Cálculo de Pi pelo Método de Monte Carlo

Relatório de Trabalho Prático

Grupo 6:
Álison Christian R. V. de Carvalho
Andrei Rezende Ono
Antônio Marcos Fontes Darienco
George Silva De Oliveira
João Pedro Correia Leite Moreira
Pablo Alonso Latapiat de Freitas
Rodolfo Ferreira Sapateiro

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Sistemas Distribuídos – 2025/01

Sumário

1. Introdução ao Problema

Os Métodos de Monte Carlo são uma ampla classe de algoritmos computacionais que utilizam amostragem aleatória repetida para obter resultados numéricos. A ideia central é usar a aleatoriedade para resolver problemas que, deterministicamente, poderiam ser muito complexos ou impossíveis de resolver. Essa técnica é amplamente aplicada em áreas como finanças (para modelar riscos), física (para simular sistemas de partículas) e computação gráfica (para renderizar imagens realistas).

Neste projeto, o método foi utilizado para um problema clássico: a estimativa do valor de π (pi). A lógica pode ser facilmente compreendida através de uma analogia com um alvo de dardos. Imagine um alvo quadrado e um círculo perfeitamente inscrito dentro dele. Se uma pessoa, sem mirar, arremessasse milhares de dardos aleatoriamente neste alvo, alguns cairiam dentro do círculo e outros na área do quadrado, mas fora do círculo.

Intuitivamente, a proporção de dardos que caem dentro do círculo em relação ao total de dardos arremessados será aproximadamente igual à proporção da área do círculo em relação à área do quadrado. Podemos traduzir isso matematicamente:

- `Área do Círculo / Área do Quadrado ≈ Dardos dentro do Círculo / Total de Dardos`

Para simplificar o cálculo, utilizamos um quadrado de lado 1 no primeiro quadrante do plano cartesiano, com um quarto de círculo de raio 1 inscrito nele. Geramos pontos com coordenadas (x, y) aleatórias, onde $0 \leq x < 1$ e $0 \leq y < 1$. Para cada ponto, verificamos se ele está dentro do quarto de círculo usando a equação do círculo: se $x^2 + y^2 \leq 1$, o ponto está dentro.

A relação das áreas nos dá a fórmula final:

- $(\text{Área do Quarto de Círculo}) / (\text{Área do Quadrado}) = (\pi * 1^2) / 4 / 1^2 = \pi / 4$

Portanto, podemos reorganizar a proporção para estimar o valor de Pi:

- $\pi \approx 4 * (\text{Número de pontos dentro do círculo} / \text{Número total de pontos})$

De acordo com a Lei dos Grandes Números, a precisão dessa estimativa aumenta significativamente conforme o número de pontos simulados cresce, tornando este um problema computacionalmente intensivo e ideal para a aplicação de técnicas de paralelismo.

2. Configuração da Máquina de Execução

Os testes para todas as soluções serão executados no seguinte ambiente de hardware:

- **Dispositivo:** Notebook Acer Nitro 5
- **Processador:** Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz
- **Memória RAM:** 16,0 GB
- **Placa de Vídeo:** NVIDIA GeForce GTX 1650 (4 GB)
- **Sistema Operacional:** Windows 10, 64 bits

3. Recursos do Projeto

Todos os materiais relacionados a este projeto, incluindo o código-fonte e as apresentações em vídeo, podem ser acessados nos seguintes links:

- Repositório do Código-Fonte (GitHub)
- Apresentações em Vídeo (Google Drive)

4. Análise das Soluções Desenvolvidas

A seguir, uma breve descrição de cada uma das três abordagens implementadas para a resolução do problema.

4.1. Solução Sequencial

Esta é a implementação fundamental que serve como linha de base para a comparação de desempenho. O algoritmo opera em uma única thread, executando um laço que, para cada iteração, gera um ponto aleatório e verifica se ele pertence ao círculo. Todo o processamento é feito de forma linear, do início ao fim.

4.2. Solução Paralela

Para explorar os recursos de processadores multi-core, a solução paralela foi desenvolvida utilizando o módulo `threading` do Python. A carga de trabalho total (o número de pontos a serem simulados) é dividida entre um número pré-definido de threads. Cada thread executa o cálculo de Monte Carlo para sua porção dos dados concorrentemente. Para garantir a integridade do resultado final, um mecanismo de `lock` é utilizado para sincronizar o acesso à variável global que acumula a contagem de pontos dentro do círculo, evitando condições de corrida.

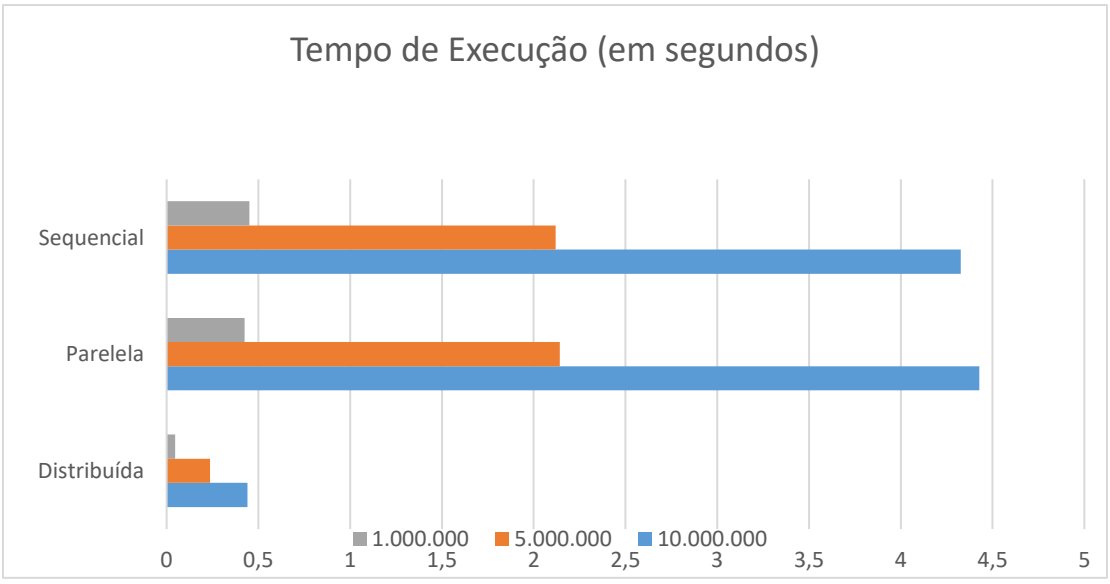
4.3. Solução Distribuída

A abordagem distribuída eleva o paralelismo para além de uma única máquina, utilizando uma arquitetura cliente-servidor baseada em `sockets`. O servidor é responsável por orquestrar o trabalho: ele aguarda a conexão de um número definido de clientes, divide a tarefa total e envia a cada cliente a sua respectiva carga de trabalho. Cada cliente processa sua tarefa de forma independente e retorna o resultado parcial para o servidor. O servidor, por sua vez, utiliza threads para gerenciar a comunicação com múltiplos clientes simultaneamente e, ao final, agrega todos os resultados parciais para calcular a estimativa final de π.

5. Análise Comparativa de Desempenho

5.1. Gráfico de Tempos de Execução

O gráfico abaixo irá comparar os tempos de execução obtidos para cada solução com diferentes cargas de trabalho (número de pontos).



5.2. Tabela de Estimativas de Pi

A tabela abaixo irá comparar as estimativas de Pi obtidas para cada solução com diferentes cargas de trabalho (número de pontos).

Número de Pontos	Solução Sequencial	Solução Paralela	Solução Distribuída
1.000.000	3,1421880	3,1411920	3,1419240
5.000.000	3,1407592	3,1417912	3,1418696
10.000.000	3,1427664	3,1423972	3,1413876

6. Análise de Escalabilidade, Gargalos e Melhorias

6.1. Solução Sequencial

- **Escalabilidade e Eficiência:** Esta abordagem é inerentemente não escalável. Sua principal limitação é o tempo de execução, que cresce de forma linear com o aumento do número de pontos, sem a possibilidade de otimização por meio de hardware paralelo.
- **Gargalos Identificados:** O gargalo fundamental é sua natureza `single-threaded`, que impede o aproveitamento de processadores com múltiplos núcleos.

6.2. Solução Paralela

- **Escalabilidade e Eficiência:** A solução demonstra uma boa escalabilidade em relação à carga de trabalho, com o tempo de execução crescendo de forma previsível e linear conforme o número de pontos aumenta. Isso indica que a estratégia de divisão de trabalho entre threads é eficaz.
- Gargalos Identificados:
    - **Global Interpreter Lock (GIL):** O principal gargalo do Python, o GIL impede a execução simultânea de múltiplas threads em tarefas intensivas de CPU. Na prática, as threads alternam o uso da CPU em vez de operarem em paralelo, o que limita o ganho em ambientes multi-core.
    - **Sincronização:** O uso do `threading.Lock` para a atualização do contador global, embora necessário, introduz um ponto de serialização no código, onde as threads podem precisar esperar umas pelas outras.
- **Proposta de Melhoria:** Para obter um paralelismo real e mitigar o impacto do GIL, a principal melhoria seria utilizar o módulo `multiprocessing`, que permite o uso pleno de múltiplos núcleos de CPU ao criar processos independentes.

6.3. Solução Distribuída

- **Escalabilidade e Eficiência:** Esta arquitetura apresenta a maior escalabilidade (horizontal), pois o desempenho pode ser melhorado adicionando mais nós (clientes) à rede. É a abordagem mais adequada para problemas de grande escala.
- **Gargalos Identificados:**
    - **Comunicação de Rede:** A latência da rede e a sobrecarga para estabelecer conexões via sockets, enviar e receber dados são os principais gargalos de desempenho.
    - **Serialização de Dados:** A conversão de dados entre formatos numéricos e bytes para a transmissão em rede consome ciclos de CPU e adiciona sobrecarga.

7. Desafios Enfrentados e Soluções Encontradas

Desafio na Solução Sequencial: Lentidão em Grandes Volumes

- **Problema:** O tempo de execução da abordagem sequencial torna-se proibitivo para a grande quantidade de pontos necessária para uma alta precisão.
- **Solução Encontrada:** A limitação de desempenho motivou a exploração de novas arquiteturas, resultando no desenvolvimento das versões paralela e distribuída para paralelizar o cômputo.

Desafio na Solução Paralela: Condições de Corrida

- **Problema:** Com múltiplas threads atualizando um contador compartilhado simultaneamente, havia o risco de *race conditions*, que poderiam corromper o resultado final.
- **Solução Encontrada:** Foi utilizado um `threading.Lock` para garantir a exclusão mútua. Ao proteger a operação de escrita na variável global com um `lock`, assegurou-se que apenas uma thread pudesse modificar o contador por vez, garantindo a atomicidade da operação.

Desafio na Solução Distribuída: Orquestração e Comunicação

- **Problema:** Era necessário desenvolver um método para o servidor gerenciar múltiplos clientes de forma concorrente e agregar os resultados apenas após todos eles terem finalizado suas tarefas.
- **Solução Encontrada:** A orquestração foi feita com `threading` no servidor, onde cada conexão de cliente foi tratada em sua própria thread. O uso de `thread.join()` no fluxo principal garantiu que o servidor aguardasse a conclusão de todas as threads antes de finalizar o cálculo.

8. Metodologia e Ferramentas Utilizadas

Para o desenvolvimento deste trabalho, além das bibliotecas padrão do Python (`threading`, `socket`), foi utilizado o auxílio do modelo de linguagem **Gemini, do Google**. A ferramenta foi empregada como uma assistente de programação para as seguintes finalidades:

- **Refatoração e Correção:** Auxílio na revisão e correção de bugs nos códigos das três implementações (sequencial, paralela e distribuída).
- **Otimização:** Sugestão de melhorias na estrutura do código, como a distribuição mais eficiente dos pontos restantes na solução paralela.

O uso da ferramenta foi supervisionado, com todas as sugestões sendo validadas e adaptadas pela equipe para garantir que os objetivos do trabalho fossem atingidos.

9. Considerações Finais

O presente trabalho cumpriu com sucesso seu objetivo de implementar e avaliar três abordagens distintas — sequencial, paralela e distribuída — para a resolução do problema computacionalmente intensivo do cálculo de Pi pelo método de Monte Carlo. As análises de desempenho demonstraram de forma clara e quantitativa a superioridade das soluções que exploram o paralelismo.

Os resultados confirmaram a hierarquia de desempenho esperada: a solução distribuída apresentou a maior eficiência, com uma redução drástica no tempo de execução, seguida pela solução paralela, que também ofereceu ganhos significativos sobre a implementação sequencial. Ficou evidente que, para problemas que podem ser divididos em tarefas independentes ("embaraçosamente paralelos"), a distribuição da carga de trabalho é uma estratégia extremamente eficaz para otimização de tempo.

Este estudo prático também destacou os importantes trade-offs entre desempenho e complexidade. Enquanto a abordagem distribuída foi a mais rápida, ela também introduziu uma maior complexidade de implementação, envolvendo a gestão de comunicação de rede, sockets e a orquestração entre múltiplos processos. A solução paralela, por sua vez, representou um excelente equilíbrio, oferecendo uma melhoria de desempenho substancial com uma complexidade de código moderada, embora limitada por fatores como o GIL do Python.

Conclui-se que a escolha da arquitetura ideal depende diretamente dos requisitos do problema, dos recursos disponíveis e da escala da computação desejada. O projeto proporcionou uma valiosa experiência prática na aplicação de conceitos fundamentais de concorrência e distribuição, como o uso de threads, locks para evitar condições de corrida, e a comunicação via sockets, solidificando o entendimento teórico da disciplina.