



High Performance Computing – 2025/2026
MSc. in Informatics – ESTiG/IPB

Practical Work 2:
Random Forests Acceleration with MPI

Leonardo Canuto Junior	a67607
Diogo Emanuel Antunes Santos	a67603

Objetivo

Acelerar uma implementação sequencial do método Random Forests usando MPI.

Questões

Letra A

Testado (não obrigatório):

```
[cad07@rocks-frontend randforest-serial]$ make
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c main.c -o main.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c utils/utils.c -o utils/utils.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c utils/data.c -o utils/data.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c utils/argparse.c -o utils/argparse.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c model/tree.c -o model/tree.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c model/forest.c -o model/forest.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c eval/eval.c -o eval/eval.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -c utils/log.c -o utils/log.o
gcc -std=c99 -O0 -g -Wall -Wextra -Wpedantic -o random-forest main.o utils/utils.o utils/data.o utils/argparse.o
model/tree.o model/forest.o eval/eval.o utils/log.o
[cad07@rocks-frontend randforest-serial]$ ./random-forest wdbc.csv --seed 0
using:
  seed: 0
  verbose log level: 1
  rows: 568, cols: 32
reading from csv file:
  "wdbc.csv"
using:
  k_folds: 5
using RandomForestParameters:
  n_estimators: 3
  max_depth: 7
  min_samples_leaf: 3
  max_features: 3
cross validation accuracy: 93.451327% (93%)
(time taken: 1.070000s)
```

Letra B

b1)

Comando usado para executar através do gprofng:

Shell

```
gprofng collect app -o random-forest.er ./random-forest wdbc.csv --seed 0
```

```
[cad07@rocks-node-7 randforest-serial]$ gprofng collect app -o random-forest.er ./random-forest wdbc.csv --seed 0
Creating experiment directory random-forest.er (Process ID: 27827) ...
using:
  seed: 0
  verbose log level: 1
  rows: 568, cols: 32
reading from csv file:
  "wdbc.csv"
using:
  k_folds: 10
using RandomForestParameters:
  n_estimators: 10
  max_depth: 7
  min_samples_leaf: 3
  max_features: 10
cross validation accuracy: 95.714286% (95%)
(time taken: 66.060000s)
[cad07@rocks-node-7 randforest-serial]$
```

b.2)

Comando usado para exibir a árvore de chamadas:

Shell

```
gprofng display text -calltree random-forest.er
```

Árvore de chamadas com maior que 30% de CPU time (19.8s)

Shell

Functions Call Tree. Metric: Attributed Total CPU Time

b.3) O maior ponto de anomalia encontrado foi dentro da função `calculate_best_data_split`, pelo profiling identificamos que ela ocupou 99,99%, equivalente a 66,055 segundos.

Dentro dela, a função `split_dataset` se destaca com total de 65,427 segundos e 99,03% (maior gargalo dentro da função, nela é evidente que há muito tempo gasto com as chamadas de `realloc` (40,228 segundos e 60,89%). É considerado má gestão de memória, onde há um realloc a cada linha, o que dispara muitas realocações desnecessárias.

Letra C

c.1) Para resolver, o primeiro passo será incluir antes do malloc, uma etapa para ter certeza nos tamanhos do `left_count` e `right_count`, para evitar que seja necessário realloc a cada linha, o que sobrecarrega a função.

Para isso, apenas é replicado o for e o if que faz os reallocs, mas utilizando para a contagem e não realocação.

Código:

```
C/C++  
for (size_t i = 0; i < rows; ++i) {  
    double *row = data[i];  
    if (row[feature_index] < value)  
        ++left_count;  
    else  
        ++right_count;  
}
```

Depois altero o malloc, para receber a quantidade correta do left e right count:

Código:

```
C/C++  
  
double **left = (double **)malloc(left_count * sizeof(double *));  
double **right = (double **)malloc(right_count * sizeof(double *));
```

Agora comento as linhas que possuem a função realloc antes de utilizá-los no for.

c.2) A versão otimizada apresentou um tempo de 21.465s, mantendo em +95% de acurácia.

```
[cad07@rocks-node-7 randforest-serial-opt]$ gprofng collect app -o random-forest.er ./random-forest wdbc.csv --seed 0
Creating experiment directory random-forest.er (Process ID: 25651) ...
using:
  seed: 0
  verbose log level: 1
  rows: 568, cols: 32
reading from csv file:
  "wdbc.csv"
using:
  k_folds: 10
using RandomForestParameters:
  n_estimators: 10
  max_depth: 7
  min_samples_leaf: 3
  max_features: 10
cross validation accuracy: 95.714286% (95%)
(time taken: 21.470000s)
```

c.3) Árvore de chamadas com maior que 30% de CPU time

Shell

Functions Call Tree. Metric: Attributed Total CPU Time

Attr. Total	Name
CPU	
sec.	%
21.465 100.00	+-<Total>
21.465 100.00	+--__libc_start_main
21.465 100.00	+--main
21.465 100.00	+--cross_validate
21.465 100.00	+--train_model
21.465 100.00	+--train_model_tree
13.740 64.01	+--grow
10.267 47.83	+--grow
7.545 35.15	+--grow
5.264 24.52	+--grow
3.112 14.50	+--grow
1.761 8.21	+--calculate_best_data_split
1.151 5.36	+--split_dataset
...	

c.4) A melhor função identificada para ser paralelizada é a função **train_model**.

Todas as outras funções são recursivas, o que dificulta de ser desenvolvido funções paralelas com trocas de mensagens. A partir da função **train_model**, é chamada a função **train_model_tree** pela quantidade de n estimators (quantidade de árvores).

O foco será paralelizar a **train_model**, assim cada **train_model_tree** pode ser executada em paralelo por outro processo.

A **train_model** apresenta um weight no call tree de 100% e CPU time de 21.465s (tempo total de execução). Pela **Lei de Amdahl**, iremos adotar 99,99%.

Letra D

Pela **Lei de Amdahl** adotada, teremos: **p = 0,9999** (99,99%).

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

Fórmula 1 - [Lei de Amdahl – Wikipédia, a encyclopédia livre](#) (2025)

$$S_T(2) = 1 / ((1 - 0,9999) + 0,9999 / 2) = 1,9998$$

$$E_T(2) = (1,9998 / 2) * 100 \approx 99,99\%$$

$$S_T(4) = 1 / ((1 - 0,9999) + 0,9999 / 4) = 3,9988$$

$$E_T(4) = (3,9988 / 4) * 100 \approx 99,97\%$$

$$S_T(6) = 1 / ((1 - 0,9999) + 0,9999 / 6) = 5,9976$$

$$E_T(6) = (5,9976 / 6) * 100 \approx 99,96\%$$

$$S_T(8) = 1 / ((1 - 0,9999) + 0,9999 / 8) = 7,9944$$

$$E_T(8) = (7,9944 / 8) * 100 \approx 99,93\%$$

$$S_T(10) = 1 / ((1 - 0,9999) + 0,9999 / 10) = 9,9910$$

$$E_T(10) = (9,9910 / 10) * 100 \approx 99,91\%$$

$$S_T(12) = 1 / ((1 - 0,9999) + 0,9999 / 12) = 11,9868$$

$$E_T(12) = (11,9868 / 12) * 100 \approx 99,89\%$$

$$S_T(14) = 1 / ((1 - 0,9999) + 0,9999 / 14) = 13,9814$$

$$E_T(14) = (13,9814 / 14) * 100 \approx 99,87\%$$

$$S_T(16) = 1 / ((1 - 0,9999) + 0,9999 / 16) = 15,9748$$

$$E_T(16) = (15,9748 / 16) * 100 \approx 99,84\%$$

d1)

N	2	4	6	8	10	12	14	16
S _T	1,99	3,99	5,99	7,99	9,99	11,98	13,98	15,97

Table 1 - Theoretical Speedups

d2)

N	2	4	6	8	10	12	14	16
E _T (%)	99,99	99,97	99,96	99,93	99,91	99,89	99,87	99,84

Table 2 - Theoretical Efficiency (in percentage)

$$\text{d3)} \quad S_T(\infty) = \frac{1}{1-p} = \frac{1}{1-0,9999} = 10000$$

O limite teórico de speed é aproximadamente 10000.

Letra E

e.1) O desenvolvimento foi realizado em duas etapas (dividida por arquivos):

- **Etapa 1:** Adaptação do arquivo **main.c** para suportar a paralelização.

No arquivo **main.c**, inicializamos o ambiente MPI e preparamos os dados para distribuição.

O **processo 0** é responsável por fazer o parse de args, random seed, ler os dados e distribuí-los para todos os processos usando **MPI_Bcast**:

```
C/C++  
if (rank == 0) {  
    data = malloc(sizeof(double) * csv_dim.rows * csv_dim.cols);  
    parse_csv(file_name, &data, csv_dim);  
}  
  
// broadcast dimensoes  
MPI_Bcast(&csv_dim.rows, 1, MPI_LONG, 0, MPI_COMM_WORLD);  
MPI_Bcast(data, n_elements, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- **Etapa 2:** Ajuste e adaptação nas funções **train_model** e dependentes (**predict_model** e **free_random_forest**), no arquivo **forest.c**.

Na função **train_model**, mudamos de construir todas as árvores sequencialmente para dividir o trabalho entre processos igualmente e com resto (*remainder*), Isso também foi desenvolvido para as funções dependentes (**predict_model** e **free_random_forest**).

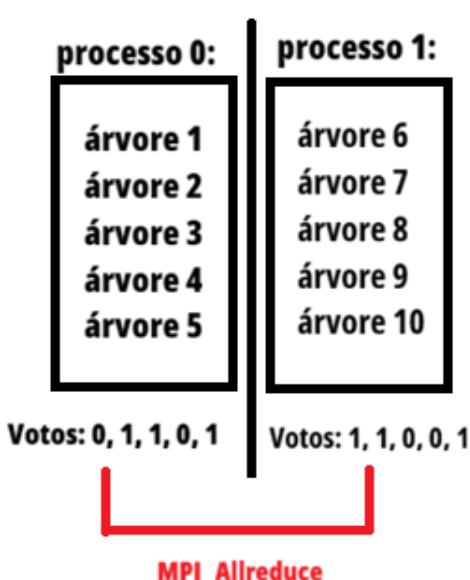
```
C/C++  
// calcula quantas arvores cada processo vai construir  
int trees_per_process = n_estimators / numtasks;  
int remainder = n_estimators % numtasks;  
  
// ind de inicio e fim para este processo  
int start_tree = rank * trees_per_process + (rank < remainder ? rank :  
remainder);  
int end_tree = start_tree + trees_per_proces
```

Foi feito a construção e criado um seed para cada árvore construída, (e enviado por **MPI_Bcast**) a partir de árvores locais, também desenvolvemos alguns log levels (2 nível que não sobrecarregar tanto o código, pois estava rodando no nível 0, mas pode ser acompanhada (o volume é alto de árvore locais, então polui visualmente).

```
C/C++  
for (int i = 0; i < local_n_trees; ++i)  
{  
    int tree_id = start_tree + i;  
  
    unsigned int tree_seed;  
    MPI_Bcast(&tree_seed, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);  
  
    if (rank == 0) {  
        tree_seed = rand();  
    }  
  
    MPI_Bcast(&tree_seed, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);  
    srand(tree_seed + tree_id);  
  
    log_if_level(2, "Rank %d: building global tree %d (local %d)\n",  
                rank, tree_id, i);  
  
    random_forest[i] = train_model_tree(data, params, csv_dim, &nodeId,  
                                         ctx);  
}
```

Na função **predict_model**, utilizamos do **MPI_Allreduce** para juntar os globais 1 e 0, assim retornando a comparação total entre as árvores locais de cada processo (com uso de **MPI_SUM**).

Desenho representativo da estrutura desenvolvida:



Cada árvore é como uma chamada da função **train_model_tree**.

A parte de votos é realizada pela função **predict_model**, nela cada processo prediz localmente da sua árvore.

No final, na função **predict_model**, é chamado um **MPI_Allreduce** e depois comparado os valores globais de predição da árvore.

Durante o processo foi identificada a necessidade também de paralelizar a **free_random_forest**, para ir limpando as árvores temporárias criadas em cada processo.

Dados do benchmarking para registro:

Serial Opt	MPI com 2	MPI com 4	MPI com 6	MPI com 8	MPI com 10	MPI com 12
192,250000	98,200000	50,430000	40,820000	30,190000	20,840000	20,680000
192,220000	98,060000	50,090000	39,990000	30,350000	20,610000	20,570000
192,570000	98,400000	50,300000	40,900000	30,150000	20,610000	20,680000
Média						
192,346666	98,220000	50,2733333	40,5700000	30,2300000	20,6866666	20,6433333

MPI com 14	MPI com 16
20,630000	20,520000
20,630000	20,540000
20,460000	20,380000
Média	
20,5733333	20,4800000

Fórmula para RDS (%):

$$\bar{T}_R = \frac{\sum_{i=1}^k T_i}{k}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^k (T_i - \bar{T}_n)^2}{k}}$$

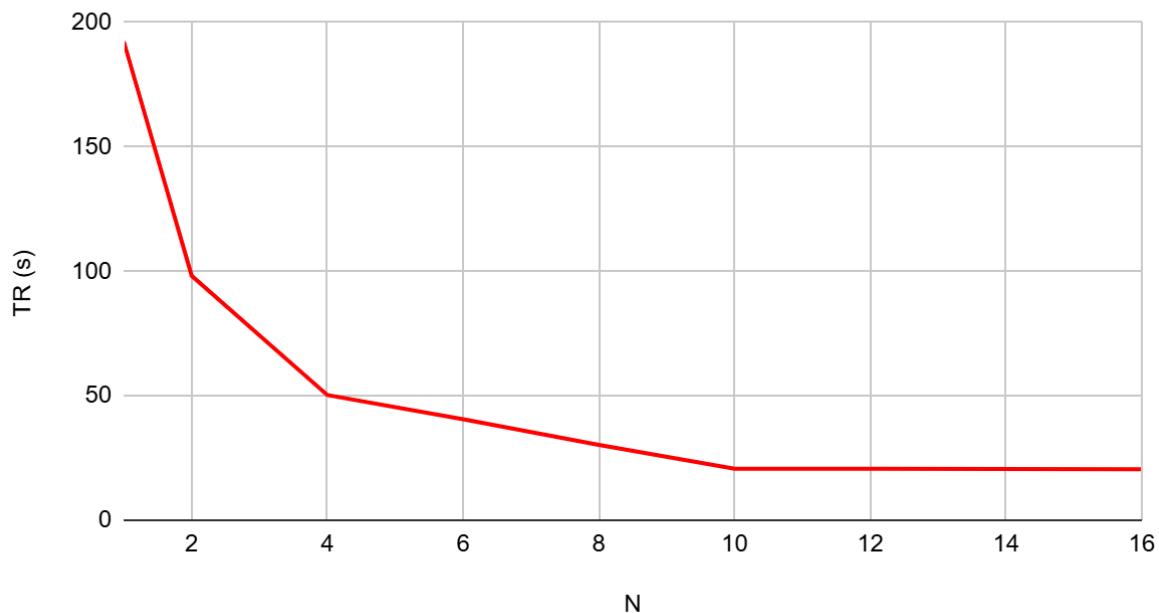
$$RDS = \frac{\sigma}{\bar{T}_n} \times 100\%$$

e.2)

<i>N</i>	1	2	4	6	8	10	12	14	16
$T_R(s)$	192,346	98,220	50,273	40,570	30,230	20,686	20,643	20,573	20,480
RSD (%)	0,1008	0,1739	0,3412	1,2420	0,3500	0,6419	0,3076	0,4770	0,4256

Table 3 - Real Exec. Times of the Optimized Serial Version (N=1), and MPI Parallel Version (N>1)

TR (s) versus N



Graph 1 - Real Exec. Times (N>=1) x N process

e.3) $T_1 = 192,346666$

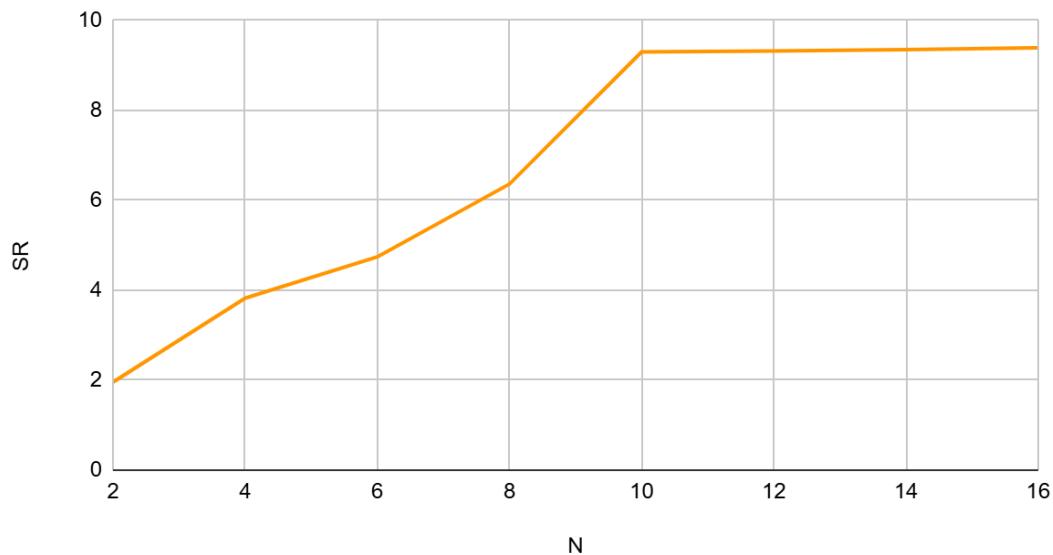
$$S_R(N) = T_1 / T_R(N)$$

$$E_R(N) = (S_R(N) / N) * 100$$

N	2	4	6	8	10	12	14	16
S_R	1,9583	3,8260	4,7411	6,3627	9,2980	9,3176	9,3493	9,3919
$E_R(%)$	97,9162	95,6504	79,0184	79,5346	92,9809	77,6468	66,7808	58,6995

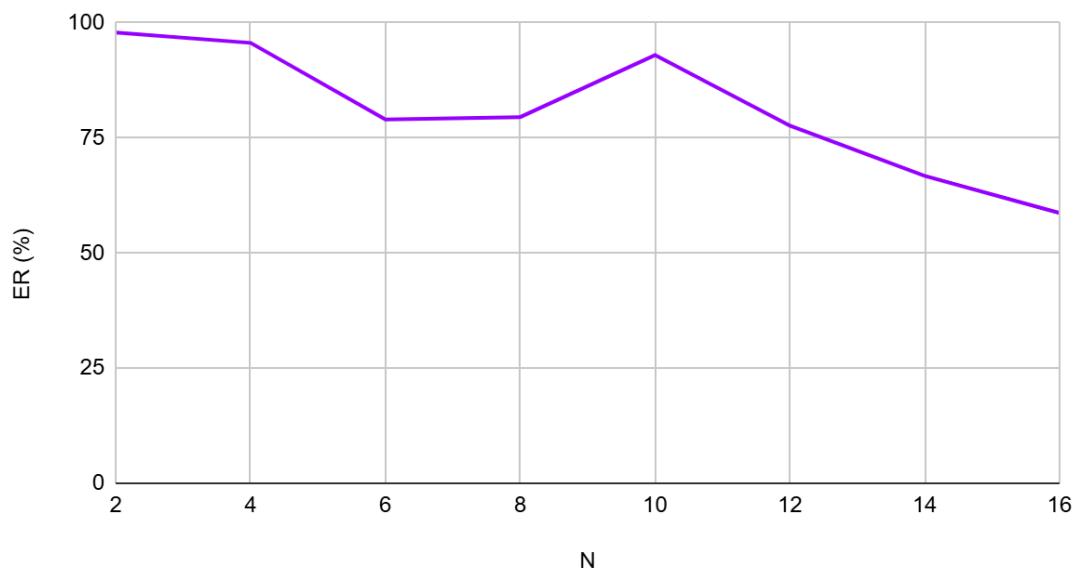
Table 4 - Real Speedups and Real Efficiencies of the MPI Parallel Version

SR versus N



Graph 2 - Real Speedups x N process

ER (%) versus N



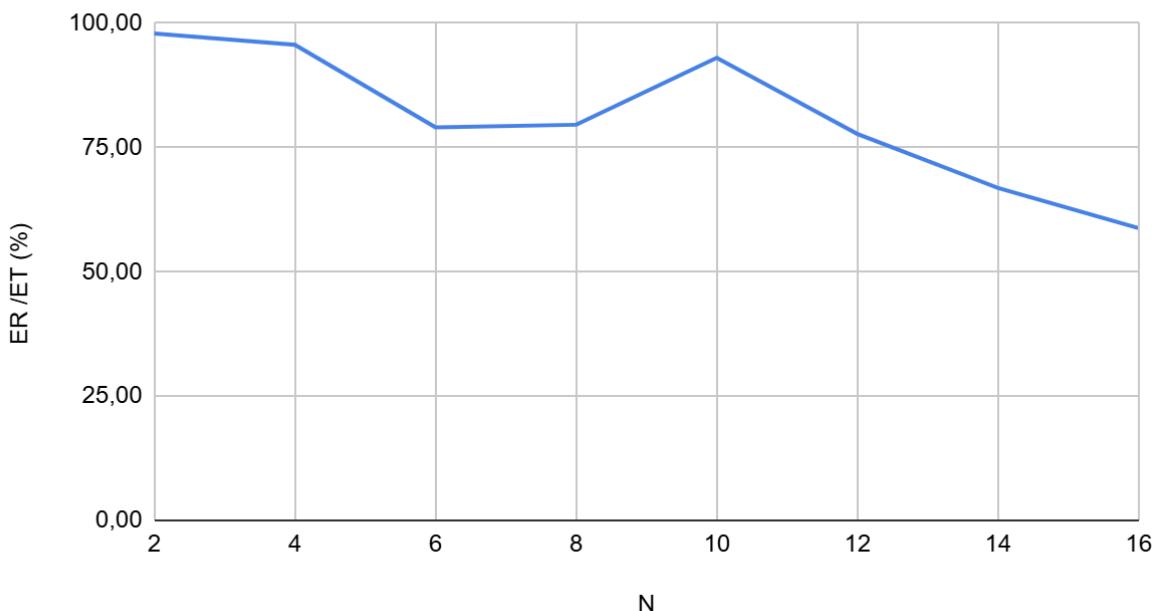
Graph 3 - Real Efficiencies x N process

$$\mathbf{e.4) E_R/E_T (\%)} = E_R/E_T * 100$$

N	2	4	6	8	10	12	14	16
$E_R/E_T(\%)$	97,926	95,679	79,050	79,590	93,065	77,732	66,868	58,794

Table 5 - Closeness of the real to ideal efficiency of the MPI Parallel Version

ER /ET (%) versus N



Graph 4 - Closeness of the real to ideal efficiency x N process

Letra F

f.1) Os resultados obtidos mostram um comportamento interessante em relação às previsões da Lei de Amdahl. Para 2, 4 e 10 processos, os resultados ficaram muito próximos do esperado, com eficiências aproximadas de 97,9%, 95,7% e 93,0% respectivamente, demonstrando speedups lineares. Isso confirma que a estratégia de paralelização no nível de árvores foi bem-sucedida, pois cada árvore é completamente independente e não há overhead significativo de comunicação durante a construção.

Porém, para [6, 8, 12, 14, 16] processos, observamos quedas significativas na eficiência, chegando a 58,7% para 16 processos. A principal razão para essa discrepância em relação às previsões de Amdahl é o desbalanceamento de carga causado pelo número fixo de árvores (`n_estimators`). Esse tempo ocioso não está previsto na Lei de Amdahl, que assume distribuição perfeita de trabalho. As melhores eficiências apresentadas foram em quantidades de processos divisíveis pelo número de árvores, como 2, 4 e 10.

Com 20 árvores e 6 processos, por exemplo: 2 processos constroem 4 árvores cada, enquanto 4 processos constroem apenas 3 árvores, resultando em tempo ocioso. Os 4

processos que terminam mais cedo (após construir as 3 árvores), ficam aguardando na sincronização do **MPI_Allreduce** enquanto os 2 primeiros processos ainda estão construindo sua quarta árvore, desperdiçando recursos computacionais.

Como complemento, foi realizado **um teste extra ao projeto com 20 processos e 20 árvores**, com um time taken de 10,82s. A eficiência real chega em 88,88%, e o valor de proximidade de E_R/E_T a aproximadamente 89,05%. **Um aumento significativo acerca dos 58,7% para 16 processos**, comprovando que a quantidade de processos divisíveis impacta em melhor eficiência do algoritmo.

Para cenários de produção em escala, recomenda-se escolher os **n_estimators** que seja múltiplo do número de processos disponíveis (ex: 24, 30 ou 36 árvores para até 12 processos), ou implementar balanceamento dinâmico de carga usando padrão master-worker onde as árvores são distribuídas sob demanda.