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

Arquiteturas de Alto Desempenho - Semana 1

Método de Monte Carlo para o cálculo do valor de π

Professor: Dr. Emerson Carlos Pedrino

Víctor Cora Colombo. RA 727356. Engenharia de Computação.

São Carlos, 13 de Novembro de 2020

Sumário

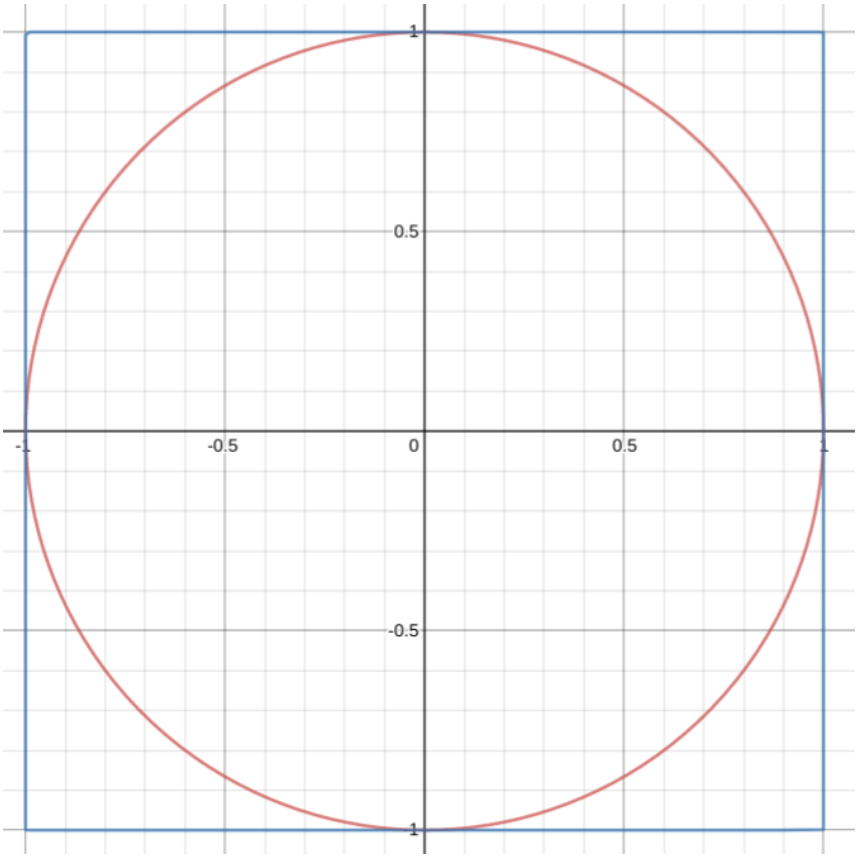
1 O método de Monte Carlo

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

Uma implementação sequencial trivial em Octave poderia ser feita como mostrado no Código 1. Essa solução é uma simples adaptação do código presente no site *blogcyberin*[2].

Código 1 – Monte Carlo sequencial

```
1  function pi = montecarlo_sequencial(n)
2    acertos = 0;
3    for i = 1:n
4      x = rand(1); % gera um numero aleatorio entre 0 e 1
5      y = rand(1); % gera um numero aleatorio entre 0 e 1
6      if (x^2 + y^2 < 1)
7        % verifica se o ponto (x,y) esta dentro da
             circunferencia
8        acertos = acertos + 1;
9      endif
10   endfor
11
12   % calcula o valor de π de acordo com a Equacao 6
13   pi = 4 * (acertos / n);
14 endfunction
```

Contudo, essa implementação resulta em um elevado tempo de execução. Para um valor de N de apenas 10^6, já temos um tempo de execução na casa dos 20 segundos. E o resultado ainda está longe de ser uma aproximação satisfatória para o valor de π, é necessário uma quantidade muito maior de pontos.

Código 2 – Resultado do Código 1

```
1 >> tic();
2 >> sequencial = montecarlo_sequencial(1e6);
3 >> toc();
4 >> printf("pi = %.7f\n", sequencial);
5 Elapsed time is 18.5034 seconds.
6 pi = 3.1394520
```

[2] https://www.blogcyberini.com/2018/09/calculando-o-valor-de-pi-via-metodo-de-monte-carlo.html

Isso acontece pois *loops* em Matlab/Octave são extremamente lentos (Bister et al. 2007). A linguagem é otimizada para funcionar com operações sobre vetores, e sofre de grandes penalizações quando usando for-loops. O que faremos então é modificar o código para remover o *loop* e transformá-lo em uma operação de vetores.

Primeiramente, serão gerados todos os pontos antecipadamente, resultando em dois vetores x e y, em que cada par (x_i, y_i) indica um ponto sorteado. Esse passo é mostrado nas linhas 2 e 3 do Código 3. Tem-se agora dois vetores, em que podemos operar diretamente sobre eles.

A ideia agora é calcular a Equação 7 para cada par (x_i, y_i). Primeiramente usa-se o operador .^2, que elevará cada entrada no vetor ao quadrado. Em seguida, soma-se cada elemento de x com seu respectivo par em y, resultando em um vetor S em que cada entrada é a soma $x_i^2 + y_i^2$. Finalmente, para cada elemento em S, verifica-se se ele satisfaz a Equação 7. O vetor resultante é uma sequência de 1 ou 0, em que 1 significa que o ponto (x_i, y_i) está dentro da circunferência, e 0 caso contrário. Finalmente, a quantidade de acertos é a soma de todas as entradas 1 desse vetor. O código completo pode ser visto no Código 3.

Código 3 – Monte Carlo sequencial melhorado

```
1  function pi = montecarlo_sequencial(n)
2    x = rand(1, n);
3    y = rand(1, n);
4
5    % .^2 significa elevar cada elemento do vetor ao
       quadrado separadamente, tambem conhecido como
       element-wise
6    % x.^2 + y.^2 < 1.0 gera um vetor de tamanho N em que
       uma entrada vale 1 se esta dentro da circunferencia,
       e 0 caso contrario
7    % sum() retorna a soma de todos os elementos do vetor
8    acertos = sum(x.^2 + y.^2 < 1.0);
9
10   pi = 4 * (acertos / n);
11 endfunction
```

Dessa vez, para $N = 10^6$, o tempo de execução foi de meros 0.049762 segundos. Isso significa que o código modificado foi 371 vezes mais rápido! Contudo a resposta do valor aproximado de π ainda não é satisfatória. Com o uso da Equação 8, é possível calcular que seu erro atual é de 0.068%. Pode parecer um erro pequeno, mas estamos acertando uma única casa decimal do resultado por enquanto.

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

Uma quantidade maior de pontos resulta em um valor mais próximo do correto. O Código 4 mostra que, para $N = 10^8$, encontramos um valor aproximado de $\pi = 3.1417232$. Isso reduz o erro para apenas 0.0042%, ou seja, um acerto de 3 casas decimais!

Código 4 – Resultado do Código 3

```
1 >> tic();
2 >> sequencial = montecarlo_sequencial(1e8);
3 >> toc();
4 >> printf("pi = %.7f\n", sequencial);
5 Elapsed time is 6.59926 seconds.
6 pi = 3.1417232
```

Contudo, já é possível notar que até mesmo o código sequencial otimizado está demorando para terminar a execução com $N = 10^8$. Se a memória do sistema não for um fator limitante (o código atual causa dificuldades até para um sistema de 8GB de RAM), o tempo de execução já dificulta aumentar a quantidade de pontos.

2.2 Implementação paralela

Para melhorar a performance a partir desse ponto, é necessário utilizar computação paralela para tirar proveito de todas as *CPUs* da máquina. Em Octave, isso é feito usando o pacote *Parallel*[3]. Para instalá-lo, basta executar o comando `pkg install -forge parallel` dentro do ambiente do Octave, e carregá-lo usando `pkg load parallel`. Pode ser necessário possuir o `octave-devel`[4] instalado em sua máquina.

Vamos agora começar a entender como paralelizar o código sequencial visto na última seção. Uma primeira tentativa poderia ser adaptar o código de forma a ficar igual ao do Código 5. Nessa função, a parte que está sendo paralelizada é a verificação de se os pontos estão contidos na circunferência (a função `fun` na linha 5).

`pararrayfun`[5] recebe como parâmetros o número de processadores a serem usados, a função que será paralelizada, e os argumentos dessa função (no caso os vetores x e y). Em seguida, o parâmetro `"Vectorized", true` indica que esse processamento será feito em vetores, o que ativará algumas melhorias na perfomance do paralelismo. `"ChunksPerProc", 8` indica que os vetores serão divididos em $8 \cdot nproc$ chunks, de forma que quando um processador ficar livre, ele pegará o próximo chunk para processar. Isso evita que um processador fique ocioso se terminar antes dos anteriores.

[3] https://wiki.octave.org/Parallel_package
[4] https://wiki.octave.org/Octave_for_GNU/Linux
[5] https://octave.sourceforge.io/parallel/package_doc/pararrayfun.html

O resultado do processamento paralelo então é um vetor `respostas_parciais` com N elementos, em que cada entrada indica se o ponto (x_i, y_i) está dentro da circunferência. Finalmente, a linha 9 soma todos os acertos e retorna o valor aproximado de pi.

Código 5 – Monte Carlo paralelo

```
1  function pi = montecarlo_paralelo(n)
2    x = rand(1, n);
3    y = rand(1, n);
4
5    fun = @(x, y) x.^2 + y.^2 < 1.0;
6
7    respostas_parciais = pararrayfun(nproc, fun, x, y,
       "Vectorized", true, "ChunksPerProc", 8);
8
9    pi = 4 * (sum(respostas_parciais) / n);
10 endfunction
```

Parece tudo certo! Vamos então executar. O resultado pode ser visto no Código 6.

Código 6 – Resultado do Código 5

```
1 >> tic();
2 >> paralelo = montecarlo_paralelo(1e8);
3 >> toc();
4 >> printf("pi = %.7f\n", paralelo);
5 Elapsed time is 8.18526 seconds.
6 pi = 3.1416203
```

Curiosamente, o resultado foi pior do que no processamento sequencial! O que está acontecendo?

É necessário entender um pouco como o `pararrayfun` funciona internamente. Primeiramente, é chamado um `fork()` para cada processador que será usado[6]. Isso causa um grande *overhead* inicial para a aplicação, na casa de 0.3 segundos. Além disso, é necessário copiar os dados que serão processados para os forks. Essa cópia é lenta, e como estamos copiando vetores gigantescos (cada processador está recebendo dois vetores de tamanho $N/nproc$), é um grande fardo para o tempo de execução.

Em seguida, é necessário retornar os valores processados e uni-los. Estamos retornando um vetor com tamanho N, outra operação extremamente custosa. Por fim esse vetor

[6] http://alberto-computerengineering.blogspot.com/2013/07/parallel-programming-in-octave.html

é somado localmente, ou seja, em um único processador. Talvez essa parte possa ser feita também de forma paralela.

Vamos então resolver todos os *overheads* discutidos. Primeiramente, a ideia é delegar a criação dos vetores para que as CPUs o façam de forma paralela. Isso significa que será passado para os forks apenas a quantidade de pontos a serem analisados em cada *chunk*, e não os pontos em si. Eles então serão responsáveis pela criação dos vetores. Isso pode ser visto nas linhas 3 e 4 do Código 7.

Código 7 – Monte Carlo paralelo modificado fun

```
1 function r = fun(n)
2     % A geracao aleatoria de pontos agora esta sob
         responsabilidade dos jobs paralelos
3     x = rand(1, n);
4     y = rand(1, n);
5
6     % fun agora soma a quantidade de acertos ao inves de
         retornar o vetor de 0 e 1 para ser processado
         localmente
7     r = sum(x.^2 + y.^2 < 1.0);
8 endfunction
```

A função `repmat` na linha 3 do Código 8 cria um `vetor` de tamanho `jobs`, em que cada entrada é o número de pontos que uma instância de processamento paralelo irá calcular. A ideia então é dividir N em vários jobs para serem processados de forma paralela.

Agora, passamos apenas o `vetor` para `pararrayfun`, de forma que removemos o enorme *overheads* de copiar quantidades gigantescas de dados, e cada fork recebe apenas um parâmetro N com o número de pontos que deve processar. O resultado é um vetor de tamanho `jobs`, em que cada entrada é a quantidade de acertos calculados em um dos jobs. Isso novamente representa uma melhora na transmissão de dados, pois a soma já foi feita dentro do job paralelo.

Por fim, é necessário apenas somar os acertos de cada job e calcular o valor aproximado de π. O código completo pode ser visto no Código 8.

Código 8 – Monte Carlo paralelo modificado

```
1 function pi = montecarlo_paralelo(n, jobs=4)
2   % repmat cria um vetor de tamanho jobs em que todas as
       entradas sao iguais a n/jobs. Esse valor indica o
       tamanho
3   vetor = repmat(ceil(n/jobs), 1, jobs);
```

```
4
5   % fun agora precisa estar precedido de @. Essa eh uma
       necessidade do pararrayfun quando fun eh uma funcao
       que esta em seu proprio arquivo
6   respostas_parciais = pararrayfun(nproc, @fun, vetor,
       "Vectorized", true, "ChunksPerProc", 8);
7
8   pi = 4 * (sum(respostas_parciais) / n);
9 endfunction
```

Vamos checar então checar se houve melhora no tempo de execução.

Código 9 – Resultado do Código 8

```
1 >> tic();
2 >> paralelo = montecarlo_paralelo(1e8);
3 >> toc();
4 >> printf("pi = %.7f\n", paralelo);
5 Elapsed time is 3.07373 seconds.
6 pi = 3.1413485
```

A melhoria é considerável! Ela representa um *speedup* de 2.15 vezes em relação ao código sequencial. Isso significa que o código está rodando em menos da metade do tempo da solução sequencial. Além disso, essa solução é muito mais escalável. É possível ter ganhos ainda melhores ao ajustar os parâmetros de `pararrayfun` e a quantidade de `jobs`.

O Código 10 mostra o arquivo *script.m* completo, em que é possível modificar os parâmetros das soluções e comparar os resultados obtidos.

Código 10 – script.m

```
1 clear all;
2
3 pkg load parallel;
4
5 n = 1e8
6 jobs = 32
7
8 tic();
9 sequencial = montecarlo_sequencial(n);
10 toc();
```

```
11 printf("Sequencial_good: %.7f\n", sequencial);
12
13 tic();
14 paralelo = montecarlo_paralelo_bad(n);
15 toc();
16 printf("Paralelo bad: %.7f\n", paralelo);
17
18 tic();
19 paralelo = montecarlo_paralelo(n, jobs);
20 toc();
21 printf("Paralelo good: %.7f\n", paralelo);
```

3 Arquitetura paralela mais adequada

A arquitetura paralela mais adequada para a implementação desse problema é a arquitetura *Multiple Instruction, Multiple Data* (MIMD). Isso se dá pelo fato de que precisamos de várias instruções executando ao mesmo tempo, no mesmo ciclo de processamento (daí o *Multiple Instruction*. Já em relação aos dados, cada CPU irá atuar sobre um conjunto diferente de dados, os pontos gerados aleatoriamente naquele job.

MIMD é uma arquitetura muito utilizada em processamento paralelo[7]. Nela, cada processador possui seu próprio *instruction pool*, podendo executar comandos de forma independente, e os processos podem trabalhar sobre seu próprio conjunto isolado de dados. Isso é exatamente o que é necessário na solução para o problema de Monte Carlo dada pelo Código 8, em que cada job gera seus próprios pontos e opera sobre eles. Se houvesse um vetor compartilhado de pontos, como é o caso do Código 5, talvez uma arquitetura *Multiple Instruction, Single Data*[8] (MISD) poderia ser utilizada.

[7] https://en.wikipedia.org/wiki/MIMD
[8] https://en.wikipedia.org/wiki/MISD

Referências Bibliográficas

[1] M Bister et al. “Increasing the speed of medical image processing in MatLab®”. Em: *Biomedical imaging and intervention journal* 3.1 (2007).