Workflow Optimization for Parallel Split Learning

Joana Tirana*, Dimitra Tsigkari†, George Iosifidis†, Dimitris Chatzopoulos*
*School of Computer Science, University College Dublin & VistaMilk SFI, Ireland
†Delft University of Technology, The Netherlands
joana.tirana@ucdconnect.ie, D.Tsigkari@tudelft.nl, G.Iosifidis@tudelft.nl, dimitris.chatzopoulos@ucd.ie

***Abstract*—Split learning (SL) has been recently proposed as a way to enable resource-constrained devices to train multi-parameter neural networks (NNs) and participate in federated learning (FL). In a nutshell, SL splits the NN model into parts, and allows clients (devices) to offload the largest part as a processing task to a computationally powerful helper. In parallel SL, multiple helpers can process model parts of one or more clients, thus, considerably reducing the maximum training time over all clients (makespan). In this paper, we focus on orchestrating the workflow of this operation, which is critical in highly heterogeneous systems, as our experiments show. In particular, we formulate the joint problem of client-helper assignments and scheduling decisions with the goal of minimizing the training makespan, and we prove that it is NP-hard. We propose a solution method based on the decomposition of the problem by leveraging its inherent symmetry, and a second one that is fully scalable. A wealth of numerical evaluations using our testbed's measurements allow us to build a solution strategy comprising these methods. Moreover, we show that this strategy finds a near-optimal solution, and achieves a shorter makespan than the baseline scheme by up to 52.3%.**

I. Introduction

Motivation. The proliferation of devices that collect voluminous data through their sensors motivated the design of client-based distributed machine learning (ML) protocols, like federated learning (FL) [1]. In FL, the training process is organized in training rounds that include local model updates at the devices (that act as *clients*) and the aggregation of all the clients' models at a server (the *aggregator*). During this process, clients keep their dataset locally, while only sharing their model's updates with the aggregator.

Some of the main challenges in FL are: 1) system heterogeneity; 2) communication overhead; 3) constrained resources, i.e., clients of limited memory and computing capacities [2]. As a result of these factors, the training time of some clients might be prohibitively long, thus, slowing down this cross-silo distributed ML process. Indeed, these clients (stragglers) increase the *training makespan*, i.e., the maximum training time over all clients, which is a key metric in highly heterogeneous systems because of the synchronous nature of FL [2], [3]. While state-of-the-art FL approaches propose ways of alleviating this phenomenon, e.g., via model pruning [4] or asynchronous protocols [5], they may compromise the accuracy of the produced model. Moreover, clients with limited memory capacity (e.g., IoT devices) might not be able to participate in FL processes that train large ML models.

Acknowledgments: This work has been supported by SFI-VistaMilk grant no 16/RC/3835 and EU Horizon project no 101092912 (MLSysOps).

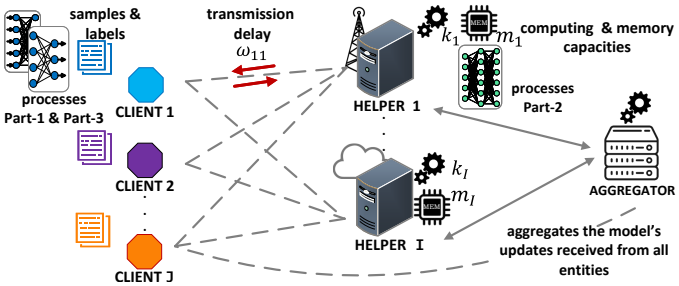

Fig. 1: Parallel SL in this work. The considered network topology, its resources, and the processing tasks per entity.

Split learning (SL) protocols have been recently proposed in order to enable resource-constrained clients to train neural networks (NNs) of millions of parameters [6]. In SL, clients offload a part of their training task to a *helper*, which could be a Virtual Machine (VM) on the cloud or a lightweight container in a base station in beyond 5G networks. Formally, a NN comprising L layers[1] $(1, \ldots, L)$ is split into three parts (part-1, part-2, and part-3) of consecutive layers $([1, \ldots, \sigma_1], [\sigma_1+1, \ldots, \sigma_2], [\sigma_2+1, \ldots, L])$ using 2 *cut layers* σ_1 and σ_2. Then, part-1 and part-3 are processed at the clients, and part-2 at the helper. This allows the resource-constrained clients to offload computationally demanding processes to the helper.

In conventional SL, the clients share the same part-2, and the helper collaborates with each client in a sequential order to train the model parts. This can lead to long delays in the training process depending on the number of participating clients. Whereas, in parallel SL [7], [8], the helper allocates a different version of part-2 for each client, allowing clients to make parallel model updates. At the end of each training round, all clients synchronize their model versions, and thus, the training makespan remains a key metric. Essentially, parallel SL is the integration of SL in the FL protocol. In fact, parallel SL reduces the makespan (when compared to the conventional SL) without compromising the model accuracy [7], [9], [10].

The presence of multiple helpers working in parallel can further reduce the training makespan [11]. Orchestrating the workflow of parallel SL in this network of entities (as depicted in Fig. 1) is one of the main challenges that the SL paradigm faces, as discussed in [2]. In detail, the factors that one needs to consider are the computation and memory resources of all entities, and the connectivity of the clients to the helpers.

[1]Throughout this manuscript, a "layer" is the NN's model part that is indivisible, i.e., it cannot be further partitioned into more layers.

Methodology & Contributions. Driven by the time measurements of our testbed and their heterogeneity even among devices of similar capabilities, we identify two key decisions: 1) the *client-helper assignments* that are tied to the helpers' memory and computing capacities; 2) the *scheduling*, i.e., the order in which each helper processes the offloaded tasks. Both decisions can be crucial for the training makespan by alleviating the effect of stragglers while fully utilizing the available resources. Hence, we formulate the problem of *jointly* making these decisions with the goal of minimizing the training makespan. To the best of our knowledge, this is the first work that studies this joint problem. We analyze this problem and its challenges both theoretically (proving it is NP-hard) and experimentally (using measurements from a realistic testbed). Therefore, we propose a solution method based on an intuitive decomposition of the problem into two subproblems, leveraging its inherent symmetry. The first one involves the assignment and the forward-propagation scheduling variables, and the second one involves the backward-propagation scheduling variables. For the former, the Alternating Direction Method of Multipliers (ADMM) is employed, while for the latter, a polynomial-time algorithm is provided. Moreover, we propose a second solution method based on load balancing, that is more scalable, and thus, ideal for large problem instances. Finally, our numerical evaluations provide insights on the performance of the proposed methods, as well as the achieved gains in makespan in representative scenarios. The contributions of this work are summarized below.

- We formulate the joint problem of client-helper assignments and scheduling decisions with the goal of minimizing the training makespan in parallel SL, and prove it is NP-hard.
- We propose a solution method that is based on the decomposition of the problem into two subproblems. For the first one, ADMM is employed, while for the second one a polynomial-time algorithm is provided.
- We extend our model to account for preemption costs, and propose a balanced-greedy algorithm with minimal overheads.
- We perform numerical evaluations of the proposed methods using collected data from our testbed.[2] These findings shape a solution strategy based on the scenario at hand.
- We show that our solution strategy finds a near-optimal solution and achieves a shorter makespan than the baseline scheme by up to 52.3%, and on average by 23.4%. Finally, we assess the impact of the number of helpers on the makespan.

II. Related Work

Client-based Distributed ML. Research on FL mainly focuses on achieving good accuracy while minimizing the wall-clock time, through client selection strategies [12], or aggregation algorithms [13], [14], or by taking into account the communication overhead [12], [15]. However, some clients might not be able to support the computation demands of such protocols, which is a less explored problem. Literature on SL tackles exactly this issue [6], [7], [16]–[18]. A large body of existing works model a system consisting of multiple clients and a single helper. In particular, it focuses on finding the NN's cut layers while trying to optimize the energy consumption [19], [20], or the training makespan [20], [21], or privacy [9]. In presence of multiple clients, the system may need to be scaled up in order to speed up the training process. In such systems, with multiple helpers, minimizing the training makespan requires a careful workflow orchestration. Close to this idea, the work in [11] jointly finds the cut layers and assignment decisions without taking into account the scheduling decisions. As our analysis shows, scheduling decisions are crucial in systems of highly heterogeneous network resources. Hence, it is clear that one needs to jointly optimize the client-helper assignments and scheduling decisions in parallel SL.

Workflow Optimization. Joint problems of assignment and scheduling decisions, such as the parallel machine scheduling problem, are often NP-hard, see, e.g., [22]–[24]. While a first approach would be to rely on methods such as branch-and-bound or column or row generation methods (like benders decomposition [25]), our experiments show that such methods may lead to high computation overheads, even for small problem instances. Different from this approach or other existing approaches (e.g., for edge computing policies [26], [27]), we decompose the problem based on the inherent structure of parallel SL operations. Next, we solve one of the resulting subproblems with ADMM from the toolbox of convex optimization [28]. This iterative method has been recently found to perform remarkably well for non-convex problems [29], [30]. The advantage of employing this method lies in its versatility, allowing us to use techniques that may constrain the problem's solution space or tune its penalty parameters [29], and thus, we tailor it to leverage the nature of the subproblem at hand.

III. System Model

Network Topology. We consider a system with a set $\mathcal{J}$ of $J=|\mathcal{J}|$ clients, e.g., IoT or handheld devices, and a set $\mathcal{I}$ of $I=|\mathcal{I}|$ helpers, that are connected over a wireless bipartite network $G=(\mathcal{J},\mathcal{I},\mathcal{E})$ with non-interfering links $\mathcal{E}$, see Fig. 1. The nodes are potentially heterogeneous in terms of hardware and/or wireless connectivity. Namely, each node $n \in \mathcal{N}:=\mathcal{J}\cup\mathcal{I}$ has computing capacity k_n (cycles/sec) and memory capacity m_n Gbytes. Further, we denote ω_{ji} the average delay[3] for transmitting one byte from client j to helper i, $\forall(j,i)\in\mathcal{E}$, and, w.l.o.g., we assume symmetric links, i.e., $\omega_{ij}=\omega_{ji}$. All nodes are connected to an aggregator, indexed $n=0$, who may collect the necessary information and orchestrate the workflow using the solution strategy we develop.

Parallel SL. The clients collaborate with the helpers to train a large NN using SL and FL. Each client owns a dataset that is divided into batches of equal size. As discussed in Sec. I, the NN is split into three parts, where σ_1 and σ_2 are the cut layers and each client computes part-1 and part-3, and offloads part-2 to a helper[4]. This SL architecture [6] protects the privacy of

[2]The evaluation code and the collected testbed's measurements are publicly available at https://github.com/jtirana98/SFL-workflow-optimization.

[3]In OFDMA-based systems, e.g., mobile networks, these assumptions are satisfied by design. In shared-spectrum systems, these parameters capture the effective (accounting for collisions) average delay.

[4]An interesting approach would be to split part-2 into more parts that are offloaded and processed by more than one helper. However, this would require a non-trivial coordination and communication among the helpers that we plan to address in future work.

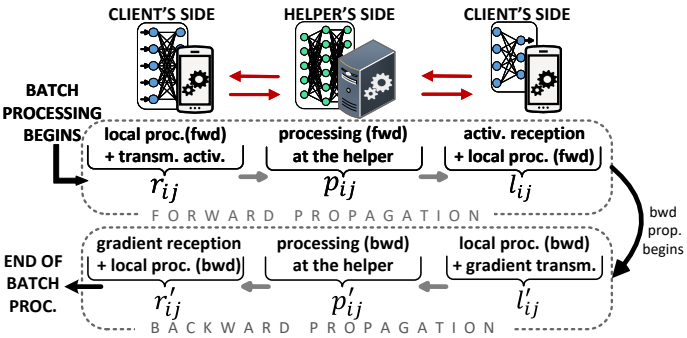

Fig. 2: The workflow of the batch processing for a single client and helper pair, and the corresponding times (processing and transmission). The *queuing delay* that a client might experience at the helper is not depicted here.

the clients' data since the samples and labels are kept locally. Finally, our analysis is oblivious to: a) the cut layers, which are decided in advance and may differ across the clients, and b) the training hyperparameters (e.g., batch size, learning rate, etc.), and thus, the resulting model accuracy is not affected.

Batch Processing Workflow. Fig. 2 depicts the steps of one *batch update* for client $j \in \mathcal{J}$ and helper $i \in \mathcal{I}$, and introduces the main time-related parameters of parallel SL. We employ a time-slotted model [31] with time intervals that, w.l.o.g., are of unit-length.[5] The client applies forward-propagation of part-1 and transmits the activations of the first cut layer (σ_1) to the helper. We denote by r_{ij} the number of time slots required for these two operations, which depends on k_j and ω_{ij}. The helper needs p_{ij} time slots to propagate these activations into part-2 (i.e., to execute the `fwd-prop` task), which depend both on the capacity k_i and the choice of cut layers, i.e., the "size" of the task. The client receives the activations of the last layer of part-2 (σ_2) from the helper and completes forward-propagation by processing part-3 and computing the loss. We denote by l_{ij} the time required for these operations, which depends on ω_{ij}, the data size, and k_j.

Then, the back-propagation of the training error starts. The client updates the weights of part-3, computes the gradients, and transmits them to the helper, consuming l'_{ij} time slots. The helper back-propagates these gradients into part-2, so as to update its weights, spending p'_{ij} time to execute this `bwd-prop` task. Afterwards, it transmits the gradients of σ_1 to the client, who then back-propagates part-1. We denote by r'_{ij} the time required for this final step. The time-related parameters (or delays) $\boldsymbol{r} = (r_{ij}, (i,j) \in \mathcal{E})$, $\boldsymbol{r'} = (r'_{ij}, (i,j) \in \mathcal{E})$, $\boldsymbol{p} = (p_{ij}, (i,j) \in \mathcal{E})$, $\boldsymbol{p'} = (p'_{ij}, (i,j) \in \mathcal{E})$, $\boldsymbol{l} = (l_{ij}, (i,j) \in \mathcal{E})$, and $\boldsymbol{l'} = (l'_{ij}, (i,j) \in \mathcal{E})$ represent average quantities[6] for these tasks or processes, and are considered available through profiling and other offline measurement methods [32], [33].

Epochs & Aggregation. The batch processing workflow is repeated for all batches. When the client has applied a batch update using all batches of data, a *local epoch* is completed. Clients repeat the processing of a local epoch for a predefined number of times until a *training round* (or global epoch) is completed. Subsequently, the updated model parts from each node (client or helper) must be sent to and aggregated at the aggregator, using methods such as `FedAvg` [1]. Typically, such training processes require hundreds of training rounds, each consisting of multiple batch updates [7], [20]. Hence, in order to minimize the maximum training time across all clients, i.e., the *training makespan*, we leverage the structural nature of the training process, and focus on the makespan of a single batch, see [11], [20]. We note that, when compared to conventional FL, the time required for the aggregation (comprising processing and transmissions) is negligible since the size of the data transferred is smaller per entity. Moreover, transmitting model updates can start even before all entities have completed a batch update, thus speeding up the procedure.

It is important to stress the inherent coupling between forward and backward propagation. When a client $j \in \mathcal{J}$ transmits part-1 activations to a helper $i \in \mathcal{I}$, the latter allocates d_j Gbytes of memory, where possibly $d_j \neq d_{j'}$ if $j \neq j'$, in order to store and process these activations. The helper stores this data during the `fwd-prop`, and reassigns the (same) memory to the gradients received from the client during the `bwd-prop`. This means that, in practice, a client cannot use a different helper for each propagation direction.

Time Horizon & Decision Variables. As mentioned earlier, we employ a time-slotted model with time intervals S_t, where $S_0 = [0,1]$, $S_t = (t, t+1], t = 1, \ldots, T$, and $T = |\mathcal{T}|$ is the number of slots in time horizon $\mathcal{T}$. The parameter T upper-bounds the batch makespan, and can be calculated as follows:

$$T := \max_{(i,j)\in\mathcal{E}} \left\{ r_{ij} + l_{ij} + r'_{ij} + l'_{ij} \right\} + \sum_{j\in\mathcal{J}} \max_{i\in\mathcal{I}} \left\{ p_{ij} + p'_{ij} \right\},$$

where the first term finds the worst-case transmission, and processing times in the network; and the second term measures the worst helper's processing time for any task.

Based on this time-slotted model, we introduce variables that inject tasks to helpers towards minimizing the makespan. In particular, we introduce the binary variables $\boldsymbol{y} = (y_{ij} \in \{0,1\}, (i,j) \in \mathcal{E})$, where $y_{ij} = 1$ if client j is assigned to helper i. Moreover, we define the slot-indexed variables $\boldsymbol{x} = (x_{ijt} \in \{0,1\}, (i,j) \in \mathcal{E}, t \in \mathcal{T})$, where $x_{ijt} = 1$ if the `fwd-prop` task of client j is processed at helper i during slot S_t, and $x_{ijt} = 0$ otherwise. Similarly, we define $\boldsymbol{z} = (z_{ijt} \in \{0,1\}, (i,j) \in \mathcal{E}, t \in \mathcal{T})$ with $z_{ijt} = 1$ if the `bwd-prop` task of client j is processed at i during S_t. These vectors fully characterize the batch processing workflow.

IV. Problem Formulation

The scheduling and assignment decision variables ($\boldsymbol{x}$, $\boldsymbol{z}$, and $\boldsymbol{y}$) should be consistent with the SL operation principles.

Scheduling Constraints. Each `fwd-prop` task can be executed only after its input becomes available, i.e., after activations of σ_1 are transmitted to the helper. Hence,

$$x_{ijt} = 0, \quad \forall t < r_{ij}, \ (i,j) \in \mathcal{E}. \tag{1}$$

In scheduling parlance, r_{ij} is the *release time* of this task, i.e., when it becomes "available" at the helper. Similarly, the

[5] We further discuss the choice of the time slot's length in Sec. IV and VII.

[6] As is common in scheduling literature and, w.l.o.g., we assume that these quantities are integers. If this assumption is violated, fractions can be handled by multiplying by a proper factor. Also, one could adopt a more conservative approach where worst-case values are considered instead of the average ones.

`bwd-prop` can start only after the gradients of $\sigma_2 + 1$ have been received by the helper (see Fig. 2), and thus,

$$z_{ij(t+l_{ij}+l'_{ij})} \leq \frac{1}{p_{ij}} \sum_{\tau=0}^{t-1} x_{ij\tau}, \quad \forall (i,j) \in \mathcal{E}, t \in \mathcal{T}. \quad (2)$$

That is, in order to assign the `bwd-prop` task of j to i at (or after) slot $t + l_{ij} + l'_{ij}$, we need to allocate enough processing time at i (at least p_{ij}) for `fwd-prop` until slot t. Essentially, (2) are *precedence constraints* that ensure the `bwd-prop` of a client's part-2 strictly succeeds its `fwd-prop`. The next constraints ensure that each helper will process a single task during any time slot (assuming single-threaded computing):

$$\sum_{j \in \mathcal{J}} (x_{ijt} + z_{ijt}) \leq 1, \quad \forall i \in \mathcal{I}, t \in \mathcal{T}. \quad (3)$$

Assignment Constraints. Regarding the assignment decisions $\boldsymbol{y}$, each client's task is assigned to a single helper:

$$\sum_{i \in \mathcal{I}} y_{ij} = 1, \quad \forall j \in \mathcal{J}. \quad (4)$$

Further, the assignments are bounded by the helper's memory:

$$\sum_{j \in \mathcal{J}} y_{ij} d_j \leq m_i, \quad \forall i \in \mathcal{I}, \quad (5)$$

and recall that this memory is used in both directions. Clearly, the assignment and scheduling constraints are tightly coupled. Indeed, when an assignment is decided, we need to ensure adequate processing time will be scheduled for the `fwd-prop` and `bwd-prop` tasks. In other words, it should hold that:

$$\sum_{t \in \mathcal{T}} x_{ijt} = y_{ij} p_{ij}, \quad \forall (i,j) \in \mathcal{E} \quad \text{and} \quad (6)$$

$$\sum_{t \in \mathcal{T}} z_{ijt} = y_{ij} p'_{ij}, \quad \forall (i,j) \in \mathcal{E}. \quad (7)$$

Completion Times. Finally, we introduce additional variables to measure some key delays. In particular, we define $\boldsymbol{\phi} = (\phi_j, j \in \mathcal{J})$, where ϕ_j is the slot when the `bwd-prop` of client $j \in \mathcal{J}$ is completed. These variables should satisfy:

$$\phi_j \geq (t+1) z_{ijt}, \quad \forall (i,j) \in \mathcal{E}, t \in \mathcal{T}. \quad (8)$$

Similarly, we introduce the overall (batch) completion time variable $c_j, \forall j \in \mathcal{J}$, which should satisfy:

$$c_j = \phi_j + \sum_{i \in \mathcal{I}} r'_{ij} y_{ij} \quad \forall j \in \mathcal{J}. \quad (9)$$

Essentially, the vector $\boldsymbol{c} = (c_j, j \in \mathcal{J})$ contains the completion times of one-batch model-training for all clients, and hence, its maximum element dictates the makespan. Naturally, all elements of $\boldsymbol{\phi}$ and $\boldsymbol{c}$ are upper-bounded by T. Finally, we observe that the quantity $\phi_j - \sum_i y_{ij}(r_{ij} + p_{ij} + l_{ij} + l'_{ij} + p'_{ij})$ is the total queuing delay that client j might experience during `fwd-prop` and `bwd-prop`.

Preemption. A strong aspect of our model is that it allows preemption, i.e., a task may be paused partway through its execution and then resumed later, if this improves the makespan. Specifically, preemptions may occur at the end of each time slot S_t. Preemptive schedules may prioritize the slowest client (straggler), thus reducing the makespan. This is in contrast to previous work that follows more rigid non-preemptive models [11], but in line with related work on edge computing, e.g., [33], [34]. We further discuss this point in Sec. VI. Finally, since the length of S_t determines the frequency of preemptions, a smaller length implies a larger benefit from preemption, i.e., shorter makespan. We investigate this point using our testbed's measurements in Sec. VII.

We can now formulate the *joint scheduling and assignment* problem that minimizes the batch makespan of parallel SL.

Problem 1 (Batch Training Makespan)**.**

$$\mathbb{P}: \underset{\boldsymbol{x},\boldsymbol{z},\boldsymbol{y},\boldsymbol{\phi},\boldsymbol{c}}{\text{minimize}} \ \max_{j \in \mathcal{J}} \{c_j\}$$
$$\text{s.t.} \quad (1) - (9),$$
$$\boldsymbol{x}, \boldsymbol{z} \in \{0,1\}^{|\mathcal{E}| \times |\mathcal{T}|}, \boldsymbol{\phi}, \boldsymbol{c} \in \{0 \,..\, T\}^J, \quad (10)$$
$$\boldsymbol{y} \in \{0,1\}^{|\mathcal{E}|}. \quad (11)$$

This min-max problem can be written as an Integer Linear Program (ILP) using standard transformations [35, Sec. 4.3.1], i.e., by introducing the worst-case makespan variable ξ and changing the objective to $\min_{\xi,\boldsymbol{x},\boldsymbol{y},\boldsymbol{z},\boldsymbol{\phi}} \xi$ with additional constraints $\xi \geq c_j, \forall j \in \mathcal{J}$. Albeit elegant, such transformations do not alleviate the computational challenges in solving large, or even medium-sized instances of $\mathbb{P}$. To exemplify, for a scenario with $J = 20$ clients, $I = 5$ helpers, and horizon $T = 636$, state-of-the-art solvers, such as Gurobi [36], achieve only a 40% optimality gap in 14 hours (off-the-shelf computer). Such long delays are not surprising due to the following result.

Theorem 1. $\mathbb{P}$ *(Problem 1) is NP-hard.*

Proof. We first define a simpler instance of $\mathbb{P}$: we assume that all helpers have enough memory for all tasks, i.e., we drop the memory constraints in (5), and $\boldsymbol{r} = \boldsymbol{r}' = \boldsymbol{l} = \boldsymbol{l}' = \boldsymbol{0}$, i.e., transmissions and propagations of part-1 and 3 are instantaneous. Therefore, all `fwd-prop` tasks are released at time 0 and each `bwd-prop` task may start instantly after the `fwd-prop` task is processed. Moreover, we assume that $\boldsymbol{p} = \boldsymbol{p}' = \boldsymbol{1}$, i.e., all tasks are of unit-length (require one time slot). We will show that there is a polynomial-time reduction from the parallel machine scheduling problem in [37] to this problem. The former is defined as follows: given a set of n jobs and a set of m parallel machines, each job should be assigned to a machine, while every machine can process at most one job at a time. The processing time of all jobs is $q = 1$, jobs are subject to precedence constraints, and the problem has as objective to find a schedule and the job-machine assignments to minimize the makespan, i.e., the time the last job will be completed at the machine. The reduction is shown by setting $n = J$, i.e., the `fwd-prop` and `bwd-prop` are the jobs (with precedence constraints), $m = I$, i.e., the helpers are the machines, and $p_{ij} = p'_{ij} = 1, \ \forall (i,j) \in \mathcal{E}$. Given this reduction and the fact that the parallel machine scheduling problem is NP-hard [22], [37], $\mathbb{P}$ is NP-hard as well. □

Given this result, we develop a multi-fold solution strategy consisting of a decomposition algorithm (Sec. V) and an informed heuristic (Sec. VI).

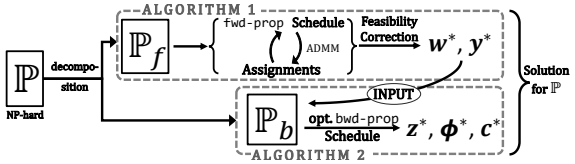

Fig. 3: The roadmap to our ADMM-based solution method.

V. Solution method

The core idea of our solution method is to decompose $\mathbb{P}$ into two subproblems (see Fig. 3): *(i)* $\mathbb{P}_f$, which minimizes the forward propagation makespan by deciding variables $\boldsymbol{x}$ and $\boldsymbol{y}$; *(ii)* $\mathbb{P}_b$, which minimizes the backward makespan by deciding $\boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{c}$. We solve $\mathbb{P}_f$ using ADMM (see discussion in Sec. II), and, for $\mathbb{P}_b$, we prove it admits a polynomial-time algorithm by leveraging its coupling with $\mathbb{P}_f$ (due to $\boldsymbol{y}$). As we will see in Sec. VII, this approach will lead to considerable speedups (up to $52\times$) when compared to exact solution methods.

A. `Fwd-prop` Optimization

Before introducing $\mathbb{P}_f$, we need some additional notation. First, we note that the time horizon that is related to `fwd-prop` can be confined to the set $\mathcal{T}_f$ with $T_f := \max_{(i,j)\in\mathcal{E}}\{r_{ij}+l_{ij}\} + \sum_{j\in\mathcal{J}} \max_{i\in\mathcal{I}} p_{ij}$. We denote by ϕ_j^f the `fwd-prop` finish time for each client $j \in \mathcal{J}$, which by definition has to satisfy the constraints (similarly to (8)):

$$\phi_j^f \geq (t+1)x_{ijt},\ \forall i \in \mathcal{I}, t \in \mathcal{T}_f. \tag{12}$$

Also, we define the `fwd-prop` completion time $c_j^f, j \in \mathcal{J}$, which is determined by ϕ_j^f and the times l_{ij}, i.e.,

$$c_j^f = \phi_j^f + \sum_{i\in\mathcal{I}} l_{ij} y_{ij},\ \forall j \in \mathcal{J}. \tag{13}$$

As before, $\boldsymbol{\phi}^f = (\phi_j^f, j \in \mathcal{J})$ and $\boldsymbol{c}^f = (c_j^f, j \in \mathcal{J})$. Collecting the above requirements, we can now formulate $\mathbb{P}_f$:

Problem 2 (`Fwd-prop` makespan)**.**

$$\begin{aligned}
\mathbb{P}_f:\ \underset{\boldsymbol{x},\boldsymbol{y},\boldsymbol{\phi}^f,\boldsymbol{c}^f}{\text{minimize}}\ & \max_{j\in\mathcal{J}} \{c_j^f\} \\
\text{s.t.}\quad & (1), (4)-(6), (11), (12), (13) \\
& \sum_{j\in\mathcal{J}} x_{ijt} \leq 1, \quad \forall i \in \mathcal{I}, t \in \mathcal{T}_f, && (14) \\
& \boldsymbol{x} \in \{0,1\}^{|\mathcal{E}|\times|\mathcal{T}|}, \boldsymbol{\phi}^f, \boldsymbol{c}^f \in \{0\,..\,T_f\}^J. && (15)
\end{aligned}$$

Comparing the constraints of this reduced problem with $\mathbb{P}$, we observe that: $\mathbb{P}_f$ replaces constraints (8) and (9) with (12) and (13); omits constraints (2) and (7); and replaces (3) with (14). This yields a simpler problem, as $\mathbb{P}_f$ has fewer variables (omits $\boldsymbol{z}$ and $T_f < T$) and less complicated constraints. However, the solution of $\mathbb{P}_f$ will not necessarily be consistent with the `bwd-prop` operations. For this, we properly tune the `bwd-prop` scheduling problem $\mathbb{P}_b$ in Sec. V-B. Despite this decomposition, there is not a known algorithm for $\mathbb{P}_f$. In fact, arguments as the ones in the proof of Theorem 1 can lead to a reduction from the unrelated machine scheduling problem with release dates, preemption, and no precedence constraints to $\mathbb{P}_f$. To the best of our knowledge, there is no polynomial-time algorithm for this problem, except for some special cases, e.g., for a specific number of machines, see [22], [38], [39].

Algorithm 1 ADMM-based `fwd-prop` Workflow

Input: $\boldsymbol{\lambda}^{(0)}, \boldsymbol{y}^{(0)} = \mathbf{0}$, ε_1, ε_2, ρ, τ_{max}, T^f

1 **for** $\tau = 1, 2, \ldots, \tau_{max}$ **do**

2 $\quad \boldsymbol{w}^{(\tau+1)} = \underset{(1),(12)-(15),(20)}{\operatorname{argmin}} \mathcal{L}\big(\boldsymbol{w}, \boldsymbol{y}^{(\tau)}, \boldsymbol{\lambda}^{(\tau)}\big)$ *schedule*

3 $\quad \boldsymbol{y}^{(\tau+1)} = \underset{(4),(5),(11)}{\operatorname{argmin}} \mathcal{L}\big(\boldsymbol{w}^{(\tau+1)}, \boldsymbol{y}, \boldsymbol{\lambda}^{(\tau)}\big)$ *assignment*

4 $\quad \lambda_{ij}^{(\tau+1)} = \lambda_{ij}^{(\tau)} + \Big(\sum_{t\in\mathcal{T}_f} x_{ijt}^{(\tau+1)} - y_{ij}^{(\tau+1)} p_{ij}\Big),\ \forall (i,j) \in \mathcal{E}$

5 $\quad$ Exit **for** if (17) and (18) are satisfied.

6 Correct $\mathbb{P}_f$ feasibility with (19).

7 **Return** $\boldsymbol{w}^* = (\boldsymbol{x}^*, \boldsymbol{\phi}^{f*}, \boldsymbol{c}^{f*})$, $\boldsymbol{y}^*$

To that end, we employ ADMM to decompose $\mathbb{P}_f$ and obtain smaller subproblems, which, it turns out, can be solved in reasonable time for many problem instances. Indeed, we observe that by relaxing the constraints in (6), we can decompose $\mathbb{P}_f$ into a (forward-only) scheduling subproblem, involving $(\boldsymbol{x}, \boldsymbol{\phi}^f, \boldsymbol{c}^f)$, and an assignment subproblem that optimizes $\boldsymbol{y}$. Then, we can solve these subproblems iteratively and penalize their solution so as to gradually recover the relaxed constraints. The first step is to define the Augmented Lagrange function:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{w}, \boldsymbol{y}, \boldsymbol{\lambda}) =& \max_{j\in\mathcal{J}} c_j^f + \sum_{(i,j)\in\mathcal{E}} \lambda_{ij} \Bigg(\sum_{t\in\mathcal{T}_f} x_{ijt} - y_{ij} p_{ij} \Bigg) \\
& + \frac{\rho}{2} \sum_{(i,j)\in\mathcal{E}} \Bigg| \sum_{t\in\mathcal{T}_f} x_{ijt} - y_{ij} p_{ij} \Bigg|,
\end{aligned} \tag{16}$$

where we define $\boldsymbol{w} = (\boldsymbol{x}, \boldsymbol{\phi}^f, \boldsymbol{c}^f)$ to streamline the notation; introduce the dual variables $\boldsymbol{\lambda} = (\lambda_{ij}, (i,j) \in \mathcal{E})$ for relaxing (6); and use the ADMM penalty parameter ρ, see [28, Ch. 3]. Note that, unlike the vanilla version of ADMM that uses the Euclidean norm ℓ_2, we penalize the constraint violation using the ℓ_1 norm so as to improve the algorithm runtime.

The detailed steps can be found in Algorithm 1. At iteration $\tau+1$, we first update the schedule $\boldsymbol{w}$ using the previous assignment $\boldsymbol{y}^{(\tau)}$ and dual variables $\boldsymbol{\lambda}^{(\tau)}$ (line 2). Next, we optimize the assignment $\boldsymbol{y}^{(\tau+1)}$ using the updated schedule $\boldsymbol{w}^{(\tau+1)}$ (line 3), and finally we correct the dual variables based on the violation of (6) in line 4. We repeat these steps until convergence is achieved or a maximum number of iterations is reached (τ_{max}).[7] As a convergence flag, we use the detection of stationary assignments and objective values (line 5):

$$\sum_{(i,j)\in\mathcal{E}} \Big| y_{ij}^{(\tau+1)} - y_{ij}^{(\tau)} \Big| < \varepsilon_1 \quad \text{and} \tag{17}$$

$$\Big| \max_{j\in\mathcal{J}} c_j^{f,(\tau+1)} - \max_{j\in\mathcal{J}} c_j^{f,(\tau)} \Big| < \varepsilon_2. \tag{18}$$

[7] The $\boldsymbol{w}$- and $\boldsymbol{y}$-subproblems (i.e., lines 2-3 of Alg. 1) could be solved either with exact methods, e.g., branch and bound, or inexact methods, e.g., through a tailored relaxation [30]. As for the former, we elaborate on the resulting overhead in Sec. VII, and the latter is in line with the fact that ADMM can (under certain conditions) tolerate inexact solutions for its subproblems [40].

Finally, we correct any remaining infeasible constraints by tuning the schedule to the final assignment $\boldsymbol{y}^*$ (line 6):

$$\boldsymbol{w}^* = \underset{(1),(12)-(15),(6)}{\operatorname{argmin}} \mathcal{L}\big(\boldsymbol{w}, \boldsymbol{y}^*, \boldsymbol{\lambda}^*\big), \tag{19}$$

where we additionally use the constraints (6) to ensure full consistency between $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$. Since the relaxed constraints in (6) concern the processing times for each client's task, we can further accelerate the convergence of Alg. 1 by creating a tighter constraint set [29]. In detail, we introduce a set of constraints that limit the search to schedules that allocate enough processing time for each client in the $\boldsymbol{w}$-subproblem:

$$\sum_{i\in\mathcal{I}} \frac{1}{p_{ij}} \sum_{t\in\mathcal{T}_f} x_{ijt} = 1, \quad \forall j \in \mathcal{J}. \tag{20}$$

Algorithm 1 is not guaranteed to converge to the optimal solution of $\mathbb{P}_f$. However, its efficacy is demonstrated with a battery of trace-driven evaluations in Sec. VII, where in most of the tested scenarios, it achieves less than 10.2% suboptimality gap, with one corner case of 14.9%.

B. `Bwd-prop` Schedule

Given the assignment $\boldsymbol{y}^*$ and `fwd-prop` schedule $\boldsymbol{w}^* = (\boldsymbol{x}^*, \boldsymbol{\phi}^{f*}, \boldsymbol{c}^{f*})$ from the solution of $\mathbb{P}_f$, we can optimize the `bwd-prop` schedule by solving the $\mathbb{P}_b$ subproblem. The latter stems from $\mathbb{P}$ by removing the constraints which do not involve $\boldsymbol{z}$; and by further replacing variables $\boldsymbol{x}$ and $\boldsymbol{y}$ with the respective values $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$ that we obtained from $\mathbb{P}_f$, wherever they appear in the constraints:

Problem 3 (`Bwd-prop` makespan; given $\boldsymbol{y}^*$, $\boldsymbol{w}^*$)**.**

$$\begin{aligned} \mathbb{P}_b: \ \underset{\boldsymbol{z},\boldsymbol{\phi},\boldsymbol{c}}{\text{minimize}} \ & \max_{j\in\mathcal{J}} c_j \\ \text{s.t.} \quad & (2),(3),(7)-(9) \\ & \boldsymbol{z} \in \{0,1\}^{|\mathcal{E}|\times|\mathcal{T}|}, \boldsymbol{\phi}, \boldsymbol{c} \in \{T_f^* \,..\, T\}^J. \end{aligned} \tag{21}$$

We stress that the variables $\boldsymbol{\phi}$ and $\boldsymbol{c}$ can be restricted in the time window starting after the `fwd-prop`, which is provided by $\mathbb{P}_f$ and denoted by T_f^*. These provisions ensure that the solution we obtain from successively solving subproblems $\mathbb{P}_f$ and $\mathbb{P}_b$ will not induce constraint violations.

Theorem 2. *$\mathbb{P}_b$ can be solved in polynomial time.*

Proof. We first observe that, since the client-helper assignments are fixed ($\boldsymbol{y}^*$), we can parallelize $\mathbb{P}_b$'s solution across the helpers. That is, we can independently focus on the `bwd-prop` tasks of the subset of clients $\mathcal{J}_i$ assigned to each helper $i \in \mathcal{I}$, where $\mathcal{J}_i := \{j \in \mathcal{J} \ : \ y_{ij}^* = 1\}$. Also, the $\boldsymbol{w}^*$ obtained by Alg. 1 dictates a subset of time slots where `bwd-prop` tasks can be scheduled. We denote by $\mathcal{T}_i$ the remaining eligible slots for helper i. We can now state the subproblem of minimizing the `bwd-prop` makespan for each helper $i \in \mathcal{I}$, while we abuse notation and drop the index i.

$$\begin{aligned} \mathbb{P}_b^i: \ \underset{\boldsymbol{z},\boldsymbol{\phi}}{\text{minimize}} \ & \max_{j\in\mathcal{J}_i} \{\phi_j + \pi_j\} \\ \text{s.t.} \quad & \sum_{t\in\mathcal{T}_j} z_{jt} = p'_j, \quad \forall j \in \mathcal{J}_i \end{aligned} \tag{22}$$

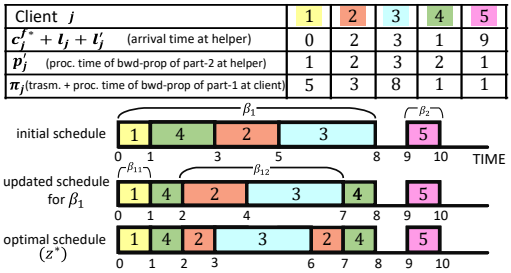

Fig. 4: Algorithm 2 for optimal `bwd-prop` schedule in a toy example of 5 clients and 1 helper.

$$z_{j(t+l_j+l'_j)} \leq \frac{1}{p_j} \sum_{\tau=0}^{t-1} x_{j\tau}^*, \ \forall j \in \mathcal{J}_i, t \in \mathcal{T}_i \tag{23}$$

$$\sum_{j\in\mathcal{J}_i} z_{jt} \leq 1, \quad \forall t \in \mathcal{T} \tag{24}$$

$$\phi_j \geq (t+1) z_{jt}, \quad \forall j \in \mathcal{J}_i, t \in \mathcal{T}_i, \tag{25}$$

where $\pi_j := \sum_{i\in\mathcal{I}} r'_{ij} y_{ij}^*, \forall j \in \mathcal{J}_i$ and $x_{j\tau}^*$ are fixed parameters. We will show that there is a polynomial-time reduction from $\mathbb{P}_b^i$ to the single machine scheduling problem of minimizing the maximum cost subject to release and precedence constraints, which is polynomially time solvable [41]. It suffices to set the release times of the jobs as the $\{c_j^{f*} + l_j + l'_j\}_{j\in\mathcal{J}_i}$ (i.e., the time the client needs to complete the back-propagation of part-3 and transmit the gradients, given the c_j^{f*}). Moreover, it suffices to set as the cost function in [41] the quantity $\phi_j + \pi_j$, i.e., the makespan of the batch update (including the clients' local computations). □

We now present the algorithm that optimally solves $\mathbb{P}_b$ based on [41] together with a worked example in a scenario of 5 clients and 1 helper, as depicted in Fig. 4.

Algorithm 2 and Worked Example. For each helper i, and *in parallel*, we find the set of assigned clients $\mathcal{J}_i$. We then order the clients according to nondecreasing $\{c_j^{f*} + l_j + l'_j\}_{j\in\mathcal{J}_i}$, which are the arrival (release) times for their `bwd-prop` tasks at the helper, and build an initial schedule where tasks are processed according to the ordering of their arrival times. This schedule naturally decomposes $\mathcal{J}_i$ into an initial set $\mathcal{B}_i$ of *blocks*. Specifically, each block $\beta \in \mathcal{B}_i$ is the smallest set of clients whose `bwd-prop` task has an arrival time at (or after) $s(\beta) := \min_{j\in\beta}\{c_j^{f*} + l_j + l'_j\}$ and can be processed before $e(\beta) := s(\beta) + \sum_{j\in\beta} p'_j$. In fact, a client's task $h \notin \beta$ is either processed no later than $s(\beta)$, i.e., $\phi_h \leq s(\beta)$, or not released at the helper before $e(\beta)$, i.e., $c_h^{f*} + l_h + l'_h \geq e(\beta)$. Essentially, each block in $\mathcal{B}_i$ represents a non-idle period for the helper. In our example, there are two blocks: $\beta_1 = \{1,4,2,3\}$ and $\beta_2 = \{5\}$ with $s(\beta_1) = 0, e(\beta_1) = 8, s(\beta_2) = 9$, and $e(\beta_2) = 10$. We can now focus on each block separately [41]. For each block $\beta \in \mathcal{B}_i$, we find client $\ell \in \beta$ such that

$$\ell := \underset{j\in\beta}{\operatorname{argmin}} \{e(\beta) + r'_j\}. \tag{26}$$

In our example, that would be client 4 for β_1 since $9 = \min\{8+5, 8+3, 8+8, 8+1\}$ and client 5 for β_2. Then, we reschedule the tasks in β such that the `bwd-prop` task

of client ℓ is processed only during time intervals (between $s(\beta)$ and $e(\beta)$) where no other client's task has arrived or is being processed. In our case, since β_2 contains a single client, no reschedulings are required within β_2, while, for β_1, client 2 "moves" to an earlier slot in the schedule (subject to its arrival time) and the task of client 4 is scheduled in slots where no other task is processed. This also decomposes the remaining set $\beta - \{\ell\}$ into a set Γ_β of subblocks (according to the rule described above). In our example, $\Gamma_1 = \{\beta_{11}, \beta_{12}\}$ as depicted in Fig. 4. Now, for each subblock in Γ_β, we find the client ℓ' based on (26) and reschedule the tasks within this subblock (based on the same rules as above). In our example, β_{11} needs no rescheduling, while, for β_{12}, ℓ' is 2 since $10 = \min\{7+3, 7+8\}$. The resulting schedule is optimal. In our case, client 3 will be processed upon arrival at the helper. The final optimal schedule has a makespan of 14, where client 3 will be the last one to finish the back-propagation of its part-1. This process runs in $\mathcal{O}(|\mathcal{J}_i|^2)$ time for helper i, so Algorithm 2 will run in $\mathcal{O}(\max_{i\in I}\{|\mathcal{J}_i|\}^2)$ time due to parallelization.

The ADMM-based solution method can be easily adapted in cases where clients own samples of different sizes. In such cases, we can simply remove from the obtained schedules $\boldsymbol{x}^*$ and $\boldsymbol{z}^*$ the clients whose samples are completely processed (after a number of batch updates) and "move" the remaining clients earlier in the schedules (subject to availability of their tasks at the helpers). Moreover, the assignments $\boldsymbol{y}^*$ do not need to change since helpers have already allocated memory for the model copies of the assigned clients.

VI. Model Extensions & A (Faster) Heuristic

Preemption Cost. In certain systems (e.g., with very limited memory), preemption might induce further delays or costs (e.g., due to context switch [42]) that need to be taken into account while deciding on the schedule. This feature can be readily incorporated in our model without affecting the proposed solution method. In detail, let us denote with μ_i the switching cost that captures the delay induced at helper $i \in \mathcal{I}$ when switching between two tasks, i.e., $x_{ijt} = 0$ and $x_{ij(t+1)} = 1$, for some client $j \in \mathcal{J}$ and time interval S_t. Then, we can directly apply the ADMM algorithm with the following modified constraint instead of (13) in $\mathbb{P}_f$:

$$c_j^f = \phi_j^f + \sum_{i\in\mathcal{I}} l_{ij} y_{ij} + \sum_{i\in\mathcal{I}}\sum_{t\in\mathcal{T}} \mu_i \left| x_{ijt} - x_{ij(t+1)} \right|, \ \forall j \in \mathcal{J},$$

where the last term captures the cost of switching tasks when a preemption occurs and when a task has just started being processed. In a similar way, we can modify the constraints in (9) in $\mathbb{P}_b$ for variables $\boldsymbol{z}$, and use exact or inexact methods to solve the problems $\mathbb{P}_b^i$, as discussed in footnote 7.

A Scalable Heuristic (balanced-greedy). Since the sub-problems of Alg. 1 are ILP problems, the ADMM-based method might induce high overhead. To exemplify, running the ADMM-based method on an ILP solver (with exact solution) for a scenario of $J = 70$ clients and $I = 10$ helpers takes 14 min. While such overhead may be tolerable in scenarios of this size, especially given the resulting time savings in total training makespan when compared to a baseline (see Sec. VII), it might not be the case in larger problem instances. Moreover, this method decides on the assignments ($\boldsymbol{y}$) without taking into account the times $\boldsymbol{p}'$ of `bwd-prop` tasks. Specifically, in our experiments, we noticed that when the processing times of `bwd-prop`, i.e., $\boldsymbol{p}'$, are much larger than the times of `fwd-prop` tasks, i.e., $\boldsymbol{p}$, long queues during `bwd-prop` may occur. This phenomenon can be alleviated by balancing the workload among the helpers. We, thus, propose a heuristic that addresses these issues: it is of low complexity and *balances* the client assignments among helpers in a greedy way.

TABLE I: Testbed devices and average computing time (in sec) for a batch update, where the batch size is 128 samples.

Device	ResNet101	VGG19
RPi 4 B Cortex-A72 (4 cores), 4GB	91.9	71.9
RPi 3 B+ Cortex-A5 (4 cores), 1GB	not enough memory	
NVIDIA Jetson Nano, 4 GB (CPU,GPU)	(143, 1.2)	(396, 2.6)
VM 8-core virtual CPU, 16GB	2	3.6
Apple M1 8-core CPU, 16GB	3.5	3.6

We propose *balanced-greedy* that consists of two steps; it first decides on the client-helper assignments, and then on the scheduling. Specifically, it starts with $\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{y} = \boldsymbol{0}$ and:

1) The assignments follow a static load balancing algorithm [43], where the load of helper i is defined as the number of assigned clients, i.e., $G_i = \sum_j y_{ij}$. For each client $j \in \mathcal{J}$, it finds the subset of helpers Q_j with enough available memory to allocate for j (i.e., $Q_j := \{i \in \mathcal{I} : m_i - \sum_h d_h y_{ih} \geq d_j\}$) and, based on Q_j, it finds the helper η with the least load, i.e., $\eta = \text{argmin}_{i\in Q_j}\{G_i\}$. Balanced-greedy assigns client j to η, i.e., $y_{\eta j} = 1$, before proceeding to the next client.

2) The scheduling decisions $\boldsymbol{x}$ and $\boldsymbol{z}$ are made at each helper in a first-come-first-served (FCFS) order [44], i.e., for the `fwd-prop` tasks, the schedule $\boldsymbol{x}$ and the completion times $\boldsymbol{c}^f$ are based on the release times $\boldsymbol{r}$, and, for the `bwd-prop` tasks, $\boldsymbol{z}$ is based on $\boldsymbol{c}^f + \boldsymbol{l} + \boldsymbol{l}'$. In contrast to the ADMM-based method, balanced-greedy is non-preemptive.

VII. Performance Evaluation

In this section, we evaluate the performance of our solution methods using measurements from our testbed's devices.

Dataset & Models. We use CIFAR-10 [45] and two NN models: *(i)* ResNet101 [46], and *(ii)* VGG19 [47] for our training tasks. They are both deep convolutional NNs with 0.42 and 2.4 million parameters, and organized in 37 and 25 layers respectively. Hence, they may push resource-constrained devices to their limits when trained locally.

Testbed. The testbed's devices are listed in Table I, where the last two were employed as helpers. We also list the collected time measurements for a batch update for ResNet101 and VGG19. One of the devices (RPi 3) cannot fully train any of the two models locally due to its memory limitations. Furthermore, Jetson GPU's training times are comparable to the helpers' times, however, in practice, the memory allocation for the GPU training can be very challenging [48].

Setup. In our simulations, the values of the input parameters of $\mathbb{P}$, i.e., $\boldsymbol{r}, \boldsymbol{r}', \boldsymbol{p}, \boldsymbol{p}', \boldsymbol{l}, \boldsymbol{l}'$, are set according to the profiling data of the testbed (for the computation times) and findings on Internet connectivity in France [49, p.56] (for the transmission

TABLE II: Suboptimality and speedup achieved by the ADMM-based method when compared to an ILP solver.

		J	I	T	**suboptimality (%)**	**speedup ($\times$)**
Scenario1	Resnet101	10	2	294	0	32.2
			5	294	2.6	13.2
		15	5	399	14.9	37
	VGG19	10	2	176	0	12.5
			5	176	0	18.5
		15	5	211	0	22
Scenario2	Resnet101	10	2	321	8.2	26.4
			5	324	10.2	22.8
		15	5	445	4.2	17.2
	VGG19	10	2	263	3.2	20.5
			5	265	0	31.2
		15	5	331	2	52

times). We explore two scenarios that represent *two levels of heterogeneity* in terms of devices, resources, and cut layers:

• **Scenario 1 (low heterogeneity):** Clients and helpers have the same parameters as in Table I, where the selection of the type of device for each client and helper is uniformly random. Moreover, the entities' memory capacities are limited by their RAM size and all the clients' NNs have the same cut layers: layers 3 and 33 for Resnet101, and layers 3 and 23 for VGG19.

• **Scenario 2 (high heterogeneity):** To capture higher heterogeneity, the input parameters are devised by interpolating the time measurements of the profiled devices. Also, the entities' memory capacities can vary from device to device, but upper-bounded by their RAM size. In this scenario, clients participate in the training with different cut layers (randomly selected).

Results and Observations. We proved that $\mathbb{P}$ is NP-hard, which led us to propose two scalable solution methods. Table II shows the suboptimality and speedup achieved by the ADMM-based method when compared to Gurobi [36], one of the fastest ILP solvers [50], that optimally solves $\mathbb{P}$. We observe that, in most cases, our method achieves less than 10.2% suboptimality, with one corner case of 14.9%. However, even in this case, there is a $37\times$ speedup when compared to the solver. We highlight that these results derive from running less than 5 iterations of Algorithm 1, while we may achieve smaller suboptimality with a larger number of iterations using techniques like the ones in [29]. Actually, ADMM may be tailored so that we can balance suboptimality and speed.

Observation 1. The ADMM-based method finds the optimal solution for $\mathbb{P}$ in several problem instances and achieves up to $52\times$ speedup when compared to an ILP solver.

We note that the numerical evaluations in Table II were performed in small instances (up to 15 clients and 5 helpers) that can be handled by ILP solvers. We observe that the lowest suboptimality gap is achieved for VGG19, which comes from the choice of cut layers (see above). In particular, Fig. 5 shows the processing times between forward and backward propagation per device for the two NNs. We see that these times can highly differ between forward and backward operation. Such asymmetries that can occur in SL further corroborate our approach towards jointly optimized assignments and scheduling.

Next, in Table II, we see that the time horizon (T) increases with the problem size, and our method achieves considerable speedups in execution time for very large T. In particular, T is directly related to the number of variables of the problem. This

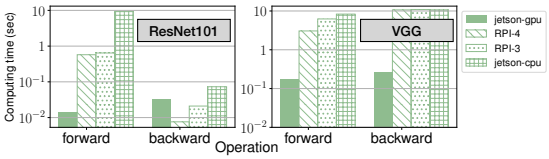

Fig. 5: Profiled computing time (ms.) of part-1 for each device.

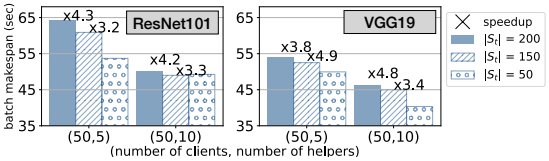

Fig. 6: Batch makespan obtained by the ADMM-based method for time slot length $|S_t|$ equal to 200 ms, 150 ms, and 50 ms, in Scenario 1. The speedup is relative to the case $|S_t| = 50$.

dependency can be critical in cases where the input parameters $\boldsymbol{r}, \boldsymbol{p}, \boldsymbol{l}$, etc. are in the order of thousands (e.g., when in ms). For this reason, we explore next how tuning the time slot's length can affect the obtained solution (batch makespan).

In Sec. IV, we discussed the impact of the time slot's length, i.e., $|S_t|$, on the frequency of preemptions. Furthermore, as $|S_t|$ decreases, the time horizon T and the number of the problem's variables increase. To exemplify, a processing time of 400ms would be translated into 2, 3, or 8 time slots when $|S_t| = 200$ ms, $|S_t| = 150$ ms, and $|S_t| = 50$ ms respectively. Since T is defined based on the input processing and transmission times, its length will be the largest when $|S_t| = 50$. Moreover, in the case where $|S_t| = 150$, the processing time of our example is interpreted as 3 slots, which can overestimate the makespan. In fact, since the helper will need a bit less than 3 slots to process the task, in a real-life implementation of the obtained schedule, it may be able to start processing the next task before the end of the 3rd slot. Therefore, in such cases, the time slot's length may affect the accuracy of the obtained schedule.

Observation 2. As the length of the considered time slots S_t increases, the obtained makespan increases, while the execution time decreases. This confirms an algorithmic tradeoff between the solution's precision and size of the solution space.

Fig. 6 depicts the makespan obtained by the ADMM-based method for 3 different $|S_t|$. The numbers on top of the bars are the speedups relative to the case $|S_t| = 50$, which has the highest overhead. We observe that the makespan is higher, in principle, as the $|S_t|$ increases. This is because large $|S_t|$ implies less frequent preemptions and a less precise solution. Of course, as $|S_t|$ increases, the length of the time horizon (T) decreases, which results in a speedup of up to 4.9%. Finally, we note that, for all the other experiments, we have used $|S_t|$ equal to 180ms, for Resnet101, and to 550ms, for VGG19.

Next, in Fig. 7, we compare the proposed methods (ADMM-based and balanced-greedy) between them, and with a *baseline scheme* that first decides on the client-helper assignments in a random way (subject, of course, to memory constraints), and then schedules the tasks in a FCFS order. This baseline could be seen as a naive real-time implementation of parallel SL without proactive decisions on assignments or scheduling.

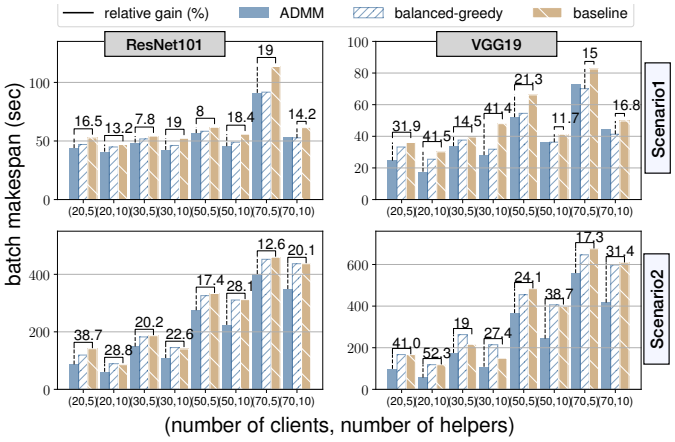

Fig. 7: Comparison of the proposed methods with the baseline.

We note that a straightforward comparison with related work (e.g., [11]) is not possible since, typically, its client-helper assignments are coupled with decisions on cut layers or other considerations, while it usually adopts a FCFS schedule.

We focus first on the comparison between the two proposed methods. We observe that the ADMM-based method finds a shorter makespan (up to 32%) than the balanced-greedy in medium-sized scenarios (1 & 2), i.e., up to 50 clients, and thus, it should be the preferred method. However, as the number of clients grows and the queuing delays risk to increase, balancing the helpers' loads provides a better solution. Indeed, in Scenario 1 (top), the balanced-greedy achieves a better makespan than the ADMM-based method, making it, hence, the method of choice.

Next, as the heterogeneity of the network resources increases (Scenario 2, bottom Fig. 7), the scheduling and assignment decisions become more crucial for the makespan. Therefore, it is not surprising that the ADMM-based method outperforms the balanced-greedy by up to 48%, and thus, it should be the preferred method. We note that the problem instances of Scenario 2 contain a few helpers with very limited memory capacities that were not fully utilized by balanced-greedy. The presence of such helpers explains the fact that the batch makespan is larger in Scenario 2 than in Scenario 1, since this implies long queuing delays in the other helpers. Nevertheless, these performance gains decrease as the number of clients increase, and given the discussion on the overhead of the two methods in Sec. VI, balanced-greedy should be preferred for very large scenarios (e.g., ≥ 100 clients in our case) to avoid excess overhead.

Observation 3. The numerical evaluations allow us to build a solution strategy based on the scenario's characteristics that achieves a shorter makespan than the baseline by up to 52.3%.

The observations above shape a *solution strategy* that comprises the two proposed methods, and it is tailored based on the scenario at hand (i.e., its heterogeneity and size). Also, we see that any improvement of one method over the other is larger in the VGG19. This reveals a dependency on the NN, since NNs may differ on the cut layers, the size of the processing

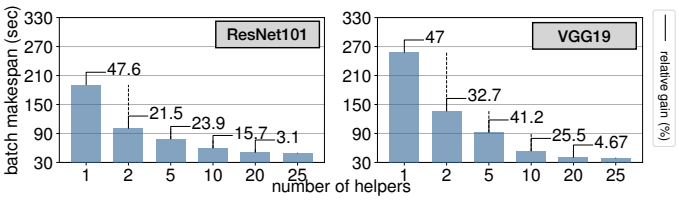

Fig. 8: Batch makespan obtained by the balanced-greedy method in Scenario 1 for $J = 100$ clients and varying I.

tasks [9], etc. We plan to further explore this dependency and its implications on the makespan in future work.

Focusing now on the comparison of our strategy (i.e., ADMM-based or balanced-greedy depending on the scenario, as discussed above) to the baseline scheme, we observe that our proposed strategy consistently outperforms the baseline, achieving a shorter makespan. In detail, the baseline scheme decides on the assignments $\boldsymbol{y}$ without taking into account processing and transmission delays, which results in a larger makespan. Essentially, this confirms the need for workflow optimization in parallel SL. We notice that, in some instances where the preferred method is the ADMM-based one (e.g., (30,5) for VGG19), the baseline with random client-helper allocation may find a shorter makespan than balanced-greedy. This is because balanced-greedy may allocate clients to slower helpers, without taking into account that queuing delays might not be long in faster helpers for such medium-sized instances.

Finally, in Fig. 8, we perform a sensitivity analysis with respect to the number of helpers in Scenario 1 where we depict the relative gains in batch makespan. Given the scenario's type and size, we employ balanced-greedy.

Observation 4. In a scenario of 100 clients and 1 helper, adding one more helper can dramatically decrease the batch makespan by up to 47.6%.

Whereas, in the presence of 10 helpers, the relative gains of adding more helpers are decreasing. Such observations provide useful insights for a real-life implementation of parallel SL and lead us towards future extensions of our approach where deployment or energy costs are included in our model.

VIII. Conclusions and Future Work

In this work, we formulated the joint problem of client-helper assignments and scheduling for parallel SL. We analyzed it both theoretically, proving it is NP-hard, and experimentally, using measurements from a realistic testbed. We proposed two solution methods, one based on the decomposition of the problem, and the other characterized by a low computation overhead. Our performance evaluations led us to build a bespoke solution strategy comprising these methods that are chosen based on the scenario's characteristics. We showed that this strategy finds a near-optimal makespan, while it can be tuned to balance suboptimality and speed. Also, it outperforms the baseline scheme by achieving a shorter makespan by up to 52.3%. A natural direction for future work would be to decide on the NN's cut layers per client in conjunction with the proposed solution strategy.

REFERENCES

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *AISTATS*. PMLR, 2017, pp. 1273–1282.

[2] W. Y. B. Lim, N. C. Luong, D. T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, and C. Miao, "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 3, pp. 2031–2063, 2020.

[3] L. L. Pilla, "Optimal task assignment for heterogeneous federated learning devices," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 661–670.

[4] Y. Jiang, S. Wang, V. Valls, B. J. Ko, W.-H. Lee, K. K. Leung, and L. Tassiulas, "Model pruning enables efficient federated learning on edge devices," *IEEE Trans. on Neural Networks and Learning Systems*, 2022.

[5] M. R. Sprague, A. Jalalirad, M. Scavuzzo, C. Capota, M. Neun, L. Do, and M. Kopp, "Asynchronous federated learning for geospatial applications," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2018, pp. 21–28.

[6] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," *arXiv preprint arXiv:1812.00564*, 2018.

[7] C. Thapa, P. C. M. Arachchige, S. Camtepe, and L. Sun, "Splitfed: When federated learning meets split learning," in *Proc. of the AAAI Conf. on Artificial Intelligence*, vol. 36, no. 8, 2022, pp. 8485–8493.

[8] J. Jeon and J. Kim, "Privacy-sensitive parallel split learning," in *International Conf. on Information Networking*. IEEE, 2020, pp. 7–9.

[9] Z. Zhang, A. Pinto, V. Turina, F. Esposito, and I. Matta, "Privacy and efficiency of communications in federated split learning," *IEEE Transactions on Big Data*, 2023.

[10] K. Palanisamy, V. Khimani, M. H. Moti, and D. Chatzopoulos, "Spliteasy: A practical approach for training ml models on mobile devices," in *Proc. of the 22nd International Workshop on Mobile Computing Systems and Applications (ACM HotMobile)*, 2021, p. 37–43.

[11] Z. Wang, H. Xu, Y. Xu, Z. Jiang, and J. Liu, "CoopFL: Accelerating federated learning with dnn partitioning and offloading in heterogeneous edge computing," *Computer Networks*, vol. 220, p. 109490, 2023.

[12] Z. Jiang, Y. Xu, H. Xu, Z. Wang, and C. Qian, "Adaptive control of client selection and gradient compression for efficient federated learning," *arXiv preprint arXiv:2212.09483*, 2022.

[13] A. Rodio, F. Faticanti, O. Marfoq, G. Neglia, and E. Leonardi, "Federated learning under heterogeneous and correlated client availability," *Proc. of IEEE INFOCOM*, pp. 1–10, 2023.

[14] C. Chen, H. Xu, W. Wang, B. Li, B. Li, L. Chen, and G. Zhang, "GIFT: Toward accurate and efficient federated learning with gradient-instructed frequency tuning," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 4, pp. 902–914, 2023.

[15] H. Liu, F. He, and G. Cao, "Communication-efficient federated learning for heterogeneous edge devices based on adaptive gradient quantization," in *Proc. of IEEE INFOCOM*, 2023, pp. 1–10.

[16] X. Liu, Y. Deng, and T. Mahmoodi, "Energy efficient user scheduling for hybrid split and federated learning in wireless uav networks," in *Proc. of IEEE ICC*, 2022, pp. 1–6.

[17] S. Wang, X. Zhang, H. Uchiyama, and H. Matsuda, "Hivemind: Towards cellular native machine learning model splitting," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 2, pp. 626–640, 2021.

[18] J. Tirana, C. Pappas, D. Chatzopoulos, S. Lalis, and M. Vavalis, "The role of compute nodes in privacy-aware decentralized ai," in *Proc. of the 6th International Workshop on Embedded and Mobile Deep Learning*, 2022, pp. 19–24.

[19] M. Kim, A. DeRieux, and W. Saad, "A bargaining game for personalized, energy efficient split learning over wireless networks," in *Wireless Communications and Networking Conf.(WCNC)*. IEEE, 2023, pp. 1–6.

[20] E. Samikwa, A. Di Maio, and T. Braun, "ARES: Adaptive resource-aware split learning for internet of things," *Computer Networks*, vol. 218, p. 109380, 2022.

[21] W. Wu, M. Li, K. Qu, C. Zhou, X. Shen, W. Zhuang, X. Li, and W. Shi, "Split learning over wireless networks: Parallel design and resource management," *IEEE Journal on Selected Areas in Communications*, vol. 41, no. 4, pp. 1051–1066, 2023.

[22] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B. Shmoys, "Sequencing and scheduling: Algorithms and complexity," *Handbooks in operations research and management science*, 1993.

[23] J. K. Lenstra, D. B. Shmoys, and É. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical programming*, vol. 46, pp. 259–271, 1990.

[24] B. Chen, C. N. Potts, and G. J. Woeginger, "A review of machine scheduling: Complexity, algorithms and approximability," *Handbook of Combinatorial Optimization: Volume1–3*, pp. 1493–1641, 1998.

[25] A. M. Geoffrion, "Generalized benders decomposition," *Journal of optimization theory and applications*, vol. 10, pp. 237–260, 1972.

[26] J. Lou, Z. Tang, W. Jia, W. Zhao, and J. Li, "Startup-aware dependent task scheduling with bandwidth constraints in edge computing," *IEEE Transactions on Mobile Computing*, 2023.

[27] H. Wang, W. Li, J. Sun, L. Zhao, X. Wang, H. Lv, and G. Feng, "Low-complexity and efficient dependent subtask offloading strategy in iot integrated with multi-access edge computing," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2023.

[28] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, 2011.

[29] S. Diamond, R. Takapoui, and S. Boyd, "A general system for heuristic solution of convex problems over nonconvex sets," *arXiv preprint arXiv:1601.07277*, 2016.

[30] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with ADMM," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[31] A. S. Schulz and M. Skutella, "Scheduling unrelated machines by randomized rounding," *SIAM journal on discrete mathematics*, vol. 15, no. 4, pp. 450–469, 2002.

[32] N. H. Tran, W. Bao, A. Zomaya, M. N. Nguyen, and C. S. Hong, "Federated learning over wireless networks: Optimization model design and analysis," in *Proc. of IEEE INFOCOM*, 2019, pp. 1387–1395.

[33] Z. Fu, J. Ren, D. Zhang, Y. Zhou, and Y. Zhang, "Kalmia: A heterogeneous qos-aware scheduling framework for dnn tasks on edge servers," in *Proc. of IEEE INFOCOM*, 2022, pp. 780–789.

[34] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in *Proc. of IEEE INFOCOM*, 2019, pp. 2287–2295.

[35] S. P. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.

[36] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023. [Online]. Available: https://www.gurobi.com

[37] S. Ullman, "Complexity of sequencing problems," *Computers and Job-shop Scheduling*, 1967.

[38] T. Gonzalez, E. L. Lawler, and S. Sahni, "Optimal preemptive scheduling of two unrelated processors," *ORSA Journal on Computing*, vol. 2, no. 3, pp. 219–224, 1990.

[39] W. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, vol. 21, no. 1, pp. 177–185, 1974.

[40] J. Eckstein and D. P. Bertsekas, "On the Douglas—Rachford splitting method and the proximal point algorithm for maximal monotone operators," *Mathematical Programming*, vol. 55, no. 1, pp. 293–318, 1992.

[41] K. R. Baker, E. L. Lawler, J. K. Lenstra, and A. H. Rinnooy Kan, "Preemptive scheduling of a single machine to minimize maximum cost subject to release dates and precedence constraints," *Operations Research*, vol. 31, no. 2, pp. 381–386, 1983.

[42] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. of the workshop on Experimental computer science*, 2007.

[43] J. M. Shah, K. Kotecha, S. Pandya, D. Choksi, and N. Joshi, "Load balancing in cloud computing: Methodological survey on different types of algorithm," in *International conference on trends in electronics and informatics (ICEI)*. IEEE, 2017, pp. 100–107.

[44] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[45] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[47] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[48] L. Bai, W. Ji, Q. Li, X. Yao, W. Xin, and W. Zhu, "Dnnabacus: Toward accurate computational cost prediction for deep neural networks," *arXiv preprint arXiv:2205.12095*, 2022.

[49] D. Belson, "State of the Internet Q4 2016 report," *Akamai Technologies*, vol. 9, no. 4, 2017.

[50] R. Anand, D. Aggarwal, and V. Kumar, "A comparative analysis of optimization solvers," *Journal of Statistics and Management Systems*, vol. 20, no. 4, pp. 623–635, 2017.