

# Estimativa de $\pi$ 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

## O Desafio: Estimar $\pi$

A estimativa de  $\pi$  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.

## 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++;
}
```

## 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);
}
```

## 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
        }
    }
}
```

## 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 regioao paralela
}
```

## 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 regioao paralela
}
```

# 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

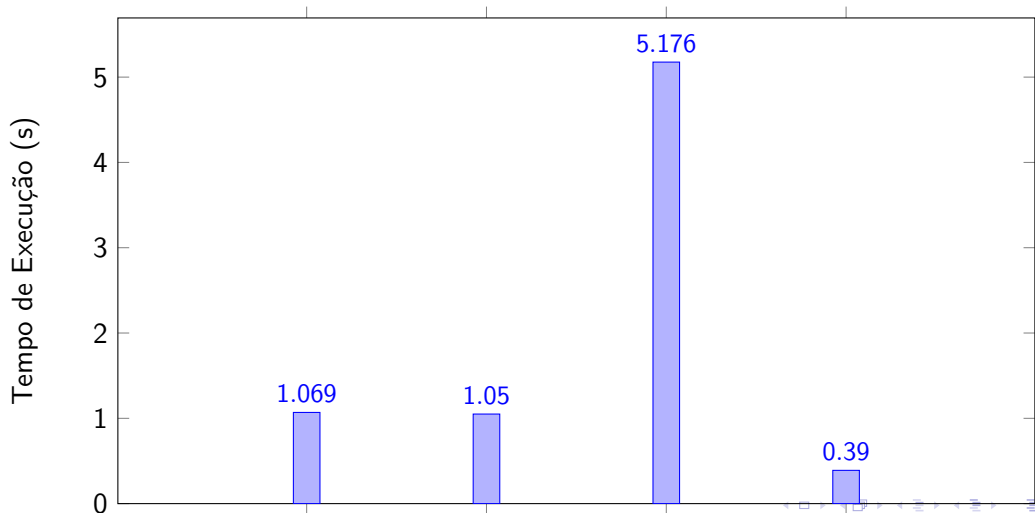
- Thread A lê 'total' (ex: 100).
- Thread B lê 'total' (ainda 100).
- Thread A escreve '101'.
- Thread B escreve '101'.

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



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

Comparativo de Desempenho das Implementações



# 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  $\pi$ .

## Análise de Desempenho (Speedup)

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

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

- 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.

## 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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <omp.h> // Cabeçalho necessário para omp_get_wtime()
5
6  // Definição global do número de passos para consistência
7  const long NUM_PASSOS = 100000000;
8
9
10 void pi_sequencial() {
11     long pontos_no_circulo = 0;
12     unsigned int seed = 12345; // Semente fixa para repetibilidade
13
14     for (long i = 0; i < NUM_PASSOS; i++) {
15         double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
16         double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0;
17         if (x * x + y * y < 1.0) {
18             pontos_no_circulo++;
19         }
20     }
21     double pi = 4.0 * pontos_no_circulo / NUM_PASSOS;
22     printf("Sequencial: pi = %f\n", pi);
23 }
24
25
26 int main() {
27     double start_time, end_time;
28
29     printf("Iniciando análise de desempenho para %ld passos.\n", NUM_PASSOS);
30
31     // Teste Sequencial
32     start_time = omp_get_wtime();
33     pi_sequencial();
34     end_time = omp_get_wtime();
35     double tempo_sequencial = end_time - start_time;
36     printf("Tempo Sequencial: %f segundos\n", tempo_sequencial);
37
38     return 0;
39 }
```

## prog\_paralel\_for.c

```
1  #define _POSIX_C_SOURCE 200809L
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <omp.h>
6
7  // Definição global do número de passos para consistência
8  const long NUM_PASSOS = 100000000;
9  long pontos_no_circulo = 0; //variavel compartilhada pelas thread
10 unsigned int seed = 12345;
11
12 //versão Paralela
13 void pi_paralel_for() {
14
15     #pragma omp parallel for
16
17     for (long i = 0; i < NUM_PASSOS; i++){
18         unsigned int seed_T = seed ^ omp_get_thread_num(); //semente unica
19         por thread
20
21         double x = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
22         double y = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
23         if (x * x + y * y < 1.0) {
24             pontos_no_circulo++; //aqui esta a condição de corrida
25         }
26     }
27
28 int main() {
29     double start_time, end_time;
30
31     printf("Iniciando analise de desempenho para %ld passos.\n", NUM_PASSOS);
32     start_time = omp_get_wtime();
33     pi_paralel_for();
34     end_time = omp_get_wtime();
35     double tempo_paralelo = end_time - start_time;
36     double pi_estimado = 4.0 * pontos_no_circulo / NUM_PASSOS;
37
38     printf("Estimativa paralela de pi = %f\n", pi_estimado);
39     printf("Tempo Paralelo: %f segundos\n", tempo_paralelo);
40
41     return 0;
42 }
```

## prog\_paralel\_for\_critical.c

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <omp.h>
5 #include <time.h>
6
7 // Definição global do número de passos para consistência
8 const long NUM_PASSOS = 100000000;
9 long pontos_no_circulo = 0; // variavel compartilhada pelas thread
10
11 // versão Paralela
12 void pi_paralel_for_critical() {
13
14     #pragma omp parallel
15     {
16         unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num();
17         #pragma omp for
18         for (long i = 0; i < NUM_PASSOS; i++){
19             double x = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
20             double y = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
21
22             if (x * x + y * y < 1.0) {
23                 #pragma omp critical
24                 {
25                     pontos_no_circulo++;
26                 }
27             }
28         }
29     } // Fim da região paralela
30 }
31
32 int main() {
33     double start_time, end_time;
34
35     printf("Iniciando analise de desempenho para %ld passos.\n", NUM_PASSOS);
36     start_time = omp_get_wtime();
37     pi_paralel_for_critical();
38     end_time = omp_get_wtime();
39     double tempo_paralelo = end_time - start_time;
40     double pi_estimado = 4.0 * pontos_no_circulo / NUM_PASSOS;
41
42     printf("Estimativa paralela de pi = %f\n", pi_estimado);
43     printf("Tempo Paralelo: %f segundos\n", tempo_paralelo);
44
45     return 0;
46 }
47
```

## prog\_paralel\_for\_critical\_private.c

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <omp.h>
5 #include <time.h>
6
7 // Definição global do número de passos para consistência
8 const long NUM_PASSOS = 100000000;
9 long pontos_no_circulo_total = 0;
10
11 //versão Paralela
12 void pi_paralel_for_critical_private() {
13
14     #pragma omp parallel
15     {
16         unsigned int seed_T = (unsigned int)time(NULL) ^ omp_get_thread_num();
17         long pontos_no_circulo_local = 0;
18         #pragma omp for
19         for (long i = 0; i < NUM_PASSOS; i++){
20             double x = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
21             double y = (double)rand_r(&seed_T) / RAND_MAX * 2.0 - 1.0;
22
23             if (x * x + y * y < 1.0) {
24                 pontos_no_circulo_local++;
25             }
26         }
27         #pragma omp critical
28         {
29             pontos_no_circulo_total += pontos_no_circulo_local;
30         }
31     } // Fim da região paralela
32 }
33
34 int main() {
35     double start_time, end_time;
36
37     printf("Iniciando analise de desempenho para %ld passos.\n", NUM_PASSOS);
38     start_time = omp_get_wtime();
39     pi_paralel_for_critical_private();
40     end_time = omp_get_wtime();
41     double tempo_paralelo = end_time - start_time;
42     double pi_estimado = 4.0 * pontos_no_circulo_total / NUM_PASSOS;
43
44     printf("Estimativa paralela de pi = %f\n", pi_estimado);
45     printf("Tempo Paralelo: %f segundos\n", tempo_paralelo);
46
47     return 0;
48 }
49
50
```