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**

Odd-even Transposition Sort **3**

Implementação do Código 4

Serial 4

Paralelo 6

Execução dos códigos **9**

Máquina usada nos testes 9

Código serial 9

Testes 10

Código Paralelo 11

Testes 12

Análise de Speedup, Eficiência e Escalabilidade **14**

SpeedUp 14

Eficiência e Escalabilidade 15

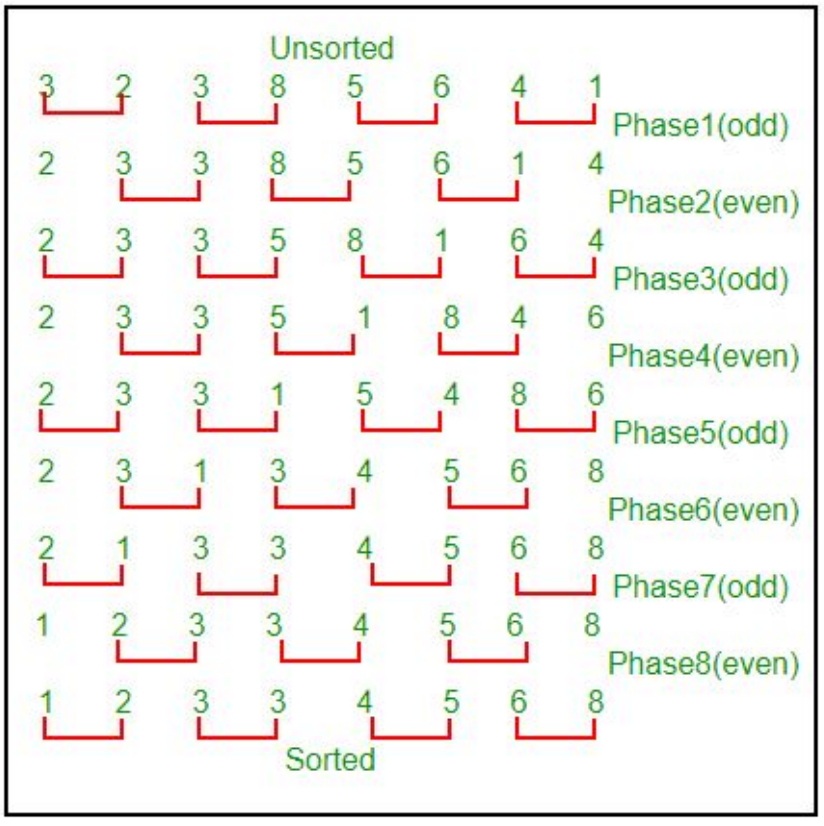
Conclusão **16**

Introdução

Neste relatório aplicamos o método MPI (Message Passing Interface), na solução Odd-even Transposition Sort

Odd-even Transposition Sort

O Odd-even sort é um algoritmo de ordenação relativamente simples. É um algoritmo de ordenação por comparação baseado no bubble sort com o qual compartilha muitas características. Ele funciona através da comparação de todos os pares indexados (ímpar, par) de elementos adjacentes na lista e, se um par está na ordem errada (o primeiro é maior do que o segundo), os elementos são trocados. O próximo passo repete isso para os pares indexados (par, ímpar) (de elementos adjacentes). Em seguida, ele alterna entre etapas de (ímpar, par) e (par, ímpar) até que a lista é ordenada. Pode ser pensado como a utilização de processadores paralelos, cada qual usando um BubbleSort, mas a partir de diferentes pontos na lista (todos os índices ímpares para a primeira etapa). Este algoritmo de ordenação é apenas ligeiramente mais difícil do que o bubble sort para implementar.



Implementação do Código

Serial

A seguir implementamos a função do método de Odd-even Transposition Sort serial na linguagem c. O código tem duas fases a par e ímpar, na primeira fase par as posições do vetores com números pares ele compara com o próximo e se for menor ele trocar as posições, e o mesmo nas posições de vetores com números ímpares.

```c
void Odd_Even(int *vet, int n)
{
   int phase,i,temp;
   for (phase = 0; phase < n; phase++)
   {
       if (phase % 2 == 0)
       {
           for (i = 1; i < n; i+=2)
           {
               if (vet[i-1]>vet[i])
               {
                   temp=vet[i];
                   vet[i]=vet[i-1];
                   vet[i-1]=temp;
               }
           }

       }
       else
       {
           for (i = 1; i < n-1; i+=2)
           {
               if (vet[i]>vet[i+1])
               {
                   temp=vet[i];
                   vet[i]=vet[i+1];
                   vet[i+1]=temp;
               }
           }
       }
   }

}
```

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 const *argv[])
{
   //========================
   gettimeofday(&start, 0);
   Odd_Even(v,n);
   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 Odd-even Transposition Sort paralelo na linguagem c.
Primeiro e vetor é criado e embaralhado, depois o vetor é dividido igualmente entre os processos, a seguir começa a Odd-even Transposition Sort é dividido entre as fases par e ímpar, na primeira fase par, o primeiro processo recebe o vetor do segundo processo, envia o seu proprio vetor para o segundo processo,depois junta ao seu próprio vetor e organizar usando a função do Odd-even Transposition Sort, na fase impar o primeiro processo recebe o vetor do segundo processo, organizar usando a mesma função, e depois organizar seu vetor para que a parte que cabe a ele fique no começo, e depois de todas as fases o começo dos vetores é organizado em um so para que tenha o vetor completo organizado.

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

MPI_Status status;

void embaralhar(int *vet, int vetSize) //função para embaralhar o vetor
{
   for (int i = 0; i < vetSize; i++)
   {
       int r = rand() % vetSize;

       int temp = vet[i];
       vet[i] = vet[r];
       vet[r] = temp;
   }
}

void Odd_Even(int *vet, int n) //função para organizar o vetor
{
   int phase, i, temp;
   for (phase = 0; phase < n; phase++)
   {
       if (phase % 2 == 0)
       {
           for (i = 1; i < n; i += 2)
           {
               if (vet[i - 1] > vet[i])
               {
                   temp = vet[i];
                   vet[i] = vet[i - 1];
                   vet[i - 1] = temp;
               }
           }
       }
       else
       {
           for (i = 1; i < n - 1; i += 2)
           {
               if (vet[i] > vet[i + 1])
               {
                   temp = vet[i];
                   vet[i] = vet[i + 1];
```

```
                    vet[i + 1] = temp;
                }
            }
        }
    }
}

int main(int argc, char **argv)
{
    //criando vetor aleatorio;
    int size = atof(argv[1]);
    int *v = malloc(sizeof(int) * size);
    for (int i = 0; i < size; i++) //criando vetor
    {
        v[i] = i;
    }
    struct timeval start, stop;
    embaralhar(v, size); //embaralhando vetor

    int numtasks, taskid;

    gettimeofday(&start, 0); //inicio da contagem de tempo
    //=====================================Inicio=====================================
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

    int local_size = size / numtasks;
    int *local_v = malloc(sizeof(int) * (local_size * 2));

    if (taskid == 0) //==========Destribuindo o vetor entre os processos============
    {
        MPI_Scatter(v, local_size, MPI_INT, local_v, local_size, MPI_INT, 0, MPI_COMM_WORLD);
    }
    else
    {
        MPI_Scatter(v, local_size, MPI_INT, local_v, local_size, MPI_INT, 0, MPI_COMM_WORLD);
    } //=========================================================

    for (int phase = 0; phase < numtasks - 1; phase++) //começando a odd even so que dessa vez usando
os processos
    {
        MPI_Barrier(MPI_COMM_WORLD);
        if (phase % 2 == 0) //fase par
        {
            if (taskid % 2 == 0)
            {
                MPI_Recv(&local_v[local_size], local_size, MPI_INT, (taskid + 1), 0, MPI_COMM_WORLD,
&status); //recebendo novo vetor
                MPI_Send(local_v, local_size, MPI_INT, (taskid + 1), 0, MPI_COMM_WORLD);
//enviando vetor
                Odd_Even(local_v, local_size * 2);
//organizando os dois vetores
            }
            else
            {
                MPI_Send(local_v, local_size, MPI_INT, (taskid - 1), 0, MPI_COMM_WORLD);
//segundo processo enviando vetor
                MPI_Recv(&local_v[local_size], local_size, MPI_INT, (taskid - 1), 0, MPI_COMM_WORLD,
&status); //segundo processo recebendo vetor
                Odd_Even(local_v, local_size * 2);
```

```
                MPI_Sendrecv(&local_v[local_size],local_size, MPI_INT, taskid, 0,local_v,local_size,
MPI_INT, taskid, 0, MPI_COMM_WORLD, &status);  //enviando e recebendo o vetor dele mesmo, para
colocar a parte que cabe a ele na primeir posição

            }
        }
        if (phase % 2 == 1) //fase impar
        {
            if (taskid % 2 == 1 && taskid != numtasks - 1)
            {
                MPI_Recv(&local_v[local_size], local_size, MPI_INT, (taskid + 1), 0, MPI_COMM_WORLD,
&status); //recebendo novo vetor
                MPI_Send(local_v, local_size, MPI_INT, (taskid + 1), 0, MPI_COMM_WORLD);
//enviando vetor
                Odd_Even(local_v, local_size * 2);
//organizando os dois vetores
            }
            else if (taskid % 2 == 0 && taskid != 0)
            {
                MPI_Send(local_v, local_size, MPI_INT, (taskid - 1), 0, MPI_COMM_WORLD);
//segundo processo enviando vetor
                MPI_Recv(&local_v[local_size], local_size, MPI_INT, (taskid - 1), 0, MPI_COMM_WORLD,
&status); //segundo processo recebendo vetor
                Odd_Even(local_v, local_size * 2);
                MPI_Sendrecv(&local_v[local_size],local_size, MPI_INT, taskid, 0,local_v,local_size,
MPI_INT, taskid, 0, MPI_COMM_WORLD, &status);   //enviando e recebendo o vetor dele mesmo, para
colocar a parte que cabe a ele na primeir posição
            }
        }
    }
    MPI_Gather(local_v, local_size, MPI_INT, v, local_size, MPI_INT, 0, MPI_COMM_WORLD); //juntando
todos os vetores organizados no vetor principal e completo

    gettimeofday(&stop, 0); //fim da contagem de tempo
    free(local_v);          //desalocando vetor
    MPI_Finalize();
//================================================Fim=============================================
===
    gettimeofday(&stop, 0); //fim da contagem de tempo

    if (taskid == 0) //primeiro processo 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);
        }

        //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;
}
```

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
#INFORMAÇÕES- Para rodar no computador pessoal.
# EXECUTAR NO TERMINAL: ./shellscript_start.sh
   #Caso não funcione, conceda permissão máxima ao este arquivo: chmod 777
shellscript_start.sh
#OPCIONAL: apagar arquivos temporários (gerados ou não pelo .c).
rm tempo_de_mm.txt
rm mpi_mm
#Compilação de Código. Modifique para o que mais se adequa a você.
gcc -g -Wall oddeven.c -o mpi_mm
#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  #números de cores utilizados
   do
           for size in  100000 106000 110000 116000 #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.
                   ./mpi_mm $size
                   #mpirun ./mpi_mm
                   #no meu PC, tive que utilizar "--oversubscrive" para rodar. No seu pode
não ser necessário.
               done
           done
   done
   exit
exit
```

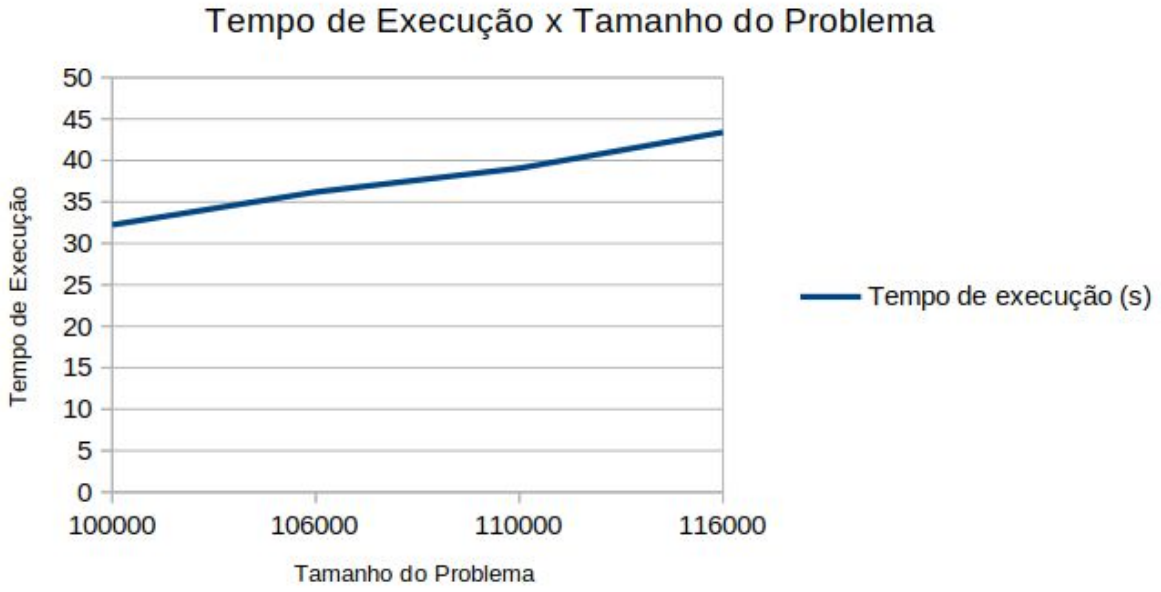
O Shell Script executa o código serial 5 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código serial que escreve o tempo que levou para executar logo depois.

Testes

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

Tamanho do Problema	Tempo de execução (s)
100000	32,23
106000	36,19
110000	39,06
116000	43,40

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



Código Paralelo

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

```
#!/bin/bash

#INFORMAÇÕES- Para rodar no computador pessoal.
# EXECUTAR NO TERMINAL: ./shellscript_start.sh
   #Caso não funcione, conceda permissão máxima ao este arquivo: chmod
777 shellscript_start.sh

#OPCIONAL: apagar arquivos temporários (gerados ou não pelo .c).
rm tempo_de_mm.txt
rm mpi_mmcopy
#Compilação de Código. Modifique para o que mais se adequa a você.
mpicc -g -Wall oddeven.c -o mpi_mmcopy
#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 2 4 8 #números de cores utilizados
   do
           for size in 100000 106000 110000 116000 #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_mmcopy $size
                   #mpirun ./mpi_mm
                   #no meu PC, tive que utilizar "--oversubscrive" 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.

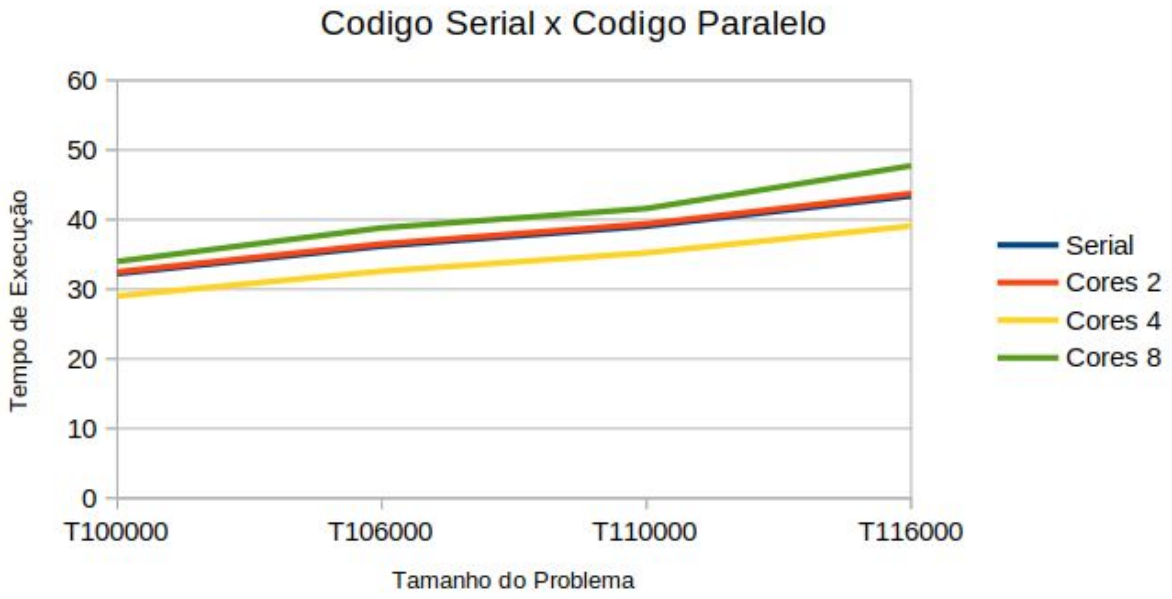
Testes

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

Nº de Cores	Tamanho do Problema	Tempo de execução (s)
Serial	100000	32,23
Serial	106000	36,19
Serial	110000	39,06
Serial	116000	43,40
2	100000	32,49
2	106000	36,51
2	110000	39,38
2	116000	43,74
4	100000	29,03
4	106000	32,59
4	110000	35,23
4	116000	39,08
8	100000	33,99
8	106000	38,81
8	110000	41,60
8	116000	47,74

No gráfico a seguir está a comparação do código serial com o paralelo, da pra observar que utilizando 2 cores o desempenho é ate inferior ao codigo serial,e depois com 4 cores ele so tem um pequeno aumento no desempenho,e com 8 cores o desempenho é menor ainda,

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 o consumo dos processadores do código paralelo usando 4 cores no linux mint.

Análise de Speedup, Eficiência e Escalabilidade

SpeedUp

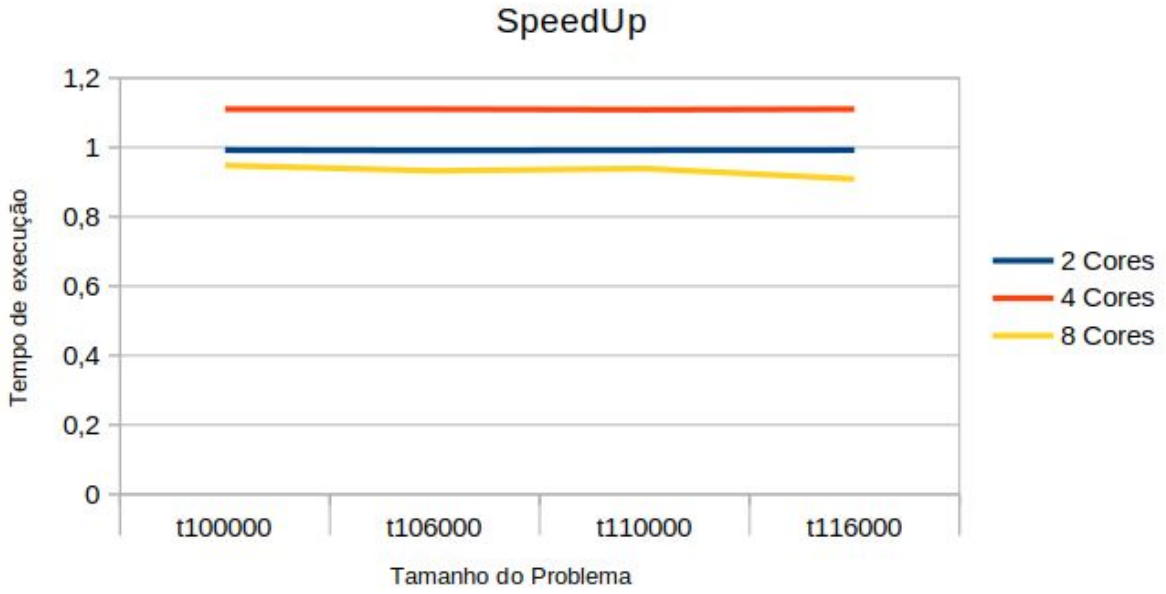
O speedup paralelo S definido como a relação entre o tempo de processamento serial Ts de um algoritmo e seu tempo de processamento paralelo Tp, tal que

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

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 do Problema	100000	106000	110000	116000
2 Cores	0,99	0,99	0,99	0,99
4 Cores	1,11	1,11	1,11	1,11
8 Cores	0,95	0,93	0,94	0,91

Em seguida temos o gráfico de SpeedUp onde podemos observar que o gráfico é constante em todos os cores.



Eficiência e Escalabilidade

Considere também a eficiência paralela Ef 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

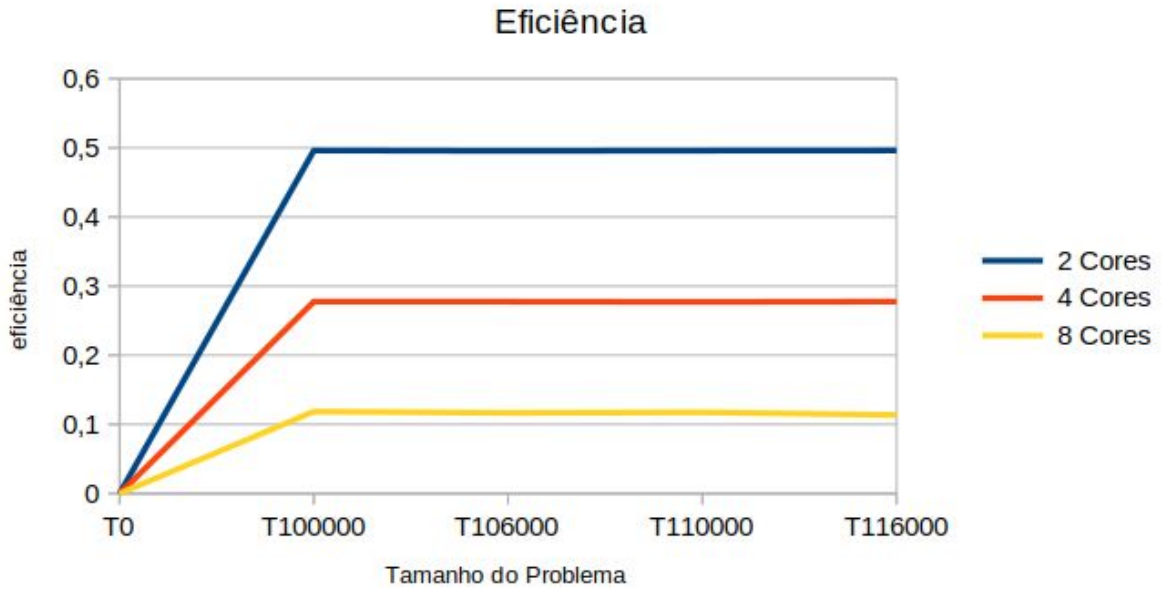
$$E_f = \frac{S}{m}.$$

A seguir a tabela referente a eficiência do código paralelo.

Eficiência			
Tamanho	2 Cores	4 Cores	8 Cores
100000	0,4960541286	0,2775210237	0,1185223835
106000	0,4955450771	0,2775628769	0,1165602627
110000	0,4958884324	0,2771477786	0,1173732522
116000	0,4960646253	0,2775948378	0,1136226383

Comentário o gráfico a seguir na conclusão.

Eficiência



Conclusão

Após vários testes e visualizando as características da máquina dá pra perceber que o código não foi muito eficiente nesse caso, provavelmente se rodar o código em uma máquina mais potente poderemos visualizar sua eficiência, pois só tivemos uma diminuição significativa no tempo de execução usando 4 cores, e no caso dessa máquina nem temos 4 cores físicos. mas mesmo assim ainda podemos dizer que o código é fracamente escalável.