Projeto AV2 - Análise de Desempenho de Algoritmos de Busca em Ambientes Concorrentes e Paralelos: Um Estudo Comparativo em Java

Autores

- **Victor Rios**
- **Nelson Mateus**

Resumo

Este projeto tem como objetivo realizar uma **análise comparativa de desempenho entre versões seriais e paralelas de algoritmos clássicos de ordenação**, desenvolvidos na linguagem **Java**. A busca por **eficiência computacional** é um dos pilares da ciência da computação moderna. Com o avanço dos processadores **multicore**, compreender como diferentes algoritmos se comportam em contextos **concorrentes e paralelos** torna-se essencial. Neste estudo, foram implementados e analisados **quatro algoritmos de ordenação** — *Quick Sort, Merge Sort, Counting Sort* e *Insertion Sort* — em suas versões **sequenciais e paralelizadas**, com **execuções controladas, medições de tempo, geração de arquivos CSV e visualização dos resultados**.

Introdução

A ordenação de dados é uma operação fundamental em ciência da computação, impactando diretamente o desempenho de sistemas em diversas aplicações. Neste trabalho, implementamos e comparamos os seguintes algoritmos: 1. **Quick Sort** 2. **Merge Sort** 3. **Counting Sort** 4. **Insertion Sort** Cada um deles foi implementado em **duas versões**:

- **Versão Sequencial:** execução linear, utilizando apenas uma thread.
- **Versão Paralela:** execução concorrente, utilizando **Java Threads** para dividir a carga de trabalho.

O repositório do projeto está disponível em: GitHub - Victorios20/algoritmos-paralelos-java

Metodologia

A metodologia adotada seguiu as etapas abaixo:

1. Implementação dos algoritmos

Cada algoritmo foi implementado em Java em duas versões (serial e paralela). A implementação paralela utilizou **multithreading**, com controle de número de threads, para permitir análises em diferentes níveis de paralelismo.

2. Estrutura de Testes

Foi desenvolvido um **framework de experimentação**, responsável por:

- Gerar **conjuntos de dados aleatórios** de diferentes tamanhos;
- Executar cada algoritmo diversas vezes (mínimo de **5 amostras**);
- Controlar o número de threads em execução;
- **Registrar tempos médios e desvios padrão**;
- **Exportar resultados em arquivos CSV**.

3. Coleta e Registro de Dados

Os resultados das execuções foram gravados em **arquivos `.csv`**, contendo:

- Nome do algoritmo;
- Tipo de execução (serial/paralela);
- Número de threads;
- Tamanho do conjunto de dados;
- Tempo médio de execução (ms).

Esses dados permitem a geração de **gráficos comparativos**, como mostrado abaixo:

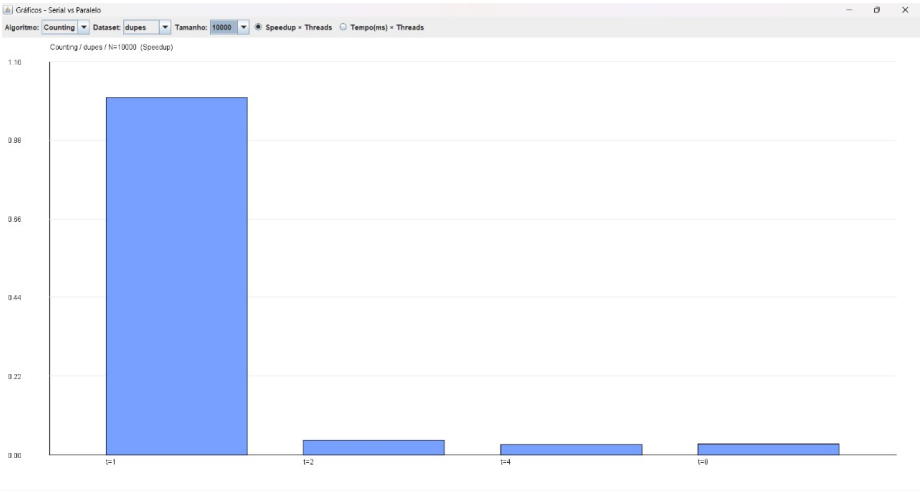

Figure 1: Gráfico Comparativo de Desempenho

(Figura 1 – Exemplo ilustrativo de comparação entre versões seriais e paralelas de algoritmos.)

Também é possível escolher algumas configurações como o algorítimo:

(Figura 2 – Exemplo ilustrativo selecionar o tipo de algorítimo.)

Como o tipo de dataset:

Figure 2: Menu 1

Figure 3: Menu 2

(Figura 3 – Exemplo ilustrativo selecionar o tipo dataset.)

Como o tamanho da entrada de dados:

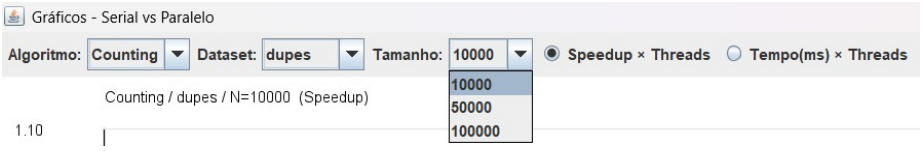

Figure 4: Menu 3

(Figura 4 – Exemplo ilustrativo selecionar o tamanho de entrada.)

Abaixo estão alguns exemplos de vizualização:

(Figura 5 – Exemplo do algoritimo insertion.)

(Figura 6 – Exemplo do algoritimo counting.)

(Figura 7 – Exemplo do algoritimo merge.)

(Figura 8 – Exemplo do algoritimo quick.)

4. Análise Estatística

A análise dos dados do csv considerou métricas de tempo médio e variação percentual de desempenho entre versões, observando o comportamento de escalabilidade conforme o aumento do número de threads.

O csv contém os dados: algoritmo, versao, dataset, tamanho, threads, amostra, tempo_ms, speedup, eficiencia.

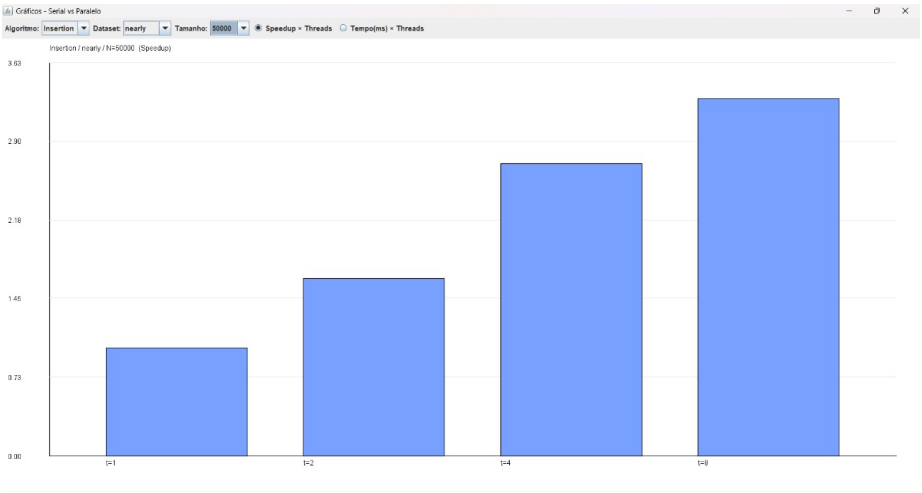

Figure 5: Insertion Sort

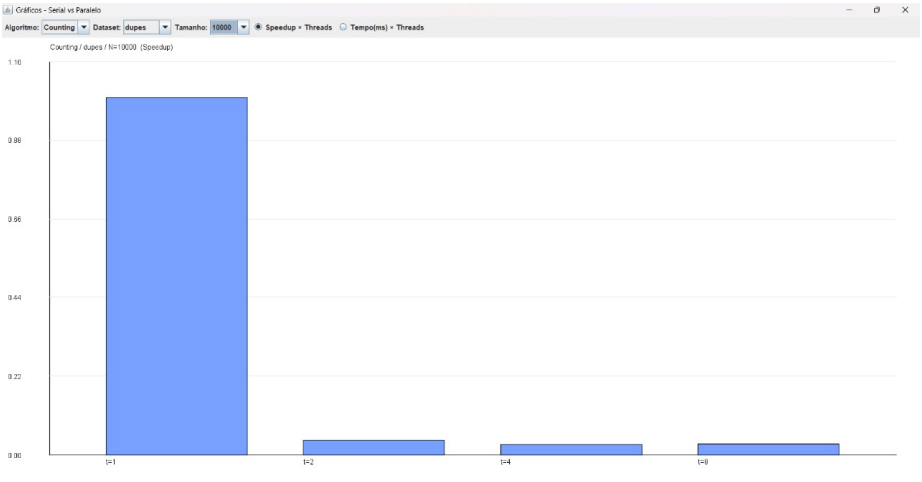

Figure 6: Counting Sort

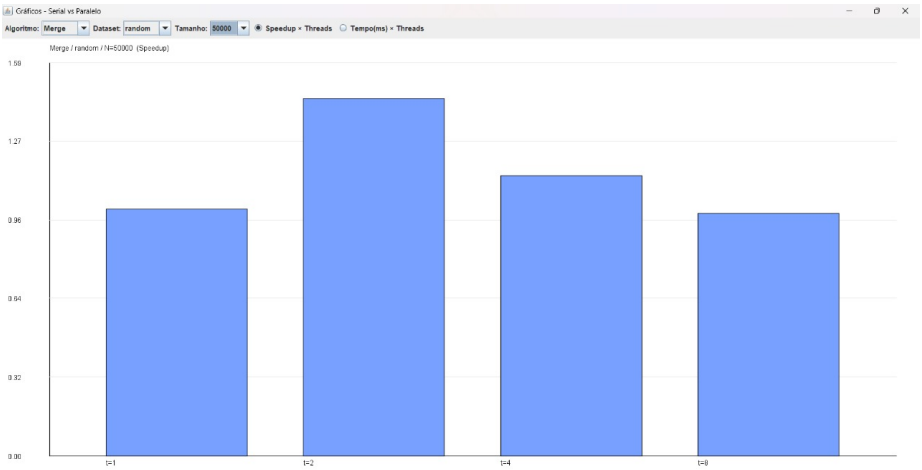

Figure 7: Merge Sort

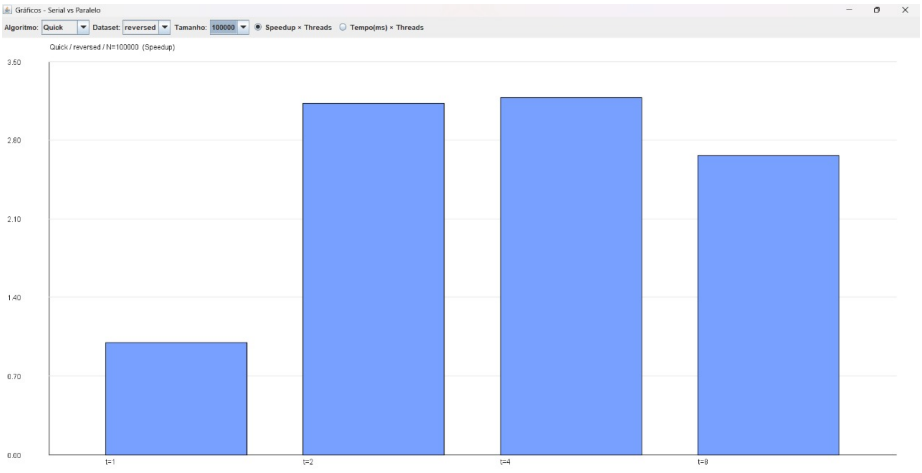

Figure 8: Quick Sort

Foi feita uma análise de dados com python dessas 1200 entradas do csv em que: - foram testados 4 algoritmos × 4 tamanhos de dataset × até 8 threads. - Cada configuração foi repetida 5 vezes para garantir média e confiabilidade.

Os resultados foram colocados em gráficos para comparação:

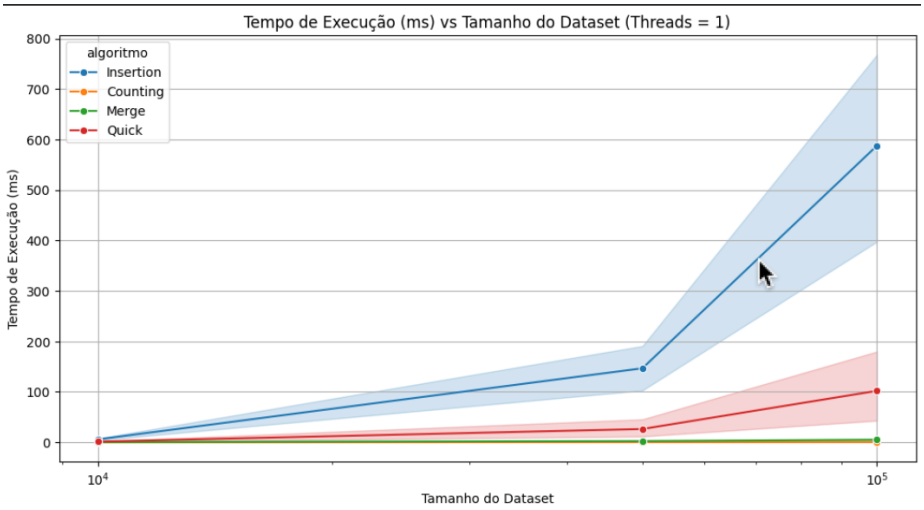

Figure 9: Tempo de execução vs tamanho do dataset

(Figura 9 – Tempo de execução vs tamanho do dataset para os 4 algoritimos - 1 thread.)

(Figura 10 – Eficiência Média vs Número de Threads para os 4 algoritimos.)

(Figura 11 – SpeedUp médio vs Número de Threads para os 4 algoritimos.)

Resultados e Discussão

Com base nessa análise estatística e nos gráficos os resultados demonstraram comportamentos distintos entre os algoritmos:

Resumo das Métricas de Desempenho

A análise das métricas de desempenho (tempo de execução, speedup e eficiência) para os algoritmos Insertion, Counting, Merge e Quick em diferentes tamanhos de dataset e número de threads revelou o seguinte: - **Tempo de Execução:** O tempo de execução geralmente aumenta com o tamanho do dataset. Algoritmos como o Quick Sort e Merge Sort tendem a ter tempos de execução menores para datasets maiores em comparação com o Insertion Sort. O Counting Sort apresenta tempos de execução relativamente baixos, especialmente para datasets com menor variação de valores. - **Speedup:** O speedup, que mede o ganho de desempenho ao usar múltiplos threads em comparação com um único

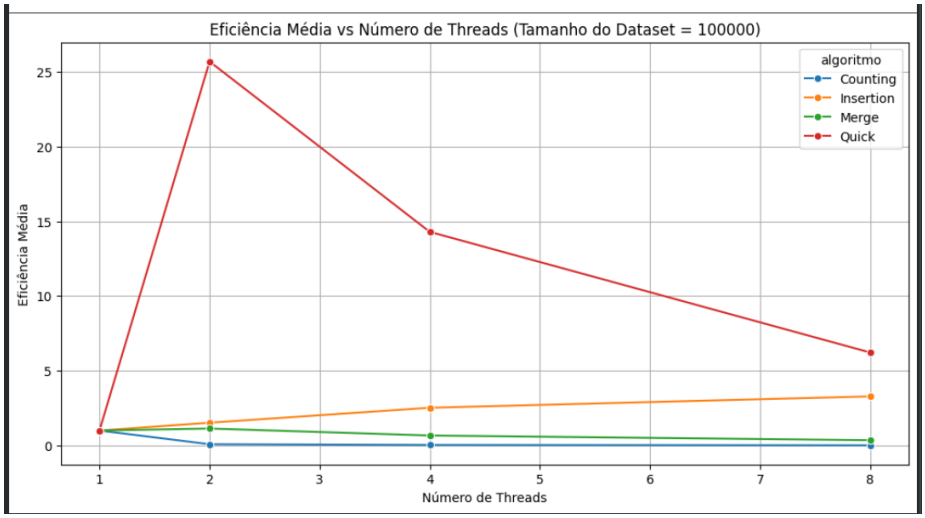

Figure 10: Eficiência Média vs Número de Threads

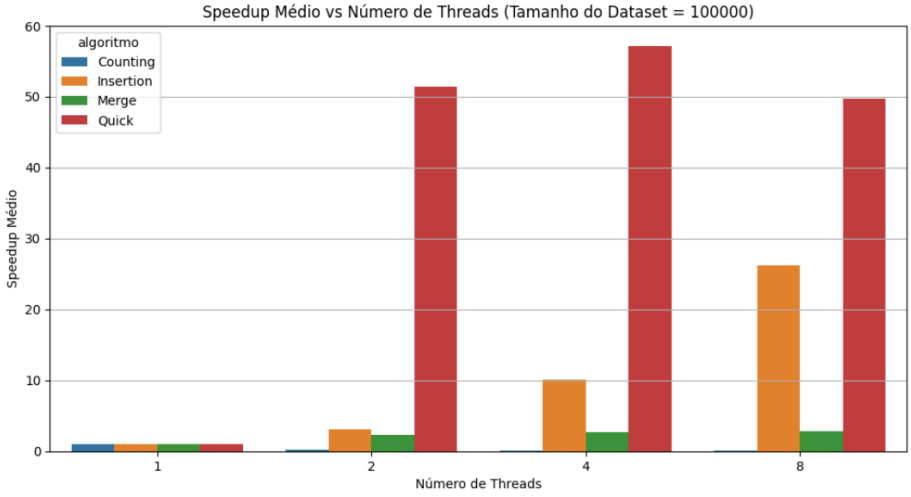

Figure 11: SpeedUp médio vs Número de Threads

thread, varia significativamente entre os algoritmos. Algoritmos paralelizáveis como Quick Sort e Merge Sort mostram speedup considerável com o aumento de threads, enquanto o Insertion Sort, sendo inerentemente serial, não apresenta speedup significativo. O Counting Sort também mostra speedup limitado devido à sua natureza. - **Eficiência:** A eficiência, calculada como speedup dividido pelo número de threads, indica quão bem os recursos de processamento adicionais estão sendo utilizados. Idealmente, a eficiência seria 1 (speedup igual ao número de threads). Observou-se que a eficiência tende a diminuir à medida que o número de threads aumenta para a maioria dos algoritmos, indicando que o overhead da paralelização pode superar os benefícios em certos pontos. O Quick Sort mostrou picos de eficiência para um número moderado de threads, mas a eficiência diminui com mais threads. O Insertion Sort e Counting Sort geralmente apresentam baixa eficiência em ambientes paralelos.

Análise das Visualizações

As visualizações geradas fornecem insights adicionais sobre o desempenho dos algoritmos: - **Tempo de Execução (ms) vs Tamanho do Dataset (Threads = 1):** Este gráfico de linha mostra claramente a diferença na escalabilidade dos algoritmos com o tamanho do dataset em execução serial. O Insertion Sort apresenta um aumento de tempo de execução muito mais acentuado do que os outros algoritmos, destacando sua ineficiência para grandes volumes de dados. Quick Sort e Merge Sort demonstram melhor escalabilidade, enquanto o Counting Sort mantém um tempo de execução relativamente baixo, confirmando sua adequação para certos tipos de dados. - **Speedup Médio vs Número de Threads (Tamanho do Dataset = 100000):** O gráfico de barras do speedup médio ilustra a eficácia da paralelização para cada algoritmo em um dataset grande. Quick Sort e Merge Sort exibem speedup significativo com o aumento de threads, embora a taxa de aumento possa diminuir. Insertion Sort e Counting Sort mostram speedup limitado ou insignificante, reforçando que não se beneficiam substancialmente da execução paralela. - **Eficiência Média vs Número de Threads (Tamanho do Dataset = 100000):** O gráfico de linha da eficiência média complementa a análise de speedup, mostrando a utilização dos recursos. A queda na eficiência para a maioria dos algoritmos com mais threads sugere custos de comunicação e sincronização entre threads. O Quick Sort demonstra uma eficiência mais alta para um número menor de threads antes de diminuir, indicando um ponto ideal para paralelização em termos de utilização de recursos.

Conclusão

Com base na análise, podemos concluir que: - Para datasets pequenos ou em execução serial, o desempenho dos algoritmos pode variar, mas para datasets grandes, algoritmos com melhor complexidade assintótica como Quick Sort e Merge Sort superam o Insertion Sort. - A paralelização é benéfica para algoritmos como Quick Sort e Merge Sort, resultando em speedup considerável,

mas a eficiência pode diminuir com um número excessivo de threads devido ao overhead. - O Counting Sort é eficiente para tipos específicos de dados, mas sua paralelização pode não ser tão eficaz quanto a de outros algoritmos para datasets gerais. - A escolha do algoritmo de ordenação mais adequado depende do tamanho do dataset, da necessidade de paralelização e das características específicas dos dados.

Referências

- Tanenbaum, A. S. *Sistemas Operacionais Modernos.* Pearson, 2016.
- Lewis, B., & Berg, D. J. *Multithreaded Programming with Java Technology.* Prentice Hall, 2000.
- Sedgewick, R., & Wayne, K. *Algorithms.* Addison-Wesley, 2011.
- Documentation Java Threads API – Oracle Docs.
- GeeksForGeeks – *Parallel Sorting Algorithms in Java.*

Execução do Projeto

Como compilar e rodar (Windows/PowerShell)

Remove-Item -Recurse -Force out -ErrorAction SilentlyContinue javac -d out (Get-ChildItem -Recurse -Filter *.java | ForEach-Object { $_.FullName }) java -cp out bench.Main

Saída

- results.csv: tempos por amostra e medianas por combinação.