Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Engenharia da Computação e Automação - DCA
DCA3703 - Programação Paralela

Tarefa 08: Coerência de Cache e Falso Compartilhamento

Discente: Quelita Míriam
Docente: Samuel Xavier de Souza

I. INTRODUÇÃO

Com o aumento da demanda por desempenho computacional, técnicas de paralelismo vêm sendo amplamente empregadas para acelerar algoritmos matemáticos. Neste contexto, esta atividade objetiva comparar diferentes formas de paralelização da estimativa de π com a biblioteca OpenMP, avaliando o impacto da geração de números aleatórios e estratégias de agregação de resultados no tempo de execução e na precisão obtida.

II. METODOLOGIA

Foi implementado um programa[1] em linguagem C que estima o valor de π utilizando o método de Monte Carlo com diferentes estratégias de paralelização. O código foi compilado com as seguintes versões testadas:

- ***rand()* + *critical***: cada thread utiliza a função *rand()* para gerar pontos aleatórios e acumula o número de acertos em uma região crítica com *#pragma omp critical*.
- ***rand()* + *vetor***: cada *thread* acumula localmente seus acertos em um vetor indexado pelo ID da *thread*, evitando regiões críticas.
- ***rand_r()* + *critical***: versão semelhante à primeira, mas utilizando *rand_r()*, uma função reentrante que permite sementes independentes por *thread*.
- ***rand_r()* + *vetor***: combinação de sementes independentes e armazenamento local por *thread*.

Todas as versões utilizam 10 milhões de tentativas para estimar o valor de π. O tempo de execução é medido utilizando a função *gettimeofday()*.

III. RESULTADOS E DISCUSSÕES

A imagem a seguir apresenta os tempos de execução e os valores estimados de π para cada versão do programa:

Versão	Tempo(s)	Valor de pi
rand() + critical	2.974439	3.1416536000
rand() + vetor	3.516753	3.1424372000
rand_r() + critical	0.080283	3.1425256000
rand_r() + vetor	0.100364	3.1423344000

[1] Código para a realização da tarefa:
https://github.com/quelita2/programacao-paralela/blob/main/topico01/tarefa08/src/main.c

Imagem 1: Resultado de Compilação

Com base nos resultados obtidos, é possível observar diferenças marcantes no desempenho das quatro versões do programa, influenciadas principalmente pelos efeitos da coerência de *cache* e do falso compartilhamento.

A terceira versão (***rand_r() + critical***) apresentou o melhor tempo de execução, sendo a mais eficiente entre todas. Isso se deve ao uso da função *rand_r()*, que permite a geração de números aleatórios de forma independente por cada *thread*, evitando conflitos de acesso. Apesar de utilizar uma região crítica para somar os acertos, essa seção é executada apenas uma vez por *thread* e de forma rápida, não causando gargalos significativos. Essa simplicidade no controle dos dados compartilhados, somada à independência das *threads*, torna essa versão a mais equilibrada em termos de paralelismo e desempenho.

A segunda versão (***rand() + vetor***) foi a que apresentou o pior desempenho. Embora evite regiões críticas, ela sofre fortemente com o falso compartilhamento, pois diferentes *threads* escrevem em posições próximas de um vetor alocado dinamicamente. Essas posições, mesmo sendo diferentes, podem compartilhar a mesma linha de cache, o que provoca interferência entre as *threads* e reduz a eficiência da execução paralela. Além disso, o uso da função *rand()* não é seguro em ambientes *multithread*, o que pode gerar contenções internas e resultados imprevisíveis.

A primeira versão (***rand() + critical***) teve desempenho melhor que a segunda versão, mas ainda assim lento, devido ao uso de *rand()* e da região crítica. A chamada *rand()* não é *thread-safe*, e quando usada junto com uma região crítica, força as *threads* a esperarem mais tempo umas pelas outras, aumentando o tempo total da execução.

A quarta versão (***rand_r() + vetor)*** teve desempenho razoável, melhor que as que usam *rand()*, mas inferior à versão com *critical*. Isso acontece porque, embora *rand_r()* melhore a geração paralela de números aleatórios, o uso do vetor ainda mantém o problema do falso compartilhamento. Cada *thread* grava em uma posição separada do vetor, mas as posições estão muito próximas na memória, o que causa competição indireta pelo cache.

Por meio desta análise é possível observar por que a terceira versão, com *rand_r()* e *critical,* foi a mais eficiente por equilibrar bem a segurança da geração paralela de dados e o acesso controlado à variável compartilhada. Enquanto a segunda versão, com *rand()* e vetor, é a pior por combinar uma função insegura com um padrão de acesso à memória que prejudica o desempenho em sistemas paralelos.

IV. CONCLUSÃO

Com base nos testes realizados, conclui-se que a escolha adequada de técnicas de paralelismo e controle de acesso à memória tem impacto direto no desempenho de programas *multithread*. A versão que utilizou *rand_r()* com região crítica demonstrou ser a mais eficiente, combinando geração segura de números aleatórios em paralelo com controle simples e eficaz sobre os dados compartilhados. Por outro lado, a pior performance foi observada na versão que combinou *rand()* com vetor, prejudicada tanto pelo uso de uma função não segura em múltiplas *threads* quanto pelos efeitos do falso

compartilhamento no acesso ao vetor. Esses resultados evidenciam a importância de se considerar o funcionamento da memória cache e o padrão de comunicação entre *threads* ao implementar algoritmos paralelos. Assim, otimizações simples como evitar compartilhamento excessivo de memória e escolher funções *thread-safe* são fundamentais para alcançar desempenho eficiente em ambientes com múltiplos núcleos.