Parallel Random Forest using MPI and OpenMP

Lorenzo Chicco, Giuseppe Curci
lorenzo.chicco@studenti.unitn.it [245704]
giuseppe.curci@studenti.unitn.it [243049]

***Abstract*—Ensembling techniques are one of the most used algorithms in machine learning for their flexibility and ease of implementation. Unfortunately, their training cost grows linearly with the number of estimators and, for large datasets, can become unsustainable. In this work, we only focus on the Random Forest Classifier algorithm, and we propose a solution leveraging both data parallelism and task parallelism with significant improvement over the sequential version. The code is available on github.**

I. Introduction

Random forest is a powerful and widely used algorithm in supervised learning, where the goal is to learn to predict an output variable given some features. It belongs to the broader family of ensemble methods, which combine multiple weak learners (decision trees) to form a robust predictor. Introduced by Breiman [1], random forests gained popularity due to their ability to handle heterogeneous variables, resist overfitting via randomisation, and benefit from post-hoc pruning.

In this work, we focus on classification trees for simplicity, although the parallelisation strategies apply equally to regression tasks. Random forests are an example of "perfectly parallelizable" problem, as each tree in the ensemble can be trained independently. Moreover, training deep trees over large datasets can become prohibitively slow without parallelisation. To address this, we implement a parallel random forest using OpenMP for multi-threading data parallelism and MPI for multi-node task parallelism, leveraging hybrid parallelism.

The paper first introduces the metrics used to evaluate classification and parallelisation. Then it explains how classification trees and random forests work, from a statistical and computational perspective, and how we parallelised the algorithm. Finally in the experimental section we present our results showing that all three parallel versions significantly improve over the sequential version.

II. Methods

A. Metrics

To assess the quality of predictions, we use standard classification metrics:

- **Accuracy**: measures the overall correctness of predictions.
$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$
- **Precision**: evaluates how many predicted positives are actually correct.
$$\text{Precision} = \frac{TP}{TP + FP}$$
- **Recall**: quantifies how many actual positives are correctly identified.
$$\text{Recall} = \frac{TP}{TP + FN}$$

Where:

- TP: True Positives
- TN: True Negatives
- FP: False Positives
- FN: False Negatives

To evaluate the quality of our parallelisation we also use common metrics:

- **Speedup**: quantifies how much faster the parallel version is compared to the sequential baseline.
$$S = \frac{T_s}{T_p}$$
where T_s is the execution time using one process and T_p is the time using p processes.
- **Efficiency**: measures how effectively the processes are utilised.
$$E = \frac{S}{p} = \frac{T_s}{p \cdot T_p}$$

Higher speedup and efficiency indicate better parallel scalability. For the hybrid implementation leveraging both MPI and OpenMP we use the same metrics. Thus we consider threads and nodes as an abstraction of execution units and the number of processes per job is obtained by multiplying the number of nodes and the number of threads.

B. Sequential Random Forest

Let a dataset D consist of N samples and M variables, where each sample is a pair $\{x_i, y_i\}_{i=1}^{N}$, with $x_i \in \mathbb{R}^{1\times M-1}$ representing the features and $y_i \in \{0, \ldots, K\}$ the corresponding dependent variable (class label). A classification decision tree partitions the input space into R regions—also called *nodes*—and assigns a class label to a new observation x_i based on the majority class among training samples in the corresponding region:

$$\hat{y}_i = \arg\max_c \hat{p}(c \mid R_j)$$

To construct the tree, we must select an optimality criterion that guides how splits are made at each node. A greedy algorithm is typically used to choose the split that optimises this criterion locally. Among the most common criteria are the *Gini index* and *Shannon entropy* (sometimes referred to as Shannon–Jensen entropy). In this work, we focus on entropy, which is also the default in many implementations.

The entropy for a region R_j is defined as:

$$D_j = -\sum_{c=1}^{K} \hat{p}(c \mid R_j) \log \hat{p}(c \mid R_j)$$

where $\hat{p}(c \mid R_j)$ is the proportion of samples in region R_j that belong to class c, and K is the total number of classes. The overall entropy of a split is the weighted sum of the entropies of its child nodes. This criterion reaches its minimum value of 0 when a split produces two nodes containing only one class each and is thus said to be *pure*.

As mentioned before, classification trees can handle both discrete and continuous variables. However, when looking for the best split in any discrete variable having K possible categories, we'd have to consider at most $2^K - 1$ combinations for each node, which for a reasonable $K = 5$ would get $2^5 - 1 = 63$ different splits to check. On the other hand, any continuous variable requires checking at most $N - 1$ possible splits with $N >> K$. We thus only consider classification trees with continuous features, which can be considered an upper bound of the workload.

The algorithm 1 explains the training procedure. The maximum depth and the minimum number of samples arguments are introduced to reduce the risk of overfitting.

The main problem with decision trees is that they tend to have a high variance. Random forest helps by leaving the bias unchanged and drastically reducing the variance. The algorithm trains many trees and introduces

Algorithm 1 Classification Decision Tree

function TRAIN_TREE
  Let M_d be the maximum depth of the tree, M_s the minimum number of samples to split a node
  if depth $\geq M_d$ **or** $|D| < M_s$ **then**
    return Leaf node with majority class in y
  end if
  parent_entropy $\leftarrow$ entropy(y)
  best_entropy $\leftarrow +\infty$
  best_feature $\leftarrow \emptyset$, best_threshold $\leftarrow \emptyset$
  for each predictor m in D **do**
    Sort m (and reorder y accordingly)
    for each i in $1, \ldots, N-1$ **do**
      avg $\leftarrow (x_i + x_{i+1})/2$
      Initialize left_node $\leftarrow \emptyset$, right_node $\leftarrow \emptyset$
      for each sample j in $1, \ldots, N$ **do**
        if $x_j <$ avg **then**
          Add y_j to left_node
        else
          Add y_j to right_node
        end if
      end for
      current_entropy $\leftarrow$ entropy(left_node, right_node)
      if current_entropy $<$ best_entropy **then**
        best_entropy $\leftarrow$ current_entropy
        best_feature $\leftarrow m$
        best_threshold $\leftarrow$ avg
      end if
    end for
  end for
  if best_entropy $<$ parent_entropy **then**
    Split D into left and right subsets using best_feature and best_threshold
    Recursively call TRAIN_TREE(left subset, y_{left}, depth + 1)
    Recursively call TRAIN_TREE(right subset, y_{right}, depth + 1)
  else
    return Leaf node with majority class in y
  end if
end function

stochasticity in the process to decorrelate as much as possible each estimator and thus minimise the variance. Algorithm 2 explains our implementation. In the original algorithm, the author proposes to use bootstrapping to train each tree. However, when using large amounts of data, this often becomes counterproductive as many observations can be redundant in terms of information. Thus, we propose to simply train on a portion of the dataset, which mostly retains the same statistics of the total training set, with an average difference across features and subsets of 0.0017 for the mean and 0.0012 for the standard deviation in the medium-sized dataset of 100 thousand rows. To further decorrelate the trees, we also use the standard procedure of limiting the choice of features to consider for the best split to a subset.

Algorithm 2 Classification Random Forest

1: **function** TRAIN_FOREST
2: Let N be the number of trees
3: **for** each tree n in $1, \ldots, N$ **do**
4: $\mathrm{S} \leftarrow \mathrm{sample(D)}$
5: $\mathrm{Tree} \leftarrow \mathrm{Train\_tree}$ on S and use at most m features at each split (with $m < M$)
6: $\mathrm{Forest_n} \leftarrow \mathrm{Tree}$
7: **end for**
8: **end function**

C. OpenMP

We apply parallelism at the level of individual decision trees, since forest-level parallelisation is already efficiently handled using MPI across nodes. Within each tree, several opportunities for parallelisation exist. However, we focus specifically on parallelising the search for the best split among numerical variables, as it is the most computationally intensive step, taking approximately 97% of the time. The following most expensive operations are the sorting of the variable and entropy computation, which together roughly represent the 2% of the total time.

Parallelising over features was considered but ultimately not used, as the number of features is typically smaller than the number of available threads. This would lead to thread underutilization and inefficient resource usage. Additionally, we experimented with parallelising the entropy computation, but observed degraded performance due to the overhead introduced by fine-grained parallelism. Our implementation uses merge sort to sort feature values at each node. While merge sort is efficient, it is not straightforward to parallelise, and therefore, we leave it for future work.

All other components of the training and inference pipeline—such as tree traversal, node creation, or inference—are computationally negligible. For instance, even on the largest dataset considered, inference time is less than one second and thus not a bottleneck.

Thus, the parallelisation consists of using

```
#pragma omp parallel for
    private(left_node, right_node)
```

on top of the for loop at line 11 in algorithm 1. The only data dependency here is due to the comparison $\mathrm{current\_entropy}$ and $\mathrm{best\_entropy}$ at line 22 which is solved using

```
#pragma omp critical
```

so that only one thread at a time can access that section. Note, however, that this is not the only solution. In fact, we also attempted to avoid the critical section by creating a local $\mathrm{best\_split}$ array for each thread and then reducing it to get the global best threshold and variable for the split. However, as it will be shown in the experiments section, this approach degraded the performance. We hypothesise that this is because, in most cases, the best split is found pretty soon, and thus the number of times the best split variable is updated is lower compared to the version without the critical section that has many local variables.

D. MPI

The MPI implementation distributes trees across all processes while process 0 handles additional orchestration responsibilities, such as determining the amount of trees each process will have to train (later used to know how many arrays of predictions it receives), sharing the shape of the dataset and other MPI communication related information. Process 0 handles data distribution and result aggregation in addition to training its assigned subset of trees, ensuring all available processes contribute to the computational workload. In traditional MPI implementations, process 0 works as the coordinator, but given that every worker process works with the same amount of data with no need for coordination during the training algorithm we decided to make process 0 both coordinator and worker. If we were to use process 0 as a lone coordinator, our efficiency would drop significantly since it would be idle for the majority of the training algorithm, wasting computational resources.

Data Distribution Strategy: Initially, we implemented a centralized sampling approach where process 0 performed data sampling for all processes and distributed the subsets individually. However, this approach proved highly inefficient as communication costs increased with the number of processes. Our current implementation adopts a broadcasting strategy: process 0 reads and broadcasts the complete dataset to all processes using `MPI_Bcast`, after which each process independently performs stratified splitting and sampling from the training set using a process-specific seed (base seed + process rank). While this approach temporarily increases memory usage—as each process holds the complete dataset during initialization—the memory is freed immediately after sampling, and the performance gains significantly outweigh the temporary memory overhead.

Load Balancing: Trees are distributed evenly across all processes to maximize computational efficiency. Each process receives $\lfloor N/P \rfloor$ trees, where N is the total number of trees and P is the number of processes. Any remaining trees (when N is not divisible by P) are distributed one each to the first $N \bmod P$ processes. This distribution strategy ensures balanced workloads with at most a one-tree difference between any two processes, while maintaining deterministic and reproducible tree assignments.

Communication Pattern: The implementation follows a point-to-point communication pattern using `MPI_Send` and `MPI_Recv`. After training, each process performs inference on the complete test set using its assigned trees, then sends predictions back to process 0 using `MPI_Send` and `MPI_Recv`. Process 0 aggregates all predictions using majority voting to produce final classifications. This approach minimizes communication overhead by requiring only one message per process, while ensuring all processes use identical test data for consistent aggregation.

Data Dependencies: The random forest algorithm exhibits no data dependencies during the tree construction phase, making it an ideal candidate for parallel processing. Each tree is built independently using the same training dataset but with different random subsets of features and samples. This independence eliminates the need for inter-process communication or synchronization during computation, allowing for embarrassingly parallel execution with minimal coordination overhead.

Timings: The start time is taken right after process 0 reads the dataset. Disk I/O is excluded since the program is not optimized for high-performance disk I/O, as discussed in Chapter 2.7 Parallel Program Design - page 65 [2]. No `MPI_Barrier` was used because process 0 naturally becomes the bottleneck due to doing the very first action of our algorithm (determining the amount of classes in the dataset) and the last (aggregating predictions and counting the votes per class).

E. OpenMP and MPI

The hybrid implementation combines the forest-level parallelism of MPI with the tree-level parallelism of OpenMP, creating a two-tier parallel structure.

Threading Model: We use `MPI_Init_thread` with the `MPI_THREAD_FUNNELED` level, which allows multiple threads within each process while restricting MPI calls to the main thread only [3]. As stated in the official documentation, this specifies that "The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread)." This design choice aligns with our implementation where OpenMP parallelizes the computationally intensive best-split search within each tree, while MPI handles inter-process coordination through the main thread.

Nested Parallelism Structure: The parallelization hierarchy operates as follows:

- **Process level (MPI)**: Trees are distributed across processes, with each process training a subset of the forest.
- **Thread level (OpenMP)**: Within each process, multiple threads collaborate on training individual trees by parallelizing the best-split search across potential split points using the same `#pragma omp parallel for` directive described in the OpenMP section

Resource Management: This hybrid approach enables efficient utilization of distributed memory architectures. MPI scales across nodes while OpenMP exploits shared memory parallelism within each node. The combination allows the algorithm to leverage both inter-node and intra-node parallelism effectively.

Performance Considerations: The hybrid model is particularly well-suited for random forest algorithms due to their computational characteristics. Pure MPI creates excessive processes on multi-core nodes, leading to unnecessary memory overhead from duplicating the training dataset across processes and potential communication bottlenecks during the minimal coordination phases. Pure OpenMP would restrict execution to single-node parallelism, severely limiting scalability for large forests that benefit from distributed computing resources. Our hybrid approach leverages MPI for distributing

trees across nodes (exploiting the algorithm's embarrassingly parallel nature) while using OpenMP for intra-node parallelization of individual tree construction. This design maximizes resource utilization by avoiding idle processes (as discussed with our process 0 coordination strategy) and scales efficiently across distributed systems without the memory duplication penalties of pure MPI implementations.

Memory Efficiency: Each process maintains its own data subset after the initial broadcast phase, and threads within each process share this memory space during tree training. This design minimizes memory overhead while enabling fine-grained parallelism where it provides the most benefit. The same data dependency resolution using `#pragma omp critical` applies as described in the OpenMP section, ensuring thread-safe updates to the best split variables.

III. Experiments

The experiments were ran on the HPC Cluster of the University of Trento. Its main specifics are the following:

- 142 CPU calculation nodes for a total of 7.674 cores
- 10 GPU calculation nodes for a total of 48.128 CUDA cores
- 2 frontend nodes (head node)
- Ram: 65TB ram
- All nodes are interconnected with 10Gb / s network and some have Infiniband connectivity, while others have Omni-Path connectivity (both low-latency and high-speed connectivity for MPI communication).
- The operating system is Linux CentOS 7, while the cluster management software is PBS Professional, currently at version 19.1.

For the experiments, we used the following setup:

- Data:
  - A standard train-test of 80% and 20% split using stratification by target variable to ensure the classes' distribution consistency between partitions
  - We test our solution using datasets of 15 columns, of which one is the target variable with 3 classes, and varying rows between 50 and 150 thousand, equivalent to a matrix size approximately between 0.75M and 2.25M; Note: we tried to use a larger dataset of 200 thousand observations, but the sequential job would last more than 6 hours and it would thus inevitably be killed since we only had access to the short queue on the HPC cluster
  - To generate the data, we use the popular machine learning library *scikit-learn*, specifically the utility function `datasets.make_classification`. Each class is modelled as a Gaussian distribution with a unique mean vector and a shared diagonal covariance matrix (i.e., no correlation between features within a cluster).
- Training:
  - After some testing, we decided to train a total of 100 decision trees as the classification performance was pretty stable with accuracy, precision and recall around 83%
  - The maximum tree depth and the `train_tree_proportion` parameter (i.e., the proportion of the available data each tree is trained on) are selected to achieve reasonably good classification metrics and set to 10 and 75% respectively. These parameters can be further fine-tuned for tasks with higher sensitivity to model performance.
  - The minimum number of samples required to split an internal node is set to 200. However, note this parameter likely doesn't have an impact on time metrics since the trees are grown to have a maximum depth of 10 and with large datasets, leaves rarely end up having less than 200 samples
  - The effective training dataset size for each tree is $(N \cdot 0.8 \cdot 0.75) \times M$, accounting for both the 80% split used for training and the 75% subsampling per tree. This effective size corresponds to the one reported in the results figures.
- Benchmarking:
  - Benchmarking: the speedup and efficiency are computed using the total time spent between training and inference of the random forest, even though the latter is negligible and it doesn't even account for 1% of the total time.

A. OpenMP

Figures 1 and 2 show the efficiency and speedup achieved using OpenMP. While the speedup always increases for the medium and the large dataset, this is not the case for the small one, which suggests overhead communication with 64 threads. This evidence is further reinforced when considering the efficiency chart, which shows that with 64 threads, the efficiency reduces

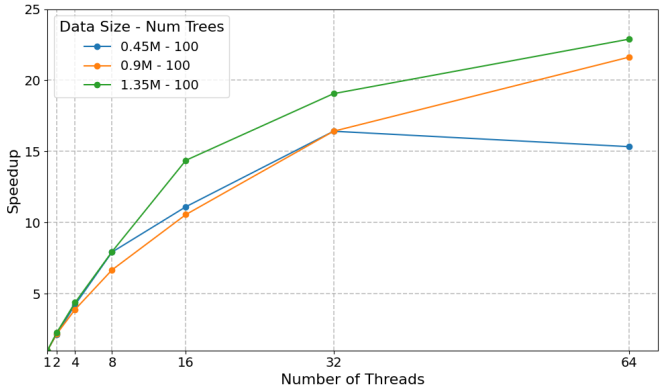

Fig. 1. Speedup for different numbers of threads and data size

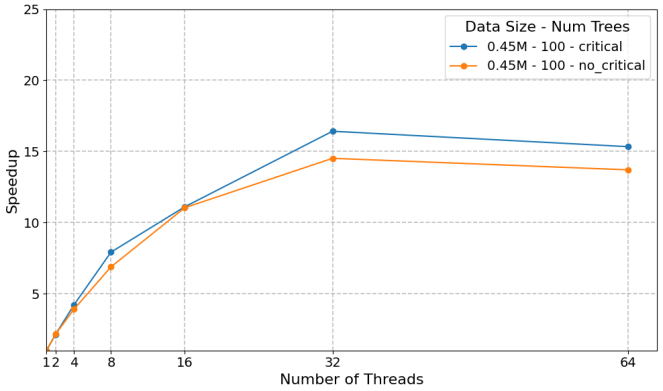

Fig. 3. Speedup for different numbers of threads using small data size and without OpenMP critical section

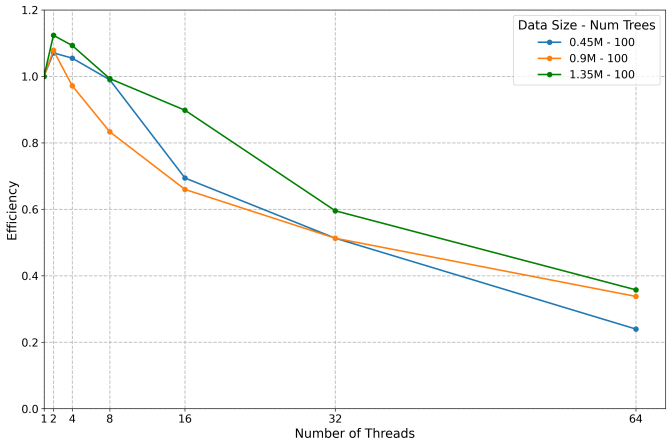

Fig. 2. Efficiency for different numbers of threads and data size

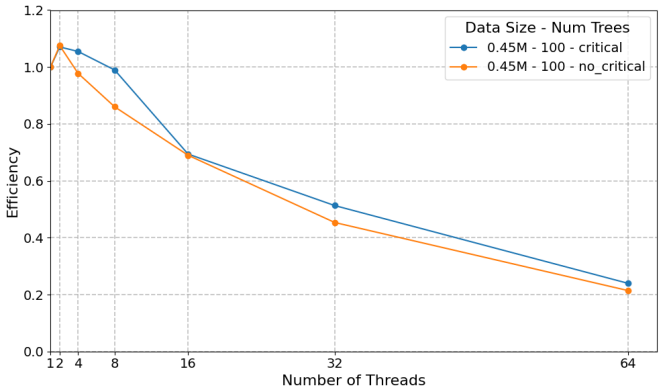

Fig. 4. Efficiency for different numbers of threads using small data size and without OpenMP critical section

drastically for all datasets, especially the small one. Given the cluster's architecture of 142 CPU nodes with 7,674 total cores (approximately 54 cores per node), the performance degradation at 64 threads can be attributed to thread oversubscription. When running 64 threads on a node with only 54 physical cores, the operating system must introduce context switching overhead and thus resource contention. This oversubscription effect is particularly pronounced for the small dataset, where the reduced computational workload per thread makes the scheduling overhead proportionally more significant, even causing speedup to decrease from 32 to 64 threads ($16.41\times$ to $15.33\times$ speedup). In the methodological section, we mentioned that we decided to use the critical section for the best_split variable because, as the figures 3 and 4 show, the performance decreases when using the other strategy.

B. MPI

Figure 5 highlights how all datasets have their speedup increase as the number of processes increases. In particular, the medium dataset achieved super-linear efficiency at 4 and 8 processes with a $4.05\times$ and $8.30\times$ speedup respecitvely. We assume this is due to better cache utilization as each process works with smaller data portions, but not as small as the smaller dataset, where the reduced computational workload per process makes coordination overhead more significant, preventing super-linear efficiency despite similar data locality benefits.

Figure 6 shows the efficiency of the MPI implementation. The small dataset worsens significantly from 4 processes to 8, stays constant and drops again at 64 processes. The medium dataset shows that the increasing processes keeps a high efficiency up until 64 processes, where there is a huge drop from 0.89 at 32 processes to

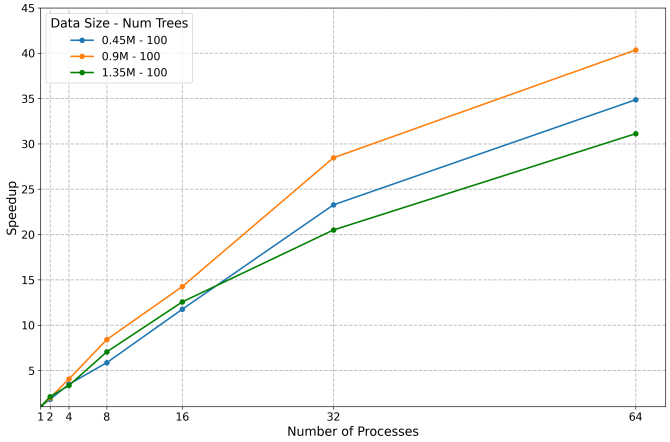

Fig. 5. MPI speedup for different numbers of processes

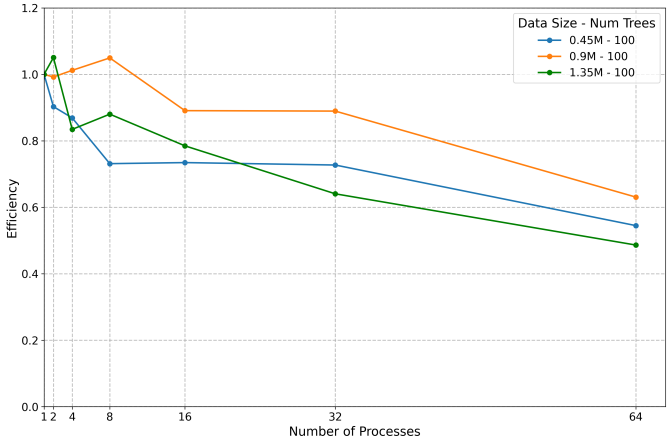

Fig. 6. MPI efficiency for different numbers of processes

0.63 at 64 processes. The large dataset has a large drop of efficiency from 16 to 64 processes, starting from 0.88 at 8 processes and reaching 0.48 at 64 processes.

The common pattern among all datasets is that efficiency drops when going from 32 to 64 processes. Using a hybrid OpenMP and MPI version will allow us to utilize the extra CPU cores through threading rather than creating more processes that would further reduce efficiency.

C. Hybrid OpenMP and MPI

Given the results obtained in the OpenMP portion, we decided to avoid using a 64 threads with 64 processes combination. 64 threads already proved to be highly inefficient, furthermore the PBS queue times proved to be extremely high to make it easy to test. The doubly parallel implementation demonstrates significant performance improvements that scale with both problem size and computational resources. For the small dataset, tables I and II, we achieved maximum speedups of $379.79\times$ using 16 threads and 64 processes, though efficiency generally decreased as the number of processes increased due to communication overhead. For the medium dataset, the results in tables III and IV, showed the most consistent scaling behavior, reaching impressive speedups of up to $1070.26\times$ with 32 threads and 64 processes while maintaining reasonable efficiency values above 0.5 for most configurations with moderate thread counts. The large dataset exhibited in tables V and VI displayed the most robust scaling characteristics at 64 threads. It achieved a maximum speedups of $663.94\times$ with 32 threads and 64 processes. Notably, it maintained better efficiency at higher process counts compared to the small dataset, but not better than the medium one. Across all dataset sizes, we observed that configurations with 8-16 threads generally provided the optimal balance between speedup and efficiency, with diminishing returns and increased overhead becoming apparent at 32-64 processes. The results indicate that our algorithm scales effectively with problem size, as larger datasets better utilize the available parallel resources and achieve higher absolute speedups while maintaining decently stable efficiency metrics.

TABLE I
SPEEDUP RESULTS FOR SMALL DATASET

T\P	1	2	4	8	16	32	64
1	1.00	1.81	3.48	5.85	11.75	23.27	34.87
2	1.70	1.66	6.97	8.83	21.87	37.07	59.58
4	1.49	4.91	9.02	25.56	40.48	75.01	115.09
8	7.98	7.52	17.62	23.53	67.15	153.62	239.49
16	12.03	21.98	42.31	32.17	139.92	202.85	379.79
32	17.34	14.28	65.02	94.06	106.40	295.00	202.70
64	4.98	15.19	57.67	73.52	181.45	43.33	–

TABLE II
EFFICIENCY RESULTS FOR SMALL DATASET

T\P	1	2	4	8	16	32	64
1	1.00	0.90	0.87	0.73	0.73	0.73	0.55
2	0.85	0.41	0.87	0.55	0.68	0.58	0.47
4	0.37	0.61	0.56	0.80	0.63	0.59	0.45
8	1.00	0.47	0.55	0.37	0.53	0.60	0.47
16	0.75	0.69	0.66	0.25	0.55	0.40	0.37
32	0.54	0.22	0.51	0.37	0.21	0.29	0.10
64	0.08	0.12	0.23	0.14	0.18	0.02	–

TABLE III
SPEEDUP RESULTS FOR MEDIUM DATASET

T\P	1	2	4	8	16	32	64
1	1.00	1.98	4.05	8.40	14.25	28.46	40.36
2	1.89	4.25	8.92	16.66	25.36	50.75	76.88
4	2.79	7.38	10.28	34.65	37.56	114.15	145.70
8	9.16	10.11	29.65	59.97	104.70	177.04	286.64
16	8.17	29.65	44.31	108.87	126.39	372.24	620.86
32	22.37	40.79	67.99	195.33	310.27	684.49	1070.26
64	6.16	20.66	55.19	211.29	124.56	121.11	–

TABLE IV
EFFICIENCY RESULTS FOR MEDIUM DATASET

T\P	1	2	4	8	16	32	64
1	1.00	0.99	1.01	1.05	0.89	0.89	0.63
2	0.95	1.06	1.11	1.04	0.79	0.79	0.60
4	0.70	0.92	0.64	1.08	0.59	0.89	0.57
8	1.14	0.63	0.93	0.94	0.82	0.69	0.56
16	0.51	0.93	0.69	0.85	0.49	0.73	0.61
32	0.70	0.64	0.53	0.76	0.61	0.67	0.52
64	0.10	0.16	0.22	0.41	0.12	0.06	–

TABLE V
SPEEDUP RESULTS FOR LARGE DATASET

T\P	1	2	4	8	16	32	64
1	1.00	2.10	3.34	7.04	12.56	20.50	31.12
2	1.81	3.54	6.44	13.01	27.67	38.22	64.17
4	2.55	6.67	13.66	21.07	47.58	50.49	83.78
8	6.60	9.28	28.50	48.84	88.17	139.16	220.05
16	14.18	23.32	49.90	70.73	108.83	196.14	416.38
32	21.80	40.70	58.20	171.98	336.68	482.20	663.94
64	16.15	52.07	124.32	240.67	421.03	541.58	–

TABLE VI
EFFICIENCY RESULTS FOR LARGE DATASET

T\P	1	2	4	8	16	32	64
1	1.00	1.05	0.83	0.88	0.78	0.64	0.49
2	0.90	0.88	0.81	0.81	0.86	0.60	0.50
4	0.64	0.83	0.85	0.66	0.74	0.39	0.33
8	0.82	0.58	0.89	0.76	0.69	0.54	0.43
16	0.89	0.73	0.78	0.55	0.43	0.38	0.41
32	0.68	0.64	0.45	0.67	0.66	0.47	0.32
64	0.25	0.41	0.49	0.47	0.41	0.26	–

IV. CONCLUSIONS

Our work aims to analyze the Random Forest algorithm performance under different parallelization techniques, leveraging both shared memory (OpenMP) and distributed memory (MPI) paradigms, as well as their hybrid combination. Due to the embarrassingly parallel nature of the Random Forest problem, where individual trees can be trained independently without inter-dependencies, all three parallelization approaches demonstrated substantial performance improvements over the sequential baseline.

The OpenMP implementation showed consistent speedup improvements across all thread counts, maintaining scalability up to 64 threads. However, the efficiency started degrading from 16 threads onward, particularly pronounced for smaller datasets where the reduced computational workload per thread makes the overhead proportionally more significant.

The MPI implementation exhibited different scaling characteristics, showing suboptimal performance at low process counts due to communication overhead dominating the computational workload, but achieving superior results at higher process counts. For medium-sized datasets, we observed super-linear speedup for 4 and 8 processes, likely attributable to improved cache utilization as each process operates on smaller, more cache-friendly data portions.

The hybrid implementation demonstrated the most performant approach with the highest absolute speedup results, reaching up to $1070.26\times$ for the medium dataset with 32 threads and 64 processes. This approach effectively combines the strengths of both paradigms: MPI exploits the algorithm's embarrassingly parallel nature across nodes, while OpenMP leverages intra-node parallelism for computationally intensive split-finding operations. However, efficiency metrics indicate diminishing returns at higher resource counts due to the substantial amount of computational resources leveraged.

Several opportunities exist for further performance optimization. Currently, variable sorting represents a sequential bottleneck in our implementation, consuming approximately 2% of the total execution time. Our current approach uses merge sort, which, while stable and predictable, involves complex data dependencies when parallelized due to its divide-and-conquer structure requiring careful synchronization during the merge phases. Switching to quicksort would facilitate parallelization while maintaining the same $O(n \log n)$ average time complexity, as quicksort's partition-based approach allows for a more natural parallel implementation. This optimization should further reduce tree construction time, particularly when the workload is sufficiently high to amortize the parallelization overhead costs associated with thread coordination and data distribution.

Another promising approach involves leveraging GPU acceleration, although Random Forest algorithms face several inherent challenges when adapted to GPU architectures. The main criticalities include:

- *irregular memory access patterns* during tree traversal and feature evaluation, which conflict with GPU requirements for coalesced memory access and result in poor memory bandwidth utilization;
- *load balancing challenges* as trees of varying depths and complexities create uneven computational workloads across GPU cores, leading to resource underutilization;

Despite these architectural challenges, specialized frameworks such as cuML have implemented random forests with "experimental multi-node multi-GPU via Dask".

V. Weak Scalability BONUS

References

[1] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[2] P. Pacheco, *An Introduction to Parallel Programming*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[3] MPI Forum, "MPI_Init_thread – Initialize the MPI execution environment with thread support," https://www.mpich.org/static/docs/v3.1/www3/MPI_Init_thread.html, 2013, accessed: 2025-06-04.