UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE – UFRN
CENTRO DE TECNOLOGIA – CT
DEPARTAMENTO DE COMPUTAÇÃO E AUTOMAÇÃO – DCA

PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA – 2023.2

LUIZ HENRIQUE ARAÚJO DANTAS – 20220044224

Questões realizadas:

1, 2, 3, 4, 7, 11, 12

NPAD: 14 (3), 15 (2)

12/19 questões

Natal/RN
Dezembro/2023

1. **O que acontece no programa greetings se, em vez de strlen(greeting) + 1, usarmos strlen(greeting) para o comprimento da mensagem a ser enviada pelos processos 1, 2, ... , comm_sz-1? O que acontece se usarmos MAX_STRING em vez de strlen(greeting) + 1? Você pode explicar estes resultados?**

Gera uma inconsistência, o último índice do vetor da string a ser recebido é o "\0" e se trata de um valor nulo, por esse motivo existe o strlen(greeting)+1 para evitar o aparecimento dessa inconsistência. Se utilizamos apenas o strlen(greeting) como o comprimento da mensagem a ser enviada, o valor nulo "\0" não existe, assim os dados do vetor enviado não pode ser representado como uma string, pois não está completa.

Utilizar MAX_STRING faz com que a string seja enviada normalmente até o valor especificado, incluindo o valor nulo "\0". Se a string a ser enviada for menor que o tamanho fixado por MAX_STRING caracteres extra estarão sendo passados juntos à string.

2. **Suponha comm_sz = 4 e suponha que x é um vetor com n=14 componentes.**

   a. Como os componentes de x seriam distribuídos entre os processos em um programa que utilizasse uma block distribution?

Para um programa que utilize uma block distribution, Pacheco afirma que os processos recebem uma quantidade uniforme de componentes. Para a situação que estamos nos referindo basta realizarmos a divisão entre a quantidade de componentes e a quantidade de processos, resultando assim em: 14/4 = 3.5. Com esse resultado, serão alocados 4 componentes para os dois primeiros processos e 3 componentes para os outros dois processos, distribuído uniformemente entre eles, totalizando assim as 14 componentes.

Processo 0: x_0, x_1, x_2, x_3

Processo 1: x_4, x_5, x_6, x_7

Processo 2: x_8, x_9, x_{10}

Processo 3: x_{11}, x_{12}, x_{13}

b. Como os componentes de x seriam distribuídos entre os processos em um programa que utilizasse uma cyclic distribution?

Para a cyclic distribution a distribuição de componentes para os processos é feita a partir do estilo round robin, onde o processo 0 recebe a componente 0, o processo 1 recebe a componente 1 e assim por diante,

Processo 0: x_0, x_4, x_8, x_{12}

Processo 1: x_1, x_5, x_9, x_{13}

Processo 2: x_2, x_6, x_{10}

Processo 3: x_3, x_7, x_{11}

c. Como os componentes de x seriam distribuídos entre os processos em um programa que utilizasse uma block-cyclic distribution com blocos de tamanho b = 2?

Para o block-cyclic distribution, a distribuição dos blocos de componentes é feita utilizando o round robin do cyclic, onde a partir do valor de *b* que os componentes serão distribuídos entre os processos.

Processo 0: x_0, x_1, x_8, x_9

Processo 1: x_2, x_3, x_{10}, x_{11}

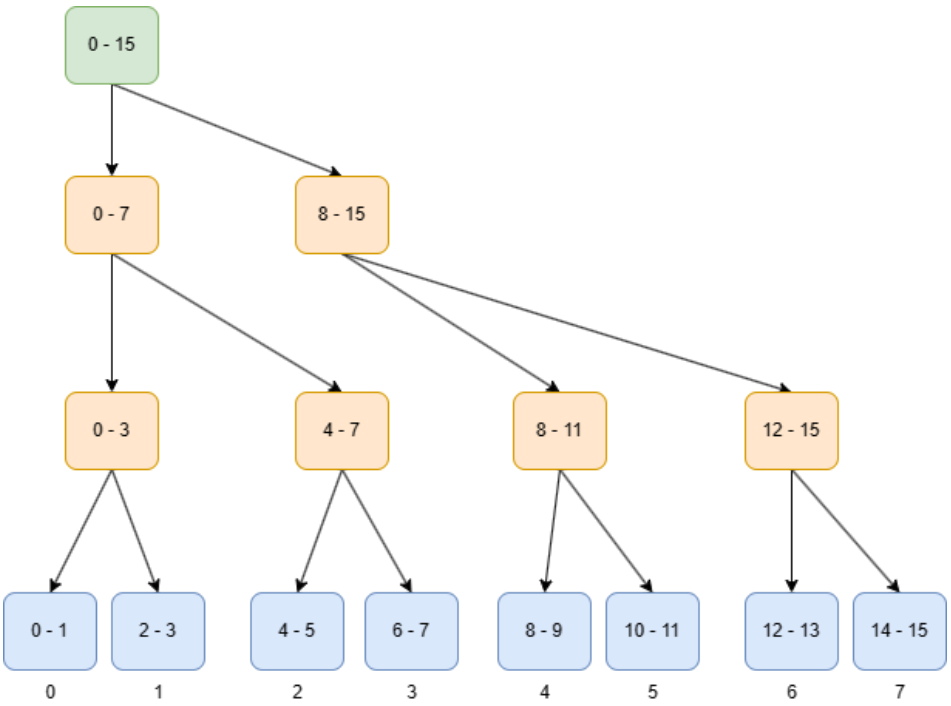
Processo 2: x_4, x_5, x_{12}, x_{13}

Processo 3: x_6, x_7

3. Suponha comm_sz = 8 e n = 16.

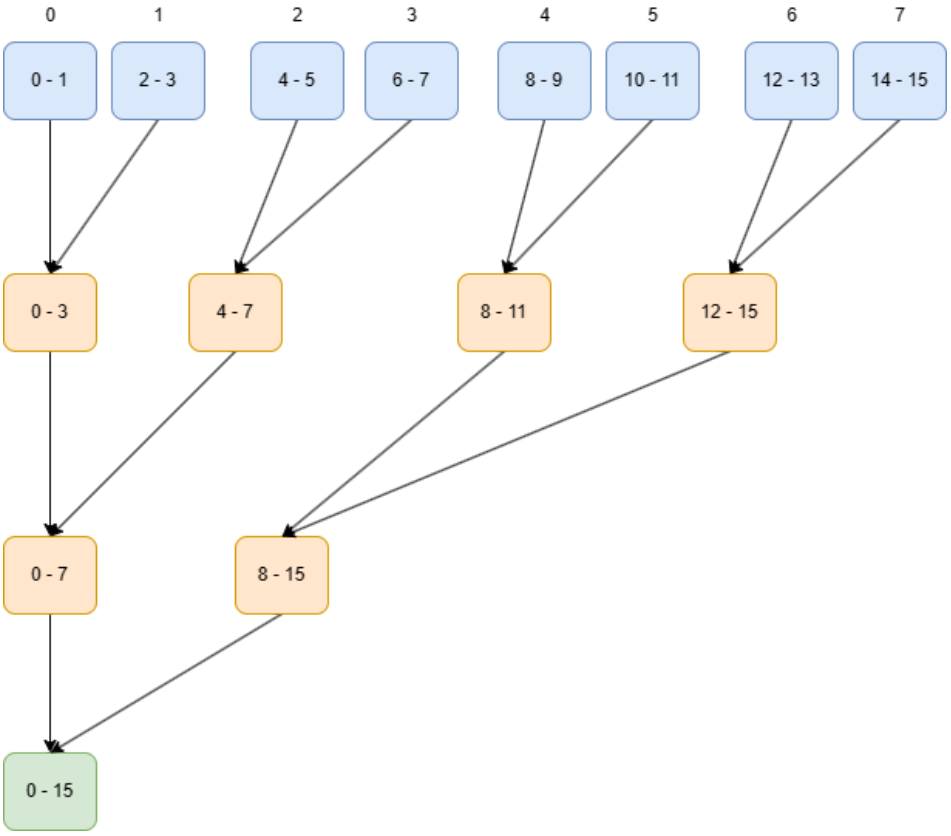
a. Desenhe um diagrama que mostre como o MPI_Scatter pode ser implementado usando a tree-structed communication com comm_sz processos quando o processo 0 precisa distribuir uma matriz contendo n elementos.

Podemos descrever o funcionamento da função MPI_Scatter, ela é utilizada para distribuir os dados de um processo para todos os processos em um comunicador, dessa forma cada processo recebe uma porção dos dados. A ideia é dividir um array de dados no processo mestre e distribuir partes iguais para cada processo no comunicador.



b. Desenhe um diagrama que mostre como o MPI_Gather pode ser implementado usando a tree-structed communication quando uma matriz de n-elementos que tenha sido distribuída entre processos comm_sz precisa ser reunida no processo 0.

Para a função MPI_Gather, lê em um dado parcial que está num processo e o envia para que seja juntado com os demais processos e assim formar o dado original no processo mestre.



4. **Escreva um programa MPI que implementa a multiplicação de vetor por escalar e o produto vetorial. O usuário deve entrar com 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. Pode-se supor que n, a ordem dos vetores, é igualmente divisível por comm_sz.**

Para essa questão, foi utilizado o programa fornecido pelo livro-texto chamado "mpi_vector_add.c" pois ele já implementa funções que lê, escreve e aloca vetores, e possui uma função para ler o tamanho n do vetor, para isso implementei três funções que são necessárias para a solução dessa questão, a Read_escalar(), produto_escalar() e mult_vetor_escalar(), nisso podemos ver que a saída do processo mestre terá um resultado baseado no conjunto de todas essas funções.

```c
int main(void)
{
    int n, local_n;
    int comm_sz, my_rank;
    double *local_x, *local_y, *local_z;
    double escalar, prod_escalar;
    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);

    Read_escalar(&escalar, my_rank, comm);

    Read_vector(local_x, local_n, n, "1", my_rank, comm);
    Print_vector(local_x, local_n, n, "Vetor 1 is", my_rank, comm);

    Read_vector(local_y, local_n, n, "2", my_rank, comm);
    Print_vector(local_y, local_n, n, "Vetor 2 is", my_rank, comm);

    produto_escalar(local_x, local_y, comm, local_n, &prod_escalar);
    if (my_rank == 0)
    {
        printf("Resultado do produto escalar: \n%lf\n", prod_escalar);
    }
```

```c
  if (my_rank == 0)
  {
      printf("Multiplicação do vetor 1 pelo escalar: \n%lf\n", escalar);
  }

  mult_vetor_escalar(local_x, escalar, local_z, local_n);
  Print_vector(local_z, local_n, n, "", my_rank, comm);

  if (my_rank == 0)
  {
      printf("Multiplicação do vetor 2 pelo escalar: \n%lf\n", escalar);
  }

  mult_vetor_escalar(local_x, escalar, local_z, local_n);
  Print_vector(local_z, local_n, n, "", my_rank, comm);

  free(local_x);
  free(local_y);
  free(local_z);

  MPI_Finalize();

  return 0;
```

```c
void Read_n(
    int *n_p,
    int *local_n_p,
    int my_rank,
    int comm_sz,
    MPI_Comm comm)
{

    if (my_rank == 0)
    {
        printf("ordem dos vetores 1 e 2: \n");
        scanf("%d", n_p);
    }
    MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

    *local_n_p = *n_p / comm_sz;
}
```

```c
void Allocate_vectors(
    double **local_x_pp,
    double **local_y_pp,
    double **local_z_pp,
    int local_n,
    MPI_Comm comm)
{
    *local_x_pp = malloc(local_n * sizeof(double));
    *local_y_pp = malloc(local_n * sizeof(double));
    *local_z_pp = malloc(local_n * sizeof(double));
}
```

```c
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;

    if (my_rank == 0)
    {
        a = malloc(n * sizeof(double));
        printf("valores do vetor %s\n", vec_name);
        for (i = 0; i < n; i++)
            scanf("%lf", &a[i]);
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
                    comm);
        free(a);
    }
    else
    {
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE, 0,
                    comm);
    }
}
```

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

    double *b = NULL;
    int i;

    if (my_rank == 0)
    {
        b = malloc(n * sizeof(double));
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
        printf("%s\n", title);
        for (i = 0; i < n; i++)
        {
            printf("%f ", b[i]);
        }
        printf("\n");
        free(b);
    }
    else
    {
        MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE, 0, comm);
    }
}
```

```c
void Read_escalar(
    double *escalar,
    int my_rank,
    MPI_Comm comm)
{
    if (my_rank == 0)
    {
        printf("defina o escalar: \n");
        scanf("%lf", escalar);
    }
    MPI_Bcast(escalar, 1, MPI_DOUBLE, 0, comm);
}
```

```c
double produto_escalar(
    double local_x[],
    double local_y[],
    MPI_Comm comm,
    int local_n,
    double *prod_scalar)
{
    double prod_escalar, local_prod_scalar = 0;
    int local_i;

    for (local_i = 0; local_i < local_n; local_i++)
    {
        local_prod_scalar += local_x[local_i] * local_y[local_i];
    }

    MPI_Reduce(&local_prod_scalar, prod_scalar, 1, MPI_DOUBLE, MPI_SUM, 0, comm);

    return prod_escalar;
}
```

```c
void mult_vetor_escalar(
    double local_x[],
    double escalar,
    double local_z[],
    int local_n)
{
    int local_i;

    for (local_i = 0; local_i < local_n; local_i++)
    {
        local_z[local_i] = local_x[local_i] * escalar;
    }
}
```

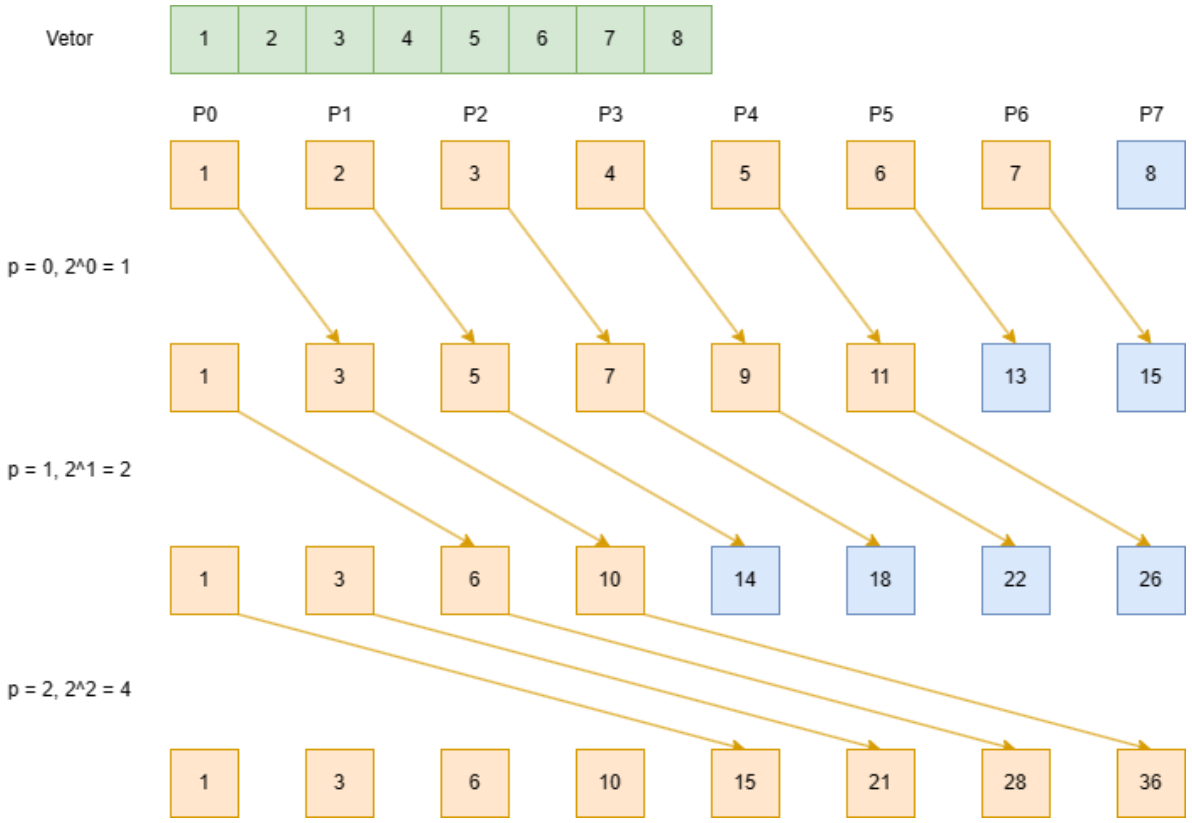
7. Apresente um diagrama mostrando que a função MPI_Scan() pode ser implementada em log2(p) passos sequenciais da mesma forma que a função MPI_Allreduce() pode.

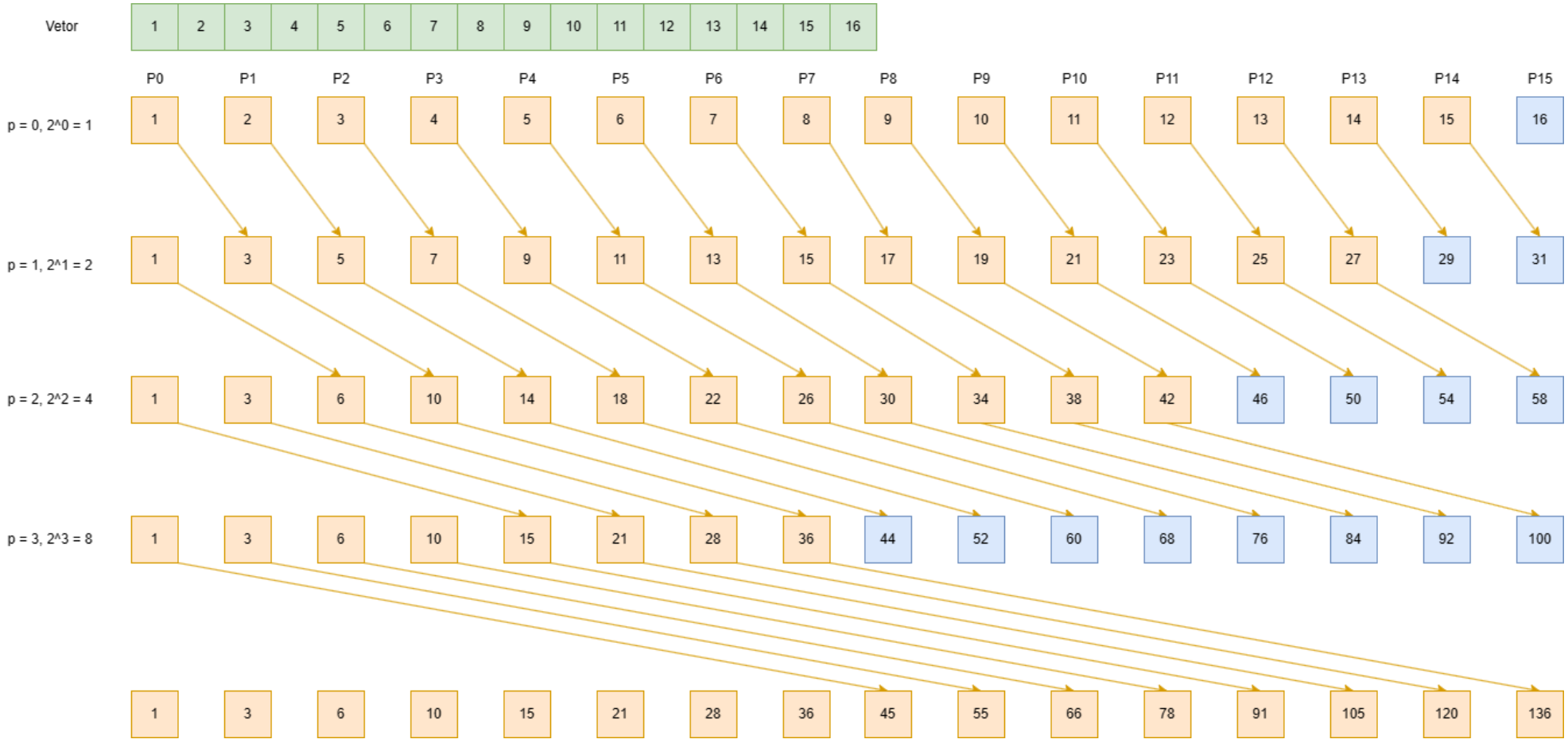
Para realizarmos a resolução desse problema temos que entender como que a função MPI_Allreduce() funciona, ela realiza uma operação de redução global, onde são combinados todos os dados dos processos e distribui o resultado final para todos os processos.

Já a MPI_Scan() realiza uma varredura para os dados distribuídos nos processos, onde serão acumulados os resultados parciais do início até o processo atual.

O diagrama representa como o MPI_Scan() funciona, os valores da esquerda representam o deslocamento no envio dos cores a cada passo, supondo que existe um vetor de 8 elementos e 8 cores disponíveis, podemos entender a execução do Scan da seguinte forma:

- No primeiro passo (p=0), cada core enviará seu valor ao seu vizinho à direita, correspondendo à o passo de valor 1, P0 enviará para P1, P1 para P2 e assim até que P6 envie para P7, P7 não enviará seu valor pois não possui vizinho à direita.
- No segundo passo (p=2), o deslocamento de envio passa a ser feito a partir do segundo vizinho mais à direita, assim P0 envia para P2, P1 envia para P3, P2 envia para P4, P3 envia para P5, P4 envia para P6 e P5 envia para P7, os processos P6 e P7 não farão envios pois não possuem vizinhos mais à direita.
- No terceiro passo (p=4), o deslocamento de envio passa a ser feito a partir do quarto vizinho mais à direita, então: P0 envia para P4, P1 envia para P5, P2 envia para P6 e P3 envia para P7, os processos P4, P5, P6 e P7 não farão nenhum envio.




Vetor
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 P11 P12 P13 P14 P15
p = 0, 2^0 = 1
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
p = 1, 2^1 = 2
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31
p = 2, 2^2 = 4
1 3 6 10 14 18 22 26 30 34 38 42 46 50 54 58
p = 3, 2^3 = 8
1 3 6 10 15 21 28 36 44 52 60 68 76 84 92 100
1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136

11. As funções MPI_Pack e MPI_Unpack oferecem uma alternativa aos tipos de dados derivados para agrupamento de dados. O MPI_Pack copia os dados a serem enviados, um bloco de cada vez, em um buffer fornecido pelo usuário. O buffer pode então ser enviado e recebido. Após a recepção dos dados, o MPI_Unpack pode ser usado para desembalá-los do buffer de recepção. 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     */);
```

Portanto, poderíamos empacotar os dados de entrada no 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 position. Quando o MPI_Pack é chamado, a posição deve referenciar o primeiro slot disponível no pack_buf. Quando MPI_Pack retorna, ele se refere para o primeiro slot disponível após os dados que acabaram de ser embalados, portanto, após o processo 0 executar este código, todos os processos podem chamar MPI_Bcast:

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

Note que o MPI datatype para um buffer embalado é MPI_PACKED. Agora o outros processos podem desempacotar 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     */);
```

Isto pode ser usado "invertendo" as etapas do MPI_Pack, ou seja, os dados são desembrulhado um bloco de cada vez, começando com posição = 0. Escreva outra

função de entrada para o programa de regras trapezoidais. Este deve-se usar o `MPI_Pack` no processo 0 e o `MPI_Unpack` nos outros processos.

O código abaixo mostra o uso de MPI_Send e MPI_Recv para enviar dados recebidos para a regra do trapézio (Pacheco, p.100). Com isso, podemos ter uma ideia de como os dados são transportados entre os processos, de forma semelhante ocorre com o MPI_Pack e MPI_Unpack, onde os dados serão empacotados e enviados a partir de um broadcast e quando for recebido pelos outros processos, será executada a função MPI_Unpack que irá desempacotar os dados.

```
1  void Get_input(
2        int      my_rank   /* in  */,
3        int      comm_sz   /* in  */,
4        double*  a_p       /* out */,
5        double*  b_p       /* out */,
6        int*     n_p       /* out */) {
7     int dest;
8
9     if (my_rank == 0) {
10       printf("Enter a, b, and n\n");
11       scanf("%lf %lf %d", a_p, b_p, n_p);
12       for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16       }
17    } else { /* my_rank != 0 */
18       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19             MPI_STATUS_IGNORE);
20       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21             MPI_STATUS_IGNORE);
22       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23             MPI_STATUS_IGNORE);
24    }
25 }  /* Get_input */
```

Program 3.5: A function for reading user input

```c
#include <mpi.h>

const int PACK_BUF_SIZE = 100;

void Get_input(int my_rank, double *a_p, double *b_p, int *n_p)
{
    int pack_buf = PACK_BUF_SIZE;
    int position = 0;
    int dest;

    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, PACK_BUF_SIZE, &position, MPI_COMM_WORLD);
        MPI_Pack(b_p, 1, MPI_DOUBLE, pack_buf, PACK_BUF_SIZE, &position, MPI_COMM_WORLD);
        MPI_Pack(n_p, 1, MPI_DOUBLE, pack_buf, PACK_BUF_SIZE, &position, MPI_COMM_WORLD);
    }

    MPI_BCast(pack_buf, PACK_BUF_SIZE, MPI_PACKED, 0, MPI_COMM_WORLD);

    if (my_rank > 0)
    {

        MPI_Unpack(pack_buf, PACK_BUF_SIZE, &position, a_p, 1, MPI_DOUBLE, MPI_COMM_WORLD);
        MPI_Unpack(pack_buf, PACK_BUF_SIZE, &position, b_p, 1, MPI_DOUBLE, MPI_COMM_WORLD);
        MPI_Unpack(pack_buf, PACK_BUF_SIZE, &position, n_p, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    }
}
```

12. Cronometre nossa implementação da regra trapezoidal que usa `MPI_Reduce`. Como você escolhe n, o número de trapezoidais? Como comparar o tempo mínimo com o tempo médio e o tempo da mediana? Quais são as velocidades? Quais são as eficiências? Com base nos dados coletados, você diria que a regra do trapézio é escalável?

O valor do número de trapezoidais deve ser alto o bastante para que possa ser possível a medição de tempo a partir da função MPI_Wtime(), com isso em mente, foram utilizadas as dimensões 200M, 400M, 500M, 1B e 2B para 1, 2, 4 e 8 processos, de maneira local. O tempo é obtido realizando a diferença entre o tempo final e o tempo inicial, com isso podemos imprimir o tempo final utilizando o processo mestre.

Com as medições realizadas, podemos obter as medidas que são solicitadas como o tempo médio e a mediana a partir da tabela de tempo de execução, utilizaremos a mediana, pois é a medida que contém menor variação entre as três citadas, com isso poderemos estimar o valor do speedup e eficiência do código.

p = 1	Número de trapézios				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	0.532	1.034	1.293	2.668	5.235
2	0.526	1.035	1.306	2.703	5.274
3	0.517	1.074	1.344	2.658	5.276
4	0.533	1.067	1.327	2.658	5.254
5	0.532	1.039	1.334	2.641	5.347

Mínimo p = 1	0.517	1.034	1.293	2.641	5.235

Tempo médio p = 1	0.528	1.049	1.321	2.665	5.277

Mediana p = 1	0.532	1.039	1.327	2.658	5.274

p = 2	Número de trapézios				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	0.269	0.547	0.761	1.373	2.899
2	0.277	0.569	0.728	1.366	2.823
3	0.262	0.550	0.656	1.393	2.889
4	0.267	0.540	0.653	1.391	2.687
5	0.283	0.560	0.686	1.366	2.769

Mínimo p = 2	0.262	0.540	0.653	1.366	2.687

Tempo médio p = 2	0.2716	0.553	0.696	1.377	2.813

Mediana p = 2	0.269	0.550	0.686	1.373	2.823

p = 4	Número de trapézios				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	0.134	0.284	0.428	0.745	1.540
2	0.140	0.298	0.412	0.903	1.596
3	0.181	0.301	0.357	0.673	2.224
4	0.143	0.288	0.549	0.757	1.740
5	0.137	0.422	0.371	0.763	1.547

Mínimo p = 4	0.134	0.284	0.357	0.673	1.540

Tempo médio p = 4	0.147	0.318	0.423	0.768	1.729

Mediana p = 4	0.140	0.298	0.412	0.757	1.596

p = 8	Número de trapézios				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	0.121	0.234	0.286	0.570	1.219
2	0.105	0.239	0.315	0.563	1.194
3	0.116	0.203	0.265	0.591	1.121
4	0.127	0.263	0.290	0.637	1.259
5	0.115	0.253	0.319	0.632	1.150

Mínimo p = 8	0.105	0.203	0.265	0.563	1.121

Tempo médio p = 8	0.116	0.238	0.295	0.598	1.188

Mediana p = 8	0.116	0.239	0.290	0.591	1.194

Tabela de tempo de execução com valores da mediana (em segundos)

Processos	Número de trapézios				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	0.532	1.039	1.327	2.658	5.274
2	0.269	0.550	0.686	1.373	2.823
4	0.140	0.298	0.412	0.757	1.596
8	0.116	0.239	0.290	0.591	1.194

Tabela de speedup

Processos	Número de trapézios				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	1	1	1	1	1
2	1.97	1.88	1.93	1.93	1.86
4	3.80	3.48	3.22	3.51	3.30
8	4.58	4.34	4.57	4.49	4.41

Tabela de eficiência

Processos	Número de trapézios				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	1	1	1	1	1
2	0.98	0.94	0.96	0.96	0.93
4	0.95	0.87	0.80	0.87	0.82
8	0.57	0.54	0.57	0.56	0.55

As velocidades e eficiências podem ser observadas nas tabelas acima.

Em relação às velocidades, vemos que observando apenas uma dimensão de dados, podemos ver que há um aumento do speedup enquanto aumentamos a quantidade de processos, porém não representa um aumento linear.

Em relação às eficiências vemos que há uma certa estabilidade enquanto aumentamos a quantidade de processos e a quantidade de trapézios, porém baseado nos dados e na diagonal principal podemos dizer que o código não é escalável pois o valor de 0.98 não é superado, também não será fracamente escalável devido à eficiência diminuir quando aumentamos proporcionalmente o tamanho do problema e a quantidade de trapézios.

14. Um ping-pong é uma comunicação na qual duas mensagens são enviadas, primeiro do processo A para o processo B (ping) e depois do processo B de volta para o processo A (pong). Medir o tempo de blocos repetidos de ping-pongs é uma maneira comum de estimar o custo de envio de mensagens. Cronometre um programa de ping-pong usando a função clock em C no seu sistema. Quanto tempo o código precisa executar antes que o clock retorne um tempo de execução diferente de zero? Como os tempos obtidos com a função clock se comparam aos tempos obtidos com o MPI Wtime?

Para essa questão, fiz com que dois processos (0 e 1) se comuniquem utilizando MPI_Send e MPI_Recv e enviem um dado respectivo, para o processo 0 será enviado o ping (valor 0) e o processo 1, quando receber esse dado, enviará como resposta o pong (valor 1), que o processo 0 irá receber.

A execução do programa no supercomputador deve ser realizado a partir de um job, onde é necessário a implementação de um arquivo shell com algumas configurações específicas.

– job-name : nome do job, para esse caso utilizei o número da questão;

– time: tempo de execução do job;

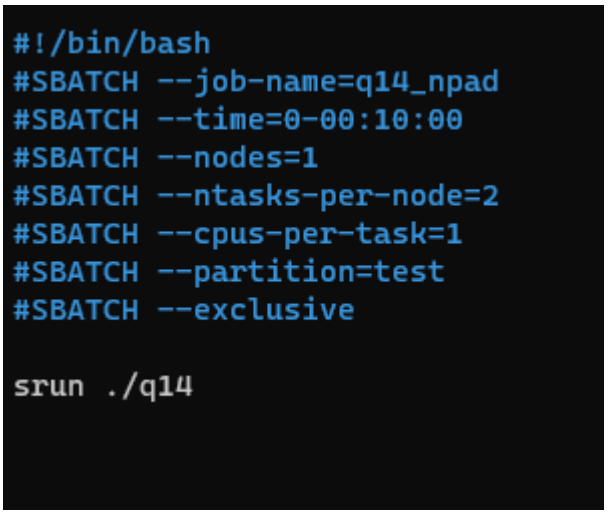
– nodes: quantidade de nós a serem utilizados;

– ntasks-per-node: quantidade de processos que serão executados em cada nó;

– cpus-per-task: quantidade de cpu's que serão utilizados para o job;

– partition: partição test;

– exclusive: alocação exclusiva de recursos;



```c
// Questão 14 -- NPAD
// compile: mpicc q14.c -o q14

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

int main(int argc, char *argv[])
{
    int my_rank, comm_sz;
    int ping = 0;
    int pong = 1;

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

    clock_t start_clock = clock();
    double start_MPI = MPI_Wtime();

    if (my_rank == 0)
    {
        MPI_Send(&ping, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        printf("Processo %d enviou: %d\n", my_rank, ping);
        MPI_Recv(&pong, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Processo %d recebeu: %d\n", my_rank, pong);
    }
    else
    {
        MPI_Recv(&ping, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Processo %d recebeu: %d\n", my_rank, ping);
        MPI_Send(&pong, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Processo %d enviou: %d\n", my_rank, pong);
    }

    clock_t finish_clock = clock();
    double finish_MPI = MPI_Wtime();

    double tempo_clock = (double)(finish_clock - start_clock) / CLOCKS_PER_SEC;
    double tempo_MPI = finish_MPI - start_MPI;

    if (my_rank == 0)
    {
        printf("Tempo de execução do clock: %f\n", tempo_clock);
        printf("Tempo de execução do MPI_Wtime: %f\n", tempo_MPI);
    }

    MPI_Finalize();
}
```

```
[lhadantas@headnode0 ~]$ mpicc q14_npad.c -o q14
[lhadantas@headnode0 ~]$ sbatch sbatch_q14.sh
Submitted batch job 312296
[lhadantas@headnode0 ~]$ tail -f slurm-312296.out

  Local host:   r1i3n2
  Local device: mlx4_0
--------------------------------------------------------------------------
Processo 0 enviou: 0
Processo 0 recebeu: 1
Tempo de execução do clock: 0.004915
Tempo de execução do MPI_Wtime: 0.005034
Processo 1 recebeu: 0
Processo 1 enviou: 1
```

Quanto tempo o código precisa executar antes que o clock retorne um tempo de execução diferente de zero?

A função clock só começa a contar quando o processamento das instruções do programa se iniciam, então todo o tempo que levou anteriormente para se iniciar e finalizar o programa não é contado, sendo medido apenas o tempo em CPU, assim apenas o tempo final é relativamente menor que o tempo utilizando a função MPI_Wtime().

Existe a possibilidade da função de clock retornar valor 0 ou muito próximo à isso pelo motivo do supercomputador possuir um ótimo poder de processamento, sendo assim as instruções rodarão tão rapidamente que o tempo da função clock não poderá ser mensurado.

Como os tempos obtidos com a função clock se comparam aos tempos obtidos com o MPI Wtime?

Os tempos da função de clock dizem respeito apenas do tempo de processamento da requisição de envio e recebimento das funções, ou seja, esse tempo que obtemos para a execução se diz respeito apenas à duração do processamento do programa.

Já o tempo medido utilizando MPI_Wtime() se diz respeito ao relógio de parede, ao tempo total da execução do programa, esse tempo vem acrescido de tempo de início, tempo de finalização, tempo ociosidade do programa, por esse motivo o tempo utilizando MPI_Wtime() é maior do que o medido a partir da função de clock.

15. Encontre as acelerações e eficiências da classificação paralela odd-even. O programa obtém acelerações lineares? É escalável? É fortemente escalável? É fracamente escalável?

Para resolver esse problema utilizando o supercomputador, estimei o tamanho dos elementos em 200 milhões, 400 milhões, 500 milhões, 1 bilhão e 2 bilhões de valores,

utilizando até 32 processos, dividindo-os em 1, 2, 4, 8, 16, 32. Assim, para rodar o job no supercomputador utilizamos um algoritmo específico, chamado sbatch_q15.sh onde existem configurações para que seja criado o job.

Utilizando o supercomputador do NPAD, foram feitas as seguintes configurações para o arquivo sbatch.sh utilizado para subir o job e executar o programa com recursos do supercomputador.

```
#!/bin/bash
#SBATCH --job-name=q15_npad
#SBATCH --time=0-00:30:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=32
#SBATCH --partition=test
#SBATCH --exclusive
#SBATCH --hint=compute_bound

#num de processos
n_processes=(1 2 4 8 16 32)

values=(200000000 400000000 500000000 1000000000 2000000000)

for n_procs in "${n_processes[@]}"; do
    for value in "${values[@]}"; do
        result=$(srun --ntasks $n_procs ./q15 g $value)
        echo "$n_procs processos e entrada $value: $result"
    done
done
```

p = 1	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	30.205	61.951	78.567	161.200	330.180
2	30.210	62.062	78.642	161.257	330.541
3	30.253	62.084	78.661	161.382	330.537
4	30.244	62.112	78.713	161.524	331.094
5	30.274	62.050	78.691	161.382	330.690

Mínimo p = 1	30.205	61.951	78.567	161.200	330.180

Tempo médio p = 1	30.037	62.051	78.454	161.349	330.410

Mediana p = 1	30.244	62.062	78.661	161.382	330.541

p = 2	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	15.313	31.344	39.721	81.397	166.765
2	15.312	31.375	39.775	81.413	166.952
3	15.328	31.381	39.757	81.430	166.784
4	15.301	31.415	39.792	81.598	166.896
5	15.320	31.390	39.775	81.456	166.871

Mínimo p = 2	15.301	31.344	39.721	81.397	166.765

Tempo médio p = 2	15.314	31.581	39.764	81.458	166.653

Mediana p = 2	15.313	31.381	39.775	81.430	166.871

p = 4	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	8.325	17.068	21.508	43.933	89.996
2	8.279	17.048	21.480	43.917	89.366
3	8.324	17.092	21.441	44.065	89.318
4	8.339	17.116	21.600	44.104	90.625

5	8.301	17.003	21.574	43.906	90.082

Mínimo p = 4	8.279	17.003	21.441	43.906	89.318

Tempo médio p = 4	8.313	17.065	21.520	44.185	89.877

Mediana p = 4	8.324	17.068	21.508	43.933	89.996

p = 8	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	5.541	11.232	14.224	28.879	58.991
2	5.440	11.146	14.090	28.676	58.889
3	5.462	11.245	14.174	28.656	58.587
4	5.519	11.249	14.186	28.902	58.892
5	5.494	11.064	14.084	28.515	58.949

Mínimo p = 8	5.440	11.064	14.084	28.515	58.587

Tempo médio p = 8	5.491	11.187	14.151	28.725	58.861

Mediana p = 8	5.494	11.232	14.174	28.676	58.892

p = 16	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	4.949	10.374	12.668	25.703	53.990
2	5.121	10.357	13.190	26.607	51.790
3	4.967	10.125	13.015	25.540	53.820
4	5.124	10.326	12.897	25.510	54.615
5	5.092	10.489	12.859	26.372	54.084

Mínimo p = 16	4.949	10.125	12.668	25.510	51.790

Tempo médio p = 16	5.050	10.334	12.925	25.946	53.659

Mediana p = 16	5.092	10.357	12.897	25.703	53.990

p = 32	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	6.465	13.261	17.194	34.155	66.516
2	6.264	13.385	16.370	31.147	69.260
3	6.399	13.195	15.766	34.468	67.833
4	6.454	13.404	16.432	31.232	63.904
5	6.315	12.732	16.776	34.051	68.536

Mínimo p = 32	6.264	12.732	15.766	31.147	63.904

Tempo médio p = 32	6.379	13.395	16.507	33.010	67.009

Mediana p = 32	6.399	13.261	16.776	31.232	66.516

Tabela de tempo de execução com valores da mediana (em segundos)

Processos	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	30.244	62.062	78.661	161.382	330.541
2	15.313	31.381	39.775	81.430	166.871
4	8.324	17.068	21.508	43.933	89.996
8	5.494	11.232	14.174	28.676	58.892
16	5.092	10.357	12.897	25.703	53.990
32	6.399	13.261	16.776	31.232	66.516

Tabela de speedup

Processos	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	1.00	1.00	1.00	1.00	1.00
2	1.97	1.97	1.97	1.98	1.97
4	3.63	3.63	3.65	3.67	3.67
8	5.50	5.52	5.54	5.62	5.61

16	5.94	5.99	6.10	6.28	6.12
32	4.71	4.67	4.68	5.17	4.96

Tabela de eficiência

Processos	Quantidade de elementos				
	$200x10^6$	$400x10^6$	$500x10^6$	$1x10^9$	$2x10^9$
1	1.00	1.00	1.00	1.00	1.00
2	0.99	0.99	0.99	0.99	0.99
4	0.91	0.91	0.91	0.92	0.92
8	0.69	0.69	0.69	0.70	0.70
16	0.37	0.37	0.38	0.39	0.38
32	0.14	0.15	0.15	0.16	0.16

O programa obtém acelerações lineares?

Não, pois observando a tabela de speedup, existe uma melhora na aceleração do código quando aumentamos a quantidade processos, porém a aceleração linear só existe quando o aumento do número de processos torna o programa p vezes mais rápido do que o programa serial. Dessa forma, para o odd-even não existe uma aceleração linear, mas existe aceleração.

É escalável? É fortemente escalável? É fracamente escalável?

Um programa escalável pode ser definido como ser capaz de manter ou aumentar sua eficiência quando aumentamos a quantidade de elementos para uma mesma quantidade de processos, observando a tabela de eficiência, o programa possui variações muito pontuais em relação à possibilidade de escalabilidade, pois há uma variação mínima, para ser escalável, sendo assim podemos definir que ele é escalável.

O programa não é fortemente escalável pois em todos os casos quando aumentamos a quantidade de processos para um mesmo tamanho de problema a eficiência diminui rapidamente.

O programa não é fracamente escalável pois quando aumentamos proporcionalmente o tamanho dos elementos e a quantidade de processos a eficiência continua diminuindo (como consta na diagonal principal da tabela de eficiência).