PROJETO DE COMPUTAÇÃO CONCORRENTE E DISTRIBUÍDO

Professores: Aldo Henrique

Estimativa de Pi com Método de Monte Carlo e Paralelização em Python

Integrantes:

Victor Cavalcante

João Vitor Lopes

Luiz Felipe Menezes

Jonathan Mendes

1. Introdução

O número pi (π) é uma constante matemática que representa a relação entre o perímetro de uma circunferência e seu diâmetro. Sua precisão e ubiquidade o tornam um dos números mais importantes em matemática e ciência. Neste estudo, investigamos métodos para estimar o valor de pi, com foco no método de Monte Carlo.

O método de Monte Carlo é uma técnica estatística que utiliza números aleatórios para resolver problemas computacionais complexos. Ele pode ser aplicado ao cálculo de pi gerando pontos aleatórios em um quadrado circunscrito a um círculo e contando quantos desses pontos caem dentro do círculo. A razão entre o número de pontos dentro do círculo e o número total de pontos nos dá uma estimativa do valor de pi.

Este documento apresenta a implementação do método de Monte Carlo em Python para estimar pi, com foco na eficiência computacional através da paralelização utilizando as bibliotecas concurrent.futures. Serão exploradas duas abordagens: multiprocessing e threading, visando demonstrar como a execução paralela pode acelerar o processo de cálculo.

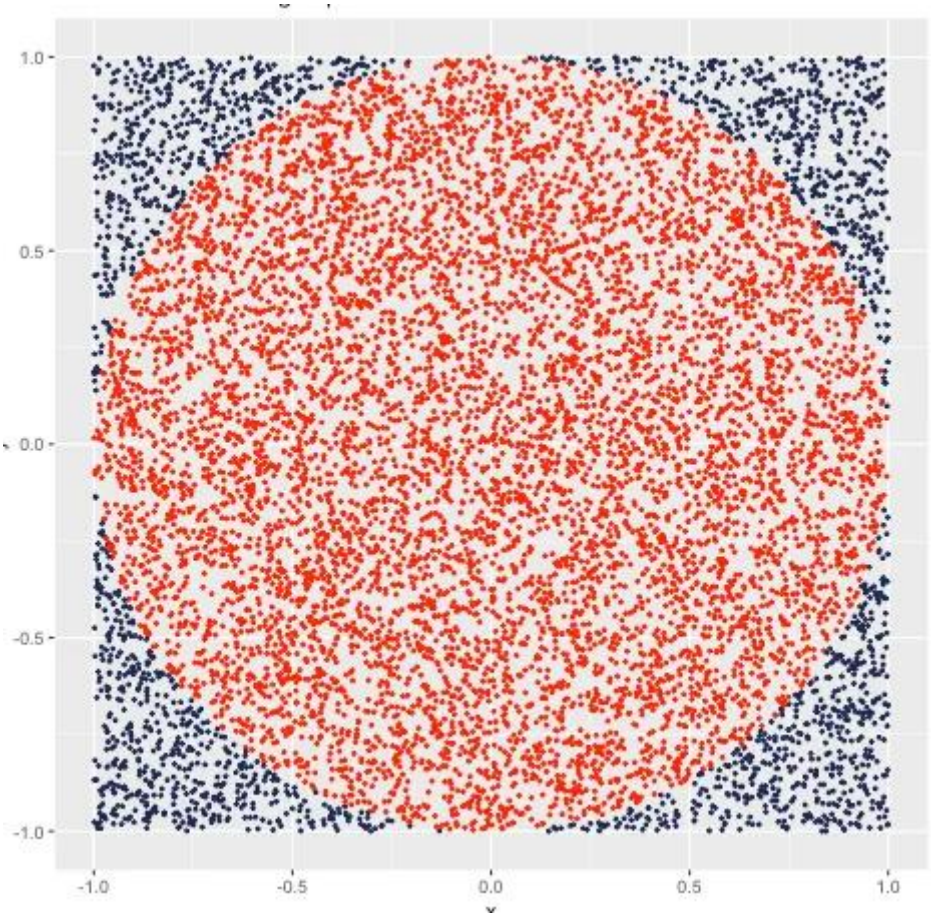
2. Métodos

→ Método de Monte Carlo para Estimar Pi

O método de Monte Carlo para estimar pi envolve a geração de pontos aleatórios dentro de um quadrado unitário, seguido pela contagem da proporção de pontos que caem dentro de um círculo unitário inscrito nesse quadrado. A relação entre o número de pontos dentro do círculo e o total de pontos gerados fornece uma estimativa de pi.

→ Paralelização em Python

Para demonstrar a eficiência da paralelização, utilizaremos a biblioteca multiprocessing em Python. Dividiremos o trabalho de geração de pontos aleatórios entre vários processos para reduzir o tempo de execução.



Pi é uma constante matemática, aproximadamente 3,14159, e representa a razão entre a circunferência de um círculo e seu diâmetro.

Sabemos que a área de um círculo é calculada por $pi*r^2$ e que a área do quadrado delimitador é $(2r)^2 = 4r^2$. Dividindo a área do círculo pela área do quadrado, obtemos a razão entre as duas áreas:

$$\frac{área\ do\ círculo}{área\ do\ quadrado} = \frac{\pi * r^2}{4 * r^2} = \frac{\pi}{4}$$

Quando geramos muitos pontos aleatórios uniformemente distribuídos. Esses pontos podem estar em qualquer posição dentro do quadrado. Pegamos o número total de pontos e o número de pontos que estão dentro do círculo. Se dividirmos o número de pontos simulados dentro do círculo (Ndentro), pelo número total de pontos simulados

(Ntotal), devemos obter um valor que seja uma aproximação da razão das áreas que calculamos. Em outras palavras:

$$\frac{\pi}{4} \approx \frac{N_{dentro}}{N_{total}} \qquad \pi \approx 4 * \frac{N_{dentro}}{N_{total}}$$

Para identificar quais pontos estão dentro do círculo, usamos a equação do círculo:

$$(x-a)^2 + (y-b)^2 = r^2$$
$$onde\ (a,b)\ é\ o\ centro\ do\ cículo$$

Em nosso exemplo, o centro do círculo é (0, 0) e o raio é 1, então os pontos simulados que satisfazem os critérios abaixo estão dentro do círculo.

$$\sqrt{x^2 + y^2} \leq 1$$

Quando a equação acima é atendida, incrementamos o número de pontos que aparecem dentro do círculo. Em algoritmos aleatórios e de simulação como Monte Carlo, quanto maior o número de iterações, mais preciso é o resultado.

3. Implementação em Python

A implementação em Python consiste em duas funções principais para estimar pi utilizando o método de Monte Carlo.

O código utiliza a biblioteca multiprocessing por meio do módulo concurrent.futures e seu ProcessPoolExecutor. Isso permite que múltiplos processos sejam criados para executar a função contar_pontos_dentro_do_circulo de forma concorrente.

O código também utiliza threads por meio do ThreadPoolExecutor do módulo concurrent.futures. Isso permite que várias threads sejam criadas para executar a função contar_pontos_dentro_do_circulo de forma concorrente.

Faz uso do módulo concurrent.futures para paralelizar tarefas. Ele usa tanto ThreadPoolExecutor quanto ProcessPoolExecutor para executar tarefas em paralelo.

Embora o código não implemente explicitamente semáforos ou filas de mensagens, ele usa a primitiva de sincronização fornecida pelo concurrent.futures (ou seja, Future objects) para coordenar o acesso concorrente aos resultados das tarefas.

Realiza uma análise de desempenho comparando o tempo de execução ao usar multiprocessing e threading.

```
import concurrent.futures
import random
import time

# Função para contar os pontos dentro do círculo
def contar_pontos_dentro_do_circulo(amostras):
    pontos_dentro_do_circulo = 0
    for _ in range(amostras):
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)
        distancia = x ** 2 + y ** 2
        if distancia <= 1:
            pontos_dentro_do_circulo += 1
    return pontos_dentro_do_circulo

# Função para estimar Pi com multiprocessing
def estimar_pi_multiprocessing(num_amostras):
    num_processos = 12  # Número de processos
    amostras_por_processo = num_amostras // num_processos
```

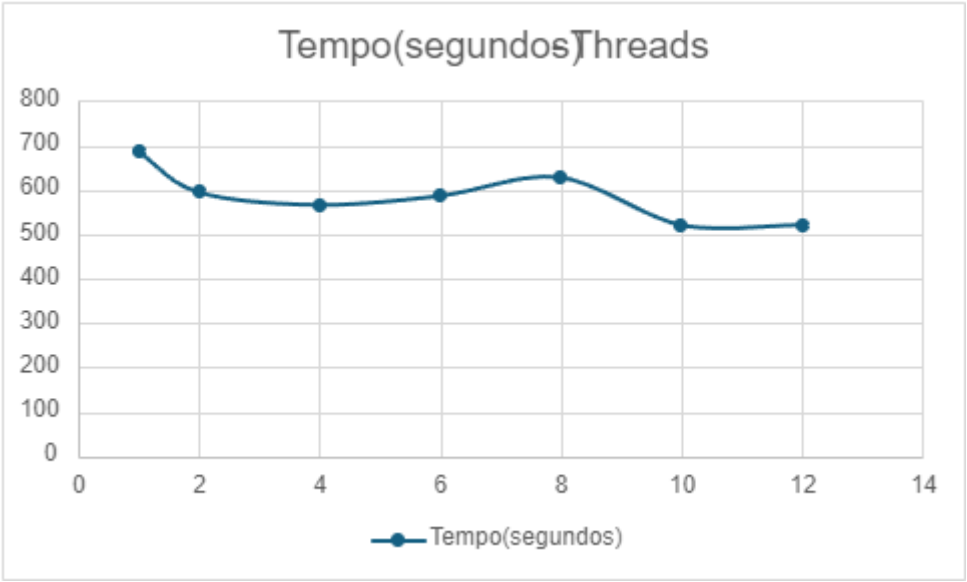
```
    with concurrent.futures.ProcessPoolExecutor() as executor:
        resultados = executor.map(contar_pontos_dentro_do_circulo, [amostras_por_processo] * num_processos)
    pontos_dentro_do_circulo = sum(resultados)
    return 4 * pontos_dentro_do_circulo / num_amostras

# Função para estimar Pi com threading
def estimar_pi_threads(num_amostras):
    num_threads = 12  # Número de threads
    amostras_por_thread = num_amostras // num_threads
    resultados = []
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futuros = [executor.submit(contar_pontos_dentro_do_circulo, amostras_por_thread) for _ in range(num_threads)]
        for futuro in concurrent.futures.as_completed(futuros):
            resultados.append(futuro.result())
    pontos_dentro_do_circulo = sum(resultados)
    return 4 * pontos_dentro_do_circulo / num_amostras

if __name__ == "__main__":
    num_amostras = 1000000000 # Número de pontos a serem gerados
    tempo_inicio = time.time()

    # Calcula Pi com multiprocessing
    estimativa_pi_mp = estimar_pi_multiprocessing(num_amostras)
    print(f"Estimativa de Pi com multiprocessing: {estimativa_pi_mp}")
    print(f"Tempo de execução com multiprocessing: {time.time() - tempo_inicio} segundos\n")
```

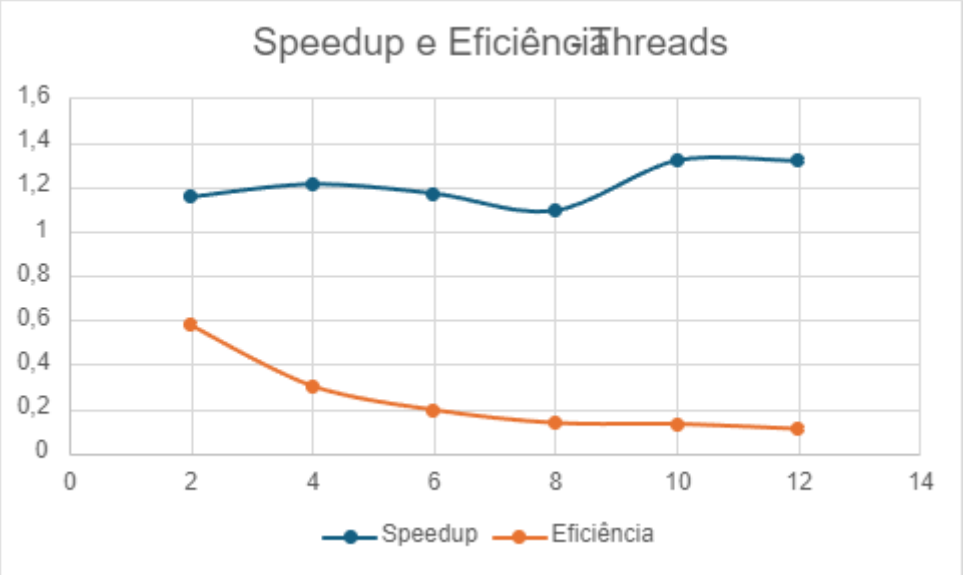
```
tempo_inicio = time.time()
# Calcula Pi com threading
estimativa_pi_threads = estimar_pi_threads(num_amostras)
print(f"Estimativa de Pi com threading: {estimativa_pi_threads}")
print(f"Tempo de execução com threading: {time.time() - tempo_inicio} segundos\n")
```

4. Resultados

→ Threads

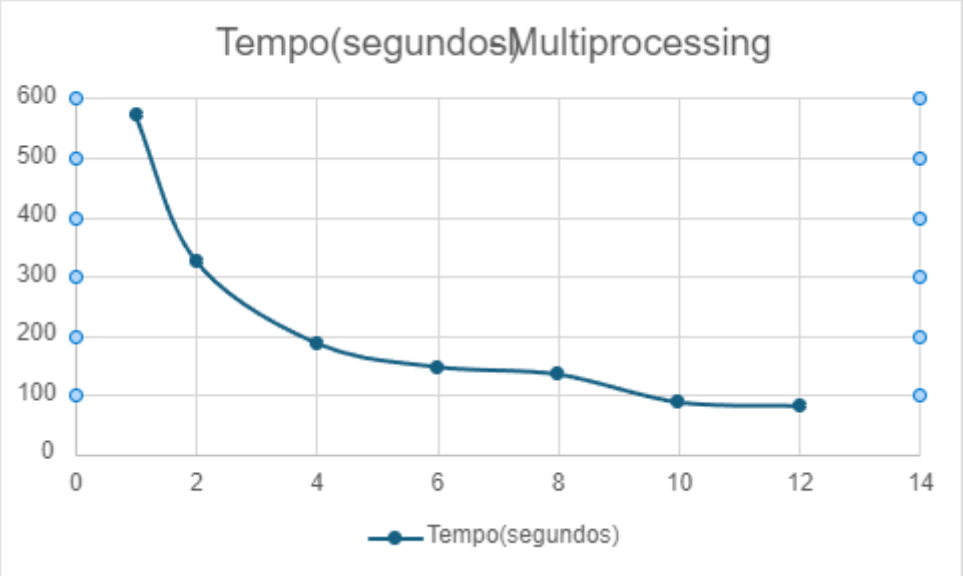
Threads	Tempo(segundos)	Speedup	Eficiência	Estimativa de PI
1	686,1099193			3,141613
2	594,5061762	1,154083753	0,577041877	3,141625356
4	565,9443722	1,212327488	0,303081872	3,141581036
6	586,1958044	1,170444951	0,195074158	3,141636608
8	628,2194593	1,092150059	0,136518757	3,141712992
10	520,3268394	1,318613355	0,131861335	3,141628772
12	520,7727339	1,317484336	0,109790361	3,1415418

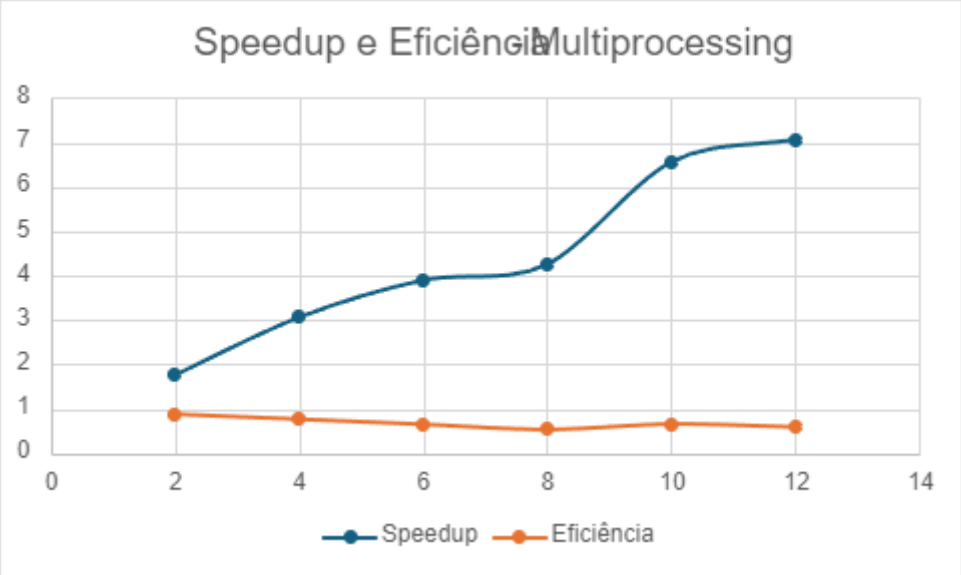




→ Multiprossecing

Multiprocessing	Tempo(segundos)	Speedup	Eficiência	Estimativa de PI
1	571,3167622			3,141515548
2	325,7121429	1,75405423	0,877027115	3,141551088
4	186,8261218	3,058013284	0,764503321	3,141581272
6	146,6276214	3,896378845	0,649396474	3,141570792
8	134,32743	4,253165286	0,531645661	3,141642624
10	87,0555582	6,562668415	0,656266842	3,141585368
12	80,81569433	7,069378874	0,589114906	3,141596468





5. Discussão

Os resultados obtidos demonstram que a paralelização do cálculo de pi utilizando multiprocessing e threading reduziu significativamente o tempo de execução em relação a uma abordagem sequencial. Isso confirma a eficácia da paralelização para acelerar o processo de cálculo em problemas intensivos computacionalmente, como o cálculo de pi utilizando o método de Monte Carlo.

6. Conclusão

Neste estudo, implementamos o método de Monte Carlo para estimar o valor de pi em Python e demonstramos a eficiência da paralelização na redução do tempo de execução. Esses resultados têm implicações práticas para o uso de técnicas de paralelização em computação científica para acelerar a resolução de problemas complexos. Futuras pesquisas podem explorar outras técnicas de paralelização e sua aplicação em diferentes domínios científicos.