UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
IDCA3703 - PROGRAMAÇÃO PARALELA

TAREFA 5 - COMPARAÇÃO ENTRE PROGRAMAÇÃO SEQUENCIAL E PARALELA
RELATÓRIO DE EXECUÇÃO

ERNANE FERREIRA ROCHA JUNIOR

NATAL/RN, 13 DE ABRIL DE 2025

SUMÁRIO

1. INTRODUÇÃO

O avanço das tecnologias de processamento tem possibilitado a execução de tarefas de forma mais eficiente e otimizada, principalmente no que diz respeito à programação paralela. O objetivo deste trabalho é comparar a execução de um algoritmo nas versões sequencial e paralela, analisando o impacto da utilização de múltiplos núcleos de processamento. Para isso, escolhemos uma aplicação que realiza o cálculo de uma operação matemática simples, mas com alto custo computacional quando executada em grande escala.

A versão sequencial do programa é executada em um único núcleo, enquanto a versão paralela aproveita múltiplos núcleos, com a implementação de uma técnica de redução para otimizar a execução. A análise do desempenho entre as duas versões será feita com base em métricas como o tempo de execução em função do tamanho de entrada e o número de *threads* utilizadas na versão paralela. Este estudo visa não apenas verificar as vantagens de se usar a programação paralela, mas também entender até que ponto a utilização de múltiplos núcleos pode realmente melhorar o desempenho, especialmente em relação ao tempo de execução, que é uma das métricas mais relevantes para qualquer tipo de otimização computacional.

2. METODOLOGIA

Para a realização deste trabalho, foram implementadas duas versões do algoritmo: uma sequencial e outra paralela. Ambas as versões possuem o mesmo comportamento lógico, mas com diferenças fundamentais na forma como as tarefas são distribuídas e processadas. A versão sequencial executa o cálculo em um único núcleo, realizando todas as operações em sequência, sem qualquer tipo de paralelismo. Já a versão paralela, implementada com a técnica de reduction, divide o trabalho entre múltiplos núcleos de processamento, utilizando o modelo de execução em paralelo com 8 *threads*, aproveitando a capacidade de múltiplos processadores para acelerar a execução.

O desempenho de cada versão foi medido em termos de tempo de execução, variando o tamanho da entrada do problema (denotado por n). Para a versão sequencial, o tempo de execução foi registrado à medida que o número de elementos aumentava, o que nos permitiu observar como o desempenho escala com o tamanho da entrada. Para a versão paralela, a mesma análise foi realizada, mas com múltiplos núcleos sendo utilizados, sendo observada a melhoria do tempo de execução conforme o número de *threads* aumentava.

As medições foram realizadas em um ambiente controlado, utilizando uma máquina com 8 núcleos de processamento. As versões do código foram executadas repetidamente para garantir a consistência dos resultados e reduzir possíveis variações decorrentes de flutuações no sistema. Após a coleta dos tempos de execução, gráficos foram gerados para comparar as duas versões, tanto para a análise individual de tempo versus tamanho da entrada, quanto para a comparação direta entre os tempos de execução sequenciais e paralelos.

Além disso, foi analisado o impacto da execução com 8 *threads* na versão paralela, comparando o tempo de execução de ambas as abordagens (sequencial e paralela) sob as mesmas condições de entrada.

3. RESULTADOS

Os testes realizados permitiram observar com clareza o comportamento das versões sequencial e paralela do algoritmo à medida que o valor de entrada, representado por *n*, aumentava. Inicialmente, ao analisar o desempenho da versão sequencial, constatamos que o tempo de execução cresce de forma linear com o aumento de *n*. Isso era esperado, dado que o algoritmo percorre todos os números até *n* para identificar quais são primos. Para valores pequenos de *n*, como 10^3 ou 10^4, o tempo de execução foi praticamente irrisório, muitas vezes inferior a milissegundos, o que torna essa versão bastante eficiente em casos simples.

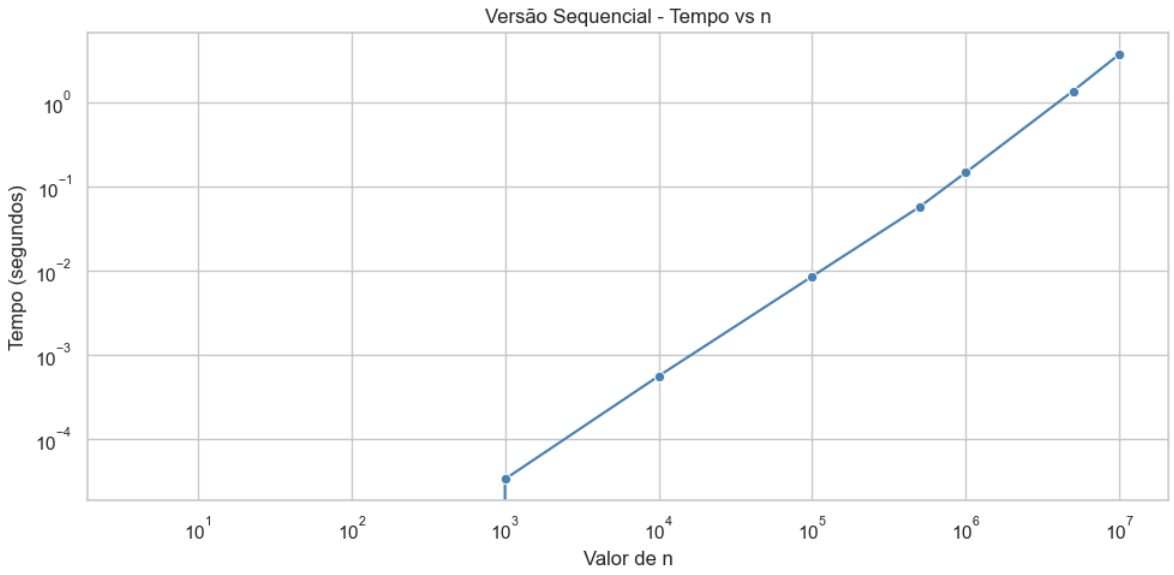

Figure 1: Versão Sequencial - Tempo vs n

Em seguida, foi observada a execução da versão paralela, inicialmente sem nenhum tipo de sincronização nas regiões críticas. Nessa abordagem, surgiram erros relacionados a condições de corrida, pois múltiplas *threads* tentavam modificar simultaneamente a mesma variável responsável por contar a quantidade de primos identificados. Como consequência, os resultados obtidos eram inconsistentes, variando a cada execução, e geralmente apresentando contagens incorretas.

Para evidenciar esse problema, a tabela a seguir mostra alguns valores de *n* e a contagem de primos retornada por ambas as versões. Note que o erro aumenta de acordo o número de *threads* aumenta para um mesmo fator *n*.

N	Primos - Sequencial	Primos - Paralelo [w/2th] (Sem proteção)	Primos - Paralelo [w/4th] (Sem proteção)	Primos - Paralelo [w/8th] (Sem proteção)
4	2	2	2	2
1000	168	168	152	150
10000	1229	1186	1201	1021
100000	9592	9473	9001	7461
500000	41538	41258	40245	35826
1000000	78498	78122	76741	70605
...	...	...	...	...

Para solucionar esse problema, foi aplicada a cláusula reduction, que garante a proteção da variável de contagem nas regiões críticas. Essa estratégia assegura que cada *thread* mantenha sua contagem parcial de primos, que ao final são combinadas de forma segura. Com isso, os resultados da versão paralela passaram a coincidir perfeitamente com os da versão sequencial, permitindo uma comparação justa entre desempenho e eficiência.

A partir desse ponto, foi possível realizar testes comparativos precisos. Notou-se que, para valores pequenos de *n*, a versão sequencial ainda apresentava melhor desempenho. Isso ocorre porque o *overhead* associado à criação e gerenciamento das múltiplas *threads* não é compensado quando o volume de dados é pequeno. No entanto, à medida que o valor de *n* cresce significativamente, esse custo inicial passa a ser diluído, e a vantagem do paralelismo começa a se manifestar.

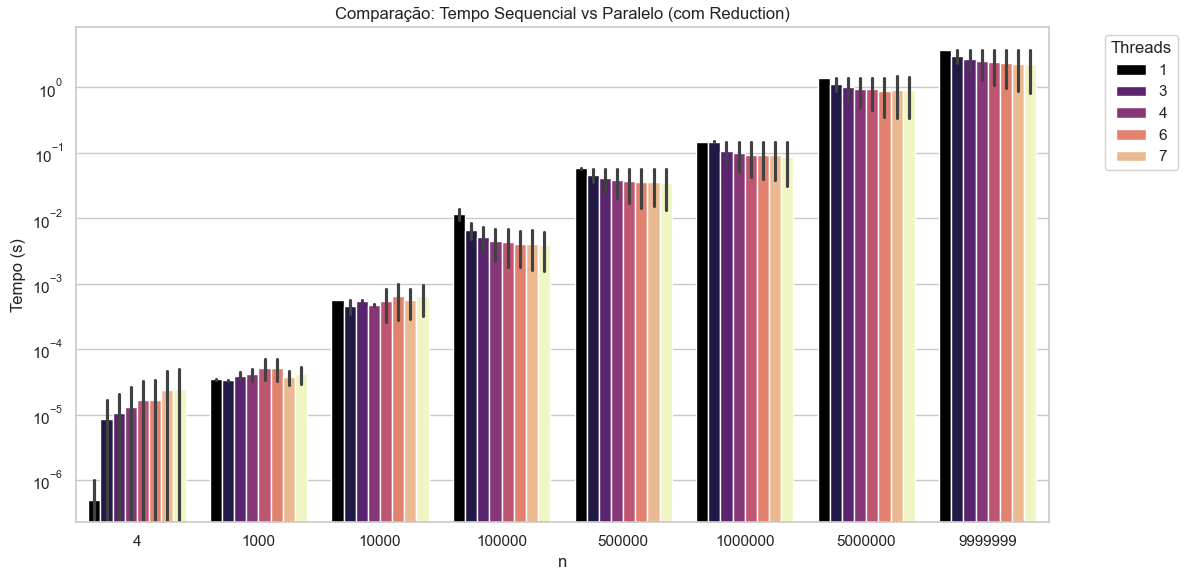

Figure 2: Comparação: Tempo Sequencial vs Paralelo (com Reduction)

Esse gráfico mostra claramente o ponto em que o desempenho da versão paralela supera o da sequencial, evidenciando o ganho obtido com o uso de múltiplos núcleos. Quando chegamos a valores

de n na ordem de 10^4 ou superiores, a execução paralela com o número máximo de *threads* disponíveis passa a ser visivelmente mais eficiente.

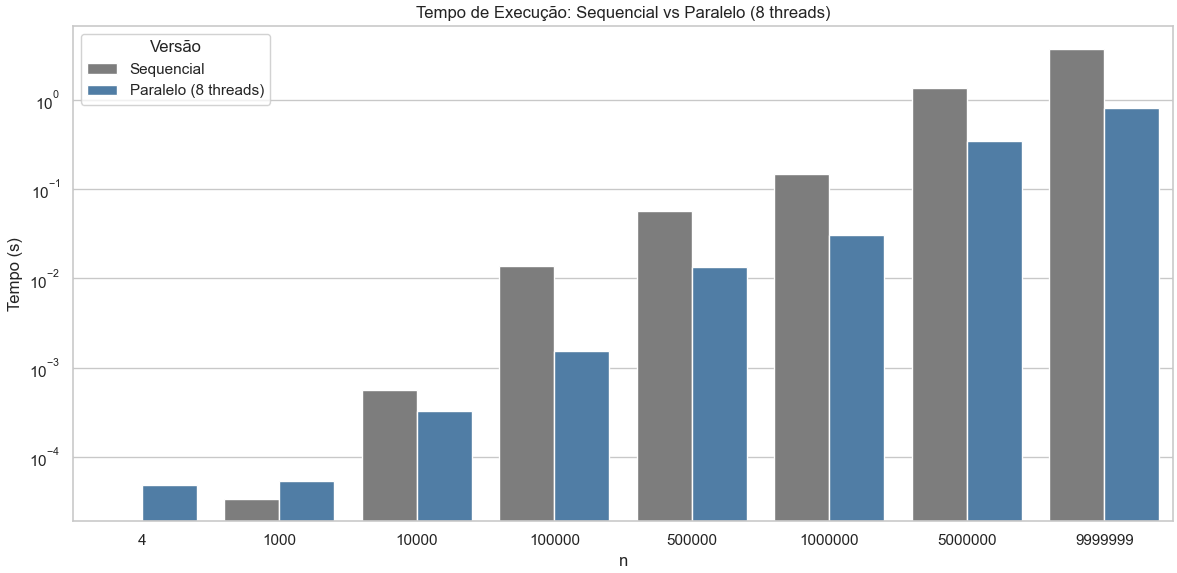

Figure 3: Tempo de Execução: Sequencial vs Paralelo (8 *threads*)

Além da comparação direta entre sequencial e paralelo com o número máximo de *threads*, também foi feita uma análise mais detalhada para entender qual seria a configuração ideal de *threads*. Para isso, o tempo de execução foi avaliado variando o número de *threads* entre 1 e 8. Foi possível observar que todas as configurações paralelas superam a versão sequencial para valores maiores de n, mas o comportamento entre elas não é estritamente linear.

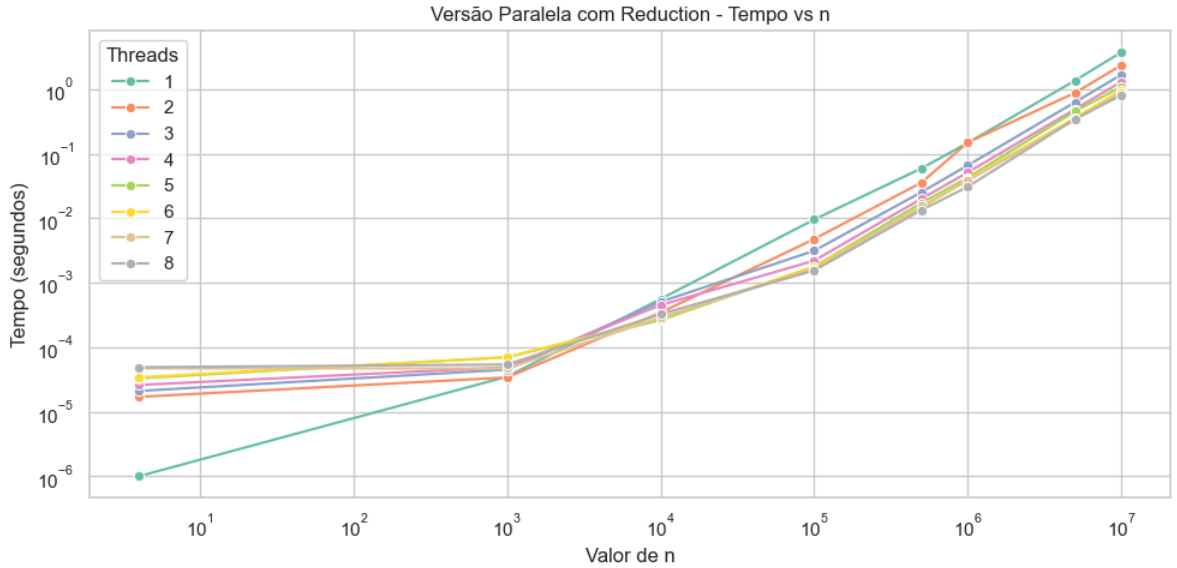

Figure 4: Versão Paralela com Reduction - Tempo vs n (1 à 8 *threads*)

Esse gráfico revela variações curiosas ao longo do tempo de execução. Por exemplo, na primeira parte da curva, como esperado, quanto mais *threads* são utilizadas, maior é o tempo de execução devido ao *overhead*. Entretanto, à medida que *n* aumenta, esse comportamento se inverte de forma abrupta. Observamos uma transição em que 6 *threads* tornam-se mais rápidas que 8, e 2 *threads* passam a superar 1.

No meio da curva, há outras transições interessantes: as posições relativas entre 3, 4, 5 e 7 *threads* mudam, com variações nos tempos que geram novas "camadas" de desempenho. Mais adiante, por volta de *n* na casa dos 10^5, ocorre uma nova virada: pela primeira vez, 8 *threads* tornam-se a configuração mais rápida, seguidas por 7, 6, 5, e assim sucessivamente até a mais lenta, com apenas 1 thread. A partir desse ponto, embora ainda ocorram pequenas oscilações, a estrutura geral se estabiliza, e fica evidente que, quanto maior o valor de *n*, menor será o tempo de execução usando o máximo de *threads* disponíveis para esse caso de experimentção.

4. CONCLUSÃO

A análise comparativa entre as versões sequencial e paralela do algoritmo de contagem de números primos permitiu observar de forma clara os impactos do paralelismo no desempenho computacional. A versão sequencial, apesar de simples e eficiente para valores baixos de entrada, torna-se cada vez menos competitiva à medida que o volume de dados cresce, justamente por não se beneficiar do uso de múltiplos núcleos de processamento. Já a versão paralela, após a devida correção das condições de corrida com o uso da cláusula reduction, demonstrou grande potencial de ganho de desempenho, principalmente para valores maiores de *n*.

Foi possível constatar que o paralelismo introduz um custo inicial significativo, associado à criação e coordenação das *threads*. Esse *overhead* torna a versão paralela inicialmente mais lenta que a sequencial, mas essa desvantagem se dilui com o aumento do volume de trabalho. A partir de determinado ponto — claramente visível nos gráficos apresentados — a vantagem do paralelismo se consolida, tornando essa versão muito mais rápida e escalável para grandes volumes de dados.

Além disso, a investigação sobre o número ideal de *threads* mostrou que, embora o uso do máximo de *threads* disponíveis tenda a produzir os melhores resultados para grandes valores de *n*, há um comportamento irregular e por vezes contraintuitivo em diferentes faixas de entrada. Essas flutuações, observadas nos gráficos, indicam que o desempenho não cresce de forma perfeitamente proporcional ao número de *threads*, sendo influenciado por diversos fatores como agendamento do sistema operacional, gerenciamento de cache e competição por recursos de hardware.

Em resumo, o paralelismo se mostrou uma estratégia altamente vantajosa para otimização de desempenho, desde que corretamente implementado e aplicado a problemas de escala suficiente. A escolha entre uma abordagem sequencial ou paralela depende, portanto, do equilíbrio entre o volume de trabalho e o custo do paralelismo — um fator crucial na tomada de decisão para aplicações de computação de alto desempenho.

5. ANEXOS

- Repositório no Github com o programa desenvolvido: https://github.com/ErnaneJ/parallel-programming-dca3703