UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
TÓPICOS ESPECIAIS EM INTERNET DAS COISAS “B” - T01
(2020.6)

Atividade 01 - Calcular o valor de Pi & Regra dos Trapézios

Paulo Eneas Rolim Bezerra

14 de outubro de 2020

1 INTRODUÇÃO

Os problemas propostos são calcular o número Pi e o valor aproximado da integral de uma função f(x) num determinado intervalo (a, b), por meio da regra dos trapézios.

Para calcular o número Pi foi utilizado o método Monte carlo que consiste em gerar pontos aleatórios e testar se eles estão dentro da área de uma circunferência unitária com origem no ponto zero do plano cartesiano.

A idéia da regra do trapézio é aproximar o valor da integral da função f(x), de ordem 2, a vários polinômios de ordem 1, desenhando diversos trapézios, e por meio do cálculo da área desses trapézios obtém-se o valor aproximado da integral da função f(x).

2. DESENVOLVIMENTO

2.1. Solução Serial Implementada em C++ para o cálculo de Pi

Para fazer o cálculo foram declaradas três variáveis uma do tipo inteiro para armazenar a precisão do cálculo que também é o tamanho do problema, e duas do tipo double uma para armazenar o valor ao final do cálculo e outra para auxiliar no laço de repetição durante o processo de cálculo, conforme visto na imagem abaixo.

```
1   #include <iostream>
2   #include <random>
3   #include <string>
4   #include <sys/time.h>
5
6   //função responsável por mostrar ao usuário como usar o programa
7   void como_usar(const std::string &programa)
8   {
9       std::cout << "Uso do programa " << programa << " precisao <valor> " << std::endl;
10  }
11
12  /*Cálculo serial de Pi utilizando o método de integração de Monte Carlo*/
13
14  int main(int argc, char const *argv[]){
15
16      int tamanho_problema;
17      double acumulador;
18      double pi_calculado;
19      std::string comando;
```

Foi desenvolvida uma interface simples para receber do usuário o tamanho do problema e testar se o valor informado serve para a realização do cálculo, vejamos na imagem abaixo:

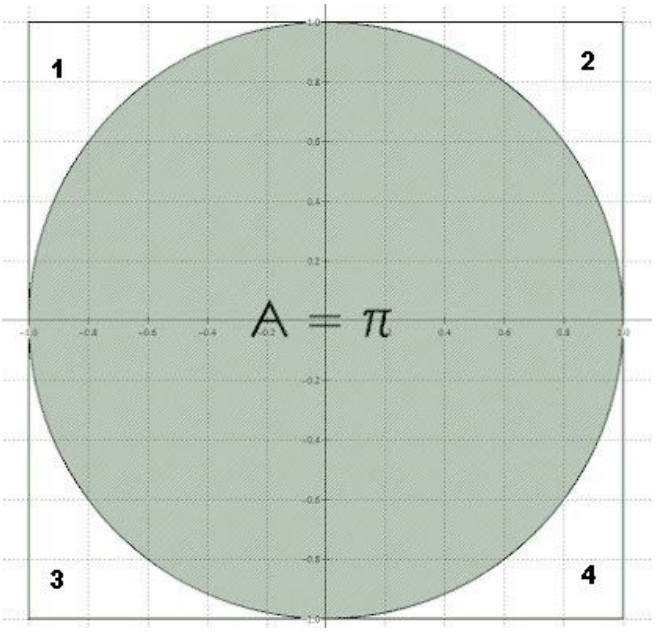
```
20
21      if (argc == 1)
22      {
23          como_usar(argv[0]);
24          return 1;
25      }
26
27      comando = argv[1];
28
29      if (comando != "precisao")
30      {
31          como_usar(argv[0]);
32          return 1;
33      }
34
35      if (argc == 2)
36      {
37          std::cout << "Informe agora o tamanho do problema para calcular PI" << std::endl;
38          std::cin >> tamanho_problema;
39      } else
40      {
41          comando = atoi(argv[2]);
42      }
```

O cálculo do valor de Pi pelo método de Monte Carlo foi implementado, nos moldes da imagem abaixo. Consiste no sorteio de dois números reais no intervalo entre os dois número naturais 0 e 1. O sorteio é realizado pelas funções da biblioteca *random* da linguagem C++, e armazenados em duas variáveis locais do tipo double.

```
44      struct timeval start, stop; /* código fornecido pelo professor para registrar o tempo de execução */
45      double tempo_exec;
46      gettimeofday(&start, 0);//início do registro do tempo de execução
47
48      /*No laço são sorteados dois número reais. É realizado um teste para saber se o número sorteado
49      está dentro da área circunferência de raio unitário*/
50      for (int i = 0; i <= tamanho_problema; i++)
51      {
52
53          std::random_device rd;
54          std::default_random_engine gen(rd());
55          std::uniform_real_distribution<> real_dis(0, 1);
56
57          double x_real_randomico = real_dis(gen);
58          double y_real_randomico = real_dis(gen);
59
60          if (x_real_randomico * x_real_randomico + y_real_randomico * y_real_randomico < 1.0)
61          {
62              acumulador++;
63          }
64
65      }
66
67      pi_calculado = (4.0 * acumulador) / tamanho_problema;
68
69      gettimeofday(&stop, 0); //fim do registro do tempo de execução
```

Logo em seguida é verificado se os números sorteados estão inseridos na área da circunferência unitária, ou seja, os números sorteados precisam satisfazer a seguinte inequação $x^2+y^2 < 1.0$, e o resultado dessa operação é armazenado na variável *acumulador*.

Para modelar o algoritmo imaginou-se que a circunferência de raio unitário está inserida num quadrado cujo lado tem tamanho 2, porém ao invés de utilizar toda a área da figura os pontos sorteados entre o intervalo (0, 1) estarão sempre no quadrante 2 da figura abaixo. Sendo assim o valor obtido será ¼ de Pi.



Após o sorteio e a testagem é feito o cálculo de Pi pela operação disposta na linha 67, imagem abaixo, onde multiplica-se por 4 o número de pontos encontrados dentro da área da circunferência e divide o resultado pela quantidade total de pontos sorteados e testados, atribuindo o resultado a variável double *pi_calculado*. Quanto maior o tamanho do problema, maior será a precisão do cálculo.

```
66
67      pi_calculado = (4.0 * acumulador) / tamanho_problema;
68
```

2.1.1. Solução Paralela Implementada em C++ para o Cálculo de Pi

Para realizar o cálculo de Pi de forma paralela foi utilizada a mesma lógica de sorteio e testes aplicada na solução serial. Todavia ao utilizar a biblioteca mpi.h alguns ajustes foram feitos, como pode ser visto no código. Foram declaradas as mesmas variáveis da versão serial, mas surgiu a necessidade de uma nova variável double *pi_local* para armazenar o valor gerado pelos processos paralelos.

```
1   #include <mpi.h>
2   #include <iostream>
3   #include <random>
4   #include <sys/time.h>
5
6   int main(int argc, char const *argv[]){
7
8       int tamanho_problema;
9       double acumulador;
10      double pi_global;
11      double pi_local;
12
13      int my_rank, comm_sz; // variáveis MPI
14
15      struct timeval start, stop; /* código fornecido pelo professor para registrar o tempo de execução */
16      double tempo_exec;
17
```

Conforme pode ser visualizado na figura abaixo, o processo mestre solicita ao usuário o valor do tamanho do problema (precisão), e em seguida realiza o envio do valor aos processos escravos por meio da função *MPI_Bcast*. Essa forma de envio da mensagem é uma das técnicas de comunicação coletiva da biblioteca MPI. Durante a execução do MPI_Bcast, um processo envia os mesmos dados para todos os processos em um comunicador, que no presente caso é o MPI_COMM_WORLD.

```
18      MPI_Init(NULL, NULL);
19      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
20      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
21
22      if (my_rank == 0)
23      {
24          std::cout << "Informe agora o tamanho do problema para calcular PI" << std::endl;
25          std::cin >> tamanho_problema;
26      }
27
28      /*função que envia para todos os processos o valor da variável tamanho_problema. Fazendo com que
29      todos os processos recebam o valor para iniciarem o sorteio dos números e o calculo de Pi*/
30      MPI_Bcast(&tamanho_problema, 1, MPI_INT, 0, MPI_COMM_WORLD);
31
32      if (tamanho_problema <= 0)
33      {
34          MPI_Finalize();
35          return 1;
```

Cada processo escravo realiza o cálculo seguindo a mesma lógica aplicada no cálculo serial, sorteando números reais no intervalo (0,1) e testando se satisfazem a equação $x^2+y^2 < 1.0$, e gravando os acertos na variável *acumulador*, e o cálculo de de Pi na variável *pi_local*.

Após o cálculo a função MPI_Reduce recebe a variável *pi_local* de cada processo, faz a soma dos resultados, e envia a saída para o processo mestre. Que por sua vez faz a impressão dos dados na tela.

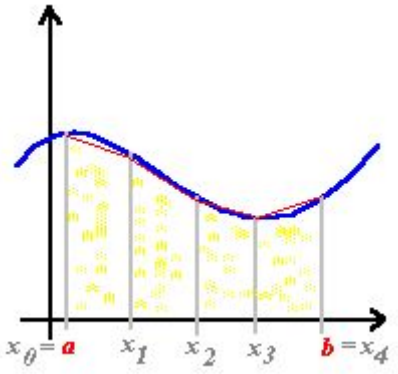
```
60
61          /*função responsável por somar e reduzir todos os dados gerados nos processos paralelos em um
62          conjunto menor de números. Na prátia essa função está recebendo todos os valores calculados
63          somando e atribuindo a variável global pi_global*/
64          MPI_Reduce(&pi_local, &pi_global, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
65
66          gettimeofday(&stop, 0); //fim do registro do tempo de execução
67
68          if (my_rank == 0)
69          {
70              std::cout << "O valor aproximado de Pi é: " << pi_global << std::endl;
71
72              tempo_exec = (double)(stop.tv_usec - start.tv_usec) / 1000000 + (double)(stop.tv_sec - start.tv_
73
74              std::cout << "O tempo de execução foi " << (std::scientific) << tempo_exec << std::endl;
75          }
76
77      }
78
79      MPI_Finalize();
80      return 0;
81  }
```

2.2. Solução Serial Implementada em C++ para a Regra dos Trapézios

No algoritmo foram declaradas duas variáveis do tipo double para armazenar o início e o final do intervalo, uma variável do tipo inteiro para armazenar a quantidade de trapézios, ou seja, o tamanho do problema, uma variável para armazenar o valor total da Integral e outra para servir de incremento para cada repetição do laço de iteração. Os valores iniciais das variáveis foram configurados manualmente no código, conforme imagem abaixo:

```
13  int main(int argc, char *arrgv[]){
14
15      double x_a, x_b; //início e fim do intervalo
16      double Int; //valor da integral;
17      double incremento;
18      int n; //quantidade de trapézios ou tamanho do problema
19
20      struct timeval start, stop; /* código fornecido pelo professor para registrar o tempo de execução */
21      double tempo_exec;
22      gettimeofday(&start, 0);//início do registro do tempo de execução
23
24
25      /*Set manual dos valores de entrada*/
26      x_a = 0.0; //ponto inicial
27      x_b = 1.0; //ponto final
28      n = 200; //quantidade de trapézios ou tamanho do problema
```

Além disso foi definida uma função contendo o f(x) = x² + x + 2, que será utilizada para realizar o cálculo aproximado da integral.



A lógica aplicada consiste em subdividir o intervalo (*x_a, x_b*) em n trapézios e armazenando o valor na variável *incremento* em seguida, em seguida o laço altera o ponto final (*x_b*) para o final da primeira das n subdivisões calculadas fora do laço. Em seguida é realizado o cálculo da área e o valor da variável *x_b* é atribuído a variável *x_a*, justamente para demarcar o próximo trapézio a ter a área calculada, e esse laço é repetido *n* vezes. O cálculo da área dos *n* trapézios vai sendo armazenada na variável *Int* a cada passagem do laço. A implementação pode ser vista na imagem abaixo:

```
30      incremento = (x_b - x_a)/n; //subdivide o cálculo conforme a quantidade de trapézios selecionada
31
32      Int = 0.0;
33
34      for (int i = 0; i < n; i++)
35      {
36          x_b = x_a + incremento; //diz até que ponto do intervalo o cálculo será feito
37
38          Int = Int + (x_b - x_a)*(funcao(x_a) + funcao(x_b))/2.0; //cálculo da área do trapézio
39
40          x_a = x_b; //atualiza a variável para o próximo vértice do trapézio ou ponto do intervalo
41      }
```

Ao final o valor da área e o tempo de execução são exibidos na tela do terminal.

2.2.1 Solução Paralela Implementada em C++ para a Regra dos Trapézios

Para implementar o código paralelo, utilizou-se a mesma lógica do algoritmo serial, porém subdividindo entre os processos a carga de trabalho, consoante figura abaixo:

```
34      /*Área Paralela*/
35      MPI_Init(NULL, NULL);
36      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); //Tamanho do comunicador ou quantos processos inicializados
37      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //Número do processo em execução
38
39      gettimeofday(&start, 0);//início do registro do tempo de execução
40
41      incremento = (x_b - x_a)/n_global;
42
43      n_local = n_global/comm_sz; //divisão do tamanho do problema pela quantidade de processos
44
45      Int_local = 0.0;
46
47      for (int i = 0; i < n_local; i++)
48      {
49          x_b = x_a + incremento; //diz até que ponto do intervalo o cálculo será feito
50
51          Int_local = Int_local + (x_b - x_a)*(funcao(x_a) + funcao(x_b))/2.0; //cálculo da área do trapézio
52
53          x_a = x_b; //atualiza a variável para o próximo vértice do trapézio ou ponto do intervalo
54      }
55
56      if (my_rank != 0)
57      {
58          MPI_Send(&Int_local, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD); //cada processo envia o resultado do cálculo
59
```

Cada processo escravo realiza uma parcela do cálculo, e o envia para o processo mestre por meio da função *MPI_Send* . O processo mestre executa um laço de repetição com a função *MPI_Recv* para receber as mensagens de todos os processos e vai somando os resultados na variável *Int_global*. Ao final o processo mestre exibe na tela o resultado do cálculo.

```
56      if (my_rank != 0)
57      {
58          MPI_Send(&Int_local, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD); //cada processo envia o resultado do cálculo
59
60      } else
61      {
62          Int_global = Int_local;
63
64          for (int q = 1; q < comm_sz; q++)
65          {
66              MPI_Recv(&Int_local, 1, MPI_DOUBLE, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); //processo mestre rec
67
68              Int_global += Int_local;
69
70          }
71
72      }
```

3. RESULTADOS

Foram realizados testes com os códigos serial e paralelo, com tamanhos de problema diferentes e o tempo de execução medido, conforme tabelas abaixo:

Tabela de Tempo de Execução - Cálculo Pi Serial		
	Tamanho do Problema	Tempo médio
Execução 01	2500000	0,005975
Execução 02	3000000	0,006597
Execução 03	3500000	0,006949
Execução 04	5000000	0,007818

Tabela de Tempo de Execução - Cálculo Pi Paralelo			
	Tamanho do Problema	Cores	Tempo médio
Execução 01	2500000	2	12,151916
Execução 02	3000000	2	14,238300
Execução 03	3500000	2	17,109610
Execução 04	5000000	2	24,753967

Tabela de Tempo de Execução - Cálculo Pi Paralelo			
	Tamanho do Problema	Cores	Tempo médio
Execução 01	2500000	4	7,944534
Execução 02	3000000	4	9,665454
Execução 03	3500000	4	11,33180
Execução 04	5000000	4	16,01014

Tabela de Tempo de Execução - Cálculo Pi Paralelo			
	Tamanho do Problema	Cores	Tempo médio
Execução 01	2500000	8	6,5729840
Execução 02	3000000	8	8,7018320
Execução 03	3500000	8	9,7722380
Execução 04	5000000	8	12,381988

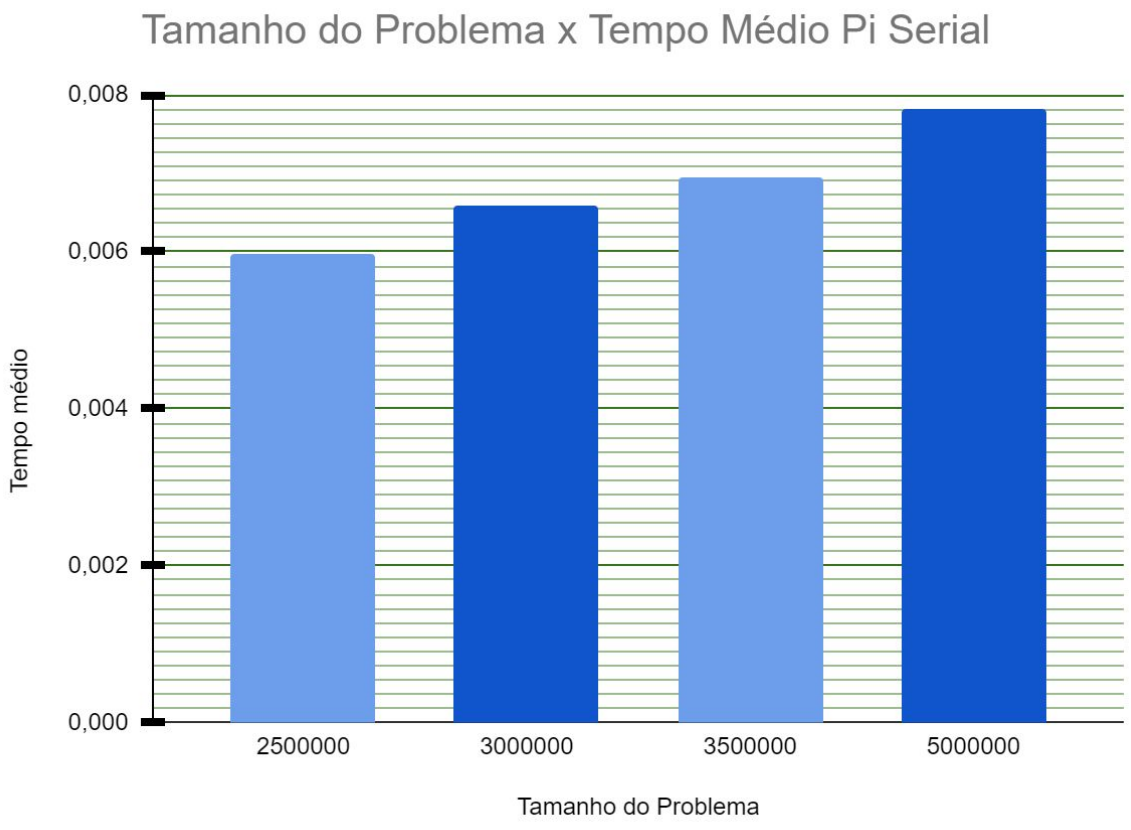
Tabela de Tempo de Execução - Regra dos Trapézios Serial		
	Tamanho do Problema	Tempo médio
Execução 01	2500000	0,0311655
Execução 02	3000000	0,0371026
Execução 03	3500000	0,0429825
Execução 04	5000000	0,0613903

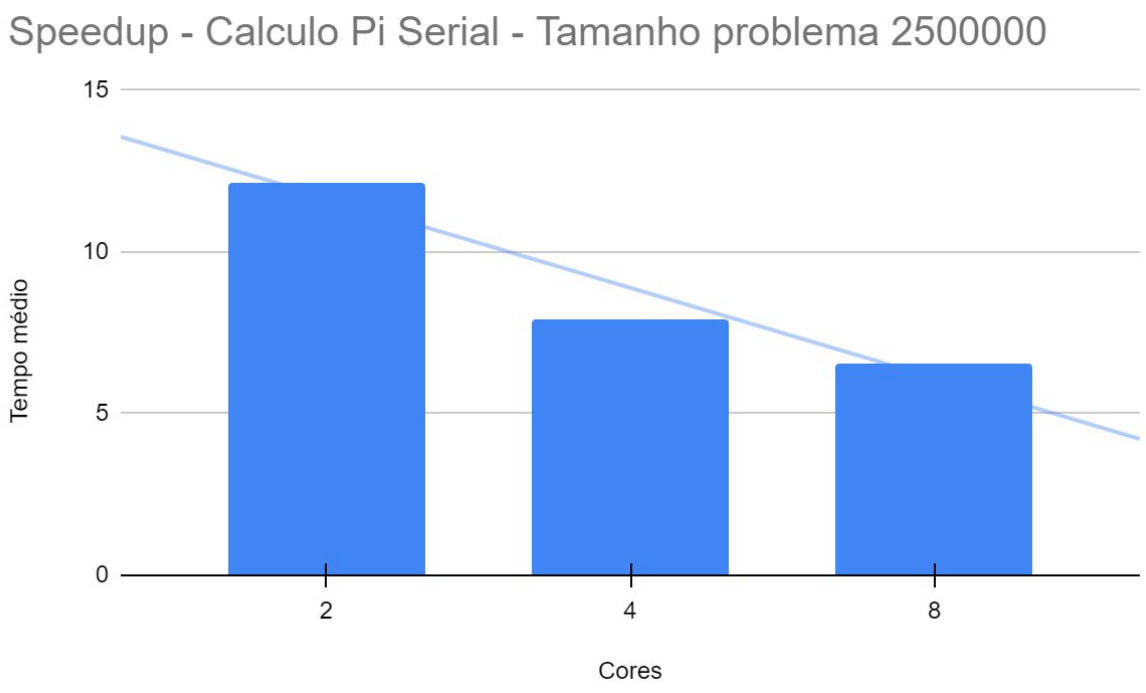
Tabela de Tempo de Execução - Regra dos Trapézios Paralelo			
	Tamanho do Problema	Cores	Tempo médio
Execução 01	2500000	2	0,0162900
Execução 02	3000000	2	0,0391720
Execução 03	3500000	2	0,0401112
Execução 04	5000000	2	0,0515822

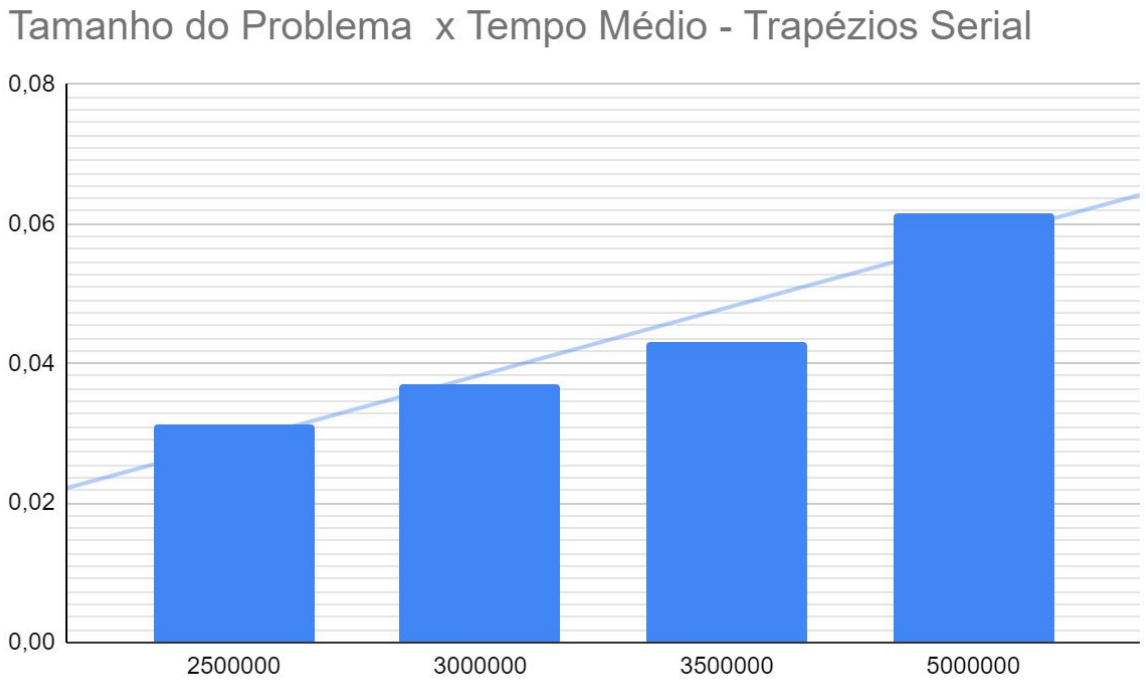
Tabela de Tempo de Execução - Regra dos Trapézios Paralelo			
	Tamanho Problema	Cores	Tempo médio
Execução 01	2500000	4	0,015472
Execução 02	3000000	4	0,021194
Execução 03	3500000	4	0,030117
Execução 04	5000000	4	0,037061

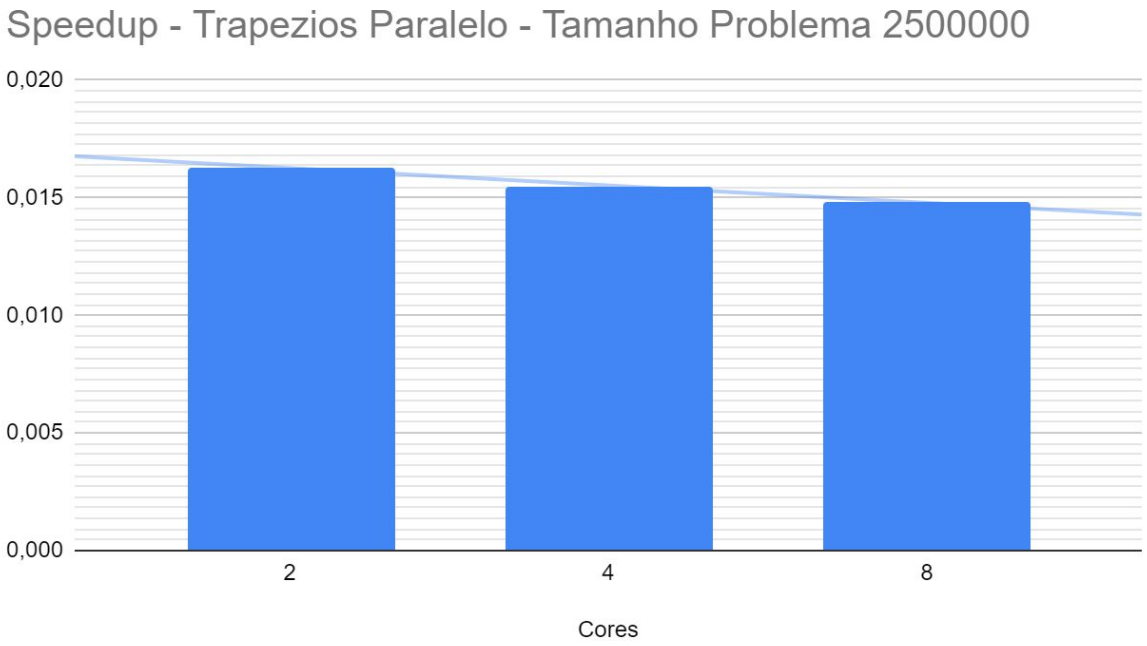
Tabela de Tempo de Execução - Regra dos Trapézios Paralelo			
	Tamanho Problema	Cores	Tempo médio
Execução 01	2500000	8	0,0148110
Execução 02	3000000	8	0,0170697
Execução 03	3500000	8	0,0258502
Execução 04	5000000	8	0,0336290

O computador de testes roda o Sistema Operacional Ubuntu 20.04.1 LTS, usa a placa mãe da fabricante Gigabyte modelo 970A-DS3P com o Base Clock (BCLK) setado em 201,51 Mhz, o processador é AMD-FX8320e de 8 núcleos e 16mb de memória cache com overclock para 3.5 Ghz, 16gb memória ram PC3-10700H DR3 O.C. 1600 MHz da fabricante Corsair com as seguintes latências principais 9, 9, 9, 24, 41.




Speedup - Calculo Pi Serial - Tamanho problema 2500000
15
10
5
0
Tempo médio
2
4
8
Cores


Tamanho do Problema x Tempo Médio - Trapézios Serial
0,08
0,06
0,04
0,02
0,00
2500000
3000000
3500000
5000000


Speedup - Trapezios Paralelo - Tamanho Problema 2500000
0,020
0,015
0,010
0,005
0,000
2
4
8
Cores