Relatório: Escopo de variáveis e regiões críticas (Estimativa Estocástica de π com OpenMP)

Aluno: Cristovão Lacerda Cronje

1. Introdução

Este relatório avalia diferentes implementações paralelizadas da estimativa estocástica de π usando OpenMP. O método de Monte Carlo foi aplicado com 100 milhões de pontos, e foram testadas cinco versões do código, cada uma explorando diferentes cláusulas do OpenMP para paralelização.

Configuração do sistema:

- **Processador:** Intel i5-3210M (2 núcleos físicos, 4 threads lógicas)
- **Compilação:** gcc -fopenmp 006_tarefa.c -o 006_tarefa -lm
- **Sistema Operacional:** Linux(Mint)

O objetivo foi analisar:

1. O impacto da condição de corrida em versões paralelas ingênuas.
2. A correção usando private, critical, firstprivate, lastprivate e default(none).
3. A precisão e eficiência de cada abordagem.

2. Metodologia

Foram implementadas 5 versões do algoritmo:

Versão	Cláusulas OpenMP	Descrição
1	#pragma omp parallel for	Paralelização ingênua com condição de corrida
2	private, critical	Correção do problema de concorrência
3	firstprivate	Geração de números aleatórios independentes por thread
4	lastprivate	Demonstração do uso de variáveis após o loop paralelo
5	default(none)	Boa prática para evitar compartilhamento acidental

- #pragma omp parallel for → Paraleliza um loop, dividindo iterações entre threads, mas sem proteção contra condições de corrida.
- private → Cada thread recebe sua própria cópia *não inicializada* de uma variável.
- critical → Garante que apenas uma thread execute uma seção de código por vez (evita concorrência).
- firstprivate → Cada thread recebe uma cópia *inicializada* com o valor da variável antes do paralelismo.
- lastprivate → Preserva o valor da última iteração do loop paralelo após o fim da região paralela.

- default(none) → Exige declaração explícita do escopo de todas as variáveis, evitando compartilhamento acidental.

Métricas avaliadas:

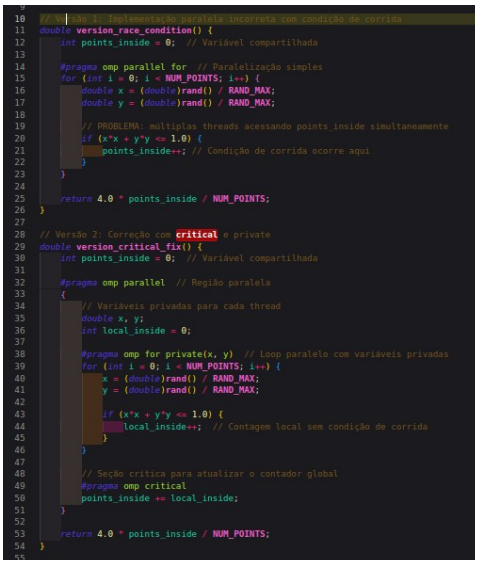
•**Estimativa de π** (comparada com PI_REF = 3.14159265358979323846).

•**Erro absoluto** (|π_estimado – π_referência|): Diferença direta entre o valor estimado e o valor real de π (quanto menor, mais preciso)

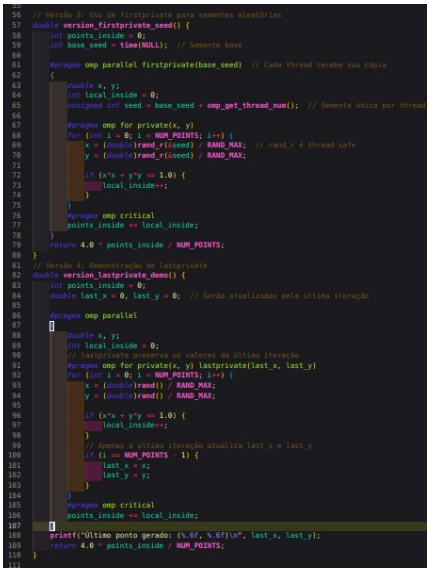
•**Dígitos corretos** (-log10(erro)): Número de casas decimais que batem com π real (ex: erro 1e-4 = 4 dígitos corretos).

- Exemplo: Se π ≈ 3.1416 e o erro é 0.0001, temos 4 dígitos corretos, pois 3.141 está exato.)

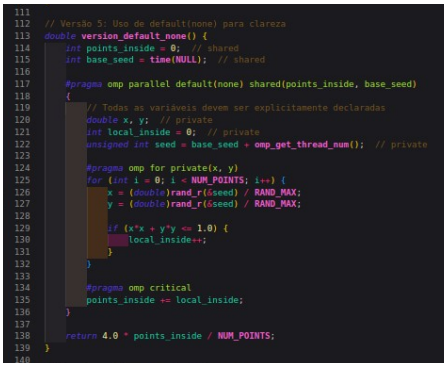
•**Código:**



```c
// Versão 1: Implementação paralela incorreta com condição de corrida
double version_race_condition() {
    int points_inside = 0;  // Variável compartilhada

    #pragma omp parallel for  // Paralelização simples
    for (int i = 0; i < NUM_POINTS; i++) {
        double x = (double)rand() / RAND_MAX;
        double y = (double)rand() / RAND_MAX;

        // PROBLEMA: múltiplas threads acessando points_inside simultaneamente
        if (x*x + y*y <= 1.0) {
            points_inside++; // Condição de corrida ocorre aqui
        }
    }

    return 4.0 * points_inside / NUM_POINTS;
}

// Versão 2: Correção com critical e private
double version_critical_fix() {
    int points_inside = 0;  // Variável compartilhada

    #pragma omp parallel  // Região paralela
    {
        // Variáveis privadas para cada thread
        double x, y;
        int local_inside = 0;

        #pragma omp for private(x, y)  // Loop paralelo com variáveis privadas
        for (int i = 0; i < NUM_POINTS; i++) {
            x = (double)rand() / RAND_MAX;
            y = (double)rand() / RAND_MAX;

            if (x*x + y*y <= 1.0) {
                local_inside++;  // Contagem local sem condição de corrida
            }
        }

        // Seção crítica para atualizar o contador global
        #pragma omp critical
        points_inside += local_inside;
    }

    return 4.0 * points_inside / NUM_POINTS;
}
```



```c
// Versão 3: Uso de firstprivate para sementes aleatórias
double version_firstprivate_seed() {
    int points_inside = 0;
    int base_seed = time(NULL);  // Semente base

    #pragma omp parallel firstprivate(base_seed)  // Cada thread recebe sua cópia
    {
        double x, y;
        int local_inside = 0;
        unsigned int seed = base_seed + omp_get_thread_num();  // Semente única por thread

        #pragma omp for private(x, y)
        for (int i = 0; i < NUM_POINTS; i++) {
            x = (double)rand_r(&seed) / RAND_MAX;  // rand_r é thread-safe
            y = (double)rand_r(&seed) / RAND_MAX;

            if (x*x + y*y <= 1.0) {
                local_inside++;
            }
        }
        #pragma omp critical
        points_inside += local_inside;
    }
    return 4.0 * points_inside / NUM_POINTS;
}
// Versão 4: Demonstração de lastprivate
double version_lastprivate_demo() {
    int points_inside = 0;
    double last_x = 0, last_y = 0;  // Serão atualizadas pela última iteração

    #pragma omp parallel
    {
        double x, y;
        int local_inside = 0;
        // lastprivate transfere os valores da última iteração
        #pragma omp for private(x, y) lastprivate(last_x, last_y)
        for (int i = 0; i < NUM_POINTS; i++) {
            x = (double)rand() / RAND_MAX;
            y = (double)rand() / RAND_MAX;

            if (x*x + y*y <= 1.0) {
                local_inside++;
            }
            // Apenas a última iteração atualiza last_x e last_y
            if (i == NUM_POINTS - 1) {
                last_x = x;
                last_y = y;
            }
        }
        #pragma omp critical
        points_inside += local_inside;
    }
    printf("Último ponto gerado: (%.6f, %.6f)\n", last_x, last_y);
    return 4.0 * points_inside / NUM_POINTS;
}
```



```c
// Versão 5: Uso de default(none) para clareza
double version_default_none() {
    int points_inside = 0;  // shared
    int base_seed = time(NULL);  // shared

    #pragma omp parallel default(none) shared(points_inside, base_seed)
    {
        // Todas as variáveis devem ser explicitamente declaradas
        double x, y;  // private
        int local_inside = 0;  // private
        unsigned int seed = base_seed + omp_get_thread_num();  // private

        #pragma omp for private(x, y)
        for (int i = 0; i < NUM_POINTS; i++) {
            x = (double)rand_r(&seed) / RAND_MAX;
            y = (double)rand_r(&seed) / RAND_MAX;

            if (x*x + y*y <= 1.0) {
                local_inside++;
            }
        }

        #pragma omp critical
        points_inside += local_inside;
    }

    return 4.0 * points_inside / NUM_POINTS;
}
```

3.**Análise dos Resultados**

Versão 1: Paralelização ingênua (#pragma omp parallel for)

•**Resultado:** π ≈ 3.11477836 (Erro: **2.68e-02**)

•**Problema:** Condição de corrida em points_inside++.

•**Efeito:** Subestimação de π devido a contagens perdidas.

Versão 2: Correção com private e critical

•**Resultado:** π ≈ 3.14159744 (Erro: **4.79e-06**)

•**Melhoria:** Eliminação da condição de corrida.

•**Mecanismo:**

•private(x, y) garante que cada thread tenha suas próprias variáveis.

•critical protege a atualização de points_inside.

Versão 3: Uso de firstprivate para sementes aleatórias

- **Resultado:** π ≈ 3.14161872 (Erro: **2.61e-05**)
- **Vantagem:** Evita repetição de números aleatórios entre threads.
- **Mecanismo:**
  - firstprivate(base_seed) garante uma semente única por thread.
  - rand_r(&seed) é **thread-safe**.

Versão 4: Demonstração de lastprivate

- **Resultado:** π ≈ 3.14149312 (Erro: **9.95e-05**)
- **Funcionalidade:** Preserva o último valor das variáveis após o loop.
- **Saída adicional:** Último ponto gerado: (0.207491, 0.929194)

Versão 5: Uso de default(none)

- **Resultado:** π ≈ 3.14146112 (Erro: **1.32e-04**)
- **Vantagem:** Força declaração explícita de escopo, evitando erros.
- **Aplicação:** #pragma omp parallel default(none) shared(points_inside, base_seed)

4. **Quadro Comparativo**

Versão	Estimativa de π	Erro	Dígitos Corretos	Problema/Correção
1	3.11477836	2.68e-02	1	Condição de corrida
2	3.14159744	4.79e-06	5	Correção com critical
3	3.14161872	2.61e-05	4	firstprivate para sementes
4	3.14149312	9.95e-05	4	lastprivate demo
5	3.14146112	1.32e-04	3	default(none)

5. Discussão:

Versão 1 (Condição de Corrida)

• Erro elevado (π ≈ 3.114 vs 3.141): Múltiplas threads acessaram simultaneamente points_inside sem sincronização(*concorrência descontrolada)*, causando perda de atualizações e subestimando a contagem de pontos dentro do círculo.

Versões 2-5 (Correções e Limitações)

- Versão 2 (private + critical):
  Resolveu a condição de corrida, mas pequenos erros persistem devido à natureza probabilística do método de Monte Carlo (variância inerente, mesmo com 100M de pontos).

- Versão 3 (firstprivate):
  Não melhorou significativamente a precisão, mas garantiu independência estatística ao fornecer sementes aleatórias únicas por thread

Possíveis Razões para Variações

- Baixa amostragem relativa: Mesmo com 100M de pontos, a estimativa de Monte Carlo ainda apresenta flutuações estatísticas.
- Overhead do critical: A sincronização introduz um pequeno atraso, porém necessário para evitar inconsistências.

6. **Conclusões**

•**Condição de corrida** A versão ingênua (sem sincronização) apresentou erro significativo ($\pi \approx 3.114$ vs 3.141) porque múltiplas threads tentavam atualizar simultaneamente a mesma variável compartilhada (points_inside), resultando em perda de contagens e estimativa incorreta.

•**private + critical:** O uso de variáveis privadas (private) com seção crítica (critical) eliminou a condição de corrida, garantindo precisão. Porém, a sincronização impõe um pequeno overhead de desempenho, já que threads precisam "esperar sua vez" (Apesar do loop ser paralelo, o critical força que as threads atualizem points_inside uma de cada vez) para atualizar o contador global.

•**Firstprivate:** Essa cláusula foi crucial para gerar números aleatórios confiáveis, pois cada thread recebeu sua própria cópia inicializada da semente (base_seed). Embora não aumente diretamente a precisão, evita repetição de valores entre threads, garantindo independência estatística.

•**Lastprivate:** Útil em cenários específicos onde o valor da última iteração é necessário após o loop paralelo (ex: coordenadas do último ponto gerado). Não impacta na precisão de π, mas demonstra como transferir informações do contexto paralelo para o serial.

•**default(none):** Forçar a declaração explícita de escopo (default(none)) previne bugs sutis de compartilhamento acidental de variáveis, especialmente em programas complexos. Isso não melhora desempenho ou precisão, mas aumenta significativamente a segurança e legibilidade do código paralelo.

7. **Reflexão Final**

•**OpenMP** é eficaz para paralelização, mas requer cuidado com **compartilhamento de memória**.

•**Boas práticas** (private, critical, default(none)) evitam erros sutis.

•**Monte Carlo** tem limitações estatísticas, mas paralelização melhora desempenho.