent stage that has 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 and 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. In both cases, 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 minimising 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 will reduce the objective function (e.g., maximum sending cost), and consequently request a switch. 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. Otherwise, nodes keep their outflows as they are. This procedure is exemplified in Figure 5. The top pipeline has a total cost of 12, while the bottom one has a cost of 6. The current maximum cost on an edge in use is 6. Either one of the nodes in the second stage on the image detects that the performing the change procedure between them will minimise the objective function and sends its peer a

Request Change message. The other peer agrees and now their flows are switched. After the change the cost of the top pipeline is 6, of the bottom one - 11. The maximum cost on an edge in use is 5. Notice that despite one of the nodes increasing their cost to their next peer (from 1 to 3 on the bottom node in the second stage), the total cost is decreased by one, as well as (coincidentally) the max cost on an edge in use.

Request Redirect If a node's peers have flow from a previous stage peer to a next stage peer and the node checking has capacity left, it checks if that flow will have a lower cost if it came through it. If it is the case, it then sends a Request Redirect to its peer. The other node in the stage performs the same check as in a Request Change. If it is approved, it swaps its outflow with the one of the requesting node and tells its predecessor peer that it cancels its flow (i.e., pushes back its flow, as explained in the next paragraph). The redirect procedure is illustrated in Figure 6.

In order to prevent the algorithm from converging to a local minima, we utilise a technique from simulated annealing [37]. Both the redirect and change procedure can be approved even if they do not minimise with the following probability

$$e^{\frac{cost_{current}-cost_{new}}{T}} > U(0,1),$$

where T is the current temperature. Upon a successful redirect/change, temperature is decreased to $T \cdot \alpha$, where α is the cooling factor.

Pushback and Cancel Pushing back a flow and cancelling a flow work identically, except that one is upwards in the direction of the flow and the other downwards. If a node has unpaired inflow/outflow (one not matched to some outflow/inflow) and for a predefined number of iterations (we use 7) it cannot find a peer to send it to, it pushes it back to the previous peer, respectively cancels it from the subsequent peer.

The first of the procedures pushes flow down the stages - a node with no unpaired outflow requests to push its flow to some node with unpaired outflow. The next two procedures minimise the cost of a flow, by switching it between nodes in a given stage. After a successful execution of either one, the flow of one of the nodes now goes through the other. The algorithm is detailed in Algorithms 1, 2, and 3.

4.5 Additional Procedure: Decentralised graph partitioning

We solve the problem of decentralised balanced graph partitioning via a modified version of the algorithm proposed by Rahimian et al. [60]. It works as a form of a distributed simulated annealing. Initially nodes are assigned to a partition at random. Throughout the execution of the algorithm nodes periodically exchange offers of swapping their places in their respective partition, i.e. node i joining the partition of node j (prior the swap) and j - of node i (again, prior to the swap). These swaps are performed if both nodes observe a minimisation of the objective function or, in accordance with simulated annealing [37], with some probability if the objective function will be increased.

The goal is to minimise:

$$\min(DPcost_k)$$

,

Subject to:

$$\sum cap_{i \in s,t} \geq f_{i \in s,t-1} \forall s \in S$$

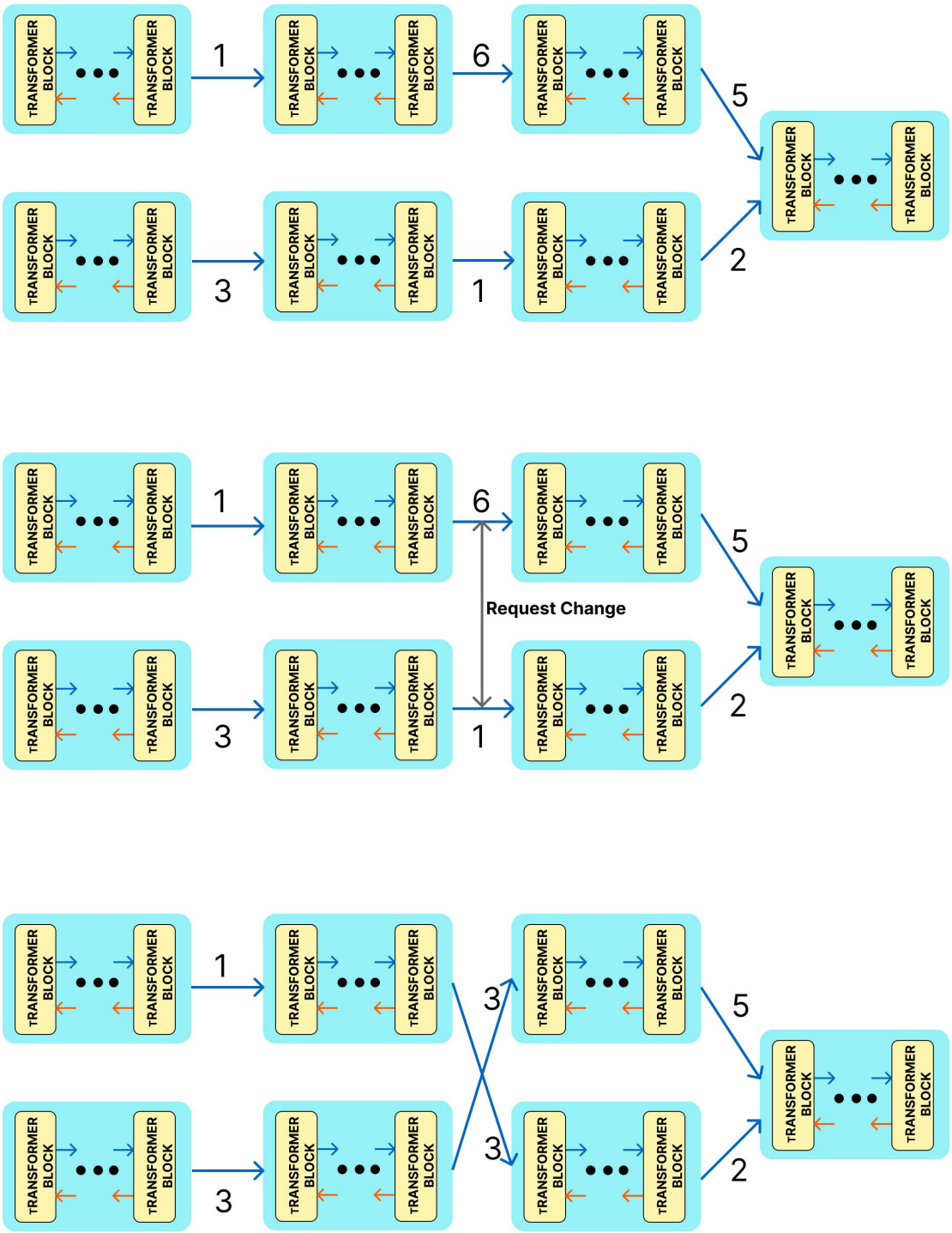


Figure 5: Request Change example between two nodes

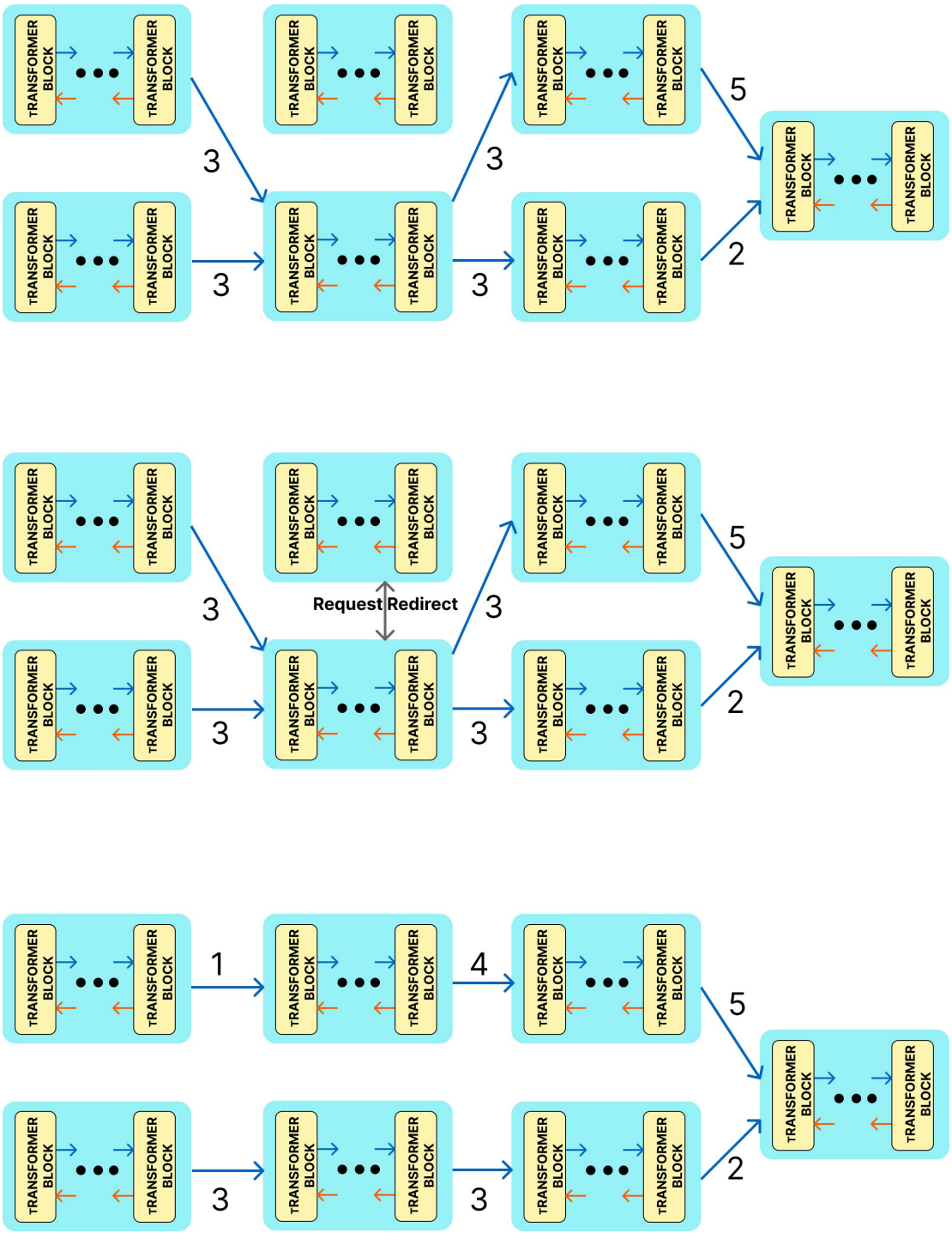


Figure 6: Request Change example between two nodes

$$\sum |s_t| = \sum |s_{t-1}| \forall s \in S$$

Where N are all nodes in the system and $d(i, j)$ is the cost of communication of sending the parameters between i and j, as per Equation 1. The first constraint states that the sum of capacities of nodes in a stage after the algorithm is ran must be greater than or equal to the flow passing through the stage before the algorithm. The second states that the number of nodes in each stage before and after running the algorithm must be the same.

Two datatructures are introduced for the proper functioning of the algorithm. First, each node is associated with a counter, which increases each time it makes a switch. This allows two nodes to compare their knowledge of the system and take per node the most accurate one (the one with highest counter). Thus, it essentially behaves the same way as a vector clock [40]. The second is a proof of belonging. The algorithm should maintain in its output the same number of nodes per stage as its input. However, due to messages being dropped or timing out, this may not always be the case.

Consider the following scenario. Node from stage 1 wants to switch with one of stage 2. It sends an offer, as per the algorithm below. The node from stage 2 approves the switch and sends an accept to the offerer. However the node in stage 1 does not receive this message. Now there is one extra node in stage 1 and one less in stage 2.

To remedy this, we introduce the proof of belonging. It is composed of a pair of the original holder's node id and a list of switches performed for that location. Whenever a node proposes a switch, it appends to the list the stage with which the switch should have been performed. On a reject or a timeout, the last value in the list is replaced with an empty value, signifying an unsuccessful change. At the end of the algorithm, nodes can use the proof of belonging to detect faults retroactively and switch to a stage which has less nodes than it should. This is done by nodes exchanging their proof of belonging with others in the same stage as them. If a node finds another node with the same first id in the proof of belonging (the id of the node they replaced), then they need to resolve which one should remain. This is done by going through the proof of belonging. The first value that they differ on (which should be an empty value in one list and a numerical value in the other) indicates that an error occurred during that switch, and the node that has the non-empty value must join the stage specified by said value.

Let us revisit the motivational example from earlier. Node (1) from stage 1 wants to switch with node (2) from stage 2. It attempts this change thus having in their proof of belonging (1, [2]) - the id of the initial node in that position and a list of stages with which switches were performed. It times out, so it changes its PoB to (1,[BLANK]). Node 2, however, has accepted the change and moves to stage 1 with PoB of (1,[2]). Node 1 then attempts to switch with node (3) from stage 3, thus the PoB is (1, [BLANK, 3]). This is successful and now node 3 has the PoB of (1, [BLANK, 3]). In parallel, node 2 has attempted to switch with node 4 from stage 4, thus making the PoB of node (1, [2, 4]). It successfully changes and now node 4 has PoB (1, [2, 4]). We now have nodes 3 and 4 in stage 1, nobody in stage 2, node 1 in stage 3, and node 2 in stage 4. Upon exchanging their proofs of belonging, 3 and 4 realise something is wrong. As their lists first differ in position 1, node 3 remains in stage 1 and node 4 goes to stage 2, thus preserving the conservation of nodes, but potentially increasing the optimisation function.

The algorithm proceeds as follows. Nodes periodically calculate their current cost to the k closest peers in the same stage as them ($cost_{current}$). If k is set to infinity, it will calculate the distance to all its neighbours in the stage. Then they attempt to swap stages with another peer, i.e. node i moves to the stage of node j and vice-versa. With probability of 0.5, each node samples randomly a few peers it knows its cost to. For each peer, it compares its data parallel cost if it were in that peer's stage ($cost_{new}$) to its current one ($cost_{current}$). If it is lower - a swap is possible. If multiple minimising swaps are possible, the one that minimises by the largest amount is chosen. If no minimising swap can be found, a swap is possible with each

sampled peer with some probability:

$$e^{\frac{cost_{current}-cost_{new}}{T}} > U(0,1),$$

where T is the current temperature.

An offer is sent to that node with proof of costs to currently known nodes in their stage (with their counter), a list of the currently known nodes in the stage as the one making the offer (with their counter), the offerer's capacity, the flow required by the offerer, the offerer's temperature, the number of peers the offerer should have in its stage κ, and its proof of belonging. Upon receiving an offer, a node first checks if it is not with outdated information (based on the counter). If it is outdated, it simply ignores it, as it knows of a more recent message from that node. If it currently has an outstanding offer it will reject the incoming one, unless it is to the node, it currently is receiving an offer from (both parties have agreed that the switch is good for them). The node will then update its known peers and their stages based on the two lists received. The recipient then calculates the proposer's sum of k smallest costs to nodes in the recipient's stage. Additionally, it performs the same calculation for its sum of k smallest costs in the proposer's sum. These constitute the new possible data parallel communication costs of the two nodes. If both new costs are lower than the two current ones, it is accepted, and a switch is performed - the recipient sends its stage, proof of belonging, temperature, the number of peers it should have in its stage κ, and its counter. If not, same as before, it is accepted with some probability. Once two nodes switch, they use the information provided to them by the other node to continue their execution (known peers in that stage, proof of belonging, flow required, etc). They both update their respective temperature to the minimum of their temperatures prior to the switch. Finally, after finalising their switch locally, each node broadcasts to its new peers the change that has occurred to the nodes in their old stage. If their peers have also switched with some other node, they forward this message to the correct recipient. If the switch is not accepted, the node sends a reject message, which also contains a list of the nodes it currently knows in its stage with their counters.

At any time a peer must know of exactly the same numbers of peers in its stage as the node it represented knew when the algorithm started. This is the κ exchanged by the two nodes. If it knows more or less, then the second constraint is violated and it should wait before switching until it has been resolved. Nodes who are offline are still "known" in the stage they disconnected from. An example of a switch between two nodes in two different stages is shown in Figure 7.

Nodes not participating in any stage (or here the special stage 255), have a communication cost to their peers of 0. Nodes in that stage do not generate offers, but instead send their information to a node they wish to replace and await an offer from them. This is to ensure the preservation of the second constraint as the proof of belonging does not take into account this "hacky" stage.

4.6 New Node Joining

We design a two-way procedure to assign a new node to the stage that is predicted to be the bottlenecked one. First, the joining node estimates the stage with the highest utilized ratio, meaning the number of flows divided by the available capacity at the stage. Then, the joining node sends a proposal to an elected data node with its capacity, the stage it wishes to join, and estimated costs to other nodes in that stage. The data node decides to accept the proposal or not based on the estimated cost improvement.

Proposal to join a stage When a new (relay) node wants to participate in the training, it first needs to discover other nodes in the system through a Distributed Hash Table (DHT) [46].

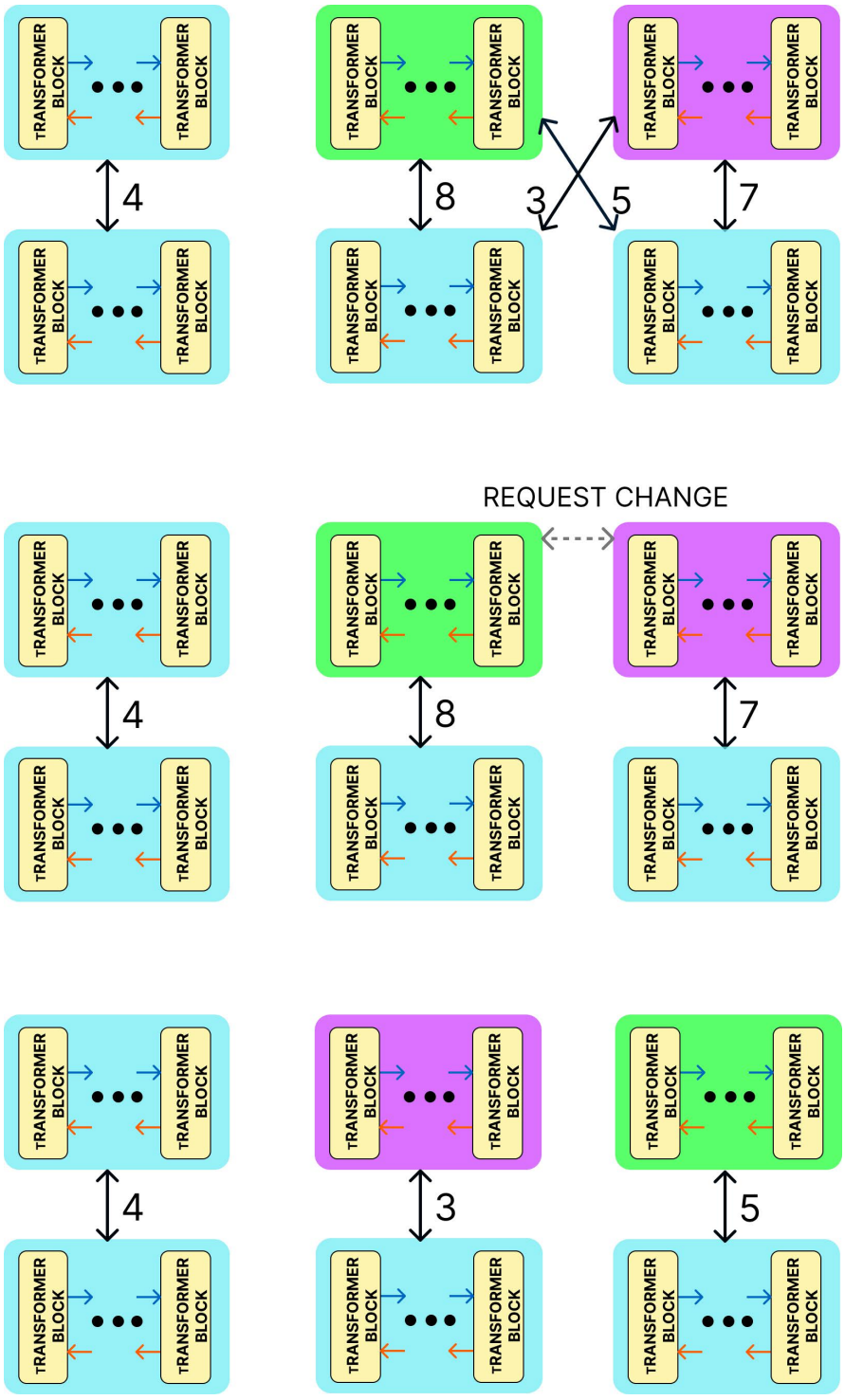


Figure 7: Example of swapping two nodes in two separate stages.

It queries existing nodes for the following information: their stage, microbatch flow capacity, and existing flows. Afterward, it computes the bottleneck factor for each stage - the ratio of its utilisation.

We first define the bottleneck factor (bf) of a stage s as the number of total microbatch flows divided by the entire capacity of a single stage.

$$bf_s = \frac{|F|}{\sum_{i \in s} cap_i}$$

The closer the bottleneck factor of a stage is to 1, the more it is a bottleneck. New nodes should join the stages with the highest value. Once nodes decide on a stage to join, they need to estimate three values. The first is the maximum cost of data parallel communication with a peer in the same stage. This value can be obtained through $d(i, j)$, where size refers to the size of the model in that stage. The second is the maximum communication of activations with any node from either the previous or subsequent stage (termed as the expected pipeline parallel cost). These are used to estimate the communication cost associated with using this new node in the training process. The last of the values it sends is its own capacity cap_i.

Selecting the new nodes Upon reception of such an offer, data nodes rebroadcast it to all known other data nodes and agree on which nodes' proposal to accept at a given stage. The key criterion is to choose a node's joining proposal that maximizes the ratio of the cost of microbatch flow divided by the overall microbatch throughput.

As the throughput and cost are dominated by the bottlenecked stage, we thus estimate the cost and throughput by the bottlenecked stage.

$$x = \text{argmax}_{s \in S}(bf_s)$$
$$z = \text{argmax}_{s \in S-X}(bf_s)$$

Where x is the stage with highest bottleneck factor, X are all stages with the same bottleneck factor as x, and z is the stage with the next highest bottle neck factor. If multiple stages have the same bf, then those closer to the first stage are preferred for node addition first.

The leader data node first computes the updated throughput for accepting new nodes into the stage x, which is determined by the stage of the smallest capacity. Thus, for each node that has sent an offer, the leader computes:

$$throughput_{new} = \min\left(\sum_{i \in x} cap_i + cap_{candidate}, \sum_{i \in z} cap_i\right)$$

The current throughput is equivalent to $\sum_{i \in x} cap_i$, as the system should be fully saturated. As the next node gets added, the throughput of the system cannot exceed the next bottleneck. The leader knows the current cost of training, which is bound by the most expensive flow times 2 (once forward, once back), which is the Pipeline Parallel cost (PPcost) plus the cost of the slowest intrastage communication, the Data Parallel cost (DPcost):

$$cost = 2 \cdot \max_{f \in F}(cost_f) + \max_{i \in N}(DPcost_{i,k})$$

Then the leader computes the updated microbatch cost, which is estimated as (i) the longest microbatch flow after including the new node and (ii) the worst intra stage communication time that is required to aggregate the results of nodes within a stage. As the actual maximum flow

cost depends on the actual routing, we can only estimate the overall flow cost by replacing the average cost per stage by the worst cross stage communication cost of adding the new node.

$$PPcost_{current} = \max_{f \in F}(cost_f) \tag{3}$$

$$PPcost_{new} = 2 \cdot PPcost_{current} - \left(\frac{PPcost_{current}}{|S|}\right) + \max\left(\frac{PPcost_{current}}{|S|},\ PPcost_{candidate}\right) \tag{4}$$

$$DPcost_{new,k} = \max\left(\max_{i \in N}(DPcost_{i,k}),\ DPcost_{candidate,k}\right) \tag{5}$$

$$cost_{new} = PPcost_{new} + DPcost_{new,k} \tag{6}$$

Where $|\mathbf{S}|$ is the number of stages. New nodes are then ranked based on:

$$\frac{cost}{throughput_{current}} - \frac{cost_{new}}{throughput_{new}}$$

The highest value greater than 0 is selected. This way nodes who will help increase the throughput without sacrificing the cost per microbatch are preferred. This process can be repeated iteratively for more nodes to be added in the next stage with highest bottleneck factor ($argmax_{s \in S} bf_s$ excluding stages that have had nodes added to them so far).

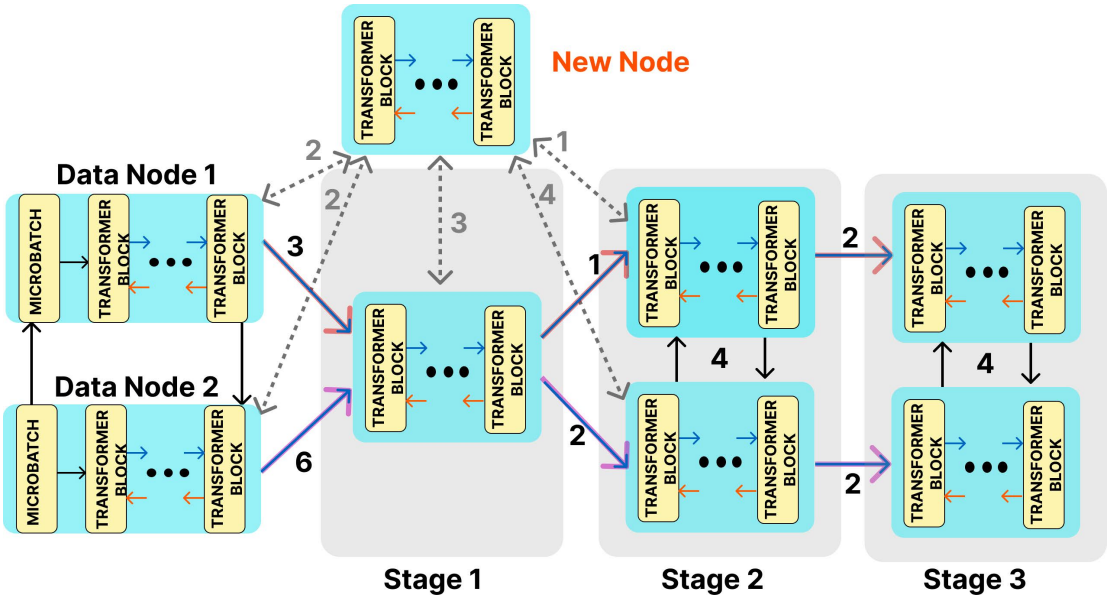


Figure 8: Example of adding a new node into stage 1 in the system that has two data nodes and three relay nodes. Circles represents the capacity of a node in terms of flows and edges represent the communication cost.

Running example We use a simple example to illustrate the computation of updated throughput and microbatch flow cost shown in Fig 8. There are 2 dataholder nodes, each sending single microbatch flow traversing in 4 stages. The circle in each stage denotes the node

and the value denotes the capacity. Stages 1, 2 and 3 have a total capacity of hosting 2, 3 and 5 microbatch flows. The existing cost of the two flows are 6 (top) and 10 (bottom).

A new node of capacity 5 proposes to join stage 1 - the bottlenecked stage. It incurs the following costs: (i) a cost of 3 in exchanging parameters with the node in stage 1; (ii) a cost of 2 in exchanging the activations of both dataholders; and (iii) two costs of 1 and 4 to exchange the activations of top and bottom node in stage 2, respectively. As a result, it also sends the estimated additional communication cost within a stage, which is 3, and the worst case additional cost to the overall flow, which is 4, i.e., the maximum of all additional communications across parent and children.

Upon receiving the proposal from this new node, the leader data node starts estimating the new throughput and the cost of including this new node. We note that multiple nodes can propose to join the same stage. The new throughout of stage 2 is thus 3, as the first stage is expanded to the capacity of $7 = 5 + 2$. It then estimates the new flow cost, which sums the maximum flow cost and the worst communication cost within the stage 3 that is still 5. The maximum flow cost is to replace the average per stage flow time by the worst flow time of introducing the new node, i.e., $10 - 10/4 + \max\{2, 2, 1, 4\}$. As a result, one can easily compute the new ratio of the flow cost divided throughput decreases.

4.7 Tolerating Crashes

When a node completes the computations of a batch, it sends back to the previous node in the path a COMPLETE message with the id of the batch. Based on this message, nodes can estimate the communication cost (network and computation latency) between them, though with one extra delay term, as what they will measure is $2 * \lambda + \frac{size}{\beta} + computation$. When sending training information, either during a forward or a backward pass, nodes expect to receive a COMPLETE reply from their peer within some fixed predefined amount of time. If they do not receive a reply, then the node assumes that the peer has left the system or is unresponsive and diverts its traffic to a different peer. If this is not possible, for example because it does not know any peers who can take in the traffic, it should send a DENY message to its upstream peer and let it redistribute the flow. This last operation can continue recursively until the source, which will have to leave this batch for the next iteration.

DENY messages essentially signal to an upstream peer that the subsequent node cannot process any more microbatches. This can happen because it doesn't know any nodes that it can send to, it has received DENY messages from every downstream peer or it has reached its maximum capacity. Nodes who have sent a DENY message are excluded from consideration until they send a CAN TAKE message (described in Section 4.8).

Forward pass Crash recovery during the forward pass of relay nodes is done as in SWARM [61]. 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, Go With The Flow uses the flow algorithm of Section 4.4, which can optimise costs of sending while also taking into account the memory constraints of each node.

Backward pass Nodes also send a complete message when they are done with their computation. If a node does not receive a COMPLETE BACKWARDS message or the transmission failed, it should inform the data node. If the data node has no outstanding microbatches for which they have not received a backwards pass, they do not need to restore the pipeline. In this case the node that detected the crash will not send the gradient of the activations, but will use the computed gradient to update its parameters during the aggregation phase. Otherwise, the

data node pings the node it sent the activations to with the microbatch path, which it knows as it had to compute to loss and it was the last one in the path. Nodes ping each other along the path. Upon a failed ping, the node that did not receive a ping forwards its activation to a new downstream peer with the path information. The downstream peer, upon calculating the activations on that layer, attempts to send the results to some node that was already on that path. If it does not receive a COMPLETE reply, it tries again to divert the traffic, until either reaching a node which was on the path and is still alive or reaching the data node, which is still technically a node on the path. Then the backwards pass can be resumed. Note that if a node earlier on from the original pipeline is found to be alive, then the process is significantly cheaper than recomputing the pipeline for that microbatch from scratch, which previous works would have to do.

If a node receives the same batch twice, before processing a backward pass, it can assume that one of the peers has recovered along the original path. This situation can happen, for example, when a node receives a COMPLETE message after it has timed out for a given microbatch. Thus now at least two nodes have computed activations for the same batch and they also remember the activations they have received. These can be freed (forgotten) by all but the first peer who sent the duplicate microbatch, to allow for more data through. The node that receives duplicate activations tell previous nodes whether they should forget their activations.

Aggregation When entering the aggregation phase, nodes first send out a STAKE message to nodes who should receive their parameters. If a node does not receive any STAKE messages before receiving a CAN TAKE message (explained in Section 4.8), then it can assume no nodes will be sending it their parameters, and thus they can end their aggregation phase. If a node has not received the parameters of a peer which has sent them a STAKE message when receiving a CAN TAKE message, then that node should perform a liveness check on its peer. If it is dead it is excluded from consideration during this aggregation phase.

4.8 Training-Aggregation Synchronization

As we have seen, when a node wants to join the system it runs the procedure of Section 4.6. Once it has been approved to join, it begins running the flow algorithm of Section 4.4 in parallel to the actual training. Every few iterations, in parallel to the routing and the training, it runs the algorithm of Section 4.5. When a fault occurs it runs one of the procedures of Section 4.7. In a decentralized training system, nodes also have to alternate between training and aggregation phases, which has been insufficiently described in previous works [61] and in previous sections. This synchronisation is necessary as nodes in the same stage need to have identical parameters when processing microbatches in a iteration. Go With the Flow relies on a simple algorithm through which nodes can signal to each other when this transition should happen.

When a node has reached its capacity of microbatches processed and has computed the backwards pass on all of them, it sends its parameters to other nodes in the same stage as it, as per [9, 36, 15], and enters the aggregation phase. Otherwise, the aggregation phase can also begin when the elected data node leader announces a starting aggregation to all its peers (using a BEGIN_AGGREGATION message), which propagate it further through the network. Upon receiving this message, nodes broadcast their model weights within their stage and collect their peers' weights. Once a node has finished its aggregation phase and knows of a downstream peer that also completed theirs, then it sends to its upstream peers another message (CAN_TAKE) that indicates that it accepts new microbatches. Nodes from the last stage do not need to wait for a CAN_TAKE message before sending their own. Thus the whole system involves several passes: a back to front formation of pipelines, a front to back forward pass, a back to front

backwards pass, a front to back relaying of BEGIN_AGGREGATION messages, and finally a back to front relaying of CAN_TAKE messages, which signals that a node has finished its aggregation phase and can begin the next iteration. This communication pattern is illustrated in Figure 9.

5 Evaluation

We demonstrate that our solution can provide significant speed up for training a large model in decentralized settings, even in the presence of node crashes. We show that our system is up to 45% faster in end-to-end runtime in the presence of faults compared to previous works. We further compare the performance of our scheduler against the optimal communication scheduler of Yuan et al.'s DT-FM [79]. We finally verify the applicability of our system in a practical scenario by showing that the model converges at rate similar of the one of GPipe [34].

5.1 Setup

The following experiments were performed on LLama architecture models with varying model sizes. We use the DAS6 cluster [5] of 5 NVIDIA RTX A4000 GPUs with 16 GB of GPU memory that are interconnected by a 100 GB Infiniband connection. We simulate geo-distributed locations by limiting the bandwidth and increasing the latency between nodes, with a maximum bandwidth between two nodes in simulated different locations of 500Mb/s. All systems were built on top of DecCom [69], as it allows for modular protocol stack construction and peer to peer communication. Details for the protocol stacks are presented in Appendix B. Node activity (uptime and downtime over the iterations) is presented in Appendix C.

To achieve a higher throughput per node, we make use of memory offloading by storing the computation graph and received model parameters/gradients to RAM. Unless stated otherwise, our decentralized optimization uses $T = 1.7$ for the initial temperature, $\alpha = 0.95$ for the cooling factor, and the aggregation group size was set to $k = \infty$. As the number of nodes per stage is relatively small (maximum 3 nodes per stage) and the number of crashes per iteration was often too high, the graph partitioning algorithm could not introduce any substantial benefit. As such, we disable it, unless explicitly specified.

5.2 Node Crashes

We evaluate the performance of our system against SWARM [61] on a small LLaMa-like model with $d_{model} = 1024$, $n_{heads} = 18$ and 16 layers. Microbatches used were of size 4 and sequence length of 512. As the model and batch sizes are significantly smaller than those in practice, we compensate by decreasing the bandwidth between nodes by a factor of 250, which is equivalent to sending an activation 250 times larger. This is necessary as the sequence length is 8 times smaller ([70] trains on a lenght of 4096), the activations sizes were roughly 12 times smaller than for actual LLaMa models of size 13b, and microbatches usually include more examples. Each setting has a world size of 18 nodes (maximum nodes that can be active at the same time). The model was split across 6 stages each servicing three transformer blocks, except the very first one which serviced 1 transformer block, the embedding layer, and handled loss computation and data retrieval. Two data nodes participate in the setting and do not leave the system until the end. They each try to push 4 microbatches during each iteration.

We compare the performance against SWARM [61], the state of the art in decentralized crash tolerant training. For a fair comparison, we modified SWARM to allow node crashes and joining of nodes. The SWARM baseline includes an additional timeout if no backwards pass

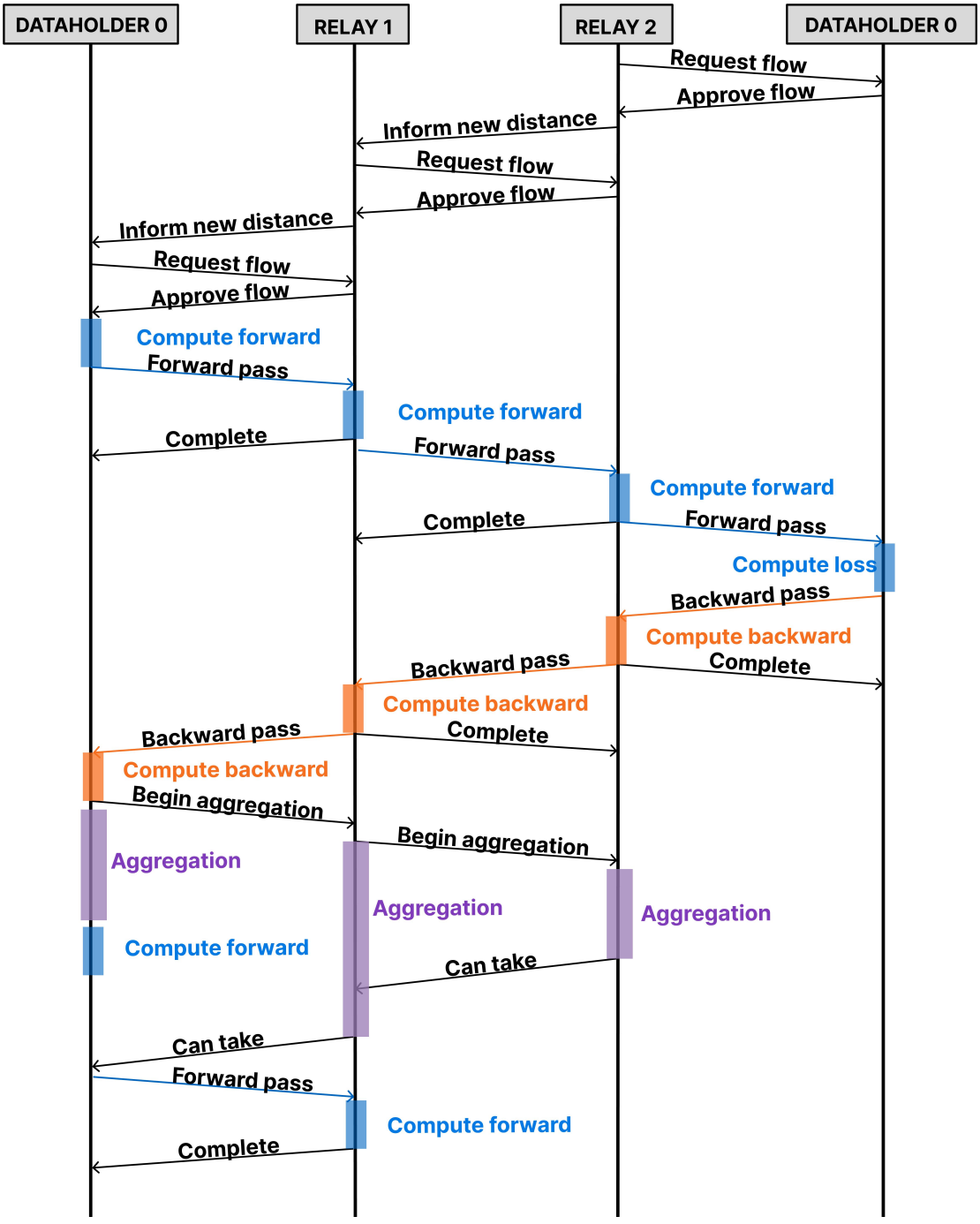


Figure 9: Communication and training happening during one iteration in a simple 3-stage pipeline without any crashes and with 1 microbatch per iteration. For ease of visualisation messages that relate to the aggregation phase are omitted.

has been received in some predefined amount of time by a data node. This triggers SWARM to rewire the nodes stochastically. Additionally, nodes can join freely in any stage they choose.

Parameters vary between experiments. The first difference is capacity. In the heterogeneous setting, relay nodes have random capacities between 1 and 3. In the homogeneous case all nodes have a capacity of 4. The second parameter is the join-leave chance. It dictates each node's chance to crash or join during any iteration. At 0%, nodes do not leave the system. At 10% they have a ten percent chance to join the system if currently inactive, or ten percent chance to crash during any iteration, if currently active. Results are presented in Table 2. Several metrics are reported (averaged across all iterations):

- Time per microbatch - the maximum time per iteration, measured from the point of view of a dataholder, divided by the number of microbatches processed in that iteration.
- Throughput per iteration - the number of microbatches processed in an iteration.
- Microbatch per time unit - the inverse of time per microbatch.
- Communication time - the summed pipeline parallel communication time for all microbatches in an iteration in minutes.
- Wasted GPU time - the summed pipeline parallel computation time in minutes for all microbatches in an iteration number, which was not included in the aggregation step or was not on the main path of a microbatch.

In all heterogeneous cases we demonstrated an improvement over SWARM and better performance even in the presence of crashes (speed up of 45% in the 10% crash case). In the homogeneous case Go With The Flow outperforms SWARM in the presence of crashes and performs similarly in the fault-free case. SWARM also has a much higher GPU wasted time, as microbatches may be sent to nodes that do not know anyone who they can send to, or on backward pass faults, the entire pipeline needs to be recomputed.

5.3 Optimality

In this section we aim to evaluate the performance of our scheduling routines, by comparing the end-to-end training time of our system against a GPipe setting with a communication optimal arrangement, calculated by the procedure DT-FM designed by Yuan et al. [79]. We additionally evaluate the performance of Go With The Flow with a working decentralised graph partitioning algorithm (Our work GP). The setting of the experiment is the same as the 0% homogeneous of the previous experiments, with some differences. Following the settings of [79] 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. The end-to-end training time between the two systems is compared in Table 3. Although the optimal computation schedule does outperform our work by almost 13%, it takes much longer time to be computed, as it involves the use of a genetic algorithm [79]. As it scales exponentially with the number of nodes it is not sustainable for large systems or systems that change ad-hoc.

5.4 Scalability with Model Size

Additionally, we evaluated the training loss of our system in a 10% heterogeneous setting against a single GPipe pipeline of 8 nodes, with 8 microbatches of size 1 and sequence length 4096. We evaluated against a setting of 10% node crashes with a maximum world size of 10 and 1 data

Table 2: Performance with crash-prone devices. Results are over 25 iterations. Per column the mean of the iterations is presented followed by the standard deviation

	Homogeneous 0%		Homogeneous 10%		Homogeneous 20%	
	SWARM	Ours	SWARM	Ours	SWARM	Ours
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	$\mathbf{8.0 \pm 0.13}$	7.16 ± 0.17	4.64 ± 0.87	$\mathbf{5.8 \pm 0.37}$	$\mathbf{6.1 \pm 1.3}$	5.72 ± 0.43
Microbatch per time	$\mathbf{1.95 \pm 0.26}$	1.8 ± 0.36	$\mathbf{1.14 \pm 0.53}$	$\mathbf{1.14 \pm 0.45}$	0.85 ± 0.43	$\mathbf{1.03 \pm 0.5}$
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%	
	SWARM	Ours	SWARM	Ours	SWARM	Ours
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	2.96 ± 0.21	$\mathbf{3.6 \pm 0.44}$	1.56 ± 4.08	$\mathbf{2.08 \pm 0.93}$	1.72 ± 3.56	$\mathbf{2.12 \pm 2.06}$
Microbatch per time	0.64 ± 0.09	$\mathbf{0.96 \pm 0.26}$	0.36 ± 0.21	$\mathbf{0.47 \pm 0.16}$	0.3 ± 0.2	$\mathbf{0.4 \pm 0.22}$
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 3: Comparison against optimal schedule

	Time per microbatch	Throughput per iteration	Microbatch per time
DT-FM [79]	0.44 ± 0.0	9.0 ± 0.0	2.27 ± 0.010
This work	0.51 ± 0.08	8.08 ± 0.08	1.99 ± 0.26
This work GP	0.62 ± 0.01	9.0 ± 0.01	1.62 ± 0.04

node. The model used in this experiment is the LLaMA-7b with $d_{model} = 4096$, $n_{heads} = 32$, and 32 layers, distributed uniformly across 8 stages. The dataset used was the Wikipedia English dataset [24]. Figure 10 shows that the larger system deployed by our system has a similar convergence of loss as the centralised GPipe pipeline.

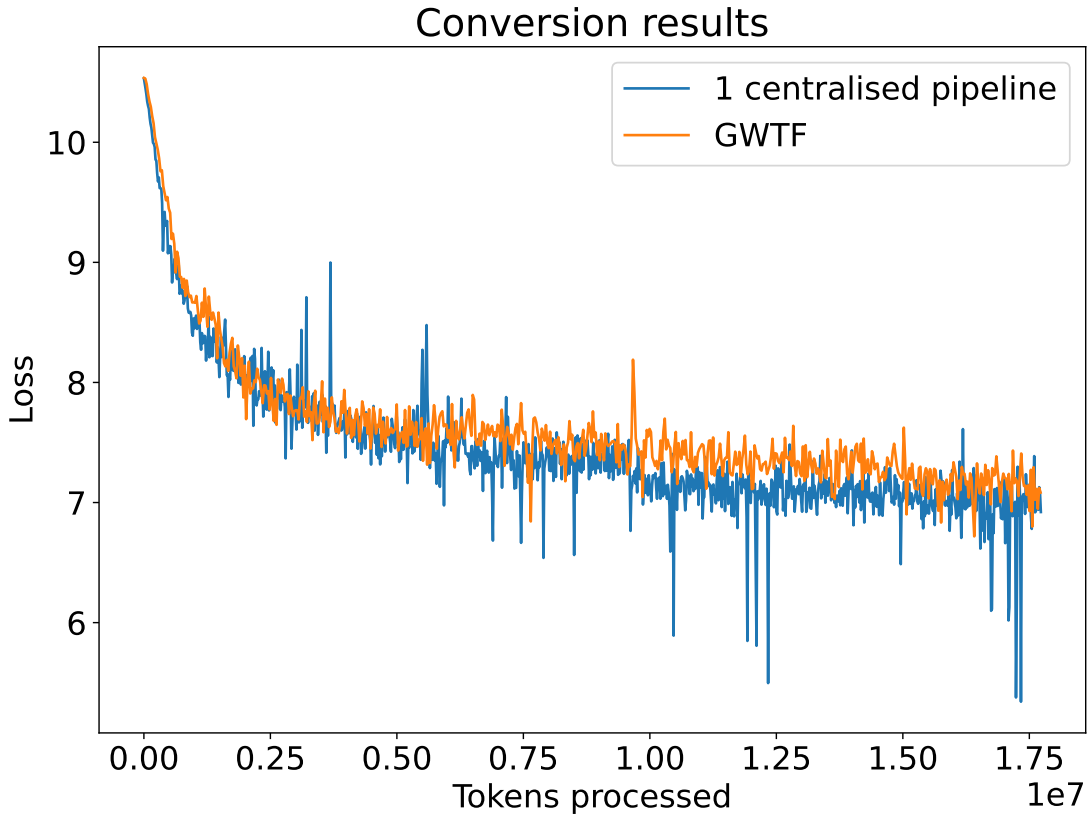


Figure 10: Loss convergence of Go with The Flow compared to the convergence of a single centralised GPipe pipeline

5.5 Decentralized Minimum Cost Flow

We evaluate our flow algorithm in 6 different settings. The first four settings involve a single source-sink node while the last have multiple source-sink nodes. Details about the tests are presented in Table 4. In all cases the source-sinks were given sufficient capacity to prevent bottlenecks. In order to compare to the optimal result as found by Fulkerson's algorithm [26], here our procedure attempts to minimise $min\left(\sum_{i,j\in\mathbf{N}} d(i,j) * f(i,j)\right)$, i.e. minimise the sum of costs of all flows. Tests 5 and 6 are not compared with the optimal baseline, as there the formulation differs, in that that a source must deliver to a specific sink. We use the approach from SWARM [61] of sending to the next stage closest node as a baseline. The results of the first two tests achieved difference from the optimal flow cost are presented in Figure 11 as fractions of the optimal cost.

Table 4: 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	$\lfloor U(1,3) \rfloor$	$\lfloor U(1,20) \rfloor$

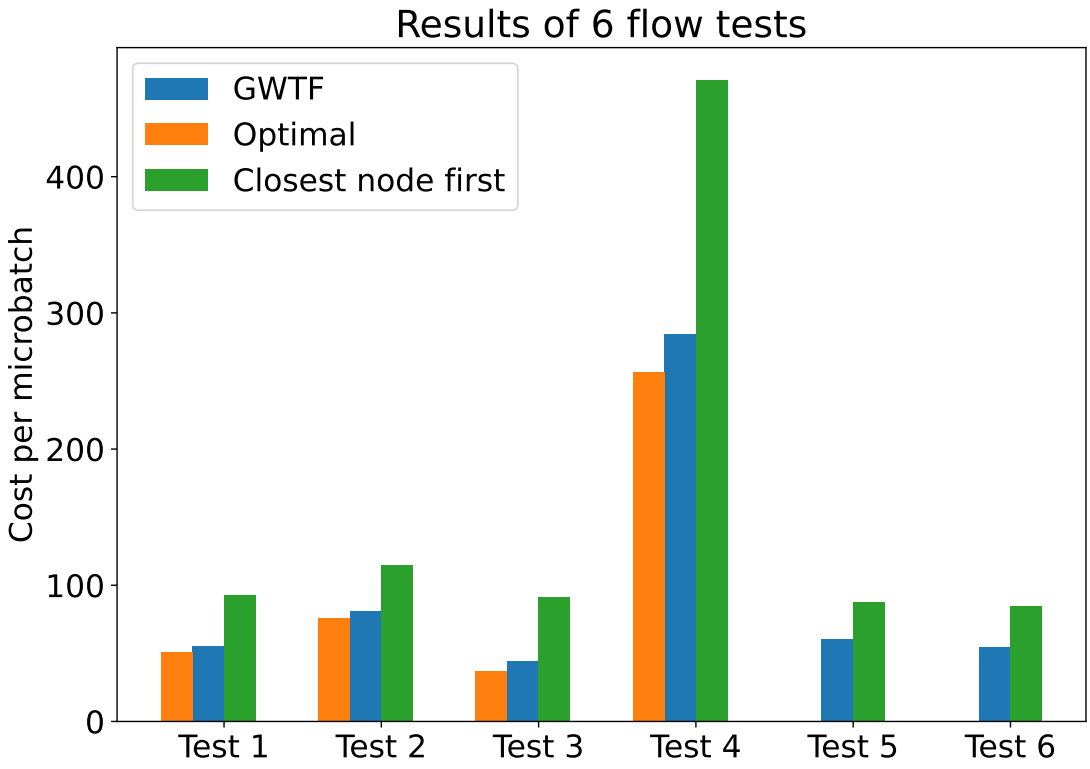


Figure 11: Average cost per microbatch in flow tests.

5.6 Handling Joining Nodes

We evaluate the algorithm for joining nodes, which is described in section 4.6. It is evaluated 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 5. To construct the system, we use the Out-of-kilter algorithm [26] to determine the minimum cost flow. Iteratively, 20 nodes are added in different stages following the algorithm in section 4.6. Since every node is a candidate for each stage, we assume that

Table 5: Node Addition Test Settings. Note that 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	$\lfloor U(1,20) \rfloor$	$\lfloor U(1,100) \rfloor$	$\phi + \lfloor U(50,100) \rfloor$

each node knows its costs for each stage. The optimal choice of node addition is determined by running the minimum cost flow algorithm [26] for each combination of S candidate nodes added to each of the S stages in the tests.

The results are presented in Figure 12 where the average improvement of 10 runs for each test is measured as $\frac{cost_{now} - cost_{after}}{cost_{now}}$ (i.e., the proportional improvement compared to before running the respective algorithm). We provide two baselines - adding highest capacity first and adding random nodes.

In all cases our work 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

5.7 Additional Experiments

As stated, in Table 3 we include an additional experiment which compares the performance of our system with the additional procedure of distributed graph partitioning. In this section we present several other experiments not included in the original paper.

5.7.1 Distributed Graph Partioning

We evaluate the distributed algorithm for balanced graph partitioning, as presented in 4.5. We evaluate it on two settings - one with 24 nodes and one with 48 nodes. For both, 6 geo-distributed locations are selected, with nodes assigned uniformly across them. Nodes are initially assigned to a random stage. The cost between nodes is defined as the time for sending the stage parameters, as per Equation 1. The algorithm aims to split the graph into 6 partitions. Initial temperature T was set to 3 and the cooling factor α to 0.95. The algorithm ran for 310 iterations for each case. Results are presented in Figure 13, where for both the average and the minimum maximum Data Parallel cost in a stage ($\max_{i \in N}(DPcost_{i,k})$) is reported over 5 runs. For the 24 node case we see on average a speed up of 4.3× times compared to before running the algorithm. For the 48 nodes, we can observe an average speed up of 4.4×.

5.7.2 GPT Architecture

In this section we repeat the experiments of Section 5.2, however with a GPT-like model. Model hyper parameters are as follows - $d_{model} = 1024$, $n_{heads} = 16$ and 16 layers. The model is split

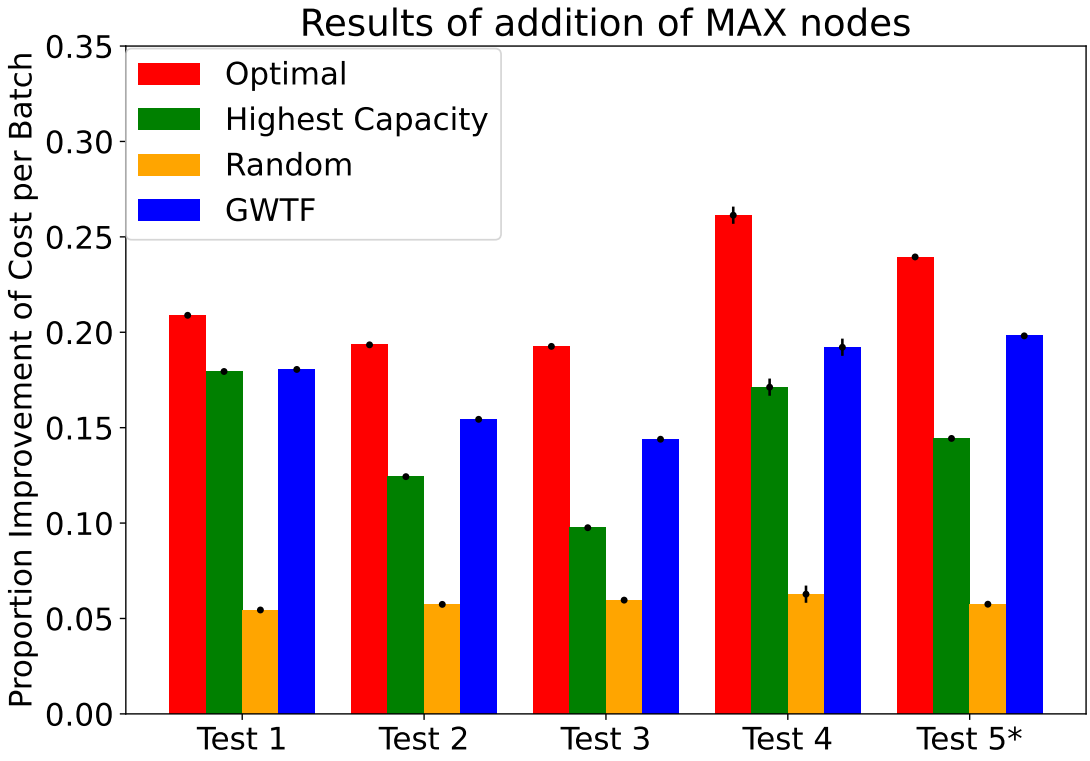
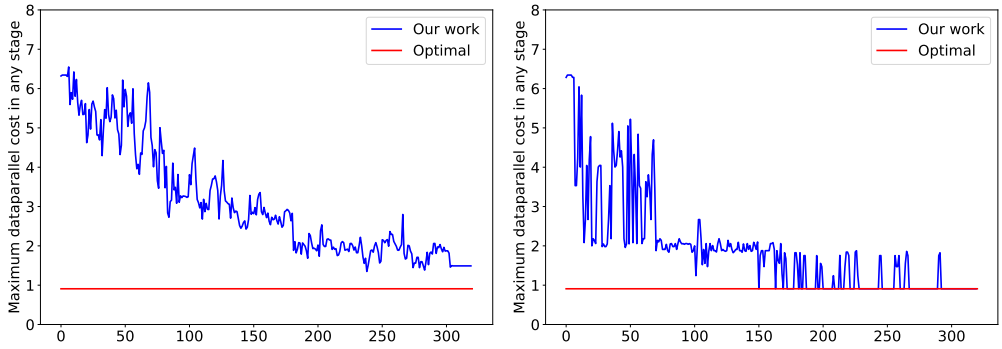


Figure 12: Average improvement of 10 runs for the new node addition tests. The variance is reported with black lines per bar. Higher means better.

across 6 stages in a system of a maximum of 18 nodes at a time. Microbatches used were of size 4 and sequence length of 512. Results are presented in Table 6.

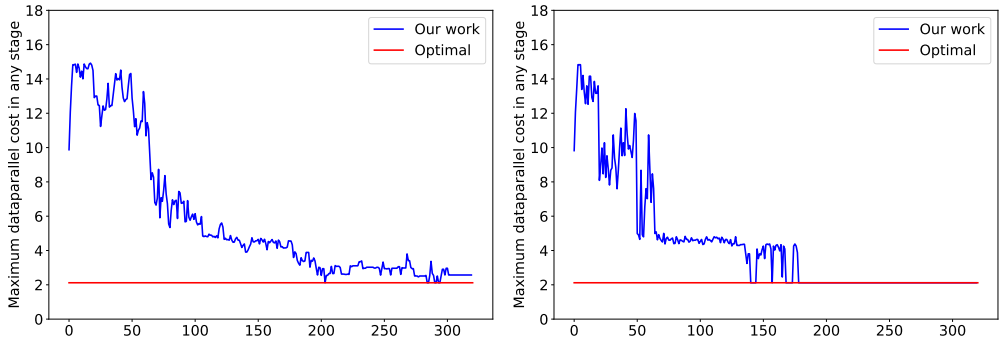
6 Conclusion and Future Work

In this work we present a novel decentralised crash tolerant training framework, which aims at minimising the time of training and increasing the throughput. We uniquely models the forward and backward pass of per training microbatch per iteration as a flow, and effectively decide its execution on clients that are of different computation and communication capacities. We describe three different procedures for minimising the training time, and two procedures for recovering from faults. We extensively evaluate Go With The Flow on decentralized training of LLama-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 our work is able to improve the training time by up to a 45% and throughput by up to a 30% increase while resulting into almost zero GPU wasted time of any joining clients. We additionally demonstrate that the system does not sacrifice convergence, even in the presence of faults.



(a) Average maximum Data Parallel cost across 5 runs in 24 node setting

(b) Minimum maximum Data Parallel cost across 5 runs in 24 node setting



(c) Average maximum Data Parallel cost across 5 runs in 48 node setting

(d) Minimum maximum Data Parallel cost across 5 runs in 48 node setting

Figure 13: Comparison in completion time for a single batch in 2 pipelines in the presence of faults

6.1 Ethical Consideration

We return back to the value analysis from the Introduction. We aimed at creating a system which satisfied three core values: right of individual expression, right of access to technology, and right to privacy. Via our framework, individuals with a common mode of expression or values can band together and utilise their independent and scattered resources to collectively train a shared model. All these individuals would require is a large corpus of data specifically targeted for their intended task. Alternatively, they can also employ parameter-efficient fine-tuning strategies to align a pretrained model with their task/values [18, 19, 32]. Since by design the system is decentralised (no central coordinator is required), individuals can train together a model, without the need of a large organisation to handle said training. The last of the issues is a bit harder to address. On one hand, data holders do not need to share their data with other participants. As such they preserve their privacy, should this data be sensitive. On the other, this also means that other participants have no way of verifying what data they are training on. This can present itself an issue, as a data holder might exploit individuals workers for a personal goal, under the guise of training for a common goal. We consider this an external issue to our work, and refer the read to the (albeit limited) literature on the manner. Model stealing is possible [53]. As such individuals could be able to obtain private data through inference on the model [76]. We attempt to remedy this issue by keeping the first and last few layers of the

Table 6: Performance with crash-prone devices for a GPT-like model. Results are over 25 iterations. Per column the mean of the iterations is presented followed by the standard deviation

	Homogeneous 0%		Homogeneous 10%		Homogeneous 20%	
	SWARM	Ours	SWARM	Ours	SWARM	Ours
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	$\mathbf{6.04 \pm 0.43}$
Microbatch per time	1.49 ± 0.18	$\mathbf{2.71 \pm 0.41}$	0.79 ± 0.39	$\mathbf{1.14 \pm 0.45}$	0.76 ± 0.42	$\mathbf{0.92 \pm 0.42}$
	Heterogeneous 0%		Heterogeneous 10%		Heterogeneous 20%	
	SWARM	Ours	SWARM	Ours	SWARM	Ours
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	$\mathbf{2.39 \pm 2.06}$
Microbatch per time	0.72 ± 0.06	$\mathbf{0.85 \pm 0.03}$	0.39 ± 0.21	$\mathbf{0.42 \pm 0.16}$	0.39 ± 0.28	$\mathbf{0.42 \pm 0.21}$

model only on the data holders.

6.2 Future Work

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 [41] 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.

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) [74]. Recent work on this matter assumes a stable, non-faulty, central node, which can store the check points [74, 22]. However, this is insufficient for our setting. Efficient decentralised checkpointing with crash-prone devices remains an unexplored topic.

Incentives Throughout this paper we assume volunteer nodes - nodes sparing some resources for no compensation. 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 [66], 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 [44, 77]

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 [33]. 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) [30]. For both scenarios, the framework presented in this work can in theory be utilised.

We hope that this works inspires many less privileged individuals who wish to develop and train Large Language Models, to unite their computing resources together and democratise access to these models. Volunteer workers of the world have nothing to lose but their idle GPU times.

References

[1] Abubakar Abid, Maheen Farooqi, and James Zou. Persistent anti-muslim bias in large language models. In Marion Fourcade, Benjamin Kuipers, Seth Lazar, and Deirdre K. Mulligan, editors, *AIES '21: AAAI/ACM Conference on AI, Ethics, and Society, Virtual Event, USA, May 19-21, 2021*, pages 298–306. ACM, 2021.

[2] Saleh Al-Takrouri and Andrey V. Savkin. A decentralized flow redistribution algorithm for avoiding cascaded failures in complex networks. *Physica A: Statistical Mechanics and its Applications*, 392(23):6135–6145, 2013.

[3] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*, pages 120–124. ACM, 2004.

[4] Eugene Bagdasaryan and Vitaly Shmatikov. Spinning language models: Risks of propaganda-as-a-service and countermeasures. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 769–786. IEEE, 2022.

[5] Henri E. Bal, Dick H. J. Epema, Cees de Laat, Rob van Nieuwpoort, John W. Romein, Frank J. Seinstra, Cees Snoek, and Harry A. G. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[6] Dario Bauso, Franco Blanchini, Laura Giarré, and Raffaele Pesenti. A decentralized solution for the constrained minimum cost flow. In *Proceedings of the 49th IEEE Conference on Decision and Control, CDC 2010, December 15-17, 2010, Atlanta, Georgia, USA*, pages 661–666. IEEE, 2010.

[7] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. In Danushka Bollegala, Ruihong Huang, and Alan Ritter, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics: System Demonstrations, ACL 2023, Toronto, Canada, July 10-12, 2023*, pages 558–568. Association for Computational Linguistics, 2023.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen,

Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[9] Marc Casas, Wilfried N. Gansterer, and Elias Wimmer. Resilient gossip-inspired all-reduce algorithms for high-performance computing: Potential, limitations, and open questions. *Int. J. High Perform. Comput. Appl.*, 33(2), 2019.

[10] Chang Che and Olivia Wang. What happens when you ask a chinese chatbot about taiwan?, Jul 2023.

[11] Tianshi Che, Ji Liu, Yang Zhou, Jiaxiang Ren, Jiwen Zhou, Victor S. Sheng, Huaiyu Dai, and Dejing Dou. Federated learning of large language models with parameter-efficient prompt tuning and adaptive optimization. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 7871–7888. Association for Computational Linguistics, 2023.

[12] Chaochao Chen, Xiaohua Feng, Jun Zhou, Jianwei Yin, and Xiaolin Zheng. Federated large language model: A position paper. *CoRR*, abs/2307.08925, 2023.

[13] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 612–623. IEEE, 2022.

[14] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023.

[15] Martijn de Vos, Sadegh Farhadkhani, Rachid Guerraoui, Anne-Marie Kermarrec, Rafael Pires, and Rishi Sharma. Epidemic learning: Boosting decentralized learning with randomized communication. *CoRR*, abs/2310.01972, 2023.

[16] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *J. Mach. Learn. Res.*, 13:165–202, 2012.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[18] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *CoRR*, abs/2203.06904, 2022.

[19] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nat. Mac. Intell.*, 5(3):220–235, 2023.

[20] Fyodor Dostoyevsky. *The brothers Karamazov*. Penguin Classics, London, England, 2024.

[21] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony (preliminary version). In Tiko Kameda, Jayadev Misra, Joseph G. Peters, and Nicola Santoro, editors, *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*, pages 103–118. ACM, 1984.

[22] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-n-run: a checkpointing system for training deep learning recommendation models. In Amar Phanishayee and Vyas Sekar, editors, *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*, pages 929–943. USENIX Association, 2022.

[23] Tao Fan, Yan Kang, Guoqiang Ma, Weijing Chen, Wenbin Wei, Lixin Fan, and Qiang Yang. FATE-LLM: A industrial grade federated learning framework for large language models. *CoRR*, abs/2310.10049, 2023.

[24] Wikimedia Foundation. Wikimedia downloads.

[25] D. R. Fulkerson. An out-of-kilter method for minimal-cost flow problems. *Journal of the Society for Industrial and Applied Mathematics*, 9(1):18–27, 1961.

[26] D. R. Fulkerson. An out-of-kilter method for minimal-cost flow problems. *Journal of the Society for Industrial and Applied Mathematics*, 9(1):18–27, 1961.

[27] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[28] Barbara B. Greene and Gerald M. Rubin. *Automatic grammatical tagging of English*. Dept. of Linguistics, Brown University, 1971.

[29] Karen Hao. Openai is giving microsoft exclusive access to its gpt-3 language model, Sep 2020.

[30] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, and Phillip B. Gibbons. Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377, 2018.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[32] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[33] Yang Hu, Connor Imes, Xuanang Zhao, Souvik Kundu, Peter A. Beerel, Stephen P. Crago, and John Paul Walters. Pipeline parallelism for inference on heterogeneous edge computing. *CoRR*, abs/2110.14895, 2021.

[34] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[35] David Hume. *A treatise of human nature*. Wildside Books, 2007.

[36] Jiyan Jiang, Wenpeng Zhang, Jinjie Gu, and Wenwu Zhu. Asynchronous decentralized online learning. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 20185–20196, 2021.

[37] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[38] Mark Johnson. How the statistical revolution changes (computational) linguistics. In Timothy Baldwin and Valia Kordoni, editors, *Proceedings of the EACL 2009 Workshop on the Interaction between Linguistics and Computational Linguistics: Virtuous, Vicious or Vacuous?*, pages 3–11, Athens, Greece, March 2009. Association for Computational Linguistics.

[39] Jakub Konečný, Brendan McMahan, and Daniel Ramage. Federated optimization: Distributed optimization beyond the datacenter. *CoRR*, abs/1511.03575, 2015.

[40] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[41] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[42] Angeliki Lazaridou, Adhiguna Kuncoro, Elena Gribovskaya, Devang Agrawal, Adam Liska, Tayfun Terzi, Mai Gimenez, Cyprien de Masson d'Autume, Tomás Kociský, Sebastian Ruder, Dani Yogatama, Kris Cao, Susannah Young, and Phil Blunsom. Mind the gap:

Assessing temporal generalization in neural language models. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 29348–29363, 2021.

[43] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 583–598. USENIX Association, 2014.

[44] Andrei Lihu, Jincheng Du, Igor Barjaktarevic, Patrick Gerzanics, and Mark Harvilla. A proof of useful work for artificial intelligence on the blockchain. *CoRR*, abs/2001.09244, 2020.

[45] Julie Beth Lovins. Development of a stemming algorithm. *Mech. Transl. Comput. Linguistics*, 11(1-2):22–31, 1968.

[46] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, volume 2429 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2002.

[47] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data. In Aarti Singh and Xiaojin (Jerry) Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282. PMLR, 2017.

[48] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proc. VLDB Endow.*, 16(9):2354–2363, 2023.

[49] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[50] Boaz Miller. Is technology value-neutral? *Science, Technology, & Human Values*, 46(1):53–80, 2021.

[51] Douglas Murray. *Bosie: A biography of lord Alfred Douglas*. Hodder amp; Stoughton, 2021.

[52] Safiya Umoja Noble. *Algorithms of oppression: How search engines reinforce racism*. New York University Press, 2018.

[53] Daryna Oliynyk, Rudolf Mayer, and Andreas Rauber. I know what you trained last summer: A survey on stealing machine learning models and defences. *ACM Comput. Surv.*, 55(14s):324:1–324:41, 2023.

[54] OpenAI. Introducing openai, Mar 2017.

[55] James B. Orlin. Max flows in o(nm) time, or better. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 765–774. ACM, 2013.

[56] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distributed Comput.*, 69(2):117–124, 2009.

[57] Sitti Rabiah. Language as a tool for communication and cultural reality discloser. *International Conference on Media, Communication and Culture*, 11 2018.

[58] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

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

[60] Fatemeh Rahimian, Amir Hossein Payberah, Sarunas Girdzijauskas, Márk Jelasity, and Seif Haridi. JA-BE-JA: A distributed algorithm for balanced graph partitioning. In *7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2013, Philadelphia, PA, USA, September 9-13, 2013*, pages 51–60. IEEE Computer Society, 2013.

[61] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. SWARM parallelism: Training large models can be surprisingly communication-efficient. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 29416–29440. PMLR, 2023.

[62] Max Ryabinin and Anton Gusev. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[63] Jean-Paul Sartre. *Existentialism is a Humanism*. Les Editions Nagel, Paris, France, 1946.

[64] Michael Shirts and Vijay S. Pande. Screen savers of the world unite! *Science*, 290(5498):1903–1904, 2000.

[65] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[66] Paul Sullivan. As bitcoin's price surges, affluent investors start to take a look. *New York Times*.

[67] Mattilda Bernstein Sycamore. *That's revolting!: Queer strategies for resisting assimilation*. Accessible Publishing Systems Pty, Ltd., 2010.

[68] The Bing Team.

[69] The Worker Thread, 2024.

[70] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[71] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[72] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[73] JP Vergne. Decentralized vs. distributed organization: Blockchain, machine learning and the future of the digital platform. *Organization Theory*, 1(4):2631787720977052, 2020.

[74] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: fast failure recovery in distributed training with in-memory checkpoints. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 364–381. ACM, 2023.

[75] Laura Weidinger, Maribeth Rauh, Nahema Marchal, Arianna Manzini, Lisa Anne Hendricks, Juan Mateos-Garcia, A. Stevie Bergman, Jackie Kay, Conor Griffin, Ben Bariach, Iason Gabriel, Verena Rieser, and William Isaac. Sociotechnical safety evaluation of generative AI systems. *CoRR*, abs/2310.11986, 2023.

[76] Laura Weidinger, Jonathan Uesato, Maribeth Rauh, Conor Griffin, Po-Sen Huang, John Mellor, Amelia Glaese, Myra Cheng, Borja Balle, Atoosa Kasirzadeh, Courtney Biles, Sasha Brown, Zac Kenton, Will Hawkins, Tom Stepleton, Abeba Birhane, Lisa Anne Hendricks, Laura Rimell, William Isaac, Julia Haas, Sean Legassick, Geoffrey Irving, and Iason Gabriel. Taxonomy of risks posed by language models. In *FAccT '22: 2022 ACM Conference on Fairness, Accountability, and Transparency, Seoul, Republic of Korea, June 21 - 24, 2022*, pages 214–229. ACM, 2022.

[77] Jiasi Weng, Jian Weng, Ming Li, Yue Zhang, and Weiqi Luo. Deepchain: Auditable and privacy-preserving deep learning with blockchain-based incentive. *IACR Cryptol. ePrint Arch.*, page 679, 2018.

[78] Terry Winograd. *Procedures as a representation for data in a computer program for understanding natural language*. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1971.

[79] Binhang Yuan, Yongjun He, Jared Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Ré, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments. In *NeurIPS*, 2022.

[80] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[81] Pan Zhou, Qian Lin, Dumitrel Loghin, Beng Chin Ooi, Yuncheng Wu, and Hongfang Yu. Communication-efficient decentralized machine learning over heterogeneous networks. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 384–395. IEEE, 2021.

[82] Michael Zimmer. *The Gaze of the Perfect Search Engine: Google as an Infrastructure of Dataveillance*, pages 77–99. 01 2008.

A System Messages

- **INTRODUCTION** - Sent when a node discovers a new node via the DHT. Includes the node's current stage, capacity, flow.
- **COMPLETE** - Message sent by a downstream peer i to an upstream peer j if i had received in this iteration a microbatch from j and i has completed processing the microbatch
- **BACKWARD COMPLETE** - Message sent by an upstream peer j to an upstream peer i if j had received in this iteration a backwards pass for a microbatch from i and j has completed processing the backwards pass.
- **DENY** - Message sent if it cannot process a given microbatch. This can occur if its memory is full or doesn't know any downstream peer to which it can sent afterwards. Receiving this message means that the microbatch has to be sent to a different peer.
- **CAN TAKE** - Signals to an upstream peer that the sender can now take microbatches. Sent if previously has sent a DENY or if a new iteration has begun and the sender has completed the aggregation phase. Can contain a list of flows - those which the node can currently take until further notice.
- **BEGIN AGGREGATION** - Signals to all nodes that they should enter the aggregation phase
- **REQUEST FLOW** - An upstream peer requests flow from a downstream one. Contains the cost of the flow requested, the intended data node (sink node), and the unique flow id as generated by the upstream peer.
- **APPROVE FLOW** - A downstream peer *accepts* a previously requested flow. Contains the cost of the flow requested, the intended data node (sink node), and the unique flow id as generated by the upstream peer.
- **REJECT FLOW** - A downstream peer *rejects* a previously requested flow. Contains the new lowest cost to the intended data node (sink node) and the intended data node (sink node).
- **PROPOSE CHANGE** - A node offers a flow exchange to another node, which has flow to a different peer but the same intended data node (sink node). Contains the intended data node, the sender's current downstream peer, the sender's intended next downstream peer, the two costs it hast to the downstream nodes, and the unique id for the flow it has to its downstream peer.
- **ACCEPT CHANGE** - Accepts a change offer. Accompanied by the sender's previous downstream peer (now the recipient's new downstream peer) and the sender's unique flow id to that peer.
- **REJECT CHANGE** - Rejects a redirect change.
- **PROPOSE REDIRECT** - A node offers a flow redirect to another node. Contains the intended data node, the sender's intended new upstream and downstream peer an the two costs it hast to the two nodes.

- **ACCEPT REDIRECT** - Accepts a redirect offer. Accompanied by the sender's previous downstream peer (now the recipient's new downstream peer) and the sender's unique flow id to that peer.
- **REJECT REDIRECT** - Rejects a redirect offer.
- **PUSHBACK FLOW** - A downstream peer pushes back the flow of an upstream peer. This flow can currently not go through that node and the upstream peer is responsible for it - figuring out to whom it sends it, if it pushes it back, etc. Contains the upstream peer's unique id.
- **CANCEL FLOW** - An upstream peer cancels the flow it had through a downstream peer. The downstream peer removes it from its inflow. Contains the upstream peer's unique id.
- **PROPOSE STAGE SWITCH** - A node sends to another node an offer to switch their stages. Contains the sender's distance to each node it knows in the recipient's stage, a list of nodes and their counters in the sender's stage, a list of nodes and their counters in the recipient's stage, the sender's proof of belonging, the sender's stage.
- **ACCEPT STAGE SWITCH** - Accepts a stage switch. Contains the sender's proof of belonging.
- **REJECT STAGE SWITCH** - Rejects a stage switch. Includes the sender's same stage peers with their counters.
- **CONTEST PROOF OF BELONGING** - A node informs another about a discrepancy detected based on the proof of belonging of the recipient and the sender. Contains the sender's proof of belonging.
- **SYSTEM CHECK** - Sent by a leader periodically. Contains a unique ID and the leader's ID. When received, a node generates an **INFORM** message, unless it has previously sent such a message with the same unique and leader IDs.
- **INFORM** - Generated when receiving a **SYSTEM CHECK** or another **INFORM** message. Contains the unique and leader IDs of the **INFORM** message which started this process. Additionally, a node puts its current stage, ID, capacity, flow, and Data Parallel cost within the message. Sent to all downstream peers.
- **REQUEST TO JOIN** - Sent by a candidate node to a known data node, which will broadcast it to other data nodes. Contains the stage a node wishes to join, its capacity, and the maximum Data Parallel and Pipeline Parallel costs to other nodes if it were to join that stage.
- **JOIN** - Sent by a data node to a candidate node. Contains the stage the candidate has been permitted to join.

B Protocol stacks

Estimating the delays between nodes is outside of the scope of this paper and we assume they are known apriori by each node with some additional noise.

B.1 Go With The Flow

The protocol stack is (in bold the implemented protocols for this experiment) - UDP Transport, Kademlia Discovery (k = 20, update interval = 30 seconds), Noise Protocol (not strict, no encryption), TCP Stream Protocol, TCP Delay, **Flow protocol**, **Training protocol**, **Control protocol** responsible for node joining/leaving/starting the different sub procedures.

B.2 SWARM

The protocol stack is (in bold the implemented protocols for this experiment) - UDP Transport, Kademlia Discovery (k = 20, update interval = 30 seconds), Noise Protocol (not strict, no encryption), TCP Stream Protocol, TCP Delay, **Training protocol** as described in SWARM [61], **Control protocol** responsible for node joining/leaving/starting the different sub procedures.

B.3 GPipe

The protocol stack is (in bold the implemented protocols for this experiment) - UDP Transport, Kademlia Discovery (k = 20, update interval = 30 seconds), Noise Protocol (not strict, no encryption), TCP Stream Protocol, TCP Delay, **Training protocol** which requires a known next and previous peer as well as known data-parallel group.

C Node participation

This appendix contains the figures of the physical nodes uptime and downtime throughout the system. In all cases, when a node comes back online again, it joins as a different logical node, prior to the one that was alive on that physical node. This way despite being limited to 18 physical nodes, the training involves an up to 100 logical nodes participating at different times. Figures 14, 15, and 16 present the node activity for the 0% crash, 10% crash, and 20% crash respectively. Numbers inside the cells represent the capacity of the logical node alive at that moment on the given physical one. Graphics for the homogeneous cases are the same, with the only difference that all nodes have a capacity of 4.

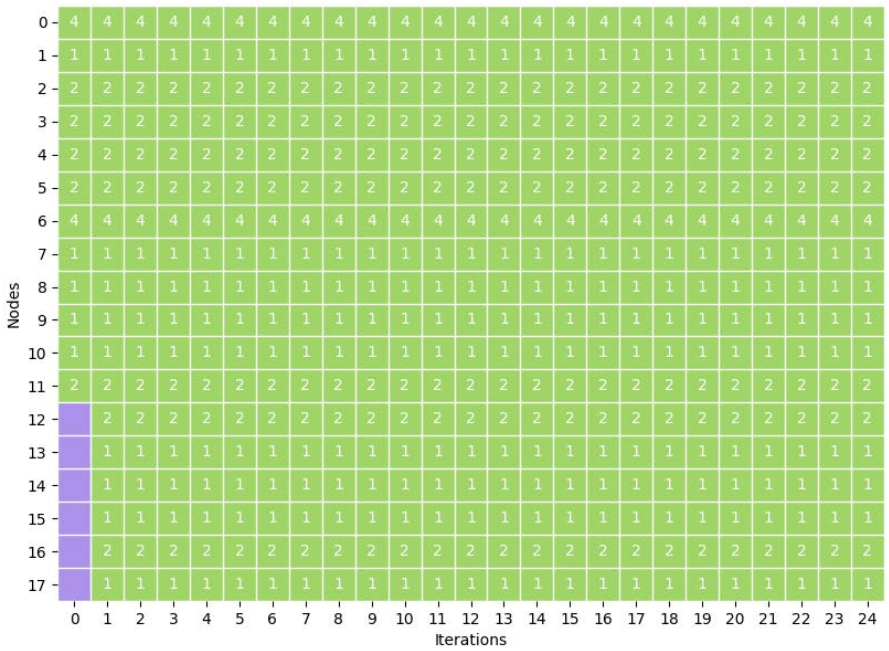


Figure 14: Node activity with 0% crash chance. Green indicates that a node is active at that iteration, purple - inactive, and red - crashed during that iteration

Algorithm 1 Initialising, Introduction, and Flow Request procedures of the flow algorithm.

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**
  $Outflow = \emptyset$ Outflow not in Inflow is outstanding outflow
  $Inflow = \emptyset$ Inflow not in Outflow is outstanding inflow
  $Sinks = \emptyset$
  $Next\ Stage = \emptyset$
  $Previous\ Stage = \emptyset$
  $Same\ Stage = \emptyset$
  $unmatched = 0$
  if *source* **then**
    $unmatched = cap$
    Add outflows to $Outflow$ to oneself
  if *sink* **then**
    $unmatched = -cap$
    Add inflows to $Inflow$ to oneself

upon event$\langle$`Flow`, `Introduction`$|j, stage, their\ unmatched\rangle$ **do**
  if $stage = S_i$ **then**
    $Same\ Stage \cup j$
  else if $stage = S_i + 1$ **then**
    $Next\ Stage \cup j$
  else if $stage = S_i - 1$ **then**
    if *nodeidissink* **then**
      $Sinks \cup j$
    $Previous\ Stage \cup j$
  Remember *their unmatched*

upon event$\langle$`DHT`, `New Peer`$|j\rangle$ **do**
  $\langle SEND$, `INTRODUCTION` $|j$, i, S_i, unmatched$\rangle$ Introduce ourselves

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

Algorithm 2 Redirect, Change, and Push Back procedures of the flow algorithm.

1: **upon event**$\langle$`Flow`, `UPDATE` *or* `REJECT`$|j, cost, target, unmatched\rangle$ **do**
2: Store node j having smallest cost to *target* of *cost* and unmatched of *unmatched*

3: **upon event**$\langle$`Flow`, `REDIRECT`$|j, k, l, their\ cost\rangle$ **do**
4: **if** don't have flow from k to l **then**
5: **return**
6: $cost_{current} = max(d(i,k), d(i,l))$
7: $cost_{new} = cost$
8: $improvement = cost_{current} - cost_{new}$
9: **if** $improvement > minimisation$ **then**
10: $\langle SEND,$ `NEW PARENT` $|l$, j, $d(j,l)$, flow to l $\rangle$
11: $\langle SEND,$ `PUSH BACK` $|k$, i, flow from k $\rangle$
12: $\langle SEND,$ `ACCEPT` $|j$, l, cost of flow to target through l - $d(i,l)$, target of flow $\rangle$
13: $T = T * \alpha$
14: **else if** $minimisation \leq 0$ *and* $e^{\frac{improvement}{T}} > U(0,1)$ **then**
15: $\langle SEND,$ `NEW PARENT` $|l$, j, $d(j,l)$, flow to l $\rangle$
16: $\langle SEND,$ `PUSH BACK` $|k$, i, flow from k $\rangle$
17: $\langle SEND,$ `ACCEPT` $|j$, l, cost of flow to target through l - $d(i,l)$, target of flow $\rangle$
18: $T = T * \alpha$

19: **upon event**$\langle$`Flow`, `NEW PARENT`$|j, their\ cost, flow\rangle$ **do**
20: Set the parent of $flow$ to j

21: **upon event**$\langle$`Flow`, `PUSH BACK`$|j$, flow to j,$\rangle$ **do**
22: $unmatched- = 1$
23: $Outflow = Outflow \setminus \{flow\ to\ j\}$

24: **upon event**$\langle$`Flow`, `CHANGE`$|j, k, l, their\ cost\rangle$ **do**
25: **if** don't have flow to k **then**
26: **return**
27: $cost_{current} = max(d(j,k), d(i,l))$
28: $cost_{new} = max(d(j,l), d(i,k))$
29: $improvement = cost_{current} - cost_{new}$
30: **if** $improvement > minimisation$ **then**
31: $\langle SEND,$ `NEW PARENT` $|k$, j, $d(i,k)$, flow to k $\rangle$
32: $\langle SEND,$ `ACCEPT CHANGE` $|j,$, k, l, cost of flow to target through k - $d(i,k)$, target of flow $\rangle$
33: Add outflow to l at cost reported by j
34: $T = T * \alpha$
35: **else if** $minimisation \leq 0$ *and* $e^{\frac{improvement}{T}} > U(0,1)$ **then**
36: $\langle SEND,$ `NEW PARENT` $|k$, j, $d(j,k)$, flow to k $\rangle$
37: $\langle SEND,$ `ACCEPT CHANGE` $|j,$, k, l, cost of flow to target through k - $d(i,k)$, target of flow $\rangle$
38: Add outflow to l at cost reported by j
39: $T = T * \alpha$

40: **upon event**$\langle$`Flow`, `ACCEPT CHANGE`$|k, l, cost\ to\ k\rangle$ **do**
41: $\langle SEND,$ `NEW PARENT` $|l$, j, $d(j,l)$, flow to l $\rangle$
42: Add outflow to k at cost reported by j
43: $T = T * \alpha$

Algorithm 3 Periodic procedure of the flow algorithm, which either requests more flow or minimised the cost of the current flow

1: **periodically**⟨`Flow`, `Periodic`⟩ **do**
2: $choices$ = nodes in $Next\ Stage$ with $unmatched < 0$
3: **if** $Inflow \setminus Outflow \neq \emptyset$ **then**
4: $choices$ = nodes in $choices$ such that they have flow to sink in $Inflow \setminus Outflow$
5: $chosen$ = node with smallest code to any target
6: **if** $chosen \neq \emptyset$ **then**
7: $\langle SEND,$ `Request` $|i, chosen$, smallest cost to target, target$\rangle$ Request flow to target
8: **else if** $chosen = \emptyset$ **then**
9: $minimisation = 0$
10: $request = \emptyset$
11: **forall** $j \in Same\ Stage$ **do**
12: **forall** k, l pairs of nodes $\in Next\ Stage$ such that j has flow to k and we have flow to l**do**
13: $cost_{current} = max(d(j,k), d(i,l))$
14: $cost_{new} = max(d(i,k), d(j,l))$
15: $improvement = cost_{current} - cost_{new}$
16: **if** $improvement > minimisation$ **then**
17: $minimisation = improvement$
18: $chosen = j$
19: $request = CHANGE$
20: **else if** $minimisation \leq 0$ *and* $e^{\frac{improvement}{T}} > U(0,1)$ **then**
21: $minimisation = improvement$
22: $chosen = j$
23: $request = CHANGE$

24: **forall** $j \in Same\ Stage$ **do**
25: **forall** k, l pairs of nodes $\in Next\ Stage$ such that j has flow from k to l**do**
26: $cost_{current} = max(d(j,k), d(j,l))$
27: $cost_{new} = max(d(i,k), d(i,l))$
28: $improvement = cost_{current} - cost_{new}$
29: **if** $improvement > minimisation$ **then**
30: $minimisation = improvement$
31: $chosen = j$
32: $request = REDIRECT$
33: **else if** $minimisation \leq 0$ *and* $e^{\frac{improvement}{T}} > U(0,1)$ **then**
34: $minimisation = improvement$
35: $chosen = j$
36: $request = REDIRECT$
37: **if** $request$ *is* $REDIRECT$ **then**
38: $\langle SEND,$ `REDIRECT` $|chosen$, flow from j and to l, our costs to them $\rangle$ Request redirect from node
39: **if** $request$ *is* $CHANGE$ **then**
40: $\langle SEND,$ `CHANGE` $|chosen$, flow from j, flow to l, our costs to them $\rangle$ Request change from node

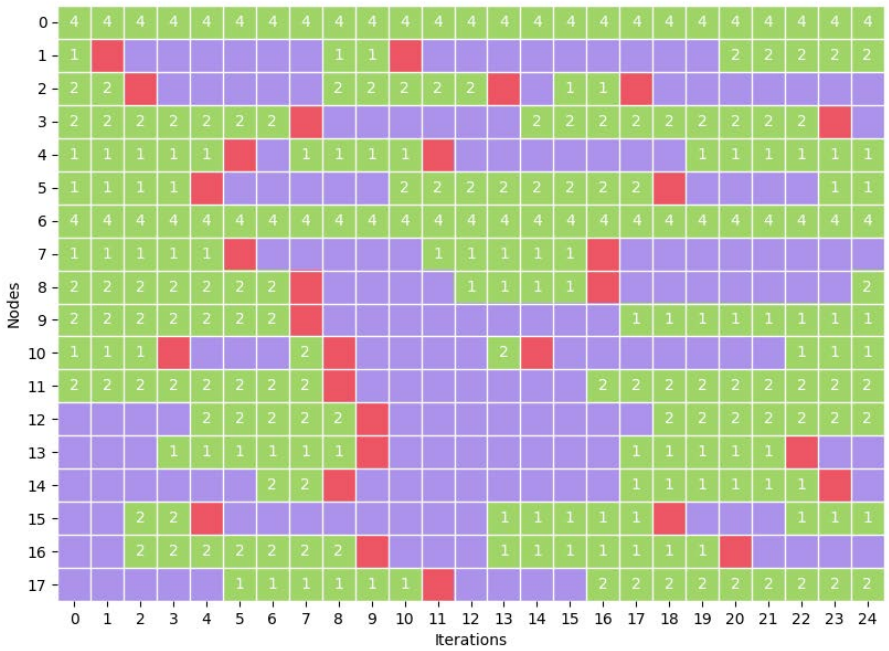


Figure 15: Node activity with 10% crash chance. Green indicates that a node is active at that iteration, purple - inactive, and red - crashed during that iteration

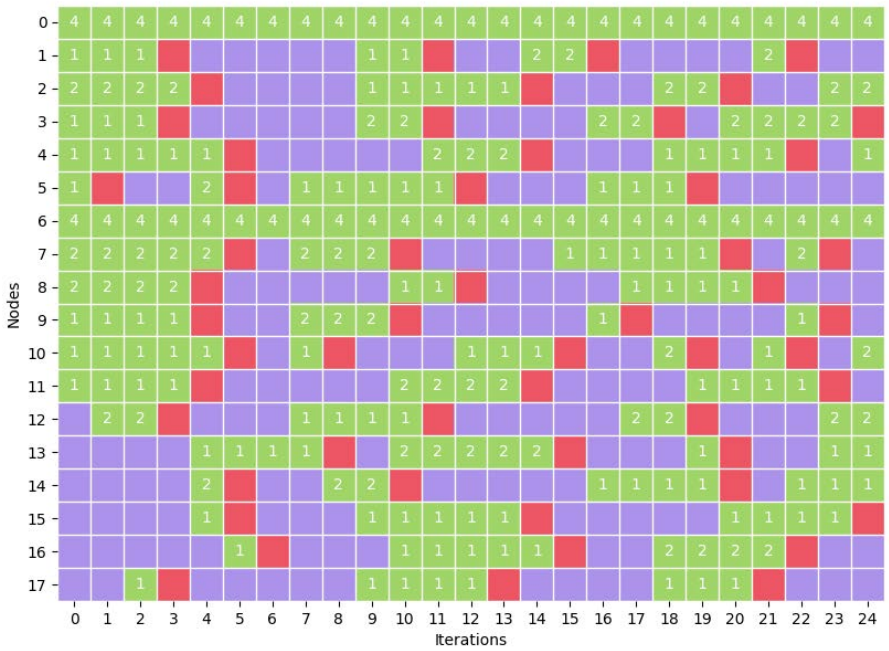


Figure 16: Node activity with 20% crash chance. Green indicates that a node is active at that iteration, purple - inactive, and red - crashed during that iteration