Universidade Federal do Rural do Semi-Árido
Centro Multidisciplinar de Pau dos Ferros
Departamento de Engenharias e Tecnologia
PEX1272 - Programação Concorrente e Distribuída
Professor: Ítalo Assis

Lista de Exercícios

QUESTÃO 36 - RESPONDIDA
QUESTÃO 37 - RESPONDIDA
QUESTÃO 38 - RESPONDIDA
QUESTÃO 39 - RESPONDIDA
QUESTÃO 40 - RESPONDIDA
QUESTÃO 41 - RESPONDIDA
QUESTÃO 42 - RESPONDIDA
QUESTÃO 43 - RESPONDIDA
QUESTÃO 44 - RESPONDIDA
QUESTÃO 45 - RESPONDIDA
QUESTÃO 46 - RESPONDIDA
QUESTÃO 47 - RESPONDIDA
QUESTÃO 48 - RESPONDIDA
QUESTÃO 49 - RESPONDIDA
QUESTÃO 50 - RESPONDIDA

4 Programação de memória distribuída com MPI

36. Modifique a regra trapezoidal para que ela estime corretamente a integral mesmo que comm_sz não divida *n* uniformemente Você ainda pode assumir que *n* ≥ *comm_sz*.

Código:

```
Get_input(my_rank, comm_sz, &a, &b, &n);
rest = n % comm_sz; // trapezios excedentes
h = (b - a) / n; /* h is the same for all processes */
local_n = n / comm_sz + (my_rank < rest); /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + (my_rank * local_n + (my_rank < rest ? my_rank : rest)) * h;
local_b = local_a + local_n * h;
local_int = Trap(local_a, local_b, local_n, h);
```

A variável *rest* calcula os trapézios excedentes, distribuímos eles para os primeiros ranks, conforme mostrado no cálculo da variável *local_n.* Para ajustar o descolamento dos trapézios adicionamos um ternário no cálculo de *local_a*.

37. Modifique o programa que apenas imprime uma linha de saída de cada processo (mpi_output.c) para que a saída seja impressa na ordem de classificação do processo: processe a saída de 0 primeiro, depois processe 1 e assim por diante.

Código:

```
int main(void) {
 int my_rank, comm_sz;
 char *message = (char *)malloc(MESSAGE_SZ * sizeof(char));
```

```
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(message, "Proc %d of %d > Does anyone have a toothpick?\n", my_rank,
          comm_sz);

  MPI_Send(message, MESSAGE_SZ, 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 i = 1; i < comm_sz; i++) {
    MPI_Recv(message, MESSAGE_SZ, MPI_CHAR, i, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("%s", message);
  }
}

free(message);
MPI_Finalize();
return 0;
```

Para isso criamos um buffer que irá guardar a saída de cada processo, cada processo irá enviar a sua saída para o rank 0, o rank 0 por sua vez imprime sua saída e processa em ordem crescente a saída de cada rank.

38. Suponha que um programa seja executado com comm_sz processos e que *x* seja um vetor com *n* componentes. Como os componentes de *x* seriam distribuídos entre os processos em um programa que usasse uma distribuição:

(a) em bloco?

Cada processo recebe um bloco de n/size iterações consecutivas, no caso de n não ser divisível por size, os primeiros processos recebem mais iterações. Podemos calcular esse bloco como n/comm_sz + (rank<n%comm_sz). Para compensar a divisão uniforme podemos calcular o deslocamento das iterações por processo como:

rank * (n / comm_sz) + (rank < n%comm_sz ? rank : comm_sz).

(b) cíclica?

A distribuição de iterações segue um esquema round-robin, onde cada processo recebe a iteração correspondente ao seu rank, com um deslocamento igual ao tamanho do comunicador comm_sz. Por exemplo:

Para o processo de rank x: x, x + comm_sz, x + 2*comm_sz,...

(c) bloco-cíclica com tamanho de bloco *b*?

A distribuição bloco-cíclica com tamanho de bloco b distribui os elementos de forma equilibrada entre os comm_sz processos em um esquema round-robin, onde cada processo recebe blocos consecutivos com deslocamento comm_sz*b. Exemplo:

O processo de rank x recebe os índices (x+comm_sz · i)*b+k onde k pertence a {0,b-1} e i é o iterador global.

Suas respostas devem ser genéricas para que possam ser usadas independentemente dos valores de comm_sz e *n*. Ao mesmo tempo, as distribuições apresentadas nas respostas devem ser "justas", de modo que, se *q* e *r* forem dois processos quaisquer, a diferença entre o número de componentes atribuídos a *q* e a *r* seja a menor possível.

39. Escreva um programa MPI que receba do usuário dois vetores e um escalar, todos lidos pelo processo 0 e distribuídos entre os processos. O primeiro vetor deve ser multiplicado pelo escalar. Para o segundo vetor, deve-se calcular o produto interno. Os resultados calculados devem ser coletados no processo 0, que os imprime. Você pode assumir que *n*, a ordem dos vetores, é divisível por comm_sz.

Saída:

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/listaMPI$ mpirun ./39.o
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Entre com o tamanho dos vetores:
5
Entre com o escalar:
2
Entre com 5 valores:
1 3 4 6 8

Entre com 5 valores:
1 2 3 4 5

Vetor 1 passado:
1 3 4 6 8
Vetor 2 passado:
1 2 3 4 5
Produto interno do vetor 2: 26520
Vetor 1 multiplicado pelo escalar 2:
2 6 8 12 16
```

Código:

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

void get_input(int *n, int *local_n, int rank, int size, int *escalar);
void block_compute(int size, int *send_counts, int *desl, int n);
void read_vector(int *vetor, int *local_vetor, int n, int *local_n, int rank,
                 int size, int *send_counts, int *desl);
```

```c
int main(int argc, char *argv[]) {
 int *vetor1, *vetor2; // vetores originais
 int rank, size;
 int n, local_n, escalar;
 int *local_vetor1, *local_vetor2;
 long long int global_produto = 1;
 int *send_counts;
 int *desl;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 send_counts = (int *)malloc(size * sizeof(int));
 desl = (int *)malloc(size * sizeof(int));

 get_input(&n, &local_n, rank, size, &escalar);
 local_vetor1 = (int *)malloc(local_n * sizeof(int));
 local_vetor2 = (int *)malloc(local_n * sizeof(int));

 vetor1 = (int *)malloc(n * sizeof(int));
 vetor2 = (int *)malloc(n * sizeof(int));

 block_compute(size, send_counts, desl, n);

 read_vector(vetor1, local_vetor1, n, &local_n, rank, size, send_counts, desl);
 read_vector(vetor2, local_vetor2, n, &local_n, rank, size, send_counts, desl);

 int local_produto = 1;
 for (int k = 0; k < local_n; k++) {
   local_vetor1[k] *= escalar;
   local_produto += local_vetor2[k] * local_vetor2[k];
 }

 if (rank == 0) {
   printf("Vetor 1 passado:\n");
   print_vetor(n, vetor1);

   printf("Vetor 2 passado:\n");
   print_vetor(n, vetor2);
 }

 MPI_Reduce(&local_produto, &global_produto, 1, MPI_INT, MPI_PROD, 0,
            MPI_COMM_WORLD);

 MPI_Gatherv(local_vetor1, local_n, MPI_INT, vetor1, send_counts, desl,
             MPI_INT, 0, MPI_COMM_WORLD);

 if (rank == 0) {
   printf("Produto interno do vetor 2: %lld\n", global_produto);
```

```
   printf("Vetor 1 multiplicado pelo escalar %d:\n", escalar);
   print_vetor(n, vetor1);
 }

 free(local_vetor1);
 free(local_vetor2);
 free(send_counts);
 free(desl);
 if (rank == 0) {
   free(vetor1);
   free(vetor2);
 }
 MPI_Finalize();

 return 0;
}

void get_input(int *n, int *local_n, int rank, int size, int *escalar) {
 if (rank == 0) {
   printf("Entre com o tamanho dos vetores:\n");
   scanf("%d", n);

   printf("Entre com o escalar: \n");
   scanf("%d", escalar);
 }

 MPI_Bcast(n, 1, MPI_INT, 0, MPI_COMM_WORLD);
 MPI_Bcast(escalar, 1, MPI_INT, 0, MPI_COMM_WORLD);
 *local_n = *n / size + (rank < (*n % size));
}

void read_vector(int *vetor, int *local_vetor, int n, int *local_n, int rank,
                 int size, int *send_counts, int *desl) {
 if (rank == 0) {
   int aux;
   printf("Entre com %d valores:\n", n);
   for (int i = 0; i < n; i++) {
     scanf("%d", &aux);
     vetor[i] = aux;
   }
   printf("\n");
 }
 MPI_Scatterv(vetor, send_counts, desl, MPI_INT, local_vetor, *local_n,
              MPI_INT, 0, MPI_COMM_WORLD);
}

void block_compute(int size, int *send_counts, int *desl, int n) {
 int rest = n % size;
```

```
 for (int r = 0; r < size; r++) {
   send_counts[r] = (n / size) + (r < (rest));
   desl[r] = r * (n / size) + (r < (rest) ? r : rest);
 }
}

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

40. Encontrar somas de prefixos é uma generalização da soma global. Em vez de simplesmente encontrar a soma de n valores,

$$x_0 + x_1 + \cdots + x_{n-1}, \quad (2)$$

as somas dos prefixos são as n somas parciais

$$x_0,\ x_0 + x_1,\ x_0 + x_1 + x_2,\ \cdots,\ x_0 + x_1 + \cdots + x_{n-1}. \quad (3)$$

(a) Elabore um algoritmo serial para calcular as n somas de prefixos de um vetor com n elementos.

Código:

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

void read_vetor(int n, int *vetor);
void print_vetor(int n, int *vetor);

int main() {
 int *vetor;
 int *soma;
 int n;

 printf("Entre com o tamanho do vetor:\n");
 scanf("%d", &n);

 vetor = (int *)malloc(n * sizeof(int));
 soma = (int *)malloc(n * sizeof(int));

 read_vetor(n, vetor);

 int aux = 0;
```

```
 for (int i = 0; i < n; i++) {
   aux += vetor[i];
   soma[i] = aux;
 }

 printf("Vetor\n");
 print_vetor(n, vetor);
 printf("Somas parcias\n");
 print_vetor(n, soma);

 return 0;
}

void read_vetor(int n, int *vetor) {
 printf("Entre com %d elementos para o vetor:\n", n);
 int aux;
 for (int i = 0; i < n; i++) {
   scanf("%d", &aux);
   vetor[i] = aux;
 }
 printf("\n");
}

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

(b) Paralelize seu algoritmo serial para um sistema com *n* processos, cada um armazenando um dos elementos de *x*.

- Sem utilizar MPI_Scan

Código:

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

void read_vetor(int n, int *vetor, int rank, int *local_sum, MPI_Comm comm);
void prefix_sum(int size, int rank, int *local_sum, MPI_Comm comm);
void print_vetor(int n, int *vetor);

int main(void) {
  int *vetor = NULL;
```

```c
  int *soma = NULL;
  int n, rank, size;
  int local_sum;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Init(NULL, NULL);

  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  n = size;
  soma = (int *)malloc(n * sizeof(int));

  read_vetor(n, vetor, rank, &local_sum, comm);
  prefix_sum(n, rank, &local_sum, comm);

  MPI_Gather(&local_sum, 1, MPI_INT, soma, 1, MPI_INT, 0, comm);

  if (rank == 0) {
    printf("A soma dos prefixos: \n");
    print_vetor(n, soma);
  }

  free(soma);
  MPI_Finalize();

  return 0;
}

void read_vetor(int n, int *vetor, int rank, int *local_sum, MPI_Comm comm) {
  vetor = (int *)malloc(sizeof(int) * n);
  if (rank == 0) {
    printf("Entre com %d elementos para o vetor:\n", n);
    for (int i = 0; i < n; i++) {
      scanf("%d", &vetor[i]);
    }
  }
  MPI_Scatter(vetor, 1, MPI_INT, local_sum, 1, MPI_INT, 0, comm);
  free(vetor);
}

void prefix_sum(int size, int rank, int *local_sum, MPI_Comm comm) {
```

```c
 int partner;
 int aux;
 if (rank > 0) {
   partner = rank - 1;
   MPI_Recv(&aux, 1, MPI_INT, partner, 0, comm, NULL);
   *local_sum += aux;
 }
 if (rank < size - 1) {
   partner = rank + 1;
   MPI_Send(local_sum, 1, MPI_INT, partner, 0, comm);
 }
}


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

Saída:

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/listaMPI/quest40$ mpirun mpi_b.o
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Entre com 4 elementos para o vetor:
1 2 3 4
A soma dos prefixos:
1 3 6 10
```

(c) Suponha $n = 2^k$ para algum inteiro positivo k. Crie um algoritmo paralelo que exija apenas k fases de comunicação.

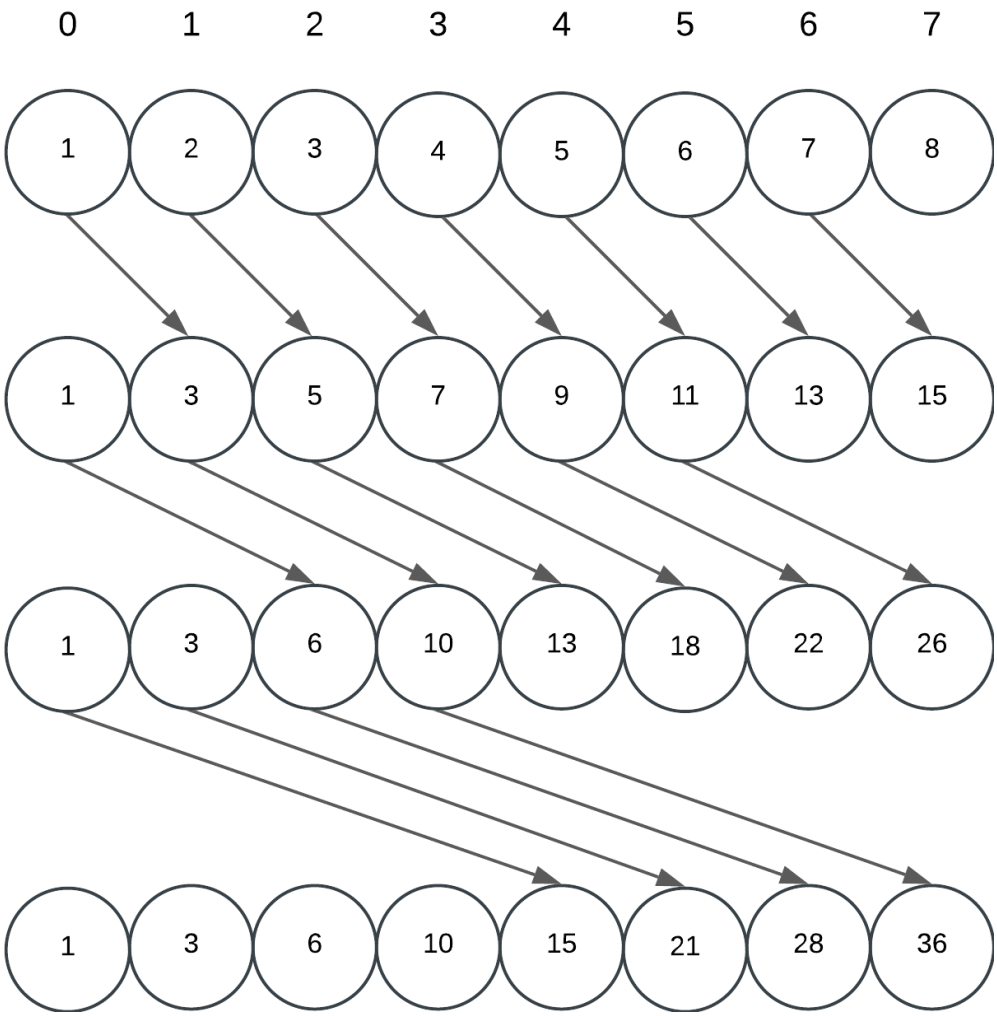
- Sem utilizar MPI_Scan

Modificações no código anterior:

```c
void prefix_sum_log(int size, int rank, int *local_sum, MPI_Comm comm) {
int step, partner;
int aux;

for (step = 1; step < size; step *= 2) {
  if (rank + step < size) {
```

```
      partner = rank + step;
      MPI_Send(local_sum, 1, MPI_INT, partner, 0, comm);
    }
    if (rank >= step) {
      partner = rank - step;
      MPI_Recv(&aux, 1, MPI_INT, partner, 0, comm, MPI_STATUS_IGNORE);
      *local_sum += aux;
    }
  }
}
```

DIAGRAMA:



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

```
int MPI Scan(
```

```
void* sendbuf p /* in */,
void* recvbuf p /* out */,
int count /* in */,
MPI Datatype datatype /* in */,
MPI Op op /* in */,
MPI Comm comm /* in */);
```

Ela opera em arrays com count elementos; sendbuf_p e recvbuf_p devem se referir a blocos de count elementos do tipo datatype. O argumento op é igual ao op para o MPI_Reduce. Escreva um programa MPI que gere um vetor aleatório de count elementos em cada processo MPI, encontre as somas dos prefixos e imprima os resultados.

Código:

```
int main(int argc, char *argv[]) {
 MPI_Comm comm = MPI_COMM_WORLD;
 int rank, size, n;
 int *sum_prefix, *vector;

 MPI_Init(&argc, &argv);
 MPI_Comm_size(comm, &size);
 MPI_Comm_rank(comm, &rank);

 get_input(&n, rank, comm);

 vector = (int *)malloc(sizeof(int) * n);
 sum_prefix = (int *)malloc(sizeof(int) * n);

 srand(rank + time(NULL));
 for (int i = 0; i < n; i++) {
   vector[i] = rand() % 10;
 }

 MPI_Barrier(comm);
 MPI_Scan(vector, sum_prefix, n, MPI_INT, MPI_SUM, comm);
 MPI_Barrier(comm);

 for (int r = 0; r < size; r++) {
   if (rank == r) {
     printf("Rank %d: \n\n", rank);
     printf("Initial Vector: \n");
     for (int i = 0; i < n; i++) {
       printf("%d ", vector[i]);
     }
     printf("\nPrefix sum: \n");
```

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

 free(vector);
 free(sum_prefix);

 MPI_Finalize();

 return 0;
}
```

Saída:

```
Rank 0:

Initial Vector:
2 0 6 8 9
Prefix sum:
2 0 6 8 9
Rank 1:

Initial Vector:
1 3 5 7 5
Prefix sum:
3 3 11 15 14
Rank 2:

Initial Vector:
9 3 7 5 4
Prefix sum:
12 6 18 20 18
Rank 3:

Initial Vector:
4 2 3 9 2
Prefix sum:
16 8 21 29 20
```

41. Uma alternativa para um allreduce estruturado em borboleta é uma estrutura de passagem em anel. Em uma passagem de anel, se houver p processos, cada processo q envia dados para o processo $q + 1$, exceto que o processo $p - 1$ envia dados para o processo 0. Isso é repetido até que cada processo tenha o resultado desejado. Assim, podemos implementar allreduce com o seguinte código:

```
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) Escreva um programa MPI que implemente esse algoritmo para o allreduce. Como seu desempenho se compara ao allreduce estruturado em borboleta?

Código:

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

void all_reduce_ring(int *sum, int rank, int p, int *my_val, MPI_Comm comm) {
 int i;
 int temp_val;
 int dest, source, sendtag, recvtag;

 dest = rank + 1;
 source = rank - 1;
 if (rank == (p - 1)) {
   source = rank - 1;
   dest = 0;
 } else if (rank == 0) {
   source = p - 1;
   dest = 1;
 }

 *sum = temp_val = *my_val;
 for (i = 1; i < p; i++) {
   MPI_Sendrecv_replace(&temp_val, 1, MPI_INT, dest, i, source, i, comm, NULL);
   *sum += temp_val;
 }
}

int main() {
 int sum, rank, size, my_val;
 MPI_Comm comm = MPI_COMM_WORLD;
```

```
 MPI_Init(NULL, NULL);
 MPI_Comm_size(comm, &size);
 MPI_Comm_rank(comm, &rank);

 sum = 0;
 my_val = 1;

 all_reduce_ring(&sum, rank, size, &my_val, comm);

 MPI_Barrier(comm);
 MPI_Allreduce(&my_val, &sum, 1, MPI_INT, MPI_SUM, comm);

 MPI_Finalize();

 return 0;
}
```

Não consegui maquinas suficientes para comparar seu desempenho de forma adequada, mas como a quantidade a quantidade de comunicações em anel tendem a uma função linear e a função em borboleta, a uma função logaritimica, se espera que em borboleta mostra melhor desempenho especialmente em casos com maior quantidades de processos.

(b) Modifique o programa MPI que você escreveu na primeira parte para que ele implemente somas de prefixos.

```
void all_reduce_ring(int *sum, int rank, int p, int *my_val, MPI_Comm comm) {
 int i;
 int temp_val;
 int dest, source, sendtag, recvtag;

 dest = rank + 1;
 source = rank - 1;
 if (rank == (p - 1)) {
   source = rank - 1;
   dest = 0;
 } else if (rank == 0) {
   source = p - 1;
   dest = 1;
 }

 *sum = temp_val = *my_val;
 for (i = 1; i < p; i++) {
   MPI_Sendrecv_replace(&temp_val, 1, MPI_INT, dest, i, source, i, comm, NULL);
   if (rank >= i)
     *sum += temp_val;
 }
```

```
}
```

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/listaMPI/quest41$ mpirun -n 4 --use-hwthread-
cpus prefix_sum
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Rank 0:
Initial valuer:
1
sum:
1

Rank 1:
Initial valuer:
2
sum:
3

Rank 2:
Initial valuer:
3
sum:
6

Rank 3:
Initial valuer:
4
sum:
10
```

42. As funções MPI_Scatter e MPI_Gather têm a limitação de que cada processo deve enviar ou receber o mesmo número de itens de dados. Quando este não for o caso, devemos utilizar as funções MPI_Gatherv e MPI_Scatterv. Consulte a documentação dessas funções e modifique seu programa da questão 39 para que ele possa lidar corretamente com o caso quando *n* não é divisível por comm_sz.

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

void get_input(int *n, int *local_n, int rank, int size, int *escalar);
void block_compute(int size, int *send_counts, int *desl, int n);
void read_vector(int *vetor, int *local_vetor, int n, int *local_n, int rank,
                 int size, int *send_counts, int *desl);

int main(int argc, char *argv[]) {
 int *vetor1, *vetor2; // vetores originais
 int rank, size;
 int n, local_n, escalar;
 int *local_vetor1, *local_vetor2;
 long long int global_produto = 1;
 int *send_counts;
 int *desl;
```

```c
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
send_counts = (int *)malloc(size * sizeof(int));
desl = (int *)malloc(size * sizeof(int));

get_input(&n, &local_n, rank, size, &escalar);
local_vetor1 = (int *)malloc(local_n * sizeof(int));
local_vetor2 = (int *)malloc(local_n * sizeof(int));

vetor1 = (int *)malloc(n * sizeof(int));
vetor2 = (int *)malloc(n * sizeof(int));

block_compute(size, send_counts, desl, n);

read_vector(vetor1, local_vetor1, n, &local_n, rank, size, send_counts, desl);
read_vector(vetor2, local_vetor2, n, &local_n, rank, size, send_counts, desl);

int local_produto = 1;
for (int k = 0; k < local_n; k++) {
  local_vetor1[k] *= escalar;
  local_produto += local_vetor2[k] * local_vetor2[k];
}

if (rank == 0) {
  printf("Vetor 1 passado:\n");
  print_vetor(n, vetor1);

  printf("Vetor 2 passado:\n");
  print_vetor(n, vetor2);
}

MPI_Reduce(&local_produto, &global_produto, 1, MPI_INT, MPI_PROD, 0,
           MPI_COMM_WORLD);

MPI_Gatherv(local_vetor1, local_n, MPI_INT, vetor1, send_counts, desl,
            MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0) {
  printf("Produto interno do vetor 2: %lld\n", global_produto);

  printf("Vetor 1 multiplicado pelo escalar %d:\n", escalar);
  print_vetor(n, vetor1);
}

free(local_vetor1);
free(local_vetor2);
free(send_counts);
```

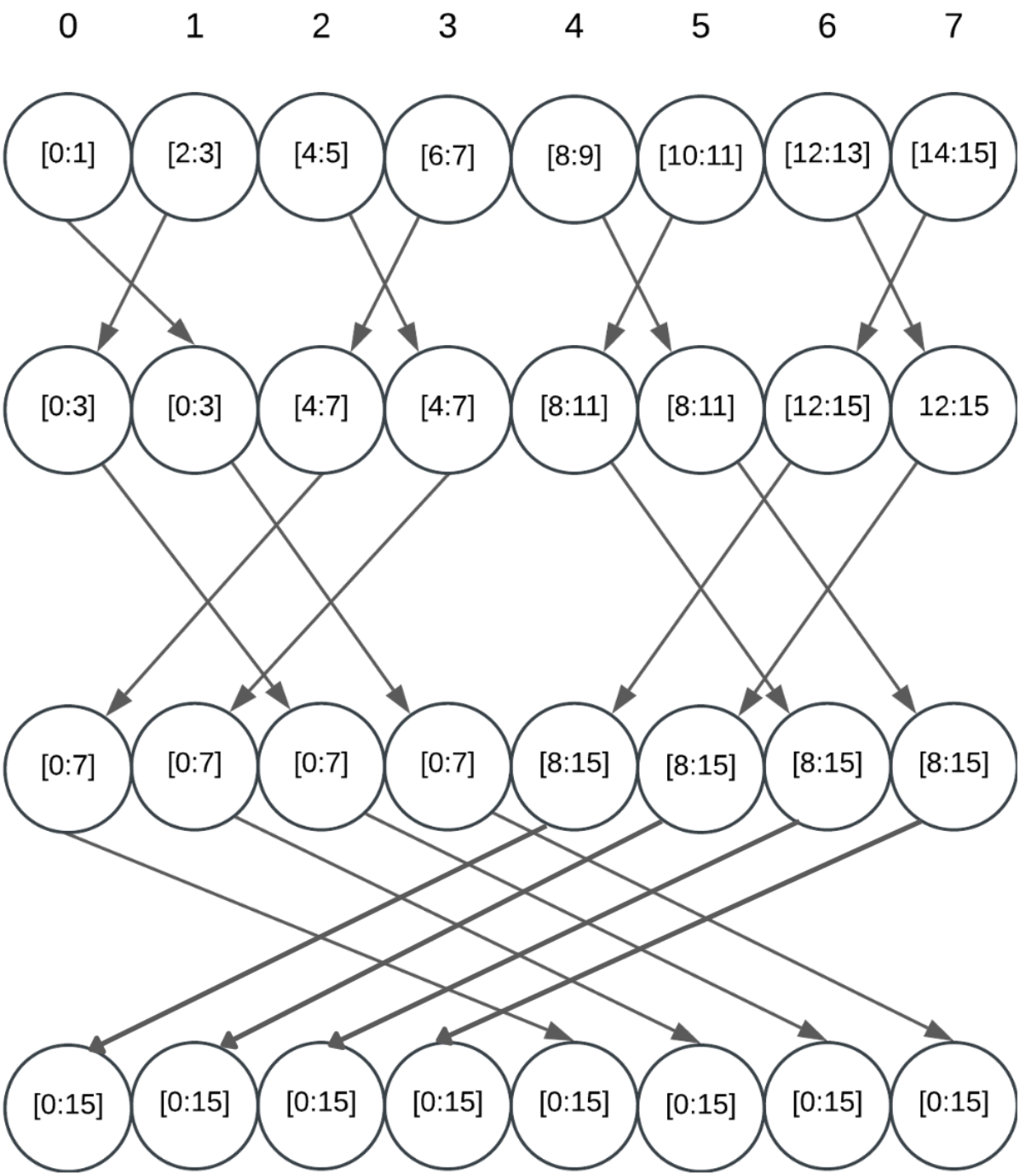
```
 free(desl);
 if (rank == 0) {
   free(vetor1);
   free(vetor2);
 }
 MPI_Finalize();

 return 0;
}

void get_input(int *n, int *local_n, int rank, int size, int *escalar) {
 if (rank == 0) {
   printf("Entre com o tamanho dos vetores:\n");
   scanf("%d", n);

   printf("Entre com o escalar: \n");
   scanf("%d", escalar);
 }

 MPI_Bcast(n, 1, MPI_INT, 0, MPI_COMM_WORLD);
 MPI_Bcast(escalar, 1, MPI_INT, 0, MPI_COMM_WORLD);
 *local_n = *n / size + (rank < (*n % size));
}

void read_vector(int *vetor, int *local_vetor, int n, int *local_n, int rank,
                 int size, int *send_counts, int *desl) {
 if (rank == 0) {
   int aux;
   printf("Entre com %d valores:\n", n);
   for (int i = 0; i < n; i++) {
     scanf("%d", &aux);
     vetor[i] = aux;
   }
   printf("\n");
 }
 MPI_Scatterv(vetor, send_counts, desl, MPI_INT, local_vetor, *local_n,
              MPI_INT, 0, MPI_COMM_WORLD);
}

void block_compute(int size, int *send_counts, int *desl, int n) {
 int rest = n % size;
 for (int r = 0; r < size; r++) {
   send_counts[r] = (n / size) + (r < (rest));
   desl[r] = r * (n / size) + (r < (rest) ? r : rest);
 }
}

void print_vetor(int n, int *vetor) {
 for (int i = 0; i < n; i++) {
```

```
    printf("%d ", vetor[i]);
  }
  printf("\n");
}
```

Saída:

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/listaMPI$ mpirun -n 4 ./39.o
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
Entre com o tamanho dos vetores:
15
Entre com o escalar:
2
Entre com 15 valores:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Entre com 15 valores:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Vetor 1 passado:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Vetor 2 passado:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Produto interno do vetor 2: 513864225
Vetor 1 multiplicado pelo escalar 2:
2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
```

43. Suponha que *comm_sz* = 8 e o vetor $x = (0, 1, 2, \cdots, 15)$ tenha sido distribuído entre os processos usando uma distribuição em bloco. Desenhe um diagrama ilustrando as etapas de uma implementação borboleta da função allgather de x.



44. A função MPI_Type_contiguous pode ser usada para construir um tipo de dados derivado de uma coleção de elementos contíguos em uma matriz. Sua sintaxe é

int MPI_Type_contiguous(

int count /* in */,
MPI Datatype old_mpi_t /* in */,
MPI Datatype* new_mpi_t_p /* out *);

Modifique as funções Read_vector e Print_vector (mpi_vector_add.c) para que elas usem um tipo de dados MPI criado por uma chamada para MPI_Type_contiguous e um argumento de contagem de 1 nas chamadas para MPI_Scatter e MPI_Gather.

```
void create_datatype(int *local_n, MPI_Datatype *vector, MPI_Comm comm) {
 MPI_Type_contiguous(*local_n, MPI_DOUBLE, vector);
 MPI_Type_commit(vector);
}
```

```
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  */,
                 MPI_Datatype vector) {

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

 if (my_rank == 0) {
   a = malloc(n * sizeof(double));
   if (a == NULL)
     local_ok = 0;
   Check_for_error(local_ok, fname, "Can't allocate temporary vector", comm);
   printf("Enter the vector %s\n", vec_name);
   for (i = 0; i < n; i++)
     scanf("%lf", &a[i]);
   MPI_Scatter(a, 1, vector, local_a, 1, vector, 0, comm);
   free(a);
 } else {
   Check_for_error(local_ok, fname, "Can't allocate temporary vector", comm);
   MPI_Scatter(a, 1, vector, local_a, 1, vector, 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 */, MPI_Datatype vector) {

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

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

Saída:

```
(base) felipehidequel@la-maquina:~/Desktop/pcd/listaMPI/quest44$ mpirun -np 2 mpi_vector_add
hwloc/linux: Ignoring PCI device with non-16bit domain.
Pass --enable-32bits-pci-domain to configure to support such devices
(warning: it would break the library ABI, don't enable unless really needed).
What's the order of the vectors?
8
Enter the vector x
1 2 3 4 5 6 7 8
x is
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
Enter the vector y
1 2 3 4 5 6 7 8
y is
1.000000 2.000000 3.000000 4.000000 5.000000 6.000000 7.000000 8.000000
The sum is
2.000000 4.000000 6.000000 8.000000 10.000000 12.000000 14.000000 16.000000
```

45. A função MPI_Type_indexed pode ser usada para construir um tipo de dados derivado de elementos arbitrários de um vetor. Sua sintaxe é

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

Ao contrário da função MPI_Type_create_struct, os deslocamentos são medidos em unidades de old_mpi_t - não em bytes. Use a função MPI_Type_indexed para criar um tipo de dados derivado que corresponda à parte triangular superior de uma matriz quadrada. Por exemplo, na matriz 4 x 4

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

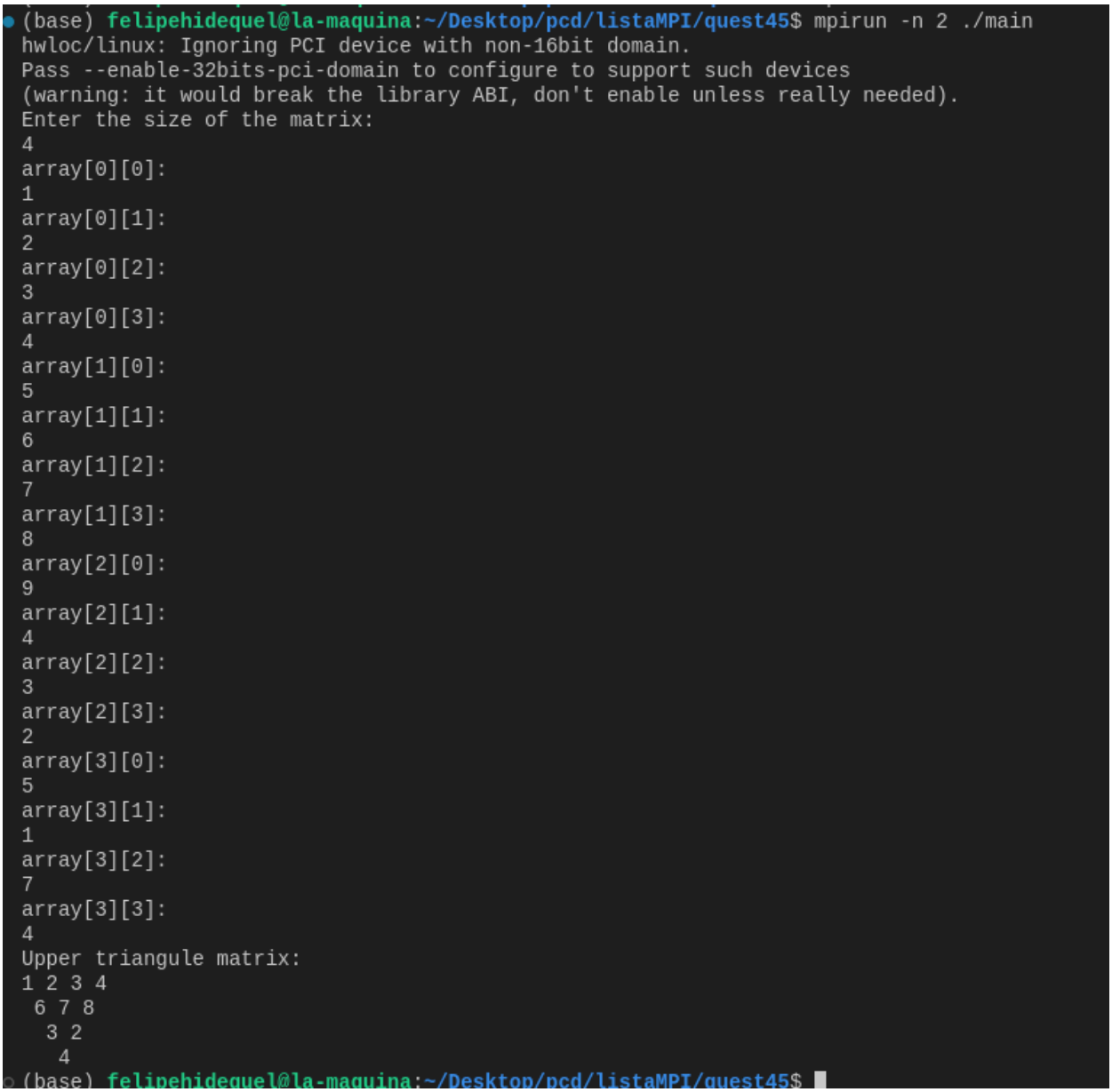
a parte triangular superior são os elementos 0, 1, 2, 3, 5, 6, 7, 10, 11, 15. O processo 0 deve ler uma matriz $n \times n$ como um vetor unidimensional, criar o tipo de dados derivado e enviar a parte triangular superior com uma única chamada de MPI_Send. O processo 1 deve receber a parte triangular superior com uma única chamada ao MPI_Recv e depois imprimir os dados recebidos.

Código:

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

void get_n(int *n, int *my_rank, int *comm_sz, MPI_Comm comm);
void create_datatype(int *array, int n, MPI_Datatype *upper_triangule);

int main() {
 int n, rank, comm_sz, *array;
 MPI_Comm comm = MPI_COMM_WORLD;
 MPI_Datatype upper_triangule;

 MPI_Init(NULL, NULL);
 MPI_Comm_size(comm, &comm_sz);
 MPI_Comm_rank(comm, &rank);

 get_n(&n, &rank, &comm_sz, comm);
 array = malloc(n * n * sizeof(int));

 create_datatype(array, n, &upper_triangule);
 if (rank == 0) {
   for (int i = 0; i < n; i++) {
     for (int j = 0; j < n; j++) {
       printf("array[%d][%d]: \n", i, j);
       scanf("%d", &array[i * n + j]);
     }
   }
```

```
    MPI_Send(array, 1, upper_triangle, 1, 0, comm);
  }

  if (rank == 1) {
    MPI_Recv(array, 1, upper_triangle, 0, 0, comm, MPI_STATUS_IGNORE);

    printf("Upper triangule matrix: \n");
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (j < i) {
          printf(" ");
          continue;
        }
        printf("%d ", array[i * n + j]);
      }
      printf("\n");
    }
  }

  free(array);
  MPI_Type_free(&upper_triangule);
  MPI_Finalize();

  return 0;
}

void get_n(int *n, int *my_rank, int *comm_sz, MPI_Comm comm) {
  if (*my_rank == 0) {
    printf("Enter the size of the matrix: \n");
    scanf("%d", n);
  }
  MPI_Bcast(n, 1, MPI_INT, 0, comm);
}

void create_datatype(int *array, int n, MPI_Datatype *upper_triangule) {
  int send_cout[n];
  int displs[n];

  for (int i = 0; i < n; i++) {
    send_cout[i] = n - i;
    displs[i] = i * n + i;
  }

  MPI_Type_indexed(n, send_cout, displs, MPI_INT, upper_triangule);
  MPI_Type_commit(upper_triangule);
}
```

Saída:



46. As funções MPI_Pack e MPI_Unpack fornecem uma alternativa aos tipos de dados deriva dos para agrupar dados. O MPI_Pack copia os dados a serem enviados, um bloco por vez, em um *buffer* fornecido pelo usuário. O *buffer* pode então ser enviado e recebido. Após o recebimento dos dados, MPI_Unpack pode ser usado para descompactá-los do *buffer* de recebimento. A sintaxe do MPI_Pack é

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

Poderíamos, portanto, empacotar os dados de entrada para o programa da regra dos trapézios com o seguinte código:

```
char pack_buf[100];
int position = 0;
MPI_Pack(&a, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&b, 1, MPI_DOUBLE, pack_buf, 100, &position, comm);
MPI_Pack(&n, 1, MPI_INT, pack_buf, 100, &position, comm);
```

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

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

Observe que o tipo de dados MPI para um *buffer* compactado é MPI_PACKED. Agora os outros processos podem descompactar os dados usando: MPI_Unpack:

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

MPI_Unpack pode ser usado "invertendo"as etapas do MPI_Pack, ou seja, os dados são descompactados um bloco por vez, começando em *position* = 0.

Escreva outra função Get_input para o programa da regra dos trapézios. Este deve usar MPI_Pack no processo 0 e MPI_Unpack nos demais processos.

```c
void Get_input(int my_rank, int comm_sz, double *a_p, double *b_p, int *n_p) {

 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);
   MPI_Pack(&a_p, 1, MPI_DOUBLE, pack_buf, 100, &position, MPI_COMM_WORLD);
   MPI_Pack(&b_p, 1, MPI_DOUBLE, pack_buf, 100, &position, MPI_COMM_WORLD);
   MPI_Pack(&n_p, 1, MPI_INT, pack_buf, 100, &position, MPI_COMM_WORLD);
 }
 MPI_Bcast(pack_buf, 100, MPI_PACKED, 0, MPI_COMM_WORLD);
 if (my_rank != 0) {
```

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

47. Cronometre a implementação do livro da regra dos trapézios que usa MPI_Reduce para diferentes números de trapézios e processos,*n* e *p*, respectivamente. Lembre-se de medir o tempo de execução ao menos 5 vezes para cada par (*n, p*).

    (a) Qual critério você utilizou para escolher *n*?

    (b) Como os tempos mínimos se comparam aos tempos médios e medianos?

    (c) Quais são os *speedups*?

    (d) Quais são as eficiências?

    (e) Com base nos dados que você coletou, você diria que a regra dos trapézios é escalável?

    Código módificado:

```
int main(int argc, char *argv[]) {
 int my_rank, comm_sz, n, local_n;
 double a, b, h, local_a, local_b;
 double local_int, total_int;
 double start, finish, local_elapsed, elapsed;

 /* Let the system do what it needs to start up MPI */
 MPI_Init(&argc, &argv);
 MPI_Barrier(MPI_COMM_WORLD);
 start = MPI_Wtime();

 /* Get my process rank */
 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

 /* Find out how many processes are being used */
 MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
 if (my_rank == 0) {
   n = strtol(argv[1], NULL, 10);
   a = 0;
   b = 1;
 }
 Get_input(my_rank, comm_sz, &a, &b, &n);

 h = (b - a) / n;       /* h is the same for all processes */
 local_n = n / comm_sz; /* So is the number of trapezoids  */

 /* Length of each process' interval of
  * integration = local_n*h.  So my interval
```

```
  * starts at: */
 local_a = a + my_rank * local_n * h;
 local_b = local_a + local_n * h;
 local_int = Trap(local_a, local_b, local_n, h);

 /* Add up the integrals calculated by each process */
 MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

 /* Print the result */
 if (my_rank == 0) {
   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.15e\n", a, b, total_int);
 }

 /* Shut down MPI */
 finish = MPI_Wtime();
 local_elapsed = finish - start;
 MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0,
            MPI_COMM_WORLD);
 if (my_rank == 0) {
   printf("Elapsed time = %.6f\n", elapsed);
 }
 MPI_Finalize();

 return 0;
} /*  main  */
```

TABELAS:

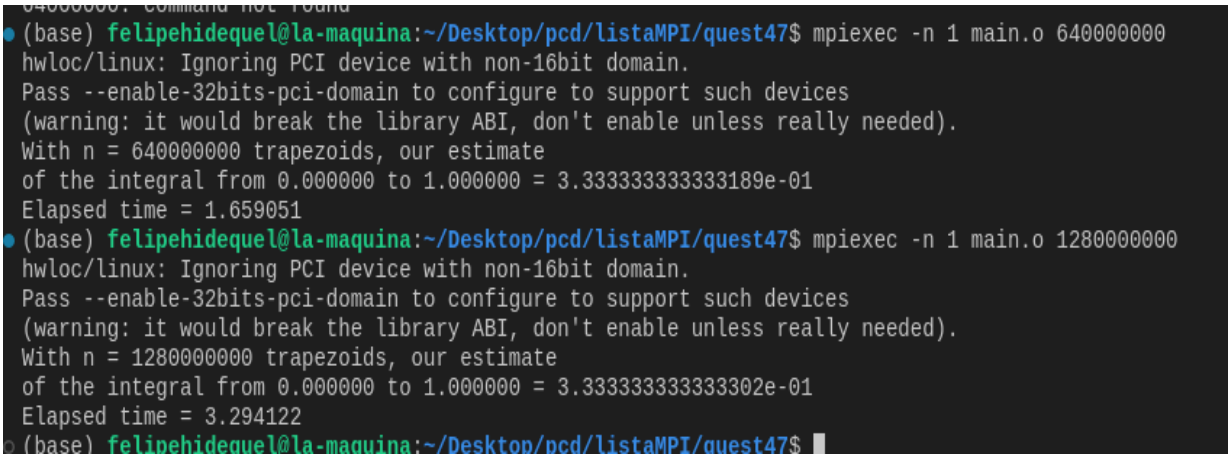
medios (segundos)				
	N			
p	$64*10^7$	$128*10^7$	$512*10^7$	$1024*10^7$
1	1.436927	3.079102	2.010693	4.154027
2	0.762086	1.657926	1.088632	2.333555
4	0.413428	0.895481	0.583419	1.303487
8	0.438046	0.893335	0.605147	1.244554

min (segundos)				
	N			
p	$64*10^7$	$128*10^7$	$512*10^7$	$1024*10^7$
1	1.43461	2.989405	2.006199	4.093263

2	0.76037	1.644573	1.079916	2.319371
4	0.406438	0.88465	0.579487	1.28648
8	0.423283	0.87945	0.591966	1.206131

max(segundos)				
	N			
p	$64*10^7$	$128*10^7$	$512*10^7$	$1024*10^7$
1	1.449027	3.095081	2.012784	4.215988
2	0.777897	1.675557	1.106954	2.358389
4	0.42111	0.90355	0.591541	1.323419
8	0.460536	0.917583	0.673619	1.315738

A. Para escolher N eu procurei um valor inicial que durasse mais de um segundo e multipliquei ele por algum fator até que o tempo de duração duplicasse:



obtido esses valores, comecei a duplicar a entrada e duplicar a quantidade de processos.

B. Os minimos e maxímos se mostraram bem proximos dos tempos medios, dando confiabilidade as amostras, se os tempos são consistentes a escolha de N foi correta. Os valores para 8 processos estão muito proximos dos de 4p, além dos testes serem realizados em uma só maquina, para executar 8 processos estavamos utilizando as threads de hardware e isso pode ser o motivo do ganho não alterar tanto com 8 p.

C.

speedup				
	N			
p	$64*10^7$	$128*10^7$	$512*10^7$	$1024*10^7$
1	100.00%	100.00%	100.00%	100.00%
2	188.55%	185.72%	184.70%	178.01%
4	347.56%	343.85%	344.64%	318.69%
8	328.03%	344.67%	332.27%	333.78%

D.

eficiência				
	N			
p	$64*10^7$	$128*10^7$	$512*10^7$	$1024*10^7$
1	100.00%	100.00%	100.00%	100.00%
2	94.28%	92.86%	92.35%	89.01%
4	86.89%	85.96%	86.16%	79.67%
8	41.00%	43.08%	41.53%	41.72%

E. Com base nessa tabela de eficiência poderiamos deduzir que o programa dos trapézios não é escalavel, já que nesses testes ele mostrou uma queda de eficiência. No entanto para verificar de forma adequada seria necessario rodar o programa em um sistema distribuído e com um tamanho de problema maior.

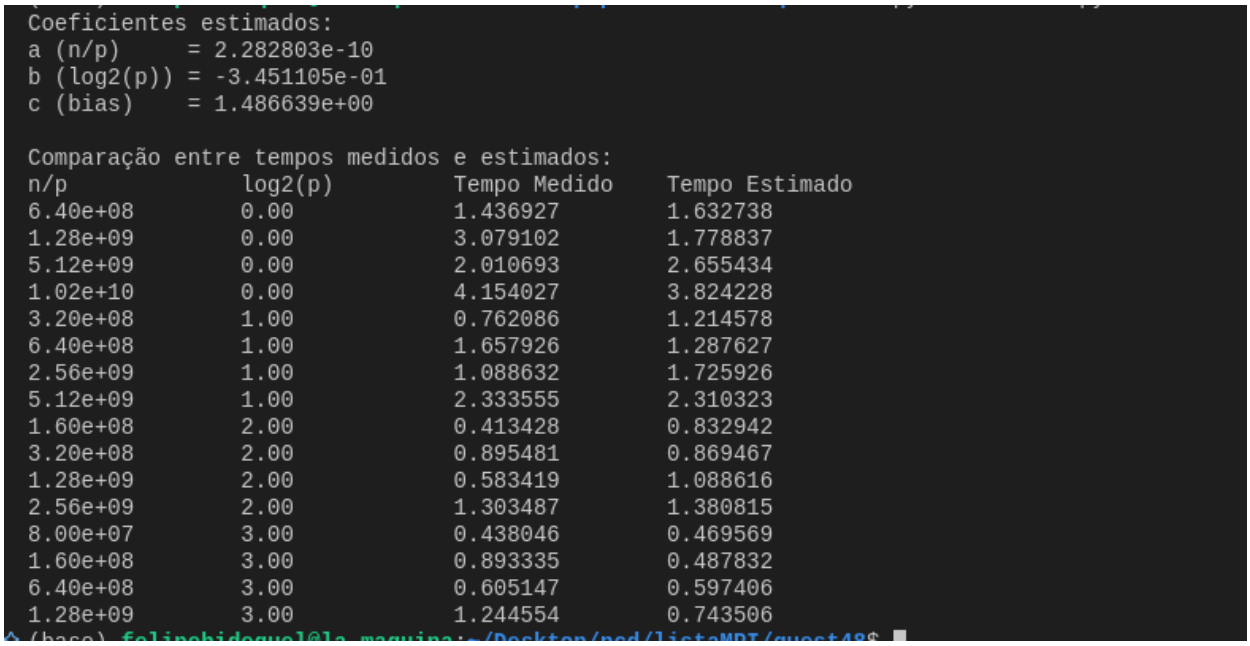
48. Embora não conheçamos os detalhes da implementação do MPI_Reduce, podemos supor que ele usa uma estrutura semelhante à árvore binária que discutimos. Se for esse o caso, esperaríamos que seu tempo de execução crescesse aproximadamente à taxa de $\log_2(p)$ (p = *comm_sz*), uma vez que existem aproximadamente $\log_2(p)$ níveis na árvore. Como o tempo de execução da regra dos trapézios serial é aproximadamente proporcional a n, o número de trapézios, e a regra dos trapézios paralela simplesmente aplica a regra serial a n/p trapézios em cada processo, com nossa suposição sobre MPI_Reduce, obtemos uma fórmula para o tempo de execução geral da regra dos trapézios paralela que se parece com

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

onde a e b são constantes.

Use a fórmula, os tempos que você mediu no Exercício 47 e seu programa favorito para fazer cálculos matemáticos (por exemplo, o MATLAB®) para obter uma estimativa de

mínimos quadrados dos valores de *a* e *b*. Comente sobre a qualidade dos tempos de execução previstos usando a fórmula.



```
Coeficientes estimados:
a (n/p)     = 2.282803e-10
b (log2(p)) = -3.451105e-01
c (bias)    = 1.486639e+00

Comparação entre tempos medidos e estimados:
n/p             log2(p)         Tempo Medido    Tempo Estimado
6.40e+08        0.00            1.436927        1.632738
1.28e+09        0.00            3.079102        1.778837
5.12e+09        0.00            2.010693        2.655434
1.02e+10        0.00            4.154027        3.824228
3.20e+08        1.00            0.762086        1.214578
6.40e+08        1.00            1.657926        1.287627
2.56e+09        1.00            1.088632        1.725926
5.12e+09        1.00            2.333555        2.310323
1.60e+08        2.00            0.413428        0.832942
3.20e+08        2.00            0.895481        0.869467
1.28e+09        2.00            0.583419        1.088616
2.56e+09        2.00            1.303487        1.380815
8.00e+07        3.00            0.438046        0.469569
1.60e+08        3.00            0.893335        0.487832
6.40e+08        3.00            0.605147        0.597406
1.28e+09        3.00            1.244554        0.743506
(base) felipebideguel@la-maquina:~/Desktop/ped/listaMPI/quest48$
```

Para cada variação de n/p e log2(p) a estimativa está bem proxima do tempo real, com algumas discrepancias principalmente para quantidades menores de processos. Enquanto o cociente a está diretamente relacionado a n/p mostra um valor pequeno (mostra que o tempo cresce devagar com o aumento de n/p) ,o conciente b que é ligado ao log2(p) é negativo, o que implica que o tempo reduz ao aumentarmos a quantidade de processos, estimativa esperada.

49. Encontre os *speedups* e as eficiências da ordenação ímpar-par paralela (mpi_odd_even.c).

(a) O programa obtém *speedups* lineares?

Para isso, o speedup tem que ser S = P, não é o que observamos nas tabelas.

(b) É escalável?

A perca de eficiência drastica ilustra na diagonal da tabela e observando linearmente, ilustra uma aplicação que não escala

(c) É fortemente escalável?

Observando linearmente as células da tabela de eficiência nota-se que a eficiência cai a medida que a quantidade de processos aumenta, isso é, o programa não é fortemente escalável.

(d) É fracamente escalável?

Observando a tabela de eficiência nota-se que a eficiência cai a medida que a quantidade de processos aumenta e a medida que tamanho do problema aumenta, isso é, o programa não é fracamente escalável.

Para medir o tempo, comentei os trechos que imprimiam os vetores e adpatei o código com MPI_Wtime().

Código:

```c
int main(int argc, char *argv[]) {
 int my_rank, p;
 char g_i;
 int *local_A;
 int global_n;
 int local_n;
 MPI_Comm comm;
 double start, end, local_elapsed, elapsed;

 MPI_Init(&argc, &argv);
 comm = MPI_COMM_WORLD;
 MPI_Barrier(comm);
 start = MPI_Wtime();

 MPI_Comm_size(comm, &p);
 MPI_Comm_rank(comm, &my_rank);

 Get_args(argc, argv, &global_n, &local_n, &g_i, my_rank, p, comm);
 local_A = (int *)malloc(local_n * sizeof(int));
 if (g_i == 'g') {
   Generate_list(local_A, local_n, my_rank);
   // Print_local_lists(local_A, local_n, my_rank, p, comm);
 } else {
   Read_list(local_A, local_n, my_rank, p, comm);
#ifdef DEBUG
   Print_local_lists(local_A, local_n, my_rank, p, comm);
#endif
 }

#ifdef DEBUG
 printf("Proc %d > Before Sort\n", my_rank);
 fflush(stdout);
#endif
 Sort(&local_A, local_n, my_rank, p, comm);

#ifdef DEBUG
 Print_local_lists(local_A, local_n, my_rank, p, comm);
 fflush(stdout);
#endif

 // Print_global_list(local_A, local_n, my_rank, p, comm);
```

```c
free(local_A);

end = MPI_Wtime();
local_elapsed = end - start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
if (my_rank == 0)
  printf("Elapsed time = %e\n", elapsed);

MPI_Finalize();

return 0;
```

Os tamanhos de problema escolhidos foram $16*10^4$,$32*10^4$,$64*10^4$ e $128*10^4$ para a variação de processos 8,4,2 e 1. Para obter essas tabelas, foram realizados 5 repetições para cada configuração e feito a mediana.

Tempos de execução em segundos:

Tempos de exeução (seg)				
	N			
p	$16*10^4$	$32*10^4$	$64*10^4$	$128*10^4$
1	2.134769	4.467662	11.30904	19.415551
2	1.372676	3.10698	6.735139	11.835656
4	0.873067	1.83487	3.965794	7.679778
8	0.479676	1.013167	2.119485	4.678075

Speedup's:

speedup				
p/n	$16*10^4$	$32*10^4$	$64*10^4$	$128*10^4$
1	100.00%	100.00%	100.00%	100.00%
2	152.38%	148.99%	162.62%	173.45%
4	237.29%	230.81%	249.52%	277.00%
8	318.01%	319.25%	335.19%	369.67%

Eficiência:

	n/p	1	2	4	8
eficiencia	$16*10^4$	100.00%	76.19%	59.32%	39.75%
	$32*10^4$	100.00%	74.50%	57.70%	39.91%

	$64*10^4$	100.00%	81.31%	62.38%	41.90%
	$128*10^4$	100.00%	86.72%	69.25%	46.21%

50. Modifique a ordenação ímpar-par paralela (mpi_odd_even.c) para que as funções Merge simplesmente troquem os ponteiros do vetor após encontrar os elementos menores ou maiores. Que efeito essa mudança tem no tempo de execução geral?

Para medirmos o tempo de execução foram feitas as seguintes modificações na função main:

```
int main(int argc, char *argv[]) {
 int my_rank, p;
 char g_i;
 int *local_A;
 int global_n;
 int local_n;
 MPI_Comm comm;
 double start, end, local_elapsed, elapsed;

 MPI_Init(&argc, &argv);
 comm = MPI_COMM_WORLD;
 MPI_Barrier(comm);
 start = MPI_Wtime();

 MPI_Comm_size(comm, &p);
 MPI_Comm_rank(comm, &my_rank);

 Get_args(argc, argv, &global_n, &local_n, &g_i, my_rank, p, comm);
 local_A = (int *)malloc(local_n * sizeof(int));
 if (g_i == 'g') {
   Generate_list(local_A, local_n, my_rank);
   // Print_local_lists(local_A, local_n, my_rank, p, comm);
 } else {
   Read_list(local_A, local_n, my_rank, p, comm);
#ifdef DEBUG
   Print_local_lists(local_A, local_n, my_rank, p, comm);
#endif
 }

#ifdef DEBUG
 printf("Proc %d > Before Sort\n", my_rank);
 fflush(stdout);
#endif
 Sort(&local_A, local_n, my_rank, p, comm);

#ifdef DEBUG
```

```
 Print_local_lists(local_A, local_n, my_rank, p, comm);
 fflush(stdout);
#endif

 // Print_global_list(local_A, local_n, my_rank, p, comm);

 free(local_A);

 end = MPI_Wtime();
 local_elapsed = end - start;
 MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);
 if (my_rank == 0)
   printf("Elapsed time = %e\n", elapsed);

 MPI_Finalize();

 return 0;
}
```

Modificações feitas nas funções merge:

```
void Merge_low(int *my_keys[], int recv_keys[], int *temp_keys[], int local_n) {
 int m_i, r_i, t_i;
 m_i = r_i = t_i = 0;
 while (t_i < local_n) {
   if ((*my_keys)[m_i] <= recv_keys[r_i]) {
     (*temp_keys)[t_i] = (*my_keys)[m_i];
     t_i++;
     m_i++;
   } else {
     (*temp_keys)[t_i] = recv_keys[r_i];
     t_i++;
     r_i++;
   }
 }
 int *temp = *my_keys;
 *my_keys = *temp_keys;
 *temp_keys = temp;
}
void Merge_high(int *local_A[], int temp_B[], int *temp_C[], int local_n) {
 int ai, bi, ci;
 ai = bi = ci = local_n - 1;
 while (ci >= 0) {
   if ((*local_A)[ai] >= temp_B[bi]) {
     (*temp_C)[ci] = (*local_A)[ai];
     ci--;
     ai--;
   } else {
```

```
    (*temp_C)[ci] = temp_B[bi];
    ci--;
    bi--;
   }
 }
 int *temp = *local_A;
 *local_A = *temp_C;
 *temp_C = temp;
} /* Merge_high */
```

Na tabelas a seguir cada celula é uma mediana de 5 amostras, o tempo de execução é em segundos:

pointers				
	N			
p	$16*10^4$	$32*10^4$	$64*10^4$	$128*10^4$
1	2.11078	4.334049	9.082627	20.885773
2	1.385198	2.908907	5.585258	12.041698
4	0.889519	1.877725	3.640075	7.539931
8	0.663739	1.35756	2.709718	5.649867
memcpy				
	N			
p	$16*10^4$	$32*10^4$	$64*10^4$	$128*10^4$
1	2.134769	4.467662	11.30904	19.415551
2	1.372676	3.10698	6.735139	11.835656
4	0.873067	1.83487	3.965794	7.679778
8	0.479676	1.013167	2.119485	4.678075

Em alguns casos a implementação com troca de ponteiros apresentou melhor desempenho principalmente de 1 a 4 processos no tamanho $64*10^4$ onde o speedup relativo atingiu até 24,51%, mas no geral, na maioria dos casos a implementação com memcpy teve um ganho maior.

5 Referência

PACHECO, Peter S. An introduction to parallel programming. Amsterdam Boston: Morgan Kaufmann, c2011. xix, 370 p. ISBN: 9780123742605.