LISTA 2

José Augusto Agripino de Oliveira

DCA0123 - Programação Concorrente e Distribuída

Questões feitas: 1, 2, 3, 4, 5, 6, 10, 11, 12 (9/19)

Questão 1 - 3.1 do livro texto: O que acontece no programa greetings se, em vez de *strlen(greeting) + 1*, usarmos *strlen(greeting)* para o comprimento da mensagem a ser enviada pelos processos 1, 2, . . , comm_sz-1? O que acontece se usarmos MAX_STRING em vez de *strlen(greeting) + 1*? Você pode explicar estes resultados?

Gera uma inconsistência, já que o último caractere de uma string deve ser nulo "\0", indicando o fim da string, por isso há strlen(greeting)+1. Quando usamos apenas strlen(greeting), há erro, pois não existe o caractere nulo, logo, o vetor não é caracterizado como uma string.

Quando usamos MAX_STRING, as strings serão enviadas normalmente, já que existe o caractere nulo. Entretanto, estamos enviando mais dados do que o necessário, perdendo a eficiência do programa.

Questão 2 - 3.6 do livro texto: Suponha *comm_sz = 4* e suponha que x é um vetor com n=14 componentes.

a. Como os componentes de x seriam distribuídos entre os processos em um programa que utilizasse uma block distribution?

A distribuição de blocos é usada quando a carga computacional é distribuída de maneira homogênea. A quantidade de elementos por processo é calculado dividindo o número total a ser paralelizado pela quantidade de processos: $14/4 = 3.5$

Processo 0: x_0, x_1, x_2, x_3

Processo 1: x_4, x_5, x_6, x_7

Processo 2: x_8, x_9, x_{10}

Processo 3: x_{11}, x_{12}, x_{13}

b. Como os componentes de x seriam distribuídos entre os processos em um programa que utilizasse uma cyclic distribution?

A distribuição cíclica é utilizada para melhorar o equilíbrio de carga computacional em uma dada estrutura de dados quando ela não possui uma carga homogênea.

Processo 0: x_0, x_4, x_8, x_{12}

Processo 1: x_1, x_5, x_9, x_{13}

Processo 2: x_2, x_6, x_{10}

Processo 3: x_3, x_7, x_{11}

c. Como os componentes de x seriam distribuídos entre os processos em um programa que utilizasse uma block-cyclic distribution com blocos de tamanho b = 2?

Já a distribuição cíclica em blocos é uma junção das duas, sendo o blocksize uma variável que influencia na forma da distribuição.

Processo 0: x_0, x_1, x_8, x_9

Processo 1: x_2, x_3, x_{10}, x_{11}

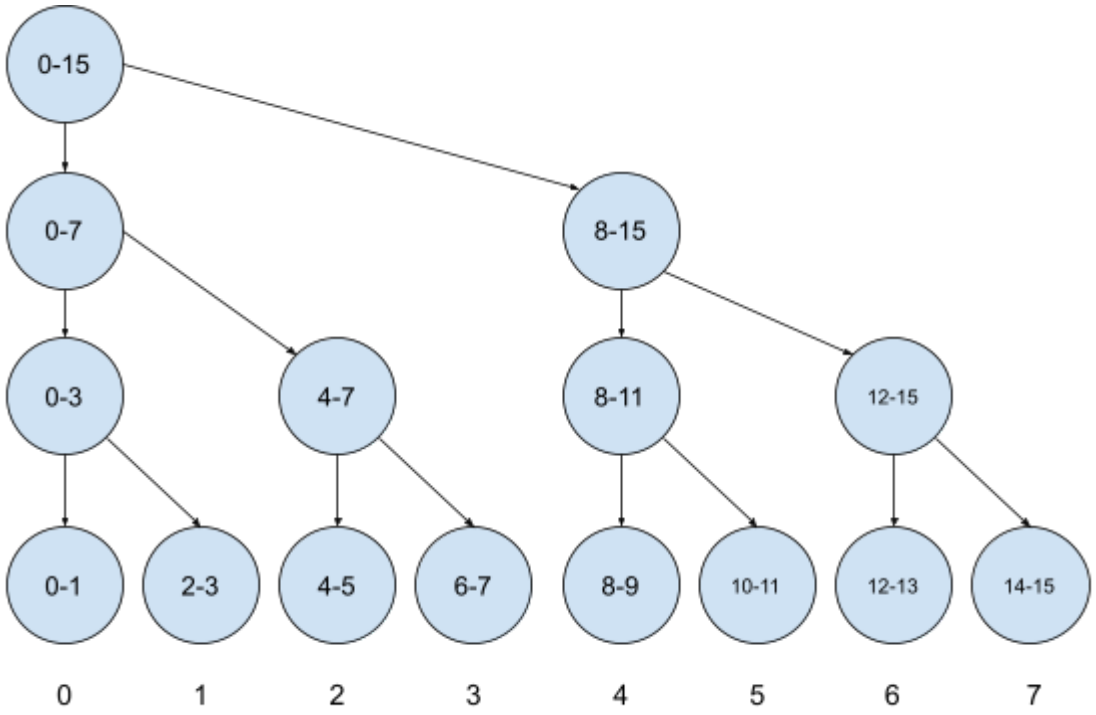
Processo 2: x_4, x_5, x_{12}, x_{13}

Processo 3: x_6, x_7

Questão 3 - 3.8 do livro texto: Suponha *comm_sz = 8* e *n = 16.*

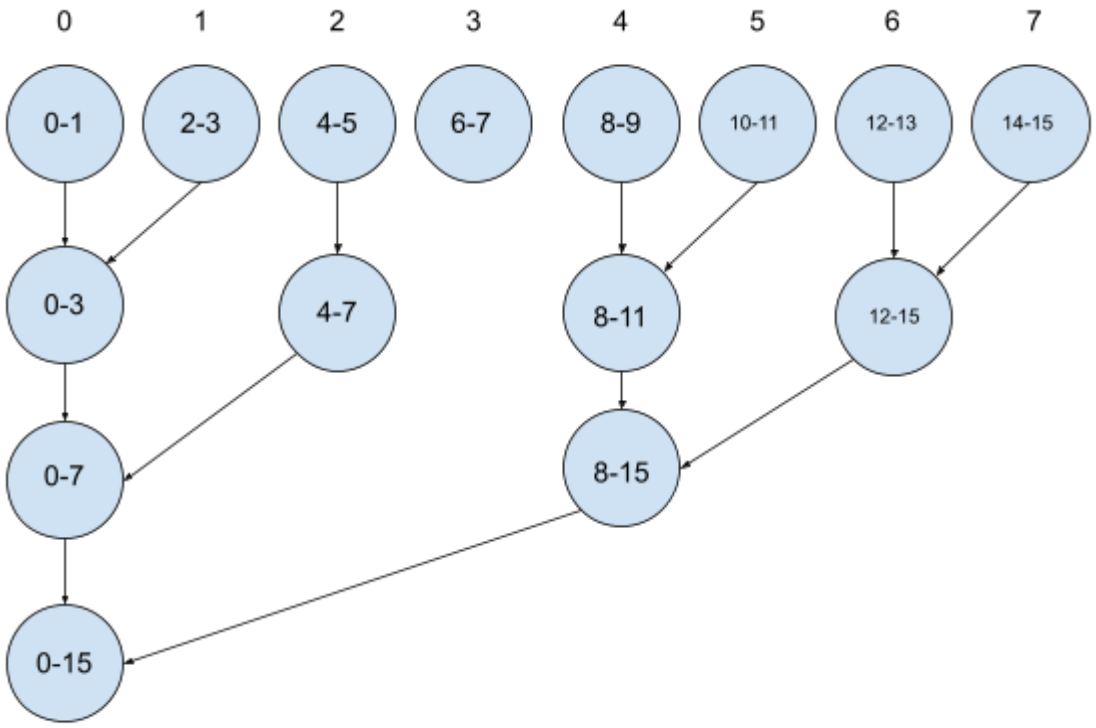
a. Desenhe um diagrama que mostre como o MPI_Scatter pode ser implementado usando a tree-structed communication com *comm_sz* processos quando o processo 0 precisa distribuir uma matriz contendo n elementos.

Como a função MPI_Scatter é utilizada para distribuir dados, temos então:



b. Desenhe um diagrama que mostre como o MPI_Gather pode ser implementado usando a tree-structed communication quando uma matriz de n-elementos que tenha sido distribuída entre processos *comm_sz* precisa ser reunida no processo 0.

A função MPI_Gather faz justamente o inverso da MPI_Scatter, que é juntar os dados distribuídos e processados por cada um dos processos em execução.



Questão 4 - 3.9 do livro texto: Escreva um programa MPI que implementa a multiplicação de vetor por escalar e o produto vetorial. O usuário deve entrar com dois vetores e um escalar, todos lidos pelo processo 0 e distribuídos entre os processos. Os resultados são calculados e coletados no processo 0, que os imprime. Pode-se supor que n, a ordem dos vetores, é igualmente divisível por comm_sz.

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

void Check_for_error(int local_ok, char fname[], char message[],
     MPI_Comm comm);
void Read_n(int* n_p, int* local_n_p, int my_rank, int comm_sz,
     MPI_Comm comm);
void Read_scalar(double* scalar, int my_rank,
     MPI_Comm comm);
void Allocate_vectors(double** local_x_pp, double** local_y_pp,
     double** local_z_pp, int local_n, MPI_Comm comm);
void Read_vector(double local_a[], int local_n, int n, char vec_name[],
     int my_rank, MPI_Comm comm);
void Print_vector(double local_b[], int local_n, int n, char title[],
     int my_rank, MPI_Comm comm);
```

```c
void Parallel_vector_dot_prod(double local_x[], double local_y[],
     MPI_Comm comm, int local_n, double* dot_product);
void Parallel_vector_mult(double scalar, double local_x[],
     double local_z[], int local_n);


/*--------------------------------------------------------------------*/
int main(void) {
  int n, local_n;
  int comm_sz, my_rank;
  double *local_x, *local_y, *local_z;
  double scalar, dot_product;
  MPI_Comm comm;

  MPI_Init(NULL, NULL);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &comm_sz);
  MPI_Comm_rank(comm, &my_rank);

  Read_n(&n, &local_n, my_rank, comm_sz, comm);
#  ifdef DEBUG
  printf("Proc %d > n = %d, local_n = %d\n", my_rank, n, local_n);
#  endif
  Allocate_vectors(&local_x, &local_y, &local_z, local_n, comm);

  Read_scalar(&scalar, my_rank, comm);
  Read_vector(local_x, local_n, n, "1", my_rank, comm);
  Print_vector(local_x, local_n, n, "Vector 1 is", my_rank, comm);
  Read_vector(local_y, local_n, n, "2", my_rank, comm);
  Print_vector(local_y, local_n, n, "Vector 2 is", my_rank, comm);

  Parallel_vector_dot_prod(local_x, local_y, comm, local_n,
&dot_product);
  if(my_rank == 0) {
   printf("O produto escalar entre os vetores 1 e 2 eh:\n%lf\n",
dot_product);
  }

  if(my_rank == 0) {
   printf("The multiplication of the vector 1 by scalar '%lf' is:",
scalar);
  }
  Parallel_vector_mult(scalar, local_x, local_z, local_n);
```

```
  Print_vector(local_z, local_n, n, "", my_rank, comm);

  if(my_rank == 0) {
   printf("The multiplication of the vector 2 by scalar '%lf' is:",
scalar);
  }
  Parallel_vector_mult(scalar, local_y, local_z, local_n);
  Print_vector(local_z, local_n, n, "", my_rank, comm);

  free(local_x);
  free(local_y);
  free(local_z);

  MPI_Finalize();

  return 0;
}  /* main */

/*--------------------------------------------------------------------*/
void Check_for_error(
     int       local_ok   /* in */,
     char      fname[]    /* in */,
     char      message[]  /* in */,
     MPI_Comm  comm       /* in */) {
  int ok;

  MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
  if (ok == 0) {
     int my_rank;
     MPI_Comm_rank(comm, &my_rank);
     if (my_rank == 0) {
        fprintf(stderr, "Proc %d > In %s, %s\n", my_rank, fname,
              message);
        fflush(stderr);
     }
     MPI_Finalize();
     exit(-1);
  }
}  /* Check_for_error */


/*--------------------------------------------------------------------*/
void Read_n(
```

```
      int*       n_p         /* out */,
      int*       local_n_p   /* out */,
      int        my_rank     /* in  */,
      int        comm_sz     /* in  */,
      MPI_Comm   comm        /* in  */) {
   int local_ok = 1;
   char *fname = "Read_n";

   if (my_rank == 0) {
      printf("What's the order of the vectors?\n");
      scanf("%d", n_p);
   }
   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
   if (*n_p <= 0 || *n_p % comm_sz != 0) local_ok = 0;
   Check_for_error(local_ok, fname,
         "n should be > 0 and evenly divisible by comm_sz", comm);
   *local_n_p = *n_p/comm_sz;
}  /* Read_n */


/*---------------------------------------------------------------------*/
void Allocate_vectors(
      double**   local_x_pp  /* out */,
      double**   local_y_pp  /* out */,
      double**   local_z_pp  /* out */,
      int        local_n     /* in  */,
      MPI_Comm   comm        /* in  */) {
   int local_ok = 1;
   char* fname = "Allocate_vectors";

   *local_x_pp = malloc(local_n*sizeof(double));
   *local_y_pp = malloc(local_n*sizeof(double));
   *local_z_pp = malloc(local_n*sizeof(double));

   if (*local_x_pp == NULL || *local_y_pp == NULL ||
       *local_z_pp == NULL) local_ok = 0;
   Check_for_error(local_ok, fname, "Can't allocate local vector(s)",
         comm);
}  /* Allocate_vectors */



/*---------------------------------------------------------------------*/
```

```c
void Read_scalar(
   double* scalar,
   int my_rank,
   MPI_Comm comm) {

       if (my_rank == 0) {
           printf("What's the scalar?\n");
           scanf("%lf", scalar);
       }
       MPI_Bcast(scalar, 1, MPI_DOUBLE, 0, comm);
}


/*--------------------------------------------------------------------*/
void Read_vector(
     double    local_a[],
     int       local_n,
     int       n,
     char      vec_name[],
     int       my_rank,
     MPI_Comm  comm) {

  double* a = NULL;
  int i;
  int local_ok = 1;
  char* fname = "Read_vector";

  if (my_rank == 0) {
     a = malloc(n*sizeof(double));
     if (a == NULL) local_ok = 0;
     Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
           comm);
     printf("Enter the vector %s\n", vec_name);
     for (i = 0; i < n; i++)
        scanf("%lf", &a[i]);
     MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
0,
        comm);
     free(a);
  } else {
     Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
```

```
            comm);
    MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
0,
        comm);
  }
}


/*--------------------------------------------------------------------*/
void Print_vector(
    double    local_b[],
    int       local_n,
    int       n,
    char      title[],
    int       my_rank,
    MPI_Comm  comm) {

  double* b = NULL;
  int i;
  int local_ok = 1;
  char* fname = "Print_vector";

  if (my_rank == 0) {
    b = malloc(n*sizeof(double));
    if (b == NULL) local_ok = 0;
    Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
          comm);
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
          0, comm);
    printf("%s\n", title);
    for (i = 0; i < n; i++)
       printf("%f ", b[i]);
    printf("\n");
    free(b);
  } else {
    Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
          comm);
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
0,
        comm);
  }
```

```
}


/*--------------------------------------------------------------------*/
void Parallel_vector_dot_prod(
     double  local_x[]  /* in  */,
     double  local_y[]  /* in  */,
     MPI_Comm comm,
     int     local_n    /* in  */,
     double* dot_product) {

     double local_dot_product;

     for (int local_i = 0; local_i < local_n; local_i++)
        local_dot_product += local_x[local_i] * local_y[local_i];

     MPI_Reduce(&local_dot_product, dot_product, 1, MPI_DOUBLE,
MPI_SUM, 0, comm);
}

void Parallel_vector_mult(
   double scalar,
   double local_x[],
   double local_z[],
   int local_n) {

   for (int local_i = 0; local_i < local_n; local_i++) {
       local_z[local_i] = local_x[local_i] * scalar;
   }
}
```

5. Estenda a Questão 4 para permitir que o tamanho dos vetores não seja múltiplo do número de processos usando MPI_Gatherv() e MPI_Scatterv().

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

```c
void Check_for_error(int local_ok, char fname[], char message[],
     MPI_Comm comm);
void Read_n(int* n_p, int* local_n_p, int my_rank, int comm_sz,
     MPI_Comm comm);
void Read_scalar(double* scalar, int my_rank,
     MPI_Comm comm);
void Allocate_vectors(double** local_x_pp, double** local_y_pp,
     double** local_z_pp, int local_n, MPI_Comm comm);
void Read_vector(double local_a[], int local_n, int n, char vec_name[],
     int my_rank, MPI_Comm comm, int comm_sz);
void Print_vector(double local_b[], int local_n, int n, char title[],
     int my_rank, MPI_Comm comm, int comm_sz);
void Parallel_vector_dot_prod(double local_x[], double local_y[],
     MPI_Comm comm, int local_n, double* dot_product);
void Parallel_vector_mult(double scalar, double local_x[],
     double local_z[], int local_n);
void Init_counts_displs(int counts[], int displs[], int n, int
comm_sz);


/*--------------------------------------------------------------------*/
int main(void) {
  int n, local_n;
  int comm_sz, my_rank;
  double *local_x, *local_y, *local_z;
  double scalar, dot_product;
  MPI_Comm comm;

  MPI_Init(NULL, NULL);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &comm_sz);
  MPI_Comm_rank(comm, &my_rank);

  Read_n(&n, &local_n, my_rank, comm_sz, comm);
#  ifdef DEBUG
  printf("Proc %d > n = %d, local_n = %d\n", my_rank, n, local_n);
#  endif
  Allocate_vectors(&local_x, &local_y, &local_z, local_n, comm);

  Read_scalar(&scalar, my_rank, comm);
  Read_vector(local_x, local_n, n, "1", my_rank, comm, comm_sz);
  Print_vector(local_x, local_n, n, "Vector 1 is", my_rank, comm,
comm_sz);
```

```c
  Read_vector(local_y, local_n, n, "2", my_rank, comm, comm_sz);
  Print_vector(local_y, local_n, n, "Vector 2 is", my_rank, comm,
comm_sz);

  Parallel_vector_dot_prod(local_x, local_y, comm, local_n,
&dot_product);
  if(my_rank == 0) {
   printf("O produto escalar entre os vetores 1 e 2 eh:\n%lf\n",
dot_product);
  }

  if(my_rank == 0) {
   printf("The multiplication of the vector 1 by scalar '%lf' is:",
scalar);
  }
  Parallel_vector_mult(scalar, local_x, local_z, local_n);
  Print_vector(local_z, local_n, n, "", my_rank, comm, comm_sz);

  if(my_rank == 0) {
   printf("The multiplication of the vector 2 by scalar '%lf' is:",
scalar);
  }
  Parallel_vector_mult(scalar, local_y, local_z, local_n);
  Print_vector(local_z, local_n, n, "", my_rank, comm, comm_sz);

  free(local_x);
  free(local_y);
  free(local_z);

  MPI_Finalize();

  return 0;
}  /* main */

/*--------------------------------------------------------------------*/
void Check_for_error(
      int       local_ok   /* in */,
      char      fname[]    /* in */,
      char      message[]  /* in */,
      MPI_Comm  comm       /* in */) {
  int ok;

  MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
```

```
  if (ok == 0) {
     int my_rank;
     MPI_Comm_rank(comm, &my_rank);
     if (my_rank == 0) {
        fprintf(stderr, "Proc %d > In %s, %s\n", my_rank, fname,
              message);
        fflush(stderr);
     }
     MPI_Finalize();
     exit(-1);
  }
}  /* Check_for_error */


/*---------------------------------------------------------------------*/
void Read_n(
     int*      n_p,
     int*      local_n_p,
     int       my_rank,
     int       comm_sz,
     MPI_Comm  comm) {
  int local_ok = 1;
  char *fname = "Read_n";
  int quotient, remainder;

  if (my_rank == 0) {
     printf("What's the order of the vectors?\n");
     scanf("%d", n_p);
  }
  MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
  if (*n_p <= 0) local_ok = 0;
  Check_for_error(local_ok, fname,
        "n should be > 0", comm);

  quotient = *n_p/comm_sz;
  remainder = *n_p % comm_sz;

  if (my_rank < remainder) {
     *local_n_p = quotient +1;
  } else {
     *local_n_p = quotient;
  }
}  /* Read_n */
```

```c
/*---------------------------------------------------------------------*/
void Allocate_vectors(
     double**   local_x_pp,
     double**   local_y_pp,
     double**   local_z_pp,
     int        local_n,
     MPI_Comm   comm) {
  int local_ok = 1;
  char* fname = "Allocate_vectors";

  *local_x_pp = malloc(local_n*sizeof(double));
  *local_y_pp = malloc(local_n*sizeof(double));
  *local_z_pp = malloc(local_n*sizeof(double));

  if (*local_x_pp == NULL || *local_y_pp == NULL ||
      *local_z_pp == NULL) local_ok = 0;
  Check_for_error(local_ok, fname, "Can't allocate local vector(s)",
        comm);
}  /* Allocate_vectors */



/*---------------------------------------------------------------------*/
void Read_scalar(
   double* scalar,
   int my_rank,
   MPI_Comm comm) {

       if (my_rank == 0) {
           printf("What's the scalar?\n");
           scanf("%lf", scalar);
       }
       MPI_Bcast(scalar, 1, MPI_DOUBLE, 0, comm);
}


/*---------------------------------------------------------------------*/
void Read_vector(
     double    local_a[],
     int       local_n,
     int       n,
```

```
      char      vec_name[],
      int       my_rank,
      MPI_Comm  comm,
      int comm_sz) {

   double* a = NULL;
   int i;
   int local_ok = 1;
   char* fname = "Read_vector";

   int* counts = NULL;
   int* displs = NULL;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      counts = malloc(comm_sz*sizeof(int));
      displs = malloc(comm_sz*sizeof(int));

      if (a == NULL || counts == NULL || displs == NULL) local_ok = 0;
      Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
            comm);

      Init_counts_displs(counts, displs, n, comm_sz);

      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatterv(a, counts, displs, MPI_DOUBLE,
            local_a, local_n, MPI_DOUBLE, 0, comm);

      free(a);
      free(displs);
      free(counts);
   } else {
      Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
            comm);
      MPI_Scatterv(a, counts, displs, MPI_DOUBLE,
            local_a, local_n, MPI_DOUBLE, 0, comm);
   }
}
```

```
/*---------------------------------------------------------------------*/
void Print_vector(
     double    local_b[],
     int       local_n,
     int       n,
     char      title[],
     int       my_rank,
     MPI_Comm  comm,
     int comm_sz) {

  double* b = NULL;
  int i;
  int local_ok = 1;
  char* fname = "Print_vector";

  int* counts = NULL;
  int* displs = NULL;

  if (my_rank == 0) {
     b = malloc(n*sizeof(double));
     counts = malloc(comm_sz*sizeof(int));
     displs = malloc(comm_sz*sizeof(int));

     if (b == NULL || counts == NULL || displs == NULL) local_ok = 0;
     Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
           comm);

     Init_counts_displs(counts, displs, n, comm_sz);

     MPI_Gatherv(local_b, local_n, MPI_DOUBLE, b, counts, displs,
           MPI_DOUBLE, 0, comm);

     printf("%s\n", title);
     for (i = 0; i < n; i++)
        printf("%f ", b[i]);
     printf("\n");

     free(b);
     free(displs);
     free(counts);
  } else {
```

```
     Check_for_error(local_ok, fname, "Can't allocate temporary
vector",
           comm);
     MPI_Gatherv(local_b, local_n, MPI_DOUBLE, b, counts, displs,
           MPI_DOUBLE, 0, comm);
  }
}  /* Print_vector */


/*--------------------------------------------------------------------*/
void Parallel_vector_dot_prod(
     double  local_x[],
     double  local_y[],
     MPI_Comm comm,
     int     local_n,
     double* dot_product) {

     double local_dot_product;

     for (int local_i = 0; local_i < local_n; local_i++)
        local_dot_product += local_x[local_i] * local_y[local_i];

     MPI_Reduce(&local_dot_product, dot_product, 1, MPI_DOUBLE,
MPI_SUM, 0, comm);
}


/*--------------------------------------------------------------------*/
void Parallel_vector_mult(
   double scalar,
   double local_x[],
   double local_z[],
   int local_n) {

   for (int local_i = 0; local_i < local_n; local_i++) {
       local_z[local_i] = local_x[local_i] * scalar;
   }
}


/*--------------------------------------------------------------------*/
void Init_counts_displs(int counts[], int displs[], int n, int comm_sz)
{
```

```
  int offset, q, quotient, remainder;

  quotient = n/comm_sz;
  remainder = n % comm_sz;
  offset = 0;
  for (q = 0; q < comm_sz; q++) {
     if (q < remainder)
        counts[q] = quotient+1;
     else
        counts[q] = quotient;
     displs[q] = offset;
     offset += counts[q];
  }
}
```

Questão 6 - Item d da questão 3.11 do livro texto: Encontrar prefix sums é uma generalização da soma global. Ao invés de simplesmente encontrar a soma de n valores,

$$x_0 + x_1 + \cdots + x_{n-1},$$

as somas de prefixos são as n somas parciais

$$x_0,\ x_0 + x_1,\ x_0 + x_1 + x_2,\ \ldots,\ x_0 + x_1 + \cdots + x_{n-1}.$$

d. O MPI fornece uma função de comunicação coletiva, MPI_Scan, que pode ser usada para calcular somas de prefixos:

```
int MPI_Scan(
      void*          sendbuf_p   /* in  */,
      void*          recvbuf_p   /* out */,
      int            count       /* in  */,
      MPI_Datatype   datatype    /* in  */,
      MPI_Op         op          /* in  */,
      MPI_Comm       comm        /* in  */);
```

Ela opera em arrays com *count* elementos; tanto *sendbuf_p* quanto *recvbuf_p* devem se referir a blocos com *count* elementos do tipo *datatype*. O argumento *op* é o mesmo que *op* para *MPI_Reduce*. Escreva um programa MPI que gera array aleatório de *count* elementos em cada processo MPI, encontra os prefixos e imprime os resultados.

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

```
void Check_for_error(int local_ok, char fname[], char message[]);
void Get_n(int argc, char* argv[], int* n_p, int* local_n_p);
void Gen_vector(double loc_vect[], int n, int loc_n);
void Read_vector(char prompt[], double loc_vect[], int n, int loc_n);
void Print_vector(char title[], double loc_vect[], int n, int loc_n);
void  Compute_prefix_sums(double  loc_vect[],  double  loc_prefix_sums[],
int n,
     int loc_n);

int my_rank, comm_sz;
MPI_Comm comm;

int main(int argc, char* argv[]) {
  double *loc_vect, *loc_prefix_sums;
  int n, loc_n;

  MPI_Init(&argc, &argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm, &my_rank);
  MPI_Comm_size(comm, &comm_sz);

  Get_n(argc, argv, &n, &loc_n);
  loc_vect = malloc(loc_n*sizeof(double));
  loc_prefix_sums = malloc(loc_n*sizeof(double));

#  ifndef DEBUG
  Gen_vector(loc_vect, n, loc_n);
#  else
  Read_vector("Enter the vector", loc_vect, n, loc_n);
#  endif
  Print_vector("Initial vector", loc_vect, n, loc_n);
  Compute_prefix_sums(loc_vect, loc_prefix_sums, n, loc_n);
  Print_vector("Prefix sums", loc_prefix_sums, n, loc_n);

  free(loc_vect);
  free(loc_prefix_sums);

  MPI_Finalize();
  return 0;
}

/*--------------------------------------------------------------------*/
```

```
void Check_for_error(
               int       local_ok,
               char      fname[],
               char      message[]) {
  int ok;

  MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
  if (ok == 0) {
     int my_rank;
     MPI_Comm_rank(comm, &my_rank);
     if (my_rank == 0) {
        fprintf(stderr, "Proc %d > In %s, %s\n", my_rank, fname,
              message);
        fflush(stderr);
     }
     MPI_Finalize();
     exit(-1);
  }
}

/*---------------------------------------------------------------------*/
void Get_n(int argc, char* argv[], int* n_p, int* local_n_p) {
  int local_ok = 1;

  if (my_rank == 0){
     if (argc != 2)
        *n_p = 0;
     else
        *n_p = strtol(argv[1], NULL, 10);
  }

  MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
  if (*n_p <= 0 || *n_p % comm_sz != 0) local_ok = 0;
  Check_for_error(local_ok, "Get_n",
        "n should be > 0 and evenly divisible by comm_sz");

  *local_n_p = *n_p / comm_sz;
}  /* Get_n */


/*---------------------------------------------------------------------*/
void Gen_vector(double loc_vect[], int n, int loc_n) {
  int i;
```

```
  double* tmp = NULL;

  if (my_rank == 0) {
     tmp = malloc(n*sizeof(double));
     srandom(1);
     for (i = 0; i < n; i++)
        tmp[i] = random()/((double) RAND_MAX);
      MPI_Scatter(tmp, loc_n, MPI_DOUBLE, loc_vect, loc_n, MPI_DOUBLE,
0,
           comm);
     free(tmp);
  } else {
      MPI_Scatter(tmp, loc_n, MPI_DOUBLE, loc_vect, loc_n, MPI_DOUBLE,
0,
           comm);
  }
}

/*--------------------------------------------------------------------*/
void Read_vector(char prompt[], double loc_vect[], int n, int loc_n) {
  int i;
  double* tmp = NULL;

  if (my_rank == 0) {
     tmp = malloc(n*sizeof(double));
     printf("%s\n", prompt);
     for (i = 0; i < n; i++)
        scanf("%lf", &tmp[i]);
      MPI_Scatter(tmp, loc_n, MPI_DOUBLE, loc_vect, loc_n, MPI_DOUBLE,
0,
           comm);
     free(tmp);
  } else {
      MPI_Scatter(tmp, loc_n, MPI_DOUBLE, loc_vect, loc_n, MPI_DOUBLE,
0,
           comm);
  }
}

/*--------------------------------------------------------------------*/
void Print_vector(char title[], double loc_vect[], int n, int loc_n) {
  int i;
  double* tmp = NULL;
```

```
  if (my_rank == 0) {
     tmp = malloc(n*sizeof(double));
     MPI_Gather(loc_vect, loc_n, MPI_DOUBLE, tmp, loc_n, MPI_DOUBLE, 0,
           comm);
     printf("%s\n   ", title);
     for (i = 0; i < n; i++)
        printf("%.2f ", tmp[i]);
     printf("\n");
     free(tmp);
  } else {
     MPI_Gather(loc_vect, loc_n, MPI_DOUBLE, tmp, loc_n, MPI_DOUBLE, 0,
           comm);
  }
}

/*--------------------------------------------------------------------*/
void  Compute_prefix_sums(double  loc_vect[],  double  loc_prefix_sums[],
int n,
     int loc_n) {
  int loc_i;
  double my_sum, pred_sum;

  loc_prefix_sums[0] = loc_vect[0];
  for (loc_i = 1; loc_i < loc_n; loc_i++)
              loc_prefix_sums[loc_i]   =   loc_prefix_sums[loc_i-1]   +
loc_vect[loc_i];

  my_sum = loc_prefix_sums[loc_n-1];
  MPI_Scan(&my_sum, &pred_sum, 1, MPI_DOUBLE, MPI_SUM, comm);
#  ifdef DEBUG
  printf("Proc %d > my_sum = %.2f, pred_sum = %.2f\n",
        my_rank, my_sum, pred_sum);
#  endif

  pred_sum -= my_sum;
  for (loc_i = 0; loc_i < loc_n; loc_i++)
     loc_prefix_sums[loc_i] += pred_sum;
}
```

10 - 3.19 do livro texto: `MPI_Type_indexed` pode ser usado para construir um tipo de

dado derivado de elementos de vetor arbitrário. Sua sintaxe é

```
int MPI_Type_indexed(
      int            count                    /* in  */,
      int            array_of_blocklengths[]  /* in  */,
      int            array_of_displacements[] /* in  */,
      MPI_Datatype   old_mpi_t                /* in  */,
      MPI_Datatype*  new_mpi_t_p)             /* out */);
```

Ao contrário do `MPI_Type_create_struct`, os deslocamentos são medidos em unidades de `old_mpi_t` - não de bytes. Use o tipo `MPI_Type_indexed` para criar um datatype derivado que corresponda à parte triangular superior de uma matriz quadrada. Por exemplo, na matriz 4 × 4

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

a parte triangular superior são os elementos 0, 1, 2, 3, 5, 6, 7, 10, 11, 15. O processo 0 deve ser lido em uma matriz n × n como uma matriz unidimensional, criar o datatype derivado e enviar a parte triangular superior com uma única chamada para o `MPI_Send`. O processo 1 deve receber a parte triangular superior com uma única chamada para a `MPI_Recv` e depois imprimir os dados que recebeu.

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

int my_rank, comm_sz;
MPI_Comm comm;

void Check_for_error(int local_ok, char fname[], char message[]);
void Get_n(int* n_p);
void Read_loc_mat(double loc_mat[], int n);
void Build_indexed_type(int n, MPI_Datatype* indexed_mpi_t_p);
void Print_loc_mat(char title[], double loc_mat[], int n);


int main(void) {
```

```
  int n;
  double *loc_mat;
  MPI_Datatype indexed_mpi_t;

  MPI_Init(NULL, NULL);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &comm_sz);
  MPI_Comm_rank(comm, &my_rank);

  if (comm_sz < 2)
     Check_for_error(0, "main", "comm size must be >= 2");

  Get_n(&n);
  loc_mat = malloc(n*n*sizeof(double));

  Build_indexed_type(n, &indexed_mpi_t);
  if (my_rank == 0) {
     Read_loc_mat(loc_mat, n);
     MPI_Send(loc_mat, 1, indexed_mpi_t, 1, 0, comm);
  } else if (my_rank ==1){
     memset(loc_mat, 0, n*n*sizeof(double));
               MPI_Recv(loc_mat,   1,   indexed_mpi_t,   0,   0,   comm,
MPI_STATUS_IGNORE);
     Print_loc_mat("Received matrix", loc_mat, n);
  }

  free(loc_mat);
  MPI_Type_free(&indexed_mpi_t);
  MPI_Finalize();
  return 0;
}


/*---------------------------------------------------------------------*/
void Check_for_error(
               int       local_ok   /* in */,
               char      fname[]    /* in */,
               char      message[]  /* in */) {
  int ok;

  MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
  if (ok == 0) {
     int my_rank;
```

```
    MPI_Comm_rank(comm, &my_rank);
    if (my_rank == 0) {
       fprintf(stderr, "Proc %d > In %s, %s\n", my_rank, fname,
             message);
       fflush(stderr);
    }
    MPI_Finalize();
    exit(-1);
  }
}


/*--------------------------------------------------------------------*/
void Get_n(int*    n_p           /* out */) {

  if (my_rank == 0) {
     printf("Enter the order of the matrix\n");
     scanf("%d", n_p);
  }

  MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

  if (*n_p <= 0)
     Check_for_error(0, "Get_n", "n must be positive");
}


/*--------------------------------------------------------------------*/
void Build_indexed_type(
     int            n                /* in  */,
     MPI_Datatype*  indexed_mpi_t_p  /* out */) {
  int i;
  int* array_of_block_lens;
  int* array_of_disps;

  array_of_block_lens = malloc(n*sizeof(int));
  array_of_disps = malloc(n*sizeof(int));

  for (i = 0; i < n ; i++) {
     array_of_block_lens[i] = n-i;
     array_of_disps[i] = i*(n+1);
  }
```

```
  MPI_Type_indexed(n, array_of_block_lens, array_of_disps, MPI_DOUBLE,
        indexed_mpi_t_p);
  MPI_Type_commit(indexed_mpi_t_p);

  free(array_of_block_lens);
  free(array_of_disps);
}


/*--------------------------------------------------------------------*/
void Read_loc_mat(
            double    loc_mat[]  /* out */,
            int       n          /* in  */) {
  int i,j;

  printf("Enter the matrix\n");
  for (i = 0; i < n; i++)
     for (j = 0; j < n; j++)
        scanf("%lf", &loc_mat[i*n + j]);
}


/*--------------------------------------------------------------------*/
void Print_loc_mat(
            char      title[]    /* in */,
            double    loc_mat[]  /* in */,
            int       n          /* in */) {
  int i,j;

  printf("Proc %d > %s\n", my_rank, title);
  for (i = 0; i < n; i++) {
     for (j = 0; j < n; j++)
        printf("%.2f ", loc_mat[i*n + j]);
     printf("\n");
  }
  printf("\n");
}
```

11. As funções `MPI_Pack` e `MPI_Unpack` oferecem uma alternativa aos tipos de dados derivados para agrupamento de dados. O `MPI_Pack` copia os dados a serem enviados, um bloco de cada vez, em um buffer fornecido pelo usuário. O buffer pode

então ser enviado e recebido. Após a recepção dos dados, o `MPI_Unpack` pode ser usado para desembalá-los do buffer de recepção. A sintaxe do `MPI_Pack` é:

```
int MPI_Pack(
      void*          in_buf          /* in     */,
      int            in_buf_count    /* in     */,
      MPI_Datatype   datatype        /* in     */,
      void*          pack_buf        /* out    */,
      int            pack_buf_sz     /* in     */,
      int*           position_p      /* in/out */,
      MPI_Comm       comm            /* in     */);
```

Portanto, poderíamos empacotar os dados de entrada no programa de regras trapezoidais com o seguinte código:

```
char pack_buf[100];
int position = 0;

MPI_Pack(&a, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&b, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&n, 1, MPI_INT, pack_buf, 100, &position, comm);
```

A chave é o argumento *position*. Quando o `MPI_Pack` é chamado, a posição deve referenciar o primeiro slot disponível no `pack_buf`. Quando `MPI_Pack` retorna, ele se refere para o primeiro slot disponível após os dados que acabaram de ser embalados, portanto, após o processo 0 executar este código, todos os processos podem chamar `MPI_Bcast`:

```
MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, comm);
```

Note que o MPI datatype para um buffer embalado é `MPI_PACKED`. Agora o outros processos podem desempacotar os dados usando: `MPI_Unpack`:

```
int MPI_Unpack(
      void*          pack_buf        /* in     */,
      int            pack_buf_sz     /* in     */,
      int*           position_p      /* in/out */,
      void*          out_buf         /* out    */,
      int            out_buf_count   /* in     */,
      MPI_Datatype   datatype        /* in     */,
      MPI_Comm       comm            /* in     */);
```

Isto pode ser usado "invertendo" as etapas do MPI_Pack, ou seja, os dados são desembrulhado um bloco de cada vez, começando com posição = 0.

Escreva outra função de entrada para o programa de regras trapezoidais. Este deve-se usar o `MPI_Pack` no processo 0 e o `MPI_Unpack` nos outros processos.

```
void Get_input(
      int       my_rank  /* in  */,
      double*   a_p      /* out */,
      double*   b_p      /* out */,
      int*      n_p      /* out */) {

   char pack_buf[PACK_BUF_SZ];
   int position = 0;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      MPI_Pack(a_p, 1, MPI_DOUBLE, pack_buf, PACK_BUF_SZ, &position,
            MPI_COMM_WORLD);
      MPI_Pack(b_p, 1, MPI_DOUBLE, pack_buf, PACK_BUF_SZ, &position,
            MPI_COMM_WORLD);
      MPI_Pack(n_p, 1, MPI_INT, pack_buf, PACK_BUF_SZ, &position,
            MPI_COMM_WORLD);
   }

   MPI_Bcast(pack_buf, PACK_BUF_SZ, MPI_PACKED, 0, MPI_COMM_WORLD);

   if (my_rank > 0) {
      MPI_Unpack(pack_buf, PACK_BUF_SZ, &position, a_p, 1, MPI_DOUBLE,
            MPI_COMM_WORLD);
      MPI_Unpack(pack_buf, PACK_BUF_SZ, &position, b_p, 1, MPI_DOUBLE,
            MPI_COMM_WORLD);
      MPI_Unpack(pack_buf, PACK_BUF_SZ, &position, n_p, 1, MPI_INT,
            MPI_COMM_WORLD);
   }
}
```

Questão 12 - 3.22 do livro texto: Cronometre nossa implementação da regra trapezoidal que usa `MPI_Reduce`. Como você escolhe n, o número de trapezoidais? Como comparar o tempo mínimo com o tempo médio e o tempo da mediana? Quais são as

velocidades? Quais são as eficiências? Com base nos dados coletados, você diria que a regra do trapézio é escalável?

O número de trapézios n é escolhido para ser suficientemente grande para que a variação entre os tempos das entradas seja perceptível.

Quanto aos tempos, existem algumas maneiras que podemos pensar em medir. Ao pegar a média entre as execuções realizadas, estamos sujeitos a sofrer interferência de overflow, ou seja, houve uma interferência externa no programa, como a execução de algum processo no núcleo em que o código está sendo executado. Podemos pensar em pegar a mediana, entretanto, ela nos fornece um valor central entre as amostras, o que não é interessante para o nosso caso, pois queremos o melhor tempo de execução sem interferência externa. Isso é gerado ao escolhermos o tempo mínimo de execução, pois é aquele que tem a menor interferência.

Os tempos de execução do programa, em milissegundos, com uma entrada de trapézios n = 1.000.000.000:

Processos	Número de trapézios n				
	$2{,}5 \times 10^9$	5×10^9	10×10^9	20×10^9	40×10^9
1	4.35	3.98	3.73	2.81	4.34
2	3.89	3.82	3.87	3.80	4.35
4	2.34	3.86	3.83	5.02	3.93
8	3.28	3.55	3.16	2.56	3.97
16	4.42	4.15	4.26	2.74	2.59

Eficiência:

	$2{,}5 \times 10^9$	5×10^9	10×10^9	20×10^9	40×10^9
2	0.57	0.52	0.48	0.37	0.49
4	0.48	0.26	0.24	0.14	0.28
8	0.17	0.14	0.15	0.14	0.13

16	0.06	0.06	0.05	0.06	0.10

De acordo com os testes realizados, mesmo com algumas entradas possuindo uma eficiência melhor ou igual que a anterior, a aplicação não é escalável, uma vez que conforme aumentamos o número de trapézios, para a maioria dos casos, a eficiência diminui. Isso configura um problema não escalável.