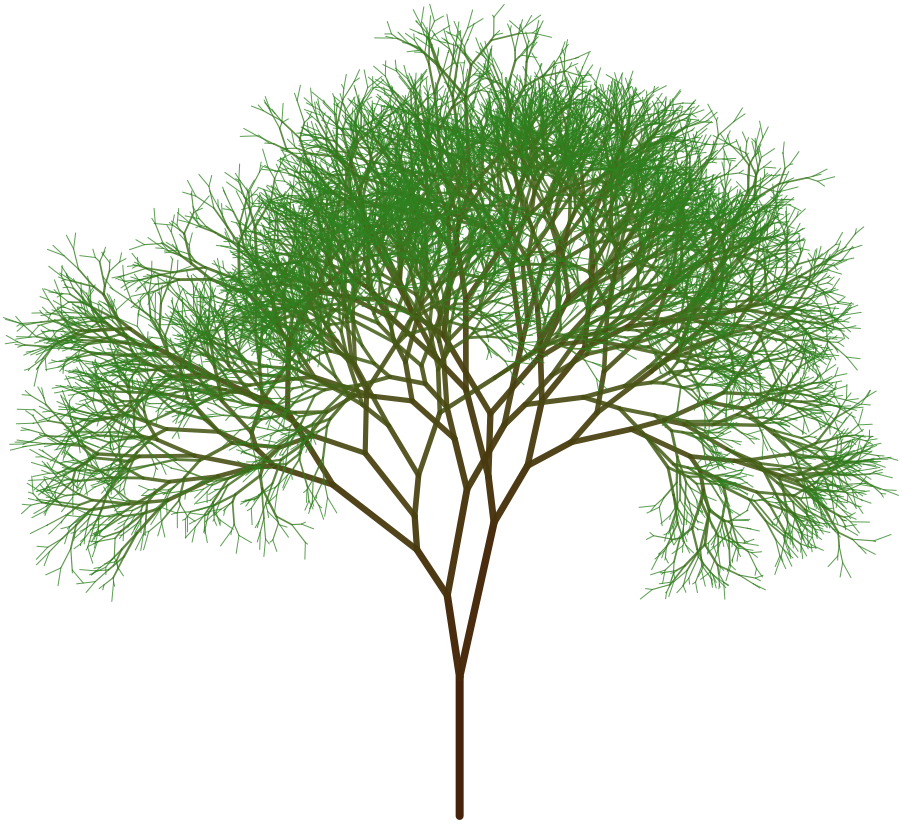
Random Forest Classifier

A fast C++ implementation



University of Pisa
Master in Computer Science - Artificial Intelligence
Parallel and Distributed Systems - Massimo Torquati

Benedetti Gabriele
23.10.2025 - 602202

Contents

1 Introduction

The Random Forest Classifier[1] is a Machine Learning model that leverages the Random Forest algorithm to perform accurate predictions on input data. This method, developed by Leo Breiman and Adele Cutler, is based on the construction of multiple decision trees, whose outputs are aggregated through a majority vote in order to produce a robust and reliable prediction.

This report assumes that the reader is already familiar with the Random Forest Classifier model, or at least has a basic understanding of how Random Forests work. Instead, the focus of this paper will be on the specific architectural and design choices made during the development.

The model has been fully implemented from scratch in C++, as no online resources were found that were both competitive with *scikit-learn*[2] and easy to use. Several available implementations were either incomplete, overly complex, or lacked access to the source code. For this reason, the development started from a very basic version of the algorithm, which was then progressively extended and refined step by step.

1.1 Design

To ensure familiarity and ease of use, the design of the model is heavily inspired by the popular *scikit-learn* library in Python. This philosophy promotes a clean and predictable interface, centered around three primary public methods: `fit`, `predict`, and `score` (to get accuracy and F1-score).

1.2 General Notes

The main classes — `RandomForestClassifer` and `DecisionTreeClassifier` — are located in the `src/` folder. The project uses the following libraries:

- cxxopts[3] (for command-line options)
- pdqsort[4] (a better alternative of `std::sort`)
- pcg32[5] (for a strong random number generation)

The implementation of the splitter with OpenMP and FastFlow, used by the Random Forest classifier to find the optimal split threshold, can be found in `include/splitters/`.

For the prediction phase, there are some **demo implementations** in `include/predictionFF/` that use FastFlow to make predictions for each test sample. These are **only experimental** and were **not used in the final version** of the project.

Following some useful commands to run in the main directory of the project:

```
make all        # Build all the project (both shared-memory and MPI version)
make tests      # Run all the shared-memory tests
make tests-mpi  # Run all the MPI tests
make help       # Check available commands
```

2 Implementation

This chapter discusses the main issues and solutions for the RandomForestClassifier and DecisionTreeClassifier, with effectiveness demonstrated through timing comparisons (pre- vs post-optimization).

A dedicated **Timer** class was implemented to record:

- number of invocations of a code block,
- total accumulated execution time,
- maximum time of a single invocation.

As the class is not thread-safe, measurements were performed on a single decision tree. The focus was first on minimizing training time of one tree, before extending parallelization to the whole forest. All experiments were run in debug mode (no compiler optimizations), on the SUSY dataset (70% training, 30% test), with a fixed random seed and bootstrap sampling enabled.

2.1 RandomForestClassifier: A Slow Start

The core mechanism of a decision tree lies in finding the best splitting criterion for its nodes. This process relies on two fundamental steps:

- for each selected feature, an optimal threshold is computed that maximizes the "purity" (e.g., Gini[6] index) of the resulting groups (`compute_threshold` function)
- the one with the best threshold is then partitioned (`split_left_right` function)

In code:

```
for (int f : selected_features) {
    auto [threshold, impurity] = compute_treshold(X, y, feature);

    if (impurity < best_impurity) {
      auto [left_X, right_X] = split_left_right(X, y, threshold, f);

      if (!left_X.empty() && !right_X.empty()) {
          best_impurity = impurity;
          best_feature = f;
          best_threshold = threshold;
      }
    }
}
```

The first implementation, however, had several critical issues.

For the `compute_threshold` function:

- **Copy overhead**: creating *feature_label_pair* vector preserves feature–label correspondence but introduces prohibitive memory and runtime costs on large datasets.
- **Significant cache misses**: since data is stored as "samples × features", operations like

  ```
  feature_label_pairs.emplace_back(X[i][f], y[i]);
  ```

  cause frequent cache misses, slowing execution.

- **Redundant impurity checks**: sorted data often yields consecutive identical impurity values, leading to unnecessary recomputations. An adaptive offset can mitigate this by skipping negligible differences.

For the `split_left_right` function:

- **Recursive submatrix creation**: repeatedly generating dataset copies at each tree depth is highly inefficient.
- **Splitting cost**: although each split is $O(n)$, the function is invoked many times. With a small adjustment, it is possible to exploit binary search ($O(\lg n)$) to optimize performance.

The following table shows the execution times of the two methods, highlighting the potential speed-up:

Function	Function Calls	Total Time	Max Single Call (s)
split_left_right	728086	59.3578s	1.3945
compute_threshold	1591836	7m 4.8506s	5.5948

Table 1 - Execution times between the two core functions

```
Training Time: 9m 8.495s
```

Execution times were prohibitively high, mainly due to the massive number of function calls (e.g., `compute_threshold` was invoked over 1.5M times) and the poor scalability of each function. Single calls reached over 5 seconds for `compute_threshold` and 1.4 seconds for `split_left_right`, which is unacceptable for simple split operations.

On smaller datasets like Iris, training times remained reasonable, and both accuracy and F1-score matched scikit-learn, confirming the correctness of the implementation despite inefficiencies.

2.2 Performance Improvements: Index based approach

To reduce memory overhead, the initial implementation was shifted to an index-based approach. Instead of duplicating parts of the dataset, arrays of integer indices were used to reference the features directly. This change allowed for more efficient sorting and partitioning, as operations were performed on lightweight integers, and enabled the use of binary search in the `split_left_right` function.

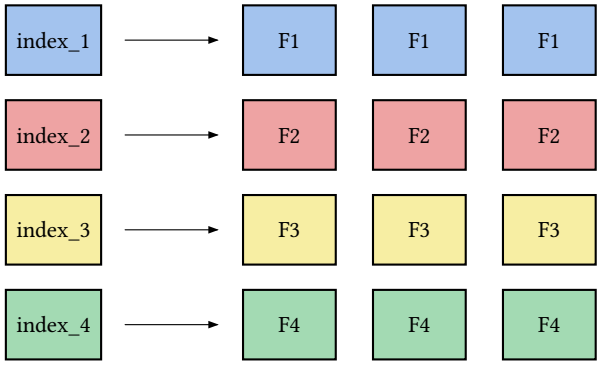


Diagram 1 - Example of indices that points to a 3×4 matrix

In practice, operations such as moving, removing or sorting indices are performed in bulk on entire rows and columns. This approach is much more efficient, as the original matrix is never modified —values are only read when necessary.

Further minor improvements included:

- **Adaptive Offset:** In the `compute_threshold` method, the step size for threshold searching was dynamically adjusted based on impurity changes to optimize search efficiency.
- **Minimum sample count:** A *min_samples_ratio* hyperparameter was introduced to prevent imbalanced splits and overfitting by ensuring child nodes met a minimum size requirement relative to their parent.

2.2.1 Cache-Miss Bottleneck

While the index-based approach reduced memory usage, it introduced a significant number of cache misses due to random memory access patterns. This negatively impacted the `compute_threshold` function, which became the main performance bottleneck.

The initial performance with the index-based approach was:

Function	Function Calls	Total Time	Max Single Call (s)
split_left_right	177532	0.3351s	0.00435
compute_threshold	288540	5m 40.278s	17.828

Table 2 - Execution times with the updated methods

```
Training Time: 5m 45.477s
```

Analysis revealed that sorting the indices was responsible for the vast majority of the execution time within `compute_threshold`.

Code Block	Function Calls	Total Time	Max Single Call (s)
threshold: sorting	288540	5m 34.280s	17.716
threshold: main	288540	5.225s	0.160

Table 3 - Execution times of the main components of `compute_threshold`

To mitigate this, the *feature_label_pair* structure was temporarily reintroduced. This vector made memory access contiguous and cache-friendly during the threshold computation, leading to a major performance boost, although it prevented the use of binary search in `split_left_right`.

Code Block	Function Calls	Total Time	Max Single Call (s)
threshold: feature_label	288540	30.625s	0.844
threshold: sorting	288540	1m 2.483s	1.771
threshold: main	288540	1.349s	0.160
split_left_right	288540	19.807s	0.977

Table 4 - Execution times using the `feature_label_pair` vector

```
Training Time: 2m 1.408s
```

2.2.2 Sorting Algorithm

The most significant improvement came from changing the sorting algorithm. Initially **pdqsort** was considered, however it was replaced with **Radix Sort** (adapted for floating-point values), ending with a sorting time drastically reduced. This allowed for reverting to the pure index-based method, which restored the highly efficient binary search in `split_left_right`.

2.2.3 Results

The combination of the index-based approach and Radix Sort yielded the best results.

Code Block	Function Calls	Total Time	Max Single Call (s)
threshold: sorting	288540	25.906s	0.735
threshold: main	288540	4.673s	0.136
split_left_right	288540	0.302s	0.003

Table 5 - Execution times using Radix Sort

```
Training Time: 35.131s
```

With the addition of compiler optimization flags, the final training time is the follow:

```
Training Time: 9.463s
```

This optimized implementation combines high performance with low memory consumption. The index-based technique requires only about **0.75 GB** of memory for a dataset with 5,000,000 samples running on 40 threads, demonstrating its efficiency and scalability.

2.3 Can we do better?

The index-based solution currently works well but suffers from a major drawback: **cache misses**, as explained in the previous sections.

This issue inevitably leads to longer training times, since most matrix accesses require values to be reloaded from memory and placed into cache. Of course, the observed overhead is not entirely due to the training phase alone—other variables are involved—but it is important to recognize that cache misses represent a significant performance penalty we are currently paying.

2.3.1 Min-Max Quantization

Random Forests do not require extreme precision for split selection, since variance reduction arises from the ensemble effect. Thus, a compressed data representation is sufficient. Min-max quantization[7] addresses this by approximating floating-point values, which can later be reconstructed when needed.

The quantized value is computed as:

$$\mathrm{q} = \left\lfloor \frac{\text{val - min}}{\text{max - min}} \times (L-1) \right\rceil \tag{1}$$

Where:
- *min*: global minimum value
- *max*: global maximum value
- L: number of levels (in our case, the maximum representable value of the chosen compressed data type)

To implement this technique in C++, it is possible to adopt the **`uint8_t`** type: an unsigned integer requiring only 8 bits, with a representable range of $[0, 255]$. In this case, $L = 255$.

The original (approximated) value can be reconstructed as:

$$\mathrm{val}_{\mathrm{approx}} = \frac{q}{L-1} \times (\max - \min) + \min \tag{2}$$

For example, with a dataset of 1B samples and 20 features:

$$\begin{gathered} \text{memory\_usage\_float} = \text{n\_samples} \times \text{n\_features} \times \text{sizeof(float)} = \\ 1000000000 \times 18 \times 4 \text{ bytes} \times 40 \approx 74.51 \text{ Gb} \\ \text{memory\_usage\_uint8\_t} = \text{n\_samples} \times \text{n\_features} \times \text{sizeof(uint8\_t)} = \\ 1000000000 \times 18 \times 1 \text{ bytes} \times 40 \approx 18.63 \text{ Gb} \end{gathered} \tag{3}$$

This represents of course a substantial reduction.

To handle the quantized data, a dedicated **TrainMatrix** class was developed.

2.4 Performance Improvements: Non-Index-Based Approach

To achieve further improvements, a first attempt involved was removing index arrays and assigning each decision tree its own working matrix. This enables in-place modifications without intermediate lookups, but at the cost of prohibitive memory usage, since consumption now scales with both samples and features.

For the SUSY dataset test case:
- 40 threads on the front-end node
- 5,000,000 samples
- 18 features

The minimum memory consumption is:

$$\begin{gathered} \text{memory\_usage} = \text{n\_samples} \times \text{n\_features} \times \text{sizeof(float)} \times \text{n\_threads} = \\ 5000000 \times 18 \times 4 \text{ bytes} \times 40 \approx 13.41 \text{ Gb} \end{gathered} \tag{4}$$

And this only for SUSY dataset! For datasets around 30 GB, memory requirements reached ≈ **1200 GB**. Quantization was applied to mitigate this, yielding significant memory savings and simplifying Radix Sort into a plain **Counting Sort** (on integers). Still, this was insufficient to make the approach viable.

On the positive side, training times per single tree dropped dramatically: with a 30 GB dataset, training required only ≈20 minutes (≈3× speedup), showing that minimizing random access to the training matrix provides major benefits.

These estimates consider only the matrix data; in practice, decision tree nodes also contribute. For example, a 30 GB dataset, the tree structure alone grew to approximately 60,279,850 nodes (which is ≈ 1Gb just for a single thread!).

2.4.1 Results

Code Block	Function Calls	Total Time	Max Single Call (s)
threshold: sorting	391468	8.812s	0.204
threshold: main	391468	2.577s	0.079
split_left_right	86380	1.911s	0.181
matrix_creation	1	2.172s	2.172

Table 6 - Execution times using Non-Index-Based Approach

```
Training Time: 17.105s
```

The improvements in the optimal split search are remarkable: sorting an array 391,468 times now takes only **8.8 seconds**, compared to the 25 seconds required by the index-based version. Also, the main for-loop now completes in **2.5 seconds**, down from 4.6 seconds in the index-based implementation.

Furthermore, cache misses were reduced by approximately 77%, which explains the substantial performance improvement.

```
perf stat -e cache-misses ./decision_tree

 Performance counter stats for './decision_tree':

     270.496.826      cache-misses
```

Finally, by enabling optimization flags, the training time was further reduced to:

```
Training Time: 2.359s
```

This approach achieved a 3x speedup, confirming that cache optimization delivers substantial benefits. Fast-math flags were not used, as they caused slight accuracy loss, though performance remained unaffected without them.

However, the method was ultimately discarded due to its excessive memory consumption. While highly effective on a single tree, any attempt at parallelization quickly exhausted available RAM, preventing full utilization of machine cores.

This naturally raises the question: is it possible to combine the strengths of both approaches —a memory-efficient index-based representation with the fast training times of the non-indexed version?

2.5 Performance Improvements: Histogram-Based Approach

Instead of scanning all features and testing every possible split, an **histogram-based**[8] strategy was introduced. Features are discretized into **256 bins** (enabled by quantization), allowing fast label counting without remapping floating-point values.

This method leverages indices while relaxing their ordering: indices need to be sorted only once, since they can be taken in any order, which improves cache locality during matrix access. Additionally, the input matrix is assumed to be in Column-Major format, since the model operates feature-wise.

The `compute_threshold` function was redesigned to build histograms for selected features.

```
for (const int idx : indices) {
    const uint8_t bin = X.getQuantized(f, idx);
    const int label = y[idx];
    if (bin_counts[bin] == 0) {
       active_bins[active_bins_size++] = bin;
    }

    histogram[bin][label]++;
    bin_counts[bin]++;
}
```

With at most 256 iterations per feature, the adaptive offset technique became unnecessary.

Finally, for data partitioning, the `split_left_right` function was updated to adopt Hoare's partitioning algorithm, providing an in-place data split around the chosen threshold.

2.5.1 Results

Code Block	Function Calls	Total Time	Max Single Call (s)
matrix_creation	1	0.414s	0.414
histogram creation	374532	6.487s	0.069
threshold: main	374532	0.220s	0.000065
split_left_right	83035	1.164s	0.069

Table 7 - Execution times of Histogram-based approach

```
Training Time: 13.597s
```

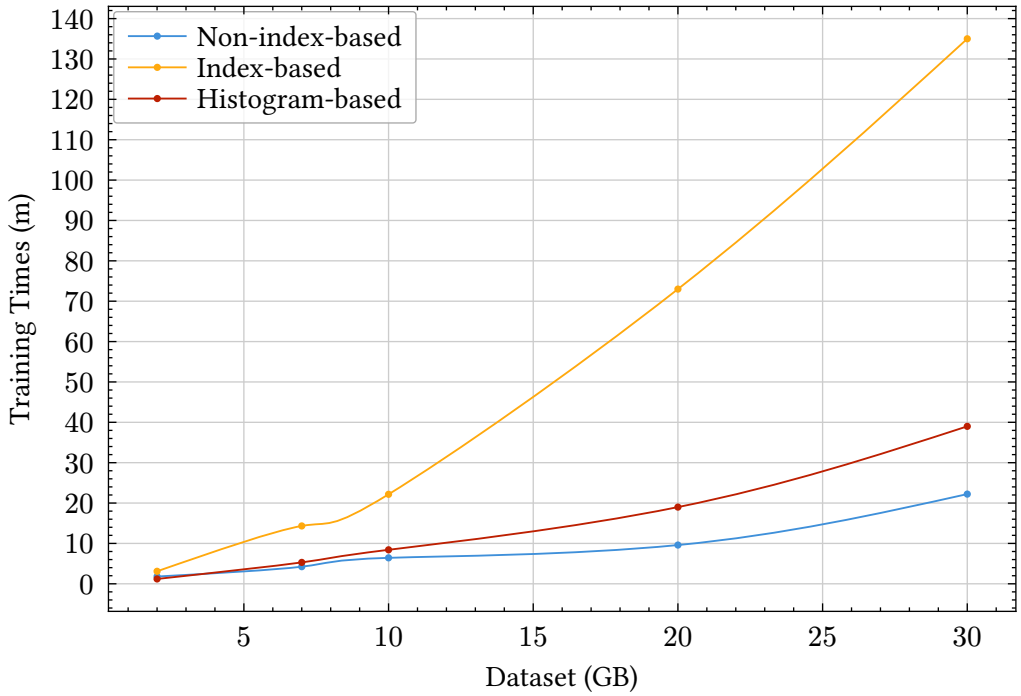
The histogram-based changes provide significant speedup:
- `compute_threshold` now only builds the histogram, eliminating sorting overhead.
- `split_left_right` benefits from pre-sorted indices, improving cache locality.
- TrainMatrix creation avoids copies, only computing the global minimum for quantization, reducing setup time.

This implementation represents the choosed version of the RandomForestClassifier for this project. With optimization flags, training time reached:

```
Training Time: 3.02s
```

Compared to previous approaches, the histogram method offers a balanced compromise between speed and memory usage. For very large datasets, splitting indices in `split_left_right` gradually reduces cache locality, slightly affecting performance despite sorted indices.

The final graph compares the three previously discussed versions.



Plot 1 - Comparison between all implemented versions

As expected, the histogram-based approach strikes a middle ground, offering a balanced compromise between performance and memory usage. For larger datasets, splitting indices in `split_left_right` gradually reduces cache locality, causing a slight performance degradation even with sorted indices.

3 Parallelization

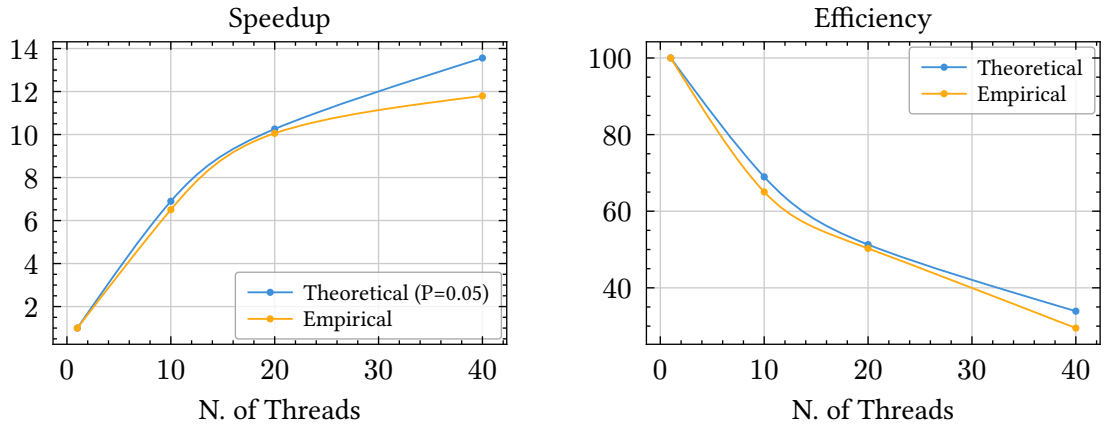
In this chapter, we analyze how the model has been parallelized, both at the tree level and within the best-split search, and evaluate the impact of these strategies on performance.

Larger datasets, such as those explored in the previous chapter, were not considered here. These require extensive parameter tuning, as they tend to generate an exponential number of nodes. For the purpose of this project, it was deemed more relevant to demonstrate the potential of parallelization on small- to medium-scale datasets like SUSY with a variable number of trees, rather than pushing memory usage to its limits.

3.1 Shared Memory

3.1.1 Training

The independence of each decision tree's construction makes Random Forest training highly suitable for shared-memory parallelization. This was achieved by parallelizing the main training loop, where trees are constructed. **OpenMP** was selected over FastFlow due to its superior performance on this task, as observed during development.



Plot 2 - Speedup and Efficiency plots of Training phase

The plots also show that both speedup and efficiency align almost perfectly with theoretical expectations. This indicates that increasing the number of threads leads to substantial improvements in computational efficiency as well as significantly reduced training times. Such behavior can be attributed to **the absence of I/O operations and major bottlenecks**, as the majority of the computation is performed in-memory.

Parallelism was implemented at **two levels**: across trees and within the best-split search, where features are evaluated independently. This nested approach, however, risks oversubscription, as tree-building threads can spawn excessive child threads for the split search, causing performance loss from context switching.

To mitigate this, **FastFlow** was chosen for the inner split-search loop. Its `parallel_reduce` implementation outperformed alternatives, primarily due to its lightweight threading model where idle threads yield resources. This contrasts with **OpenMP**'s fixed worker pool, which retains threads and exacerbates oversubscription in nested parallelism. Within the `parallel_reduce`, features are explored concurrently, and splits are pruned if their impurity is worse than the current best.

This hybrid model excels when available CPU cores outnumber the trees.
For example using:

- 40-thread
- 15 trees
- SUSY dataset

The hybrid OpenMP/FastFlow solution perform as follow:

```
Full OpenMP version
Training Time: 17.512s

Hybrid OpenMP / FF
Training Time: 13.298s
```

This corresponds to a speedup of approximately **1.32x**! Moreover, if the construction time of a single tree is high, reducing the number of threads assigned to tree building and reallocating them to the split search phase can further improve efficiency.

3.1.2 Prediction

The prediction phase is implemented in batch mode, processing a set of samples together for performance reasons, as single-sample evaluation is inefficient for metrics like accuracy.

The core operation is a nested loop iterating through samples and then trees:

```
for (int sample = 0; sample < n_samples; sample++) {
  for (int tree = 0; tree < trees.size(); ++tree) {
    const int pred = trees[tree].predict(X[i]);
    all_votes[sample][pred]++;
  }
}
```

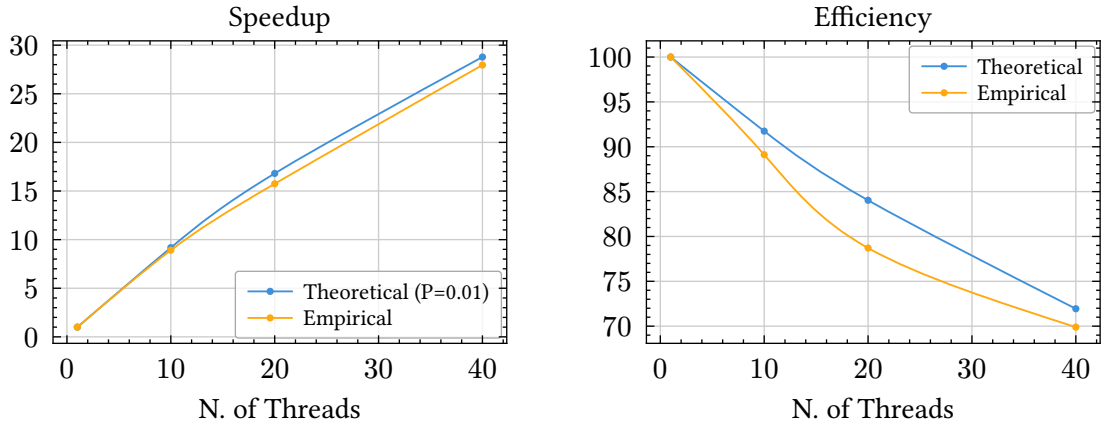
The prediction for each sample is then obtained simply by selecting the maximum value.

The computational cost is: $O\left(n_{\text{samples}} \times n_{\text{trees}} \times \text{tree}_{\text{height}}\right)$. Since $\text{tree}_{\text{height}}$ is typically very small, this simplifies to $O\left(n_{\text{samples}} \times n_{\text{trees}}\right)$. This complexity implies that parallelizing only the outer loop over samples is not sufficient for optimal performance (ideally, each sample should be evaluated across multiple trees in parallel).

To address this, the nested loop structure was identified as an ideal candidate for OpenMP's collapse pragma. This directive flattens the two loops into a single iteration space, enabling maximum parallelism.

```
#pragma omp parallel for collapse(2)
for (int sample = 0; sample < n_samples; sample++) {
  for (int tree = 0; tree < trees.size(); ++tree) {
    const int pred = trees[tree].predict(X[i]);
    all_votes[sample][pred]++;
  }
}
```

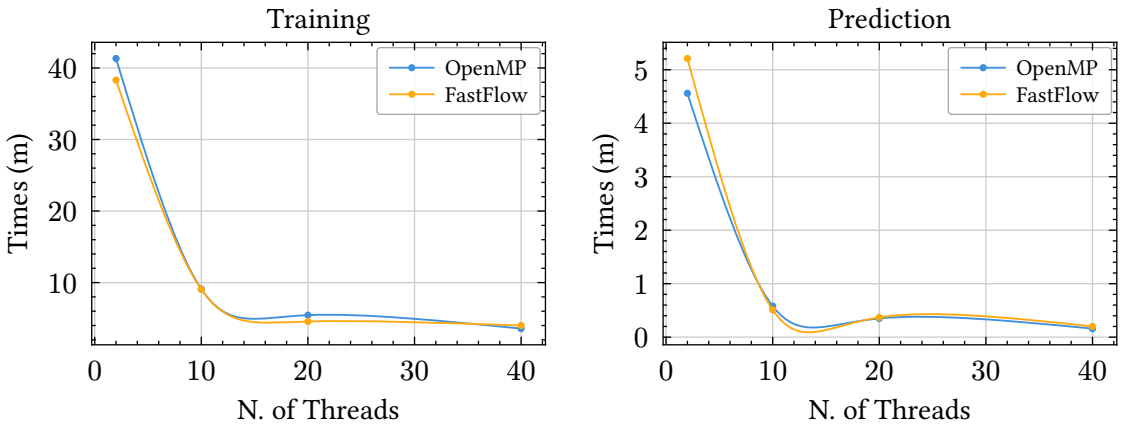
A tiling strategy was also applied to this loop to further enhance performance by improving cache utilization.



Plot 3 - Speedup and Efficiency plots of Prediction phase

In this case as well, the measured data closely follow the theoretical expectations. This behavior arises from the fact that the prediction phase is almost perfectly parallelizable (hence the parallel fraction P approaches 1), and the near absence of bottlenecks allows for significant performance improvements as additional levels of parallelism are introduced.

3.1.3 OpenMP vs FastFlow



As shown in the plots, the two implementations exhibit comparable performance. The **training** phase consists of a parallel for-loop without any concurrent regions, so the similar execution times are expected. The OpenMP implementation was chosen for the final model because its scheduling is easier to customize and the code is more straightforward to write and maintain.

A similar situation occurs in the **prediction** phase: a *Farm* pattern was used for FastFlow and a collapsed for-loop for OpenMP, as described earlier. Specifically, in FastFlow:

- The **Emitter** generates tasks specifying which samples and trees to process (in fixed-size batches).
- The **Workers** perform the actual prediction, producing votes for each sample.
- The **Collector** aggregates the votes from all Workers.

The overall workload is very similar between the two implementations, which explains the comparable execution times. However, FastFlow requires more memory for intermediate data structures and increases code complexity, which is why the OpenMP implementation was ultimately preferred.

3.2 Distributed Memory

In distributed-memory systems, the forest is partitioned across cluster nodes, where each node trains a subset of trees. To minimize communication overhead, trained trees remain on their respective nodes rather than being sent back to the master. During prediction, each node leverages shared-memory parallelism (OpenMP/FastFlow) to compute a local majority vote, then this partial results are sent back to the master node and aggregated to get a final output.

3.2.1 Model Cost

$$T_{\text{total}} = T_{\text{scatter}} + T_{\text{train}} + T_{\text{predict}} + T_{\text{gather}} \tag{5}$$

Where:
- $T_{\text{scatter}} = T_{\text{send}} \times (r-1)$
- $T_{\text{train}} = (n_{\text{trees}}/r) \times T_{\text{single\_tree}}$
- $T_{\text{predict}} = n_{\text{samples}} \times n_{\text{trees}} \times \text{tree}_{\text{height}}$
- $T_{\text{gather}} = T_{\text{receive}} \times (r-1)$

Where $T_{\text{single\_tree}}$ can be formulated as follow, given D as the maximum depth that a single tree can reach during training phase, N the number of samples and F the numeber of features:

$$T_{\text{single\_tree}} \approx N \times F \times \min\{D, \log(N)\} \tag{6}$$

The primary bottlenecks are the initial data distribution (T_{scatter}) and the training phase (T_{train}). The scatter phase becomes critical with large datasets, as the entire training set must be broadcasted to all $r-1$ worker nodes, with overhead growing linearly with both node count and dataset size. The training phase represents the most computationally intensive operation, particularly with high-dimensional data (large F), deep trees (large D), or numerous samples (large N), though it benefits from excellent parallelization with theoretical linear speedup.

In contrast, prediction and gathering phases (T_{predict} and T_{gather}) represent negligible overhead since test datasets are typically much smaller and only prediction vectors are transmitted. Overall efficiency depends on the computation-to-communication ratio: when $T_{\text{train}} \gg T_{\text{scatter}}$, the system achieves near-optimal scalability.

3.2.2 Results

The experiments were conducted using the SUSY dataset, with a maximum of 1000 trees per model.

This limitation was imposed by the cluster's 30-minute runtime cap, which made it impossible to run larger experiments. As a result, some of the measured speedups may seem modest when increasing the number of cores, mainly because the problem size is relatively small. However, the experiments were still more than enough to show the expected scaling trends and to validate the parallel behavior of the implementation.

Note: to produce valid plots, the sequential version was ran on a private server.

3.2.3 Times Analysis

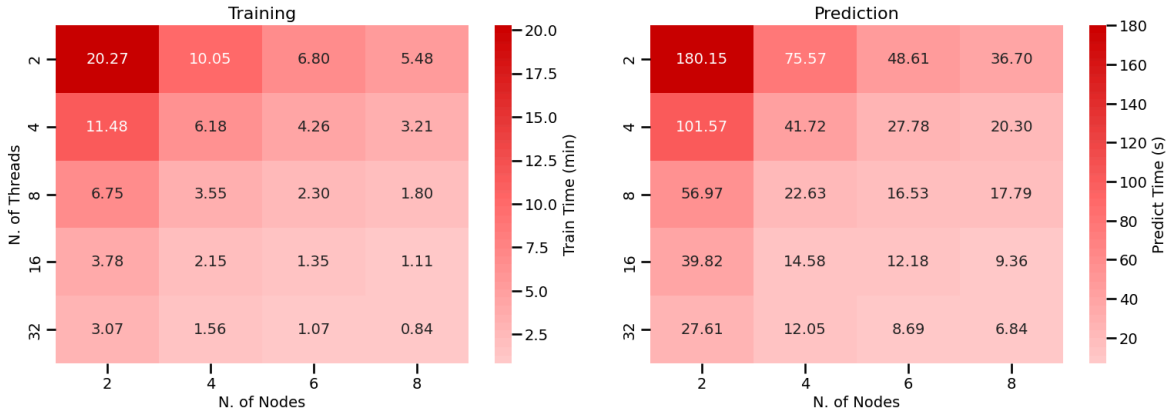


Figure 1 - Heatmaps of training and prediction time using 1K trees on SUSY dataset

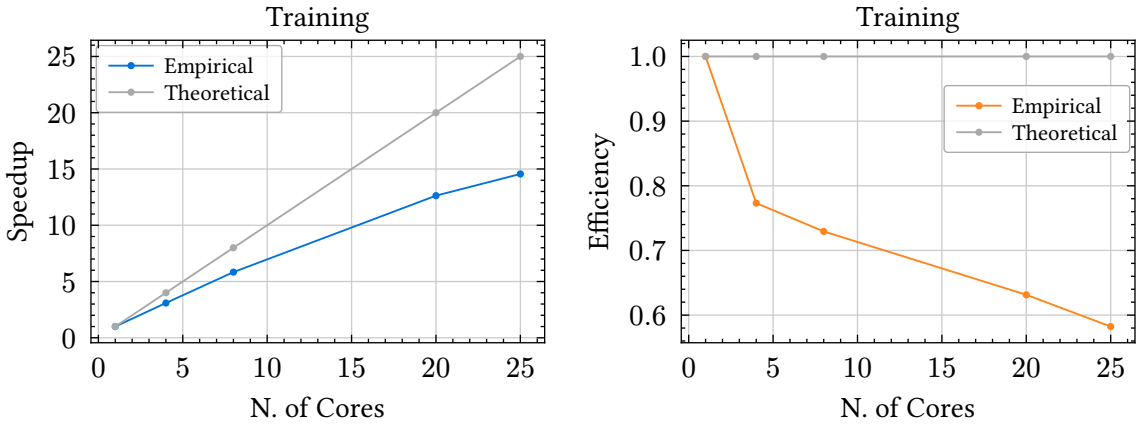
The heatmaps provide a clear overview of how training and prediction times evolve as the number of nodes and threads increases. For the training phase, the speedup is particularly impressive: the runtime decreases from approximately 20 minutes to just 0.84 minutes, corresponding to a **2281% improvement**.

The relatively small problem size becomes evident once a certain threshold is reached, beyond which further increases in resources yield only marginal gains. Nevertheless, the smooth gradient observed in the heatmap strongly indicates that the developed model scales efficiently with the number of available cores.

As a curiosity, training 10,000 trees on 8 nodes with 32 threads took only **7.295 minutes** — a remarkable result that highlights the scalability and efficiency of the implementation.

3.2.4 Strong scaling

Given N. of Cores = N. of nodes × N. of Threads:

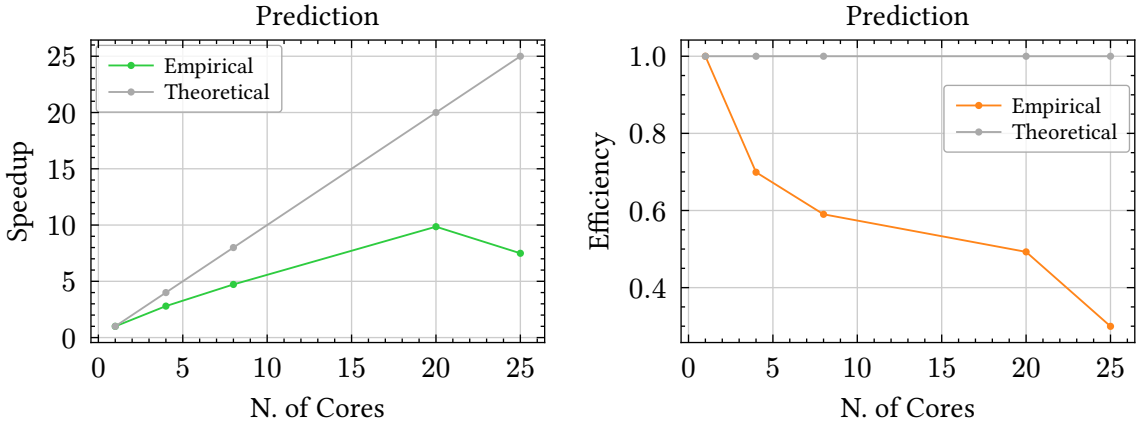


Plot 4 - Strong scaling on Training

In the context of strong scaling, the problem size is kept fixed while the number of available computational resources is increased to analyze how performance evolves.

Since the trees to create is relatively small, the number of cores was chosen to best highlight the expected sub-linear speedup behavior. In fact, performance improved up to ≈ **16 cores**, but after that, adding more cores didn't help much. This is the expected result because the task wasn't large enough to need more processing power.

Importantly, the observed sub-linear speedup closely matches theoretical predictions, while efficiency gradually decreases from nearly 1.0 to approximately 0.65. This degradation is expected in strong scaling scenarios and is primarily caused by **memory bandwidth saturation**, OpenMP **synchronization overhead**, and the inherent **sequential portions** described by Amdahl's Law. The implementation uses **OpenMP dynamic scheduling** to handle the variability in tree construction times, resulting in smooth efficiency degradation rather than abrupt performance collapse.



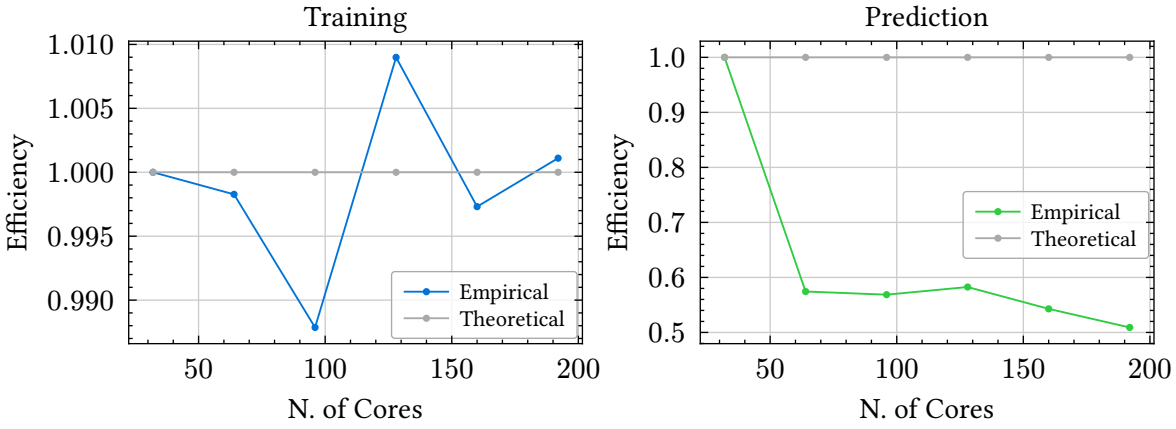
Plot 5 - Strong scaling on Prediction

In contrast to the training phase, the prediction phase exhibits a markedly different scaling behavior. While the speedup curve also tends to flatten after approximately **16 cores**, the efficiency shows a sharp drop as soon as multiple nodes are involved. Initially, the efficiency is close to 1 when running on a single node, indicating excellent resource utilization.

However, as soon as additional nodes are introduced, **MPI communication overhead dominates**, causing a drastic reduction in parallel efficiency.

3.2.5 Weak Scaling

Given N. of Cores = N. of nodes × N. of Threads:



Plot 6 - Weak scaling on Training and Prediction“

For the training phase, the weak scaling efficiency remains consistently around ± 1, indicating near-ideal scalability as the number of cores increases. This behavior suggests an excellent load balance across processes and minimal communication overhead. The absence of significant I/O or MPI bottlenecks plays a key role here, as the tree structures remain distributed, allowing the algorithm to maintain high efficiency even when scaling to a large number of cores.

In contrast, the prediction phase shows a different trend. While the efficiency is close to 100% when running on a single node, it drops sharply once inter-node communication is introduced. This decline highlights the non-negligible overhead associated with MPI communication during prediction, where the computational workload per process is comparatively lower and communication costs dominate.

4 Conclusions

I am quite satisfied with the implementation achieved so far. However, there is room for improvement, particularly in reducing access to the shared matrix by multiple threads. This matrix is accessed millions of times concurrently, which can lead to memory bus contention and cache thrashing. One possible optimization would be to pre-load the most frequently used or selected features so that each thread maintains its own local copy. This could reduce contention, but it must be carefully balanced to avoid excessive memory usage, especially with large datasets where such replication could introduce significant overhead.

Another interesting future direction could be to explore a gradient-based implementation similar to LightGBM. This would allow the model to support more advanced learning techniques and potentially improve both training efficiency and prediction accuracy, while still leveraging the distributed architecture developed in this work.

Bibliography

[1] L. Breiman, “Random Forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[2] Scikit-learn development team, “sklearn.ensemble.RandomForestClassifier.” 2023.

[3] J. Jarrold, “cxxopts.” 2023.

[4] O. R. L. Peters, “pdqsort.” 2021.

[5] M. E. O'Neill, “PCG: A Family of Better Random Number Generators.” 2014.

[6] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, “Classification and Regression Trees,” in *Wadsworth International Group*, 1984, pp. 28–32.

[7] B. Jacob *et al.*, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[8] Scikit-learn development team, “Histogram-Based Gradient Boosting.” 2023.