Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Engenharia da Computação e Automação - DCA
DCA3703 - Programação Paralela

Tarefa 05: Comparação entre programação sequencial e paralela

Discente: Quelita Míriam
Docente: Samuel Xavier de Souza

I. INTRODUÇÃO

A computação paralela é uma abordagem utilizada para aumentar o desempenho de algoritmos por meio da execução simultânea de instruções. Nesse contexto, este trabalho tem como objetivo desenvolver um programa em linguagem C capaz de contar a quantidade de números primos entre 2 e um valor máximo definido pelo usuário. Posteriormente, o código é paralelizado utilizando a diretiva *#pragma omp parallel for*, sem alterar a lógica original da contagem. Com isso, buscamos comparar o desempenho entre a versão sequencial e a paralela, avaliando o tempo de execução e possíveis divergências nos resultados.

A tarefa também permite a análise de aspectos importantes da programação paralela, como condições de corrida, sincronização de variáveis compartilhadas, e balanceamento de carga entre threads.

II. METODOLOGIA

O algoritmo[1] baseia-se em uma função que verifica se um número é primo, utilizando uma abordagem simples de divisão até a raiz quadrada do número. Duas versões do programa foram executadas:

- **Versão sequencial**: Um laço *for* percorre todos os números de 2 até n, incrementando uma variável de contagem a cada número primo identificado.
- **Versão paralela**: O mesmo laço foi paralelizado com a diretiva *#pragma omp parallel for*. Contudo, a variável de contagem foi utilizada sem nenhuma forma de proteção contra condições de corrida.

Foram comparados os tempos de execução das versões sequencial e paralela. Para isso, utilizamos a função *gettimeofday()* para medir o tempo de execução de cada parte do código. O resultado foi analisado para entender a eficiência da paralelização, bem como os problemas enfrentados, como a perda de contagem de números primos no caso da falta de proteção à zona crítica.

O código é compilado a partir de "*gcc main.c -o main -fopenmp -lm && ./main*" para os resultados e "*python plot.py*" para o plot de um gráfico para mais análises.

III. RESULTADOS E DISCUSSÕES

Ao compilar o código, foram observados a seguinte tabela contendo os tempos de execução e resultados da versão sequencial e da versão paralela (sem proteção):

[1] Link do código desenvolvido para a tarefa:
https://github.com/quelita2/programacao-paralela/blob/main/topico01/tarefa05/src/main.c

n	Primos (Seq)	Tempo (s) Seq	Primos (Par)	Tempo (s) Par	Diferença
1000	168	0.00004	152	0.00036	16
10000	1229	0.00124	1156	0.00040	73
100000	9592	0.02662	9373	0.01216	219
1000000	78498	0.43210	77939	0.22315	559
10000000	664579	10.03191	662978	5.30198	1601

Imagem 1: Resultado da compilação com condição de corrida

Na versão sequencial, a contagem de números primos foi precisa e constante em todas as execuções. Enquanto na versão paralela sem proteção da variável de contagem, os resultados foram inconsistentes, não batendo com os resultados da versão sequencial. Em diversas execuções, a contagem final foi menor do que a da versão sequencial, apresentando diferença na contagem de primos à medida que o número máximo é crescido, como mostrado na imagem da tabela de resultados acima.

Esse comportamento incorreto é causado por condições de corrida, pois múltiplas threads acessam e modificam simultaneamente a variável de contagem, resultando na perda de incrementos. Para corrigir esse problema, é necessário utilizar uma região crítica, como *#pragma omp critical*.

n	Primos (Seq)	Tempo (s) Seq	Primos (Par)	Tempo (s) Par	Diferença
1000	168	0.00004	168	0.00116	0
10000	1229	0.00120	1229	0.00045	0
100000	9592	0.02750	9592	0.01303	0
1000000	78498	0.41289	78498	0.19010	0
10000000	664579	9.84542	664579	5.31698	0

Imagem 2: Resultado da compilação sem condição de corrida

Agora, por meio da diretiva *#pragma omp critical* há proteção para a região crítica, logo, o resultado final da contagem não é afetado. Por isso, a diferença entre os resultados é nulo, pois são os mesmos para a versão sequencial e paralela.

O uso da diretiva evita erros, mas introduz sincronização, ou seja, as threads ficam esperando para acessar a variável e isso pode reduzir o desempenho, especialmente se muitas threads tentam acessar esse bloco crítico com frequência.

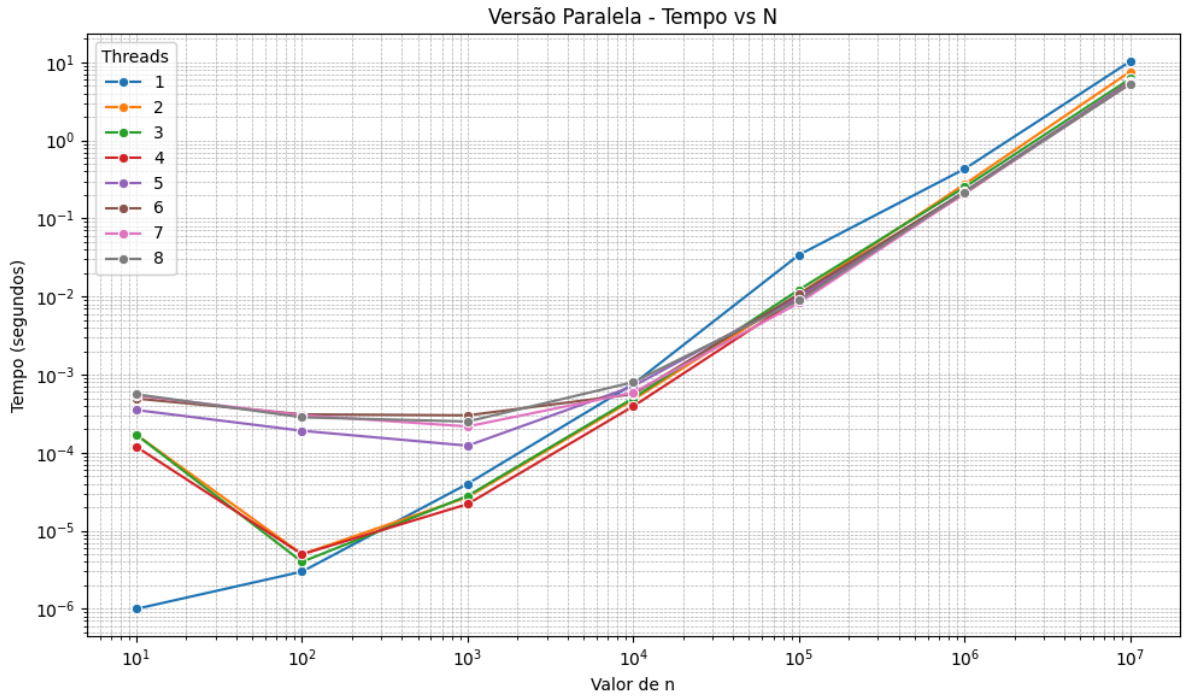

Gráfico 1: Tempo de execução vs N (número máximo) com a proteção à variável crítica para *multithreading*

O gráfico mostra o tempo de execução da versão paralela com *#pragma omp critical*, variando o número de threads de 1 a 8. Para valores pequenos de n, o uso de múltiplas *threads* não traz ganho de desempenho devido à sobrecarga de criação e sincronização das threads. Com o aumento de n, o tempo de execução melhora com mais *threads*, mas essa melhora se estabiliza a partir de 4 ou 5 *threads*. Isso ocorre porque a diretiva *critical* cria um gargalo, impedindo que múltiplas threads executem simultaneamente a região crítica, limitando o ganho de paralelismo.

São esses os desafios comuns em programação paralela:

- Desequilíbrio de carga: algumas threads terminam suas tarefas antes de outras, causando tempo ocioso.
- Condições de corrida: o uso incorreto de variáveis compartilhadas compromete a precisão dos resultados.
- Latência e sincronização: a comunicação e coordenação entre threads pode impactar negativamente o desempenho.

IV. CONCLUSÃO

A experiência com a paralelização da contagem de números primos em C permitiu uma compreensão prática dos desafios e vantagens da programação paralela. Foi possível perceber que a simples paralelização de um laço não garante melhorias de desempenho sem uma atenção cuidadosa às zonas críticas, balanceamento de carga, e sincronização de threads.

A versão paralela só se torna viável e confiável quando há proteção adequada das variáveis compartilhadas. Além disso, o uso de diretivas como *#pragma omp critical* do OpenMP mostrou uma solução eficiente para evitar condições de corrida e melhorar o desempenho da aplicação.

Por fim, esta tarefa reforça a importância de testar, validar e otimizar cuidadosamente algoritmos paralelos, especialmente em contextos onde a confiabilidade do resultado é essencial.