UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO – UFERSA
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS – CMPF
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

PEX1272 - PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA - T01

Bruno Victor Paiva Da Silva

Lista para avaliação da 3ª unidade

Capitulo 03

3.9 EXERCISES

Questão 3.2 - **RESPONDIDA**
Questão 3.4 - **RESPONDIDA**
Questão 3.6 - **RESPONDIDA**
Questão 3.9 - **RESPONDIDA**
Questão 3.11 - **RESPONDIDA**
Questão 3.12 - **RESPONDIDA**
Questão 3.13 - **RESPONDIDA**
Questão 3.16 - **RESPONDIDA**
Questão 3.17 - **RESPONDIDA**
Questão 3.19 - **RESPONDIDA**
Questão 3.20 - **RESPONDIDA**
Questão 3.22 - **RESPONDIDA**
Questão 3.23 - **NÃO RESPONDIDA**
Questão 3.27 - **RESPONDIDA**
Questão 3.28 - **RESPONDIDA**

3.2. Modifique a regra trapezoidal para que ela estime corretamente a integral, mesmo que comm_sz não seja divisível por "n". (Você ainda pode assumir que n ≥ comm_sz.)

O problema no cálculo da integral quando comm_sz não é divisível por “n” acontece, pois há trapézios que não são considerados no cálculo, para resolver isso, são distribuídas fatias a mais - fatias essas que não seriam incluídas no cálculo - para os processos iniciais de maneira uniforme.

Para isso é feita uma diferença no cálculo do local_n, local_a e consequentemente no local_b quando o comm_sz não é divisível pelo número de trapézios.

```C/C++
over = (n % comm_sz);

h = (b-a)/n;
local_n = (n / comm_sz);

if(my_rank < over){
      local_n++;
      local_a = a + my_rank * local_n * h;
      local_b = local_a + local_n * h;
} else {
      local_a = a + (my_rank * local_n + over) * h;
      local_b = local_a + local_n * h;
}

local_int = Trap(local_a, local_b, local_n, h);
```

3.4. 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 0s primeiro, depois processe 1s e assim por diante.
Há diferentes formas garantir tal ordem, uma delas é fazendo com que todas as impressões seja feita pelo processo 0, onde os outros iriam mandar a string para ele e ele se garantiria (com auxílio de um for) a impressão na ordem correta.

```C/C++
if(my_rank != 0){
      sprintf(question, "Proc %d of %d > Does anyone have a
      toothpick?", my_rank, comm_sz);
```

```
      MPI_Send(question, strlen(question)+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 i = 1; i < comm_sz; i++){
            MPI_Recv(question, MAX_STRING, MPI_CHAR, i, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", question);
      }
}
```

Outra maneira mais simples seria utilizar MPI_Barrier para garantir a sincronização dos processos.

```C/C++
for (int i = 0; i < comm_sz; i++){
      MPI_Barrier(MPI_COMM_WORLD);
      if(my_rank == i){
            printf("Proc %d of %d > Does anyone have a
      toothpick?\n", my_rank, comm_sz);
      }
}
```

3.6. Suponha que comm_sz = 4 e suponha que x seja um vetor com n = 14 componentes.

a. Como os componentes de x seriam distribuídos entre os processos em um programa que usasse uma distribuição em blocos?

Como comm_sz = 4 e n = 14, vemos que caso dividirmos os componentes em blocos iguais, acabariam faltando componentes a serão utilizados. Diante disso, uma abordagem para isso seria redistribuir alguns componentes para alguns processos, com isso uns teriam blocos maiores que outros.

Processo	Componente
0	0, 1, 2 e 3

1	4, 5, 6 e 7
2	8, 9, 10
3	11, 12 e 13

b. Como os componentes de x seriam distribuídos entre os processos em um programa que usasse uma distribuição cíclica?

Da mesma forma, alguns processos iram executar/manipular/processar componentes a mais que outros nessa distribuição também.

Processo	Componente
0	0, 4, 8 e 12
1	1, 5, 9 e 13
2	2, 6 e 10
3	3, 7 e 11

c. Como os componentes de x seriam distribuídos entre os processos em um programa que usasse uma distribuição cíclica de blocos com tamanho de bloco b = 2?

Da mesma forma, alguns processos iram executar/manipular/processar componentes a mais que outros nessa distribuição também, mas agora de maneira diferente.

Processo	Componente
0	0, 1, 8 e 9
1	2, 3, 10 e 11
2	4, 5, 12 e 13
3	6 e 7

Você deve tentar tornar suas distribuições gerais para poderem ser usadas independentemente de quais sejam comm_sz e n. Você também deve tentar fazer suas distribuições “justas” de modo que se "q" e “r” forem dois processos quaisquer, a diferença entre o número de componentes atribuídos a "q" e o número de componentes atribuídos a “r” seja tão pequena quanto possível.

3.9. Escreva um programa MPI que implemente a multiplicação de um vetor por um escalar e um produto escalar. O usuário deve inserir dois vetores e um escalar, todos lidos pelo processo 0 e distribuídos entre os processos. Os resultados são calculados e coletados no processo 0, que os imprime. Você pode assumir que "n", a ordem dos vetores, é divisível igualmente por comm_sz.

Como "n" será divisível por comm_sz, a divisão dos blocos serão iguais para todos os processos. Com isso, podemos utilizar o MPI_Scatter para distribuir os valores (a, b) para os processos.

```C/C++
int main(void) {
      int my_rank, comm_sz;
      int order_vector, local_n;
      double scalar;

      double local_result = 0.0;
      double global_result = 0.0;

      double* vector_a, *vector_b;
      double* local_vector_a, *local_vector_b;

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

      if (my_rank == 0) {
            printf("Enter vector order:\n");
            scanf("%d", &order_vector);

            vector_a = (double *) malloc(order_vector *
      sizeof(double));
            vector_b = (double *) malloc(order_vector *
      sizeof(double));

            printf("\nInsert elements into vectors:\n");
            printf("Vector A:\n");

            for (int i = 0; i < order_vector; i++){
                  scanf("%lf", &vector_a[i]);
            }
```

```
            printf("Vector B:\n");
            for (int i = 0; i < order_vector; i++){
                  scanf("%lf", &vector_b[i]);
            }

            printf("\nEnter scalar:\n");
            scanf("%lf", &scalar);
      }

      MPI_Bcast(&order_vector, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Bcast(&scalar, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

      local_n = order_vector / comm_sz;

      local_vector_a = (double *) malloc(local_n *
sizeof(double));
      local_vector_b = (double *) malloc(local_n *
sizeof(double));

      MPI_Scatter(vector_a, local_n, MPI_DOUBLE, local_vector_a,
local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      MPI_Scatter(vector_b, local_n, MPI_DOUBLE, local_vector_b,
local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

      for (int i = 0; i < local_n; i++){
            local_result += local_vector_a[i] * local_vector_b[i]
      * scalar;
      }

      MPI_Reduce(&local_result, &global_result, 1, MPI_DOUBLE,
MPI_SUM, 0, MPI_COMM_WORLD);

      if(my_rank == 0){
            printf("\nThe result: %.2lf\n", global_result);

            free(vector_a);
            free(vector_b);
      }

      free(local_vector_a);
```

```
      free(local_vector_b);

      MPI_Finalize();
      return 0;
}
```

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

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

as somas dos prefixos são as "n" somas parciais

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

a. Elabore um algoritmo serial para calcular as "n" somas de prefixos de uma matriz com "n" elementos.

```
C/C++
int vector[8] = {5, 1, 8, 4, 3, 6, 4, 3};
int sum_prefixs[8];

for (int i = 0; i < 8; i++){
      sum_prefixs[i] = vector[i];

      if (i > 0)
            sum_prefixs[i] += sum_prefixs[i-1];
}
```

b. Paralelize seu algoritmo serial para um sistema com "n" processos, cada um armazenando um dos x.

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

int main(){
```

```
      int comm_sz, my_rank;
      int order,local_order;

      int *vector;
      int *local_vector;
      int last_sum = 0;

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

      if(my_rank == 0){
            printf("Enter the order of the vector: ");
            scanf("%d", &order);

            if (comm_sz != order) {
                  fprintf(stderr, "This program requires number
of processes = vector order.\n");
                  MPI_Finalize();
                  return 1;
            }
      }

      MPI_Bcast(&order, 1, MPI_INT, 0, MPI_COMM_WORLD);

      local_order = order/comm_sz;
      vector = (int *) malloc(order * sizeof(int));
      local_vector = (int *) malloc(local_order * sizeof(int));

      if(my_rank == 0){
            printf("Enter values in vector: \n");

            for (int i = 0; i < order; i++){
                  scanf("%d", &vector[i]);
            }

            printf("\n");
      }
```

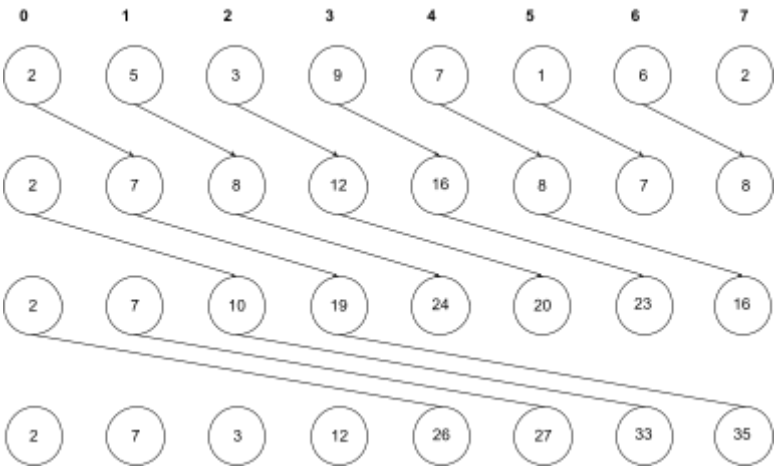
```
      MPI_Scatter(vector, local_order, MPI_INT, local_vector,
local_order, MPI_INT, 0, MPI_COMM_WORLD);

      for (int i = 0; i < local_order; i++){
            printf("Process %d[%d] = %d", my_rank, i,
local_vector[i]);
      }

      printf("\n");
      MPI_Barrier(MPI_COMM_WORLD);
      if(my_rank == 0){
            local_vector[0] += last_sum;
            for (int i = 1; i < local_order; i++){
                  local_vector[i] += local_vector[i - 1];
            }

            printf("\nProcess %d sum_prefix = %d\n", my_rank,
local_vector[local_order - 1]);

            MPI_Send(&local_vector[local_order - 1], order,
MPI_INT, my_rank + 1, 0 , MPI_COMM_WORLD);
      } else {
         MPI_Recv(&last_sum, order, MPI_INT, my_rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUSES_IGNORE);

            local_vector[0] += last_sum;
            for (int i = 1; i < local_order; i++){
                  local_vector[i] += local_vector[i - 1];
            }

            printf("Process %d sum_prefix = %d\n", my_rank,
local_vector[local_order - 1]);

            if(my_rank < comm_sz -1){
                  MPI_Send(&local_vector[local_order - 1] , 1 ,
MPI_INT , my_rank + 1 , 0 , MPI_COMM_WORLD);
            }
      }
```

```
    return 0;
}
```

c. Suponha n = 2k para algum inteiro positivo k. Você pode criar um algoritmo serial e uma paralelização do algoritmo serial de modo que o algoritmo paralelo exija apenas k fases de comunicação?

O código serial do item A que resolve o problema da soma dos prefixos requer um total de iterações igual ao tamanho do problema, que no caso seria o tamanho do vetor.

Para fazer com necessite de apenas k fazes de comunicação desenvolvi esse esquema ao lado para chegar no resultado final.



C/C++

```
/* File:  Q11_mpi_sum_prefix_CP.c
 * Compile: mpicc -g -Wall -o Q11_mpi_sum_prefix_CP
Q11_mpi_sum_prefix_CP.c -lm
 * Run:    mpiexec -n <number of processes> ./Q11_mpi_sum_prefix_CP
 */

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

int main(int argc, char** argv) {
      int my_rank, comm_sz;

      int vector[4] = { 2, 5, 3, 9 };
      int sum_prefix[4];

      int size = sizeof(vector) / sizeof(int);
      int iterator = log2(size);

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

```
	for (int i = 0; i < size; i++) {
		sum_prefix[i] = vector[i];
	}

	if (my_rank < (size - 1)){
		MPI_Send(&sum_prefix[my_rank], 1, MPI_INT, my_rank +
1, 0, MPI_COMM_WORLD);
	}

	for (int i = 1; i <= iterator; i++){
		MPI_Barrier(MPI_COMM_WORLD);
		if (my_rank > pow(2, i - 1) - 1){
			int acres;
			MPI_Recv(&acres, 1, MPI_INT, my_rank - pow(2, i -
1), 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

			sum_prefix[my_rank] += acres;
		}

		if (my_rank < (size - pow(2, i))){
			MPI_Send(&sum_prefix[my_rank], 1, MPI_INT,
my_rank + pow(2, i), 0, MPI_COMM_WORLD);
		}
	}

	if (my_rank == 0){
		printf("Vector: { ");
		for (int i = 0; i < size; i++){
			if (i != size - 1){
				printf("%d, ", vector[i]);
			} else {
				printf("%d }\n", vector[i]);
			}
		}
	}

	for (int i = 0; i < size; i++){
		if (my_rank == i && i == 0){
			printf("Sum_prefix: { %d, ", sum_prefix[i]);
		} else if(my_rank == i && i == size - 1){
```

```
            printf("%d }\n", sum_prefix[i]);
        } else if(my_rank == i){
            printf("%d, ", sum_prefix[i]);
        }
    }

    MPI_Finalize();

    return 0;
}
```

d. 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  */);
```

Opera em arrays com elementos de contagem; tanto sendbuf_p quanto recvbuf_p devem se referir a blocos de elementos de contagem do tipo de dados. O argumento op é o mesmo que op para MPI_Reduce. Escreva um programa MPI que gere uma matriz aleatória de elementos de contagem em cada processo MPI, encontre as somas dos prefixos e imprima os resultados.

C/C++

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

int main(){
    int comm_sz, my_rank;
    int order;

    int *vector;
```

```
      int *sum_prefixs;



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

      if(my_rank == 0){
            printf("Enter the order of the vector: ");
            scanf("%d", &order);
      }

      MPI_Bcast(&order, 1, MPI_INT, 0, MPI_COMM_WORLD);

      vector = (int *) malloc(order * sizeof(int));
      sum_prefixs = (int *) malloc(order * sizeof(int));

      srand(time(NULL) + my_rank);
      for (int i = 0; i < order; i++){
            vector[i] = rand() % 100;
      }

      printf("\n");
      for (int i = 0; i < order; i++){
            printf("\nProcess %d[%d] = %d", my_rank, i,
vector[i]);
      }

      printf("\n");
      MPI_Barrier(MPI_COMM_WORLD);
      MPI_Scan(vector, sum_prefixs, order, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);

      printf("\n");
      for (int i = 0; i < order; i++){
            printf("Process %d: sum[%d] = %d\n", my_rank, i,
sum_prefixs[i]);
      }
```

```
    free(vector);
    free(sum_prefixs);

    return 0;
}
```

3.12. Uma alternativa para uma estrutura de redução coletiva (allreduce) em formato de borboleta é uma estrutura de passagem de anel (ring-pass). Em uma passagem de anel, se houver p processos, cada processo q envia dados para o processo q + 1, com exceção do processo p - 1, que envia dados para o processo 0. Isso é repetido até que cada processo tenha o resultado desejado. Portanto, podemos implementar o 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 allreduce. Como seu desempenho se compara ao allreduce estruturado em borboleta?

C/C++

```
/* File:  Q12_mpi_ring_pass.c
 * Compile: mpicc -g -Wall -o Q12_mpi_ring_pass
Q12_mpi_ring_pass.c
 * Run:   mpiexec -n <number of processes> ./Q12_mpi_ring_pass
 */

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

int main(int argc, char const *argv[]){
    int my_rank, comm_sz;
    int sum, temp_val, my_val;
    int source, dest;
```

```
	int* local_array = NULL;

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

	if (my_rank == 0){
		local_array = (int*)malloc(comm_sz * sizeof(int));

		srand(time(NULL) + my_rank);
		for (int i = 0; i < comm_sz; i++){
			local_array[i] = rand() % 10;
			printf("D[%d] = %d\n", i, local_array[i]);
		}

		MPI_Scatter( local_array , 1 , MPI_INT , &my_val , 1 ,
MPI_INT , 0 , MPI_COMM_WORLD);

	} else {
		MPI_Scatter( local_array , 1 , MPI_INT , &my_val , 1 ,
MPI_INT , 0 , MPI_COMM_WORLD);
	}

	if (my_rank == 0){
		source = comm_sz - 1;
		dest = 1;
	} else if (my_rank == (comm_sz - 1)){
		source = comm_sz - 2;
		dest = 0;
	} else{
		source = my_rank - 1;
		dest = my_rank + 1;
	}

	sum = temp_val = my_val;
	for (int i = 1; i < comm_sz; i++){
		MPI_Sendrecv_replace( &temp_val , 1 , MPI_INT , dest , 0
, source , 0 , MPI_COMM_WORLD , MPI_STATUS_IGNORE);
		sum += temp_val;
```

```
      }

      MPI_Gather( &sum , 1 , MPI_INT , local_array , 1 , MPI_INT , 0 ,
MPI_COMM_WORLD);

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

  }

      free(local_array);

      MPI_Finalize();

      return 0;
}
```

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

```C/C++
/* File:  Q12_mpi_sum_prefix_ring_pass.c
 * Compile: mpicc -g -Wall -o Q12_mpi_sum_prefix_ring_pass
Q12_mpi_sum_prefix_ring_pass.c
 * Run:    mpiexec -n <number of processes>
./Q12_mpi_sum_prefix_ring_pass
 */

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

int main(int argc, char const *argv[]){
      int my_rank, comm_sz;
      int sum, temp_val, my_val;
```

```
	int source, dest;
	int* local_array = NULL;

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

	if (my_rank == 0){
		local_array = (int*)malloc(comm_sz * sizeof(int));

		srand(time(NULL) + my_rank);
		for (int i = 0; i < comm_sz; i++){
			local_array[i] = rand() % 10;
			printf("D[%d] = %d\n", i, local_array[i]);
		}

		MPI_Scatter( local_array , 1 , MPI_INT , &my_val , 1 ,
MPI_INT , 0 , MPI_COMM_WORLD);

	} else{
		MPI_Scatter( local_array , 1 , MPI_INT , &my_val , 1 ,
MPI_INT , 0 , MPI_COMM_WORLD);
	}

	if (my_rank == 0) {
		source = comm_sz - 1;
		dest = 1;
	} else if (my_rank == (comm_sz - 1)){
		source = comm_sz - 2;
		dest = 0;
	} else{
		source = my_rank - 1;
		dest = my_rank + 1;
	}

	temp_val = my_val;
	sum = 0;

	if (my_rank == 0){
```

```
            MPI_Send( &temp_val , 1 , MPI_INT , dest , 0 ,
MPI_COMM_WORLD);
            sum += temp_val;
      } else if (my_rank < (comm_sz - 1)){
            MPI_Recv( &sum , 1 , MPI_INT , source , 0 ,
MPI_COMM_WORLD , MPI_STATUS_IGNORE);
    sum += temp_val;
            MPI_Send( &sum , 1 , MPI_INT , dest , 0 ,
MPI_COMM_WORLD);
      } else {
            MPI_Recv( &sum , 1 , MPI_INT , (comm_sz - 1) - 1 , 0 ,
MPI_COMM_WORLD , MPI_STATUS_IGNORE);
            sum += temp_val;
      }

      MPI_Gather( &sum , 1 , MPI_INT , local_array , 1 , MPI_INT , 0 ,
MPI_COMM_WORLD);

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

      free(local_array);

      MPI_Finalize();

      return 0;
}
```

3.13. 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 MPI_Gatherv e MPI_Scatterv. Consulte as páginas de manual dessas funções e modifique seu programa de soma vetorial, produto escalar, para que ele possa lidar corretamente com o caso quando "n" não é divisível igualmente por comm_sz.

Como há o problema de quando “n” não divisível por comm_sz, temos que garantir que os processos possam receber quantidades diferentes de dados. Para isso temos que utilizar

MPI_Scatterv para poder distribuir os dados (de tamanhos diferentes e iguais) para os processos.
A alteração no código vai se dá basicamente na declaração e atribuição das variáveis (arrays) que devem ser passadas como parâmetro no MPI_Scatterv e alguns detalhes decorrente dessas variáveis.

C/C++

```
int main(void) {
      int my_rank, comm_sz;
      int order_vector, local_n;
      double scalar;

      double *vector_a, *vector_b;
      double *local_vector_a, *local_vector_b;

      double local_result = 0.0;
      double global_result = 0.0;

      int rest, local_add, current_displ = 0;
      int *send_counts, *displs;

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

      if (my_rank == 0) {
            printf("Enter vector order:\n");
            scanf("%d", &order_vector);

            vector_a = (double *) malloc(order_vector *
sizeof(double));
            vector_b = (double *) malloc(order_vector *
sizeof(double));

            printf("\nInsert elements into vectors:\n");
            printf("Vector A:\n");

            for (int i = 0; i < order_vector; i++){
                  scanf("%lf", &vector_a[i]);
            }

            printf("Vector B:\n");
```

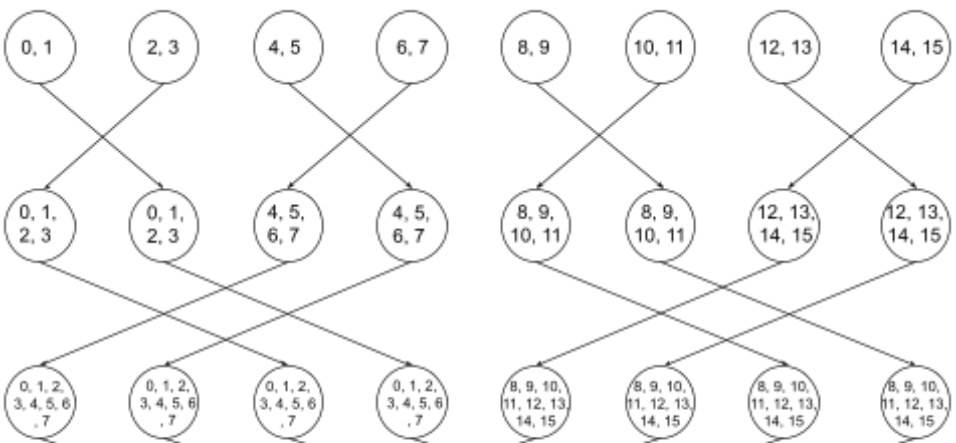
```
            for (int i = 0; i < order_vector; i++){
                  scanf("%lf", &vector_b[i]);
            }

            printf("\nEnter scalar:\n");
            scanf("%lf", &scalar);
      }

      MPI_Bcast(&order_vector, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Bcast(&scalar, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

      local_n = order_vector / comm_sz;
      rest = order_vector % comm_sz;

      if (my_rank < rest){
            local_add = 1;
      } else {
            local_add = 0;
      }

      local_vector_a = (double *) malloc((local_n + local_add) *
sizeof(double));
      local_vector_b = (double *) malloc((local_n + local_add) *
sizeof(double));

      send_counts = (int *) malloc(comm_sz * sizeof(int));
      displs = (int *) malloc(comm_sz * sizeof(int));

      for (int i = 0; i < comm_sz; i++) {
            send_counts[i] = local_n + (i < rest ? 1 : 0);
            displs[i] = current_displ;
            current_displ += send_counts[i];
      }

      MPI_Scatterv(vector_a, send_counts, displs, MPI_DOUBLE,
local_vector_a, local_n + local_add, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
      MPI_Scatterv(vector_b, send_counts, displs, MPI_DOUBLE,
local_vector_b, local_n + local_add, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
```

```
      for (int i = 0; i < send_counts[my_rank]; i++){
            local_result += local_vector_a[i] * local_vector_b[i]
* scalar;
      }

      MPI_Reduce(&local_result, &global_result, 1, MPI_DOUBLE,
MPI_SUM, 0, MPI_COMM_WORLD);

      if(my_rank == 0){
            printf("\nThe result: %.2lf\n", global_result);

            free(vector_a);
            free(vector_b);
      }

      free(local_vector_a);
      free(local_vector_b);
      free(send_counts);
      free(displs);

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

3.16. Suponha que comm_sz = 8 e o vetor x = (0, 1, 2, . . . , 15) tenha sido distribuído entre os processos usando uma distribuição em blocos. Desenhe um diagrama ilustrando as etapas de uma implementação borboleta de allgather de x.

Processo	Componentes
0	0 e 1
1	2 e 3
2	4 e 5
3	6 e 7

Processo	Componentes
4	8 e 9
5	10 e 11
6	12 e 13
7	14 e 15



3.17. MPI_Type_contiguous pode ser usado 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 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.

Read_vector:

C/C++

```
void Read_vector(char prompt[], double local_vec[], int n, int
local_n, int my_rank, MPI_Comm comm) {
      double* vec = NULL;
      int i, local_ok = 1;

      MPI_Datatype new_type_double;
      MPI_Type_contiguous(1, MPI_DOUBLE, &new_type_double);
      MPI_Type_commit(&new_type_double);

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

            if (vec == NULL)
                  local_ok = 0;

            Check_for_error(local_ok, "Read_vector", "Can't
      allocate temporary vector", comm);
            printf("Enter the vector %s\n", prompt);

            for (i = 0; i < n; i++)
                  scanf("%lf", &vec[i]);

            MPI_Scatter(vec, 1, new_type_double, local_vec, 1,
new_type_double, 0, comm);
            free(vec);
      } else {
```

```
            Check_for_error(local_ok, "Read_vector", "Can't
allocate temporary vector", comm);
            MPI_Scatter(vec, 1, new_type_double, local_vec, 1,
new_type_double, 0, comm);
      }

      MPI_Type_free(&new_type_double);
} /* Read_vector */
```

Print_vector:

C/C++

```
void Print_vector(char title[], double local_vec[], int n, int
local_n, int my_rank, MPI_Comm comm) {
      double* vec = NULL;
      int i, local_ok = 1;

      MPI_Datatype new_type_double;
      MPI_Type_contiguous(1, MPI_DOUBLE, &new_type_double);
      MPI_Type_commit(&new_type_double);

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

            if (vec == NULL)
                  local_ok = 0;

            Check_for_error(local_ok, "Print_vector", "Can't
allocate temporary vector", comm);

            MPI_Gather(local_vec, 1, new_type_double, vec, 1,
new_type_double, 0, comm);

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

            printf("\n");
```

```
            free(vec);
      } else {
            Check_for_error(local_ok, "Print_vector", "Can't
allocate temporary vector", comm);
            MPI_Gather(local_vec, 1, new_type_double, vec, 1,
new_type_double, 0, comm);
      }

      MPI_Type_free(&new_type_double);
} /* Print_vector */
```

3.19. MPI_Type_indexed pode ser usado para construir um tipo de dados derivado de elementos arbitrários de array. 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 de MPI_Type_create_struct, os deslocamentos são medidos em unidades de old_mpi_t – não em bytes. Use 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 × 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 × n como uma matriz unidimensional, criar o tipo de dados derivado e envie a parte triangular superior com uma única chamada para MPI_Send. O processo 1 deve receber a parte triangular superior com uma única chamada a MPI_Recv e depois imprimir os dados recebidos.

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

```c
int main(void) {
	int my_rank, comm_sz;
	int order_matrix, quant;

	int *matrix, *matrix_upper;

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

	if (comm_sz != 2) {
		fprintf(stderr, "This program requires exactly 2 processes.\n");
		MPI_Finalize();
		return 1;
	}

	if (my_rank == 0){
		printf("Enter matrix order:\n");
		scanf("%d", &order_matrix);
	}

	MPI_Bcast(&order_matrix, 1, MPI_INT, 0, MPI_COMM_WORLD);

	if (my_rank == 0) {
		MPI_Datatype upper_triangle_type;

		int blocklengths[order_matrix];
		int displacements[order_matrix];

		for (int i = 0; i < order_matrix; i++){
			blocklengths[i] = order_matrix - i;
			displacements[i] = (i * order_matrix) + (i);
		}

		MPI_Type_indexed(order_matrix, blocklengths, displacements, MPI_INT, &upper_triangle_type);
		MPI_Type_commit(&upper_triangle_type);
```

```
            matrix = (int *) malloc(order_matrix * order_matrix *
sizeof(int));

            printf("\nInsert elements into matrix:\n");
            for (int i = 0; i < order_matrix; i++){
                  for (int j = 0; j < order_matrix; j++){
                        printf("Element [%d][%d]: ", i, j);
                        scanf("%d", &matrix[i * order_matrix + j]);
                  }
            }

            MPI_Send(matrix, 1, upper_triangle_type, 1, 0,
MPI_COMM_WORLD);
      } else {
            quant = order_matrix * (order_matrix + 1) / 2;

            matrix_upper = (int *) malloc((quant) * sizeof(int));

            MPI_Recv(matrix_upper, quant, MPI_INT, 0, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            printf("\n");
            for (int i = 0; i < quant; i++){
                  if(i == quant - 1){
                        printf("%d\n", matrix_upper[i]);
                  } else {
                        printf("%d, ", matrix_upper[i]);
                  }
            }
      }

      if (my_rank == 0){
            free(matrix);
      }

      free(matrix_upper);

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

3.20. As funções MPI_Pack e MPI_Unpack fornecem uma alternativa aos tipos de dados derivados para agrupar dados. 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 de regras trapezoidais com o seguinte código:

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

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

A chave é o argumento da position. Quando MPI_Pack é chamado, a posição deve referir-se ao primeiro slot disponível em 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:

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     */);
```

Isso pode ser usado "invertendo" as etapas em MPI_Pack, ou seja, os dados são descompactados um bloco por vez, começando com position = 0.
Escreva outra função Get_input para o programa de regras trapezoidais. Este deve usar MPI_Pack no processo 0 e MPI_Unpack nos demais processos.

C/C++

```
void Get_input(int my_rank, int comm_sz, double* a, double* b, int*
n) {
      char pack_buf[100];
      int position;

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

            position = 0;

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

             MPI_Bcast(pack_buf , 100 , MPI_PACKED , 0 ,
      MPI_COMM_WORLD);
      } else {
            position = 0;

            MPI_Bcast( pack_buf , 100 , MPI_PACKED , 0 ,
MPI_COMM_WORLD);

            MPI_Unpack( pack_buf , 100 , &position , a , 1 ,
MPI_DOUBLE , MPI_COMM_WORLD);
            MPI_Unpack( pack_buf , 100 , &position , b , 1 ,
MPI_DOUBLE , MPI_COMM_WORLD);
            MPI_Unpack( pack_buf , 100 , &position , n , 1 , MPI_INT ,
MPI_COMM_WORLD);
  }
} /* Get_input */
```

3.22. Cronometre nossa implementação da regra trapezoidal que usa MPI_Reduce. Como você escolherá "n", o número de trapézios? Como os tempos mínimos se comparam aos tempos médios e medianos? Quais seus speedups? Quais são as eficiências? Com base nos dados que você coletou, você diria que a regra trapezoidal é escalável?

A escolha de “n” terá como base o tempo de execução do problema, enquanto o tempo de execução não for relevante, o tamanho do problema deverá aumentar. Tendo isso em mente, a escolha do tamanho do problema foi $5 * 10^7$ **.**

Tempo de execução sequencial e paralelos (em segundos):

comm_sz	**Números de trapézios (n)**		
	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
1	1,448637	3,042828	6,162827
2	0,902760	1,478527	3,469713
4	0,536016	1,013781	2,281294

Tempos	**Números de trapézios (n)**		
	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
Medianos	0,902760	1,478527	3,469713
Médios	0,98157	1,886304	3,984210

A diferença do tempo mediano para o médio não é tão grande, mas levando em consideração a escala dos tempos na tabela, a diferença entre eles acaba sendo um pouco significativa.

Speedup’s:

comm_sz	**Números de trapézios (n)**		
	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
1	1,00	1,00	1,00
2	1,604676	2,058013	1,776178
4	2,702600	3,001465	2,701461

Efficiency’s:

comm_sz	Números de trapézios (n)		
	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
1	1,00	1,00	1,00
2	0,802338	1,029007	0,888089
4	0,675650	0,750366	0,675365

Não é escalável, pois apesar de sua eficiência ser ótima em uma situação, ela não consegue manter sua eficiência constante.
Ela nem mesmo pode ser fortemente escalável devido não manter a eficiente constante quando aumenta apenas o número de processos, assim como fracamente escalável, pois não consegue manter a eficiência constante quando aumenta o número de processos na mesma escala do tamanho.

3.27. Encontre os speedup's e eficiências da classificação paralela ímpar-par. O programa obtém acelerações lineares? É escalável? É fortemente escalável? É fracamente escalável?

Tempo de execução onde o serial é o código MPI com 1 processo:

comm_sz	Números de trapézios (n)		
	$7,5 * 10^7$	$15 * 10^7$	$30 * 10^7$
1	11,379302	23,436915	52,460848
2	6,729476	15,675234	43,736653
4	4,857437	10,006563	31,934490

Speedup's:

comm_sz	Números de trapézios (n)		
	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
1	1,00	1,00	1,00
2	1,690964	1,495156	1,199471
4	2,342656	2,342154	1,642765

Efficiency's:

comm_sz	Números de trapézios (n)

	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
1	1,00	1,00	1,00
2	0,845482	0,747578	0,599736
4	0,585664	0,585539	0,410691

Com base nessas eficiências, é possível o algoritmo não é escalável diante de que a sua eficiência não se mantém constante a medida em que há o aumento do número de trabalho e de processos na mesma proporção.

De outra maneira:

Tempo de execução com o algoritmo totalmente serial:

comm_sz	**Números de trapézios (n)**		
	$2,5 * 10^3$	$5 * 10^3$	$10 * 10^3$
1	1,030256	4,802957	20,453372
2	0,001664	0,005142	0,007226
4	0,001479	0,002746	0,006116

Speedup's:

comm_sz	**Números de trapézios (n)**		
	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
1	1,00	1,00	1,00
2	619,1442	934,06398	2830,5248
4	696,5896	1749,0739	3344,2400

Efficiency's:

comm_sz	**Números de trapézios (n)**		
	$5 * 10^7$	$10 * 10^7$	$20 * 10^7$
1	1,00	1,00	1,00
2	309,5721	467,03199	1415,2624
4	174,1474	437,26848	836,06001

O cálculo do tempo de execução tomando esse modelo, observar que o algoritmo é escalável sim.

3.28. Modifique a classificação de transposição paralela ímpar-par para que as funções Merge simplesmente troquem os ponteiros da matriz após encontrar os elementos menores ou maiores. Que efeito essa mudança tem no tempo de execução geral?

comm_sz	Números de trapézios (n)		
	$7,5 * 10^7$	$15 * 10^7$	$30 * 10^7$
1	11,379302	23,436915	52,460848
2	6,729476	15,675234	43,736653
4	4,857437	10,006563	31,934490

comm_sz	Números de trapézios (n)		
	$7,5 * 10^7$	$15 * 10^7$	$30 * 10^7$
1	11.569591	23.385201	51.085460
2	6.410680	16.252951	38.824618
4	4.477462	9.928167	35.083344

É possível observar que não grande diferença no que diz respeito ao tempo de execução do problema, pois os tempos são quase os mesmo. O motivo disso acontecer é porque o método de realizar a troca de ponteiros comparado a cópia dos valores no algoritmo não impacta tanto no tempo de execução.