



**UNIVERSIDADE FEDERAL DO RURAL DO SEMI-ÁRIDO
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS
DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIA
PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA
DOCENTE: ÍTALO AUGUSTO SOUZA DE ASSIS**

ERIKY ABREU VELOSO

PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

**PAU DOS FERROS – RN
2025**

QUESTÕES RESPONDIDAS

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

PROGRAMAÇÃO DE MEMÓRIA DISTRIBUÍDA COM MPI

• QUESTÃO 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 $n \geq comm_sz$.

RESPOSTA

Considerando os testes realizados no código `mpi_trap1.c`, com os valores fixos de $a = 0$, $b = 3$ e $n = 1024$, e variando apenas o número de processos (`comm_sz`), observa-se na Figura 1 que, quando o número de processos não é um divisor exato de n , como nas duas últimas execuções, o valor estimado da integral apresenta discrepâncias.

Figura 1. Execução do código “`mpi_trap1.c`” com diferentes números de processos

```
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/ipp-source-use/ch3$ mpiexec -n 1 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.000004291534424e+00
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/ipp-source-use/ch3$ mpiexec -n 2 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.000004291534424e+00
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/ipp-source-use/ch3$ mpiexec -n 3 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 8.973662840668112e+00
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/ipp-source-use/ch3$ mpiexec -n 5 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 8.894946975633502e+00
```

Esse erro ocorre pelo fato de o restante da divisão ser totalmente ignorado durante o processo de distribuição das tarefas entre os processos. Pode-se observar no Código 1 que o intervalo local do cálculo da integral depende diretamente da variável `local_n`, que representa o número de trapézios a ser calculado por cada processo. No entanto, `local_n` armazena apenas o valor inteiro da divisão e não direciona o restante da divisão para ser calculado por algum processo.

Código 1. Trecho do código “`mpi_trap1.c`”

C/C++

```
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);
```

Dessa forma, quando o número de processos não é um divisor exato de n , alguns trapézios deixam de ser atribuídos, ou seja, não são calculados por nenhum processo. Como resultado, a área correspondente a esses trapézios não é somada ao valor total, gerando um resultado incorreto para a integral. Entretanto, realizando a modificação exposta no Código 2, atribuímos os trapézios restantes para os processos iniciais, sendo possível calcular a área total da figura.

Código 2. Trecho do código “mpi_trap1.c” (Modificado)

```
C/C++
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: */

resto = n%comm_sz;
if (resto == 0){
    local_a = a + my_rank*local_n*h;
    local_b = local_a + local_n*h;
}else{
    if(my_rank < resto){
        local_n += 1;
        local_a = a + my_rank*local_n*h;
        local_b = local_a + local_n*h;
    }else{
        local_a = a + my_rank*local_n*h + resto*h;
        local_b = local_a + local_n*h;
    }
}
```

A Figura 2 apresenta a saída do código mpi_trap1.c após a modificação, com os mesmos valores de processos adotados na Figura 1, evidenciando a integral correta em todos os casos testados.

Figura 2. Execução do código “mpi_trap1.c” (Modificado)

```
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_36$ mpiexec -n 1 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.000004291534424e+00
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_36$ mpiexec -n 2 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.000004291534424e+00
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_36$ mpiexec -n 3 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.000004291534424e+00
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_36$ mpiexec -n 5 ./mpi_trap1
With n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 9.000004291534424e+00
```

- QUESTÃO 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.

RESPOSTA

Aplicando a modificação do código mpi_output.c exposta no Código 3, a saída é impressa na ordem de classificação do processo. A modificação consiste no processo zero imprimir sua mensagem, e posteriormente receber a mensagem ordenadamente de todos os outros núcleos.

Código 3. Função main do código “mpi_output.c” (Modificado)

```
C/C++  
  
int main(void) {  
    int my_rank, comm_sz;  
    char message[MAX_STRING];  
  
    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, strlen(message)+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 k = 1; k < comm_sz; k++){  
            MPI_Recv(message, MAX_STRING, MPI_CHAR, k, 0, MPI_COMM_WORLD,  
MPI_STATUS_IGNORE);  
            printf("%s", message);  
        }  
    }  
  
    MPI_Finalize();  
    return 0;  
} /* main */
```

A saída do Código 3, aplicando seis processos, pode ser visualizada a seguir, na Figura 3.

Figura 3. Execução do código “mpi_output.c” (Modificado)

```
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_37$ mpiexec -n 6 ./mpi_output
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

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

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.

Para todas os três tópicos, suponha que o programa seja executado com `comm_sz` = 4 processos e que x seja um vetor com n = 18 componentes.

RESPOSTA A _____

Em uma distribuição em bloco, os elementos do vetor são divididos em segmentos contíguos, de modo que cada processo receba uma porção contínua de elementos. Para garantir uma distribuição justa, o número total de elementos n é dividido pelo número de processos `comm_sz`, e o quociente $q = n / \text{comm_sz}$ representa o número mínimo de elementos que cada processo deve receber. O restante $r = n \% \text{comm_sz}$ indica quantos processos precisarão receber um elemento a mais para que todos os n elementos sejam distribuídos. Assim, os r primeiros processos recebem $q + 1$ elementos, enquanto os demais recebem exatamente q elementos. Dessa forma, a diferença entre o número de elementos atribuídos a qualquer par de processos é no máximo 1. Na Tabela 1 pode-se observar um exemplo de como ocorre a distribuição.

Tabela 1. Distribuição em blocos

Processos	Componentes
0	0, 1, 2, 3, 4
1	5, 6, 7, 8, 9
2	10, 11, 12, 13
3	14, 15, 16, 17

RESPOSTA B

Na distribuição cíclica, os elementos de x são atribuídos alternadamente aos processos. O elemento $x[0]$ vai para o processo P_0 , $x[1]$ vai para o processo P_1 , e assim sucessivamente até $x[\text{comm_sz}-1]$, momento em que o processo P_0 recebe novamente um elemento, iniciando um novo ciclo, realizando novamente uma distribuição cíclica. Esse padrão se repete até o fim do vetor, e garante que os elementos sejam distribuídos uniformemente de forma intercalada. Como consequência, a diferença entre o número de elementos atribuídos a quaisquer dois processos será também, no máximo, 1, tornando a distribuição justa. Na Tabela 2 pode-se observar como ocorre a distribuição cíclica de 18 componentes.

Tabela 2. Distribuição cíclicas

Processos	Componentes
0	0, 4, 8, 12, 16
1	1, 5, 9, 13, 17
2	2, 6, 10, 14
3	3, 7, 11, 15

RESPOSTA C

Na distribuição bloco-cíclica com tamanho de bloco b , o vetor é primeiramente dividido em blocos de b elementos. Cada um desses blocos é então atribuído ciclicamente aos processos. Ou seja, o primeiro bloco de b elementos é enviado ao processo P_0 , o segundo bloco ao processo P_1 , e assim por diante, reiniciando em P_0 após o último processo. Se houver blocos restantes, n não ser múltiplo de b , o último bloco pode conter menos que b elementos, e mesmo assim será atribuído seguindo a mesma ordem cíclica. Essa abordagem garante uma distribuição equilibrada dos dados, onde a diferença no número total de elementos entre dois processos quaisquer é, no máximo, igual ao tamanho do bloco b . A seguir, na Tabela 3, pode-se observar um exemplo de distribuição bloco-cíclica, considerando $b = 3$.

Tabela 2. Distribuição cíclicas

Processos	Componentes
0	0, 1, 2, 12, 13, 14
1	2, 4, 5, 15, 16, 17
2	6, 7, 8
3	9, 10, 11

- QUESTÃO 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 a sua norma. 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`.

RESPOSTA

O Código 4, exposto a seguir, obteve como base de desenvolvimento o código disponibilizado pelo material de apoio denominado de “`mpi_vector_add.c`”, sendo modificado para atender ao solicitado na questão.

Código 4. Código “`mpi_vector_39.c`”

```
C/C++
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <math.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_vector(double** local_x_pp, int local_n, MPI_Comm comm);
void Read_vector(double local_a[], int local_n, int n, char vec_name[], int my_rank, MPI_Comm comm);
void Print_vector(double local_b[], int local_n, int n, char title[], int my_rank, MPI_Comm comm);
void Set_scale(int* scale, int my_rank, MPI_Comm comm);

int main(void) {
    int my_rank, comm_sz, n_vector, local_n_vector, scale;
    double *local_vector_1, *local_vector_2;
    double sum = 0, normalization=0;

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

    Read_n(&n_vector, &local_n_vector, my_rank, comm_sz, MPI_COMM_WORLD);

    Allocate_vector(&local_vector_1, local_n_vector, MPI_COMM_WORLD);
```

```

    Read_vector(local_vector_1, local_n_vector, n_vector, "ONE", my_rank,
MPI_COMM_WORLD);

    Print_vector(local_vector_1, local_n_vector, n_vector, "ONE", my_rank,
MPI_COMM_WORLD);

    Set_scale(&scale, my_rank, MPI_COMM_WORLD);

    for(int i = 0; i < local_n_vector; i++){
        local_vector_1[i] = local_vector_1[i]*scale;
    }

    Print_vector(local_vector_1, local_n_vector, n_vector, "THE VECTOR ONE
MULTIPLIED BY THE SCALAR", my_rank, MPI_COMM_WORLD);

    //START VECTOR TWO
    Allocate_vector(&local_vector_2, local_n_vector, MPI_COMM_WORLD);

    Read_vector(local_vector_2, local_n_vector, n_vector, "TWO", my_rank,
MPI_COMM_WORLD);

    Print_vector(local_vector_2, local_n_vector, n_vector, "TWO", my_rank,
MPI_COMM_WORLD);

    for(int i = 0; i < local_n_vector; i++){
        local_vector_2[i] = local_vector_2[i]*local_vector_2[i];
        sum += local_vector_2[i];
    }

    MPI_Reduce(&sum, &normalization, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    if(my_rank == 0){
        normalization = sqrt(normalization);
        printf("THE NORM OF VECTOR TWO IS %.2lf\n", normalization);
    }

    free(local_vector_1);
    free(local_vector_2);
    MPI_Finalize();
    return 0;
} /* main */

// Check_for_error
-----

void Check_for_error(
    int        local_ok    /* in */,

```

```

        char      fname[]      /* in */,
        char      message[]    /* in */,
        MPI_Comm  comm         /* in */) {
    int ok;

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

// Read_n
-----

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 */

// Allocate_vector
-----

void Allocate_vector(
    double**  local_x_pp /* out */,
    int       local_n     /* in  */,

```

```

    MPI_Comm comm /* in */) {
int local_ok = 1;
char* fname = "Allocate_vector";

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

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

// Read_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 */) {

    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]);
        }
        /*printf("VETOR %s: ", vec_name);
        for (i = 0; i < n; i++){
            printf("%.2f", a[i]);
        }*/
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
comm);
        free(a);
    } else {
        Check_for_error(local_ok, fname, "Can't allocate temporary vector",
comm);
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
comm);
    }
}

```

```

} /* Read_vector */

// Print_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";

    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 IS: ", title);
        for (i = 0; i < n; i++)
            printf("%.2f ", 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 */

// Set_scale
-----

void Set_scale(
    int* scale /* out */,
    int my_rank /* in */,
    MPI_Comm comm /* in */)
{
    if (my_rank == 0) {
        printf("Set the scalar:\n");
    }
}

```

```

        scanf("%d", &scale);
    }
    MPI_Bcast(&scale, 1, MPI_INT, 0, comm);
}

```

A Figura 4 apresenta a saída do Código 4, considerando $n = 8$ (a ordem dos vetores) e $\text{comm_sz} = 4$, logo n é divisível por comm_sz .

Figura 4. Execução do Código 4

```

eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_39$ mpiexec -n 4 ./mpi_vector_39
What's the order of the vectors?
8
Enter the vector ONE
1 2 3 4 5 6 7 8
ONE IS: 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00
Set the scalar:
5
THE VECTOR ONE MULTIPLIED BY THE SCALAR IS: 5.00 10.00 15.00 20.00 25.00 30.00 35.00 40.00
Enter the vector TWO
1 2 3 4 5 6 7 8
TWO IS: 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00
THE NORM OF VECTOR TWO IS 14.28

```

- QUESTÃO 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.

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

- Sem utilizar MPI_Scan

(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

(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.

RESPOSTA A _____

Código 5. Código “vector_prefixes”

```
C/C++
#include <stdio.h>
#include <stdlib.h>

void Read_n(int* n_p);
void Allocate_vector(double** x_pp, int n);
void Read_vector(double a[], int n, char vec_name[]);
void Print_vector(double b[], int n, char title[]);
void Element_prefixes(double d[], double c[], int n);

/*-----*/
int main(void) {
    int n;
    double *vector_x, *vector_result;

    Read_n(&n);
    Allocate_vector(&vector_x, n);
    Read_vector(vector_x, n, "x");
    Print_vector(vector_x, n, "VECTOR X:");

    Allocate_vector(&vector_result, n);
    for(int i=0; i < n; i++){
        Element_prefixes(vector_result, vector_x, i);
    }

    Print_vector(vector_result, n, "VECTOR RESULT:");

    free(vector_x);
    free(vector_result);

    return 0;
} /* main */

//Read_n
-----

void Read_n(int* n_p /* out */) {
    printf("What's the order of the vectors?\n");
    scanf("%d", n_p);
    if (*n_p <= 0) {
        fprintf(stderr, "Order should be positive\n");
        exit(-1);
    }
} /* Read_n */
```


//Allocate_vector

```
-----  
void Allocate_vector(  
    double** x_pp /* out */,  
    int      n     /* in  */) {  
    *x_pp = malloc(n*sizeof(double));  
    if (*x_pp == NULL) {  
        fprintf(stderr, "Can't allocate vectors\n");  
        exit(-1);  
    }  
} /* Allocate_vectors */
```

//Read_vector

```
-----  
void Read_vector(  
    double a[] /* out */,  
    int    n    /* in  */,  
    char   vec_name[] /* in  */) {  
    int i;  
    printf("Enter the vector %s\n", vec_name);  
    for (i = 0; i < n; i++)  
        scanf("%lf", &a[i]);  
} /* Read_vector */
```

//Print_vector

```
-----  
void Print_vector(  
    double b[] /* in */,  
    int    n    /* in */,  
    char   title[] /* in */) {  
    int i;  
    printf("%s\n", title);  
    for (i = 0; i < n; i++)  
        printf("%.2f ", b[i]);  
    printf("\n");  
} /* Print_vector */
```

//Element_prefixes

```
-----  
void Element_prefixes(  
    double d[] /* out */,  
    double c[] /* in  */,  
    int    n    /* in  */) {  
    d[n] = 0;  
    for(int i = 0; i <= n; i++){  
        d[n] += c[i];  
    }  
}
```

A Figura 5 apresenta a saída do Código 5, considerando o algoritmo serial, com $n = 8$ (a ordem dos vetores).

Figura 5. Execução do Código 5

```
c:\Users\eriky\OneDrive\Documentos\PCD\UNID. III\LISTA III\questao_40\output>.\"vector_prefixes.exe"
What's the order of the vectors?
8
Enter the vector x
1 2 3 4 5 6 7 8
VECTOR X:
1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00
VECTOR RESULT:
1.00 3.00 6.00 10.00 15.00 21.00 28.00 36.00
```

RESPOSTA B

Código 6. Código “mpi_vector_b_prefixes.c”

```
C/C++
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>

void Check_for_error(int local_ok, char fname[], char message[], MPI_Comm comm);
void Read_n(int* n_p, int my_rank, int comm_sz, MPI_Comm comm);
void Allocate_vector(double** x_pp, int n, MPI_Comm comm);
void Read_vector(double a[], int n, char vec_name[], int my_rank, MPI_Comm comm);

/*-----*/
int main(void) {
    int n_total, comm_sz, my_rank, i;
    double *vector_x, *vector_result;
    double location = 0;

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

    Read_n(&n_total, my_rank, comm_sz, MPI_COMM_WORLD);
    Allocate_vector(&vector_x, n_total, MPI_COMM_WORLD);
    Read_vector(vector_x, n_total, "VECTOR X", my_rank, MPI_COMM_WORLD);
    for(i = 0; i <= my_rank; i++){
        location += vector_x[i];
    }
    Allocate_vector(&vector_result, n_total, MPI_COMM_WORLD);
```

```

    if(my_rank==0){
        vector_result[0] = location;
        for(i = 1; i < n_total; i++){
            MPI_Recv(&vector_result[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
NULL);
        }
    }else{
        MPI_Send(&location, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }

    if(my_rank==0){
        printf("VECTOR RESULT: ");
        for (i = 0; i < n_total; i++)
        {
            printf("%.2f ", vector_result[i]);
        }
        printf("\n");
        free(vector_result);
        free(vector_x);
        MPI_Finalize();
        return 0;
    } /* main */

// Check_for_error
-----

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

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

```

//Read_n

```
void Read_n(
    int*      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) local_ok = 0;
    Check_for_error(local_ok, fname, "n should be > 0 and evenly divisible by comm_sz", comm);
} /* Read_n */
```

//Allocate_vector

```
void Allocate_vector(
    double**  x_pp  /* out */,
    int       n      /* in  */,
    MPI_Comm  comm   /* in  */) {
    int local_ok = 1;
    char* fname = "Allocate_vector";

    *x_pp = malloc(n*sizeof(double));

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

//Read_vector

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

    int i;
```

```

int local_ok = 1;
char* fname = "Read_vector";

if (a == NULL) local_ok = 0;
Check_for_error(local_ok, fname, "Vector not allocated", comm);

if (my_rank == 0) {
    printf("Enter the vector %s\n", vec_name);
    for (i = 0; i < n; i++){
        scanf("%lf", &a[i]);
    }
}
MPI_Bcast(a, n, MPI_DOUBLE, 0, comm);
} /* Read_vector */

```

A Figura 6 apresenta a saída do Código 6, considerando $n = 4$ (a ordem dos vetores) e $comm_sz = 4$, logo cada processo armazena um dos elementos de x .

Figura 6. Execução do Código 6

```

eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_40$ mpiexec -n 4 ./mpi_vector_b_prefixes
What's the order of the vectors?
4
Enter the vector VECTOR X
1 2 3 4
VECTOR RESULT: 1.00 3.00 6.00 10.00

```

RESPOSTA C

Código 7. Código “mpi_vector_c_prefixes.c”

```

C/C++
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <math.h>

void Check_for_error(int local_ok, char fname[], char message[], MPI_Comm comm);
void Read_n(int* k_p, int* n_p, int my_rank, int comm_sz, MPI_Comm comm);
void Allocate_vector(double** x_pp, int n, MPI_Comm comm);
void Read_vector(double a[], int n, char vec_name[], int my_rank, MPI_Comm comm);
void Element_prefixes(double d[], double c[], int n);
/*-----*/

```

```

int main(void) {
    int k, n, comm_sz, my_rank, i, start, duo_i, factor_one = 1 ,
    factor_two = 2, location, displacement;
    double *vector_x, *vector_result;

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

    Read_n(&k, &n, my_rank, comm_sz, MPI_COMM_WORLD);
    Allocate_vector(&vector_x, n, MPI_COMM_WORLD);
    if(my_rank==0){
        printf("%d ELEMENTS\n", n);
    }
    Read_vector(vector_x, n, "VECTOR X", my_rank, MPI_COMM_WORLD);
    displacement = n / comm_sz;
    Allocate_vector(&vector_result, n, MPI_COMM_WORLD);

    start = my_rank*displacement;
    for(i = my_rank*displacement; i<(start+displacement); i++){
        Element_prefixes(vector_result, vector_x, i);
    }

    while (factor_one < comm_sz)
    {
        if((my_rank % factor_one) == 0){
            if((my_rank % factor_two) == 0){
                duo_i = my_rank + factor_one;
                //printf("MY RANK: %d, MEU DUO %d\n", my_rank, duo_i);
                location = start+displacement;
                MPI_Recv(&vector_result[location], displacement, MPI_DOUBLE,
duo_i, 0, MPI_COMM_WORLD, NULL);
            }else{
                duo_i = my_rank - factor_one;
                MPI_Send(&vector_result[start], displacement, MPI_DOUBLE,
duo_i, 0, MPI_COMM_WORLD);
                //printf("MY RANK: %d, MEU DUO %d\n", my_rank, duo_i);
                //return 0;
            }
        }
        factor_one *= 2;
        factor_two *= 2;
        displacement *= 2;
    }
    if(my_rank == 0){
        printf("VECTOR RESULT: ");
        for(i = 0; i < n ; i++){

```

```

        printf("%.2f ", vector_result[i]);
    }
    printf("\n");
}

free(vector_result);
free(vector_x);
MPI_Finalize();
return 0;
} /* main */

// Check_for_error
-----

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

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

//Read_n
-----

void Read_n(
    int*      k_p        /* out */,
    int*      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("Enter the value of k?\n");
    scanf("%d", k_p);
    *n_p = pow(2, *k_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);
} /* Read_n */

```

```

//Allocate_vector

```

```

void Allocate_vector(
    double** x_pp /* out */,
    int n /* in */,
    MPI_Comm comm /* in */) {
    int local_ok = 1;
    char* fname = "Allocate_vector";

    *x_pp = malloc(n*sizeof(double));

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

```

```

//Read_vector

```

```

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

    int i;
    int local_ok = 1;
    char* fname = "Read_vector";

    if (a == NULL) local_ok = 0;
    Check_for_error(local_ok, fname, "Vector not allocated", comm);

    if (my_rank == 0) {
        printf("Enter the vector %s\n", vec_name);
        for (i = 0; i < n; i++){

```



```

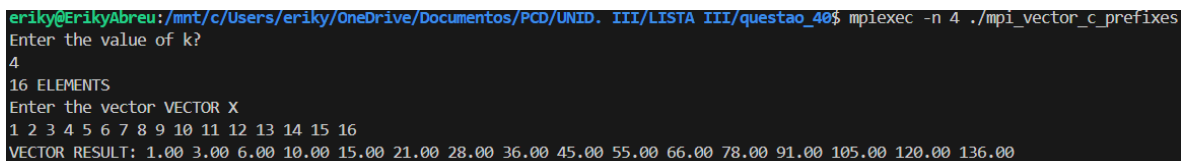
        scanf("%lf", &a[i]);
    }
}
MPI_Bcast(a, n, MPI_DOUBLE, 0, comm);
} /* Read_vector */

//Element_prefixes
-----
void Element_prefixes(
    double d[] /* out */,
    double c[] /* in */,
    int n /* in */) {
    d[n] = 0;
    for(int i = 0; i <= n; i++){
        d[n] += c[i];
    }
}

```

A Figura 7 apresenta a saída do Código 7, considerando $k = 4$, logo obtemos $n = 2^k = 16$. Adotando `comm_sz = 4`, obtemos n é divisível por `comm_sz`. Além disso, importante ressaltar, que o Código 7 apresenta uma troca de informações no formato de distribuição em árvore, garantindo apenas k fases de comunicação.

Figura 7. Execução do Código 7



```

eriky@ErikyAbreu: /mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_40$ mpiexec -n 4 ./mpi_vector_c_prefixes
Enter the value of k?
4
16 ELEMENTS
Enter the vector VECTOR X
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
VECTOR RESULT: 1.00 3.00 6.00 10.00 15.00 21.00 28.00 36.00 45.00 55.00 66.00 78.00 91.00 105.00 120.00 136.00

```

RESPOSTA D

Código 8. Código “`mpi_vector_d_scan.c`”

```

C/C++
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <time.h>

void Check_for_error(int local_ok, char fname[], char message[], MPI_Comm
comm);

```

```

void Read_n(int* n_p, int my_rank, int comm_sz, MPI_Comm comm);
void Allocate_vector(double** x_pp, int n, MPI_Comm comm);
/*-----*/
int main(void) {
    int k, n, comm_sz, my_rank, i;
    double *vector_x, *vector_result;

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

    Read_n(&n, my_rank, comm_sz, MPI_COMM_WORLD);
    Allocate_vector(&vector_x, n, MPI_COMM_WORLD);
    srand(time(NULL) + my_rank);
    for(i = 0; i < n; i++){
        vector_x[i] = rand() % 100;
    }

    Allocate_vector(&vector_result, n, MPI_COMM_WORLD);

    MPI_Scan(vector_x, vector_result, n, MPI_DOUBLE, MPI_SUM,
MPI_COMM_WORLD);

    for(i = 0; i < comm_sz; i++){
        if(my_rank == i){
            printf("I am process %d and these are my data\n", my_rank);
            printf("Vector: ");
            for(k = 0; k < n; k++){
                printf("%.2f ", vector_x[k]);
            }
            printf("\n");
            printf("Vector Result: ");
            for(k = 0; k < n; k++){
                printf("%.2f ", vector_result[k]);
            }
            printf("\n \n");
            fflush(stdout);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    free(vector_result);
    free(vector_x);
    MPI_Finalize();
    return 0;
} /* main */

```

```
// Check_for_error
```

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

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

```
//Read_n
```

```
void Read_n(
    int*      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("Enter the value of k?\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);
} /* Read_n */
```

```
//Allocate_vector
```

```
void Allocate_vector(
```

```

    double**  x_pp  /* out */,
    int       n     /* in  */,
    MPI_Comm  comm  /* in  */) {
int local_ok = 1;
char* fname = "Allocate_vector";

*x_pp = malloc(n*sizeof(double));

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

```

A Figura 8 apresenta a saída do Código 8, considerando $n = 8$ (a ordem dos vetores) e $\text{comm_sz} = 4$, realizando o uso da função de comunicação coletiva, `MPI_Scan`.

Figura 8. Execução do Código 8

```

eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_40$ mpiexec -n 4 ./mpi_vector_d_scan
Enter the value of k?
4
I am process 0 and these are my data
Vector: 87.00 25.00 82.00 27.00
Vector Result: 87.00 25.00 82.00 27.00

I am process 1 and these are my data
Vector: 88.00 25.00 71.00 3.00
Vector Result: 175.00 50.00 153.00 30.00

I am process 2 and these are my data
Vector: 13.00 63.00 82.00 74.00
Vector Result: 188.00 113.00 235.00 104.00

I am process 3 and these are my data
Vector: 81.00 6.00 1.00 34.00
Vector Result: 269.00 119.00 236.00 138.00

```

- QUESTÃO 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?

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

RESPOSTA A _____

Código 9. Código `allreduce` em anel

```
C/C++
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

void Allreduce_ring(int *total, int my_rank, int size, int *my_val, int n,
MPI_Comm comm);

/*-----*/
int main(int argc, char* argv[]) {
    int comm_sz, my_rank, n;
    int *my_vals = NULL;
    int *total = NULL;
    float time_start, time_finish, timer, time_a;

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

    if (argc != 2) {
        if (my_rank == 0)
```

```

        fprintf(stderr, "Uso: %s <tamanho_do_vetor>\n", argv[0]);
        MPI_Finalize();
        exit(1);
    }

    n = atoi(argv[1]);
    my_vals = (int*)malloc(n * sizeof(int));
    total = (int*)malloc(n * sizeof(int));

    srand(time(NULL) + my_rank);
    for (int i = 0; i < n; i++) {
        my_vals[i] = rand() % 100;
    }
    //printf("I am process %d and my val: ", my_rank);
    for (int i = 0; i < n; i++) {
        //printf("%d ", my_vals[i]);
    }
    printf("\n");
    fflush(stdout);

    if(my_rank==0){
        //printf("----Allreduce_ring----\n");
        fflush(stdout);
    }

    MPI_Barrier(MPI_COMM_WORLD);
    time_start = MPI_Wtime();
    Allreduce_ring(total, my_rank, comm_sz, my_vals, n, MPI_COMM_WORLD);
    time_finish = MPI_Wtime();
    timer = time_finish - time_start;
    MPI_Reduce(&timer, &time_a, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if(my_rank == 0){
        printf("TIME: %f\n", time_a);
    }

    //printf("Processo %d, total: ", my_rank);
    for (int i = 0; i < n; i++) {
        //printf("%d ", total[i]);
    }
    fflush(stdout);

    free(my_vals);
    MPI_Finalize();
    return 0;
} /* main */

```

```
// Allreduce_ring
-----

void Allreduce_ring(int *total, int my_rank, int size, int *my_vals, int n,
MPI_Comm comm) {
    int i;
    int *temp_vals = (int*)malloc(n * sizeof(int));
    int dest, source;

    dest = (my_rank + 1) % size;
    source = (my_rank - 1 + size) % size;

    if (my_rank == (size - 1)) {
        dest = 0;
    } else if (my_rank == 0) {
        source = (size - 1) % size;
    }

    for (i = 0; i < n; i++) {
        total[i] = temp_vals[i] = my_vals[i];
    }

    for (int step = 1; step < size; step++) {
        MPI_Sendrecv_replace(temp_vals, n, MPI_INT, dest, step, source, step,
comm, NULL);
        for(int i=0; i<n; i++){
            total[i] += temp_vals[i];
        }
    }
} /* Allreduce_ring */
```

Código 10. Código allreduce em borboleta

```
C/C++

void Allreduce_butterfly(int *total, int my_rank, int size, int *my_vals,
int n, MPI_Comm comm) {
    int etapas = (int)log2(size);
    int parceiro;
    int* temp_vals = (int*)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        total[i] = my_vals[i];
    }

    for (int i = 0; i < etapas; i++) {
        parceiro = my_rank ^ (1 << i);
```

```

    for (int j = 0; j < n; j++) {
        temp_vals[j] = total[j];
    }

    MPI_Sendrecv_replace(temp_vals, n, MPI_INT, parceiro, 0, parceiro,
0, comm, MPI_STATUS_IGNORE);

    for (int j = 0; j < n; j++) {
        total[j] += temp_vals[j];
    }
}

free(temp_vals);
}

```

Como observado na Figura 8, a diferença de desempenho observada é significativa: a versão baseada em comunicação borboleta executou a mesma tarefa em um menor tempo requerido pela versão em anel. Isso comprova a superioridade da abordagem em borboleta no que diz respeito à eficiência de comunicação em sistemas paralelos, especialmente quando o volume de dados por processo é elevado.

Figura 8. Comparação entre os tempos de execução

```

eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_41$ mpiexec -n 4 ./mpi_41_a 200000000
TIME: 17.132097
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_41$ mpiexec -n 4 ./mpi_41_a_borboleta 200000000
TEMPO: 7.011594 segundos

```

RESPOSTA B

Código 11. Código para somas de prefixos

```

C/C++
void Prefix_sum_ring(int *prefix, int my_rank, int size, int *my_val, int n,
MPI_Comm comm) {
    int *recv_buf = (int*)malloc(n * sizeof(int));
    int source = (my_rank - 1 + size) % size;
    int dest = (my_rank + 1) % size;

    for (int i = 0; i < n; i++)
        prefix[i] = my_val[i];
}

```



```

int *temp_sum = (int*)malloc(n * sizeof(int));
if (my_rank == 0) {
    for (int i = 0; i < n; i++)
        temp_sum[i] = my_val[i];
    MPI_Send(temp_sum, n, MPI_INT, dest, 0, comm);
} else {
    MPI_Recv(recv_buf, n, MPI_INT, source, 0, comm, MPI_STATUS_IGNORE);
    for (int i = 0; i < n; i++)
        prefix[i] = recv_buf[i] + my_val[i];

    if (my_rank != size - 1) {
        for (int i = 0; i < n; i++)
            temp_sum[i] = prefix[i];
        MPI_Send(temp_sum, n, MPI_INT, dest, 0, comm);
    }
}

free(recv_buf);
free(temp_sum);
}

```

- QUESTÃO 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`.

RESPOSTA _____

Código 9. Código “`mpi_vector_42.c`”

```
C/C++
#include <stdio.h>
#include <mpi.h>
#include <stdlib.h>
#include <math.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_vector(double** local_x_pp, int local_n, MPI_Comm comm);
void Read_vector(double local_a[], int sendcounts[], int display[], int
local_n, int n, char vec_name[], int my_rank, MPI_Comm comm);
void Print_vector(double local_b[], int sendcounts[], int display[], int
local_n, int n, char title[], int my_rank, MPI_Comm comm);
void Set_scale(int* scale, int my_rank, MPI_Comm comm);

int main(void) {
    int my_rank, comm_sz, n_vector, local_n_vector, scale, *vector_send,
*vector_display;
    double *local_vector_1, *local_vector_2;
    double sum = 0, normalization=0;

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

    Read_n(&n_vector, &local_n_vector, my_rank, comm_sz, MPI_COMM_WORLD);

    Allocate_vector(&local_vector_1, local_n_vector, MPI_COMM_WORLD);

    if (my_rank == 0) {
        vector_send = malloc(comm_sz * sizeof(int));
        vector_display = malloc(comm_sz * sizeof(int));
```

```

    }

    MPI_Gather(&local_n_vector, 1, MPI_INT, vector_send, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    if (my_rank == 0){
        vector_display[0] = 0;
        for(int i = 0; i < comm_sz-1; i++){
            vector_display[i+1] = vector_display[i] + vector_send[i];
        }
    }

    Read_vector(local_vector_1, vector_send, vector_display, local_n_vector,
n_vector, "ONE", my_rank, MPI_COMM_WORLD);

    Print_vector(local_vector_1, vector_send, vector_display, local_n_vector,
n_vector, "ONE", my_rank, MPI_COMM_WORLD);

    Set_scale(&scale, my_rank, MPI_COMM_WORLD);

    for(int i = 0; i < local_n_vector; i++){
        local_vector_1[i] = local_vector_1[i]*scale;
    }

    Print_vector(local_vector_1, vector_send, vector_display, local_n_vector,
n_vector, "THE VECTOR ONE MULTIPLIED BY THE SCALAR", my_rank,
MPI_COMM_WORLD);

    //START VECTOR TWO
    Allocate_vector(&local_vector_2, local_n_vector, MPI_COMM_WORLD);

    Read_vector(local_vector_2, vector_send, vector_display, local_n_vector,
n_vector, "TWO", my_rank, MPI_COMM_WORLD);

    Print_vector(local_vector_2, vector_send, vector_display, local_n_vector,
n_vector, "TWO", my_rank, MPI_COMM_WORLD);

    for(int i = 0; i < local_n_vector; i++){
        local_vector_2[i] = local_vector_2[i]*local_vector_2[i];
        sum += local_vector_2[i];
    }

    MPI_Reduce(&sum, &normalization, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

    if(my_rank == 0){
        normalization = sqrt(normalization);
        printf("THE NORM OF VECTOR TWO IS %.21f\n", normalization);
    }

```

```

    }

    free(local_vector_1);
    free(local_vector_2);
    if(my_rank==0){
        free(vector_send);
    }
    MPI_Finalize();
    return 0;
} /* main */

// Check_for_error
-----

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

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

// Read_n
-----

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, resto;
    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 and evenly divisible by
comm_sz", comm);
    resto = *n_p % comm_sz;
    if(resto == 0){
        *local_n_p = *n_p/comm_sz;
    }else{
        if(my_rank<resto){
            *local_n_p = (*n_p/comm_sz) +1;
        }else{
            *local_n_p = *n_p/comm_sz;
        }
    }
}
/* Read_n */

// Allocate_vector
-----

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

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

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

// Read_vector
-----

void Read_vector(
    double    local_a[]    /* out */,
    int       sendcounts[] ,
    int       display[],
    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]);
    }
    MPI_Scatterv(a, sendcounts, display, 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_Scatterv(a, sendcounts, display, MPI_DOUBLE, local_a, local_n,
MPI_DOUBLE, 0, comm);
}
} /* Read_vector */

// Print_vector
-----

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

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

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

```

```

        MPI_Gatherv(local_b, local_n, MPI_DOUBLE, b, recvcunts, displs,
MPI_DOUBLE, 0, comm);
        printf("%s IS: ", title);
        for (i = 0; i < n; i++)
            printf("%.2f ", b[i]);
        printf("\n");
        free(b);
    } else {
        Check_for_error(local_ok, fname, "Can't allocate temporary vector",
comm);
        MPI_Gatherv(local_b, local_n, MPI_DOUBLE, b, recvcunts, displs,
MPI_DOUBLE, 0, comm);
    }
} /* Print_vector */

// Set_scale
-----

void Set_scale(
    int* scale          /* out */,
    int my_rank         /* in */,
    MPI_Comm comm       /* in */)
{
    if (my_rank == 0) {
        printf("Set the scalar:\n");
        scanf("%d", scale);
    }
    MPI_Bcast(scale, 1, MPI_INT, 0, comm);
}

```

A Figura 9 apresenta a saída do Código 9, considerando $n = 7$ (a ordem dos vetores) e $\text{comm_sz} = 4$, logo, obtém-se um valor de n não divisível por comm_sz .

Figura 9. Execução do Código 4

```

eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_42$ mpiexec -n 4 ./mpi_vector_42
What's the order of the vectors?
7
Enter the vector ONE
1 2 3 4 5 6 7
ONE IS: 1.00 2.00 3.00 4.00 5.00 6.00 7.00
Set the scalar:
2
THE VECTOR ONE MULTIPLIED BY THE SCALAR IS: 2.00 4.00 6.00 8.00 10.00 12.00 14.00
Enter the vector TWO
7 6 5 4 3 2 1
TWO IS: 7.00 6.00 5.00 4.00 3.00 2.00 1.00
THE NORM OF VECTOR TWO IS 11.83

```

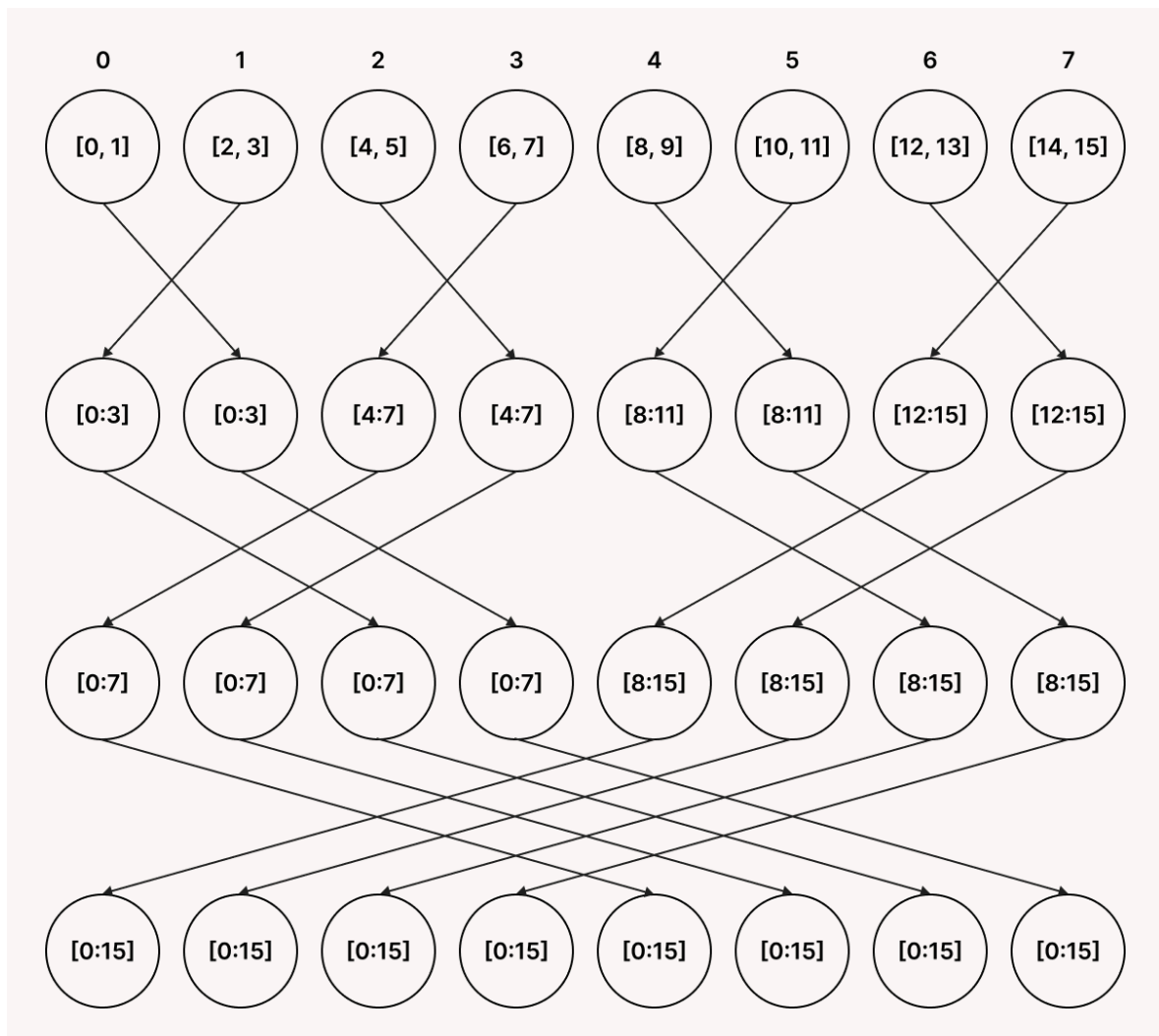
• QUESTÃO 43

Suponha que $\text{comm_sz} = 8$ e o vetor $x = (0, 1, 2, \dots, 15)$ 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 .

RESPOSTA

O MPI_Allgather permite que cada processo envie seus próprios dados para todos os outros processos, e no final todos os processos tenham uma cópia completa dos dados de todos os processos. A Figura 10 ilustra as etapas de uma implementação borboleta da função, supondo que obtenhamos um $\text{comm_sz} = 8$ e o vetor $x = (0, 1, 2, \dots, 15)$, sendo o mesmo distribuído em blocos para os processos.

Figura 10. Diagrama ilustrando uma implementação borboleta



- QUESTÃO 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`.

RESPOSTA

Código 10. Funções modificadas do código “`mpi_vector_add_44.c`”

```
C/C++
int main(void) {
    int n, local_n;
    int comm_sz, my_rank;
    double *local_x, *local_y, *local_z;
    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);

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

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

    Parallel_vector_sum(local_x, local_y, local_z, local_n);
    Print_vector(local_z, local_n, n, "The sum is", my_rank, comm, BLOCK);
}
```

```

MPI_Type_free(&BLOCK);
free(local_x);
free(local_y);
free(local_z);

MPI_Finalize();

return 0;
} /* main */

//Read_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 BLOCK      /* 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]);
        MPI_Scatter(a, 1, BLOCK, local_a, 1, BLOCK, 0,
            comm);
        free(a);
    } else {
        Check_for_error(local_ok, fname, "Can't allocate temporary vector",
            comm);
        MPI_Scatter(a, 1, BLOCK, local_a, 1, BLOCK, 0,
            comm);
    }
} /* Read_vector */

//Print_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 BLOCK     /* 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, 1, BLOCK, b, 1, BLOCK,
              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, BLOCK, b, 1, BLOCK, 0,
              comm);
}
} /* Print_vector */

```

A Figura 11 representa a execução do código “mpi_vector_add_44.c” modificado para o MPI_Scatter e MPI_Gather trabalhar apenas com um elemento do MPI_Type BLOCK, criado com a função MPI_Type_contiguous. No Código 10 são apresentadas as funções do código que sofreram modificações.

Figura 11. Execução do código “mpi_vector_add_44.c” (modificado)

```
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_44$ mpiexec -n 4 ./mpi_vector_add_44.c
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
```

- QUESTÃO 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

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

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.

RESPOSTA

Código 11. Código “`mpi_vector_45.c`”

```
C/C++  
  
#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 my_rank, int comm_sz, MPI_Comm comm);  
void Print_matrix(double *matrix, int n);  
void Print_matrix_up(double *matrix, int n);  
  
/*-----*/
```

```

int main(void) {
    int n, comm_sz, my_rank, *blocklengths, *displacements;
    double *matrix, *matrix_tri_up;
    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, my_rank, comm_sz, comm);
    blocklengths = malloc(n * sizeof(double));
    displacements = malloc(n * sizeof(double));
    for (int i = 0; i < n; i++) {
        blocklengths[i] = n - i;
        displacements[i] = i * n + i;
    }

    MPI_Datatype upper_triangle_type;
    MPI_Type_indexed(n, blocklengths, displacements, MPI_DOUBLE,
&upper_triangle_type);
    MPI_Type_commit(&upper_triangle_type);

    if (my_rank == 0) {
        matrix = malloc(n * n * sizeof(double));

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                matrix[i * n + j] = rand()%100;
            }
        }

        printf("FULL MATRIZ:\n");
        Print_matrix(matrix, n);

        MPI_Send(matrix, 1, upper_triangle_type, 1, 0, MPI_COMM_WORLD);

    } else if (my_rank == 1) {
        int num_elements = n * (n + 1) / 2;
        matrix_tri_up = malloc(num_elements * sizeof(double));

        MPI_Recv(matrix_tri_up, num_elements, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD, NULL);

        printf("\nUPPER TRIANGULAR MATRIX:\n");

        Print_matrix_up(matrix_tri_up, n);
    }
}

```

```

        free(matrix_tri_up);
    }

    if (matrix){
        free(matrix);
    }
    free(blocklengths);
    free(displacements);
    MPI_Type_free(&upper_triangle_type);
    MPI_Finalize();
    return 0;
}

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

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

//Read_n-----
void Read_n(
    int*      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 headquarters?\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);
} /* Read_n */

//Print_matrix-----
-----
void Print_matrix(
    double *matrix /* in */,
    int n /* in */) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++){
            printf("%.2f\t", matrix[i * n + j]);
        }
        printf("\n");
    }
} /* Print_vector */

//Print_matrix_up-----
-----
void Print_matrix_up(
    double *matrix_up /* in */,
    int n /* in */) {

    int k = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (j < i){
                printf("-\t");
            } else {
                printf("%.2f\t", matrix_up[k++]);
            }
        }
        printf("\n");
    }
} /* Print_vector_up */

```

A Figura 12 representa a execução do Código 11. Supondo que o valor de $n = 4$, e compreendendo que os vetores `blocklengths` e `displacements` foram obtidos da seguinte maneira:

```
for (int i = 0; i < n; i++) {
```



```

        blocklengths[i] = n - i;
        displacements[i] = i * n + i;
    }

```

Foi possível utilizar a função `MPI_Type_indexed` para criar um novo tipo e facilitar a comunicação entre o processo 0 e 1.

Figura 12. Execução do Código 11



```

eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_45$ mpiexec -n 4 ./mpi_vector_45
What's the order of the headquarters?
4
FULL MATRIZ:
83.00  86.00  77.00  15.00
93.00  35.00  86.00  92.00
49.00  21.00  62.00  27.00
90.00  59.00  63.00  26.00

UPPER TRIANGULAR MATRIX:
83.00  86.00  77.00  15.00
-      35.00  86.00  92.00
-      -      62.00  27.00
-      -      -      26.00

```

• QUESTÃO 46

As funções `MPI_Pack` e `MPI_Unpack` fornecem uma alternativa aos tipos de dados derivados 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.

RESPOSTA _____

O `MPI_Pack` é uma função que permite empacotar dados de diferentes tipos e

estruturas em um único buffer contínuo de memória. No Código 12, ele é utilizado para empacotar os três valores, `double a_p`, `double b_p` e `int n_p`, em um único *buffer*, permitindo a realização de um único `MPI_Bcast` para disseminar essas informações a todos os processos, reduzindo a quantidade de chamadas de comunicação e melhorando a eficiência.

Código 12. Função `Get_input` do código “`mpi_trap_46.c`”

```
C/C++
//Get_input-----
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p, int* n_p)
{
    int size_a, size_b, size_n, total_size;
    int position = 0;
    char* pack_buf;

    MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &size_a);
    MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &size_b);
    MPI_Pack_size(1, MPI_INT, MPI_COMM_WORLD, &size_n);

    total_size = size_a + size_b + size_n;
    pack_buf = (char*) malloc(total_size);

    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, total_size, &position,
MPI_COMM_WORLD);
        MPI_Pack(b_p, 1, MPI_DOUBLE, pack_buf, total_size, &position,
MPI_COMM_WORLD);
        MPI_Pack(n_p, 1, MPI_INT, pack_buf, total_size, &position,
MPI_COMM_WORLD);
    }
    MPI_Bcast(pack_buf, total_size, MPI_PACKED, 0, MPI_COMM_WORLD);

    position = 0;

    MPI_Unpack(pack_buf, total_size, &position, a_p, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
    MPI_Unpack(pack_buf, total_size, &position, b_p, 1, MPI_DOUBLE,
MPI_COMM_WORLD);
    MPI_Unpack(pack_buf, total_size, &position, n_p, 1, MPI_INT,
MPI_COMM_WORLD);

    free(pack_buf);
} /* Get_input */
```

Figura 13. Execução do código “mpi_trap_46.c”

```
eriky@ErikyAbreu:/mnt/c/Users/eriky/OneDrive/Documentos/PCD/UNID. III/LISTA III/questao_46$ mpiexec -n 4 ./mpi_trap_46
Enter a, b, and n
3 9 100
With n = 100 trapezoids, our estimate
of the integral from 3.000000 to 9.000000 = 2.340036000000000e+02
```

• QUESTÃO 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) .

- Qual critério você utilizou para escolher n ?
- Como os tempos mínimos se comparam aos tempos médios e medianos?
- Quais são os *speedups*?
- Quais são as eficiências?
- Com base nos dados que você coletou, você diria que a regra dos trapézios é escalável?

RESPOSTA A _____

Os valores de n escolhidos para os testes foram 2.147.483.648, 4.294.967.296, 8.589.934.592 e 17.179.869.184 (equivalentes a 2^{31} , 2^{32} , 2^{33} e 2^{34}). O valor mínimo foi adotado visando garantir que o tempo de execução de cada processo fosse superior a um segundo, uma vez que tempos inferiores podem estar mais sujeitos a variações e ruídos do sistema, comprometendo a precisão das medições. A partir disso, os valores foram duplicados a cada novo teste, de forma a observar o comportamento do desempenho em diferentes escalas de carga computacional.

RESPOSTA B _____

Os valores de tempos mínimos, no geral, não destoaram muito dos valores de tempo médio e medianos, expondo que o número de outliers foram pequenos, mesmo assim, os dados no qual foram realizadas as análises de *speedups* e eficiências, foram obtidos através da mediana, como exposto na Tabela 6.

Tabela 4. Tempos mínimos da execução

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	5,6704	12,1644	20,7907	42,2647
2	3,2517	6,7918	11,2628	22,9998
4	2,1600	4,4591	6,5513	13,363

Tabela 5. Tempos médios da execução

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	6,12018	12,48862	20,96136	42,83886
2	3,73772	7,22326	11,53298	23,38672
4	2,24694	4,50124	6,67698	13,40696

Tabela 6. Tempos medianos da execução

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	6,0597	12,5721	20,8969	42,6701
2	3,6865	7,2944	11,5223	23,4017
4	2,2202	4,5017	6,6425	13,3833

RESPOSTA CTabela A. *Speedups*

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	1	1	1	1
2	1,6438	1,7235	1,8136	1,8234
4	2,7293	2,7927	3,1459	3,1883

RESPOSTA D

Tabela B. Eficiências

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	1	1	1	1
2	0,8219	0,8618	0,9068	0,9117
4	0,6823	0,6982	0,7865	0,7971

RESPOSTA E

Baseado nos dados apresentados na Tabela B, observamos que a eficiência do programa diminui à medida que o número de processos aumenta, especialmente para quantidades menores de problemas. Essa queda de eficiência sugere que o programa pode não ser escalável em sua forma atual, pelo menos para os tamanhos de problema testados. No entanto, para avaliar corretamente a escalabilidade, seria necessário executar o programa em um ambiente com maior número de núcleos.

• QUESTÃO 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

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

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.

RESPOSTA

Utilizando como auxílio a linguagem de programação Python, foi possível obter os valores dos coeficientes de $a = 2,4479 * 10^{-9}$, $b = 0,4609$ e um fator adicional $c = 0,9442$, logo a fórmula adaptada, adicionando os parâmetros obtidos é:

$$T_{parallel}(n, p) \approx 2,4479 * 10^{-9} * \frac{n}{p} + 0,4609 * \log_2(p) + 0,9442$$

O valor de a representa que quanto maior o $\frac{n}{p}$, maior o tempo de execução, como esperado. O valor de b evidencia que à medida que o número de processos aumenta, há um ganho marginal, mas também um custo de comunicação.

Tabela C. Tempos medianos da execução

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	6,0597	12,5721	20,8969	42,6701
2	3,6865	7,2944	11,5223	23,4017
4	2,2202	4,5017	6,6425	13,3833

Tabela D. Tempos estimados

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	6,2009	11,4577	21,9712	42,9982
2	4,0335	6,6619	11,9186	22,4321
4	3,1802	4,4944	7,1228	12,3795

Realizando a comparação entre os tempos expostos na Tabela C e Tabela D, podemos concluir que a equação possui uma boa aderência geral, a maioria das estimativas está muito próxima dos valores reais, especialmente para $\log_2(p) = 0$ e $\log_2(p) = 1$. Isso mostra que o modelo representa bem os dados. Os erros absolutos geralmente estão abaixo de 1 segundo, como exposto na Tabela E.

Tabela E. Erros absolutos

Quantidade de Processos	Números de trapézios			
	2^{31}	2^{32}	2^{33}	2^{34}
1	0,141	1,114	1,074	0,328
2	0,347	0,633	0,396	0,969
4	0,978	0,007	0,480	1,003

• QUESTÃO 49

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

- (a) O programa obtém *speedups* lineares?
- (b) É escalável?
- (c) É fortemente escalável?
- (d) É fracamente escalável?

RESPOSTA _____

Os dados apresentados foram obtidos a partir da mediana de cinco execuções consecutivas para cada combinação de valores de p (número de processos) e n (tamanho do vetor), utilizando o código `mpi_odd_even.c`. Os valores de p foram definidos como potências de 2, limitados a 4 em função da quantidade de núcleos disponíveis na máquina utilizada. O menor valor de n foi determinado como a menor potência de 2 que resultasse em tempo de execução superior a um segundo com 4 processos, visando garantir medições mais estáveis e representativas.

Tabela 7. Tempos medianos da execução (segundos)

Quantidade de Processos	Componentes			
	2^{26}	2^{27}	2^{28}	2^{29}
1	7.4176	15.6206	31.3949	64.3917
2	4.0665	8.4230	17.5015	36.3644
4	2.4720	5.0496	10.4648	22.8861

Tabela 8. Speedups

Quantidade de Processos	Componentes			
	2^{26}	2^{27}	2^{28}	2^{29}
1	1	1	1	1
2	1,8241	1,8545	1,7938	1,7707
4	3,0006	3,0934	3,0000	2,8136

Tabela 9. Eficiências

Quantidade de Processos	Componentes			
	2^{26}	2^{27}	2^{28}	2^{29}
1	1	1	1	1
2	0,9120	0,9273	0,8969	0,8854
4	0,7502	0,7734	0,7500	0,7034

RESPOSTA A

O programa não obtém *speedups* lineares, como observado na Tabela 7, o tempo de execução não reduz na mesma proporção que o número de processos aumenta, em outras palavras, como observado na Tabela 8, o valor de p é consideravelmente diferente do T_{serial} dividido pelo $T_{\text{paralelizado}}$.

RESPOSTA B

Em geral, um problema é escalável se puder lidar com tamanhos de problema cada vez maiores, sem perdas consideráveis na eficiência, logo, nota-se baseado nos valores obtidos na Tabela 9, que em todas as colunas a eficiência é reduzida a medida que se aumenta a quantidade de processos.

RESPOSTA C

Se aumentarmos o número de processos e mantivermos a eficiência fixa sem aumentar o tamanho do problema, o problema será fortemente escalável. Baseado na Tabela 9, pode-se observar que para todos os 4 valores empregados aos tamanhos dos problemas, ocorre a redução da eficiência, logo, a redução não é fortemente escalável.

RESPOSTA D

Analisando as diagonais da Tabela 9, pode-se notar que a eficiência não se manteve fixa aumentando o tamanho do problema na mesma proporção em que aumentamos o número de processos, logo o problema não será fracamente escalável.

• QUESTÃO 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?

RESPOSTA _____

As Tabelas 10 e 11 apresentam os tempos medianos de cinco execuções para cada par de valores de entrada. Elas comparam o desempenho das funções `Merge_low` e `Merge_high` ao utilizarem duas abordagens distintas para copiar os dados do vetor temporário para o vetor principal: a utilização da função `memcpy` (Tabela 11) e a troca simples de ponteiros (Tabela 10).

Tabela 10. Tempos medianos da execução com ponteiros(segundos)

Quantidade de Processos	Componentes			
	2^{26}	2^{27}	2^{28}	2^{29}
1	7.3150	15.1524	30.5372	64.0336
2	4.1691	8.3334	17.0678	36.1292
4	2.4884	5.0838	10.2981	22.4900

Tabela 11. Tempos medianos da execução com a função `memcpy` (segundos)

Quantidade de Processos	Componentes			
	2^{26}	2^{27}	2^{28}	2^{29}
1	7.4176	15.6206	31.3949	64.3917
2	4.0665	8.4230	17.5015	36.3644
4	2.4720	5.0496	10.4648	22.8861

A análise geral dos dados evidencia que, especialmente para conjuntos com maior quantidade de elementos, a estratégia de troca de ponteiros resultou em tempos de execução menores, demonstrando maior eficiência na maioria dos testes realizados.

Código 13. Modificações realizadas nas funções Merge_low e Merge_high

```
C/C++
void Merge_low(int** my_keys_pp, int* recv_keys, int** temp_keys_pp, int
local_n) {
    int m_i, r_i, t_i;
    int* temp;

    m_i = r_i = t_i = 0;
    while (t_i < local_n) {
        if ((*my_keys_pp)[m_i] <= recv_keys[r_i]) {
            (*temp_keys_pp)[t_i] = (*my_keys_pp)[m_i];
            t_i++; m_i++;
        } else {
            (*temp_keys_pp)[t_i] = recv_keys[r_i];
            t_i++; r_i++;
        }
    }

    temp = *my_keys_pp;
    *my_keys_pp = *temp_keys_pp;
    *temp_keys_pp = temp;
} /* Merge_low */

/*-----*/
void Merge_high(int** local_A_pp, int* temp_B, int** temp_C_pp, int local_n)
{
    int ai, bi, ci;
    int* temp;

    ai = local_n-1;
    bi = local_n-1;
    ci = local_n-1;
    while (ci >= 0) {
        if ((*local_A_pp)[ai] >= temp_B[bi]) {
            (*temp_C_pp)[ci] = (*local_A_pp)[ai];
            ci--; ai--;
        } else {
            (*temp_C_pp)[ci] = temp_B[bi];
            ci--; bi--;
        }
    }

    temp = *local_A_pp;
    *local_A_pp = *temp_C_pp;
    *temp_C_pp = temp;
} /* Merge_high */
```