EXERCÍCIO 1 – MUDANDO AS CORES

O código abaixo ilustra o uso da comunicação ponto-a-ponto e a comunicação coletiva. Nesse código, uma matriz de três cores (verde, branco, vermelho e) é distribuída para todos os processos criados no início do programa MPI. Cada processo de muda de cor e envia a sua cor alterada atributo para o processo mestre (o), que mostra na tela a cor original e a cor alterada de cada processo.

Depois de ter definido todas as variáveis necessárias que serão usadas no programa, é especificado qual será o processo de *root*. Esse será usado para transmitir, por *broadcast*, as cores para todos os outros processos e para imprimir as cores de cada um dos processos. Processo o foi escolhido como o processo de raiz.

Em seguida, foram geradas as três cores distintas dinamicamente que serão utilizadas nos processos, uma matriz dinâmica *vetorCores* é criada para armazenar as cores: branco, vermelho e verde.

```
N_cores = 3;
colorArray = (int*) malloc(sizeof(int) * n_cores);
for (k = 0; k < n_cores; k++)
    vetorCores[k] = k;
```

É necessário inicializar o MPI, acessando o *rank* e total de processos que estão sendo executados.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &procID);
MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

Então é enviado por broadcast todo o vetor de cores para cada processo utilizando o MPI_BCAST e cada um dos processores irá definir sua cor como verde (ultimo elemento do vetor.

```
MPI_Bcast(vetorCores, n_cores, MPI_INT, root, MPI_COMM_WORLD);
pcolor = vetorCores[2];
```

O *rank* do processador que chama o *MPI_Bcast* não foi especificado, pois todos os processadores no comunicador *MPI_COMM_WORLD* comunicador deve realizar esta chamada. Neste caso, é o processo *root* que envia o *vetorCores*, para cada um dos processos no comunicador.

Além disso, não existe nenhuma parâmetro *TAG* incluído na função *MPI_Bcast*. Esse parâmetro não é necessário porque nesta operação de comunicação coletiva, o número de elementos e o tipo de dados destes elementos são idênticas em todos os processos.

Em seguida, todos os outros processos que não sejam *root* enviam sua cor de volta para o processo *root*, que então imprime a cor de cada um dos processos *nproc*. São usadas as diretivas MPI_Send e MPI_RECV para a comunicação entre os processos.

Em seguida, é mudada a cor de cada processo, dependendo se o numero do processo é par ou ímpar (branco ou vermelho) e então a cor alterada é enviada para o *root*, para que esse imprima na tela novamente.

Exemplo de saída na tela:

```
processo 0 possui a cor verde
processo 1 possui a cor verde
processo 2 possui a cor verde
processo 3 possui a cor verde
processo 0 possui a cor branca
processo 1 possui a cor vermelha
processo 2 possui a cor branca
processo 3 possui a cor vermelha
```

Código Fonte:

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

main (int argc, char* argv[]){
     int procID, nproc, root, origem, target, tag;
     int k, n_cores, pcolor;
     int *vetorCores;
     char cores[10];
     MPI_Status status;
     root = 0; // O root será o processo zero
     n_cores = 3;
     vetor_Cores = (int*) malloc(sizeof(int) * n_cores);
     for (k = 0; k < n_cores; k++)
           vetorCores[k] = k;

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &procID);
     MPI_Comm_size(MPI_COMM_WORLD, &nproc);

     // Broadcast the array of colors to all processes
     MPI_Bcast(vetorCores, n_cores, MPI_INT, root, MPI_COMM_WORLD);

     pcolor = colorArray[2]; // Seleciona a cor 2 (verde) para todos os processos

     tag = pcolor;
     target = 0;

     if (procID != root){
          MPI_Send(&pcolor, 1, MPI_INT, target, tag, MPI_COMM_WORLD);
     }
     else{
          for (origem = 0; origem < nproc; origem++){
               if (origem != 0){
                  MPI_Recv(&pcolor, 1, MPI_INT, origem, tag, MPI_COMM_WORLD, &status);
               }
               switch(pcolor){
                 case 0: sprintf(cores, "Branca");
                         break;
                 case 1: sprintf(cores, "Vermelha");
                         break;
                 case 2: sprintf(cores, "Verde");
                         break;
                 default: printf("Cor Invalida \n");
               }
               printf("Processador %d possui a cor %s\n", origem, cores);
          }
          printf("\n\n");
     }
     pcolor = procID%2;
     if (pcolor == 0){
          sprintf(cores, "Branca");
     }
     else if (pcolor == 1){
          sprintf(cores, "Vermelha");
     }
     else if (pcolor == 2){
          sprintf(cores, "Verde");
     }

     // Access new process colors
     if (procID != root){
          MPI_Send(cores, 10, MPI_CHAR, root, tag, MPI_COMM_WORLD);
     }
     else{
          printf("Processador %d possui a cor %s\n", root, cores);
          for (origem = 1; origem < nproc; origem++){
               MPI_Recv(cores, 10, MPI_CHAR, origem, tag, MPI_COMM_WORLD, &status);
               printf("proc %d has color %s\n", origem, color);
          }
     }
     free(colorArray);
     MPI_Finalize();
}
```

Exercício 2 - Confirmar envio e recebimento

```
#include <stdio.h>
#include <mpi.h>
#include <string.h>
       int main(int argc, char* argv[]) {
         int rank, tam, tag=99;
         char msg[20];
         MPI_Status, status;
         MPI_Init(&argc, &argv);
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
         MPI_Comm_size(MPI_COMM_WORLD, &tam);
         if (rank == 0) {
             strcpy( msg, "Olá Mundo!\n");
             MPI_Send( msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
             // Envia uma mensagem para processo "1"
             }
         else  {
             strcpy( msg, "Olá turma!\n" );
             printf(msg);
             MPI_Recv( msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
             // Recebe uma mensagem provinda do processo "0"
             printf(msg);
             }
         MPI_Finalize();
         return 0;
}
```

Exercício 3 - Mais um de Broadcast

```
#include <stdio.h>
#include <mpi.h>
#include <string.h>
       int main(int argc, char* argv[]) {
         int rank, tam, tag=99;
         char msg[20];
         MPI_Status, status;
         MPI_Init(&argc, &argv);
         MPI_Comm_rank(MPI_COMM_WORLD, &rank);
         MPI_Comm_size(MPI_COMM_WORLD, &tam);
         if (rank == 0) {
              strcpy( msg, "Olá Mundo!\n");
              MPI_Bcast( msg, 20, MPI_CHAR, 0, MPI_COMM_WORLD );
              // Executa Broadcast com conteúdo no processo "0"
              }
         else  {
              MPI_Bcast( msg, 20, MPI_CHAR, 0, MPI_COMM_WORLD );
              // Executa Broadcast com conteúdo no processo "0"
              printf("Processo %d msg = ", rank );
              printf(msg);
              }
         MPI_Finalize();
         return 0;
}
```

Exercício 4 - Distribuindo os dados utilizando Scatter

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char* argv[]) {
       int rank, tam, tag=99, origem=0;
       float matsend[4][4] ={{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11,
       12},{13, 14, 15, 16}};
       float matrecv[4];
       MPI_Status, status;
       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       MPI_Comm_size(MPI_COMM_WORLD, &tam);
       if (tam == 4) {
              MPI_Scatter( matsend, 4, MPI_FLOAT, matrecv, 4, MPI_FLOAT,
                           origem, MPI_COMM_WORLD );
              // Executa Scatter com conteúdo no processo origem="0"
              printf("rank= %d Resultado: %f %f %f %f\n", rank, matrecv[0],
                      matrecv[1], matrecv[2], matrecv[3]);
              }
        else {
              if (rank==0)
                 printf("Deve ser executado com número de processos
                       igual a 4.  Finalizado sem sucesso!\n");
        }
       MPI_Finalize();
       return 0;
}
```

Exercício 5 - Distribuindo os dados utilizando Gather

```
#include <stdio.h>
#include <mpi.h>
       int main(int argc, char* argv[]) {
       int rank, tam, tag=99, origem=0, i;
       float matsend[4][4]; float matrecv[4];
       MPI_Status, status;
       MPI_Init(&argc, &argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
       MPI_Comm_size(MPI_COMM_WORLD, &tam);
       if (tam == 4) {
              for(i=0; i<4; i++)
                     matrecv[i] = (float)rank*4 + i + 1;
              MPI_Gather(matrecv, 4, MPI_FLOAT, matsend, 4, MPI_FLOAT,
                          origem, MPI_COMM_WORLD );
              // Executa Gather com conteúdo de todos os processos
              if( rank == origem )
                     for(i=0; i<4; i++)
                            printf("Resultado: %f %f %f %f\n", matsend[i][0],
              matsend[i][1], matsend[i][2], matsend[i][3]);
              }
       else {
              if (rank==0)
                 printf("Deve ser executado com número de processos
                       igual a 4.  Finalizado sem sucesso!\n");
              }
       MPI_Finalize();
       return 0;
}
```

Exercício 6 - Utilizando o Reduce

```
#include <stdio.h>
#include <mpi.h>
       int main(int argc, char* argv[]) {
              int rank, tam, tag=99, origem=0, i, result;
              int sendbuff, recvbuff;
              MPI_Init(&argc, &argv);
              MPI_Comm_rank(MPI_COMM_WORLD, &rank);
              MPI_Comm_size(MPI_COMM_WORLD, &tam);
              sendbuff = rank;
              MPI_Reduce( &sendbuff, &recvbuff, 1, MPI_INT, MPI_MAX, 0,
              MPI_COMM_WORLD );
              // Executa Reduce para operação paralela
              if( rank == origem )
                     printf("Maior: %d\n", recvbuff);
              MPI_Finalize();
              return 0;
       }
```

Exercício 7 - Multiplicação de Vetores e Soma todos os elementos do vetores A e B

```
#include <stdio.h>
#include <mpi.h>
       int main(int argc, char* argv[]) {

         double soma, soma_local, a[256], b[256];
         int rank, TAM, tg=99, i, n = 256, primeiro, ultimo;
         MPI_Init(&argc,&argv);
         MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
         MPI_Comm_rank(MPI_COMM_WORLD,&myid);
         primeiro = rank * n/tam;
         ultimo = (rank + 1) * n/tam;
       /*  for (i = primeiro; i < ultimo; i++) { */
         for (i = 0; i < n; i++) {
            a[i] = i * 0.5;
            b[i] = i * 2.0;
         }
         soma_local = 0;
         for (i = primeiro; i < ultimo; i++) {
               soma_local = soma_local + a[i]*b[i];
         }
         MPI_Reduce (&soma_local, &soma, 1, MPI_DOUBLE, MPI_SUM,
                        MPI_COMM_WORLD);
         if (rank==0){
            printf ("soma = %f", soma);
            for (i=0; i<n; i++){
               printf ("Vetor a[%n] = %f", i,a[i]);
               printf ("Vetor b[%n] = %f", i,b[i]);
               printf ("Vetor c[%n] = %f", i,c[i]);
            }
         MPI_Finalize();
         Return 0;
       }
```