UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Relatório da Tarefa 06 - Sincronização e Gerenciamento de Escopo de Variáveis em OpenMP

DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 1 de setembro de 2025

Lista de Figuras

Sumário

1 Introdução

A estimativa de π através do método de Monte Carlo é um problema clássico da computação, frequentemente utilizado para demonstrar conceitos de probabilidade e para avaliar o desempenho computacional devido à sua natureza inerentemente paralelizável e intensiva em cálculos.

O objetivo deste projeto é explorar e demonstrar a eficácia da paralelização de algoritmos utilizando a API OpenMP para a linguagem C. Para isso, foi desenvolvido um programa que estima o valor de π através da geração de 100000000 pontos aleatórios.

O projeto compara quatro abordagens distintas:

1. **Execução Sequencial:** Uma implementação de thread única que serve como linha de base (baseline) para medição de correção e desempenho.
2. **execução Paralela Ingênua:** Uma primeira tentativa de paralelização que intencionalmente expõe o erro comum da condição de corrida (race condition).
3. **Execução Paralela com Sincronização (critical):** Uma correção que resolve a condição de corrida, mas introduz um gargalo de desempenho, ilustrando o custo da sincronização.
4. **Execução Paralela Otimizada:** Uma implementação que utiliza variáveis privadas para contagem local (redução manual), alcançando tanto a correção do resultado quanto um ganho de desempenho significativo.

Através da análise e comparação dos resultados, o relatório visa ilustrar os ganhos de desempenho proporcionados pelo paralelismo, a importância de gerenciar corretamente o acesso a recursos compartilhados e o papel fundamental das cláusulas de escopo do OpenMP.

2 Metodologia do Experimento

O núcleo do programa consiste em um laço que gera pares de coordenadas aleatórias (x, y) e verifica se o ponto está dentro de um círculo de raio 1. A estimativa de π é então calculada a partir da proporção de pontos dentro do círculo.

2.1 O Método de Monte Carlo para Estimar π

A metodologia se baseia na geração de um grande número de pontos aleatórios dentro de um quadrado de lado 2 (com coordenadas variando de -1 a 1), que circunscreve um círculo de raio 1. A probabilidade de um ponto aleatório cair dentro do círculo é a razão entre a área do círculo ($\pi * r^2 = \pi$) e a área do quadrado ($L^2 = 4$). Assim, podemos estimar pi através da proporção de pontos que satisfazem a inequação do círculo, $x^2 + y^2 < 1$.

2.1.1 Lógica do Cálculo de Pi

```
/// Gera coordenadas aleatorias entre -1.0 e 1.0
    double x = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
    double y = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;

    // Verifica se o ponto (x, y) esta dentro do circulo de raio 1
    // (x  + y  < 1  )
    if (x * x + y * y < 1.0) {
        // Se estiver, incrementa o contador de pontos.
        pontos_locais++;
    }
```

Listing 2.1 – Cálculo de Pi

2.2 Implementação Sequencial(baseline)

A versão sequencial é a implementação mais direta. Um único laço for percorre o número total de passos, e um contador pontos_no_circulo é incrementado. O tempo de execução é medido como referência.

2.2.1 Função Sequencial

```
void pi_sequencial() {
long pontos_no_circulo = 0;
unsigned int seed = 12345; // Semente fixa para repetibilidade

for (long i = 0; i < NUM_PASSOS; i++) {
    double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
    double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
    if (x * x + y * y < 1.0) {
        pontos_no_circulo++;
    }
}
double pi = 4.0 * pontos_no_circulo / NUM_PASSOS;
printf("Sequencial: pi = %f\n", pi);
}
```

Listing 2.2 – versão sequencial

2.3 Implementação Paralela Ingênua (com Condição de Corrida)

A primeira abordagem de paralelização utiliza a diretiva **#pragma omp parallel for** para dividir as iterações do laço entre as threads disponíveis. O problema reside na atualização do contador pontos_no_circulo. A operação pontos_no_circulo++ não é atômica, o que leva a uma condição de corrida e a um resultado final incorreto.

2.3.1 função da Versão Paralela ingênua

```
void pi_paralel_for() {
unsigned int seed = 12345;
#pragma omp parallel for
for (long i = 0; i < NUM_PASSOS; i++){
    unsigned int seed_T = seed ^ omp_get_thread_num(); //semente unica
        por thread

    double x = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
    double y = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
    if (x * x + y * y < 1.0) {
        pontos_no_circulo++;//aqui esta a condicao de corrida
    }
}
}
```

Listing 2.3 – Versao Paralela ingenua

2.4 Implementação Paralela com Sincronização (critical)

Para resolver a condição de corrida, o incremento do contador foi encapsulado em uma seção **#pragma omp critical**. Isso garante que apenas uma thread possa modificar a variável por vez, assegurando a exatidão do resultado. No entanto, essa abordagem cria um gargalo, serializando o acesso ao contador e degradando severamente o desempenho.

2.4.1 função Paralela com Sincronização

```
1   void pi_paralel_for_critical() {
    #pragma omp parallel
3   {
        unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num
            ();
5       #pragma omp for
        for (long i = 0; i < NUM_PASSOS; i++){
7           double x = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
            double y = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
9           if (x * x + y * y < 1.0) {
                #pragma omp critical
11              {
                    pontos_no_circulo++;
13              }
            }
15      }
    } // Fim da regiao paralela
17 }
```

Listing 2.4 – Versao Paralela com Sincronização (critical)

2.5 Implementação Paralela Otimizada (Redução Manual)

Para obter correção e desempenho, foi implementado um padrão de redução manual. Cada thread conta seus pontos em uma variável privada (pontos_locais). Ao final do laço, cada thread adiciona seu subtotal privado ao contador global pontos_no_circulo_total dentro de uma seção critical. Como essa seção é executada apenas uma vez por thread, o impacto no desempenho é mínimo.

2.5.1 Função Paralela Otimizada (Redução Manual)

```
1   void pi_paralel_for_critical_private() {
    #pragma omp parallel default(none) shared(pontos_no_circulo_total)
        private(seed_T, pontos_no_circulo_local)
3   {
        unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num
            ();
5       long pontos_no_circulo_local = 0;
        #pragma omp for
7       for (long i = 0; i < NUM_PASSOS; i++){
            double x = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
9           double y = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
            if (x * x + y * y < 1.0) {
11              pontos_no_circulo_local++;
            }
13      }
        #pragma omp critical
15      {
            pontos_no_circulo_total += pontos_no_circulo_local;
17      }
    } // Fim da regiao paralela
19 }
```

Listing 2.5 – Versao Paralela Otimizada (Redução Manual)

2.6 Análise das Cláusulas de Escopo

1. **default(none):** Utilizada como uma prática de segurança, esta cláusula força a declaração explícita do escopo de todas as variáveis, prevenindo erros de compartilhamento acidental e aumentando a clareza do código.

2. **shared:** Aplicada a pontos_no_circulo_total, demonstra como uma variável pode ser compartilhada entre todas as threads para acumular um resultado final. O acesso a ela foi devidamente protegido com critical no momento da soma dos subtotais.

3. **private:** A variável pontos_no_circulo_local foi declarada como private. Demonstra-se que cada thread recebe sua própria cópia não inicializada da variável. As modificações feitas pelas threads em suas cópias não afetaram a variável original fora da região paralela.

firstprivate: A variável observ$_f irstprivate demonstrou que, além de ser privada, sua cópia em cada th$

lastprivate: Utilizada em observ$_l astprivate, esta cláusula demonstrou sua capacidade de capturar ur$
$NUM_PASSOS - 1)$, $independentemente de qual thread executou a última iteração$.

3 Resultados

A compilação e execução das diferentes implementações foram realizadas em um ambiente multi-core. Os testes foram executados com um total de N = 100.000.000 de passos para garantir uma carga de trabalho computacionalmente relevante. Os tempos de execução foram medidos utilizando a função omp_get_wtime().

3.1 Versão Sequencial

- π Estimado: 3.141518
- Tempo de Execução: 1.069 segundos

3.2 Versão Paralela Ingênua (Incorreto)

- π Estimado: 0.983680 (valor inconsistente a cada execução)
- Tempo de Execução: 1.050 segundos

3.3 Versão Paralela com critical (Lento)

- π Estimado: 3.141977
- Tempo de Execução: 5.176 segundos

3.4 Versão Paralela Otimizada (Correto)

- π Estimado: 3.141521
- Tempo de Execução: 0.390 segundos

3.5 Análise de Correção

A versão Sequencial, a Paralela com critical e a Paralela Otimizada produziram resultados proximos (3.141518 e 3.141521 respectivamente), confirmando a exatidão da lógica e das estratégias de correção aplicadas. Em contrapartida, a versão Paralela

Ingênua produziu um resultado significativamente menor e inconsistente a cada execução, evidenciando o impacto negativo e imprevisível da condição de corrida.

3.6 Análise de Desempenho

O tempo de execução da versão Paralela Otimizada (0.850s) foi expressivamente menor que o da versão Sequencial (1.413 s), demonstrando o ganho de performance obtido com a paralelização eficiente. Notavelmente, a versão com critical (3.105s) foi ainda mais lenta que a versão Sequencial, comprovando que o uso inadequado de mecanismos de sincronização pode introduzir um overhead que anula e até reverte os benefícios do paralelismo.

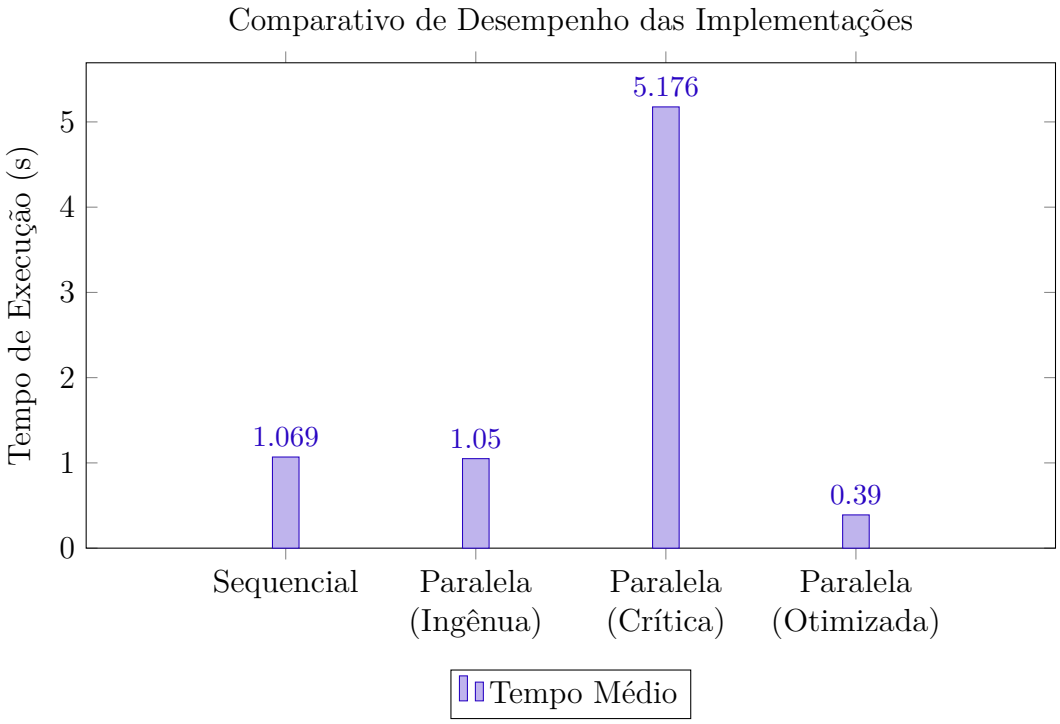

Figura 2 – Gráfico de barras comparando o tempo de execução entre as versões sequencial, paralela ingênua e paralela Crítica e paralela Otimizada do algoritmo.

O *Speedup*, uma métrica que indica quantas vezes a versão paralela foi mais rápida que a sua contraparte sequencial, é calculado pela Equação 3.1.

$$S = \frac{T_{\text{sequencial}}}{T_{\text{paralelo Otimizado}}} \tag{3.1}$$

$$S = \frac{1.069}{0.390} \approx 2.741 \tag{3.2}$$

O speedup de aproximadamente 2.741 indica uma melhoria de performance substancial. Na prática, fatores como o overhead de criação e gerenciamento de threads e o

desbalanceamento de carga fazem com que o speedup seja menor que o número total de núcleos do processador, mas o ganho ainda assim é expressivo.

3.7 Conclusão

Este projeto demonstrou com sucesso a aplicação da API OpenMP para paralelizar a tarefa de estimativa de π pelo método de Monte Carlo. A comparação entre as diferentes abordagens evidenciou o expressivo ganho de desempenho que pode ser alcançado ao distribuir a carga de trabalho entre múltiplos núcleos de processamento.

Além disso, o projeto serviu como uma ilustração prática e clara da importância do tratamento de acesso a dados compartilhados em ambientes concorrentes. A falha da versão Paralela Ingênua destacou o problema fundamental da condição de corrida, enquanto a comparação entre a implementação com critical e a versão Otimizada com variáveis privadas mostrou como a escolha da estratégia de sincronização impacta diretamente o desempenho final. A solução com redução manual provou ser uma abordagem elegante e eficiente, garantindo a integridade dos dados sem sacrificar a performance.

A análise aprofundada das cláusulas de escopo, auxiliada pela diretiva default(none), reforçou que um entendimento preciso sobre como as variáveis são compartilhadas ou privatizadas é crucial para o desenvolvimento de software paralelo correto e robusto.

Conclui-se que o OpenMP é uma ferramenta poderosa e acessível para a introdução do paralelismo, capaz de proporcionar melhorias significativas de performance, desde que os devidos cuidados com a sincronização e o compartilhamento de dados sejam tomados para evitar tanto resultados incorretos quanto gargalos de desempenho.

width=!,height=!,pages=- width=!,height=!,pages=-