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.

```
1  #include <stdio.h>
2  #include <mpi.h>
3  #include <string.h> /* For strlen */
4
5  const int MAX_STRING = 100;
6  int main(void) {
7      char phrase[MAX_STRING];
8      int my_rank, comm_sz;
9
10     MPI_Init(NULL, NULL);
11     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
12     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
13
14     if (my_rank != 0) {
15         sprintf(phrase, "Proc %d of %d > Does anyone have a toothpick
               ?", my_rank, comm_sz);
16         MPI_Send(phrase, strlen(phrase)+1, MPI_CHAR, 0, 0,
              MPI_COMM_WORLD);
17     } else {
18         printf("Proc %d of %d > Does anyone have a toothpick?\n",
              my_rank, comm_sz);
19         for (int q = 1; q < comm_sz; q++) {
20             MPI_Recv(phrase, MAX_STRING, MPI_CHAR, q, 0,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
21             printf("%s\n",phrase);
22         }
23     }
24
25     MPI_Finalize();
26     return 0;
27 }  /* 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.

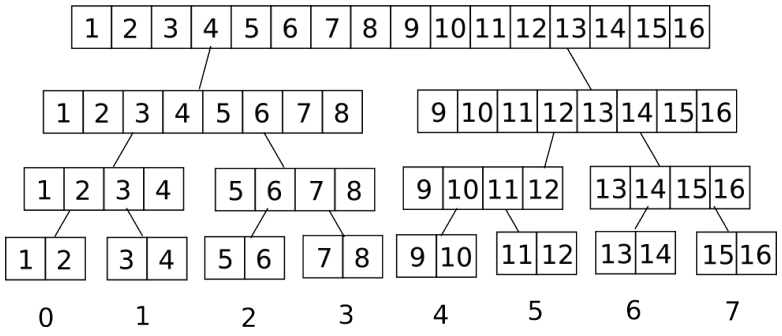

Figura 1: Scatter em comunição baseada em árvore

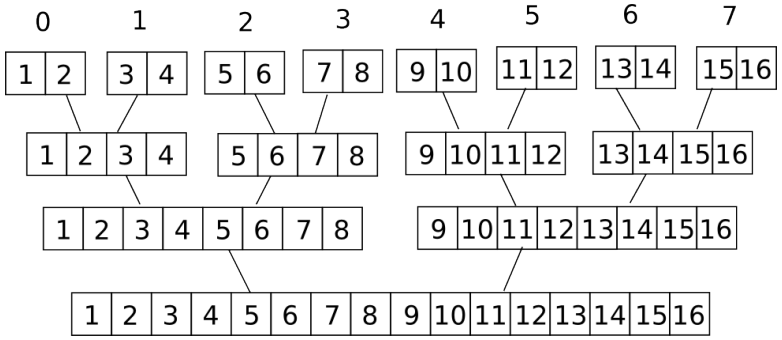

Figura 2: Gather em comunição baseada em árvore

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` .

```
/*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);
```

```
38 void Read_vector(double local_a[], int local_n, int n, char vec_name
       [],
39         int my_rank, MPI_Comm comm);
40 void Read_scalar(double* scalar_p, int my_rank, int comm_sz, MPI_Comm
       comm);
41 void Print_vector(double local_b[], int local_n, int n, char title[],
42         int my_rank, MPI_Comm comm);
43 void Parallel_vector_dotproduct(double local_x[], double local_y[],
44         double local_z[], double* result, int local_n, int n, MPI_Comm
              comm);
45 void Parallel_scalar_mutiplication(double scalar, double local_x[],
46       double local_z[], int local_n);
47
48
49 /*-------------------------------------------------------------------
       */
50 int main(void) {
51    int n, local_n;
52    int comm_sz, my_rank;
53    double *local_x, *local_y, *local_z;
54    double scalar, result;
55    MPI_Comm comm;
56
57    MPI_Init(NULL, NULL);
58    comm = MPI_COMM_WORLD;
59    MPI_Comm_size(comm, &comm_sz);
60    MPI_Comm_rank(comm, &my_rank);
61
62    Read_n(&n, &local_n, my_rank, comm_sz, comm);
63
64    Allocate_vectors(&local_x, &local_y, &local_z, local_n, comm);
65
66    Read_vector(local_x, local_n, n, "x", my_rank, comm);
67    Print_vector(local_x, local_n, n, "x is", my_rank, comm);
68    Read_vector(local_y, local_n, n, "y", my_rank, comm);
69    Print_vector(local_y, local_n, n, "y is", my_rank, comm);
70
71    Read_scalar(&scalar, my_rank, comm_sz, comm);
72
73    Parallel_scalar_mutiplication(scalar, local_x, local_x, local_n);
74    Print_vector(local_x, local_n, n, "now x is", my_rank, comm);
75    Parallel_vector_dotproduct(local_x, local_y, local_z, &result,
          local_n,
76                               n, comm);
77
```

```
78      if (my_rank == 0) {
79          printf("The result of (scalar*x[]).y[] is %lf \n", result);
80      }
81
82      free(local_x);
83      free(local_y);
84      free(local_z);
85
86      MPI_Finalize();
87
88      return 0;
89  }   /* main */
90
91  /*--------------------------------------------------------------------
92   * Function:  Check_for_error
93   * Purpose:   Check whether any process has found an error.  If so,
94   *            print message and terminate all processes.  Otherwise,
95   *            continue execution.
96   * In args:   local_ok:  0 if calling process has found an error, 1
97   *               otherwise
98   *            fname:     name of function calling Check_for_error
99   *            message:   message to print if there's an error
100  *            comm:      communicator containing processes calling
101  *                       Check_for_error:  should be MPI_COMM_WORLD.
102  *
103  * Note:
104  *    The communicator containing the processes calling
     Check_for_error
105  *    should be MPI_COMM_WORLD.
106  */
107 void Check_for_error(
108       int         local_ok   /* in */,
109       char        fname[]    /* in */,
110       char        message[]  /* in */,
111       MPI_Comm    comm       /* in */) {
112     int ok;
113
114     /* Pega o minimo do vetor
115          Se o minimo for zero, aconteceu algum erro,
116          caso contrario, todos os processos retornaram 1 e estao ok
117     */
118     MPI_Allreduce(&local_ok, &ok, 1, MPI_INT, MPI_MIN, comm);
119     if (ok == 0) {
120        int my_rank;
121        MPI_Comm_rank(comm, &my_rank);
```

```
122         if (my_rank == 0) {
123             fprintf(stderr, "Proc %d > In %s, %s\n", my_rank, fname,
124                   message);
125             fflush(stderr);
126         }
127         MPI_Finalize();
128         exit(-1);
129     }
130 }  /* Check_for_error */
131
132
133 /*---------------------------------------------------------------------
134  * Function:  Read_n
135  * Purpose:   Get the order of the vectors from stdin on proc 0 and
136  *            broadcast to other processes.
137  * In args:   my_rank:    process rank in communicator
138  *            comm_sz:    number of processes in communicator
139  *            comm:       communicator containing all the processes
140  *                        calling Read_n
141  * Out args:  n_p:        global value of n
142  *            local_n_p:  local value of n = n/comm_sz
143  *
144  * Errors:    n should be positive and evenly divisible by comm_sz
145  */
146 void Read_n(
147         int*       n_p         /* out */,
148         int*       local_n_p   /* out */,
149         int        my_rank     /* in  */,
150         int        comm_sz     /* in  */,
151         MPI_Comm   comm        /* in  */) {
152     int local_ok = 1;
153     char *fname = "Read_n";
154
155     if (my_rank == 0) {
156         printf("What's the order of the vectors?\n");
157         scanf("%d", n_p);
158     }
159     MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
160     if (*n_p <= 0 || *n_p % comm_sz != 0) local_ok = 0;
161     Check_for_error(local_ok, fname,
162           "n should be > 0 and evenly divisible by comm_sz", comm);
163     *local_n_p = *n_p/comm_sz;
164 }  /* Read_n */
165
166
```

```
167 /*---------------------------------------------------------------------
168  * Function:  Allocate_vectors
169  * Purpose:   Allocate storage for x, y, and z
170  * In args:   local_n:  the size of the local vectors
171  *            comm:     the communicator containing the calling
     processes
172  * Out args:  local_x_pp, local_y_pp, local_z_pp:  pointers to memory
173  *               blocks to be allocated for local vectors
174  *
175  * Errors:    One or more of the calls to malloc fails
176  */
177 void Allocate_vectors(
178       double**   local_x_pp  /* out */,
179       double**   local_y_pp  /* out */,
180       double**   local_z_pp  /* out */,
181       int        local_n     /* in  */,
182       MPI_Comm   comm        /* in  */) {
183    int local_ok = 1;
184    char* fname = "Allocate_vectors";
185
186    *local_x_pp = malloc(local_n*sizeof(double));
187    *local_y_pp = malloc(local_n*sizeof(double));
188    *local_z_pp = malloc(local_n*sizeof(double));
189
190    if (*local_x_pp == NULL || *local_y_pp == NULL ||
191        *local_z_pp == NULL) local_ok = 0;
192    Check_for_error(local_ok, fname, "Can't allocate local vector(s)",
193          comm);
194 }  /* Allocate_vectors */
195
196
197 /*---------------------------------------------------------------------
198  * Function:   Read_vector
199  * Purpose:    Read a vector from stdin on process 0 and distribute
200  *             among the processes using a block distribution.
201  * In args:    local_n:  size of local vectors
202  *             n:        size of global vector
203  *             vec_name: name of vector being read (e.g., "x")
204  *             my_rank:  calling process' rank in comm
205  *             comm:     communicator containing calling processes
206  * Out arg:    local_a:  local vector read
207  *
208  * Errors:     if the malloc on process 0 for temporary storage
209  *             fails the program terminates
210  *
```

```
211  * Note:
212  *    This function assumes a block distribution and the order
213  *    of the vector evenly divisible by comm_sz.
214  */
215 void Read_vector(
216       double     local_a[]   /* out */,
217       int        local_n     /* in  */,
218       int        n           /* in  */,
219       char       vec_name[]  /* in  */,
220       int        my_rank     /* in  */,
221       MPI_Comm   comm        /* in  */) {
222
223    double* a = NULL;
224    int i;
225    int local_ok = 1;
226    char* fname = "Read_vector";
227
228    if (my_rank == 0) {
229       a = malloc(n*sizeof(double));
230       if (a == NULL) local_ok = 0;
231       Check_for_error(local_ok, fname, "Can't allocate temporary
              vector",
232             comm);
233       printf("Enter the vector %s\n", vec_name);
234       for (i = 0; i < n; i++)
235          scanf("%lf", &a[i]); // reads a double (long float)
236       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
             MPI_DOUBLE, 0,
237             comm);
238       free(a);
239    } else {
240       Check_for_error(local_ok, fname, "Can't allocate temporary
              vector",
241             comm);
242       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
             MPI_DOUBLE, 0,
243             comm);
244    }
245 }  /* Read_vector */
246
247 /*--------------------------------------------------------------------
248  * Function:  Read_scalar
249  * Purpose:   Get the the scalar number from stdin on proc 0 and
250  *            broadcast to other processes.
251  * In args:   my_rank:    process rank in communicator
```

```
252  *            comm_sz:    number of processes in communicator
253  *            comm:       communicator containing all the processes
254  *                        calling Read_n
255  * Out args:  scalar_p:        global value of n
256  *
257  */
258 void Read_scalar(
259       double*    scalar_p        /* out */,
260       int        my_rank    /* in  */,
261       int        comm_sz    /* in  */,
262       MPI_Comm   comm       /* in  */) {
263
264    if (my_rank == 0) {
265       printf("What's the scalar value?\n");
266       scanf("%lf", scalar_p); // reads double
267    }
268    MPI_Bcast(scalar_p, 1, MPI_DOUBLE, 0, comm);
269
270 }  /* Read_scalar */
271
272 /*--------------------------------------------------------------------
273  * Function:  Print_vector
274  * Purpose:   Print a vector that has a block distribution to stdout
275  * In args:   local_b:  local storage for vector to be printed
276  *            local_n:  order of local vectors
277  *            n:        order of global vector (local_n*comm_sz)
278  *            title:    title to precede print out
279  *            comm:     communicator containing processes calling
280  *                      Print_vector
281  *
282  * Error:     if process 0 can't allocate temporary storage for
283  *            the full vector, the program terminates.
284  *
285  * Note:
286  *    Assumes order of vector is evenly divisible by the number of
287  *    processes
288  */
289 void Print_vector(
290       double     local_b[]  /* in */,
291       int        local_n    /* in */,
292       int        n          /* in */,
293       char       title[]    /* in */,
294       int        my_rank    /* in */,
295       MPI_Comm   comm       /* in */) {
296
```

```
297     double* b = NULL;
298     int i;
299     int local_ok = 1;
300     char* fname = "Print_vector";
301
302     if (my_rank == 0) {
303        b = malloc(n*sizeof(double));
304        if (b == NULL) local_ok = 0;
305        Check_for_error(local_ok, fname, "Can't allocate temporary
               vector",
306              comm);
307        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE
               ,
308              0, comm);
309        printf("%s\n", title);
310        for (i = 0; i < n; i++)
311           printf("%f ", b[i]);
312        printf("\n");
313        free(b);
314     } else {
315        Check_for_error(local_ok, fname, "Can't allocate temporary
               vector",
316              comm);
317        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE
               , 0,
318              comm);
319     }
320  }  /* Print_vector */
321
322
323  /*--------------------------------------------------------------------
324   * Function:  Parallel_vector_dotproduct
325   * Purpose:   Add a vector that's been distributed among the
        processes
326   * In args:   local_x:  local storage of one of the vectors being
        added
327   *            local_y:  local storage for the second vector being
        added
328   *            local_n:  the number of components in local_x, local_y,
329   *                      and local_z
330   * Out arg:   local_z:  local storage for the sum of the two vectors
331   */
332  void Parallel_vector_dotproduct(
333        double  local_x[]  /* in  */,
334        double  local_y[]  /* in  */,
```

```
335         double  local_z[]  /* out */,
336         double* result     /* out */,
337         int     local_n    /* in  */,
338         int     n          /* in  */,
339         MPI_Comm  comm     /* in */) {
340     int local_i;
341
342     for (local_i = 0; local_i < local_n; local_i++){
343         local_z[local_i] = local_x[local_i] + local_y[local_i];
344     }
345
346     double local_total_z = 0.0;
347     for (local_i = 0; local_i < local_n; local_i++){
348         local_total_z += local_z[local_i];
349     }
350
351     MPI_Reduce(&local_total_z, result, 1, MPI_DOUBLE, MPI_SUM, 0, comm)
        ;
352
353 }  /* Parallel_vector_dotproduct */
354
355 /*--------------------------------------------------------------------
356  * Function:  Parallel_scalar_mutiplication
357  * Purpose:   Add a vector that's been distributed among the
       processes
358  * In args:   scalar:  storage for the scalar value
359  *            local_x:  local storage for the second vector being
       added
360  *            local_n:  the number of components in local_x, local_y,
361  *                      and local_z
362  * Out arg:   local_z:  local storage for the scalar mutiplication of
363  *                      the vector
364  */
365 void Parallel_scalar_mutiplication(
366         double  scalar     /* in  */,
367         double  local_x[]  /* in  */,
368         double  local_z[]  /* out */,
369         int     local_n    /* in  */) {
370     int local_i;
371
372     for (local_i = 0; local_i < local_n; local_i++)
373         local_z[local_i] = local_x[local_i] * scalar;
374 }  /* 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:

```
1 int MPI_Scan(
2       void*          sendbuf_p   /* in  */,
3       void*          recvbuf_p   /* out */,
4       int            count       /* in  */,
5       MPI_Datatype   datatype    /* in  */,
6       MPI_Op         op          /* in  */,
7       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.

```
1 #include <stdio.h>
2 #include <time.h>
3 #include <stdlib.h>
4
5 int main(){
6     int n = 15;
```

```
7       srand(time(NULL));
8
9       int vec[n];
10      int prefix_sum[n];
11
12      for(int i = 0; i < n; i++){
13          vec[i] = rand() % 10; // number between 0 and 9
14          printf("%d ", vec[i]);
15      }
16      printf("\n");
17
18      for(int i = 0; i < n; i++){
19          prefix_sum[i] = 0;
20          for(int j = 0; j <= i; j++){
21              prefix_sum[i] += vec[j];
22          }
23          printf("%d ", prefix_sum[i]);
24      }
25  }
```

b.

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4  #include <mpi.h>
5
6  int main(){
7      int comm_sz; /* Number of processes */
8      int my_rank; /* My process rank */
9      MPI_Init(NULL, NULL);
10     MPI_Comm comm = MPI_COMM_WORLD;
11     MPI_Comm_size(comm, &comm_sz);
12     MPI_Comm_rank(comm, &my_rank);
13
14     int vec_i;
15     int prefix_sum = 0;
16     int n = comm_sz;
17     int vec[n];
18
19     if (my_rank == 0){
20         srand(time(NULL));
21         for(int i = 0; i < n; i++){
22             vec[i] = rand() % 10; // number between 0 and 9
23             printf("%d ", vec[i]);
24         }
```

```
25          printf("\n");
26
27          MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
28      }
29      else {
30          MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
31      }
32
33      // envia para todos os processo maiores do que ele mesmo
34      for(int i = my_rank; i < comm_sz; i++){
35          MPI_Send(&vec_i, 1, MPI_INT, i, 0, comm);
36      }
37
38      // recebe de todos os processos menores que ele mesmo
39      prefix_sum += vec_i;
40      for(int i = 0; i < my_rank; i++){
41          int aux_vec_i = 0;
42          MPI_Recv(&aux_vec_i, 1, MPI_INT, i, 0, comm,
                MPI_STATUS_IGNORE);
43          prefix_sum += aux_vec_i;
44      }
45
46
47      // Imprime o resultado na tela
48      if (my_rank != 0) {
49          MPI_Send(&prefix_sum, 1, MPI_INT, 0, 1, comm);
50      } else {
51          printf("%d ", vec_i);
52          for (int q = 1; q < comm_sz; q++) {
53              int aux;
54              MPI_Recv(&aux, 1, MPI_INT, q, 1, comm, MPI_STATUS_IGNORE)
                    ;
55              printf("%d ", aux);
56          }
57          printf("\n");
58      }
59
60
61      MPI_Finalize();
62      return 0;
63  }
```

c. **Serial**

```
1  #include <stdio.h>
2  #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);
```

```
8  int main(){
9      int comm_sz; /* Number of processes */
10     int my_rank; /* My process rank */
11     MPI_Init(NULL, NULL);
12     MPI_Comm comm = MPI_COMM_WORLD;
13     MPI_Comm_size(comm, &comm_sz);
14     MPI_Comm_rank(comm, &my_rank);
15
16     int vec_i;
17     int n = comm_sz;
18     int vec[n];
19
20     if (my_rank == 0){
21         srand(time(NULL));
22         for(int i = 0; i < n; i++){
23             vec[i] = 1; //rand() % 10; // number between 0 and 9
24         }
25         print_vec(vec,n);
26         MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
27     }
28     else {
29         MPI_Scatter(vec, 1, MPI_INT, &vec_i, 1, MPI_INT, 0, comm);
30     }
31
32     for(int k = 0; k < log2(n); k++){
33         for(int i = pow(2,k) - 1; i < pow(2,log2(n)); i = i + pow(2,k
               +1)){
34             for(int j = 1; j <= pow(2,k); j++){
35                 if (my_rank == i){
36                     MPI_Send(&vec_i, 1, MPI_INT, i+j, 0, comm);
37                     printf("This is %d, Sending to %d\n", my_rank, i+
                           j);
38                 } else if (my_rank == i + j){
39                     int aux_vec_i = 0;
40                     printf("This is %d, Receiving from %d\n",my_rank,
                           i);
41                     MPI_Recv(&aux_vec_i, 1, MPI_INT, i, 0, comm,
                           MPI_STATUS_IGNORE);
42                     vec_i += aux_vec_i;
43                 }
44             }
45         }
46     }
47
48     int vec_sum[n];
```

```
49      if (my_rank == 0) {
50          MPI_Gather(&vec_i, 1, MPI_INT, vec_sum, 1, MPI_INT, 0, comm);
51          print_vec(vec_sum,n);
52      } else {
53           MPI_Gather(&vec_i, 1, MPI_INT, vec_sum, 1, MPI_INT, 0, comm);
54      }
55
56      MPI_Finalize();
57      return 0;
58  }
59
60  void print_vec(int vec[], int n){
61      for(int i = 0; i < n; i++){
62          printf("%d ",vec[i]);
63      }
64      printf("\n");
65  }
```

d.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <time.h>
4   #include <mpi.h>
5
6   void print_vec(int vec[], int n);
7
8   int main(void) {
9       int comm_sz; /* Number of processes */
10      int my_rank; /* My process rank */
11      MPI_Init(NULL, NULL);
12      MPI_Comm comm = MPI_COMM_WORLD;
13      MPI_Comm_size(comm, &comm_sz);
14      MPI_Comm_rank(comm, &my_rank);
15
16      int count = comm_sz;
17      int vec[count];
18      int sum[count];
19
20      srand(my_rank+1);
21      for(int i = 0; i < count; i++){
22          vec[i] = rand() % 10; // number between 0 and 9
23          sum[i] = 0;
24      }
25
26      MPI_Scan(vec, sum, count, MPI_INT, MPI_SUM, comm);
```

```
27
28      if (my_rank != 0) {
29          MPI_Send(vec, count, MPI_INT, 0, 0, comm);
30          MPI_Send(sum, count, MPI_INT, 0, 0, comm);
31      } else {
32          printf("rank %d - [", my_rank );
33          print_vec(vec,count);
34          printf("] => [");
35          print_vec(sum,count);
36          printf("]\n");
37          for (int q = 1; q < comm_sz; q++) {
38              MPI_Recv(vec, count, MPI_INT, q, 0, comm,
                    MPI_STATUS_IGNORE);
39              MPI_Recv(sum, count, MPI_INT, q, 0, comm,
                    MPI_STATUS_IGNORE);
40              printf("rank %d - [", q);
41              print_vec(vec,count);
42              printf("] => [");
43              print_vec(sum,count);
44              printf("]\n");
45          }
46      }
47
48      MPI_Finalize();
49      return 0;
50  }
51
52  void print_vec(int vec[], int n){
53      for(int i = 0; i < n; i++){
54          printf("%d ",vec[i]);
55      }
56  }
```

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:

```
1 sum = temp_val = my_val;
2   for (i = 1; i < p; i++) {
3     MPI_Sendrecv_replace(&temp_val, 1, MPI_INT, dest,
4     sendtag, source, recvtag, comm, &status);
5     sum += temp_val;
6 }
```

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`.