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 Pthreads

SUMÁRIO

Introdução

Neste relatório aplicamos o método Pthread, Para multiplicar duas matrizes quadradas

Matrizes quadradas

Basicamente teremos a multiplicação de duas matrizes quadradas de tamanho n

Implementação do Código

Serial

A seguir implementamos a função para multiplicar matrizes quadradas em forma serial na linguagem c. basicamente a função matrizmult recebe duas matrizes que vão se multiplicadas membro por membro da mesma posição depois colocada na terceira matriz recebida

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

int **AlocaMatriz(int m, int n) //função para alocar a matriz dinamicamente
{
   int **M;
   int i;
   M = (int **)malloc(sizeof(int *) * m);
   if (M == NULL)
   {
       printf("Memoria insuficiente.\n");
       exit(1);
   }
   for (i = 0; i < m; i++)
   {
       M[i] = (int *)malloc(sizeof(int) * n);
       if (M[i] == NULL)
       {
           printf("Memoria insuficiente.\n");
           exit(1);
```

```
        }
    }
    return M;
}
void LiberaMatriz(int **M, int m)//função para liberar a matriz
{
    int i;
    for (i = 0; i < m; i++)
        free(M[i]);
    free(M);
}

void matrizmult(int **a, int **b, int **v, int n)//função para
multiplicar duas matrizes e colocar em outra
{

    for (int linha = 0; linha < n; linha++)
    {
        for (int coluna = 0; coluna < n; coluna++)
        {
            v[linha][coluna] = a[linha][coluna] * b[linha][coluna];
        }
    }
}
void imprimirmatriz(int **m, int n, char matriznome) //função para
imprimir determinada matriz
{
    printf("Matriz %c[%i][%i]\n", matriznome, n, n);
    for (int linha = 0; linha < n; linha++)
    {
        for (int coluna = 0; coluna < n; coluna++)
        {
            printf("%i\t", m[linha][coluna]);
        }
        printf("\n");
    }
}
int main(int argc, char const *argv[])
{
    struct timeval start, stop;//variaveis para guardar o tempo

    srand(13);
    double n = atof(argv[1]);
```

```
   int **a = AlocaMatriz(n, n);//alocando matriz a
   int **b = AlocaMatriz(n, n);//alocando matriz b
   int **v = AlocaMatriz(n, n);//alocando matriz v

   for (int linha = 0; linha < n; linha++) //definindo valores
aleatorios para cada posição das matrizes a e b
   {

       for (int coluna = 0; coluna < n; coluna++)
       {
           a[linha][coluna] = rand() % 1000;
           b[linha][coluna] = rand() % 1000;
       }
   }
```

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
   matrizmult(a, b, v, n);//multiplica as matrizes
   //imprimirmatriz(a,n,'a');
   //imprimirmatriz(b,n,'b');
   //imprimirmatriz(v,n,'v');

   gettimeofday(&stop, 0);//finaliza a contagem do tempo

   LiberaMatriz(a, n);//libera a matriz a
   LiberaMatriz(b, n);//libera a matriz b
   LiberaMatriz(v, n);//libera a matriz v
   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);

   return 0;
}close(fp);



   return 0;
}
```

Paralelo

A seguir implementamos a função da multiplicação de matrizes quadradas em forma paralela na linguagem c.A função matrizmult é chamada em forma paralela usando Pthread passando a quantidade de threads determinada, logo depois é dada para cada thread qual parte das matrizes eles devem multiplicar, que em seguida multiplicam membro por membro usando as variáveis globais

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

int thread_count;

int **AlocaMatriz(int m, int n) //função para alocar a matriz
dinamicamente
{
   int **M;
```

```
    int i;
    M = (int **)malloc(sizeof(int *) * m);
    if (M == NULL)
    {
        printf("Memoria insuficiente.\n");
        exit(1);
    }
    for (i = 0; i < m; i++)
    {
        M[i] = (int *)malloc(sizeof(int) * n);
        if (M[i] == NULL)
        {
            printf("Memoria insuficiente.\n");
            exit(1);
        }
    }
    return M;
}
void LiberaMatriz(int **M, int m)//função para liberar a matriz
{
    int i;
    for (i = 0; i < m; i++)
        free(M[i]);
    free(M);
}

void imprimirmatriz(int **m, int n, char matriznome)//função para
imprimir determinada matriz
{
    printf("Matriz %c[%i][%i]\n", matriznome, n, n);
    for (int linha = 0; linha < n; linha++)
    {
        for (int coluna = 0; coluna < n; coluna++)
        {
            printf("%i\t", m[linha][coluna]);
        }
        printf("\n");
    }
}
//variaveis globais
static double n;
int **a;
int **b;
```

```c
int **v;

void *matrizmult(void *rank)//função para  multiplicar as matrizes
usando pthread
{
   long my_rank = (long)rank;
   int local_n = n / thread_count;
   int my_first = my_rank * local_n;
   int my_last = (my_rank + 1) * local_n - 1;

   for (int linha = my_first; linha <= my_last; linha++)
   {
       for (int coluna = 0; coluna < n; coluna++)
       {
           v[linha][coluna] = a[linha][coluna] * b[linha][coluna];
       }
   }
   return NULL;
}

int main(int argc, char const *argv[])
{
   struct timeval start, stop;//variaveis para guardar o temp  o
   gettimeofday(&start, 0);
   srand(13);

   n = atoi(argv[2]);

   a = AlocaMatriz(n, n);//alocando matriz a
   b = AlocaMatriz(n, n);//alocando matriz b
   v = AlocaMatriz(n, n);//alocando matriz v

   for (int linha = 0; linha < n; linha++)//definindo valores
aleatorios para cada posição das matrizes a e b
   {

       for (int coluna = 0; coluna < n; coluna++)
       {
           a[linha][coluna] = rand() % 1000;
           b[linha][coluna] = rand() % 1000;
       }
   }
```

```
   gettimeofday(&start, 0);//começar a marcar o tempo

   long thread;
   pthread_t *thread_handles;

   thread_count = strtol(argv[1], NULL, 10);//variavel que recebe o
numero de threads
   thread_handles = malloc(thread_count * sizeof(pthread_t));

   for (thread = 0; thread < thread_count; thread++)//iniciando a
função matrizmult para cada thread
   {
       pthread_create(&thread_handles[thread], NULL, matrizmult, (void
*)thread);
   }

   for (thread = 0; thread < thread_count; thread++)//esperando cada
thread finalizar o processamento
   {
       pthread_join(thread_handles[thread], NULL);
   }

   gettimeofday(&stop, 0);//finaliza a contagem do tempo

   free(thread_handles);

   //imprimirmatriz(a, n, 'a');
   //imprimirmatriz(b, n, 'b');
   //imprimirmatriz(v, n, 'v');

   LiberaMatriz(a, n);//libera a matriz a
   LiberaMatriz(b, n);//libera a matriz b
   LiberaMatriz(v, n);//libera a matriz v

   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);

   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=0-00:20
#SBATCH --hint=compute_bound

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mem=64000
#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 35000 40000 45000 54000 #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
                    ./mpi_mm $size
                done
            done

   done

   exit
```

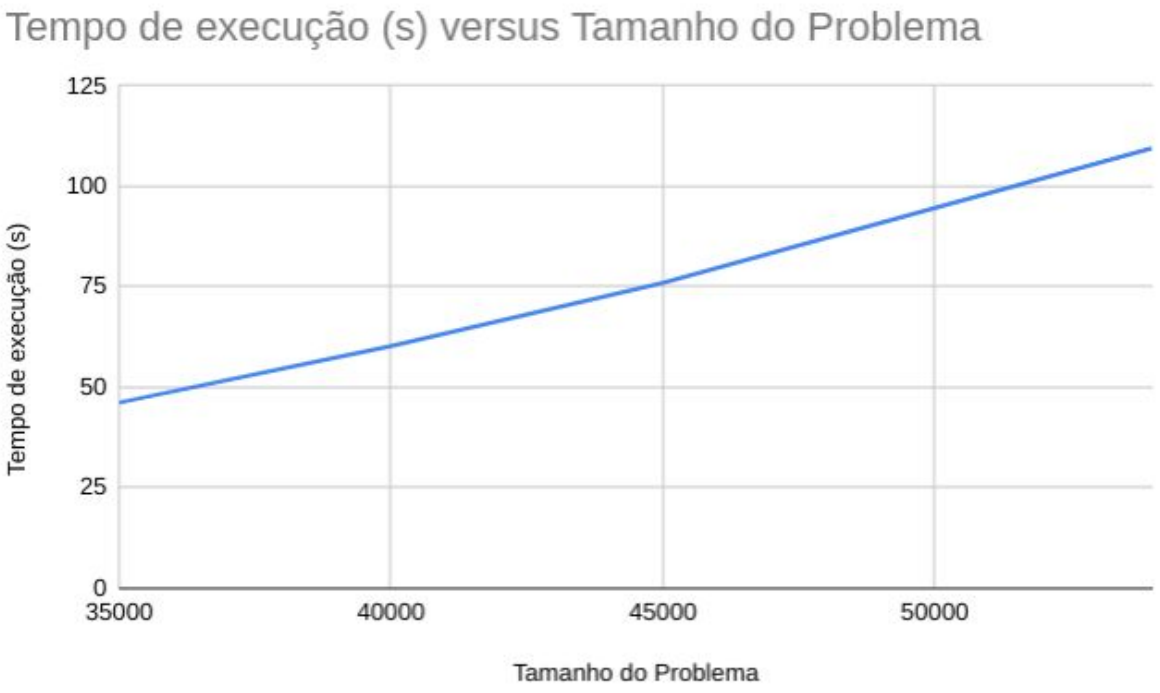
O Shell Script executa o código serial 10 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código serial que escreve o tempo que levou para executar logo depois.

Testes

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

Tamanho do Problema	Tempo de execução (s)
35000	46,2065977
40000	60,2918313
45000	75,9786856
54000	109,5106342

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

#SBATCH --partition=full
#SBATCH --job-name=decvaz
#SBATCH --output=decvazOutput.out
#SBATCH --error=decvazError.err
#SBATCH --time=0-00:50
#SBATCH --hint=compute_bound

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mem=64000
#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 35000 40000 45000 54000 #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
                   ./mpi_mm $cores $size
               done
           done

   done

   exit
```

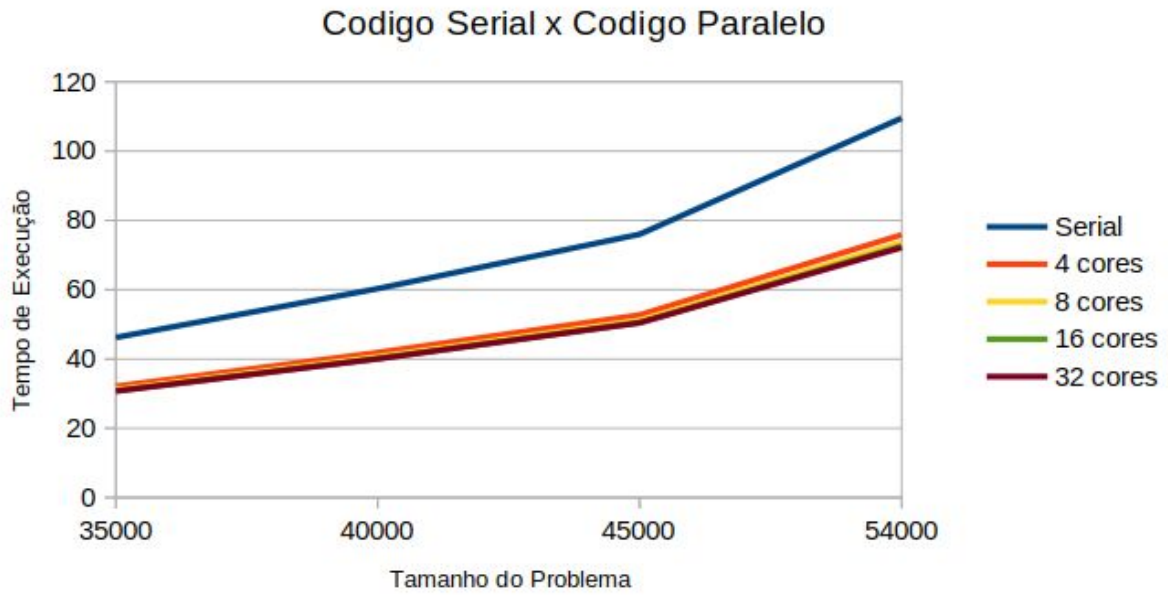
O Shell Script executa o código paralelo 10 vezes, com 4 tamanhos do problema diferentes, e ao final de cada execução escreve o tamanho do problema que junto com o próprio código paralelo que escreve o tempo que levou para executar logo depois.

Testes

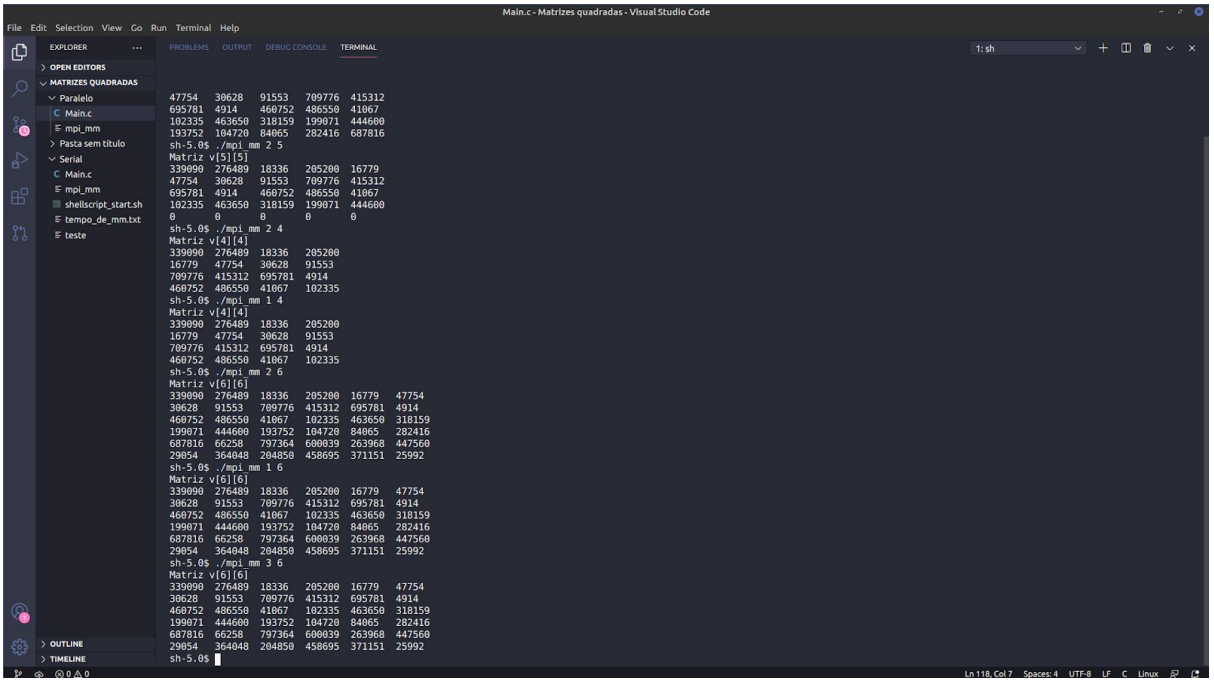
Ao executar o códigos 10 vezes em cada tamanho do problema fiz uma média como mostra a tabela a seguir

Cores	Tamanho do Problema	Tempo de execução (s)
Serial	35000	46,2065977
Serial	40000	60,2918313
Serial	45000	75,9786856
Serial	54000	109,5106342
4	35000	32,1024349
4	40000	41,844199
4	45000	52,6895454
4	54000	75,842796
8	35000	31,1840194
8	40000	40,6444686
8	45000	51,1605463
8	54000	73,6210413
16	35000	30,7343998
16	40000	40,0898433
16	45000	50,4833449
16	54000	72,5848022
32	35000	30,7209362
32	40000	40,0217589
32	45000	50,4310499
32	54000	72,1519967

No gráfico a seguir está a comparação do código serial com o paralelo, da pra observar que com 4 cores ele so tem um pequeno aumento no desempenho,e com 8 cores o desempenho é menor ainda, percebe se que o aumento de desempenho se limita por causa do codigo



A seguir o código paralelo rodando no visual studio code



Análise de Speedup, Eficiência e Escalabilidade

SpeedUp

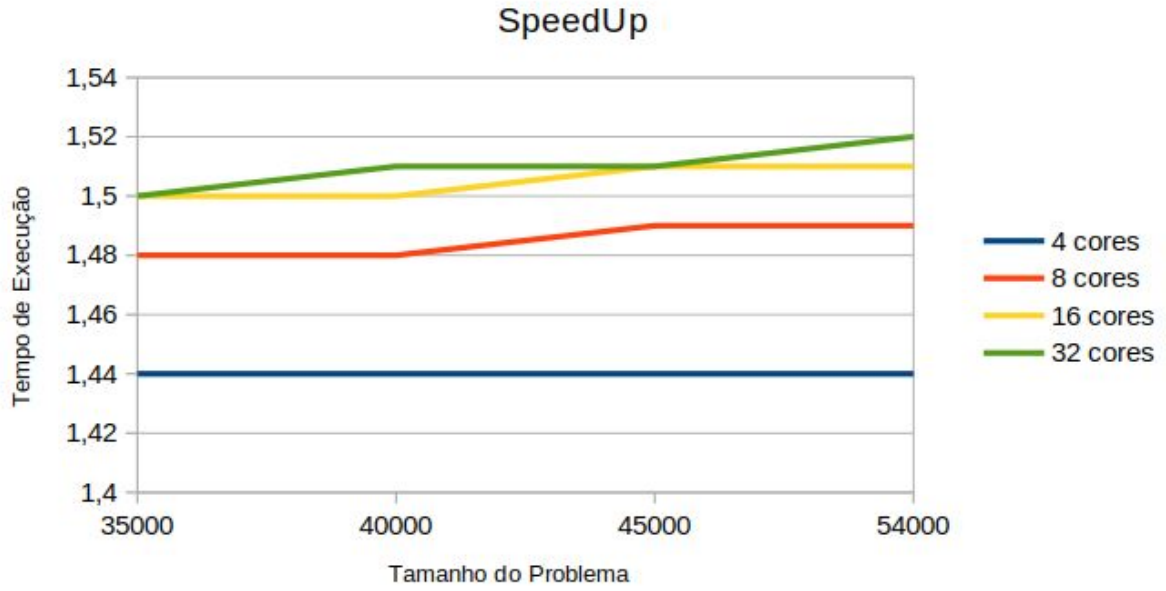
O speedup paralelo S definido como a relação entre o tempo de processamento serial Ts de um algoritmo e seu tempo de processamento paralelo Tp, tal que

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

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				
Tamanho	35000	40000	45000	54000
4 cores	1,44	1,44	1,44	1,44
8 cores	1,48	1,48	1,49	1,49
16 cores	1,50	1,50	1,51	1,51
32 cores	1,50	1,51	1,51	1,52

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

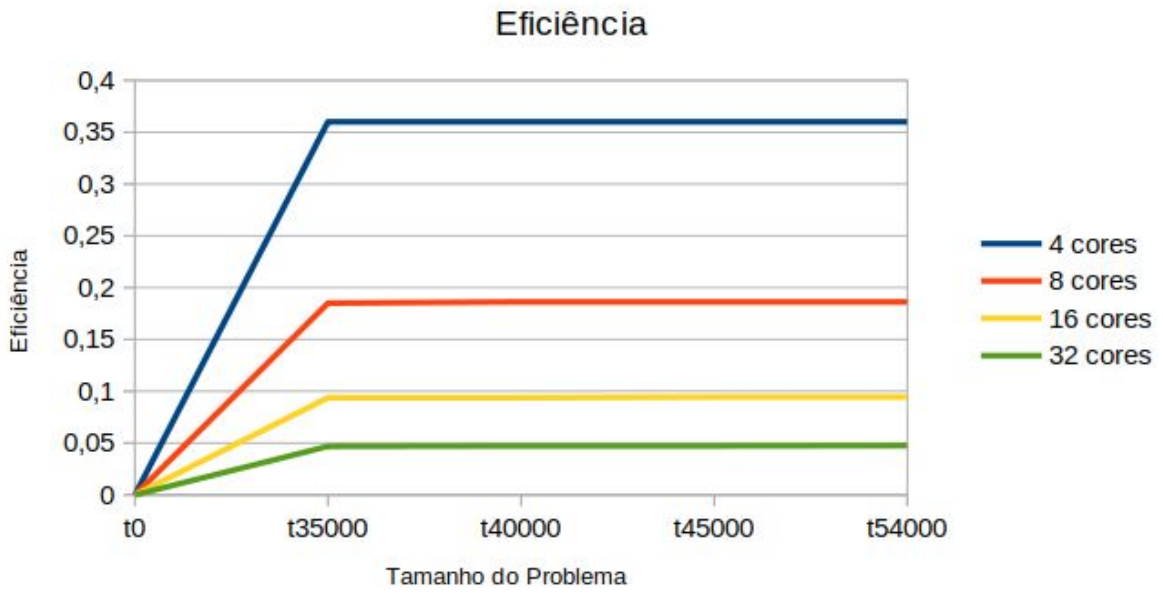
$$E_f = \frac{S}{m}.$$

A seguir a tabela referente a eficiência do código paralelo.

Eficiência			
Tamanho	4 cores	8 cores	16 cores
35000	0,36	0,185	0,09375
40000	0,36	0,18625	0,09375
45000	0,36	0,18625	0,094375
54000	0,36	0,18625	0,094375

Comentário o gráfico a seguir na conclusão.

Eficiência



Conclusão

Após vários testes dá pra perceber que o código não foi muito eficiente nesse caso, provavelmente por causa, da dificuldade que tive de rodar os códigos no supercomputador onde o tamanhos dos problemas era muito grande e pra entregar o trabalho a tempo tive que incluir o tempo de geração de matriz ao tempo total, mas mesmo assim ainda podemos dizer que o código é fracamente escalável.