TPP2: Divisão e Conquista.

1. Introdução:

Este relatório descreve a Implementação de um algoritmo paralelo de divisão e conquista para ordenar um vetor, usando o algoritmo Bubble Sort e qsort. Este relatório irá apresentar explicar o código, além de apresentar e analisar os resultados.

2. Algoritmo:

O algoritmo começa com a inicialização do MPI, após isso será verificado e é o processo raiz da divisão em conquista, no caso do processo 0 começará a contar o tempo para a execução e após isso é gerado o vetor. Os demais processos ficam travados até o recebimento de sua parte do vetor.

A parte de divisão e conquista funciona de forma semelhante para todos os processos. Os processos que receberem um vetor com tamanho menor que o delta (tamanho estabelecido para começar a ordenação), irão fazer a ordenação de sua parte e enviar para o pai. Os demais processos enviam seus vetor dividido para seus filhos, e ficam travados até receberem os vetores ordenados, após isso ocorre a intercalação.

A última etapa trata do envio de vetores para o pai, quando o processo 0 recebe os vetores de seus dois filhos, o tempo é contabilizado e o algoritmo é encerrado.

O algoritmo de divisão e conquista funciona e forma semelhante para os dois algoritmos de ordenação, quanto a sua divisão. A diferença ocorre quando o tamanho do vetor é menor que o delta, quando é chamada a ordenação

3. BS:

Com o objetivo de testar o *Bubble Sort,* foram testado casos utilizando de 1 até 1023 processos, ou seja, tivemos árvores que tinha em seu último nível as seguintes quantidades de processos: 1, 2, 4, 8 16, 32, 64, 128, 256, 512. O *speed-up* resultante é uma curva crescente que se manteve acima do *speed-up* ideal, o que torna muito vantajoso utilizar a técnica de divisão em conquista junto *Bubble Sort*, pois compensa utilizar vários processos para resolver vários pedaços de uma tarefa pesada, ou seja, vale a pena fazer esta divisão.

A eficiência se apresentou de forma crescente até a utilização de 63 processos(32 processos com vetores menor que delta), todos os testes até este apresentavam uma grande vantagem no tempo ao acrescentar mais processos,

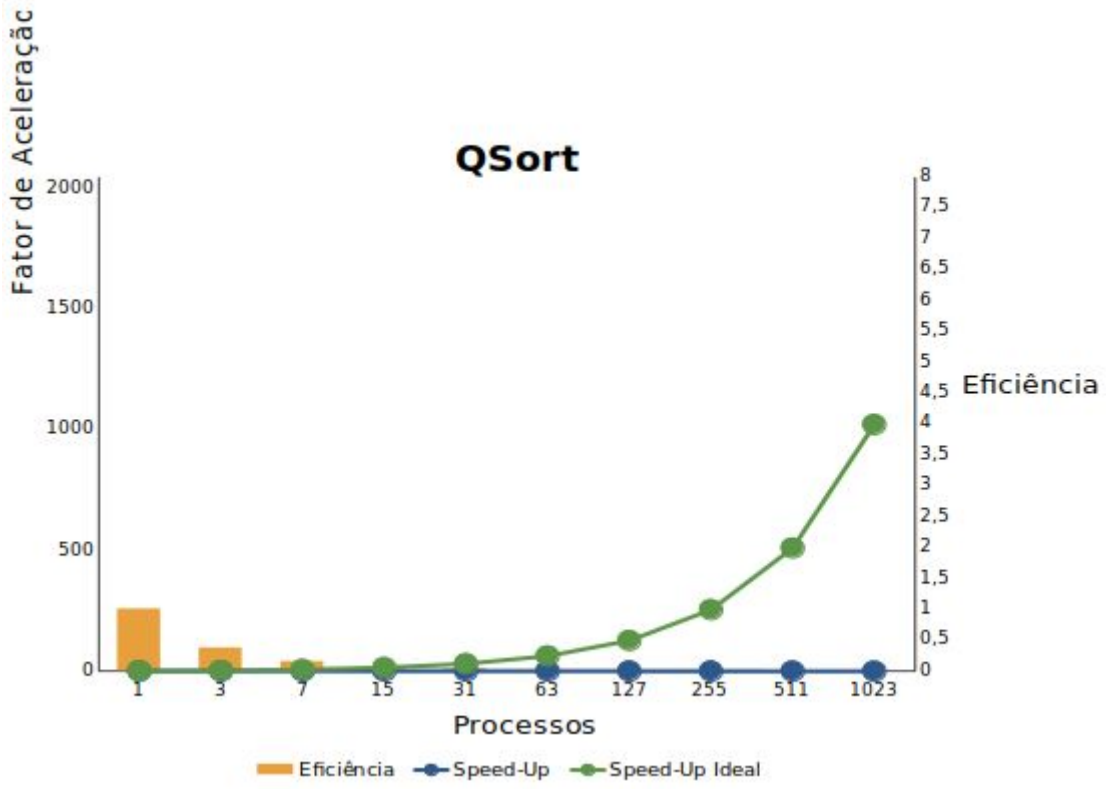
após isso, a eficiência apresentou quedas para uma quantidade maior de processos. Esta queda está muito ligada ao tamanho do vetor, caso o vetor fosse maior a eficîencia ainda seria maior com mais processos. Com isso, podemos observar que com 127 processos o custo de dividir já começa a apresentar um custo considerável, para o tamanho de vetor trabalhado, e acaba que não ganhamos tanto ao acrescentar mais processos.

Processos	Tempo	Speed-Up	Speed-Up Ideal	Eficiência
1	**4440**	**1**	**1**	**1**
3	**1080**	**4,11**	**3**	**1,37**
7	**409**	**10,86**	**7**	**1,55**
15	**71**	**62,54**	**15**	**4,17**
31	**21**	**211,43**	**31**	**6,82**
63	**9**	**493,33**	**63**	**7,83**
127	**4,9**	**906,12**	**127**	**7,13**
255	**2,54**	**1748,03**	**255**	**6,86**
511	**1,51**	**2940,4**	**511**	**5,75**
1023	**1,09**	**4073,39**	**1023**	**3,98**

4. QSort:

Com o objetivo de testar o *QSort,* foram testados os mesmos casos do *Bubble Sort* . O *speed-up* resultante é fica muito abaixo do *speed-up* ideal, e a eficiência se mostra de forma crescente desde o início. Isto ocorre pelo fato do

QSort ser um algoritmo muito rápido, então ocorre que a divisão acaba sendo mais pesada que o algoritmo.



Podemos ver na tabela abaixo que ao utilizar mais de 7 processos, o tempo começa a aumentar, ou seja, o custo de divisão acaba sendo maior que o custo do *QSort*. Isso acaba resultando que o Speed-Up seja menor que o ideal, e a eficiência do algoritmo caia ao acrescentar mais processos.

Processos	Tempo	Speed-Up	Speed-Up Ideal	Eficiência
1	**0,08**	**1**	**1**	**1**
3	**0,07**	**1,11**	**3**	**0,37**
7	**0,07**	**1,06**	**7**	**0,15**
15	**0,08**	**0,98**	**15**	**0,07**
31	**0,09**	**0,85**	**31**	**0,03**
63	**0,1**	**0,77**	**63**	**0,01**
127	**0,15**	**0,5**	**127**	**0**
255	**0,23**	**0,33**	**255**	**0**
511	**0,37**	**0,21**	**511**	**0**

1023	**0,91**	**0,08**	**1023**	**0**

Código

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

#define VETOR_SIZE 100000

void bs(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++;
            }
        }

        int *interleaving(int vetor[], int tam){
          int *vetor_auxiliar;
          int i1, i2, i_aux;

          vetor_auxiliar = (int *)malloc(sizeof(int) * tam);

          i1 = 0;
          i2 = tam / 2;

          for (i_aux = 0; i_aux < tam; i_aux++){
            if (((vetor[i1] <= vetor[i2]) && (i1 < (tam / 2)))
                || (i2 == tam))
              vetor_auxiliar[i_aux] = vetor[i1++];
            else
              vetor_auxiliar[i_aux] = vetor[i2++];
          }

          return vetor_auxiliar;
        }

        int main(int argc, char** argv){
          MPI_Status status;
          int my_rank;       // Identificador deste processo
          int proc_n;       // Numero de processos disparados pelo usuario na linha de
comando (np)
          int mid;
          int *vetor_aux;
          int tam_vetor = VETOR_SIZE;
          int nivel,pai;
          int delta = 25000;
          double t1,t2;

          MPI_Init(&argc , &argv); // funcao que inicializa o MPI, todo o codigo paralelo
estah abaixo

          MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // pega pega o numero do
processo atual (rank)
          MPI_Comm_size(MPI_COMM_WORLD, &proc_n);  // pega informacao do
numero de processos (quantidade total)
```

```
        int *vetor = (int *)malloc(sizeof(int)*tam_vetor);
        // recebo vetor

        if ( my_rank != 0 ){
          MPI_Recv(vetor, VETOR_SIZE, MPI_INT, MPI_ANY_SOURCE, 0,
MPI_COMM_WORLD,&status);
          pai = status.MPI_SOURCE; //conferir
          MPI_Get_count(&status, MPI_INT, &tam_vetor);
        }
        else{
            t1 = MPI_Wtime();
          int i = 0;
          for(; i < tam_vetor; i++){// sou a raiz e portanto gero o vetor - ordem reversa
             vetor[i] = tam_vetor - i;
          }
          mid = tam_vetor / 2;
        }
        // dividir ou conquistar?

        if ( tam_vetor <= delta ){
          printf("%d\n", my_rank);
       bs(tam_vetor,vetor);
          vetor_aux = vetor;
        }  // conquisto
        else{
          // dividir
          // quebrar em duas partes e mandar para os filhos
          printf("%d\n", my_rank);
          MPI_Send(&vetor[0], tam_vetor/2, MPI_INT, (2*my_rank)+1, 0,
MPI_COMM_WORLD);
          MPI_Send(&vetor[tam_vetor/2], tam_vetor/2, MPI_INT, (2*my_rank)+2,
0,MPI_COMM_WORLD);
          // receber dos filhos

          MPI_Recv(&vetor[0], tam_vetor/2, MPI_INT, (2*my_rank)+1, 0,
MPI_COMM_WORLD,
           &status);
          MPI_Recv(&vetor[tam_vetor/2], tam_vetor/2, MPI_INT, (2*my_rank)+2, 0,
MPI_COMM_WORLD,
           &status);

          vetor_aux = interleaving(vetor, tam_vetor);// intercalo vetor inteiro
        }

        // mando para o pai
```

```
  if ( my_rank !=0 ){
    MPI_Send(vetor_aux, tam_vetor, MPI_INT, pai, 0,
       MPI_COMM_WORLD);
  }
  else{
      t2 = MPI_Wtime(); // termina a contagem do tempo
      printf("\nTempo de execucao: %f\n\n", t2-t1);
  }

  MPI_Finalize();
  free(vetor_aux);
  return 1;
}
```