Supercomputação
Entrega - Atividade 14
Arthur Tamm

Introdução

Nesta atividade, implementamos o cálculo do valor de pi utilizando o método de Monte Carlo. O algoritmo baseia-se em gerar pontos aleatórios dentro de um quadrado unitário e calcular a proporção de pontos que caem dentro de um círculo inscrito nesse quadrado.

Desenvolvemos três versões do algoritmo:

1. Versão Sequencial (pi_serial.cpp)
2. Primeira Tentativa de Paralelização (pi_parallel1.cpp)
3. Melhoria na Paralelização (pi_parallel2.cpp)

Cada versão foi executada 10 vezes, e os resultados foram coletados para análise.

Parte 1: Implementação Sequencial

Descrição da Implementação

Na versão sequencial, geramos N=10e6 pontos aleatórios dentro do quadrado unitário. Para cada ponto gerado, verificamos se ele está dentro do círculo unitário centrado na origem.

Resultados

- Valor médio estimado de pi: 3.1417
- Tempo médio de execução: aproximadamente 2.0 segundos

Reflexão sobre a Implementação

A implementação sequencial foi direta e sem grandes dificuldades. A sequência de números aleatórios foi gerada corretamente utilizando um gerador adequado e uma distribuição uniforme. O maior desafio foi garantir a precisão no cálculo e na contagem dos pontos dentro do círculo.

Parte 2: Primeira Tentativa de Paralelização

Descrição da Paralelização

Na primeira tentativa de paralelização, utilizamos a diretiva **#pragma omp parallel** para distribuir o trabalho entre múltiplas threads. O laço de iteração foi paralelizado usando **#pragma omp for**, e a variável **inside_circle** foi tratada com a cláusula **reduction(+:inside_circle)** para evitar condições de corrida.

Para a geração de números aleatórios, compartilhamos um único gerador (std::mt19937) entre todas as threads e protegemos o sorteio usando **#pragma omp critical**, evitando assim acessos simultâneos ao gerador.

Reflexão sobre a Geração de Números Aleatórios

A geração de números aleatórios pode ser um obstáculo em um ambiente paralelo porque o acesso simultâneo ao gerador por múltiplas threads pode causar condições de corrida e resultados incorretos. Proteger o gerador com uma seção crítica garante a correção, mas introduz um gargalo de desempenho, já que apenas uma thread pode acessar o gerador por vez.

Resultados

- Valor médio estimado de pi: 3.1417
- Tempo médio de execução: aproximadamente 2.6 segundos

Impacto no Desempenho

A solução adotada, embora correta, impactou negativamente o desempenho do código. O uso de **omp critical** criou um ponto de serialização no código, eliminando os benefícios da paralelização na geração de números aleatórios. Como resultado, a versão paralela ficou mais lenta que a sequencial.

Parte 3: Melhorando a Paralelização

Descrição das Melhorias

Para melhorar a paralelização, atribuímos a cada thread seu próprio gerador de números aleatórios. Inicializamos o gerador de cada thread com uma semente única, combinando o tempo atual com o número da thread (omp_get_thread_num()). Dessa forma, eliminamos a necessidade de proteger o gerador com omp critical, permitindo que as threads operem de forma independente.

Reflexão sobre as Mudanças

A mudança permitiu paralelizar efetivamente a geração de números aleatórios. Cada thread passou a gerar seus próprios números sem interferir nas demais, eliminando o gargalo causado pela seção crítica. Observamos que o valor estimado de pi não sofreu alterações significativas, mantendo a precisão.

Resultados

- Valor médio estimado de pi: 3.1411
- Tempo médio de execução: aproximadamente 0.19 segundos

Melhoria no Tempo de Execução

Houve uma melhoria significativa no tempo de execução em comparação com as versões anteriores. A eliminação da seção crítica e a independência das threads permitiram aproveitar plenamente o paralelismo, resultando em um desempenho muito superior.

Conclusão e Comparação

Tabela Comparativa dos Resultados

Versão	Valor médio estimado de pi	Tempo médio de execução (s)
Sequencial	3.1417	2.0
Paralelização Inicial	3.1417	2.6
Paralelização Melhorada	3.1411	0.19

Respostas às Perguntas

1. **Houve uma melhoria significativa no tempo de execução entre a versão sequencial e as versões paralelas?**

Sim, houve uma melhoria significativa no tempo de execução na segunda versão paralela. Enquanto a primeira tentativa de paralelização resultou em um tempo maior devido ao uso de omp critical, a melhoria na paralelização permitiu reduzir o tempo de execução de aproximadamente 2.0 segundos para 0.19 segundos, aproveitando eficientemente o uso de múltiplas threads.

2. **A estimativa de pi permaneceu precisa em todas as versões?**

Sim, a estimativa de pi manteve-se precisa em todas as versões. Houve pequenas variações nos valores estimados devido à natureza aleatória do método de Monte Carlo, mas todos os valores estavam próximos do valor real de pi, demonstrando a consistência dos resultados.

3. **Quais foram os maiores desafios ao paralelizar o algoritmo, especialmente em relação aos números aleatórios?**

O maior desafio foi lidar com a geração de números aleatórios em um ambiente paralelo. O uso de um gerador compartilhado entre threads introduziu condições de corrida e a necessidade de sincronização com omp critical, o que prejudicou o desempenho. A solução foi criar geradores independentes para cada thread com sementes diferentes, eliminando a necessidade de sincronização e permitindo a geração eficiente de números aleatórios em paralelo.

4. **O uso de threads trouxe benefícios claros para este problema específico?**

Sim, o uso de threads trouxe benefícios claros na versão com a paralelização melhorada. Ao eliminar os gargalos e permitir que cada thread operasse independentemente, conseguimos reduzir significativamente o tempo de execução. Isso demonstra que, quando bem implementada, a paralelização pode oferecer vantagens substanciais em termos de desempenho para este tipo de problema computacionalmente intensivo.

Considerações Finais

A atividade permitiu explorar os desafios e benefícios da paralelização em algoritmos que dependem de geração de números aleatórios. Através das implementações, ficou evidente a importância de considerar cuidadosamente a forma como os recursos são compartilhados entre threads para evitar impactos negativos no desempenho. A solução encontrada não só melhorou o tempo de execução como também manteve a precisão dos resultados, evidenciando o potencial da programação paralela quando aplicada corretamente.