Implementation and Evaluation of Cross-Silo Federated Learning without Gradient Descent

Chiavazza Edoardo

Università d'Informatica di Torino

Abstract. Federated Learning has emerged as a framework for enhancing AI systems while preserving user privacy and protecting private enterprise data. While traditionally relying on neural networks and gradient descent-based algorithms through weight or gradient exchange, this approach limits applicability in scenarios requiring alternative models with greater interpretability or superior performance in specific contexts.
Building on "Boosting the Federation: Cross-Silo Federated Learning without Gradient Descent", this study implements and evaluates decision tree-based techniques. We implement three algorithms: classical AdaBoost and two MPI-based federated variants, evaluating their strong and weak scaling performance alongside ensemble accuracy.

1 Introduction

Federated Learning (FL) is a paradigm where multiple clients collaborate on machine learning tasks using private data under an aggregator's coordination. Local data remains isolated, with learning occurring in rounds where clients compute model updates using private data, aggregate results on the server, and broadcast for subsequent rounds.

Two main federated settings exist: cross-device FL involves numerous edge devices (thousands) with limited computational power and reliability, while cross-silo FL involves organizations (typically 2-100 parties) where communication and computation constraints are less restrictive.

This work proposes cross-silo FL algorithms for classification, drawing inspiration from AdaBoost and distributed boosting literature. The algorithms impose minimal constraints on client learning settings, accommodating models not specifically designed for FL, such as decision trees and SVMs. Models trained at each epoch combine to form ensemble learners that strengthen performance by iterating and combining results.

2 AdaBoost Algorithm

Freund and Schapire proved that AdaBoost reduces ensemble error exponentially fast in the number T of combined weak models, provided weak learners consistently outperform random guessing. This holds even under adversarial behavior,

though federated scenarios often yield lower accuracy than sequential versions due to data subset limitations.

Crucially, if at least one client produces a model better than random guessing over the entire dataset, distributed AdaBoost can drive ensemble error to its minimum exponentially fast while preserving data privacy.

Algorithm 1: SAMME Algorithm

Input: Training data $(x_1, y_1), \ldots, (x_N, y_N)$ where $y_i \in \{1, \ldots, K\}$;
Number of iterations M;
Output: Ensemble classifier $H(x)$

Initialize sample weights: $D_1(i) = \frac{1}{N},\ \forall i = 1, \ldots, N$;
for $m = 1$ ***to*** M **do**
  Train weak classifier h_m on training data using weights D_m;
  Compute error: $\varepsilon_m = \frac{\sum_{i=1}^{N} D_m(i) \cdot [\![h_m(x_i) \neq y_i]\!]}{\sum_{i=1}^{N} D_m(i)}$;
  Compute classifier weight: $\alpha_m = \log\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right) + \log(K-1)$;
  Update sample weights: $D_{m+1}(i) = D_m(i) \cdot \exp\left(\alpha_m \cdot [\![h_m(x_i) \neq y_i]\!]\right)$;
  Normalize weights: $D_{m+1}(i) = \frac{D_{m+1}(i)}{\sum_{j=1}^{N} D_{m+1}(j)}$;
end
Construct final classifier: $H(x) = \arg\max_k \sum_{m=1}^{M} \alpha_m \cdot [\![h_m(x) = k]\!]$;

AdaBoost.SAMME (Stagewise Additive Modeling using Multi-class Exponential loss) extends AdaBoost for multi-class classification ($K \geq 2$ classes), building additive models by sequentially combining weak learners where each focuses on previously misclassified samples.

3 Federated AdaBoost Algorithms

Our federated framework uses Message Passing Interface (MPI), a standardized protocol for parallel computing in distributed memory architectures. MPI enables explicit message passing between processes across clusters or multicore systems, aligning with federated learning's core tenet of preserving data privacy without raw data exchange.

Experiments were conducted on HPC4AI, a high-performance computing platform in Turin, Italy, simulating federated environments. A central process manages initial dataset partitioning to client processes, but once initialization completes, all computations occur in isolation with no further inter-client communication.

3.1 Federated AdaBoost (Master-Slave Version)

AdaBoost.F allows clients to train single models per round, retaining only the best performer while the master optimizes weight distributions. This avoids model averaging that might hinder local data specialization, exploring hypothesis spaces that include non-overly-specific models.

Algorithm 2: AdaBoost.F (Aggregator)

Input: C: number of clients
T: ensemble dimension
K: number of classes
Output: $\text{ens}(\mathbf{x}) \triangleq vote([h^{t*}]_{t=1}^{T}, [\alpha^t]_{t=1}^{T}, \mathbf{x})$

Initialize $N \leftarrow \sum_{c=1}^{C} \texttt{receive}_n(c)$;
for $t \leftarrow 1$ **to** T **do**
  $\mathbf{h}^t \leftarrow \|_{c=1}^{C} \texttt{receive}_h(c)$; `// Collect C hypotheses`
  $\texttt{broadcast}_h(\mathbf{h}^t)$; `// Send to clients`
  $\mathbf{E}^t \leftarrow [\texttt{receive}_e(c)]_{c=1}^{C}$; `// Error matrix`
  $c^{t*} \leftarrow \arg\min_{c} \sum_{c'=1}^{C} \mathbf{E}_{c,c'}^{t}$;
  $\epsilon^{t*} \leftarrow \frac{1}{N} \sum_{c=1}^{C} \mathbf{E}_{c,c^{t*}}^{t}$;
  $\alpha^t \leftarrow \log\left(\frac{1-\epsilon^{t*}}{\epsilon^{t*}}\right) + \log(K-1)$;
  $\texttt{broadcast}_\alpha(\alpha^t)$;
  $\texttt{broadcast}_c(c^{t*})$;
$\texttt{broadcast}_{\text{stop}}(\text{STOP})$; `// Termination`

Algorithm 3: AdaBoost.F (Client)

Input: $\mathcal{A}$: weak learner
$\mathbf{X} \in \mathbb{R}^{n \times m}$: training data
$\mathbf{y} \in \{1, \ldots, K\}^n$: labels

$\texttt{send}_n(\text{aggr}, n)$; `// Report data size`
Initialize $\mathbf{d} \leftarrow \mathbf{1}$; `// Uniform distribution`
repeat
  $h \leftarrow \mathcal{A}\left(\mathbf{X}, \mathbf{y}, \frac{\mathbf{d}}{\|\mathbf{d}\|_1}\right)$; `// Train model`
  $\texttt{send}_h(\text{aggr}, h)$;
  $\mathbf{h}^t \leftarrow \texttt{receive}_h(\text{aggr})$; `// Receive hypotheses`
  $\epsilon \leftarrow \left[\mathbf{d}^\top [\![\mathbf{y} \neq h_c(\mathbf{X})]\!]\right]_{c=1}^{|\mathbf{h}^t|}$;
  $\texttt{send}_e(\text{aggr}, \epsilon)$; `// Send errors`
  $\alpha^t \leftarrow \texttt{receive}_\alpha(\text{aggr})$;
  $c^{t*} \leftarrow \texttt{receive}_c(\text{aggr})$;
  Update $\mathbf{d} \leftarrow \left[d_i \exp\left(-\alpha^t (-1)^{[\![h_{c^{t*}}(\mathbf{x}_i) \neq y_i]\!]}\right)\right]_{i=1}^{n}$;
until $\texttt{receive}_{stop}(aggr)$;

This architecture involves clients transmitting locally trained models to a master, who broadcasts all models back to clients. Each client computes weighted errors for all models, forming matrix $\mathbf{E}$ at the aggregator to identify the best model:

$$c^{t*} \leftarrow \arg\min_{c} \sum_{c'=1}^{C} \mathbf{E}_{c,c'}^{t} \tag{1}$$

These errors update training example weights, with higher weights increasing influence in subsequent rounds to better explore challenging solution space regions.

3.2 Federated AdaBoost (Distributed)

The decentralized version addresses centralized architecture limitations, particularly single points of failure. By distributing critical operations (model selection, error computation, weight calculation) across all nodes, we improve system robustness and fault tolerance.

Algorithm 4: Federated AdaBoost (Distributed)

Input: C: number of clients
T: ensemble dimension
K: number of classes
Output: $\text{ens}(\mathbf{x}) \triangleq vote([h^{t*}]_{t=1}^{T}, [\alpha^t]_{t=1}^{T}, \mathbf{x})$

Initialize $W \leftarrow 1/C$;
for $t \leftarrow 1$ **to** T **do**
  $h \leftarrow \mathcal{A}\left(\mathbf{X}, \mathbf{y}, \frac{\mathbf{d}}{\|\mathbf{d}\|_1}\right)$; // Train model
  $\texttt{send}_h(\text{all}, h)$; // Broadcast to all clients
  $\mathbf{h}^t \leftarrow \texttt{receive}_h(\text{all})$; // Receive hypotheses
  $\epsilon \leftarrow \left[\mathbf{d}^\top [\![\mathbf{y} \neq h_c(\mathbf{X})]\!]\right]_{c=1}^{|\mathbf{h}^t|}$;
  $\texttt{send}_e(\text{all}, \epsilon)$; // Send errors to all
  $\mathbf{E}^t \leftarrow [\texttt{receive}_e(c)]_{c=1}^{C}$; // Error matrix
  $c^{t*} \leftarrow \arg\min_c \sum_{c'=1}^{C} \mathbf{E}_{c,c'}^t$;
  $\texttt{send}_c(\text{all}, c^{t*})$; // Send choice to all
  $\mathbf{M}^t \leftarrow [\texttt{receive}_c(c)]_{c=1}^{C}$; // Receive all choices
  $m^{t*} \leftarrow \arg\max_v \text{count}_{\mathbf{M}}(v)$; // Best model by vote
  $\epsilon^{t*} \leftarrow \frac{1}{C} \sum_{c=1}^{C} \mathbf{E}_{c,m^{t*}}^t$;
  $\alpha^t \leftarrow \log\left(\frac{1-\epsilon^{t*}}{\epsilon^{t*}}\right) + \log(K-1)$;
  Update $\mathbf{d} \leftarrow \left[d_i \exp\left(-\alpha^t (-1)^{[\![h_{m^{t*}}(\mathbf{x}_i) \neq y_i]\!]}\right)\right]_{i=1}^{n}$;

This fully decentralized approach operates through: independent local model training, all-to-all model broadcasting, local evaluation of all received models, collaborative best model selection via voting, and local weight updates. The iterative process builds robust ensembles without requiring central coordination.

4 Experimental Setting

We compared two federated algorithms against centralized SAMME on HPC4AI infrastructure in Turin, Italy. Testing various weak learner counts ($T \in \{1, 5, 10, 20, 30, 40, 50, 100\}$) with Decision Trees as base learners, we explored different hyperparameter combinations including tree depth, splitting criteria, and minimum samples per leaf. Both algorithms terminate when weak classifier error reaches 0.5.

The algorithms are agnostic to weak learner choice and could employ different models. We tested varying client numbers to analyze impacts on speedup and accuracy, assuming perfect participation without failures or dropouts.

4.1 Dataset

The Forest Cover Type dataset predicts forest cover from cartographic variables, containing 581,012 observations across 55 variables from four wilderness areas in Roosevelt National Forest, Colorado. Class distribution shows imbalance: Class 1 (91,244), Class 2 (121,478), Class 3 (15,530), Class 4 (1,177), Class 5 (4,119), Class 6 (7,608), Class 7 (8,844). The test set comprises 174,303 points with distribution: Class 1 (56,000), Class 2 (72,000), Class 3 (38,453), Class 4 (350), Class 5 (1,200), Class 6 (5,900), Class 7 (14,400).

4.2 Performance Metrics

Strong scaling maintains fixed problem size while increasing processors, measuring execution time decrease. With T_1 as single-processor time and T_p as p-processor time:

$$S(p) = \frac{T_1}{T_p} \tag{2}$$

Weak scaling increases problem size proportionally with processors, maintaining constant per-processor workload. Perfect scaling yields constant execution time:

$$E(p) = \frac{T_1}{p \cdot T_p} \tag{3}$$

5 Results

The following table shows the results of the various algorithms. For the two federated versions we put only the best model for speedup or accuracy

Table 1. Performance comparison of different algorithms

Algorithm	N Clients	N Epochs	Time (s)	Precision (%)
SAMME	1	1	29.68	77.0
AdaBoost (FMS)	252	1	0.49	67.3
AdaBoost (FMS)	108	30	15.02	73.1
AdaBoost (FD)	252	1	0.29	64.8
AdaBoost (FD)	108	30	8.47	70.1

In the next sections, the results of the two algorithms will be analyzed in more detail.

5.1 Master-Slave Version

Strong scaling analysis reveals significant deviations from ideal linear speedup. Figure 1 shows real speedup plateauing around 60× at 252-288 processes, with initial efficiency of 57% at 50 processes declining to 29% at 180 processes. Performance degradation beyond 288 processes indicates communication overhead outweighing parallelization benefits.

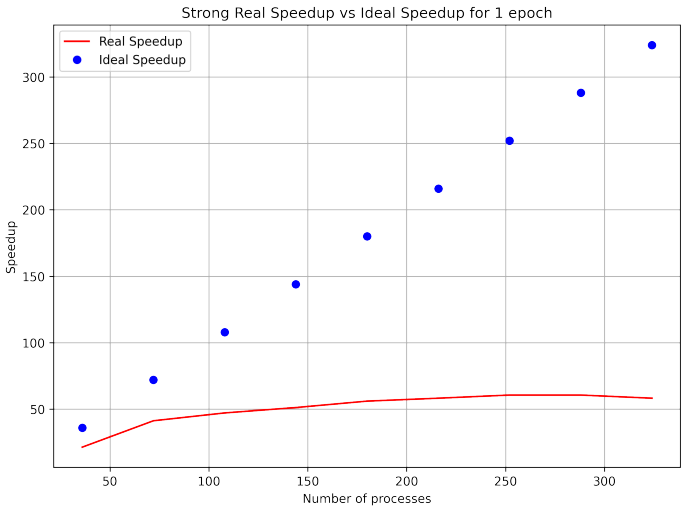

Fig. 1. Strong scaling comparison for single training epoch

Multi-epoch configurations (Figure 2) exhibit similar patterns, with fewer epochs achieving marginally superior speedup. All configurations peak around 252-288 processes, with more pronounced decline in higher-epoch configurations, consistent with Amdahl's Law limitations.

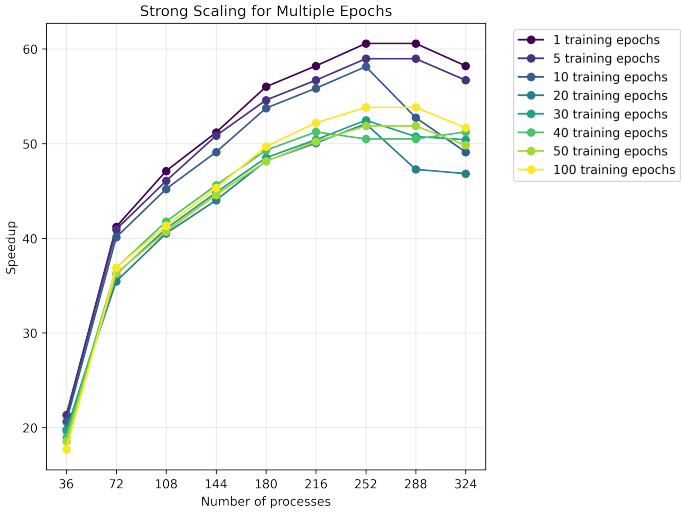

Fig. 2. Strong scaling across multiple training epochs

Parallel efficiency analysis (Figure 3) shows maximum 60% efficiency at 36 processes, declining to 18% at 324 processes. The substantial communication overhead is evident even at optimal configurations.

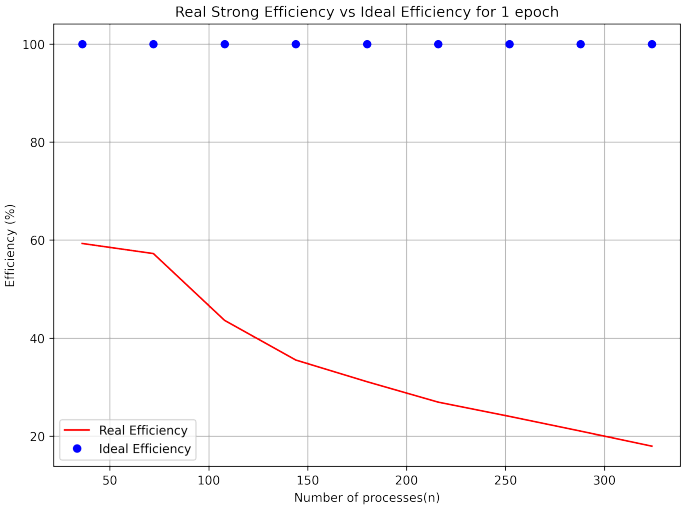

Fig. 3. Parallel efficiency for single epoch

Multi-epoch efficiency measurements (Figure 4) confirm that scaling limitations are algorithmic rather than workload-specific, with all configurations converging to 15-18% efficiency at 324 processes.

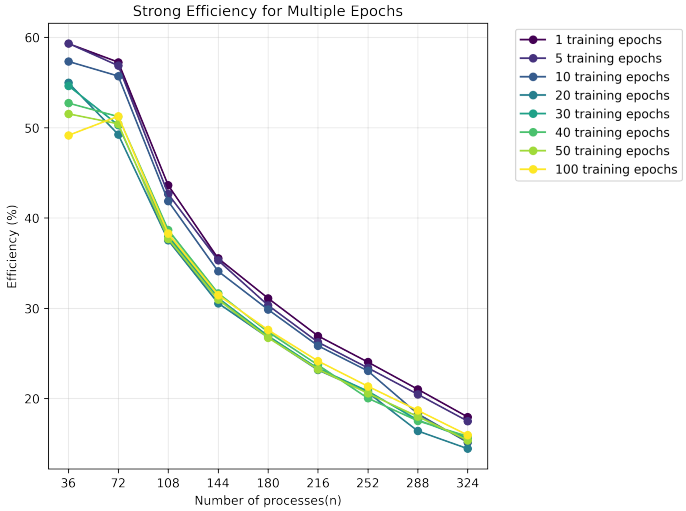

Fig. 4. Efficiency across multiple training epochs

Weak scaling analysis (Figure∼5) demonstrates near-optimal performance (≈1.0) at 36-72 processes, degrading to 0.69 at 324 processes. This pattern reveals excellent initial scaling with communication bottlenecks emerging at higher process counts.

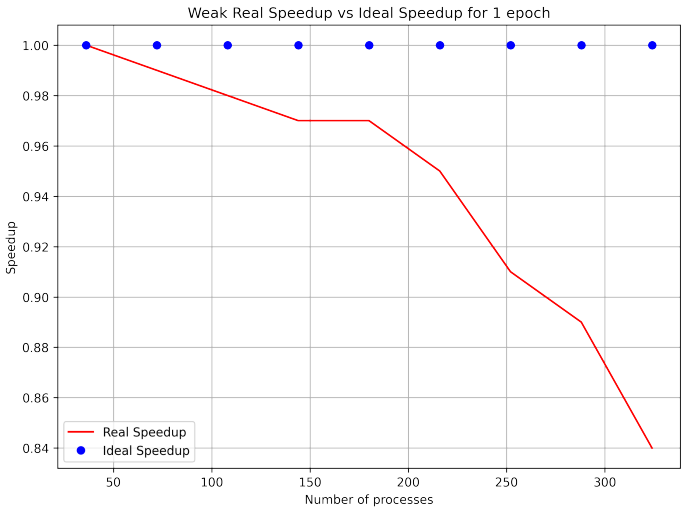

Fig. 5. Weak scaling for single epoch

5.2 Decentralized Version

The decentralized implementation employs fully distributed coordination where nodes independently train models, exchange them through collective communications, collaboratively select optimal models, and locally update weights. This approach eliminates single points of failure while maintaining AdaBoost's iterative improvement methodology.

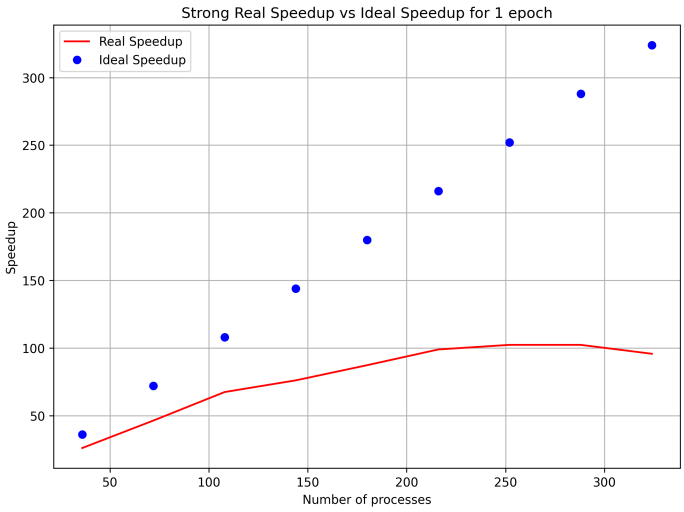

Fig. 6. Strong scaling for decentralized version (single epoch)

Figure 6 demonstrates the strong scaling characteristics of the decentralized approach for a single epoch. The decentralized version exhibits different scaling behavior compared to the master-slave implementation due to its peer-to-peer communication pattern. Notable characteristics include more consistent initial scaling and different bottleneck patterns related to collective communication overhead.

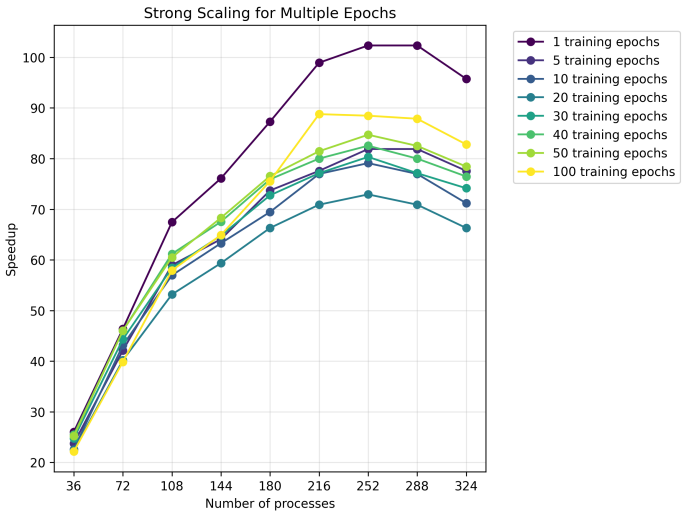

Fig. 7. Strong scaling across multiple epochs - decentralized version

The multi-epoch analysis (Figure 7) reveals that the decentralized approach maintains more consistent performance across different epoch configurations. Unlike the master-slave version, the performance gap between different epoch counts is less pronounced, suggesting better load balancing in the distributed coordination mechanism.

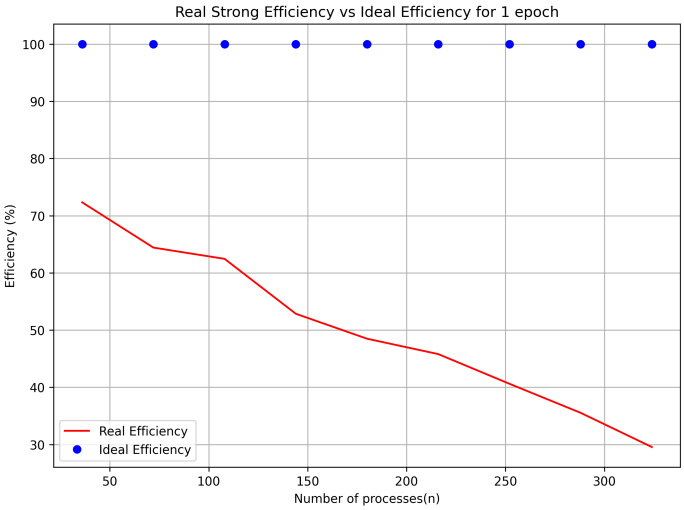

Fig. 8. Parallel efficiency for decentralized version (single epoch)

Parallel efficiency analysis (Figure 8) shows the decentralized version achieving higher initial efficiency compared to the master-slave approach. The efficiency degradation is more gradual, indicating better utilization of computational resources across the scaling range. Maximum efficiency reaches approximately 75% at lower process counts, significantly higher than the master-slave version's 60%.

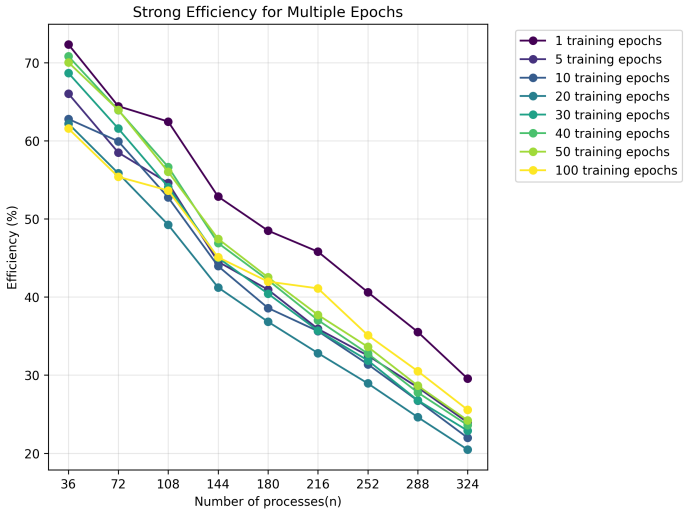

Fig. 9. Efficiency across multiple epochs - decentralized version

Multi-epoch efficiency measurements (Figure 9) demonstrate that the decentralized approach maintains superior efficiency across all tested configurations. The convergence behavior at high process counts is less severe, with final efficiency values remaining above 25% even at 324 processes, compared to the master-slave version's 15-18%.

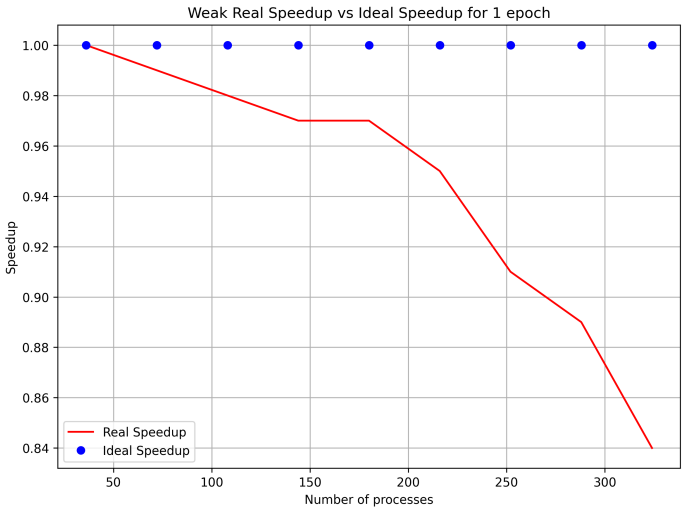

Fig. 10. Weak scaling for decentralized version (single epoch)

Weak scaling analysis (Figure∼10) reveals superior performance compared to the master-slave version. The decentralized approach maintains near-optimal scaling ($\approx$ 0.95) across a broader range of process counts (36-144 processes), with more gradual degradation thereafter. Final performance at 324 processes reaches approximately 0.78, substantially better than the master-slave version's 0.69.

5.3 Comparison master-salve and decentralized algorithms

The two algorithms achieve fair results. The former with a master-slave architecture suffers from some bottleneck problems. The master node is stressed at multiple points which causes delays and idle nodes. The major stress points are receiving all the models and having to broadcast them to the various nodes. Also, the calculation of the total error and alpha is done only by the master, while the client nodes wait for the two measurements to update the weights. The positive factor is that the master node having all the data available, the calculation of alpha and total error is better. So the algorithm is slower but more accurate than the decentralized version.
The second is considerably faster, as many of the bottleneck problems have been solved. The problem is that it also increases the commonality and synchronization ratio. In fact, each node must broadcast its model to the other nodes and expect to receive n - 1 models where n is the number of processors. This communication phase is the most expensive phase. Another synchronization point is in the common model choice that requires each node to broadcast its choice to the other nodes and expect to receive the choices of the other nodes. At this point the calculation of the error and alpha is done locally, so it is faster than

the master-save algorithm.
The first algorithm can be improved by replicating several master nodes that are responsible for receiving a portion from the total number of models. Once the masters have calculated the errors of the obtained models they communicate with each other to obtain the final result. The final result is broadcasted to the other nodes by the master that received the model initially. This would partially solve the fact that the master node in multiple parts of the algorithm acts as a bottleneck at the cost of adding some synchronization cost among the master nodes.
In conclusion, the master slave algorithm is a good algorithm when one wants to quickly train a set of weak learners with a fair amount of accuracy, while the decentralized version is likely to fail to capture the complexity of the dataset.