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 OpenMP

SUMÁRIO

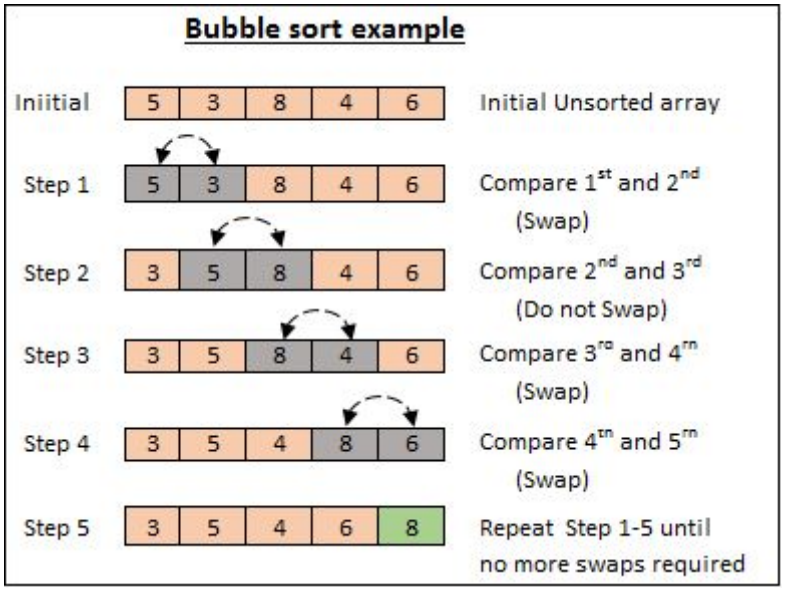
Introdução

Neste relatório aplicamos o método OpenMP, Para fazer a ordenação de um vetor utilizando o algoritmo de ordenação bubble sort.

Bubble Sort

O bubble sort, ou ordenação por flutuação (literalmente "por bolha"), é um algoritmo de ordenação dos mais simples. A ideia é percorrer o vector diversas vezes, e a cada passagem fazer flutuar para o topo o maior elemento da sequência. Essa movimentação lembra a forma como as bolhas em um tanque de água procuram seu próprio nível, e disso vem o nome do algoritmo.

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos do vetor. No pior caso, são feitas n^2 operações. A complexidade desse algoritmo é de ordem quadrática. Por isso, ele não é recomendado para programas que precisem de velocidade e operem com quantidade elevada de dados.



Implementação do Código

Serial

A seguir temos o código serial em c, que cria um vetor ordenado, e desordena usando uma função, logo depois ele executa a função bubble sort, que rodar o vetor n vezes ordenando ele como explicado no código.

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

```c
#include <time.h>

int *Alocavetor(int m) //função para alocar a vetor dinamicamente
{
   int *M;
   M = (int *)malloc(sizeof(int *) * m);
   if (M == NULL)
   {
       printf("Memoria insuficiente.\n");
       exit(1);
   }
   return M;
}

void embaralhar(int *vet, int vetSize) //função para embaralhar
elementos do vetor
{
   srand(13);
   for (int i = 0; i < vetSize; i++)
   {
       int r = rand() % vetSize;

       int temp = vet[i];
       vet[i] = vet[r];
       vet[r] = temp;
   }
}
void imprimirvetor(int *m, int n, char vetornome) //função para
imprimir determinado vetor
{
   printf("vetor %c[%i]\n", vetornome, n);
   for (int linha = 0; linha < n; linha++)
   {
       printf("%i\t", m[linha]);
   }
   printf("\n");
}

void swap(int *a, int *b)//função para trocar um elemento do vetor com
o proximo
{
   int temp = *a;
   *a = *b;
```

```
    *b = temp;
}

void bubbleSort(int *v, int n)//função que realizar as comparações e
trocas
{
    if (n < 1)//caso o n seja 0 a função termina, pois ja chegou na
execução necessaria
        return;

    for (int i = 0; i < n; i++)//rodar o vetor comparando um elemento
com o outro, e caso esteja fora de posição ele troca
        if (v[i] > v[i + 1])
            swap(&v[i], &v[i + 1]);
    bubbleSort(v, n - 1);//chama a função recursivamente
}
int main(int argc, char const *argv[])
{
    struct timeval start, stop; //variaveis para guardar o tempo
    int n = atof(argv[1]);//recebe o tamanho do problema

    int *a = Alocavetor(n);                 //alocando vetor a
    for (int linha = 0; linha <= n; linha++) //definindo valores
aleatorios para cada posição das vetores a e b
    {
        a[linha] = linha + 1;
    }
    //imprimirvetor(a, n, 'a');//imprime vetor
    embaralhar(a, n);//desordena o vetor
    //imprimirvetor(a, n, 'a');//imprime vetor
```

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.

```
    gettimeofday(&start, 0); //começar a marcar o tempo
    bubbleSort(a, n - 1);//chamando a função passando o vetor e o
tamanho
    //imprimirvetor(a, n, 'a');//imprime vetor
    gettimeofday(&stop, 0); //finaliza a contagem do tempo
```

```
    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)); //imprimindo tempo no
arquivo tempo_de_mm.txt
    fclose(fp);
    free(a);
    return 0;
}
```

Paralelo

A seguir temos o código paralelo em c, que cria um vetor ordenado, e desordena usando uma função, logo depois ele abre a região paralela usando OpenMP e o processo 0 executa a função bubble sort, que vai sendo executa e quando chega na quarta posição do vetor ele abre uma nova task que ou ele possa executar quando acabar ou outro processo que está esperando, assim ele executa o bubble sort n vezes e ou final o vetor é verificado, para saber se a ordenação ocorreu.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <omp.h>
```

```c
int global=0;

int *AlocaMatriz(int m) //função para alocar a matriz dinamicamente
{
   int *M;
   M = (int *)malloc(sizeof(int *) * m);
   if (M == NULL)
   {
       printf("Memoria insuficiente.\n");
       exit(1);
   }
   return M;
}

void embaralhar(int *vet, int vetSize)//função para embalhar elementos
do vetor
{
   srand(11);
   for (int i = 0; i < vetSize; i++)
   {
       int r = rand() % vetSize;

       int temp = vet[i];
       vet[i] = vet[r];
       vet[r] = temp;
   }
}
void imprimirmatriz(int *m, int n, char matriznome)//função para
imprimir determinado vetor
{
   printf("Matriz %c[%i]\n", matriznome, n);
   for (int linha = 0; linha < n; linha++)
   {
       printf("%i\t", m[linha]);
   }
   printf("\n");
}

void swap(int *a, int *b)//função para trocar um elemento do vetor com
o proximo
{
   int temp = *a;
   *a = *b;
```

```
    *b = temp;
}

void bubbleSort(int *v, int n,int contador)//função que realizar as
comparações e trocas e gerar as task
{
    if (contador < 1)//caso o contador seja 0 a função termina, pois ja
chegou na execução necessaria
        return;

    for (int i = 0; i < n; i++) //roda o vetor comparando um elemento
com o outro, e caso esteja fora de posição ele troca
    {
        if (i==4 && n>0)//cada vez que chega na quarta posição do vetor
ele abre uma task para que o proximo processo comece a rodar o vetor
        {
            #pragma omp task//abrindo uma task com o contador--
            {
                bubbleSort(v, n,contador-1);
            }
        }

        if (v[i] > v[i + 1])
        {
            swap(&v[i], &v[i + 1]);
        }

    }
    global++;//Contando quantas vezes o bubble sort foi executado
    return;
}
void corretude(int *v, int n)//função para verificar se o vetor foi
ordenado
{
    for (int i = 0; i < n; i++)
    {
        if (v[i] > v[i + 1])
        {
            printf("o vetor não esta ordenado.\n");
            return;
        }
    }
    printf("o vetor esta ordenado.\n");
```

```
    return;

}

int main(int argc, char const *argv[])
{
    struct timeval start, stop; //variaveis para guardar o tempo
    int numThreads = atoi(argv[1]);//recebe o numero de threads
    int n = atoi(argv[2]);//recebe o tamanho do problema

    int *a = AlocaMatriz(n);                    //alocando matriz a
    for (int linha = 0; linha <= n; linha++) //definindo valores
aleatorios para cada posição das matrizes a e b
    {
        a[linha] = linha + 1;
    }
    //imprimirmatriz(a, n, 'a');//imprime vetor
    embaralhar(a, n);//desordena o vetor
    //imprimirmatriz(a, n, 'a');//imprime vetor
    gettimeofday(&start, 0); //começar a marcar o tempo
    #pragma omp parallel num_threads(numThreads) shared(a) shared(n)
default(none)//abrindo região paralela
    {
        int my_rank=omp_get_thread_num();
        //int num_threads=omp_get_num_threads();
        if (my_rank==0)//o processo 0 executa a função e os outros
processos ficam esperando as task serem geradas
        {
            bubbleSort(a, n,n - 1);
        }
    }


    gettimeofday(&stop, 0); //finaliza a contagem do tempo
    //imprimirmatriz(a, n, 'a');//imprime vetor
    corretude(a, n-1);//verifica se o vetor foi ordenado
    printf("o bubbleSort foi executado %i vezes\n",global);//imprime
quantas vezes o bubble sort foi executado

    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)); //imprimindo tempo no
arquivo tempo_de_mm.txt
    fclose(fp);
    free(a);
    return 0;
}
```

Execução dos códigos

Código serial

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

```
#!/bin/bash

#SBATCH --partition=cluster
#SBATCH --job-name=decvaz
#SBATCH --output=decvazOutput.out
#SBATCH --error=decvazError.err
#SBATCH --time=0-02:00
#SBATCH --hint=compute_bound
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32

#No Supercomputador, 1 nó = 32 Cores (ou CPUs)
#Lembrar que: TASK = PROCESSO

#A configuração acima reserva 1 nó inteiro (32 cpus no mesmo
processador), vai rodar apenas 1 processo por vez,
# mas este processo terá acesso a 32 cores
```

```
#Loop principal de execuções. São 10 tentativas

   tentativas=10 #Quantas vezes o código será executado

   for cores in 1 #números de cores utilizados
   do
               for size in 80000 100000 130000 160000 #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
                   ./teste $size
               done
           done

   done

   exit
```

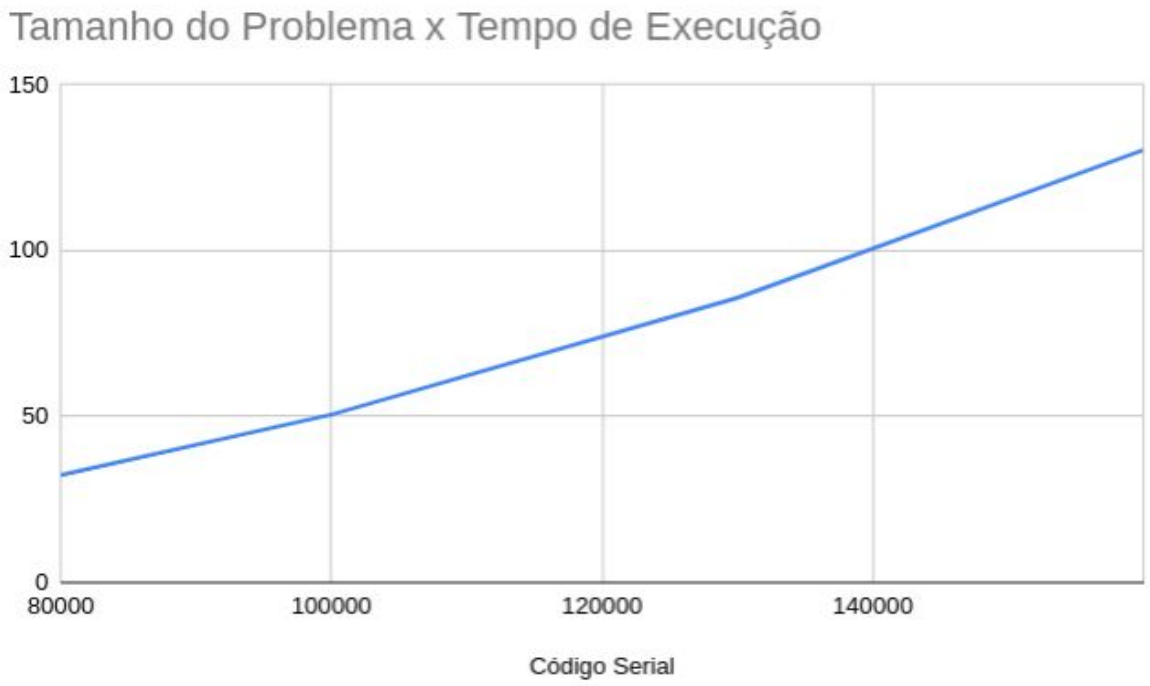
O Shell Script executa o código serial 10 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código serial que escreve o tempo que levou para executar logo depois.

Testes

Ao executar o códigos 10 vezes em cada tamanho do problema fiz uma média como mostra a tabela a seguir

Código Serial	
80000	32,3741064
100000	50,6297051
130000	85,7994692
160000	130,3942423

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



Código Paralelo

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

```
#!/bin/bash

#SBATCH --partition=cluster
#SBATCH --job-name=decvaz
#SBATCH --output=decvazOutput.out
#SBATCH --error=decvazError.err
#SBATCH --time=0-02:00
#SBATCH --hint=compute_bound
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32

#No Supercomputador, 1 nó = 32 Cores (ou CPUs)
#Lembrar que: TASK = PROCESSO
```

```
#A configuração acima reserva 1 nó inteiro (32 cpus no mesmo
processador), vai rodar apenas 1 processo por vez,
# mas este processo terá acesso a 32 cores

#Loop principal de execuções. São 10 tentativas

   tentativas=10 #Quantas vezes o código será executado

   for cores in 4 8 16 32 #números de cores utilizados
   do
              for size in 80000 100000 130000 160000 #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
                  ./teste $cores $size
              done
          done

   done

   exit
```

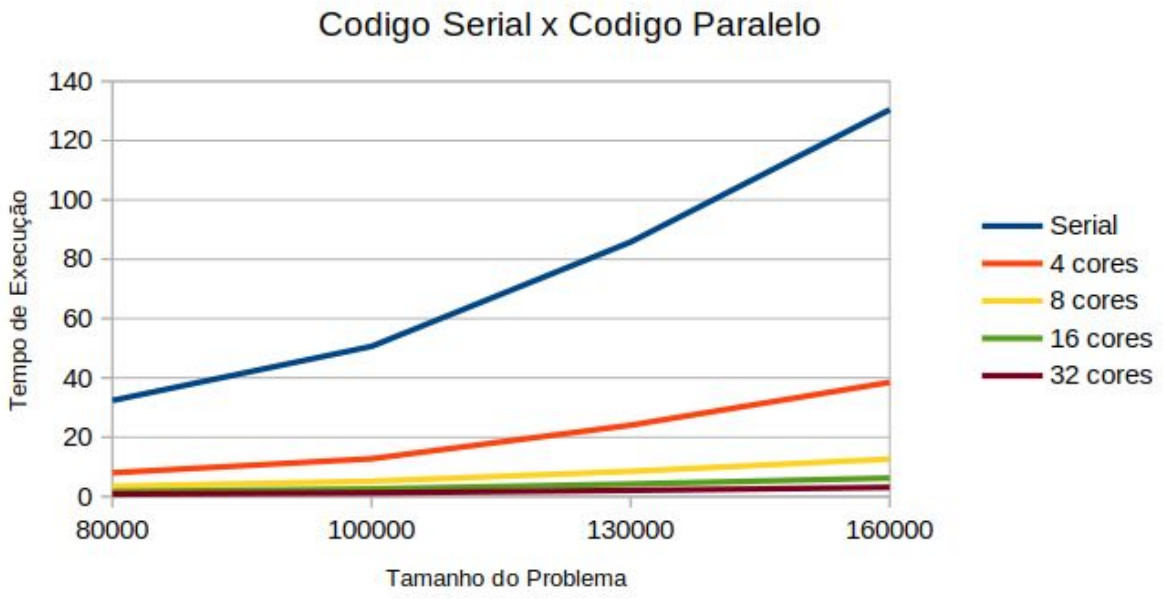
O Shell Script executa o código paralelo 10 vezes, com 4 tamanhos de problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código paralelo que escreve o tempo que levou para executar logo depois.

Testes

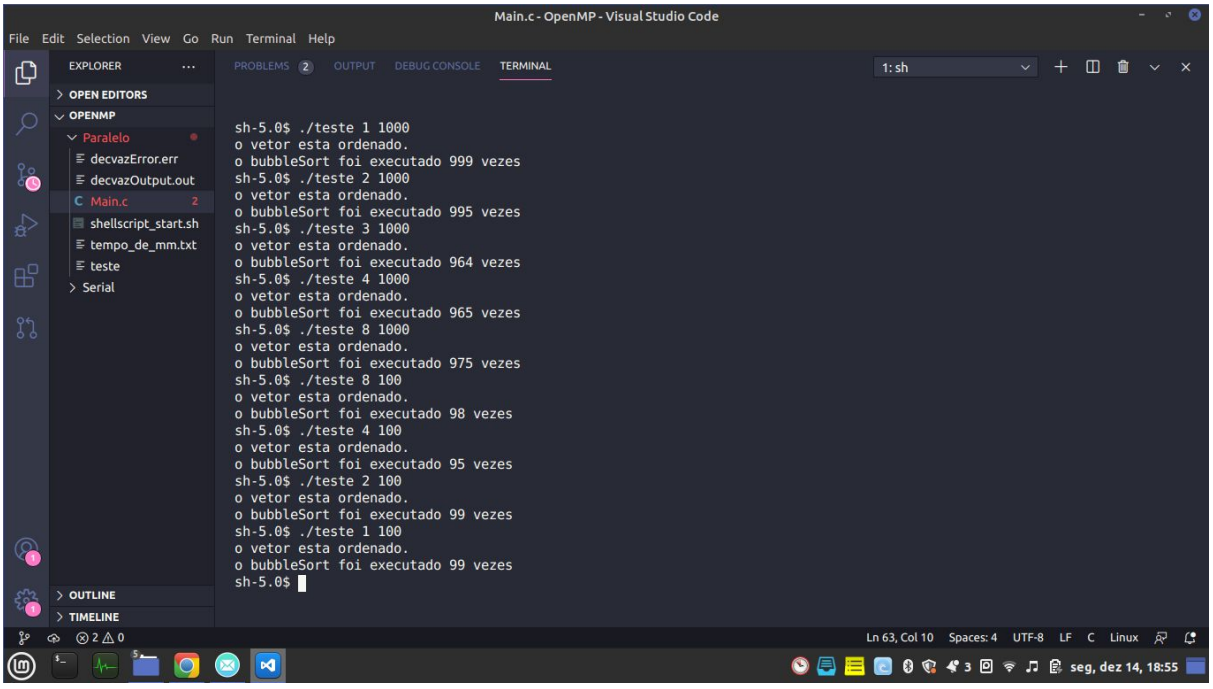
Ao executar o códigos 10 vezes em cada tamanho do problema fiz uma média como mostra a tabela a seguir

Cores	Tamanho do Problema	Tempo de execução (s)
Serial	80000	32,3741064
Serial	100000	50,6297051
Serial	130000	85,7994692
Serial	160000	130,3942423
4	80000	7,9916791
4	100000	12,7531003
4	130000	24,0465801
4	160000	38,5130024
8	80000	3,4167194
8	100000	5,1577353
8	130000	8,4921312
8	160000	12,6482994
16	80000	1,6951645
16	100000	2,5587674
16	130000	4,2067682
16	160000	6,2195929
32	80000	0,8243415
32	100000	1,2488015
32	130000	2,0704428
32	160000	3,0917853

No gráfico a seguir está a comparação do código serial com o paralelo, da pra observar que em todas os casos o Código Paralelo é EXTREMAMENTE mais rápido do que o serial, chegando a menos de um segundo em alguns casos com 32 cores.



A seguir o código paralelo rodando no visual studio code



Análise de Speedup, Eficiência e Escalabilidade

SpeedUp

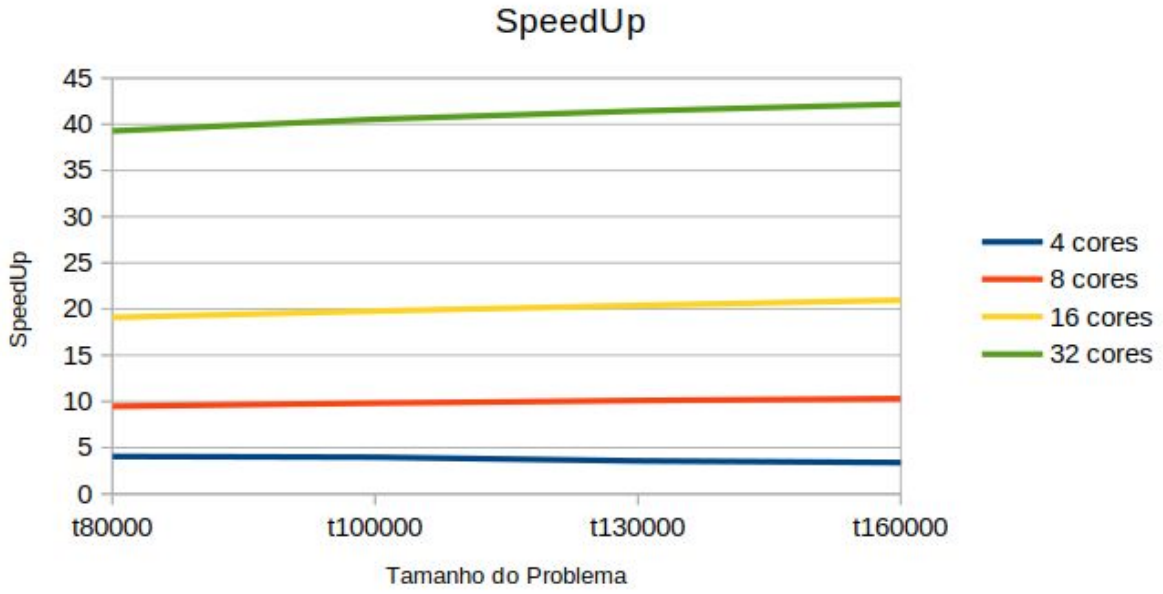
O speedup paralelo S definido como a relação entre o tempo de processamento serial Ts de um algoritmo e seu tempo de processamento paralelo Tp, tal que

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

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				
	t80000	t100000	t130000	t160000
4 cores	4,050976772	3,969991916	3,568052873	3,385719995
8 cores	9,475201973	9,816266666	10,10340834	10,30923116
16 cores	19,09791433	19,78675557	20,39557806	20,96507672
32 cores	39,27268784	40,54263636	41,44015435	42,17441693

Em seguida temos o gráfico de SpeedUp onde podemos observar que o gráfico é constante em todas 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

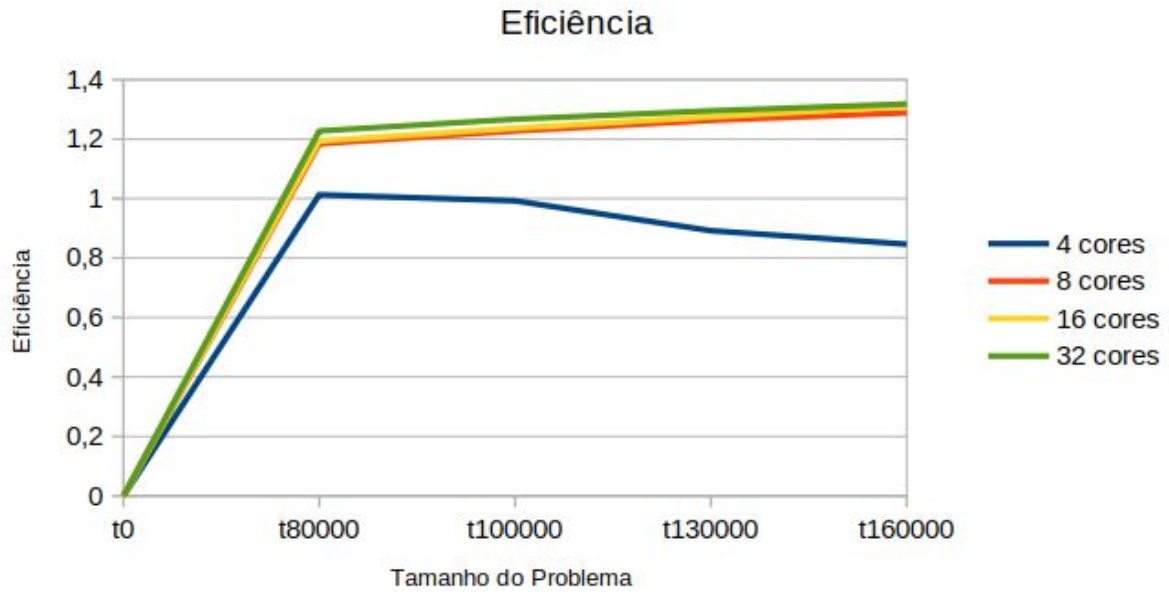
$$E_f = \frac{S}{m}.$$

A seguir a tabela referente a eficiência do código paralelo.

Eficiência				
Tamanho	4 cores	8 cores	16 cores	32 cores
t80000	1,012744193	1,184400247	1,193619646	1,227271495
t100000	0,9924979791	1,227033333	1,236672223	1,266957386
t130000	0,8920132181	1,262926043	1,274723629	1,295004823
t160000	0,8464299988	1,288653895	1,310317295	1,317950529

Comentário do gráfico a seguir na conclusão.

Eficiência



Conclusão

Após vários testes dá pra perceber que o código foi muito eficiente, até demais, pensei sobre as possíveis motivos e um deles foi que o código paralelo está errado de alguma forma, mas em todas a execuções eu rodei uma função no vetor que foi ordenado, e usando a mesma função em vetores que ordenei erradamente, e sempre a função indicava que o vetor não estava ordenado se foi o caso, em todas as execução no supercomputador o código imprimiu em um arquivo que o vetor estava ordenado,e coloquei um contador para ver se o código não estava sendo executado menos vezes do que o serial, sendo assim ,assumindo que o código está sem erros, podemos assumir que o código é fortemente escalável.