# Relatório Atividade 14 - SuperComputação

Pedro Drumond

## Parte 1: Implementação Sequencial

**Valor estimado de Pi: 3.13776**
**Tempo de execução: 0.003266 segundos**

**Geração de Números Aleatórios**:

- A função rand() foi usada para gerar os números aleatórios, e srand(time(0)) foi utilizado para garantir que a sequência de números aleatórios seja diferente a cada execução.
- O valor de Pi foi estimado com uma precisão razoável, mesmo para um número relativamente pequeno de pontos (N=100000). A precisão da estimativa pode ser melhorada aumentando o valor de N.

**Dificuldades**:

- A implementação não apresentou grandes dificuldades, já que o algoritmo é simples e a geração de números aleatórios com rand() é eficiente. No entanto, para grandes valores de N, o tempo de execução aumenta linearmente, o que pode ser uma limitação em sistemas com capacidade de processamento limitada.

## Parte 2: Primeira Tentativa de Paralelização

**Valor estimado de Pi: 3.13648**
**Tempo de execução: 0.0286079188808799 segundos**

**Geração de Números Aleatórios em Ambiente Paralelo:**

- A função rand() gera a mesma sequência de números em todas as threads quando é chamada simultaneamente em um ambiente paralelo, pois utiliza um único estado global. Isso pode afetar a precisão dos resultados.
- A solução para esse problema é usar geradores de números aleatórios independentes para cada thread. A função rand_r(), que recebe uma semente como argumento, poderia ser uma solução melhor, já que é reentrante e thread-safe.

**Impacto no Desempenho:**

- Antes, com o uso de rand(), várias threads estavam competindo por acesso ao gerador de números aleatórios global, o que gerava contenção e um gargalo no desempenho. Como resultado, o tempo de execução da versão paralela era maior que o esperado.
- Com o uso de rand_r(), cada thread possui seu próprio gerador de números aleatórios, o que elimina a contenção e permite que as threads operem de forma independente. Isso resultou em um tempo de execução significativamente menor.

**Melhoria no Tempo de Execução**:

- O tempo de execução caiu drasticamente para 0.000601 segundos, em comparação com os resultados anteriores, onde a paralelização com rand() levou cerca de 0.0286 segundos. Isso mostra o quanto a escolha de um gerador de números aleatórios adequado pode impactar o desempenho em um ambiente paralelo.
- A versão sequencial levava cerca de 0.0033 segundos, então a versão paralela com rand_r() é agora muito mais rápida do que a versão sequencial, especialmente em um cenário onde o número de pontos aumentaria significativamente.

## Parte 3: Melhorando a Paralelização

**Valor estimado de Pi: 3.14676**
**Tempo de execução : 0.00544430036097765 segundos**

Na Parte 3 da atividade, a paralelização do algoritmo de Monte Carlo foi aprimorada ao introduzir geradores de números aleatórios independentes para cada thread, utilizando o Mersenne Twister (std::mt19937) e a distribuição uniforme (std::uniform_real_distribution). Essa abordagem, teoricamente, deveria garantir uma geração de números aleatórios de alta qualidade, sem contenção entre as threads, resolvendo os problemas associados ao uso de um gerador compartilhado.

Contudo, ao analisar os resultados comparando com a Parte 2, observamos que essa mudança não teve o efeito esperado em termos de desempenho.

Geração de Números Aleatórios: Paralelizada de Forma Eficaz?

Sim, a geração de números aleatórios foi paralelizada de maneira eficaz no sentido de que:

1. Cada thread teve seu próprio gerador de números aleatórios. Isso eliminou qualquer possibilidade de contenção entre as threads, que era uma limitação na abordagem sequencial com rand().

2. A utilização de std::mt19937 garantiu que os números aleatórios gerados por cada thread fossem de alta qualidade e independentes entre si, evitando repetições ou padrões indesejados que poderiam ocorrer em um ambiente compartilhado.

No entanto, apesar da eficiência na paralelização, o custo computacional de usar um gerador mais sofisticado introduziu uma sobrecarga significativa que impactou negativamente o desempenho.

**Mudança no Valor de Pi:**

O valor de Pi estimado na Parte 3 (com std::mt19937) foi 3.14676, enquanto na Parte 2 (com rand_r()) foi 3.14712. Essa diferença é muito pequena e está dentro das variações esperadas para o método de Monte Carlo, que depende da aleatoriedade.

Portanto, a mudança no valor de Pi foi insignificante, sugerindo que, para este problema específico, o uso de um gerador de números aleatórios mais sofisticado como o std::mt19937 não trouxe uma melhoria significativa na precisão.

**Impacto no Tempo de Execução:**

Ao contrário do que seria esperado, o tempo de execução piorou significativamente com a mudança:

- Parte 2 (com rand_r()): 0.0006 segundos
- Parte 3 (com std::mt19937): 0.0054 segundos

A diferença de tempo foi substancial. A introdução de std::mt19937 e std::uniform_real_distribution aumentou a complexidade da geração de números aleatórios, resultando em uma sobreposição de custos computacionais que superou os benefícios da paralelização. Isso significa que, apesar de a paralelização ter sido eficaz no sentido de eliminar a contenção, a eficiência foi prejudicada pela maior sofisticação do gerador de números aleatórios utilizado.

Observações:

1. Trade-off entre Qualidade e Desempenho:
    - Enquanto std::mt19937 é conhecido por gerar números pseudo-aleatórios de alta qualidade, essa melhoria não trouxe uma vantagem significativa para o problema em questão, pois o método de Monte Carlo para calcular Pi não requer números aleatórios de altíssima precisão para obter uma estimativa precisa.
    - A função rand_r(), embora menos sofisticada, foi mais rápida e suficientemente precisa para estimar o valor de Pi com uma margem de erro aceitável.

2. Simplicidade vs. Sofisticação:
    - Em problemas como este, onde a quantidade de números aleatórios gerados é alta, mas a precisão dos números não precisa ser absoluta, simples é melhor. A escolha de rand_r() se mostrou mais adequada por sua simplicidade e velocidade, enquanto std::mt19937 introduziu uma sobrecarga que não foi compensada com uma melhoria notável nos resultados.
3. Eficácia da Paralelização:
    - Embora a paralelização tenha sido implementada de forma eficaz em termos de distribuição de trabalho entre threads e geração independente de números aleatórios, o impacto final no desempenho foi negativo devido à maior complexidade do gerador de números aleatórios utilizado. Em ambientes paralelos, é importante equilibrar a qualidade com o custo computacional, e nesse caso, o uso de rand_r() foi a opção mais eficiente.

**Conclusão:**

| Versão | Valor Estimado de Pi | Tempo de Execução (seg) |
|---|---|---|
| Parte 1 (Sequencial) | 3.13766 | 0.003266 |
| Parte 2 (Paralelo com rand_r()) | 3.14712 | 0.000601 |
| Parte 3 (Paralelo com std::mt19937) | 3.14676 | 0.005444 |

**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 Parte 2, onde o uso de rand_r() como gerador de números aleatórios permitiu uma grande redução no tempo de execução. A versão paralela (Parte 2) foi mais de 5 vezes mais rápida que a versão sequencial (Parte 1). No entanto, a Parte 3, apesar de utilizar um gerador de números aleatórios de maior qualidade (std::mt19937), resultou em um tempo de execução maior que a versão sequencial, devido à sobrecarga computacional causada pelo gerador mais complexo.

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

Sim, a estimativa de Pi permaneceu precisa em todas as versões, com pequenas variações normais para o método de Monte Carlo, que depende da aleatoriedade dos pontos gerados. A variação entre as versões foi pequena (em torno de 0,01), o que está dentro do esperado para um cálculo com 100.000 pontos

aleatórios. Mesmo com diferentes abordagens para a geração de números aleatórios, o valor de Pi se manteve consistente.

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

O maior desafio foi garantir que as threads gerassem números aleatórios de forma independente, sem conflitos ou contenção, e sem sacrificar o desempenho. Na Parte 1, a geração de números aleatórios era simples com rand(), mas não apropriada para paralelização. Ao usar rand_r() na Parte 2, conseguimos evitar a contenção com um gerador rápido e thread-safe, resultando em melhor desempenho. No entanto, o uso de um gerador de alta qualidade como std::mt19937 na Parte 3 introduziu uma sobrecarga considerável, mostrando que, embora seja possível gerar números aleatórios de alta qualidade em paralelo, isso pode aumentar o custo computacional sem trazer ganhos significativos para o problema específico.

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

Sim, o uso de threads trouxe benefícios claros na Parte 2, onde a paralelização do sorteio de pontos reduziu drasticamente o tempo de execução. No entanto, a Parte 3 mostrou que o paralelismo pode trazer sobrecargas se forem usados componentes computacionalmente caros, como geradores de números aleatórios mais complexos. O desempenho máximo foi alcançado quando usamos um gerador de números aleatórios eficiente em termos de paralelismo (rand_r()), que permitiu que as threads trabalhassem de maneira independente e eficiente, sem a sobrecarga que ocorreu ao utilizar std::mt19937.