Tarefa 6 - Estimativa Estocástica de π com OpenMP

Método de Monte Carlo

O método de Monte Carlo estima π 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: *π ≈ 4 × (pontos_dentro_círculo / 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 π 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 π
• Com milhões de pontos, centenas de milhares podem ser perdidos
• Resultado: π calculado será sempre menor que o valor real

Resultados Inconsistentes Observados:

Execução	π Estimado	Erro	Tempo (s)	Explicação
1	0.913476	70.923%	2.8614	Com 250M pontos: perdeu ~70% dos incrementos
2	0.865560	72.448%	2.7982	
3	0.911949	70.972%	2.8158	

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:** 4 threads executando como se fosse 1 thread
• **Overhead de sincronização:** Cada critical section tem custo computacional

📊 **Análise de Performance (250M pontos, ~196M hits):**
• **Tempo total:** 21.28s (vs 3.62s sequencial)
• **Speedup:** 0.17x (na verdade ficou **6x mais lento!**)
• **Critical sections executadas:** ~196.000.000 (uma para cada ponto dentro do círculo)
• **Tempo perdido:** Threads ficam 80% 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 (250M pontos, ~196M hits)
`#pragma omp critical` dentro do loop	A cada incremento	**~196.000.000 sincronizações!** Cada hit = 1 critical section
`#pragma omp parallel` + acumulação local	Uma por thread	**4 sincronizações** Uma por thread no final

🔧 **Funcionamento Passo a Passo:**

1. **Criação das Threads:** `#pragma omp parallel` cria 4 threads
2. **Variáveis Locais:** Cada thread tem sua própria `pontos_locais = 0`
3. **Divisão do Trabalho:** `#pragma omp for` divide 250M iterações entre 4 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:** 1.28s vs 21.28s do critical (16.7x mais rápido!)
- **Speedup:** 2.8x comparado ao sequencial (com 4 threads)
- **Precisão:** Mantém a correção completa dos resultados
- **Escalabilidade:** Performance melhora com mais threads disponíveis

Performance Comparada

Versão	Tempo (s)	π Estimado	Erro (%)	Speedup	Observações
Sequencial	3.6101	3.141578	0.000%	1.0x	Referência (250M pontos)
Race Condition	2.45-2.48	0.843-0.880	71-73%	1.5x	**Catastrófico - perdeu ~73% dos incrementos**
Critical	21.3381	3.141593	0.000%	0.17x	Correto, mas **5.9x mais lento**
Reestruturado	0.9039	3.141587	0.000%	4.0x	**Melhor performance com 4 threads**

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 = ???
Thread 3: 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
```

```
Thread 3: contador = 50 → 65
💡 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
Thread 3: soma_total += 25 → 70
⚠ Requer sincronização!

⬆ DEPOIS: soma_total = 70
```

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 das Cláusulas OpenMP (250 Milhões de Pontos)

Cláusula	Tempo (s)	π Estimado	Erro (%)	Speedup vs Sequencial	Características Principais
PRIVATE	0.9532	3.141587	0.000%	**3.8x**	Quase igual ao reestruturado - Inicialização manual
LASTPRIVATE	1.0889	3.141587	0.000%	**3.3x**	Performance excelente - Preserva última iteração
FIRSTPRIVATE	1.0608	3.141548	0.001%	**3.4x**	Overhead moderado - Copia automaticamente
SHARED	7.3141	3.141587	0.000%	**0.49x**	**Mais lenta que sequencial!** - Atomic mata performance

> 🧿 **Por que apenas SHARED é lenta?**
>
> 📊 **Comparação com a Versão Reestruturada (0.9039s):**

• **PRIVATE (0.9532s):** Quase idêntica - diferença de apenas 49ms
• **FIRSTPRIVATE (1.0608s):** Overhead mínimo - diferença de 157ms
• **LASTPRIVATE (1.0889s):** Overhead pequeno - diferença de 185ms
• **SHARED (7.3141s):** ⚠ **CATASTRÓFICA** - 7.7x mais lenta!

🔍 **Explicação Técnica:**

✔ **PRIVATE, FIRSTPRIVATE e LASTPRIVATE:**
• Fazem **acumulação local** (igual ao reestruturado)
• Cada thread conta seus pontos em variável **privada**
• Sincronização acontece apenas **1x por thread** (4 vezes total)
• Resultado: **Performance quase ótima**

✖ **SHARED com Atomic:**
• Usa `contador_compartilhado` com `#pragma omp atomic`
• **Sincronização a CADA iteração** (250 milhões de vezes!)
• Threads competem pelo mesmo contador a cada ponto processado
• Resultado: **Overhead de sincronização mata a performance**

💡 **Conclusão:** SHARED é lenta porque força sincronização massiva, enquanto as outras cláusulas mantêm o padrão eficiente de acumulação local do reestruturado.

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 (250 Milhões de Pontos - Execução Atualizada)

1. **Condição de Corrida Catastrófica:** A versão incorreta com `#pragma omp parallel for` apresenta erros extremos (71-73%), perdendo ~73% dos incrementos e tornando o programa completamente inútil.
2. **Critical Section - Gargalo Extremo:** A correção com `#pragma omp critical` funciona, mas é 5.9x mais lenta que a versão sequencial (21.3s vs 3.6s) devido à sincronização excessiva.
3. **Reestruturação Bem-Sucedida:** A abordagem com `#pragma omp parallel` + `#pragma omp for` e acumulação local consegue speedup excepcional de **4.0x** (0.9039s vs 3.6101s), demonstrando paralelização eficiente.
4. **Cláusulas com Dataset Completo:** Todas as demonstrações das cláusulas usam o dataset completo (250M pontos), confirmando comportamento correto e revelando diferenças sutis de performance entre elas.
5. **Performance Ranking:** PRIVATE (0.95s) ≈ REESTRUTURADO (0.90s) > FIRSTPRIVATE (1.06s) > LASTPRIVATE (1.09s) >> SHARED (7.31s)

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 70-72% observados com dataset grande)
2. **Critical sections** resolvem o problema mas introduzem overhead extremo (6x mais lento que sequencial com 196M sincronizações)
3. **Reestruturação inteligente** é fundamental - separar `#pragma omp parallel` de `#pragma omp for` permitiu speedup de 2.8x com 4 threads
4. **Escalabilidade confirmada** - Com datasets grandes, a diferença entre abordagens corretas e incorretas fica ainda mais evidente

5. **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

```c
#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: x² + y² ≤ 1
        if (x*x + y*y <= 1.0) {
            pontos_dentro++;
        }
    }

    // Estimativa de π usando a fórmula: π ≈ 4 × (pontos_dentro / 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 (4 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=== CLÁUSULA: PRIVATE ===\n");
    printf("Antes: pontos_locais=%ld, thread_id=%d (serão perdidos)\n", pontos_locais, thread_id);

    #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
            }
        }

        #pragma omp critical
        {
            printf("Thread %d: %ld pontos\n", thread_id, pontos_locais);
            pontos_dentro += pontos_locais;
        }
    }

    printf("Depois: pontos_locais=%ld, thread_id=%d (valores originais inalterados)\n", pontos_locais,
thread_id);

    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=== CLÁUSULA: FIRSTPRIVATE ===\n");
    printf("Antes: contador_inicial=%ld, multiplicador=%d (serão copiados)\n", contador_inicial,
multiplicador);

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

        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)

        #pragma omp critical
        {
            printf("Thread %d: contador=%ld, mult=%d, pontos=%ld\n",
                   thread_id, contador_inicial, multiplicador, pontos_locais);
            pontos_dentro += pontos_locais;
        }
    }
```

```
    printf("Depois: contador_inicial=%ld, multiplicador=%d (valores originais preservados)\n",
contador_inicial, multiplicador);

    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=== CLÁUSULA: SHARED ===\n");
    printf("Variáveis compartilhadas: contador=%ld, progresso=%.1f%%\n", contador_compartilhado,
progresso * 100);

    #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
            contador_compartilhado++;

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

        #pragma omp critical
        {
            printf("Thread %d: %ld pontos\n", thread_id, pontos_locais);
            pontos_dentro += pontos_locais;
        }
    }

    printf("Final: contador=%ld, progresso=%.1f%% (modificadas por todas threads)\n",
           contador_compartilhado, progresso * 100);

    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=== CLÁUSULA: LASTPRIVATE ===\n");
    printf("Antes: ultimo_indice=%d, thread_da_ultima_iteracao=%d\n", ultimo_indice,
thread_da_ultima_iteracao);

    #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
            // Mas apenas os valores da ÚLTIMA iteração serão preservados
            ultimo_indice = i;
            thread_da_ultima_iteracao = thread_id;
        }

        #pragma omp critical
        pontos_dentro += pontos_locais;
    }

    printf("Depois: ultimo_indice=%d, thread_da_ultima_iteracao=%d (valores da última iteração)\n",
           ultimo_indice, thread_da_ultima_iteracao);
```

```
    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() {
    // Configurar o número de threads para 4
    omp_set_num_threads(4);

    long num_pontos = 250000000; // 250 milhões de pontos para demonstração com formatação limpa

    printf("=== ESTIMATIVA DE π USANDO MÉTODO DE MONTE CARLO ===\n");
    printf("Número de pontos: %ld\n", num_pontos);
    printf("Número de threads configuradas: %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 completo para demonstrações com 250 milhões de pontos
    testar_implementacao("CLÁUSULA PRIVATE", estimar_pi_private, num_pontos);

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

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

    testar_implementacao("CLÁUSULA LASTPRIVATE", estimar_pi_lastprivate, num_pontos);

    return 0;
}
```