Cálculo de Pi através do método de Monte Carlo utilizando MPI

Leandro Silva Ferreira Junior

1. Introdução

Tendo como objetivo a aproximação do valor de pi, nosso algoritmo utilizará a comparação da área de um círculo de raio r com a de um quadrado de lado 2r para fins de cálculo como será mostrado a seguir.

Para simplificar o algoritmo e sua implementação, utilizamos um valor unitário para o raio do círculo, dessa forma, temos que $r = 1$, então, a área do círculo consiste em:

$$Ac = \pi.r^2 = \pi.1^2 = \pi.$$

Da mesma forma temos a área do quadrado $Aq = l^2$ onde $l = 2r = 2$, então:

$$Aq = (2r)^2 = 2^2 = 4$$

Da proporção entre as duas áreas temos que:

$$\frac{Ac}{Aq} = \frac{\pi}{4}$$

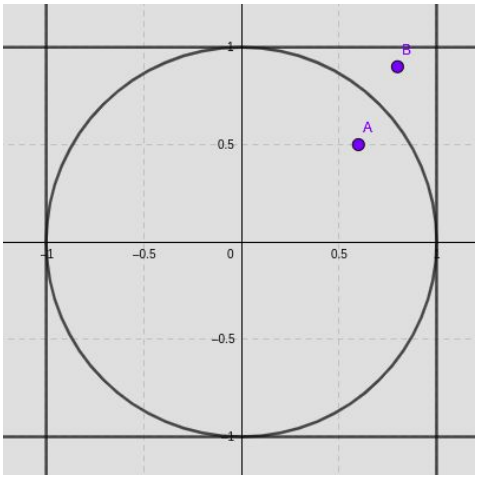
e finalmente, isolando π:

$$\pi = 4 \frac{Ac}{Aq}.$$

O método de Monte Carlo consiste em um teste estatístico que utiliza-se de amostragens aleatórias para calcular uma probabilidade. No nosso caso, usaremos o Monte Carlo para calcular a aproximação da proporção entre as áreas do círculo e do quadrado, o que implica no cálculo de pi utilizando tal método:

$\frac{Ac}{Aq}$ calculado através de Monte Carlo.

Tomando como parâmetro um círculo e um quadrado de centro (0,0), para realizar o cálculo devemos gerar pontos aleatórios com coordenadas x e y entre -1 e 1, assim garantimos que o ponto está dentro do quadrado, e verificamos se o ponto se encontra dentro/sob a circunferência cuja equação é $x^2 + y^2 = 1$.



Da equação geral do círculo temos que

$$r^2 = x^2 + y^2 \text{ , logo, } r = \sqrt{x^2 + y^2}$$

Como $r = 1$, se $-1 \leq \sqrt{x^2 + y^2} \leq 1$, o ponto estará contido no círculo.

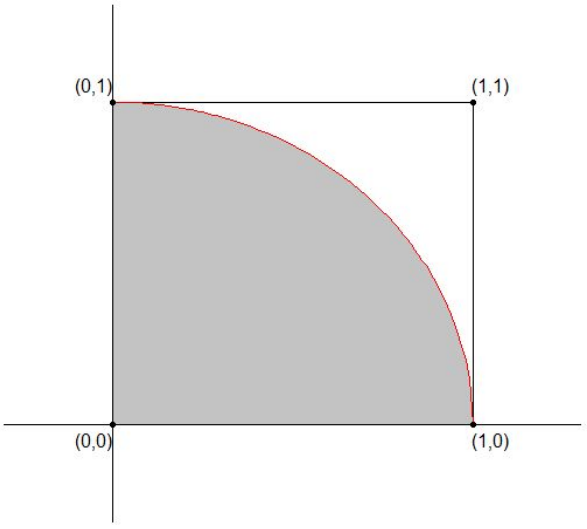
Quanto maior for o número de pontos sorteados aleatoriamente para calcular a proporção entre pontos dentro do círculo e pontos totais, mais preciso será o retorno da aplicação do método na obtenção do valor de pi.

Partindo dessa premissa, a implementação dos códigos paralelos visa otimizar o tempo de cálculo de quantos destes números aleatórios estão no círculo, podendo dividir a quantidade de números (tamanho do problema) pelo número de cores disponíveis para execução. Assim podemos aplicar tamanhos de problemas extremamente grandes sem resultar em um tempo de execução tão alto.

Discutiremos a seguir a maneira como o código serial e o código paralelo foram implementados, bem como os tamanhos de problema utilizados nos experimentos e por último as análises dos possíveis ganhos em eficiência e speedup.

2. Desenvolvimento

Para facilitar a implementação do algoritmo, os valores gerados para x e y não irão de -1 a 1 e sim de 0 a 1, desta forma estaremos trabalhando apenas com um único quadrante do círculo e do quadrado:



Dada a situação, o cálculo de $\frac{Ac}{Aq}$ corrigido seria $\frac{Ac/4}{Aq/4}$ que nada mais é do que o próprio $\frac{Ac}{Aq}$, e a condição para verificar se o ponto faz parte do arco será $\sqrt{x^2 + y^2} \leq 1$.

2.1 Implementação Serial (linguagem C)

A implementação do algoritmo de Monte Carlo para calcular Pi no código serial consiste em um laço que será executado n vezes o tamanho do problema, que no caso é a quantidade de números aleatórios gerados. Neste laço são gerados dois pontos aleatórios entre 0 e 1: x e y e em seguida é verificado se o ponto de P(x,y) se encontra dentro do círculo de $raio = (x^2) + (y^2)$, se a condição for verdadeira um contador incrementa uma unidade.

Para gerar os valores aleatórios de x e y, foi utilizada a função $rand()$, que gera valores de 0 até $RAND\_MAX$. Dividindo o valor gerado pelo próprio $RAND\_MAX$, garantimos que os valores para x e y flutuam sempre entre 0 e 1.

Após a execução do laço, teremos o número total de pontos calculados dentro da circunferência e podemos fazer a aproximação de $\frac{Ac}{Aq}$ dividindo o contador de pontos dentro pelo n total. Seguindo a equação $\pi = 4 \frac{Ac}{Aq}$, basta multiplicar a aproximação por 4 e iremos obter o valor aproximado de π.

```
for (int i=0; i<size; i++) {

    x = (double)rand() / RAND_MAX; //Gera x entre 0 e 1
    y = (double)rand() / RAND_MAX; //Gera y entre 0 e 1
    raio = sqrt(x*x + y*y);

    if (raio<=1) {
        pdentro++;
    }
}

valorpi=((double)pdentro/size)*4;       //Calcula Pi
```

No código foi inserido um “printf” que imprime o pi calculado para cada iteração, esse valor foi salvo no arquivo “pi_serial.out”, dessa forma é possível verificar se o cálculo feito está correto.

```
printf("Size %d PI %.10f \n", size, valorpi); //Imprime o valor de Pi em serial_pi.out
```

Como nosso objetivo final é analisar e comparar os tempos de execução do programa, antes de iniciar qualquer cálculo, foi inserido um medidor de tempo. A mesma coisa foi feita após o fim do cálculo de pi. A diferença entre esses dois tempos será o tempo de execução do programa.

```
gettimeofday(&comeco, 0); //Mede o tempo inicial
gettimeofday(&final, 0); //Mede o tempo final
```

Ao final do código, é impresso no arquivo “serial_pi.txt” o tempo de execução a cada vez que o programa for rodado, além disso foi incluída uma condição para parar o programa caso o arquivo .txt não consiga ser aberto.

```
char nomedoarquivo[] = "serial_pi.txt";
FILE *fp;
fp = fopen(nomedoarquivo, "a");    //Abre o arquivo txt

if (fp == NULL) {
  exit(1);
}
else {
    fprintf(fp, "%.3f\t", (final.tv_sec+final.tv_usec*1e-6)-(comeco.tv_sec+comeco.tv_usec*1e-6));
    fclose(fp);
}
```

2.2 Implementação Paralela (também em linguagem C, utilizando MPI):

A primeira diferença da implementação em MPI consiste em dividir o tamanho do problema (n) pelo número de cores (p) disponíveis, assim o laço que gera os pontos aleatórios

e calcula o contador de pontos dentros será executado localmente para cada thread com tamanho $\frac{n}{p}$.

Para implementar isso, já dentro da região paralela, o laço de execução irá até $\frac{n}{p}$:

```
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD, &threads);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

gettimeofday(&comeco, NULL);

for (int i=0; i < size/threads; i++) {
  //Calculo de Pi
  x = (double)rand() / RAND_MAX;
  y = (double)rand() / RAND_MAX;
  raio = sqrt(x*x + y*y);

    if (raio<=1) {
        pdentrolocal++;
    }
}
```

Em uma situação ideal, se duplicarmos o número de cores, o tempo de execução do cálculo tenderia à metade do inicial.

Após o cálculo das variáveis locais (pdentrolocal), foi utilizada a função MPI_Reduce, com a operação MPI_SUM para somar esses valores em uma variável global (pdentro) e retorná-la ao rank 0.

O rank 0 então realiza as operações de calcular o π, imprimir a saída do valor de π no arquivo "paralelo_pi.out", calcular o tempo de execução e salvá-lo no arquivo txt, pois a região paralela ainda está aberta.

```
if(rank==0) {

    valorpi=((double)pdentro/size)*4; //Calcula Pi

    gettimeofday(&final, NULL); //Mede o tempo final
    printf("Size %d PI %.10f \n", size, valorpi); //Imprime o valor de Pi em paralelo_pi.out

    char saida[] = "paralelo_pi.txt";
    FILE *fp;
    fp = fopen(saida, "a");    //Abre o arquivo txt

    if (fp == NULL) {
        exit(1);
    }
    else {
        fprintf(fp, "%.3f\t", (final.tv_sec+final.tv_usec*1e-6)-(comeco.tv_sec+comeco.tv_usec*1e-6));
        fclose(fp);
    }
}
```

Somente após todas essas etapas, fecharemos a região paralela utilizando a função MPI_Finalize.

3. Experimentos

3.1 Parâmetros

O testes foram realizados para 4 tamanhos de problema diferentes: 100, 200, 400 e 800 milhões de pontos. Para cada tamanho, o código foi executado 20 vezes, e o tempo de execução que utilizaremos consiste da média desses 20 tempos.

Para organizar todos os casos de testes do código, utilizamos um shellscript para o código serial e outro para o paralelo. O script serial nos dará como resposta 4 tempos médios, um para cada tamanho.

No caso do script paralelo, foi inserido mais um laço para o número de cores, desta forma, os testes para os 4 tamanhos serão realizados utilizando 2, 4, 8, 16 e 32 cores, resultando em 20 tempos médios como resposta.

3.2 SpeedUp e Eficiência

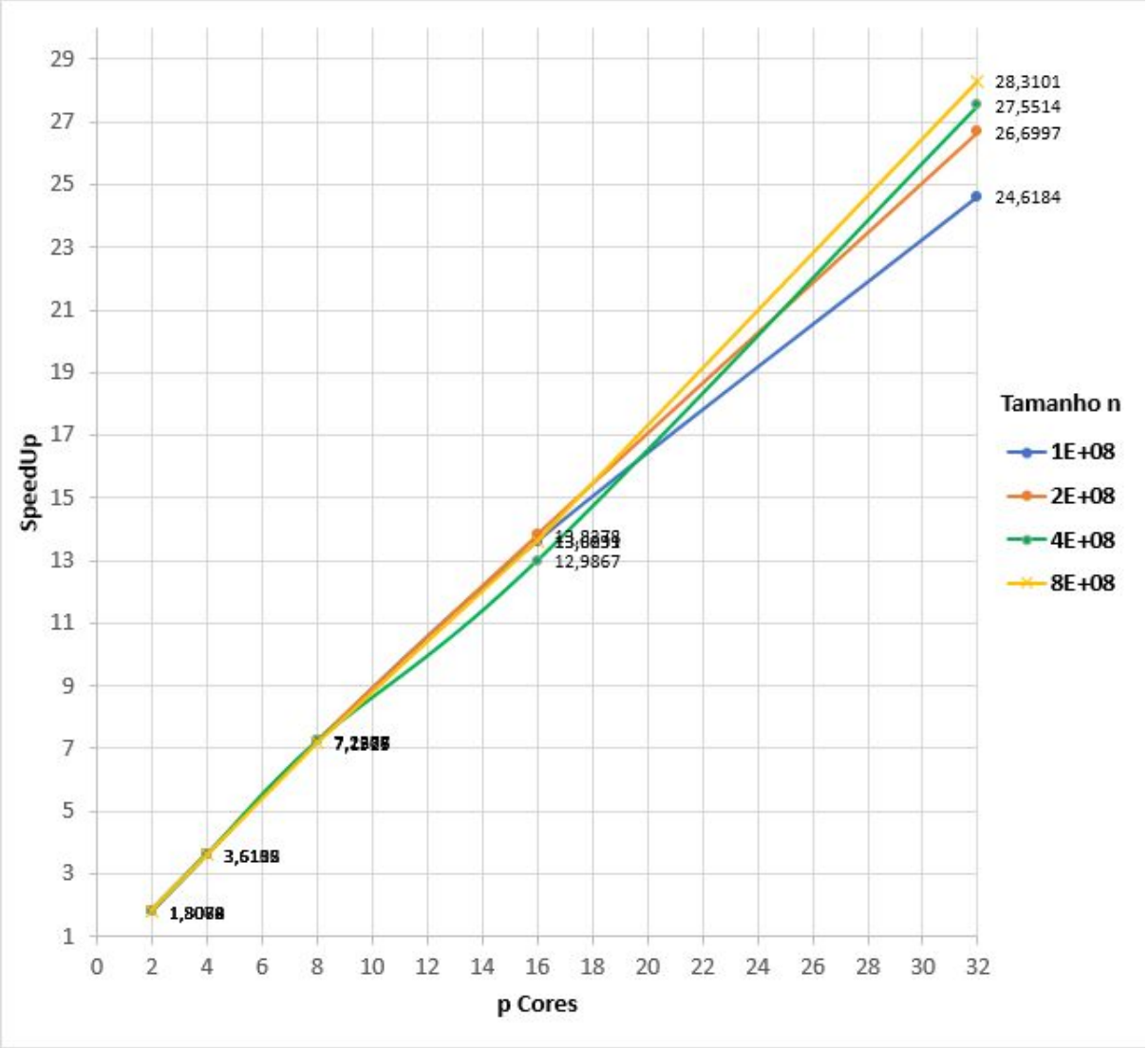
Após concluída a execução dos scripts, os arquivos de saída foram importados para a plataforma do Excel, onde foi calculado a média dos 20 tempos, os valores de SpeedUp, e os valores de Eficiência onde

$S = Ts / Tp.$ e $Ef = S / p\ cores$

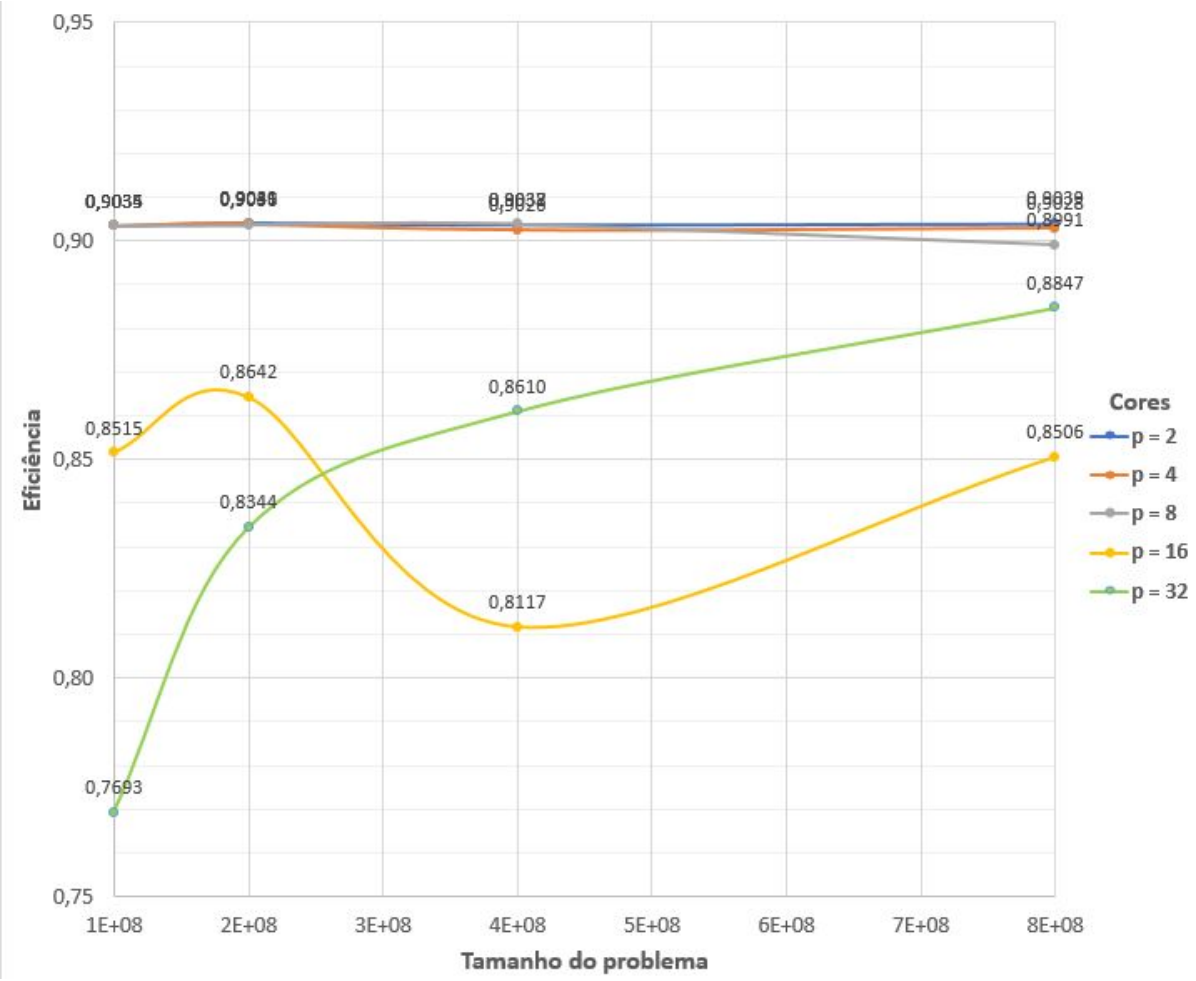
N Cores	Tamanho	Média Tempo	SpeedUp	Eficiência
Serial	1E+08	3,061	--	--
Serial	2E+08	6,122	--	--
Serial	4E+08	12,244	--	--
Serial	8E+08	24,488	--	--
2	1E+08	1,694	1,8069	0,9035
2	2E+08	3,386	1,8082	0,9041
2	4E+08	6,774	1,8074	0,9037
2	8E+08	13,546	1,8078	0,9039
4	1E+08	0,847	3,6136	0,9034
4	2E+08	1,693	3,6155	0,9039
4	4E+08	3,391	3,6103	0,9026
4	8E+08	6,781	3,6112	0,9028
8	1E+08	0,424	7,2277	0,9035
8	2E+08	0,847	7,2286	0,9036
8	4E+08	1,693	7,2308	0,9038
8	8E+08	3,405	7,1925	0,8991
16	1E+08	0,225	13,6239	0,8515
16	2E+08	0,443	13,8278	0,8642
16	4E+08	0,943	12,9867	0,8117
16	8E+08	1,799	13,6091	0,8506
32	1E+08	0,124	24,6184	0,7693
32	2E+08	0,229	26,6997	0,8344
32	4E+08	0,444	27,5514	0,8610
32	8E+08	0,865	28,3101	0,8847

Em seguida, foram plotados dois gráficos:

O gráfico de SpeedUps, que diz quantas vezes o código paralelo é mais rápido do que o serial, analisando os 4 tamanhos de problemas diferentes:



E o gráfico de Eficiência, que diz o quão bem os cores estão sendo utilizados no paralelismo do código, a análise é feita para os 5 tamanhos de cores utilizados:



4 Conclusões

Analisando os resultados de tempo e speedups, é possível perceber que há um ganho considerável de tempo de execução do algoritmo paralelo em relação ao algoritmo serial, isso evidencia o quanto paralelizar um problema otimiza a sua execução. O padrão de comportamento do gráfico de SpeedUps evidencía a Lei de Gustafson, pois ao aumentarmos o tamanho do problema, aumentamos a sua fração paralela, e proporcionalmente os tempos de execução e speedups.

Ao analisar a escalabilidade do algoritmo, foi concluído que, para os tamanhos utilizados, a única situação onde o código é escalável é com 32 cores. Analisando como um todo, o algoritmo é não-escalável, já que para um número fixo de cores, ao aumentarmos o tamanho do problema, a eficiência decresce em alguns momentos.

Das hipóteses, temos que para tamanhos de problemas muito mais altos o algoritmo tenha a sua escalabilidade.