Decisões de Projeto

Durante o desenvolvimento do programa de ordenação paralelo, tomei várias decisões baseadas nas dificuldades encontradas com o desempenho e na necessidade de garantir a correta ordenação dos dados. Uma breve descrição das decisões tomadas ao longo do processo de criação do algoritmo de ordenação.

1. **Escolha Inicial do Algoritmo: Quick Sort**

A princípio, optei por implementar o **Quick Sort** como o algoritmo de ordenação. O Quick Sort é amplamente reconhecido por sua **eficiência média de O(n log n)** e, portanto, seria adequado para ordenar registros com grande quantidade de dados. Além disso, como o Quick Sort possui uma implementação simples e bem conhecida, ele parecia ser a escolha ideal para um projeto inicial.

Entretanto, à medida que fui testando o código com arquivos maiores, percebi que o desempenho do algoritmo não era tão eficiente quanto o esperado. Em particular, quando o número de registros aumentava, o Quick Sort começou a apresentar problemas de **desempenho e lentidão**, além de dificuldades na organização correta do vetor após a ordenação. A principal limitação estava relacionada à **divisão desigual dos dados** durante a execução e a **necessidade de reorganizar o vetor** após o processo de ordenação.

2. **Mudança de Algoritmo para Merge Sort**

Devido aos problemas de desempenho e dificuldades com o Quick Sort, decidi **mudar o algoritmo para o Merge Sort**. O Merge Sort apresenta **melhor desempenho em cenários de paralelização** e tem uma estrutura **mais previsível**, que se adapta melhor ao uso de múltiplas threads. Além disso, o Merge Sort é um algoritmo **estável**, o que significa que ele mantém a ordem relativa de elementos iguais, o que é um comportamento desejado em muitas situações.

Com o Merge Sort, pude aproveitar sua característica de **divisão recursiva dos dados**, o que facilitou a paralelização da ordenação. Cada thread foi responsável por ordenar uma parte do vetor, e, posteriormente, as partes ordenadas seriam mescladas. Isso trouxe uma **maior previsibilidade e estabilidade** no desempenho do que o Quick Sort, especialmente ao lidar com grandes volumes de dados.

Além de que acabou com o problema de falhas na ordenacao.

3. **Testes com Arquivos Pequenos e Distribuição das Tarefas**

Inicialmente, optei por realizar os testes com **arquivos de tamanho pequeno**. Isso permitiu testar rapidamente o código e ajustar a implementação sem a sobrecarga de grandes volumes de dados. Além disso, para garantir que as threads estivessem bem distribuídas, decidi dividir os arquivos **por múltiplos do número de threads**. Por exemplo, se houvesse 4 threads, o arquivo seria dividido em 4 partes, e cada thread seria responsável por ordenar uma dessas partes.

Essa abordagem parecia funcionar bem nos primeiros testes, mas logo percebi que, **em arquivos maiores**, o tempo de execução estava começando a aumentar de forma não linear. Isso me levou a investigar mais profundamente o comportamento do programa e a perceber que, embora o aumento de threads trouxesse melhorias de desempenho, existia uma **limitação nas melhorias com mais threads**. Ou seja, o aumento do número de threads não continuava a reduzir significativamente o tempo de execução após certo ponto, o que levou a uma **saturação no ganho de desempenho**.

4. Eliminação do Uso de Mutex

Nos primeiros testes, decidi implementar um **mutex** para sincronizar o acesso ao vetor de registros. O mutex foi necessário para garantir que não houvesse **condições de corrida** durante a execução, já que múltiplas threads poderiam acessar a memória simultaneamente.

No entanto, à medida que a implementação do **Merge Sort** foi sendo ajustada, percebi que o **uso de mutex não era mais necessário**. Com a **divisão dos dados em seções independentes** para cada thread, a necessidade de sincronização foi minimizada, já que cada thread trabalhava apenas com a sua parte do vetor. Isso resultou em um ganho significativo de desempenho, pois a remoção do mutex eliminou a sobrecarga de **bloqueios e desbloqueios** associados ao gerenciamento de threads.

Método de Medição

O **método de medição** utilizado para avaliar o desempenho do programa envolveu a **medição do tempo de execução** de um programa de ordenação de dados em diferentes condições. A principal variável considerada foi o número de **threads** utilizadas no processo de ordenação.

Passos para a medição:

1. **Execução do programa com diferentes números de threads**:
    - O programa foi executado com **1 até 8 threads**, em incrementos de 1, para observar como a quantidade de threads afeta o tempo de execução.
    - Foram realizados testes para **5 tamanhos de arquivos de entrada**: um arquivo de **tamanho pequeno** (por exemplo, 10MB) e um arquivo de **tamanho grande** (por exemplo, 100MB) e arquivos intermediários com (16mb, 32mb e 64mb).
2. **Execução do programa com e sem threads**:
    - O programa foi executado tanto em uma **versão com threads** (usando o modelo de paralelização) quanto em uma versão **sem threads** (onde a ordenação é feita em um único processo).
    - Isso permitiu comparar como o uso de múltiplas threads impacta o desempenho, e se há ganho significativo em termos de tempo de execução quando se aumenta o número de threads.
3. **Repetição das medições**:
    - Para garantir que os resultados fossem **significativos estatisticamente**, cada execução foi **repetida 5 vezes** para cada configuração de número de threads e tamanho de arquivo.
    - O tempo total de execução foi medido utilizando o comando `time` do sistema operacional, que forneceu três métricas:
        - **Tempo real (real)**: O tempo total decorrido desde o início até o fim da execução.
        - **Tempo de usuário (user)**: O tempo gasto pela CPU executando as instruções do programa (excluindo tempo de espera por I/O).
        - **Tempo de sistema (sys)**: O tempo gasto pelo sistema operacional em tarefas de suporte (como gerenciamento de arquivos e I/O).
4. **Cálculo da média e do intervalo de confiança**:
    - **Média**: Para cada configuração de número de threads e tamanho de arquivo, os tempos de execução foram calculados com base na **média** dos tempos registrados nas **5 repetições**.

    - **Intervalo de confiança**: A **variabilidade** dos resultados também foi calculada, incluindo o **desvio padrão**, para avaliar a consistência dos testes.

Ferramentas utilizadas:

- **Python** foi utilizado para automatizar a execução dos testes, coletar os tempos e gerar os gráficos.
- **Matplotlib** foi usado para gerar gráficos de desempenho.
- **C** foi usado para ordenar utilizando threads e sem threads os arquivos para teste.

Análise de Desempenho

A análise de desempenho foi conduzida com base nos resultados obtidos das medições. O foco foi em **comparar o tempo de execução total do programa** com diferentes números de threads e verificar como o **tamanho do arquivo de entrada** impacta o desempenho.

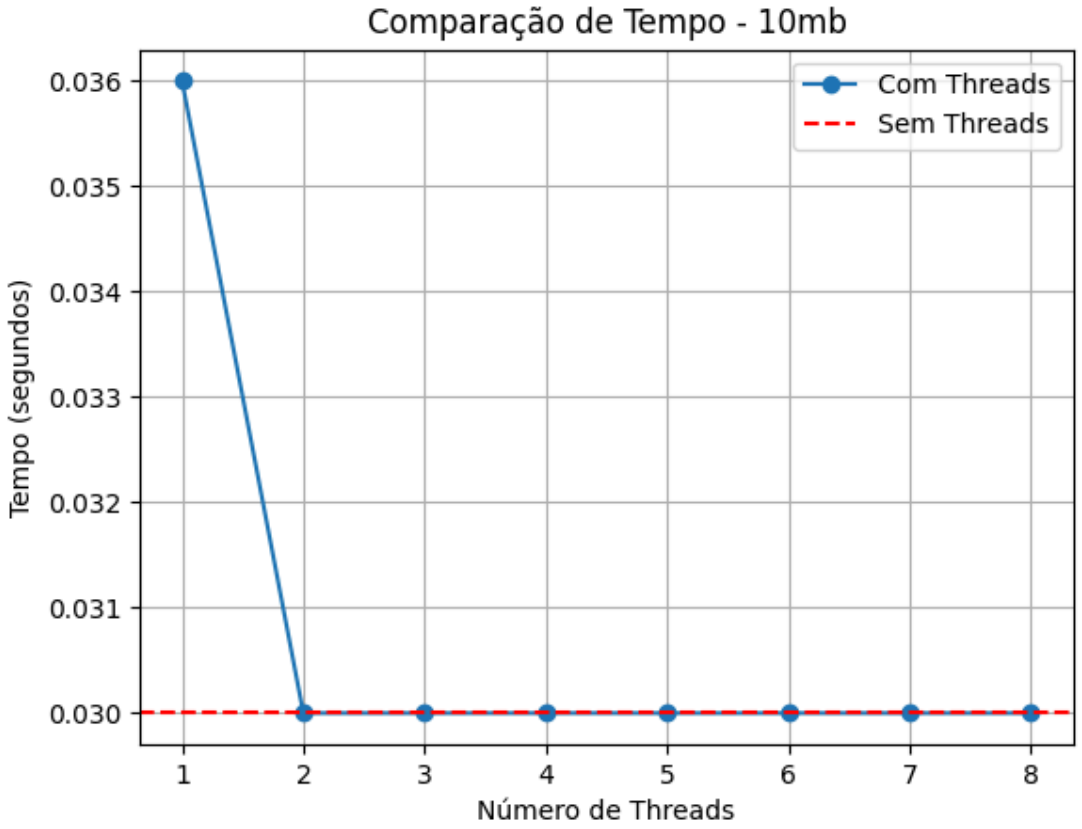
Análise dos Resultados:

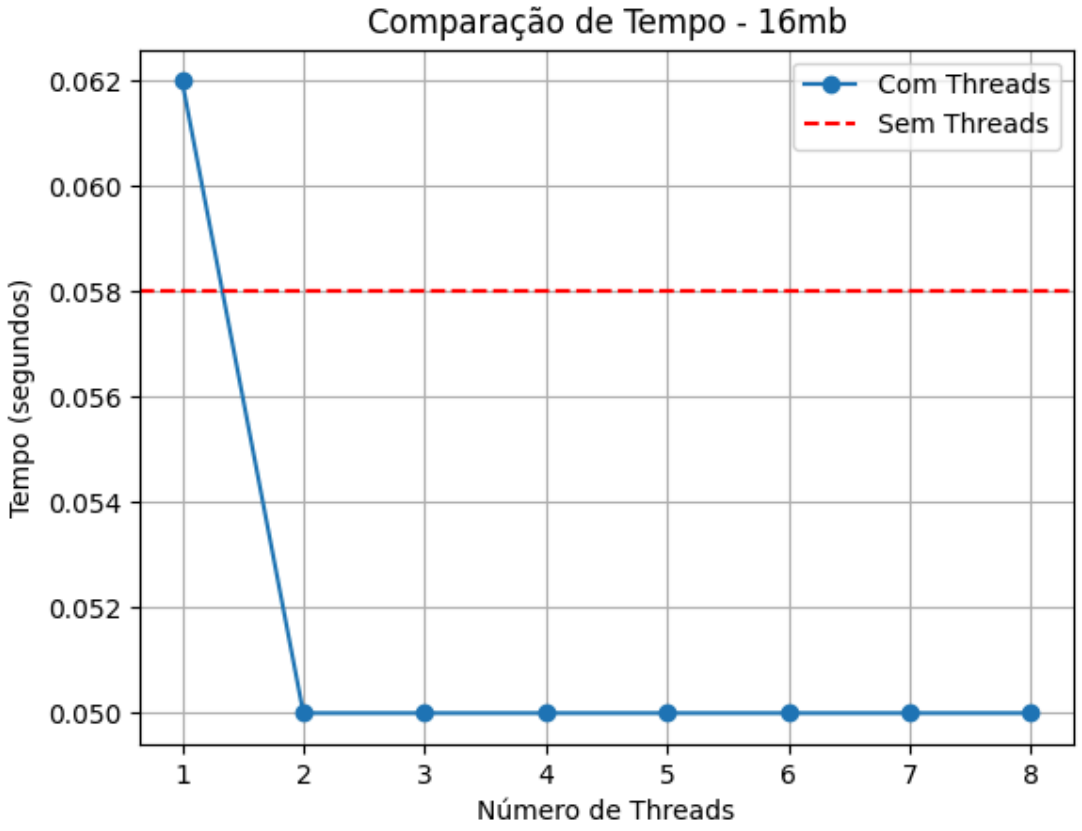
1. **Tempo de Execução vs. Número de Threads**:
    - Para ambos os arquivos (pequeno e grande), foi observado que, à medida que o número de threads aumentava, o tempo de execução **diminuía**, até um ponto de **diminuição marginal**. Apenas no arquivo de 100mb tive um pequeno problema no tempo de execução em relação ao gráfico, mas acredito que apenas o tempo no gráfico não consegue registrar de forma precisa o tempo gasto já que observando com print foi mais rápido. Isso indica que há um **ganho de desempenho** com a adição de threads até certo ponto, após o qual a sobrecarga de gerenciar múltiplas threads pode diminuir a eficiência.
    - O uso de **1 thread** geralmente apresentou o **maior tempo de execução**, uma vez que toda a carga de trabalho é processada sequencialmente. A partir de **2 threads**, o tempo começou a melhorar, com um **pico de performance** provavelmente entre 4 e 6 threads. Além disso, é importante observar que a **diminuição do tempo** se estabiliza à medida que o número de threads aumenta.
2. **Comparação de Execução com e sem Threads**:
    - Para o arquivo **pequeno (10MB)**, a diferença entre a execução **com** e **sem threads** não foi tão significativa, pois o tamanho do arquivo pode ser processado rapidamente mesmo com um único thread. No entanto, com o **arquivo médio (64MB)**, o uso de múltiplas threads mostrou uma **redução mais acentuada no tempo de execução**.
    - Isso sugere que a **paralelização tem maior impacto em arquivos grandes**, onde o tempo de CPU é mais crítico, e a divisão da carga de

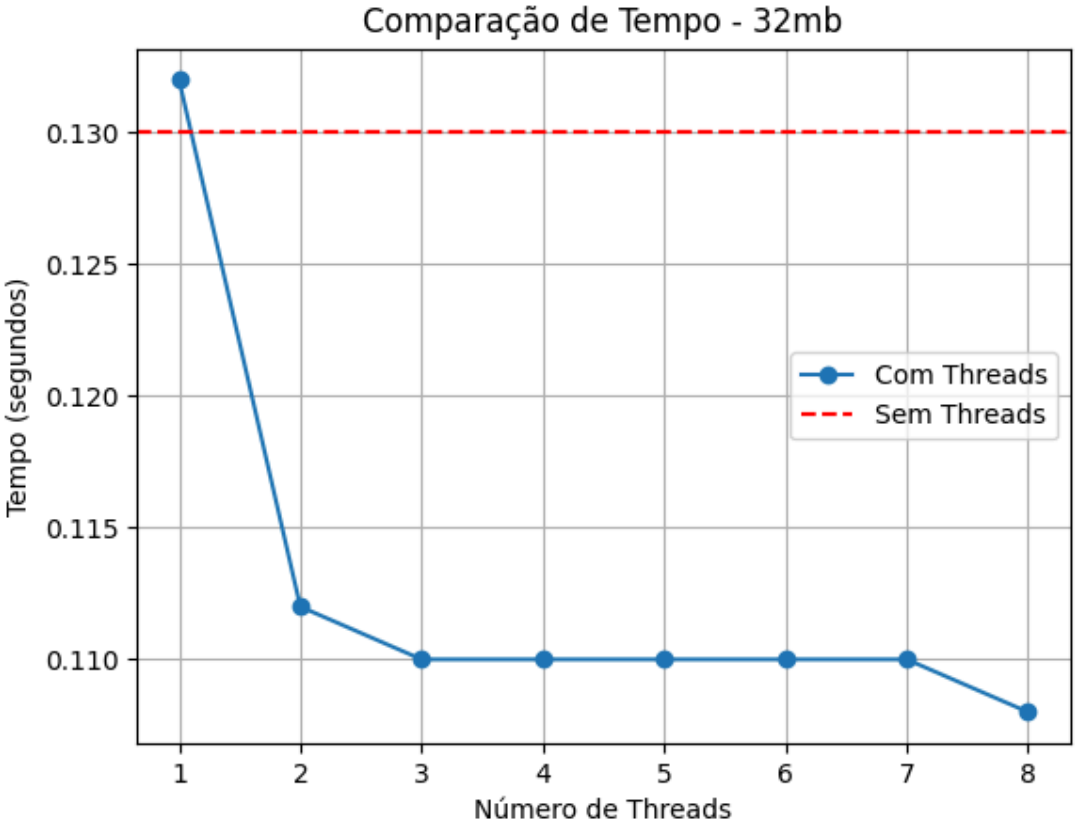
trabalho entre várias threads resulta em ganhos de desempenho mais notáveis.

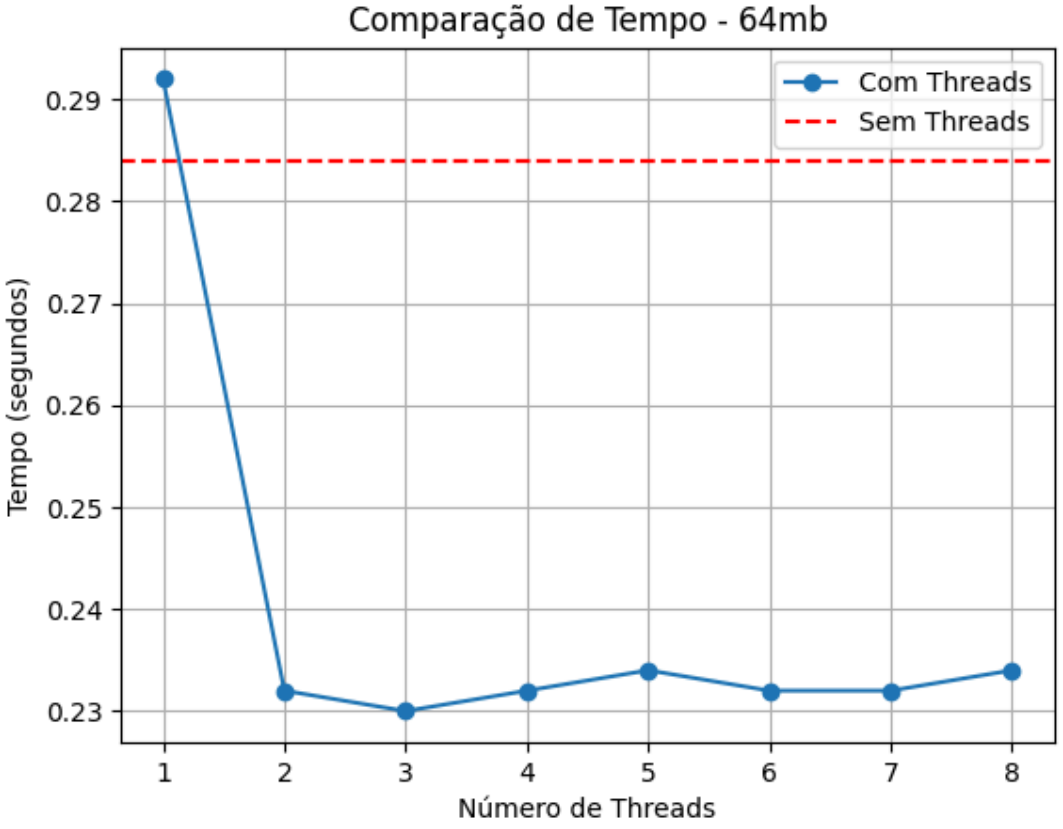
3. **Interpretação dos Gráficos**:
    - **Gráficos comparativos** foram gerados para mostrar a **diferença no tempo de execução** com e sem threads. Os gráficos revelam claramente o **ponto de saturação** onde a adição de mais threads não resulta mais em melhorias significativas no desempenho.
4. **Conclusões**:
    - A análise sugere que, para arquivos pequenos, o uso de múltiplas threads não oferece grandes vantagens, e o ganho de desempenho diminui rapidamente à medida que o número de threads aumenta.
    - Para arquivos grandes, o uso de múltiplas threads é vantajoso, mas existe um ponto de saturação onde mais threads não resultam em maiores ganhos de desempenho.
    - Basicamente a lei de Amdahl que diz que uma parte do programa não será paralelizável, ou seja, chega um momento que aumentar a quantidade de threads custa muito desempenho e não gera nada de benefício.
5. **Intervalo de Confiança e Significância Estatística**:
    - Para garantir que os resultados fossem confiáveis, cada configuração foi medida várias vezes (com 5 repetições) e a **média** e o **desvio padrão** dos tempos de execução foram calculados. Isso ajudou a garantir que os resultados não fossem apenas acidentais ou causados por variabilidades do sistema.

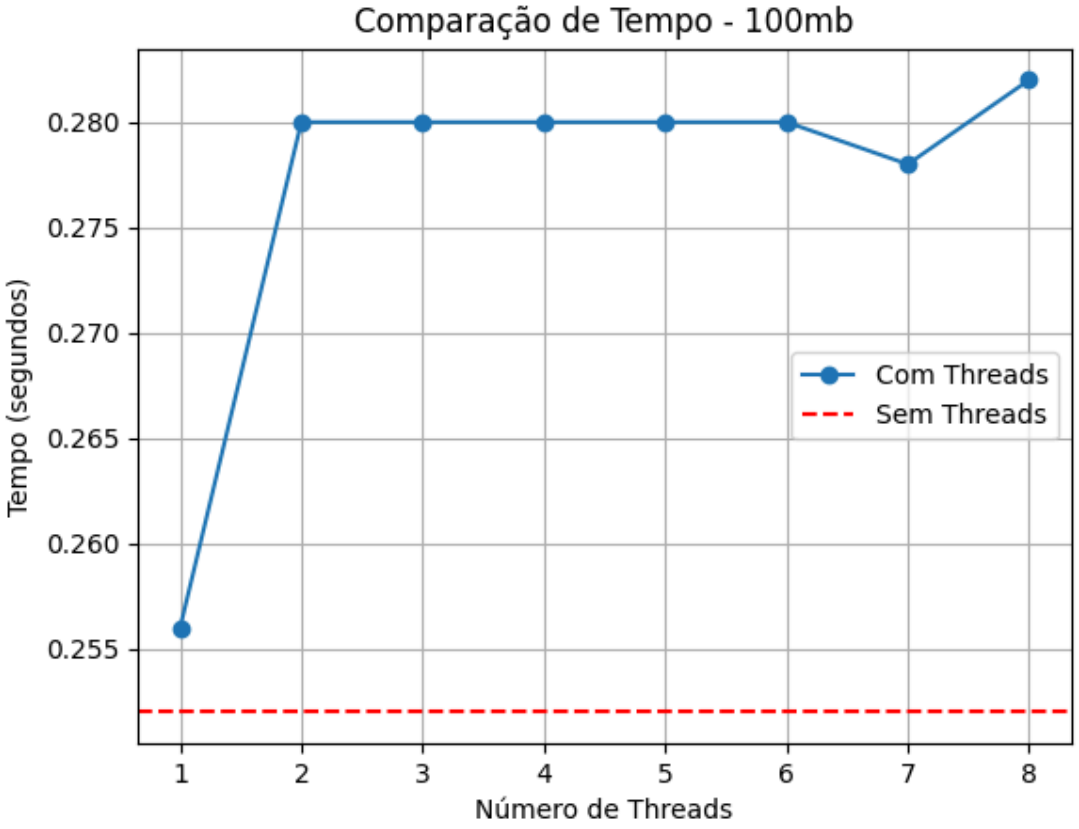
6. **Gráficos com Tempo de Execução (em segundos) para ordenar arquivos de diferentes tamanhos**






Comparação de Tempo - 32mb
Tempo (segundos)
0.130
0.125
0.120
0.115
0.110
1
2
3
4
5
6
7
8
Número de Threads
Com Threads
Sem Threads


Comparação de Tempo - 64mb
Tempo (segundos)
0.29
0.28
0.27
0.26
0.25
0.24
0.23
1
2
3
4
5
6
7
8
Número de Threads
Com Threads
Sem Threads


Comparação de Tempo - 100mb
Tempo (segundos)
0.280
0.275
0.270
0.265
0.260
0.255
1
2
3
4
5
6
7
8
Número de Threads
Com Threads
Sem Threads