# Fast Simulation of Federated and Decentralized Learning Algorithms

**Scheduling Algorithms for Minimisation of Variability in Federated Learning Simulations**

**Todor Slavov**[1]

**Supervisor(s): Jérémie Decouchant**[1]**, Bart Cox**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Todor Slavov
Final project course: CSE3000 Research Project
Thesis committee: Jérémie Decouchant, Bart Cox, Qing Wang

## Abstract

Federated Learning (FL) systems often suffer from high variability in the final model due to inconsistent training across distributed clients. This paper identifies the problem of high variance in models trained through FL and proposes a novel approach to mitigate this issue through scheduling simulations subject to precedence constraints. By effectively scheduling the execution of client tasks and parameter server updates, we aim to reduce the variance in the final aggregated model. Through a series of experiments, we demonstrate that our proposed scheduling method significantly reduces model variance, while not impacting the time of simulation drastically. Additionally, we propose 2 algorithms to solve the problem of scheduling under precedence constraints - Ant Colony Optimisation, and an Evolutionary Algorithm - to minimize the makespan of simulations.

## 1 Introduction

As machine learning (ML) solutions keep gaining popularity they are getting applied to more and more complex problems and require ever-increasing volumes of data for training. In current times user devices carry large amounts of data that can be highly valuable for learning. With an outlook on sensitivity and privacy concerns the paradigm of federated learning (FL) has been gaining traction in recent years. FL systems have the advantage of utilising those large quantities of data while preserving user privacy and capturing the heterogeneity of data. To do that a parameter server distributes a global model which is trained by user/client devices on their local data. Those client nodes send their updates to the parameter server which aggregates them to a local model, repeating this in an iterative manner.

Deploying FL systems presents many technical challenges, including coordination of devices, model synchronization, and navigating the variability in device capabilities and network conditions. To surmount these obstacles, researchers often turn to simulations, as a way to assess the efficacy of FL algorithms. In an FL simulation, one or potentially multiple machines (usually a smaller number than the number of clients) do the work of a parameter server and client nodes. For instance, one machine might switch between being a parameter server to aggregate updates and being a client node to learn from data. Alternatively, one machine might be assigned a parameter server and several others might be client nodes.

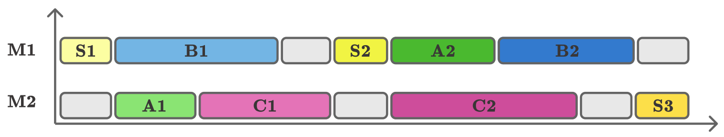


Figure 1: Simulation of the synchronous FL example from Fig 6b on two machines. Grey boxes represent idle times and coloured boxes represent learning times. Here two machines take turns executing the aggregation jobs of the Server and the learning jobs of Alice, Bob, and Charlie. M1 and M2 reffer to Machine 1 and Machine 2 respectively. Each task named, for example, S2 reffers to the 2$^{nd}$ task of the (S)erver.

Simulations allow for easier communication, management and synchronisation, letting the researcher access a finished model faster and easier than through the deployment of the actual system. Still, two main challenges must be carefully examined when designing a simulation for an FL system.

- The final trained models can vary greatly depending on the order of client updates, because of the non-IID distribution of data [1].
- FL simulations can take a long time due to the learning process of clients and the computational constraints of the system used for simulation [2].

As we will see in the next section, both of these challenges can be addressed by studying the optimisation problem of scheduling under precedence constraints.

The remainder of this paper is organized as follows: Section 2 introduces the notation and provides a definition and motivation for the scheduling problem we are addressing. In Section 3, we explain the methodology used to solve this problem. Section 4 reviews relevant literature on FL simulations. Section 5 elaborates on the approach taken to answer the research questions. Section 6 highlights the contributions made by this paper, and Section 7 presents the results we obtained. In Section 8, we discuss the ethical considerations and responsible research practices related to our work. In Section 9, we open a discussion and outline potential directions for future research. Finally, Section 10 summarizes the conclusions we have drawn.

## 2 Background

This section presents the pieces of the puzzle that need to be understood to grasp the problem at hand. Subsection 2.1 gives background on FL deployments. Subsection 2.2 introduces the FedAsync algorithm and Sbsection 2.3 explains FL simulations and their differences and similarities with FL deployments. Subsection 2.4 gives background on the theory and applications of scheduling problems.

### 2.1 Deployment

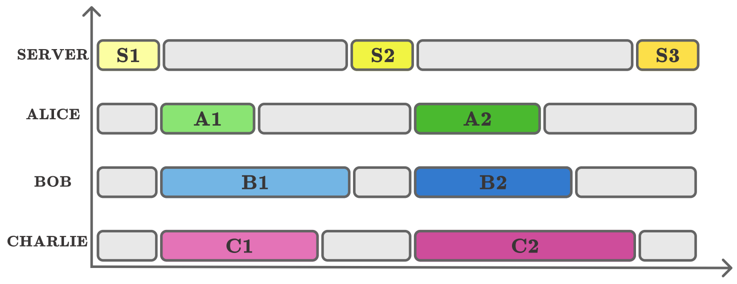
Two main flavours of Federated Learning deployments exist: synchronous and asynchronous FL.

#### Synchronous Federated Learning

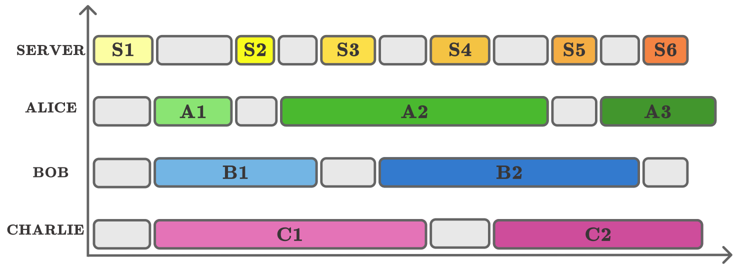
In synchronous FL [3], the parameter server delegates the global model to a subset of client nodes to train locally . Once the clients have completed their local training, they send their updates back to the parameter server, which waits to receive updates from all selected clients before aggregating them into a new global model. This new global model is then redistributed to the clients for the next round of training. One significant advantage of synchronous FL is that it ensures consistency across the global model, as all updates are based on the same version of the model. This can lead to more stable convergence and potentially better final model performance. However, the downside is that the overall training time can be significantly increased due to the necessity of waiting for the slowest client (often referred to as the ”straggler problem”). This can be particularly problematic in environments where clients have highly variable computation and communication capabilities.

#### Asynchronous Federated Learning

In asynchronous FL [4, 5, 6], the parameter server updates the global model immediately upon receiving a local update from any client . The newly updated global model is then sent back to the client that just finished its local update, allowing for continuous and dynamic model improvement without the need for synchronization points. This approach can significantly reduce idle times and accelerate the learning process, making it more suitable for environments with heterogeneous client performance and unreliable communication networks. However, asynchronous FL comes with its own set of challenges. The primary concern is the potential for stale updates, where some clients may be training on outdated versions of the global model. This can lead to less stable convergence and may require additional mechanisms, such as staleness-aware update schemes, to mitigate the impact of outdated information. Despite these challenges, asynchronous FL is often more flexible and can better handle real-world scenarios where uniformity in client capabilities cannot be assumed.



(a) Synchronous Federated Learning



(b) Asynchronous Federated Learning

Figure 2: The coloured bars represent working time and the grey bars represent idle time. Different shades of the same colour indicate that different tasks belong to the same client or server. The notation C2 means that we are referring to the 2nd task of (C)harlie.

## 2.2 FedAsync

FedAsync [4] is an asynchronous Federated Learning algorithm designed to enhance the efficiency and performance of federated learning systems. Unlike synchronous federated learning, where the global model is updated only after receiving updates from all clients, FedAsync updates the global model immediately upon receiving a local update from any client 1. This approach reduces idle time and can lead to faster convergence, especially in heterogeneous environments where clients have varying computational capabilities and network conditions.

FedAsync has been widely adopted in various federated learning applications due to its flexibility and ability to handle stale updates effectively. The algorithm uses a staleness function to adjust the impact of older updates, ensuring that more recent updates have a greater influence on the global model. In this paper, we will use FedAsync as a baseline for comparison and extend it to further improve its performance in specific scenarios.

**Algorithm 1** FedAsync

```
1:  Process Server ($\alpha \in (0,1)$) :
2:      Initialize $x_0$, $\alpha_t \leftarrow \alpha, \forall t \in [T]$
3:      Run Scheduler() thread and Updater() thread
            asynchronously in parallel

4:  Thread Scheduler :
5:      Periodically trigger training tasks on some workers,
            and send the global model with time stamp

6:  Thread Updater :
7:  for epoch $t \in [T]$ do
8:      Receive the pair $(x_{\text{new}}, \tau)$ from any worker
9:      Optional: $\alpha_t \leftarrow \alpha \times s(t-\tau)$, $s(\cdot)$ is a function of the
            staleness
10:     $x_t \leftarrow (1-\alpha_t)x_{t-1} + \alpha_t x_{\text{new}}$
11: end for

12: Process Worker :
13: for i $\in [n]$ in parallel do
14:     if triggered by the scheduler then
15:         Receive the pair of the global model and its time
                stamp $(x_t, t)$ from the server
16:         $\tau \leftarrow t$, $x_i^{\tau,0} \leftarrow x_t$
17:         Define $g_{x_t}(x;z) = f(x;z) + \frac{\rho}{2}\|x - x_t\|^2$,
                where $\rho > \mu$
18:         for local iteration $h \in [H_i^\tau]$ do
19:             Randomly sample $z_i^{\tau,h} \sim \mathcal{D}_i$
20:             Update $x_i^{\tau,h} \leftarrow x_i^{\tau,h-1} - \gamma \nabla g_{x_t}(x_i^{\tau,h-1}; z_i^{\tau,h})$
21:         end for
22:         Push $(x_i^{\tau,H_i^\tau}, \tau)$ to the server
23:     end if
24: end for
```

## 2.3 Simulation

Simulations of Federated Learning (FL) systems offer a versatile and controlled environment to explore various architectural configurations, including distributed, centralized, single-server, and multi-server setups. These simulations enable the execution of the learning process across different machines, which collaborate to develop a collective model without sharing raw data. In a simulated environment, some machines handle the actual computation and model training, while others may manage orchestration and aggregation tasks. *By the end of the simulation, a fully trained model is produced,* reflecting the combined efforts of all participating nodes.

One of the significant advantages of using simulations for FL systems is the elimination of practical concerns such as communication delays, network navigation, client dropouts, and synchronization issues. In a simulated setting, these factors can be controlled or abstracted away, allowing researchers and developers to focus on optimizing algorithms and strategies without the overhead of real-world implementation challenges. This streamlined approach facilitates faster iteration and testing of FL models, paving the way for more efficient and robust deployment in actual scenarios.

## 2.4 Scheduling

### Applications and Types

Scheduling is an operational decision-making process affecting company and organization performance and its ability to add value and to respect contracts. The application of scheduling is wide, starting from manufacturing and production systems to information processing environments as well as transportation and distribution systems. Typical scheduling problems include sequencing batches in continuous and discrete manufacturing environments to minimize setup times and/or maximize throughput while meeting due dates, assigning gates in airports, scheduling tasks in computing processing units, managing project activities in teams, healthcare, and timetabling.

Scheduling problems can be classified based on the nature of the tasks and the constraints involved. Some common types include:

- **Single-machine problems** - The simplest environment with a single machine processing all jobs.
- **Flow-shop problems** - Involve multiple machines arranged in series, where jobs follow the same sequence of machines.
- **Job-shop problems** - Feature multiple machines with jobs following different sequences, tailored to specific job requirements.
- **Parallel machines problems** - Involve multiple machines working in parallel, where jobs can be assigned to any machine.

### 3-Field Notation

Scheduling problems are often described using the three-field notation introduced by Graham et al. [7]. This notation comprises three fields, $\alpha|\beta|\gamma$, where the first field, $\alpha$, defines the scheduling environment: F for flow shop, J for job shop, P for parallel machines, and O for open shop. A specific number may indicate the number of machines. The second field, $\beta$, specifies job characteristics, such as preemption, ready times, and additional resources. The third field, $\gamma$, specifies the performance index or objective, such as minimization of makespan or maximum lateness. For example, $1|s_{jk}|C_{\max}$ denotes a single-machine problem with sequence-dependent setup times aiming to minimize the maximum completion time (makespan); $J_m||C_{\max}$ denotes a job-shop problem with $m$ machines and makespan minimization.

### Complexity and Solution Approaches

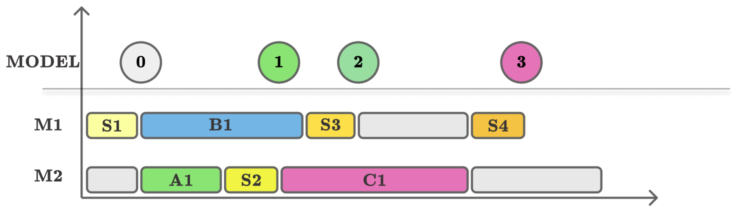
Over the years, numerous scheduling problems have been thoroughly examined, and a significant number of them have proven to be intractable [8, 9, 10, 11, 12]. One factor contributing to the complexity of these problems is the incorporation of precedence constraints [13, 14], such as ensuring that tasks follow a specific order like client update sequences. Exact solutions are rare, and heuristics and metaheuristics are frequently employed to find feasible solutions. Techniques like column generation, Lagrangian relaxation, or branch and cut are used when the problem is formulated as a mixed integer linear programming (MILP) model. For dynamic and stochastic scheduling, dispatching rules are often designed to handle uncertainty and variability. Problems with specific constraints, such as sequence-dependent setup times, can be addressed using graph theory.

## 3 Problem Description

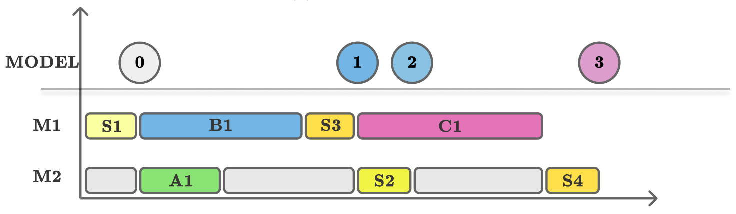
In Subsection 3.1 we describe the challenges that arise when simulating FL system, thus motivating the importance of the problem this paper studies. In Subsection 3.2 we formalise the scheduling problem and we go over notation.

### 3.1 Problem Motivation

Due to the non-IID distribution of data over client nodes in Federated Learning, if client $A$ finishes its update and aggregation before client $B$, the resulting global model will be different than if the order was reversed. The more heterogenous the setting becomes the bigger the variability can be. In order to be fair in comparing different ML models and FL algorithms, we would like to minimise this variability. One way to do so is by repeating simulations multiple times and looking at the set of results, but this process is costly and takes a lot of time. The approach studied in this paper is to take a sequence of client updates and fix all simulation schedules to this order of client updates.



(a) First scenario



(b) Second scenario

Figure 3: The numbered circles above represent the state of the global model during a simulation. The difference in colour indicates the different states the global model goes through.

Simulating an FL system requires the actual learning process to be executed, and thus it can take a long time. As we will see later scheduling simulation tasks via some sub-optimal policy such as first come first serve or Round Robin can significantly slow down simulations. With all of this in mind, we are interested in scheduling the simulation of client updates, in such a way that we minimise the completion time of the simulation while remaining consistent with a pre-specified execution order.

### 3.2 Problem Formulation

We are given a Directed Acyclic Graph $\mathcal{G} = (\mathcal{J}, \mathcal{E})$ specifying the job execution order we have to follow such as Fig 4. The set of jobs we have to simulate $\mathcal{J}$ is the set of vertices in the graph. A job $\mathcal{J}_i$ finishing before a job $\mathcal{J}_j$ starts, is represented with the edge $(\mathcal{J}_i, \mathcal{J}_j)$. We can only schedule $\mathcal{J}_j$ after $\mathcal{J}_i$ has finished executing.

Each job $\mathcal{J}_i$ takes $p_i$ time and it has to be scheduled for simulation on one of $m$ identical machines without preemption. Each machine can handle a single job at a time. Each job $\mathcal{J}_i$ belongs to a client $C_i$ from the deployment.

Finally, in any feasible schedule, we can compute the completion time $C_i$ of any job $\mathcal{J}_i$ as the sum of all the processing times of all jobs that were simulated on the same machine as $C_i$. Defining the makespan (the latest time of completion of any job) $C_{\max} = \max_{i \in \mathbb{Z}^+}\{C_i\}$ we aim to obtain a schedule that minimizes the makespan. In the popular three-field notation [7, 15] our scheduling problem can be stated as follows:

$$\mathrm{P} \mid prec \mid C_{\max} \tag{1}$$

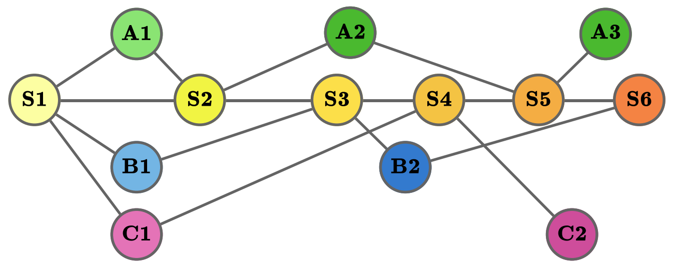


Figure 4: Precedence graph for the example system from Fig 6b. Any Node job connected to a Node job left of them has to wait to be simulated. For example, S4 can be simulated only after C1 has finished simulating. S2 can only be simulated after A1 has finished simulating and so on.

## 4 Related Work

### 4.1 Scheduling of P | *prec* | $C_{\max}$

The problem of scheduling tasks with precedence constraints on identical machines to minimize makespan is a well-known NP-Hard problem, as proven by Ullman [8]. This implies that finding an exact polynomial-time algorithm for this problem is highly unlikely. To tackle such complex scheduling issues, heuristic approaches like Evolutionary Algorithms (EAs) and Ant Colony Optimization (ACO) are explored.

**Ant Colony Optimisation**

Ant Colony Optimization, on the other hand, is inspired by the foraging behavior of ants and utilizes a colony of artificial ants to construct solutions iteratively, guided by pheromone trails and heuristic information. This approach is well-suited for combinatorial optimization problems and has been shown to perform well in scenarios where tasks have complex precedence constraints. ACO algorithms can adapt to dynamic changes and provide good-quality solutions relatively quickly. However, they may suffer from premature convergence and require fine-tuning of parameters like pheromone evaporation rate and the balance between exploration and exploitation. Liu et al. present an ACO algorithm for the $\mathrm{P} \mid prec, ST_{SD} \mid \mathrm{C}_{\max}$ problem [16]. Our problem is not concerned with sequence-dependent setup times, and for that reason, as well as the special structure of the precedence DAG in Section 6 we propose our own ACO algorithm for our version of the $\mathrm{P} \mid prec \mid C_{\max}$ problem.

**Evolutionary Algorithms**

Evolutionary Algorithms, inspired by natural selection and genetics, iteratively evolve a population of candidate solutions by applying operations such as mutation, crossover, and selection. EAs are particularly effective for large, complex search spaces due to their flexibility and ability to escape local optima. However, they can be computationally expensive and require careful tuning of parameters such as population size and mutation rate. Tayachi et al. present an EA for a problem related to ours, denoted as $\mathrm{P} \mid prec, p_j, C_{i,j} \mid \mathrm{C}_{\max}$[17]. However, their solution includes extraneous elements such as communication delays and does not consider the special structure of the precedence DAG in the problem we are interested. Therefore, in Section 6, we introduce our own Evolutionary Algorithm specifically designed to solve the $\mathrm{P} \mid prec \mid C_{\max}$ problem.

### 4.2 Simulation with Flower

Flower [18] is a comprehensive framework designed to facilitate the development and deployment of federated learning systems. Its appeal comes from its customizable and extendable nature, allowing users to tailor the framework to suit different use cases and research needs. Flower supports various machine learning frameworks, including TensorFlow, PyTorch, and scikit-learn, making it very compatibile and flexibile. However, Flower does not currently support asynchronous federated learning, which can limit its usfuleness, for instance in our case. Additionally, the framework lacks the capability to specify execution schedules readily, which is a desirable feature.

## 5 Methodology and Experimental Setup

To evaluate the performance and variability of Federated Learning algorithms under different conditions, we designed two experiments using the MNIST dataset. The first experiment explores how the variance of the accuract of the final trained model is impacted by imposing precedence constraints on the simulation schedule . The second experiment investigates the makespan of simulations under schedules obtained from different algorithms. We use the Flower framework with PyTorch to implement and run these experiments.

### 5.1 Experimental Setup

**Dataset and Model**

We use the MNIST dataset, a standard benchmark for image classification tasks. The dataset consists of 60,000 training images and 10,000 test images, each depicting a handwritten digit from 0 to 9. The classification model used in our experiments is a Multi-Layer Perceptron (MLP) with the following architecture:

- **Input layer** - 784 neurons (28x28 pixel images)
- **Hidden layer** - 128 neurons with ReLU activation
- **Output layer** - 10 neurons with softmax activation

**Client Distribution**

The MNIST dataset is split among multiple clients in a highly unbalanced manner, reflecting a non-IID distribution [19]. Specifically, each client receives a different number of data samples. The distribution of digit classes varies across clients, simulating real-world scenarios where data is unevenly distributed. The distribution of the global test set remains IID, having approximately the same amount of data points for each label.

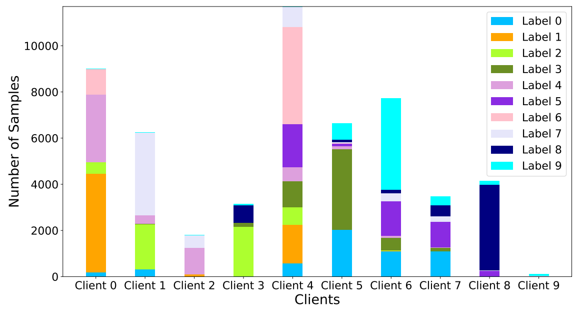


Figure 5: Distribution of the MNIST dataset over 10 clients. The data is distributed in an non-IID manner, different clients having different amounts of total data, as well as different proportions of data labels.

## 5.2 DAG Generation

A high-level idea for generating the precedence DAGs given a system specification is to assume probability distributions for the time each client and the parameter server take to finish their jobs. Then we can "execute" the algorithm without spending time on actual learning. By accumulating the processing times for learning and aggregation, we can keep track of when different client tasks start and finish.

Specifically, we use a priority queue to manage the times each task can start. When a task is popped from the queue, we draw from the distribution to determine how long it will take to simulate it. Once the task is completed, we update the queue by adding its succeeding tasks, along with their start times, based on the completion time of the current task. This allows us to efficiently simulate the execution order and timing of tasks in the system. We provide pseudocode describing this procedure below 2.

**Algorithm 2** Generating Precedence DAGs

```
 1: Input: Probability distributions for client processing times $\mathcal{D}_c$
    and server processing times $\mathcal{D}_s$
 2: Output: Precedence DAG

 3: Initialize a priority queue $Q$
 4: Initialize an empty DAG $G$
 5: Assume: Initial tasks and their start times are known
 6: for each initial task $t_i$ do
 7:     Draw processing time $p(t_i)$ from $\mathcal{D}_c$
 8:     Insert $(s(t_i), s(t_i) + p(t_i))$ into $Q$
 9: end for

10: while $Q$ is not empty do
11:     Pop client task $t_c$ with min completion time $c(t_c)$ from $Q$
12:     Draw server processing time $p(t_s)$ from $\mathcal{D}_s$
13:     Compute completion time $c(t_s) \leftarrow c(t_c) + p(t_s)$
14:     Update DAG: Add $t_s$ and $t_c$ and their edges to $G$
15: end while
16: Return: Precedence DAG $G$
```

## 5.3 Variance Reduction Experiment

In this experiment, we simulate two sets of FL deployments, each repeated five times. As a baseline, we simulate the FedAsynch algorithm without following any precedence constraints. We then compare these simulations to five additional simulations of the FedAsynch algorithm, this time scheduled according to a predefined set of precedence constraints. For both setups, we record the final model accuracy and compute the sample mean and variance. We expect to see a similar sample mean for both setups, but a lower variance for the simulations with precedence constraints.

## 5.4 Makespan Minimisation Experiment

In this experiment, we generate a set of precedence constraints and obtain schedules for simulation using the algorithms described in Section 4: the Ant Colony Optimization algorithm, the Evolutionary Algorithm, the 2-approximation algorithm, and a greedy First Come First Serve algorithm as a benchmark. We run simulations adhering to this schedule and track the total simulation times. We expect the Ant Colony Optimization, Evolutionary Algorithm, and 2-approximation algorithm to outperform the First Come First Serve algorithm in terms of minimising the makespan of the simulation.

## 5.5 Implementation

The experiments are implemented using the Flower framework, a flexible and scalable framework for federated learning. PyTorch is used for building and training the MLP model. The Flower framework handles client-server communication and orchestration of the asynchronous training process. The Flower framework does not provide asynchronous Federated Learning out-of-the-box and is thus extended. Furthermore, Flower does not provide support for submitting precedence constraints on schedules of simulations; this is also implemented.

## 6 Proposed Solution

We propose two heuristic algorithms to solve the scheduling problem - Ant Colony Optimisation, and an Evolutionary Algorithm. Both are described below and pseudocode is provided. Furthermore, we propose a consistent heuristic that is used in combination with the A$^*$ algorithm to find optimal schedules for small problem instances.

## 6.1 Ant Colony Optimisation

The Ant Colony Optimisation algorithm works by simulating the behavior of ants that construct solutions incrementally. Each ant builds a solution by probabilistically selecting jobs based on pheromone trails and heuristic information. Pheromone trails represent the quality of solutions found so far, with stronger trails indicating better solutions. The heuristic function assigns higher preference to jobs with smaller duration. Ants deposit pheromone on edges (transitions between partial schedules) proportional to the quality of the final solution. Over iterations, pheromones evaporate to encourage the exploration of new solutions, but they build up on edges that lead to good solutions. This balance between exploitation of known solutions and exploration of new ones allows ACO to effectively search for optimal or near-optimal schedules and escape local optima. We provide pseudocode for a general Ant Colony Optimisation algorithm below 3.

## 6.2 Evolutionary Algorithm

Evolutionary algorithms are heuristic optimization techniques inspired by natural evolution processes. They maintain a population of candidate solutions (individuals) representing potential solutions

**Algorithm 3** Ant Colony Optimization for Scheduling under Precedence Constraints

```
1: Initialize parameters: $\alpha, \beta, \rho, q$, num_ants, num_iterations
2: Initialize pheromone trails $\tau_{ij}$ for all jobs $i$ and $j$
3: Initialize best_schedule
4: while not reached num_iterations do
5:     Initialize ants with empty solutions
6:     for each ant do
7:         Construct current_schedule by probabilistically selecting jobs based on $\tau_{ij}$ and heuristic
8:         if current_schedule is better than best_schedule then
9:             Update best_schedule
10:        end if
11:        Update pheromone trails based on current_schedule quality
12:    end for
13:    Evaporate pheromone trails: $\tau_{ij} \leftarrow (1-\rho) \cdot \tau_{ij}$
14: end while
15: return best_schedule
```

to a problem. Each individual's fitness is evaluated based on a predefined objective function - the makespan of the schedule. In the context of our problem - scheduling under precedence constraints - each candidate solution (schedule) is represented as a permutation of jobs over machines. The algorithm proceeds through iterations (generations), where new solutions are generated through selection, crossover, and mutation operations.

- **Selection** favors individuals with higher fitness, ensuring better solutions survive and potentially improve over generations.
- **Crossover** combines parts of two parent solutions to create new offspring, promoting exploration of the search space. We take the schedules of the two parents and combine them ensuring constraints are still met.
- **Mutation** introduces small random changes to maintain diversity and prevent premature convergence. We randomly swap execution order of jobs.

By iteratively applying these operations, evolutionary algorithms improve the population until a satisfactory solution is found. We provide pseudocode for the general Evolutionary below 4.

**Algorithm 4** Evolutionary Algorithm for Scheduling under Precedence Constraints

```
1: Initialize population of schedules randomly
2:
3: while termination criteria not met do
4:     Select parents with highest fitness
5:     Crossover to generate new offspring
6:     Mutate offspring
7:     Select individuals for next generation
8: end while
9: return Best schedule found
```

## 6.3 The A* algorithm

A* (A-star) is a widely used pathfinding and graph traversal algorithm known for its efficiency and accuracy in finding the shortest path in a weighted graph. It employs a best-first search approach, where it uses a heuristic function to estimate the cost to reach the goal from any given node, combined with the actual cost to reach that node from the start. For A* to guarantee an optimal solution, the heuristic function $h(n)$ must be admissible, meaning it never overestimates the true cost to reach the goal. Additionally, if the heuristic is consistent (or monotonic), which means for any node $n$ and its successor $n'$, the estimated cost $h(n)$ is no greater than the cost from $n$ to $n'$ plus the estimated cost from $n'$ to the goal, then A* is guaranteed to find the shortest path and also ensures optimal efficiency.

**Algorithm 5** A* Search Algorithm

```
1: function A*(start, goal)
2:     openSet ← {start}
3:     cameFrom ← an empty map
4:     gScore ← map with default value of $\infty$
5:     gScore[start] ← 0
6:     fScore ← map with default value of $\infty$
7:     fScore[start] ← heuristic(start, goal)
8: while openSet is not empty do
9:     current ← node in openSet with lowest fScore[current]
10:    if current = goal then
11:        return reconstruct_path(cameFrom, current)
12:    end if
13:    openSet ← openSet \{current}
14:    for each neighbor of current do
15:        tentative_gScore ← gScore[current] + dist(current, neighbor)
16:        if tentative_gScore ¡ gScore[neighbor] then
17:            cameFrom[neighbor] ← current
18:            gScore[neighbor] ← tentative_gScore
19:            fScore[neighbor] ← gScore[neighbor] + heuristic(neighbor, goal)
20:            if neighbor not in openSet then
21:                openSet ← openSet ∪ {neighbor}
22:            end if
23:        end if
24:    end for
25: end while
26: return failure
```

**Optimistic Fit**

We propose the Optimistic Fit (OF) heuristic and argue that it is admissible. To evaluate a partial schedule, OF considers the remaining jobs and (optimistically) assumes that they can all be scheduled on all machines perfectly, meaning that no machine has idle time and all machines finish at the same time. This provides a lower bound to the actual distance from the solution since any solution that adheres to the precedence constraints and the non-preemptive nature of the jobs will acquire *at least* that much more makespan. This characteristic makes the Optimistic Fit heuristic admissible.

**Longest Chain**

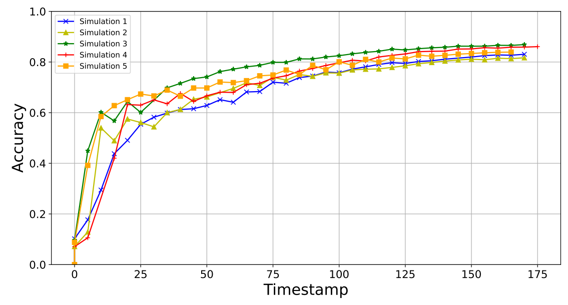
Another admissible heuristic is the Longest Chain (LC) heuristic. For a job finishing on some machine, it calculates the makespan of the longest chain of jobs that are successors to each other. This heuristic is admissible since all the jobs on the chain have to be executed, and they can only be executed one at a time. The fastest way to execute them is to have no idle time before the jobs on the chain, thus the sum of their processing times is a lower bound on the remaining time. Therefore, the Longest Chain heuristic is also admissible.

## 7 Results

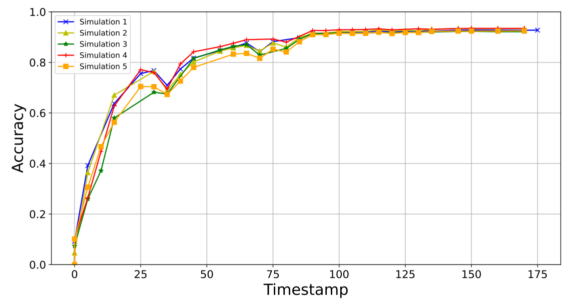
In this section, we present the results of the experiments conducted. Subsection 7.1 details the results of the Variance Reduction Experiment. In Subsection 7.2, we provide the outcomes related to the optimal minimization of the makespan of schedules under precedence constraints. Subsection 7.3 presents the results for the minimization of the makespan of schedules under precedence constraints, but without enforcing the optimality condition.

### 7.1 Variance Reduction Experiment

The results of the two sets of five simulations are presented and compared in the figures below 6 as well as in the table below 2.



(a) Five FL Simulations without Precedence Constraints



(b) Five FL Simulations under Precedence Constraints

Figure 6: Two sets of five FL simulations showing accuracy over time. Each marker shows the global model accuracy measured against a global dataset. The simulations in the former set of experiments have a varying final model accuracy, whereas the ones in the latter set of experiments converge to approximately the same final accuracy.

| | S 1 | S 2 | S 3 | S 4 | S 5 | $\mu$ | $\sigma^2$ |
|---|---|---|---|---|---|---|---|
| **normal** | 0.830 | 0.816 | 0.830 | 0.830 | 0.830 | 0.843 | $4.5 \times 10^{-4}$ |
| | 232.31 | 248.87 | 247.11 | 235.18 | 239.21 | 240.22 | 50.7 |
| **prec** | 0.926 | 0.936 | 0.923 | 0.930 | 0.929 | 0.929 | $2.3 \times 10^{-5}$ |
| | 314.31 | 321.45 | 328.41 | 309.81 | 326.1 | 319.6 | 64.3 |

Table 1: The five final model accuracies are given for two sets of simulations, one under no constraints, and one under precedence constraints. The sample mean and sample variance are given as well. On the lower rows we give the time each simulation took.

The F-test [20] is a statistical method used to compare the variances of two or more samples. Specifically, when comparing sample variances using the F-test, one calculates the ratio of the variances of two independent samples. This ratio follows an F-distribution under the Null Hypothesis that the variances of the populations from which the samples are drawn are equal.

To achieve a confidence level of $p = 0.05$ the critical value of the F-test is 6.3882. The ratio of the variance under no constraints $\sigma_{no}^2 = 4.5e^{-4}$ and the variance under precedence constraints $\sigma_{prec}^2 = 2.3e^{-5}$ is given by $\sigma_{no}^2 \div \sigma_{prec}^2 = 19.886$.

This value is higher than the critical value of the F-test. Thus we reject the Null Hypothesis, that the variances of the two populations are equal. With a sufficient level of confidence, we conclude that the variance of separate simulations under the same set of precedence constraints is smaller than that of simulations under no constraints. This happens without significantly increasing the time each simulation takes.

### 7.2 Makespan Minimisation - Dijkstra's and A$^*$

| | | 4 | 9 | 13 | 15 | 18 |
|---|---|---|---|---|---|---|
| | **Nodes** | 28 | 1 646 | 1 012 715 | 5 428 190 | |
| **Dijkstra's** | **Time** | 0.08 | 0.2 | 2.68 | 21.889 | |
| | **Makespan** | 15 | 21 | 35 | 36 | |
| | **Nodes** | 12 | 884 | 68 231 | 2 019 696 | |
| **A$^*$ – LC** | **Time** | 0.04 | 0.16 | 0.218 | 5.110 | |
| | **Makespan** | 15 | 21 | 35 | 36 | |
| | **Nodes** | 28 | 1 132 | 163 022 | 337 268 | 5 230 117 |
| **A$^*$ – OF** | **Time** | 0.07 | 0.19 | 0.489 | 3.754 | 14.695 |
| | **Makespan** | 15 | 21 | 35 | 36 | 45 |

Table 2: Dijkstra's algorithm, $A^*$ with LC heuristic, and $A^*$ with OF heuristic are assessed on 5 problem instances with 4, 9, 13, 15, and 18 jobs to be scheduled on 2 machines. For each algorithm, it is given how many nodes were expanded while traversing the search space, how much time (in seconds) it took the algorithm to arrive to a solution, and what was the makespan of the final schedule.

The 3 exact algorithms given above are assessed on several problem instances of varying sizes. As the search space grows exponentially, the slower algorithms can not find a solution for larger problem instances due to machine and time constraints. The pure Dijkstra's algorithm and $A^*$ with the LC heuristic can not find a solution to the problem instance with 18 jobs in a reasonable time. $A^*$ with OF manages to find a solution for the problem instance with 18 jobs but fails for larger instances.

For each problem instance, all 3 algorithms arrive at solutions with the same makespan, showing empirically that the two proposed heuristics are indeed admissible. By extension the found solutions are optimal.

### 7.3 Makespan Minimisation - Heuristic Approaches

The two proposed heuristic algorithms - Ant Colony Optimisation and Evolutionary Algorithm are assesed on various problem instances and benchmarked against a randomly constructed schedule that satisfies the precedence constraints. First they are assesed on the same 5 problem instances as the exact algorithms were. Then they are assesed on 4 larger problem instances.

| | | 4 | 9 | 13 | 15 | 18 |
|---|---|---|---|---|---|---|
| **Random** | **Time** | $4.3{\times}10^{-5}$ | $7.7{\times}10^{-5}$ | $1.1{\times}10^{-4}$ | $8.1{\times}10^{-5}$ | $1.3{\times}10^{-4}$ |
| | **Makespan** | 15 | 23 | 36 | 44 | 51 |
| **ACO** | **Time** | 0.004 | 0.016 | 0.025 | 0.031 | 1.036 |
| | **Makespan** | 15 | 24 | 36 | 39 | 47 |
| **EA** | **Time** | 0.138 | 0.325 | 0.389 | 0.434 | 0.511 |
| | **Makespan** | 15 | 21 | 35 | 36 | 45 |

Table 3: Random schedule, ACO, and EA assesed on 5 problem instances with 4, 9, 13, 15, and 18 jobs to be scheduled on 2 machines. For each algorithm, it is given how much time (in seconds) it took the algorithm to arrive to a solution, and what was the makespan of the final schedule.

| | | 50 | 100 | 250 | 1000 |
|---|---|---|---|---|---|
| **Random** | **Time** | $2.9{\times}10^{-4}$ | $1.1{\times}10^{-3}$ | $1.3{\times}10^{-3}$ | $5.9{\times}10^{-3}$ |
| | **Makespan** | 99 | 296 | 618 | 3019 |
| **ACO** | **Time** | 2.146 | 3.501 | 4.712 | 67.98 |
| | **Makespan** | 90 | 275 | 512 | 2458 |
| **EA** | **Time** | 1.414 | 3.533 | 16.14 | 216.79 |
| | **Makespan** | 78 | 269 | 554 | 2552 |

Table 4: Random schedule, ACO, and EA assesed on 5 problem instances with 50, 100, 250, and 1000 jobs to be scheduled on 3 machines. For each algorithm, it is given how much time (in seconds) it took the algorithm to arrive to a solution, and what was the makespan of the final schedule.

From the first set of experiments 3 we can conclude that our heuristic approaches do not deviate much from the optimal solution for small problem instances. From the second set of experiments 4 we can conlude that the ACO algorithm outperforms the EA when it comes to minimising the makespan of simulations for larger problem instances, and it takes less time to converge to a solution.

## 8 Responsible Research

In this study, ensuring the reproducibility of our results was a primary focus to uphold the principles of responsible research. To facilitate this, we have made all relevant code and data publicly accessible in a dedicated GitHub repository[1]. This allows other researchers to replicate our experiments, validate our findings, and build upon our work with confidence. By providing clear documentation and version control within the repository, we aim to promote transparency, enhance the credibility of our results, and contribute to the collective advancement of knowledge in our field.

## 9 Discussion and Future Work

### 9.1 Setup Times are Unimportant

An initial idea when considering what scheduling problem is appropriate for the problem was the inclusion of sequence-dependent setup times. This means that whenever any job begins execution on a given machine, it incurrs a time needed for the machine to be set up based on the previously executed task.

It was initially considered to adopt this scheduling problem, due to the fact that big chunks of data - local datasets and model parameters - have to move around the simulation machine whenever a different client begins a new round of training. It was decided that those setup times can be ignored, since whenever a model finishes training, it has to be aggregated to the global parameter server. This means that either the server's parameters have to be loaded or the client's parameters have to loaded. Thus, no benefit can be extracted from considering sequence dependent setup times.

[1]https://github.com/todor-slavov/fl-simulations-and-scheduling

### 9.2 Scheduling Implementation in Flower

The current implementation allows us to enforce precedence constraints on the schedule of simulation. It would be nice to further be able to control the simulation in it's entirety. This would allow us to study the impacts of scheduling under precedence constraints into greater detail, and minimise the overhead from imposing those constraints.

## 10 Conclusion

This paper addresses the challenge of high variance in Federated Learning simulation by proposing a novel approach - scheduling simulations under precedence constraints. Our research demonstrates that this approach can significantly reduce model variance without markedly increasing simulation time.

We propose two admissible heuristics for search algorithms and empirically demonstrate their effectiveness in combination with the $A^*$ algorithm. We propose two heuristic algorithms - an Ant Colony Optimisation and an Evolutionary Algorithm and benchmark them against randomly generated schedules. They consistently outperform the benchmark and keep close to the optimal solution values.

In summary, our work demonstrates that strategic scheduling can reduce model variance and improve the robustness of FL systems, offering a promising direction for more consistent and reliable FL deployments.

## References

[1] Q. Li, Y. Diao, Q. Chen, and B. He, "Federated learning on non-iid data silos: An experimental study," in *2022 IEEE 38th international conference on data engineering (ICDE)*, pp. 965–978, IEEE, 2022.

[2] F. Granqvist, C. Song, Á. Cahill, R. van Dalen, M. Pelikan, Y. S. Chan, X. Feng, N. Krishnaswami, V. Jina, and M. Chitnis, "pfl-research: simulation framework for accelerating research in private federated learning," *arXiv preprint arXiv:2404.06430*, 2024.

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

[4] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous federated optimization," *arXiv preprint arXiv:1903.03934*, 2019.

[5] B. Cox, A. Mălan, J. Decouchant, and L. Y. Chen, "Asynchronous byzantine federated learning," *arXiv preprint arXiv:2406.01438*, 2024.

[6] Y. Zuo, B. Cox, J. Decouchant, and L. Y. Chen, "Asynchronous multi-server federated learning for geo-distributed clients," *arXiv preprint arXiv:2406.01439*, 2024.

[7] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," in *Annals of discrete mathematics*, vol. 5, pp. 287–326, Elsevier, 1979.

[8] J. D. Ullman, "Np-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[9] J. Bruno, E. G. Coffman Jr, and R. Sethi, "Scheduling independent tasks to reduce mean finishing time," *Communications of the ACM*, vol. 17, no. 7, pp. 382–387, 1974.

[10] R. Sethi, "On the complexity of mean flow time scheduling," *Mathematics of Operations Research*, vol. 2, no. 4, pp. 320–330, 1977.

[11] M. R. Garey and D. S. Johnson, "Complexity results for multiprocessor scheduling under resource constraints," *SIAM journal on Computing*, vol. 4, no. 4, pp. 397–411, 1975.

[12] M. R. Garey and D. S. Johnson, "Scheduling tasks with nonuniform deadlines on two processors," *Journal of the ACM (JACM)*, vol. 23, no. 3, pp. 461–467, 1976.

[13] J. K. Lenstra and A. Rinnooy Kan, "Complexity of scheduling under precedence constraints," *Operations Research*, vol. 26, no. 1, pp. 22–35, 1978.

[14] J. R. Correa and A. S. Schulz, "Single-machine scheduling with precedence constraints," *Mathematics of Operations Research*, vol. 30, no. 4, pp. 1005–1021, 2005.

[15] P. Brucker, "Scheduling algorithms," *Journal-Operational Research Society*, vol. 50, pp. 774–774, 1999.

[16] G. B. Liu, Q. Zhu, and J. Zhang, "Aco algorithm for scheduling complex unrelated parallel machine," *Advanced Materials Research*, vol. 268, pp. 297–302, 2011.

[17] D. Tayachi, *Solving the P/Prec, pj,Cij/CmaxUsing an Evolutionary Algorithm*, pp. 385–397. Cham: Springer International Publishing, 2018.

[18] D. J. Beutel, T. Topal, A. Mathur, X. Qiu, J. Fernandez-Marques, Y. Gao, L. Sani, H. L. Kwing, T. Parcollet, P. P. d. Gusmão, and N. D. Lane, "Flower: A friendly federated learning research framework," *arXiv preprint arXiv:2007.14390*, 2020.

[19] B. Cox, L. Y. Chen, and J. Decouchant, "Aergia: leveraging heterogeneity in federated learning systems," in *Proceedings of the 23rd ACM/IFIP International Middleware Conference*, pp. 107–120, 2022.

[20] R. S. Witte and J. S. Witte, *Statistics*. John Wiley & Sons, 2017.