UNIVERSIDADE FEDERAL DE SÃO CARLOS
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

Arquiteturas de Alto Desempenho

Estimativa de π por Monte Carlo utilizando MPI

Guilherme Gomes Arcencio - 769731 - Engenharia de Computação
Guilherme Silva Castro - 769763 - Ciência da Computação

1 Introdução

Neste experimento, foi utilizado o método de Monte Carlo para estimar o valor de π de duas formas diferentes: uma implementação totalmente sequencial e outra que lança mão de arquiteturas de memória distribuída utilizando MPI. Ao mensurar o tempo gasto em cada uma delas, foi possível encontrar experimentalmente o *speedup* ganho com o paralelismo distribuído.

2 Desenvolvimento teórico

O método de Monte Carlo consiste em utilizar uma grande quantidade de números gerados aleatoriamente para estimar um valor [1]. Para usá-lo para estimar o valor de π, considera-se a equação da área do círculo:

$$A = \pi r^2$$

e , considerando um círculo de raio unitário, obtém-se a relação

$$A = \pi \, .$$

Então, se A corresponder à área de apenas um quarto do círculo, tem-se

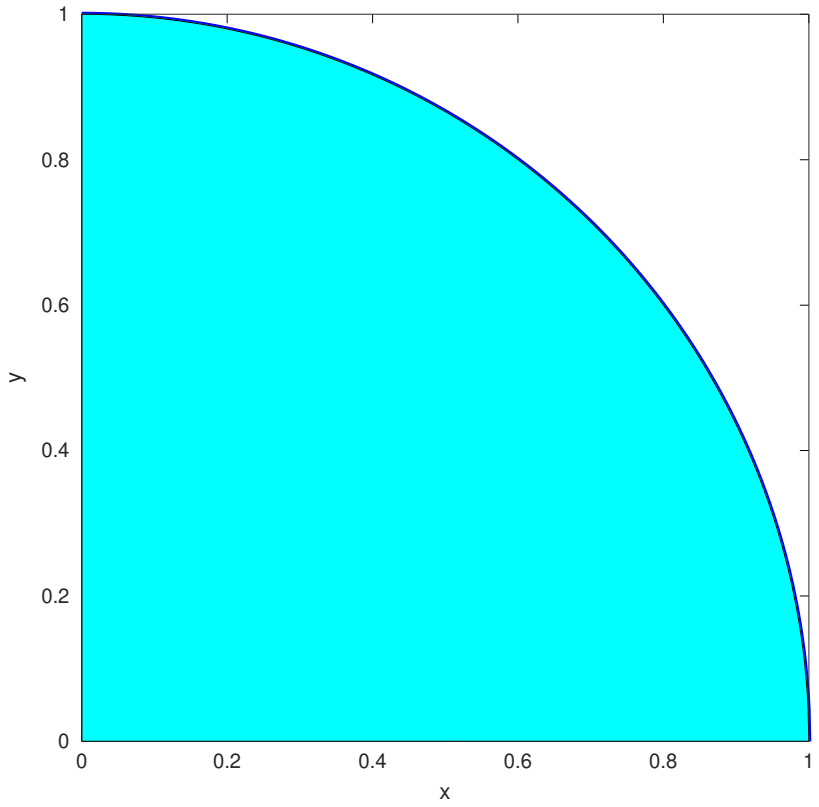
$$A = \frac{\pi}{4} \, .$$

Assim, caso seja possível encontrar ou estimar a área A de um único quadrante de um círculo de raio unitário, como mostra a Figura 1, será possível estimar π através da relação $\pi = 4A$. O uso do método de Monte Carlo está na estimativa da área A.

Ao escolher aleatoriamente um ponto (x, y) no quadrado que circunscreve a região circular, isto é, tal que $0 \leq x, y \leq 1$, a probabilidade de que ele se encontre dentro do círculo é $\frac{A}{A_{\text{quadrado}}} = A$, já que o quadrado tem lado e área 1. Ao gerar-se um número grande de pontos, espera que a razão entre os que estão dentro do círculo e a quantidade total aproxime-se cada vez mais da probabilidade mencionada.

Portanto, utilizando o método de Monte Carlo, geram-se N pontos e calcula-se quantos desses pontos satisfazem $x^2 + y^2 < 1$, construindo assim uma estimativa para a área $A \approx \frac{N_{x^2+y^2<1}}{N}$ e, por conseguinte, uma estimativa para $\pi \approx 4\frac{N_{x^2+y^2<1}}{N}$. Quanto maior o valor de N, mais precisa será a estimativa.

Figura 1: Região do primeiro quadrante do plano (sombreada) cuja área pode ser utilizada para estimar π.



3 Desenvolvimento prático

Foram testadas duas implementações da aplicação do método de Monte Carlo para a estimativa do valor de π. Para um mesmo valor de N, foram implementadas uma versão totalmente sequencial e uma versão paralela que simulava uma arquitetura de memória distribuída com MPI (*Message Passing Interface*). Ambas as versões foram implementadas em Python e seus desempenhos foram medidos através do tempo de execução de cada uma.

O código da versão sequencial é mostrado a seguir.

```
1 from random import random
2 from time import perf_counter
3
4 TOTAL_PONTOS = 100000000
5
6
7 def monte_carlo(num_pontos: int) -> int:
```

```
    pontos_dentro = 0
    for _ in range(num_pontos):
        x = random()
        y = random()
        # Verificando os pontos interiores
        if x**2 + y**2 < 1:
            pontos_dentro += 1

    return pontos_dentro


def main():
    start = perf_counter()

    results = monte_carlo(num_pontos=TOTAL_PONTOS)
    pi = 4 * (results / TOTAL_PONTOS)

    end = perf_counter()

    print(pi)
    print(end - start)


if __name__ == "__main__":
    main()
```

Já a implementação paralela, mostrada a seguir, utiliza uma implementação de MPI para simular quatro computadores em *cluster*.

```
from random import random
from time import perf_counter

from mpi4py import MPI

TOTAL_POINTS = 100000000
MASTER_NODE = 0

comm = MPI.COMM_WORLD
n_nodes = comm.Get_size()
```

```
11 rank = comm.Get_rank()
12
13
14 def monte_carlo(num_points: int) -> int:
15     pontos_dentro = 0
16     for _ in range(num_points):
17         x = random()
18         y = random()
19         # Verificando os pontos interiores
20         if x**2 + y**2 < 1:
21             pontos_dentro += 1
22
23     return pontos_dentro
24
25
26 def get_num_points() -> int:
27     points = TOTAL_POINTS // n_nodes
28     remainder = TOTAL_POINTS % n_nodes
29     if rank < remainder:
30         points += 1
31
32     return points
33
34
35 def main_master():
36     start = perf_counter()
37
38     points = get_num_points()
39     points_in = monte_carlo(points)
40     data = comm.gather(points_in, root=MASTER_NODE)
41
42     pi = 4.0 * sum(data) / TOTAL_POINTS
43
44     end = perf_counter()
45
46     print(pi)
47     print(end - start)
48
49
```

```
50 def main_worker():
51     points = get_num_points()
52     points_in = monte_carlo(points)
53     comm.gather(points_in, root=MASTER_NODE)
54 
55 
56 if __name__ == "__main__":
57     if rank == MASTER_NODE:
58         main_master()
59     else:
60         main_worker()
```

Cada implementação foi executada 5 vezes e a média de seus tempos de execução foram consideradas.

4 Discussão dos resultados

A partir dos resultados obtidos e mostrados na Tabela 1, observa-se o *speedup* alcançado pela implementação paralela. Ainda que esta simulasse uma arquitetura distribuída de 4 processadores, o *speedup* foi de apenas 2, o que indica a presença de *overheads* de comunicação.

Tabela 1: Comparação de tempo de execução e *speedup* entre as implementações sequencial e vetorial de Monte Carlo.

Tempo de execução sequencial (s)	Tempo de execução paralelo (s)	*Speedup*
15.6	7.4	2.1

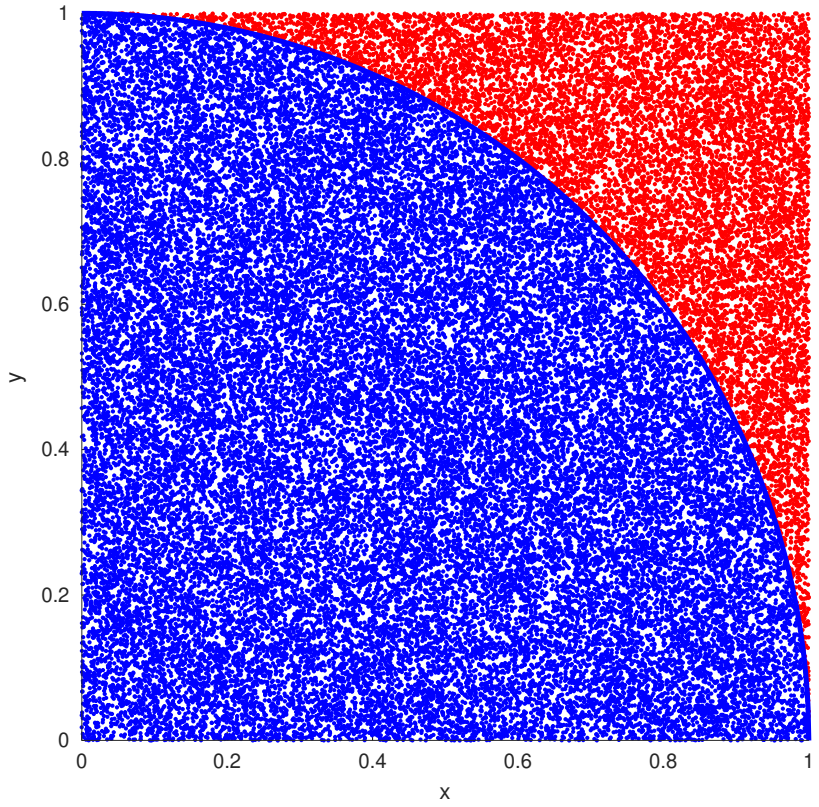
Uma visualização dos pontos aleatórios é mostrada na Figura 2. Nela, é possível observar que a proporção entre pontos dentro do círculo e pontos totais é muito próxima da proporção entre a área do círculo e a área do quadrado. O código, em Octave, utilizado para gerar a figura é mostrado a seguir.

```
1 TOTAL_PONTOS = 50000;
2 
3 % Geração e classificação dos pontos
4 coordenadas = rand(TOTAL_PONTOS, 2);
5 dentro_circulo = ...
6     coordenadas(:, 1).^2 + coordenadas(:, 2).^2 < 1;
7 fora_circulo = ...
```

```
    coordenadas(:, 1).^2 + coordenadas(:, 2).^2 >= 1;

coordenadas_dentro = coordenadas(dentro_circulo, :);
coordenadas_fora = coordenadas(fora_circulo, :);

x_circulo = linspace(0, 1, 1000);
y_circulo = sqrt(1 - x_circulo.^2);

figure;
% Pontos internos
scatter(
    coordenadas_dentro(:, 1),
    coordenadas_dentro(:, 2),
    "b", "filled"
)
hold on;

% Pontos externos
scatter(
    coordenadas_fora(:, 1),
    coordenadas_fora(:, 2),
    "r", "filled"
);

% Circunferência
plot(x_circulo, y_circulo, 'b', 'LineWidth', 10);

% Ajustes de tamanho
axis([0 1 0 1], "equal");
xlabel('x');
ylabel('y');

set(gcf, "units", "inches");
width = get(gcf, "Position")(3);
height = get(gcf, "Position")(4);

set(gcf, "paperunits", "inches");
set(gcf, "papersize", [width, height]);
```

```
46 print -dpdf 'plot.pdf';
```

Figura 2: Visualização dos pontos gerados aleatoriamente, sendo os azuis dentro do círculo e vermelhos fora.



5 Conclusões

Através da aplicação do método de Monte Carlo para estimar o valor de π foi possível observar a diferença de desempenho entre uma implementação totalmente sequencial e uma versão paralela que emula uma arquitetura de memória distribuída. A versão paralela, simulando 4 nós em 4 processadores obteve um speedup de 2, reduzindo o tempo de execução do método pela metade.

Referências

1 FELIPE, Henrique. **Calculando o valor de Pi via método de Monte Carlo**. 2018. Disponível em: <https://www.blogcyberini.com/2018/09/calculando-o-valor-de-pi-via-metodo-de-monte-carlo.html>. Acesso em: 5 abr. 2025.