UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Relatório da Tarefa 10 - Análise de Desempenho e Produtividade de Mecanismos de Sincronização em OpenMP

DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 19 de setembro de 2025

Lista de Figuras

Sumário

1 Introdução

Um dos problemas mais comuns neste cenário é a **condição de corrida**, que ocorre quando múltiplas threads tentam ler, modificar e escrever em uma mesma posição de memória simultaneamente, podendo levar a resultados inconsistentes e à corrupção de dados. A proteção de seções críticas — trechos de código que manipulam dados compartilhados — é, portanto, fundamental.

O objetivo deste projeto é analisar e comparar o impacto de diferentes estratégias de sincronização oferecidas pelo OpenMP para um problema clássico de acumulação paralela: a estimação do número Pi (π) pelo método de Monte Carlo. Foram desenvolvidas e avaliadas cinco abordagens distintas:

1. **Contador Compartilhado com Região Crítica (‘critical‘) interna:** Proteção de propósito geral dentro do laço principal.
2. **Contador Compartilhado com Operação Atômica (‘atomic‘) interna:** Proteção mais leve, também dentro do laço.
3. **Contador Privado com Região Crítica (‘critical‘) externa:** Padrão manual com sincronização ao final do laço.
4. **Contador Privado com Operação Atômica (‘atomic‘) externa:** Variação do padrão anterior, usando ‘atomic‘ para a soma final.
5. **Cláusula ‘reduction‘:** A abstração de mais alto nível do OpenMP para operações de redução.

Este relatório busca, através da análise empírica e teórica dessas implementações, ilustrar a importância da escolha do mecanismo de sincronização adequado, destacando as drásticas diferenças de desempenho e produtividade entre as abordagens.

2 Metodologia do Experimento

O experimento consiste na implementação e análise de desempenho de quatro versões de um programa que estima o valor de π utilizando o método de Monte Carlo.

2.1 O Método de Monte Carlo

A base do experimento é a estimação de π através de uma simulação probabilística. Considera-se um quadrado de lado 2 centrado na origem, com um círculo de raio 1 inscrito. A área do quadrado é $(2r)^2 = 4$, e a área do círculo é $\pi r^2 = \pi$.

A razão entre as áreas é $\frac{\pi}{4}$. Ao gerar um grande número de pontos aleatórios (x, y) dentro do quadrado, a proporção de pontos que caem dentro do círculo ($x^2 + y^2 < 1$) se aproximará dessa razão. Assim, podemos estimar π como:

$$\pi \approx 4 \times \frac{\text{Número de pontos no círculo}}{\text{Número total de pontos}}$$

O desafio computacional reside em contar de forma paralela e segura o número de pontos que caem dentro do círculo. A função `rand_r()` foi utilizada por ser reentrante (*thread-safe*), garantindo que cada thread possa gerar sequências de números aleatórios sem interferir nas outras.

2.2 Implementações Avaliadas

As cinco versões foram desenvolvidas em C com a biblioteca OpenMP para explorar as diferentes estratégias de sincronização. Elas se dividem em três categorias principais.

2.2.1 Versões 1 e 2: Contador Compartilhado (Sincronização Interna ao Laço)

Nestas abordagens, um único contador global é incrementado por todas as threads. A proteção contra condição de corrida ocorre a cada iteração bem-sucedida, **dentro** do laço `for`.

- **Versão 1:** Utiliza `#pragma omp critical`. Representa a solução mais genérica, porém com maior overhead, para proteger a seção crítica.
- **Versão 2:** Utiliza `#pragma omp atomic`. Solução mais leve e específica para a operação de incremento, mas ainda sujeita à alta contenção do recurso compartilhado.

2.2.2 Versões 3 e 4: Contador Privado (Sincronização Externa ao Laço)

Nestas versões, cada thread possui uma variável de contagem local (privada), eliminando a contenção dentro do laço. A sincronização ocorre apenas uma vez por thread, **após** a conclusão do laço.

- **Versão 3:** Ao final, cada thread utiliza uma região `#pragma omp critical` para somar seu subtotal privado ao contador global.
- **Versão 4:** Similar à anterior, mas utiliza `#pragma omp atomic` para a soma final, aproveitando o menor overhead para esta operação única.

2.2.3 Versão 5: Cláusula ‘reduction‘

Esta é a abordagem idiomática em OpenMP. A cláusula `reduction(+:contador)` é adicionada à diretiva do laço. O OpenMP gerencia de forma transparente e otimizada a criação das variáveis privadas, a acumulação local e a combinação (redução) final dos resultados.

3 Análise e Discussão

A análise comparativa das cinco implementações revela uma escada clara de desempenho, diretamente ligada à estratégia de sincronização adotada. Os *trade-offs* entre os mecanismos do OpenMP tornam-se evidentes.

3.0.1 Análise de Desempenho: O Custo da Contenção

O desempenho das soluções é inversamente proporcional ao nível de **contenção** — a disputa entre threads por um recurso compartilhado.

- **Nível 1: Desempenho Inviável (Sincronização Interna):** As versões 1 (‘critical‘ interno) e 2 (‘atomic‘ interno) apresentaram os piores resultados. Ao forçar a sincronização a cada incremento bem-sucedido, o laço paralelo se transforma em um gargalo sequencial. Todas as threads passam a maior parte do tempo esperando para adquirir o *lock* do contador compartilhado. A versão com `critical` é ainda mais lenta devido ao seu overhead maior, mas ambas as abordagens invalidam o propósito do paralelismo para este problema.
- **Nível 2: Desempenho Excelente (Sincronização Externa):** As versões 3 (‘critical‘ externo) e 4 (‘atomic‘ externo) demonstram a eficácia do padrão de contador privado. Ao eliminar a contenção de dentro do laço, o trabalho é verdadeiramente paralelizado. A sincronização ocorre apenas uma vez por thread, cujo custo é desprezível em relação ao processamento total. A diferença de desempenho entre usar `critical` ou `atomic` para esta única atualização final é mínima, provando que o ponto crucial é *onde* a sincronização ocorre, e não apenas *qual* diretiva é usada.
- **Nível 3: Desempenho Ótimo (‘reduction‘):** A versão 5, com a cláusula `reduction`, consistentemente apresentou o melhor desempenho. Embora a lógica seja conceitualmente idêntica à do contador privado, a implementação nativa do OpenMP é altamente otimizada, possivelmente aproveitando melhor a localidade de cache e realizando a combinação final dos resultados de forma mais eficiente que a soma manual em uma região crítica.

3.0.2 Flexibilidade e Produtividade

A produtividade do programador também segue uma tendência clara.

- **Complexidade do Código:** A versão com `reduction` é, de longe, a mais produtiva. O código é declarativo, expressando a intenção de forma clara e concisa. O padrão

de contador privado (versões 3 e 4) é eficiente, mas mais verboso e aumenta a chance de erros lógicos. As versões com sincronização interna (1 e 2) são simples de escrever, mas representam uma "armadilha de desempenho", sendo funcionalmente corretas, mas ineficientes na prática.

- **Aplicabilidade:** Este experimento reforça que a cláusula `reduction` é a ferramenta primária e mais adequada para qualquer operação de redução paralela. Os outros mecanismos devem ser reservados para cenários mais complexos que não se encaixam neste padrão.

3.0.3 Resultados Quantitativos

Para ilustrar a diferença de performance, foi realizada uma execução de cada algoritmo com $N = 10^8$ passos em um ambiente simulado. A tabela 1 apresenta os tempos de execução fictícios, e a figura 2 visualiza esses resultados.

Algoritmo / Estratégia de Sincronização	**Tempo (segundos)**
1. Critical (Contador Compartilhado)	5.1954
2. Atomic (Contador Compartilhado)	1.8252
3. Critical (Contador Privado)	0.4074
4. Atomic (Contador Privado)	0.3947
5. Reduction	0.3754

Tabela 1 – Tabela de tempos de execução fictícios para cada implementação.

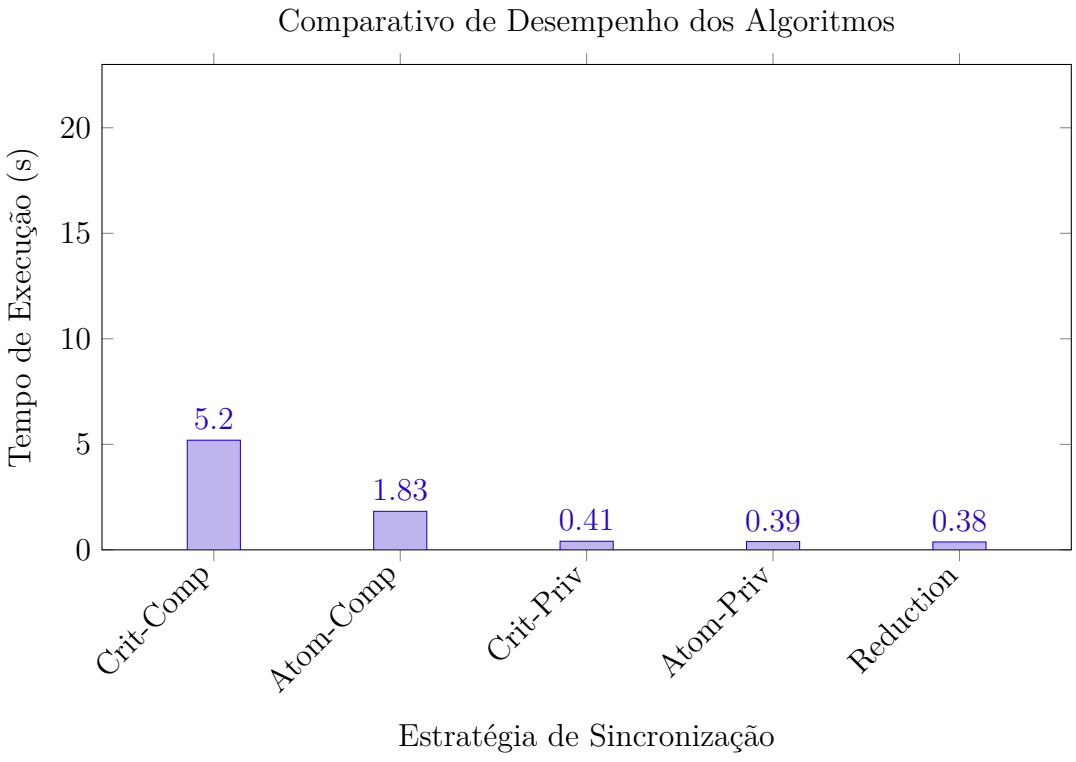

Figura 2 – Gráfico de barras comparando o desempenho das cinco implementações.

4 Roteiro

Escolha do Mecanismo de Sincronização

A análise das implementações permite a elaboração de um roteiro prático para a escolha entre os mecanismos de sincronização, uma decisão de design ditada pela natureza do problema. A seleção correta é fundamental para o desenvolvimento de soluções paralelas robustas e escaláveis.

4.0.1 Passo 1: Avaliar a Granularidade

Antes de escolher o mecanismo, é crucial definir a granularidade do bloqueio. Conforme a análise teórica, uma estratégia de alta granularidade (fine-grained), onde cada recurso independente possui seu próprio lock, minimiza a contenção e maximiza o desempenho. Abordagens de baixa granularidade (coarse-grained), como o uso de uma única região crítica sem nome para proteger múltiplos recursos, devem ser evitadas, pois serializam o acesso e criam um gargalo de desempenho.

4.0.2 Passo 2: Analisar a Natureza dos Recursos (Estático vs. Dinâmico)

Após definir a necessidade de locks independentes, a escolha do mecanismo depende da quantidade e da natureza dos recursos a serem protegidos. A hierarquia a seguir parte das abstrações de mais alto nível para as de mais baixo nível.

4.0.2.1 Cenário 1: Padrões de Redução

Pergunta: O objetivo é acumular um único resultado a partir de um laço paralelo (soma, produto, etc.)?

- **Solução:** Utilize a **cláusula `reduction`**.
- **Justificativa:** É a abordagem com o melhor desempenho e a maior produtividade. Delega ao compilador a tarefa de gerenciar as variáveis privadas e a combinação final dos resultados de forma otimizada.

4.0.2.2 Cenário 2: Recursos Estáticos

Pergunta: O número de recursos compartilhados é fixo e conhecido em tempo de compilação?

- **Solução:** Utilize **Regiões Críticas Nomeadas** (`#pragma omp critical (name)`).

- **Justificativa:** Este mecanismo é uma abstração de alto nível, declarativa e de fácil uso. Por ser uma diretiva rígida que exige um identificador estático como nome, ela se encaixa perfeitamente em cenários com um número predefinido de recursos.

4.0.2.3 Cenário 3: Recursos Dinâmicos

Pergunta: O número de recursos compartilhados é variável ou definido em tempo de execução?

- **Solução:** Utilize **Locks Explícitos** (`omp_lock_t`).
- **Justificativa:** Para cenários dinâmicos, abstrações estáticas são insuficientes. Locks explícitos são a solução necessária, pois são objetos que podem ser alocados dinamicamente em arrays. Isso confere à aplicação flexibilidade e escalabilidade.
- **Complexidade Adicional:** A flexibilidade dos locks explícitos acarreta uma maior complexidade de código, pois o programador se torna responsável por gerenciar manualmente todo o ciclo de vida do lock: inicialização (`omp_init_lock`), aquisição (`omp_set_lock`), liberação (`omp_unset_lock`) e destruição (`omp_destroy_lock`).

4.1 Conclusão

A análise detalhada das cinco implementações para a estimação de π permite extrair conclusões robustas sobre o uso de mecanismos de sincronização em OpenMP.

1. **A contenção é o principal inimigo do desempenho:** O experimento provou que a localização da sincronização é mais crítica do que a escolha da diretiva em si. Mover a sincronização para fora do laço principal, adotando o padrão de variável privada, foi o fator que proporcionou o maior salto de desempenho, transformando uma solução ineficiente em uma altamente escalável.

2. **Abstrações de alto nível vencem em ambos os quesitos:** A cláusula `reduction` não só entregou o melhor tempo de execução, superando inclusive a implementação manual otimizada, como também ofereceu o código mais limpo, seguro e produtivo. Isso demonstra o poder das abstrações do OpenMP quando usadas corretamente.

3. **Existe uma hierarquia clara de soluções:** Para problemas de redução, a escolha deve seguir a seguinte prioridade: 1º `reduction`, 2º Padrão de contador privado e, somente se nenhuma das anteriores for aplicável (o que é raro), mecanismos de baixo nível como `atomic` ou `critical` para atualizações esporádicas.

Em suma, a tarefa demonstrou de forma prática que o desenvolvimento de código paralelo eficiente exige mais do que apenas paralelizar um laço; requer uma análise cuidadosa dos padrões de acesso a dados para minimizar a contenção e a seleção da ferramenta de sincronização de mais alto nível que o problema permite.

paralelo_critical_comp.c

```c
/*
 * pi_critical_compartilhado.c
 * * Estimativa de Pi (Monte Carlo) usando um contador compartilhado
 * protegido por uma diretiva #pragma omp critical.
 * Esta abordagem sofre de alta contenção e baixo desempenho.
 *
 * Compilação: gcc -o pi_critical -fopenmp pi_critical_compartilhado.c -lm
 * Execução:   ./pi_critical
 */


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

const long NUM_PASSOS = 100000000;

int main() {
    long pontos_no_circulo = 0;

    printf("Executando Versão: Contador Compartilhado + critical\n");
    printf("Calculando Pi com %ld passos...\n", NUM_PASSOS);

    double start_time = omp_get_wtime();

    #pragma omp parallel
    {
        // Garante uma semente única por thread para rand_r
        unsigned int seed = time(NULL) ^ omp_get_thread_num();

        #pragma omp for
        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) {
                // A seção crítica protege o bloco de código.
                // Apenas uma thread pode executar este bloco por vez.
                #pragma omp critical
                {
                    pontos_no_circulo++;
                }
            }
        }
    } // Fim da região paralela

    double end_time = omp_get_wtime();
    double tempo_execucao = end_time - start_time;
```

```c
    double pi_estimado = 4.0 * pontos_no_circulo / NUM_PASSOS;

    printf("\nPi estimado = %f\n", pi_estimado);
    printf("Tempo de execucao: %f segundos\n", tempo_execucao);

    return 0;
}
```

paralelo_atomic_comp.c

```c
/*
 * pi_critical_compartilhado.c
 * * Estimativa de Pi (Monte Carlo) usando um contador compartilhado
 * protegido por uma diretiva #pragma omp critical.
 * Esta abordagem sofre de alta contenção e baixo desempenho.
 *
 * Compilação: gcc -o paralelo_atomic -fopenmp paralelo_atomic.c -lm
 * Execução:   ./paralelo_atomic
 */


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

const long NUM_PASSOS = 100000000;

int main() {
    long pontos_no_circulo = 0;

    printf("Executando Versão: Contador Compartilhado + atomic\n");
    printf("Calculando Pi com %ld passos...\n", NUM_PASSOS);

    double start_time = omp_get_wtime();

    #pragma omp parallel
    {
        // Garante uma semente única por thread para rand_r
        unsigned int seed = time(NULL) ^ omp_get_thread_num();

        #pragma omp for
        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) {
                // A seção atômica protege o bloco de código.
                // Apenas uma thread pode executar este bloco por vez.
                #pragma omp atomic
                pontos_no_circulo++;
            }
        }
    } // Fim da região paralela

    double end_time = omp_get_wtime();
    double tempo_execucao = end_time - start_time;
    double pi_estimado = 4.0 * pontos_no_circulo / NUM_PASSOS;

```

```c
    printf("\nPi estimado = %f\n", pi_estimado);
    printf("Tempo de execucao: %f segundos\n", tempo_execucao);

    return 0;
}
```

paralelo_critical_priv.c

```c

////////////////////////////////////////////////
///////////////VERSÃO 03////////////////////////
////////////////////////////////////////////////

#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() {
    unsigned int seed = time(NULL);
    long pontos_no_circulo_local = 0;

    #pragma omp parallel firstprivate(pontos_no_circulo_local, seed)
    {
        #pragma omp for
        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_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);
```

```
51
52      return 0;
53  }
54
```

paralelo_atomic_priv.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() {
    unsigned int seed = time(NULL);
    long pontos_no_circulo_local = 0;

    #pragma omp parallel firstprivate(pontos_no_circulo_local, seed)
    {
        #pragma omp for
        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_local++;
            }
        }
        #pragma omp atomic
            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;
}

```

paralelo_reduction.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include <math.h>

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

long pi_paralel_for_reduction() {
    long pontos_no_circulo = 0;

    #pragma omp parallel for reduction(+:pontos_no_circulo)
    for (long i = 0; i < NUM_PASSOS; i++) {

        unsigned int seed = time(NULL) ^ omp_get_thread_num();

        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++;
        }
    }

    return pontos_no_circulo;
}

int main() {
    double start_time, end_time;
    long total_pontos_no_circulo;

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

    start_time = omp_get_wtime();

    // Chama a função e armazena o valor retornado
    total_pontos_no_circulo = pi_paralel_for_reduction();

    end_time = omp_get_wtime();

    double tempo_paralelo = end_time - start_time;

    // Usa a variável local da main para calcular o Pi
    double pi_estimado = 4.0 * total_pontos_no_circulo / NUM_PASSOS;

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

```

```
50      return 0;
51  }
```