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

Arquiteturas de Alto Desempenho - Semana 2

Método de Monte Carlo para o cálculo do valor de π

Implementação em Python

Professor: Dr. Emerson Carlos Pedrino

Víctor Cora Colombo. RA 727356. Engenharia de Computação.

São Carlos, 20 de Novembro de 2020

Sumário

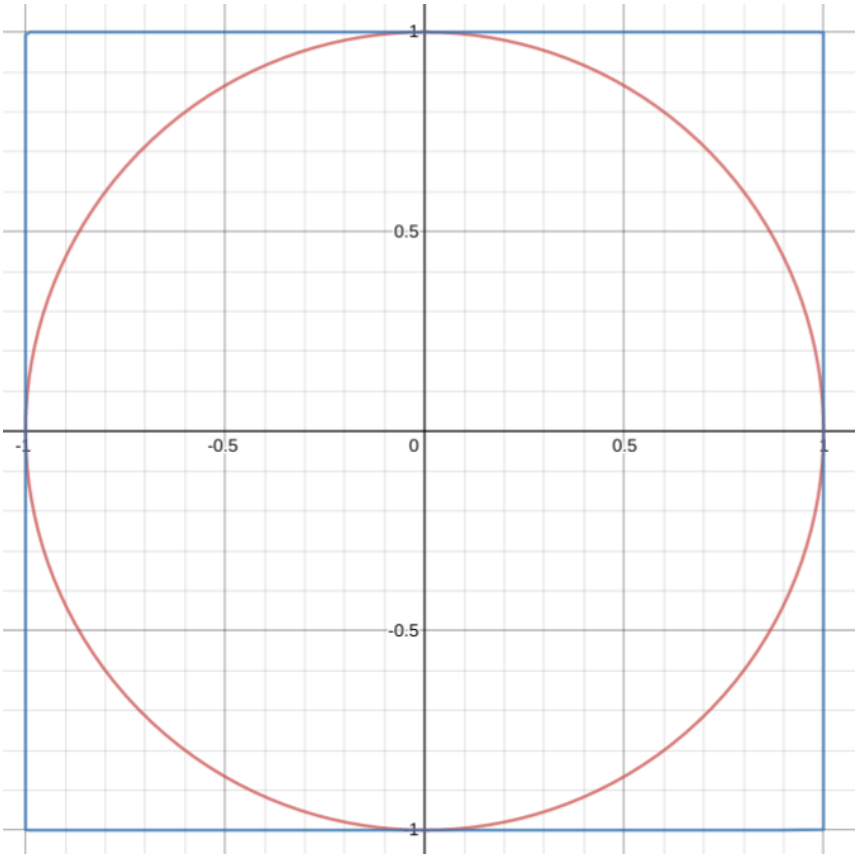
1 O método de Monte Carlo

1.1 Lógica por trás do método

O método de Monte Carlo para o cálculo do valor de π consiste no uso de uma grande quantidade de números aleatórios para aproximar seu valor de forma probabilística.

A ideia principal do método é inscrever uma circunferência de raio r em um quadrado de lado $2r$. Então sorteia-se um ponto aleatoriamente dentro do quadrado e verifica se está dentro da circunferência. Se está, é considerado um acerto. Repete-se esse processo para uma grande quantidade de pontos. A Figura 1 representa a situação descrita.

Figura 1 – Circunferência inscrita no quadrado.



Fonte: Feito pelo autor na plataforma Desmos[1].

O que acontece é que, com uma distribuição uniforme de pontos, a chance de um ponto dentro da circunferência ser sorteado é proporcional à sua área. É possível calcular a porcentagem da área do quadrado que está preenchida pela circunferência usando a Equação 1. Dessa forma, sorteando-se um ponto aleatoriamente, a probabilidade dele estar contido na circunferência é igual a P.

$$P = \frac{A_{circ}}{A_{quadrado}} \tag{1}$$

[1] https://www.desmos.com/calculator

Sabendo que a área da circunferência é dada pela Equação 2, podemos transformar a Equação 1 na Equação 3.

$$A_{circ} = \pi \cdot r^2 \tag{2}$$

$$\pi = \frac{P \cdot A_{quadrado}}{r^2} \tag{3}$$

Voltemos agora à ideia de sortear pontos. Digamos que um ponto escolhido estar dentro da circunferência significa um acerto. A probabilidade P do ponto estar dentro da circunferência, para uma quantidade N de pontos, é dada pela Equação 4, com $N_{acertos}$ a quantidade de acertos.

$$P = \frac{N_{acertos}}{N} \tag{4}$$

Para N grande o suficiente, é possível aproximar a Equação 3 para a Equação 5. O valor do raio r pode ser escolhido de forma arbitrária. Neste caso, será utilizado o raio unitário $r = 1$.

$$\pi \approx \frac{N_{acertos}}{N} \cdot A_{quadrado} \tag{5}$$

Falta apenas descobrir a área do quadrado. Esse valor é dado por $A_{quadrado} = L^2$, com L sendo seu lado. Como comentado anteriormente, $L = 2r$, portanto sua área é $A_{quadrado} = 4r^2$. Como o raio é unitário, pode-se derivar a Equação 5 para a Equação 6.

$$\pi \approx \frac{N_{acertos}}{N} \cdot 4 \tag{6}$$

1.2 Verificando que um ponto está contido na circunferência

Um ponto (x, y) está dentro da circunferência se satisfizer a Equação 7.

$$x^2 + y^2 < 1 \tag{7}$$

Para facilitar o código que será introduzido na próxima seção (na hora de gerar os números aleatórios), serão utilizados $0 < x < 1$ e $0 < y < 1$. Isso significa usar apenas $1/4$ da área original. Basta então dividir a área por 4 e multiplicar novamente no final. Portanto, a equação resultante continua sendo a Equação 6.

2 Implementação do método de Monte Carlo

2.1 Implementação sequencial

A implementação sequencial em Python é facilmente derivada da definição do método de Monte Carlo. Basta gerar N pontos aleatoriamente, e verificar se (x_i, y_i) satisfazer a Equação 7. Se sim, soma-se 1 a `acertos`. No final, utiliza-se a Equação 6 para encontrar o valor aproximado de π.

Diferentemente do Octave, Python é bom em lidar com *loops*. Portanto, será usado um `for`, em que cada iteração gerará um ponto aleatoriamente, e checará se a Equação 7 é satisfeita. Isso acarreta numa melhoria significativa do uso de memória em relação à implementação em Octave com vetores, pois não estão sendo guardados todos os N pontos na memória ao mesmo tempo.

O Código 1 mostra uma função que retorna a quantidade de acertos para um parâmetro N.

Código 1 – montecarlo.py

```
1  # Eh necessario ter o numpy instalado no seu ambiente
2  # $ pip install numpy
3  import numpy as np
4
5
6  def montecarlo(N, seed=1234):
7      np.random.seed(seed=seed)
8
9      acertos = 0
10
11     for i in range(N):
12         # gera o ponto (x_i, y_i)
13         x = np.random.random_sample()
14         y = np.random.random_sample()
15
16         # checa se (x_i, y_i) esta dentro da circunferencia
17         if (x**2 + y**2 < 1):
18             acertos += 1
19
20     return acertos
```

O Código 2 utiliza o Código 1 para retornar a quantidade de acertos de forma sequencial. Esse valor é então usado para calcular o resultado da Equação 6 e estimar o valor de π.

Código 2 – sequential.py

```
1  import sys
2
3  # eh importante ter o arquivo montecarlo.py no mesmo
       diretorio
4  from montecarlo import montecarlo
5
6
7  def sequential(N):
8      acertos = montecarlo(N)
9      return (acertos / N) * 4
10
11
12 if __name__ == "__main__":
13     if (len(sys.argv) != 2):
14         print("Quantidade incorreta de parametros")
15         print("Uso: $ python /path/to/sequential.py N")
16         sys.exit()
17
18     # N vem como parametro do usuario
19     N = int(sys.argv[1])
20
21     pi = sequential(N)
22     print("pi = %.7f" % (pi))
```

O resultado da execução da implementação sequencial é dada pelo Código 3. Foram testados $N = 5 \cdot 10^4$ e $N = 10^7$. No primeiro caso, com o uso da Equação 8, vemos que a estimativa foi $\pi = 3.1400800$, o que representa um erro de 0.04815% em relação ao valor real. Já para $N = 10^7$ a estimativa foi de $\pi = 3.1413620$, com erro de 0.00734%, indicando uma melhora considerável.

$$\delta = |\frac{\pi_{aprox} - \pi_{real}}{\pi_{real}}| \cdot 100\% \tag{8}$$

O custo desta melhoria é o tempo que o programa leva para finalizar. Partimos de uma execução que consumia 0.272 segundos do tempo da CPU, para uma execução de cerca de 13 segundos.

Código 3 – Resultado da execução de sequential.py

```
$ time python sequential.py 50000
pi = 3.1400800

real    0m0.272s
user    0m0.379s
sys     0m0.236s

$ time python sequential.py 10000000
pi = 3.1413620

real    0m13.008s
user    0m13.045s
sys     0m0.264s
```

Dessa forma, precisamos novamente recorrer ao uso da computação paralela para permitir a escalabilidade desse método.

2.2 Implementação paralela

Para diminuir o tempo de execução e permitir o aumento da precisão por meio do aumento da quantidade de pontos N, será mostrado como implementar uma solução paralela em Python para o problema da aproximação do valor de π pelo método de Monte Carlo.

A linguagem Python não possui uma maneira nativa de executar programas em múltiplas CPUs. Para isso, será usada a biblioteca `mpi4py`[2]. A versão mais recente da biblioteca no momento da escrita desse tutorial é a 3.0.3.

A biblioteca requer que o usuário tenha instalado no seu sistema uma implementação do padrão *Message Passing Interface*[3] (MPI). Nesse tutorial será usado o *open-mpi*[4], uma opção *open-source* disponível nos repositórios de todos os principais sistemas operacionais, mas qualquer outra implementação pode ser utilizada. No Ubuntu, sua instalação pode ser feita da seguinte forma:

```
$ sudo apt install openmpi-bin libopenmpi-dev
```

No openSUSE o comando é:

```
$ sudo zypper install openmpi openmpi-devel.
```

Para outras distribuições, verifique o instalador de pacotes do seu sistema.

[2] https://mpi4py.readthedocs.io/en/stable/
[3] https://computing.llnl.gov/tutorials/mpi/
[4] https://www.open-mpi.org/

Pode ser necessário usar o comando `mpi-selector` para dizer ao sistema que deve utilizar o *open-mpi*. Execute o comando a seguir e tudo pronto:

```
$ mpi-selector set openmpi
```

Agora, para finalmente instalar a biblioteca `mpi4py`, devemos indicar ao `pip` onde está o executável do *open-mpi*. Para encontrá-lo no seu sistema, rode o comando:

```
$ find / -name mpicc
```

A resposta desse comando deve conter alguma linha parecida com `/usr/lib64/mpi/gcc/openmpi/bin/mpicc`.

Com tudo isso feito, basta executar o seguinte comando:

```
$ env MPICC=/path/to/mpicc pip install mpi4py
```

Não se esqueça de mudar o valor após `MPICC=` para a localização do executável na sua máquina.

Se tudo foi instalado com sucesso, o seguinte comando deve apresentar uma mensagem de *"Hello World"* cinco vezes:

```
$ mpiexec -n 5 python -m mpi4py.bench helloworld
```

Um aviso sobre não ser possível encontrar uma interface de rede pode aparecer, mas iremos ignorá-la, visto que não causará problemas nos exemplos a seguir.

Com tudo instalado, podemos finalmente seguir para como utilizar a biblioteca para criar programas que utilizem múltiplas CPUs! Primeiramente, é importante entender como a biblioteca funciona. O `mpi4py` agrupa os processos paralelos em um comunicador. O comunicador padrão é `COMM_WORLD`, que une cada processo com todos os outros.

Usaremos `COMM_WORLD` como nosso comunicador. Existem três funções importantes dentro desse comunicador. A primeira é `Get_rank()`, que retorna o identificador do processo, que o difere dos outros processos em paralelo. Têm-se então que $0 \leq rank \leq P - 1$, com P o número de processos que foram criados.

A segunda é `Get_size()`, que retorna a quantidade total P de processos que foram criados. Com isso, qualquer processo pode saber quantos irmãos ele possui. Por fim, `gather(resposta, root)` une as respostas de todos os jobs paralelos em um único processo. Por exemplo `gather(foo(), root=0)` retorna uma lista em que cada posição é o valor retornado por `foo()` em algum dos processos. `root=0` indica que as respostas serão unidas no processo cujo rank é 0. Esse processo então será responsável por unificar as respostas e apresentar o resultado final. Há muitas outras funções no `mpi4py`, mas essas três são suficientes para implementar uma solução paralela para o método de Monte Carlo.

O Código 4 mostra a implementação da solução paralela do problema. A linha 28 define o comunicador, usando `COMM_WORLD` para permitir aos processos se comunicarem com todos os outros. A linha 31 então chama `parallel()` para cada processo, iniciando o cálculo em paralelo. Na linha 34, os resultados de todos os processos são unidos. É importante notar que `resultados` só receberá o valor no processo de rank 0. Nesse processo, `resultados` é uma lista de tamanho `comm.Get_size()`, ou seja, o número de

processos definidos. Todos os outros processos terão esse valor como nulo (`resultados = None`). Por fim, o bloco nas linhas 37-39 indicam que apenas o processo de rank 0 irá somar os resultados e printá-lo na tela.

A função `parallel(N, comm)` é responsável apenas por chamar o Código 1, de forma que cada processo irá executar o método de Monte Carlo para uma quantidade $N/size$ de pontos.

Código 4 – parallel.py

```
1  import sys
2
3  from mpi4py import MPI
4
5  # eh importante ter o arquivo montecarlo.py no mesmo
       diretorio
6  from montecarlo import montecarlo
7
8
9  def parallel(N, comm=MPI.COMM_WORLD):
10     rank = comm.Get_rank()
11     size = comm.Get_size()
12
13     # usamos seed=rank para ter resultados diferentes para
          cada processo
14     # cada processo eh responsavel por calcular o
          resultado para uma parcela menor do problema, de
          tamanho N/size
15     return montecarlo(int(N/size), seed=rank)
16
17
18 if __name__ == "__main__":
19     if (len(sys.argv) != 2):
20         print("Quantidade incorreta de parametros")
21         print("Uso: $ python /path/to/parallel.py N")
22         sys.exit()
23
24     # N vem como parametro do usuario
25     N = int(sys.argv[1])
26
```

```
27      # COMM_WORLD eh um comunicador que agrupa todos os
            processos
28      comm = MPI.COMM_WORLD
29
30      # cada processo calcula o numero de acertos
            individualmente
31      acertos = parallel(N, comm)
32
33      # os valores resultantes sao reunidos no processo de
            rank 0
34      resultados = comm.gather(acertos, root=0)
35
36      # o processo de rank 0 eh responsavel por somar os
            resultados e apresenta-los
37      if comm.Get_rank() == 0:
38          pi = (sum(resultados) / N) * 4
39          print("pi = %.7f" % (pi))
```

O resultado da execução da implementação paralela é dada pelo Código 5. Foram testados, novamente, $N = 5 \cdot 10^4$ e $N = 10^7$. Para executar o código, é necessário usar o comando `mpiexec`, com o parâmetro `-n 4`, indicando que serão criados 4 processos, seguido do script Python a ser executado. A quantidade de processos não precisa ser igual nem menor que o número de processadores disponíveis na máquina. Se forem criados mais processos que processadores, eles apenas se alternarão dentro das CPUs. Esse parâmetro então pode ser usado para tunar a execução.

No primeiro caso, para $N = 5 \cdot 10^4$, vemos que a estimativa foi $\pi = 3.1407200$, o que parece indicar que a implementação está correta e retornando um valor praticamente idêntico ao da execução sequencial. Contudo, no tempo de execução, vemos algo estranho. O período aumentou de 0.272 segundos no sequencial para 0.780 segundos no paralelo. Isso se deve ao *overhead* de inicialização da biblioteca e dos processos que ela cria.

Para $N = 10^7$, já vemos que houve uma melhora em relação à execução sequencial. O tempo foi de 13.008 segundos para 7.601 segundos, um *speedup* de 1.711, indicando que a implementação paralela foi 1.7 vezes mais rápida que que a sequencial para essa quantidade de pontos.

Código 5 – Resultado da execução de parallel.py

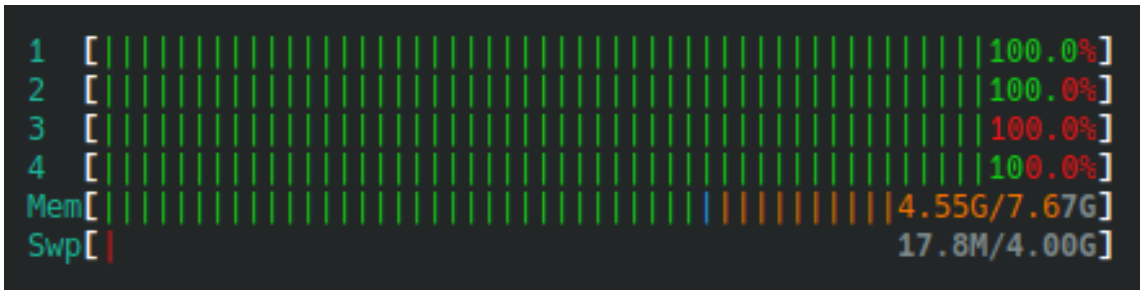
```
1 $ time mpiexec -n 4 python parallel.py 50000
2 ...
3
```

```
pi = 3.1407200

...

real    0m0.780s
user    0m1.355s
sys     0m0.469s

$ time mpiexec -n 4 python parallel.py 10000000
...

pi = 3.1415092

...

real    0m7.601s
user    0m25.179s
sys     0m0.715s
```

A ideia que fica então é que, a partir de uma certa quantidade de pontos, a implementação paralela passa a escalar melhor. Uma implementação sequencial teria problemas para aumentar a precisão por meio do aumento de pontos, visto que isso acarretaria em um grande período de execução. Com a solução paralela, basta adicionar mais processadores à máquina e aumentar a quantidade de pontos.

A Figura 2 mostra o uso das CPUs do sistema na execução da implementação paralela. Veja que todos os 4 processadores estão sendo 100% utilizados, indicando o paralelismo.

Figura 2 – Uso das CPUs na implementação paralela.



Fonte: Feito pelo autor com o comando `htop`

O Código 6 mostra o `script.py`, que pode ser usado para rodar todos os exemplos desse tutorial de forma automática.

Código 6 – script.py

```
1 import os
2
3 dir_path = os.path.dirname(os.path.realpath(__file__))
4 seq_file = dir_path + "/sequential.py"
5 paral_file = dir_path + "/parallel.py"
6
7 print("RUNNING: time python %s 50000" % (seq_file))
8 os.system("time python %s 50000" % (seq_file))
9
10 print("\n\n\n\nRUNNING: time python %s 10000000" %
    (seq_file))
11 os.system("time python %s 10000000" % (seq_file))
12
13 print("\n\n\n\nRUNNING: time mpiexec -n 4 python %s 50000"
    % (paral_file))
14 os.system("time mpiexec -n 4 python %s 50000" %
    (paral_file))
15
16 print("\n\n\n\nRUNNING: time mpiexec -n 4 python %s
    10000000" % (paral_file))
17 os.system("time mpiexec -n 4 python %s 10000000" %
    (paral_file))
```

3 Arquitetura paralela mais adequada

Uma arquitetura paralela adequada para a implementação desse problema seria a arquitetura *Single Instruction, Multiple Data* (SIMD). Isso se dá pelo fato de que temos o mesmo conjunto de instruções executando nos processos, daí o *Single Instruction*. Já em relação aos dados, cada CPU irá atuar sobre um conjunto diferente de dados, os pontos gerados aleatoriamente naquele processo, por isso é *Multiple Data*.

SIMD é uma arquitetura muito utilizada em processamento paralelo para a "solução de problemas computacionalmente intensivos da área científica e de engenharia"[5]. Nela, os processadores compartilham um único *instruction pool*, executando um único comando

[5] http://www.rc.unesp.br/igce/demac/balthazar/gpacp/bibliografia/A-06%20-%20Processamento%20Paralelo%20e%20de%20Alto%20Desempenho%20-%20NAT%20-%20Unive/SIMD.pdf

por vez. Contudo, esses comandos são operados sobre dados diferentes, o que confere seu paralelismo.

É importante notar que as implementações mostradas neste trabalho seguem a arquitetura *Multiple Instruction, Multiple Data* (MIMD), pois estão sendo executadas em um computador pessoal, que implementa uma versão de pequena escala dessa arquitetura. Portanto, MIMD também é uma opção plausível e perfeitamente adequada para esse problema.