



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO
TÓPICOS ESPECIAIS EM INTERNET DAS COISAS “B”
COMPUTAÇÃO PARALELA - (IMD0291)
PROF. KAYO GONCALVES E SILVA
2020.6

Relatório

DAWERTON EDUARDO CARLOS VAZ

Aplicação de computação paralela utilizando MPI

SUMÁRIO

Introdução	3
Problemas	3
PI pelo Método Monte Carlo	3
Como verificar se um ponto está dentro da circunferência?	5
Modelando a solução do problema	5
Implementação do Código	7
Serial	7
Paralelo	8
Execução dos códigos	10
Máquina usada nos testes	10
Código serial	10
Testes	11
Código Paralelo	12
Testes	13
Análise de Speedup, Eficiência e Escalabilidade	16
SpeedUp	16
Eficiência e Escalabilidade	17
Conclusão	18
Regra dos Trapézios	18
Implementação do Código	20
Função usada	20
Serial	21
Paralelo	22
Execução dos códigos	23
Máquina usada nos testes	23
Código serial	24
Testes	25
Código Paralelo	26
Testes	27
Análise de Speedup, Eficiência e Escalabilidade	30
SpeedUp	30
Eficiência e Escalabilidade	31
Conclusão	32

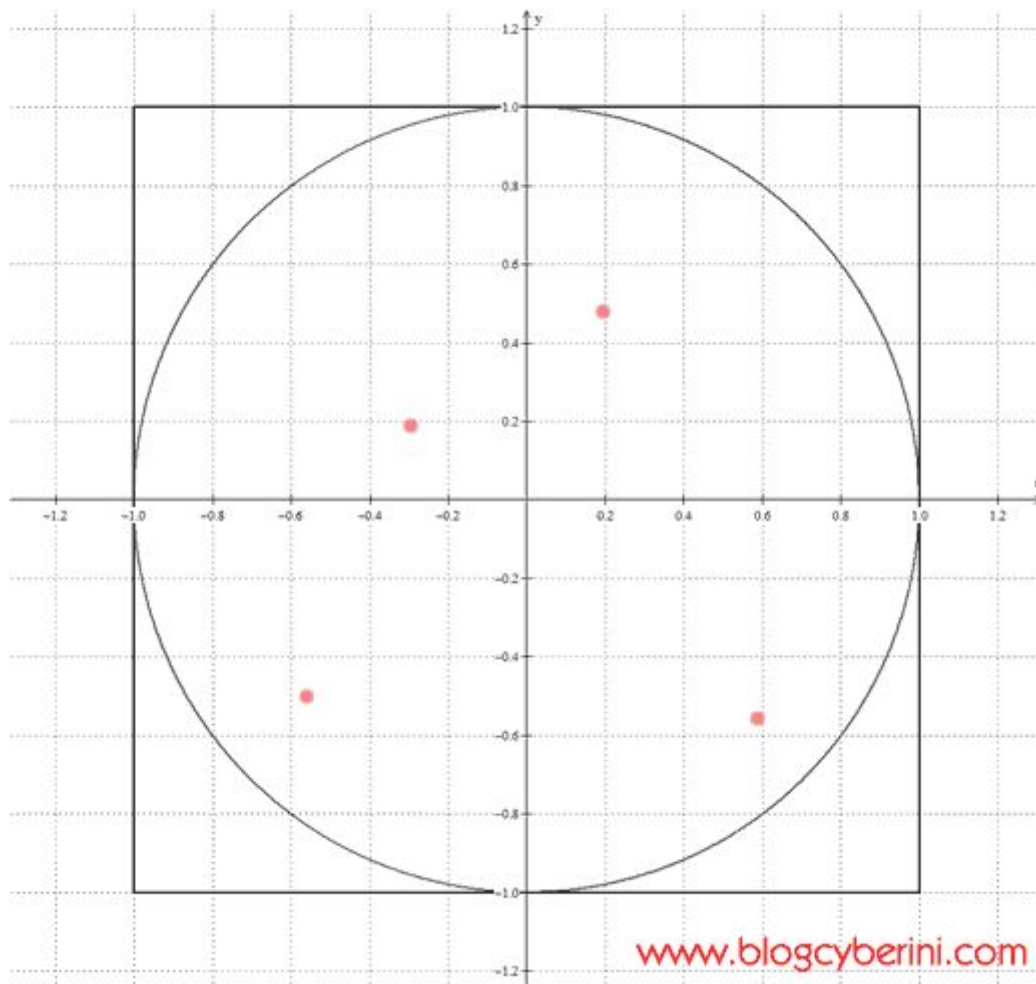
Introdução

Problemas

Neste relatório aplicamos o método MPI (Message Passing Interface) em dois algoritmos diferentes. O algoritmo que calcula o número PI através do método de monte carlo. E a aplicação da regra dos trapézios para calcular a área de uma função aplicada a dois pontos distintos.

PI pelo Método Monte Carlo

Método de Monte Carlo é um termo utilizado para se referir a *qualquer método que resolve um problema gerando números aleatórios e observando se uma dada fração desses números satisfaz uma propriedade previamente estabelecida*. Para calcular a área da circunferência unitária, utilizaremos o método de integração de Monte Carlo. A ideia é colocar a circunferência dentro de uma figura, cuja área seja fácil de calcular, e sortear pontos aleatórios dentro da figura. Utilizaremos um quadrado.



Se o ponto sorteado estiver dentro da circunferência, então marcamos um acerto. Ao final dos sorteios, espera-se que a área da circunferência seja proporcional à taxa de acertos e à área do quadrado. Esse valor será uma aproximação para π .

"Espera-se" porque o método é estocástico, isto é, os resultados obtidos são aleatórios. Porém, para uma grande amostra de pontos, ele deve convergir para o resultado esperado, ou seja, π .

A taxa de acertos é a razão entre o número de acertos e o total de sorteios realizados. Este valor pode ser visto como uma aproximação da probabilidade de sortear um ponto e ele estar dentro da circunferência.

A probabilidade exata é igual a razão entre a área da circunferência e a área do quadrado

$$P = \frac{A_{\text{circ}}}{A_{\text{quad}}}$$

Onde P é a probabilidade do ponto estar dentro da circunferência, A_{circ} é a área da circunferência e A_{quad} é a área do quadrado. Na circunferência unitária, temos $A_{\text{circ}} = \pi$, que implica em

$$P = \frac{\pi}{A_{\text{quad}}}$$

$$\pi = P \cdot A_{\text{quad}}$$

Suponha que sorteamos N pontos e que N_{acertos} é o número de acertos, onde

$N_{\text{acertos}} \leq N$. Então, P será aproximadamente igual a $\frac{N_{\text{acertos}}}{N}$ portanto o valor aproximado de π é dado por

$$\pi \approx \frac{N_{\text{acertos}}}{N} \cdot A_{\text{quad}}$$

Quanto maior o número de sorteios, melhor será a aproximação.

Como verificar se um ponto está dentro da circunferência?

Por simplicidade, vamos utilizar a circunferência unitária com centro na origem, definida pela equação

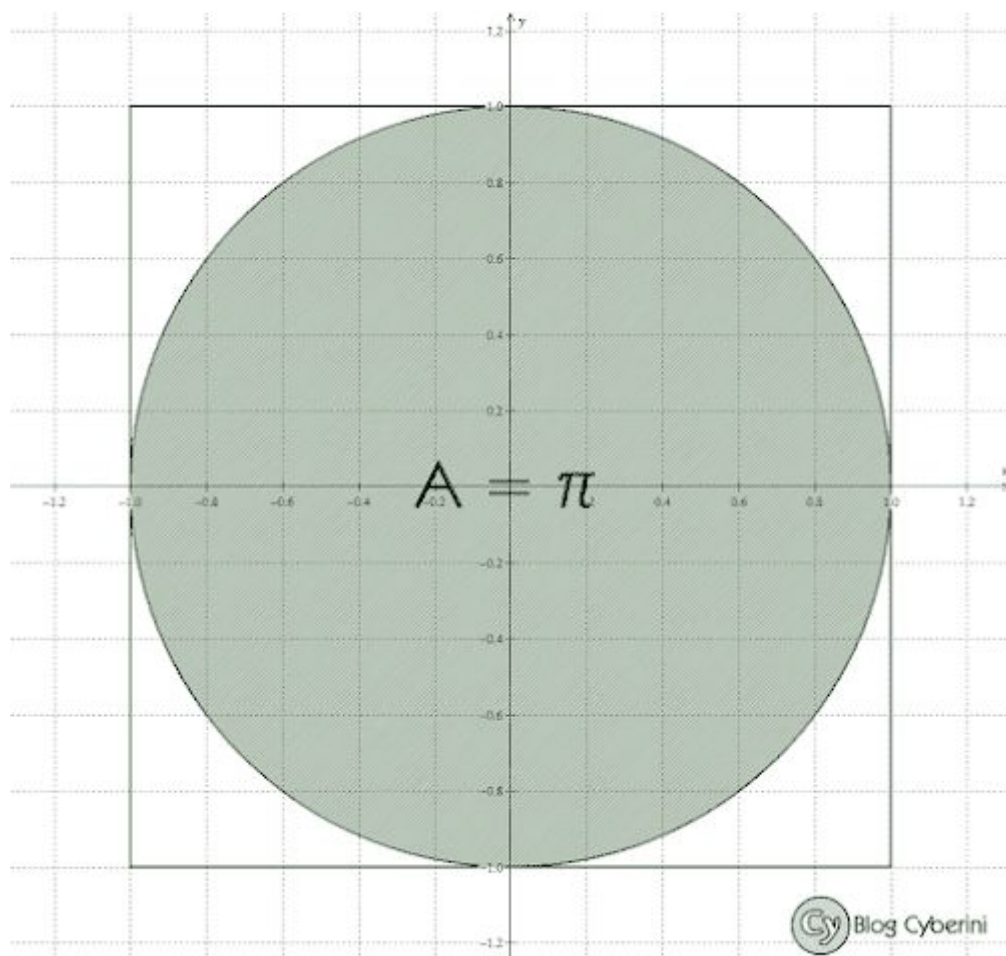
$$x^2 + y^2 = 1$$

Dado um ponto (x,y) , se o ponto está dentro da circunferência, então ele deve satisfazer a seguinte condição

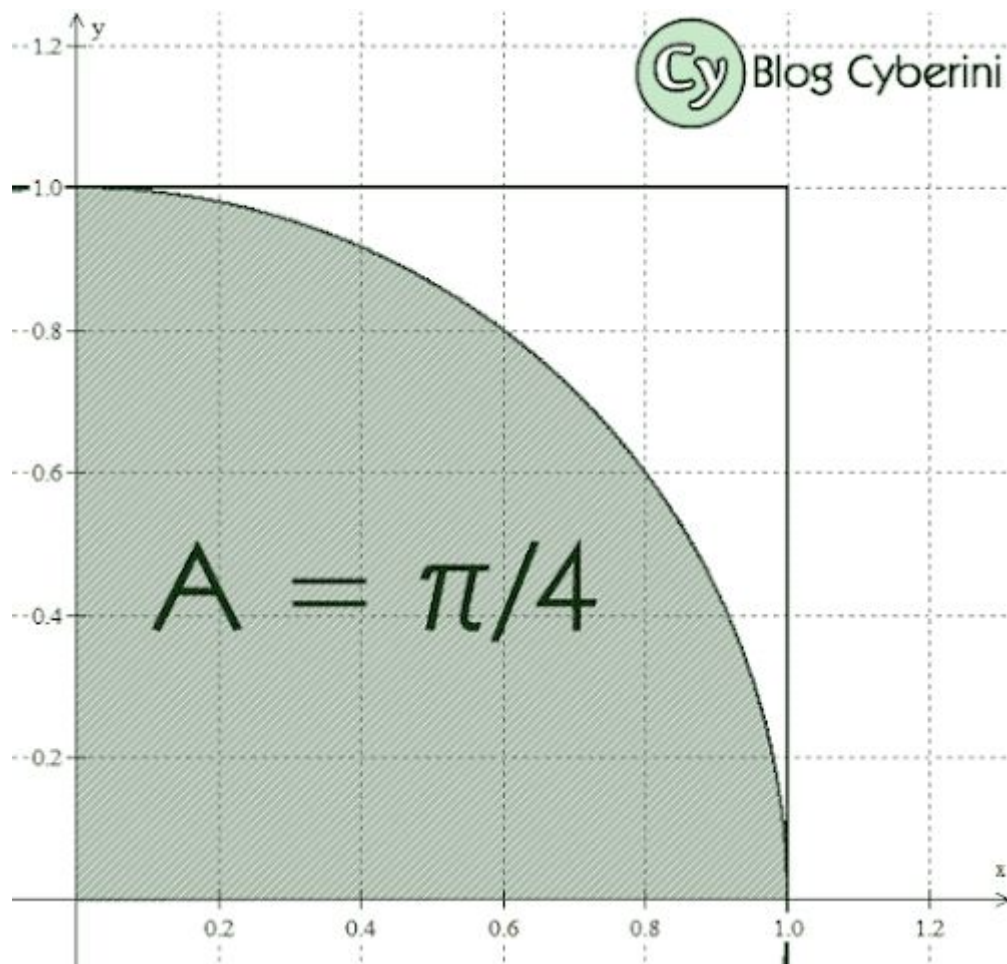
$$x^2 + y^2 < 1$$

Modelando a solução do problema

Para os cálculos, escolheremos um quadrado de lado 2 circunscrito na circunferência de raio 1.



No entanto, ao invés de utilizarmos as figuras completas, utilizaremos apenas a parte delas situada no primeiro quadrante. Em outras palavras, teremos 1/4 de circunferência dentro de um quadrado de lado 1.



Ou seja, ao invés de estimarmos π , estimaremos $\pi/4$. Depois é só multiplicar a aproximação por quatro para obter π .

Faremos isso por pura conveniência, pois normalmente as linguagens de programação oferecem métodos/funções que calculam números reais aleatórios entre 0 e 1 (na verdade, pseudo aleatórios).

O algoritmo em pseudocódigo é

```
01. monteCarloPi(n)
02. |   acertos ← 0
03. |   para i ← 0 até n
04. |   |   x ← sorteie um número real entre 0 e 1
05. |   |   y ← sorteie um número real entre 0 e 1
06. |   |   se(x * x + y * y < 1)
07. |   |   |   acertos ← acertos + 1
08. |   |   fim_se
09. |   fim_para
10. |   retorne 4 * acertos / n
11. fim_monteCarloPi
```

Implementação do Código

Serial

A seguir implementamos a função do método de monte carlo serial na linguagem c. O código gera um ponto no espaço formado por (x,y) e calcular sua distância até o ponto (0,0). Caso o valor seja menor do que 1.0, a variável count será incrementada e servirá como um contador de pontos que estão dentro da esfera de raio unitário. Por fim, PI é calculado e seu valor retornado à função principal.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double montecarlo(double n) {
    srand( SEED );
    int i, count;
    double x,y,z,pi;
    count = 0;
    for(i = 0; i < n; ++i) {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        z = x * x + y * y;
        if( z <= 1 ) count++;
    }
    pi = (double) count/n * 4;
    return(pi);
}
```

Em Seguida temos o código para executar a função e depois adicionar em um arquivo o tempo de execução.

O código guarda o início da marcação de tempo na variável start e depois de rodar a função marca o final da execução na variável stop, depois criar uma arquivo e escreve a diferença do start e stop, marcando assim o tempo de de execução da função.

```
int main(int argc, char **argv) {
    struct timeval start, stop;
    gettimeofday(&start, 0);
    montecarlo(atof(argv[1]));
    gettimeofday(&stop, 0);

    FILE *fp;
    char outputFilename[] = "tempo_de_mm.txt";
    fp = fopen(outputFilename, "a");
    if (fp == NULL) {
        fprintf(stderr,"Can't open output file %s!\n",
outputFilename);
    }
}
```

```

        exit(1);
    }
    fprintf(fp, "\t%f ", (double)(stop.tv_usec - start.tv_usec) /
1000000 + (double)(stop.tv_sec - start.tv_sec));
    fclose(fp);
    return 0;
}

```

Paralelo

A seguir implementamos a função do método de monte carlo paralelo na linguagem c. O problema é dividido pelo número de processos com o método MPI, em seguida cada um dos processos calcula os acertos e manda para o processo ZERO pra ele fazer o cálculo do PI.

```

#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <math.h>
#include <time.h>
MPI_Status status;
#define SEED time(NULL)
int main(int argc, char **argv)
{
    struct timeval start, stop;
    int n = atoi(argv[1]);
    gettimeofday(&start, 0); //começa a contagem do tempo
    int numtasks, taskid;
    srand( SEED );
    int i, count, mycount;
    double x, y, z;
    double pi=0;
    mycount = 0;

    MPI_Init(&argc, &argv); //Iniciamos a região paralela
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid); //definimos o numero de cada um
dos processos
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks); //numero de processos
    for(i = 0; i < (n/numtasks); ++i) //cada um dos processos calcula uma
parte do problema
    {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;

```



```

        z = x * x + y * y;
        if( z <= 1 ) mycount++;
    }

    if (taskid == 0) //Processo 0
    {
        count = mycount;
        for (int proc=1; proc<numtasks; proc++) {

MPI_Recv(&mycount,1,MPI_DOUBLE,proc,0,MPI_COMM_WORLD,&status);//O
processo ZERO recebe os outros resultados
            count +=mycount;

        }
        pi=(double)count/(n)*4;//calculo do pi
        gettimeofday(&stop, 0);//Para a contagem do tempo
    }
    else
    {
        MPI_Send(&mycount,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);//Cada processo
envia o resultado para o ZERO
    }

    if (taskid == 0) //O processo ZERO escreve o tempo de execução no
arquivo
    {
        FILE *fp;
        char outputFilename[] = "tempo_de_mm.txt";

        fp = fopen(outputFilename, "a");
        if (fp == NULL) {
            fprintf(stderr, "Can't open output file %s!\n",
outputFilename);
            exit(1);
        }

        fprintf(fp, "\t%f ", (double)(stop.tv_usec - start.tv_usec) /
1000000 + (double)(stop.tv_sec - start.tv_sec));

        fclose(fp);
    }

MPI_Finalize();//fecha as interações dos processos
    return 0;
}

```

Execução dos códigos

Máquina usada nos testes

Nos testes foi utilizado um notebook da marca asus de modelo ASUS Z450UA-WX010 com as seguintes configurações:

Processador Intel(R) Core(TM) i3-6100U CPU @ 2.30GHz

Número do processador: i3-6100U

Número de núcleos: 2

Nº de threads: 4

Memória ram: 4.0 GB

Armazenamento: SSD 241.1 GB

Sistema operacional: Linux Mint 20 Cinnamon versão 4.6.7

Código serial

Para os testes do código serial foi utilizado o seguinte código em Shell Script.

```
#!/bin/bash
rm tempo_de_mm.txt #apagar arquivos temporários
rm mpi_mm #apagar arquivos temporários
gcc -g -Wall mpi_mm.c -o mpi_mm #Compilação de Código.
#OBRIGATÓRIO: Laço de iteração para resgate dos tempos de acordo com
"cores" e "size"
#Loop principal de execuções. São 10 tentativas
tentativas=5 #Quantas vezes o código
for cores in 1 #números de cores utilizados
do
    for size in 1000000000 1200000000 1400000000 1600000000
    #tamanho do problema
    do
        echo -e "\n$cores\t$size\t\t\t\t\t" >> "tempo_de_mm.txt"

        for tentativa in $(seq $tentativas) #Cria uma vetor de 1
a "tentativas"
        do
            #Executar o código.
            ./mpi_mm $size
        done
    done
done
exit
exit
```

O Shell Script executa o código serial 5 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código serial que escreve o tempo que levou para executar logo depois.

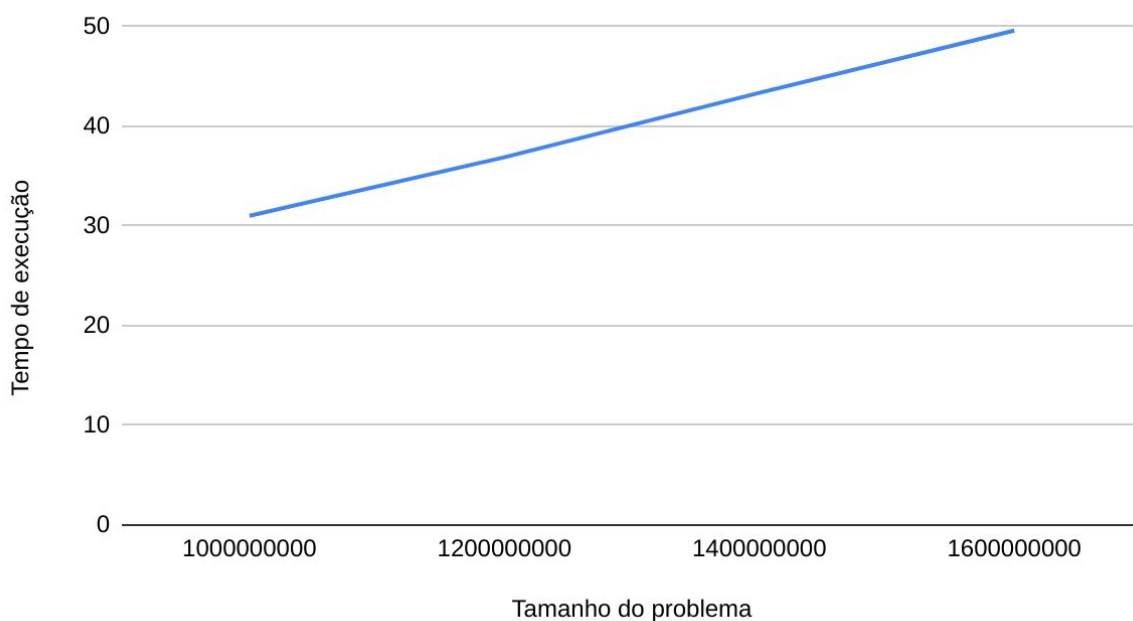
Testes

Ao executar o códigos 5 vezes em cada tamanho do problema fiz uma média como mostra a tabela a seguir

Código Serial	
Tamanho do problema	Tempo de execução (s)
1000000000	31,021515
1200000000	36,895995
1400000000	43,360778
1600000000	49,605349

No gráfico a seguir observa-se que a cada vez que o tamanho do problema aumentar, o tempo para execução ficar maior.

Tempo de execução x Tamanho do problema



Código Paralelo

Para os testes do código paralelo foi utilizado o seguinte código em Shell Script.

```
#!/bin/bash

#OPCIONAL: apagar arquivos temporários (gerados ou não pelo .c).
rm tempo_de_mm.txt
rm mpi_mmcoppy

#Compilação de Código. Modifique para o que mais se adequa a você.
mpicc -g -Wall mpi_mmcoppy.c -o mpi_mmcoppy

#OBRIGATÓRIO: Laço de iteração para resgate dos tempos de acordo com
"cores" e "size"

#Loop principal de execuções. São 10 tentativas
tentativas=5 #Quantas vezes o código será executado dado um par
(cores,size)
for cores in 1 2 3 4 8 #números de cores utilizados
do
    for size in 1000000000 1200000000 1400000000 1600000000
#tamanho do problema
do
    echo -e "\n$cores\t$size\t\t\c" >> "tempo_de_mm.txt"

    for tentativa in $(seq $tentativas) #Cria uma vetor de 1
a "tentativas"
do
        #Executar o código. Modifique para o que mais se
adequa a você.

        mpirun -np $cores ./mpi_mmcoppy $size
        #mpirun ./mpi_mm
        #no meu PC, tive que utilizar "--oversubscribe" para
rodar. No seu pode não ser necessário.

    done

done

done
exit
```

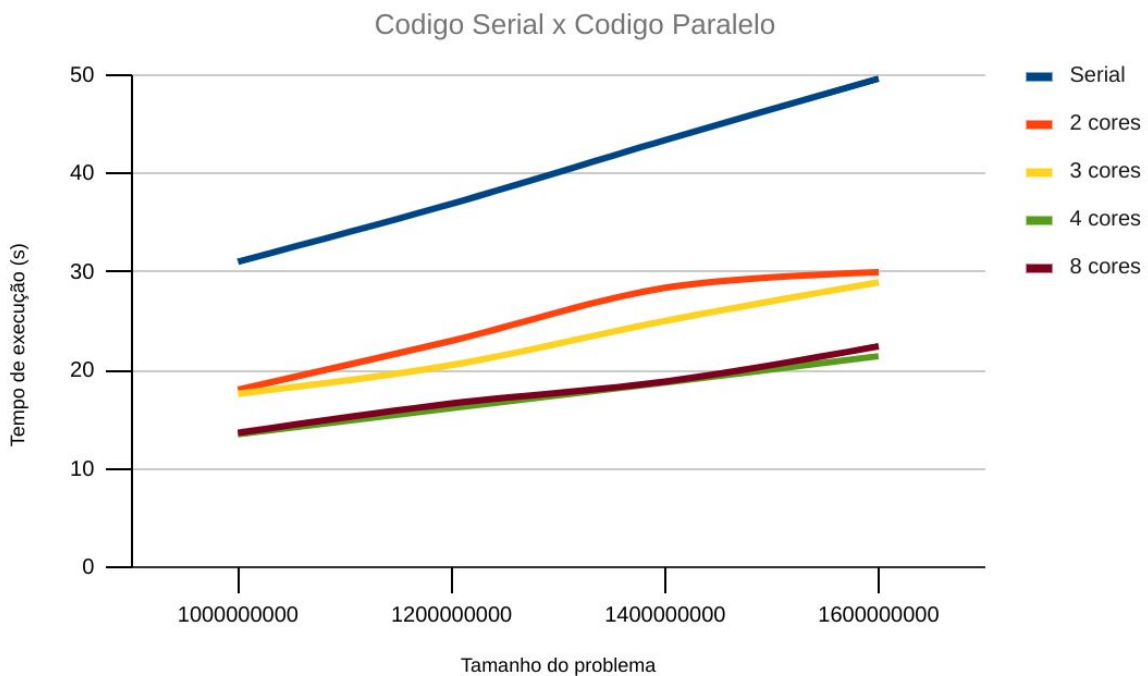
O Shell Script executa o código paralelo 5 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código paralelo que escreve o tempo que levou para executar logo depois.

Testes

Ao executar o códigos 5 vezes em cada tamanho do problema fiz uma média como mostra a tabela a seguir

Nº de Cores	Tamanho do Problema	Tempo de execução (s)
Serial	1000000000	31,021515
Serial	1200000000	36,895995
Serial	1400000000	43,360778
Serial	1600000000	49,605349
2	1000000000	18,015364
2	1200000000	22,993633
2	1400000000	28,369306
2	1600000000	29,942837
3	1000000000	17,612865
3	1200000000	20,529306
3	1400000000	25,011405
3	1600000000	28,924482
4	1000000000	13,513448
4	1200000000	16,170236
4	1400000000	18,762078
4	1600000000	21,427537
8	1000000000	13,652799
8	1200000000	16,628170
8	1400000000	18,857380
8	1600000000	22,430831

No gráfico a seguir está a comparação do código serial com o paralelo, da pra observar que utilizando 2 cores o desempenho aumenta quase 100%, e depois com 3 cores ele so tem um pequeno aumento no desempenho, e depois utilizando 4 cores o desempenho aumentar novamente, e com 8 cores não tem muita diferença dá pra ver que no final o problema utilizando 4 cores ainda tem um desempenho melhor, considerando que o processador do notebook utilizado é 2 cores e 4 threads percebe se que o aumento de desempenho se limita ao número de threads do processador.



A seguir alguns exemplos dos códigos rodando no terminal do linux mint.

Execução do código serial, vale lembrar que na execução para os testes não teve nenhum tipo de escrita na tela para não atrapalhar no desempenho

```
eduardovaz@eduardo-pc: ~/Programação/MPI/PI/Serial
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
eduardovaz@eduardo-pc:~/Programação/MPI/PI/Serial$ ./shellscript_start.sh
PI=3.141659
PI=3.141519
PI=3.141608
PI=3.141622
PI=3.141616
PI=3.141688
PI=3.141648
PI=3.141578
PI=3.141609
PI=3.141569
PI=3.141614
PI=3.141565
PI=3.141570
PI=3.141578
PI=3.141560
PI=3.141500
PI=3.141612
PI=3.141619
PI=3.141581
PI=3.141637
eduardovaz@eduardo-pc:~/Programação/MPI/PI/Serial$
```

Execução do código paralelo, vale lembrar que na execução para os testes não teve nenhum tipo de escrita na tela para não atrapalhar no desempenho.

```
eduardovaz@eduardo-pc: ~/Programação/MPI/PI/Paralelo
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
eduardovaz@eduardo-pc:~/Programação/MPI/PI/Paralelo$ ./shellscript_start.sh
PI=3.141680
PI=3.141510
PI=3.141541
PI=3.141822
PI=3.141295
PI=3.141609
PI=3.141571
PI=3.141601
PI=3.141596
PI=3.141659
PI=3.141546
PI=3.141584
PI=3.141624
PI=3.141622
PI=3.141624
PI=3.142058
PI=3.141525
PI=3.141872
PI=3.141353
PI=3.141816
PI=3.141528
PI=3.141628
PI=3.141716
```

Análise de Speedup, Eficiência e Escalabilidade

SpeedUp

O speedup paralelo S definido como a relação entre o tempo de processamento serial T_s de um algoritmo e seu tempo de processamento paralelo T_p , tal que

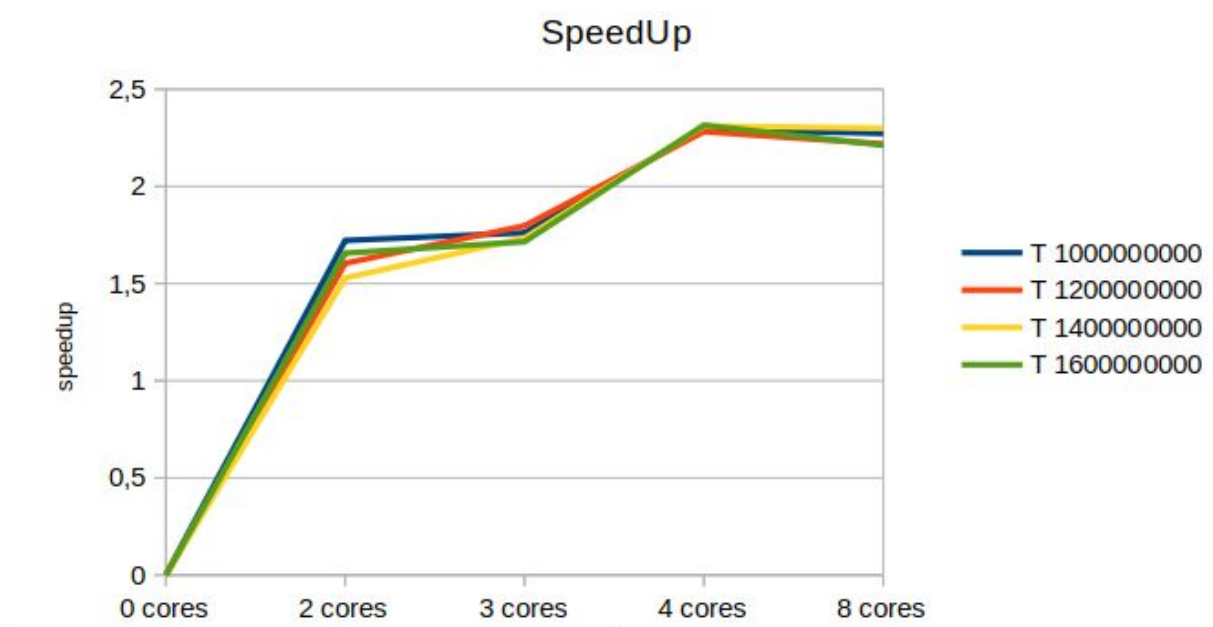
$$S = \frac{T_s}{T_p}.$$

O speedup expressa quantas vezes o algoritmo paralelo é mais rápido do que o sequencial.

A seguir a tabela referente ao SpeedUp do código paralelo comparado com o serial.

Speedup				
Tamanho	1000000000	1200000000	1400000000	1600000000
2 cores	1,72	1,60	1,53	1,66
3 cores	1,76	1,80	1,73	1,71
4 cores	2,30	2,28	2,31	2,32
8 cores	2,27	2,22	2,30	2,21

Em seguida temos o gráfico de SpeedUp onde podemos observar claramente que no final o gráfico fica constante.



Eficiência e Escalabilidade

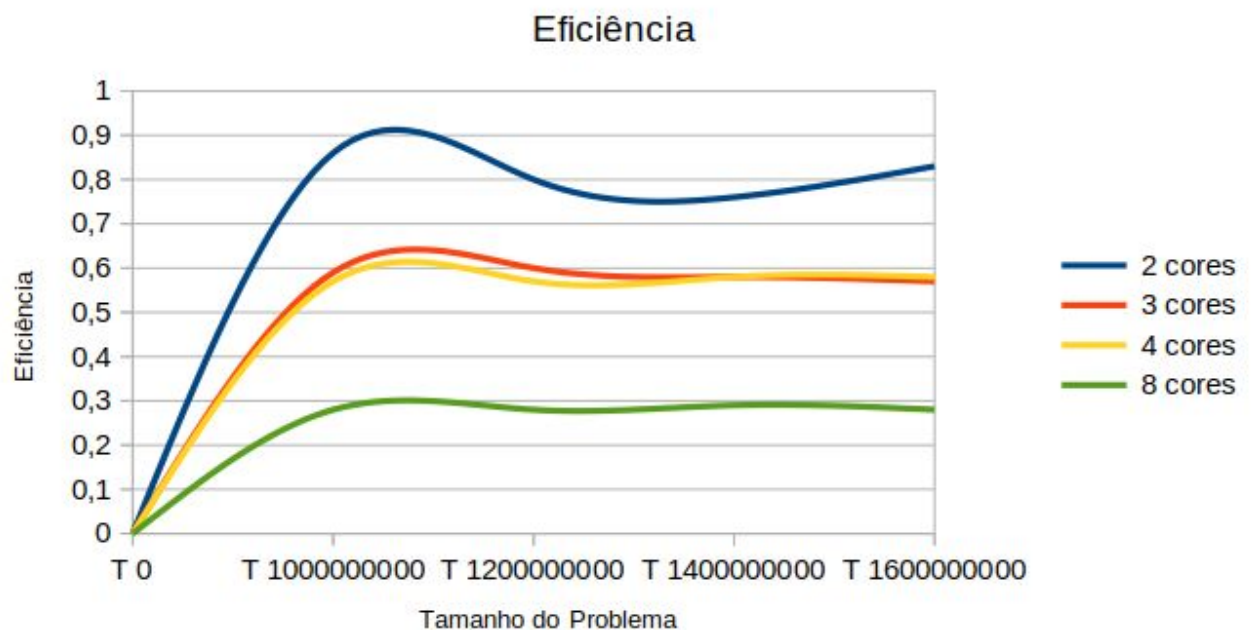
Considere também a eficiência paralela E_f como a razão entre o speedup e o número de núcleos de processamento m , o que indica o quão bem os núcleos de processamento estão sendo utilizados na computação, na forma

$$E_f = \frac{S}{m}.$$

A seguir a tabela referente a eficiência do código paralelo.

Eficiência				
Tamanho	2 cores	3 cores	4 cores	8 cores
1000000000	0,86	0,59	0,57	0,28
1200000000	0,80	0,60	0,57	0,28
1400000000	0,76	0,58	0,58	0,29
1600000000	0,83	0,57	0,58	0,28

Comentário o gráfico a seguir na conclusão.

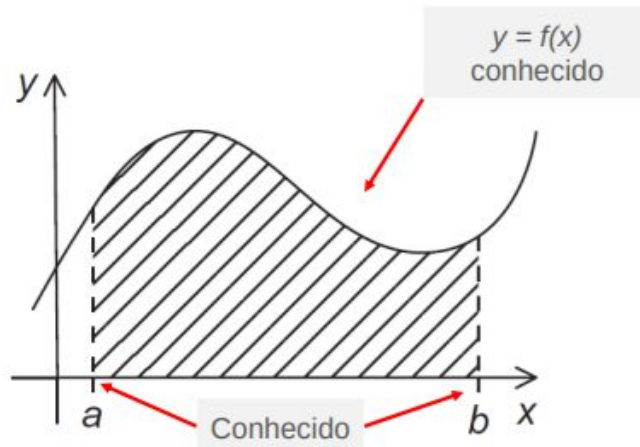


Conclusão

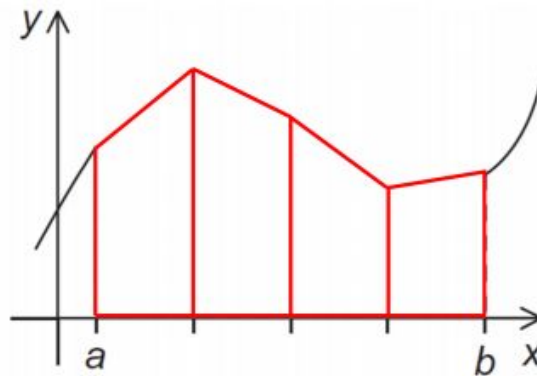
Após os vários testes foi comprovada a eficiência do MPI, apesar de que no gráfico podemos observar uma pequena queda no segundo tamanho do problema utilizando 2 cores, observando as outras linhas podemos ver que continuam constantes conforme o tamanho do problema aumenta, desconsiderando o caso com 2 cores podemos dizer que o código é fracamente escalável.

Regra dos Trapézios

A regra dos trapézios serve para calcular a área entre dois pontos de uma função qualquer, como no exemplo a seguir.



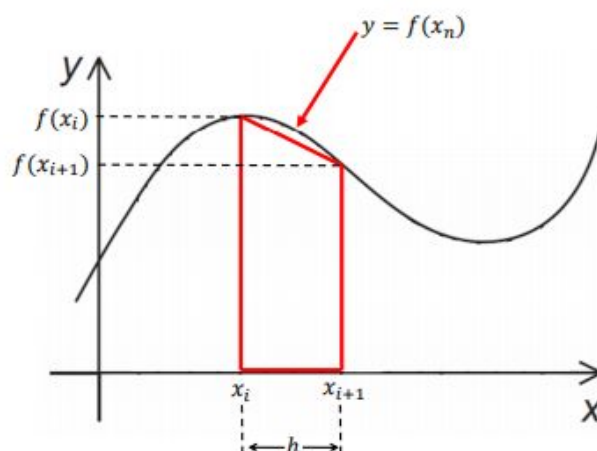
Para fazer o cálculo dividimos a área em vários trapézios como no exemplo a seguir



Calculando cada trapézio com a fórmula a seguir.

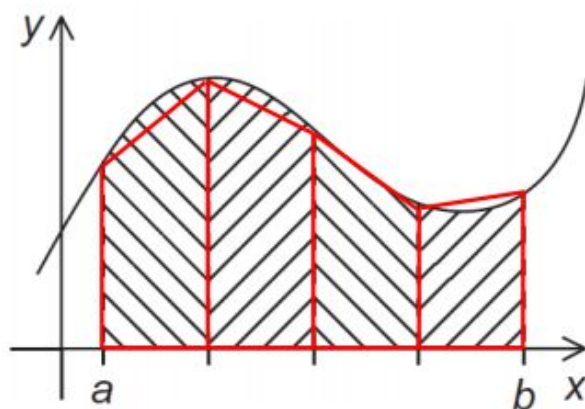
$$\text{Área de um trapézio} = \frac{h}{2} [f(x_i) + f(x_{i+1})]$$

$$h = \frac{b - a}{n} \quad \leftarrow \text{Nº de trapézios}$$



E depois para conseguir a área entre os dois pontos é só somar a área dos trapézios

Veja porém que essa aproximação, no exemplo da figura a seguir, deixou muito a desejar, visto que uma parte expressiva da área sob o gráfico da função foi negligenciada. O que fazer para melhorar essa aproximação?



A estratégia para melhorar a qualidade da aproximação é dividir os trapézios em diversos sub trapézios menores, aproximando o valor da área real conforme o número de trapézios aumenta.

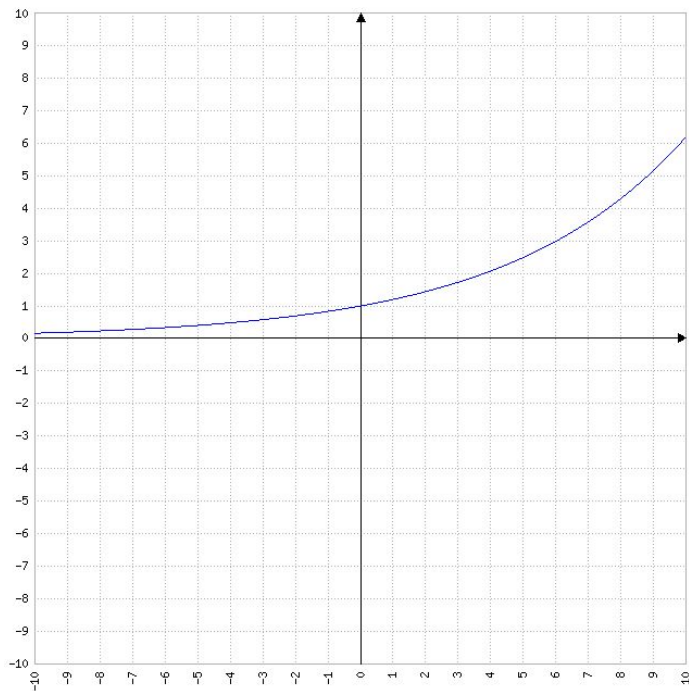
Implementação do Código

Função usada

Para fazer o cálculo da área aplicamos a regra dos trapézios em uma função exponencial bem simples representada no código a seguir

```
#include <math.h>
double f(int n){ return pow(1.2, n);}
```

A seguir temos a representação da função na forma de gráfico.



Serial

A seguir a implementação do código em sua forma serial. Como dito anteriormente o código divide a área da função em n trapézios calcula a área deles e depois soma para obter a área aproximada entre os pontos da função.

Logo em seguida temos o código para fazer a marcação do tempo em um arquivo de texto.

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
double f(int n){ return pow(1.2, n);} //função
int main(int argc, char const *argv[])
{
    struct timeval start, stop;
    gettimeofday(&start, 0);
    int x_i;
    double a=1;
    double b=20;
    long double n=atof(argv[1]);
    long double h=(b-a)/n;
    double area_total=(f(a)+f(b))/2;
    for (int i = 1; i < n; i++) {
        x_i=a+i*h;
        area_total +=f(x_i);
    }
    area_total =h*area_total;
    gettimeofday(&stop, 0);
    FILE *fp;
    char outputFilename[] = "tempo_de_mm.txt";
    fp = fopen(outputFilename, "a");
    if (fp == NULL) {
        fprintf(stderr, "Can't open output file %s!\n", outputFilename);
        exit(1);
    }
    //testes de impressão no arquivo
    fprintf(fp, "\t%f ", (double)(stop.tv_usec - start.tv_usec) / 1000000 +
(double)(stop.tv_sec - start.tv_sec));
    fclose(fp);
    return 0;
}
```

Paralelo

A seguir temos a implementação do código paralelo. Assim como o serial ele divide a área entre os dois pontos da função em n trapézios, calcula e soma a área dos trapézios, mas o número de trapézios é dividido entre os processos para que cada uma calcule uma parte e envia para o processo ZERO que faz a soma de todos para dar a área final mais rapidamente. E logo depois o processo ZERO escreve o tempo que levou para executar o código em forma paralelo em um arquivo de texto.

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#include <math.h>
#include <time.h>

MPI_Status status;

double f(int n){ return pow(1.2, n);}

int main(int argc, char **argv)
{
    struct timeval start, stop;
    gettimeofday(&start, 0);
    int x_i;
    double a=1;
    double b=20;
    long double n=atof(argv[1]);
    long double h=(b-a)/n;
    int numtasks,taskid;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    double local_n=n/numtasks;
    double local_a=a+taskid*local_n*h;
    double local_b=local_a+local_n*h;
    double local_area=(f(local_a)+f(local_b))/2;
    double area_total;
    for (int i = 1; i < local_n; i++)
    {
        x_i=local_a+i*h;
        local_area +=f(x_i);
    }
    local_area =h*local_area;

    if (taskid == 0)
    {
        area_total = local_area;
        for (int proc=1; proc<numtasks; proc++) {
            MPI_Recv(&local_area,1,MPI_DOUBLE,proc,0,MPI_COMM_WORLD,&status);
            area_total +=local_area;
        }
    }
}
```

```

    }

    printf("area total=%f\n",area_total);
    gettimeofday(&stop, 0);
}
else
{
    MPI_Send(&local_area,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
}
if (taskid == 0)
{
    FILE *fp;
    char outputFilename[] = "tempo_de_mm.txt";

    fp = fopen(outputFilename, "a");
    if (fp == NULL) {
        fprintf(stderr, "Can't open output file %s!\n", outputFilename);
        exit(1);
    }
    fprintf(fp, "\t%f ", (double)(stop.tv_usec - start.tv_usec) / 1000000 +
(double)(stop.tv_sec - start.tv_sec));
    fclose(fp);
}

MPI_Finalize();
return 0;
}

```

Execução dos códigos

Máquina usada nos testes

Nos testes foi utilizado um notebook da marca asus de modelo ASUS Z450UA-WX010 com a seguintes configurações:

Processador Intel(R) Core(TM) i3-6100U CPU @ 2.30GHz

Número do processador: i3-6100U

Número de núcleos: 2

Nº de threads: 4

Memória ram: 4.0 GB

Armazenamento: SDD 241.1 GB

Sistema operacional: Linux Mint 20 Cinnamon versão 4.6.7

Código serial

Para os testes do código serial foi utilizado o seguinte código em Shell Script.

```
#!/bin/bash
#OPCIONAL: apagar arquivos temporários (gerados ou não pelo .c).
rm tempo_de_mm.txt
rm Tfunction
#Compilação de Código. Modifique para o que mais se adequa a você.
gcc -g -Wall Tfunction.c -o Tfunction -lm
#OBRIGATÓRIO: Laço de iteração para resgate dos tempos de acordo com
"cores" e "size"
#Loop principal de execuções. São 10 tentativas
tentativas=1 #Quantas vezes o código será executado dado um par
(cores,size)

for cores in 1 #números de cores utilizados
do
    for size in 800000000 1000000000 1100000000 1150000000
#tamanho do problema
do
    echo -e "\n$cores\t$t$size\t\t\t\t\t" >> "tempo_de_mm.txt"

    for tentativa in $(seq $tentativas) #Cria uma vetor de 1
a "tentativas"
do
    #Executar o código. Modifique para o que mais se
adequa a você.

    ./Tfunction $size
    #mpirun ./mpi_mm
    #no meu PC, tive que utilizar "--oversubscribe" para
rodar. No seu pode não ser necessário.
done
done
done
exit
exit
```

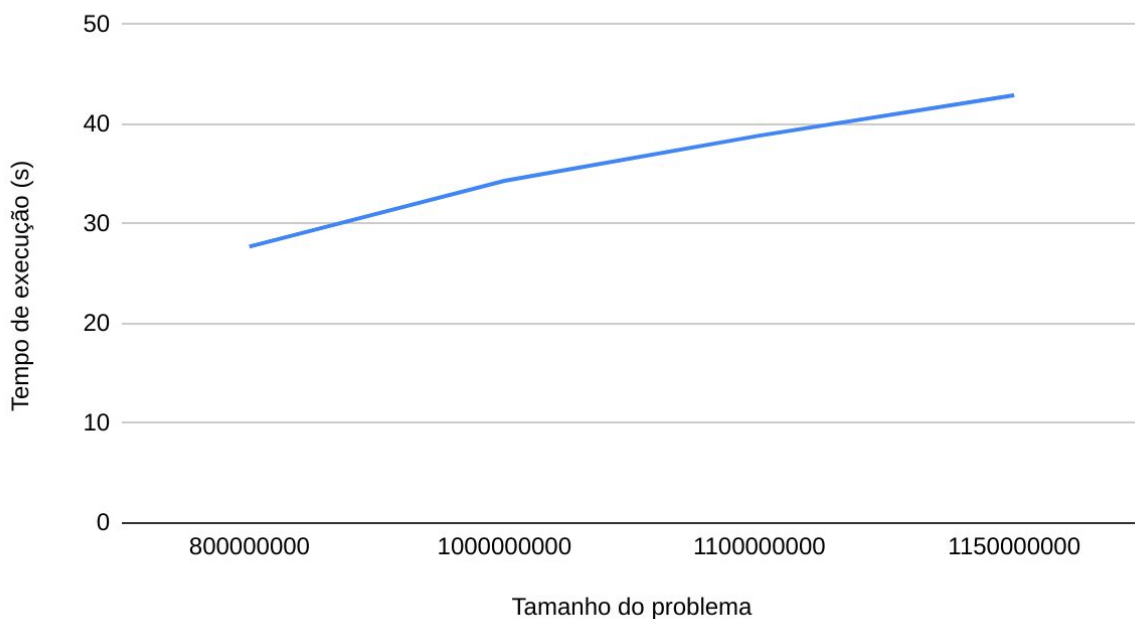
O Shell Script executa o código serial 5 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código serial que escreve o tempo que levou para executar logo depois.

Testes

Ao executar o códigos 5 vezes em cada tamanho do problema fiz uma média como mostra a tabela a seguir

Código Serial	
Tamanho do problema	Tempo de execução (s)
800000000	27,7245138
1000000000	34,3380216
1100000000	38,8590576
1150000000	42,9191612

Tempos de execução x Tamanho do problema



Código Paralelo

Para os testes do código paralelo foi utilizado o seguinte código em Shell Script.

```
#!/bin/bash
#OPCIONAL: apagar arquivos temporários (gerados ou não pelo .c).
rm tempo_de_mm.txt
rm tfunction

#Compilação de Código. Modifique para o que mais se adequa a você.
mpicc -g -Wall Tfunction.c -o tfunction -lm

#OBRIGATÓRIO: Laço de iteração para resgate dos tempos de acordo com
"cores" e "size"

#Loop principal de execuções. São 10 tentativas
tentativas=5 #Quantas vezes o código será executado dado um par
(cores,size)

for cores in 1 2 3 4 8 #números de cores utilizados
do
    for size in 800000000 1000000000 1100000000 1150000000
#tamanho do problema
do
        echo -e "\n$cores\t$size\t\t\tc" >> "tempo_de_mm.txt"

        for tentativa in $(seq $tentativas) #Cria uma vetor de 1
a "tentativas"
do
            #Executar o código. Modifique para o que mais se
adequa a você.

            mpirun -np $cores ./tfunction $size
            #mpirun ./mpi_mm
            #no meu PC, tive que utilizar "--oversubscribe" para
rodar. No seu pode não ser necessário.

        done
    done
done
exit
exit
```

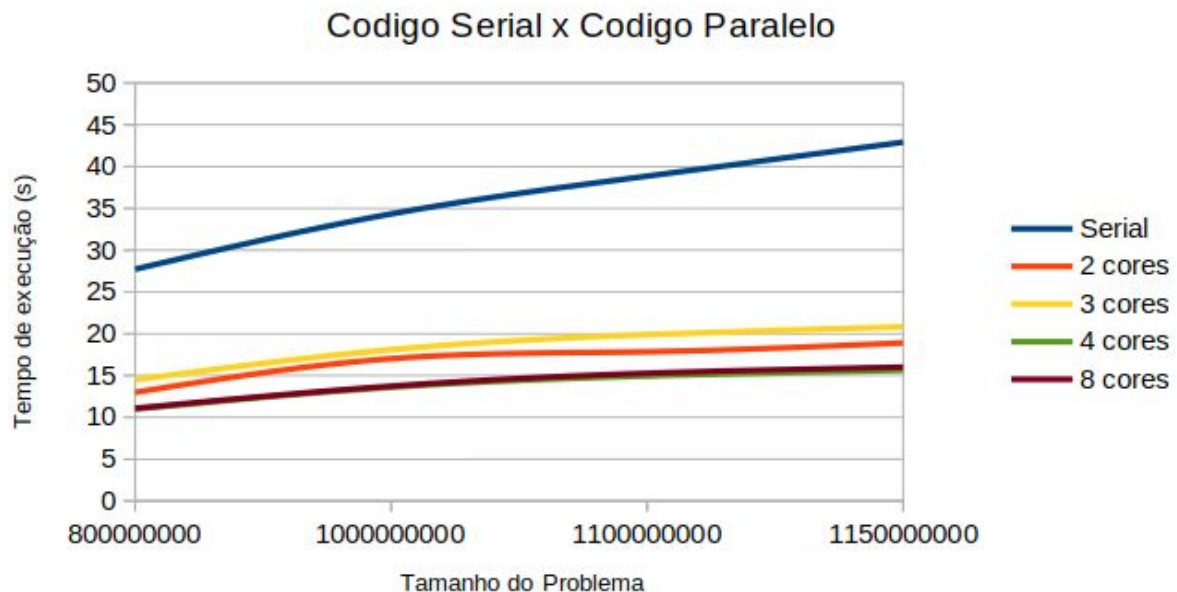
O Shell Script executa o código paralelo 5 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código paralelo que escreve o tempo que levou para executar logo depois.

Testes

Ao executar o códigos 5 vezes em cada tamanho do problema fiz uma média como mostra a tabela a seguir

Código Serial		
Nº de cores	Tamanho do problema	Tempo de execução (s)
Serial	800000000	27,7245138
Serial	1000000000	34,3380216
Serial	1100000000	38,8590576
Serial	1150000000	42,9191612
2	800000000	12,9868884
2	1000000000	17,0172258
2	1100000000	17,8483564
2	1150000000	18,9056596
3	800000000	14,494943
3	1000000000	18,1015072
3	1100000000	19,887353
3	1150000000	20,818699
4	800000000	10,9759208
4	1000000000	13,5915274
4	1100000000	14,9412672
4	1150000000	15,6237844
8	800000000	11,0854872
8	1000000000	13,7334148
8	1100000000	15,2752956
8	1150000000	16,0011776

No gráfico a seguir está a comparação do código serial com o paralelo. Aqui temos o mesmo problema do Pi utilizando Monte Carlo, pelo notebook ter apenas 2 cores fisicos e 4 threads o desempenho só é significativo quando quando utilizamos até 4 processos, e utilizando de 3 a 4 cores o desempenho nem é tão significativo assim.

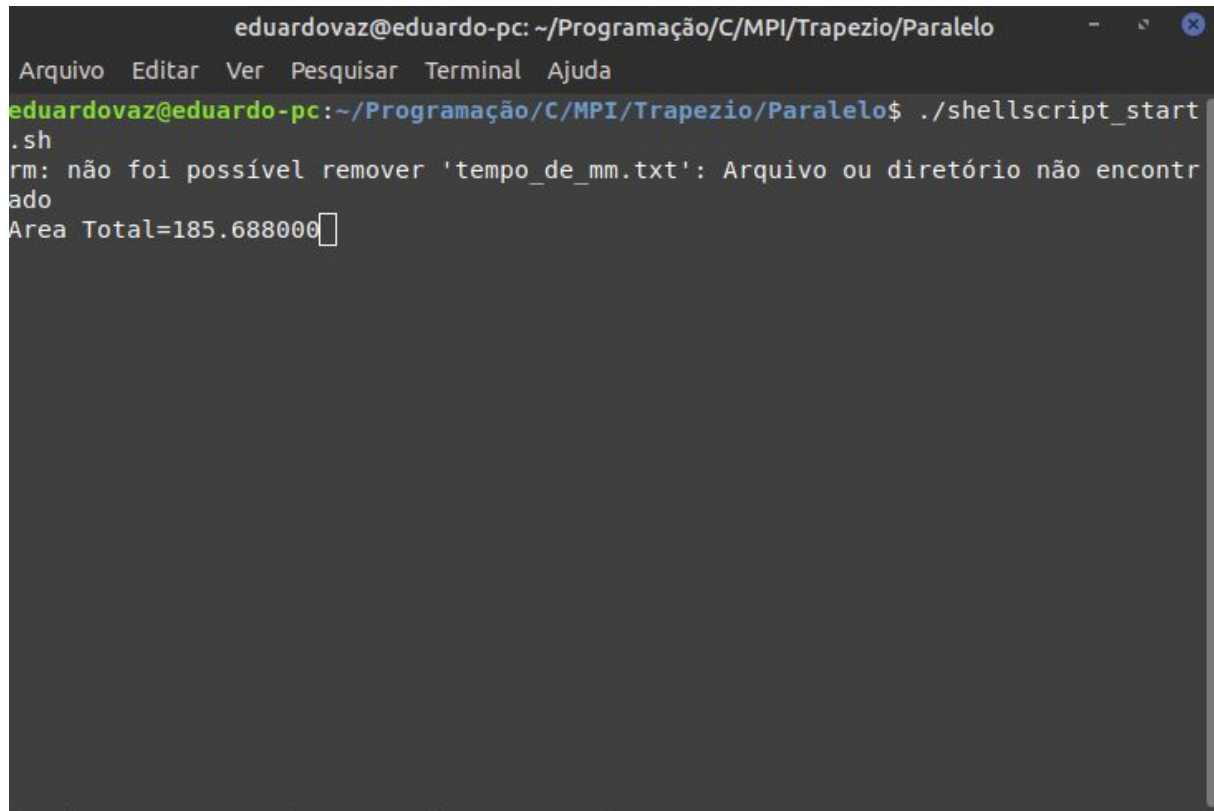


A seguir alguns exemplos dos códigos rodando no terminal do linux mint.

Execução do código serial, vale lembrar que na execução para os testes não teve nenhum tipo de escrita na tela para não atrapalhar no desempenho

```
eduardovaz@eduardo-pc: ~/Programação/C/MPI/Trapezio/Serial
Arquivo Editar Ver Pesquisar Terminal Ajuda
eduardovaz@eduardo-pc:~/Programação/C/MPI/Trapezio/Serial$ ./shellscript_start.s
h
area total=185.688000area total=185.688000
```

Execução do código paralelo, vale lembrar que na execução para os testes não teve nenhum tipo de escrita na tela para não atrapalhar no desempenho.



```
eduardovaz@eduardo-pc: ~/Programação/C/MPI/Trapezio/Paralelo
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
eduardovaz@eduardo-pc:~/Programação/C/MPI/Trapezio/Paralelo$ ./shellscript_start
.sh
rm: não foi possível remover 'tempo_de_mm.txt': Arquivo ou diretório não encontrado
Area Total=185.688000
```

Análise de Speedup, Eficiência e Escalabilidade

SpeedUp

O speedup paralelo S definido como a relação entre o tempo de processamento serial T_s de um algoritmo e seu tempo de processamento paralelo T_p , tal que

$$S = \frac{T_s}{T_p}.$$

O speedup expressa quantas vezes o algoritmo paralelo é mais rápido do que o sequencial.

A seguir a tabela referente ao SpeedUp do código paralelo comparado com o serial.

Speedup				
Tamanho	800000000	1000000000	1100000000	1150000000
2 cores	2,13	2,02	2,18	2,27
3 cores	1,91	1,90	1,95	2,06
4 cores	2,53	2,53	2,60	2,75
8 cores	2,50	2,50	2,54	2,68

Em seguida temos o gráfico de SpeedUp onde podemos observar claramente que no final o gráfico fica constante.



Eficiência e Escalabilidade

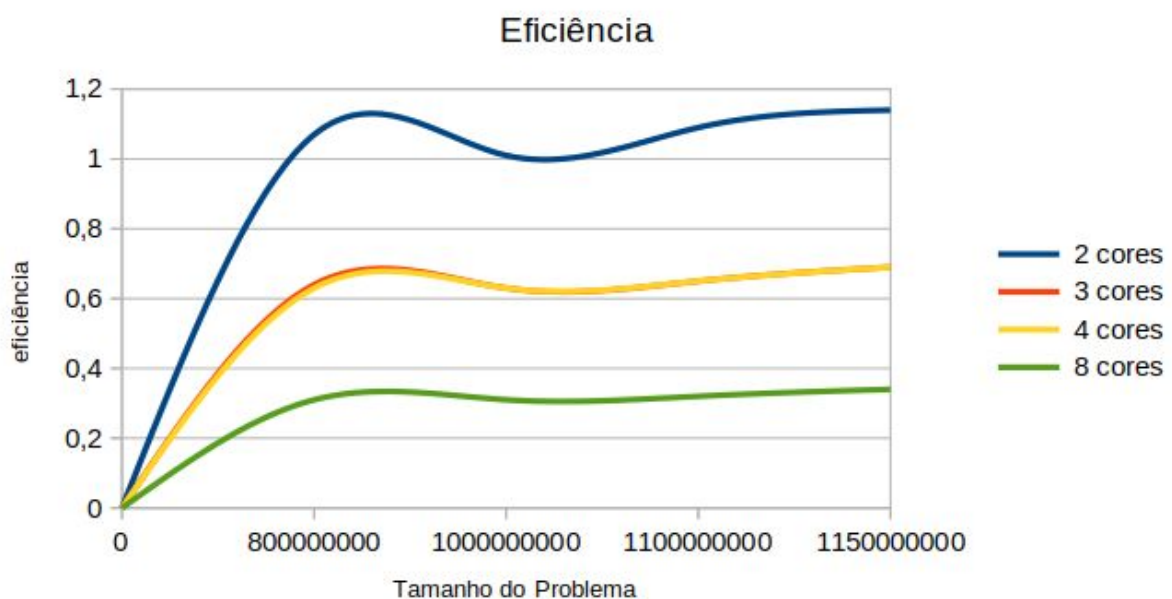
Considere também a eficiência paralela E_f como a razão entre o speedup e o número de núcleos de processamento m , o que indica o quão bem os núcleos de processamento estão sendo utilizados na computação, na forma

$$E_f = \frac{S}{m}.$$

A seguir a tabela referente a eficiência do código paralelo.

Eficiência				
Tamanho	2 cores	3 cores	4 cores	8 cores
800000000	1,07	0,64	0,63	0,31
1000000000	1,01	0,63	0,63	0,31
1100000000	1,09	0,65	0,65	0,32
1150000000	1,14	0,69	0,69	0,34

O gráfico a seguir será comentado na conclusão.



Conclusão

Após os vários testes foi comprovada a eficiência do MPI, como no problema de PI, apesar de que no gráfico podemos observar uma pequena queda no segundo tamanho do problema utilizando 2 cores, observando as outras linhas podemos ver que continuam constantes conforme o tamanho do problema aumenta, e novamente temos a mesma conclusão pois os gráficos de eficiência dos dois problemas são extremamente parecidos.