Universidade Federal do Rio Grande do Norte
Departamento de Engenharia da Computação e Automação
DCA3703 - Programação Paralela
Tarefa 8 - Coerência de Cache e falso compartilhamento
Aluno: Daniel Bruno Trindade da Silva

1 Introdução

O número π é uma constante matemática fundamental que pode ser estimada através de métodos estocásticos, como a simulação de Monte Carlo. Este trabalho tem como objetivo implementar e comparar diferentes estratégias de paralelização da estimativa de π utilizando a biblioteca OpenMP em linguagem C.

Além da implementação, foi feita uma análise do tempo da execução de cada versão, considerando os efeitos da coerência de cache e do falso compartilhamento, fenômenos que impactam o desempenho de aplicações paralelas em arquiteturas modernas.

2 Metodologia

O método estocástico de Monte Carlo para estimar o valor de π é uma técnica probabilística que usa números aleatórios para resolver um problema matemático. Para estimar π utilizaremos um circulo de raio 1 circunscrito em um quadrado de lado 2, ambos posicionados no centro do eixo de coordenadas como mostrado na figura 1:

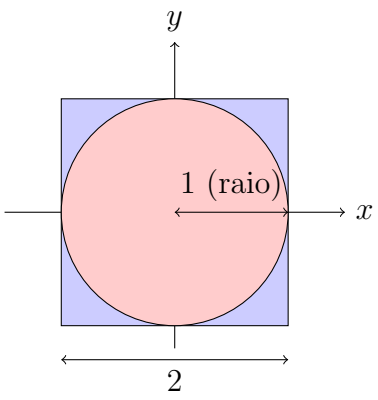

Figure 1: Circulo de raio 1 circunscrito em um quadrado de lado 2

Em seguida geraremos muitos pares (x,y) com valores aleatórios entre 0 e 2, ou seja, dentro da área do quadrado, a proporção de pontos encontrados dentro do círculo em relação ao total gerado se aproxima da razão entre as áreas:

$$\frac{\text{Pontos no Círculo}}{\text{Total de Pontos}} \approx \frac{\pi}{4}$$

Então para estimarmos o valor de π pelo método de Monte Carlo temos:

$$\pi \approx 4 \times \frac{\text{Pontos no Círculo}}{\text{Total de Pontos}}$$

Assim, a base de nosso código será composto por um laço de repetição que gerará n pares aleatórios (x,y) e testará se os pontos estão dentro ou não do circulo. Ao final utilizaremos a proporção de acertos pelo número de pares (x, y) para estimarmos o valor de π.

Nessa tarefa foram desenvolvidos quatro versões do código mudando alguns pontos da abordagem para a paralelização como se segue:

- A primeira utiliza a função `rand()` para geração de números aleatórios, com cada thread acumulando seus acertos em uma variável privada e somando ao total através de uma região crítica `#pragma omp critical`.
- A segunda versão também usa `rand()`, mas cada thread armazena seus acertos em uma posição distinta de um vetor compartilhado, realizando a acumulação final de forma serial após a região paralela.
- A terceira e quarta versões seguem a mesma lógica das duas primeiras, porém substituindo rand() por `rand_r()`, uma função *thread-safe* que permite melhor paralelização ao eliminar conflitos no gerador de números aleatórios.

Em todas as versões foi utilizada a função `gettimeofday()` para registrar os tempos de execução e a quantidades de pontos geradas para todas as versões foi de 1.000.000.000.

3 Resultados

Cada experimento utilizou 10^8 pontos gerados aleatoriamente e foi executado no mesmo ambiente computacional para garantir a comparação justa.

Os resultados obtidos foram:

- **Versão 1 — rand() com região crítica:**
  - Tempo de execução: 13,270068 segundos
  - Estimativa de π: 3,141687
- **Versão 2 — rand_r() com região crítica:**
  - Tempo de execução: 0,6296 segundos
  - Estimativa de π: 3,141577
- **Versão 3 — rand() com vetor compartilhado:**
  - Tempo de execução: 13,9920 segundos
  - Estimativa de π: 3,141661
- **Versão 4 — rand_r() com vetor compartilhado:**
  - Tempo de execução: 0,9164 segundos
  - Estimativa de π: 3,141621

Observa-se que todas as versões forneceram estimativas bastante próximas do valor real de π (aproximadamente 3,14159265), com diferenças esperadas para um método estocástico baseado em amostragem aleatória.

Quanto ao desempenho, foi possível observar que:

- As versões que utilizam `rand_r()` apresentaram tempos de execução significativamente menores, devido à natureza *thread-safe* desta função, que evita conflitos no acesso ao gerador de números aleatórios.
- A eliminação da região crítica através do uso de um vetor compartilhado melhorou o desempenho, porém o uso da função `rand()` ainda limitou a velocidade nas versões correspondentes.
- Mesmo nas versões vetorizadas, o falso compartilhamento pode ter impactado levemente o desempenho, pois múltiplas threads escrevem em posições adjacentes de memória, afetando a eficiência da cache.

4 Resultados

Cada experimento utilizou 10^8 pontos gerados aleatoriamente e foi executado no mesmo ambiente computacional para garantir a comparação justa.

Os resultados obtidos estão apresentados na Tabela 1.

Versão	**Descrição**	**Tempo (s)**	**Estimativa de π**
1	`rand()` com `critical`	13,2701	3,141687
2	`rand_r()` com `critical`	0,6296	3,141577
3	`rand()` com vetor compartilhado	13,9920	3,141661
4	`rand_r()` com vetor compartilhado	0,9164	3,141621

Table 1: Tempos de execução e estimativas de π para cada versão do código.

Observa-se que todas as versões forneceram estimativas bastante próximas do valor real de π (aproximadamente 3,14159265), com diferenças esperadas para um método estocástico baseado em amostragem aleatória.

Quanto ao desempenho, foi possível observar que:

- As versões que utilizam `rand_r()` apresentaram tempos de execução significativamente menores, devido à natureza *thread-safe* desta função, que evita conflitos no acesso ao gerador de números aleatórios.
- A eliminação da região crítica através do uso de um vetor compartilhado melhorou o desempenho, porém o uso da função `rand()` ainda limitou a velocidade nas versões correspondentes.
- Mesmo nas versões vetorizadas, o falso compartilhamento pode ter impactado levemente o desempenho, pois múltiplas threads escrevem em posições adjacentes de memória, afetando a eficiência da cache.

5 Análise dos Resultados

A análise dos resultados obtidos permite identificar diferentes fatores que influenciam o desempenho das versões paralelas da estimativa estocástica de π. Dois aspectos principais devem ser destacados: o impacto da função geradora de números aleatórios utilizada (`rand()` versus `rand_r()`) e a organização do acesso à memória compartilhada pelas threads.

5.1 Impacto da função geradora de números aleatórios

As versões que utilizam `rand_r()` apresentaram tempos de execução significativamente menores em comparação às versões que utilizam `rand()`. Isso ocorre porque `rand()` é uma função tradicionalmente *não thread-safe*, ou seja, seu uso simultâneo por múltiplas threads gera contenção em torno de variáveis internas globais, introduzindo atrasos devido à necessidade de sincronização implícita.

Já `rand_r()` é uma função *thread-safe* que opera sobre uma semente local passada por argumento. Cada thread pode, portanto, gerar seus próprios números aleatórios de forma independente, sem interferência entre si. Essa independência elimina a contenção no gerador de números aleatórios e permite que as threads realmente executem em paralelo, maximizando o ganho de desempenho com o uso de múltiplos núcleos.

5.2 Impacto da estratégia de acumulação e falso compartilhamento

Outra diferença entre as implementações está relacionada à forma como as threads acumulam os acertos:

- Nas versões com `critical`, cada thread utiliza uma variável local para contar seus acertos, mas a atualização da variável global de hits ocorre dentro de uma região crítica. Como apenas uma thread pode executar dentro da região crítica por vez, existe uma limitação na escalabilidade do desempenho, especialmente conforme o número de threads aumenta.

- Nas versões vetorizadas, cada thread escreve seus acertos em uma posição exclusiva de um vetor compartilhado. Isso elimina a necessidade de regiões críticas durante a execução principal, favorecendo a paralelização e melhorando o desempenho geral.

No entanto, o uso de vetores compartilhados pode introduzir um outro fenômeno conhecido como **falso compartilhamento** (*false sharing*). O falso compartilhamento ocorre quando múltiplas threads acessam e modificam variáveis que estão armazenadas em posições de memória próximas e que pertencem à mesma linha de cache. Mesmo que cada thread acesse apenas sua própria posição do vetor, a arquitetura do sistema pode fazer com que modificações próximas invalidem linhas de cache, resultando em atualizações frequentes e ineficientes entre os caches dos processadores.

O falso compartilhamento gera penalidades de desempenho, pois aumenta a comunicação e a sincronização implícita entre os núcleos, reduzindo o ganho esperado de paralelismo.

Apesar disso, nos resultados obtidos, o impacto do falso compartilhamento foi relativamente pequeno em comparação ao ganho expressivo obtido pela eliminação da região crítica e pela adoção da função `rand_r()`.

6 Conclusão

Neste trabalho, foi realizada a implementação e análise de quatro versões paralelas para a estimativa de π utilizando o método de Monte Carlo com OpenMP.

Observou-se que o uso da função `rand_r()`, por ser *thread-safe*, proporcionou ganhos significativos de desempenho em relação à função `rand()`, eliminando a contenção entre as threads. Além disso, a substituição de regiões críticas por vetores compartilhados também melhorou a eficiência, apesar do leve impacto do falso compartilhamento.

Conclui-se que, para otimizar aplicações paralelas, é essencial adotar geradores de números aleatórios adequados ao ambiente multi-thread e estruturar o acesso à memória de forma a minimizar conflitos de cache.