Método de Monte Carlo

Método de Monte Carlo

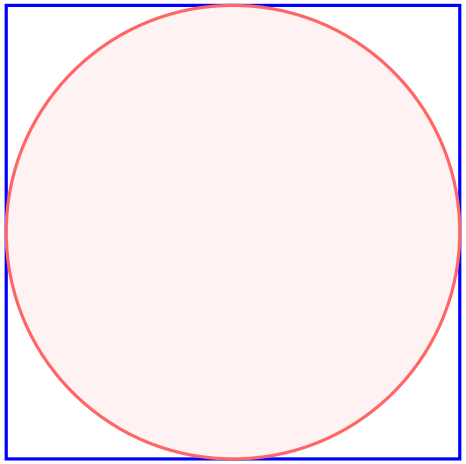
O valor de π pode ser calculado de várias maneiras. Considere o método de Monte Carlo de aproximação do PI:

- Considere um círculo com raio r inscrito num quadrado de lado $2r$.
- A área do círculo é πr^2 e a área do quadrado é $4r^2$.
- A razão entra a área do círculo e a área do quadrado é:

$$\pi r^2 / 4r^2 = \pi / 4$$

Método de Monte Carlo

Se você gerar n pontos dentro do quadrado, aproximadamente $n \cdot \pi/4$ cairá dentro do círculo.



Método de Monte Carlo

O valor de π pode ser obtido por:

$$\begin{aligned} n \cdot \pi/4 &= m \\ \pi/4 &= m/n \\ \pi &= 4 \cdot m/n \end{aligned}$$

onde m é o número de pontos que aleatoriamente cairam dentro do círculo.

calcula_pi_sequencial()

```
 1  numero_de_pontos = 10000
 2  pontos_no_circulo = 0
 3  para i = 1 até numero_de_pontos
 4       faça gere 2 números entre 0 e 1
 5       x = numero_aleatorio1
 6       y = numero_aleatorio2
 7       se (x, y) está dentro do círculo então
 8            pontos_no_circulo+ = 1
 9  pi = 4.0 * pontos_no_circulo/numero_de_pontos
10  devolva pi
```

Método de Monte Carlo

Isto leva a um algoritmo inerentemente paralelo (embarassingle parallel):

- Quebre o loop de modo que possa ser executado em diferentes tarefas simultâneas.
- Cada tarefa executa uma porção do loop um certo número de vezes.
- Todas as tarefas são independentes das demais
- A tarefa mestre recebe o resultado das demais usando primitiva *send* e *receive*.

calcula_pi_paralelo()

```
 1  numero_de_pontos = 100000000
 2  pontos_no_circulo = 0
 3  p = numero_de_tarefas
 4  r = numero_de_pontos/p
 5  para i = 1 até r
 6      faça gere 2 números entre 0 e 1
 7      x = numero_aleatorio1
 8      y = numero_aleatorio2
 9      se (x, y) está dentro do círculo então
10          pontos_no_circulo+ = 1
11  se id == 0 então ▷ mestre
12      receba dos escravos os pontos_no_circulo
13      calcule o valor de pi (use os valores do mestre e escravos)
14  senão ▷ escravo
15      envia para o mestre os pontos_no_circulo
```

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MESTRE 0

int esta_dentro_do_circulo(double x, double y){
   return ((x*x + y*y) <= 1.0);
}

int main(int argc, char* argv[]){
   int i, id, p, r;
   int tag = 5, source;
   int numero_de_pontos = 100000000;
   int pontos_no_circulo_local = 0, pontos_no_circulo = 0;
   double x, y;
```

```
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &p);
r = numero_de_pontos/p;
srand48(id);
pontos_no_circulo_local = 0;
for (i = 0; i < r; i++) {
   x = drand48();
   y = drand48();
   if (esta_dentro_do_circulo(x,y)) {
      pontos_no_circulo_local++;
   }
}
```

```
if (id == MESTRE) {
    pontos_no_circulo = pontos_no_circulo_local ;

    for (source = 1; source < p; source++) {
       MPI_Recv(&pontos_no_circulo_local, 1, MPI_INT,
                    MPI_ANY_SOURCE, tag, MPI_COMM_WORLD,&status);
       pontos_no_circulo += pontos_no_circulo_local;
    }

    printf("pontos no círculo: %d\n", pontos_no_circulo);
    double pi = (4.0 * pontos_no_circulo) / numero_de_pontos;
    printf("Pi = %24.16f\n", pi);
 }
```

```
  if (id != MESTRE) {   //  escravos
      MPI_Send(&pontos_no_circulo_local, 1, MPI_INT,
                                  MESTRE, tag, MPI_COMM_WORLD);
   }

   MPI_Finalize();
   return 0;
}
```

Compilação e Execução

Para compilar

```
$ mpicc monte_carlo.c -o monte_carlo
```

Para executar

```
$ mpirun -np 4 ./monte_carlo
```

Um exemplo de comunicação coletiva

Um exemplo de comunicação coletiva

Como calcular $1^2 + 2^2 + \cdots N^2$ utilizando MPI?

▷ dividir a computação entre p processadores.

▷ cada processador computa sua soma local de $(id * r + 1)^2 + (id * r + 2)^2 + \cdots + (id * r + 1 + r)^2$, onde id é o identificador do processo e $r = N/p$.

Um exemplo de comunicação coletiva

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define MESTRE 0

int main(int argc, char* argv[]){
   int id, p, n;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
if (id == MESTRE){
   printf("Entre com o valor de N: ");
   scanf("%d", &n);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

int soma_local = 0, soma_global = 0;
int r = n/p;
int inicio = (id * r) + 1;
int fim = inicio + r;

for(int i = inicio; i < fim; i++){
   soma_local += (i * i);
}
```

```
   MPI_Reduce(&soma_local, &soma_global, 1, MPI_INT,
                           MPI_SUM, 0, MPI_COMM_WORLD);

   if (id == MESTRE){
      printf("\n Total : %d\n", soma_global);
   }

   MPI_Finalize();
   return 0;
}
```

Compilação e Execução

Para compilar

```
$ mpicc soma_quadrado.c -o soma_quadrado
```

Para executar

```
$ mpirun -np 4 ./soma_quadrado
```

Algoritmo da Soma no modelo CGM

Algoritmo da Soma no modelo CGM

- Entrada: n elementos $v(1), v(2), \cdots, v(n)$
- Saída: $S = v(1) + v(2) + \cdots + v(n)$

  (1) p processadores

  (2) Cada processador recebe n/p elementos,
  efetua a soma localmente e envia o resultado para p_1

  (3) p_1 efetua a soma de $S = S_1 + S_2 + \cdots + S_p$

Algoritmo da Soma no modelo CGM

- Entrada: número do processador i;
  número p de processadores; $B = A((i-1)r + 1 : r)$;
  $r = n/p$
- Saída: p_i calcula $z = B(1) + B(2) + \cdots + B(n)$ e envia o resultado para p_1. p_1 calcula $S = z_1 + z_2 + \cdots + z_n$

(1) $z = B(1) + B(2) + \cdots + B(n)$

(2) se $i = 1$ então $S = z$
  caso contrário $envia(z, p_1)$

(3) se $i = 1$ então
  para $i = 2$ até p faça
    $receba(s[i], p)$

(4) $S = S_1 + S_2 + \cdots + S_p$

Algoritmo da Soma no modelo CGM

complexidade:

Passo 1: cada p_i efetua r operações

Passo 2: p_1 efetua uma operação e os demais p_i's enviam uma msg.

Passo 3: p_1 recebe $p-1$ mensagens.

Passo 4: p_1 efetua $p-1$ operações.

Um rodada de comunicação.

tempo : $O(n/p)$

Soma no modelo CGM - implementação

```
#include<mpi.h>

  // Passo 1. Inicializar
  // Passo 2. Envie os dados para as tarefas diferente de 0
  // Passo 3. Receba uma msg da tarefa 0
  // Passo 4. Calcule a soma
  // Passo 5. Receba as somas parciais
  // Passo 6. Imprima os valores da soma
  // Passo 7. Finalize o MPI
```

Passo 1

```
// Passo 1. Inicializar
MPI_init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

tam = (int)TAMMAX/size;      // tamanho do subvetor
```

Passo 2 - Enviando dados

```
// Passo 2. Envie os dados para as tarefas diferente de 0
if (rank == 0){
   for (i = 1; i < size; i++)  {
      for (j = 0; j < tam; j++) {
         subvetor[j] = vetor_dados[tam * i + j];
      }
      MPI_Send(subvetor, tam, MPI_INT, i, tag, MPI_COMM_WORLD);
   } // fim for

   for (j = 0; j < tam; j++) { // vetor da tarefa 0;
      subvetor[j] = vetor_dados[j];
   }
}
```

Passo 3 - Recebendo dados

```
// Passo 3. Receba uma msg da tarefa 0
if (rank != 0) {
   MPI_Recv(subvetor, tam, MPI_INT, 0, tag,
                           MPI_COMM_WORLD, &status);
}
```

Passo 4 - Efetuando a soma

```
// Passo 4. Calcule a soma
 for(i = 0; i < tam; i++) {
    somap = somap + subvetor[i];
 }
```

Passo 5 - Recebendo somas parciais

```
// Passo 5. Receba as somas parciais
MPI_Reduce(&somap, &soma, 1, MPI_INT,
                             MPI_SUM, 0, MPI_COMM_WORLD);
```

Passo 6 e 7 - Finalizando

```
// Passo 6. Imprima os valores da soma
if (rank == 0) {
   printf("soma: %d\n", soma);
}


// Passo 7. Finalize o MPI
MPI_Finalize();
return 0;
```

Compilação e Execução

Para compilar

```
$ mpicc soma.c -o soma
```

Para executar

```
$ mpirun -np 4 ./soma
```

Soma de prefixos

Soma de prefixos

Dado um vetor A de n elementos $A[0], A[1], \cdots, A[n-1]$ a computação de prefixos calcula os valores:

```
A[0]
A[0] op A[1]
A[0] op A[1] op A[2]
...
A[0] op ... op A[n-1]
```

onde op é uma operação binária associativa.

Soma de prefixos

- Exemplo: op é adição
- Vetor de entrada:

$$[3, 7, 5, 1, 2, 4, 0, 9]$$

- Vetor de saída:

$$[3, 10, 15, 16, 18, 22, 22, 31]$$

Soma de prefixos - CGM

- Entrada: n elementos $v(1), v(2), \cdots, v(n)$
- Saída: $S(k) = v(1) + \cdots + v(k)$, $1 \leq k \leq n$
  - (1) p_0 envia n/p elementos para cada p_i
  - (2) Cada p_i calcula a soma $T(i)$ de seus elementos e envia o resultado para os processadores $j > i$
  - (3) p_0 calcula a soma $ST(i) = T(1) + \cdots + T(i)$, $1 \leq i \leq p$
  - (4) Cada p_i calcula a soma de prefixos local
    - $S(k) = ST(i-1) + v((i-1) * r + 1) + \cdots + v(k)$,
    - $(i-1) * r + 1 \leq k \leq i * r$, $r = n/p$

Soma de prefixos - implementação

```
#include <mpi.h>


  // passo 1. Inicialização
  MPI_init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  tam = (int)TAMMAX/size;       // tamanho do subvetor
```

Soma de prefixos - implementação

```
// passo 2. Envia os dados as tarefas filhos
MPI_Scatter(vetor_dados, tam, MPI_INT, subvetor, tam,
                              MPI_INT, root, MPI_COMM_WORLD);


// passo 3. Calcula a soma em cada tarefa
for (i = 0; i < tam; i++)
   somap = somap + subvetor[i];
```

Soma de prefixos - implementação

```
// passo 4. Receba as somas parciais
MPI_Scan(&somap, &soma, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);


// passo 5. Calcula as somas parciais em cada tarefa
soma_pre[0] = soma - somap + subvetor[0]
for (i = 1; i < tam; i++)
   soma_pre[i] = soma_pre[i-1] + subvetor[i];
```

Soma de prefixos - implementação

```
// passo 6. Imprima o valor da soma
for (i = 0; i < tam; i++)
   printf("rank %d e soma_pre[%d] = %d\n", rank, i, soma_pre[i]);



// passo 7. Finalize o MPI
MPI_Finalize();
return 0;
```

Fim