Instituto Federal do Espírito Santo

Gustavo Saraiva Mariano
Pedro Henrique Albani Nunes
Tertuliano dos Santos Junior

Introdução à Programação Multithread com PThreads

Sistemas Operacionais
Professor Flávio Giraldeli Bianca

SERRA/ES
2025

Instituto Federal do Espírito Santo

Gustavo Saraiva Mariano
Pedro Henrique Albani Nunes
Tertuliano dos Santos Junior

Introdução à Programação Multithread com PThreads

Sistemas Operacionais
Professor Flávio Giraldeli Bianca

Trabalho apresentado ao Instituto Federal,
com o objetivo de introduzir os aprendizados
de multithread ao contexto de multiprogramação,
testando o mesmo exemplo de programa,
primeiramente em modo serial e após em modo paralelo.

SERRA/ES
2025

INTRODUÇÃO

A thread constitui a unidade básica de alocação e utilização da CPU, podendo ser entendida como um “miniprocesso” capaz de executar fluxos de instruções de forma concorrente dentro de um processo. As threads podem ser categorizadas em dois níveis principais: threads de usuário, cujo gerenciamento é realizado por bibliotecas de software específicas no espaço do usuário, e threads de kernel , cuja administração ocorre diretamente pelo núcleo do sistema operacional. Nesse contexto, para que uma thread de usuário realize uma chamada ao sistema (system call), é necessário estabelecer um vínculo com uma thread de kernel associada. Diante da crescente demanda por maior eficiência computacional e com os avanços nas arquiteturas de multiprocessamento, tecnologias baseadas nos conceitos de multithreading e paralelismo têm se desenvolvido significativamente. Neste trabalho, são realizados testes de desempenho envolvendo programas compostos por matrizes quadradas, cuja finalidade é determinar e contar a quantidade de números primos existentes em cada estrutura. Os algoritmos serão inicialmente executados de maneira serial e, posteriormente, de forma paralela, considerando o número de núcleos disponíveis no processador do equipamento utilizado pelos usuários nos testes.

DESENVOLVIMENTO

O desenvolvimento do trabalho consistirá nas comparações entre o processamento das matrizes de modo serial e depois em modo paralelo, com diferentes números de núcleos. Nesta seção, serão exibidos as tabelas de cada modo e o tamanho da matriz testada, e para o caso de paralelismo, o tamanho de cada macrobloco.

Tabelas de Testes

Tabela de Teste em Modo Serial			
tamanho matriz	Processador	AMD Ryzen 7 5700X3D	AMD Ryzen 5 5600
Matriz 20000 x 20000		15,776s	14,625s

Tabela de Teste em Modo Paralelo AMD Ryzen 7 5700X3D Matriz 20000 x 20000					
tamanho macrobloco	threads	2 Threads	4 Threads	8 Threads	16 Threads
1 x 1		42,013s	52,482s	61,522s	68,616s
1000 x 1000		8,003s	5,366s	2,153s	1,650s
5000 x 5000		8,030s	5,471s	3,116s	1,660s
10000 x 10000		8,021s	6,171s	5,478s	4,082s
20000 x 20000		15,841s	15,831s	15,824s	15,815s

Gráfico Referente a Tabela de Teste em Modo Paralelo do Processador AMD Ryzen 7 5700X3D com Macrobloco de tamanho 1000 x 1000

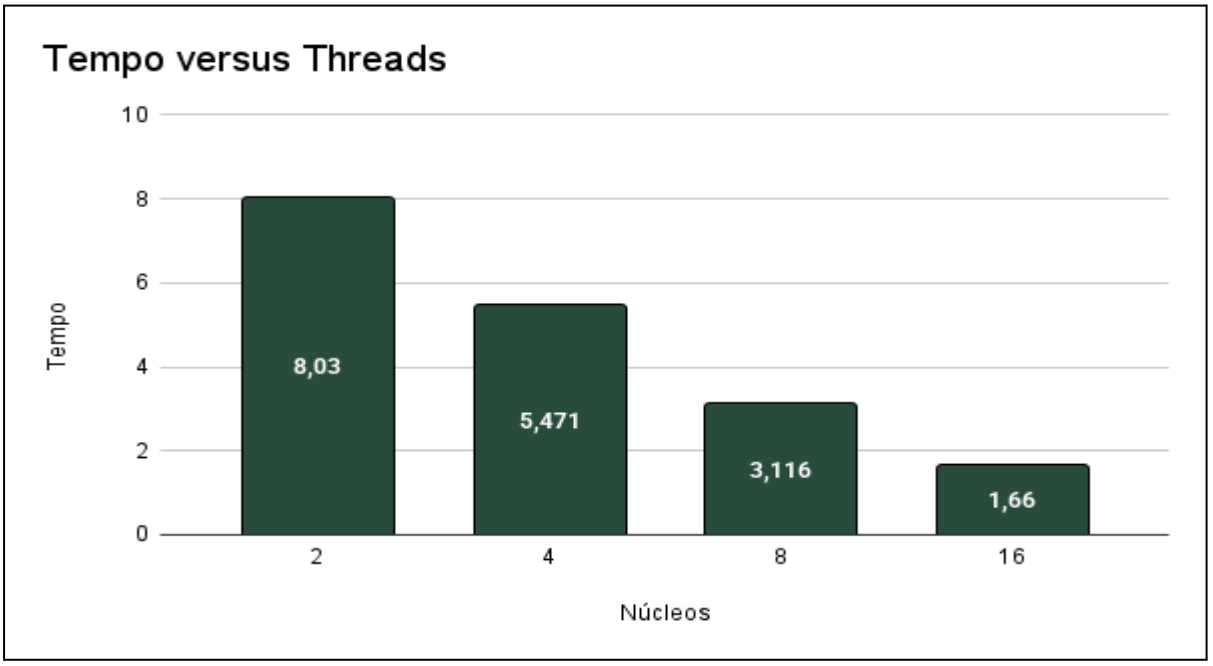
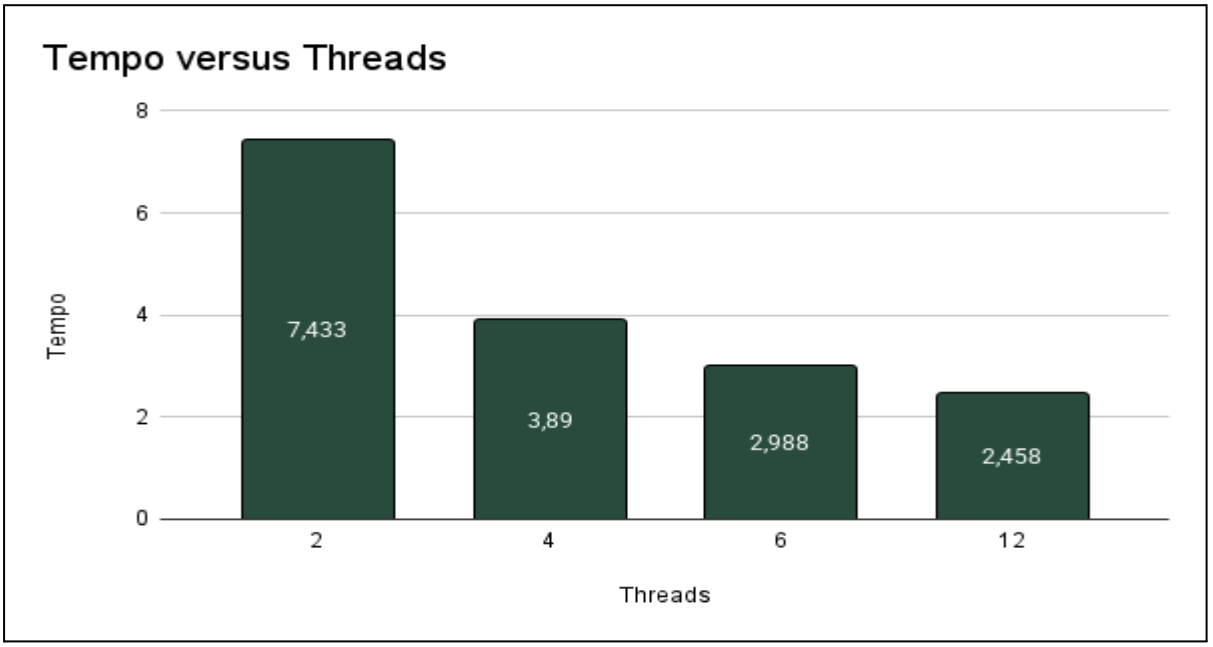


Tabela de Teste em Modo Paralelo AMD Ryzen 5 5600 Matriz 20000 x 20000					
tamanho macrobloco	threads	2 Threads	4 Threads	6 Threads	12 Threads
1 x1		39,039s	43,171s	49,096s	61,292s
1000 x 1000		7,457s	3,986s	2,716s	1,964s
5000 x 5000		7,433s	3,890s	2,988s	2,458s
10000 x 10000		7,380s	3,197s	3,911s	3,949s
20000 x 20000		14,555s	14,593s	14,624s	14,605s

Gráfico Referente a Tabela de Testes em Modo Paralelo do Processador AMD Ryzen 5 5600



TESTE MATRIZ 30000 x 30000

Tabela de Teste em Modo Serial AMD Ryzen 7 5700X3D			
30000 x 30000			
Tabela de Teste em Modo Paralelo AMD Ryzen 7 5700X3D			
tamanho macrobloco	threads	8 Threads	16 Threads
5000 x 5000		5,295s	4,313s
15000 x 15000		13,078s	13,609s
30000 x 30000		36,962s	35,816s

CONCLUSÃO

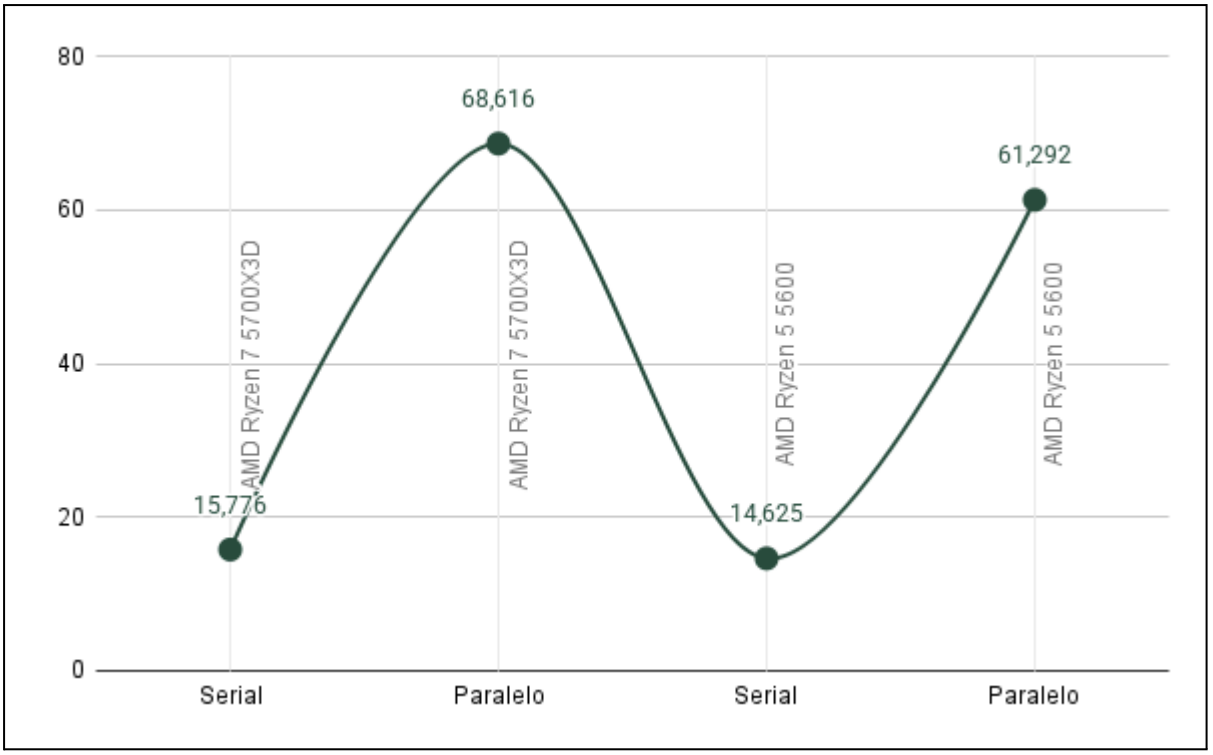
Analisando os resultados dos testes feitos, pode-se observar que migrar do processamento serial para o processamento paralelo geralmente resulta em uma redução significativa do tempo de processamento. No entanto, é crucial ressaltar que essa otimização não é universal e nem sempre acontece em todos os cenários. Para conseguir obter um desempenho favorável na paralelização depende de um equilíbrio cuidadoso entre os diversos fatores críticos.

Primeiramente, a granularidade das tarefas, que é definida pelo tamanho dos macroblocos, exerce uma forte influência. Quando os macroblocos estavam com tamanhos pequenos (1 x 1), o programa paralelo otimizado (utilizando todos os núcleos disponíveis de cada CPU) ficou até mais lento que o serial. O motivo para isso reside no ***overhead*** gerado em cada etapa do processo: desde o momento de requisitar um macrobloco, entrar nele, processar seus elementos e, por fim, sinalizar sua conclusão. Por outro lado, macroblocos muito grandes (com o mesmo tamanho da matriz) resultam em um baixo grau de paralelismo efetivo, subutilizando os núcleos disponíveis e se aproximando do desempenho do modo serial. Isso demonstra que a busca pela “faixa ideal” dos macroblocos é essencial para otimizar o balanceamento de carga de trabalho entre as threads.

Em segundo lugar, a quantidade de threads criadas é um elemento chave para a otimização. Enquanto números adequados de threads para cada CPU acelerou o processo significativamente, testes realizados com número de threads exorbitantemente grandes, como 700 threads, comprometeram totalmente o desempenho. Um número excessivo de de threads gerou ***overhead*** de gerenciamento, com o aumento de trocas de contexto e contenção por recursos utilizados por outras threads, levando a tempos de execução iguais ou até piores que a versão serial.

	Modo SERIAL	Modo PARALELO (macrobloco 1 x 1)
AMD Ryzen 7 5700X3D	15,776s	68,616s
AMD Ryzen 5 5600	14,625s	61,292s

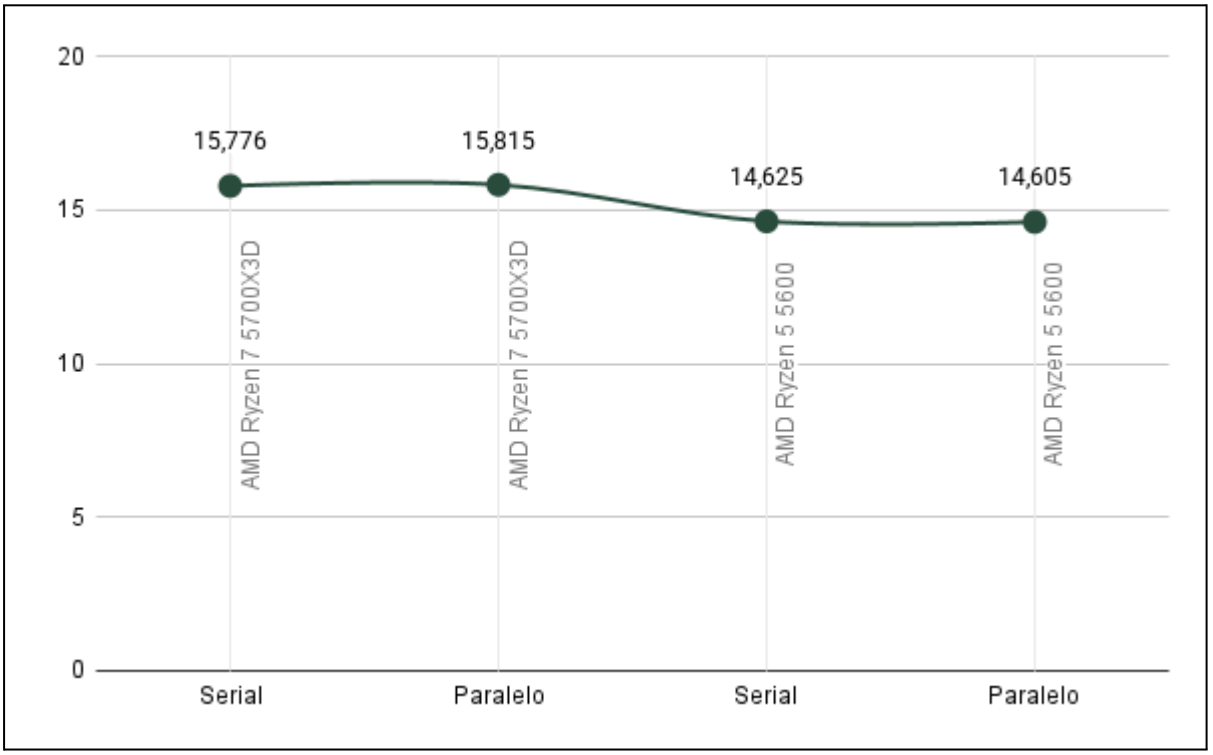
Gráfico Referente a Comparação Entre Modo Serial e Modo Paralelo(macrobloco 1 x 1) dos Processadores AMD Ryzen 7 5700X3D e AMD Ryzen 5 5600



Por outro lado, se o macrobloco for gigante (por exemplo a matriz inteira, 20000 x 20000), o tempo paralelo fica quase igual ao serial, porque só uma ou poucas threads trabalham de verdade, enquanto as outras ficam ociosas:

	Modo SERIAL	Modo PARALELO (macrobloco 20000 x 20000) (todos os núcleos)
AMD Ryzen 7 5700X3D	15,776s	15,815s
AMD Ryzen 5 5600	14,625s	14,605s

Gráfico Referente a Comparação Entre Modo Serial e Modo Paralelo(macrobloco 20000 x 20000) dos Processadores AMD Ryzen 7 5700X3D e AMD Ryzen 5 5600



Na faixa de tamanho 1000 x 1000, 5000 x 5000 dos macroblocos, o trabalho fica bem mais distribuído, e o overhead do paralelismo não atrapalha o processamento.

Quanto mais threads (até o limite de cada processador), melhor o desempenho. Por exemplo, no Ryzen 7 com macroblocos de tamanho 1000 x 1000:

- **1 núcleo** → 16,024 segundos
- **8 núcleos** → 2,153 segundos
- ***speedup:*** 7,44 vezes mais rápido quando usado todos os núcleos físicos.

No Ryzen 5, a mesma coisa:

- **1 núcleo** → 14,783 segundos
- **6 núcleos** → 2,716 segundos
- ***speedup:*** 5,44 vezes mais rápido quando usado todos os núcleos físicos.

Ao analisar o *speedup* obtido em cada processador, a Lei de Amdahl, que determina que a aceleração máxima de um programa é limitada pela parte do código que é inerentemente serial e não pode ser paralelizado, torna-se evidente e confirma que a utopia de 100% de paralelismo é inatingível, pois sempre haverá *overhead* de sincronização e gerenciamento das threads, além das partes do algoritmo que não são paralelizáveis, como dito anteriormente. Apesar da limitação teórica, o uso de multithread se mostra sim vantajoso. No Ryzen 7, o *speedup* alcançado foi de 7,44 vezes, comparado o tempo de execução de 1 thread com o tempo de 6 threads. Da mesma forma, no Ryzen 5, o *speedup* obtido foi de 5,44. Embora esses valores evoluíram linearmente, por conta da Lei de Amdahl, mostram a eficiência do paralelismo para tarefas intensas. O teste de uma matriz de 30000 x 30000 também evidencia isso, mostrando tempos de 36,962 segundos para 8 Threads e 35,816 segundos para 16 Threads no Ryzen 7, indicando que o paralelismo se mantém eficaz mesmo em cargas maiores de processamento.

Em relação ao uso de *mutex* para sincronização, tornou-se obrigatório proteger as variáveis compartilhadas (**`contadorPrimos`** e **`proximoMacrobloco`**) para garantir a integridade dos dados. Rodando o código com uma matriz de 20000 x 20000 e macroblocos com tamanho pequeno de 1 x 1, pudemos observar a vital importância dos semáforos. Na função onde existe a presença de semáforos protegendo a região crítica, foram encontrados **43.539.032 números primos**, valor que se mostrou igual ao da busca serial, confirmando a correção do algoritmo paralelo.

No entanto, ao rodar a função sem *mutex* (desativando as proteções), com as mesmas configurações de matriz e macroblocos, apesar de ter processado em quase metade do tempo comparado com a presença de semáforos, foi encontrado um número incorreto de primos: **49.143.168 números primos**. Esta discrepância gritante demonstra claramente o fenômeno de condição de corrida. A ausência de sincronização permitiu que múltiplas threads acessassem e modificassem as variáveis compartilhadas simultaneamente de forma desordenada, gerando resultados errados e inconsistentes entre as execuções.

Em contrapartida, notamos que em testes com macroblocos de tamanhos maiores, mesmo sem a proteção dos semáforos, a contagem de primos permaneceu consistente com a busca serial, pois a diminuição da frequência de acesso às regiões críticas reduziu a probabilidade de que as condições de corrida se manifestassem durante a execução. Sendo assim, a utilização de mutex (semáforo) é fundamental e necessário para garantir que o programa funcione corretamente e produza resultados confiáveis na programação paralela.

O que você pode aprender com esse trabalho?

Com esse trabalho, conseguimos aprender a complexidade e a delicadeza de se trabalhar com programação multithread. É legal analisar como a busca paralela pode ser drasticamente mais rápida que a busca serial em diferentes casos, mas às vezes, quando a aplicação do paralelismo não é feita da maneira correta o resultado pode ser prejudicial, tornando o processo mais lento comparado a forma serial.

Esta avaliação foi uma excelente maneira de provar o conhecimento teórico adquirido em aula, visto que, ao fazer um algoritmo paralelo é possível notar a necessidade dos elementos de controle de fluxo e tratamento de suas regiões críticas, assim como impactos que a falta ou a aplicação errada deles causa no resultado final.

Ao analisar os resultados, aprendemos que projetar um algoritmo paralelo eficaz exige um equilíbrio muito rígido entre dividir o trabalho e controlar a coordenação entre as threads, e que a Lei de Amdahl, que em AOC só aprendemos o conceito, vimos como ela acontece e é aplicada na prática, mostrando que as progressões de paralelismo não são lineares, muito menos 100% reais e perfeitas.

De forma resumida, esse trabalho nos fez amarrar as pontas soltas da nossa compreensão da matéria. Ao codificar uma tarefa de forma paralela, o uso dos semáforos e identificação de pontos críticos testa a aplicação da matéria, fazendo com que haja uma validação e uma compreensão mais aprofundada do assunto. Após alguns erros no código, a atividade convida a revisitar os materiais e expandir o conhecimento como um desafio que precisa ser vencido.