UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
Departamento de Engenharia de Computação e Automação

Programação Paralela
Juliane da Silva Santos

Relatório 5:
Comparação entre programação sequencial e paralela

1 Introdução

Um programa é visto como programação sequencial, quando o mesmo possui uma série de instruções sequenciais que devem ser executadas em um único processador. Já um programa paralelo é considerado programação paralela quando este é visto como um conjunto de partes que podem ser resolvidas concorrentemente, ou seja, cada parte é igualmente constituída por uma série de instruções sequenciais, mas que no seu conjunto podem ser executadas simultaneamente em vários processadores.

Na terefa 5, investigamos a eficiência da paralelização por meio da diretiva `#pragma omp parallel for`, da biblioteca OpenMP, na contagem de números primos em um intervalo de 2 até um valor máximo n. A partir da comparação com a versão sequencial do mesmo algoritmo, busca-se avaliar os ganhos de desempenho e refletir sobre os desafios iniciais da programação paralela.

O código completo pode ser visto no repositório disposto na plataforma GitHub.

2 Metodologia

O experimento foi realizado com as seguintes etapas:

- Implementação de uma função `eh_primo(int num)` que identifica números primos.

- Construção de duas versões da função de contagem: Uma versão sequencial e uma outra paralela utilizando OpenMP com a diretiva `#pragma omp parallel for`.

- Medição do tempo de execução de cada versão com a função **gettimeofday**().

- Execução do programa para diferentes valores de n, variando de 1.000.000 a 10.000.000 com incrementos de 1.000.000.

- Armazenamento dos dados obtidos (tempo e quantidade de primos) em um arquivo CSV.

- Geração de gráficos comparativos com a biblioteca Plotly em Python.

2.1 Implementação

A implementação foi feita em linguagem C com diretivas OpenMP. A seguir, estão descritas as principais partes do código:

- Definições e verificação para saber se um número é primo:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <omp.h>

int eh_primo(int num) {
    if (num < 2) return 0;
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0) return 0;
    }
    return 1;
}
```

Iniciamos o programa checando se o número é menor do que 2, caso seja, retornamos 0, caso contrario, entramos no primeiro loop do programa, onde de fato ocorre a verificação para saber se o numero é primo ou não. O loop começa em 2 (o primeiro número primo) e vai até a raiz quadrada do número (i*i <= num). Para cada i no intervalo, verifica se num é divisível por i (num % i == 0). Se encontrar algum divisor, retorna 0 (não é primo). Se o loop terminar sem encontrar divisores, retorna 1 (verdadeiro), indicando que o número é primo. Se encontrar algum divisor, retorna 0 (falso), senão retorna 1 (verdadeiro).

- Contagem de primos de forma sequencial e paralela:

```c
int contar_primos_sequencial(int max) {
    int count = 0;
    for (int i = 2; i <= max; i++) {
        if (eh_primo(i)) count++;
    }
    return count;
}

int contar_primos_paralelo(int max) {
    int count = 0;
    #pragma omp parallel for
    for (int i = 2; i <= max; i++) {
        if (eh_primo(i)) count++;
    }
```

```
15      return count;
16  }
```

A primeira função conta quantos números primos existem desde 2 até um valor máximo especificado (max), usando um método sequencial (passo a passo). Ele começa criando uma variável count inicializada em zero, que armazenará o total de números primos encontrados. O loop irá percorre todos os números inteiros de 2 até o valor máximo (max), um por vez. Para cada número (i), chama a função `eh_primo(i)` para verificar se é primo. Se o número for primo (quando `eh_primo(i)` retorna 1), incrementa o contador count em 1. Se não for primo, simplesmente passa para o próximo número. Após verificar todos os números, retorna o valor total armazenado em count, que representa quantos números primos existem até max.

A versão em paralelo faz a mesma contagem, mas de forma paralela, aproveitando múltiplos núcleos do processador. Inicializa, igualmente ao código sequencial, um contador count em zero, mas antes do for, utiliza uma diretiva `#pragma omp parallel for` que divide automaticamente o trabalho entre várias threads. O loop é distribuído entre os núcleos disponíveis do processador. Cada thread processa uma parte diferente do intervalo de números simultaneamente, entretanto todas as threads compartilham a verificação da primalidade dos números, quando encontra um primo, incrementa o contador compartilhado count++

- Calculo do tempo de execução:

```
1  double tempo_execucao(struct timeval inicio, struct timeval fim) {
2      return (fim.tv_sec - inicio.tv_sec) + (fim.tv_usec - inicio.tv_usec) /
           1e6;
3  }
```

Para medir o tempo de execução, foi recomendado em sala de aula o uso da função gettimeofday(), que captura o tempo total de execução (wall-clock time), incluindo pausas, esperas e operações de I/O. Essa abordagem difere da função clock(), que mede apenas o tempo de CPU efetivamente utilizado pelo processo, ignorando períodos de inatividade, como espera por I/O ou suspensão do programa. Para facilitar a medição do tempo, foi desenvolvida a função tempo_execucao(), que utiliza gettimeofday() e retorna o tempo atual em milissegundos. Essa função é chamada duas vezes: antes de chamarmos a função sequencial e paralela e no final, para o cálculo final do tempo de execução realizamos uma subtração do tempo final pelo inicial.

- Main

```
1  int main() {
2      struct timeval t1, t2;
3      int total_seq, total_par;
4
5      fprintf(fp, "MAX,Tempo_Sequencial,Primos_Sequencial,Tempo_Paralelo,
           Primos_Paralelo\n");
6
7      for (int max = 1000000; max <= 10000000; max += 1000000) {
8          // Sequencial
```

```
9           gettimeofday(&t1, NULL);
10          total_seq = contar_primos_sequencial(max);
11          gettimeofday(&t2, NULL);
12          double tempo_seq = tempo_execucao(t1, t2);
13
14          // Paralelo
15          gettimeofday(&t1, NULL);
16          total_par = contar_primos_paralelo(max);
17          gettimeofday(&t2, NULL);
18          double tempo_par = tempo_execucao(t1, t2);
19
20      return 0;
21  }
```

Neste bloco iremos chamar as funções criadas anteriormente e apresentar os resultados em um arquivo csv para a plotagem do gráfico e em prints na tela para a analise dos resultados.

3 Resultados e Discussão

Os testes mostraram que a versão paralela é significativamente mais rápida que a sequencial, especialmente para valores maiores de n, como mostra o gráfico 1.

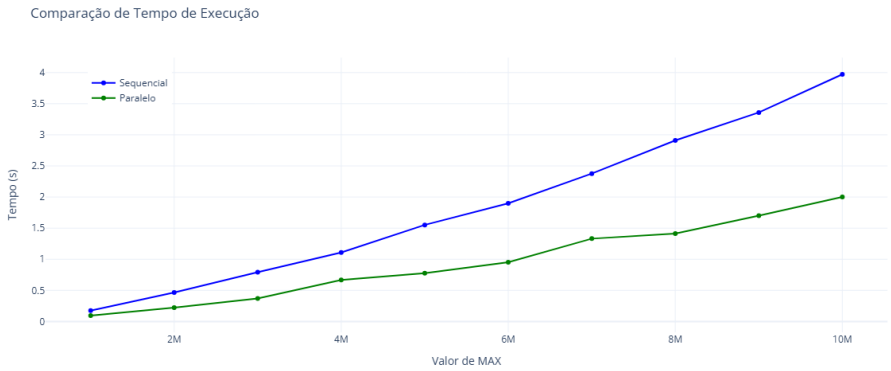

Figura 1: Comparação entre a versão sequencial (azul) e o paralelo (verde)

Apesar do tempo de execução ser menor para a versão em paralelo, é possível observar uma divergência nos resultados (quantidade de primos) entre as versões, como apresentada na tabela a seguir.

Tamanho	Operação	Contagem
1000000	Sequencial	78498 primos
	Paralelo	77316 primos
2000000	Sequencial	148933 primos
	Paralelo	147398 primos
4000000	Sequencial	283146 primos
	Paralelo	281144 primos
6000000	Sequencial	412849 primos
	Paralelo	410143 primos
10000000	Sequencial	664579 primos
	Paralelo	661216 primos

Esse resultado pode ser um exemplo de condição de corrida. Segundo Pacheco [1], uma condição de corrida ocorre quando duas ou mais threads acessam uma variável compartilhada simultaneamente, e pelo menos uma dessas operações é uma operação de escrita, podendo causar comportamentos indefinidos ou resultados incorretos. No programa paralelo todas as threads estão acessando e modificando simultaneamente a variável count. Como a variavel não está protegida contra acessos concorrentes duas ou mais threads podem sobrescrever o valor umas das outras, resultando em contagens erradas.

4 Conclusão

Este experimento demonstrou que a paralelização com OpenMP pode trazer ganhos significativos de desempenho em algoritmos computacionalmente intensivos, como a contagem de números primos. No entanto, também revelou que a programação paralela exige atenção a detalhes como concorrência e sincronização, sob pena de comprometer a exatidão dos dados.

A utilização das diretivas de paralelismo, conforme proposto por Pacheco (2011), aliada à compreensão da arquitetura de computadores discutida por Stallings (2010), é essencial para o desenvolvimento de aplicações eficientes e corretas.

Referências

[1] Peter. PACHECO. *An introduction to parallel programming*. Morgan Kaufmann, Massachusetts, 2011.