UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
IDCA3703 - PROGRAMAÇÃO PARALELA

TAREFA 10 - MECANISMOS DE SINCRONIZAÇÃO: ATOMIC E REDUCTION
RELATÓRIO DE EXECUÇÃO

ERNANE FERREIRA ROCHA JUNIOR

NATAL/RN, 30 DE ABRIL DE 2025

SUMÁRIO

1. INTRODUÇÃO

A computação paralela é uma ferramenta essencial para o processamento eficiente de grandes volumes de dados e simulações numéricas. No contexto da linguagem *C*, a biblioteca *OpenMP* oferece uma abordagem prática para o desenvolvimento de aplicações paralelas em arquiteturas multicore, fornecendo diversas diretivas para controle de paralelismo e sincronização entre *threads*. Um exemplo clássico de aplicação paralelizável é a estimativa do número π (pi) pelo método de Monte Carlo, que se baseia em simulações probabilísticas para determinar a razão entre pontos aleatórios dentro de um quarto de círculo e o total de pontos lançados em um quadrado.

Neste experimento, objetiva-se reimplementar uma estimativa de π utilizando diferentes estratégias de sincronização de variáveis compartilhadas entre *threads* em *OpenMP: #pragma omp critical*, *#pragma omp atomic* e a cláusula *reduction*. Além disso, foram implementadas versões alternativas propositalmente ineficientes, que realizam sincronizações com maior frequência, para demonstrar o impacto negativo dessas práticas no desempenho da aplicação. A comparação entre as abordagens busca avaliar o custo de cada mecanismo em termos de desempenho (tempo de execução) e a sua aplicabilidade em contextos práticos, considerando também a clareza e produtividade no desenvolvimento do código.

2. METODOLOGIA

Para realizar a estimativa de π, foi implementado um algoritmo baseado no método de Monte Carlo, onde pares de números aleatórios (x, y) são gerados dentro do intervalo $[0, 1]$ e testados quanto à condição $x^2 + y^2 \leq 1$. A razão entre o número de pontos que satisfazem essa condição e o total de pontos lançados aproxima o valor de $\frac{\pi}{4}$. Multiplicando esse valor por 4 obtém-se a estimativa final de π. O número total de lançamentos foi definido como $N = 999.999.999$, um valor suficientemente grande para evidenciar diferenças de desempenho entre as abordagens paralelas.

Todas as implementações foram desenvolvidas em *C*, utilizando a biblioteca *OpenMP* para paralelizar o laço principal. As *threads* geradas utilizam geradores de números pseudoaleatórios independentes por meio da função *rand_r*(), que é *thread-safe,* com *seeds* diferentes baseados no tempo e no identificador da *thread*, garantindo a independência estatística dos experimentos.

Três versões principais do algoritmo foram comparadas:

1. **Versão com *#pragma omp critical***: cada *thread* acumula um contador local de acertos e o valor final é somado à variável compartilhada usando uma região crítica.
2. **Versão com *#pragma omp atomic***: semelhante à anterior, mas a soma à variável compartilhada é feita com operação atômica.
3. **Versão com cláusula *reduction*:** a variável de contagem é reduzida automaticamente pelas *threads*, sem a necessidade de instruções de sincronização explícitas.

Adicionalmente, foram desenvolvidas duas versões ineficientes: uma com *#pragma omp critical* e outra com *#pragma omp atomic* utilizadas a cada acerto (isto é, a cada iteração bem-sucedida do teste de condição), o que simula uma má prática de sincronização excessiva. Todas as versões foram compiladas com suporte à *OpenMP* usando *gcc* com a *flag -fopenmp* e executadas em ambiente controlado, medindo-se o tempo total de execução de cada versão com auxílio da função *gettimeofday*().

3. RESULTADOS

Os testes foram realizados utilizando o mesmo valor de $N = 999.999.999$ pontos para todas as versões do algoritmo, assegurando comparabilidade entre os experimentos. O tempo de execução e o valor estimado de π foram registrados para cada implementação. A tabela abaixo resume os resultados obtidos:

Versão	Estratégia de Sincronização	π (Estimado)	Tempo de Execução (s)
v1	*#pragma omp critical* (*local count*)	3.141576695141577	03.098
v2	*#pragma omp atomic* (*local count*)	3.141576695141577	03.095
v3	*reduction(+:count)*	3.141576695141577	03.048
v4	*#pragma omp atomic (por acerto*)	3.141631403141631	34.350
v5	*#pragma omp critical* (*por acerto*)	3.141676511141676	60.818

Observa-se que as três primeiras versões — que utilizam contadores locais com sincronização posterior (v1, v2, v3) — apresentam tempos de execução muito semelhantes, todos em torno de 3 segundos, com estimativas de π praticamente idênticas. A versão com reduction se destaca levemente em desempenho, pois evita chamadas explícitas de sincronização e é otimizada internamente pelo compilador.

Em contrapartida, as versões ineficientes (v4 e v5), nas quais a sincronização ocorre a cada acerto no laço de simulação, apresentam tempos drasticamente maiores — mais de **10x** mais lentas — devido à contenção intensa entre *threads*, que bloqueiam umas às outras continuamente ao acessar a variável compartilhada.

Esses resultados ilustram claramente o impacto do tipo e frequência de sincronização sobre o desempenho de aplicações paralelas, mesmo em algoritmos simples como o de Monte Carlo.

4. CONCLUSÃO

A partir dos experimentos realizados com diferentes mecanismos de sincronização no algoritmo de Monte Carlo para estimativa de π, foi possível observar com clareza como a escolha adequada da estratégia pode impactar significativamente o desempenho de aplicações paralelas com *OpenMP*.

As versões que utilizam contadores locais seguidos de uma sincronização única (com *critical*, *atomic* ou *reduction*) apresentaram excelente desempenho e produziram estimativas consistentes de π. Dentre elas, a versão com cláusula *reduction* demonstrou leve superioridade, tanto por eliminar a necessidade de código adicional de sincronização quanto por possibilitar otimizações automáticas feitas pelo compilador.

Em contraste, as versões que aplicam sincronização diretamente dentro do laço de contagem — especialmente com *critical* a cada acerto — apresentaram desempenho substancialmente inferior, evidenciando o alto custo da contenção por acesso frequente a regiões críticas.

Com base nos resultados e na análise realizada, é possível estabelecer um roteiro prático para a escolha do mecanismo de sincronização:

- Usar ***reduction*** sempre que for possível expressar a operação em forma de redução (ex.: somas, médias, mínimos/máximos), pela simplicidade e eficiência.
- Usar ***atomic*** quando o acesso à variável compartilhada for pontual e a operação puder ser representada com operações atômicas simples, como incrementos.
- Usar ***critical*** apenas quando as operações críticas forem complexas ou não suportadas por *atomic*, e evitar seu uso em trechos executados com alta frequência.
- Evitar sincronização dentro de laços intensos, especialmente quando múltiplos *threads* podem tentar acessar uma região crítica simultaneamente, o que degrada seriamente a escalabilidade.

Portanto, a escolha consciente e contextualizada do mecanismo de sincronização não apenas melhora o desempenho da aplicação, mas também contribui para a sua clareza, escalabilidade e manutenibilidade.

5. ANEXOS

- Repositório no Github com o programa desenvolvido: https://github.com/ErnaneJ/parallel-programming-dca3703