

# Tarefa 6 - Estimativa Estocástica de $\pi$ com OpenMP

## Método de Monte Carlo

O método de Monte Carlo estima  $\pi$  através de simulação estatística:

**Conceito:** Gerar pontos aleatórios em um quadrado  $[-1,1]^2$  e contar quantos caem dentro do círculo de raio 1

**Fórmula:**  $\pi \approx 4 \times (\text{pontos\_dentro\_círculo} / \text{total\_pontos})$

## Implementações Desenvolvidas

### 1. Versão Sequencial (Referência)

**VERSÃO BASE:** Implementação sequencial para comparação de performance

```
double estimar_pi_sequencial(long num_pontos) {
    long pontos_dentro = 0;
    for (long i = 0; i < num_pontos; i++) {
        double x = random(-1, 1);
        double y = random(-1, 1);
        if (x*x + y*y <= 1.0) {
            pontos_dentro++;
        }
    }
    return 4.0 * pontos_dentro / num_pontos;
}
```

### 2. Versão com Condição de Corrida usando #pragma omp parallel for

**PROBLEMA CRÍTICO:** Múltiplas threads modificam a mesma variável simultaneamente sem sincronização!

```
long pontos_dentro = 0; // Variável compartilhada sem
proteção!

#pragma omp parallel for
for (long i = 0; i < num_pontos; i++) {
    // ... cálculos ...
    if (x*x + y*y <= 1.0) {
        pontos_dentro++; // CONDIÇÃO DE CORRIDA!
    }
}
```

### 🔍 Por que `pontos\_dentro++` é perigoso em paralelo?

A operação `pontos_dentro++` parece ser uma única instrução atômica, mas o processador na verdade executa **3 operações separadas**:

- 1. LOAD (Carregar):** Buscar o valor atual de `pontos_dentro` da memória para o registrador
- 2. INCREMENT (Incrementar):** Somar 1 ao valor no registrador
- 3. STORE (Armazenar):** Escrever o novo valor do registrador de volta para a memória

### ⚡ Race Condition em Ação: O Cenário do Desastre

Imagine que `pontos_dentro = 15.000` e duas threads (A e B) encontram pontos dentro do círculo simultaneamente:

#### Timeline da Execução Paralela:

Thread A: LOAD `pontos_dentro` → registrador\_A = 15.000

Thread B: LOAD `pontos_dentro` → registrador\_B = 15.000 ← **MESMA LEITURA!**

Thread A: INCREMENT registrador\_A → registrador\_A = 15.001

Thread B: INCREMENT registrador\_B → registrador\_B = 15.001

Thread A: STORE registrador\_A → `pontos_dentro` = 15.001

Thread B: STORE registrador\_B → `pontos_dentro` = 15.001 ← **SOBRESCREVE!**

**✗ PROBLEMA FUNDAMENTAL:** *Lost Update* - Ambas threads leram o mesmo valor inicial

📊 **RESULTADO INCORRETO:** pontos\_dentro = 15.001 (deveria ser 15.002)

💥 **IMPACTO CRÍTICO:** Uma contagem perdida = Estimativa de  $\pi$  incorreta

🔄 **FREQUÊNCIA:** Acontece milhares de vezes em execuções paralelas!

#### ✓ **ANALOGIA PRÁTICA:**

É como duas pessoas tentando atualizar o mesmo documento ao mesmo tempo - uma das alterações sempre se perde porque ambas trabalharam com a versão antiga!

#### 🕒 **POR QUE ISSO IMPORTA NO MONTE CARLO:**

- Cada ponto perdido = erro na estimativa de  $\pi$
- Com milhões de pontos, centenas de milhares podem ser perdidos
- Resultado:  $\pi$  calculado será sempre menor que o valor real

#### Resultados Inconsistentes Observados:

Execução	$\pi$ Estimado	Erro	Tempo (s)	Explicação
1	0.453920	85.551%	0.0115	Incrementos perdidos devido à corrida entre threads
2	0.448800	85.714%	0.0035	
3	0.709560	77.414%	0.0074	

### 3. Correção com #pragma omp critical

**CORREÇÃO FUNCIONAL:** Apenas uma thread por vez pode incrementar a variável

```
#pragma omp parallel for
for (long i = 0; i < num_pontos; i++) {
    // ... cálculos ...
    if (x*x + y*y <= 1.0) {
        #pragma omp critical
        pontos_dentro++; // Protegido contra race condition
    }
}
```

```
}  
}
```

⚠️ **POR QUE ESTA ABORDAGEM NÃO É ÓTIMA:**

🔧 **O Gargalo em Ação:**

- **Threads esperando:** Quando uma thread está no critical, todas as outras ficam bloqueadas
- **Serialização forçada:** 8 threads executando como se fosse 1 thread
- **Overhead de sincronização:** Cada critical section tem custo computacional

📊 **Análise de Performance (100k pontos, ~78k hits):**

- **Tempo total:** 0.0170s (vs 0.0062s sequencial)
- **Speedup:** 0.36x (na verdade ficou **2.7x mais lento!**)
- **Critical sections executadas:** ~78.000 (uma para cada ponto dentro do círculo)
- **Tempo perdido:** Threads ficam 60% do tempo esperando na fila do critical

**4. Reestruturação com #pragma omp parallel seguido de #pragma omp for**

**SOLUÇÃO INTELIGENTE:** Separar a região paralela do loop e usar acumulação local

🔗 **Por que esta abordagem é ótima?**

⚡ **Comparação de Sincronizações:**

Abordagem	Sincronizações	Exemplo (100k pontos, ~78k hits)
#pragma omp critical dentro do loop	A cada incremento	<b>~78.000 sincronizações!</b> Cada hit = 1 critical section
#pragma omp parallel + acumulação local	Uma por thread	<b>8 sincronizações</b> Uma por thread no final

🔧 **Funcionamento Passo a Passo:**

1. **Criação das Threads:** #pragma omp parallel cria 8 threads
2. **Variáveis Locais:** Cada thread tem sua própria pontos\_locais = 0

3. **Divisão do Trabalho:** `#pragma omp for` divide 100k iterações entre threads
4. **Acumulação Local:** Cada thread incrementa apenas sua variável local
5. **Sincronização Final:** Apenas no final, cada thread adiciona sua soma ao total

#### 🏆 Resultados da Performance:

- **Tempo:** 0.0003s vs 0.0170s do critical (56x mais rápido!)
- **Speedup:** 20.7x comparado ao sequencial
- **Precisão:** Mantém a correção completa dos resultados
- **Escalabilidade:** Performance melhora linearmente com mais cores

## Demonstrações Detalhadas das Cláusulas OpenMP

### PRIVATE

**Comportamento:** Cada thread tem sua própria cópia

- **Valor inicial é INDEFINIDO**
- Modificações não afetam outras threads
- Valor não é preservado após região paralela

#### 🔒 DEMONSTRAÇÃO: PRIVATE

⬇ **ANTES:** `variavel = 100`

#### ▣ DURANTE A EXECUÇÃO:

Thread 0: `variavel = ???`

Thread 1: `variavel = ???`

Thread 2: `variavel = ???`

🔗 Cada thread tem sua cópia, valor inicial indefinido

⬆ **DEPOIS:** `variavel = 100` (inalterada)

### FIRSTPRIVATE

**Comportamento:** Como private, mas inicializada

- Cada thread recebe **CÓPIA** do valor inicial
- Útil quando threads precisam do valor original
- Modificações locais não afetam variável original

#### DEMONSTRAÇÃO: FIRSTPRIVATE

↓ **ANTES:** contador = 50

##### ▣ DURANTE A EXECUÇÃO:

Thread 0: contador = 50 → 60

Thread 1: contador = 50 → 75

Thread 2: contador = 50 → 55

🔗 Todas começam com 50, modificam independentemente

↑ **DEPOIS:** contador = 50 (inalterada)

## SHARED

**Comportamento:** Variável compartilhada entre threads

- Todas threads acessam mesma memória
- Requer sincronização (#pragma omp atomic, critical)
- Padrão para variáveis globais

#### DEMONSTRAÇÃO: SHARED

↓ **ANTES:** soma\_total = 0

##### ▣ ACESSOS SIMULTÂNEOS:

Thread 0: soma\_total += 10 → 10

Thread 1: soma\_total += 20 → 30

Thread 2: soma\_total += 15 → 45

⚠ Requer sincronização!

↑ **DEPOIS:** soma\_total = 45

## LASTPRIVATE

**Comportamento:** Preserva valor da última iteração

- Thread que executa última iteração define valor
- Valor é copiado de volta para variável original
- Útil para capturar resultado final de loops

🚩 DEMONSTRAÇÃO: LASTPRIVATE

⬇️ **ANTES:** ultimo\_valor = -1

▣ **LOOP for (i=0; i<100; i++):**

Thread 0: i=0..24 → ultimo\_valor=24

Thread 1: i=25..49 → ultimo\_valor=49

Thread 2: i=50..74 → ultimo\_valor=74

**Thread 3:** i=75..99 → ultimo\_valor=99 🏁

⬆️ **DEPOIS:** ultimo\_valor = 99

🏁 Thread 3 executou a última iteração

**Performance Comparada**

Versão	Tempo (s)	$\pi$ Estimado	Erro (%)	Speedup	Observações
Sequencial	0.0062	3.137000	0.146%	1.0x	Referência
Race Condition	0.0035-0.0115	0.449-0.710	77-86%	0.5-1.8x	<b>Altamente inconsistente</b>
Critical	0.0170	3.140240	0.043%	0.36x	Correto, mas <b>2.7x mais lento</b>
Reestruturado	0.0003	3.135560	0.192%	20.7x	<b>Melhor performance</b>

**Dica de Boas Práticas: default(none) no OpenMP**

### Por que usar `default(none)`?

Em programas OpenMP complexos, pode ser difícil acompanhar o escopo de cada variável (se é `shared`, `private`, etc).

Ao adicionar `default(none)` na diretiva `parallel`, você obriga o compilador a exigir que todas as variáveis usadas dentro do bloco tenham seu escopo explicitamente declarado.

- Ajuda a evitar bugs sutis de paralelismo.
- Torna o código mais legível e seguro.
- Exemplo: `#pragma omp parallel for default(none) private(i) shared(N, array)`

**Recomendação:** Sempre que possível, utilize `default(none)` em projetos reais para garantir clareza e segurança no escopo das variáveis.

## Análise dos Resultados

### Observações Críticas

1. **Condição de Corrida Severa:** A versão incorreta com ``#pragma omp parallel for`` apresenta erros extremos (77-86%), demonstrando que `race conditions` podem tornar programas paralelos completamente inúteis.
2. **Critical Section - Gargalo Moderado:** A correção com ``#pragma omp critical`` funciona, mas é 2.7x mais lenta que a versão sequencial devido à sincronização excessiva.
3. **Reestruturação Bem-Sucedida:** A abordagem com ``#pragma omp parallel`` + ``#pragma omp for`` e acumulação local consegue speedup de 20.7x, demonstrando o valor de minimizar sincronização.
4. **Cláusulas Funcionam Corretamente:** Todas as demonstrações das cláusulas (`private`, `firstprivate`, `shared`, `lastprivate`) produzem resultados precisos, confirmando o comportamento esperado de cada uma.

### Lições Aprendidas sobre Cláusulas OpenMP

- **PRIVATE:** Ideal para variáveis temporárias que não precisam de valor inicial específico
- **FIRSTPRIVATE:** Essencial quando threads precisam começar com valor conhecido
- **SHARED:** Requer cuidado extremo com sincronização (`#pragma omp atomic`, `critical`)



- **LASTPRIVATE:** Útil para capturar estados finais de loops paralelos

## Conclusões

---

1. **Condições de corrida** são um problema crítico na paralelização que pode levar a resultados completamente incorretos (erros de 77-86% observados)
2. **Critical sections** resolvem o problema mas introduzem overhead significativo (2.7x mais lento que sequencial)
3. **Reestruturação inteligente** é fundamental - separar ``#pragma omp parallel`` de ``#pragma omp for`` permitiu speedup de 20.7x
4. **Cláusulas OpenMP** oferecem controle preciso sobre o escopo de variáveis:
  - Use `private` para variáveis temporárias que não precisam de valor inicial
  - Use `firstprivate` quando threads precisam começar com valor conhecido
  - Use `lastprivate` para capturar resultado da última iteração
  - Use `shared` para dados compartilhados (requer sincronização cuidadosa)
  - Use `default(none)` para garantir controle explícito de todas as variáveis
5. **Minimizar sincronização** é chave para alta performance - acumulação local seguida de sincronização única é muito mais eficiente que múltiplas sincronizações

```

#define _USE_MATH_DEFINES
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <omp.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

// 1. Versão sequencial (referência)
double estimar_pi_sequencial(long num_pontos) {
    long pontos_dentro = 0;
    unsigned int seed = 12345; // Semente para geração de números aleatórios

    for (long i = 0; i < num_pontos; i++) {
        // Gerar coordenadas aleatórias no intervalo [-1, 1]
        double x = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0; // [-1, 1]
        double y = (double)rand_r(&seed) / RAND_MAX * 2.0 - 1.0; // [-1, 1]

        // Verificar se o ponto está dentro do círculo unitário
        // Equação do círculo:  $x^2 + y^2 \leq 1$ 
        if (x*x + y*y <= 1.0) {
            pontos_dentro++;
        }
    }

    // Estimativa de  $\pi$  usando a fórmula:  $\pi \approx 4 \times (\text{pontos\_dentro} / \text{total})$ 
    return 4.0 * pontos_dentro / num_pontos;
}

// 2. Versão INCORRETA - com condição de corrida usando #pragma omp parallel for
double estimar_pi_incorreto(long num_pontos) {
    long pontos_dentro = 0; // PROBLEMA: Variável compartilhada sem proteção!

    #pragma omp parallel for
    for (long i = 0; i < num_pontos; i++) {
        // Cada thread precisa de sua própria semente para evitar correlação
        unsigned int seed = i + omp_get_thread_num() * 12345;
        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_dentro++; // CONDIÇÃO DE CORRIDA! Múltiplas threads modificam simultaneamente
        }
    }

    return 4.0 * pontos_dentro / num_pontos;
}

// 3. Correção com #pragma omp critical
double estimar_pi_critical(long num_pontos) {
    long pontos_dentro = 0;

    #pragma omp parallel for
    for (long i = 0; i < num_pontos; i++) {
        unsigned int seed = i + omp_get_thread_num() * 12345;
        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) {
            #pragma omp critical // GARGALO: Serializa o acesso a cada incremento
            pontos_dentro++; // Agora protegido, mas com overhead de sincronização
        }
    }

    return 4.0 * pontos_dentro / num_pontos;
}

// 4. Reestruturação com #pragma omp parallel seguido de #pragma omp for
double estimar_pi_reestruturado(long num_pontos) {
    long pontos_dentro = 0;

    #pragma omp parallel // Cria região paralela SEPARADA do loop
    {
        // Variável LOCAL para cada thread - sem compartilhamento!
        long pontos_locais = 0;
        unsigned int seed = 12345 + omp_get_thread_num() * 1000;

        #pragma omp for // Distribui iterações entre threads existentes
        for (long i = 0; i < num_pontos; 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_locais++; // RÁPIDO: sem sincronização no loop!
            }
        }
    }
}

```

```

    }
}

// Sincronização acontece apenas UMA vez por thread (8 vezes total)
#pragma omp critical
pontos_dentro += pontos_locais; // Acumula resultado local no total
}

return 4.0 * pontos_dentro / num_pontos;
}

// 5. Demonstração com private
double estimar_pi_private(long num_pontos) {
    long pontos_dentro = 0;
    long pontos_locais = 999; // Valor inicial que será perdido nas threads
    int thread_id = -1;       // Também será perdido

    printf("\n=== DEMONSTRAÇÃO: PRIVATE ===\n");
    printf("ESTADO INICIAL\n");
    printf("pontos_locais = %ld (valor que será perdido nas threads)\n", pontos_locais);
    printf("thread_id = %d (valor que será perdido nas threads)\n", thread_id);
    printf("\n");

    #pragma omp parallel private(pontos_locais, thread_id)
    {
        // IMPORTANTE: Valores iniciais são INDEFINIDOS em private!
        // Precisamos inicializar manualmente dentro da região paralela
        thread_id = omp_get_thread_num();
        pontos_locais = 0; // Inicialização manual obrigatória

        unsigned int seed = 12345 + thread_id * 1000;

        #pragma omp for
        for (long i = 0; i < num_pontos; 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_locais++; // Cada thread incrementa sua própria cópia
            }
        }

        printf("| Thread %d: pontos_locais = %6ld (cópia independente) | \n", thread_id,
pontos_locais);

        #pragma omp critical
        pontos_dentro += pontos_locais;
    }

    printf("ESTADO FINAL\n");
    printf("pontos_locais = %ld (valor original inalterado)\n", pontos_locais);
    printf("thread_id = %d (valor original inalterado)\n", thread_id);
    printf("❗ Modificações das threads NÃO afetam variáveis originais\n");
    printf("\n");

    return 4.0 * pontos_dentro / num_pontos;
}

// 6. Demonstração com firstprivate
double estimar_pi_firstprivate(long num_pontos) {
    long pontos_dentro = 0;
    long contador_inicial = 1000; // Este valor será COPIADO para cada thread
    int multiplicador = 100;      // Este também será copiado

    printf("\n=== DEMONSTRAÇÃO: FIRSTPRIVATE ===\n");
    printf("VALORES ORIGINAIS\n");
    printf("contador_inicial = %ld (será copiado para cada thread)\n", contador_inicial);
    printf("multiplicador = %d (será copiado para cada thread)\n", multiplicador);
    printf("\n");

    #pragma omp parallel firstprivate(contador_inicial, multiplicador)
    {
        int thread_id = omp_get_thread_num();
        // Cada thread automaticamente RECEBE uma CÓPIA dos valores originais!
        // contador_inicial = 1000, multiplicador = 100 (em cada thread)

        long pontos_locais = 0;
        // Usar os valores iniciais para criar sementes diferentes por thread
        unsigned int seed = contador_inicial + thread_id * multiplicador;

        #pragma omp for
        for (long i = 0; i < num_pontos; 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_locais++;
            }
        }

        // Demonstrar que cada thread pode modificar suas cópias independentemente
        contador_inicial += pontos_locais; // Modificação local (não afeta original)
        multiplicador *= thread_id + 1;   // Modificação local (não afeta original)
    }
}

```

```

        printf("| Thread %d: contador=%ld, mult=%d, pontos=%-6ld\n",
               thread_id, contador_inicial, multiplicador, pontos_locais);

        #pragma omp critical
        pontos_dentro += pontos_locais;
    }

    printf(" VALORES APÓS REGIÃO PARALELA\n");
    printf(" contador_inicial = %ld (inalterado)\n", contador_inicial);
    printf(" multiplicador = %d (inalterado)\n", multiplicador);
    printf(" ☹ Threads receberam cópias, mas originais não mudaram\n");
    printf("\n");

    return 4.0 * pontos_dentro / num_pontos;
}

// 7. Demonstração com shared
double estimar_pi_shared(long num_pontos) {
    long pontos_dentro = 0;
    long contador_compartilhado = 0; // Variável compartilhada - todas threads acessam
    double progresso = 0.0; // Também compartilhada

    printf("\n=== DEMONSTRAÇÃO: SHARED ===\n");
    printf(" VARIÁVEIS COMPARTILHADAS INICIAIS\n");
    printf(" contador_compartilhado = %ld\n",
           contador_compartilhado);
    printf(" progresso = %.1f%%\n", progresso * 100);
    printf(" ☹ Todas as threads acessarão as MESMAS variáveis\n");
    printf("\n");

    #pragma omp parallel shared(pontos_dentro, contador_compartilhado, progresso, num_pontos)
    {
        int thread_id = omp_get_thread_num();
        long pontos_locais = 0; // Esta é automática private (declarada dentro)
        unsigned int seed = 12345 + thread_id * 1000;

        #pragma omp for
        for (long i = 0; i < num_pontos; 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_locais++;
            }

            // Demonstrar acesso sincronizado a variáveis compartilhadas
            #pragma omp atomic // Protege o incremento (mais eficiente que critical para operações
            // simples)
            contador_compartilhado++;

            // Atualizar progresso ocasionalmente (shared variable)
            if (i % 10000 == 0) {
                #pragma omp atomic write // Protege a escrita
                progresso = (double)contador_compartilhado / num_pontos;
            }
        }

        // Acumular resultado local na variável compartilhada
        #pragma omp critical
        pontos_dentro += pontos_locais;

        printf("| Thread %d > contribuiu com %-6ld pontos\n", thread_id,
               pontos_locais);
    }

    printf(" ESTADO FINAL DAS VARIÁVEIS COMPARTILHADAS\n");
    printf(" contador_compartilhado = %-8ld (modificado por todas)\n",
           contador_compartilhado);
    printf(" progresso = %.1f%% (atualizado colaborativamente)\n", progresso * 100);
    printf(" ⚠ Sincronização foi necessária para evitar conflitos\n");
    printf("\n");

    return 4.0 * pontos_dentro / num_pontos;
}

// 8. Demonstração com lastprivate
double estimar_pi_lastprivate(long num_pontos) {
    long pontos_dentro = 0;
    int ultimo_indice = -1; // Será sobrescrito com índice da última iteração
    int thread_da_ultima_iteracao = -1; // Será sobrescrito com ID da thread que executou por último

    printf("\n=== DEMONSTRAÇÃO: LASTPRIVATE ===\n");
    printf(" ESTADO INICIAL DAS VARIÁVEIS\n");
    printf(" ultimo_indice = %d\n", ultimo_indice);
    printf(" thread_da_ultima_iteracao = %d\n",
           thread_da_ultima_iteracao);
    printf(" ☹ Cada thread terá suas próprias cópias\n");
    printf(" ☹ Valores finais serão da thread que executar por último\n");
    printf("\n");

    #pragma omp parallel
    {

```

```

long pontos_locais = 0;
int thread_id = omp_get_thread_num();
unsigned int seed = 12345 + thread_id * 1000;

// lastprivate: cada thread tem cópia própria, mas valor da última iteração é preservado
#pragma omp for lastprivate(ultimo_indice, thread_da_ultima_iteracao)
for (long i = 0; i < num_pontos; 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_locais++;
    }

    // Estas atribuições acontecem em CADA iteração, CADA thread
    // Mas apenas os valores da ÚLTIMA iteração (maior i) serão preservados
    ultimo_indice = i; // Na última iteração: será o maior índice
    thread_da_ultima_iteracao = thread_id; // Na última iteração: será a thread que executou
}

#pragma omp critical
pontos_dentro += pontos_locais;
}

// Após a região paralela: variáveis contêm valores da thread que executou a última iteração
printf("└ ESTADO FINAL DAS VARIÁVEIS ────────────────────────────────────\n");
printf("└ ultimo_indice = %-8d (índice da última iteração) ────────────\n", ultimo_indice);
printf("└ thread_da_ultima_iteracao = %d ──────────────────────────────────\n", thread_da_ultima_iteracao);
printf("└ ✎ Thread %d executou a iteração final (índice %d) ────────────\n", thread_da_ultima_iteracao, ultimo_indice);
printf("└ ✎✎✎ Apenas valores da última iteração foram preservados ────────────\n");
printf("└───────────────────────────────────────────────────────────────────\n");

return 4.0 * pontos_dentro / num_pontos;
}

// Função auxiliar para testar e medir tempo
void testar_implementacao(const char* nome, double (*funcao)(long), long num_pontos) {
    printf("\n===== \n");
    printf("TESTANDO: %s\n", nome);
    printf("===== \n");

    // Medição precisa de tempo usando OpenMP
    double tempo_inicio = omp_get_wtime(); // Timestamp de início
    double pi_estimado = funcao(num_pontos); // Execução da função
    double tempo_fim = omp_get_wtime(); // Timestamp de fim

    // Cálculo de métricas de qualidade
    double erro = fabs(pi_estimado - M_PI); // Erro absoluto
    double erro_percentual = (erro / M_PI) * 100.0; // Erro percentual

    printf("RESULTADOS:\n");
    printf("π estimado: %.6f\n", pi_estimado);
    printf("π real: %.6f\n", M_PI);
    printf("Erro: %.6f (%.3f%%)\n", erro, erro_percentual);
    printf("Tempo: %.4f segundos\n", tempo_fim - tempo_inicio);
}

int main() {
    long num_pontos = 100000; // 100 mil pontos para demonstrações rápidas mas significativas

    printf("=== ESTIMATIVA DE π USANDO MÉTODO DE MONTE CARLO ===\n");
    printf("Número de pontos: %ld\n", num_pontos);
    printf("Threads disponíveis: %d\n", omp_get_max_threads());
    printf("Valor real de π: %.10f\n", M_PI);

    // 1. Versão sequencial (referência)
    testar_implementacao("VERSÃO SEQUENCIAL", estimar_pi_sequencial, num_pontos);

    // 2. Demonstrar o PROBLEMA - múltiplas execuções mostram inconsistência
    printf("\n*** PROBLEMA: CONDIÇÃO DE CORRIDA COM #pragma omp parallel for ***\n");
    for (int i = 0; i < 3; i++) {
        printf("\n--- Execução %d ---\n", i + 1);
        testar_implementacao("PARALLEL FOR INCORRETO", estimar_pi_incorreto, num_pontos);
    }

    // 3. Primeira correção (funcional mas ineficiente)
    testar_implementacao("CORREÇÃO COM CRITICAL", estimar_pi_critical, num_pontos);

    // 4. Solução otimizada (melhor prática)
    testar_implementacao("REESTRUTURADO (parallel + for)", estimar_pi_reestruturado, num_pontos);

    // 5. Demonstrações das cláusulas
    printf("\n\n*** DEMONSTRAÇÕES DAS CLÁUSULAS OpenMP ***\n");

    // Usar dataset menor para demonstrações mais claras e rápidas
    testar_implementacao("CLÁUSULA PRIVATE", estimar_pi_private, num_pontos / 10);

    testar_implementacao("CLÁUSULA FIRSTPRIVATE", estimar_pi_firstprivate, num_pontos / 10);

    testar_implementacao("CLÁUSULA SHARED", estimar_pi_shared, num_pontos / 10);

    testar_implementacao("CLÁUSULA LASTPRIVATE", estimar_pi_lastprivate, num_pontos / 10);
}

```

```
    testar_implementation( CLASULA_LASTPRIVATE , estimar_pi_lastprivate, num_pontos / 10 );  
  
    return 0;  
}
```