**Relatório de Entrega de Trabalho**
**Disciplina de Programação Paralela (PP) - Prof. César De Rose**
**Alunos:** Rafael Rios e Rodrigo Silveira
**Exercício:** TPP2: MPI Divisão e Conquista (D&C)

# 1 Implementação

O objetivo deste trabalho foi implementar um algoritmo de divisão e conquista para compreender aplicações paralelas, usando a biblioteca MPI, o problema proposto consiste em ordenar um vetor de 1000000 elementos utilizando os algoritmos bubblesort e quicksort. No inicio, o processo 0 cria e inicializa o vetor enquanto os outros aguardam o recebimento do mesmo junto com a mensagem do tamanho do vetor recebido (MPI_Get_count). O processamento do que é feito com o vetor é realizado de modo igual por todos os processos no código: caso o tamanho do vetor recebido seja menor ou igual ao delta pré estabelecido, o vetor é ordenado, caso contrário, o vetor é separado e enviado para os filhos, que são escolhidos pelas fórmulas 2*(rank do processo)+1 e 2*(rank do processo)+2, que mapeiam uma árvore binária a um vetor, garantindo que cada filho só terá um pai e que cada pai terá no máximo 2 filhos. Outra implementação foi o controle para tamanhos de vetor não divisíveis por dois (Ex: 15625), quando isso ocorre, o filho da esquerda recebe a metade do vetor truncada (Ex: 0-7812) e o da direita recebe o restante a partir do índice seguinte (Ex: 7813-15624), esse controle é feito comparando a metade do tamanho do vetor, com esse tamanho somado de 0.5 e depois truncado, caso esses valores forem idênticos, o vetor é divisível por dois, caso contrário deve ser dividido como explicado anteriormente. O mesmo código foi usado para a execução do algoritmo ordenado localmente pelos pais. Antes de demonstrar o seu funcionamento, deve-se explicar duas variáveis: "tam_local"é o tamanho do vetor que será executado localmente pelo pai e "tam_divide"é o tamanho a ser dividido igualmente pelos filhos. O controle é feito através de um valor definido (LOCAL), se for local, "tam_local"recebe delta e "tam_divide"recebe o tamanho do vetor subtraído por delta, se não for local, "tam_local"recebe 0 e "tam_divide"recebe o tamanho inteiro do vetor a ser dividido. Após o envio do vetor para os filhos, se for o algoritmo com execução local, ele ordena a sua parte do vetor, depois, o pai recebe o vetor ordenado dos filhos (levando em conta se o tamanho enviado era divisível por dois) e organiza-o de modo a ficar ordenado, mostrando o tempo de execução no final, pelo processo 0.

# 2 Dificuldades encontradas

Embora a implementação do algoritmo tenha sido relativamente simples e sem maiores complicações, tivemos dificuldades em alguns pequenos detalhes, como a elaboração de um método para garantir a ordem certa de envio dos vetores para os filhos, escolhendo quais processos receberiam o vetor e a resolução do problema para quando o tamanho do vetor a ser particionado não seria divisível por dois.

# 3 Testes

O código foi executado para três casos com o intuito de mostrar as diferenças nos tempos de ordenamento. No primeiro teste, o pai divide o vetor pela metade para os filhos até que se receba um vetor com tamanho menor ou igual a um delta pré estabelecido conforme o número de processos (caso A), sendo executado até 31 nodos (HT). Em um segundo momento, o algoritmo foi testado para um número de processos maiores que o HT (caso C) e, por último, realizaram-se alterações no código para o pai ordenar uma parte do vetor e dividir o resto para os filhos (caso B), com a finalidade de melhorar ainda mais o tempo de execução. Foram realizados testes com alocação de 2 nodos na máquina grad, executando o algoritmo de divisão e conquista para 7, 15, 31, 63, 127 e 255 processos. O valor de delta para o caso B foi escolhido dividindo o tamanho do vetor pelo número de processos, de modo a balancear a ordenação dos vetores, já que cada processo fará a sua ordenação. Para valores de delta em que foi usada uma porcentagem fixa, obtivemos resultados piores, visto que a ordenação tornava-se desbalanceada entre os processos. O delta para o caso A e C é dividido pela metade conforme aumenta-se o número de processos.

# 4 Análise de desempenho

Para o algoritmo quicksort, não houve uma mudança significativa nos tempos medidos, já que o algoritmo apresenta uma complexidade muito baixa. Entretanto, para o algoritmo bubblesort, notou-se que conforme aumentava-se o número de processos, o tempo de execução praticamente diminuía pela metade para todos os casos, mesmo passando do ponto de HT (acima de 31 processos). O caso B com ordenação local obteve o melhor tempo de execução, visto que ao diminuir o tamanho do vetor a ser ordenado houve um menor custo de processamento na função bubblesort, agilizando o processo. Curiosamente, houve uma grande melhora no tempo de execução após o HT, mesmo havendo um número de processos maior do que o número de núcleos disponíveis. Acreditamos que isso se deve ao tempo ocioso dos processos pais enquanto aguardam a resposta dos filhos e, no caso B, o grau de processamento local pode não estar otimizado, fazendo com que o processo pai termine de processar muito antes de seus filhos. O speed-up melhorou com o aumento do número de processos para todos os casos, como mostra a Figura 1, e a eficiência estabilizou por volta de 4.0 após 15 processos para os casos A e C e em 13.0 para o caso B, também após 15 processos, com o tempo de execução diminuindo.

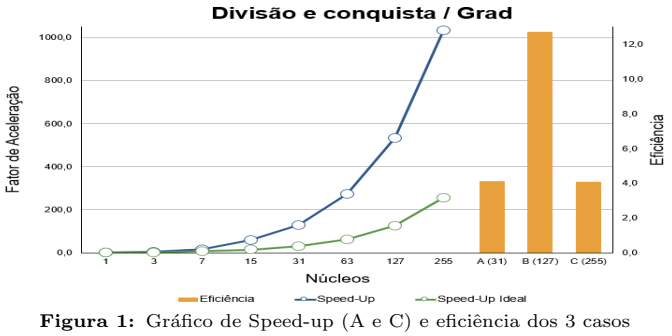


**Figura 1:** Gráfico de Speed-up (A e C) e eficiência dos 3 casos

# 5 Observações Finais

Apesar de o código ser extremamente compacto e simples, ele se mostrou muito eficiente, melhorando muito o tempo de execução do problema e apresentando um desempenho melhor que o mestre/escravo, já que não possui o gargalo de comunicação com o mestre. Notou-se a melhora no desempenho ao realizar a ordenação localmente, de modo a não deixar os processos pais ociosos e diminuindo o tamanho do vetor final a ser ordenado.

```c
#include <stdio.h>
#include "mpi.h"
#include <stdlib.h>
#include <string.h>

void print_line(int c, int* line){
        int i;
        for(i = 0 ; i < c ; i++)
                printf("%d ", line[i]);
        printf("\n");
}

void bubble_sort (int *a, int n) {
    int i, t, j = n, s = 1;
    while (s) {
        s = 0;
        for (i = 1; i < j; i++) {
            if (a[i] < a[i - 1]) {
                t = a[i];
                a[i] = a[i - 1];
                a[i - 1] = t;
                s = 1;
            }
        }
        j--;
    }
}

int *interleaving3(int vetor[], int tam, int offset1, int offset2, int offset3){
  int *vetor_auxiliar;
  int i1, i2, i3, i_aux;
  vetor_auxiliar = (int *)malloc(sizeof(int) * tam);
  i1 = offset1;
  i2 = offset2;
  i3 = offset3;
  for (i_aux = 0; i_aux < tam; i_aux++) {
    if (((vetor[i1] <= vetor[i2]) && (i1 < i2) && (vetor[i1] <= vetor[i3])) || ((i2 == offset3) && (i3 ==
    tam))){
      vetor_auxiliar[i_aux] = vetor[i1++];
    }else if (((vetor[i2] <= vetor[i1]) && (i2 < i3) && (vetor[i2] <= vetor[i3])) || ((i1 == offset2) && (
    i3 == tam))){
      vetor_auxiliar[i_aux] = vetor[i2++];
    }else
      if (i3 != tam){
        vetor_auxiliar[i_aux] = vetor[i3++];
      }else{
        if (((vetor[i1] <= vetor[i2]) && (i1 < i2)))
          vetor_auxiliar[i_aux] = vetor[i1++];
        else
          vetor_auxiliar[i_aux] = vetor[i2++];
      }
  }
  return vetor_auxiliar;
}

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 cmpfunc (const void * a, const void * b) {
   return ( *(int*)a - *(int*)b );
}

#define VETOR_SIZE 1000000
#define delta 70000
#define LOCAL 1

/*              7       15      31      63      127     255
normal          250000  125000  62500   31250   15625   8000
local           150000  70000   34000   16000   8000    4000
*/

void main(int argc, char** argv){
  int i, tam_vetor, tam_local, tam_divide;
  int my_rank, rank_pai;       // Identificador deste processo
  int proc_n;         // Numero de processos disparados pelo usuario na linha de comando (np)
  int message;        // Buffer para as mensagens
  MPI_Status status; // estrutura que guarda o estado de retorno
  int (*vetor) = malloc(VETOR_SIZE * sizeof *vetor);
  double t1, t2;

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

  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // pega o numero do processo atual (rank)
  MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

  if ( my_rank != 0 ){
        MPI_Recv(vetor, VETOR_SIZE, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);  //
    nao sou a raiz, tenho pai
        rank_pai = status.MPI_SOURCE;
        MPI_Get_count(&status, MPI_INT, &tam_vetor);  // descubro tamanho da mensagem recebida
  }
  else{
        t1 = MPI_Wtime();
        tam_vetor = VETOR_SIZE;              // defino tamanho inicial do vetor
        for(i = 0 ; i < tam_vetor ; i++)      // sou a raiz e portanto gero o vetor - ordem reversa
                vetor[i] = VETOR_SIZE - i;
  }
  if( tam_vetor <= delta ){
        //qsort(vetor, tam_vetor, sizeof(int), cmpfunc);
        bubble_sort(vetor, tam_vetor);
  }
   else{
        float tam_filhos;
        if(LOCAL){      //Se for local, ordena da posicao 0 ate o delta, e divide o resto igualmente entre
     os filhos
                tam_local = delta;
                tam_divide = tam_vetor-delta;
        }else{          //Se nao for local, divide todo o vetor igualmente entre os filhos
                tam_local = 0;
                tam_divide = tam_vetor;
        }
        tam_filhos = tam_divide/2.0;
        if(tam_filhos != (int)(tam_filhos+0.5)){    //se for um tamanho de vetor com casa decimal, manda
    tam_vetor/2 e tam_vetor/2+1
                MPI_Send(&vetor[tam_local], (int)(tam_divide/2), MPI_INT, 2*my_rank+1, 0, MPI_COMM_WORLD);
      // mando metade inicial do vetor
                MPI_Send(&vetor[tam_local + (int)(tam_divide/2)], (int)(tam_divide/2+1), MPI_INT, 2*
    my_rank+2, 0, MPI_COMM_WORLD);  // mando metade final
        }
        else{             //se nao for um tamanho de vetor com casa decimal
                MPI_Send(&vetor[tam_local], tam_divide/2, MPI_INT, 2*my_rank+1, 0, MPI_COMM_WORLD);  //
    mando metade inicial do vetor
                MPI_Send(&vetor[tam_local + tam_divide/2], tam_divide/2, MPI_INT, 2*my_rank+2, 0,
    MPI_COMM_WORLD);  // mando metade final
        }
        if(LOCAL){        //se for execucao local, ordeno a minha parte do vetor
                //qsort(vetor, tam_local, sizeof(int), cmpfunc);
                bubble_sort(vetor, tam_local);
        }
        if(tam_filhos != (int)(tam_filhos+0.5)){    //se for um tamanho de vetor com casa decimal, manda
    tam_vetor/2 e tam_vetor/2+1
                MPI_Recv(&vetor[tam_local], (int)(tam_divide/2), MPI_INT, 2*my_rank+1, MPI_ANY_TAG,
    MPI_COMM_WORLD, &status);   //recebo metade inicial
                MPI_Recv(&vetor[tam_local + (int)(tam_divide/2)], (int)(tam_divide/2+1), MPI_INT, 2*
```

```
    my_rank+2, MPI_ANY_TAG, MPI_COMM_WORLD, &status); //recebo metade final
142         }
143         else{             //se nao for um tamanho de vetor com casa decimal
144                 MPI_Recv(&vetor[tam_local], tam_divide/2, MPI_INT, 2*my_rank+1, MPI_ANY_TAG,
    MPI_COMM_WORLD, &status);  //recebo metade inicial
145                 MPI_Recv(&vetor[tam_local + tam_divide/2], tam_divide/2, MPI_INT, 2*my_rank+2, MPI_ANY_TAG
    , MPI_COMM_WORLD, &status); //recebo metade final
146         }
147         if(LOCAL) vetor = interleaving3(&vetor[0], tam_vetor, 0, tam_local, tam_local + tam_divide/2); //
    ordena para local
148         else vetor = interleaving(vetor, tam_vetor);    //ordena para 2 vetores
149   }
150   if ( my_rank !=0 ){
151         MPI_Send(vetor, tam_vetor, MPI_INT, rank_pai, 0, MPI_COMM_WORLD);  // tenho pai, retorno vetor
    ordenado pra ele
152   }
153   else  {
154         t2 = MPI_Wtime();
155         printf("Tempo de execucao: %f\n", t2-t1);
156         //print_line(VETOR_SIZE, vetor);
157   }
158   MPI_Finalize();
159   free(vetor);
160 }
```