Estimativa de π com OpenMP

Um Estudo sobre Sincronização e Gerenciamento de Escopo de Variáveis

Werbert Arles de Souza Barradas

Universidade Federal do Rio Grande do Norte (UFRN)
Disciplina de Programação Paralela - DCA3703

31 de agosto de 2025

Introdução

O Desafio: Estimar π

A estimativa de π com Monte Carlo é uma tarefa computacionalmente intensiva, ideal para testar os limites do paralelismo.

Objetivo do Estudo

Analisar o ganho de desempenho (speedup), demonstrar o problema da condição de corrida e explorar soluções de sincronização e gerenciamento de escopo em OpenMP.

As 4 Abordagens Analisadas

- **Sequencial:** Nossa linha de base (baseline) para medir corretude e desempenho.
- **Paralela Ingênua:** Expõe a condição de corrida.
- **Paralela com `critical`:** Garante a correção, mas com um alto custo de desempenho.
- **Paralela Otimizada:** Usa variáveis privadas para alcançar correção e velocidade.

Metodologia: A Lógica Central

Cálculo de Pi (Método de Monte Carlo)

A lógica principal se baseia em gerar um ponto aleatório (x, y) e verificar se ele pertence ao círculo unitário, conforme a inequação $x^2 + y^2 < 1$.

```
// Gera coordenadas aleatorias entre -1.0 e 1.0
double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
// Verifica se o ponto (x, y) esta dentro do circulo
if (x * x + y * y < 1.0) {
    // Se estiver, incrementa o contador de pontos.
    pontos_no_circulo++;
}
```

Metodologia: Implementações Iniciais

1. Versão 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);
}
```

Metodologia: Implementações Iniciais

2. 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
        }
    }
}
```

Metodologia: As Correções

3. Correção com `critical` (Lenta)

```
void pi_paralel_for_critical() {
    #pragma omp parallel
    {
        unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num
            ();
        #pragma omp for
        for (long i = 0; i < NUM_PASSOS; i++){
            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) {
                #pragma omp critical
                {
                    pontos_no_circulo++;
                }
            }
        }
    } // Fim da regiao paralela
}
```

Metodologia: As Correções

4. Versão Otimizada (Rápida)

```
void pi_paralel_for_critical_private() {
    #pragma omp parallel default(none) shared(pontos_no_circulo_total) private(
        seed_T, pontos_no_circulo_local)
    {
        unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num();
        long pontos_no_circulo_local = 0;
        #pragma omp for
        for (long i = 0; i < NUM_PASSOS; i++){
            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_local++;
            }
        }
        #pragma omp critical
        {
            pontos_no_circulo_total += pontos_no_circulo_local;
        }
    } // Fim da regiao paralela
}
```

Análise das Cláusulas de Escopo

As cláusulas de escopo gerenciam o compartilhamento de variáveis em uma região paralela.

- **shared:** Variáveis compartilhadas entre todas as threads. O acesso deve ser sincronizado para evitar condições de corrida.
- **private:** Cada thread recebe uma cópia local da variável, trabalhando de forma isolada.
- **firstprivate:** Similar a 'private', mas a cópia local é inicializada com o valor da variável original.
- **lastprivate:** A variável privada da última thread a completar o laço tem seu valor transferido de volta para a variável original.

O Papel de `default(none)`

Esta cláusula de segurança força a declaração explícita do escopo de todas as variáveis, prevenindo erros e tornando o código mais claro e seguro.

Por Que a Versão Ingênua Falha?

A Operação Não-Atômica

A instrução `x++` não é única. O processador a executa em três passos:

1. Ler o valor de 'x' da memória.
2. Incrementar o valor no registrador.
3. Escrever o novo valor de volta.

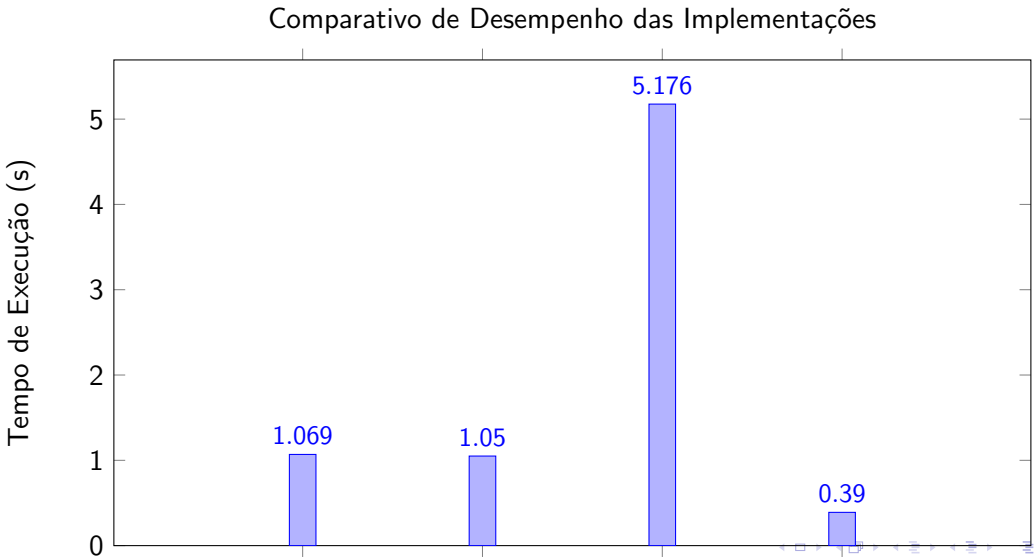
Múltiplas threads podem executar o passo 1 antes que qualquer uma chegue ao passo 3.

Cenário de Conflito

- Thread A lê 'total' (ex: 100).
- Thread B lê 'total' (ainda 100).
- Thread A escreve '101'.
- Thread B escreve '101'.

Resultado: Dois pontos foram encontrados, mas o contador foi incrementado apenas uma vez.

Comparativo de Tempo de Execução para $N = 10^8$



Análise dos Resultados

Análise de Correção

- A versão Paralela (Ingênua) produz um resultado incorreto.
- As versões **Sequencial**, **Paralela (Crítica)** e **Paralela (Otimizada)** chegam ao valor correto de π.

Análise de Desempenho (Speedup)

Comparando a versão Sequencial com a Otimizada, que são as duas corretas e relevantes para performance:

$$S = \frac{T_{sequencial}}{T_{paralelo}} = \frac{1.069s}{0.390s} \approx 2.74$$

- A paralelização otimizada resultou em um programa **2.74 vezes mais rápido**.
- A versão com `critical` (5.176s) foi a mais lenta de todas, provando o alto custo de sincronização excessiva.

Conclusão

Resultados do Estudo

- O experimento demonstrou o potencial de ganho de desempenho do OpenMP e os riscos da programação concorrente.
- A paralelização otimizada gerou um **speedup significativo de 2.74x**.
- O acesso não sincronizado a recursos compartilhados leva a resultados incorretos.
- A escolha da estratégia de sincronização é crucial: `critical` no laço degrada a performance, enquanto a redução manual a otimiza.

Implicação Prática

Compreender e aplicar mecanismos de sincronização e o gerenciamento de escopo de variáveis é fundamental para o desenvolvimento de software paralelo que seja não apenas rápido, mas também correto.

prog_sequencial.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h> // Cabeçalho necessário para omp_get_wtime()

// Definição global do número de passos para consistência
const long NUM_PASSOS = 100000000;


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);
}


int main() {
    double start_time, end_time;

    printf("Iniciando análise de desempenho para %ld passos.\n", NUM_PASSOS);

    // Teste Sequencial
    start_time = omp_get_wtime();
    pi_sequencial();
    end_time = omp_get_wtime();
    double tempo_sequencial = end_time - start_time;
    printf("Tempo Sequencial: %f segundos\n", tempo_sequencial);

    return 0;
}
```

prog_paralel_for.c

```c
#define _POSIX_C_SOURCE 200809L

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

// Definição global do número de passos para consistência
const long NUM_PASSOS = 100000000;
long pontos_no_circulo = 0;//variavel conpartilhada pelas thread
unsigned int seed = 12345;

//versão Paralela
void pi_paralel_for() {

        #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 condição de corrida
            }
        }
}

int main() {
    double start_time, end_time;

    printf("Iniciando analise de desempenho para %ld passos.\n", NUM_PASSOS);
    start_time = omp_get_wtime();
    pi_paralel_for();
    end_time = omp_get_wtime();
    double tempo_paralelo = end_time - start_time;
    double pi_estimado = 4.0 * pontos_no_circulo / NUM_PASSOS;

    printf("Estimativa paralela de pi = %f\n", pi_estimado);
    printf("Tempo Paralelo: %f segundos\n", tempo_paralelo);

    return 0;
}
```

prog_paralel_for_critical.c

```c

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// Definição global do número de passos para consistência
const long NUM_PASSOS = 100000000;
long pontos_no_circulo = 0;//variavel conpartilhada pelas thread

//versão Paralela
void pi_paralel_for_critical() {

    #pragma omp parallel
    {
        unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num();
        #pragma omp for
        for (long i = 0; i < NUM_PASSOS; i++){
            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) {
                #pragma omp critical
                {
                    pontos_no_circulo++;
                }
            }
        }
    } // Fim da região paralela
}

int main() {
    double start_time, end_time;

    printf("Iniciando analise de desempenho para %ld passos.\n", NUM_PASSOS);
    start_time = omp_get_wtime();
    pi_paralel_for_critical();
    end_time = omp_get_wtime();
    double tempo_paralelo = end_time - start_time;
    double pi_estimado = 4.0 * pontos_no_circulo / NUM_PASSOS;

    printf("Estimativa paralela de pi = %f\n", pi_estimado);
    printf("Tempo Paralelo: %f segundos\n", tempo_paralelo);

    return 0;
}

```

prog_paralel_for_critical_private.c

```c

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// Definição global do número de passos para consistência
const long NUM_PASSOS = 100000000;
long pontos_no_circulo_total = 0;

//versão Paralela
void pi_paralel_for_critical_private() {

    #pragma omp parallel
    {
        unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num();
        long pontos_no_circulo_local = 0;
        #pragma omp for
        for (long i = 0; i < NUM_PASSOS; i++){
            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_local++;
            }
        }
        #pragma omp critical
        {
            pontos_no_circulo_total += pontos_no_circulo_local;
        }

    } // Fim da região paralela
}

int main() {
    double start_time, end_time;

    printf("Iniciando analise de desempenho para %ld passos.\n", NUM_PASSOS);
    start_time = omp_get_wtime();
    pi_paralel_for_critical_private();
    end_time = omp_get_wtime();
    double tempo_paralelo = end_time - start_time;
    double pi_estimado = 4.0 * pontos_no_circulo_total / NUM_PASSOS;

    printf("Estimativa paralela de pi = %f\n", pi_estimado);
    printf("Tempo Paralelo: %f segundos\n", tempo_paralelo);

    return 0;
}

```