Lista Programação Concorrente e distribuída

Resolução feita por Samuel Cavalcanti

Samuel Cavalcanti

Table of contents

1. Lista de Execícios de Programação Concorrente e distribuída

- Capítulo 1 (7 questões)
- Capítulo 2 (13 questões)
- Capítulo 3 (16 questões)
- Capítulo 5 (12 questões)

Capítulo 3 foram realizadas as questões: 2, 4, 6, 9, 11, 12, 13, 16, 17, 19

A versão em pdf desse documento pode ser encontrada em samuel-cavalcanti.github.io/parallel_programming_ufrn/assets/pdf/document.pdf

2. Chapter 1

2.1 Capítulo 1

Capítulo 1: 1-6, 9; (7 questões)

1. Devise formulas for the functions that calculate ***my_first_i*** and ***my_last_i*** in the global sum example. Remember that each core should be assigned roughly the same number of elements of computations in the loop. Hint: First consider the case when ***n*** is evenly divisible by ***p***

- Resposta Questão 1

2. We've implicitly assumed that each call to ***Compute_next_value*** requires roughly the same amount of work as the other calls. How would you change your answer to the preceding question if call $i = k$ requires $k + 1$ times as much work as the call with $i = 0$? So if the first call ($i = 0$) requires 2 milliseconds, the second call ($i = 1$) requires 4, the third ($i = 2$) requires 6, and so on.

- Resposta Questão 2

3. Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, . . .).

- Resposta Questão 3

4. As an alternative to the approach outlined in the preceding problem we can use C's bitwise operators to implement the tree-structured global sum. In order to see how this works, it helps to write down the binary (base 2) representation of each of the core ranks, and note the pairings during each stage

- Resposta Questão 4

5. What happens if your pseudo-code in Exercise 1.3 or Exercise 1.4 is run when the number of cores is not a power of two (e.g., 3, 5, 6, 7) ? Can you modify the pseudo-code so that it will work correctly regardless of the number of cores ?

- Resposta Questão 5

6. Derive formulas for the number of receives and additions that core 0 carries out using:
   a. the original pseudo-code for a global sum
   b. the tree-structured global sum.
   Make a table showing the numbers of receives and additions carried out by core 0 when the two sums are used with 2, 4, 8, . . . , 1024 cores.

- Resposta Questão 6

7.

8.

9. Write an essay describing a research problem in your major that would benefit from the use of parallel computing. Provide a rough outline of how parallelism would be used. Would you use task- or data-parallelism ?

- Resposta Questão 9

2.2 Question 1

2.2.1 Questão 1

Devise formulas for the functions that calculate ***my_first_i*** and ***my_last_i*** in the global sum example. Remember that each core should be assigned roughly the same number of elements of computations in the loop. Hint: First consider the case when n is evenly divisible by p

Observações

Nó magnânimo mundo da programação, é bastante comum adotar a seguinte convenção na hora dividir um vetor em pedaços. Onde o primeiro valor é o indicie que está contido no vetor e segundo valor é limite, por exemplo:

$$vetor = [a_0, b_1, c_2, d_3]$$

$$vetor[1:3] == [b_1, c_2]$$

$$vetor[1:2] == [b_1]$$

Essa será a convenção adota aqui.

Para resolver o problema primeiro, temos que pensar no caso mais simples onde n é divisível pro p, ou seja:

$$\frac{n}{p} = a$$

$$n = ap$$

nesse caso, podemos dar pedaços de tamanho a para cada processo. Onde o primeiro pedaço teria o primeiro índice em 0 e terminaria no indicie $a - 1$. o segundo pedaço começaria em a e terminaria em $2 * a - 1$. ou seja..

$$p_0 = [0 : a]$$

$$p_1 = [a : 2a]$$

$$p_2 = [2a : 3a]$$

$$p_n = [an : (n+1)a]$$

No entanto e se n não for divisível por p, ou seja:

$$n = ap + r$$

onde r é o resto da divisão. Pensando de forma simples, podemos atribuir todo o "restante" no processo 0 e repetir a mesma ideia, ou seja, o processo 0 teria o tamanho $a + r$ e outros processos teria o tamanho a.

p	**tamanho**
0	a+r
1	a
2	a
3	a
..	...
r-1	a
r	a
r +1	a
..	...

Pensando em termos de intervalo ficaria assim:

$$p_0 = [0 : a + r]$$

$$p_1 = [a + r : 2a + r]$$

$$p_2 = [2a + r : 3a + r]$$

$$p_n = [an + r : (n + 1)a + r]$$

Essa primeira ideia é justamente a função **new_range**. Perceba que podemos melhorar ainda mais o algoritmo, podemos dividir r para os primeiros p_r cores até que não "reste" mais nada atribuindo $+1$ no tamanho dos p_r cores.

p	**tamanho**
0	a+1
1	a+1
2	a+1
3	a+1
..	...
r-1	a+1
r	a
r +1	a
..	...

Pensando em termos de intervalo:

$$p_0 = [0 : a + 1]$$

$$p_1 = [a + 1 : 2a + 1]$$

$$p_2 = [2a + 1 : 3a + 1]$$

$$p_{r-1} = [(r - 1)a + 1, (r)a + 1]$$

$$p_r = [ra + 1, (r + 1)a]$$

$$p_{r+1} = [(r + 1)a : (r + 2)a]$$

$$p_n = [an + r : (n + 1)a]$$

Essa é a função **new_range_2**. Para testar ambas as funções foi criado o programa question_1.c.

```
struct range
{
    int first;
    int last;
};
struct range new_range(int thread_index, int p, int n)
{
    struct range r;

    int division = n / p;
    int rest = n % p;

    if (rest == 0)
    {
        r.first = thread_index * division;
        r.last = (thread_index + 1) * division;
    }
    else
    {
        r.first = thread_index == 0 ? 0 : thread_index * division + rest;
        r.last = (thread_index + 1) * division + rest;
    }

    if (r.last > n)
        r.last = n;
```

```
    return r;
}

struct range new_range_2(int thread_index, int p, int n)
{
    struct range r;

    int division = n / p;
    int rest = n % p;

    if (thread_index < rest)
    {
        r.first = thread_index * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = thread_index * division + rest;
        r.last = r.first + division;
    }

    return r;
}
```

Saída:

```
First  0 Last  20 m Last - First: 20
First  20 Last  40 m Last - First: 20
First  40 Last  60 m Last - First: 20
First  60 Last  80 m Last - First: 20
First  80 Last  100 m Last - First: 20
First  0 Last  25 m Last - First: 25
First  25 Last  50 m Last - First: 25
First  50 Last  75 m Last - First: 25
First  75 Last  99 m Last - First: 24
First  99 Last  123 m Last - First: 24
Test question 1 success
OLD new range
First  0 Last  27 m Last - First: 27
First  27 Last  51 m Last - First: 24
First  51 Last  75 m Last - First: 24
First  75 Last  99 m Last - First: 24
First  99 Last  123 m Last - First: 24
```

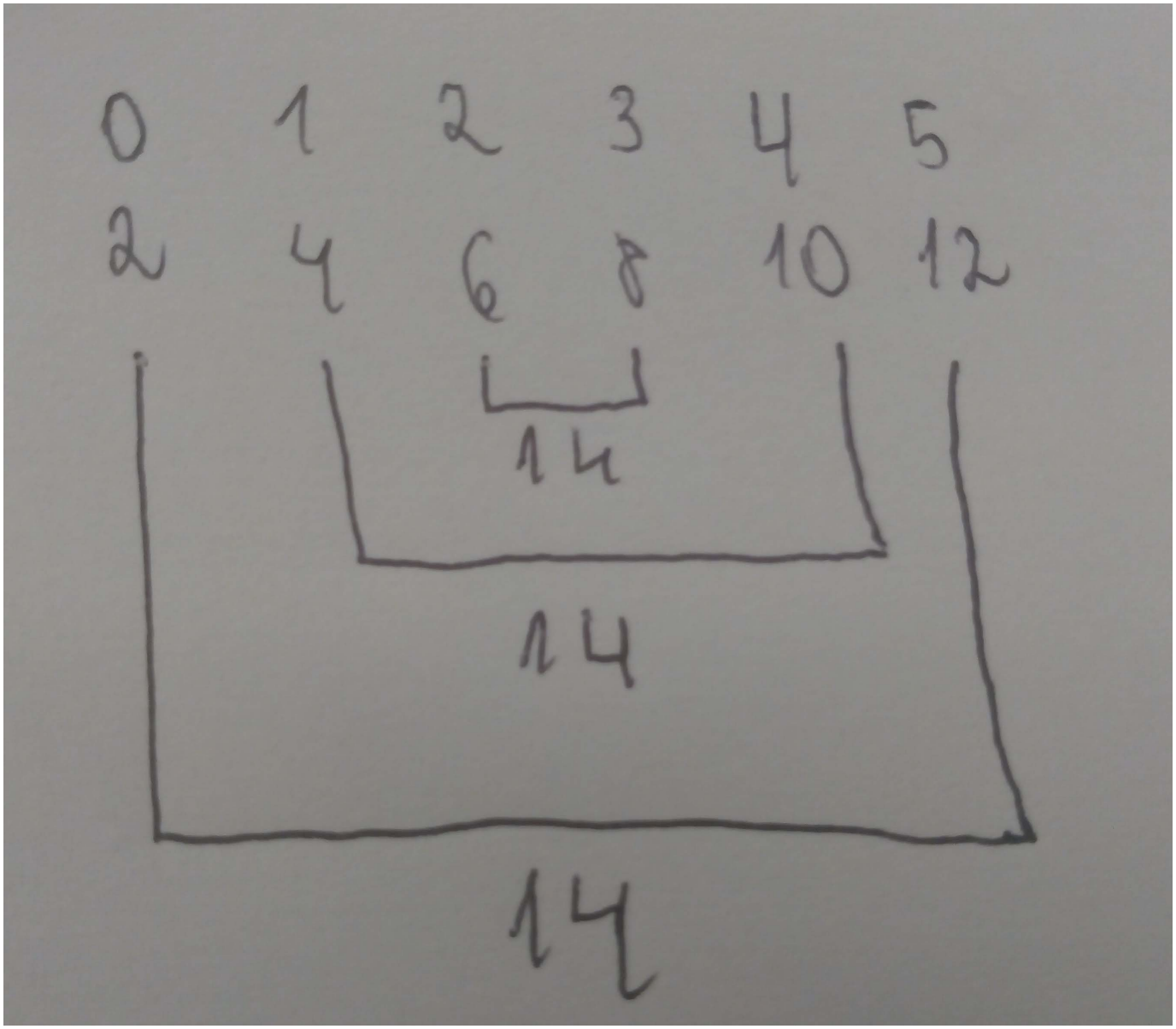
2.3 Question 2

2.3.1 Questão 2

We’ve implicitly assumed that each call to ***Compute_next_value*** requires roughly the same amount of work as the other calls. How would you change your answer to the preceding question if call $i = k$ requires $k + 1$ times as much work as the call with $i = 0$? So if the first call ($i = 0$) requires 2 milliseconds, the second call ($i = 1$) requires 4, the third ($i = 2$) requires 6, and so on.

Sabendo que desenvolvemos um algoritmo muito bom na questão 1 que busca distribuir da forma mais igual possível. Uma solução possível para esse problema seria mapear esse problema para o problema anterior através da soma de gauss.

Somatório de Gauss



Exemplo, Supondo que $k = 10$, temos o seguinte vetor de índice:
$indices = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, utilizando a lógica da somatório de gauss e organizando os indicies temos um array normalizado:

normalized_array $= [[0, 9], [1, 8], [2, 7], [3, 6], [4, 5]]$ onde o custo de cada índice do *normalized_array* é igual, por tanto podemos usar o algoritmo da questão 1 aplicado ao *normalized_array* resultando:

Thread	normalized_array
1	0,1
2	2,3
3	4

Thread	Compute_next_value	cost
1	0, 9,1, 8	44
2	2, 7,3, 6	44
3	4, 5	22

se consideramos que $\frac{k}{2}$ é divisível pro p o resultados são ainda melhores, como $k = 16$ e $p = 4$

Thread	Compute_next_value	cost
1	0, 15, 1, 14	68
2	2, 13, 3, 12	68
3	4, 11, 5, 10	68
4	6, 9, 7, 8	68

2.4 Question 3

2.4.1 Questão 3

Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, . . .).

Para criar a árvore, foi considerado que o vetor principal já foi igualmente espaçado entre as *p* threads, usando o algoritmo da questão 1.

Representação de um Nó

Neste caso foi representado o nó, como uma estrutura que possui um vetor de vizinhos e outro ponteiro para um vetor de inteiros, na prática, o ponteiro para um vetor de inteiros, seria usando o design pattern chamado **Future**, ou um **Option\<Future>**, ou **Option\<thread>**.

Também foi criado dois construtores um construtor que representa, a inicialização do Nó por meio dos seus vizinhos a esquerda e direita, usando na criação de nós intermediários da árvore, e a inicialização do Nó por meio do seu dado, os nós inicializados por meio dos dados abstrai os **núcleos** ou as **threads** que estão sendo usadas para executar o algoritmo de alto custo computacional.

```
class Node
{
public:
    std::vector<Node *> neighborhoods;
    std::vector<int> *data;
    Node(Node *left, Node *right)
    {
        this->neighborhoods.push_back(left);
        this->neighborhoods.push_back(right);
        this->data = nullptr;
    }
    Node(std::vector<int> *data)
    {
        this->data = data;
    }
    ~Node()
    {
        delete this->data;
    }
};
```

Função que cria a Árvore

Para criar a Árvore foi feita uma função recursiva que a partir do nível mais baixo da árvore cria a raiz, ou seja, a partir um vetor com *p* Nós ,a função vai sendo chamada recursivamente, onde a cada chamada vai-se criando um nível acima da árvore, até que se atinja a raiz, onde a cada nível o número de nós é dividido por 2. **Caso o número de nós inicial não for divisível pro 2, o algoritmo não funciona**

```
std::vector<Node *> create_tree_from_core_nodes(std::vector<Node *> nodes)
{
    auto size = nodes.size();

    if (size / 2 == 1)
    {
        auto left = nodes[0];
        auto right = nodes[1];
        receive_value(left, right);  // Left receive value from right
        return {left};
    }

    auto new_nodes = std::vector<Node *>{};

    for (auto i = 0; i < size; i += 2)
    {
        auto left = nodes[i];
        auto right = nodes[i + 1];
        receive_value(left, right); // Left receive value from right
        new_nodes.push_back(left);
    }

    return create_tree_from_core_nodes(new_nodes);
```

```
}

Node *create_new_tree(std::vector<Node *> nodes)
{
    return create_tree_from_core_nodes(nodes)[0];
}
```

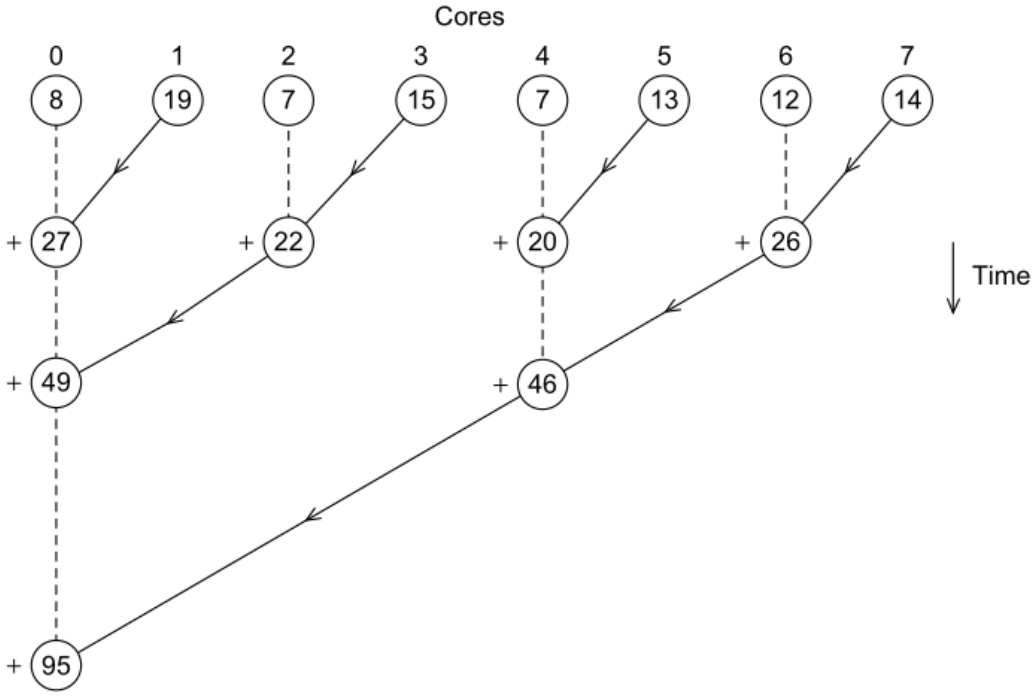


FIGURE 1.1

Multiple cores forming a global sum

Após criar a árvore basta percorrer-lá recursivamente, lembrando que na prática ***compute_data***, seria um *join*, ou um *await*, de uma thread.

```
int compute_data(std::vector<int> *data)
{
    auto total = 0;
    auto size = data->size();
    for (auto i = 0; i < size; i++)
    {
        total += data->at(i);
    }

    return total;
}
int compute_node(Node &node)
{

    int result_data = node.data == nullptr ? 0 : compute_data(node.data);

    for (auto neighborhood : node.neighborhoods)
        result_data += compute_node(*neighborhood);

    return result_data;
}
```

2.5 Question 4

2.5.1 Questão 4

As an alternative to the approach outlined in the preceding problem we can use C’s bitwise operators to implement the tree-structured global sum. In order to see how this works, it helps to write down the binary (base 2) representation of each of the core ranks, and note the pairings during each stage

Semelhante ao questão 3 sendo a diferença utilizar o bitwise **<<** para dividir o tamanho atual da função recursiva:

```
std::vector<Node *> create_new_tree_bitwise(std::vector<Node *> nodes)
{

    auto size = nodes.size();

    if (size >> 1 == 1) // alteração.
    {
        auto left = nodes[0];
        auto right = nodes[1];
        receive_value(left, right);
        return {left};
    }

    auto new_nodes = std::vector<Node *>{};

    for (auto i = 0; i < size; i += 2)
    {
        auto left = nodes[i];
        auto right = nodes[i + 1];
        receive_value(left, right); // Left receive value from right
        new_nodes.push_back(left);
    }

    return create_new_tree_bitwise(new_nodes);
}
```

2.6 Question 5

2.6.1 Questão 5

What happens if your pseudo-code in Exercise 1.3 or Exercise 1.4 is run when the number of cores is not a power of two (e.g., 3, 5, 6, 7) ? Can you modify the pseudo-code so that it will work correctly regardless of the number of cores ?

Entendendo o que acontece quanto os número de núcleos não é da potencia de 2

Se por exemplo, o número de cores, ou Nós for 3 por exemplo, existirá nós que serão "esquecidos" no algoritmo, por tanto o algoritmo não funcionará corretamente.

```
auto size = nodes.size();// size = 3

if (size >> 1 == 1) // alteração.
{
    auto left = nodes[0];
    auto right = nodes[1];
    // node[2] foi esquecido
    receive_value(left, right); // Left receive value from right
    return {left};
}
```

ou se o por exemplo o número de nós for 7, a última iteração do laço ***for***, os indices ***i***, ***i+1***, serão respectivamente 6 e 7, ou seja será acessado um endereço inválido ***7***, uma vez que os indices vão de 0 até 6.

```
    auto new_nodes = std::vector<Node *>{};

    for (auto i = 0; i < size; i += 2)
    {
        auto left = nodes[i];
        auto right = nodes[i + 1];
        receive_value(left, right); // Left receive value from right
        new_nodes.push_back(left);
    }

    return create_new_tree_bitwise(new_nodes);
```

Para isso foram feitas as seguintes modificações:

- adicionado condicionamento para verificar se o tamanho é igual 3
- alterado o bitwise para apenas um comparador ***size ==2***
- verificado se o tamanho dos nós é par, caso não seja adicionado uma logica extra.

```
std::vector<Node *> create_new_tree_bitwise(std::vector<Node *> nodes)
{

    auto size = nodes.size();

    if (size == 2)
    {
        auto left = nodes[0];
        auto right = nodes[1];
        receive_value(left, right);
        return {left}; // Construtor C++ Moderno.
    }
    if (size == 3)
    {
        auto left = nodes[0];
        auto middle = nodes[1];
        auto right = nodes[2];
        receive_value(left, middle); // Left receive value from middle
        receive_value(left, right);  // Left receive value from right
        return {left};               // Construtor C++ Moderno.
    }

    auto new_nodes = std::vector<Node *>{};

    if (size % 2 != 0) // lógica extra.
    {
        size = size - 1;
        new_nodes.push_back(nodes[size]);
    }

    for (auto i = 0; i < size; i += 2)
```

```
    {
        auto left = nodes[i];
        auto right = nodes[i + 1];
        receive_value(left, right); // Left receive value from right
        new_nodes.push_back(left);
    }

    return create_new_tree_bitwise(new_nodes);
}
```

Explicando a lógica extra

Além de adicionar uma verificação para saber se o tamanho é par, foi adicionado dois comandos extras, o primeiro é alterar o tamanho (size), para um valor menor, uma vez que estávamos acessando um índice maior que o permitido. Segundo foi adicionar o nó que não será percorrido pelo laço para o vetor ***new_nodes*** que será a entrada da próxima função recursiva

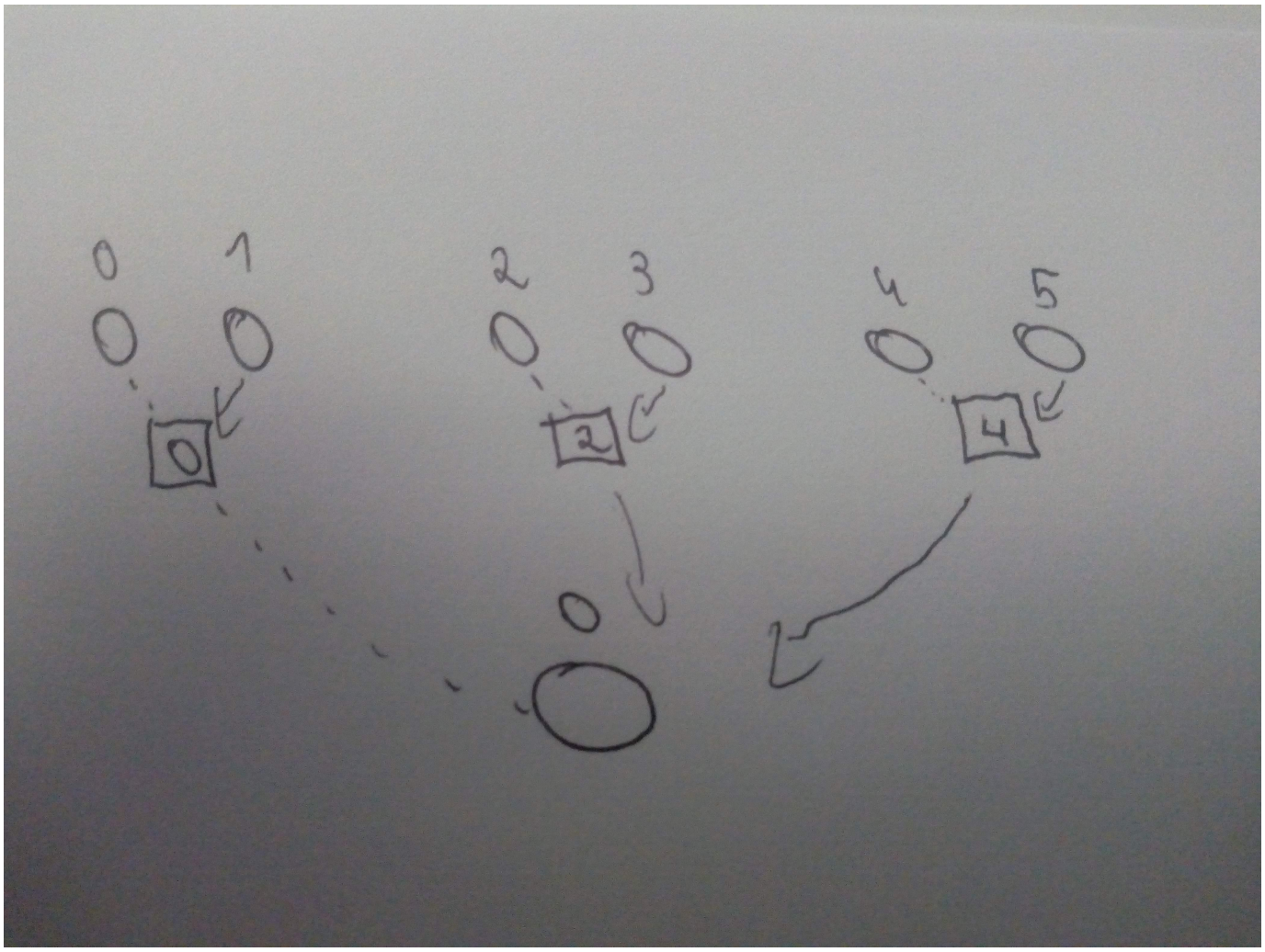
```
if (size % 2 != 0) // verificação se é par.
   {
       size = size - 1; //1
       new_nodes.push_back(nodes[size]); // 2
   }
```

Explicando o novo caso base

Percebemos que além do 2/2 == 1, a divisão inteira de 3/2 também é igual 1. Por tanto além do caso base de quando o tamanho do vetor de nós ser igual a 2, temos que tratar também quando o número de nós ser igual a 3.

```
if (size == 2)
   {
       auto left = nodes[0];
       auto right = nodes[1];
       receive_value(left, right);
       return {left}; // Construtor C++ Moderno.
   }
   if (size == 3)
   {
       auto left = nodes[0];
       auto middle = nodes[1];
       auto right = nodes[2];
       receive_value(left, middle); // Left receive value from middle
       receive_value(left, right);  // Left receive value from right
       return {left};               // Construtor C++ Moderno.
   }
```

Como no exemplo abaixo, onde a segunda iteração do algoritmo o número de nós é 3.



2.7 Question 6

2.7.1 Questão 6

Derive formulas for the number of receives and additions that core 0 carries out using:
a. the original pseudo-code for a global sum
b. the tree-structured global sum.
Make a table showing the numbers of receives and additions carried out by core 0 when the two sums are used with 2, 4, 8, . . . , 1024 cores.

Tabela mostrando o número de receives, ou joins que o core 0 terá

- A coluna Cores representa o número de núcleos, que estão em potência de 2.
- A coluna Naive, é o número de receives que o core 0 terá caso utilizado a abordagem ingenua do core 0 esperar todos os outros e somar todo de uma vez.
- A coluna Tree é o número de receives que o core 0 terá utilizando a abordagem em árvore.

Cores	Naive	Tree
2	1	1
4	3	2
8	7	3
16	15	4
32	31	5
64	63	6
128	127	7
256	255	8
512	512	9
1024	1023	10

Podemos observar claramente que a abordagem ingênua segue a formula, $p - 1$ e quando usamos a árvore, percebemos que a cada 2 núcleos, o número de ligações amentar em $p = 2^n$, ou seja, $n = log_2(p)$. Podemos ver o número de ligações crescendo dois

núcleos na imagem abaixo

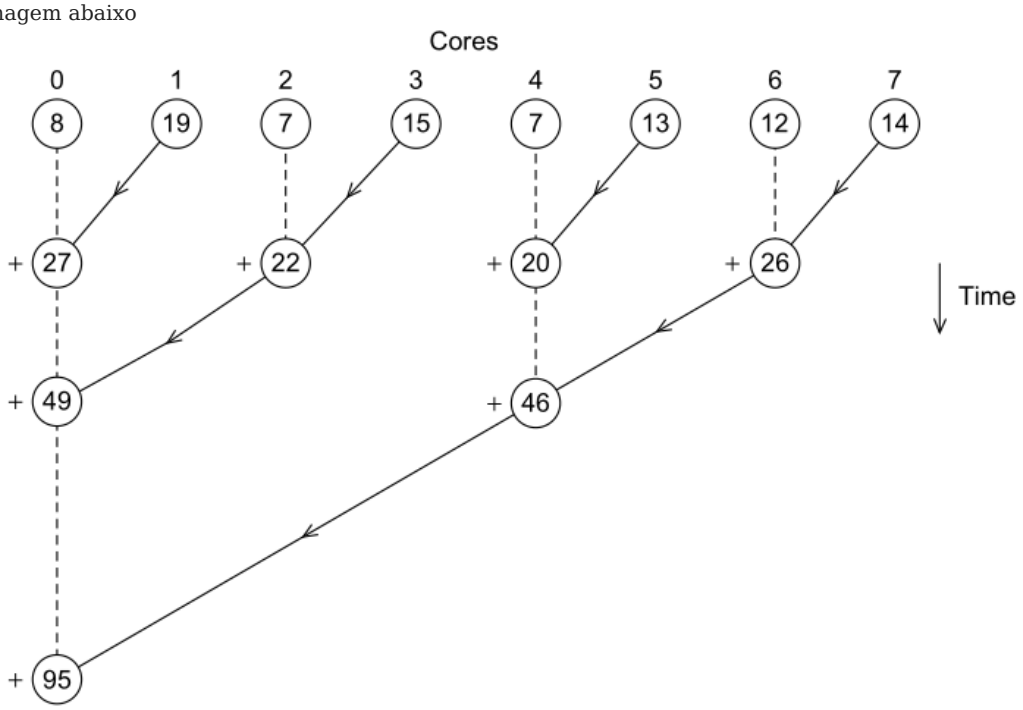


FIGURE 1.1

Multiple cores forming a global sum

2.8 Question 9

2.8.1 Questão 9

Write an essay describing a research problem in your major that would benefit from the use of parallel computing. Provide a rough outline of how parallelism would be used. Would you use task- or data-parallelism ?

TinyML in Context of web browser

Em contexto de aplicações web, especificamente a camada front-end, vem se popularizando frameworks que utilizam o WebAssembly (Wasm). O WebAssembly é um novo tipo de código que pode ser executado em browsers modernos — se trata de uma linguagem de baixo nível como assembly, com um formato binário compacto que executa com performance quase nativa e que fornece um novo alvo de compilação para linguagens como C/C++, para que possam ser executadas na web developer.mozilla.org. Através do Wasm é possível implementar algoritmos de Tiny machine learning (TinyML) para classificação, analise de dados sem a necessidade de comunicação com backend. TinyML é amplamente definido como uma rápida e crescente área de aprendizado de máquina e suas aplicações que inclui hardware, algoritmos e software capazes de performar analise de dados em dispositivos de baixo consumo de energia, tipicamente em milliwatts, assim habilitado variados casos de uso e dispositivos que possuem bateria. Em um navegador, ou durante a navegação o usuário está a todo momento produzindo dados que muitas vezes estão sendo enviados de forma bruta ou quase bruta para a camada de aplicação ou back-end, onde nela é gasto processamento e memória para a primeira etapa de classificação ou analise dos dados. Uma vez analisado desempenho de técnicas e algoritmos de TinyML utilizando WebAssembly, pode ser possível transferir a responsabilidade da analise dos dados para o front-end. Em contexto de navegador quase tudo é paralelo ou distribuído, uma aba ou *tab* em inglês é um processo diferente. Criar um uma extensão, que faça aquisição e analise dos dados de diferentes abas, seria criar um sistema que se comunica com diferentes processos por meio de mensagens e o algoritmo de aprendizado pode fazer uso de idealmente uma ou duas threads para realizar a analise rapidamente. Por tanto é um sistema que é task-and-data parallel.

3. Chapter 2

3.1 Capítulo 2

Capítulo 2: 1-3, 5, 7, 10, 15-17, 19-21, 24; (13 questões)

1. When we were discussing floating point addition, we made the simplifying assumption that each of the functional units took the same amount of time. Suppose that fetch and store each take 2 nanoseconds and the remaining operations each take 1 nanosecond.

a. How long does a floating point addition take with these assumptions ?

b. How long will an unpipelined addition of 1000 pairs of floats take with these assumptions ?

c. How long will a pipelined addition of 1000 pairs of floats take with these assumptions ?

d. The time required for fetch and store may vary considerably if the operands/results are stored in different levels of the memory hierarchy. Suppose that a fetch from a level 1 cache takes two nanoseconds, while a fetch from a level 2 cache takes five nanoseconds, and a fetch from main memory takes fifty nanoseconds. What happens to the pipeline when there is a level 1 cache miss on a fetch of one of the operands? What happens when there is a level 2 miss ?

- Resposta questão 1

2. Explain how a queue, implemented in hardware in the CPU, could be used to improve the performance of a write-through cache.

- Resposta questão 2

3. Recall the example involving cache reads of a two-dimensional array (page 22). How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops? What happens if MAX = 8 and the cache can store four lines ? How many misses occur in the reads of A in the first pair of nested loops ? How many misses occur in the second pair ?

- Resposta questão 3

4.

5. Does the addition of cache and virtual memory to a von Neumann system change its designation as an SISD system ? What about the addition of pipelining? Multiple issue? Hardware multithreading ?

- Resposta questão 5

6.

7. Discuss the differences in how a GPU and a vector processor might execute the following code:

```
sum = 0.0;
for (i = 0; i < n; i++) {
   y[i] += a*x[i];
   sum += z[i]*z[i];
}
```

- Resposta questão 7

8.

9.

10. Suppose a program must execute 10^{12} instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10^6 seconds (about 11.6 days). So, on average, the single processor system executes 10^6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute 10^{12} /p instructions and each processor must send 10^9 (p − 1) messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to things such as waiting for messages.

a. Suppose it takes 10^{-9} seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run ?

b. Suppose it takes 10^{-3} seconds to send a message. How long will it take the program to run with 1000 processors ? Resposta questão 10

11.

12.

13.

14.

15. a. Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable x in its cache, and it executes the assignment x = 5. Finally suppose that core 1 doesn't have x in its cache, and after core 0's update to x, core 1 tries to execute y = x. What value will be assigned to y ? Why ?

    b. Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to y ? Why ?

    c. Can you suggest how any problems you found in the first two parts might be solved ?

    - Resposta questão 15

16. a. Suppose the run-time of a serial program is given by $T_{serial} = n^2$, where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time $T_{parallel} = \frac{n^2}{p} + log_2(p)$. Write a program that finds the speedups and efficiencies of this program for various values of n and p. Run your program with $n = 10, 20, 40, \ldots, 320$, and $p = 1, 2, 4, \ldots, 128$. What happens to the speedups and efficiencies as p is increased and n is held fixed? What happens when p is fixed and n is increased ?

    b. Suppose that $T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$. Also suppose that we fix p and increase the problem size.
    - Show that if $T_{overhead}$ grows more slowly than T_{serial} , the parallel efficiency will increase as we increase the problem size. - Show that if, on the other hand, $T_{overhead}$ grows faster than T_{serial}, the parallel efficiency will decrease as we increase the problem size.

    - Resposta questão 16

17. A parallel program that obtains a speedup greater than p—the number of processes or threads—is sometimes said to have **superlinear speedup**. However, many authors don't count programs that overcome "resource limitations" as having superlinear speedup. For example, a program that must use secondary storage for its data when it's run on a single processor system might be able to fit all its data into main memory when run on a large distributed-memory system. Give another example of how a program might overcome a resource limitation and obtain speedups greater than p

    - Resposta questão 17

18.

19. Suppose $T_{serial} = n$ and $T_{parallel} = \frac{n}{p} + log_2(p)$, where times are in microseconds. If we increase p by a factor of k, find a formula for how much we'll need to increase n in order to maintain constant efficiency. How much should we increase n by if we double the number of processes from 8 to 16 ? Is the parallel program scalable ?

    - Resposta questão 19

20. Is a program that obtains linear speedup strongly scalable ? Explain your answer.

    - Resposta questão 20

21. Bob has a program that he wants to time with two sets of data, *input_data1* and *input_data2*. To get some idea of what to expect before adding timing functions to the code he's interested in, he runs the program with two sets of data and the Unix shell command *time*:

    ```
    $ time ./bobs prog < input data1
    real 0m0.001s
    user 0m0.001s
    sys 0m0.000s
    $ time ./bobs prog < input data2
    real 1m1.234s
    user 1m0.001s
    sys 0m0.111s
    ```

    The timer function Bob is using has millisecond resolution. Should Bob use it to time his program with the first set of data ? What about the second set of data ? Why or why not ?

    - Resposta questão 21

22.

23.

24. If you haven't already done so in Chapter 1, try to write pseudo-code for our tree-structured global sum, which sums the elements of *loc_bin_cts*. First consider how this might be done in a shared-memory setting. Then consider how this might be done in a distributed-memory setting. In the shared-memory setting, which variables are shared and which are private ?

    - Resposta questão 24

3.2 Question 01

3.2.1 Questão 1

When we were discussing floating point addition, we made the simplifying assumption that each of the functional units took the same amount of time. Suppose that fetch and store each take 2 nanoseconds and the remaining operations each take 1 nanosecond.
a. How long does a floating point addition take with these assumptions ?

b. How long will an unpipelined addition of 1000 pairs of floats take with these assumptions ?

c. How long will a pipelined addition of 1000 pairs of floats take with these assumptions ?

d. The time required for fetch and store may vary considerably if the operands/results are stored in different levels of the memory hierarchy. Suppose that a fetch from a level 1 cache takes two nanoseconds, while a fetch from a level 2 cache takes five nanoseconds, and a fetch from main memory takes fifty nanoseconds. What happens to the pipeline when there is a level 1 cache miss on a fetch of one of the operands? What happens when there is a level 2 miss ?

1.a

Instructions	Time in nanosecond
Fetch	2
Store	2
Functional OP	1

"As an alternative, suppose we divide our floating point adder into seven separate pieces of hardware or functional units. The first unit will fetch two operands, the second will compare exponents, and so on." (Página 26)

O Author do livro considera que existe sete operações, considerando que duas delas são fetch e store custa 2 nanosegundos e o restante 1 nanosegundo.

$$1 \times 5 + 2 \times 2 = 9 \text{ nanosegundos}$$

1.b

Considerando que exitem 1000 pares de valores vão serem somados:

$$1000 \times 9 = 9000 \text{ nanosegundos}$$

1.c

foi pensado o seguinte: No momento que o dado passa pelo fetch, e vai para a próxima operação já é realizado o fetch da segunda operação. Executando o pipeline:

Tempo em nanosegundos	Fetch	OP1	OP2	OP3	OP4	OP5
0	1	wait	wait	wait	wait	wait
2	2	1	wait	wait	wait	wait
3	2	wait	1	wait	wait	wait
4	3	2	wait	1	wait	wait
5	3	wait	2	wait	1	wait
6	4	3	wait	2	wait	1
7	4	wait	3	wait	2	wait
8	5	4	wait	3	wait	2
9	5	wait	4	wait	3	wait
10	6	5	wait	4	wait	3
11	6	wait	5	wait	4	wait

Percebe-se que a primeira instrução irá ser finalizada ou sumir na tabela quanto for 9 segundos ou seja,a primeira instrução dura 9 nanosegundos, no entanto, no momento em que a primeira instrução é finalizada,a segunda já começa a ser finalizada ou seja, demora apenas 2 nanosegundos até segunda operação ser finalizada e mais 2 nanosegundos para a terceira ser finalizada e assim por diante. Por tanto para executar todos os 1000 dados, o custo total fica:

$$9 + 999 * 2 = 2007$$

1.d

No caso, considerando que a cache nível 1 não falhe a tabela continua mesma, pois o fetch e store custam o mesmo 2 nanosegundos:

Tempo em nanosegundos	Fetch	OP1	OP2	OP3	OP4	OP5
10	6	5	wait	4	wait	3
11	6	wait	5	wait	4	wait

mas se imaginarmos que na 12 iteração o Fetch e Store passa a custar 5 nanosegundos:

Tempo em nanosegundos	Fetch	OP1	OP2	OP3	OP4	OP5
10	6	5	wait	4	wait	3
11	6	wait	5	wait	4	wait
12	6	wait	wait	5	wait	4
13	6	wait	wait	wait	5	4
14	6	wait	wait	wait	5	4
15	7	6	wait	wait	5	4
16	7	wait	6	wait	wait	5

Quando mais lento fica a transferência para a memória principal, mais nítido fica o gargalo de Von Neumann, ou seja, percebe-se que a performance do processador fica limitado a taxa de transferência de dados com a memória principal.

3.3 Question 02

3.3.1 Questão 2

Explain how a queue, implemented in hardware in the CPU, could be used to improve the performance of a write-through cache.

Como observado na questão 1 cada momento que a escrita é debilitada, fica nítido o gargalo de Von Neuman se considerarmos que uma escrita na cache é uma escrita na memória principal, então cada Store iria demorar ***50*** nano segundos. Colocando uma fila e supondo que ela nunca fique cheia, a CPU não irá gastar tanto tempo no Store, mas uma vez a fila cheia, a CPU terá que aguardar uma escrita na memória principal.

Tabela com cache miss questão 1

Tempo em nanosegundos	**Fetch**	**OP1**	**OP2**	**OP3**	**OP4**	**OP5**
10	6	5	wait	4	wait	3
11	6	wait	5	wait	4	wait
12	6	wait	wait	5	wait	4
13	6	wait	wait	wait	5	4
14	6	wait	wait	wait	5	4
15	7	6	wait	wait	5	4
16	7	wait	6	wait	wait	5

3.4 Question 03

3.4.1 Questão 3

Recall the example involving cache reads of a two-dimensional array (page 22). How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops? What happens if MAX = 8 and the cache can store four lines ? How many misses occur in the reads of A in the first pair of nested loops ? How many misses occur in the second pair ?

Exemplo envolvendo leituras na cache em um array bidimensional

```
double A[MAX][MAX], x[MAX], y[MAX];

//. . .
// Initialize A and x, assign y = 0 */
//. . .


/* First pair of loops */
for (i = 0; i < MAX; i++)
    for (j = 0; j < MAX; j++)
        y[i] += A[i][j]*x[j];

//. . .
// Assign y = 0 */
//. . .

/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j];
```

Cache line	Elements of A
0	$A[0][0], A[0][1], A[0][2], A[0][3]$
1	$A[1][0], A[1][1], A[1][2], A[1][3]$
2	$A[2][0], A[2][1], A[2][2], A[2][3]$
3	$A[3][0], A[3][1], A[3][2], A[3][3]$

How does a larger matrix and a larger cache affect the performance of the two pairs of nested loops ?

Supondo que a cache tenha a mesma proporção do que a Matrix, o número de cache miss seria igual ao número de linhas da matriz, como apontado no exemplo dado no livro, quando o processador Pede o valor $A[0][0]$, baseado na ideia de vizinhança, a cache carrega todas as outras colunas da linha 0, portanto é plausível pensar que o número de miss é igual ao número de linhas, ou seja, o número miss é igual a $\boldsymbol{MAX}$, pois a cache tem o mesmo número de linhas que a matrix A, suponto que não preciso me preocupar com $\boldsymbol{x}$ e $\boldsymbol{y}$.

if MAX = 8 and the cache can store four lines ? How many misses occur in the reads of A in the first pair of nested loops ? How many misses occur in the second pair ?

FIRST PAIR OF LOOPS

Tendo a a cache armazenando metade dos valores de uma linha da Matriz A, então para cada linha da Matriz, vai haver duas cache miss, a primeira np $A[i][0]$ e a segunda no $A[i][4]$. Outro ponto é que como a cache só possui 4 linhas, então após ocorrer os cache misses $A[0][0]$,$A[0][4]$ e $A[1][0]$, $A[1][4]$ toda a cache terá sindo preenchida, ou seja, Tento a matriz 8 linhas e para cada linha tem 2 cache miss por tanto:

$8 * 2 = 16$ cache miss

como tanto a primeira parte quando na segunda parte, percorre-se todas as linhas irá haver 16 cache miss, suponto que não preciso me preocupar com $\boldsymbol{x}$ e $\boldsymbol{y}$.

Cache line	Elements of A
0	$A[0][0], A[0][1], A[0][2], A[0][3]$
1	$A[0][4], A[0][5], A[0][6], A[0][7]$
2	$A[1][0], A[1][1], A[1][2], A[1][3]$
3	$A[1][4], A[1][5], A[1][6], A[1][7]$

SECOND PAIR OF LOOPS

No segundo par de loops, vemos que o segundo laço for, percorre os valores: $A[0][0]$, $A[1][0]$, $A[2][0]$ ... $A[7][0]$, para quando $j = 0$. Isso faz com que todo hit seja miss, ou seja iremos ter miss para cada acesso em A, portanto:

$$8 * 8 = 64 \text{ cache miss}$$

```
/* Second pair of loops */
for (j = 0; j < MAX; j++)
    for (i = 0; i < MAX; i++)
        y[i] += A[i][j]*x[j]; // cada acesso um miss.
```

3.5 Question 05

3.5.1 Questão 5

Does the addition of cache and virtual memory to a von Neumann system change its designation as an SISD system ? What about the addition of pipelining? Multiple issue? Hardware multithreading ?

Um SISD system ou Single Instruction Single Data system, são sistemas que executam uma única instrução por vez e sua taxa de transferência de dados é de um item por vez também.

the addition of cache and virtual memory

Adicionar um cache e memória virtual, pode ajudar a reduzir o tempo que **única** instrução é lida da memória principal, mas não aumenta o número de instruções buscadas na operação Fetch ou na Operação Store, por tanto o sistema continua sendo Single Instruction Single Data.

the addition of pipelining

Como demostrado na questão 1, ao adicionar um pipeline, podemos realizar a mesma instrução complexa em múltiplos dados, ou seja, Single Instruction Multiple Data System, portanto sim.

the addition of Multiple issue and Hardware multithreading

No momento em que possibilitamos uma máquina executar antecipadamente uma instrução ou possibilitamos a execução de múltiplas threads, nesse momento então a máquina está executando várias instruções em vários dados ao mesmo tempo, por tanto o sistema se torna Multiple Instruction Multiple Data system, ou seja, a designação muda.

3.6 Question 07

3.6.1 Questão 7

Discuss the differences in how a GPU and a vector processor might execute the following code:

```
sum = 0.0;
for (i = 0; i < n; i++) {
   y[i] += a*x[i];
   sum += z[i]*z[i];
}
```

CPU com vetorização

Um processo de vetorização em cima desse laço for dividiria as entradas em chucks ou blocos de dados e executaria em paralelo a instrução complexa. Algo como:

```
//executando o bloco paralelamente.
y[0] += a*x[0];
y[1] += a*x[1];
y[2] += a*x[2];
y[3] += a*x[3];

z[0]*z[0] // executando em paralelo
z[1]*z[1] // executando em paralelo
z[2]*z[2] // executando em paralelo
z[3]*z[3] // executando em paralelo
// somando tudo depois
sum+=  z[0]*z[0] + z[1]*z[1] + z[2]*z[2] + z[3]*z[3];
```

GPU

Atualmente essa operação em GPU é muito mais interessante, pois hoje podemos compilar ou gerar instruções complexas que podem ser executas em GPU. A primeira vantagem seria separar o calculo do sum:

```
    sum += z[i]*z[i];
```

do calculo do y

```
     y[i] += a*x[i];
```

ficando assim:

SHADERS GLSL EQUIVALENTE SUPONTO QUE N = 4

```
# version 330

layout(location = 0) in vec4 x;
layout(location = 1) in mat4 a;
layout(location = 2) in vec4 y;

/* o buffer gl_Position é ré-inserido no y */
void main()
{
    gl_Position =  a*x + y;
}
//
```

```
# version 330

layout(location = 0) in mat4 z;
uniform float sum;

/* transpose é uma função que calcula a transposta já existem no Glsl */
void main()
{
    mat4 temp =  transpose(z) * z;

    sum = 0;
     for (int i = 0; i < 4; i++)
     // desde que o laço for seja baseado em constantes ou variáveis uniforms
     // esse laço for é possível.
```

```
    {
        sum += temp[i];
    }

    // recupera o valor no index 0
    gl_Position = vec4(sum, 0.0, 0.0, 0.0);
}
```

A grande vantagem de usar os shaders seria dependendo do tamanho do vetor de dados, executar as instruções de uma só vez em todos os dados, na prática assim como a vetorização envia em blocos, na GPU você também enviaria em blocos, comumente chamados de buffers, a grande diferença seria justamente no fato que um bloco na GPU possui um tamanho muito maior que o bloco da vetorização.

3.7 Question 10

3.7.1 Questão 10

Suppose a program must execute 10^{12} instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in 10^6 seconds (about 11.6 days). So, on average, the single processor system executes 10^6 or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses p processors, each processor will execute $\frac{10^{12}}{p}$ instructions and each processor must send $10^9(p-1)$ messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to things such as waiting for messages.

1. Suppose it takes 10^{-9} seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run ?
2. Suppose it takes 10^3 seconds to send a message. How long will it take the program to run with 1000 processors ?

main.py

```
import datetime

NUMBER_OF_INSTRUCTIONS = 10**12
NUMBER_OF_MESSAGES = 10**9
AVERANGE_SECOND_PER_INSTRUCTIONS = (10**6) / NUMBER_OF_INSTRUCTIONS


def cost_time_per_instruction(instructions: int) -> float:
    return AVERANGE_SECOND_PER_INSTRUCTIONS * instructions


def number_of_instructions_per_processor(p: int) -> int:
    return NUMBER_OF_INSTRUCTIONS/p


def number_of_messagens_per_processor(p: int) -> int:
    return NUMBER_OF_MESSAGES * (p-1)


def simulate(time_per_message_in_seconds: float, processors: int):
    print(
        f'time to send a message: {time_per_message_in_seconds} processors: {processors}')
    instructions = number_of_instructions_per_processor(processors)
    number_of_messages = number_of_messagens_per_processor(processors)
    each_process_cost_in_seconds = cost_time_per_instruction(instructions)

    total_messages_in_seconds = time_per_message_in_seconds * number_of_messages

    result = total_messages_in_seconds + each_process_cost_in_seconds
    result_date = datetime.timedelta(seconds=result)

    print(f'executing instructions is {instructions}')
    print(f'spend sending messages is {total_messages_in_seconds}')

    print(f'total time in seconds: {result}')
    print(f'total time in HH:MM:SS {result_date}')


def a():

    time_per_message_in_seconds = 1e-9
    processors = 1e3
    simulate(time_per_message_in_seconds, processors)


def b():
    time_per_message_in_seconds = 1e-3
    processors = 1e3
    simulate(time_per_message_in_seconds, processors)


def main():
    print('A:')
    a()
    print('B:')
    b()
```

```
if __name__ == '__main__':
    main()
```

Executando:

```
python chapter_2/question_10/main.py
A:
time to send a message: 1e-09 processors: 1000.0
executing instructions is 1000000000.0
spend sending messages is 999.0000000000001
total time in seconds: 1999.0
total time in HH:MM:SS 0:33:19
B:
time to send a message: 0.001 processors: 1000.0
executing instructions is 1000000000.0
spend sending messages is 999000000.0
total time in seconds: 999001000.0
total time in HH:MM:SS 11562 days, 12:16:40
```

11562 dias são 32 anos.

3.8 Question 15

3.8.1 Questão 15

1. Suppose a shared-memory system uses snooping cache coherence and write-back caches. Also suppose that core 0 has the variable x in its cache, and it executes the assignment x = 5. Finally suppose that core 1 doesn’t have x in its cache, and after core 0’s update to x, core 1 tries to execute y = x. What value will be assigned to y ? Why ?
2. Suppose that the shared-memory system in the previous part uses a directory-based protocol. What value will be assigned to y ? Why ?
3. Can you suggest how any problems you found in the first two parts might be solved ?

1 What value will be assigned to y ? Why ?

Não é possível determinar qual valor será atribuído ao y independentemente se for write-back ou write-through, uma vez que não houve uma sincronização entre os cores sobre o valor de x. A atribuição de y = x do core 1 pode ocorrer antes ou depois das operações no core 0.

shared-memory system in the previous part uses a directory-based protocol What value will be assigned to y ? Why ?

Com o sistema de arquivos, ao core 0 irá notificar o a memória principal que a consistência dos dados foi comprometida, no entanto, ainda não dá para saber qual o valor de y, uma vez que a atribuição de y = x do core 1 pode ocorrer antes ou depois das operações no core 0.

Can you suggest how any problems you found in the first two parts might be solved ?

Existe dois problemas, o problema da consistência do dados, temos que garantir que ambos os cores façam alterações que ambas sejam capaz de ler e o segundo é um mecanismo de sincronização, onde por exemplo, o core 1 espera o core 0 finalizar o seu processamento com a variável x para ai sim começar o seu. Podemos utilizar por exemplo um mutex, onde inicialmente o core 0 faria o lock e ao finalizar ele entrega a chave a qual, o core 1 pegaria.

3.9 Question 16

3.9.1 Questão 16

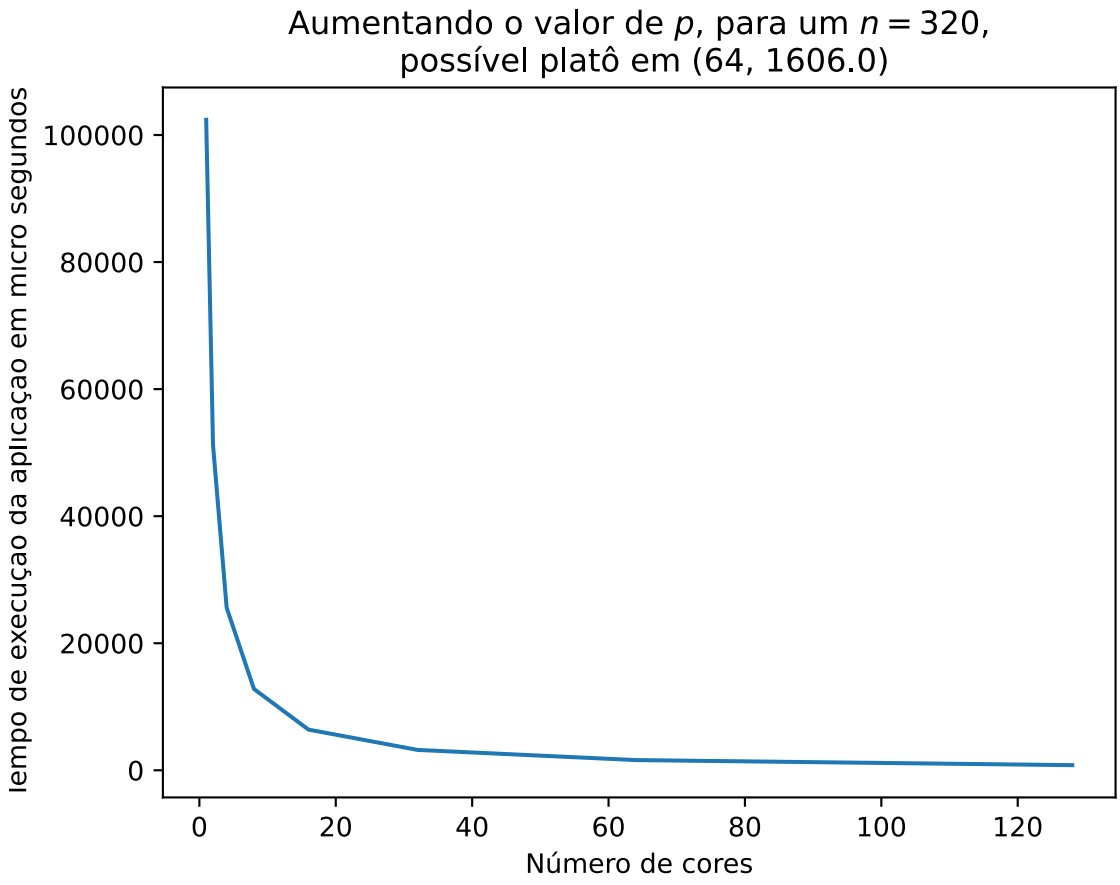
a. Suppose the run-time of a serial program is given by $T_{serial} = n^2$, where the units of the run-time are in microseconds. Suppose that a parallelization of this program has run-time $T_{parallel} = \frac{n^2}{p} + log_2(p)$. Write a program that finds the speedups and efficiencies of this program for various values of n and p. Run your program with $n = 10, 20, 40, \ldots, 320$, and $p = 1, 2, 4, \ldots, 128$. What happens to the speedups and efficiencies as p is increased and n is held fixed? What happens when p is fixed and n is increased ?

b. Suppose that $T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$. Also suppose that we fix p and increase the problem size.

- Show that if $T_{overhead}$ grows more slowly than T_{serial} , the parallel efficiency will increase as we increase the problem size.
- Show that if, on the other hand, $T_{overhead}$ grows faster than T_{serial}, the parallel efficiency will decrease as we increase the problem size.

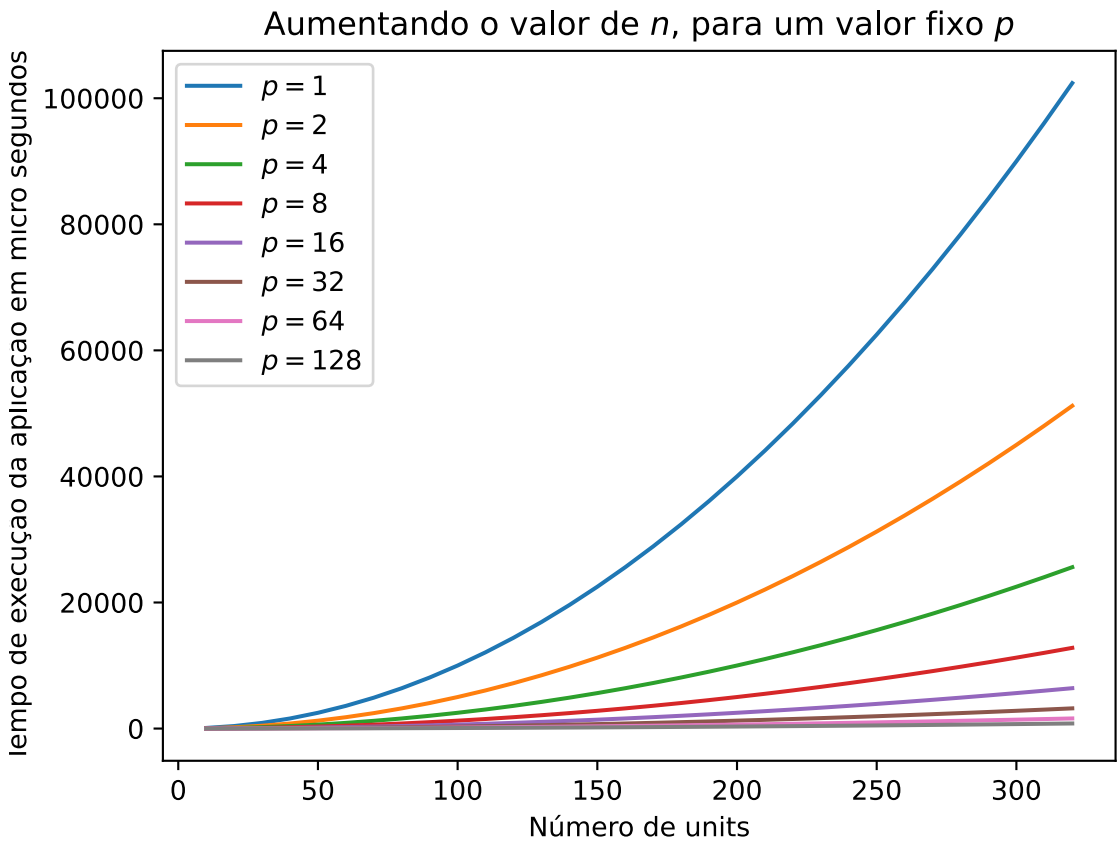
16 a

WHAT HAPPENS TO THE SPEEDUPS AND EFFICIENCIES AS P IS INCREASED AND N IS HELD FIXED ?



Podemos observar a Amadahl's law, "a lei de Amadahl diz: a menos que virtualmente todo o programa serial seja paralelizado, o possível speedup, ou ganhado de performance será muito limitado, independentemente dos números de cores disponíveis." Nó exemplo em questão, podemos observar que a partir de 32 cores, o tempo da aplicação fica estagnado por volta dos 1606 micro segundos.

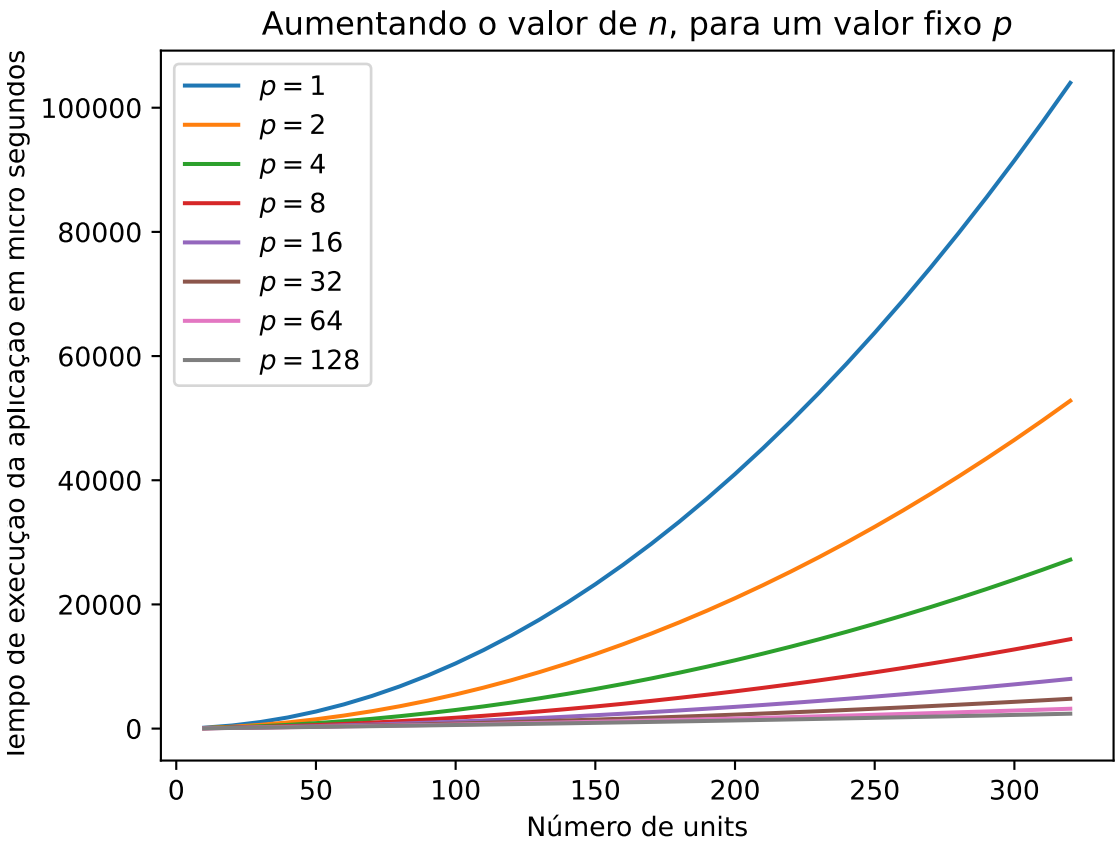
WHAT HAPPENS WHEN P IS FIXED AND N IS



Mesmo com da lei de Amdahl, podemos observar que o aumento de performance é bastante significante, valendo a pena paralelizar a aplicação, também pelo fato que os hardwares atualmente possuem mais de um core.

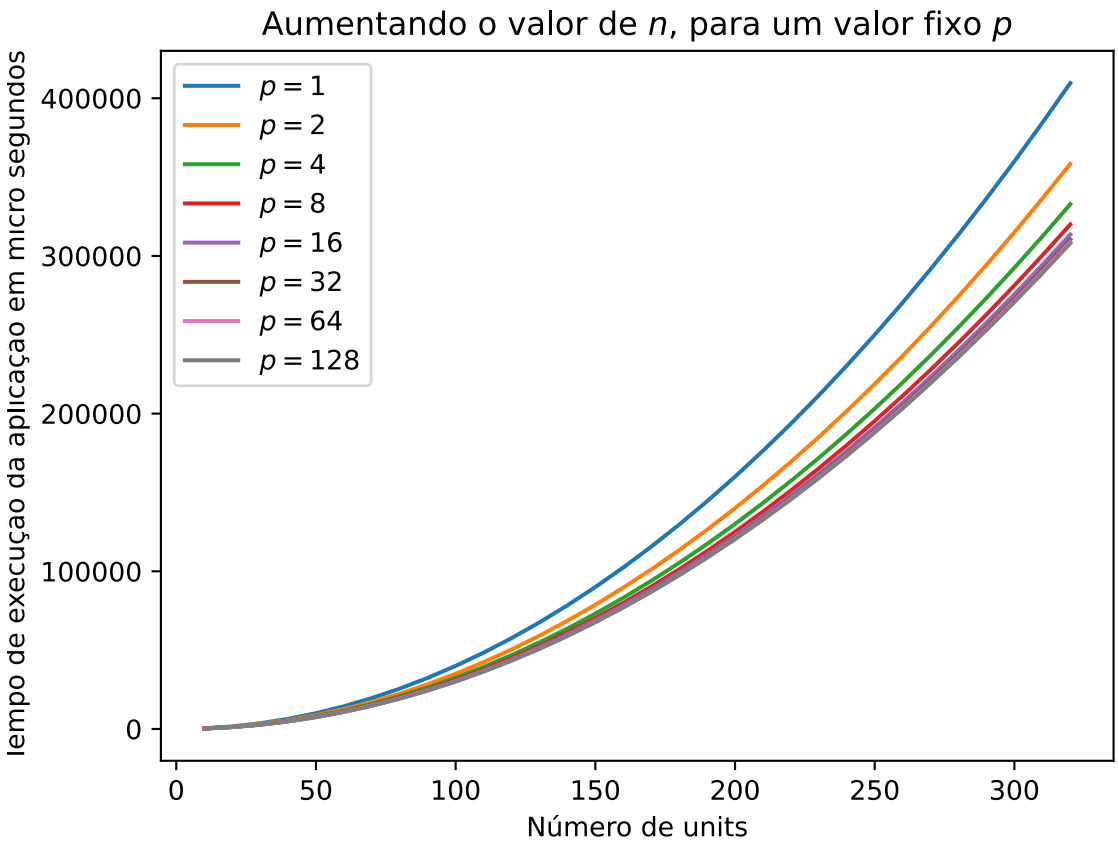
16 b

SHOW THAT IF $T_{overhead}$ GROWS MORE SLOWLY THAN T_{serial} , THE PARALLEL EFFICIENCY WILL INCREASE AS WE INCREASE THE PROBLEM SIZE.



Sabendo que o tempo serial é o quadrático da entrada, considerei o tempo de overhead sendo: $T_{overhead} = 5n$, ou seja, uma função linear. Comparando com o gráfico da letra ***a*** dificilmente é notado uma diferença entre os gráficos, podendo até ser desconsiderado.

SHOW THAT IF, ON THE OTHER HAND, $T_{overhead}$ GROWS FASTER THAN T_{serial}, THE PARALLEL EFFICIENCY WILL DECREASE AS WE INCREASE THE PROBLEM SIZE.



Sabendo que o tempo serial é o quadrático da entrada, considerei o tempo de overhead sendo: $T_{overhead} = 3T_{serial} = 3n^2$, ou seja, o overhead cresce 3 vezes mais que o serial. Comparando com o gráfico da letra ***a*** podemos observar que paralelizar não é uma boa opção, pois nem com $p = 128$ consegue ser melhor do que com $p = 1$, ou seja, não paralelizar, ou seja, a solução serial sem o overhead.

Observações

Para essa atividade foi utilizado Python na sua versão ***3.10.4*** para instalar as dependências do script e criar os gráficos:

```
# para criar um ambiente virtual, suponto que tenha python 3.10.4 instalado
python -mnenv .env
source .env/bin/activate
# instalando as dependências e executando a aplicação
pip install -r requirements.txt
python main.py
```

3.10 Question 17

3.10.1 Questão 17

A parallel program that obtains a speedup greater than p—the number of processes or threads—is sometimes said to have **superlinear speedup**. However, many authors don’t count programs that overcome “resource limitations” as having superlinear speedup. For example, a program that must use secondary storage for its data when it’s run on a single processor system might be able to fit all its data into main memory when run on a large distributed-memory system. Give another example of how a program might overcome a resource limitation and obtain speedups greater than p

Multiplicação Matricial

Como demonstrado por george hotz em **can you multiply a matrix? (noob lesson)** quando otimizado o algoritmo de Multiplicação de matrizes de forma que busca extrair ao máximo o uso da cache, se obtém mais que o dobro de de Gflops. No entanto também é observado Thermal Throttling que reduz a frequência do processador, impossibilitado o superlinear speedup com todos os cores. O que aconteceu, foi que toda a matriz é capaz de ser carregada na cache compartilhada, mas não é capaz de ser totalmente carregada na cache exclusiva do core, então quando se dividiu a matriz entre os cores a performance mais que dobra pois ela se soma com a vantagem da cache.

vídeo da implementação https://www.youtube.com/watch?v=VgSQ1GOC86s.

3.11 Question 19

3.11.1 Questão 19

Suppose $T_{serial} = n$ and $T_{parallel} = \frac{n}{p} + log_2(p)$, where times are in microseconds. If we increase p by a factor of k, find a formula for how much we'll need to increase n in order to maintain constant efficiency. How much should we increase n by if we double the number of processes from 8 to 16 ? Is the parallel program scalable ?

Encontre uma formula para o quanto nós teremos que aumentar n para obter uma eficiência constante.

$E(p) = \frac{T_{serial}}{pT_{parallel}}$

$E(p) = \frac{n}{p(\frac{n}{p} + log_2(p))}$

$E(p) = \frac{n}{n + plog_2(p)}$

$E(kp) = \frac{n}{n + kplog_2(kp)}$

Se considerarmos a constante A o número de vezes que temos que aumentar n para obter uma eficiência constante, logo:

$E_a(kp) = \frac{An}{An + kplog_2(kp)}$

$E_a(kp) = E(p)$

$\frac{An}{An + kplog_2(kp)} = \frac{n}{n + plog_2(p)}$

$An = \frac{n(nA + kplog_2(kp))}{n + plog_2(p)}$

$A = \frac{nA + kplog_2(kp)}{n + plog_2(p)}$

$A = \frac{nA}{n + plog_2(p)} + \frac{kplog_2(kp)}{n + plog_2(p)}$

$A - \frac{nA}{n + plog_2(p)} = \frac{kplog_2(kp)}{n + plog_2(p)}$

$A[1 - \frac{n}{n + plog_2(p)}] = \frac{kplog_2(kp)}{n + plog_2(p)}$

$A[\frac{n + plog_2(p)}{n + plog_2(p)} - \frac{n}{n + plog_2(p)}] = \frac{kplog_2(kp)}{n + plog_2(p)}$

$A\frac{plog_2(p)}{n + plog_2(p)} = \frac{kplog_2(kp)}{n + plog_2(p)}$

$Aplog_2(p) = kplog_2(kp)$

$A = \frac{kplog_2(kp)}{plog_2(p)}$

$A = \frac{klog_2(kp)}{log_2(p)}$

Quantas vezes nós devemos aumentar n se nós dobrarmos o número de cores de 8 para 16

$A(k, p) = \frac{klog_2(kp)}{log_2(p)}$ se $k = 2$ e $p = 8$ então:

$A = \frac{2log_2(16)}{log_2(8)}$

$A = \frac{2(4)}{3}$

$A = \frac{8}{3}$

O programa paralelo é escalável ?

Dado a definição do autor sim. Para o autor escalável é quando a eficiência de um programa paralelo se mantém constante, ou seja, se existe uma taxa que relaciona o crescimento do tamanho do problema com o crescimento do número de threads/ processos, então o programa paralelo é fracamente escalável (***weakly scalable***), .

3.12 Question 20

3.12.1 Questão 20

Is a program that obtains linear speedup **strongly** scalable ? Explain your answer.

Dado a definição do autor sim. Para o autor escalável é quando a eficiência de um programa paralelo se mantém constante. Linear speedup pode ser escrito pela seguinte expressão:

$S = \frac{T_{serial}}{T_{parallel}} = p$, onde p é número de cores e S é o speedup.

Portanto dado que eficiência é dado pela seguinte expressão:

$E = \frac{T_{serial}}{pT_{parallel}}$, onde T_{serial} é o tempo da aplicação em serial e $T_{parallel}$ o tempo da aplicação em paralelizada.

se o speedup for linear, ou seja, $S = p$, temos que

$E = \frac{S}{p}$, portanto

$E = \frac{p}{p} = 1$, Como a eficiência é constante, logo, por definição a aplicação é fortemente escalável (***strongly scalable***).

3.13 Question 21

3.13.1 Questão 21

Bob has a program that he wants to time with two sets of data, *input_data1* and *input_data2*. To get some idea of what to expect before adding timing functions to the code he's interested in, he runs the program with two sets of data and the Unix shell command *time*:

```
$ time ./bobs prog < input data1
real 0m0.001s
user 0m0.001s
sys 0m0.000s
$ time ./bobs prog < input data2
real 1m1.234s
user 1m0.001s
sys 0m0.111s
```

The timer function Bob is using has millisecond resolution. Should Bob use it to time his program with the first set of data ? What about the second set of data ? Why or why not ?

Segundo a referência o commando time, retorna três valores:

- real, que o tempo total desde a inicialização até a terminação do programa
- user, o total de cpu usado pelo programa em modo usuário, ou seja,o a quantidade de cpu usada, eliminando chamadas do sistema e o tempo que o sistema ficou bloqueado ou aguardando outros processos.
- Sys, o tempo de cpu usado pelo kernel/ SO para esse processo em específico.

Primeiro timer

Na primeira chamada observamos que o tempo coletado é praticamente 0, ou seja, o tempo levado para executar o programa esta fora da resolução do relógio do sistema, por tanto não podemos concluir nada sobre a primeira chamada e se essa for a primeira chamada é bem provável que a próxima também dê praticamente 0, uma vez que a aplicação pode ter pouco tamanho de entrada se comparado a máquina em questão.

Segundo timer

Já no segundo timer podemos observar informações sobre a aplicação, estão dentro da resolução do relógio e que maior parte da aplicação foi gasta em em user mode. Bob pode fazer proveito dessas informações.

3.14 Question 24

3.14.1 Questão 24

If you haven't already done so in Chapter 1, try to write pseudo-code for our tree-structured global sum, which sums the elements of loc_bin_cts. First consider how this might be done in a shared-memory setting. Then consider how this might be done in a distributed-memory setting. In the shared-memory setting, which variables are shared and which are private ?

Um código em C++, mas sem o uso de estruturas de dados de programação paralela, pode ser observado na resposta da questão 5 Question_5.cpp. Também foi implementado o **mesmo** algoritmo em rust e nessa implementação foi utilizado threads, smart pointers e mutex para resolver o problema. O código pode ser observado aqui: main.rs

Main.rs

```rust
use std::{
    sync::{Arc, Mutex},
    thread::JoinHandle,
};

#[derive(Debug)]
struct Node {
    data: Vec<i32>,
    neighborhoods: Mutex<Vec<JoinHandle<i32>>>,
}

impl Node {
    fn from_value(data: i32) -> Node {
        Node {
            data: vec![data],
            neighborhoods: Mutex::new(vec![]),
        }
    }

    fn compute(self) -> i32 {
        /*
            Em termos de memória essa função desaloca toda memória
            usada pela estrutura Node e retorna um inteiro de 32bits.
            Dado que foi utilizado safe rust e o código compila, logo esse
            código está livre data race e como não referências cíclicas
            também está livre de memory leak.
         */
        let result: i32 = self.data.iter().sum();

        let neighborhoods = self.neighborhoods.into_inner().unwrap();

        let neighborhoods_sum: i32 = neighborhoods
            .into_iter()
            .map(|handle| handle.join().expect("Unable to lock neighborhood"))
            .sum();

        result + neighborhoods_sum
    }
}

fn start_to_compute_node(node: Node) -> JoinHandle<i32> {
    std::thread::spawn(move || {
        let result = node.compute();
        std::thread::sleep(std::time::Duration::from_micros(500));

        result
    })
}

fn receive_value(left: Arc<Node>, right: Arc<Node>) {
    let right = Arc::try_unwrap(right).unwrap();

    let mut left_neighborhoods = left
        .neighborhoods
        .lock()
        .expect("Unable to lock neighborhood");

    left_neighborhoods.push(start_to_compute_node(right))
}

fn create_new_tree_bitwise(mut nodes: Vec<Arc<Node>>) -> Vec<Arc<Node>> {
    let size = nodes.len();

    match size {
        2 => {
            let left = nodes.remove(0);
            let right = nodes.remove(0);
```

```
            receive_value(left.clone(), right);

            vec![left]
        }

        3 => {
            let left = nodes.remove(0);
            let middle = nodes.remove(0);
            let right = nodes.remove(0);

            receive_value(left.clone(), middle);
            receive_value(left.clone(), right);

            vec![left]
        }

        _ => {
            let mut new_nodes = vec![];

            let mut size = size;

            if size % 2 != 0 {
                size = size - 1;

                new_nodes.push(nodes.remove(size - 1));
            }

            for i in (0..size).step_by(2) {
                let left = nodes.remove(0);
                let right = nodes.remove(0);
                println!("i: {} left: {:?} right: {:?}", i, left, right);
                receive_value(left.clone(), right);
                new_nodes.push(left);
            }
            println!("Next iteration");

            create_new_tree_bitwise(new_nodes)
        }
    }
}

fn main() {
    let data = vec![8, 19, 7, 15, 7, 13, 12, 14];

    let nodes: Vec<Arc<Node>> = data
        .clone()
        .iter()
        .map(|&v| Node::from_value(v))
        .map(|node| Arc::new(node))
        .collect();

    let root = create_new_tree_bitwise(nodes)[0].clone();

    let root = Arc::try_unwrap(root).unwrap();

    let total = root.compute();

    assert!( total ==  data.iter().sum::<i32>());
    println!(
        "total: {} data sum {} ",
        total,
        data.iter().sum::<i32>()
    );
}

#[test]
fn tree_sum() {
    /*
        Teste com várias entradas
    */
    let data_tests = vec![
        vec![8, 19, 7, 15, 7, 13, 12, 14],
        vec![8, 19, 7, 15, 7, 13, 12],
        vec![8, 19, 7, 15, 7, 13],
        vec![8, 19, 7, 15, 7],
        vec![8, 19, 7, 15],
        vec![8, 19, 7],
        vec![8, 19],
    ];

    for data in data_tests {
        let nodes: Vec<Arc<Node>> = data
            .clone()
            .iter()
            .map(|&v| Node::from_value(v))
            .map(|node| Arc::new(node))
            .collect();

        let root = create_new_tree_bitwise(nodes)[0].clone();

        let root = Arc::try_unwrap(root).unwrap();

        assert_eq!(root.compute(), data.iter().sum::<i32>());
```

```
    }
}
```

Observação

caso tenha o rust instalado você pode observar a execução do caso de teste com o comando **cargo**.

```
cargo test
```

O teste é mesmo teste feito em c++.

4. Chapter 3

4.1 Capítulo 3

Capítulo 3: 2, 4, 6, 9, 11, 12, 13, 16, 17, 19, 20, 22, 23, 27 e 28 (16 questões);

1.

2. Modify the trapezoidal rule so that it will correctly estimate the integral even if *comm_sz* doesn't evenly divide n. (You can still assume that n $\geq$ *comm_sz*.).

- Resposta questão 2

3.

4. Modify the program that just prints a line of output from each process (*mpi_output.c*) so that the output is printed in process rank order: process 0s output first, then process 1s, and so on.

- Resposta questão 4

5.

6. Suppose *comm_sz* = 4 and suppose that $\mathbf{x}$ is a vector with $n = 14$ components.

a. How would the components of $\mathbf{x}$ be distributed among the processes in program that used a block distribution ?

b. How would the components of $\mathbf{x}$ be distributed among the processes in a program that used a cyclic distribution ?

c. How would the components of x be distributed among the processes in a program that used a block-cyclic distribution with blocksize $b = 2$?

- Resposta questão 6

7.

8.

9. Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by *comm_sz*

- Resposta questão 9

10.

11. Finding **prefix sums** is a generalization of global sum. Rather than simply finding the sum of n values,

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

the prefix sums are the n partial sums

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

a. Devise a serial algorithm for computing the n prefix sums of an array with n elements.

b. Parallelize your serial algorithm for a system with n processes, each of which is storing one of the *x_is*.

c. Suppose $n = 2^k$ for some positive integer k. Can you devise a serial algorithm and a parallelization of the serial algorithm so that the parallel algorithm requires only k communication phases ?

d. MPI provides a collective communication function, MPI Scan, that can be used to compute prefix sums:

```
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  */);
```

It operates on arrays with *count* elements; both *sendbuf_p* and *recvbuf_p* should refer to blocks of *count* elements of type *datatype*. The *op* argument is the same as op for *MPI_Reduce*. Write an MPI program that generates a random array of *count* elements on each MPI process, finds the prefix sums, and print the results.

- Resposta questão 11

12. An alternative to a butterfly-structured allreduce is a **ring-pass** structure. In a ring-pass, if there are p processes, each process q sends data to process $q+1$, except that process $p-1$ sends data to process 0. This is repeated until each process has the desired result. Thus, we can implement allreduce with the following code:

```
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. Write an MPI program that implements this algorithm for allreduce. How does its performance compare to the butterfly-structured allreduce ?

b. Modify the MPI program you wrote in the first part so that it implements prefix sums.

- Resposta questão 12

13. MPI Scatter and MPI Gather have the limitation that each process must send or receive the same number of data items. When this is not the case, we must use the MPI functions MPI Gatherv and MPI Scatterv. Look at the man pages for these functions, and modify your vector sum, dot product program so that it can correctly handle the case when n isn't evenly divisible by *comm_sz*.

- Resposta questão 13

14.

15.

16. Suppose $comm\_sz = 8$ and the vector $\mathbf{x} = (0, 1, 2, \ldots, 15)$ has been distributed among the processes using a block distribution. Draw a diagram illustrating the steps in a butterfly implementation of allgather of $\mathbf{x}$

- Resposta questão 16

17. *MPI_Type_contiguous* can be used to build a derived datatype from a collection of contiguous elements in an array. Its syntax is

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

Modify the *Read_vector* and *Print_vector* functions so that they use an MPI datatype created by a call to *MPI_Type_contiguous* and a *count* argument of 1 in the calls to *MPI_Scatter* and *MPI_Gather*.

- Resposta questão 17

18.

19. *MPI_Type_indexed* can be used to build a derived datatype from arbitrary array elements. Its syntax is

```
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 */);
```

Unlike *MPI_Type_create_struct*, the displacements are measured in units of *old_mpi_t* --not bytes. Use *MPI_Type_indexed* to create a derived datatype that corresponds to the upper triangular part of a square matrix. For example in the 4×4 matrix.

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

the upper triangular part is the elements 0, 1, 2, 3, 5, 6, 7, 10, 11, 15. Process 0 should read in an $n \times n$ matrix as a one-dimensional array, create the derived datatype, and send the upper triangular part with a single call to *MPI_Send*. Process 1 should receive the upper triangular part with a single call ot *MPI_Recv* and then print the data it received.

- Resposta questão 19

20. The functions *MPI_Pack* and *MPI_Unpack* provide an alternative to derived datatypes for grouping data. *MPI_Pack* copies the data to be sent, one block at a time, into a user-provided buffer. The buffer can then be sent and received. After the data is received, *MPI_Unpack* can be used to unpack it from the receive buffer. The syntax of *MPI_Pack* is

```
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 */);
```

We could therefore pack the input data to the trapezoidal rule program with the following code:

```
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);
```

The key is the position argument. When *MPI_Pack* is called, position should refer to the first available slot in *pack_buf*. When *MPI_Pack* returns, it refers to the first available slot after the data that was just packed, so after process 0 executes this code, all the processes can call *MPI_Bcast*:

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

Note that the MPI datatype for a packed buffer is MPI_PACKED. Now the other processes can unpack the data using: *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 */);
```

This can be used by “reversing” the steps in *MPI_Pack*, that is, the data is unpacked one block at a time starting with position = 0. Write another Get input function for the trapezoidal rule program. This one should use *MPI_Pack* on process 0 and *MPI_Unpack* on the other processes.

21.

22. Time our implementation of the trapezoidal rule that uses *MPI_Reduce*. How will you choose n, the number of trapezoids ? How do the minimum times compare to the mean and median times ? What are the speedups ? What are the efficiencies ? On the basis of the data you collected, would you say that the trapezoidal rule is scalable ?

23. Although we don’t know the internals of the implementation of *MPI_Reduce*, we might guess that it uses a structure similar to the binary tree we discussed. If this is the case, we would expect that its run-time would grow roughly at the rate of $log_2(p)$, since there are roughly $log_2(p)$ levels in the tree. (Here, $p = comm\_sz$.) Since the run-time of the serial trapezoidal rule is roughly proportional to n, the number of trapezoids, and the parallel trapezoidal rule simply applies the serial rule to $\frac{n}{p}$ trapezoids on each process, with our assumption about *MPI_Reduce*, we get a formula for the overall run-time of the parallel trapezoidal rule that looks like
$T_{parallel}(n, p) \approx a \times \frac{n}{p} + b log_2(p)$
for some constants a and b.

a. Use the formula, the times you’ve taken in Exercise 3.22, and your favorite program for doing mathematical calculations to get a least-squares estimate of the values of a and b.

b. Comment on the quality of the predicted run-times using the formula and the values for a and b computed in part (a).

24.

25.

26.

27. Find the speedups and efficiencies of the parallel odd-even sort. Does the program obtain linear speedups? Is it scalable ? Is it strongly scalable ? Is it weakly scalable ?

28. Modify the parallel odd-even transposition sort so that the Merge functions simply swap array pointers after finding the smallest or largest elements. What effect does this change have on the overall run-time ?

4.2 Question 02

4.2.1 Questão 2

Modify the trapezoidal rule so that it will correctly estimate the integral even if *comm_sz* doesn't evenly divide n. (You can still assume that n $\geq$ *comm_sz*.).

Código antes da alteração

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


double trap(double a, double b, long int n);

double function(double x);

int main(int argc, char *argv[])
{

  int my_rank;
  int comm_sz;

  int message_tag = 0;
  // resceber argumentos globais via linha de commando
  MPI_Init(&argc, &argv);

  const double A = atof(argv[1]);
  const double B = atof(argv[2]);
  const int N = atoi(argv[3]);



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

  double h = (B - A) / N;

  int local_n = N / comm_sz;
  /*
    Como podemos observar que quando a divisão
    inteira N/comm_sz o n local, não corresponde
    ao número total de trapézios desejados.
    Ex: 1024/3 ==  341, sendo que 341*3 == 1023
   */
  double local_a = A + my_rank * local_n * h´;
  double local_b = local_a + local_n * h;
  double result_integral = trap(local_a, local_b, local_n);

  if (my_rank != 0)
  {
    MPI_Send(&result_integral, 1, MPI_DOUBLE, 0, message_tag, MPI_COMM_WORLD);
  }
  else // my_rank ==0
  {

    for (int source = 1; source < comm_sz; source++)
    {
      double result_integral_source;
      MPI_Recv(&result_integral_source, 1, MPI_DOUBLE, source, message_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      result_integral += result_integral_source;
    }
    printf("with n = %d trapezoids, our estimate\n", N);
    printf("of the integral from %f to %f = %.15e\n", A, B, result_integral);
  }

  MPI_Finalize();

  return 0;
}

double trap(double a, double b, long int n)
{

  double h = (b - a) / n;

  double approx = (function(a) + function(b)) / 2.0;
  for (int i = 1; i < n - 1; i++)
  {
    // printf("f_%i \n", i);
    double x_i = a + i * h;

    approx += function(x_i);
```

```
  }

  return h * approx;
}

double function(double x) { return x * x; }
```

compilação e execução

```
mpicc trap_rule.c -o trap
# a =0, b =3, n = 1024
mpiexec -n 4 ./trap 0 3 1024
```

Alteração no código

Sabendo que pode ocorrer divisões cujo o número não pertence aos números racionais, por tanto se faz necessário utilizar a seguinte expressão:

$$\frac{N}{D} = a \times D + \text{resto}$$

exemplo:

$$\frac{1024}{3} = 341 \times 3 + 1$$

Podemos resolver o problema simplesmente despejando os trapézios restantes em um processo ou podemos dividir a "carga" entre os processos. Dividir a carga entre os processos de forma mais igualitária possível, foi resolvido no exercício 1 capítulo 1.

```
struct range
{
    int first;
    int last;
};

struct range new_range_2(int thread_index, int p, int n)
{
    struct range r;

    int division = n / p;
    int rest = n % p;

    if (thread_index < rest)
    {
        r.first = thread_index * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = thread_index * division + rest;
        r.last = r.first + division;
    }

    return r;
}
...

int main(int argc, char *argv[])
{

    ...
    /*
    onde thread_index é o rank,
    o número de cores é o número de processos,
    o tamanho do vetor é o número de trapézios
    */
    struct range r = new_range_2(my_rank,comm_sz,N);

    double h = (B - A) / N;

    /*
    perceba que o número local de trapézios é
    o tamanho do intervalo calculado
     */
    int local_n = r.last -r.first;
    double local_a = A + r.first * h;
    double local_b = A + r.last * h;
    double result_integral = trap(local_a, local_b, local_n);

    printf("local n: %i local a: %f local b %f \n", local_n, local_a, local_b);
}
```

Simulações feitas offline:

```
mpicc trap_rule.c -o trap; mpiexec -n 3 ./trap 1 3 1024
local n: 342 local a: 1.000000 local b 1.667969
local n: 341 local a: 1.667969 local b 2.333984
local n: 341 local a: 2.333984 local b 3.000000
with n = 1024 trapezoids, our estimate
of the integral from 1.000000 to 3.000000 = 8.633069768548012e+00
```

```
mpicc trap_rule.c -o trap; mpiexec -n 4 ./trap 1 3 2024
local n: 506 local a: 1.000000 local b 1.500000
local n: 506 local a: 1.500000 local b 2.000000
local n: 506 local a: 2.000000 local b 2.500000
local n: 506 local a: 2.500000 local b 3.000000
with n = 2024 trapezoids, our estimate
of the integral from 1.000000 to 3.000000 = 8.645439504647232e+00
```

```
mpicc trap_rule.c -o trap; mpiexec -n 3 ./trap 0 3 1024
local n: 342 local a: 0.000000 local b 1.001953
local n: 341 local a: 1.001953 local b 2.000977
local n: 341 local a: 2.000977 local b 3.000000
with n = 1024 trapezoids, our estimate
of the integral from 0.000000 to 3.000000 = 8.959068736061454e+00
```

código final

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

struct range
{
  int first;
  int last;
};

struct range new_range_2(int thread_index, int p, int n)
{
  struct range r;

  int division = n / p;
  int rest = n % p;

  if (thread_index < rest)
  {
    r.first = thread_index * (division + 1);
    r.last = r.first + division + 1;
  }
  else
  {
    r.first = thread_index * division + rest;
    r.last = r.first + division;
  }

  return r;
}
``
double trap(double a, double b, long int n);

double function(double x);

int main(int argc, char *argv[])
{

  int my_rank;
  int comm_sz;

  int message_tag = 0;

  MPI_Init(&argc, &argv);

  const double A = atof(argv[1]);
  const double B = atof(argv[2]);
  const int N = atoi(argv[3]);

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

  struct range r = new_range_2(my_rank, comm_sz, N);

  double h = (B - A) / N;

  int local_n = r.last - r.first;
  double local_a = A + r.first * h;
  double local_b = A + r.last * h;
  double result_integral = trap(local_a, local_b, local_n);

  printf("local n: %i local a: %f local b %f \n", local_n, local_a, local_b);
```

```
  if (my_rank != 0)
  {
    MPI_Send(&result_integral, 1, MPI_DOUBLE, 0, message_tag, MPI_COMM_WORLD);
  }
  else // my_rank ==0
  {

    for (int source = 1; source < comm_sz; source++)
    {
      double result_integral_source;
      MPI_Recv(&result_integral_source, 1, MPI_DOUBLE, source, message_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      result_integral += result_integral_source;
    }
    printf("with n = %d trapezoids, our estimate\n", N);
    printf("of the integral from %f to %f = %.15e\n", A, B, result_integral);
  }

  MPI_Finalize();

  return 0;
}

double trap(double a, double b, long int n)
{

  double h = (b - a) / n;
  // printf("H: %f\n",h);

  double approx = (function(a) + function(b)) / 2.0;
  for (int i = 1; i < n - 1; i++)
  {
    // printf("f_%i \n", i);
    double x_i = a + i * h;

    approx += function(x_i);
  }

  return h * approx;
}

double function(double x) { return x * x; }
```

4.3 Question 04

4.3.1 Questão 4

Modify the program that just prints a line of output from each process (*mpi_output.c*) so that the output is printed in process rank order: process 0s output first, then process 1s, and so on.

```c
#include <stdio.h>
#include <mpi.h>
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);
    printf("Proc %d of %d > Does anyone have a toothpick?\n",
           my_rank, comm_sz);

    MPI_Finalize();
    return 0;
}
```

Sabendo que o acesso ao stdout está sendo requisitado por todos os processos e que não existe por padrão um mecanismo que ordene as mensagens. então uma proposta para resolver o problema é justamente definir que a responsabilidade de exibir mensagens no stdout será data ao processo de rank 0 e que todo processo que deseja utilizar o stdout, enviará uma mensagem para o processo de rank 0.

```c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <mpi.h>
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);
    int tag = 0;
    const char template[] = "Proc %d of %d > Does anyone have a toothpick?\n";
    size_t template_size = 47;

    if (my_rank == 0)
    {
        printf(template, my_rank, comm_sz);
        for (size_t i = 1; i < comm_sz; i++)
        {
            char message[template_size];
            MPI_Recv(message, template_size + 1, MPI_CHAR, i, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s", message);
        }
    }
    else
    {
        char message[template_size];
        sprintf(message, template, my_rank, comm_sz);
        MPI_Send(message, template_size + 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

```
mpicc -g -Wall -o questio_4 main.c;mpiexec -n 4 ./questio_4
# saída
Proc 0 of 4 > Does anyone have a toothpick?
Proc 1 of 4 > Does anyone have a toothpick?
Proc 2 of 4 > Does anyone have a toothpick?
Proc 3 of 4 > Does anyone have a toothpick?
```

4.4 Question 06

4.4.1 Questão 6

Suppose $comm\_sz = 4$ and suppose that $\mathbf{x}$ is a vector with $n = 14$ components.

1. How would the components of $\mathbf{x}$ be distributed among the processes in program that used a block distribution ?
2. How would the components of $\mathbf{x}$ be distributed among the processes in a program that used a cyclic distribution ?
3. How would the components of x be distributed among the processes in a program that used a block-cyclic distribution with blocksize $b = 2$?

block distribution

Sabendo que a função que calcula os intervalos da distribuição de blocos é a mesma função do Questão 1 cap 1 :

```
struct range
{
    int first;
    int last;
};

struct range new_range_2(int thread_index, int p, int n)
{
    struct range r;

    int division = n / p;
    int rest = n % p;

    if (thread_index < rest)
    {
        r.first = thread_index * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = thread_index * division + rest;
        r.last = r.first + division;
    }

    return r;
}
```

criando um script em c e o executando temos que:

```
gcc questao_6.c -o questao_6; ./questao_6
Process 0 First  0 Last  4 m Last - First: 4
Process 1 First  4 Last  8 m Last - First: 4
Process 2 First  8 Last  11 m Last - First: 3
Process 3 First  11 Last  14 m Last - First: 3
```

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

cyclic distribution

Sabendo que a cyclic distribution é a transposta da matriz, pois é uma matriz quadrada,então:

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

block-cyclic distribution

Um block-cyclic distribution de blocksize $b = 2$, é um cyclic distribution, onde cada componente é um bloco de tamanho tamanho 2. Sabendo que o tamanho do vetor e 14, logo haverá $\frac{14}{2}$ componentes. então para encontrar a distribuição temos que:

- achar o cyclic distribution de 7 componentes virtuais
- substituir cada um dos 7 componentes virtuais em dois reais

CYCLIC DISTRIBUTION DE 7 COMPONENTES VIRTUAIS

processos		
0	B_1	B_5
1	B_2	B_6
2	B_3	B_7
3	B_4	

SUBSTITUINDO OS BLOCOS.

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

4.5 Question 09

4.5.1 Questão 9

Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by *comm_sz*

question_9.c

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct input
{
    double *v_1;
    double *v_2;
    double scalar;
    int vector_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

double local_dot(double *v_1, double *v_2, int size);
double *local_scalar_multiply(double *v, int size, double scalar);

double get_global_dot(double *local_dot);
double *send_global_vector_to_root(double *local_vector, int local_size, int global_size, int my_rank);
void print_vector(double *vector, int size);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;
    int input_size;

    int message_tag = 0;



    /*
        Decidir pegar os dados de entrada através
        dos argumentos passados pelos programa
        então preciso iniciar o init com argc
        e argv
    */
    MPI_Init(&argc, &argv);

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

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);

    double local_dot_result = local_dot(local_input.v_1, local_input.v_2,
                                        local_input.vector_size);

    double *local_scalar_v1 = local_scalar_multiply(local_input.v_1,
                                                    local_input.vector_size, local_input.scalar);
    double *local_scalar_v2 = local_scalar_multiply(local_input.v_2,
                                                    local_input.vector_size, local_input.scalar);

    double dot_result = get_global_dot(&local_dot_result);

    double *scalar_v1 = send_global_vector_to_root(local_scalar_v1, local_input.vector_size,
                                                   local_input.vector_size * comm_sz, my_rank);
    double *scalar_v2 = send_global_vector_to_root(local_scalar_v2, local_input.vector_size,
                                                   local_input.vector_size * comm_sz, my_rank);

    if (my_rank == 0)
    {
        printf("\t\t Result\n");
        printf("Dot: %lf\n", dot_result);
        printf("scalar v1: ");
```

```
        print_vector(scalar_v1, local_input.vector_size * comm_sz);
        printf("\n");
        printf("scalar v2: ");
        print_vector(scalar_v2, local_input.vector_size * comm_sz);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

double local_dot(double *v_1, double *v_2, int size)
{
    double result = 0;
    for (int i = 0; i < size; i++)
    {
        result += v_1[i] * v_2[i];
    }

    return result;
}

double *local_scalar_multiply(double *v, int size, double scalar)
{
    double *result = malloc(size * sizeof(double));

    for (int i = 0; i < size; i++)
    {
        result[i] = v[i] * scalar;
    }

    return result;
}

double get_global_dot(double *local_dot)
{
    double result_dot;
    MPI_Reduce(local_dot, &result_dot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    return result_dot;
}

double *send_global_vector_to_root(double *local_vector, int local_size, int global_size, int my_rank)
{
    double *global_vector = NULL;

    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
    }
    MPI_Gather(local_vector, local_size, MPI_DOUBLE, global_vector, local_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;

    if (my_rank == 0)
    {
        /*
            como pedido na questão o processo 0
            deve enviar dados de entrada para
            os outros processo então somente
            no processo 0 será lido os argumentos
            de entrada  que são:
            argc == 5 (total de argumentos)
            argv[0] == ./question_9 (argv[0] sempre é o nome do executável )
            argv[1] == 5 (scalar)
            argv[2] == "1,1,1,1" (vetor 1)
            argv[3] ==  "2,2,2,2" (vetor 2)
            argv[4] == 4 (tamanho dos vetores)
        */
        total_input = read_input(argc, argv);

        printf("\t\tInput\n");
        printf("vector size %i\n", total_input.vector_size);
        printf("v_1: ");
        print_vector(total_input.v_1, total_input.vector_size);
        printf("\n");
        printf("v_2: ");
        print_vector(total_input.v_2, total_input.vector_size);
        printf("\n");
        printf("Scalar: %lf\n", total_input.scalar);
    }

    /*
        com MPI_Bcast agora todos os processos tem o mesmo valor das variáveis
        total_input.vector_size e total_input.scalar
    */
    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&total_input.scalar, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);
    /*
         eu uso a função que calcula os indices do vetor para
         calcular o tamanho do vetor por uma questão de aprendizado
         e verifica que dá o mesmo resultado.
         Na prática o tamanho do vetor local é o (tamanho global)/(total de processos)
         (tamanho global)/(total de processos) == total_input.vector_size/comm_sz
    */
    local_input.vector_size = r.last - r.first;
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.v_2 = malloc(local_input.vector_size * sizeof(double));
    local_input.scalar = total_input.scalar;

    /*
        MPI_Scatter divide o vetor total e envia cada parte para os processos
        MPI_Scatter segue o block distribution ou seja, processo 0 fica com
        os pedaço [0:a], processo 1 fica com o pedaço [a:2a]...
        e assim por diante, sendo a = total_input.vector_size/comm_sz
    */
    MPI_Scatter(total_input.v_1, local_input.vector_size, MPI_DOUBLE, local_input.v_1, local_input.vector_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_input.v_2, local_input.vector_size, MPI_DOUBLE, local_input.v_2, local_input.vector_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return local_input;
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }

    return r;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

input read_input(int argc, char *argv[])
{

    if (argc != 5)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.scalar = atoi(argv[1]);

    result.vector_size = atoi(argv[4]);

    result.v_1 = read_vector(argv[2], result.vector_size);
    result.v_2 = read_vector(argv[3], result.vector_size);
```

```
    return result;
}
```

teste

```
mpirun -n 4 ./question_9 5 "1,1,1,1" "2,2,2,2" 4
               Input
vector size 4
v_1: [1.000000,1.000000,1.000000,1.000000]
v_2: [2.000000,2.000000,2.000000,2.000000]
Scalar: 5.000000
                Result
Dot: 8.000000
scalar v1: [5.000000,5.000000,5.000000,5.000000]
scalar v2: [10.000000,10.000000,10.000000,10.000000]
```

4.6 Question 11

4.6.1 Questão 11

Finding **prefix sums** is a generalization of global sum. Rather than simply finding the sum of n values,

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

the prefix sums are the n partial sums

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

1. Devise a serial algorithm for computing the n prefix sums of an array with n elements. 2. Parallelize your serial algorithm for a system with n processes, each of which is storing one of the x_is. 3. Suppose $n = 2^k$ for some positive integer k. Can you devise a serial algorithm and a parallelization of the serial algorithm so that the parallel algorithm requires only k communication phases ? 4. MPI provides a collective communication function, MPI Scan, that can be used to compute prefix sums:

```
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  */);
```

It operates on arrays with *count* elements; both *sendbuf_p* and *recvbuf_p* should refer to blocks of *count* elements of type *datatype*. The *op* argument is the same as op for *MPI_Reduce*. Write an MPI program that generates a random array of *count* elements on each MPI process, finds the prefix sums, and print the results.

Prefix sums serial

question_11_serial.cpp

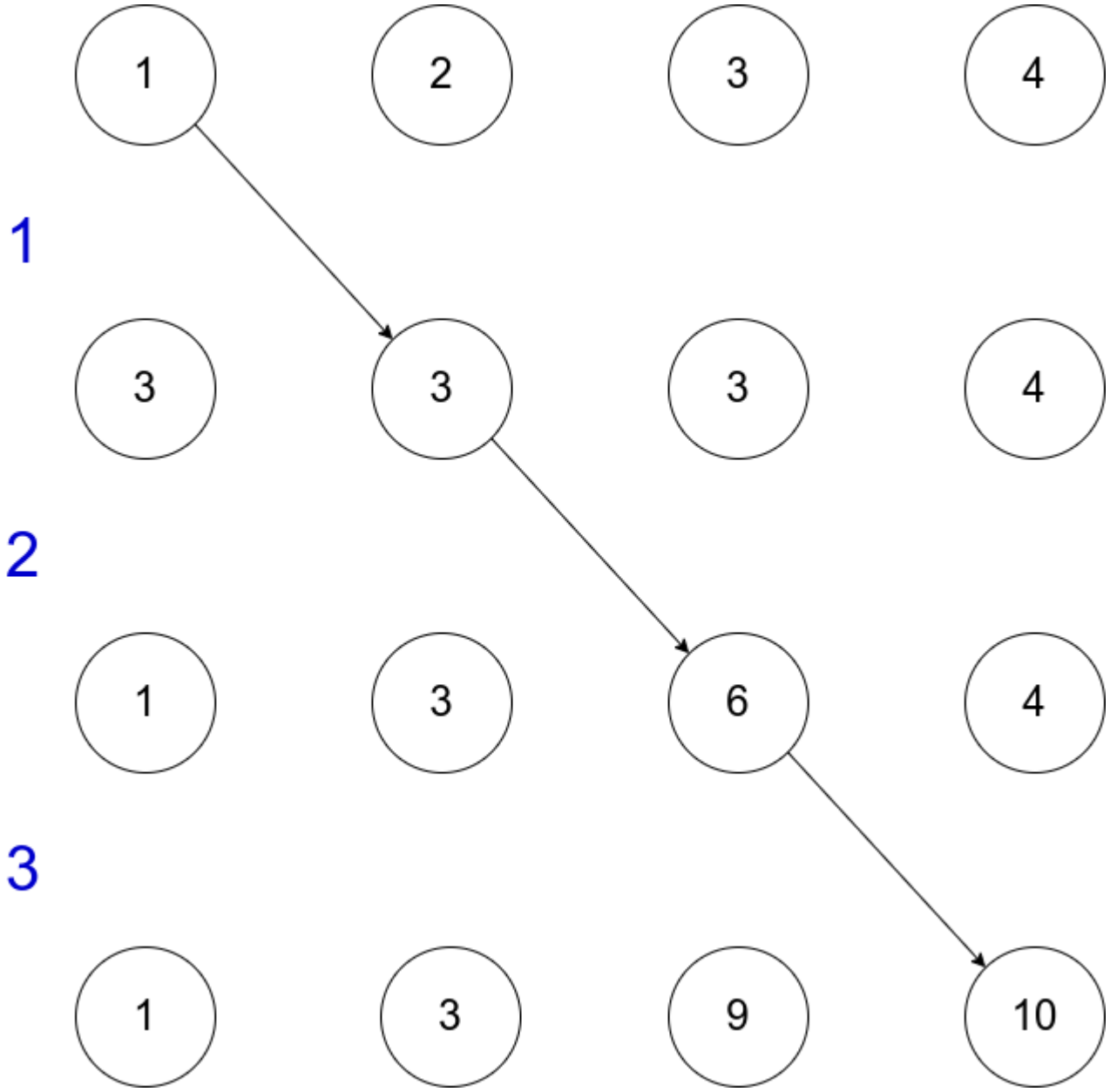
```cpp
#include <iostream>
#include <vector>
#include <assert.h>

std::vector<int> calculate_n_prefix_sums(std::vector<int> &v, int &n);
int prefix_sum(std::vector<int> &v, int &n);
void test_prefix_sums();

int main(int argc, char const *argv[])
{
    test_prefix_sums();

    return 0;
}

std::vector<int> calculate_n_prefix_sums(std::vector<int> &v, int &n)
{
    assert(n >= 1);

    std::vector<int> result(n);
    result[0] = v[0];

    for (int i = 1; i < n; i++)
        result[i] = result[i - 1] + v[i];

    return result;
}

void test_prefix_sums()
{
    std::vector<int> v{1, 2, 3, 4, 5, 6};
    auto n = 1;
    auto prefixs = calculate_n_prefix_sums(v, n);

    assert(prefixs.size() == 1);
    assert(prefixs[0] == v[0]);

    n = 2;
    prefixs = calculate_n_prefix_sums(v, n);
```

```
    assert(prefixs.size() == 2);
    assert(prefixs[0] == v[0]);
    assert(prefixs[1] == v[0] + v[1]);

    n = 3;
    prefixs = calculate_n_prefix_sums(v, n);

    assert(prefixs.size() == 3);
    assert(prefixs[0] == v[0]);
    assert(prefixs[1] == v[0] + v[1]);
    assert(prefixs[2] == v[0] + v[1] + v[2]);

    std::cout << "Passed in tests" << std::endl;
}
```

```
c++ question_11_serial.cpp -o question_11_serial
./question_11_serial
Passed in tests
```

Prefix sums MPI

A figura a baixo, é um exemplo onde a entrada é é um vetor $v = [1, 2, 3, 4]$ e o número de processos é igual a 4. Logo cada processo terá apenas um número. A ideia é focar só na transferência dos valores entre os processos. Perceba que o processo raiz apenas envia o seu valor para o próximo nó e que o último processo apenas recebe os valor do nó anterior e que o número de sincronizações é $p - 1$, onde p é um números de processos, no exemplo $p = 4$, por tanto o número de sincronizações é 3.



question_11_mpi.c

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct input
{
    double *v_1;
    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

void print_vector(double *vector, int size);

double *send_global_vector_to_root(double *local_vector, int local_size,
```

```
                                int global_size, int my_rank);
double *calculate_n_prefix_sums(double v[], int size);

void update_prefix_sums_vector(double vector[], int size, double value);
double receive_last_prefix_sum(int source_node, int message_tag);
void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;

    int message_tag = 0;

    MPI_Init(&argc, &argv);

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

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);
    double *local_prefix_sum = calculate_n_prefix_sums(local_input.v_1,
                                                       local_input.vector_size);

    if (my_rank != 0) // is not root process
    {
        double sum_source_node = receive_last_prefix_sum(my_rank - 1,
                                                         message_tag);

        update_prefix_sums_vector(local_prefix_sum,
                                  local_input.vector_size, sum_source_node);
    }

    if (my_rank != comm_sz - 1) // is not last process
    {
        double last_prefix_sum = local_prefix_sum[local_input.vector_size - 1];
        MPI_Send(&last_prefix_sum, 1, MPI_DOUBLE,
                 my_rank + 1, message_tag, MPI_COMM_WORLD);
    }

    double *global_prefix_sum = send_global_vector_to_root(local_prefix_sum,
                                                           local_input.vector_size,
                                                           local_input.input_size,
                                                           my_rank);
    if (my_rank == 0)
    {
        printf("\t\tResult\n");
        printf("Prefix sum: ");
        print_vector(global_prefix_sum, local_input.input_size);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank)
{
    double *global_vector = NULL;

    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
    }
    MPI_Gather(local_vector, local_size, MPI_DOUBLE, global_vector,
               local_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;

    total_input.v_1 = NULL;

    if (my_rank == 0)
    {
        total_input = read_input(argc, argv);

        printf("\t\tInput\n");
        printf("vector size %i\n", total_input.vector_size);
        printf("vector input: ");
        print_vector(total_input.v_1, total_input.vector_size);
        printf("\n");
    }

    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);
```

```
    local_input.vector_size = r.last - r.first;
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.input_size = total_input.vector_size;

    MPI_Scatter(total_input.v_1, local_input.vector_size, MPI_DOUBLE,
                local_input.v_1, local_input.vector_size, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    return local_input;
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }

    return r;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

input read_input(int argc, char *argv[])
{

    if (argc != 3)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.vector_size = atoi(argv[1]);
    result.v_1 = read_vector(argv[2], result.vector_size);

    return result;
}

void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag)
{
    int source_node = my_rank - 1;
    double last_prefix_sum_source_node;
    MPI_Recv(&last_prefix_sum_source_node,
             1,
             MPI_DOUBLE,
             source_node,
             message_tag,
             MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}

double receive_last_prefix_sum(int source_node, int message_tag)
```

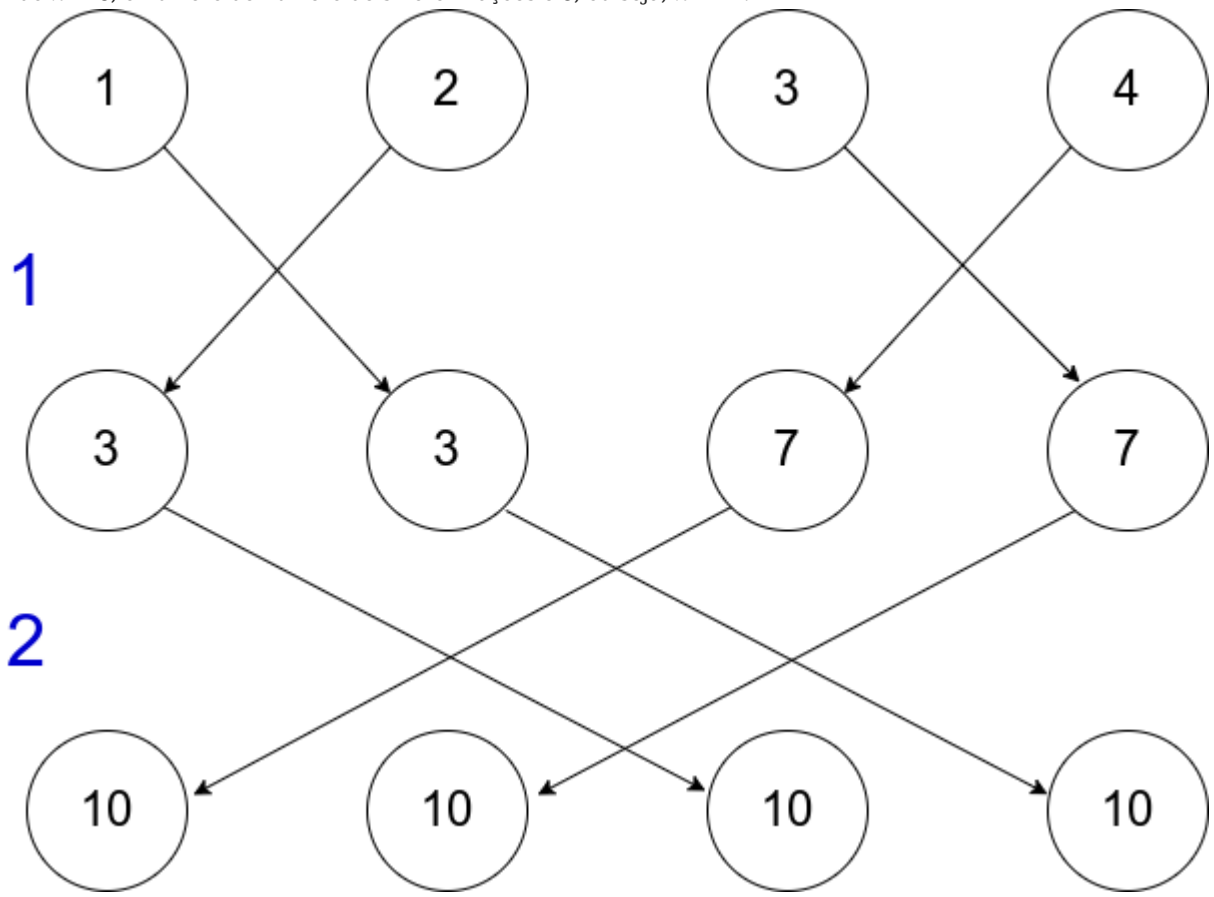
```
{
    double last_prefix_sum_source_node;
    MPI_Recv(&last_prefix_sum_source_node, 1, MPI_DOUBLE,
             source_node, message_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    return last_prefix_sum_source_node;
}

double *calculate_n_prefix_sums(double v[], int size)
{

    double *result = malloc(size * sizeof(double));

    result[0] = v[0];

    for (int i = 1; i < size; i++)
        result[i] = result[i - 1] + v[i];

    return result;
}

void update_prefix_sums_vector(double vector[], int size, double value)
{
    for (int i = 0; i < size; i++)
        vector[i] += value;
}
```

```
mpicc question_11_mpi.c -o question_11_mpi
mpiexec -n 4 ./question_11_mpi 8 1,2,3,4,5,6,7,8

                Input
vector size 8
vector input: [1.000000,2.000000,3.000000,4.000000,5.000000,6.000000,7.000000,8.000000]
                Result
Prefix sum: [1.000000,3.000000,6.000000,10.000000,15.000000,21.000000,28.000000,36.000000]
```

Prefix sums MPI butterfly structured

Nó caso para reduzirmos o número de sincronizações, temos que utilizar a estrutura butterfly, idealmente nós não deveríamos implementar a estrutura, uma vez que **implementar** ela é de forma ótima é responsabilidade do próprio MPI. Deveríamos apenas utilizar e passar os parâmetros adequados. - "Alternatively, we might have the processes exchange partial results instead of using one-way communications. Such a communication pattern is sometimes called a butterfly (see Figure 3.9). Once again, we don’t want to have to decide on which structure to use, or how to code it for optimal performance." Perceba que utilizando a estrutura butterfly atente os requisitos. $n = 4$, implica o número de sincronizações é 2 e caso tenha número de cores suficiente,

quando $n = 8$, o número de número de sincronizações é 3, ou seja, $n = 2^k$.



question_11_butterfly.c

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct input
{
    double *v_1;
    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

void print_vector(double *vector, int size);

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank);
double *calculate_n_prefix_sums(double v[], int size);

void update_prefix_sums_vector(double vector[], int size, double value);

void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;

    int message_tag = 0;

    int sync_step = 1;
```

```
    int n = 1;

    MPI_Init(&argc, &argv);

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

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);
    double *local_prefix_sum = calculate_n_prefix_sums(local_input.v_1, local_input.vector_size);

    /*
        butterfly algorithm
    */
    double last_sum = local_prefix_sum[local_input.vector_size - 1];
    for (int k = 0; (1 << k) < comm_sz; k++)
    // k varia entre  0,1,2,4,8,16 ... até ser maior que comm_sz
    {
        int partner = my_rank ^ (1 << k); // por algum motivo isso funciona.
        double last_sum_partner;

        MPI_Sendrecv(&last_sum, 1, MPI_DOUBLE, partner, message_tag,
                     &last_sum_partner, 1, MPI_DOUBLE, partner, message_tag,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        /*
          o valores do  processo p é atualizado pelos valores do processo p -1
          mas os valores do processo p -1 não é atualizado com os valores do processo p.
          Exemplo:
                processo 1 atualiza o seu vetor local com o último valor do processo 0, mas
                o processo 0 não atualiza seu vetor local com o último valor do processo 1.
        */
        if (my_rank > partner)
            update_prefix_sums_vector(local_prefix_sum,
                                      local_input.vector_size, last_sum_partner);

        /*
            mesmo que processo 0 não atualiza seu vetor local com o último valor do processo 1.
            o processo 0 deve enviar o seu valor + o do processo 1 para o processo 2.
        */
        last_sum += last_sum_partner;
    }

    double *global_prefix_sum = send_global_vector_to_root(local_prefix_sum, local_input.vector_size,
                                                           local_input.input_size, my_rank);
    if (my_rank == 0)
    {
        printf("\t\tResult\n");
        printf("Prefix sum: ");
        print_vector(global_prefix_sum, local_input.input_size);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank)
{
    double *global_vector = NULL;

    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
    }
    MPI_Gather(local_vector, local_size, MPI_DOUBLE,
               global_vector, local_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;

    total_input.v_1 = NULL;

    if (my_rank == 0)
    {
        total_input = read_input(argc, argv);

        printf("\t\tInput\n");
        printf("vector size %i\n", total_input.vector_size);
        printf("vector input: ");
        print_vector(total_input.v_1, total_input.vector_size);
        printf("\n");
    }

    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);

    local_input.vector_size = r.last - r.first;
```

```
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.input_size = total_input.vector_size;

    MPI_Scatter(total_input.v_1, local_input.vector_size, MPI_DOUBLE,
                local_input.v_1, local_input.vector_size, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    return local_input;
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }

    return r;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

input read_input(int argc, char *argv[])
{

    if (argc != 3)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.vector_size = atoi(argv[1]);
    result.v_1 = read_vector(argv[2], result.vector_size);

    return result;
}

void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag)
{
    int source_node = my_rank - 1;
    double last_prefix_sum_source_node;
    MPI_Recv(&last_prefix_sum_source_node, 1, MPI_DOUBLE,
             source_node, message_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

double *calculate_n_prefix_sums(double v[], int size)
{

    double *result = malloc(size * sizeof(double));

    result[0] = v[0];
```

```
    for (int i = 1; i < size; i++)
        result[i] = result[i - 1] + v[i];

    return result;
}

void update_prefix_sums_vector(double vector[], int size, double value)
{
    for (int i = 0; i < size; i++)
        vector[i] += value;
}
```

```
mpicc question_11_butterfly.c -o question_11_butterfly
mpiexec -n 4 ./question_11_butterfly 8 1,2,3,4,5,6,7,8

               Input
vector size 8
vector input: [1.000000,2.000000,3.000000,4.000000,5.000000,6.000000,7.000000,8.000000]
               Result
Prefix sum: [1.000000,3.000000,6.000000,10.000000,15.000000,21.000000,28.000000,36.000000]
```

Prefix sum SCAN function

question_11_scan.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct input
{
    double *v_1;
    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

void print_vector(double *vector, int size);

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank);
double *calculate_n_prefix_sums(double v[], int size);

void update_prefix_sums_vector(double vector[], int size, double value);

void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;

    int message_tag = 0;

    int sync_step = 1;
    int n = 1;

    MPI_Init(&argc, &argv);

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

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);
    double *local_prefix_sum = calculate_n_prefix_sums(local_input.v_1,
                                                       local_input.vector_size);

    double last_sum = local_prefix_sum[local_input.vector_size - 1];
    /*
        https://stackoverflow.com/questions/10801154/gathering-results-of-mpi-scan

         MPI_Scan é semelhante ao Reduce , só que a cada soma parcial, ele devolve o
         seu valor para todos os processos.
         considerando a entrada: [1,2,3,4,5,6,7,8] para 4 processos:

                                           p0   p1     p2    p3
         divisão da entrada entre processo: [1,2] [3,4]  [5,6] [7,8]
         o MPI_Scan será:
```

```
         sendbuf:  3  ,  7 , 11 ,  15, a suas somas parciais serão:
         recvbuf:  3  , 3+7 , 3+7+11, 3+7+15

    */
    double partial_sum_scan_buf;
    MPI_Scan(&last_sum, &partial_sum_scan_buf, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

    // para eliminar a o last sum somado duas vezes.
    double value = partial_sum_scan_buf - last_sum;
    update_prefix_sums_vector(local_prefix_sum, local_input.vector_size, value);

    double *global_prefix_sum = send_global_vector_to_root(local_prefix_sum,
                                                           local_input.vector_size,
                                                           local_input.input_size,
                                                           my_rank);
    if (my_rank == 0)
    {
        printf("\t\tResult\n");
        printf("Prefix sum: ");
        print_vector(global_prefix_sum, local_input.input_size);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank)
{
    double *global_vector = NULL;

    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
    }
    MPI_Gather(local_vector, local_size, MPI_DOUBLE, global_vector,
               local_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;

    total_input.v_1 = NULL;

    if (my_rank == 0)
    {
        total_input = read_input(argc, argv);

        printf("\t\tInput\n");
        printf("vector size %i\n", total_input.vector_size);
        printf("vector input: ");
        print_vector(total_input.v_1, total_input.vector_size);
        printf("\n");
    }

    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);

    local_input.vector_size = r.last - r.first;
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.input_size = total_input.vector_size;

    MPI_Scatter(total_input.v_1, local_input.vector_size, MPI_DOUBLE,
                local_input.v_1, local_input.vector_size, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    return local_input;
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }

    return r;
```

```
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

input read_input(int argc, char *argv[])
{

    if (argc != 3)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.vector_size = atoi(argv[1]);
    result.v_1 = read_vector(argv[2], result.vector_size);

    return result;
}

void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag)
{
    int source_node = my_rank - 1;
    double last_prefix_sum_source_node;
    MPI_Recv(&last_prefix_sum_source_node, 1, MPI_DOUBLE,
             source_node, message_tag, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}

double *calculate_n_prefix_sums(double v[], int size)
{

    double *result = malloc(size * sizeof(double));

    result[0] = v[0];

    for (int i = 1; i < size; i++)
        result[i] = result[i - 1] + v[i];

    return result;
}

void update_prefix_sums_vector(double vector[], int size, double value)
{
    for (int i = 0; i < size; i++)
        vector[i] += value;
}
```

```
mpicc question_11_scan.c -o question_11_scan
mpiexec -n 4 ./question_11_scan 8 1,2,3,4,5,6,7,8
                Input
vector size 8
vector input: [1.000000,2.000000,3.000000,4.000000,5.000000,6.000000,7.000000,8.000000]
                Result
Prefix sum: [1.000000,3.000000,6.000000,10.000000,15.000000,21.000000,28.000000,36.000000]
```

4.7 Question 12

4.7.1 Questão 12

An alternative to a butterfly-structured allreduce is a **ring-pass** structure. In a ring-pass, if there are p processes, each process q sends data to process $q+1$, except that process $p-1$ sends data to process 0. This is repeated until each process has the desired result. Thus, we can implement allreduce with the following code:

```
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;
}
```

1. Write an MPI program that implements this algorithm for allreduce. How does its performance compare to the butterfly-structured allreduce ?
2. Modify the MPI program you wrote in the first part so that it implements prefix sums.

benchmark butterfly vs ring pass

CÓDIGO

bench.c

```
/*
mpicc  bench.c -o bench && mpirun  -n 4 ./bench 8 1,2,3,4,5,6,7,8  && rm bench
*/
#define _GNU_SOURCE

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
typedef struct input
{
    double *v_1;
    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

void print_vector(double *vector, int size);

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank);

double sum_elements(double *vector, int size);

double global_sum_ring_pass(double local_sum, int my_rank,
                            int comm_sz, int message_tag);

double global_sum_butterfly(double local_sum, int my_rank,
                            int comm_sz, int message_tag);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;

    int message_tag = 0;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);

    double sum = sum_elements(local_input.v_1, local_input.vector_size);

    double ring_pass_start, ring_pass_end;

    MPI_Barrier(MPI_COMM_WORLD);
    ring_pass_start = MPI_Wtime();
    double sum_ring_pass = global_sum_ring_pass(sum, my_rank, comm_sz, message_tag);
    ring_pass_end = MPI_Wtime();

    // Blocks until all processes in the communicator have reached this routine.
    MPI_Barrier(MPI_COMM_WORLD); // https://www.mpich.org/static/docs/v3.2/www3/MPI_Barrier.html

    double butterfly_start, butterfly_end;

    // returns Time in seconds since an arbitrary time in the past.
    butterfly_start = MPI_Wtime(); // https://www.mpich.org/static/docs/v3.2/www3/MPI_Wtime.html
    double sum_butterfly = global_sum_butterfly(sum, my_rank, comm_sz, message_tag);
    butterfly_end = MPI_Wtime();

    double ring_pass_time = ring_pass_end - ring_pass_start;
    double butterfly_time = butterfly_end - butterfly_start;

    double *vector_input = send_global_vector_to_root(local_input.v_1, local_input.vector_size,
                                                      local_input.input_size, my_rank);

    double *ring_pass_time_vector = send_global_vector_to_root(&ring_pass_time,
                                                               1, comm_sz, my_rank);

    double *butterfly_time_vector = send_global_vector_to_root(&butterfly_time,
                                                               1, comm_sz, my_rank);

    if (my_rank == 0)
    {
        double mean_time_butterfly = sum_elements(butterfly_time_vector, comm_sz) / comm_sz;
        double mean_time_ring_pass = sum_elements(ring_pass_time_vector, comm_sz) / comm_sz;

        // printf("\t\tInput\n");
        // printf("vector size %i\n", local_input.input_size);
        // printf("vector input: ");
        // print_vector(vector_input, local_input.input_size);
        // printf("\n");

        // printf("\t\tResult\n");

        // printf("sum-ring-pass: %lf sum-butterfly: %lf\n", sum_ring_pass, sum_butterfly);
        // printf("#butterfly-time vector: ");
        // print_vector(butterfly_time_vector, comm_sz);
        // printf("\n");

        // printf("#ring-pass-time vector: ");
        // print_vector(ring_pass_time_vector, comm_sz);
        // printf("\n");

        //     algorithm   time-in-nsec   number-process

        printf("butterfly-time %lf %i\n", mean_time_butterfly, comm_sz);
        printf("ring-pass-time %lf %i\n", mean_time_ring_pass, comm_sz);
    }

    MPI_Finalize();

    return 0;
}

double sum_elements(double *vector, int size)
{
    double sum = 0;
    for (int i = 0; i < size; i++)
    {
        sum += vector[i];
    }
    return sum;
}

double global_sum_ring_pass(double local_sum, int my_rank, int comm_sz, int message_tag)
{
    /*  seja p, o número de processos, e q um rank de um processo.

       processo q enviar dado para q + 1, caso  q + 1 > p então q envia para 0

       analogamente, processo q recebe dado q -1,caso q == 0 então q recebe p-1
   */
    int last_process = comm_sz - 1;

    int dest = my_rank == last_process ? 0 : my_rank + 1;

    int source = my_rank == 0 ? last_process : my_rank - 1;

    double global_sum = 0;
    double value = local_sum;
```

```
    for (int phase = 0; phase < comm_sz; phase++)
    {
        MPI_Sendrecv_replace(&value, 1, MPI_DOUBLE, dest, message_tag,
                             source, message_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        global_sum += value;
    }

    return global_sum;
}

double global_sum_butterfly(double local_sum, int my_rank,
                            int comm_sz, int message_tag)
{

    double global_sum = local_sum;
    for (int k = 0; (1 << k) < comm_sz; k++)
    // k varia entre  0,1,2,4,8,16 ... até ser maior que comm_sz
    {
        int partner = my_rank ^ (1 << k); // por algum motivo isso funciona.
        double last_sum_partner;

        MPI_Sendrecv(&global_sum, 1, MPI_DOUBLE, partner, message_tag,
                     &last_sum_partner, 1, MPI_DOUBLE, partner, message_tag,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        /*
            mesmo que processo 0 não atualiza seu vetor local com o último valor do processo 1.
            o processo 0 deve enviar o seu valor + o do processo 1 para o processo 2.
        */
        global_sum += last_sum_partner;
    }

    return global_sum;
}

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank)
{
    double *global_vector = NULL;

    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
    }
    MPI_Gather(local_vector, local_size, MPI_DOUBLE, global_vector,
               local_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;

    total_input.v_1 = NULL;

    if (my_rank == 0)
    {
        total_input = read_input(argc, argv);
    }

    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);

    local_input.vector_size = r.last - r.first;
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.input_size = total_input.vector_size;

    MPI_Scatter(total_input.v_1, local_input.vector_size, MPI_DOUBLE,
                local_input.v_1, local_input.vector_size, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    return local_input;
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }
```

```
    return r;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

input read_input(int argc, char *argv[])
{

    if (argc != 3)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.vector_size = atoi(argv[1]);
    result.v_1 = read_vector(argv[2], result.vector_size);

    return result;
}
```

run_bench.sh

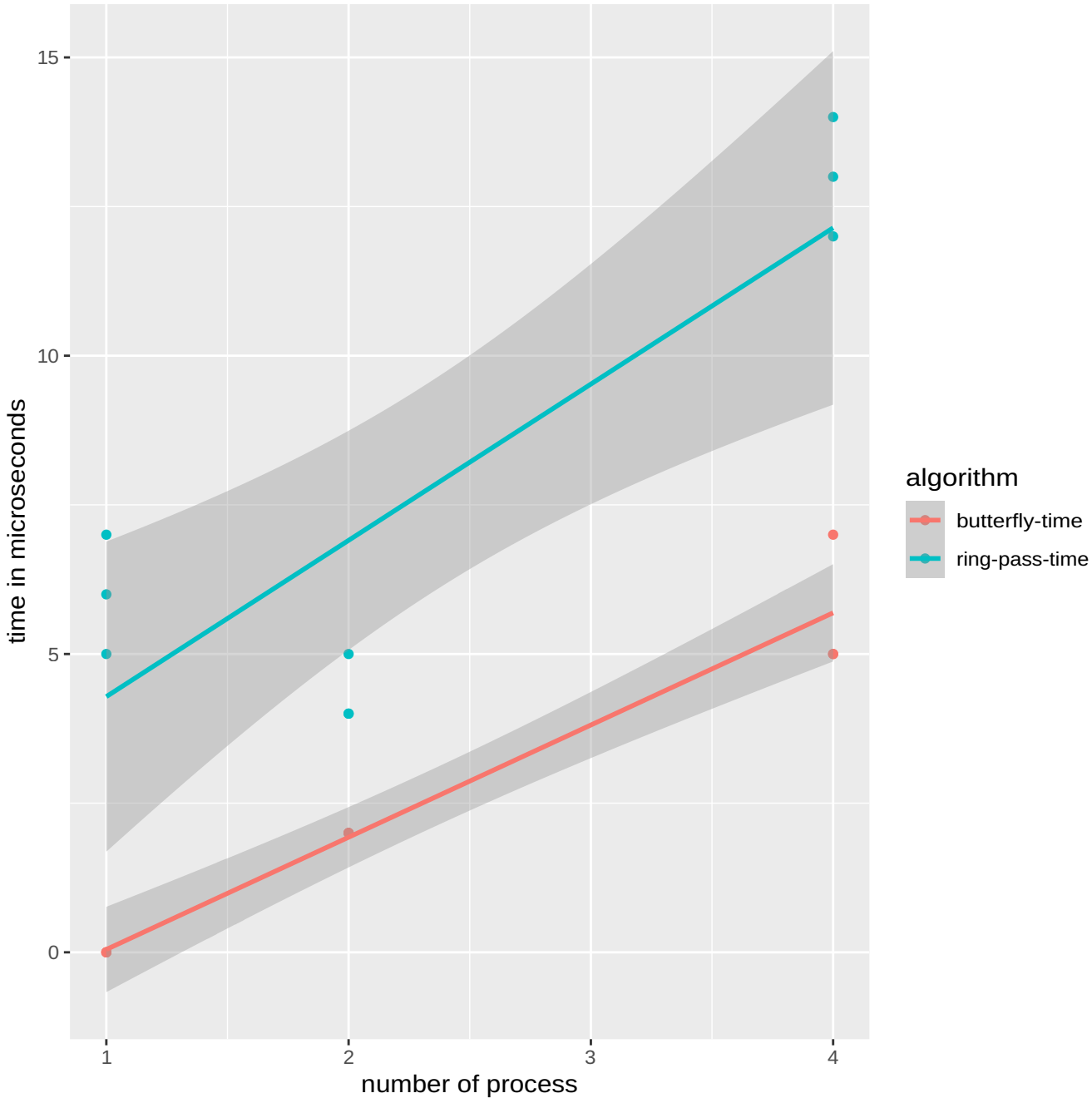
```
#!/bin/bash



A="-n 4 ./bench 12 1,2,3,4,5,6,7,8,9,0,1,2"
B="-n 4 ./bench 8 1,2,3,4,5,6,7,8"
C="-n 4 ./bench 4 1,2,3,4"
# D="-n 3 ./bench 9 1,2,3,4,5,6,7,8,9"
# E="-n 3 ./bench 6 1,2,3,4,5,6"
# F="-n 3 ./bench 3 1,2,3"
G="-n 2 ./bench 8 1,2,3,4,5,6,7,8"
H="-n 2 ./bench 4 1,2,3,4"
I="-n 2 ./bench 2 1,2"
J="-n 1 ./bench 8 1,2,3,4,5,6,7,8"
K="-n 1 ./bench 4 1,2,3,4"
L="-n 1 ./bench 2 1,2"

inputs_mpirun=("$A" "$B" "$C" "$G" "$H" "$I" "$J" "$K" "$L")


echo "algorithm time process" > data.dat
for input_mpi_run in "$A" "$B" "$C" "$G" "$H" "$I" "$J" "$K" "$L";
do
    mpicc  bench.c -o bench
    mpirun $input_mpi_run >> data.dat
done

# creating graphs
./plot_data.r


# cleaning
```

```
rm bench
rm Rplots.pdf
```

plot_data.r

```
#!/bin/Rscript

#algorithm time process
t = read.table('data.dat',header=TRUE)

t["time"] = t["time"] *1e6

library("ggplot2")
plot <- ggplot(t, aes(process,time,colour=algorithm)) +ylab("time in microseconds") + xlab("number of process") + geom_point() + geom_smooth(method=lm)
ggsave("butterfly_vs_ring_pass.svg")
```

GRÁFICOS



Ring pass prefix sums

ring_pass_prefix_sums.c

```c
/*
mpicc  ring_pass_prefix_sums.c -o ring_pass_prefix_sums && mpirun  -n 4 ./ring_pass_prefix_sums 8 1,2,3,4,5,6,7,8  && rm ring_pass_prefix_sums
*/
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct input
{
    double *v_1;
    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

void print_vector(double *vector, int size);

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank);
double *calculate_n_prefix_sums(double v[], int size);

void update_prefix_sums_vector(double vector[], int size, double value);
double receive_last_prefix_sum(int source_node, int message_tag);
void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;

    int message_tag = 0;

    MPI_Init(&argc, &argv);

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

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);
    double *local_prefix_sum = calculate_n_prefix_sums(local_input.v_1,
                                                       local_input.vector_size);

    /*  seja p, o número de processos, e q um rank de um processo.

     processo q enviar dado para q + 1, caso  q + 1 > p então q envia para 0

     analogamente, processo q recebe dado q -1,caso q == 0 então q recebe p-1
 */
    int last_process = comm_sz - 1;

    int dest = my_rank == last_process ? 0 : my_rank + 1;

    int source = my_rank == 0 ? last_process : my_rank - 1;

    double value;

    for (int phase = 1; phase < comm_sz; phase++)
    {
        if (phase != dest)
            value = 0;
        else
            value = local_prefix_sum[local_input.vector_size - 1];

        /*  [1,2] ,[3,4],[5,6], [7,8]
             [3]    [7]   [11]  [15]
            phase 1:
              p_0  3+0   -> 0
              p_1  7+3   -> 10
              p_2  11+0  -> 11
              p_3  15+0  -> 15
            phase 2:
              p_0  3+0   -> 0
              p_1  10+0  -> 10
              p_2  11+10 -> 21
              p_3  15+0  -> 15
            phase 3:
```

```
            p_0  3+0   -> 0
            p_1  10+0  -> 10
            p_2  21+0  -> 21
            p_3  15+21 -> 36
        */

        MPI_Sendrecv_replace(&value, 1, MPI_DOUBLE, dest, message_tag,
                             source, message_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        update_prefix_sums_vector(local_prefix_sum, local_input.vector_size, value);
    }

    double *global_prefix_sum = send_global_vector_to_root(local_prefix_sum,
                                                           local_input.vector_size,
                                                           local_input.input_size,
                                                           my_rank);

    double *vector_input = send_global_vector_to_root(local_input.v_1, local_input.vector_size,
                                                      local_input.input_size, my_rank);

    if (my_rank == 0)
    {
        printf("\t\tInput\n");
        printf("vector size %i\n", local_input.input_size);
        printf("vector input: ");
        print_vector(vector_input, local_input.input_size);
        printf("\n");

        printf("\t\tResult\n");
        printf("Prefix sum: ");
        print_vector(global_prefix_sum, local_input.input_size);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

double *send_global_vector_to_root(double *local_vector, int local_size,
                                   int global_size, int my_rank)
{
    double *global_vector = NULL;

    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
    }
    MPI_Gather(local_vector, local_size, MPI_DOUBLE, global_vector,
               local_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;

    total_input.v_1 = NULL;

    if (my_rank == 0)
    {
        total_input = read_input(argc, argv);
    }

    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);

    local_input.vector_size = r.last - r.first;
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.input_size = total_input.vector_size;

    MPI_Scatter(total_input.v_1, local_input.vector_size, MPI_DOUBLE,
                local_input.v_1, local_input.vector_size, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    return local_input;
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
```

```
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }

    return r;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

input read_input(int argc, char *argv[])
{

    if (argc != 3)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.vector_size = atoi(argv[1]);
    result.v_1 = read_vector(argv[2], result.vector_size);

    return result;
}

void update_prefix_sums(int my_rank, double *local_prefix_sum,
                        int vector_size, int message_tag)
{
    int source_node = my_rank - 1;
    double last_prefix_sum_source_node;
    MPI_Recv(&last_prefix_sum_source_node,
             1,
             MPI_DOUBLE,
             source_node,
             message_tag,
             MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}

double receive_last_prefix_sum(int source_node, int message_tag)
{
    double last_prefix_sum_source_node;
    MPI_Recv(&last_prefix_sum_source_node, 1, MPI_DOUBLE,
             source_node, message_tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    return last_prefix_sum_source_node;
}

double *calculate_n_prefix_sums(double v[], int size)
{

    double *result = malloc(size * sizeof(double));

    result[0] = v[0];

    for (int i = 1; i < size; i++)
        result[i] = result[i - 1] + v[i];

    return result;
}

void update_prefix_sums_vector(double vector[], int size, double value)
{
    for (int i = 0; i < size; i++)
```

```
        vector[i] += value;
}
```

```
mpicc ring_pass_prefix_sums.c -o ring_pass_prefix_sums && mpirun  -n 4 ./ring_pass_prefix_sums 8 1,2,3,4,5,6,7,8  && rm ring_pass_prefix_sums

                Input
vector size 8
vector input: [1.000000,2.000000,3.000000,4.000000,5.000000,6.000000,7.000000,8.000000]
                Result
Prefix sum: [1.000000,3.000000,6.000000,10.000000,15.000000,21.000000,28.000000,36.000000]
```

```
mpicc ring_pass_prefix_sums.c -o ring_pass_prefix_sums && mpirun  -n 2 ./ring_pass_prefix_sums 8 1,2,3,4,5,6,7,8  && rm ring_pass_prefix_sums

                Input
vector size 8
vector input: [1.000000,2.000000,3.000000,4.000000,5.000000,6.000000,7.000000,8.000000]
                Result
Prefix sum: [1.000000,3.000000,6.000000,10.000000,15.000000,21.000000,28.000000,36.000000]
```

4.8 Question 13

4.8.1 Questão 13

MPI Scatter and MPI Gather have the limitation that each process must send or receive the same number of data items. When this is not the case, we must use the MPI functions MPI Gatherv and MPI Scatterv. Look at the man pages for these functions, and modify your vector sum, dot product program so that it can correctly handle the case when n isn't evenly divisible by *comm_sz*.

question_13.c

```c
/*
mpicc question_13.c -o question_13 && mpirun -n 4 ./question_13 5 "1,1,1,1,1" "2,2,2,2,2" 6 && rm question_13
*/
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct input
{
    double *v_1;
    double *v_2;
    double scalar;
    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

double local_dot(double *v_1, double *v_2, int size);
double *local_scalar_multiply(double *v, int size, double scalar);

double get_global_dot(double *local_dot);
double *send_global_vector_to_root(double *local_vector, int local_size, int global_size, int my_rank, int comm_sz);
void print_vector(double *vector, int size);

void generate_counts_and_displacements(int *counts, int *displacements, int input_size, int comm_sz);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;
    int input_size;

    int message_tag = 0;

    MPI_Init(&argc, &argv);

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

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);

    double local_dot_result = local_dot(local_input.v_1, local_input.v_2, local_input.vector_size);

    double *local_scalar_v1 = local_scalar_multiply(local_input.v_1, local_input.vector_size, local_input.scalar);
    double *local_scalar_v2 = local_scalar_multiply(local_input.v_2, local_input.vector_size, local_input.scalar);

    double dot_result = get_global_dot(&local_dot_result);

    double *scalar_v1 = send_global_vector_to_root(local_scalar_v1, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);

    double *scalar_v2 = send_global_vector_to_root(local_scalar_v2, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);

    double *input_v_1 = send_global_vector_to_root(local_input.v_1, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);

    double *input_v_2 = send_global_vector_to_root(local_input.v_2, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);

    if (my_rank == 0)
```

```
    {

        printf("\t\tInput\n");
        printf("vector size %i\n", local_input.input_size);
        printf("v_1: ");
        print_vector(input_v_1, local_input.input_size);
        printf("\n");
        printf("v_2: ");
        print_vector(input_v_2, local_input.input_size);
        printf("\n");
        printf("Scalar: %lf\n", local_input.scalar);

        printf("\t\t Result\n");
        printf("Dot: %lf\n", dot_result);
        printf("scalar v1: ");
        print_vector(scalar_v1, local_input.input_size);
        printf("\n");
        printf("scalar v2: ");
        print_vector(scalar_v2, local_input.input_size);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

double local_dot(double *v_1, double *v_2, int size)
{
    double result = 0;
    for (int i = 0; i < size; i++)
    {
        result += v_1[i] * v_2[i];
    }

    return result;
}

double *local_scalar_multiply(double *v, int size, double scalar)
{
    double *result = malloc(size * sizeof(double));

    for (int i = 0; i < size; i++)
    {
        result[i] = v[i] * scalar;
    }

    return result;
}

double get_global_dot(double *local_dot)
{
    double result_dot;
    MPI_Reduce(local_dot, &result_dot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    return result_dot;
}

double *send_global_vector_to_root(double *local_vector, int local_size, int global_size, int my_rank, int comm_sz)
{
    double *global_vector = NULL;
    int *displacements = NULL;
    int *counts = NULL;
    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
        displacements = malloc(comm_sz * sizeof(int));
        counts = malloc(comm_sz * sizeof(int));
        generate_counts_and_displacements(counts, displacements, global_size, comm_sz);
    }

    MPI_Gatherv(local_vector, local_size, MPI_DOUBLE, global_vector, counts, displacements,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);

    free(displacements);
    free(counts);
    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;
    int *displacements = NULL;
    int *counts = NULL;
    if (my_rank == 0)
    {
        total_input = read_input(argc, argv);
        displacements = malloc(comm_sz * sizeof(int));
        counts = malloc(comm_sz * sizeof(int));
        generate_counts_and_displacements(counts, displacements, total_input.vector_size, comm_sz);
    }

    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&total_input.scalar, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);

    local_input.vector_size = r.last - r.first;
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.v_2 = malloc(local_input.vector_size * sizeof(double));
    local_input.scalar = total_input.scalar;
    local_input.input_size = total_input.vector_size;

    MPI_Scatterv(total_input.v_1, counts, displacements, MPI_DOUBLE,
                 local_input.v_1, local_input.vector_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    MPI_Scatterv(total_input.v_2, counts, displacements, MPI_DOUBLE,
                 local_input.v_2, local_input.vector_size, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    free(counts);
    free(displacements);

    return local_input;
}

void generate_counts_and_displacements(int *counts, int *displacements, int input_size, int comm_sz)
{

    for (size_t rank = 0; rank < comm_sz; rank++)
    {
        range r = range_block_partition(rank, comm_sz, input_size);
        /*
                          0 1 2 3 4
            input_vector =[a,b,c,d,e]

            p_0 = [a,b], counts[p_0] == 2, displacement[p_0] =0
            p_1 = [c], counts[p_1] == 1, displacement[p_1] == 2
            p_2 = [d], counts[p_2] == 1, displacement[p_2] == 3
            p_3 = [e], counts[p_3] == 1, displacement[p_3] == 4

            p_0 = input_vector[ displacement[p_0]: displacement[p_0] + counts[p_0] ] = [a,b]
        */
        counts[rank] = r.last - r.first;
        displacements[rank] = r.first;
    }
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }

    return r;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}
```

```
input read_input(int argc, char *argv[])
{

    if (argc != 5)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.scalar = atoi(argv[1]);

    result.vector_size = atoi(argv[4]);

    result.v_1 = read_vector(argv[2], result.vector_size);
    result.v_2 = read_vector(argv[3], result.vector_size);

    return result;
}
```

```
mpicc question_13.c -o question_13 && mpirun -n 4 ./question_13 5 "1,1,1,1,1,1" "2,2,2,2,2,2" 6 && rm question_13
               Input
vector size 6
v_1: [1.000000,1.000000,1.000000,1.000000,1.000000,1.000000]
v_2: [2.000000,2.000000,2.000000,2.000000,2.000000,2.000000]
Scalar: 5.000000
                Result
Dot: 12.000000
scalar v1: [5.000000,5.000000,5.000000,5.000000,5.000000,5.000000]
scalar v2: [10.000000,10.000000,10.000000,10.000000,10.000000,10.000000]
```

4.9 Question 16

4.9.1 Questão 16

Suppose $comm\_sz = 8$ and the vector $\mathbf{x} = (0, 1, 2, \ldots, 15)$ has been distributed among the processes using a block distribution. Draw a diagram illustrating the steps in a butterfly implementation of allgather of $\mathbf{x}$

connections

para calcular os vizinhos foi criado o seguinte script

connections.py

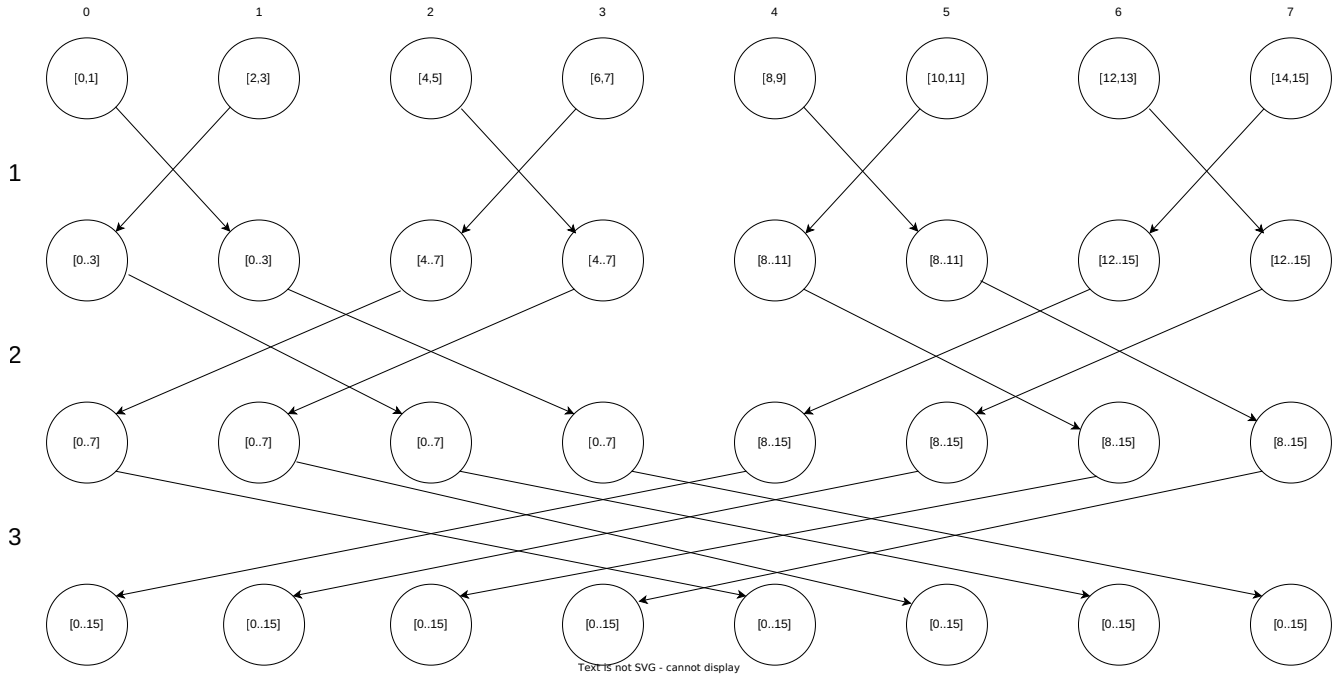
```
def main():

    def partner(rank,phrase):
        v = 2**phrase
        return rank ^ v

    for rank in range(8):
        partners = [partner(rank,phrase) for phrase in range(3)]
        print(f'P_{rank} -> {partners}')

if __name__ == '__main__':
    main()
```

Saida do script

```
P_0 -> [1, 2, 4]
P_1 -> [0, 3, 5]
P_2 -> [3, 0, 6]
P_3 -> [2, 1, 7]
P_4 -> [5, 6, 0]
P_5 -> [4, 7, 1]
P_6 -> [7, 4, 2]
P_7 -> [6, 5, 3]
```

Lê-se o resultado da seguinte maneira, processo p envia dados para seus processos parceiros onde o indicie da lista de processos parceiros representam as fases. Depois foi desenhado o diagrama de acordo

diagram



4.10 Question 17

4.10.1 Questão 17

MPI_Type_contiguous can be used to build a derived datatype from a collection of contiguous elements in an array. Its syntax is

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

Modify the *Read_vector* and *Print_vector* functions so that they use an MPI datatype created by a call to *MPI_Type_contiguous* and a *count* argument of 1 in the calls to *MPI_Scatter* and *MPI_Gather*.

question_13.c

```
/*
mpicc question_17.c -o question_17 && mpirun -n 4 ./question_17 5 "1,1,1,1,1" "2,2,2,2,2" 6 && rm question_17
*/
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef struct input
{
    double *v_1;
    double *v_2;
    double scalar;
    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

double local_dot(double *v_1, double *v_2, int size);
double *local_scalar_multiply(double *v, int size, double scalar);

double get_global_dot(double *local_dot);
double *send_global_vector_to_root(double *local_vector, int local_size, int global_size, int my_rank, int comm_sz);
void print_vector(double *vector, int size);

void generate_counts_and_displacements(int *counts, int *displacements, int input_size, int comm_sz);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

MPI_Datatype create_double_vector_type(int size);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;
    int input_size;

    int message_tag = 0;

    MPI_Init(&argc, &argv);

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

    input local_input = read_local_input(argc, argv, my_rank, comm_sz);

    double local_dot_result = local_dot(local_input.v_1, local_input.v_2, local_input.vector_size);

    double *local_scalar_v1 = local_scalar_multiply(local_input.v_1, local_input.vector_size, local_input.scalar);
    double *local_scalar_v2 = local_scalar_multiply(local_input.v_2, local_input.vector_size, local_input.scalar);

    double dot_result = get_global_dot(&local_dot_result);

    double *scalar_v1 = send_global_vector_to_root(local_scalar_v1, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);

    double *scalar_v2 = send_global_vector_to_root(local_scalar_v2, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);
```

```
    double *input_v_1 = send_global_vector_to_root(local_input.v_1, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);

    double *input_v_2 = send_global_vector_to_root(local_input.v_2, local_input.vector_size,
                                                   local_input.input_size, my_rank, comm_sz);

    if (my_rank == 0)
    {

        printf("\t\tInput\n");
        printf("vector size %i\n", local_input.input_size);
        printf("v_1: ");
        print_vector(input_v_1, local_input.input_size);
        printf("\n");
        printf("v_2: ");
        print_vector(input_v_2, local_input.input_size);
        printf("\n");
        printf("Scalar: %lf\n", local_input.scalar);

        printf("\t\t Result\n");
        printf("Dot: %lf\n", dot_result);
        printf("scalar v1: ");
        print_vector(scalar_v1, local_input.input_size);
        printf("\n");
        printf("scalar v2: ");
        print_vector(scalar_v2, local_input.input_size);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

double local_dot(double *v_1, double *v_2, int size)
{
    double result = 0;
    for (int i = 0; i < size; i++)
    {
        result += v_1[i] * v_2[i];
    }

    return result;
}

double *local_scalar_multiply(double *v, int size, double scalar)
{
    double *result = malloc(size * sizeof(double));

    for (int i = 0; i < size; i++)
    {
        result[i] = v[i] * scalar;
    }

    return result;
}

double get_global_dot(double *local_dot)
{
    double result_dot;
    MPI_Reduce(local_dot, &result_dot, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    return result_dot;
}

MPI_Datatype create_double_vector_type(int size)
{
    MPI_Datatype custom;
    // https://www.codingame.com/playgrounds/349/introduction-to-mpi/custom-types
    //  além de criar um novo tipo tem que commita-lo.
    MPI_Type_contiguous(
        size,       /* in  */
        MPI_DOUBLE, /* in  */
        &custom /* out */);

    MPI_Type_commit(&custom);
    return custom;
}
double *send_global_vector_to_root(double *local_vector, int local_size, int global_size, int my_rank, int comm_sz)
{
    double *global_vector = NULL;
    int *displacements = NULL;
    int *counts = NULL;

    MPI_Datatype local_vector_type = create_double_vector_type(local_size);

    if (my_rank == 0)
    {
        global_vector = malloc(global_size * sizeof(double));
        displacements = malloc(comm_sz * sizeof(int));
        counts = malloc(comm_sz * sizeof(int));
        generate_counts_and_displacements(counts, displacements, global_size, comm_sz);
    }
```

```
    MPI_Gatherv(local_vector, 1, local_vector_type,
                global_vector, counts, displacements, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    free(displacements);
    free(counts);
    return global_vector;
}
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz)
{
    input local_input;
    input total_input;
    int *displacements = NULL;
    int *counts = NULL;

    if (my_rank == 0)
    {
        total_input = read_input(argc, argv);
        displacements = malloc(comm_sz * sizeof(int));
        counts = malloc(comm_sz * sizeof(int));
        generate_counts_and_displacements(counts, displacements, total_input.vector_size, comm_sz);
    }

    MPI_Bcast(&total_input.vector_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&total_input.scalar, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    range r = range_block_partition(my_rank, comm_sz, total_input.vector_size);

    local_input.vector_size = r.last - r.first;
    local_input.v_1 = malloc(local_input.vector_size * sizeof(double));
    local_input.v_2 = malloc(local_input.vector_size * sizeof(double));
    local_input.scalar = total_input.scalar;
    local_input.input_size = total_input.vector_size;

    MPI_Datatype local_vector_type = create_double_vector_type(local_input.vector_size);

    MPI_Scatterv(total_input.v_1, counts, displacements, MPI_DOUBLE,
                 local_input.v_1, 1, local_vector_type, 0, MPI_COMM_WORLD);

    MPI_Scatterv(total_input.v_2, counts, displacements, MPI_DOUBLE,
                 local_input.v_2, 1, local_vector_type, 0, MPI_COMM_WORLD);

    free(counts);
    free(displacements);

    return local_input;
}

void generate_counts_and_displacements(int *counts, int *displacements, int input_size, int comm_sz)
{

    for (size_t rank = 0; rank < comm_sz; rank++)
    {
        range r = range_block_partition(rank, comm_sz, input_size);
        /*
                          0 1 2 3 4
            input_vector =[a,b,c,d,e]

            p_0 = [a,b], counts[p_0] == 2, displacement[p_0] =0
            p_1 = [c], counts[p_1] == 1, displacement[p_1] == 2
            p_2 = [d], counts[p_2] == 1, displacement[p_2] == 3
            p_3 = [e], counts[p_3] == 1, displacement[p_3] == 4

            p_0 = input_vector[ displacement[p_0]: displacement[p_0] + counts[p_0] ] = [a,b]
        */
        counts[rank] = r.last - r.first;
        displacements[rank] = r.first;
    }
}

range range_block_partition(int my_rank, int total_process, int input_size)
{
    range r;

    int division = input_size / total_process;
    int rest = input_size % total_process;

    if (my_rank < rest)
    {
        r.first = my_rank * (division + 1);
        r.last = r.first + division + 1;
    }
    else
    {
        r.first = my_rank * division + rest;
        r.last = r.first + division;
    }

    return r;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
```

```
    int current_vector_index = 0;

    for (size_t i = 0; input[i] != '\0'; i++)
    {

        if (input[i] != ',')
        {

            v[current_vector_index] = atoi(&input[i]);
            current_vector_index++;
        }
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

input read_input(int argc, char *argv[])
{

    if (argc != 5)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.scalar = atoi(argv[1]);

    result.vector_size = atoi(argv[4]);

    result.v_1 = read_vector(argv[2], result.vector_size);
    result.v_2 = read_vector(argv[3], result.vector_size);

    return result;
}
```

```
mpicc question_17.c -o question_17 && mpirun -n 4 ./question_17 5 "1,1,1,1,1,1" "2,2,2,2,2,2" 6 && rm question_17

               Input
vector size 6
v_1: [1.000000,1.000000,1.000000,1.000000,1.000000,1.000000]
v_2: [2.000000,2.000000,2.000000,2.000000,2.000000,2.000000]
Scalar: 5.000000
                Result
Dot: 12.000000
scalar v1: [5.000000,5.000000,5.000000,5.000000,5.000000,5.000000]
scalar v2: [10.000000,10.000000,10.000000,10.000000,10.000000,10.000000]
```

4.11 Question 19

4.11.1 Questão 19

MPI_Type_indexed can be used to build a derived datatype from arbitrary array elements. Its syntax is

```
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 */);
```

Unlike *MPI_Type_create_struct*, the displacements are measured in units of *old_mpi_t* --not bytes. Use *MPI_Type_indexed* to create a derived datatype that corresponds to the upper triangular part of a square matrix. For example in the 4×4 matrix.

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

the upper triangular part is the elements 0, 1, 2, 3, 5, 6, 7, 10, 11, 15. Process 0 should read in an $n \times n$ matrix as a one-dimensional array, create the derived datatype, and send the upper triangular part with a single call to *MPI_Send*. Process 1 should receive the upper triangular part with a single call ot *MPI_Recv* and then print the data it received.

question_19.c

```c
/*
mpicc -lm question_19.c -o question_19 && mpirun -n 2 ./question_19 "0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15" 16 && rm question_19
*/
#include <mpi.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
typedef struct input
{
    double *v_1;

    int vector_size;
    int input_size;
} input;

input read_input(int argc, char *argv[]);
input read_local_input(int argc, char *argv[], int my_rank, int comm_sz);
double *read_vector(char *input, int vector_size);

void print_vector(double *vector, int size);
void print_matrix(double *matrix, int rank);
void generate_counts_and_displacements(int *counts, int *displacements, int input_size, int comm_sz);

typedef struct range
{
    int first;
    int last;
} range;

range range_block_partition(int my_rank, int total_process, int input_size);

MPI_Datatype create_upper_triangular_type(int rank_matrix);

int main(int argc, char *argv[])
{

    int my_rank;
    int comm_sz;
    int input_size;

    // https://www.rookiehpc.com/mpi/docs/mpi_type_indexed.php

    MPI_Init(&argc, &argv);

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

    if (comm_sz != 2)
```

```
    {
        printf("esse exercício utiliza apenas de  2 processo.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int matrix_rank;
    input local_input;
    int source = 0;
    int dest = 1;
    double *upper_triangle = NULL;

    int message_tag = 0;

    if (my_rank == source)
    {

        local_input = read_input(argc, argv);
        // o rank de uma matriz quadrada é a raiz do seu número de elementos
        matrix_rank = sqrt(local_input.input_size);
        printf("\t\tInput\n");
        printf("vector size %i\n", local_input.input_size);
        printf("rank %i\n", matrix_rank);
        printf("\t\tmatrix \n");
        print_matrix(local_input.v_1, matrix_rank);
        printf("\n");

        MPI_Datatype triangular_type = create_upper_triangular_type(matrix_rank);

        MPI_Send(local_input.v_1, 1, triangular_type, dest, 0, MPI_COMM_WORLD);
    }

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

    if (my_rank == dest)
    {
        int size = matrix_rank*(matrix_rank +1)/2;
        upper_triangle = malloc(size * sizeof(double));
        for (size_t i = 0; i < size; i++)
        {
            upper_triangle[i] = 0.0;
        }
        MPI_Recv(upper_triangle, size, MPI_DOUBLE, source, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        printf("\t\tResult\n");
        printf("upper triangular part of a square matrix: \n");
        print_vector(upper_triangle, size);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

MPI_Datatype create_upper_triangular_type(int rank_matrix)
{
    MPI_Datatype custom;

    /*
        v[rank_matrix][rank_matrix] =
        {
            {0, 1 ,2, 3}, v[0,0:4]
            {4, 5, 6, 7}, v[1,1:4]
            {8, 9, 10,11},v[2,2:4]
            {12,13,14,15} v[3,3:4]
        }
    */
    int *lengths = malloc(rank_matrix * sizeof(int));
    int *displacements = malloc(rank_matrix * sizeof(int));

    for (size_t i = 0; i < rank_matrix; i++)
    {
        int begin = i;
        int end = rank_matrix;
        lengths[i] = end - begin; // end - begin == len(v[i][i:4])
        /*
        mapeando  matrix v[row][col] para sua posição em um vetor 1d  v[row*width + col]
        portanto v[2][2] == v[2*rank_matrix + 2]
        */
        displacements[i] = rank_matrix * i + i;
    }
    /*
    o número de elementos da matrix triangular superior é
    metade do número da matriz, portanto: n*n/n == n
    */
    /*

    /*
     MPI_Type_indexed creates an MPI datatype as a sequence of blocks,
     each made by replicating an existing MPI_Datatype a certain number of times.
     Also, each block is located at a certain displacement from the beginning of the MPI datatype.

     https://www.rookiehpc.com/mpi/docs/mpi_type_indexed.php
    */
```

```
    MPI_Type_indexed(rank_matrix, lengths, displacements, MPI_DOUBLE, &custom);

    MPI_Type_commit(&custom);

    return custom;
}

double *read_vector(char *input, int vector_size)
{

    double *v = malloc(vector_size * sizeof(double));
    int current_vector_index = 0;

    const char delimiter[] = ",";

    for (size_t i = 0; i < vector_size; i++)
    {

        v[i] = strtod(input, &input);
        input++; // increment comma
    }

    return v;
}

void print_vector(double *vector, int size)
{
    printf("[");
    for (int i = 0; i < size - 1; i++)
    {
        printf("%lf,", vector[i]);
    }
    printf("%lf]", vector[size - 1]);
}

void print_matrix(double *matrix, int rank)
{

    for (size_t i = 0; i < rank; i++)
    {
        print_vector(&matrix[i * rank], rank);
        printf("\n");
    }
}

input read_input(int argc, char *argv[])
{

    if (argc != 3)
    {
        printf("Número de entradas erradas\n");
        exit(1);
    }

    input result;

    result.vector_size = atoi(argv[2]);
    result.input_size = result.vector_size;
    result.v_1 = read_vector(argv[1], result.vector_size);

    return result;
}
```

```
mpicc -lm question_19.c -o question_19 && mpirun -n 2 ./question_19 "0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15" 16 && rm question_19

                Input
vector size 16
rank 4
                matrix
[0.000000,1.000000,2.000000,3.000000]
[4.000000,5.000000,6.000000,7.000000]
[8.000000,9.000000,10.000000,11.000000]
[12.000000,13.000000,14.000000,15.000000]

                Result
upper triangular part of a square matrix:
[0.000000,1.000000,2.000000,3.000000,5.000000,6.000000,7.000000,10.000000,11.000000,15.000000]
```