Análise de Desempenho e Produtividade de Mecanismos de Sincronização em OpenMP

Estudo de Caso: Estimação de π com Monte Carlo

Werbert Arles de Souza Barradas

Universidade Federal do Rio Grande do Norte (UFRN)
Disciplina de Programação Paralela - DCA3703

18 de setembro de 2025

Roteiro

1. Introdução
2. Metodologia
3. Análise e Resultados
4. Roteiro Prático
5. Conclusão

Introdução: O Problema da Condição de Corrida

- A manipulação de dados compartilhados em paralelo exige mecanismos de sincronização para evitar **condições de corrida**.
- O objetivo do projeto é analisar o impacto de diferentes estratégias de sincronização do OpenMP para um problema de acumulação paralela.
- **Estudo de Caso:** Estimação de π com o método de Monte Carlo, que requer a contagem segura de eventos.

Metodologia: O Problema Proposto

Lógica do Algoritmo

- Gerar N pontos aleatórios em um quadrado de lado 2.
- Contar quantos pontos caem em um círculo inscrito de raio 1.
- $\pi \approx 4 \times \frac{\text{Pontos no Círculo}}{\text{Total de Pontos}}$
- `rand_r()` garante a geração de números aleatórios de forma *thread-safe*.

Desafio

- Como gerenciar a atualização de um contador global de "pontos no círculo" de forma correta e eficiente?
- A operação `contador++` é a seção crítica que precisa ser protegida.

Metodologia: As Cinco Versões Analisadas

Foram implementadas cinco versões em C com OpenMP:

Sincronização Interna (Contador Compartilhado)

1. **Uso de** `#pragma omp critical`
2. **Uso de** `#pragma omp atomic`

Sincronização Externa (Contador Privado)

3. **Uso de** `#pragma omp critical` (após o laço)
4. **Uso de** `#pragma omp atomic` (após o laço)

Abstração de Alto Nível

5. **Uso da Cláusula** `reduction(+:contador)`

Metodologia: As Cinco Versões Analisadas

Sincronização Interna (Contador Compartilhado) #pragma omp critical

```
#pragma omp parallel
{
    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) {
            #pragma omp critical
            {
                pontos_no_circulo++;
            }
        }
    }
}
```

Metodologia: As Cinco Versões Analisadas

Sincronização Interna (Contador Compartilhado) #pragma omp atomic

```
#pragma omp parallel
{
    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) {
            #pragma omp atomic
            pontos_no_circulo++;
        }
    }
}
```

Metodologia: As Cinco Versões Analisadas

Sincronização Externa (Contador Privado) #pragma omp critical

```
    #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;
        }

    }
```

Metodologia: As Cinco Versões Analisadas

Sincronização Externa (Contador Privado) #pragma omp atomic

```
    #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;

    }
```

Metodologia: As Cinco Versões Analisadas

Abstração de Alto Nível
#pragma omp parallel for reduction(+:pontos_no_circulo)

```
#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++;
    }
}
```

Resultados Quantitativos (Dados Fictícios)

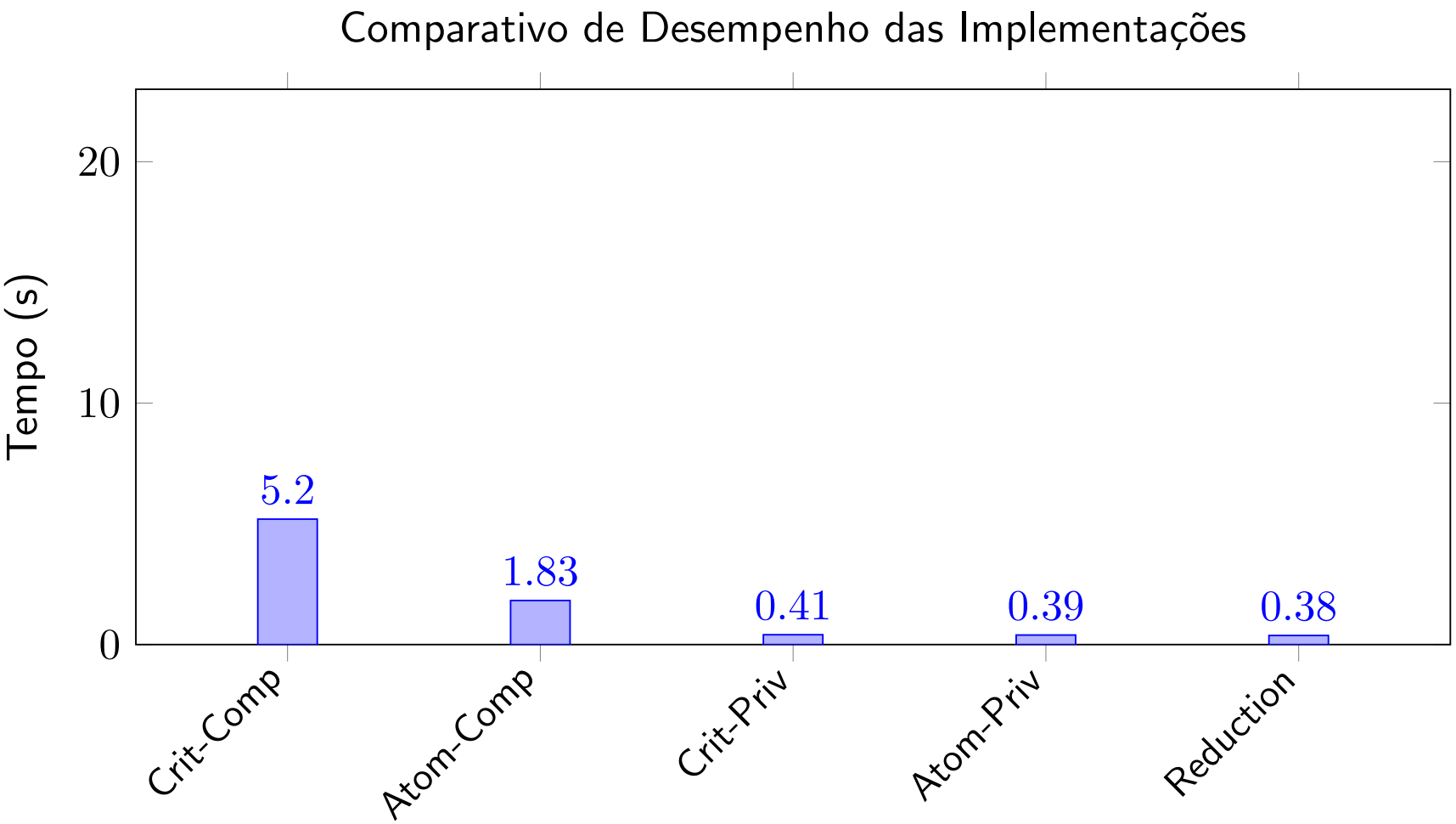

Figura: Gráfico comparando o desempenho das cinco implementações. A redução de tempo nas abordagens de contador privado e ‘reduction‘ é drástica.

Análise de Desempenho: O Impacto da Contenção

A contenção é o principal gargalo

O local da sincronização define o desempenho.

Sincronização Interna

- Versões 1 e 2 (‘critical‘/‘atomic‘ no laço)
- **Resultado:** Altíssima contenção.
- O paralelismo é anulado.
- Desempenho quase sequencial.

Sincronização Externa

- Versões 3, 4 e 5 (Privado/‘reduction‘)
- **Resultado:** Contenção mínima ou zero.
- Trabalho efetivamente paralelo.
- Desempenho ordens de magnitude superior.

Análise de Produtividade

`reduction`: O Vencedor em Produtividade

- **Código Declarativo:** Você expressa a *intenção* ("quero uma soma"), não o mecanismo.
- **Mais Limpo e Conciso:** Reduz a quantidade de código e a complexidade.
- **Menos Propenso a Erros:** O OpenMP gerencia a criação de variáveis privadas e a sincronização final, evitando erros comuns do programador.

Roteiro: Fundamentos e Padrões de Redução

Passo 1: Definir a Estratégia de Granularidade

A estratégia de **alta granularidade (fine-grained)**, onde cada recurso independente possui seu próprio lock, é o princípio fundamental.

- Minimiza a contenção e maximiza o desempenho.
- Evita gargalos sequenciais causados por locks de baixa granularidade.

Passo 2: Identificar Padrões de Redução

- **Pergunta:** O objetivo é acumular um único resultado (soma, produto, etc.) em um laço paralelo?

Roteiro: Fundamentos e Padrões de Redução

Passo 1: Definir a Estratégia de Granularidade

A estratégia de **alta granularidade (fine-grained)**, onde cada recurso independente possui seu próprio lock, é o princípio fundamental.

- Minimiza a contenção e maximiza o desempenho.
- Evita gargalos sequenciais causados por locks de baixa granularidade.

Passo 2: Identificar Padrões de Redução

- **Pergunta:** O objetivo é acumular um único resultado (soma, produto, etc.) em um laço paralelo?
- **Solução Ideal:** Utilize a **cláusula** `reduction`.
- **Justificativa:** É a abordagem de mais alto nível, com melhor desempenho e maior produtividade.

Roteiro: Cenários Estáticos vs. Dinâmicos

Se não for uma redução, analise a natureza dos recursos:

Cenário Estático

- **Recursos:** Número fixo, conhecido em tempo de compilação.
- **Solução:** `#pragma omp critical (name)`.
- **Vantagem:** Simples e declarativo, implementa alta granularidade para um conjunto fixo de locks.

Cenário Dinâmico

- **Recursos:** Número variável, definido em tempo de execução.
- **Solução:** Array de `omp_lock_t`.
- **Vantagem:** Flexível e escalável, a única solução para problemas dinâmicos.

Conclusão

- A comparação das 5 versões demonstrou que **evitar a contenção** é a estratégia mais importante para o desempenho.
- O padrão de **variável privada** (manual ou via ‘reduction‘) é a solução correta para problemas de acumulação.
- A cláusula `reduction` provou ser a melhor solução, vencendo em **desempenho** (otimização do compilador) e **produtividade** (código limpo e declarativo).
- **Regra Geral:** Sempre prefira a abstração de mais alto nível que o OpenMP oferece para resolver o seu problema.

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;

```

```
51      printf("\nPi estimado = %f\n", pi_estimado);
52      printf("Tempo de execucao: %f segundos\n", tempo_execucao);
53
54      return 0;
55  }
```

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  }
```