UNIVERSIDADE AUTÓNOMA DE LISBOA

# Trabalho prático

Sistemas operativos

10 Junho 2022


Elementos do grupo:

Natalia Hurley Guerrero – 30006438

Daniel Filipe Costa Miguel – 30008914

Syed Hammad Ur Rehman Asghar – 30008767

Igor Gerson Gonçalves Paulo - 30009443


## INTRODUCCÃO

Neste trabalho, utilizamos a simulação de Monte Carlo, para obter aproximações do número pi, basado na equação pi = 4 x (Área dum círculo / área do quadrado). Geramos um número determinado de pontos, encerrados num quadrado, com o vértice no ponto (0,0).

## Relação entre funções:

- **main()** chama **simulate(threads, total_points)** 4 vezes cada vez com um par diferente de (threads, total_points).
- **simulate()** divide o trabalho entre threads criados com função **monte_carlo(props)** que gera e calcula numero de pontos dentro do circulo.
- **monte_carlo()** utiliza **point_in_circle()** para verifica ponto ser dentro do circulo.
- **simulate()** utiliza of **accuracy(aproximação_pi)** para avaliar a qualidade de aproximação.

## Descrição das funções:

### A funcão monte_carlo():

Para realizar a simulação de Monte Carlo para estimar o valor de pi, na função precisávamos de:

- **total_points**: Número de pontos aleatórios para gerar.
- **radius**: O raio do círculo

Um pointer para um inteiro: Onde vamos guardar o número de pontos dentro do círculo

Utilizamos total_points como o limite de repetição do ciclo em que gerar as coordenadas (x, y) aleatórios, x e y sendo entre -1(radius) e 1(radius) usando a função rand().

## points_in_circle():

### A função para verificação de coordenada ser dentro da região do círculo:

Utilizamos a condição $\boldsymbol{x^2 + y^2 \leq r^2}$ que representa todos os pontos dentro do círculo cujo origem seja em (0, 0).

Cada vez que esta condição for True para qualquer coordenada aleatória gerada, incrementamos o variável points_inside declarada no função monte_carlo por 1.

Afinal o valor do points_inside é guardado no endereço do pointer.

## A função simulate():

Criamos esta função para realizar uma simulação com threads dado:

- **threads**: O número de threads para usar na simulação.
- **total_points**: Numero de pontos aleatórios para gerar.
- **radius**: O raio do círculo.

Começamos por declarar

- O array de **Ids dos threads**.
- O array de **points_inside** para guardar os resulados de cada simulação feito por cada thread.
- O array de **struct props**, a struct que usamos para passar a informação para a função dos threads.

Usamos a função monte_carlo (descrita em cima) como a função de cada thread.

O **struct props** conte os variáveis necessário para o thread function monte_carlo:

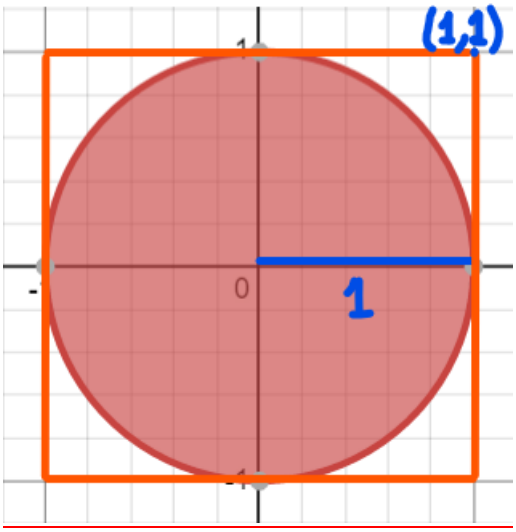
- total_points (para um thread): para cada thread, vai ser total_points (do simulação)/threads(numero de threads)
- Um pointer para uma posição do array points_inside: por exemplo, &points_inside[1]

- O radius

Um loop é usada para criar o número dado de threads, usando a função monte_carlo como função de cada thread e o respetivo struct props é passado como parâmetro da função.

Depois de acabarem de executar todos os threads, temos os resultados do monte_carlo do cada thread guardado no array points_inside.
Somamos o array points inside para obter total_points_inside.
Afinal calculamos a aproximação do pi como: 4 * (total_points_inside)/total_points
E imprimimos os resulados com analise.

DESENVOLVIMENTO




Definimos a variável "r" como o raio do círculo (neste caso vai ser uma unidade) e a variável "a" é o lado do quadrado (o dobro do raio). Definimos como o número de pontos do quadrado na linha total_points. Criamos um "for" loop no qual geram-se um número aleatório de pontos até um limite (definido em total_points previamente), restando uma unidade para que fique entre (-1,1), para ter a origem em (0,0) graficamente. Se os pontos "x" e "y" existiram dentro dos parámetros estabelecidos na função points_in_circle (descrita em cima) acumulam-se os pontos na variável points_inside.

Já obtido o número de pontos, os quais cumprem os parámetros da área, aplicamos a simulação Monte Carlo; no qual π = 4 * [points_inside / total_points], mas substituindo com

os pontos obtidos, ou seja, π = 4 * [(Área círculo) / (Área retángulo)] O valor do float retornado da seria, portanto, a nossa aproximação ao valor de π.

Declaramos os parámetros da função do thread (monte_carlo()), em formato "struct", para passar depois aos argumentos do thread. Sabendo que cada thread precisa a sua próprio numero de total_points e um pointer para um inteiro onde possa guardar os seu resultado do points_inside (pontos gerados dentro do circulo).

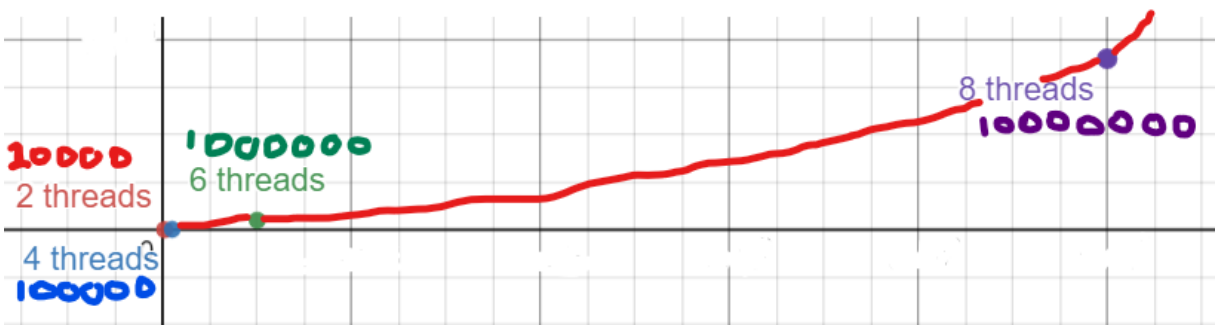
Declaramos as funções: point_in_circle (para saber se um certo ponto fica dentro do circulo), accuracy (para calcular a qualidade da estimação), o thread a chamar a função Monte Carlo, e a função "simulate" (para correr uma simulação com um certo numero de threads e total points). Criamos um array de os valores que toma a função cada vez que vai ser realizada. Realizamos a função "simulate" 4 vezes, uma vez para cada valor de número de pontos utilizado, e cada vez com um número diferente de threads.

Criamos um thread com a função pthread_create, com os IDs dos threads, que realiza a função Monte Carlo, nos quais os argumentos props são utilizados. Juntamos os valores dos threads. Criamos uma variável sum, para somar cada valor obtido em cada thread para ter o número total de pontos dentro do circulo e somar. O thread chama a função; Passamos cada um dos apontadores dos parámetros ao tipo correto. O apontador para points_inside aqui adquiere o valor de points_inside (dereferencing). Criamos logo uma variável float de "accuracy" para saber qual é a qualidade da estimativa; partimos do conhecimento do valor real de pi (3,141592, neste caso) e dividimos o resultado de cada aproximação entre o valor real e multiplicamos por 100. Para quando o valor estimado está por cima do real (por exemplo, 3,15), restamos a diferença a pi e voltamos a fazer a aproximação da diferença.

## ANÁLISE DOS RESULTADOS:

```
--------------------------------------------------------------------------------

Generated points: 20000
Points in circle: 15628
Threads: 2
Time taken: 0.006472
Actual Pi: 3.141592
Approximation: 3.125600
Accuracy: 99.490959%

--------------------------------------------------------------------------------

Generated points: 100000
Points in circle: 78536
Threads: 4
Time taken: 0.042056
Actual Pi: 3.141592
Approximation: 3.141440
Accuracy: 99.995163%

--------------------------------------------------------------------------------

Generated points: 1000000
Points in circle: 785653
Threads: 6
Time taken: 1.031037
Actual Pi: 3.141592
Approximation: 3.142612
Accuracy: 99.967537%

--------------------------------------------------------------------------------

Generated points: 10000000
Points in circle: 7853960
Threads: 8
Time taken: 18.521469
Actual Pi: 3.141592
Approximation: 3.141584
Accuracy: 99.999741%

--------------------------------------------------------------------------------
```

Como podemos ver, quanto maior seja o número de pontos gerados, melhor vai ser a qualidade do resultado, mas demora mais tempo a executar a programa.




Se vimos para gráfico da tempo como função de pontos de simulação, podemos ver que o tempo demorado cresce exponencialmente.