PARALELIZAÇÃO DO ALGORITMO BUBBLE SORT USANDO OPENMP[1]

Edson Ricardo da Costa[2] <edson.costa@edu.pucrs.br>
Gabriel Fanto Stundner[3] <gabriel.stundner@edu.pucrs.br>

Pontifícia Universidade Católica do Rio Grande do Sul – Escola Politécnica – Curso de Engenharia de Software
Av. Ipiranga, 6681 Prédio 32 Sala 418 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

29 de setembro de 2022

RESUMO

O presente artigo relata como foi implementada a solução para a paralelização do algoritmo Bubble Sort através do uso de diretivas da ferramenta OpenMP, buscando o melhor desempenho, sendo válido para obtenção de nota do primeiro trabalho da disciplina de Programação Paralela.

Palavras-chave: Programação; Bubble; Algoritmo.

ABSTRACT

Title: *"Parallelization of the bubble sort algorithm using OpenMP"*

This article reports how the solution for the parallelization of the Bubble Sort algorithm was implemented through the use of directives from the OpenMP tool, seeking the best performance, being valid for obtaining a grade of the first work of the Parallel Programming discipline.

Key-words: *Schedule; Bubble; Algorithm.*

1 INTRODUÇÃO

Foi requisitado na cadeira de Programação Paralela da Faculdade de Engenharia de Software,na instituição PUCRS para que a gente verifique o funcionamento do Algoritmo do Bubble Sort em sua estrutura sequencial e na forma paralelizada utilizando a API de multi-processos para a linguagem C chamada OpenMP. Este relatório tem como intuito explicar como foi organizado os códigos, as alterações feitas, os testes e mostrar o desempenho do Algoritmo em um conjunto de testes nas duas estruturas de código, no fim mostrando em detalhes como foi o desempenho entre as duas estruturas.

2 ESTRUTURA GERAL

2.1 Processo de Acesso ao grad

O primeiro passo para o desenvolvimento do projeto foi fazer o acesso remoto ao GRAD, que é um servidor criado para testar projetos de desenvolvimento paralelo, onde todos recebem um acesso do professor no inicio da disciplina.

Enquanto estamos dentro da PUCRS podemos ter um acesso rápido e direto ao GRAD pelos computadores logados na rede interna EDUROAM, mas enquanto estamos em outra rede ou precisa ser feito um acesso remoto, precisamos ter acesso ao servidor de uso externo chamado SPARTA.

Para acessar o SPARTA precisamos ter um programa ou console que possa utilizar o SSH para acessos em diferentes máquinas, em nosso caso utilizamos o CMD do Windows para esse acesso, onde já vem por padrão esse programa instalado.

O comando de acesso ao SPARTA precisa ser rodado em um CMD como administrador, utilizando a seguinte estrutura:

```
ssh portoalegre\matricula@sparta.pucrs.br
```

Como mostrado acima, onde está escrito matricula deve ser colocado o código da matricula do aluno, se for o primeiro acesso ele vai pedir que diga que a conexão é segura e que aceita o acesso.

O SPARTA pede uma senha de acesso, essa senha é a sua senha de login nos computadores da PUCRS.

Agora que estamos logados no SPARTA, devemos fazer o acesso ao servidor GRAD, onde devemos usar o seguinte comando no servidor do SPARTA:

```
ssh -o PasswordAuthentication=yes usuarioGRAD@grad..lad.pucrs.br
```

Como visto no acesso ao SPARTA, onde está escrito usuarioGRAD deve ser colocado o código de usuário que você recebeu de seu professor no inicio do semestre, além que foi passado em detalhes como acessar e modificar a senha de acesso, guarde bem o seu código e sua senha para facilitar o acesso quando necessário.

Ele também vai pedir para ver se você concorda em criar a conexão com o GRAD e depois pedir a senha de acesso.

2.2 Preparando o ambiente GRAD

Agora que temos a configuração de acesso preparado, temos que arrumar o nosso usuário do GRAD para podermos rodar os programas do Bubble Sort, onde devemos seguir os seguintes passos.

Primeiro passo é criar o diretório onde iremos trabalhar com o Trabalho 1 da Disciplina de Programação Paralela, que vai ser chamar por nossa convenção de **trab_i**

```
mkdir trab_i
```

Com o comando **mkdir** podemos criar um diretório, que iremos chamar de **trab_i** em referencia

como **trabalho I.**

Iremos acessar o novo diretório utilizando o comando **cd** do Linux:

```
cd trab_i
```

Dentro do novo diretório, iremos criar os diretórios **out** e **tmp,** onde todos os resultados finais dos algoritmos rodados ficarão salvos no diretório **out** e todos os processos temporários ficarão no diretório **tmp**

```
mkdir out
mkdir tmp
```

Iremos trabalhar com os algoritmos direto no diretório **trab_i** onde teremos os arquivos fontes do código salvos nele.

2.3 Criando os arquivos fontes

Foi passado para os alunos o código do algoritmo sequencial do Bubble Sort na linguagem C, onde o professor deixa comentado cada parte da estrutura e alguns exemplos internos, nesse arquivo temos a primeira parte comum de todo arquivo C com os cabeçalhos das bibliotecas que vão ser usadas no programa, onde no caso temos a biblioteca padrão da linguagem C chamada **stdio.h** e a biblioteca da API do OpenMP chamado **omp.h**.

```
#include<stdio.h>
#include<omp.h>
```

No arquivo exemplo também temos algumas constantes que são utilizados dentro do programa, sendo elas um tratamento no tamanho do vetor utilizado no algoritmo.

```
#define MAX_ARRAY_SIZE 25000
#define INI_ARRAY_SIZE 2500
#define INC_ARRAY_SIZE 2500
```

Depois vem o algoritmo do Bubble Sort com a inicialização do Vetor a partir de um valor externo que é passado na hora que é rodado o programa, chamado **NUM_ARRAYS,** como mostra o código abaixo

```
int arrays[NUM_ARRAYS][MAX_ARRAY_SIZE];

void BubbleSort(int n, int* vetor) {
        int c = 0, d, troca, trocou = 1;
        while (c < (n-1) && trocou) {
                trocou = 0;
                for (d = 0 ; d < n - c - 1; d++) {
                        if (vetor[d] > vetor[d+1]) {
                                troca = vetor[d];
                                vetor[d] = vetor[d+1];
                                 vetor[d+1] = troca;
                                 trocou = 1;
                          }

                    }
                c++;
            }
}
```

Esse é o algoritmo do Bubble Sort implementado pelo professor, após isso em nosso arquivo temos a **Main** que é a parte do programa que é inicializado e rodado o algoritmo acima.

Dentro da nossa Main, temos a seguinte estrutura:

```
int main() {
  int i, j, array_size;
  double tempo;

  for (array_size = INI_ARRAY_SIZE; array_size <= MAX_ARRAY_SIZE; array_size += INC_ARRAY_SIZE) {

      // INICIALIZA OS ARRAYS A SEREM ORDENADOS
      #pragma omp parallel for private(j) num_threads(omp_get_num_procs())
      for (i=0 ; i<NUM_ARRAYS; i++) {
          for (j=0 ; j<array_size; j++) {
              if (i%5 == 0)
                 arrays[i][j] = array_size-j;
              else
                 arrays[i][j] = j+1;
          }
      }

      // REALIZA A ORDENACAO
      tempo = -omp_get_wtime();
      for (i=0 ; i<NUM_ARRAYS; i++) {
          BubbleSort(array_size, &arrays[i][0]);
      }
      tempo += omp_get_wtime();

      // VERIFICA SE OS ARRAYS ESTAO ORDENADOS
      for (i=0 ; i<NUM_ARRAYS; i++)
          for (j=0 ; j<array_size-1; j++)
              if (arrays[i][j] > arrays[i][j+1])
                 return 1;

      // MOSTRA O TEMPO DE EXECUCAO
      printf("%d %lf\n",array_size, tempo);
  }
  return 0;
}
```

A nossa main possui a inicialização das variáveis necessárias para o processo, um for onde ele vai andar pelo vetor até o tamanho máximo do vetor definido com a entrada no programa e é feito um for interno que foi paralelizado utilizando o OpenMP que vai pegar o valor do j e fazer um processo onde ele vai fazendo os valores do j internamente ser controlado aos poucos para que não demore tanto tempo de inicializar os valores que vão ser ordenados pelo algoritmo.

Depois de inicializado os valores do Vetor, ele inicia o algoritmo e vai salvando o tempo de duração que levou para ordenar os valores dentro do vetor.

Após ordenado, verificar se os valores estão realmente ordenados e no fim imprime o tempo que levou para ordenar, sendo esse valor salvo em um arquivo **out** que iremos definir.

2.4 Trabalhando encima do programa do Bubble Sort

Agora que foi entendido o programa com o algoritmo do Bubble Sort passado pelo professor, devemos migrar esse programa para o GRAD, onde usamos o programa **nano** do Linux.

Acessamos o servidor GRAD, depois o diretório trab_i e usamos o seguinte comando para criar o arquivo **bss.c**, que é o nosso programa Sequencial do Bubble Sort (Bubble Sort Sequencial = BSS):

```
nano bss.c
```

Com isso, foi aberto o programa nano com um arquivo vazio com o nome que foi passo a ele, depois disso copiamos o código passado pelo professor e colamos no programa, para salvar o arquivo pelo nano usamos os comandos de teclado **Ctrl+O** para salvar as alterações e **Ctrl+X** para fechar o programa.

Como o programa passado pelo professor é o Sequencial, não precisamos mudar nada nesse arquivo, pondendo agora fazer os testes encima dele.

Para termos o Bubble Sort Paralelo precisamos fazer uma alteração no programa, onde iremos criar um novo arquivo no trab_i chamado **bsp.c** (Bubble Sort Paralelo = BSP).

```
nano bsp.c
```

Ele é idêntico ao BSS tirando uma alteração feito na linha 47, onde no momento de fazer a ordenação é utilizando o OpenMP para dividir em Threads definidas na entrada do programa para realizar o algoritmo, como mostra na imagem abaixo:

```
45      // REALIZA A ORDENACAO
46      tempo = -omp_get_wtime();
47      #pragma omp parallel for num_threads(NUM_THREADS)
48      for (i=0 ; i<NUM_ARRAYS; i++) {
49          BubbleSort(array_size, &arrays[i][0]);
50      }
51      tempo += omp_get_wtime();
52
```

Com essa alteração na linha 47 podemos definir quantas Threads quisermos para agilizar o algoritmo do Bubble Sort para vermos se isso diminui o tempo de conclusão da ordenação, onde iremos testar com diferentes tamanhos de Vetor e diferentes números de Threads, esse é o momento que foi especificado para nós fazermos.

2.5 Testes Sequenciais

Foi passado para nós na descrição do trabalho que devemos trabalhar com Vetores de 3 tamanhos sendo eles 25, 150 e 500.

Para testarmos, o primeiro passo é criar o arquivo compilado de cada um dos tamanhos listados, onde podemos utilizar os seguintes comandos para passarmos ao nosso programa o tamanho do vetor e o nome do arquivo que vai ser gerado que vai ficar no nosso diretório principal do trabalho chamado trab_i.

Os comandos são:

```
gcc -o bss-25 -DNUMARRAYS=25 -fopenmp bss.c
```

```
gcc -o bss-150 -DNUMARRAYS=150 -fopenmp bss.c
```

```
gcc -o bss-500 -DNUMARRAYS=500 -fopenmp bss.c
```

Esses comandos estão dizendo que eles vão usar o compilador do C chamado GCC onde vai enviar os dados compilados para um arquivo, como por exemplo o **bss-25** e passando os valores para o NUM_ARRAYS interno do programa em C o valor de cada tamanho definido na tarefa, além disso está chamando a biblioteca do OpenMP antes de chamar o arquivo com o programa sequencial.

Depois de criado o arquivo compilado com os 3 tamanhos definidos, o seu GRAD deve estar mais ou menos dessa forma:

```
pp03008@grad:~/trab_i$ ls
bsp.c  bss-150  bss-25  bss-500  bss.c  out  tmp
pp03008@grad:~/trab_i$
```

Agora, devemos criar o arquivo **Batchjob** para pegar os tempos que serão gerados por cada um dos compilados de teste, como o seguinte arquivo que criamos para os testes sequenciais:

```
 1 #!/bin/bash
 2 #SBATCH --export=ALL
 3 #SBATCH -N 1
 4 #SBATCH -t 300
 5 #SBATCH --exclusive
 6 #SBATCH --no-requeue
 7 #SBATCH -o bjs.out
 8 #SBATCH -D /home/pp03008/trab_i
 9 echo bss-25
10 echo Initial Time is 'date'
11 ./bss-25 > ./tmp/bss-25.tmp
12 echo Final Time is 'date'
13 echo bss-150
14 echo Initial Time is 'date'
15 ./bss-150 > ./tmp/bss-150.tmp
16 echo Final Time is 'date'
17 echo bss-500
18 echo Initial Time is 'date'
19 ./bss-500 > ./tmp/bss-500.tmp
20 echo Final Time is 'date'
```

Para uma descrição do que o programa faz, da linha 1 ao 8 são as configurações que o Batchjob precisa para rodar o programa, onde na linha 8 colocamos o caminho até o diretório do nosso trabalho 1, onde mostra o meu usuário no GRAD chamado pp03008.

Esse programa vai pegar cada um dos programas compilados do Sequencial e enviando os dados para arquivos temporários e impresso no console do GRAD os tempos levados para rodar cada um dos arquivos criados temporários e também os tempos levados para ordenar os dados.

Para rodar o batchjob no GRAD, devemos usar o seguinte comando:

```
sbatch bjs
```

Com esse comando ele vai criar um arquivo chamado **bjs.out** e deve mostrar a seguinte mensagem

```
pp03008@grad:~/trab_i$ sbatch bjs
Submitted batch job 5009
```

```
pp03008@grad:~/trab_i$ ls
bjs  bjs.out  bsp.c  bss-150  bss-25  bss-500  bss.c  out  tmp
```

O que ele fez foi salvar em arquivos temporários com os tempos levados, sendo salvos no diretório **/tmp** e depois de rodado esse batchjob ele vai criar os seguintes arquivos

```
pp03008@grad:~/trab_i/tmp$ ls
bss-150.tmp  bss-25.tmp  bss-500.tmp
pp03008@grad:~/trab_i/tmp$
```

Se abrirmos qualquer um desses arquivos, podemos ver o tempo que levam para cada número de Vetores desde o tamanho inicial 2500 até o tamanho máximo 25000, abaixo o caso do sequencial com 25 vetores:

```
2500 0.145271
5000 0.563803
7500 1.266211
10000 2.244684
12500 3.509148
15000 5.057411
17500 6.879037
20000 8.985266
22500 11.369629
25000 14.033251
```

Além de ter os tempos, podemos ver a fila de processos que devem ser rodados, utilizando o comando **squeue**:

```
pp03008@grad:~/trab_i/tmp$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
              4813 LAD_geral testepar  pp03007 PD       0:00     16 (PartitionNodeLimit)
              4814 LAD_geral testepar  pp03007 PD       0:00     16 (PartitionNodeLimit)
              5009 LAD_geral      bjs  pp03008  R      19:37      1 grad01
```

Agora que temos os tempos dos testes sequenciais, temos os seguintes valores

TAMANHO	25 VETORES	150 VETORES	500 VETORES
2500	0.145271	0.848389	2.811093
5000	0.563803	3.370703	11.214021
7500	1.266211	7.560595	25.203426
10000	2.244684	13.451512	44.835352
12500	3.509148	21.032109	70.113094
15000	5.057411	30.312985	100.996584
17500	6.879037	41.262056	137.531501
20000	8.985266	53.897714	179.640866
22500	11.369629	68.198829	227.392042
25000	14.033251	84.198272	280.749107

Criando um gráfico no excel fica dessa forma:

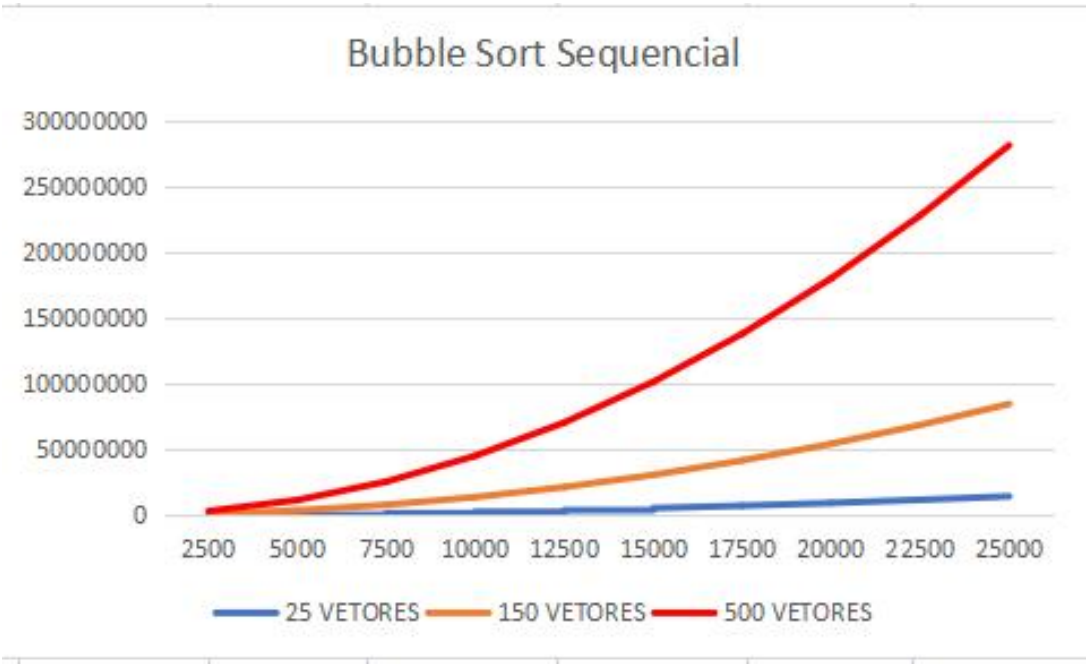

Tabela 1 - Gráfico do Bubble Sort Sequencial

2.6 Testes em Paralelo

Agora que verificamos os valores do Algoritmo Sequencial do Bubble Sort, vamos fazer os testes do Algoritmo Paralelo, onde devemos rodar 4 testes para cada tamanho do vetor, em cada tamanho (25,150 e 500) iremos testar com 2,4,8,16 Threads cada, seguindo por esse roteiro:

NÚMERO DE VETORES	NÚMERO DE THREADS	ARQUIVO GERADO
25 Vetores	2 Threads	bsp-25.2
25 Vetores	4 Threads	bsp-25.4
25 Vetores	8 Threads	bsp-25.8
25 Vetores	16 Threads	bsp-25.16
150 Vetores	2 Threads	bsp-150.2
150 Vetores	4 Threads	bsp-150.4
150 Vetores	8 Threads	bsp-150.8
150 Vetores	16 Threads	bsp-150.16
500 Vetores	2 Threads	bsp-500.2
500 Vetores	4 Threads	bsp-500.4
500 Vetores	8 Threads	bsp-500.8
500 Vetores	16 Threads	bsp-500.16

Para cada um deles iremos rodar os seguinte comando

```
gcc -o bsp-25.2 -DNUM_ARRAYS=25 -DNUM_THREADS=2 -fopenmp bsp.c
```

Estamos passando nesse comando o tamanho do vetor e o número de Threads que iremos utilizar, onde são salvos no arquivos definidos e ficam salvos no diretório **trab_i.**

Rodando todos os comandos seguindo esse padrão, ele vai criar um arquivo compilado do nosso Bubble Sort Paralelo para cada um deles, como na imagem do GRAD abaixo:

```
pp03008@grad:~/trab_i$ ls
bjs  bjs.out  bsp.c  bss-150  bss-25  bss-500  bss.c  out  tmp
pp03008@grad:~/trab_i$ gcc -o bsp-25.2 -DNUM_ARRAYS=25 -DNUM_THREADS=2 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-25.4 -DNUM_ARRAYS=25 -DNUM_THREADS=4 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-25.8 -DNUM_ARRAYS=25 -DNUM_THREADS=8 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-25.16 -DNUM_ARRAYS=25 -DNUM_THREADS=16 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-150.2 -DNUM_ARRAYS=150 -DNUM_THREADS=2 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-150.4 -DNUM_ARRAYS=150 -DNUM_THREADS=4 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-150.8 -DNUM_ARRAYS=150 -DNUM_THREADS=8 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-150.16 -DNUM_ARRAYS=150 -DNUM_THREADS=16 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-500.2 -DNUM_ARRAYS=500 -DNUM_THREADS=2 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-500.4 -DNUM_ARRAYS=500 -DNUM_THREADS=4 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-500.8 -DNUM_ARRAYS=500 -DNUM_THREADS=8 -fopenmp bsp.c
pp03008@grad:~/trab_i$ gcc -o bsp-500.16 -DNUM_ARRAYS=500 -DNUM_THREADS=16 -fopenmp bsp.c
pp03008@grad:~/trab_i$ ls
bjs      bsp-150.16  bsp-150.4  bsp-25.16  bsp-25.4  bsp-500.16  bsp-500.4  bsp.c    bss-25   bss.c  tmp
bjs.out  bsp-150.2   bsp-150.8  bsp-25.2   bsp-25.8  bsp-500.2   bsp-500.8  bss-150  bss-500  out
pp03008@grad:~/trab_i$
```

Agora que temos os arquivos compilados de todos os processos desejados, vamos criar o Batchjob para pegarmos os tempos de cada um deles.

2.6.1 Paralelo de 2 Threads

Vamos criar um Batchjob para avaliar os programas de cada um que possuem 2 Threads como no programa abaixo criado.

```
#!/bin/bash
#SBATCH --export=ALL
#SBATCH -N 1
#SBATCH -t 300
#SBATCH --exclusive
#SBATCH --no-requeue
#SBATCH -o bjp2.out
#SBATCH -D /home/pp03008/trab_i
echo bsp-25.2
echo Initial Time is `date`
./bsp-25.2 > ./tmp/bsp-25.2.tmp
echo Final Time is `date`
echo bsp-150.2
echo Initial Time is `date`
./bsp-150.2 > ./tmp/bsp-150.2.tmp
echo Final Time is `date`
echo bsp-500.2
echo Initial Time is `date`
./bsp-500.2 > ./tmp/bsp-500.2.tmp
echo Final Time is `date`
```

Ele vai fazer o mesmo que o programa do Sequencial fazia, ele vai salvar os tempos levados dentro de arquivos temporários na pasta **/tmp**

Os tempos levados para rodar o programa de 2 Threads foram

```
pp03008@grad:~/trab_i$ cat bjp2.out
bsp-25.2
Initial Time is Thu 29 Sep 2022 02:49:00 PM -03
Final Time is Thu 29 Sep 2022 02:49:33 PM -03
bsp-150.2
Initial Time is Thu 29 Sep 2022 02:49:33 PM -03
Final Time is Thu 29 Sep 2022 02:52:15 PM -03
bsp-500.2
Initial Time is Thu 29 Sep 2022 02:52:15 PM -03
```

2.6.2 Paralelo de 4 Threads

Vamos criar um Batchjob para os programas que utilizaram as 4 Threads, como abaixo:

```
#!/bin/bash
#SBATCH --export=ALL
#SBATCH -N 1
#SBATCH -t 300
#SBATCH --exclusive
#SBATCH --no-requeue
#SBATCH -o bjp4.out
#SBATCH -D /home/pp03008/trab_i
echo bsp-25.4
echo Initial Time is `date`
./bsp-25.4 > ./tmp/bsp-25.4.tmp
echo Final Time is `date`
echo bsp-150.4
echo Initial Time is `date`
./bsp-150.4 > ./tmp/bsp-150.4.tmp
echo Final Time is `date`
echo bsp-500.4
echo Initial Time is `date`
./bsp-500.4 > ./tmp/bsp-500.4.tmp
echo Final Time is `date`
```

Tempo que levou para rodar o programa de 4 Threads foram

```
pp03008@grad:~/trab_i$ cat bjp4.out
bsp-25.4
Initial Time is Thu 29 Sep 2022 02:54:35 PM -03
Final Time is Thu 29 Sep 2022 02:54:57 PM -03
bsp-150.4
Initial Time is Thu 29 Sep 2022 02:54:57 PM -03
Final Time is Thu 29 Sep 2022 02:56:27 PM -03
bsp-500.4
Initial Time is Thu 29 Sep 2022 02:56:27 PM -03
```

2.6.3 Paralelo de 8 Threads

Arquivo Batchjob para 8 Threads:

```
#!/bin/bash
#SBATCH --export=ALL
#SBATCH -N 1
#SBATCH -t 300
#SBATCH --exclusive
#SBATCH --no-requeue
#SBATCH -o bjp8.out
#SBATCH -D /home/pp03008/trab_i
echo bsp-25.8
echo Initial Time is `date`
./bsp-25.8 > ./tmp/bsp-25.8.tmp
echo Final Time is `date`
echo bsp-150.8
echo Initial Time is `date`
./bsp-150.8 > ./tmp/bsp-150.8.tmp
echo Final Time is `date`
echo bsp-500.8
echo Initial Time is `date`
./bsp-500.8 > ./tmp/bsp-500.8.tmp
echo Final Time is `date`
```

Tempo que levou para rodar o programa de 8 Threads:

```
pp03008@grad:~/trab_i$ cat bjp8.out
bsp-25.8
Initial Time is Thu 29 Sep 2022 03:01:06 PM -03
Final Time is Thu 29 Sep 2022 03:01:19 PM -03
bsp-150.8
Initial Time is Thu 29 Sep 2022 03:01:19 PM -03
Final Time is Thu 29 Sep 2022 03:02:06 PM -03
bsp-500.8
Initial Time is Thu 29 Sep 2022 03:02:06 PM -03
```

2.6.4 Paralelo de 16 Threads

Arquivo batchjob para 16 Threads:

```
#!/bin/bash
#SBATCH --export=ALL
#SBATCH -N 1
#SBATCH -t 300
#SBATCH --exclusive
#SBATCH --no-requeue
#SBATCH -o bjp16.out
#SBATCH -D /home/pp03008/trab_i
echo bsp-25.16
echo Initial Time is `date`
./bsp-25.16 > ./tmp/bsp-25.16.tmp
echo Final Time is `date`
echo bsp-150.16
echo Initial Time is `date`
./bsp-150.16 > ./tmp/bsp-150.16.tmp
echo Final Time is `date`
echo bsp-500.16
echo Initial Time is `date`
./bsp-500.16 > ./tmp/bsp-500.16.tmp
echo Final Time is `date`
```

Tempo que levou para rodar o programa de 16 Threads:

```
pp03008@grad:~/trab_i$ cat bjp16.out
bsp-25.16
Initial Time is Thu 29 Sep 2022 03:04:41 PM -03
Final Time is Thu 29 Sep 2022 03:04:52 PM -03
bsp-150.16
Initial Time is Thu 29 Sep 2022 03:04:52 PM -03
Final Time is Thu 29 Sep 2022 03:05:41 PM -03
bsp-500.16
Initial Time is Thu 29 Sep 2022 03:05:41 PM -03
```

2.6.5 Verificação dos tempos paralelos

A primeira coisa que podemos perceber que quanto mais Threads tiver, menos tempo leva para o batchjob ser executado, onde o de 2 Threads levou muito tempo para finalizar e sumir do **squeue.**

Agora, temos o GRAD com todos os arquivos necessários para avaliação dos dados, onde o nosso GRAD no diretório **trab_i** ficou assim:

```
pp03008@grad:~/trab_i$ ls
bjp16      bjp2.out  bjp8      bjs.out     bsp-150.4  bsp-25.2  bsp-500.16  bsp-500.8  bss-25   out
bjp16.out  bjp4      bjp8.out  bsp-150.16  bsp-150.8  bsp-25.4  bsp-500.2   bsp.c      bss-500  tmp
bjp2       bjp4.out  bjs       bsp-150.2   bsp-25.16  bsp-25.8  bsp-500.4   bss-150    bss.c
```

Os nossos arquivos temporários ficaram assim:

```
pp03008@grad:~/trab_i/tmp$ ls
bsp-150.16.tmp  bsp-150.4.tmp  bsp-25.16.tmp  bsp-25.4.tmp  bsp-500.16.tmp  bsp-500.4.tmp  bss-150.tmp  bss-500.tmp
bsp-150.2.tmp   bsp-150.8.tmp  bsp-25.2.tmp   bsp-25.8.tmp  bsp-500.2.tmp   bsp-500.8.tmp  bss-25.tmp
```

Criando as colunas no Excel, tivemos os seguintes tempos ocorridos durante a ordenação do Bubble Sort:

2 Threads

TAMANHO	25 VETORES	150 VETORES	500 VETORES
2500	0.089040	0.437959	1.418085
5000	0.342705	1.702341	5.658364
7500	0.764872	3.819259	12.610441
10000	1.350071	6.812948	22.637582
12500	2.123446	10.612460	35.387975
15000	3.057697	15.155849	50.542067
17500	4.157397	20.635187	69.414170
20000	5.430380	26.961461	90.208068
22500	6.870220	34.117302	114.295068
25000	8.472090	42.456950	140.437870

4 Threads

TAMANHO	25 VETORES	150 VETORES	500 VETORES
2500	0.089728	0.356551	0.727601
5000	0.358339	1.087655	3.323521
7500	0.744450	2.040596	6.560009
10000	0.905611	3.621209	11.479186
12500	1.409052	6.040686	18.024650
15000	2.406080	8.440838	25.887586
17500	2.747725	11.599201	35.115578
20000	3.597891	14.847584	46.228246
22500	4.621597	18.588149	58.779251
25000	5.686604	23.000723	72.311338

8 Threads

TAMANHO	25 VETORES	150 VETORES	500 VETORES
2500	0.062400	0.197695	0.477011
5000	0.181038	0.617625	1.698619
7500	0.402832	1.288746	3.505712
10000	0.715274	2.209626	6.133914
12500	1.086809	3.029291	9.831776
15000	1.089384	4.312581	14.155795
17500	1.463849	5.876826	18.856177
20000	1.981115	7.610784	24.701886
22500	2.433470	9.648986	31.375474
25000	3.149969	11.869321	38.730086

16 Threads

TAMANHO	25 VETORES	150 VETORES	500 VETORES
2500	0.049016	0.133997	0.426847
5000	0.136922	0.513404	1.654682
7500	0.269870	1.160235	3.739618
10000	0.481178	2.057494	6.619397
12500	0.749150	3.208174	10.353273
15000	1.070980	4.615182	14.899664
17500	1.471243	6.275613	20.274100
20000	1.896840	8.167229	26.407447
22500	2.398935	10.049538	33.527564
25000	2.962104	12.766063	41.532838

Esses foram todos os tempos gerados pelos Programas Paralelos utilizando o Batchjob, com isso podemos fazer os gráficos de tempo, Speedup e de Eficiência

2.7 Avaliação dos dados encontrados

Agora que temos os dados, vamos fazer uma avaliação de performance de tempo com todos os dados que temos, onde o primeiro é a performance de Tempo do processo de cada um dos paralelos.

2.7.1 Avaliação de Tempo

Pegando os valores dos tempos de 2,4,8,16 Threads com 25 Vetores temos essa avaliação feita no gráfico:

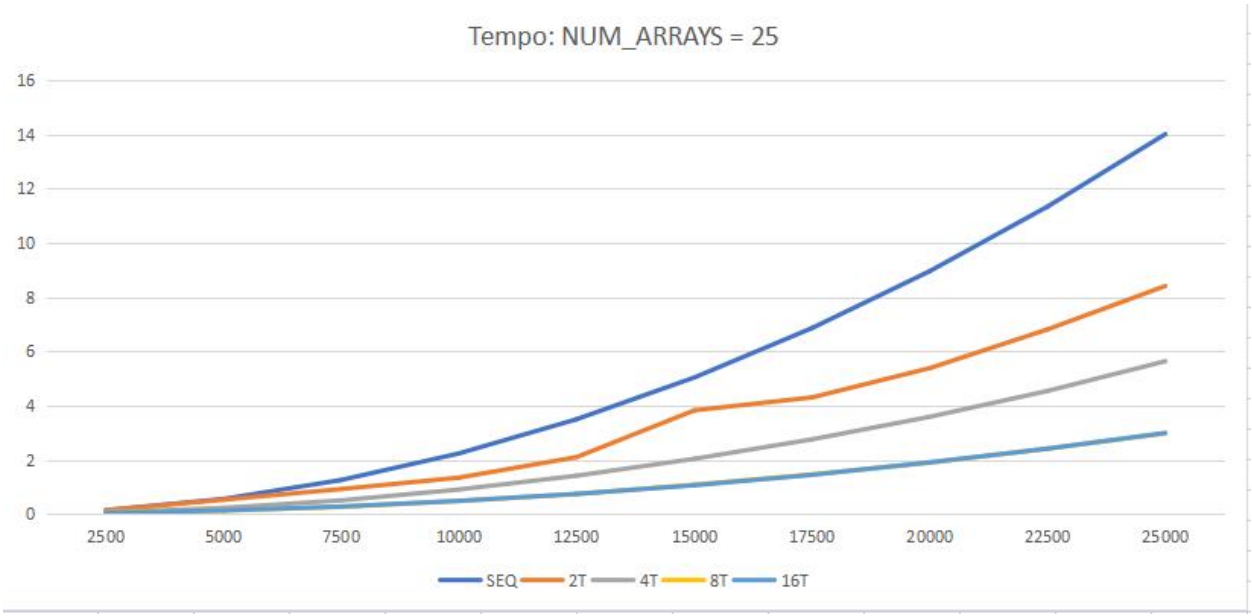

Gráfico 2 - Tempo com 25 vetores

Podemos avaliar que o Sequencial leva muito mais tempo do que os processos utilizando Threads, onde com somente 25 vetores o que leva menos tempo é o de 16 Threads, não conseguimos ver o de 8 Threads porque ele leva quase o mesmo tempo que o de 16 Threads para esse tamanho de vetores.

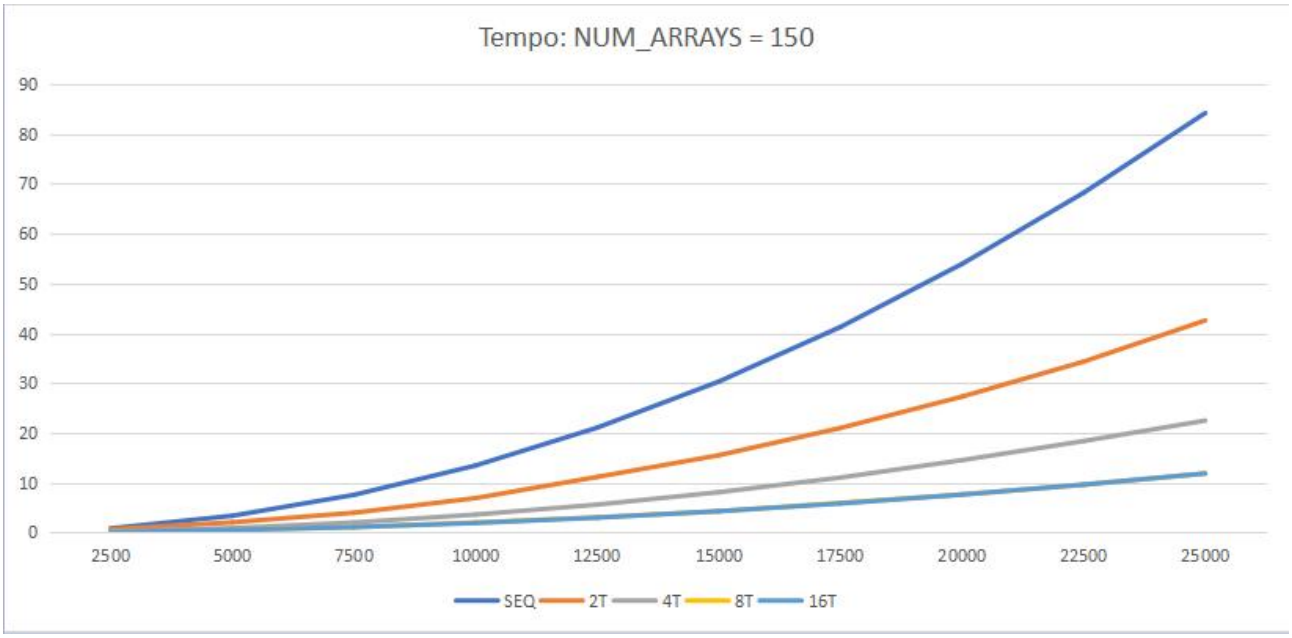

Gráfico 3 - Tempo com 150 vetores

De novo o sequencial é o que leva mais tempo, mas podemos ver que com 2 Threads não tem uma curva de tempo como a que tem 25 vetores, também vemos que o de 8 Threads leva quase o mesmo tempo que o de 16 Threads para esse tamanho de vetores.

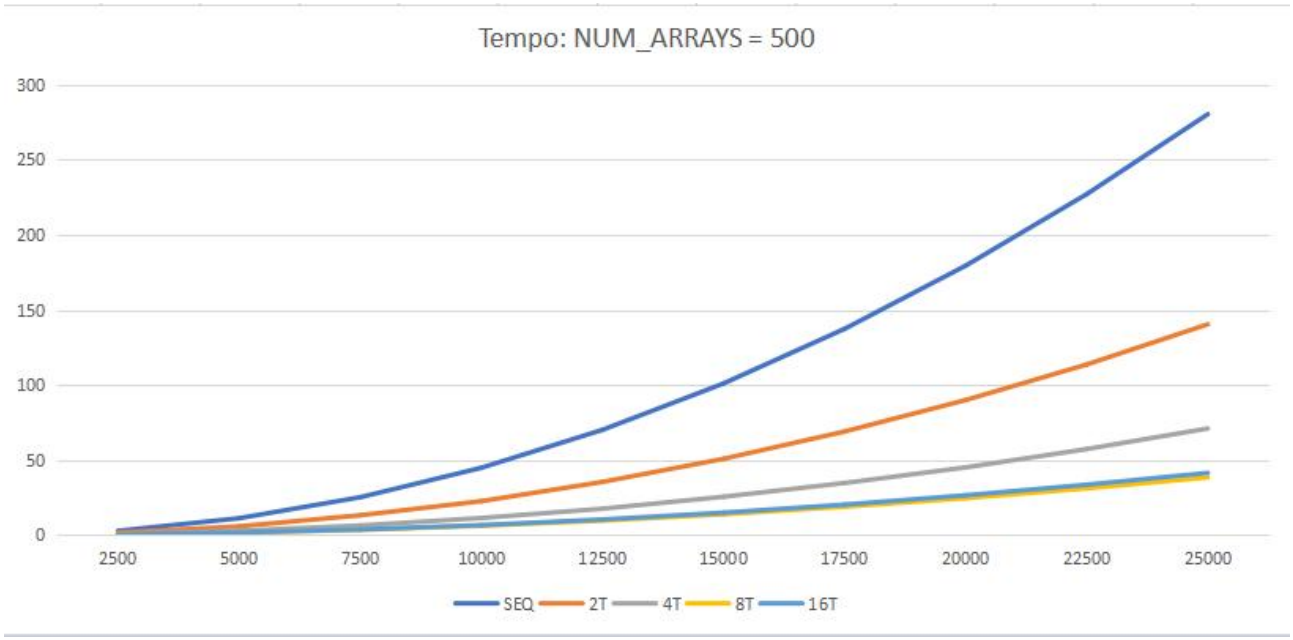

Grafico 4 - Tempo com 500 vetores

Aqui podemos ver que começa a aparecer a linha de 8 Threads, onde conseguimos ver que quantos mais vetores tivermos melhor vamos vendo que quem leva menos tempo são 8 Threads de vez de 16, porque podemos ver que começa a aparecer a linha mais abaixo das outras.

2.7.2 Avaliação de SpeedUp

Speedup é um valor calculado em relação a aceleração de desempenho relativo a dois sistemas ou mais que processam o mesmo problema, é um cálculo feito verificando o tempo gasto pelo número de processos, como mostra a imagem abaixo:

Speedup pode ser definido como a relação entre o tempo gasto para executar uma tarefa com um único processador e o tempo gasto com N processadores, ou seja, Speedup é a Medida do ganho em tempo.

$$S = \frac{T(1)}{T(N)}$$

Onde 'S' é o speedup e 'T'(N) é o tempo gasto para 'N' processadores

Definição

Fórmulas:

- $n \in \mathbb{N}$, o número de threads de execução,
- $B \in [0, 1]$, fração de um algoritmo estritamente serial,

O tempo $T(n)$ que um algoritmo demora para terminar a execução utilizando n thread(s) de execução, corresponde a:

$$T(n) = T(1)\left(B + \frac{1}{n}(1 - B)\right)$$

Portanto,o speedup teórico $S(n)$ que pode ser obtido pela execução de um dado algoritmo, em um sistema capaz da execução de n threads de execução, é:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)\left(B + \frac{1}{n}(1 - B)\right)} = \frac{1}{B + \frac{1}{n}(1 - B)}$$

Imagem 15 - Fonte: https://pt.wikipedia.org/wiki/Lei_de_Amdahl

Com as nossas Threads chegamos a uma avaliação dos valores e apresentando em Gráfico como abaixo:

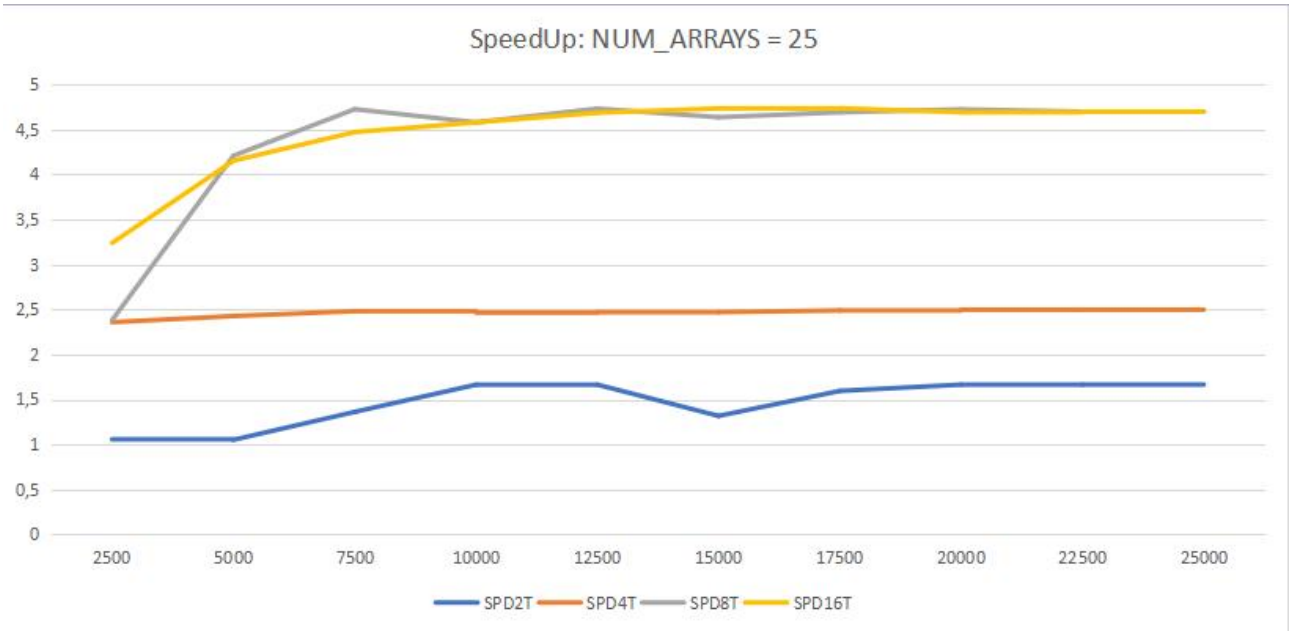

Gráfico 5 - Speedup dos tempos com 25 Vetores

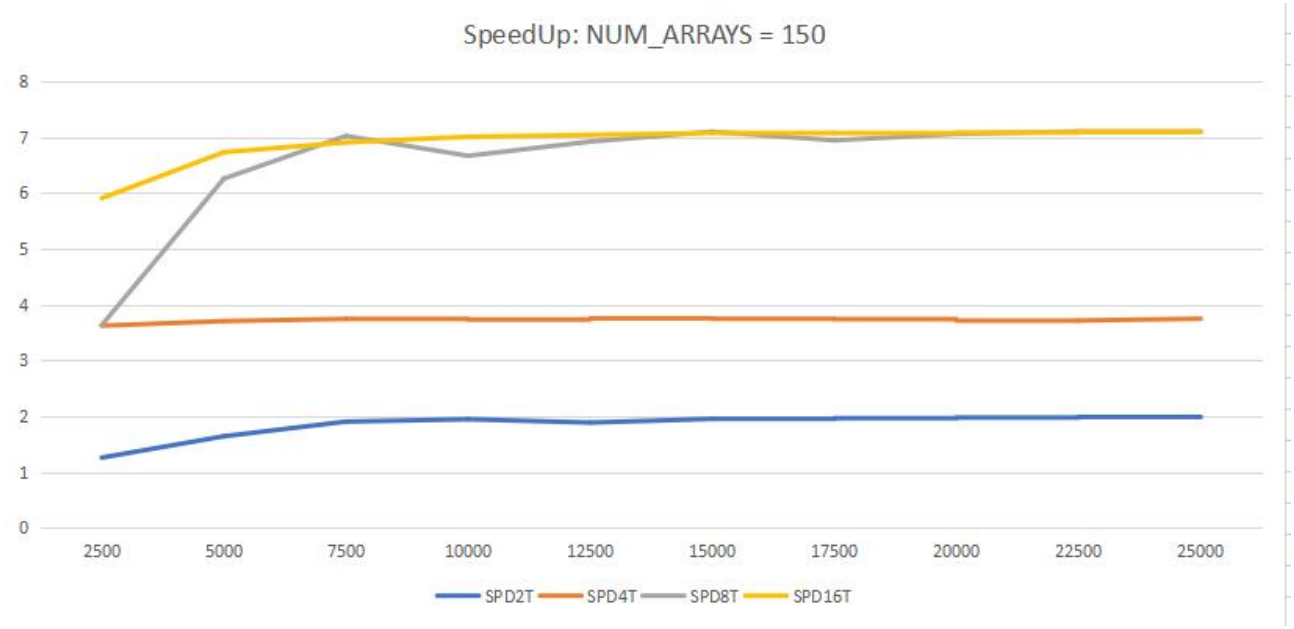

Gráfico 6 - Speedup dos tempos com 150 Vetores

Gráfico 7 - Speedup dos tempos com 500 valores

Podemos avaliar pelos gráficos que todos os testes começam por um valor inicial alto e se tornam uma contante, sendo que o de 8 Threads ele tem um aumento brusco e depois continua constante durante o processo dos algoritmos.

2.7.3 Avaliação da Eficiência

A eficiência também é um cálculo de avaliação, onde é calculado o quanto foi eficiente o conjunto de Treads para agilizar o processo do Bubble Sort, nesse caso é verificado uma média de tempo utilizado pelo número de Threads para resolver, como mostra os gráficos abaixo:

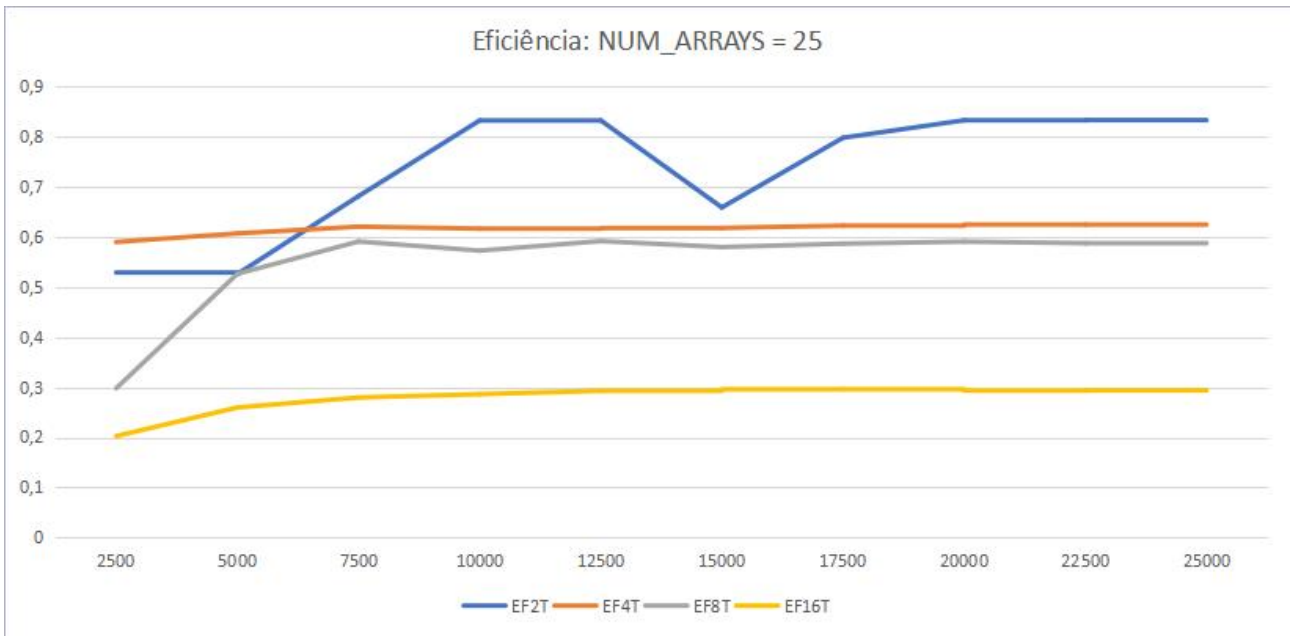

Gráfico 8 - Eficiência com 25 Vetores

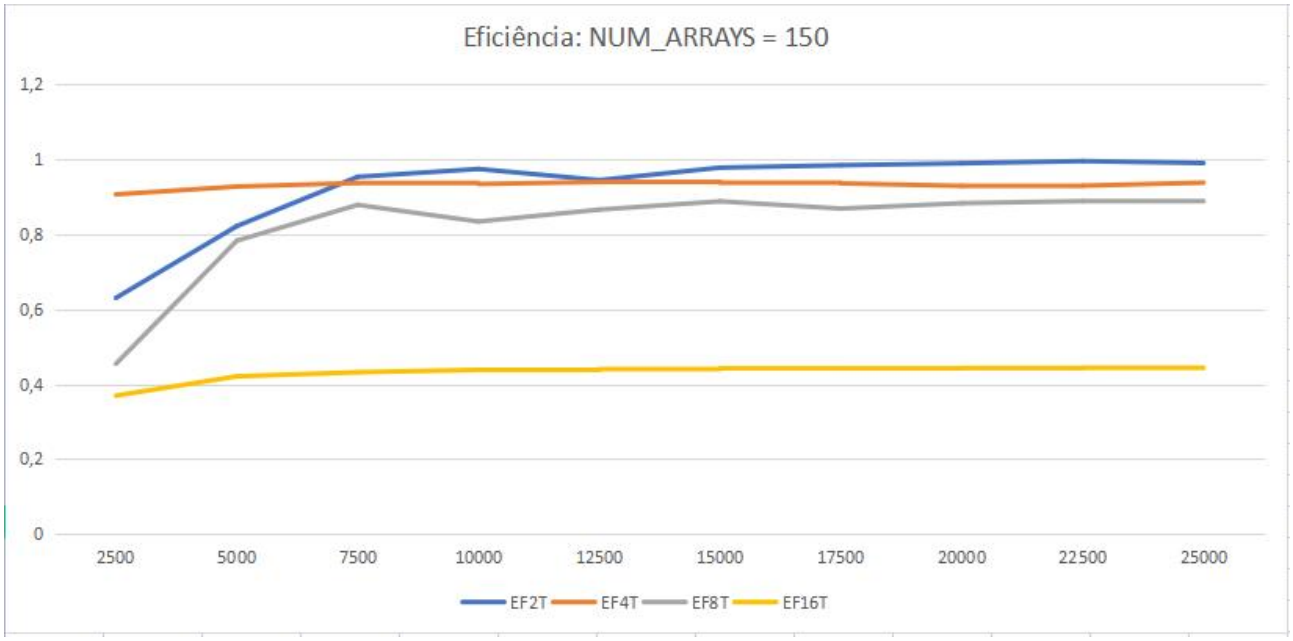

Gráfico 9 - Eficiência com 150 Vetores

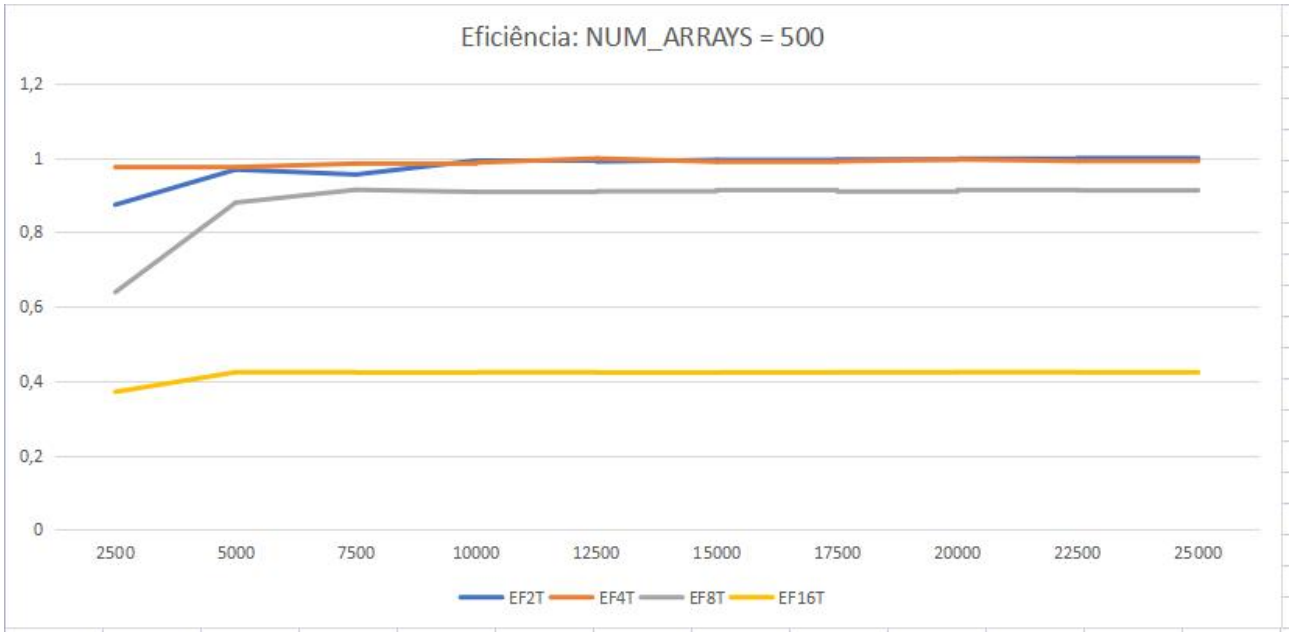

Gráfico 10 - Eficiência com 500 Vetores

Com todos os gráficos podemos ver que o de 16 Threads é o mais eficiente para o Bubble Sort em todos os testes, agilizando muito o tempo levado para finalizar a ordenação

3 CONCLUSÃO

Após visto em todo o processo dos testes tanto sequencial quanto paralelo, podemos ver que Paralelizando o programa do Bubble Sort para dividir em Threads a ordenação dos valores agilizou muito o tempo de conclusão da ordenação, e quanto mais Threads tiver melhor para diminuir o tempo.

Uma informação encontrada também na avaliação dos dados foi que com 2 Threads teve uma melhora perto do Sequencial, mas continuou levando um tempo além do desejado para uma performance.

Concluindo, paralelizando o algoritmo e tendo disponível 16 Threads para o processo, temos uma performance muito boa do Bubble Sort.

Foi muito interessante utilizar o OpenMP para melhorar o desempenho de programas, paralelizando o processo que mais consome tempo e performance dentro do código, isso vai ser útil de poder entender como ele pode auxiliar para diminuir o tempo de estruturas de repetição e recursividades em códigos C entre outros.

O acesso ao GRAD é temporário por semestre da cadeira de Programação Paralela, portanto não pode ser acessado para uso experimental, mas o acesso ao SPARTA é liberado para todos os alunos da Faculdade PUCRS na Escola Politécnica.

REFERÊNCIAS

Lei de Amdahl; Wikipedia, Disponível em: https://pt.wikipedia.org/wiki/Lei_de_Amdahl

Teodorowisch, Roland: Conteúdos passados em aula; PUCRS, Porto Alegre - Rio Grande do Sul, 2022