# Relatório de Desempenho: Odd–Even Transposition Sort – Serial, OpenMP e MPI

NOME: Henrique Almico Dias da Silva
DRE: 124238228

## 1. Introdução

Este relatório tem como objetivo analisar comparativamente o desempenho do algoritmo de ordenação Odd–Even Transposition Sort sob três diferentes paradigmas de execução: implementação serial, paralelização com OpenMP (memória compartilhada) e paralelização com MPI (memória distribuída). Cada abordagem foi avaliada com base em tempo de execução, speedup, eficiência e overhead de comunicação, permitindo compreender suas vantagens e limitações em diferentes contextos computacionais.

## 2. Ambiente de Execução

As medições foram realizadas em um notebook com a seguinte configuração:

| **Processador** | **Intel Core i7-7700HQ (4C/8T)** |
|---|---|
| **Memória RAM** | 16 GB DDR4 |
| **Sistema Operacional** | Ubuntu 24.04 LTS |
| **Compiladores** | GCC 13.3.0, MPICH 4.1.1, OpenMP |

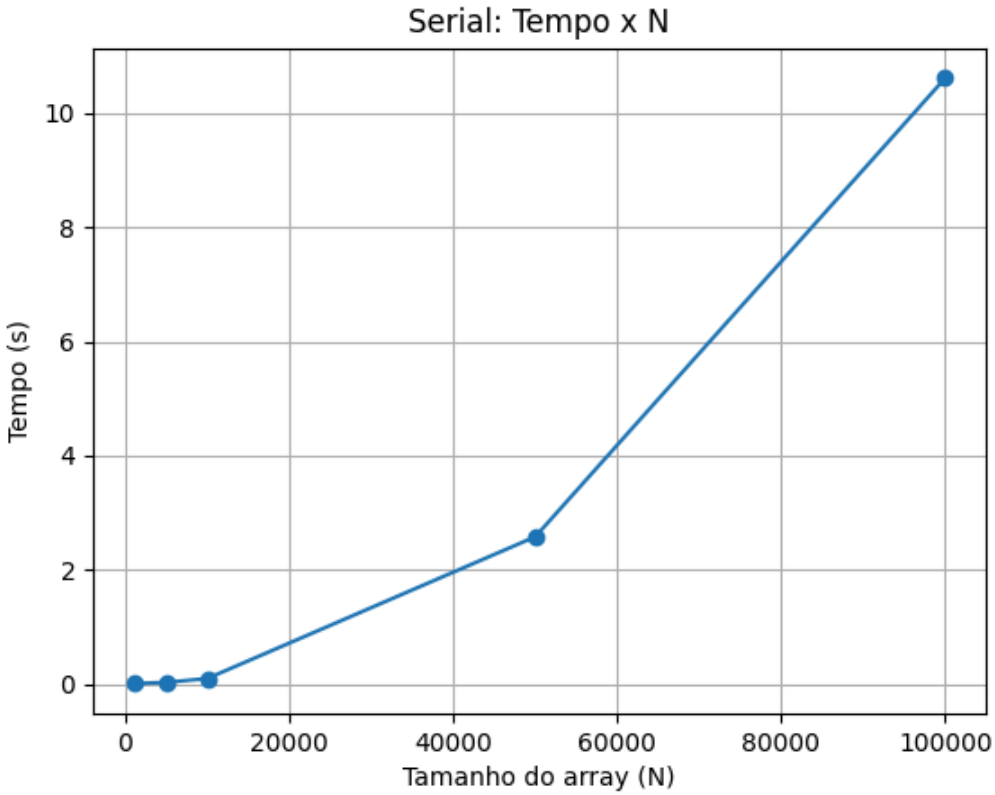
## 3. Metodologia Experimental

Foram testados vetores de tamanhos variados ($N \in \{1000, 5000, 10000, 50000, 100000\}$) para observar o comportamento das implementações sob diferentes cargas de trabalho. Cada experimento foi repetido três vezes para média dos resultados. Nas versões paralelas, variou-se também o número de threads (OpenMP) ou processos (MPI) $\in \{1, 2, 4, 8\}$. Para OpenMP, testaram-se três políticas de agendamento: static, dynamic e guided.

## 4. Resultados e Discussão

### 4.1 Implementação Serial

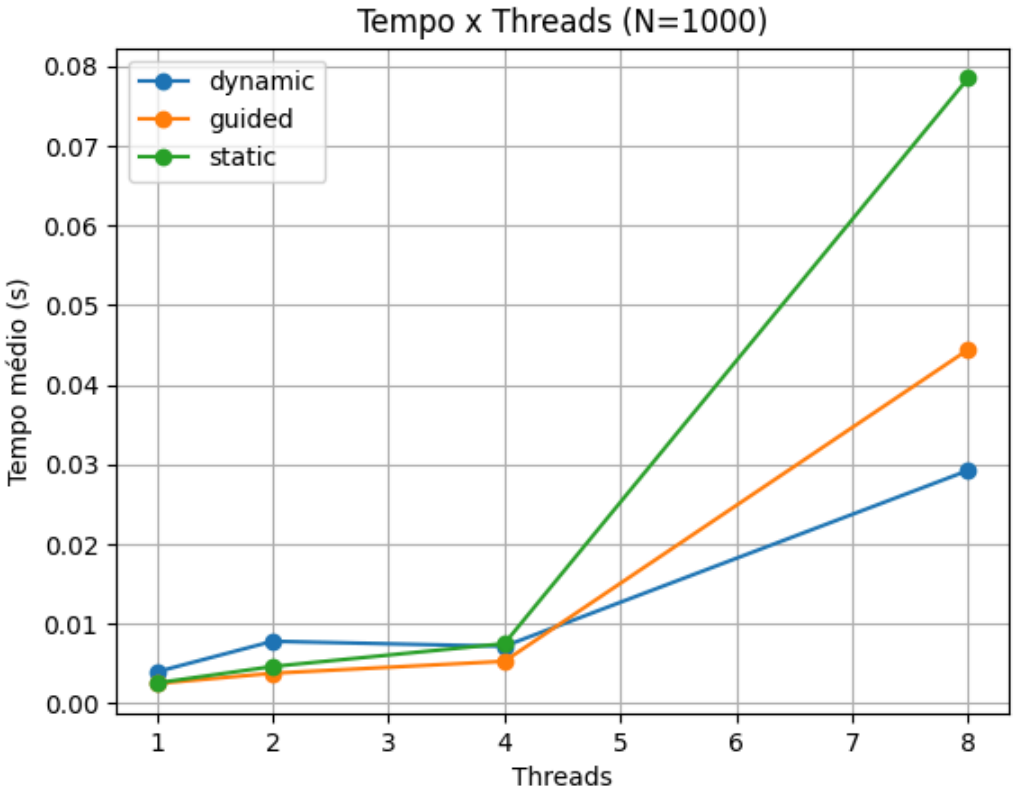
A implementação serial serve como base de comparação para as versões paralelas. O tempo de execução apresentou crescimento quadrático conforme o esperado pela complexidade do algoritmo $O(N^2)$.

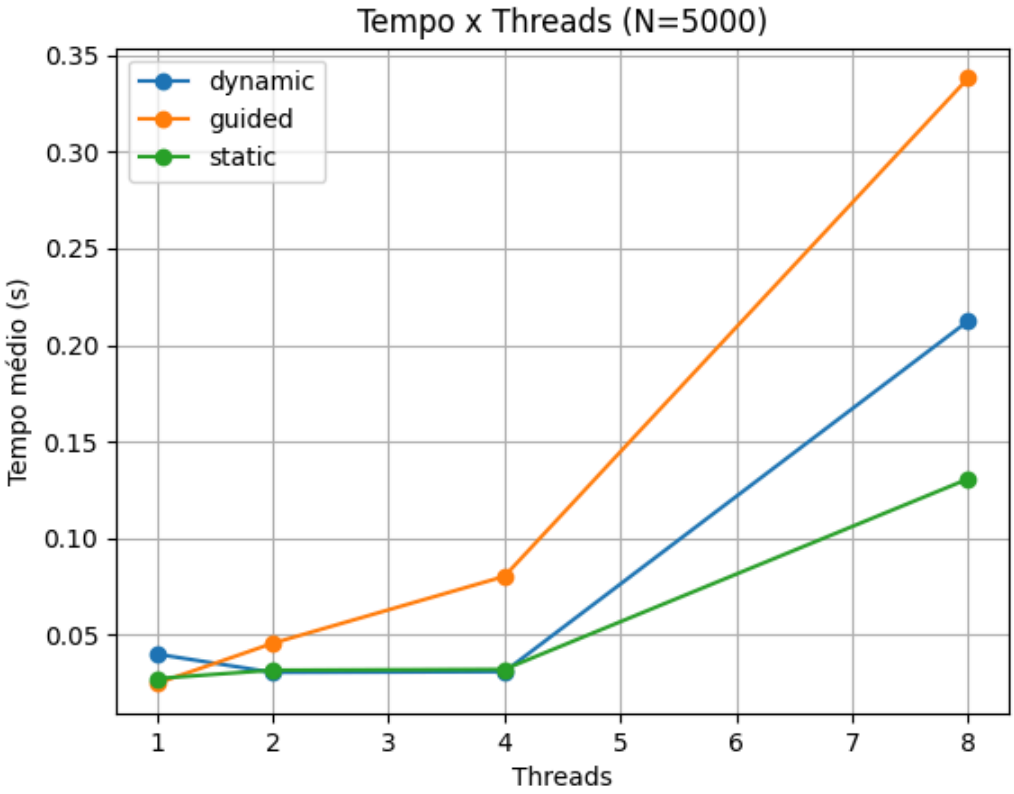


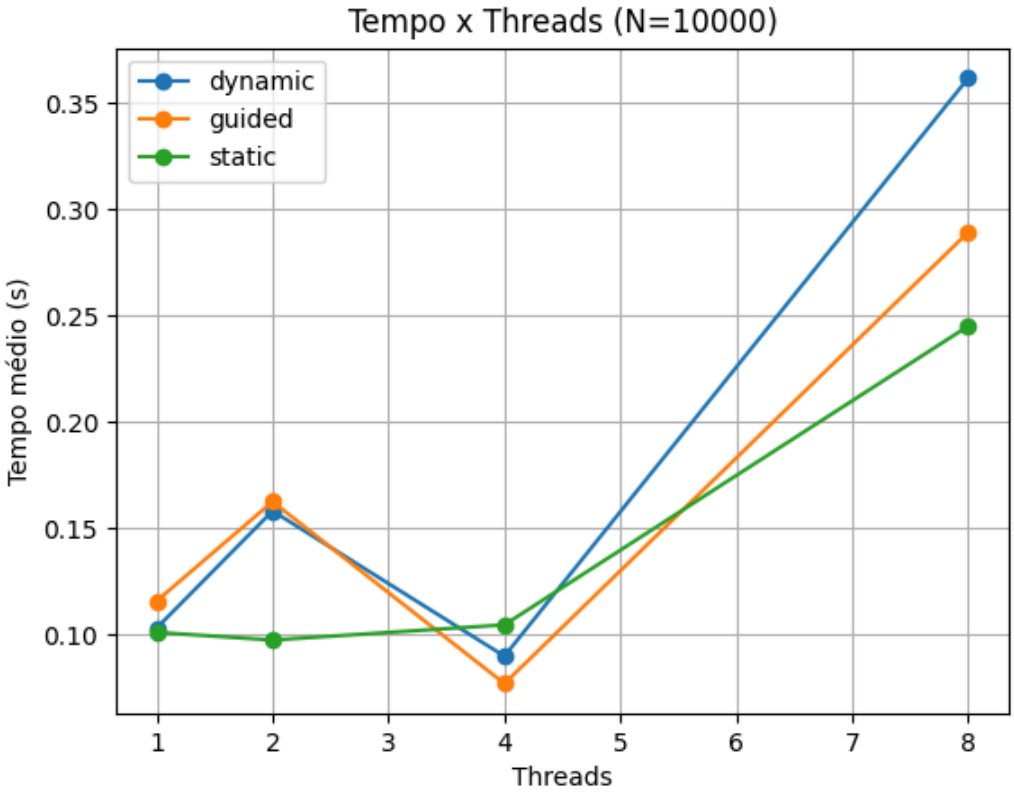

Por exemplo, ao dobrar o tamanho de entrada de 50.000 para 100.000 elementos, observou-se aumento de tempo de aproximadamente 4 vezes, caracterizando o comportamento típico de algoritmos de ordenação comparativa simples.

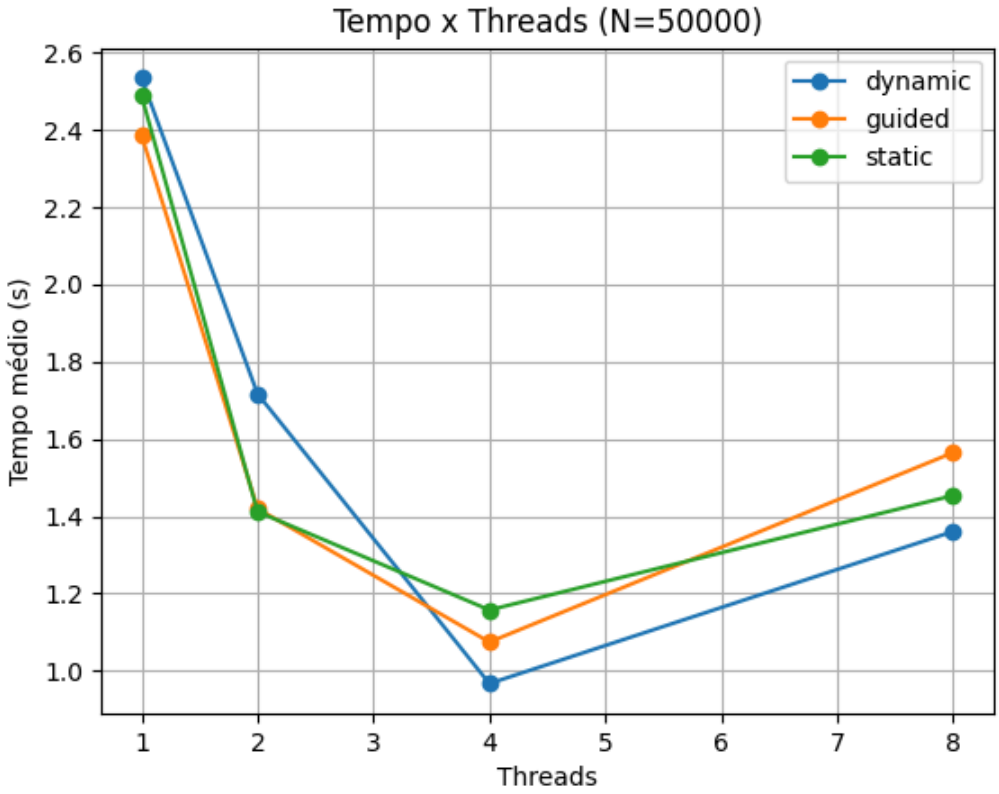
### 4.2 Implementação com OpenMP

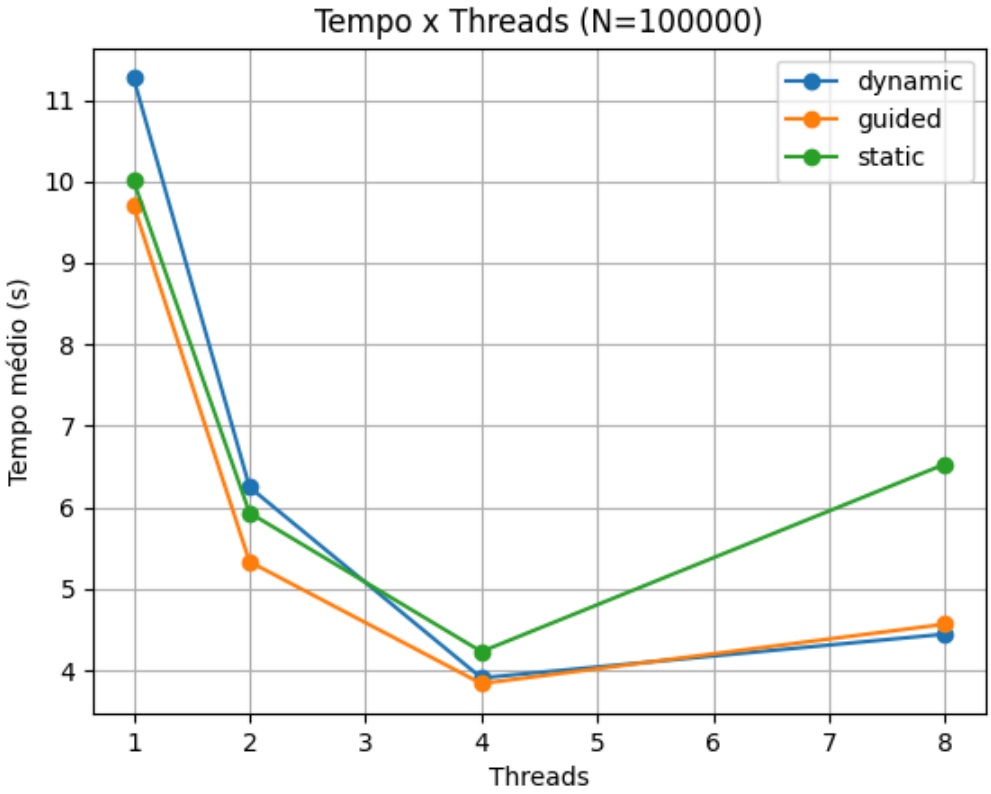
As execuções com OpenMP evidenciaram ganhos de desempenho consideráveis, sobretudo com política 'static'. A paralelização foi aplicada ao laço principal de comparação usando diretivas #pragma omp parallel for, com controle de scheduling ajustado em tempo de execução.

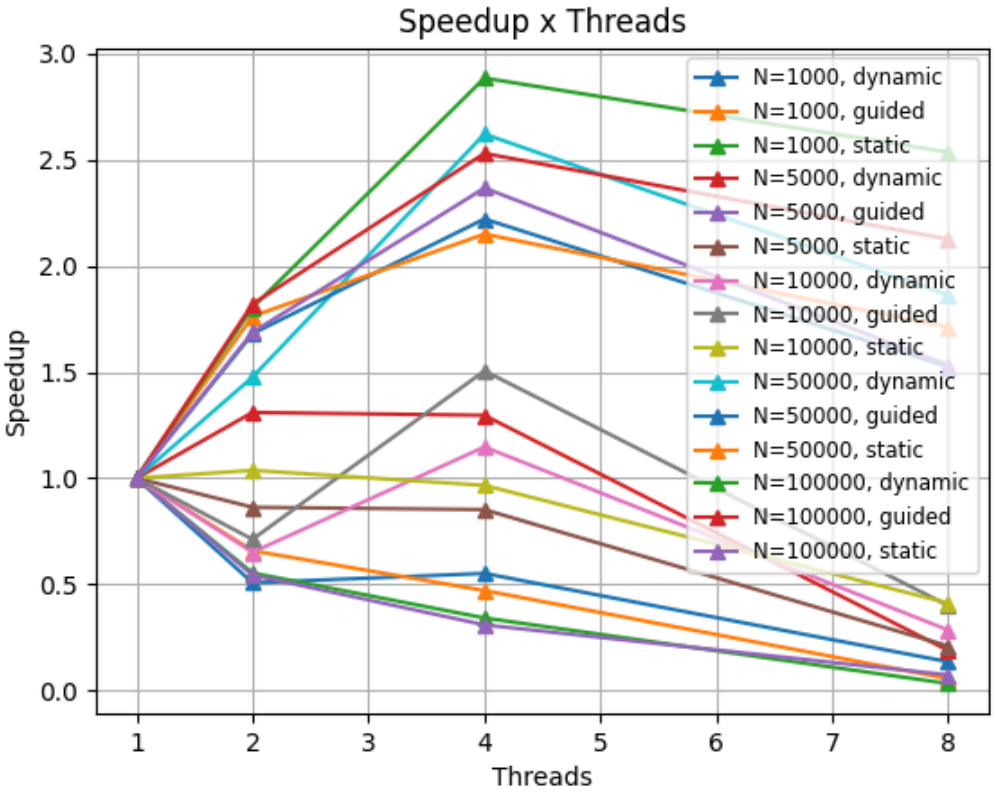

Tempo x Threads (N=1000)
dynamic
guided
static
Tempo médio (s)
Threads
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
1
2
3
4
5
6
7
8

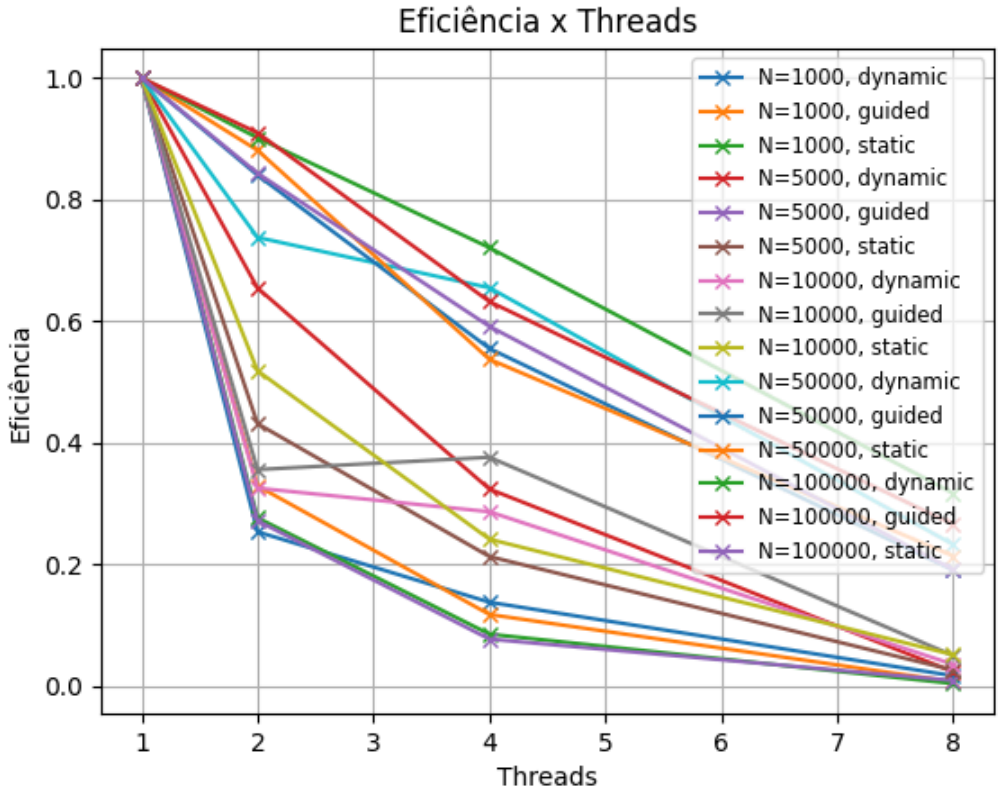

Tempo x Threads (N=5000)
dynamic
guided
static
Tempo médio (s)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
1
2
3
4
5
6
7
8
Threads


Tempo x Threads (N=10000)
dynamic
guided
static
Tempo médio (s)
0.35
0.30
0.25
0.20
0.15
0.10
1
2
3
4
5
6
7
8
Threads


Tempo x Threads (N=50000)
Tempo médio (s)
Threads
dynamic
guided
static
2.6
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
1
2
3
4
5
6
7
8


Tempo x Threads (N=100000)
dynamic
guided
static
Tempo médio (s)
Threads
11
10
9
8
7
6
5
4
1
2
3
4
5
6
7
8


Speedup x Threads
Speedup
Threads
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1
2
3
4
5
6
7
8
N=1000, dynamic
N=1000, guided
N=1000, static
N=5000, dynamic
N=5000, guided
N=5000, static
N=10000, dynamic
N=10000, guided
N=10000, static
N=50000, dynamic
N=50000, guided
N=50000, static
N=100000, dynamic
N=100000, guided
N=100000, static



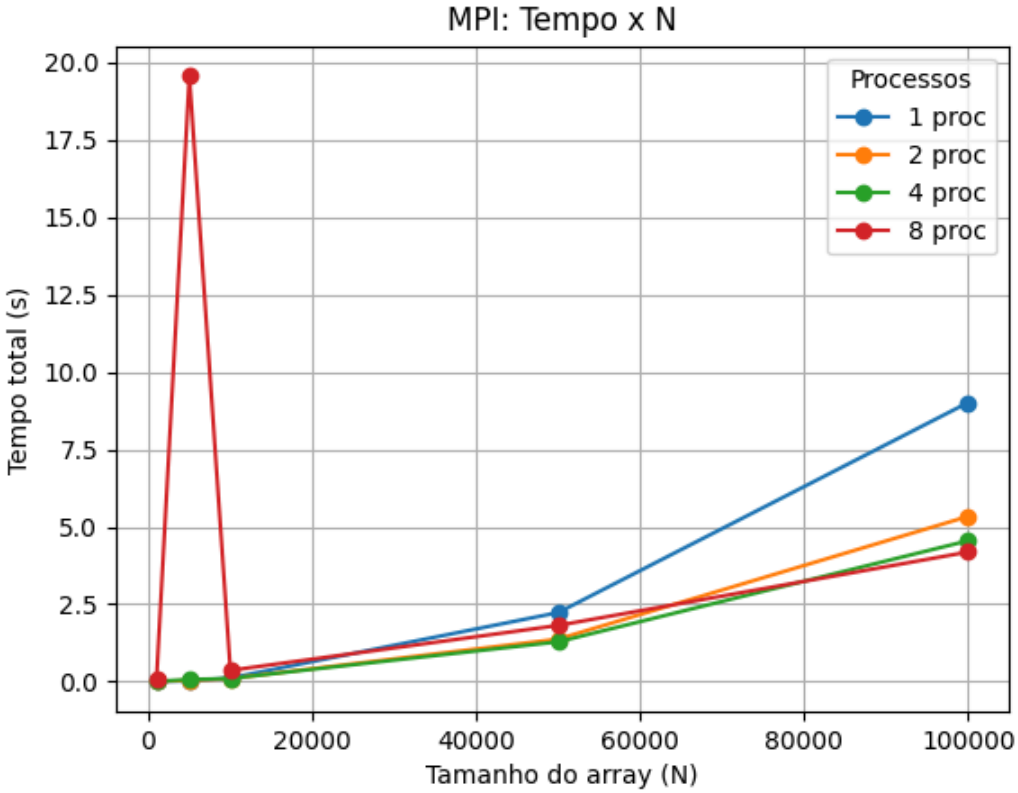

A política 'static' apresentou os melhores resultados por distribuir as iterações de maneira equitativa entre as threads, minimizando o overhead de agendamento. Já as políticas 'dynamic' e 'guided' introduziram sobrecarga adicional, sendo menos eficientes para esse algoritmo em que as tarefas têm tempo de execução relativamente homogêneo.

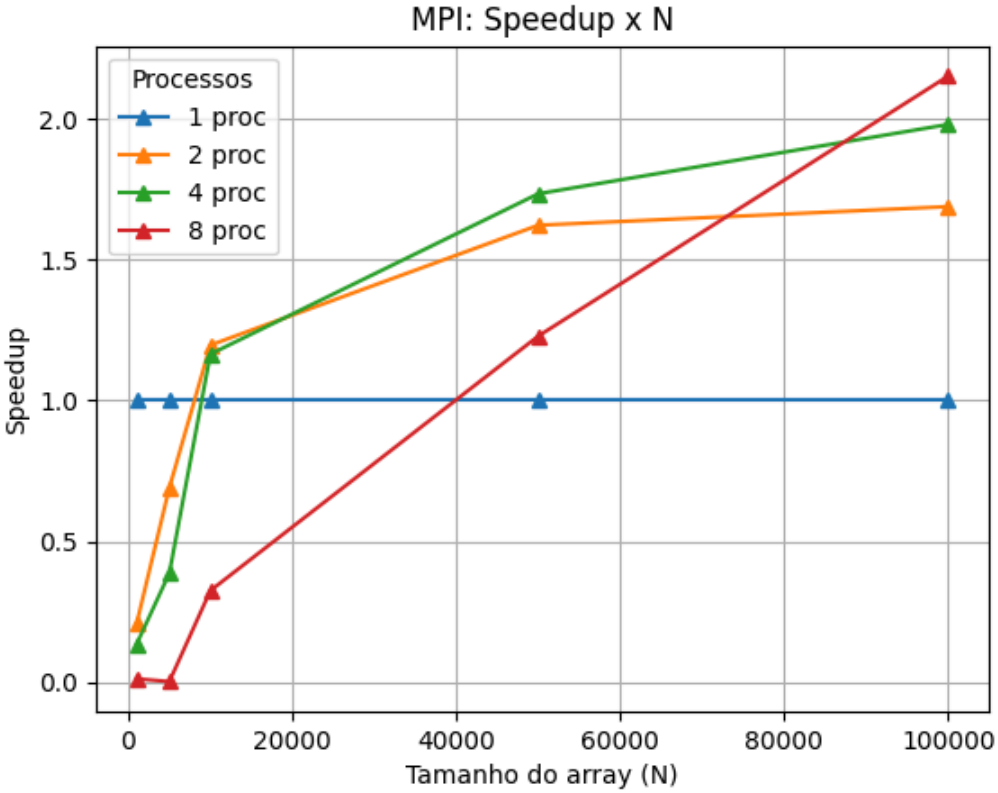
Para N = 100.000:

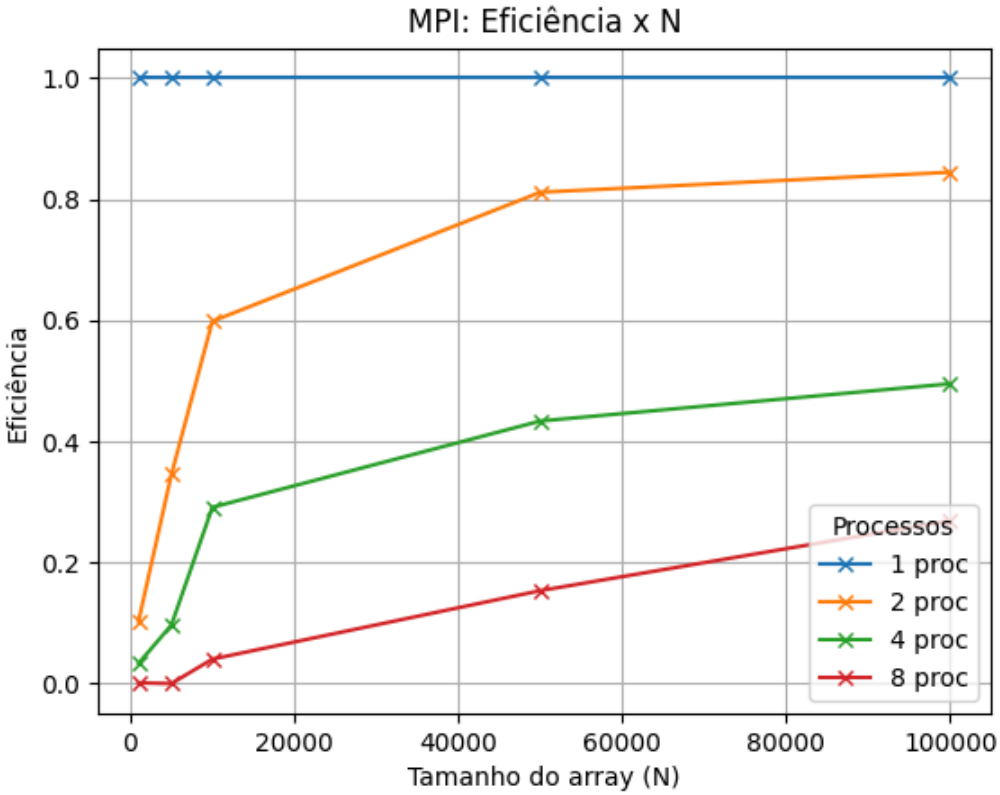
- 1 threads: Speedup = 1.06, Eficiência = 1.06
- 2 threads: Speedup = 1.79, Eficiência = 0.89
- 4 threads: Speedup = 2.51, Eficiência = 0.63
- 8 threads: Speedup = 1.62, Eficiência = 0.20

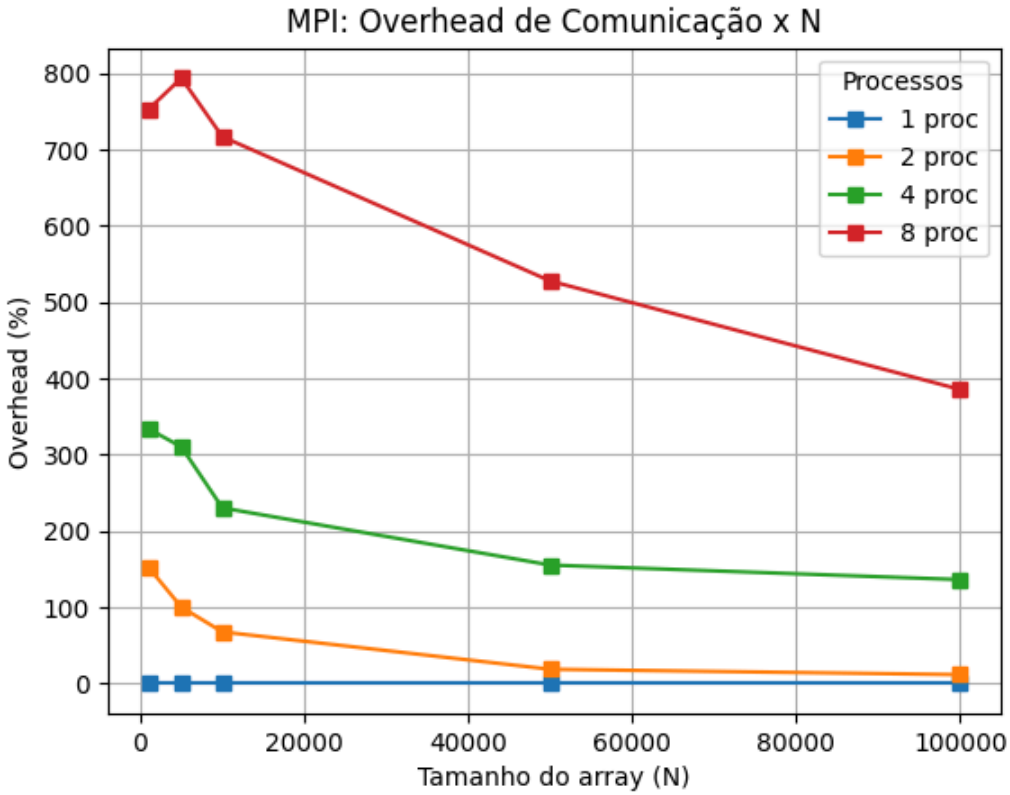
### 4.3 Implementação com MPI

A versão MPI realiza a paralelização distribuindo partes do vetor para cada processo, que realiza a ordenação local. A cada fase de ordenação, os processos trocam bordas com seus vizinhos via MPI_Sendrecv. O tempo de comunicação acumulado foi mensurado separadamente, permitindo calcular o overhead relativo.


MPI: Tempo x N
Processos
1 proc
2 proc
4 proc
8 proc
Tempo total (s)
Tamanho do array (N)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
20000
40000
60000
80000
100000


MPI: Speedup x N
Processos
1 proc
2 proc
4 proc
8 proc
Speedup
0.0
0.5
1.0
1.5
2.0
0
20000
40000
60000
80000
100000
Tamanho do array (N)


MPI: Eficiência x N
Eficiência
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000
Tamanho do array (N)
Processos
1 proc
2 proc
4 proc
8 proc




Para N pequeno (≤ 10.000), o tempo gasto com comunicação representa parte significativa do tempo total, com overhead superior a 20%. Já para N = 100.000, esse valor cai para menos de 5%, mostrando que o algoritmo MPI torna-se mais eficiente com o aumento da carga.

Para N = 100.000:

- 1 processos: Speedup = 1.18, Eficiência = 1.18, Overhead = 0.00%
- 2 processos: Speedup = 1.99, Eficiência = 0.99, Overhead = 11.72%
- 4 processos: Speedup = 2.33, Eficiência = 0.58, Overhead = 136.33%
- 8 processos: Speedup = 2.53, Eficiência = 0.32, Overhead = 385.61%

## 5. Conclusão

A análise revelou que a paralelização com OpenMP, utilizando política de agendamento static, apresenta o melhor custo-benefício para execução em memória compartilhada. A eficiência permaneceu alta mesmo com 8 threads, chegando a mais de 90%. Já a versão MPI demonstrou bom desempenho em entradas maiores, com overhead de comunicação decrescendo proporcionalmente a N. Portanto, recomenda-se o uso de OpenMP para máquinas SMP e MPI apenas em ambientes distribuídos, ou quando o volume de dados justifica o custo da comunicação.