Relatório aula 14

Diogo Pereira Lobo

Supeeeeeeer Computação

Monte Carlo 1:

Não, os números foram gerados de maneira pseudo-aleatória, com o uso da função uniform\_real\_distribution. Utilizando uma seed com base no tempo de chamada da função, houve uma tentativa maior de randomização, garantindo, quase sempre, um novo parâmetro para cada vez que a função é acionada.

valor de pi: 3.1458

tempo de execução: 0.0256994 s

Monte Carlo 2:

O principal problema nesse caso é que várias threads estão competindo pela geração no número aleatório, mudando o estado da função.

O gerador de números pseudo-aleatórios (default\_random\_engine) mantém um estado interno para gerar a sequência, e cada chamada a distribution(generator) avança esse estado.

A cada thread executada. Isso cria uma condição de corrida, onde, devido a tentativa conjunta de gerar o valor aleatório, mudando o estado do gerador, pode causar resultados imprevisíveis e corromper a geração dos números

A semente é gerada fora do loop paralelo e é compartilhada por todas as threads. Embora isso não gere sequências completamente independentes, o gerador (default\_random\_engine) é determinístico, e cada thread está utilizando uma sequência da mesma fonte. O estado interno do gerador está sendo atualizado de forma sequencial, garantindo que, mesmo com uma semente única, os números gerados para cada thread não sejam iguais.

Isso faz com que, na prática, as threads estejam gerando amostras mais bem distribuídas, o que melhora a precisão no cálculo de pi. A aleatoriedade não é ótima, mas ainda assim é suficiente para produzir resultados mais consistentes.

Esses fatores foram corrigidos no código 3.

Local:

valor de pi: 3.14684

tempo de execução: 0.00681883 s

cluster:

valor de pi: 3.13312

tempo de execução: 0.0125038 s

Monte Carlo 3

Não precisei realizar a retirada da região critica, pois estou compartilhando entre as threads, por meio do reduction(), a variável de soma final. Todavia, apliquei a mudança para corrigir o problema do valor aleatório discutido na sessão anterior. Trazendo a geração do número para dentro da região paralelizada, cada thread pôde ter sua própria semente, aumentando o fator de aleatoriedade discutido anteriormente.

Sim, houve alteração no valor de pi, aproximando-o um pouco do valor real. Ademais, o tempo de execução diminuiu consideravelmente, possivelmente devido a perca da condição de corrida imposta na versão anterior.

cluster

valor de pi: 3.14328

tempo de execucao: 0.00457324 s

local

valor de pi: 3.14624

tempo de execucao: 0.00577071 s

|  |  |  |  |
| --- | --- | --- | --- |
| pi | sequencial | Paralela 1 | Paralela 2 |
| 3.1458 | 0.0256994 s |  |  |
| 3.13312 |  | 0.0125038 s |  |
| 3.14328 |  |  | 0.00457324 s |

1. Apenas houve uma grande diferença no pento da última paralela. Os dois primeiros códigos possuem tempo semelhante, devido, principalmente a condição e corrida imposta.
2. Sim, em todas as versões, o cálculo de pi ficou bem próximo do seu valor reduzido de 3,14.
3. Entender o problema de paralelizar a criação das sementes, para melhorar a aleatoriedade e retirar a condição de corrida.
4. Sim, como pode-se observar no código 3, que é cerca de 10 vezes mais rápido que os outros, devido ao uso de threads, e consegue chegar em valores bem próximos de pi.