Minimization of the Training Makespan in Hybrid Federated Split Learning

Tirana, Joana; Tsigkari, Dimitra; Iosifidis, George; Chatzopoulos, Dimitris

DOI
10.1109/TMC.2025.3533033

Publication date
2025

Document Version
Final published version

Published in
IEEE Transactions on Mobile Computing

Citation (APA)
Tirana, J., Tsigkari, D., Iosifidis, G., & Chatzopoulos, D. (2025). Minimization of the Training Makespan in Hybrid Federated Split Learning. *IEEE Transactions on Mobile Computing*, *24*(6), 5400-5417. https://doi.org/10.1109/TMC.2025.3533033

Minimization of the Training Makespan in Hybrid Federated Split Learning

Joana Tirana, Dimitra Tsigkari, George Iosifidis, and Dimitris Chatzopoulos

***Abstract*—Parallel Split Learning (SL) allows resource-constrained devices that cannot participate in Federated Learning (FL) to train deep neural networks (NNs) by splitting the NN model into parts. In particular, such devices (clients) may offload the processing task of the largest model part to a computationally powerful helper, and multiple helpers may be employed and work in parallel. In hybrid federated and split learning (HFSL), on the other hand, devices can participate in the training process through any of the two protocols (SL and FL), depending on the system's characteristics. This could considerably reduce the maximum training time over all clients (makespan), especially in highly heterogeneous scenarios. In this paper, we study the joint problem of the training protocol selection, client-helper assignments, and scheduling decisions, to minimize the training makespan. We prove this problem is NP-hard and propose two solution methods: one based on the decomposition of the problem by leveraging its inherent symmetry, and a second fully scalable one. Through numerical evaluations using our testbed's measurements, we build a solution strategy comprising these methods. Moreover, this strategy finds a near-optimal solution and achieves a shorter makespan than the baseline schemes by up to 71%.**

***Index Terms*—Federated learning, split learning, distributed learning optimization.**

I. Introduction

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) [2]. 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 [3]. As a result of these factors, the training time of some clients might be prohibitively long, thus, slowing down this cross-device 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 [3], [4]. While state-of-the-art FL approaches propose ways of alleviating this phenomenon, e.g., via model pruning [5] or asynchronous protocols [6], they may compromise the accuracy of the produced model. Moreover, clients with limited memory capacity (e.g., IoT devices) might be unable to participate in FL processes that train large ML models.

In the cloud-edge computing schema, when the extreme-edge devices (the clients in our case) cannot support the computing or the memory demands that are required to keep up with the system's needs, a solution is to offload part of the computing burden into the cloud, or into another (more powerful) edge node device. In that context, split learning (SL) protocols have been recently proposed to enable resource-constrained clients to train neural networks (NNs) of millions of parameters [7]. 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 beyond 5 G networks. Formally, an 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* $\{\sigma_1, \sigma_2\} \in \boldsymbol{\sigma}$. 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 [8], [9], 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. Parallel SL reduces

Received 24 July 2024; revised 4 December 2024; accepted 16 January 2025. Date of publication 23 January 2025; date of current version 7 May 2025. This work was supported in part by the Horizon Europe research and innovation program of the European Union under Grant 101092912, project MLSysOps, in part by the EU Horizon Europe project under Grant 101093006 (TaRDIS), in part by the Spanish Ministry of Economic Affairs and Digital Transformation and the European Union-NextGenerationEU through the project 6G-RIEMANN under Grant TSI-063000-2021-147, in part by the National Growth Fund through the Dutch 6 G flagship project "Future Network Services", and in part by the European Commission under Grant 101139270 (ORIGAMI) and Grant 101192462 (FLECON-6G). Part of this work appeared in the proceedings of IEEE INFOCOM 2024 [1]. Recommended for acceptance by L. Guo. *(Corresponding author: Joana Tirana.)*

Joana Tirana and Dimitris Chatzopoulos are with the School of Computer Science, University College Dublin, Dublin 4 Dublin, Ireland (e-mail: joana.tirana@ucdconnect.ie; dimitris.chatzopoulos@ucd.ie).

Dimitra Tsigkari is with Telefónica Research, 28050 Madrid, Spain (e-mail: dimitra.tsigkari@telefonica.com).

George Iosifidis is with the Department of Software Technology, Delft University of Technology, 2628 Delft, The Netherlands (e-mail: g.iosifidis@tudelft.nl).

Digital Object Identifier 10.1109/TMC.2025.3533033

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

the makespan (when compared to the conventional SL) without compromising the model accuracy [8], [10], [11].

An even more practical approach is considering a hybrid federated and (parallel)[2] split learning (HFSL) [12], [13] setting, to manage both resource-constrained clients and more powerful ones (capable of supporting on-device training). In this setup, clients may train the NN model either through SL in collaboration with a helper or through FL, i.e., train the entire model locally. The motivation for the hybrid approach is that clients with higher-end devices may train through FL without increasing the makespan. Consequently, this will reduce the load and alleviate the resource demands on the helpers which will assist only the resource-constrained clients.

Also, in SL, the presence of multiple helpers working in parallel can further reduce the training makespan [14]. Because it can easily support a larger-scaled system while considering managing clients in different geographical locations. Hence, helpers are placed in multiple locations to serve clients by providing faster communication links. But, orchestrating the workflow of HFSL in this network of entities (as depicted in Fig. 1) is one of the main challenges that the HFSL paradigm faces, as discussed in [3]. In detail, the factors that one needs to consider are the computation-memory resources of all entities and the connectivity of the clients to the helpers.

Methodology & Contributions. Driven by the time measurements of our testbed and the heterogeneity even among devices of similar capabilities, we identify three key decisions: 1) the *training protocol* employed by each client (FL or SL), and in the case of SL, we further need to decide: 2) the *client-helper assignment*, which is tied to the helpers' memory and computing capacities; 3) the *scheduling*, i.e., the order in which each helper processes the offloaded tasks. These 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 to minimize the makespan.

We extend the model and the optimization problem we constructed in our previous work [1], which was solely focused on the workflow of SL. Hence, modifications must be made to support the combination of FL and SL. 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 (by proving it is NP-hard) and experimentally (by 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 training protocol selection for each client, the assignment, and the forward-propagation scheduling variables. 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 into 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.

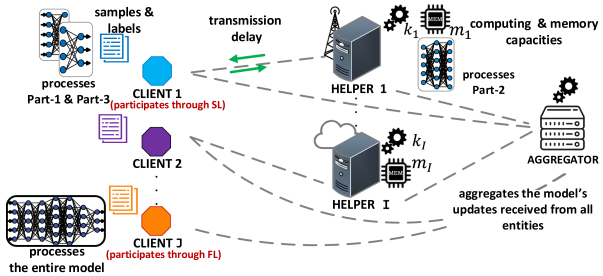

Fig. 1. The hybrid federated and split learning setting considered in this work, the considered network topology, its resources, and the processing tasks per entity. Some clients, e.g., client 1, participate in the training through parallel SL, by offloading part-2 to a helper, while others, e.g., client J, participate through FL and perform on-device training.

- Formulate the joint problem of training-protocol selection, client-helper assignments, and scheduling decisions to minimize the training makespan in HFSL, and prove it is NP-hard.
- Propose a solution method based on decomposing the problem into two subproblems. For the first one, ADMM is employed, while a polynomial-time algorithm is provided for the second one.
- Propose a heuristic algorithm with minimal overheads.
- Extend our model in two different ways, to account for (i) the preemption cost, and (ii) the energy consumption cost.
- Perform numerical evaluations of the proposed methods using collected data from our testbed.[3] These findings shape a solution strategy based on the scenario at hand.
- Show that our solution strategy finds a near-optimal solution and achieves a shorter makespan than the baseline schemes by up to 71%, and on average by 25%.

The rest of the paper is organized as follows. First, in Section II, we provide an overview of related work. Then, in Sections III and IV, we present the system model and the formulation of the joint problem of training protocol selection, client-helper assignment, and scheduling decisions, respectively. These are followed by the solution methods in Sections V and VI-A. Then, we present extensions of our model that include additional costs in the objective function; the preemption cost in Section VI-B, and the energy consumption in Section VII. Section VIII presents the numerical evaluations and, finally, Section IX concludes the paper.

II. Related Work

A. Client-Based Distributed ML

Research on FL mainly focuses on achieving good accuracy while minimizing the wall-clock time, through client selection

[2]This work focuses on parallel SL, as described above, and not on conventional SL, thus, we drop the word parallel, unless otherwise specified.

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

strategies [15], or aggregation algorithms [16], [17], or by taking into account the communication overhead [15], [18]. However, some clients might be unable to support the computation demands of such protocols (e.g., on-device training is not feasible due to limited memory capacity, or clients miss aggregation deadlines due to network delays of slow computing). This has motivated the creation of SL [7], in which the computing demands on the clients are relaxed by offloading a big part of the model into a helper. Consequently, SL reduces the risk of missed synchronization deadlines and enables the training of deeper ML models.

Going a step further, SL combined with FL, i.e., SplitFed or parallel SL [8], [19] can lead to further acceleration and scalability. Like in FL, clients can train their model parts in parallel while preserving the FL's convergence guarantees.

B. Optimizing SL

Usually, existing works model a system consisting of multiple clients and a single helper. In particular, they focus on finding the NN's cut layers while trying to optimize the energy consumption [20], [21], the training makespan [22], or privacy [10]. However, in the presence of multiple clients, the system may need to be scaled up, in order to: (i) speed up the training process; having a single helper will result in large queuing delays which leads to the *starvation* of some clients, (ii) satisfy the memory demands of SL; the helper's memory demands increase as the number of participants is increased [23], and (iii) provide robustness because having one helper translates to having a single point of failure. The first two points directly impact the makespan, while the last point concerns fault tolerance.

To minimize the training makespan for systems with multiple helpers requires careful workflow orchestration. Close to this idea, the work in [14] 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.

C. Hybrid Federated and Split Learning (HFSL)

HFSL was first introduced in [12], [13] and was inspired by the heterogeneity of devices participating in distributed ML training. In the presence of high heterogeneity, the computational load of helpers may be reduced if powerful devices run the training tasks locally. Specifically, in [12], selected clients will employ the FL protocol or SL, while in [13], clients select between FL and the SplitFed [8] protocol. This motivated us to expand and update accordingly our original model [1] towards a hybrid setting, where some clients may participate in the training through SL, and others - typically fast devices- through FL.

The main objective in existing related work is minimizing energy consumption without considering the makespan and the effects of increased training delay. Therefore, we not only expand the original model in [1] to consider the hybrid FL and SL approach but we also propose an extension that considers both makespan and energy consumption through a balanced objective. Our analysis shows that, when targeting solely on reducing the per-iteration energy, this can lead to a significant increase in training time. In this case, clients need to be online and available for a longer time which leads to a considerable consumption of their battery capacity. Moreover, even though the clients in [12] may train through SL, the model does not consider the queuing delay at the helper (as our model does).

D. Workflow Optimization

Joint problems of assignment and scheduling decisions, such as the parallel machine scheduling problem, are often NP-hard, see, e.g., [24], [25], [26]. While a first approach would be to rely on methods such as branch-and-bound or column or row generation methods (like benders decomposition [27]), 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 [28], [29]), we decompose the problem based on the inherent structure of the SL operations. Next, we solve one of the resulting subproblems with ADMM from the toolbox of convex optimization [30]. ADMM is an iterative method that can converge to the optimal under mild assumptions and has been employed in a variety of networking problems, e.g., [31], [32]. Moreover, it has been recently found to perform remarkably well for non-convex or integer problems [33], [34], [35] with even partial convergence guarantees in some cases. 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 and stopping criteria [33], and thus, we tailor it to leverage the nature of the subproblem at hand.

III. System Model & Workflow of Training

This section formally presents the entities of the HFSL system and their role, as well as the main steps of the training workflow (i.e., the training phase). Further, it introduces the relevant decision variables of the considered problem. Table I at the end of this section summarizes the introduced problem parameters and variables.

A. Network Topology & Entities

We consider a system with a set $\mathcal{J}$ of $J = |\mathcal{J}|$ clients, e.g., IoT or handheld devices, that wish to collaborate and train an NN model using their data. There is also a set $\mathcal{I}$ of $I = |\mathcal{I}|$ available 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. In the HFSL setting, clients may train the model either locally (through FL) or partially locally and partially offload part of the training to helpers. Therefore, extending the model presented in [1], the set of helpers in HFSL is updated to $\mathcal{I}' := \mathcal{J} \cup \mathcal{I}$, where a node with index $i > I$ is in practice a client (i.e., the client j with index equal to $i \bmod |\mathcal{I}|$ has a corresponding index $I + j$ in the set $\mathcal{I}'$). Similarly, $\mathcal{E}$ is extended to $\mathcal{E}'$ to contain the "loopback" links for clients using their own devices. However, as is explained in

TABLE I
SUMMARY OF NOTATION

Symbol	Definition
$\mathcal{J}$	Set of clients
$\mathcal{I}$	Set of helpers
$\mathcal{I}'$	The union set of helpers and clients
L	Length of NN model (i.e., number of layers)
$\mathcal{T}$	Time horizon, has length T
σ_1, σ_2	First and last cut layer
k_n	Computing capacity of node n
m_n	Memory capacity of node n
bw_{ij}	Bandwidth of comm. link between client j and helper i
d_j	Memory footprint of client j when offloading part-2
r_j	`fwd-prop` for part-1
r'_j	`bwd-prop` for part-1
l_j	`fwd-prop` for part-3
l'_i	`bwd-prop` for part-3
p_{ij}	Helper i applying `fwd-prop` of part-3
p'_{ij}	Helper i applying `bwd-prop` of part-3
$tr_{ij}^{a_1}$, $tr_{ij}^{a_2}$	Sending activation of first and second cut layer
$tr_{ij}^{gr_1}$, $tr_{ij}^{gr_2}$	Sending gradients of first and second cut layer
y_{ij}	Variable for assignments & train. protocol selection
x_{ijt}	Variable for scheduling `fwd-prop` tasks
z_{ijt}	Variable for scheduling `bwd-prop` tasks
c_j	Completion time of whole batch update
P_j^c	Computing power consumption of client j
P_j^t	Transmission power consumption of client j
P_j^r	Receiving power consumption of client j

the following sections, the under-study system model does not require established communication links between clients.[4]

The nodes are potentially heterogeneous in terms of hardware and/or wireless connectivity. Namely, each node $i \in \mathcal{I}'$ has computing capacity k_i (cycles/sec) and memory capacity m_i Gbytes. Further, we denote by bw_{ji} the bandwidth of the network link between client j and helper i, $\forall (j,i) \in \mathcal{E}$, and, w.l.o.g., we assume symmetric links, i.e., $bw_{ij} = bw_{ji}$. All nodes are connected to an aggregator, indexed $n=0$, which may collect the necessary information and orchestrate the workflow using the solution strategy we develop.

B. Workflow of HFSL

The clients can either collaborate with the helpers to train a large NN through SL or train the model locally through FL, if it is allowed by the capabilities of their devices. This decision is made individually for each client while minimizing the makespan. Each client owns a dataset that is divided into batches of equal size. During training every batch is iteratively propagated into the model, to update the model weights; we refer to this operation as *batch update*.

When SL is selected for a client, 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, while it offloads part-2 to a helper. This SL architecture [7] protects the privacy of the clients' data since the samples and labels are kept locally. Whereas, when FL is selected, the client is in charge of the whole model; part-1, part-2, and part-3. Also, our analysis is oblivious to a) the cut layers, which are decided in advance and may differ across the clients, b) the ML application in question, and c) the training hyperparameters (e.g., batch size, learning rate, etc.), ensuring that the resulting model accuracy is unaffected.

Epochs & Aggregation. The batch processing workflow is repeated for all batches. When the clients have 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` [2]. Typically, such training processes require hundreds of training rounds, each consisting of multiple batch updates [8]. Hence, 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 [14], [21]. Moreover, transmitting model updates can even start before all entities have completed a batch update, thus speeding up the procedure. Further, note that in SL clients transfer only part-1 and part-3 to the aggregator.

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 SL. We employ a time-slotted model [37] with time intervals that, w.l.o.g., are of unit-length.[5] The client applies `fwd-prop` of part-1 and transmits the activations of the first cut layer (σ_1) to the helper. We denote by r_j and $tr_{ij}^{a_1}$, respectively the number of time slots required for these two operations, which depends on k_j and bw_{ij}, respectively. The helper needs p_{ij} time slots to `fwd-prop` these activations into part-2, 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 `fwd-prop` by processing part-3 and computing the loss. We denote by l_i and $tr_{ij}^{a_2}$, respectively the time required for these operations, which depends on bw_{ij}, the data size, and k_j, respectively.

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'_j and $tr_{ij}^{gr_2}$ time slots. The helper back-propagates these gradients into part-2, 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'_j and $tr_{ij}^{gr_1}$ time slots the time required for these final steps.

When the FL protocol is selected for a client, the only difference in the training workflow (when compared to SL) is the absence of activation/gradient exchange with any helper, i.e., $tr_{ij}^{a_1} = tr_{ij}^{a_2} = tr_{ij}^{gr_1} = tr_{ij}^{gr_2} = 0$, for $i > I$.

Finally, the time-related parameters (or delays) $\boldsymbol{r} = (r_j, j \in \mathcal{J})$, $\boldsymbol{r'} = (r'_j, j \in \mathcal{J})$, $\boldsymbol{p} = (p_{ij}, (i,j) \in \mathcal{E}')$, $\boldsymbol{p'} = (p'_{ij}, (i,j) \in \mathcal{E}')$, $\boldsymbol{l} = (l_j, j \in \mathcal{J})$, $\boldsymbol{l'} = (l'_j, j \in \mathcal{J})$ and $\boldsymbol{tr} = (tr_{ij}^{a_k}, (i,j) \in \mathcal{E}', k \in \boldsymbol{\sigma})$, $\boldsymbol{tr} = (tr_{ij}^{gr_k}, (i,j) \in \mathcal{E}', k \in \boldsymbol{\sigma})$ represent average

[4] While device-to-device communications may offer high speed, which could alleviate the communication delays in SL, they also come with technical challenges, e.g., privacy and security issues [36]. Moreover, such communications are typically feasible only for devices within the same region. In contrast, we consider a more general scenario where devices may be located anywhere.

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

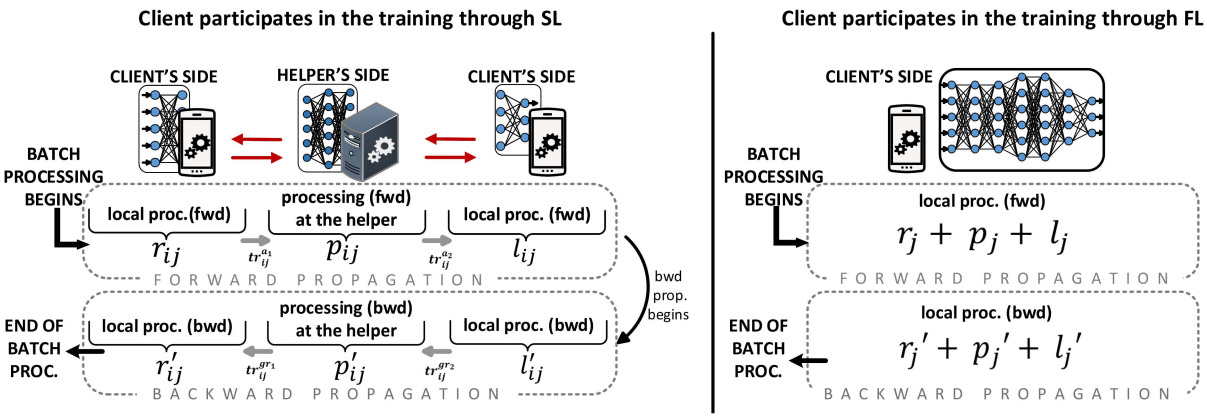

Fig. 2. The workflow of the batch update for a single client-helper pair in the SL setting, and the corresponding times, i.e., processing and transmission (left). The *queuing delay* that a client might experience at the helper is not depicted here. In contrast, in the FL setting (right), no helper is employed, and, thus, no transmission and queuing delays occur.

quantities[6] for these tasks, and are considered available through profiling and other offline measurement methods [38], [39].

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`. In practice, a client cannot use a different helper for each propagation direction.

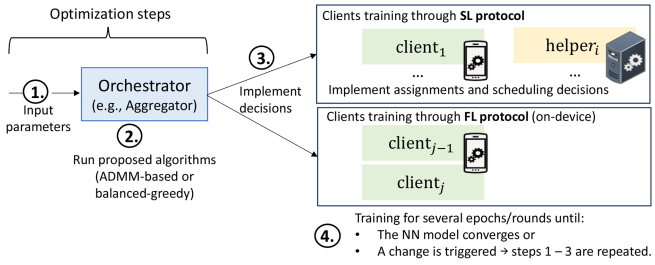

Fig. 3. Overview of the workflow of our system model.

C. Time Horizon & Decision Variables

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 is calculated as follows:

$$T := \max_{(i,j)\in\mathcal{E}'} \left\{ r_j + l_j + r'_j + l'_j + \sum_{k\in\sigma} (tr^{a_k}_{ij} + tr^{gr_k}_{ij}) \right\} + \max \left\{ \sum_{j\in\mathcal{J}} \max_{i\in\mathcal{I}} \left\{ p_{ij} + p'_{ij} \right\}, \max_{j\in\mathcal{J}} \left\{ p_{(I+j)j} + p'_{(I+j)j} \right\} \right\}, \quad (1)$$

where the first term finds the worst-case transmission and processing times for part-1 and part-3. The second term measures the worst-case for processing part-2, which can be implemented by the helpers or by each client individually.

Based on this time-slotted model, we introduce variables that inject tasks to helpers to minimize 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. However, we note that $i \in \mathcal{I}'$, and hence, y not only defines the client-helper assignment, it also implicitly determines the training protocol for each client. In detail, if $y_{ij} = 1$ with $i \in \mathcal{I}$ or index-wise it has a value smaller than $|\mathcal{I}|$, then the client j will train through the SL protocol and model part-2 of client j will be offloaded to helper i. However, if $i = j + |\mathcal{I}|$ (i.e., $i > |\mathcal{I}|$), this implies that the client does not offload the model part to any helper and trains the entire model locally, as in FL. In the next section, the problem constraints will be defined in a way that best reflects the dual role of the variables $\boldsymbol{y}$. 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

In this section, we formulate the optimization problem of minimizing the training makespan in HFSL. The role of this optimization in the training workflow in the HFSL setting is depicted in Fig. 3 and is managed by the orchestrator (e.g., the aggregator). This section contains a detailed analysis of its constraints and the definition of its objective function. Also, it presents the proof that the problem is NP-hard.

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

A. Problem Constraints Formulation

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

$$x_{ijt} = 0, \quad \forall t < r_j + tr_{ij}^{a_1}, \ (i,j) \in \mathcal{E}'. \tag{2}$$

In scheduling parlance, $r_j + tr_{ij}^{a_1}$ is the *release time* of this task, i.e., when it becomes "available" at the helper. Similarly, the `bwd-prop` can start only after the gradients of $\sigma_2 + 1$ have been received by the helper (see Fig. 2), and thus,

$$z_{ij(t+l_j+l'_j+tr_{ij}^{a_2}+tr_{ij}^{gr_2})} \leq \frac{1}{p_{ij}} \sum_{\tau=0}^{t-1} x_{ij\tau}, \quad \forall (i,j) \in \mathcal{E}', t \in \mathcal{T}. \tag{3}$$

That is, in order to assign the `bwd-prop` task of j to i at (or after) slot $t + l_j + l'_j + tr_{ij}^{a_2} + tr_{ij}^{gr_2}$, as we need to allocate enough processing time at i (at least p_{ij}) for `fwd-prop` until slot t. Essentially, (3) 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):

$$\sum_{j \in \mathcal{J}} (x_{ijt} + z_{ijt}) \leq 1, \quad \forall i \in \mathcal{I}', t \in \mathcal{T}. \tag{4}$$

Training Protocol Selection and Assignment Constraints. Regarding the decisions $\boldsymbol{y}$, each client's task is either assigned to a single helper (i.e., the SL protocol is selected) or processed locally at the client (i.e., the FL protocol is selected):

$$\sum_{i \in \mathcal{I}'} y_{ij} = 1, \quad \forall j \in \mathcal{J}. \tag{5}$$

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

$$\sum_{j \in \mathcal{J}} y_{ij} d_j \leq m_i, \quad \forall i \in \mathcal{I}', \tag{6}$$

However, clients may offload the intermediate model part only to helpers, and not to other clients (see footnote 4):

$$y_{ij} = 0, i > I \text{ and } i \neq I + j \tag{7}$$

Also, the assignment and scheduling constraints are tightly coupled. 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:

$$\sum_{t \in \mathcal{T}} x_{ijt} = y_{ij} p_{ij}, \quad \forall (i,j) \in \mathcal{E}' \text{ and} \tag{8}$$

$$\sum_{t \in \mathcal{T}} z_{ijt} = y_{ij} p'_{ij}, \quad \forall (i,j) \in \mathcal{E}'. \tag{9}$$

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:

$$\phi_j \geq (t+1) z_{ijt}, \quad \forall (i,j) \in \mathcal{E}', t \in \mathcal{T}. \tag{10}$$

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

$$c_j = \phi_j + \sum_{i \in \mathcal{I}'} (r'_j + tr_{ij}^{gr_1}) y_{ij} \quad \forall j \in \mathcal{J}. \tag{11}$$

Essentially, the vector $\boldsymbol{c} = (c_j, j \in \mathcal{J})$ contains the completion times of one batch update 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 $\phi_j - \sum_i y_{ij}(r_j + tr_{ij}^{a_1} + p_{ij} + tr_{ij}^{a_2} + l_j + l'_j + tr_{ij}^{gr_2} + p'_{ij})$ 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. Unlike preemptions in the scheduling literature [26], in the SL context, a processing task can be resumed only on the same helper due to the memory footprint of the task. 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 [14], but in line with related work on edge computing, e.g., [39], [40]. We further discuss this point in Section VI-B. 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 Section VIII.

B. Problem Definition and NP-Hardness

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

Problem 1 (Batch Training Makespan).

$$\mathbb{P}: \underset{\boldsymbol{x},\boldsymbol{z},\boldsymbol{y},\boldsymbol{\phi},\boldsymbol{c}}{\text{minimize}} \ \max_{j \in \mathcal{J}} \{c_j\}$$

$$\text{s.t.} \quad (2)\text{-}(11),$$

$$\boldsymbol{x}, \boldsymbol{z} \in \{0,1\}^{|\mathcal{E}'| \times |\mathcal{T}|}, \boldsymbol{\phi}, \boldsymbol{c} \in \{0..T\}^J, \tag{12}$$

$$\boldsymbol{y} \in \{0,1\}^{|\mathcal{E}'|}. \tag{13}$$

This min-max problem can be written as an Integer Linear Program (ILP) using standard transformations [41, Sec. 4.3.1], i.e., by introducing the worst-case makespan variable ψ and changing the objective to $\min_{\psi,\boldsymbol{x},\boldsymbol{y},\boldsymbol{z},\boldsymbol{\phi}} \psi$ with additional constraints $\psi \geq c_j, \forall j \in \mathcal{J}$. 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 [42], achieve only a 40% optimality gap in 14 hours (off-the-shelf computer). Such long delays are not surprising due to the following result [1].

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

Proof: We first define a simpler instance of $\mathbb{P}$: we assume that all devices train through SL, e.g., due to their limited capabilities, hence, $y_{ij} = 0$ for all $i > \mathcal{I}$. Further, we assume that all helpers have enough memory for all tasks, i.e., we drop

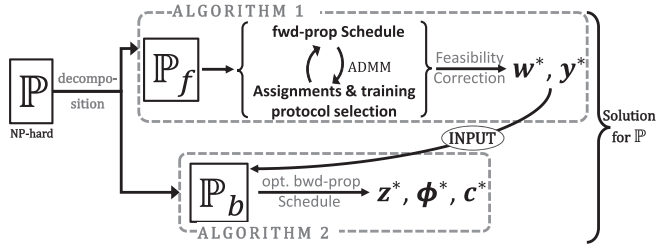

Fig. 4. The roadmap to our ADMM-based solution method.

the memory constraints in (6), and $\boldsymbol{r} = \boldsymbol{r}' = \boldsymbol{l} = \boldsymbol{l}' = \boldsymbol{p}' = \mathbf{0}$, $\boldsymbol{tr}^{a_1} = \boldsymbol{tr}^{a_2} = \boldsymbol{tr}^{gr_1} = \boldsymbol{tr}^{gr_2} = \mathbf{0}$, i.e., propagations and transmissions of part-1 and 3 and the backward propagation at the helpers are instantaneous. Therefore, in this instance, only the `fwd-prop` tasks need to be scheduled and they are released at time 0. Moreover, we assume $I = 2$, i.e., the system contains only 2 helpers, and $p_{ij} = p_j \ \ \forall j$, i.e., the length of all tasks is independent of the helper. We will show that there is a polynomial-time reduction from the parallel (identical) machine scheduling problem of 2 machines, denoted $P2 \mid\mid C_{\max}$, see [24], [43]. The former is defined as follows: given a set of n jobs and a set of 2 parallel identical machines, each job should be assigned to a machine, while every machine can process at most one job at a time, with the objective to find a schedule and the job-machine assignments to minimize the makespan. The reduction is shown by setting $n = J$, i.e., the `fwd-prop` tasks are the jobs, $J = 2$, i.e., the helpers are the 2 identical machines, hence $p_{ij} = p_j \ \forall j$. Finally, we note that in this specific instance, enabling preemption will not have any impact on the makespan, since, by design, the preemption cannot be from one helper to the other in our problem. Given this reduction and the fact that the parallel machine scheduling problem is NP-hard [24], [43], $\mathbb{P}$ is NP-hard as well. □

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

V. Solution Method

The core idea of our solution method is to decompose $\mathbb{P}$ into two subproblems (see Fig. 4): *(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-propagation makespan by deciding $\boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{c}$. We solve $\mathbb{P}_f$ using ADMM (see discussion in Section 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 Section VIII, this approach will lead to considerable speedups (up to $16\times$) 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

$$T_f := \max_{(i,j)\in\mathcal{E}', k\in\boldsymbol{\sigma}} \{r_j + l_j + tr_{ij}^{a_k}\} + \max\left\{\max_{j\in\mathcal{J}} p_{(I+j)j}, \sum_{j\in\mathcal{J}} \max_{i\in\mathcal{I}} p_{ij}\right\}$$

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 (10)):

$$\phi_j^f \geq (t+1)x_{ijt}, \ \forall i \in \mathcal{I}', t \in \mathcal{T}_f. \tag{14}$$

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_j, i.e.,

$$c_j^f = \phi_j^f + \sum_{i\in\mathcal{I}'} (l_j + tr_{ij}^{a_2}) y_{ij}, \ \forall j \in \mathcal{J}. \tag{15}$$

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**).*

$$\mathbb{P}_f: \ \underset{\boldsymbol{x},\boldsymbol{y},\boldsymbol{\phi}^f,\boldsymbol{c}^f}{\text{minimize}} \ \max_{j\in\mathcal{J}} \left\{c_j^f\right\}$$

$$\text{s.t.} \quad (2), (5)-(8), (13), (14), (15)$$

$$\sum_{j\in\mathcal{J}} x_{ijt} \leq 1, \quad \forall i \in \mathcal{I}', t \in \mathcal{T}_f, \tag{16}$$

$$\boldsymbol{x} \in \{0,1\}^{|\mathcal{E}'|\times|\mathcal{T}|}, \boldsymbol{\phi}^f, \boldsymbol{c}^f \in \{0..T_f\}^J. \tag{17}$$

Comparing the constraints of this reduced problem with $\mathbb{P}$, we observe that: $\mathbb{P}_f$ replaces constraints (10) and (11) with (14) and (15); omits constraints (3) and (9); and replaces (4) with (16). 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 Section 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 [24], [44], [45].

To that end, we employ ADMM to decompose $\mathbb{P}_f$ and obtain smaller subproblems, which, it turns out, can be solved in a reasonable time for many problem instances. Indeed, we observe that by relaxing the constraints in (8), 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 recover the relaxed constraints gradually. The first step is to define the Augmented Lagrange function:

$$\mathcal{L}(\boldsymbol{w}, \boldsymbol{y}, \boldsymbol{\lambda}) = \max_{j\in\mathcal{J}} c_j^f + \sum_{(i,j)\in\mathcal{E}'} \lambda_{ij} \left(\sum_{t\in\mathcal{T}_f} x_{ijt} - y_{ij} p_{ij} \right) + \frac{\rho}{2} \sum_{(i,j)\in\mathcal{E}'} \left| \sum_{t\in\mathcal{T}_f} x_{ijt} - y_{ij} p_{ij} \right|, \tag{18}$$

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 $\quad \boldsymbol{w}^{(\tau+1)} = \underset{(2),(14)-(17),(22)}{\operatorname{argmin}} \mathcal{L}\big(\boldsymbol{w}, \boldsymbol{y}^{(\tau)}, \boldsymbol{\lambda}^{(\tau)}\big)$ *schedule*

3 $\quad \boldsymbol{y}^{(\tau+1)} = \underset{(5),(6),(13)}{\operatorname{argmin}} \mathcal{L}\big(\boldsymbol{w}^{(\tau+1)}, \boldsymbol{y}, \boldsymbol{\lambda}^{(\tau)}\big)$ *assign. & protocol selection*

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 (19) and (20) are satisfied.

6 Correct $\mathbb{P}_f$ feasibility with (21).

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

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 (8); and use the ADMM penalty parameter ρ, see [30, 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 (8) in line 4. We repeat these steps until convergence is achieved or a maximum number of iterations is reached ($\tau_{\max}$).[7] As a convergence flag, we use the detection of stationary assignments and objective values (line 5):

$$\sum_{(i,j) \in \mathcal{E}} \left| y_{ij}^{(\tau+1)} - y_{ij}^{(\tau)} \right| < \varepsilon_1 \quad \text{and} \tag{19}$$

$$\left| \max_{j \in \mathcal{J}} c_j^{f,(\tau+1)} - \max_{j \in \mathcal{J}} c_j^{f,(\tau)} \right| < \varepsilon_2. \tag{20}$$

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

$$\boldsymbol{w}^* = \underset{(2),(14)-(17),(8)}{\operatorname{argmin}} \mathcal{L}\left(\boldsymbol{w}, \boldsymbol{y}^*, \boldsymbol{\lambda}^*\right), \tag{21}$$

where we additionally use the constraints (8) to ensure full consistency between $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$. We remind that this step concerns only clients for which the SL protocol was selected.

Furthermore, we can accelerate the convergence of Algorithm 1 by creating a tighter constraint set [33]. 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:

$$\sum_{i \in \mathcal{I}'} \frac{1}{p_{ij}} \sum_{t \in \mathcal{T}_f} x_{ijt} = 1, \quad \forall j \in \mathcal{J}. \tag{22}$$

Moreover, one can employ other techniques that speed up the convergence of ADMM, such as varying penalty parameter ρ, varying order of solving the subproblems, etc. An analysis of such techniques can be found in [30].

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 Section VIII, where in most of the tested scenarios, it achieves less than 9.4% suboptimality gap, with one corner case of 15.9%.

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 easily. 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 $\boldsymbol{y}^*$, $\boldsymbol{w}^*$).*

$$\begin{aligned} \mathbb{P}_b: \ \underset{\boldsymbol{z}, \boldsymbol{\phi}, \boldsymbol{c}}{\text{minimize}} \ & \max_{j \in \mathcal{J}} c_j \\ \text{s.t.} \ \ & (3), (4), (9) - (11) \\ & \boldsymbol{z} \in \{0,1\}^{|\mathcal{E}'| \times |\mathcal{T}|}, \boldsymbol{\phi}, \boldsymbol{c} \in \{T_f^* .. T\}^J. \end{aligned} \tag{23}$$

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. Moreover, if for a client j we have $y_{(I+j)j} = 1$ (i.e., the FL training protocol is selected for this client), then scheduling decisions are not necessary for this client. Hence, $\mathbb{P}_b$ will be solved only for the subset of clients which the SL training protocol was selected during the previous step.

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 Algorithm 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.

$$\begin{aligned} \mathbb{P}_b^i: \ \underset{\boldsymbol{z}, \boldsymbol{\phi}}{\text{minimize}} \ & \max_{j \in \mathcal{J}_i} \{\phi_j + \pi_j\} \\ \text{s.t.} \ & \sum_{t \in \mathcal{T}_j} z_{jt} = p'_j, \quad \forall j \in \mathcal{J}_i \\ & z_{j(t+l_j+l'_j+tr_{ij}^{a_2}+tr_{ij}^{gr_2})} \leq \frac{1}{p_j} \sum_{\tau=0}^{t-1} x_{j\tau}^*, \ \forall j \in \mathcal{J}_i, t \in \mathcal{T}_i \\ & \sum_{j \in \mathcal{J}_i} z_{jt} \leq 1, \quad \forall t \in \mathcal{T} \\ & \phi_j \geq (t+1) z_{jt}, \quad \forall j \in \mathcal{J}_i, t \in \mathcal{T}_i, \end{aligned}$$

[7] The $\boldsymbol{w}$- and $\boldsymbol{y}$-subproblems (i.e., lines 2-3 of Algorithm 1) could be solved either with exact methods, e.g., branch and bound, or inexact methods, e.g., through a tailored relaxation [35]. As for the former, we elaborate on the resulting overhead in Section VIII, and the latter is in line with the fact that ADMM can (under certain conditions) tolerate inexact solutions for its subproblems [46].

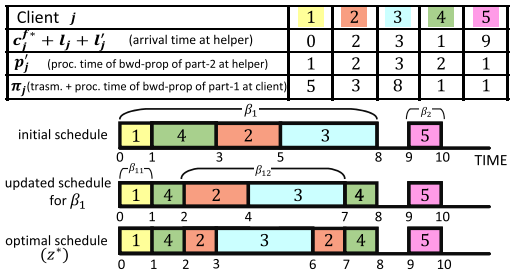

Fig. 5. Algorithm 2 for optimal bwd-prop schedule in a toy example of 5 clients and 1 helper.

where $\pi_j := \sum_{i\in\mathcal{I}}(r_j + tr_{ij}^{a_1})' y_{ij}^*, \forall j \in \mathcal{J}_i$ 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 polynomial time solvable [47]. It suffices to set the release times of the jobs as the $\{c_j^{f*} + l_j + l'_j + tr_{ij}^{a_2} + tr_{ij}^{gr_2}\}_{j\in\mathcal{J}_i}$ (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 [47] 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 [47] together with a worked example in a scenario of 5 clients and 1 helper, as depicted in Fig. 5.

Algorithm 2 and Worked Example: [8] 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 [47]. For each block $\beta \in \mathcal{B}_i$, we find client $\ell \in \beta$ such that

$$\ell := \underset{j\in\beta}{\operatorname{argmin}}\{e(\beta) + r'_j\}. \tag{24}$$

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. 5. Now, for each subblock in Γ_β, we find the client ℓ' based on (24) 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.

C. Time Complexity

While estimating the time complexity of Algorithm 2 is straightforward, this is not the case for Algorithm 1. In practice, the time complexity of Algorithm 1 depends on the number of iterations (i.e., $\tau_{\max}$) which may depend on the stopping criterion [30] and on the methods employed to solve the $w-$ and $y-$subproblems (i.e., exactly or inexactly, as discussed in footnote 7). In principle, these subproblems are NP-hard (since they are ILPs), and thus, cannot be solved exactly in polynomial time unless $P = NP$. Therefore, the time complexity of the ADMM-based method is at least that of Algorithm 2 but may or may not be polynomial. It is important to stress that even if the time complexity is not polynomial (in case of exact solutions of the subproblems), the ADMM-based method provides the framework to decompose the $\mathbb{P}$ problem into smaller subproblems. This decomposition leads to considerable speedups in running time when compared to an optimization solver that solves $\mathbb{P}$ (see Section VIII-B).

D. Discussion on Robustness to System Changes

Although the proposed approach focuses on offline optimization decisions (i.e., proactive decisions that concern a time period ahead), in this section, we discuss how our solution method can be easily adapted in cases where the clients have samples of different sizes and where changes in the system occur during training. In the former case, the ADMM-based solution method can be adapted by simply removing from the obtained schedules $\boldsymbol{x}^*$ and $\boldsymbol{z}^*$ the clients whose samples are completely processed (after a number of batch updates) and "moving" 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. In the latter case, changes in the system such as client dropouts or fluctuations in the estimated time parameters can be handled either during or at the end of the batch processing/training. Given that the orchestrator is the entity with an overall view of the system, it may re-calculate and update the decisions concerning the

[8] For simplicity, we drop here the parameters that concern the transmission delays. We can assume they are captured inside the computing delays as in [1].

makespan. Meanwhile, the helpers may need to adapt their schedules before receiving the updated decisions.

Due to the nature of the training operations (see Section III-B), the protocol selection and client-helper assignments cannot change during the training round since the activations/gradients of part-2 are stored at the helper. Therefore, only the scheduling variables x and z (for forward and backward propagation respectively) may be updated during the training round by re-solving the w-subproblem and $\mathbb{P}_b$. However, while the helpers are anticipating the new scheduling decisions, they will go through a transition phase, during which they *shift* the scheduled jobs to fill the idle slots. We note that, before that, the helpers need to wait for a short time for the originally scheduled jobs. Consequently, at the end of the batch update, the helpers can proceed with the incomplete delayed jobs; possibly causing an increase in the makespan. The shifted phase will last only for a short transition period since re-solving the w-subproblem and re-running Algorithm 2 is relatively fast, as demonstrated by the numerical evaluations in Section VIII-F. After this, the new `fwd-prop` and `bwd-prop` schedules will be adopted by the helpers, leading to a "fixed schedule" phase before the end of the training round, when a new solution to the problem $\mathbb{P}$ with the new information (following the system changes) will be computed.

VI. A (Faster) Heuristic and Preemption Cost

A. A Scalable Heuristic (Balanced-Greedy)

Since the subproblems of Algorithm 1 are ILP problems, the ADMM-based method might induce an 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 14min. While such overhead may be tolerable in scenarios of this size, especially given the resulting time savings in total training makespan compared to baseline schemes (see Section VIII), 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 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.

We propose *balanced-greedy* that can be run by the orchestrator and consists of three steps; it first decides on the training protocol per client, then on the client-helper assignments (for the clients for which the SL protocol was selected), and finally, on the scheduling at the helpers. Specifically, it starts with all decision variables being zero, i.e., $\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{y} = \boldsymbol{0}$, and:

1) First, the orchestrator decides on the training protocol of each client. In detail, the worst-case offloading delay for each client $j \in \mathcal{J}$ is estimated as follows:

$$\max_{i\in\mathcal{I}} \left\{ r_j + r'_j + l_j + l'_j + \sum_{k\in\sigma} (tr_{ij}^{a_k} + tr_{ij}^{gr_k}) + (p_{ij} + p'_{ij}) \frac{J}{I} \right\},$$

where a helper's processing time is multiplied by the fraction $\frac{J}{I}$ to represent the on-average queuing delay. Then, if client j has sufficient memory to support on-device training, and if $r_j + r'_j + l_j + l'_j + p_{(j+I)j}$ (i.e., the on-device training delay), is smaller than the expected worst-case offloading delay, the FL protocol is selected for this client. Otherwise, the SL protocol is selected. This step takes $O(JI)$ time since, for each client, it needs to iterate through all the helpers in order to estimate the worst-case offloading delay.

2) This step concerns only clients for which the SL protocol was selected in the first step. Specifically, for these clients, the assignments follow a static load balancing algorithm [48]. For this, the load of helper $i \in \mathcal{I}$ is defined as the number of assigned clients, i.e., $G_i = \sum_j y_{ij}$, and the free memory capacity of i is defined as $G'_i = m_i - \sum_{j\in\mathcal{J}} d_j y_{ij}$. Then, for each client $j \in \mathcal{J}$, balanced-greedy finds the subset of helpers Q_j with enough available memory to allocate for j (i.e., $Q_j := \{i \in \mathcal{I} : G'_i \geq d_j\}$) 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., $y_{\eta j} = 1$ and updates G_η and G'_η before proceeding to the next client. This step takes $O(JI)$ time as it iterates through all clients that have been selected to perform SL and, for each such client, it iterates through the helpers in Q_j, which is a subset of $\mathcal{I}$.

3) The scheduling decisions $\boldsymbol{x}$ and $\boldsymbol{z}$ are made at each helper in a first-come-first-served (FCFS) order [49], i.e., for the `fwd-prop` tasks, the schedule $\boldsymbol{x}$ and the completion times $\boldsymbol{c}^f$ are based on the release times $\boldsymbol{r} + \boldsymbol{tr}^{\boldsymbol{a_1}}$, and, for the `bwd-prop` tasks, $\boldsymbol{z}$ is based on $\boldsymbol{c}^f + \boldsymbol{l} + \boldsymbol{l}' + \boldsymbol{tr}^{\boldsymbol{a_2}} + \boldsymbol{tr}^{\boldsymbol{gr_2}}$. In contrast to the ADMM-based method, balanced-greedy is non-preemptive. For this final step, balanced-greedy does not require any offline computation and it can be solved with a small buffer of jobs at each helper.

As a result, balanced-greedy will run in $O(JI)$ time. We note that this time complexity is, in principle, smaller than the one of the ADMM-based method, see Section V-C.

Finally, we discuss briefly how the balanced-greedy may adapt to system changes. Specifically, when delays or client's device failures occur during the training round, the processing order at the helpers is naturally shifted by the FCFS order to adapt to these changes. Unlike the ADMM-based method (see Section V-D), there is no need to re-compute the scheduling decisions during the batch processing. At the end of the training round, protocol selection and assignment decisions may be re-computed based on the new information on the system by steps 1 and 2 of the balanced-greedy algorithm.

B. Preemption Cost

In certain systems, preemption might induce further delays or costs (e.g., due to context switch [50]) that need to be taken into account when deciding on a schedule. This feature can be readily incorporated into 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 (15) in $\mathbb{P}_f$:

$$c_j^f = \phi_j^f + \sum_{i\in\mathcal{I}}(l_j + tr_{ij}^{a_2})y_{ij} + \sum_{i\in\mathcal{I}}\sum_{t\in\mathcal{T}}\mu_i\left|x_{ijt} - x_{ij(t+1)}\right|, \forall j \in \mathcal{J},$$

where the last term captures the cost of switching tasks when a preemption occurs and when a task has just started being processed. Note that this cost is considered only for the helpers; not for clients. As was already discussed, clients in on-device training do not essentially make any scheduling decisions, and as an extension do not need to interrupt the training tasks. Similarly, we can modify the constraints in (11) 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.

VII. Energy Cost With HFSL Extension

The primary motivation of SL is the edge-friendly attribute which aims to relax the computing and memory demands on the edge. However, minimizing the training delay and the energy consumption are not considered two orthogonal objectives. For instance, an energy-aware system may suggest offloading as many layers as possible into the helpers, but, if there is not a sufficient number of helpers this can lead to an increased makespan. Hence, there is a trade-off between the energy consumption and the training makespan, when it comes to deciding whether the client will offload. In ARES [21], the trade-off between these two objectives is solved by introducing a user-defined parameter, α (where $0 \leq \alpha \leq 1$) that quantifies the importance of each objective (more details below). This implies that there is a high competitive relationship between these two objectives. However, we will show that this point is not valid as the system scales. Consequently, additional hyperparameters, which introduce a more complex interface for the user, require better management.

In what follows, we present an extension of the system model that considers the energy consumption on the client devices, and then in Section VIII we will study the relationship between makespan and energy. Let P_j^c be the power consumption for computing for client j, and P_j^t, P_j^r be the power consumption for transfer and receiving operations respectively (both measured in $Watt$). To compute the energy consumption we multiply the power consumption with the processing or communication delay of the corresponding operation. Finally, let E_j be the total energy that client j consumes during a batch update, which can be computed as:

$$\begin{aligned} E_j = &\sum_{i\in\mathcal{I}} y_{ij}(P_j^t(tr_{ij}^{a_1} + tr_{ij}^{gr_2}) + P_j^r(tr_{ij}^{a_2} + tr_{ij}^{gr_1})) \\ &+ P_j^c(r_j + r'_j + l_j + l'_j) \\ &+ P_j^c y_{(I+j)j}(p_{(I+j)j} + p'_{(I+j)j}), \quad \forall j \in \mathcal{J}. \end{aligned} \tag{25}$$

The first term of the equation computes the energy consumption due to transfer and receiving cost, hence we do not iterate through $\mathcal{I}'$, but instead only through the real helpers; there will be a communication cost only if the client offloads part-2 into a helper ($y_{ij} = 1$ for $j \leq I$). Otherwise, if $y_{ij} = 1$ for $j \geq I$ we do not want to consider the communication cost, but only the on-device computing cost, as is presented in the last term.

Now we can formulate the *joint scheduling and assignment* problem that minimizes the batch makespan of HFSL while considering the cost of the energy consumption, using the user-defined α hyper-parameter.

Problem 4 (Batch Training Makespan and Energy Consumption for HFSL).

$$\begin{aligned} \mathbb{P}_{energy}: \underset{\boldsymbol{x},\boldsymbol{z},\boldsymbol{y},\boldsymbol{\phi},\boldsymbol{c}}{\text{minimize}} \; & \alpha \max_{j\in\mathcal{J}}\{c_j\} + (1-\alpha)|S_\tau| \sum_{\mathcal{J}} E_j \\ \text{s.t.} \quad & (2)-(11), (25) \\ & \boldsymbol{x}, \boldsymbol{z} \in \{0,1\}^{|\mathcal{E}'|\times|\mathcal{T}|}, \\ & \boldsymbol{\phi}, \boldsymbol{c} \in \{0..T\}^J, \\ & \boldsymbol{y} \in \{0,1\}^{|\mathcal{E}'|}. \end{aligned}$$

Moreover, in order to produce an energy value measured in $Joules$, we need to multiply the output of (25) with the length of the slot, $|S_\tau|$. However, this will result in the objective function having a sum of values that belong to different metrics. Consequently, we compute the *proportional values* of the two variables. Specifically, each objective is translated into a percentage value, representing the portion of the maximum possible value that it achieves.[9] The maximum value for the makespan is the length of the time horizon, while the maximum energy consumption is:

$$\begin{aligned} E^{\max} = \sum_{j\in\mathcal{J}} &\max\{P_j^c(p_{(I+j)j} + p'_{(I+j)j}), \\ &\max_{i\in\mathcal{I}}\{P_j^t(tr_{ij}^{a_1} + tr_{ij}^{gr_2}) + P_j^r(tr_{ij}^{a_2} + tr_{ij}^{gr_1})\}\} \\ &+ P_j^c(r_j + r'_j + l_j + l'_j). \end{aligned} \tag{26}$$

The first term selects the maximum between the energy consumption of on-device computing (for FL) or transfer/receiving energy consumption of the slowest link (for SL). Recall that a client can either have the communication cost, in case of offloading, or the computing cost. The last term contains the computing cost for part-1 and part-3.

We note that the results of Theorem 1 could be easily extended to the problem $\mathbb{P}_{energy}$. Moreover, one could extend (with small adjustments) the proposed solution methods (ADMM-based/balanced-greedy) to solve $\mathbb{P}_{energy}$. However, due to space constraints, we focus here only on exploring the tradeoff between energy consumption and makespan, and therefore, in Section VIII-B, we showcase this tradeoff by exactly solving the problem using an optimization solver.

VIII. 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 [51] and two NN models: *(i)* ResNet101 [52], and *(ii)* VGG19 [53] for our training tasks. They are both deep convolutional NNs with 0.42 and 2.4

[9]The objective variables are divided by their maximum. Hence the two variables will have the same scale, i.e., a number between 0, and 1.

TABLE II
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

million parameters and are 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 II, where the last two were employed as helpers. We also list the collected time measurements for a batch update for ResNet101 and VGG19. These measurements show the diversity of testbed devices in terms of computing capabilities. One of the devices (RPi 3) cannot fully train any of the two models locally due to its memory limitations. Furthermore, while helpers are typically assumed to be more powerful (in terms of computation) than clients in the literature, e.g., [21], we do not restrict the evaluation of the proposed methods by this assumption. In particular, as shown in Table II, 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 [54].

Setup: In our simulations, the values of the input parameters of $\mathbb{P}$, i.e., $\boldsymbol{r}, \boldsymbol{r'}, \boldsymbol{p}, \boldsymbol{p'}, \boldsymbol{l}, \boldsymbol{l'}, \boldsymbol{tr^a}, \boldsymbol{tr^{gr}}$, are set according to the profiling data of the testbed (for the computation times) and findings on Internet connectivity in France [55, p.56] (for the transmission times). In detail, the connections between clients and helpers are of: i) low bandwidth, i.e., less than 4Mbps, ii) medium bandwidth at $4-10$ and $10-15$ Mbps, and iii) high bandwidth at $15-20$ Mbps, following the corresponding percentages given in [55, p.56]. We consider 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 II, where the selection of the type of device for each client and helper is uniformly random. Moreover, all the clients' NNs have the same cut layers: layers 2 and 30 for Resnet101, and layers 3 and 20 for VGG19.
- *Scenario 2 (high heterogeneity):* To capture higher heterogeneity, clients participate in the training by using different cut layers, and the input parameters are devised by interpolating the time measurements of the profiled devices. In detail, each input parameter is calculated by selecting a random value between the smallest and the largest profiled measurement for the corresponding task.[10] This way, we construct scenarios that capture realistic and highly-heterogeneous environments.

Moreover, in both scenarios, the memory capacities of all entities (i.e., clients and helpers) can vary from device to device but are upper-bounded by their RAM size.

[10]The available code contains detailed measurements with the profiled data and implementation for constructing the test scenarios.

A. Protocol Selection Decision (FL or SL)

First, we study the impact of the protocol selection decision on the makespan. Considering the ResNet101, we explore the benefits of the HFSL setting while changing a small subset of the system's parameters. Specifically, in Fig. 6 the number of clients and helpers remains the same, while, changing the types of (i) clients' devices and (ii) communication links between clients and helpers. In detail, we consider a set of 10 clients and 2 helpers of type VM (see Table II). Initially, the profiled data of RPi 4 are used for the clients, while the bandwidth of all network connections belongs to the fastest class, of the measurements of [55, p.56]. As we proceed, we alter the computing characteristic for a portion of the clients by slowing them down and similarly changing the bandwidth from the fastest class to the slowest one.

In this set of experiments, part-2 has fewer layers when compared to [1] since, otherwise, the on-device processing for part-2 is longer than the one of the helpers, and would not allow us to observe the benefits of the HFSL setting. The first row of Table III shows the relative gain in makespan in the HFSL setting for the corresponding experiments from Fig. 6.

At first glance, we notice that when there are slow clients but good connections, having on-device training will not alter the makespan, except in the case where there are no slow connections and no slow devices, i.e., $(0\%, 0\%)$, in which there is a slight acceleration of 4.8%. However, as more communication links get slower it is preferable to use on-device training even if the client's device is slower.

Observation 1. HFSL can decrease the makespan by up to 59% in the presence of slow network connections and devices that can process part-2 fast.

This happens because model part-2 is small and the processing time on the devices is not significantly different from the processing time on the helper (see also Section VIII-E). As a result, the communication delay may be larger than the on-device computation delay. However, we would like to highlight that these split cut do not fully benefit from the helper's availability. When having a larger part-2 it is more beneficial to offload, because, the on-device processing time gets larger, while the offloading delay gets smaller. This is shown in the second row in Table III, where the acceleration of HFSL is smaller compared to the previous case because fewer clients perform FL. Whereas, when there are more fast clients with slow communication links we can see the effects of FL, where the makespan is decreased up to 5.6%.

In general, this indicative exploration example demonstrates the importance of having a balance between the SL and FL, which is achieved through HFSL.

B. Comparison With the Optimal Solution

We proved that $\mathbb{P}$ is NP-hard, which led us to propose two scalable solution methods. Table IV shows the suboptimality and speedup achieved by the ADMM-based method when compared to Gurobi [42], one of the fastest ILP solvers [56], that optimally solves $\mathbb{P}$. The table shows the effectiveness of the ADMM-based method since it finds the optimal solution in most scenarios.

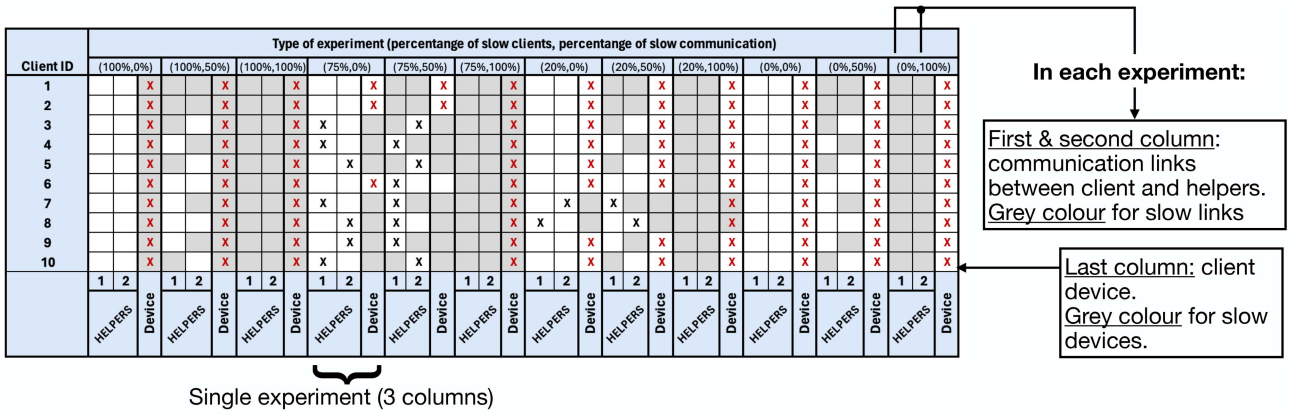

Fig. 6. Training protocol selection decisions in a scenario of 10 clients and 2 helpers. For each experiment conducted, we alter the type of clients' devices and the communication throughput between clients and helpers. The figure shows how the clients are allocated. If the client offloads its model part into one of the helper this is indicated with a black "X" at the corresponding cell. Whereas, when the client does not offload there is a red 'X' at the third column of the corresponding experiment.

TABLE III
RELATIVE GAIN IN MAKESPAN IN THE HFSL SETTING VERSUS CLIENTS TRAINING ONLY THROUGH SL (AS IN [1])

% of slow clients	100			75			20			0		
% of slow communication	0	50	100	0	50	100	0	50	100	0	50	100
small partition (Fig. 6)	0%	41.1%	42.3%	0%	20%	42%	0%	22.1%	41.1%	4.8%	57.8%	59%
large partition	0%	0%	0%	0%	1.17%	0%	0%	1.17%	0%	0%	1.17%	5.6%

TABLE IV
SUBOPTIMALITY AND SPEEDUP ACHIEVED BY THE ADMM-BASED METHOD COMPARED TO AN ILP SOLVER FOR HFSL

		J	I	T'	suboptimality (%)	speedup ($\times$)
Scenario1	Resnet101	10	2	266	0	14.51
			5	266	9.4	14
		15	5	291	15.9	11.3
	VGG19	10	2	125	0	12.8
			5	125	0	13
		15	5	172	0	15.7
Scenario2	Resnet101	10	2	264	2.7	4
			5	237	0	4
		15	5	271	0	2.6
	VGG19	10	2	268	0	6.1
			5	261	0	10.5
		15	5	267	1.8	5.3

There are some corner cases (e.g., 9.4%, 15.9%), but even then there is a significant speedup compared to the solver, i.e., $\times 14$, $\times 11.3$, respectively. We note that the numerical evaluations in Table IV were performed in small instances (up to 15 clients and 5 helpers) that ILP solvers can handle. Moreover, we observe that the lowest suboptimality gap is achieved for VGG19, which comes from the choice of cut layers (see above). In particular, Fig. 7 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 operations. Such asymmetries that can occur in SL further corroborate our approach towards jointly optimized assignments and scheduling.

Observation 2. The proposed ADMM-based method finds the optimal solution for $\mathbb{P}$ in most problem instances and achieves up to $15.7\times$ speedup compared to an ILP solver.

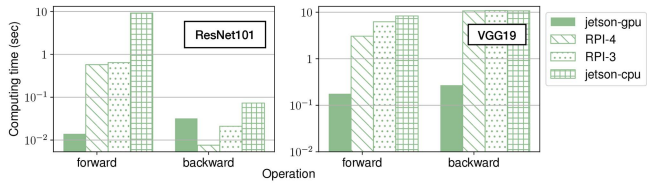

Fig. 7. Profiled computing time (ms.) of part-1 for each device.

When compared to the case where all clients train through SL (as in [1]), the computing speedup of the ADMM-based algorithm for the HFSL is not as large. This is due to two main reasons: (i) for the same input parameters, the time horizon T may be larger than in [1], because, in HFSL, we use (1), which also accounts for the on-device delay. Additionally, (ii) the space of combinatorial choices becomes larger; each client may be assigned to any of the $I+1$ helpers (the additional one is due to the FL selection), whereas in [1], there are I options. Nevertheless, we achieve a makespan closer to the optimal value when compared to [1].

C. Impact of the Time Slot's Length

In Section 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 $|S_t| = 50$.

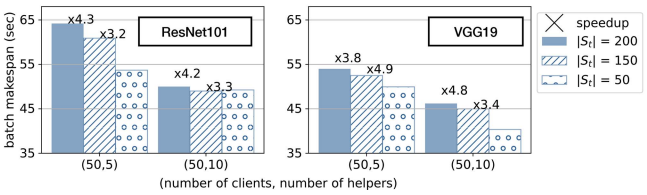

Fig. 8. Batch makespan obtained by the ADMM-based method for time slot length $|S_t|$ equal to 1500ms, 1000ms, and 500ms, in Scenario 1. Speedup is relative to the case $|S_t| = 500$.

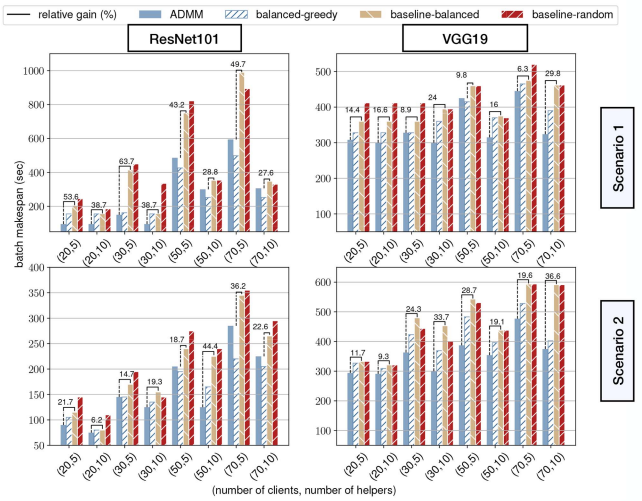

Fig. 9. Comparison of the proposed methods with the baselines for HFSL.

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 3 rd slot. Therefore, in such cases, the time slot's length may affect the accuracy of the obtained schedule.

Observation 3. As the length of the considered time slots S_t increases, the obtained makespan increases, while the execution time decreases. This shows an algorithmic tradeoff between the solution's precision and size of the solution space.

Fig. 8 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 $|S_t| = 500$, which has the highest overhead. In principle, the makespan is higher as the $|S_t|$ increases. 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 $\times 5$.

D. Comparison With Two Baselines

Next, in Fig. 9, we compare the proposed methods (ADMM-based and balanced-greedy) between them, and with two baseline schemes. The first one is the *baseline-random* that first decides on the training protocol selection and client-helper assignments in a random way (subject, of course, to memory constraints), and then schedules the tasks in an FCFS order. This baseline could be seen as a naive real-time implementation of SL without proactive decisions on assignments or scheduling. The second one, *baseline-balanced*, is less naive as it considers the first step from balanced greedy (Section VI-A) for training protocol selection. Then, if SL protocol is selected for a client, this client will be randomly assigned to a helper. We note that a straightforward comparison with related work (e.g., [12], [14]) is not possible since these works consider only the client-protocol selection (there is only one helper).

We focus first on comparing the two proposed methods. First, for medium-sized scenarios (1 & 2), i.e., up to 50 clients, the ADMM-based method outperforms the baseline-random up to 71%, and up to 63% the baseline-balanced. While the balanced-greedy achieves an improvement up to 60% and 63%, respectively. Even though the balanced-greedy and the baseline-balanced are two very close approaches, we notice that controlling the helpers' load plays an important role even for a medium-sized number of clients. In fact, as the number of clients grows and the queuing delays risk increases even more, balancing the helpers' loads provides a solution closer or even better to the ADMM-based method. Even in scenario 2, where scheduling and balancing become more crucial, the two approaches tend to provide close outputs, as the system scales. Hence, given the discussion on the overhead of the two methods in Section VI-A, balanced-greedy should be preferred for very large scenarios (e.g., ≥ 100 clients in our case) to avoid excess overhead.

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 ResNet101. This reveals a dependency on the NN, since NNs may differ on the cut layers, the size of the processing tasks [10], etc. We further explore this dependency and its implications on the makespan in the following sections.

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 schemes, we observe that our proposed strategy consistently outperforms the two baselines, achieving a shorter makespan. In detail, the baseline schemes decide 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 SL.

Observation 4. The numerical evaluations allow us to build a solution strategy based on the scenario's characteristics that achieves a shorter makespan than the fair-baseline by up to 63.7%, and up to 71.6% for the baseline-random.

E. Sensitivity Analysis

In Fig. 10, 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.

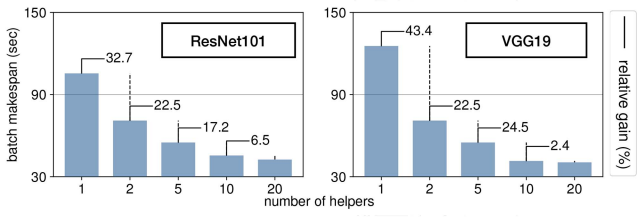

Fig. 10. Batch makespan obtained by the balanced-greedy method in Scenario 1 for $J = 100$ clients and varying I.

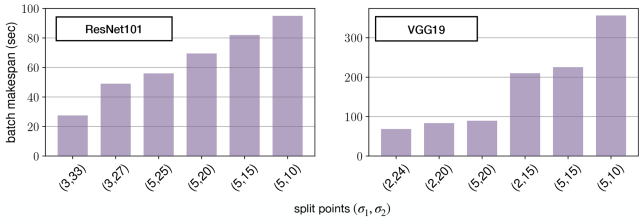

Fig. 11. Batch makespan obtained using balanced-greedy in Scenario 1 for $J = 100$, $I = 20$ and varying split points.

Observation 5. In a scenario of 100 clients and 1 helper, adding one more helper can dramatically decrease the batch makespan by up to 43.4%.

Whereas, in the presence of 10 helpers, the relative gains of adding more helpers are decreasing. Such observations provide useful insights for real-life implementation of HFSL. Finally, extending the analysis of Table III, Fig. 11 compares the output of balanced-greedy for different definitions of σ_1 and σ_2; altering the split policy.

Observation 6. Offloading larger model parts into the helpers can reduce the batch makespan by up to 71% in ResNet101 and 80% for VGG19.

As part-2 gets larger, the obtained makespan gets shorter. In practice, this fluctuation is shaped by the clients who train through SL. Because when part-2 becomes larger, the gain from using the helpers becomes even more considerable. This output indicates the importance of SL for systems that involve small IoT devices like the one we have in our testbed.

F. Preliminary Results on Robustness to System Changes

Following the discussions in Sections V-D and VI-A concerning the proposed solutions' robustness and adaptation to system changes, we explore the impact of the suggested methods on the makespan. Since this in not the main focus of our work, we present here some preliminary results that showcase that the proposed methods can be easily adapted to handle system changes.

Considering the Resnet101 in a setting of 5 helpers ($I = 5$) and of two different client scales (i.e., $J = 30$ and $J = 50$ clients), Table VI presents the changes in the makespan when two different types of system changes occur. First, we focus on the first (i.e., top-half) part of Table VI which studies the case where the release times of the processing tasks are delayed (i.e., devices or communication links become slower). In particular, in this tested scenario, the release times for almost 20% of the clients are increased by 100%. In this case, we expect the

TABLE V
NUMERICAL VALUES FOR POWER CONSUMPTION ASSIGNED TO THE DEVICES USED BUT THE CLIENTS

Device	Variable	Value ($Watt$)	Source
RPI 4	P^c	$6-7$	See footnote 11
	P^t	2.5	PowerPi [57]
	P^r	1.5	PowerPi [57]
NVIDIA Jetson Nano	P^c	7.2	ARES [21]
	P^t	2.3	ARES [21]
	P^r	2.2	ARES [21]

TABLE VI
RELATIVE CHANGES IN MAKESPAN (%) WITH RESPECT TO THE ORIGINAL OPTIMIZATION DECISIONS DUE TO TWO DIFFERENT SYSTEM CHANGES (DELAYS IN PROFILED TIMES AND CLIENT'S DEVICE FAILURES)

Type of change	J	Algorithm	shifted	fixed sched.	new sol.
Delays	30	ADMM	↑ 16%	↑ 8%	0%
		BG	↑ 11%	—	↑ 11%
	50	ADMM	↑ 29%	↑ 7.4%	↑ 7.4%
		BG	↑ 15.5%	—	↑ 15.5%
Failures	30	ADMM	0%	↓ 24%	↓ 24%
		BG	↓ 29%	—	↓ 29%
	50	ADMM	↓ 3.4%	↓ 27.5%	↓ 46%
		BG	↓ 17%	—	↓ 33%

Increases in makespan are denoted by ↑ and decreases by ↓. For these experiments, the ResNet101 model was trained in a setting of $I = 5$ helpers, and two different cases of J (number of clients) were considered. BG refers to the balanced-greedy.

makespan to increase or remain the same with respect to the original one (before any system change) and the goal is to minimize this increase. In the ADMM-based method, as expected, the shifted phase is the one that has the greatest increase in the makespan. However, this phase lasts only one batch update as the computation of `fwd-prop` and `bwd-prop` schedules lasts less than 3 seconds; the corresponding delay for a batch update is more than 100 sec. as shown in Fig. 9. Re-calculating the scheduling decisions (column "fixed schedule") can decrease considerably the increase in the makespan. Finally, in the new training round, i.e., after re-running the ADMM-based method (column "new solution"), the resulting makespan turns out to be the same as the original one or to have a small increase of 7.4%. In the latter case, this is due to changes in the protocol selection or re-assignment of clients to different helpers.

For the balanced-greedy, the FCFS scheduling order manages to keep the relative increase in makespan in the range of $11-15\%$. We observe that there is no change between the shifted schedule and the makespan of the next round (i.e., new solution). This implies that, in the new solution, the decisions on protocol selection and assignments remain the same as before. This happens because, in balanced-greedy, only the load (i.e., number of clients) is considered for the protocol selection and assignment decisions, and not the release times. However, in the case of device failures that we study next, the balanced-greedy will update the assignments in order to balance the load at the helpers. Finally, we note that there is no value in the column of fixed schedule for balanced-greedy since the scheduling decisions are by design online, as discussed in Section VI-A. Thus, there is no need to re-calculate the scheduling decisions as in the ADMM-based method.

Focusing now on the case of client device failures (i.e., bottom-half of Table VI), we consider that 15% and 20% of the client devices become unavailable during training, respectively for $J = 30$ and $J = 50$. We note that, in principle,

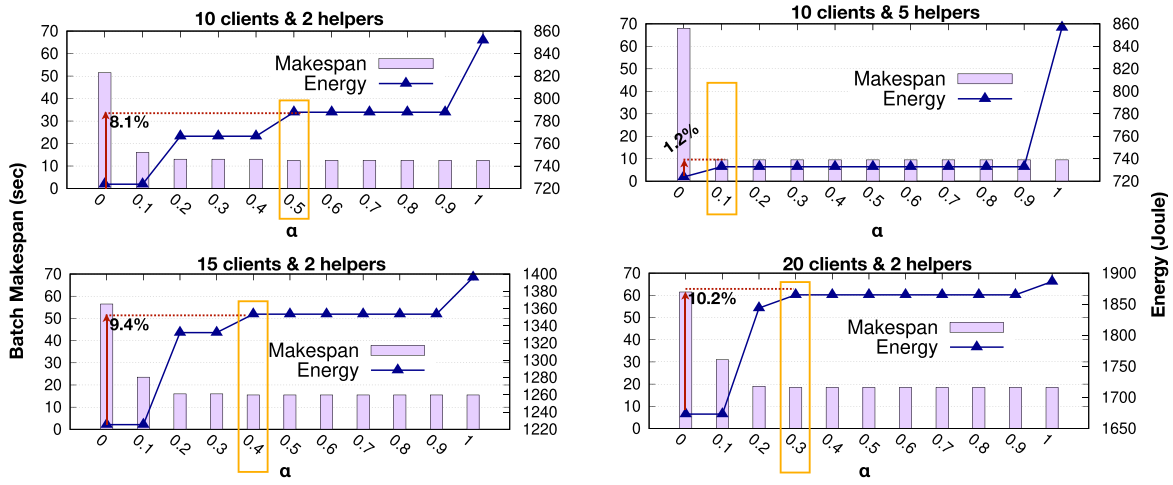

Fig. 12. Total clients' energy versus Batch makespan for different values of α. The box shows the smallest α in which we obtain the best makespan, while the arrows show the energy increase compared to $\alpha = 0$ (i.e., most energy-efficient case).

the makespan is expected to decrease since the processing load at the helpers will be reduced. Therefore, the goal here would be to maximize this decrease (as opposed to minimizing the increase in the previous case), i.e., find new decisions that will achieve the shortest makespan. We see that, for the ADMM-based method, the most impaired case is the shifted phase, which has a makespan (almost) equal to the prior one. However, the relative change in makespan is dramatically improved in the next training round (i.e., new solution) when compared to the shifted phase. Moreover, we observe that, in the case of $J = 30$, this improvement is already achieved in the phase of fixed schedule, i.e., without need of calculating a new solution. Further, for $J = 50$, where a larger number of clients become unavailable, the clients can be redistributed among the helpers, reducing up to 50% of their original load. A similar behavior is observed in the case of balanced-greedy, too.

Of course, changes usually happen gradually in a system. For simplicity, the experiments shown in Table VI consider a large number of changes occurring at the same time, thus, leading to a significant change in the makespan. However, the presented findings demonstrate how these changes may be efficiently handled to minimize the makespan.

G. Exploring Energy Consumption

Table V presents the input values used for the parameters corresponding to the power consumption when solving $\mathbb{P}_{energy}$. For the computing power consumption of the RPis, we use the benchmarks from the RPi Drupal project[11] while for the communication operations (transferring and receiving), we use the profiled data from the power cost model of PowerPi [57], which is also used in related work, e.g., [58]. Moreover, for the Jetson device, we use the measurements from ARES [21] which can be verified from additional sources.[12] We notice that in both devices computing operations require more power than communication operations. Hence, energy-wise, it's preferable that IoT devices offload their model parts to helpers, which aligns with the initial motivation of SL (as we discussed in Section I).

Fig. 12 shows the makespan and the total energy consumption on the client devices as α changes for different numbers of clients and helpers. When having $\alpha = 0$, the preferred protocol for the clients is SL (i.e., offload part-2 to helper) since the communication operations cost less than the computing ones. Moreover, each client will offload to the helper with the fastest communication link, neglecting the helper's computing delay or the queuing delay due to increased load. This may result in the over-utilization of a helper, while the other helpers remain idle. As an extension, the makespan becomes larger for smaller values of α. Whereas, when $\alpha = 1$, the energy consumption increases up to 15.5% (on average 13.5%). But, it achieves a makespan on average $\times 4.5$ faster than $\alpha = 0$. This difference is mainly because of clients for whom the FL protocol is selected. However, one of the objectives is multiplied by zero when having these extreme values, and consequently, it is neglected when solving the problem. This can be seen by the fact that when $\alpha = 0$ the makespan becomes extremely large.[13] But, when $\alpha = 0.1$ we get the same energy consumption with a better makespan. Similarly, we can use $\alpha = 0.9$ instead of $\alpha = 1$, to obtain the best makespan, with less energy consumption. As a result, we can define two modes regarding the trade-off between energy and makespan.

Observation 7. ***Energy-efficient mode:*** When $\alpha = 0.1$ we (almost) always obtain the optimal energy consumption, with an improved makespan compared to $\alpha = 0$.

Observation 8. ***Delay-efficient mode:*** When $\alpha = 0.9$ we obtain the optimal makespan, with a smaller energy consumption compared to $\alpha = 1$.

Note that, in the *energy-efficient* mode, the makespan needs to be controlled too ($\alpha \neq 0$) since, by increasing the per-batch delay, the training process will take longer in total. As a result,

[11] https://www.pidramble.com/wiki/benchmarks/power-consumption

[12] https://www.macnica.co.jp/en/business/semiconductor/manufacturers/nvidia/products/141900/

[13] In fact, the makespan becomes equal to the length of the time horizon.

client devices need to remain available for a longer time, which can affect the energy consumption and the device's battery autonomy. On the other hand, for the intermediate values of α, we do not notice significant fluctuations. This is due to two factors. The first reason is regarding energy, when α falls between $0.1 < \alpha < 0.9$ the allocation decision is mainly relied on the helper selection, not on offloading, like in $\alpha = 0$ or $\alpha = 1$. This decision is affected by the power consumption due to the communication operations, which are smaller than the computing ones. This results in a small fluctuation in energy for those α values. The second reason concerns the makespan. As is shown in Fig. 12, we obtain the best possible makespan even from a small α value (shown in orange boxes). This is due to the presence of the *max* function in the objective function. As the slowest client determines the makespan, changes for other clients will not affect the makespan as soon as the best allocation and scheduling have been found. Therefore, all $0.1 < \alpha < 0.9$ values can be represented by the *balanced mode*, as observed below.

Observation 9. ***Balanced mode:*** When $\alpha = 0.5$ we obtain the optimal makespan, while the energy consumption is larger than $\alpha = 0$, but smaller than $\alpha = 1$.

To conclude, the presented performance evaluation for all α values allows us to define three distinct system modes (energy-efficient, delay-efficient, balanced).

IX. Conclusions and Future Work

In this work, we formulated the joint problem of training protocol selection, client-helper assignments, and scheduling for HFSL. We analyzed it 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 chosen methods 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 schemes by achieving a shorter makespan by up to 63% and 71%. Finally, we explore and analyze the tradeoff between makespan and energy consumption. This led to the introduction of three user modes which could provide the flexibility to a system administrator to select according to their preferences and goals. An interesting direction for future work would be to study the formulated problem from the perspective of the online optimization framework that makes decisions as clients' processing tasks arrive or as the system's state (e.g., entities' availability, connectivity, etc.) changes.

References

[1] J. Tirana, D. Tsigkari, G. Iosifidis, and D. Chatzopoulos, "Workflow optimization for parallel split learning," in *Proc. IEEE Conf. Comput. Commun.*, 2024, pp. 1331–1340.

[2] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Proc. 20th Int. Conf. Artif. Intell. Statist.*, 2017, pp. 1273–1282.

[3] W. Y. B. Lim et al., "Federated learning in mobile edge networks: A comprehensive survey," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 3, pp. 2031–2063, 3rd Quarter, 2020.

[4] L. L. Pilla, "Optimal task assignment for heterogeneous federated learning devices," in *Proc. 2021 IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 661–670.

[5] Y. Jiang et al., "Model pruning enables efficient federated learning on edge devices," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 12, pp. 10374–10386, Dec. 2023.

[6] M. R. Sprague et al., "Asynchronous federated learning for geospatial applications," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, Springer, 2018, pp. 21–28.

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

[8] C. Thapa, P. C. M. Arachchige, S. Camtepe, and L. Sun, "SplitFed: When federated learning meets split learning," in *Proc. AAAI Conf. Artif. Intell.*, 2022, pp. 8485–8493.

[9] J. Jeon and J. Kim, "Privacy-sensitive parallel split learning," in *Proc. Int. Conf. Inf. Netw.*, 2020, pp. 7–9.

[10] Z. Zhang, A. Pinto, V. Turina, F. Esposito, and I. Matta, "Privacy and efficiency of communications in federated split learning," *IEEE Trans. Big Data*, vol. 9, no. 5, pp. 1380–1391, Oct. 2023.

[11] K. Palanisamy, V. Khimani, M. H. Moti, and D. Chatzopoulos, "SplitEasy: A practical approach for training ML models on mobile devices," in *Proc. 22nd Int. Workshop Mobile Comput. Syst. Appl.*, 2021, pp. 37–43.

[12] X. Liu, Y. Deng, and T. Mahmoodi, "Energy efficient user scheduling for hybrid split and federated learning in wireless UAV networks," in *Proc. IEEE Int. Conf. Commun.*, 2022, pp. 1–6.

[13] X. Liu, Y. Deng, and T. Mahmoodi, "Wireless distributed learning: A new hybrid split and federated learning approach," *IEEE Trans. Wireless Commun.*, vol. 22, no. 4, pp. 2650–2665, Apr. 2023.

[14] Z. Wang, H. Xu, Y. Xu, Z. Jiang, and J. Liu, "CoopFL: Accelerating federated learning with DNN partitioning and offloading in heterogeneous edge computing," *Comput. Netw.*, vol. 220, 2023, Art. no. 109490.

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

[16] A. Rodio, F. Faticanti, O. Marfoq, G. Neglia, and E. Leonardi, "Federated learning under heterogeneous and correlated client availability," in *Proc. IEEE Conf. Comput. Commun.*, 2023, pp. 1–10.

[17] C. Chen et al., "GIFT: Toward accurate and efficient federated learning with gradient-instructed frequency tuning," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 4, pp. 902–914, Apr. 2023.

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

[19] J. Tirana, S. Lalis, and D. Chatzopoulos, "MP-SL: Multihop parallel split learning," 2024, *arXiv:2402.00208*.

[20] M. Kim, A. DeRieux, and W. Saad, "A bargaining game for personalized, energy efficient split learning over wireless networks," in *Proc. Wireless Commun. Netw. Conf.*, 2023, pp. 1–6.

[21] E. Samikwa, A. Di Maio, and T. Braun, "ARES: Adaptive resource-aware split learning for Internet of Things," *Comput. Netw.*, vol. 218, 2022, Art. no. 109380.

[22] W. Wu et al., "Split learning over wireless networks: Parallel design and resource management," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 4, pp. 1051–1066, Apr. 2023.

[23] Y. Mu and C. Shen, "Communication and storage efficient federated split learning," in *Proc. IEEE Int. Conf. Commun.*, 2023, pp. 2976–2981.

[24] E. L. Lawler, J. K. Lenstra, A. H. R. Kan, and D. B. Shmoys, "Sequencing and scheduling: Algorithms and complexity," in *Handbooks in Operations Research and Management Science*. Amsterdam, The Netherlands: Elsevier, 1993.

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

[26] B. Chen, C. N. Potts, and G. J. Woeginger, "A review of machine scheduling: Complexity, algorithms and approximability," in *Handbook of Combinatorial Optimization*, vol. 1–3. Berlin, Germany: Springer, 1998, pp. 1493–1641.

[27] A. M. Geoffrion, "Generalized benders decomposition," *J. Optim. Theory Appl.*, vol. 10, pp. 237–260, 1972.

[28] J. Lou, Z. Tang, W. Jia, W. Zhao, and J. Li, "Startup-aware dependent task scheduling with bandwidth constraints in edge computing," *IEEE Trans. Mobile Comput.*, vol. 23, no. 2, pp. 1586–1600, Feb. 2024.

[29] H. Wang et al., "Low-complexity and efficient dependent subtask offloading strategy in IoT integrated with multi-access edge computing," *IEEE Trans. Netw. Service Manag.*, vol. 21, no. 1, pp. 621–636, Feb. 2024.
[30] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Found. Trends Mach. Learn.*, vol. 3, pp. 1–22, 2011.
[31] Q. Chen, F. R. Yu, T. Huang, R. Xie, J. Liu, and Y. Liu, "Joint resource allocation for software-defined networking, caching, and computing," *IEEE/ACM Trans. Netw.*, vol. 26, no. 1, pp. 274–287, Feb. 2018.
[32] D. Tsigkari, G. Iosifidis, and T. Spyropoulos, "Quid pro quo in streaming services: Algorithms for cooperative recommendations," *IEEE Trans. Mobile Comput.*, vol. 23, no. 2, pp. 1753–1768, Feb. 2024.
[33] S. Diamond, R. Takapoui, and S. Boyd, "A general system for heuristic solution of convex problems over nonconvex sets," 2016, *arXiv:1601.07277*.
[34] A. Themelis and P. Patrinos, "Douglas–Rachford splitting and ADMM for nonconvex optimization: Tight convergence results," *SIAM J. Optim.*, vol. 30, no. 1, pp. 149–181, 2020.
[35] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with ADMM," in *Proc. AAAI Conf. Artif. Intell.*, 2018, pp. 3466–3473.
[36] S. Zhang, J. Liu, H. Guo, M. Qi, and N. Kato, "Envisioning device-to-device communications in 6G," *IEEE Netw.*, vol. 34, no. 3, pp. 86–91, May/Jun. 2020.
[37] A. S. Schulz and M. Skutella, "Scheduling unrelated machines by randomized rounding," *SIAM J. Discrete Math.*, vol. 15, no. 4, pp. 450–469, 2002.
[38] 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. IEEE Conf. Comput. Commun.*, 2019, pp. 1387–1395.
[39] 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. IEEE Conf. Comput. Commun.*, 2022, pp. 780–789.
[40] 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. IEEE Conf. Comput. Commun.*, 2019, pp. 2287–2295.
[41] S. P. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge, U.K.: Cambridge Univ. Press, 2004.
[42] Gurobi Optimization, LLC, "Gurobi optimizer reference manual," 2023. [Online]. Available: https://www.gurobi.com
[43] J. K. Lenstra, A. R. Kan, and P. Brucker, "Complexity of machine scheduling problems," *Ann. Discrete Math.*, vol. 1, pp. 343–362, 1977.
[44] T. Gonzalez, E. L. Lawler, and S. Sahni, "Optimal preemptive scheduling of two unrelated processors," *ORSA J. Comput.*, vol. 2, no. 3, pp. 219–224, 1990.
[45] W. Horn, "Some simple scheduling algorithms," *Nav. Res. Logistics Quart.*, vol. 21, no. 1, pp. 177–185, 1974.
[46] J. Eckstein and D. P. Bertsekas, "On the Douglas—Rachford splitting method and the proximal point algorithm for maximal monotone operators," *Math. Program.*, vol. 55, no. 1, pp. 293–318, 1992.
[47] 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 Res.*, vol. 31, no. 2, pp. 381–386, 1983.
[48] 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 *Proc. Int. Conf. Trends Electron. Inform.*, 2017, pp. 100–107.
[49] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge, U.K.: Cambridge Univ. Press, 2013.
[50] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. Workshop Exp. Comput. Sci.*, 2007, pp. 1182–1189.
[51] A. Krizhevsky, "Learning multiple layers of features from tiny images," in *Handbook of Systemic Autoimmune Diseases*, 2009.
[52] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[53] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[54] L. Bai, W. Ji, Q. Li, X. Yao, W. Xin, and W. Zhu, "DNNAbacus: Toward accurate computational cost prediction for deep neural networks," 2022, *arXiv:2205.12095*.
[55] D. Belson, "State of the internet Q4 report," 2016. [Online]. Available: https://www.akamai.com/site/en/documents/state-of-the-internet/q4-2016-state-of-the-internet-connectivity-report.pdf
[56] R. Anand, D. Aggarwal, and V. Kumar, "A comparative analysis of optimization solvers," *J. Statist. Manage. Syst.*, vol. 20, no. 4, pp. 623–635, 2017.
[57] F. Kaup, P. Gottschling, and D. Hausheer, "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi," in *Proc. 39th Annu. IEEE Conf. Local Comput. Netw.*, 2014, pp. 236–243.
[58] A. C. Valera, N. Clayton, W. K. Seah, and T. Zheng, "Optimal transmission scheduling in data-intensive audio sensor networks," in *Proc. IEEE Glob. Commun. Conf.*, 2023, pp. 7049–7054.

Joana Tirana received the diploma (MSc-equivalent) degree in electrical and computer engineering from the University of Thessaly in Volos, Greece, in 2021. She is currently working toward the PhD degree with the University College Dublin, since January 2022. Her research area is Decentralized AI, focusing on protocols for distributed and collaborative learning like federated learning and finding ways to optimize these techniques by using AI task offloading and split learning.

Dimitra Tsigkari received the degree in mathematics from the Aristotle University of Thessaloniki, Greece, the master's degree in mathematics and applications from Pierre-et-Marie-Curie University, France, and the PhD degree in computer science, telecommunications, and electronics from EURECOM and Sorbonne University, France. She was a postdoctoral researcher with TU Delft, Netherlands. She is currently a research scientist with Telefonica Research, Spain. Her research interests lie in the area of network optimization with a focus on edge/cloud computing and distributed learning.

George Iosifidis received the diploma degree in electronics and telecommunications engineering from Greek Air Force Academy, Athens, in 2000, and the PhD degree from the University of Thessaly, in 2012. He was an assistant professor with Trinity College Dublin from 2016 to 2020. He is currently an associate professor with the Delft University of Technology. His research interests lie in the broad area of network optimization and economics.

Dimitris Chatzopoulos received the diploma and MSc degrees in computer engineering and communications from the University of Thessaly, Greece, and the PhD degree in computer science and engineering from the Hong Kong University of Science and Technology. He is an assistant professor with the School of Computer Science, University College Dublin. His research interests include privacy-preserving and AI-enabled decentralized applications for mobile and distributed systems.