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

Docente: Ítalo Assis

Discente: João Gustavo Souza Lima

Lista de Exercícios
Programação de memória distribuída com MPI

Questoes respondidas:

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

Pau dos Ferros/RN
Novembro/2025

1. Modifique a regra trapezoidal para que ela estime corretamente a integral mesmo que comm_sz não divida *n* uniformemente. Você ainda pode assumir que $n \geq comm\_sz$.

```
int my_rank, comm_sz;
double a = 0.0, b = 1.0;
int n = 1000;
double h, local_a, local_b;
int base_n, resto, local_n;

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

h = (b - a) / n;
base_n = n / comm_sz;
resto = n % comm_sz;

if (my_rank < resto) {
   local_n = base_n + 1;
   local_a = a + my_rank * local_n * h;
} else {
   local_n = base_n;
   local_a = a + (resto * (base_n + 1)
               + (my_rank - resto) * base_n) * h;
}
local_b = local_a + local_n * h;

double local_integral = Trap(local_a, local_b, local_n, h);
double total_integral;

MPI_Reduce(&local_integral, &total_integral, 1, MPI_DOUBLE,
         MPI_SUM, 0, MPI_COMM_WORLD);

if (my_rank == 0)
   printf("Integral = %.15f\n", total_integral);
MPI_Finalize();
```

2. 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.

```
int main(void) {
   int my_rank, comm_sz;

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

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

   MPI_Finalize();
   return 0;
}
```

3. 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:

   p = comm_sz -> numero de processos

   n = tamanho do vetor x

   q = my_rank

   a. em bloco?

      div = n/p

      resto = n%p

      Se q < r então o processo recebe um elemento extra.

         quantidade = div + 1

         deslocamento = quantidade*q

      Se q >= r então o processo recebe apenas div elementos.

         quantidade = div

         deslocamento = q * div + resto

b. cíclica?

Cada processo recebe elementos de forma espaçada

Processo q recebe todos os índices i onde:

i%p = q

sequencia de indices do processo q (rank):

q, q + P, q 2P, q + 3P…

c. bloco-cíclica com tamanho de bloco *b*?

Mistura da distribuição por blocos com a cíclica.

Em vez de enviar um elemento por processo, enviamos blocos de tamanho b.

Os blocos são enviados ciclicamente.

O processo q recebe os blocos:

k(blocos) = q + P * i

onde i representa a rodada (i=0,1,2)

então os índices desse bloco será

[(k)*b], [(k) * b] + 1, … , [(k)*b] + (b-1)

4. 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.

```
int main() {
int my_rank, comm_sz;
int n, local_n;
double escalar;

double *v1 = NULL, *v2 = NULL;
double *local_v1, *local_v2;
double local_sum = 0.0, global_sum = 0.0;

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("Digite o tamanho dos vetores (n): ");
   fflush(stdout);
   scanf("%d", &n);

   printf("Digite o escalar: ");
   fflush(stdout);
   scanf("%lf", &escalar);

   v1 = malloc(n * sizeof(double));
   v2 = malloc(n * sizeof(double));
```

```
    printf("Digite os elementos do vetor 1:\n");
    for (int i = 0; i < n; i++){
        fflush(stdout);
        scanf("%lf", &v1[i]);
        }

    printf("Digite os elementos do vetor 2:\n");
    for (int i = 0; i < n; i++){
        fflush(stdout);
        scanf("%lf", &v2[i]);
    }
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&escalar, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

local_n = n / comm_sz;

local_v1 = malloc(local_n * sizeof(double));
local_v2 = malloc(local_n * sizeof(double));

MPI_Scatter(v1, local_n, MPI_DOUBLE,
          local_v1, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

MPI_Scatter(v2, local_n, MPI_DOUBLE,
          local_v2, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

for (int i = 0; i < local_n; i++)
    local_v1[i] *= escalar;

for (int i = 0; i < local_n; i++)
    local_sum += local_v2[i] * local_v2[i];

MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE,
          MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Gather(local_v1, local_n, MPI_DOUBLE,
          v1, local_n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

if (my_rank == 0) {
    printf("\nVetor 1 após multiplicação pelo escalar:\n");
    for (int i = 0; i < n; i++)
        printf("%.2f ", v1[i]);
    printf("\n");

    printf("\nNorma do vetor 2: %.6f\n", sqrt(global_sum));
}

free(local_v1);
free(local_v2);
if (my_rank == 0) {
    free(v1);
    free(v2);
```

```
    }

    MPI_Finalize();
    return 0;
}
```

5. 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

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

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

```
void prefix_sum(int x[], int y[], int n) {
    int soma = 0;

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

b) Suponha $n = 2^k$ para algum inteiro positivo k. Crie um algoritmo paralelo para um sistema com n processos, cada um armazenando um dos elementos de x, que exija apenas k fases de comunicação.

i) Sem utilizar MPI_Scan

```
for (d = 0; d < k; d++) {
      dist = 1 << d;   // 2^d
      int received_val;
      if (my_rank >= dist) {
         MPI_Recv(&received_val, 1, MPI_INT,
               my_rank - dist, 0,
               MPI_COMM_WORLD, &status);
         prefix_value += received_val;
      }
      if (my_rank + dist < comm_sz) {
         MPI_Send(&prefix_value, 1, MPI_INT,
               my_rank + dist, 0,
               MPI_COMM_WORLD);
      }
   }
```

c) 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.

```
int my_rank, comm_sz;
int count = 5;
int i;

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

int* local_vec = malloc(count * sizeof(int));
int* prefix_vec = malloc(count * sizeof(int));

srand(time(NULL) + my_rank);

for (i = 0; i < count; i++)
   local_vec[i] = (rand() % 9) + 1;

MPI_Scan(local_vec, prefix_vec, count, MPI_INT, MPI_SUM,
MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);

printf("Proc %d vetor prefixo: ", my_rank);
for (i = 0; i < count; i++)
   printf("%d ", prefix_vec[i]);
printf("\n");

free(local_vec);
free(prefix_vec);

MPI_Finalize();
```

6. 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?

```
#define MSG_SIZE 100000000
int main() {
   int my_rank, comm_sz;
   MPI_Status status;

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

   int *my_array = malloc(MSG_SIZE * sizeof(int));
   int *temp_array = malloc(MSG_SIZE * sizeof(int));
   int *recv_array = malloc(MSG_SIZE * sizeof(int));

   for (int i = 0; i < MSG_SIZE; i++)
      my_array[i] = 1;

   /*              RING ALLREDUCE                    */
   for (int i = 0; i < MSG_SIZE; i++)
      temp_array[i] = my_array[i];

   int dest = (my_rank + 1) % comm_sz;
   int source = (my_rank - 1 + comm_sz) % comm_sz;

   MPI_Barrier(MPI_COMM_WORLD);
   double start = MPI_Wtime();

   for (int step = 1; step < comm_sz; step++) {

      MPI_Sendrecv_replace(temp_array, MSG_SIZE, MPI_INT, dest, 0, source,
0,MPI_COMM_WORLD, &status);

      for (int i = 0; i < MSG_SIZE; i++)
         my_array[i] += temp_array[i];
   }

   double end = MPI_Wtime();
   double local_time = end - start;

   double ring_time;
   MPI_Reduce(&local_time, &ring_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

```
    /*              BUTTERFLY ALLREDUCE                  */
    for (int i = 0; i < MSG_SIZE; i++)
        my_array[i] = 1;

    MPI_Barrier(MPI_COMM_WORLD);
    start = MPI_Wtime();

    int steps = 0;
    while ((1 << steps) < comm_sz) steps++;

    for (int i = 0; i < steps; i++) {
        int partner = my_rank ^ (1 << i);

        MPI_Sendrecv(my_array, MSG_SIZE, MPI_INT, partner, 0, recv_array, MSG_SIZE, MPI_INT,
partner, 0, MPI_COMM_WORLD, &status);

        for (int j = 0; j < MSG_SIZE; j++)
            my_array[j] += recv_array[j];
    }

    end = MPI_Wtime();
    local_time = end - start;

    double butterfly_time;
    MPI_Reduce(&local_time, &butterfly_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    if (my_rank == 0) {
        printf("\nTempo total (Ring):      %.6f s\n", ring_time);
        printf("Tempo total (Butterfly): %.6f s\n", butterfly_time);
        printf("\n(msg = %d)\n", MSG_SIZE);
        printf("\n(comm_sz = %d processos)\n\n", comm_sz);
    }
    free(my_array);
    free(temp_array);
    free(recv_array);

    MPI_Finalize();
    return 0;
}
```

	40M	60M	80M	100M
Ring (s)	4.0996	7.0048	9.5600	20.0943
Butterfly (s)	1.3017	1.7822	2.1322	12.7051
Razão	3.15	3.93	4.48	1.58

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

```
int main(){
    int my_rank, comm_sz;
    MPI_Status status;

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

    int my_val = my_rank + 1;
    int prefix_sum = my_val;
    int temp = my_val;

    int dest   = (my_rank + 1) % comm_sz;
    int source = (my_rank - 1 + comm_sz) % comm_sz;

    for (int step = 1; step < comm_sz; step++) {

        MPI_Sendrecv_replace(&temp, 1, MPI_INT,
                             dest, 0,
                             source, 0,
                             MPI_COMM_WORLD, &status);

        int sender = (my_rank - step + comm_sz) % comm_sz;

        if (sender < my_rank)
            prefix_sum += temp;
    }

    printf("Processo %d: prefix_sum = %d\n", my_rank, prefix_sum);

    MPI_Finalize();
    return 0;
}
```

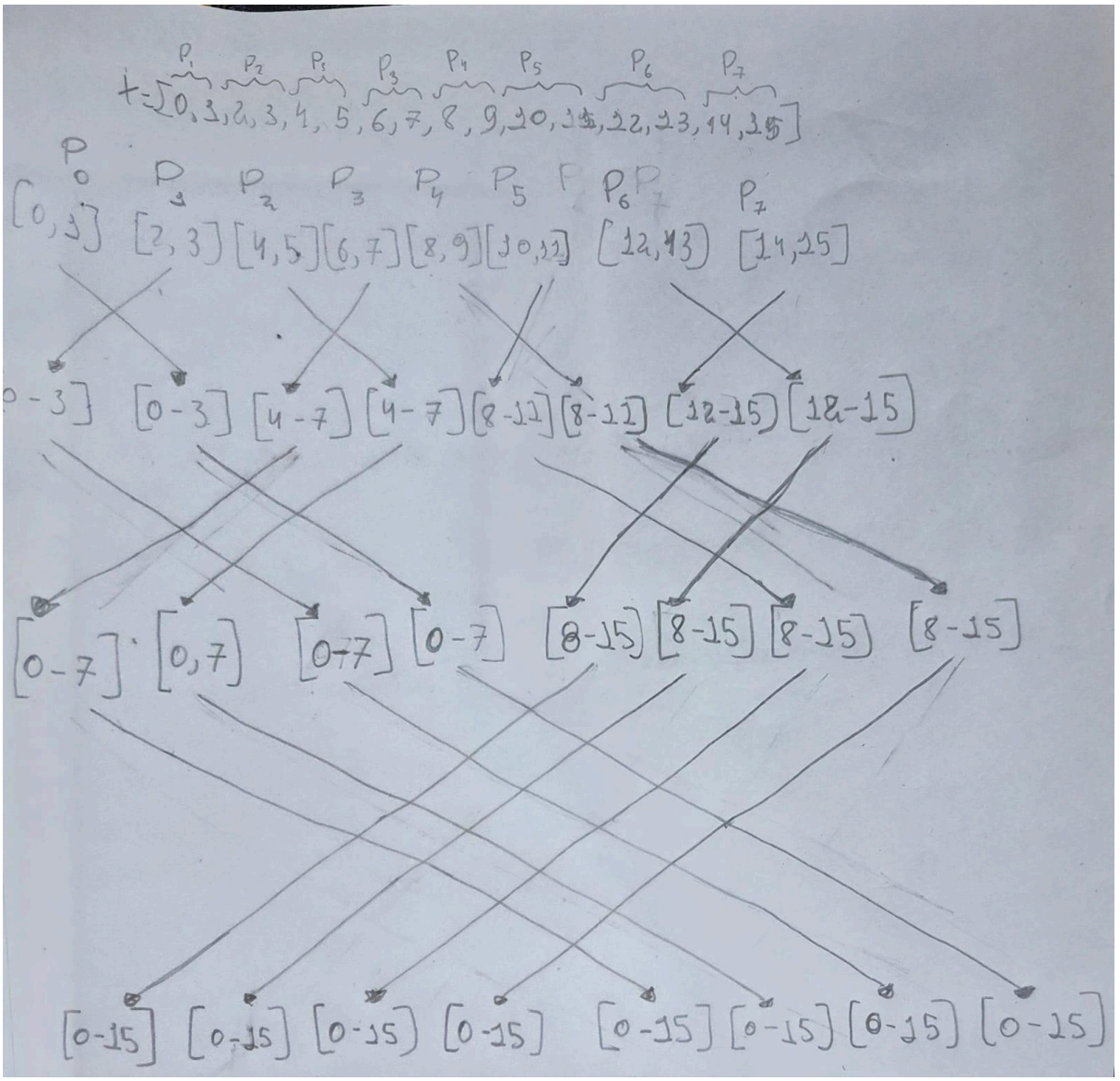
7. As funções MPI_Scatter e MPI_Gather têm a limitação de que cada processo deve enviar ou receber o mesmo número de itens de dados. Quando este não for o caso, devemos utilizar as funções MPI_Gatherv e MPI_Scatterv. Consulte a documentação dessas funções e modifique seu programa da questão 4 para que ele possa lidar corretamente com o caso quando *n* não é divisível por comm_sz.

```
int main(int argc, char* argv[]) {
   int my_rank, comm_sz;
   int n;
   double escalar;

   double *v1 = NULL, *v2 = NULL;
   double *local_v1, *local_v2;
   double local_sum = 0.0, global_sum = 0.0;

   int *sendcounts = NULL;
   int *displs = NULL;
   int local_n;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   if (my_rank == 0) {
      printf("Digite o tamanho dos vetores (n): ");
      fflush(stdout);
      scanf("%d", &n);

      printf("Digite o escalar: ");
      fflush(stdout);
      scanf("%lf", &escalar);

      v1 = malloc(n * sizeof(double));
      v2 = malloc(n * sizeof(double));

      printf("Digite os elementos do vetor 1:\n");
      for (int i = 0; i < n; i++){
         fflush(stdout);
         scanf("%lf", &v1[i]);
         }

      printf("Digite os elementos do vetor 2:\n");
      for (int i = 0; i < n; i++){
         fflush(stdout);
         scanf("%lf", &v2[i]);
      }

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

      int div = n / comm_sz;
      int resto = n % comm_sz;
```

```
    int offset = 0;
    for (int i = 0; i < comm_sz; i++) {
        sendcounts[i] = div + (i < resto ? 1 : 0);
        displs[i] = offset;
        offset += sendcounts[i];
    }
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&escalar, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

int div = n / comm_sz;
int resto = n % comm_sz;
local_n = div + (my_rank < resto ? 1 : 0);

local_v1 = malloc(local_n * sizeof(double));
local_v2 = malloc(local_n * sizeof(double));

MPI_Scatterv(v1, sendcounts, displs, MPI_DOUBLE,
            local_v1, local_n, MPI_DOUBLE,
            0, MPI_COMM_WORLD);

MPI_Scatterv(v2, sendcounts, displs, MPI_DOUBLE,
            local_v2, local_n, MPI_DOUBLE,
            0, MPI_COMM_WORLD);

for (int i = 0; i < local_n; i++)
    local_v1[i] *= escalar;

for (int i = 0; i < local_n; i++)
    local_sum += local_v2[i] * local_v2[i];

MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);

MPI_Gatherv(local_v1, local_n, MPI_DOUBLE,
            v1, sendcounts, displs, MPI_DOUBLE,
            0, MPI_COMM_WORLD);

if (my_rank == 0) {
    printf("\nVetor 1 após multiplicação pelo escalar:\n");
    for (int i = 0; i < n; i++)
        printf("%.2f ", v1[i]);
    printf("\n");

    printf("\nNorma do vetor 2: %.6f\n", sqrt(global_sum));

    free(v1);
    free(v2);
    free(sendcounts);
    free(displs);
}

free(local_v1);
```

```
    free(local_v2);

    MPI_Finalize();
    return 0;
}
```

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



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

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

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

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

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

   MPI_Type_contiguous(local_n, MPI_DOUBLE, &block_type);
   MPI_Type_commit(&block_type);

   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_type, local_a, 1, block_type, 0, comm);
      free(a);
   } else {
      Check_for_error(local_ok, fname, "Can't allocate temporary vector",
comm);
      MPI_Scatter(a, 1, block_type, local_a, 1, block_type, 0, comm);
   }

   MPI_Type_free(&block_type);
}
```

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

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

   MPI_Type_contiguous(local_n, MPI_DOUBLE, &block_type);
   MPI_Type_commit(&block_type);

   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_type,
                 b, 1, block_type,
                 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_type, b, 1, block_type, 0, comm);
   }

   MPI_Type_free(&block_type);
}
```

10. 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.

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

   if (comm_sz < 2) {
      if (my_rank == 0) printf("Execute com pelo menos 2 processos.\n");
      MPI_Finalize();
      return 0;
   }

   int n = 4;
   int N = n * n;

   MPI_Datatype upper_type;
   int *blocklengths = malloc(n * sizeof(int));
   int *displs = malloc(n * sizeof(int));

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

   MPI_Type_indexed(n, blocklengths, displs, MPI_INT, &upper_type);
   MPI_Type_commit(&upper_type);
```

```
    if (my_rank == 0) {
        int* matrix = malloc(N * sizeof(int));

        for (int i = 0; i < N; i++)
            matrix[i] = i;

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

        free(matrix);
    }
    else if (my_rank == 1) {
        int total = n*(n+1)/2;

        int *recvbuf = malloc(total * sizeof(int));

        MPI_Recv(recvbuf, total, upper_type, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        printf("Processo 1 recebeu os elementos da triangular superior:\n");
        for (int i = 0; i < total; i++)
            printf("%d ", recvbuf[i]);
        printf("\n");

        free(recvbuf);
    }

    MPI_Type_free(&upper_type);
    free(blocklengths);
    free(displs);

    MPI_Finalize();
    return 0;
}
```

11. 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.

```
void Get_input(double* a_p, double* b_p, int* n_p, int my_rank, MPI_Comm
comm) {
   char pack_buf[100];
   int position = 0;

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

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

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

   if (my_rank != 0) {
      position = 0;
      MPI_Unpack(pack_buf, 100, &position, a_p, 1, MPI_DOUBLE, comm);
      MPI_Unpack(pack_buf, 100, &position, b_p, 1, MPI_DOUBLE, comm);
      MPI_Unpack(pack_buf, 100, &position, n_p, 1, MPI_INT,    comm);
   }
}
```

12. Cronometre a implementação do livro da regra dos trapézios que usa `MPI_Reduce` para diferentes números de trapézios e processos, n e p, respectivamente.

Tempo médio				
Processos (p)	**Nós**	**10 000 000 000**	**20 000 000 000**	**40 000 000 000**
1	1	10.9201	21.8422	43.7903
2	1	5.5969	11.1846	22.4021
4	1	2.7863	5.5927	11.1664
8	1	1.4641	2.9173	5.8354
16	1	0.7393	1.4789	2.9564
24	1	0.5278	1.0189	2.0394
48	2	0.2570	0.5148	1.0241
96	4	0.1316	0.2633	0.5195

Tempo mínimo				
Processos (p)	**Nós**	**10 000 000 000**	**20 000 000 000**	**40 000 000 000**
1	1	10.9177	21.8327	43.7784
2	1	5.5806	11.1694	22.3583
4	1	2.7731	5.5806	11.1392
8	1	1.4553	2.9090	5.8272
16	1	0.7391	1.4785	2.9561
24	1	0.5097	1.0109	2.0279
48	2	0.2553	0.5115	1.0196
96	4	0.1293	0.2578	0.5119

Tempo mediano				
Processos (p)	**Nós**	**10 000 000 000**	**20 000 000 000**	**40 000 000 000**
1	1	10.9205	21.8426	43.7904
2	1	5.5981	11.1816	22.3978
4	1	2.7874	5.5954	11.1688
8	1	1.4591	2.9155	5.8340
16	1	0.7393	1.4785	2.9563
24	1	0.5114	1.0175	2.0384
48	2	0.2569	0.5139	1.0246
96	4	0.1300	0.2592	0.5192

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

A escolha do tamanho de n foi com base no resultado do programa com o tempo superior a 1s e o aumento foi com base no crescimento de n*k sendo k=2, ou seja, foi dobrado o valor para cada objetivo de medida. Os valores de n foram escolhidos suficientemente grande para garantir que o tempo de computação dominasse o overhead de comunicação.

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

O tempo mínimo é sempre ligeiramente menor que os tempos médio e mediano, mas todos são muito próximos. O tempo mínimo apresenta valores ligeiramente inferiores aos tempos médio e mediano, que permanecem muito próximos entre si. As pequenas diferenças observadas indicam baixa variabilidade entre as execuções, sendo mais perceptíveis apenas em configurações com maior número de processos, onde efeitos de comunicação podem introduzir atrasos ocasionais.

(c) Quais são os *speedups*?

Processos (p)	10 000 000 000	20 000 000 000	40 000 000 000
1	1.00	1.00	1.00
2	1.95	1.95	1.95
4	3.92	3.91	3.92
8	7.46	7.49	7.50
16	14.77	14.77	14.81
24	20.69	21.44	21.47
48	42.49	42.43	42.77
96	83.00	82.96	84.26

(d) Quais são as eficiências?

Processos (p)	10 000 000 000	20 000 000 000	40 000 000 000
1	1.00	1.00	1.00
2	0.98	0.98	0.98
4	0.98	0.98	0.98
8	0.93	0.94	0.94
16	0.92	0.92	0.93
24	0.86	0.89	0.89
48	0.88	0.88	0.89
96	0.86	0.86	0.88

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

Sim. Um programa é escalável se consegue usar eficientemente um número crescente de processos. Os resultados mostram que, ao aumentar o tamanho do problema juntamente com o número de processos, a eficiência paralela se mantém aproximadamente constante.

script:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=24
#SBATCH -p sequana_cpu_dev
#SBATCH -J mpi_trap_np_scaling
#SBATCH --exclusive
#SBATCH --time=00:15:00
#SBATCH --output=resultado_questao12_%j.log

echo "Job ID: $SLURM_JOB_ID"
echo "Nodes allocated: $SLURM_JOB_NODELIST"
echo "====================================="

cd /scratch/pex1272-ufersa/joao.lima2/questoes12E13/ || exit 1

module load gcc/14.2.0_sequana
module load openmpi/gnu/4.1.4_sequana

# Compila
mpicc -O2 -Wall -o mpi_trap_time mpi_trap_time.c

A=0.0
B=1.0

# Diferentes números de trapézios
N_LIST="10000000000 20000000000 40000000000"

########################################
# Loop em n
########################################
for N in $N_LIST
do
   echo ""
   echo "########################################"
   echo "### Experimentos com n = $N"
   echo "########################################"
   echo ""

   ####################################
   # 1 NÓ — dobrando processos
   ####################################
   echo "===== 1 NODE SCALING ====="
   for NP in 1 2 4 8 16 24
   do
      echo ""
      echo ">>> n=$N | p=$NP"
      echo "$A $B $N" | mpirun -np $NP ./mpi_trap_time
   done

   ####################################
   # 2 NÓS COMPLETOS
   echo ""
   echo "===== 2 NODES (48 processes) ====="
```

```
    echo ">>> n=$N | p=48"
    echo "$A $B $N" | mpirun -np 48 ./mpi_trap_time

    ######################################
    # 4 NÓS COMPLETOS
    ######################################
    echo ""
    echo "===== 4 NODES (96 processes) ====="
    echo ">>> n=$N | p=96"
    echo "$A $B $N" | mpirun -np 96 ./mpi_trap_time
done

echo ""
echo "FIM DO JOB"
```

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

Processos (p)	Tempo médio (s)	Speedup
1	8.8850	1.00
2	4.6488	1.91
4	2.4629	3.61
8	1.4018	6.34
16	0.8884	10.00
24	0.7927	11.21
48	0.6011	14.78
96	0.5184	17.14

Processos (p)	Eficiência
1	1.00
2	0.96
4	0.90
8	0.79
16	0.63
24	0.47
48	0.31
96	0.18

(a) O programa obtém *speedups* lineares?

Não, o programa de ordenação ímpar–par não obtém speedups lineares.

(b) É escalável?

Não, o programa de ordenação ímpar–par não é escalável. Os resultados mostram que a eficiência paralela decresce significativamente à medida que o número de processos aumenta, indicando que o crescimento do custo de comunicação e sincronização não é compensado pelo aumento do tamanho do problema.

(c) É fortemente escalável?

Não, o programa de ordenação ímpar–par não é fortemente escalável. Para um tamanho de problema fixo, o aumento do número de processos não resulta em redução proporcional do tempo de execução, devido ao elevado custo de comunicação e sincronização inerente ao algoritmo, o que leva a speedups sublineares e queda acentuada da eficiência.

(d) É fracamente escalável?

Não, o programa de ordenação ímpar–par não é fracamente escalável. Mesmo com o aumento proporcional do tamanho do problema, o custo de comunicação e sincronização cresce com o número de processos, impedindo que o tempo de execução permaneça aproximadamente constante.

script:

```
#!/bin/bash
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=24
#SBATCH -p sequana_cpu_dev
#SBATCH -J mpi_odd_even_scaling
#SBATCH --exclusive
#SBATCH --time=00:15:00
#SBATCH --output=resultado_questao13_%j.log

echo "Job ID: $SLURM_JOB_ID"
echo "Nodes allocated: $SLURM_JOB_NODELIST"
echo "====================================="

cd /scratch/pex1272-ufersa/joao.lima2/questoes12E13/ || exit 1

module load gcc/14.2.0_sequana
module load openmpi/gnu/4.1.4_sequana

# Compila
mpicc -O2 -Wall -o mpi_odd_even_time mpi_odd_even_time.c

GLOBAL_N=96000000   # divisível por 1,2,4,8,16,24,48,96
```

```
########################################
# 1 NÓ — dobrando processos
########################################
echo "===== 1 NODE SCALING ====="
for NP in 1 2 4 8 16 24
do
   echo ""
   echo ">>> Executando com $NP processo(s) em 1 nó"
   mpirun -np $NP ./mpi_odd_even_time g $GLOBAL_N
done

########################################
# 2 NÓS COMPLETOS
########################################
echo ""
echo "===== 2 NODES (48 processes) ====="
mpirun -np 48 ./mpi_odd_even_time g $GLOBAL_N

########################################
# 4 NÓS COMPLETOS
########################################
echo ""
echo "===== 4 NODES (96 processes) ====="
mpirun -np 96 ./mpi_odd_even_time g $GLOBAL_N

echo ""
echo "FIM DO JOB"
```

14. 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? Fixe uma quantidade de processos (ex.: 8 processos) e analise a influência do tamanho dos vetores no tempo de execução.

```
void Merge_low(int **my_keys, int recv_keys[], int **temp_keys, int local_n){
  int m_i, r_i, t_i;

  m_i = r_i = t_i = 0;
  while (t_i < local_n){
    if ((*my_keys)[m_i] <= recv_keys[r_i]){
      (*temp_keys)[t_i] = (*my_keys)[m_i];
      t_i++;
      m_i++;
    }
    else{
      (*temp_keys)[t_i] = recv_keys[r_i];
      t_i++;
      r_i++;
    }
  }
  int *aux = *my_keys;
  *my_keys = *temp_keys;
  *temp_keys = aux;
} /* Merge_low */
```

```
void Merge_high(int **local_A, int temp_B[], int **temp_C, int local_n){
  int ai, bi,ci;

  ai = local_n - 1;
  bi = local_n - 1;
  ci = local_n - 1;

  while (ci >= 0){
    if ((*local_A)[ai] >= temp_B[bi]){
      (*temp_C)[ci] = (*local_A)[ai];
      ci--;
      ai--;
    }
    else{
      (*temp_C)[ci] = temp_B[bi];
      ci--;
      bi--;
    }
  }
  int *aux = *local_A;
  *local_A = *temp_C;
  *temp_C = aux;
} /* Merge_high */
```

Tamanho	Tempo Original (s)	Tempo Modificado (s)
80 milhões	2.4197	2.2220
160 milhões	5.2791	4.5266
320 milhões	12.3739	11.3484

Referência

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