Programacão Concorrente em Algoritmos de Ordenação

Carlos Eduardo da Silva Santos - 119065432

Julia Deroci Lopes - 117257871

Relatório Final

Programação Concorrente (ICP-361) — 2025/1

1. Descrição do Problema Geral

Este trabalho realiza uma comparação entre as versões sequencial e concorrente do algoritmo Merge Sort, aplicadas à ordenação de grandes volumes de dados simulados.

O cenário hipotético abordado envolve o monitoramento de dados demográficos de uma população, com registros compostos por idade, peso e altura. Foram testados três cenários: dados aleatórios, ordenados em ordem crescente e em ordem decrescente, com até 1 milhão de registros. O objetivo é avaliar o desempenho e os ganhos de eficiência da abordagem concorrente em relação à sequencial nos três casos propostos.

1.1. Arquivos de Entrada

Para isso, foi desenvolvido um gerador de dados que cria arquivos CSV com até 1 milhão de registros, contendo valores aleatórios para idade, peso e altura. A partir desse arquivo original (desordenado), foram produzidas duas novas versões: uma com os dados ordenados em ordem crescente e outra em ordem decrescente, segundo uma prioridade de ordenação: primeiro pela idade, depois pelo peso e, por fim, pela altura.

Dessa forma, teremos a seguinte entrada e saída esperada para cada um dos três casos:

- ***Entrada:*** *Um arquivo CSV contendo os dados simulados de forma desordenada, crescente ou decrescente. Exemplo:*

altura	peso	idade
1.7	101.7	65
1.47	78.8	32
1.46	69.3	74
1.7	106.1	60
1.81	114.6	53
1.62	87.3	52
1.73	46.9	84
1.64	115.8	41
1.48	102.6	39
1.62	69.1	15

Figure 1. Arquivo csv desordenado de entrada.

- ***Saída:*** *Um arquivo CSV ordenado de forma crescente, seguindo a prioridade de ordenação:* ***idade, peso e altura****. Exemplo:*

altura	peso	idade
1.62	69.1	15
1.47	78.8	32
1.48	102.6	39
1.64	115.8	41
1.62	87.3	52
1.81	114.6	53
1.7	106.1	60
1.7	101.7	65
1.46	69.3	74
1.73	46.9	84

Figure 2. Arquivo csv de saída.

2. Projeto da Solução Concorrente

A solução proposta explora o paralelismo natural do algoritmo Merge Sort, que divide recursivamente um vetor em subpartes, ordena essas partes separadamente e, por fim, as mescla. Essa divisão recursiva é especialmente adequada para aplicação de programação concorrente, pois permite que diferentes subpartes sejam processadas simultaneamente.

2.1. Divisão das Tarefas

A implementação utiliza múltiplas threads para ordenar diferentes segmentos do vetor de forma concorrente, aproveitando o fato de que o Merge Sort divide recursivamente o vetor ao meio até chegar em subpartes pequenas. A cada divisão, o programa tenta criar uma thread para ordenar cada metade de forma paralela, passando os índices de início e fim da partição que será tratada. Esse processo continua recursivamente, com cada thread podendo gerar novas divisões se ainda houver threads disponíveis, caso contrário, a execução recursiva continua de forma sequencial na própria thread atual.

Para evitar a criação excessiva de threads e o sobrecarregamento do sistema, o número total permitido é definido pelo usuário e controlado por uma variável global protegida por mutex. Após as ordenações paralelas, as partes resultantes são unidas por meio do algoritmo de merge, que compara elemento por elemento das duas metades já ordenadas e monta um segmento final também ordenado. Essa etapa de junção é sempre feita de forma sequencial, a fim de garantir que nenhuma região da memória seja acessada ou modificada simultaneamente por diferentes threads.

Podemos dizer que a execução de cada thread envolve:

- *Ordenar um segmento do vetor com base no algoritmo Merge Sort.*
- *Retornar à thread principal após a conclusão.*
- *Liberar sua vaga para permitir a criação de novas threads nas próximas divisões.*

3. Testes de Corretude

A verificação da corretude do código concorrente foi realizada por meio da análise dos arquivos CSV contendo 100 registros. Para isso, tanto a versão sequencial quanto a versão concorrente do algoritmo foram executadas utilizando os mesmos arquivos de entrada. A expectativa é que ambas produzam resultados idênticos, o que confirma a preservação da lógica do algoritmo.

No contexto do Merge Sort concorrente, a verificação da corretude concentrou-se em dois critérios principais:

- ***Ordenação correta:*** *O arquivo CSV gerado deve estar corretamente ordenado de forma crescente.*
- ***Preservação dos elementos:*** *: Todos os elementos presentes no CSV de entrada devem estar no CSV de saÍda, sem perdas ou duplicatas indevidas. Isso garante que a concorrencia não causou sobrescrita incorreta ou perda de dados.*

4. Avaliação de Desempenho

Para começarmos a avaliar o desempenho, nós utilizamos o método de cálculo de aceleração, e eficiência que foi aprendida durante o período, para fazermos a comparação entre o Merge Sort sequencial e o concorrente. O código para cálculo de tempo, foi feito utilizando a biblioteca "timer.h" também utilizada durante o período em um dos laboratórios.

Com os resultados obtidos, foram gerados gráficos que evidenciam os ganhos de desempenho. Como mencionado anteriormente, os testes foram realizados com arquivos CSV contendo 100, 40 mil, 400 mil e 1 milhão de registros. Cada um desses arquivos foi preparado em três formas distintas de ordenação: crescente, decrescente e desordenada.

Para a ordenação dos dados, utilizamos o algoritmo Merge Sort em duas versões: uma sequencial e outra concorrente, com execução paralela utilizando 2, 4 e 6 threads. Cada configuração foi testada cinco vezes para garantir consistência nos resultados.

Abaixo as tabelas com a média dos tempos de processamento obtidos nos testes:

Tempo Médio de Execução - Concorrente					
Crescente	Nº	100	40.000	400.000	1.000.000
	2 Threads	0,000138	0,01329040	0,062816	0,156571
	4 Threads	0,000225	0,01452980	0,054834	0,154512
	6 Threads	0,001096	0,01338140	0,063773	0,164748
Decrescente	Nº	100	40.000	400.000	1.000.000
	2 Threads	0,000144	0,01023000	0,061316	0,165587
	4 Threads	0,000292	0,01111860	0,054484	0,175464
	6 Threads	0,001499	0,01447560	0,058750	0,188416
Desordenado	Nº	100	40.000	400.000	1.000.000
	2 Threads	0,000270	0,02107200	0,086764	0,244492
	4 Threads	0,000338	0,01693800	0,074499	0,219721
	6 Threads	0,001815	0,02151180	0,082644	0,231349

Figure 3. Tabela Tempo Médio de Execução Concorrente.

Tempo Médio de Execução - Sequencial				
Tipo	**100**	**40.000**	**400.000**	**1.000.000**
Crescente	0,000088	0,006474	0,070773	0,166860
Decrescente	0,000100	0,007349	0,064641	0,210829
Desordenado	0,000128	0,010318	0,101096	0,262499

Figure 4. Tabela Tempo Médio de Execução Sequencial.

4.1. Aceleração/Eficiência

$$A(n,t) = \frac{T_{sequencial}(n,t)}{T_{concorrente}(n,t)}$$

$$E(n,t) = \frac{A(n,t)}{t}$$

4.2. Especificações da Máquina

A máquina utilizada na realização dos testes possui as seguintes configurações:

- *Sistema Operacional: Linux Mint 21 Cinnamon*
- *Versão do Cinnamon: 5.4.12*
- *Kernel do Linux: 5.15.0-101-generic*
- *Processador: 12ª geração Intel® Core™ i5-12400, com 6 núcleos.*

5. Discussão dos Resultados

Nesta seção, discutem-se os resultados obtidos a partir das análises realizadas ao longo deste trabalho.

5.1. Tempo Médio de Execução

No gráfico 5, observamos o comportamento do tempo de processamento quando os dados do CSV estão sem qualquer ordenação. De modo geral, verifica-se que o tempo de execução cresce proporcionalmente com o aumento no tamanho do arquivo, como esperado. Por exemplo, os tempos para arquivos com 100 registros são praticamente insignificantes, enquanto os arquivos com 1.000.000 de registro alcançam tempos superiores a 0,2 segundos.

Com relação à paralelização, nota-se um ganho significativo de desempenho ao se passar da versão sequencial para versão concorrente, especialmente para os arquivos maiores. O tempo para processar 1.000.000 registros cai significamente. Esse ganho continua razoável até 4 threads, porém, ao utilizar 6 threads, o tempo volta a subir ligeiramente, sugerindo que há um ponto ótimo em que o paralelismo contribui, mas que, a partir de certo limite, o overhead gerado pela concorrência supera os ganhos.

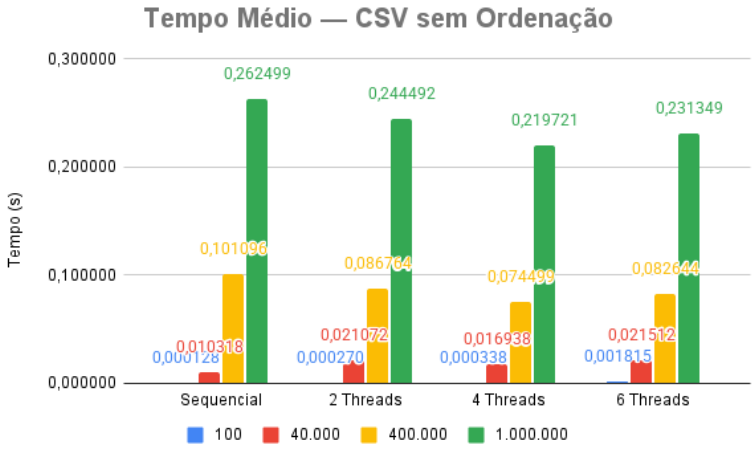

Figure 5. Gráfico do Tempo Médio de Execução — CSV sem Ordenação.

O gráfico 6 mostra que a concorrência traz ganhos de desempenho apenas com grandes volumes de dados. Para entradas pequenas, como 100 e 40.000 registros, a versão sequencial é tão eficiente quanto ou até melhor, devido ao baixo custo e ausência de overhead. Já com 400.000 e 1.000.000 de elementos, o uso de 4 threads se destaca como o mais eficiente, superando tanto a versão sequencial quanto outras configurações concorrentes. O uso de 6 threads não melhora o desempenho e pode até piorar, indicando que o excesso de threads gera mais custo de sincronização do que benefício.

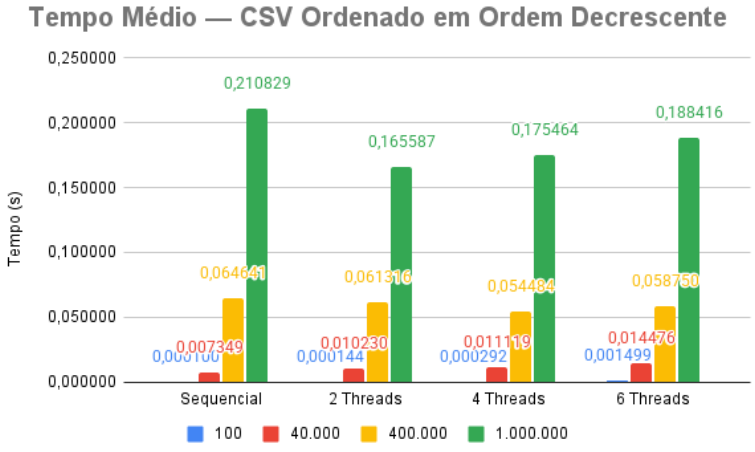

Figure 6. Gráfico do Tempo Médio de Execução — CSV Ordenado em Ordem Decrescente.

Já no gráfico 7, com os dados do CSV organizados em ordem crescente, é onde se nota o melhor desempenho entre todos os cenários. A ordenação crescente beneficia significativamente o tempo de processamento, especialmente para arquivos maiores.

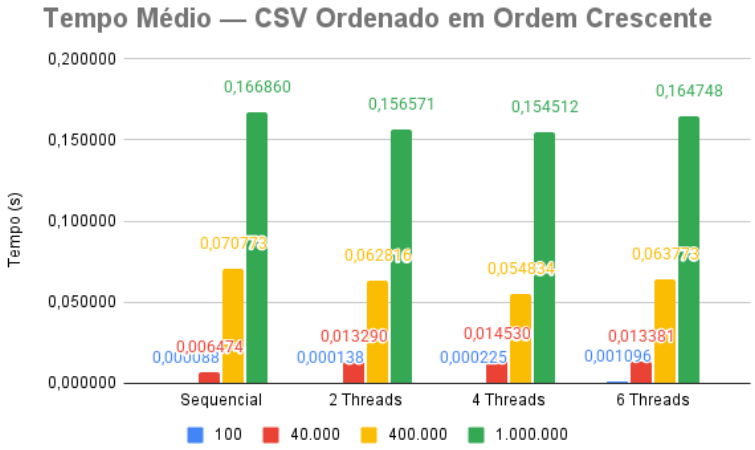

Figure 7. Gráfico do Tempo Médio de Execução — CSV Ordenado em Ordem Crescente.

5.2. Análise de Aceleração

O gráfico 8 apresenta a aceleração do algoritmo concorrente com dados em ordem aleatória, ou seja, sem qualquer pré-ordenamento. Observa-se que, para o menor conjunto de dados, o uso de múltiplas threads resulta em uma desaceleração. Isso indica que, para entradas muito pequenas, o custo associado à criação e sincronização das threads supera os benefícios da divisão de trabalho, tornando o desempenho inferior ao da versão sequencial.

À medida que o volume de dados aumenta para 40.000, 400.000 e 1.000.000 registros, os ganhos de desempenho com o uso de múltiplas threads tornam-se mais evidentes. No caso de 400.000 elementos, por exemplo, a aceleração obtida com 2 threads é de 17% ; com 4 threads, sobe para 35%; e com 6 threads, atinge 22%. No entanto, esses ganhos não crescem proporcionalmente ao número de threads.

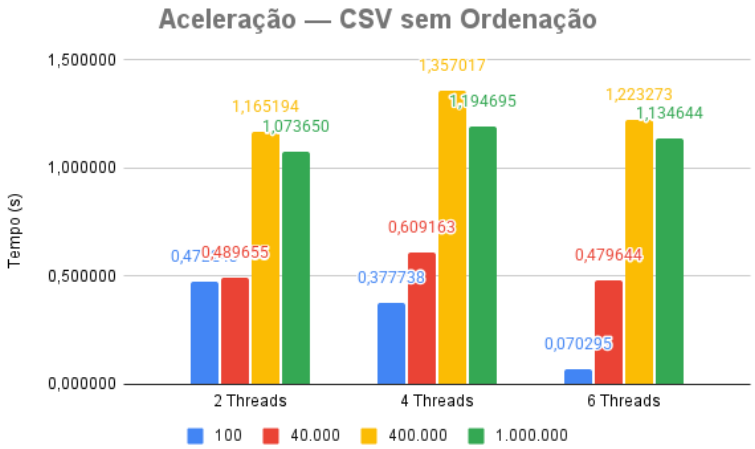

Figure 8. Gráfico de Aceleração CSV sem Ordenação.

Nos cenários com dados ordenados, tanto em ordem crescente quanto decrescente, apresentados nos gráficos 9 e 10, observa-se um padrão semelhante de aceleração.

Destacam-se, por exemplo, ganhos de até 29% com 400.000 registros em ordem crescente utilizando 4 threads, e de até 27% com 1.000.000 de registros em ordem decrescente utilizando 2 threads. Em ambos os casos, os ganhos com a execução concorrente tornam-se mais expressivos conforme o volume de dados aumenta.

Entretanto, observa-se também uma tendência de saturação do paralelismo entre 4 e 6 threads, com ganhos marginais ou até mesmo queda no desempenho a partir desse ponto. Isso indica que o aumento no número de threads não se traduz, necessariamente, em ganhos proporcionais de desempenho.

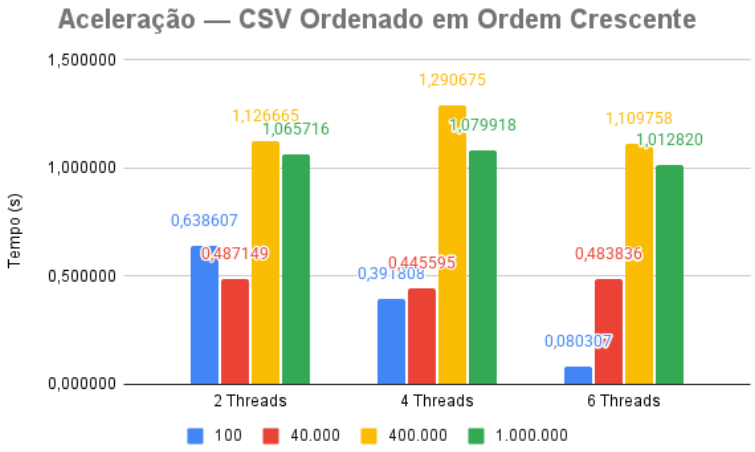

Figure 9. Gráfico Aceleração — CSV Ordenado em Ordem Crescente.

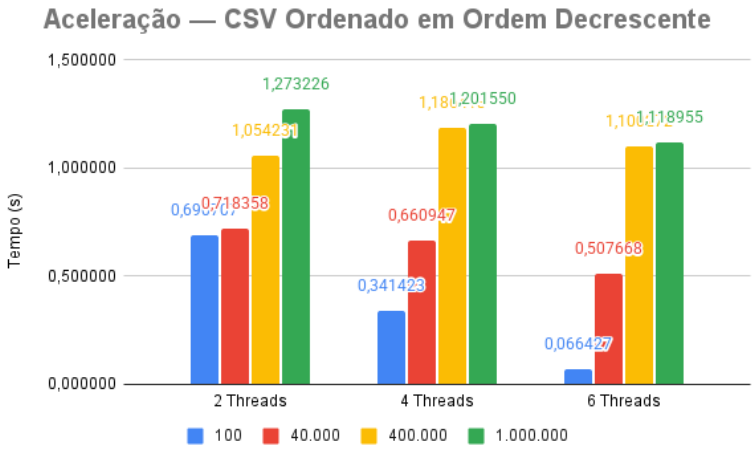

Figure 10. Gráfico de Aceleração — CSV Ordenado em Ordem Decrescente.

5.3. Análise de Eficiência

De modo geral, a eficiência aumenta em todos os cenários apresentados nos gráficos 11, 13 e 12 com o uso de até 4 threads. No entanto, em alguns casos, observa-se uma queda na eficiência ao se utilizar 6 threads. Esse comportamento sugere que, embora a paralelização traga benefícios iniciais, há um ponto a partir do qual o aumento no

número de threads deixa de compensar, devido ao acréscimo no custo de sincronização, comunicação entre threads e possível contenção de recursos.

Essa limitação é particularmente evidente em conjuntos de dados menores, nos quais o overhead da concorrência supera os ganhos obtidos com a divisão de tarefas. Em conjuntos maiores, embora os ganhos sejam mais consistentes, a escalabilidade também encontra um limite, indicando que o algoritmo possui um grau de paralelismo finito além do qual os benefícios se estabilizam ou mesmo se reduzem.

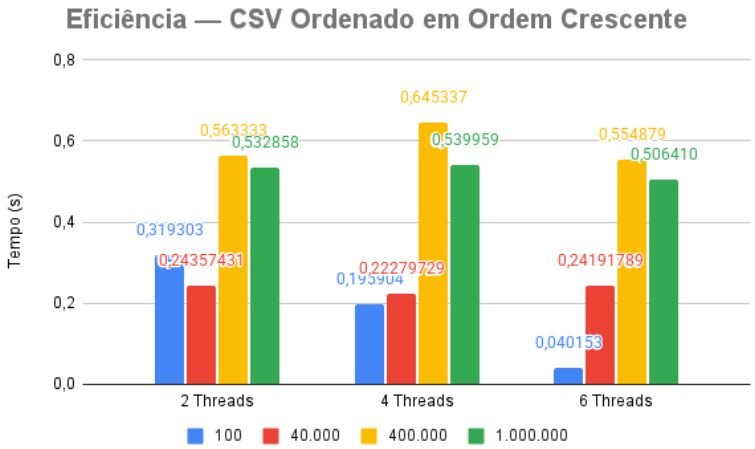

Figure 11. Gráfico de Eficiência — CSV Ordenado em Ordem Crescente.

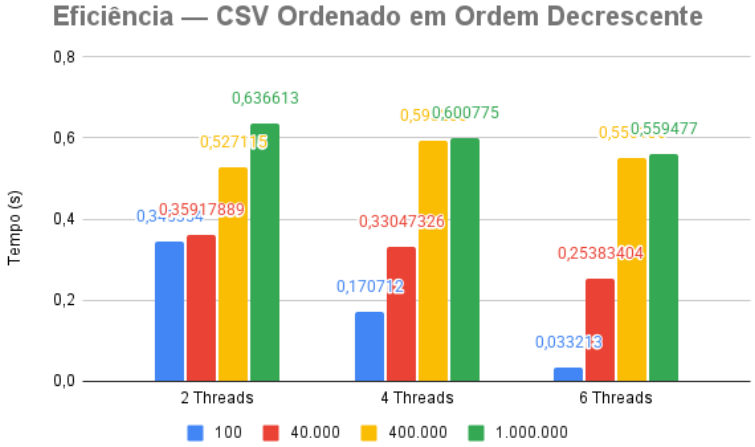

Figure 12. Gráfico de Eficiência — CSV Ordenado em Ordem Decrescente.

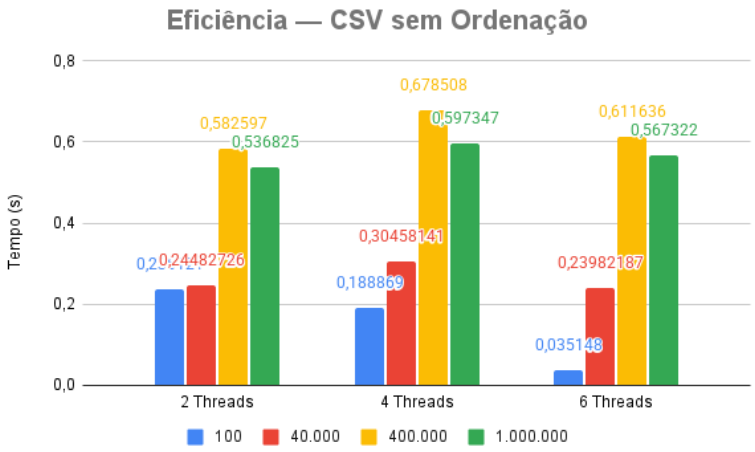

Figure 13. Gráfico de Eficiência — CSV sem Ordenação.

6. Referências

Nosso trabalho toma como referência os livros [1], [2]. Além do código desenvolvido e disponibilizado por nós no repositório [3].

[1] *Peter Pacheco.* An Introduction to Parallel Programming. *Elsevier, 2011. s.l.*

[2] *Randal E. Bryant and David R. O'Hallaron.* Computer systems: a programmer's perspective. *Pearson, Boston, 3rd edition, 2016.*

[3] *Carlos Eduardo and Julia Deroci. Merge sort concorrente. Disponível em:* `https://github.com/Dudu300599/MergeSort_Concorrente`*, 2025. Acessado em: 4 jun. 2025.*