Universidade Federal do Rio Grande do Norte
Departamento de Engenharia de Computacão e Automação

Programação Concorrente e Distribuida

Segunda Lista de Exercícios

Natal-RN, Brasil
[Outubro de 2017]

Professor

Prof. Samuel Xavier - DCA/UFRN

Aluno

Igor Macedo Silva - Bacharelando em Engenharia de Computação

Sumário

Lista de Figuras

Lista de Tabelas

1 Descrição

Lista da 2a unidade Descrição: Apresentar as respostas a todas as questões de exercício do livro texto, com exceção às questões: 3.3, 3.15 e 3.18.

Instruções para resolução (LEIAM!):

- Procure responder corretamente todas as questões da lista;
- Suas respostas serão validadas de forma oral por amostragem - geralmente de 2 à 3 defesas orais;
- Se não conseguir responder alguma questão, procure esclarecer as dúvidas em tempo em sala de aula com o professor, pelo SIGAA, com um colega, ou por e-mail. Se necessário, é possível marcar um horário para tirar dúvidas na sala do professor;
- Não serão aceitas respostas "mágicas", ou seja, quando a resposta está na lista entregue mas você não sabe explicar como chegou a ela. Sua nota nesse caso será 0 (zero). Mesmo que não saiba explicar apenas parte da sua resposta;
- Procure entregar a resolução da lista de forma organizada. Isso pode favorecer a sua nota;
- Os códigos dos programas requisitados (ou as partes relevantes) deverão aparecer no corpo da resolução da questão;
- A resolução da lista deverá ser entregue em formato PDF em apenas 1 (um) arquivo;
- O envio da resolução pode ser feito inúmeras vezes. Utilize-se disso para manter sempre uma versão atualizada das suas respostas e evite problemas com o envio próximo ao prazo de submissão devido a instabilidades no SIGAA;
- A lista com o número das questões respondidas deve aparecer na primeira folha da lista. Não será aceita alteração nessa lista.
- Procure preparar sua defesa oral para cada questão. Explicações diretas e sem arrodeios favorecerão a sua nota;
- A defesa deverá ser agendada com antecedência. Para isso, indique por email (samuel@dca.ufrn.br) no mínimo 3 horários dentro dos intervalos disponíveis em pelo menos 3 turnos diferentes. Caso não tenha disponibilidade em 3 turnos diferentes, deverá apresentar uma justificativa.
- Os horários disponíveis serão disponibilizados em uma notícia na turma virtual e serão atualizados a medida que os agendamentos forem sendo fixados.
- A defesa oral leva apenas de 10 a 15 minutos em horários fixados com antecedência. Não será tolerado que o aluno chegue atrasado para a sua prova.

Período: Inicia em 20/09/2017 às 00h00 e finaliza em 11/10/2017 às 23h59

2 Questões

2.1 Questão 3.1

What happens in the greetings program if, instead of `strlen(greeting) + 1`, we use `strlen(greeting)` for the length of the message being sent by processes `1, 2, ..., comm_sz- 1`? What happens if we use `MAX_STRING` instead of `strlen(greeting) + 1`? Can you explain these results?

Neste caso, o `+ 1` indica que o caractere de terminação da string também deve ser incluido no envio da mensagem. Se substituirmos por apenas `strlen(greeting)` a mensagem pode ser impressa corretamente ou não, dependendo do conteúdo presente no buffer de recebimento. Caso o buffer de recebimento esteja preenchido com zeros ("\0"), o comando `printf()` vai conseguir imprimir a mensagem corretamente mesmo que não exista um terminador nulo na mensagem enviada.

Em testes feitos localmente, as mensagens sempre foram exibidas corretamente, pois os buffers estavam sempre sendo iniciados com zero em suas posições de memória.

2.2 Questão 3.2

Modify the trapezoidal rule so that it will correctly estimate the integral even if `comm_sz` doesn't evenly divide n. (You can still assume that n $\geq$ `comm_sz`.)

Se `comm_sz` não divide perfeitamente n, devemos alocar os trapézios restantes nos processos de maneira mais deliberada. o pseudocódigo poderia ser:

```
1  get a, b, n;
2  h = (b - a)/n;
3  local_n = n/comm_sz; //Devemos garantir que a divisao sera inteira
4
5  n_mod_comm = n % comm_sz;
6  local_a = a + (my_rank*local_n* +
7                 my_rank*((int)(my_rank < n_mod_comm))+
8                 n_mod_comm*((int)(my_rank >= n_mod_comm && n_mod_comm
                     > 0)))*h;
9
10
11 local_b = local_a + (local_n + (int)(my_rank < n_mod_comm))*h;
12 local_integral = Trap(local_a, local_b, local_n, h);
```

O trecho da linha 7 se refere ao acrescimo incremental que deve acontecer ao h para cada rank. Isto é, no caso de `n_mod_comm = 3`, o primeiro `local_a` deve receber um acrescimo de 0, o segundo, de 1, o terceiro, de 3 e assim sucessivamente. Isso acontece pois é uma compensação ao `local_b` que está sendo acrescido de 1 até o momento em que todos os trapézios extras (no caso de n não exatamente divisivel por `comm_sz`) forem alocados em algum processo. E isso vai acontecer somente quando `my_rank` $\geq$ `n_mod_comm`

2.3 Questão 3.4

Modify the program that just prints a line of output from each process (`mpi_output.c`) so that the output is printed in process rank order: process 0s output first, then process 1s, and so on.

```c
#include <stdio.h>
#include <mpi.h>
#include <string.h> /* For strlen */

const int MAX_STRING = 100;
int main(void) {
    char phrase[MAX_STRING];
    int my_rank, comm_sz;

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

    if (my_rank != 0) {
        sprintf(phrase, "Proc %d of %d > Does anyone have a toothpick
            ?", my_rank, comm_sz);
        MPI_Send(phrase, strlen(phrase)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
    } else {
        printf("Proc %d of %d > Does anyone have a toothpick?\n",
            my_rank, comm_sz);
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(phrase, MAX_STRING, MPI_CHAR, q, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n",phrase);
        }
    }

    MPI_Finalize();
    return 0;
} /* main */
```

O princípio da resulução desta questão é perceber que para lidar com o não-determinismo do output de um programa MPI, nós devemos mandar todas as mensagens de saída para um único processo, neste caso o processo 0. Dessa forma, apenas um processo é responsável por gerenciar as mensagens de saída e, assim, podemos controlar a ordem de saída das mensagens.

2.4 Questão 3.5

In a binary tree, there is a unique shortest path from each node to the root. The length of this path is often called the depth of the node. A binary tree in which every nonleaf has two children is called a full binary tree, and a full binary tree in which every leaf has the same depth is sometimes called a complete binary tree. See Figure 3.14. Use the principle of mathematical induction to prove that if T is a complete binary tree with n leaves, then the depth of the leaves is log 2 (n).

Para fazer a indução vamos considerar $\log_2(n) = d$, onde d é a profundidade.
Então,

$$\log_2(n) = d \implies 2^d = n \tag{1}$$

Considerando o caso base em que d = 0,

$$\begin{aligned} 2^d &= n \\ 2^0 &= n \\ 2^0 &= 1 \end{aligned} \tag{2}$$

Tomamos que $d = k$ e assumimos que $2^k = n$ é verdadeiro. Então, aplicamos o passo de indução, onde $d = k + 1$, e para a próxima profundidade, $n_i = n_{i-1} \cdot 2$. Logo,

$$\begin{aligned} 2^{k+1} &= n \cdot 2 \\ &= 2^k 2^1 \\ &= 2^{k+1} \end{aligned} \tag{3}$$

Portanto, $2^d = n$ e, logo $log_2(n) = d$, onde n é o número de folhas e d é a profundidade.

2.5 Questão 3.6

Suppose comm_sz = 4 and suppose that x is a vector with n = 14 components.

a. How would the components of x be distributed among the processes in a program that used a block distribution?

b. How would the components of x be distributed among the processes in a program that used a cyclic distribution?

c. How would the components of x be distributed among the processes in a program that used a block-cyclic distribution with blocksize b = 2?

You should try to make your distributions general so that they could be used regardless of what comm sz and n are. You should also try to make your distributions "fair" so that if q and r are any two processes, the difference between the number of components assigned to q and the number of components assigned to r is as small as possible.

Processos	Bloco	Cíclico	Bloco-Cíclico (tamanho do bloco = 2)
0	0, 1, 2, 3	0, 4, 8, 12	0 1, 8 9
1	4, 5, 6, 7	1, 5, 9, 13	2 3, 10 11
2	8, 9, 10,	2, 6, 10,	4 5, 12 13
3	11, 12, 13,	3, 7, 11,	6 7

Tabela 1: Distribuição dos elementos do vetor

2.6 Questão 3.7

What do the various MPI collective functions do if the communicator contains a single process?

O processo MPI não terá qualquer outro processo para enviar os dados, logo o processamento deve ser feito nesse único processo. É um principio que permite que os processo MPI sejam executados corretamente independente de número de nós/processos disponíveis.

A forma que isso é implementado é enviar os dados para si mesmo, algo que várias funções coletivas do MPI executam. Em alguns casos, para reduzir o movimento desnecessário de dados na memória, o MPI fornece uma flag chamada MPI_IN_PLACE, que permite que o buffer de recebimento seja o mesmo do de envio, melhorando o desempenho. Essa flag não funciona em todas as funções coletivas.

`https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node145.htm`

2.7 Questão 3.8

Suppose `comm_sz = 8` and `n = 16`.

a. Draw a diagram that shows how MPI Scatter can be implemented using tree-structured communication with `comm_sz` processes when process 0 needs to distribute an array containing n elements.

b. Draw a diagram that shows how MPI Gather can be implemented using tree-structured communication when an n-element array that has been distributed among `comm_sz` processes needs to be gathered onto process 0.

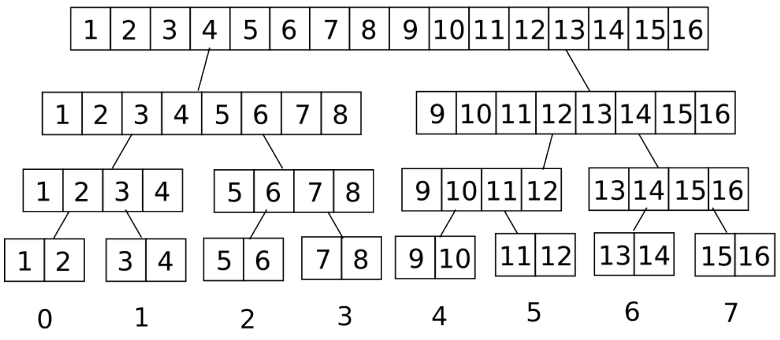


Figura 1: Scatter em comunição baseada em árvore

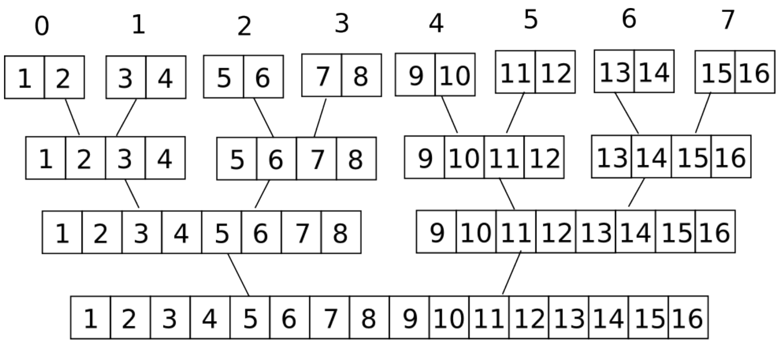


Figura 2: Gather em comunição baseada em árvore

2.8 Questão 3.9

Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by `comm_sz` .

```c
/*Esse codigo foi modificado de mpi_vector_add.c */

/* File:      mpi_vector_add.c
 *
 * Purpose:   Implement parallel vector addition using a block
 *            distribution of the vectors.  This version also
 *            illustrates the use of MPI_Scatter and MPI_Gather.
 *
 * Compile:   mpicc -g -Wall -o mpi_vector_add mpi_vector_add.c
 * Run:       mpiexec -n <comm_sz> ./vector_add
 *
 * Input:     The order of the vectors, n, and the vectors x and y
 * Output:    The sum vector z = x+y
 *
 * Notes:
 * 1.  The order of the vectors, n, should be evenly divisible
 *     by comm_sz
 * 2.  DEBUG compile flag.
 * 3.  This program does fairly extensive error checking.  When
 *     an error is detected, a message is printed and the processes
 *     quit.  Errors detected are incorrect values of the vector
 *     order (negative or not evenly divisible by comm_sz), and
 *     malloc failures.
 *
 * IPP:  Section 3.4.6 (pp. 109 and ff.)
 */

#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 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 Read_scalar(double* scalar_p, int my_rank, int comm_sz, MPI_Comm
    comm);
void Print_vector(double local_b[], int local_n, int n, char title[],
      int my_rank, MPI_Comm comm);
void Parallel_vector_dotproduct(double local_x[], double local_y[],
      double local_z[], double* result, int local_n, int n, MPI_Comm
          comm);
void Parallel_scalar_mutiplication(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, result;
   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);

   Allocate_vectors(&local_x, &local_y, &local_z, local_n, comm);

   Read_vector(local_x, local_n, n, "x", my_rank, comm);
   Print_vector(local_x, local_n, n, "x is", my_rank, comm);
   Read_vector(local_y, local_n, n, "y", my_rank, comm);
   Print_vector(local_y, local_n, n, "y is", my_rank, comm);

   Read_scalar(&scalar, my_rank, comm_sz, comm);

   Parallel_scalar_mutiplication(scalar, local_x, local_x, local_n);
   Print_vector(local_x, local_n, n, "now x is", my_rank, comm);
   Parallel_vector_dotproduct(local_x, local_y, local_z, &result,
       local_n,
                                     n, comm);

   if (my_rank == 0) {
      printf("The result of (scalar*x[]).y[] is %lf \n", result);
   }

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

   MPI_Finalize();

   return 0;
}  /* main */

/*---------------------------------------------------------------------
 * Function:  Check_for_error
 * Purpose:   Check whether any process has found an error.  If so,
 *            print message and terminate all processes.  Otherwise,
 *            continue execution.
 * In args:   local_ok:  0 if calling process has found an error, 1
 *               otherwise
 *            fname:     name of function calling Check_for_error
 *            message:   message to print if there's an error
 *            comm:      communicator containing processes calling
 *                       Check_for_error:  should be MPI_COMM_WORLD.
 *
 * Note:
 *    The communicator containing the processes calling
   Check_for_error
 *    should be MPI_COMM_WORLD.
 */
void Check_for_error(
      int       local_ok   /* in */,
      char      fname[]    /* in */,
      char      message[]  /* in */,
      MPI_Comm  comm       /* in */) {
   int ok;

   /* Pega o minimo do vetor
      Se o minimo for zero, aconteceu algum erro,
      caso contrario, todos os processos retornaram 1 e estao 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 */


/*-------------------------------------------------------------------
 * Function:   Read_n
 * Purpose:    Get the order of the vectors from stdin on proc 0 and
 *             broadcast to other processes.
 * In args:    my_rank:    process rank in communicator
 *             comm_sz:    number of processes in communicator
 *             comm:       communicator containing all the processes
 *                         calling Read_n
 * Out args:   n_p:        global value of n
 *             local_n_p:  local value of n = n/comm_sz
 *
 * Errors:     n should be positive and evenly divisible by comm_sz
 */
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 */


```

```c
/*-------------------------------------------------------------------
 * Function:   Allocate_vectors
 * Purpose:    Allocate storage for x, y, and z
 * In args:    local_n:  the size of the local vectors
 *            comm:      the communicator containing the calling
    processes
 * Out args:   local_x_pp, local_y_pp, local_z_pp:  pointers to memory
 *               blocks to be allocated for local vectors
 *
 * Errors:     One or more of the calls to malloc fails
 */
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 */


/*-------------------------------------------------------------------
 * Function:    Read_vector
 * Purpose:     Read a vector from stdin on process 0 and distribute
 *              among the processes using a block distribution.
 * In args:     local_n:  size of local vectors
 *              n:        size of global vector
 *              vec_name: name of vector being read (e.g., "x")
 *              my_rank:  calling process' rank in comm
 *              comm:     communicator containing calling processes
 * Out arg:     local_a:  local vector read
 *
 * Errors:      if the malloc on process 0 for temporary storage
 *              fails the program terminates
 *
 * Note:
 *    This function assumes a block distribution and the order
 *   of the vector evenly divisible by comm_sz.
 */
void Read_vector(
      double     local_a[]   /* out */,
      int        local_n     /* in  */,
      int        n           /* in  */,
      char       vec_name[]  /* in  */,
      int        my_rank     /* in  */,
      MPI_Comm   comm        /* in  */) {

   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]); // reads a double (long float)
      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);
   }
}  /* Read_vector */

/*------------------------------------------------------------------
 * Function:  Read_scalar
 * Purpose:   Get the the scalar number from stdin on proc 0 and
 *            broadcast to other processes.
 * In args:   my_rank:    process rank in communicator
 *              comm_sz:     number of processes in communicator
 *              comm:        communicator containing all the processes
 *                           calling Read_n
 * Out args:   scalar_p:         global value of n
 *
 */
void Read_scalar(
        double*    scalar_p         /* out */,
        int        my_rank     /* in  */,
        int        comm_sz     /* in  */,
        MPI_Comm   comm        /* in  */) {

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

}  /* Read_scalar */

/*-------------------------------------------------------------------
 * Function:  Print_vector
 * Purpose:   Print a vector that has a block distribution to stdout
 * In args:   local_b:  local storage for vector to be printed
 *            local_n:  order of local vectors
 *            n:        order of global vector (local_n*comm_sz)
 *            title:    title to precede print out
 *            comm:     communicator containing processes calling
 *                      Print_vector
 *
 * Error:     if process 0 can't allocate temporary storage for
 *            the full vector, the program terminates.
 *
 * Note:
 *    Assumes order of vector is evenly divisible by the number of
 *    processes
 */
void Print_vector(
        double     local_b[]   /* in */,
        int        local_n     /* in */,
        int        n           /* in */,
        char       title[]     /* in */,
        int        my_rank     /* in */,
        MPI_Comm   comm        /* in */) {

```

```
   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);
   }
}  /* Print_vector */


/*------------------------------------------------------------------
 * Function:  Parallel_vector_dotproduct
 * Purpose:   Add a vector that's been distributed among the
    processes
 * In args:   local_x:  local storage of one of the vectors being
    added
 *            local_y:  local storage for the second vector being
    added
 *            local_n:  the number of components in local_x, local_y,
 *                      and local_z
 * Out arg:   local_z:  local storage for the sum of the two vectors
 */
void Parallel_vector_dotproduct(
      double   local_x[]  /* in  */,
      double   local_y[]  /* in  */,
      double   local_z[]  /* out */,
      double*  result     /* out */,
      int      local_n    /* in  */,
      int      n          /* in  */,
      MPI_Comm comm       /* in */) {
   int local_i;

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

  double local_total_z = 0.0;
  for (local_i = 0; local_i < local_n; local_i++){
      local_total_z += local_z[local_i];
  }

  MPI_Reduce(&local_total_z, result, 1, MPI_DOUBLE, MPI_SUM, 0, comm)
     ;

}  /* Parallel_vector_dotproduct */

/*---------------------------------------------------------------------
 * Function:   Parallel_scalar_mutiplication
 * Purpose:    Add a vector that's been distributed among the
     processes
 * In args:    scalar:   storage for the scalar value
 *             local_x:  local storage for the second vector being
     added
 *             local_n:  the number of components in local_x, local_y,
 *                       and local_z
 * Out arg:    local_z:  local storage for the scalar mutiplication of
 *                       the vector
 */
void Parallel_scalar_mutiplication(
      double   scalar     /* in  */,
      double   local_x[]  /* in  */,
      double   local_z[]  /* out */,
      int      local_n    /* in  */) {
   int local_i;

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

2.9 Questão 3.10

In the `Read_vector` function shown in Program 3.9, we use `local_n` as the actual argument for two of the formal arguments to `MPI_Scatter` : `send_count` and `recv_count`. Why is it OK to alias these arguments?

É possível alinhar os dois argumentos porque `send_count` deve ser o número de elementos, de acordo com `send_type`, que serão enviados para cada processo. E, de forma semelhante, o `recv_count` deve ser o número de elementos recebidos do procesos raíz, de acordo com `recv_type`. Portanto, neste caso, ambos os argumentos devem receber o valor de `local_n`.

2.10 Questão 3.11

Finding **prefix sums** is a generalization of global sum. Rather than simply finding the sum of n values,

$$x_0 + x_1 + ... + x_{n-1},$$

the prefix sums are the n partial sums

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

a. Devise a serial algorithm for computing the n prefix sums of an array with n elements.

b. Parallelize your serial algorithm for a system with n processes, each of which is storing one of the x_is.

c. Suppose $n = 2^k$ for some positive integer k. Can you devise a serial algorithm and a parallelization of the serial algorithm so that the parallel algorithm requires only k communication phases?

d. MPI provides a collective communication function, `MPI_Scan` , that can be used to compute prefix sums:

```
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  */);
```

It operates on arrays with `count` elements; both `sendbuf_p` and `recvbuf_p` should refer to blocks of `count` elements of type `datatype`. The `op` argument is the same as `op` for `MPI_Reduce`. Write an MPI program that generates a random array of count elements on each MPI process, finds the prefix sums, and prints the results.

a.

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

int main(){
    int n = 15;
```

```c
    srand(time(NULL));

    int vec[n];
    int prefix_sum[n];

    for(int i = 0; i < n; i++){
        vec[i] = rand() % 10; // number between 0 and 9
        printf("%d ", vec[i]);
    }
    printf("\n");

    for(int i = 0; i < n; i++){
        prefix_sum[i] = 0;
        for(int j = 0; j <= i; j++){
            prefix_sum[i] += vec[j];
        }
        printf("%d ", prefix_sum[i]);
    }
}
```

b.

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

int main(){
    int comm_sz; /* Number of processes */
    int my_rank; /* My process rank */
    MPI_Init(NULL, NULL);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &comm_sz);
    MPI_Comm_rank(comm, &my_rank);

    int vec_i;
    int prefix_sum = 0;
    int n = comm_sz;
    int vec[n];

    if (my_rank == 0){
        srand(time(NULL));
        for(int i = 0; i < n; i++){
            vec[i] = rand() % 10; // number between 0 and 9
            printf("%d ", vec[i]);
        }
```

```
            printf("\n");

            MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
        }
        else {
            MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
        }

        // envia para todos os processo maiores do que ele mesmo
        for(int i = my_rank; i < comm_sz; i++){
            MPI_Send(&vec_i, 1, MPI_INT, i, 0, comm);
        }

        // recebe de todos os processos menores que ele mesmo
        prefix_sum += vec_i;
        for(int i = 0; i < my_rank; i++){
            int aux_vec_i = 0;
            MPI_Recv(&aux_vec_i, 1, MPI_INT, i, 0, comm,
                MPI_STATUS_IGNORE);
            prefix_sum += aux_vec_i;
        }


        // Imprime o resultado na tela
        if (my_rank != 0) {
            MPI_Send(&prefix_sum, 1, MPI_INT, 0, 1, comm);
        } else {
            printf("%d ", vec_i);
            for (int q = 1; q < comm_sz; q++) {
                int aux;
                MPI_Recv(&aux, 1, MPI_INT, q, 1, comm, MPI_STATUS_IGNORE)
                    ;
                printf("%d ", aux);
            }
            printf("\n");
        }


        MPI_Finalize();
        return 0;
}
```

c. **Serial**

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

```c
#include <stdlib.h>
#include <math.h>

int main(){
    int n = 8;
    srand(time(NULL));

    int vec[n];

    for(int i = 0; i < n; i++){
        vec[i] = 1; //rand() % 10; // number between 0 and 9
        printf("%d ", vec[i]);
    }
    printf("\n");

    for(int k = 0; k < log2(n); k++){
        printf("k is %d\n",k);
        for(int i = pow(2,k) - 1; i < pow(2,log2(n)); i = i + pow(2,k
            +1)){
            printf("i is %d\n",i);
            for(int j = 1; j <= pow(2,k); j++){
                printf("vec[%d] = vec[%d] + vec[%d]\n",i+j,i,i+j);
                vec[i+j] = vec[i] + vec[i+j];
            }
        }
        for(int i = 0; i < n; i++){
            printf("%d ", vec[i]);
        }
        printf("\n");
    }

    for(int i = 0; i < n; i++){
        printf("%d ", vec[i]);
    }
    printf("\n");
}
```

Paralelo

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

void print_vec(int vec[], int n);
```

```c
int main(){
    int comm_sz; /* Number of processes */
    int my_rank; /* My process rank */
    MPI_Init(NULL, NULL);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &comm_sz);
    MPI_Comm_rank(comm, &my_rank);

    int vec_i;
    int n = comm_sz;
    int vec[n];

    if (my_rank == 0){
        srand(time(NULL));
        for(int i = 0; i < n; i++){
            vec[i] = 1; //rand() % 10; // number between 0 and 9
        }
        print_vec(vec,n);
        MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
    }
    else {
        MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
    }

    for(int k = 0; k < log2(n); k++){
        for(int i = pow(2,k) - 1; i < pow(2,log2(n)); i = i + pow(2,k+1)){
            for(int j = 1; j <= pow(2,k); j++){
                if (my_rank == i){
                    MPI_Send(&vec_i, 1, MPI_INT, i+j, 0, comm);
                    printf("This is %d, Sending to %d\n", my_rank, i+j);
                } else if (my_rank == i + j){
                    int aux_vec_i = 0;
                    printf("This is %d, Receiving from %d\n",my_rank, i);
                    MPI_Recv(&aux_vec_i, 1, MPI_INT, i, 0, comm, MPI_STATUS_IGNORE);
                    vec_i += aux_vec_i;
                }
            }
        }
    }

    int vec_sum[n];
    if (my_rank == 0) {
       MPI_Gather(&vec_i, 1, MPI_INT, vec_sum, 1, MPI_INT, 0, comm);
       print_vec(vec_sum,n);
    } else {
        MPI_Gather(&vec_i, 1, MPI_INT, vec_sum, 1, MPI_INT, 0, comm);
    }

    MPI_Finalize();
    return 0;
}

void print_vec(int vec[], int n){
    for(int i = 0; i < n; i++){
        printf("%d ",vec[i]);
    }
    printf("\n");
}
```

d.

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

void print_vec(int vec[], int n);

int main(void) {
    int comm_sz; /* Number of processes */
    int my_rank; /* My process rank */
    MPI_Init(NULL, NULL);
    MPI_Comm comm = MPI_COMM_WORLD;
    MPI_Comm_size(comm, &comm_sz);
    MPI_Comm_rank(comm, &my_rank);

    int count = comm_sz;
    int vec[count];
    int sum[count];

    srand(my_rank+1);
    for(int i = 0; i < count; i++){
        vec[i] = rand() % 10; // number between 0 and 9
        sum[i] = 0;
    }

    MPI_Scan(vec, sum, count, MPI_INT, MPI_SUM, comm);

    if (my_rank != 0) {
        MPI_Send(vec, count, MPI_INT, 0, 0, comm);
        MPI_Send(sum, count, MPI_INT, 0, 0, comm);
    } else {
        printf("rank %d - [", my_rank );
        print_vec(vec,count);
        printf("] => [");
        print_vec(sum,count);
        printf("]\n");
        for (int q = 1; q < comm_sz; q++) {
            MPI_Recv(vec, count, MPI_INT, q, 0, comm,
                MPI_STATUS_IGNORE);
            MPI_Recv(sum, count, MPI_INT, q, 0, comm,
                MPI_STATUS_IGNORE);
            printf("rank %d - [", q);
            print_vec(vec,count);
            printf("] => [");
            print_vec(sum,count);
            printf("]\n");
        }
    }

    MPI_Finalize();
    return 0;
}

void print_vec(int vec[], int n){
    for(int i = 0; i < n; i++){
        printf("%d ",vec[i]);
    }
}
```

Output: [vec] => [sum]

```
rank 0 - [3 6 7 5 3 ] => [3 6 7 5 3 ]
rank 1 - [0 9 8 5 1 ] => [3 15 15 10 4 ]
rank 2 - [6 5 8 0 5 ] => [9 20 23 10 9 ]
rank 3 - [1 3 4 6 3 ] => [10 23 27 16 12 ]
rank 4 - [5 5 0 2 6 ] => [15 28 27 18 18 ]
```

2.11 Questão 3.12 *

An alternative to a butterfly-structured allreduce is a ring-pass structure. In a ring-pass, if there are p processes, each process q sends data to process q + 1, except that process p - 1 sends data to process 0. This is repeated until each process has the desired result. Thus, we can implement allreduce with the following code:

```
sum = temp val = my val;
  for (i = 1; i < p; i++) {
    MPI_Sendrecv_replace(&temp_val, 1, MPI_INT, dest,
    sendtag, source, recvtag, comm, &status);
    sum += temp val;
}
```

a. Write an MPI program that implements this algorithm for allreduce. How does its performance compare to the butterfly-structured allreduce?

b. Modify the MPI program you wrote in the first part so that it implements prefix sums.

2.12 Questão 3.13

`MPI_Scatter` and `MPI_Gather` have the limitation that each process must send or receive the same number of data items. When this is not the case, we must use the MPI functions `MPI_Gatherv` and `MPI_Scatterv`. Look at the man pages for these functions, and modify your vector sum, dot product program so that it can correctly handle the case when n isn't evenly divisible by `comm_sz`.

```
/*Esse codigo foi modificado de mpi_vector_add.c */

/* File:     mpi_vector_add.c
 *
 * Purpose:  Implement parallel vector addition using a block
 *           distribution of the vectors.  This version also
 *           illustrates the use of MPI_Scatter and MPI_Gather.
 *
 * Compile:  mpicc -g -Wall -o mpi_vector_add mpi_vector_add.c
 * Run:      mpiexec -n <comm_sz> ./vector_add
 *
 * Input:    The order of the vectors, n, and the vectors x and y
 * Output:   The sum vector z = x+y
 *
 * Notes:
 * 1.  The order of the vectors, n, should be evenly divisible
 *     by comm_sz
 * 2.  DEBUG compile flag.
 * 3.  This program does fairly extensive error checking.  When
 *     an error is detected, a message is printed and the processes
 *     quit.  Errors detected are incorrect values of the vector
 *     order (negative or not evenly divisible by comm_sz), and
 *     malloc failures.
 *
 * IPP:  Section 3.4.6 (pp. 109 and ff.)
 */

```

```
#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* rest, int my_rank, int comm_sz,
      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(int rest, double local_a[], int local_n, int n, char vec_name[],
      int my_rank, MPI_Comm comm);
void Read_scalar(double* scalar_p, int my_rank, int comm_sz, MPI_Comm comm);
void Print_vector(double local_b[], int local_n, int n, char title[],
      int my_rank, MPI_Comm comm);
void Parallel_vector_dotproduct(double local_x[], double local_y[],
      double local_z[], double* result, int local_n, int n, MPI_Comm comm);
void Parallel_scalar_mutiplication(double scalar, double local_x[],
    double local_z[], int local_n);


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

   Allocate_vectors(&local_x, &local_y, &local_z, local_n, comm);

   Read_vector(rest, local_x, local_n, n, "x", my_rank, comm);
   Print_vector(local_x, local_n, n, "x is", my_rank, comm);
   printf("Done ok" );
   Read_vector(rest, local_y, local_n, n, "y", my_rank, comm);
   Print_vector(local_y, local_n, n, "y is", my_rank, comm);

   Read_scalar(&scalar, my_rank, comm_sz, comm);

   Parallel_scalar_mutiplication(scalar, local_x, local_x, local_n);
   Print_vector(local_x, local_n, n, "now x is", my_rank, comm);
   Parallel_vector_dotproduct(local_x, local_y, local_z, &result,
      local_n,
                                        n, comm);

   if (my_rank == 0) {
      printf("The result of (scalar*x[]).y[] is %lf \n", result);
   }

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

   MPI_Finalize();

   return 0;
}  /* main */

/*-------------------------------------------------------------------
 * Function:  Check_for_error
 * Purpose:   Check whether any process has found an error.  If so,
 *            print message and terminate all processes.  Otherwise,
 *            continue execution.
 * In args:   local_ok:  0 if calling process has found an error, 1
 *               otherwise
 *            fname:     name of function calling Check_for_error
 *            message:   message to print if there's an error
 *            comm:      communicator containing processes calling
 *                       Check_for_error:  should be MPI_COMM_WORLD.
 *
 * Note:
 *    The communicator containing the processes calling
 Check_for_error
 *    should be MPI_COMM_WORLD.
 */
void Check_for_error(
      int        local_ok   /* in */,
      char       fname[]    /* in */,
      char       message[]   /* in */,
      MPI_Comm   comm        /* in */) {
   int ok;

   /* Pega o minimo do vetor
        Se o minimo for zero, aconteceu algum erro,
        caso contrario, todos os processos retornaram 1 e estao 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 */


/*-------------------------------------------------------------------
 * Function:   Read_n
 * Purpose:    Get the order of the vectors from stdin on proc 0 and
 *             broadcast to other processes.
 * In args:    my_rank:    process rank in communicator
 *             comm_sz:    number of processes in communicator
 *             comm:       communicator containing all the processes
 *                         calling Read_n
 * Out args:   n_p:        global value of n
 *             local_n_p:  local value of n = n/comm_sz
 *
 * Errors:     n should be positive and evenly divisible by comm_sz
 */
void Read_n(
      int*       n_p         /* out */,
      int*       local_n_p   /* out */,
      int*       rest        /* 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 ) local_ok = 0;
   Check_for_error(local_ok, fname,
         "n should be > 0. ", comm);
   *local_n_p = *n_p/comm_sz + 1;
   *rest = *n_p % comm_sz;
}  /* Read_n */


/*------------------------------------------------------------------
 * Function:  Allocate_vectors
 * Purpose:   Allocate storage for x, y, and z
 * In args:   local_n:  the size of the local vectors
 *            comm:     the communicator containing the calling
    processes
 * Out args:  local_x_pp, local_y_pp, local_z_pp:  pointers to memory
 *              blocks to be allocated for local vectors
 *
 * Errors:    One or more of the calls to malloc fails
 */
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
/*-------------------------------------------------------------------
 * Function:   Read_vector
 * Purpose:    Read a vector from stdin on process 0 and distribute
 *             among the processes using a block distribution.
 * In args:    local_n:  size of local vectors
 *             n:        size of global vector
 *             vec_name: name of vector being read (e.g., "x")
 *             my_rank:  calling process' rank in comm
 *             comm:     communicator containing calling processes
 * Out arg:    local_a:  local vector read
 *
 * Errors:     if the malloc on process 0 for temporary storage
 *             fails the program terminates
 *
 * Note:
 *    This function assumes a block distribution and the order
 *   of the vector evenly divisible by comm_sz.
 */
void Read_vector(
      int        rest         /* in  */,
      double     local_a[]    /* out */,
      int        local_n      /* in  */,
      int        n            /* in  */,
      char       vec_name[]   /* in  */,
      int        my_rank      /* in  */,
      MPI_Comm   comm         /* in  */) {

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

   int comm_sz;
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   int* sendcounts = malloc(sizeof(int)*comm_sz);
   int* displs = malloc(sizeof(int)*comm_sz);

   // Calculating send counts and displacement
   int sum = 0;
   int rem = rest;
    for (int i = 0; i < comm_sz; i++) {
        sendcounts[i] = n/comm_sz;
        if (rem > 0) {
            sendcounts[i]++;
            rem--;
        }

        displs[i] = sum;
        sum += sendcounts[i];
    }

    // print calculated send counts and displacements for each process
    if (0 == my_rank) {
       for (int i = 0; i < comm_sz; i++) {
           printf("sendcounts[%d] = %d\tdispls[%d] = %d\n", i, sendcounts[i], i, displs[i]);
       }
    }

    a = malloc(n*sizeof(double));
    if (a == NULL) local_ok = 0;
    Check_for_error(local_ok, fname, "Can't allocate temporary vector",
          comm);

    if (my_rank == 0) {
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]); // reads a double (long float)
    }

    // divide the data among processes as described by sendcounts and displs
    // MPI_Scatterv(&data, sendcounts, displs, MPI_CHAR, &rec_buf, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Scatterv(a, sendcounts, displs, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0, comm);

    // print what each process received
     printf("%d: ", my_rank);
     for (int i = 0; i < sendcounts[my_rank]; i++) {
         printf("%lf\t", local_a[i]);
     }
     printf("\n");

     free(a);
     free(sendcounts);
     free(displs);
}  /* Read_vector */
```

```c

/*-------------------------------------------------------------------
 * Function:  Read_scalar
 * Purpose:   Get the the scalar number from stdin on proc 0 and
 *            broadcast to other processes.
 * In args:   my_rank:    process rank in communicator
 *            comm_sz:    number of processes in communicator
 *            comm:       communicator containing all the processes
 *                        calling Read_n
 * Out args:  scalar_p:        global value of n
 *
 */
void Read_scalar(
      double*    scalar_p        /* out */,
      int        my_rank     /* in  */,
      int        comm_sz     /* in  */,
      MPI_Comm   comm        /* in  */) {

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

}  /* Read_scalar */

/*-------------------------------------------------------------------
 * Function:  Print_vector
 * Purpose:   Print a vector that has a block distribution to stdout
 * In args:   local_b:  local storage for vector to be printed
 *            local_n:  order of local vectors
 *            n:        order of global vector (local_n*comm_sz)
 *            title:    title to precede print out
 *            comm:     communicator containing processes calling
 *                      Print_vector
 *
 * Error:     if process 0 can't allocate temporary storage for
 *            the full vector, the program terminates.
 *
 * Note:
 *    Assumes order of vector is evenly divisible by the number of
 *    processes
 */
void Print_vector(
      double     local_b[]  /* in */,
      int         local_n    /* in */,
      int         n          /* in */,
      char        title[]    /* in */,
      int         my_rank    /* in */,
      MPI_Comm    comm       /* in */) {

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

   int comm_sz;
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   int* recvcounts = malloc(sizeof(int)*comm_sz);
   int* displs = malloc(sizeof(int)*comm_sz);

   // Calculating send counts and displacement
   int sum = 0;
   int rem = n%comm_sz;
   for (int i = 0; i < comm_sz; i++) {
       recvcounts[i] = n/comm_sz;
       if (rem > 0) {
           recvcounts[i]++;
           rem--;
       }

       displs[i] = sum;
       sum += recvcounts[i];
   }

   if (0 == my_rank) {
       for (int i = 0; i < comm_sz; i++) {
           printf("recvcounts[%d] = %d\tdispls[%d] = %d\n", i,
               recvcounts[i], i, displs[i]);
       }
   }


   b = malloc(n*sizeof(double));
   if (b == NULL) local_ok = 0;
   Check_for_error(local_ok, fname, "Can't allocate temporary vector
       ",
          comm);

  if (my_rank == 0) {
      MPI_Gatherv(local_b, recvcounts[my_rank], MPI_DOUBLE, b,
         recvcounts, displs,
           MPI_DOUBLE, 0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");

   } else {
      MPI_Gatherv(local_b, recvcounts[my_rank], MPI_DOUBLE, b,
         recvcounts, displs,
               MPI_DOUBLE, 0, comm);
   }

   free(b);
}  /* Print_vector */


/*-----------------------------------------------------------------
 * Function:   Parallel_vector_dotproduct
 * Purpose:    Add a vector that's been distributed among the
   processes
 * In args:    local_x:  local storage of one of the vectors being
   added
 *             local_y:  local storage for the second vector being
   added
 *             local_n:  the number of components in local_x, local_y,
 *                       and local_z
 * Out arg:    local_z:  local storage for the sum of the two vectors
 */
void Parallel_vector_dotproduct(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      double* result     /* out */,
      int     local_n    /* in  */,
      int     n          /* in  */,
      MPI_Comm  comm     /* in */) {
   int local_i;

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

  double local_total_z = 0.0;
```

```
    for (local_i = 0; local_i < local_n; local_i++){
        local_total_z += local_z[local_i];
    }

    int my_rank;
    MPI_Comm_rank(comm, &my_rank);
    printf("rank: %d > %lf\n", my_rank, local_total_z );

    MPI_Reduce(&local_total_z , result , 1, MPI_DOUBLE, MPI_SUM, 0, comm)
        ;

}  /* Parallel_vector_dotproduct */

/*---------------------------------------------------------------------
 * Function:  Parallel_scalar_mutiplication
 * Purpose:   Add a vector that's been distributed among the
     processes
 * In args:   scalar:   storage for the scalar value
 *            local_x:  local storage for the second vector being
     added
 *            local_n:  the number of components in local_x, local_y,
 *                      and local_z
 * Out arg:   local_z:  local storage for the scalar mutiplication of
 *                      the vector
 */
void Parallel_scalar_mutiplication(
      double   scalar     /* in  */,
      double   local_x[]  /* in  */,
      double   local_z[]  /* out */,
      int      local_n    /* in  */) {
   int local_i;

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

2.13 Questão 3.14

a. Write a serial C program that defines a two-dimensional array in the main function. Just use numeric constants for the dimensions: `int two d[3][4];`
Initialize the array in the main function. After the array is initialized, call a function that attempts to print the array. The prototype for the function should look something like this.
`void Print two d(int two d[][], int rows, int cols);`
After writing the function try to compile the program. Can you explain why it won't compile?

b. After consulting a C reference (e.g., Kernighan and Ritchie [29]), modify the program so that it will compile and run, but so that it still uses a two-dimensional C array.

O código abaixo não compila pois o compilador precisa saber as dimensões para fazer a aritmética de ponteiros corretamente. Isto é, se o compilador não tiver as dimensões do array quando ele é passado para uma função (e passado como ponteiro), a expressão array[x][y] não pode ser calculada, pois não há como sabe onde uma linha ou coluna começa ou termina.

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

void Print_two_d(int two_d[][], int rows, int cols);

int main(){
    int two_d[3][4] = {{1,1,1,1},
                        {2,2,2,2},
                        {3,3,3,3}};

   Print_two_d(two_d,3,4);
   return 0;
}

void Print_two_d(int two_d[][], int rows, int cols){
    int i, j;
    for(i = 0; i < rows; i++){
        for(j = 0; j < cols; j++){
            printf("%d\n", two_d[i][j]);
        }
    }
}
```

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

void Print_two_d(int two_d[][4], int rows, int cols);

int main(){
    int two_d[3][4] = {{1,1,1,1},
                        {2,2,2,2},
                        {3,3,3,3}};

   Print_two_d(two_d,3,4);
   return 0;
}

void Print_two_d(int two_d[][4], int rows, int cols){
    int i, j;
    for(i = 0; i < rows; i++){
        for(j = 0; j < cols; j++){
            printf("%d\n", two_d[i][j]);
        }
    }
}
```

2.14 Questão 3.16

Suppose comm_sz = 8 and the vector x = (0, 1, 2, ... , 15) has been distributed among the processes using a block distribution. Draw a diagram illustrating the steps in a butterfly implementation of allgather of x.

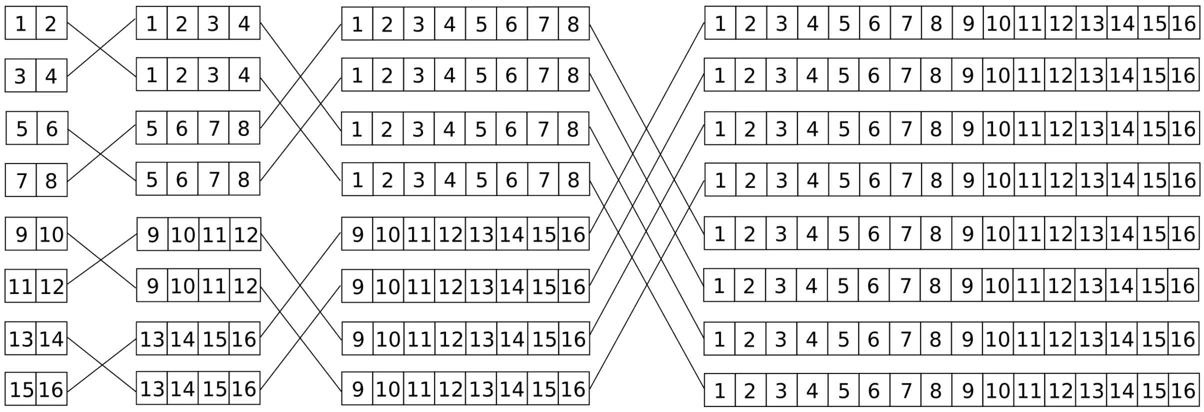


Figura 3: Allgather em comunicação butterfly

2.15 Questão 3.17

MPI_Type_contiguous can be used to build a derived datatype from a collection of contiguous elements in an array. Its syntax is

```
int MPI_Type_contiguous(
    int           count       /* in  */,
    MPI_Datatype  old_mpi_t   /* in  */,
    MPI_Datatype* new_mpi_t_p /* out */);
```

Modify the Read_vector and Print_vector functions so that they use an MPI datatype created by a call to MPI_Type_contiguous and a count argument of 1 in the calls to MPI_Scatter and MPI_Gather.

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

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

    MPI_Datatype CONTIGUOUS;
    MPI_Type_contiguous(local_n, MPI_DOUBLE, &CONTIGUOUS);
    MPI_Type_commit(&CONTIGUOUS);

    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]); // reads a double (long float)
        MPI_Scatter(a, 1, CONTIGUOUS, local_a, 1, CONTIGUOUS, 0, comm);
        free(a);
    } else {
        Check_for_error(local_ok, fname, "Can't allocate temporary
            vector",
                comm);
        MPI_Scatter(a, 1, CONTIGUOUS, local_a, 1, CONTIGUOUS, 0, comm);
    }
}  /* Read_vector */
```

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

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

    MPI_Datatype CONTIGUOUS;
    MPI_Type_contiguous(local_n, MPI_DOUBLE, &CONTIGUOUS);
    MPI_Type_commit(&CONTIGUOUS);

    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, 1, CONTIGUOUS, b, 1, CONTIGUOUS, 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, 1, CONTIGUOUS, b, 1, CONTIGUOUS, 0, comm);
    }
}  /* Print_vector */
```

2.16 Questão 3.19

`MPI_Type_indexed` can be used to build a derived datatype from arbitrary array elements. Its syntax is

```
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 */);
```

Unlike `MPI_Type_create_struct`, the displacements are measured in units of `old_mpi_t` - not bytes. Use `MPI_Type_indexed` to create a derived datatype that corresponds to the upper triangular part of a square matrix. For example, in the 4 x 4 matrix

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

the upper triangular part is the elements 0, 1, 2, 3, 5, 6, 7, 10, 11, 15. Process 0 should read in an n x n matrix as a one-dimensional array, create the derived datatype, and send the upper triangular part with a single call to `MPI_Send`. Process 1 should receive the upper triangular part with a single call to `MPI_Recv` and then print the data it received.

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

int main(int argc, char *argv[])
{
    int rank, size, i;
    MPI_Datatype type;
    int n = 4;

    int blocklen[n], displacement[n];
    for (int i = n, j = 0;
          i > 0 && j < n; i--, j++)
          { blocklen[j] = i;}

    for (int i = 0, j = 0;
          i < n, j < n; i+=5, j++)
          { displacement[j] = i;}

    int buffer[n*n];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2)
    {
        printf("Please run with 2 processes.\n");
        MPI_Finalize();
        return 1;
    }

    MPI_Type_indexed(n, blocklen, displacement, MPI_INT, &type);
    MPI_Type_commit(&type);

    if (rank == 0)
    {
        printf("Enter the matrix: ");
        fflush(stdout);
        for (int i = 0; i < n*n; i++){
            scanf("%d", &buffer[i]);
        }

        MPI_Send(buffer, 1, type, 1, 123, MPI_COMM_WORLD);
    }

    if (rank == 1)
    {
        for (i=0; i<n*n; i++)
            buffer[i] = -1;
        MPI_Recv(buffer, 1, type, 0, 123, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        for (i=0; i<n*n; i++)
            printf("buffer[%d] = %d\n", i, buffer[i]);
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

2.17 Questão 3.20

The functions MPI_Pack and MPI_Unpack provide an alternative to derived datatypes for grouping data. MPI_Pack copies the data to be sent, one block at a time, into a user-provided buffer. The buffer can then be sent and received. After the data is received, MPI_Unpack can be used to unpack it from the receive buffer. The syntax of MPI_Pack is

```
    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      */);
```

We could therefore pack the input data to the trapezoidal rule program with the following code:

```
    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);
```

The key is the position argument. When MPI_Pack is called, position should refer to the first available slot in pack buf . When MPI_Pack returns, it refers to the first available slot after the data that was just packed, so after process 0 executes this code, all the processes can callMPI_Bcast :

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

Note that the MPI datatype for a packed buffer is MPI_PACKED. Now the other processes can unpack the data using: 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     */);
```

This can be used by “reversing” the steps in `MPI_Pack` , that is, the data is unpacked one block at a time starting with position = 0. Write another Get input function for the trapezoidal rule program. This one should use `MPI_Pack` on process 0 and `MPI_Unpack` on the other processes.

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
        int* n_p) {
    //int dest;

    MPI_Comm comm = MPI_COMM_WORLD;
    char pack_buf[100];
    int position = 0;

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);

        printf("rank %d: %d\n", my_rank, *n_p);
        printf("rank %d: %lf\n", my_rank, *b_p);
        printf("rank %d: %lf\n", my_rank, *a_p);

        MPI_Pack(a_p, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
        MPI_Pack(b_p, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
        MPI_Pack(n_p, 1, MPI_INT, pack_buf, 100, &position, comm);

        MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, comm);
    }
    else { /* my_rank != 0 */
        MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, comm);

        MPI_Unpack(pack_buf, 100, &position, a_p, 1, MPI_DOUBLE, comm);
        MPI_Unpack(pack_buf, 100, &position, b_p, 1, MPI_DOUBLE, comm);
        MPI_Unpack(pack_buf, 100, &position, n_p, 1, MPI_INT, comm);
    }
} /* Get_input */
```

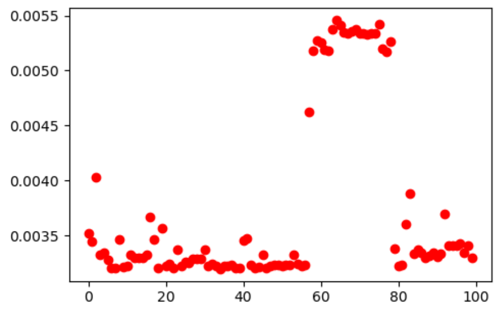
2.18 Questão 3.21

How does your system compare to ours? What run-times does your system get for matrix-vector multiplication? What kind of variability do you see in the times for a given value of `comm_sz` and n? Do the results tend to cluster around the minimum, the mean, or the median?

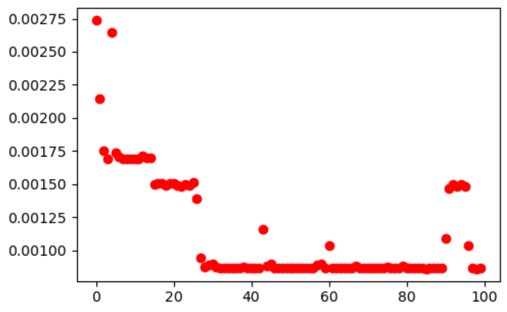
	Ordem da Matriz				
comm_sz	1024	2048	4096	8192	16384
1	3.7561	12.8853	52.0139	210.4686	846.9655
2	2.0700	10.2071	41.1858	107.3081	438.0811
4	1.7275	6.8219	25.0998	64.2229	245.5204
8	1.7007	4.1896	14.3932	56.5015	219.8544
16	2.1960	6.0016	20.8711	76.8699	236.1451

Tabela 2: Tempo de execução

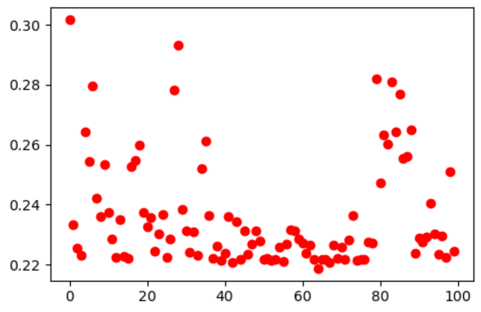
A medida que o comm_sz aumenta, é possível ver que o tempo de execução diminui. Contudo, a execução nem sempre tem um comportamento constante. Isso pode ser observado na Figura 4.



(a) `comm_sz` = 1, Ordem = 1024, (s)



(b) `comm_sz` = 4, Ordem = 1024, (s)



(c) `comm_sz` = 16, Ordem = 16384, (s)

Figura 4: Comportamento dos tempos de execução

2.19 Questão 3.22

Time our implementation of the trapezoidal rule that uses `MPI_Reduce`. How will you choose n, the number of trapezoids? How do the minimum times compare to the mean and median times? What are the speedups? What are the efficiencies? On the basis of the data you collected, would you say that the trapezoidal rule is scalable?

Recall that programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be strongly scalable. Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be weakly scalable.

A escolha do n é feita de forma que o algoritmo tome tempo de execução suficiente para uma medição segura e que ele possa ser dobrado a cada rodada. Em 100 rodadas de cada configuração, montamos as tabelas abaixo.

Em análise da Tabela 7 vemos que o algoritmo tem um momento de escalabilidade fraca entre p = 2 e p = 4, mas volta a cair em p = 8. Provavelmente, o overhead de comunicação ainda é bem alto para esses p's, mas deve ser superado em p's maiores.

	Ordem da Matriz			
comm_sz	1024	2048	4096	8192
1	1.58214537e-05	3.064394e-05	6.87956822e-05	0.000125017164
2	1.09386417e-05	1.88255289e-05	5.77306725e-05	6.33239711e-05
4	2.1030905e-05	2.27093698e-05	2.16841662e-05	3.60274319e-05
8	1.62243825e-05	3.75008588e-05	2.96688098e-05	2.9134753e-05

Tabela 3: Tempo de execução (s) (média)

	Ordem da Matriz			
comm_sz	1024	2048	4096	8192
1	1.597404e-05	3.004074e-05	6.890297e-05	0.0001249313
2	1.096725e-05	2.002716e-05	5.793571e-05	6.29425e-05
4	1.907349e-05	2.217293e-05	2.098083e-05	3.504753e-05
8	1.597404e-05	2.908707e-05	2.598763e-05	2.598763e-05

Tabela 4: Tempo de execução (mediana)

	Ordem da Matriz			
comm_sz	1024	2048	4096	8192
1	1.478195e-05	2.884865e-05	5.984306e-05	0.0001239777
2	1.001358e-05	1.478195e-05	5.698204e-05	6.198883e-05
4	1.788139e-05	2.193451e-05	2.098083e-05	3.385544e-05
8	1.28746e-05	2.69413e-05	2.31266e-05	2.479553e-05

Tabela 5: Tempo de execução (s) (mínimo)

	Ordem da Matriz			
comm_sz	1024	2048	4096	8192
1	1.0	1.0	1.0	1.0
2	1.44638193058	1.62778640445	1.191666045465	1.97424706361
4	0.752295429036	1.34939631834	3.17262289753	3.47005482786
8	0.975165230479	0.81715301944	2.31878806948	4.29099790206

Tabela 6: Tempo de Speedup da média

	Ordem da Matriz			
comm_sz	1024	2048	4096	8192
1	1.0	1.0	1.0	1.0
2	0.723190965291	0.813893202225	0.595833022732	0.987123531803
4	0.188073857259	0.337349079586	0.793155724383	0.867513706965
8	0.12189565381	0.10214412743	0.289848508685	0.536374737757

Tabela 7: Tempo de eficiência da média

2.20 Questão 3.23

Although we don't know the internals of the implementation of `MPI_Reduce`, we might guess that it uses a structure similar to the binary tree we discussed. If this is the case, we would expect that its run-time would grow roughly at the rate of $log_2(p)$, since there are roughly $log_2(p)$ levels in the tree. (Here, p = `comm_sz`.) Since the run-time of the serial trapezoidal rule is roughly proportional to n, the number of trapezoids, and the parallel trapezoidal rule simply applies the serial rule to n/p trapezoids on each process, with our assumption about `MPI_Reduce`, we get a formula for the overall run-time of the parallel trapezoidal rule that looks like

$$T_{parallel}(n, p) \approx a \times \frac{n}{p} + b \cdot \log_2(p)$$

for some constants a and b.

a. Use the formula, the times you've taken in Exercise 3.22, and your favorite R program for doing mathematical calculations (e.g., MATLAB) to get a least-squares estimate of the values of a and b.

b. Comment on the quality of the predicted run-times using the formula and the values for a and b computed in part (a).