UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
IDCA3703 - PROGRAMAÇÃO PARALELA

TAREFA 8 - COERÊNCIA DE CACHE E FALSO COMPARTILHAMENTO
RELATÓRIO DE EXECUÇÃO

ERNANE FERREIRA ROCHA JUNIOR

NATAL/RN, 27 DE ABRIL DE 2025

SUMÁRIO

1. INTRODUÇÃO

O método de Monte Carlo é uma técnica probabilística amplamente utilizada para resolver problemas matemáticos complexos que podem ser formulados de maneira estatística. Entre suas aplicações mais clássicas está a estimativa do valor de π, baseada na geração de pontos aleatórios dentro de um quadrado que circunscreve um círculo. A proporção de pontos que caem dentro do círculo, em relação ao total de pontos gerados, fornece uma aproximação do valor de π.

Com a evolução dos sistemas computacionais e o surgimento de arquiteturas paralelas, tornou-se viável explorar o paralelismo para acelerar a execução desses algoritmos, principalmente utilizando *OpenMP* para programação concorrente em ambientes de memória compartilhada. Dentro desse contexto, o presente trabalho propõe a comparação de diferentes abordagens para a estimativa de π, combinando técnicas de paralelismo com geradores de números aleatórios tradicionais (*rand*()) e reentrantes (*rand_r*()).

Além de avaliar o impacto do uso de regiões críticas e de estruturas de dados compartilhadas no desempenho das soluções, busca-se analisar também a precisão dos resultados e as diferenças de tempo de execução obtidas para cada abordagem. O objetivo final é entender melhor como escolhas de implementação e sincronização afetam a eficiência de algoritmos paralelos em problemas estocásticos.

2. METODOLOGIA

Para a realização deste trabalho, foi desenvolvido um programa em linguagem *C* utilizando a biblioteca *OpenMP* para implementar diferentes versões de um estimador estocástico do número π. A estratégia baseia-se no método de Monte Carlo, em que pontos aleatórios são gerados dentro de um quadrado de lado unitário e verificam-se quantos destes caem dentro do círculo inscrito. O valor de π é então estimado pela razão entre os pontos internos ao círculo e o total de pontos, multiplicada por quatro.

Quatro versões do algoritmo foram implementadas, variando-se o modo de geração de números aleatórios e a estratégia de acumulação dos resultados parciais de cada thread. As versões são as seguintes:

- **Versão 1**: Utiliza a função *rand*() para geração de números aleatórios e sincroniza a atualização da variável compartilhada count por meio de uma região crítica (*#pragma omp critical*).
- **Versão 2**: Também baseada em *rand*(), mas cada *thread* armazena seu resultado em um vetor compartilhado, eliminando a necessidade de regiões críticas durante a execução paralela; a soma é realizada após o término da região paralela.
- **Versão 3**: Substitui *rand*() por *rand_r*(), que permite a geração de números aleatórios de forma independente por *thread*, ainda utilizando uma região crítica para a acumulação dos resultados.
- **Versão 4**: Combina *rand_r*() com o uso de vetor compartilhado, maximizando a independência entre *threads* e evitando o uso de regiões críticas durante a geração e contagem dos pontos.

Cada versão foi executada utilizando o número padrão de *threads* configurado no ambiente de execução, sem especificação manual do número de *threads*, permitindo que o *OpenMP* ajuste conforme a disponibilidade do sistema. Em todas as execuções, o número de pontos gerados foi mantido constante, com
$N = 9.999.999$, garantindo a comparabilidade entre as abordagens.

O tempo de execução de cada versão foi medido utilizando uma função auxiliar baseada em *gettimeofday*(), registrando o tempo decorrido para o processamento completo de cada abordagem. Ao final da execução, foram impressos o valor estimado de π e o tempo gasto, possibilitando a análise de desempenho e precisão.

3. RESULTADOS

Após a implementação das quatro versões do estimador estocástico de π, foram realizadas as execuções para coleta dos resultados de precisão e desempenho. As estimativas obtidas para π e os respectivos tempos de execução mostraram diferenças sutis na precisão, mas variações perceptíveis no tempo de processamento entre as abordagens.

A tabela a seguir resume os valores estimados de π e os tempos de execução registrados para cada uma das versões:

Versão	**Estimativa de π**	**Tempo (s)**
Versão 3 - *critical* + *rand_r*()	3.141924714192471	0.033
Versão 4 - *array* + *rand_r*()	3.141924714192471	0.048
Versão 1 - *critical* + *rand*()	3.126013512601351	0.073
Versão 2 - *array* + *rand*()	3.133681113368112	0.056

Observa-se que as versões que utilizam *rand_r*(), que é *thread-safe*, apresentaram estimativas mais próximas do valor real de π (aproximadamente $3,1416$) em comparação com as versões que utilizam *rand*(). Além disso, a versão que combinou *rand_r*() com vetor de acumulação (*array* + *rand_r*()) obteve uma precisão praticamente idêntica àquela da versão com região crítica, com tempo de execução competitivo.

Em termos de desempenho, as versões que evitam o uso de regiões críticas durante a acumulação dos resultados — ou seja, aquelas que utilizam um vetor para armazenar os resultados locais das threads — apresentaram tempos de execução menores, em especial a combinação de *rand*() com vetor (*array* + *rand*()), que reduziu consideravelmente o tempo em comparação à abordagem com *critical*. Porém, o uso de *rand_r*(), mesmo com vetor, não resultou no menor tempo absoluto, indicando que o custo adicional do gerador de números aleatórios *thread-safe* também impacta a performance e as ocorrências de falso compartilhamento também.

4. CONCLUSÃO

O estudo realizado evidenciou as diferenças práticas entre distintas estratégias de paralelização aplicadas à estimação estocástica de π. As quatro versões desenvolvidas permitiram comparar não apenas o impacto do controle de concorrência (*critical versus* vetor de acumulação), mas também o efeito do método de geração de números aleatórios (*rand*() *versus rand_r*()).

Os resultados mostraram que o uso de *rand_r*(), associado a *seeds* independentes por *thread*, promoveu maior precisão na estimativa de π, evidenciando a importância de evitar conflitos em geradores de números aleatórios em ambientes paralelos. Em relação ao desempenho, embora a eliminação de regiões críticas por meio de vetores de acumulação tenha se mostrado eficaz para reduzir o tempo de execução, a diferença entre as versões que usam *rand*() e *rand_r*() revelou que a escolha da função de geração de números aleatórios também influencia diretamente o tempo gasto, mesmo em abordagens eficientes de acumulação.

Assim, conclui-se que, para aplicações onde tanto precisão quanto desempenho são críticos, a combinação de um gerador seguro para múltiplas threads (*rand_r*()) com técnicas de acumulação local (vetores), trabalhando-se para evitar o falso compartilhamento, oferece o melhor equilíbrio. Porém, para casos onde é tolerável o uso seguro de regiões criticas, em termos de velocidade, esse cenário pode ser o mais interessante. Ou seja, contextos onde a precisão é menos exigente e o tempo de execução é prioritário, pode ser aceitável trabalhar com controle adequado de concorrência para acelerar a execução.

O exercício também reforçou a necessidade de considerar cuidadosamente a natureza das funções utilizadas dentro de regiões paralelas e destacou a importância da escolha da estratégia de sincronização para a obtenção de programas paralelos mais eficientes e corretos. Uma simples modificação, pode mudar significativamente tanto a precisão nos resultados quanto o tempo necessário para obtê-los.

5. ANEXOS

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