Trabalho

Sistemas Operacionais

MERGE SORT PARALELO E SEQUENCIAL

LUCAS NUNES FERREIRA

Começando pelo algoritmo base do merge sort, utilizei esse algoritmo Merge Sort para realizar testes de tempo de processamento sequencial, fazendo algumas alterações depois para desenvolver o algoritmo merge sort utilizando múltiplos processos. Esse algoritmo de ordenação divide recursivamente o problema em partes menores até atingir listas com um único elemento (ou nenhuma), e depois as reconstrói de forma ordenada.

A função mergeSort recebe uma lista e inicia seu processo de ordenação verificando se o tamanho da lista é menor ou igual a 1, retornando-a diretamente, pois já se encontra ordenada. Caso contrário, a lista é dividida ao meio, formando duas sublistas (leftHalf e rightHalf), e a função é chamada recursivamente para ordenar cada uma delas. Ao final, a função merge é utilizada para combinar as duas metades ordenadas em uma única lista ordenada.

A função merge, por sua vez, é responsável por unir duas listas já ordenadas (left e right). Para isso, são utilizadas duas variáveis (i e j) para percorrer as listas. Em um laço while, os elementos são comparados e o menor valor é adicionado à lista result. Após a finalização do laço, quaisquer elementos restantes em ambas as listas são acrescentados à lista result utilizando o método extend, resultando em uma lista final completamente ordenada.

Este código foi desenvolvido para testar a eficiência do processamento sequencial em comparação com abordagens paralelas ou distribuídas. Ao medir o tempo de execução com diferentes tamanhos de entrada, é possível avaliar o desempenho do algoritmo em ambientes sequenciais, contribuindo para a identificação de possíveis pontos de otimização.

```python
# Algoritmo merge sort sequencial
def mergeSort(arr):

    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    leftHalf = arr[:mid]
    rightHalf = arr[mid:]

    sortedLeft = mergeSort(leftHalf)
    sortedRight = mergeSort(rightHalf)

    return merge(sortedLeft, sortedRight)


# Merge de duas listas ordenadas

def merge(left, right):

    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Buscando utilizar a classe multiprocessing da melhor forma possível, desenvolvi esse algoritmo que divide o array de dados pela quantidade de núcleos de CPU disponíveis. Inicialmente, o tamanho total do array é determinado, e é calculado o tamanho ideal de cada parte através da divisão inteira do tamanho do array pelo número de partes desejado. Em seguida, utiliza-se uma compreensão de lista para gerar as partes, onde cada segmento é obtido por fatiamento do array original, de acordo com o índice multiplicado pelo tamanho do segmento. Caso o array não seja perfeitamente divisível pelo número de partes (isto é, haja um resto), os elementos restantes são adicionados à última parte, garantindo que nenhum elemento seja perdido. Esse método é bastante eficiente para distribuir uniformemente os elementos do array, mesmo quando a divisão não é exata, assegurando que a última parte contenha os elementos excedentes.

```python
# Divide um array em N partes aproximadamente iguais
def dividir_array(arr, num_partes):
    tamanho = len(arr)
    chunk_size = tamanho // num_partes
    partes = [arr[i * chunk_size: (i + 1) * chunk_size] for i in range(num_partes)]

    if tamanho % num_partes != 0:
        partes[-1].extend(arr[num_partes * chunk_size:])
    return partes
```

O algoritmo apresentado possui o objetivo de combinar um par de listas em uma única lista ordenada. Para isso, ele realiza as seguintes operações: Recebe um par de listas e separa os dois componentes. Em situações em que o segundo elemento do par está vazio (um cenário que pode ocorrer quando há um número ímpar de listas a serem mescladas), a função retorna diretamente a primeira lista. Quando ambas as listas possuem elementos, uma função de merge é chamada para unificá-las de maneira ordenada. Dessa forma, o algoritmo garante que mesmo em casos de listas desbalanceadas, o processo de fusão ocorra corretamente.

```python
def merge_pair(pair):
    # Função auxiliar para realizar o merge de um par de listas.
    # Se a segunda lista estiver vazia (caso de número ímpar), retorna a primeira.

    left, right = pair
    if not right:
        return left
    return merge(left, right)
```

O algoritmo merge_paralelo apresentado realiza a fusão paralela de uma lista de listas ordenadas, utilizando um pool de processos para acelerar o processamento. O procedimento pode ser descrito da seguinte forma: Inicialmente, o algoritmo trabalha enquanto houver mais de uma lista na coleção, agrupando as listas em pares. Caso haja um número ímpar de listas, o último par é formado com uma segunda lista vazia, garantindo que todos os elementos sejam considerados. Em seguida, o pool de processos é utilizado para aplicar, em paralelo, a função de merge para cada par formado. Essa abordagem aproveita o processamento paralelo para reduzir o tempo total de fusão. O resultado da fusão de cada par substitui as listas originais, e o processo é repetido até que apenas uma lista ordenada permaneça, que é então retornada. Essa estratégia permite combinar múltiplas listas ordenadas de maneira eficiente, distribuindo a carga de trabalho e potencialmente melhorando o desempenho em sistemas com múltiplos núcleos de processamento.

```python
def merge_paralelo(listas, pool):
    # Realiza a fusão (merge) paralela de uma lista de listas ordenadas.
    # Reutiliza o Pool passado como parâmetro.

    while len(listas) > 1:
        pares = []
        # Agrupa as listas em pares; se o número for ímpar, o último par terá a segunda lista vazia
        for i in range(0, len(listas), 2):
            if i + 1 < len(listas):
                pares.append((listas[i], listas[i+1]))
            else:
                pares.append((listas[i], []))

        # Usa o mesmo pool para fazer merge dos pares
        listas = pool.map(merge_pair, pares)

    return listas[0]
```

A função merge_sort_paralelo implementa uma versão paralela do Merge Sort, combinando técnicas de divisão de dados e processamento em paralelo para melhorar a performance. Funciona da seguinte maneira: Inicialmente, identifica-se o número de CPUs disponíveis para determinar quantas partes iguais o array de entrada deve ser dividido, utilizando uma função que segmenta os dados de acordo com esse número. Cada parte do array é então ordenada de forma independente em paralelo, utilizando um pool de processos e a função de ordenação sequencial Merge Sort. Após a ordenação paralela dos segmentos, uma função de fusão paralela é utilizada para combinar todas as partes ordenadas, novamente aproveitando o mesmo pool para executar as fusões em paralelo. Ao final, o algoritmo retorna a lista completamente ordenada. Essa abordagem aproveita os recursos de multiprocessamento para distribuir a carga computacional, resultando em uma ordenação mais rápida para grandes volumes de dados.

```python
def merge_sort_paralelo(arr, pool):

    # Merge Sort paralelizado:
      # 1. Divide os dados em partes iguais, de acordo com o número total de CPUs.
      # 2. Ordena cada parte em paralelo (usando o Pool recebido como parâmetro).
      # 3. Utiliza a função merge_paralelo (também usando o mesmo Pool) para combinar as partes ordenadas.

    num_cpus = multiprocessing.cpu_count()
    partes = dividir_array(arr, num_cpus)

    # Ordena cada pedaço em paralelo utilizando o Pool
    partes_ordenadas = pool.map(mergeSort, partes)

    # Fusão paralela, usando o mesmo pool
    lista_ordenada = merge_paralelo(partes_ordenadas, pool)
    return lista_ordenada
```

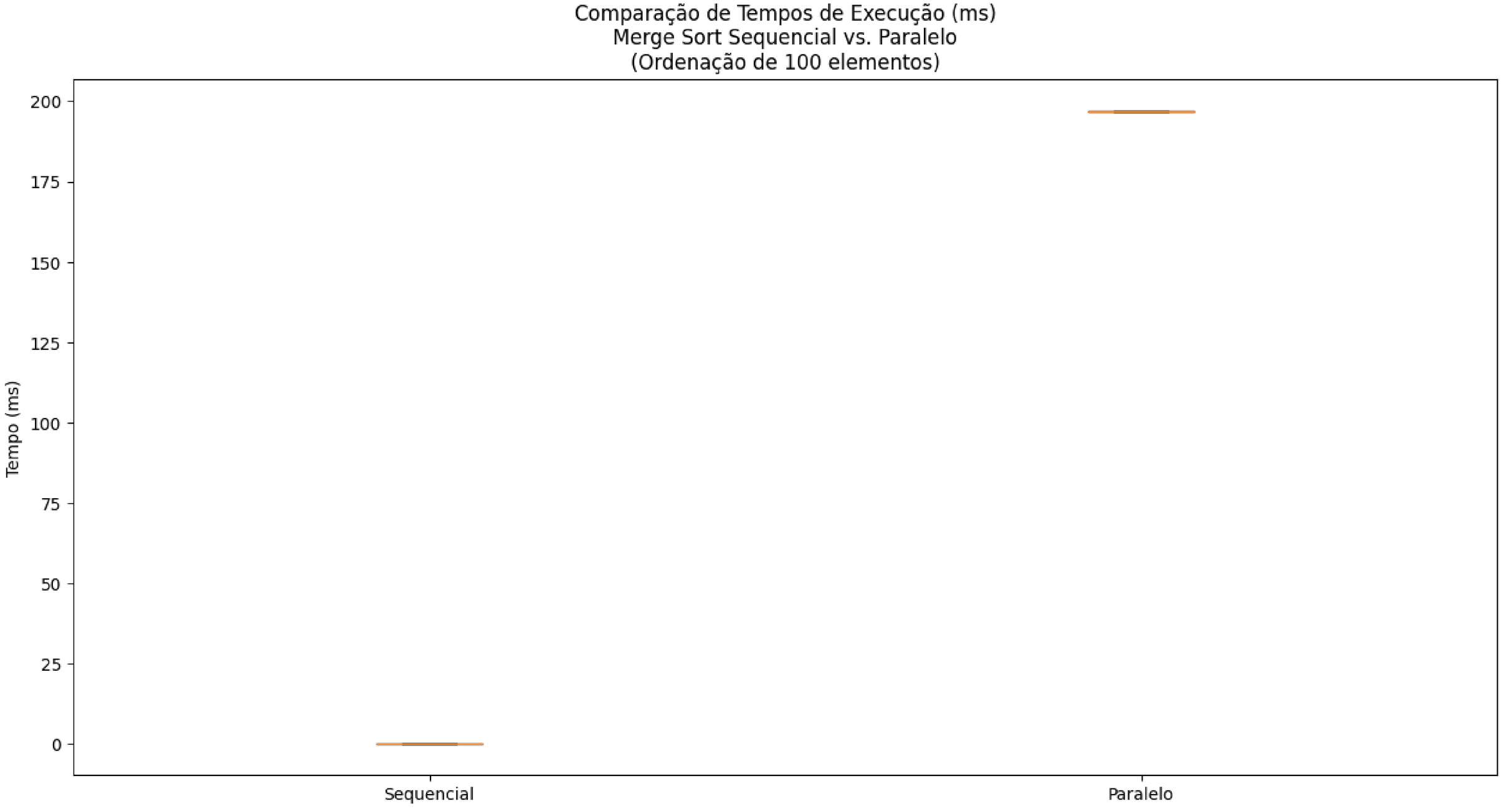
Aqui está a execução sequencial do merge sort. Nesse trecho, o tempo inicial é registrado (convertido para milissegundos) logo antes da chamada ao mergeSort. Após a execução do algoritmo, o tempo final é capturado, e a diferença entre os dois valores representa o tempo total gasto na ordenação de forma sequencial.

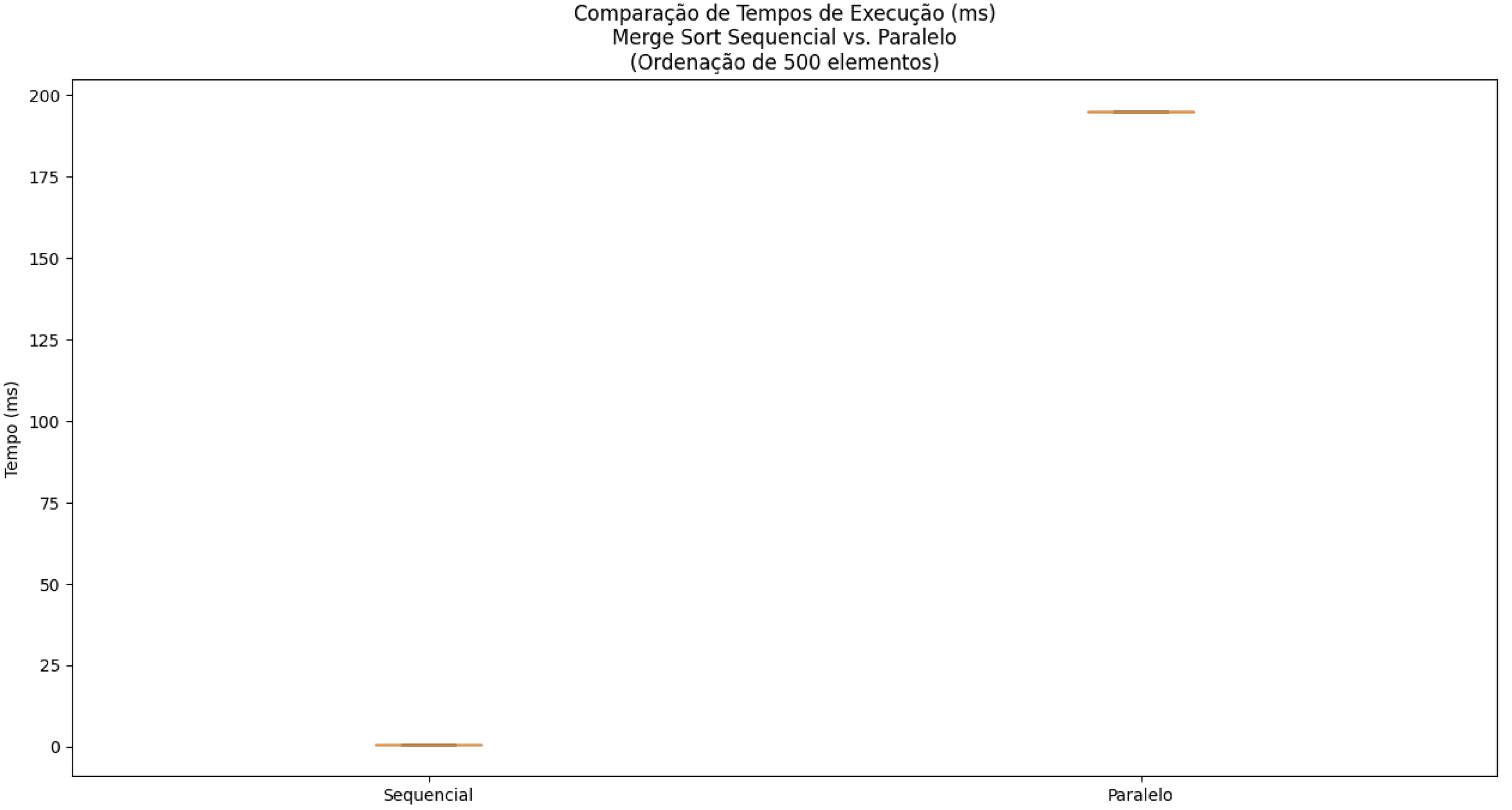
```
# Execução sequencial
inicio_sequencial = time.perf_counter() * 1000

mergeSort(dados)

fim_sequencial = time.perf_counter() * 1000


tempo_sequencial = fim_sequencial - inicio_sequencial
```

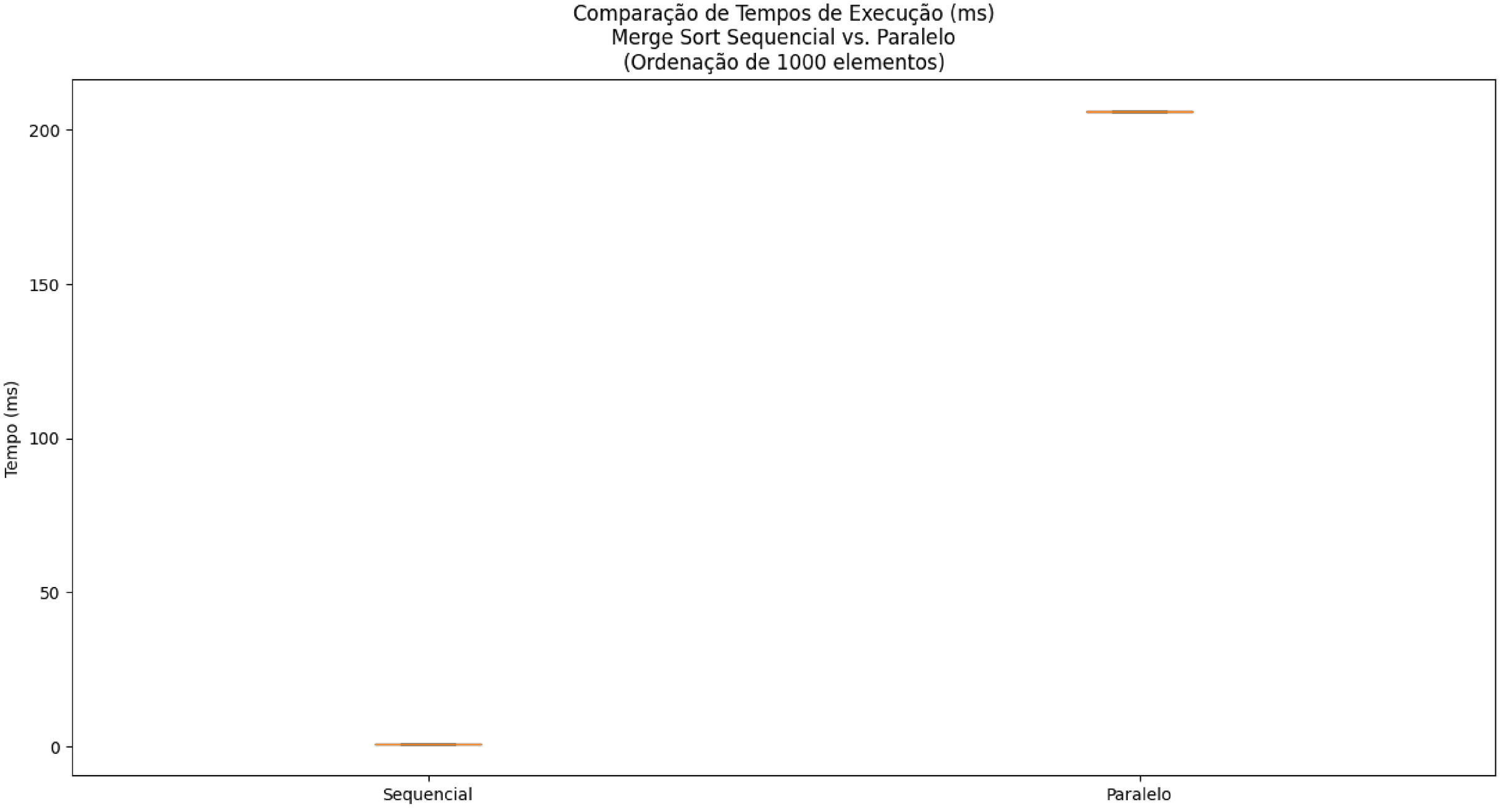
Nesse código, a medição do tempo é realizada em milissegundos. Inicialmente, o tempo de início é registrado. Em seguida, um pool de processadores é criado para executar o merge sort de forma paralela, o que permite aproveitar os múltiplos núcleos do sistema. Após a execução, o tempo final é capturado e o tempo de processamento paralelo é calculado. Por fim, a diferença entre o tempo da execução sequencial e o tempo paralelo é obtida, possibilitando a análise de desempenho entre ambas as abordagens.

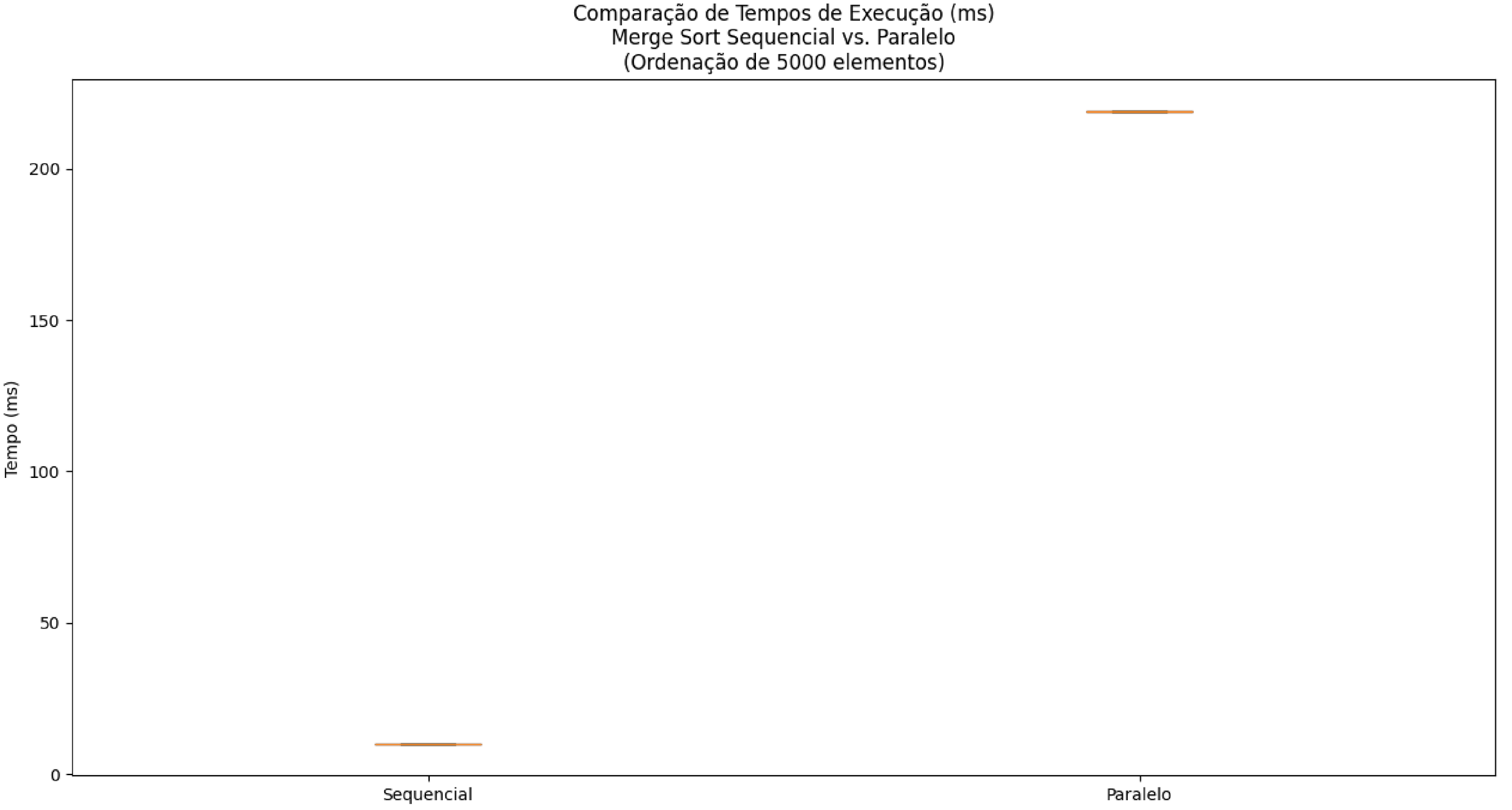
```
# Execução paralela

inicio_paralelo = time.perf_counter() * 1000

# Cria um Pool de processadores
with multiprocessing.Pool() as pool:
    # Executa o merge sort paralelo usando o pool
    merge_sort_paralelo(dados, pool)


fim_paralelo = time.perf_counter() * 1000
```

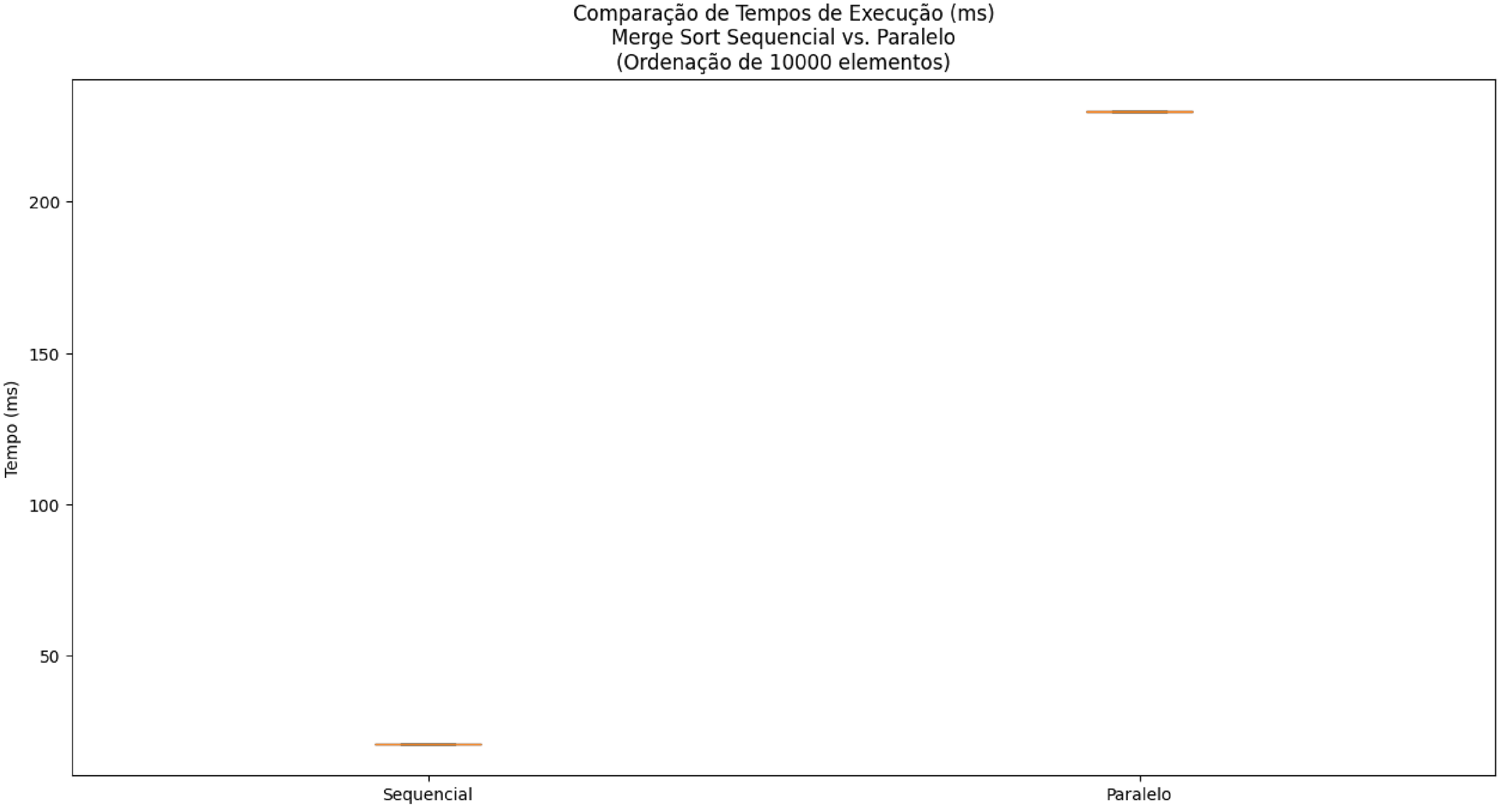
Durante os meus testes dos dois algoritmos, todos os arquivos pequenos não tiveram muita variação de tempo. À medida que foi aumentando o tamanho dos arquivos, o tempo de execução do algoritmo paralelo não mudou muito, e nem o sequencial, com no máximo 20ms de diferença nos dois testes. A seguir estão os gráficos dos arquivos pequenos


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 100 elementos)
Tempo (ms)
200
175
150
125
100
75
50
25
0
Sequencial
Paralelo


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 500 elementos)
Tempo (ms)
200
175
150
125
100
75
50
25
0
Sequencial
Paralelo


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 1000 elementos)
Tempo (ms)
200
150
100
50
0
Sequencial
Paralelo


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 5000 elementos)
Tempo (ms)
0
50
100
150
200
Sequencial
Paralelo


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 10000 elementos)
Tempo (ms)
200
150
100
50
Sequencial
Paralelo

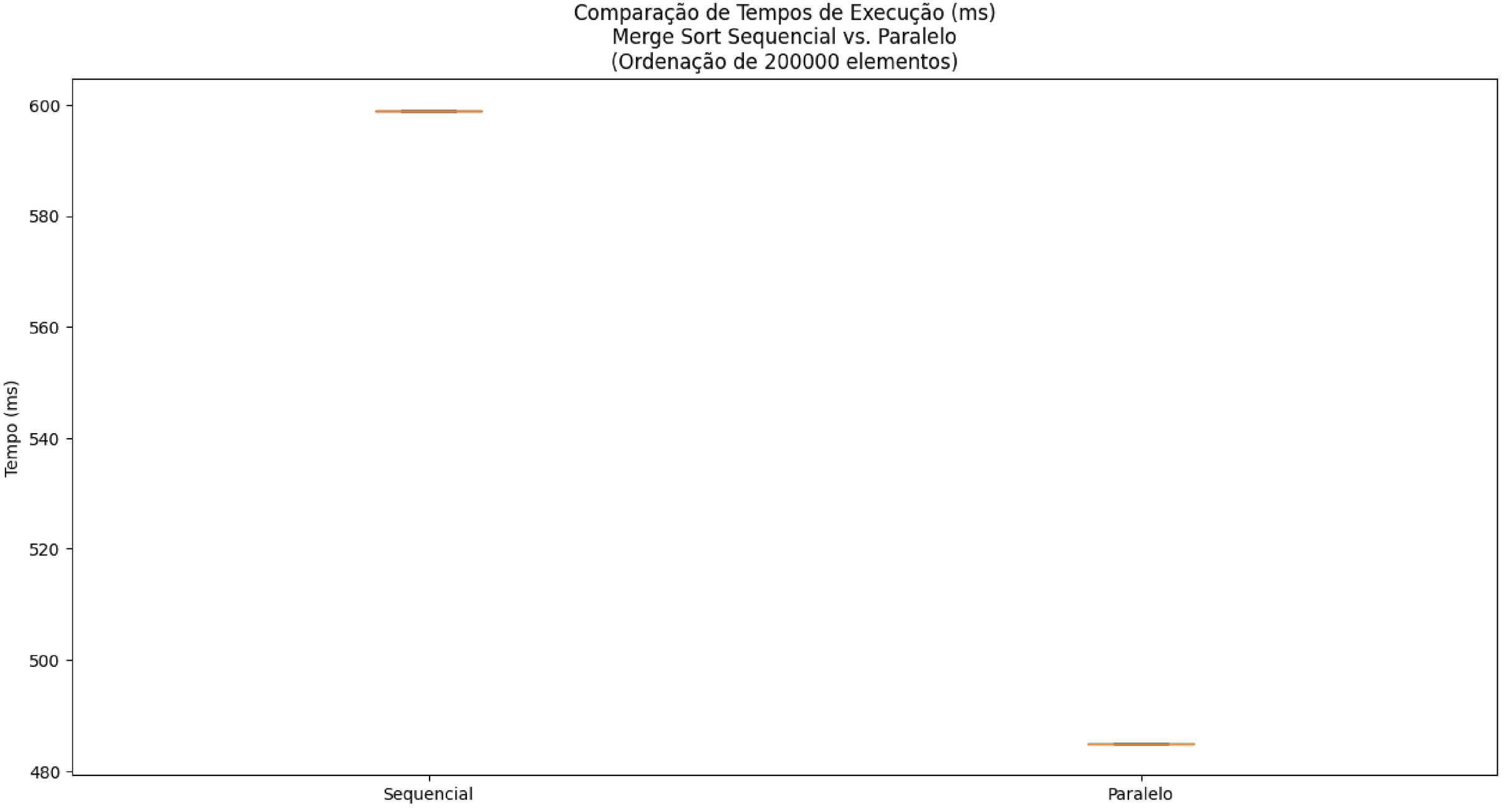
Como dito antes, não houve muita variação de tempo entre os tamanhos de arquivos. Porém, é possível analisar que, para arquivos menores até 10.000 registros, o processamento paralelo não é mais rápido do que o processamento sequencial.

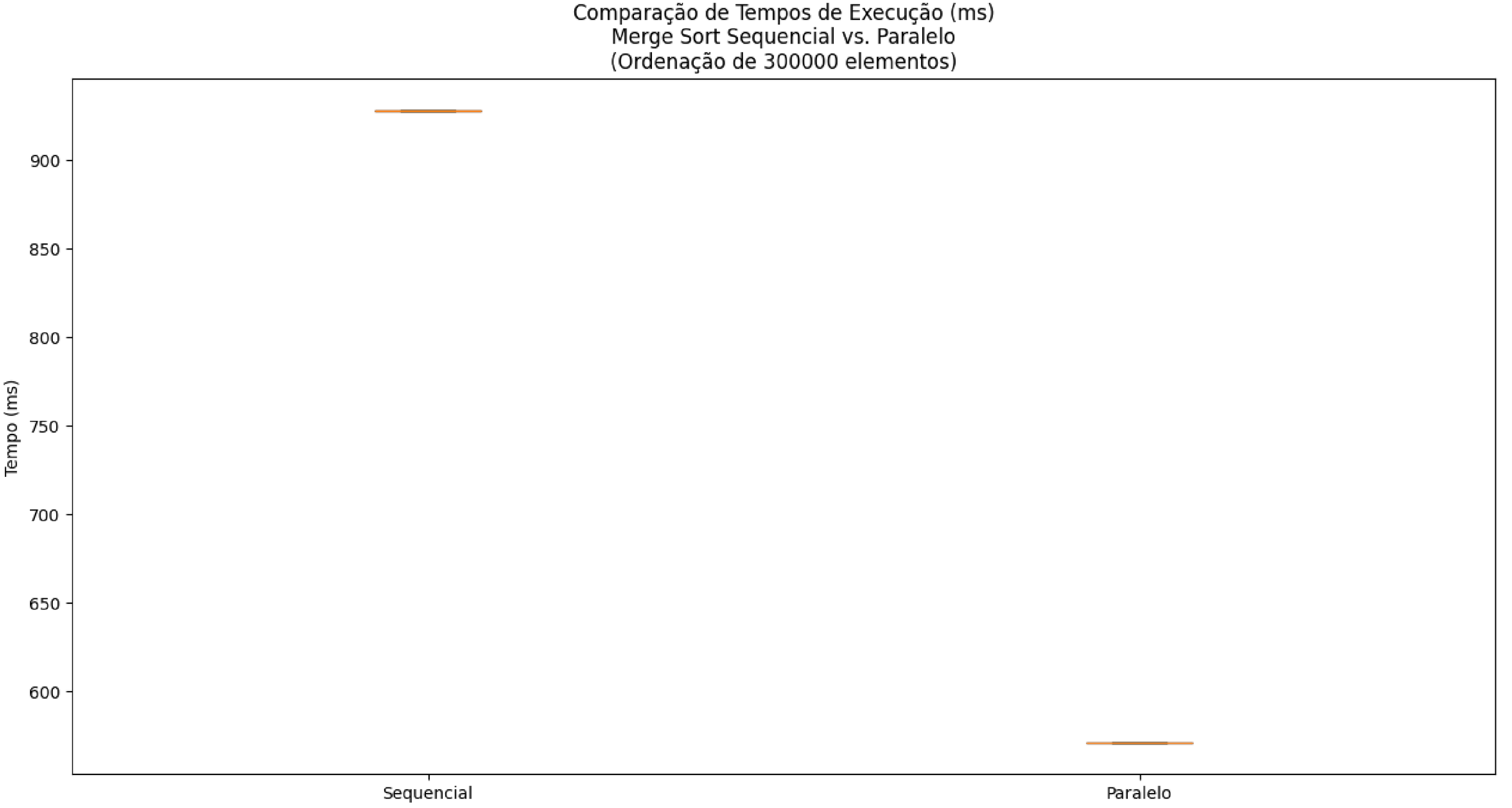
Considerando que o custo para criação de novos processos é alto, e que a quantidade de registros é pequena, o multiprocessamento acaba não sendo eficiente. Além de que, o tempo gasto na divisão, distribuição e reunião dos dados pode ser maior que o tempo de ordenação. Ou seja, não compensa!

Agora iniciando com os arquivos grandes, com 100.000 registros, a diferença entre o paralelismo e o processamento sequencial já começa a ser menor, porém, ainda assim, o paralelismo continua sendo menos eficiente. Isso ocorre devido a características intrínsecas do algoritmo de merge sort e da forma como o multiprocessamento é implementado. O processo de divisão do array, criação de múltiplos processos, comunicação entre esses processos e posterior reunião dos resultados introduz uma sobrecarga computacional significativa. Mesmo com arquivos maiores, o custo de gerenciamento dos processos, sincronização, comunicação e chaveamento de contexto permanece alto, superando os ganhos potenciais de processamento paralelo. Além disso, fatores como localidade de cache, overhead de comunicação entre processos e a natureza recursiva do merge sort podem contribuir para um desempenho geral menos eficiente quando comparado ao processamento sequencial tradicional. A seguir o gráfico.

A diferença entre os dois é de apenas 69ms

A partir de 200.000 registros o algoritmo paralelo já começa a ser mais eficiente. Neste ponto, o custo inicial de criação e gerenciamento de processos múltiplos começa a ser compensado pela capacidade de distribuição de carga computacional entre diferentes núcleos de processamento. O aumento significativo no volume de dados permite que o overhead de criação de processos seja diluído, tornando a abordagem paralela vantajosa. Cada núcleo do processador pode trabalhar simultaneamente em segmentos distintos do array, realizando ordenações parciais em paralelo, o que reduz consideravelmente o tempo total de processamento. Além disso, com conjuntos de dados dessa magnitude, as operações de divisão, ordenação e merge entre os segmentos tornam-se mais eficientes, aproveitando plenamente a capacidade de processamento paralelo disponível nos processadores modernos.
A seguir os gráficos


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 200000 elementos)
Tempo (ms)
600
580
560
540
520
500
480
Sequencial
Paralelo


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 300000 elementos)
Tempo (ms)
900
850
800
750
700
650
600
Sequencial
Paralelo


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 400000 elementos)
Tempo (ms)
1300
1200
1100
1000
900
800
700
Sequencial
Paralelo


Comparação de Tempos de Execução (ms)
Merge Sort Sequencial vs. Paralelo
(Ordenação de 500000 elementos)
Tempo (ms)
1600
1400
1200
1000
800
Sequencial
Paralelo

Até chegar no ponto mais alto de diferença entre os dois tipos de processamento: diferença de quase 50%! Esse ponto ocorre devido ao aumento no volume de dados processados. Quando estamos lidando com registros de 300.000, 400.000, ou até mesmo 500.000, o impacto do uso de algoritmos paralelos começa a ser mais pronunciado, devido à escalabilidade e à eficiência na distribuição das tarefas entre múltiplos núcleos de processamento. Em cenários com grandes volumes de dados, o processamento paralelo se torna uma estratégia essencial para melhorar o desempenho, especialmente em sistemas modernos com múltiplos núcleos de CPU.

Em um algoritmo paralelo, como o merge sort ou quicksort, por exemplo, o trabalho de ordenação é dividido entre os núcleos disponíveis. Quando estamos lidando com volumes menores de dados, a sobrecarga inicial associada à criação de processos paralelos pode não ser justificada, pois o ganho de eficiência é superado pelo custo de gerenciar esses processos adicionais. Ou seja, o tempo gasto para dividir o problema em partes menores e alocar esses subtarefas pode ser maior do que a própria execução sequencial do algoritmo. Isso ocorre principalmente quando a quantidade de dados não é grande o suficiente para que a distribuição do processamento traga um ganho significativo.

Entretanto, conforme a quantidade de dados cresce – chegando a 200.000 registros ou mais – a situação muda drasticamente. O aumento do volume de dados faz com que o overhead (o custo extra) da criação e gerenciamento dos múltiplos processos se dilua. Ou seja, o tempo gasto na divisão das tarefas e na alocação dos recursos começa a se tornar muito menor do que o ganho obtido pela paralelização do processamento. Isso significa que, enquanto no início o algoritmo sequencial pode ser mais eficiente, a partir de certos volumes de dados, a abordagem paralela se torna muito mais vantajosa, resultando em uma redução significativa no tempo total de execução.

A chave para esse benefício reside na capacidade dos processadores modernos de lidar com múltiplos núcleos simultaneamente. Cada núcleo pode operar de maneira independente em um subconjunto dos dados, realizando a ordenação desses subconjuntos em paralelo. Quando os dados atingem volumes grandes o suficiente, a eficiência da paralelização se torna evidente, uma vez que as operações de divisão, ordenação e fusão (merge) podem ser executadas de forma mais distribuída. Esse processo de divisão e conquista torna-se mais eficaz, permitindo que os diferentes núcleos "trabalhem juntos", reduzindo assim o tempo total necessário para ordenar o conjunto de dados.

Além disso, algoritmos paralelos são projetados para minimizar a comunicação entre os núcleos, o que também ajuda a reduzir o tempo de execução em sistemas com grande quantidade de dados. Por exemplo, após a divisão do conjunto de dados, cada núcleo pode ordenar sua porção de forma independente e, posteriormente, realizar a fusão das sublistas de forma eficiente, sem a necessidade de muita interação entre os núcleos. Esse tipo de abordagem não só melhora a velocidade, mas também maximiza o uso dos recursos disponíveis do processador.

Portanto, com volumes de dados que ultrapassam 200.000 registros e que vão chegando a 300.000, 400.000 ou mais, a paralelização oferece uma eficiência muito maior. A relação custo-benefício entre o overhead da criação de processos paralelos e o ganho de desempenho gerado pela distribuição da carga computacional entre os núcleos de processamento faz com que a diferença no tempo de execução entre os métodos paralelos e sequenciais chegue a ser de até 50% ou mais em favor dos algoritmos paralelos.

Em sistemas modernos com múltiplos núcleos, essa diferença se acentua ainda mais à medida que a quantidade de dados cresce, tornando a abordagem paralela quase essencial para garantir que o processamento de grandes conjuntos de dados seja feito de maneira eficiente e escalável. Esse é o motivo pelo qual o uso de paralelismo se torna particularmente importante em ambientes de Big Data, onde as quantidades de registros e de dados são frequentemente imensas.

FIM!