

DISCIPLINA

Introdução à Computação Paralela - MPI -

Prof. Kayo Gonçalves

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

MPI - Message Passing Interface

- Padrão para comunicação de dados em computação paralela
- Utilizado **principalmente** em Memória distribuída (cluster)
- Baseado em envio/recebimento de mensagens
- Termo utilizado é “**processos**”

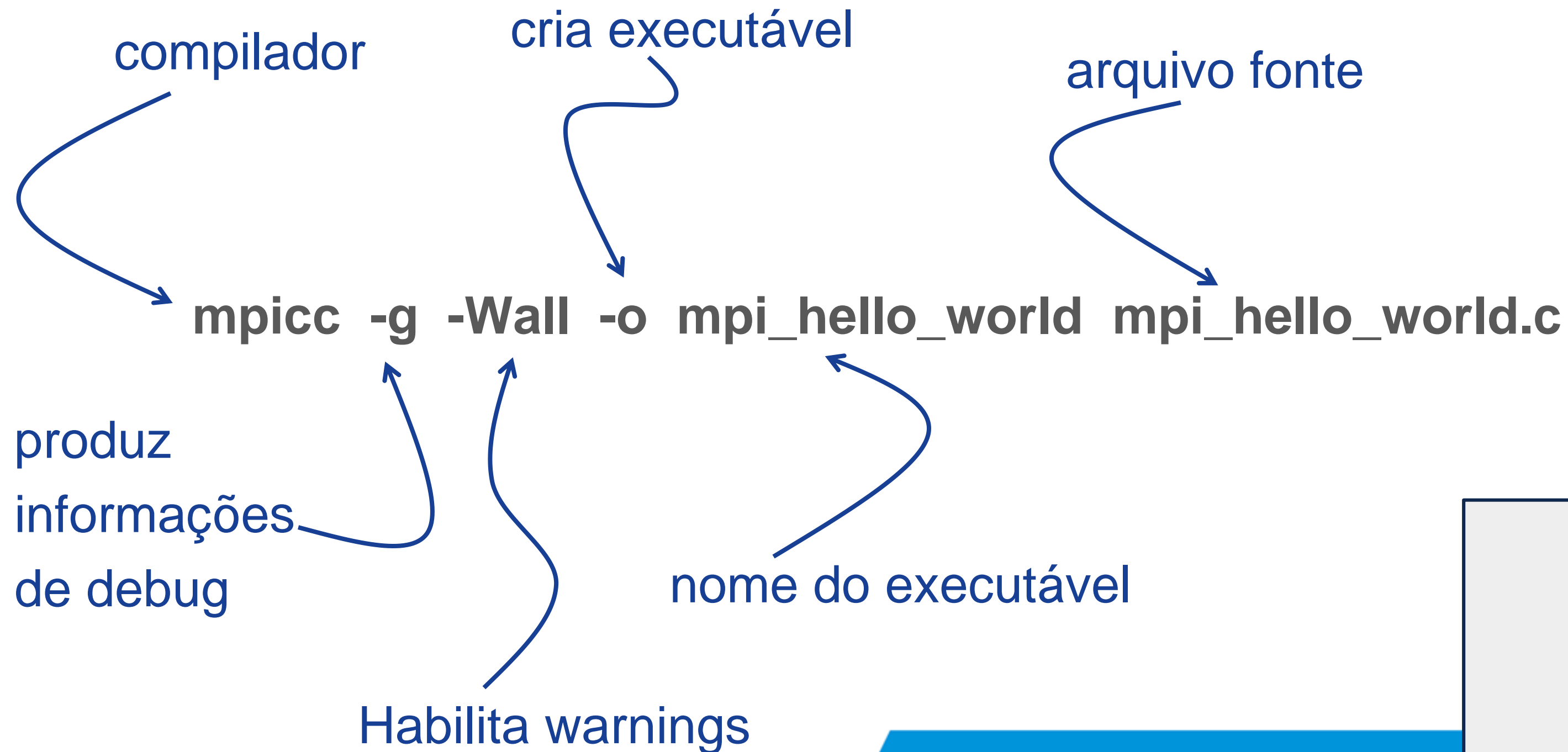


MPI - Message Passing Interface

- **Objetivo:** Criar vários processos e executá-los em paralelo.
 - O cluster deve estar previamente configurado.
- Cada processo é identificado com um número inteiro não-negativo começando por 0 (zero)
 - p processos são numerados **0, 1, 2, ..., $p-1$** .
 - Boa prática utilizar **mestre 0**



MPI – Compilar em C/C++



MPI – Executar em C/C++

`mpiexec -n <nº de processos> <executável>`

`mpiexec -n 1 ./mpi_hello_world`  1 processo

`mpiexec -n 4 ./mpi_hello_world`

4 processos 

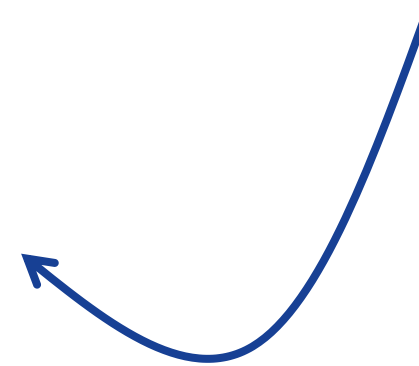


MPI – Resultado

```
mpiexec -n 1 ./mpi_hello_world  
Proc 0 of 1 > Hello World!
```

```
mpiexec -n 4 ./mpi_hello_world  
Proc 0 of 4 > Hello World!  
Proc 1 of 4 > Hello World!  
Proc 2 of 4 > Hello World!  
Proc 3 of 4 > Hello World!
```

$p=1$ não é código serial



MPI – Não determinismo

```
mpiexec -n 4 ./mpi_hello_world
```

Proc 0 of 4 > Hello World!

Proc 1 of 4 > Hello World!

Proc 2 of 4 > Hello World!

Proc 3 of 4 > Hello World!

```
mpiexec -n 4 ./mpi_hello_world
```

Proc 3 of 4 > Hello World!

Proc 1 of 4 > Hello World!

Proc 2 of 4 > Hello World!

Proc 0 of 4 > Hello World!

Que sorte



É, né?





Hello World

MPI – Identificar em C/C++

- Escrito em C/C++
 - Possui “main”
 - Pode usar stdio.h, string.h, etc.
- Precisa adicionar **mpi.h** no cabeçalho
- Identificadores MPI começam com “MPI_”
 - A primeira letra após “_” é sempre maiúscula
 - Ajuda evitar confusão
- Exemplo: MPI_Init (...)



Hello World C/C++ (sem MPI)

```
#include <stdio.h>

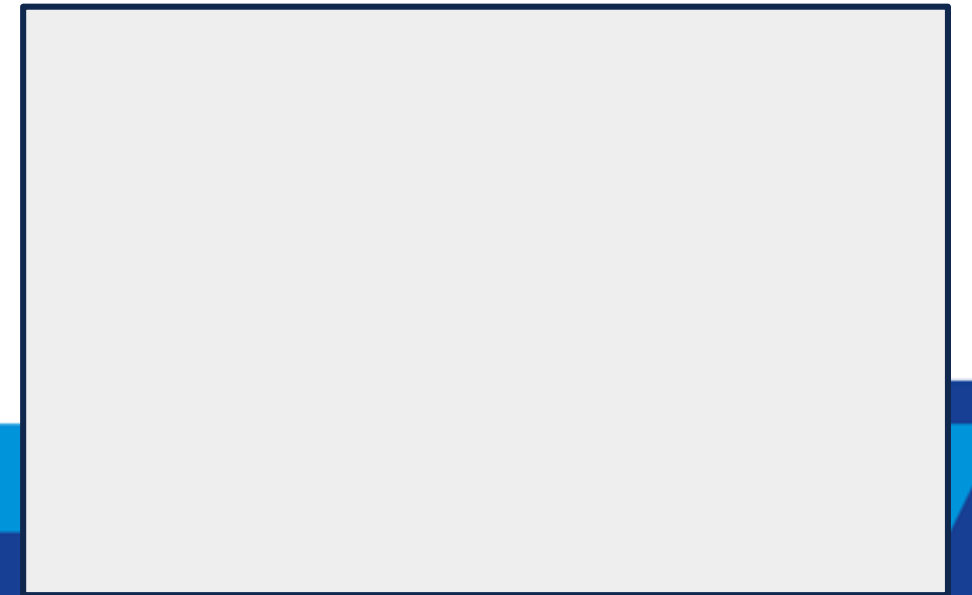
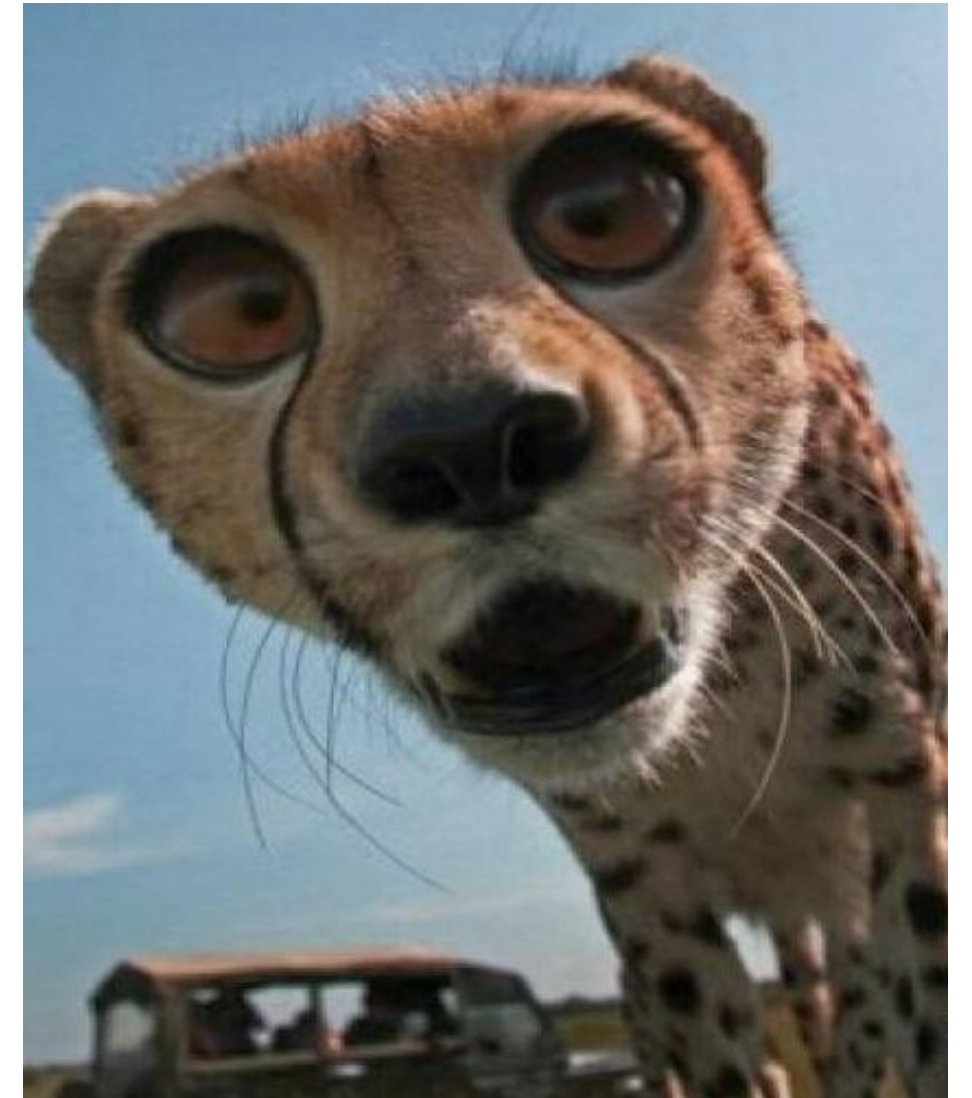
int main(void) {
    printf("hello, world\n");

    return 0;
}
```



Hello World C/C++ (com MPI)

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(void) {
5      int my_rank, comm_sz;
6
7      MPI_Init(NULL, NULL);
8      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
9      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
10
11     printf("Proc %d of %d > Hello World!\n",
12           my_rank, comm_sz);
13
14     MPI_Finalize();
15     return 0;
16 } /* main */
```



PALMA, PALMA, NÃO PRIEMOS CÂNICO!

CHAPOLIN COLORADO



Chapolin Colorado

EUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU!

Analizando código (1)

```
1 #include <stdio.h>
2 #include <mpi.h>
3
4 int main(void) {
```

← Uso livre de bibliotecas

← Obrigatório

← Pode criar constantes



Analizando código (2)

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

```
6  
7     MPI_Init(NULL, NULL);
```

```
8     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
```

```
9     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Executado em série
(1 core ou processo)

Executado em paralelo
(n cores ou processos)

Todas as variáveis são copiadas para os processos criados em MPI_Init(..).

Cada processo terá uma cópia local de **comm_sz**, **my_rank**

Analizando código (3)

```
4 int main(void) {  
5     int my_rank, comm_sz;  
6  
7     MPI_Init(NULL, NULL);  
8     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
9     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Setup Inicial do MPI.
Abre a região paralela
Usaremos NULL,NULL

```
int MPI_Init(  
    int*      argc_p  /* in/out */,  
    char***   argv_p  /* in/out */);
```

Analizando código (4)

```
4 int main(void) {  
5     int my_rank, comm_sz;  
6  
7     MPI_Init(NULL, NULL); ←  
8     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
9     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

MPI_Init(...) define um comunicador que consiste de todos os processos criados quando o programa é inicializado.

O nome do comunicador é **MPI_COMM_WORLD**

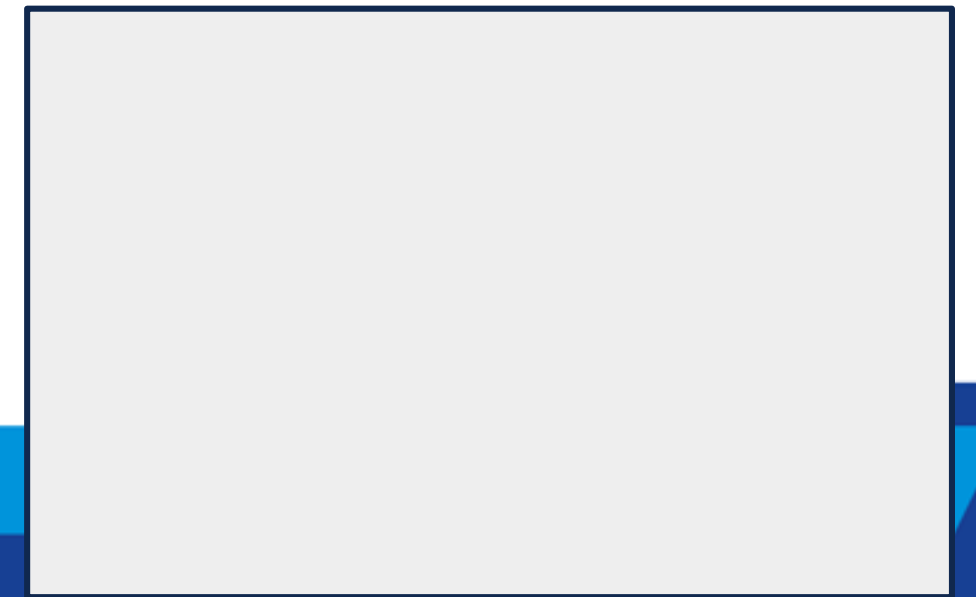
Analizando código (5)

```
4 int main(void) {  
5     int my_rank, comm_sz;  
6  
7     MPI_Init(NULL, NULL);  
8     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
9     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

**Estão rodando
em paralelo**

PENSAMENTO PARALELO NÃO-DETERMINÍSTICO

DICA: Imagine que os cores tem velocidades diferentes



Analizando código (6)

```
4 int main(void) {  
5     int my_rank, comm_sz;  
6  
7     MPI_Init(NULL, NULL);  
8     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
9     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Armazena em **comm_sz** o número de processos do comunicador

```
int MPI_Comm_size(  
    MPI_Comm comm      /* in */,  
    int* comm_sz_p     /* out */);
```

Analizando código (7)

```
4 int main(void) {  
5     int my_rank, comm_sz;  
6  
7     MPI_Init(NULL, NULL);  
8     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
9     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Armazena em
my_rank o número
de processo

```
int MPI_Comm_rank(  
    MPI_Comm comm    /* in */,  
    int* my_rank_p    /* out */);
```

Analizando código (8)

```
10
11  printf("Proc %d of %d > Hello World!\n",
12         my_rank, comm_sz);
13
14  MPI_Finalize();
15  return 0;
16 }
```

/* main */

Cada processo irá imprimir



Analizando código (9)

```
10
11 printf("Proc %d of %d > Hello World!\n",
12         my_rank, comm_sz);
13
14 MPI_Finalize();
15 return 0;
16 }
```

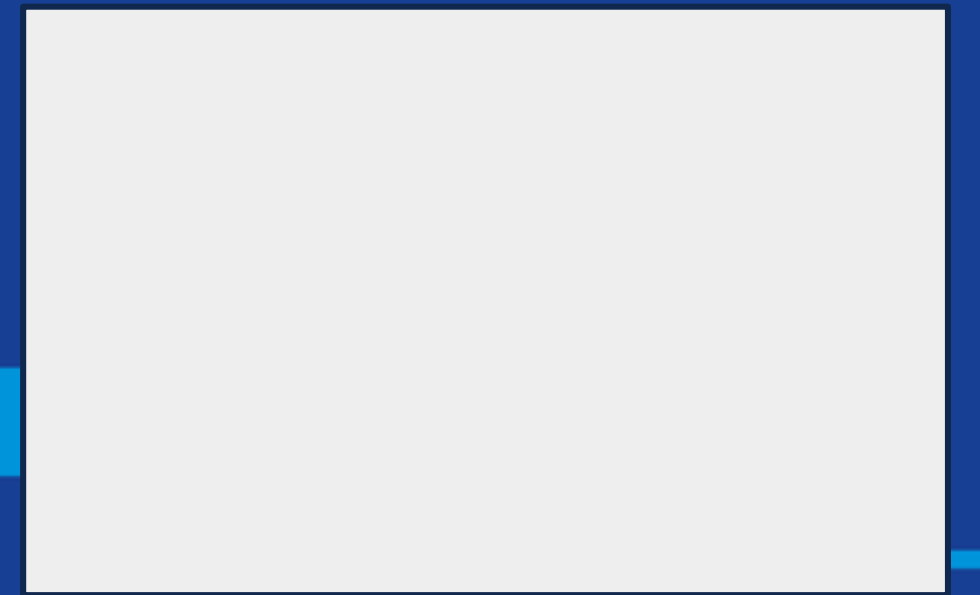
/ main */*

Diz ao MPI que finalize a região paralela. Limpe tudo alocado para este programa.

```
int MPI_Finalize(void);
```

Troca de mensagens entre processos

Saudações P2P



Troca de Mensagens

```
1 #include <stdio.h>
2 #include <string.h> /* For strlen */
3 #include <mpi.h>    /* For MPI functions, etc */
4
5 const int MAX_STRING = 100;
6
7 int main(void) {
8     char    greeting[MAX_STRING];
9     int     comm_sz; /* Number of processes */
10    int     my_rank; /* My process rank */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17        sprintf(greeting, "Greetings from process %d of %d!",
18                my_rank, comm_sz);
19        MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20                MPI_COMM_WORLD);
21    } else {
22        printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23        for (int q = 1; q < comm_sz; q++) {
24            MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25                    0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26            printf("%s\n", greeting);
27        }
28    }
29
30    MPI_Finalize();
31    return 0;
32 } /* main */
```

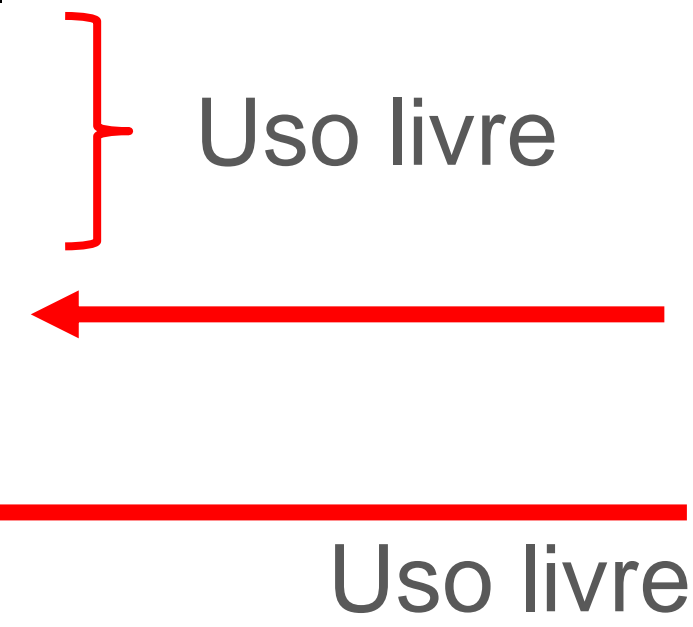


Analizando código (1)

```
1 #include <stdio.h>
2 #include <string.h>  /* For strlen */
3 #include <mpi.h>      /* For MPI functions , etc */
4
5 const int MAX_STRING = 100;
6
7 int main(void) {
```

Uso livre

Uso livre



Analizando código (2)

```
7 | int main(void) {  
8 |     char        greeting[MAX_STRING];  
9 |     int          comm_sz;  /* Number of processes */  
10 |    int          my_rank;   /* My process rank      */  
11 |  
12 |    MPI_Init(NULL, NULL);  
13 |    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
14 |    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Executado em série
(1 core ou processo)

Executado em paralelo
(n cores ou processos)

Todas as variáveis são copiadas para os processos criados em MPI_Init(..).

Cada processo terá uma cópia local de **greetings**, **comm_sz**, **my_rank** e **MAX_STRING**

Analizando código (3)

```
7  int main(void) {
8      char        greeting[MAX_STRING];
9      int         comm_sz;  /* Number of processes */
10     int         my_rank;  /* My process rank      */
11
12     MPI_Init(NULL, NULL);
13     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Setup Inicial do MPI.
Abre a região paralela
Usaremos NULL,NULL

```
int MPI_Init(
    int*      argc_p  /* in/out */,
    char***   argv_p  /* in/out */);
```

Analizando código (4)

```
7  int main(void) {  
8      char        greeting[MAX_STRING];  
9      int         comm_sz;  /* Number of processes */  
10     int         my_rank;  /* My process rank      */  
11  
12     MPI_Init(NULL, NULL);  
13     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
14     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Setup Inicial do MPI.
Abre a região paralela
Usaremos NULL,NULL

MPI_Init(...) define um comunicador que consiste de todos os processos criados quando o programa é inicializado.

O nome do comunicador é **MPI_COMM_WORLD**

Analizando código (5)

```
7  int main(void) {  
8      char        greeting[MAX_STRING];  
9      int          comm_sz;  /* Number of processes */  
10     int          my_rank;  /* My process rank      */  
11  
12     MPI_Init(NULL, NULL);  
13     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
14     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

**Estão rodando
em paralelo**

PENSAMENTO PARALELO NÃO-DETERMINÍSTICO

DICA: Imagine que os cores tem velocidades diferentes

Analizando código (6)

```
7  int main(void) {
8      char    greeting[MAX_STRING];
9      int     comm_sz;  /* Number of processes */
10     int     my_rank;  /* My process rank      */
11
12     MPI_Init(NULL, NULL);
13     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Armazena em **comm_sz** o número de processos do comunicador

```
int MPI_Comm_size(
    MPI_Comm comm      /* in */,
    int* comm_sz_p     /* out */);
```

Analizando código (7)

```
7  int main(void) {
8      char        greeting[MAX_STRING];
9      int         comm_sz;  /* Number of processes */
10     int         my_rank;  /* My process rank      */
11
12     MPI_Init(NULL, NULL);
13     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Armazena em
my_rank o número
de processo

```
int MPI_Comm_rank(
    MPI_Comm comm    /* in */,
    int*      my_rank_p /* out */);
```

Analizando código (8)

```
15
16  if (my_rank != 0) {
17      sprintf(greeting, "Greetings from process %d of %d!",
18              my_rank, comm_sz);
19      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20              MPI_COMM_WORLD);
21  } else {
22      printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23      for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25                  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27      }
28  }
```

Escravos
ENVIAM MSG p/
mestre

Mestre recebe
msgs e escreve
na tela



Analizando código (9)

```
15 |
16 |     if (my_rank != 0) {
17 |         sprintf(greeting, "Greetings from process %d of %d!",
18 |             my_rank, comm_sz);
19 |         MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20 |             MPI_COMM_WORLD);
21 |     } else {
```

```
int MPI_Send(
```

```
void*      msg_buf_p      /* in */,
int        msg_size       /* in */,
MPI_Datatype msg_type     /* in */,
int        dest           /* in */,
int        tag            /* in */,
MPI_Comm   communicator   /* in */);
```

Armazena a msg
“greetings ...” na
variável **greetings**

Escravo envia msg
para o mestre



Analizando código (10)

mensagem

Tipo da mensagem

tag (sempre usaremos 0)

```
MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
```

comunicador

destino

tamanho da mensagem

The diagram illustrates the components of the `MPI_Send` function call. Arrows point from descriptive labels to the corresponding arguments in the code: `greeting` is the message; `strlen(greeting)+1` is the message size; `MPI_CHAR` is the message type; `0` is the destination; `0` is the tag; and `MPI_COMM_WORLD` is the communicator. A large gray rectangle is present in the bottom right corner of the slide.


Analizando código (11)

```
21  } else {
22      printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
23      for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25                  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27      }
28  }
```

Mestre escreve msg
própria



Mestre recebe
msgs 1 por 1 e
escreve msg na
tela



```
int MPI_Recv(
    void*      msg_buf_p    /* out */,
    int        buf_size     /* in  */,
    MPI_Datatype buf_type    /* in  */,
    int        source        /* in  */,
    int        tag           /* in  */,
    MPI_Comm   communicator /* in  */,
    MPI_Status* status_p     /* out */);
```

Analizando código (12)

mensagem

Tipo da mensagem

tag (sempre usaremos 0)

status da msg recebida
sempre usaremos esta

```
MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

tamanho do buffer

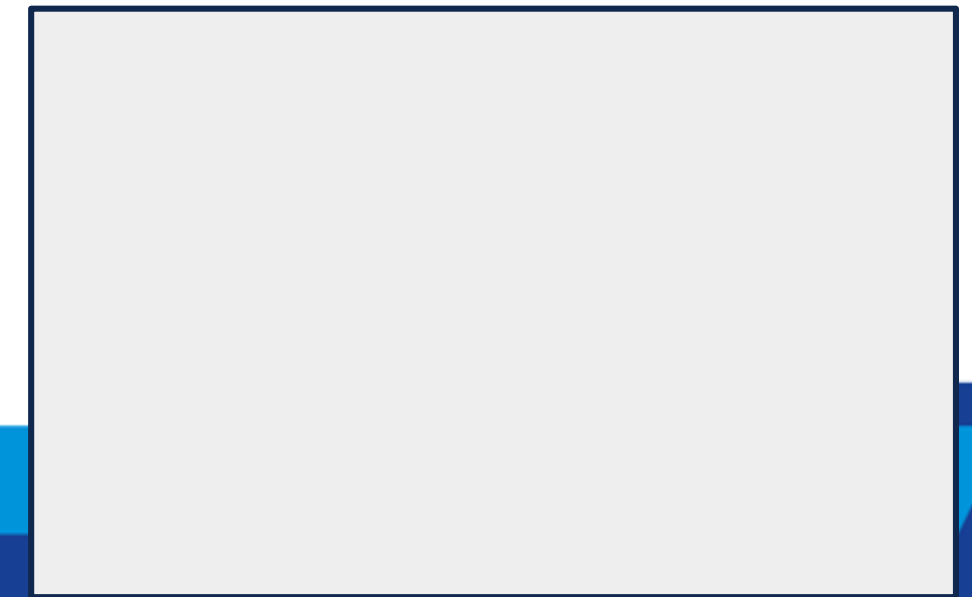
fonte

comunicador

A diagram illustrating the parameters of the MPI_Recv function call. The function call is: MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);. Annotations with arrows point to each argument: 'mensagem' points to 'greeting', 'tamanho do buffer' points to 'MAX_STRING', 'Tipo da mensagem' points to 'MPI_CHAR', 'fonte' points to 'q', 'tag (sempre usaremos 0)' points to '0', 'comunicador' points to 'MPI_COMM_WORLD', and 'status da msg recebida sempre usaremos esta' points to 'MPI_STATUS_IGNORE'. A large grey rectangle is present in the bottom right corner.

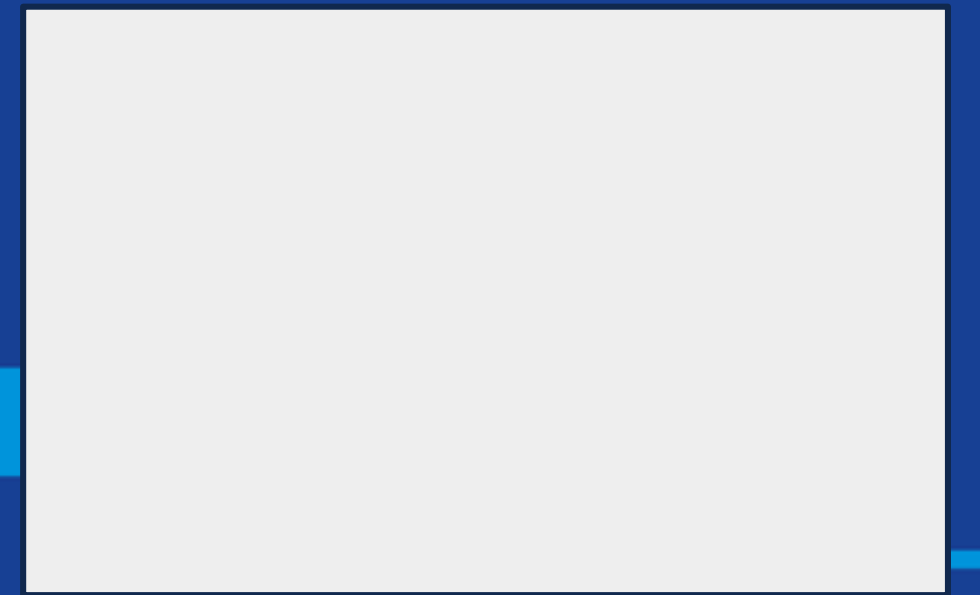
Tipos de dados MPI

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	



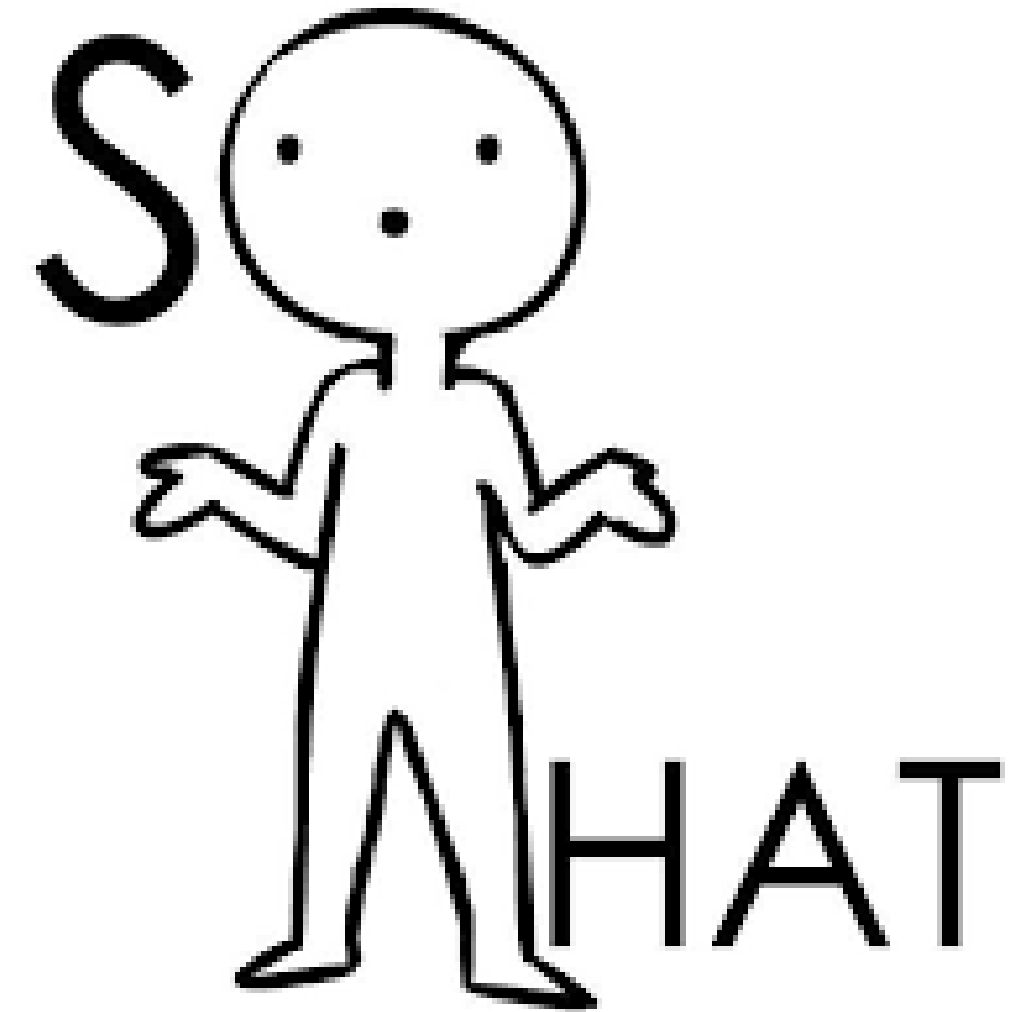
Entendendo melhor

MPI_Send & MPI_Recv



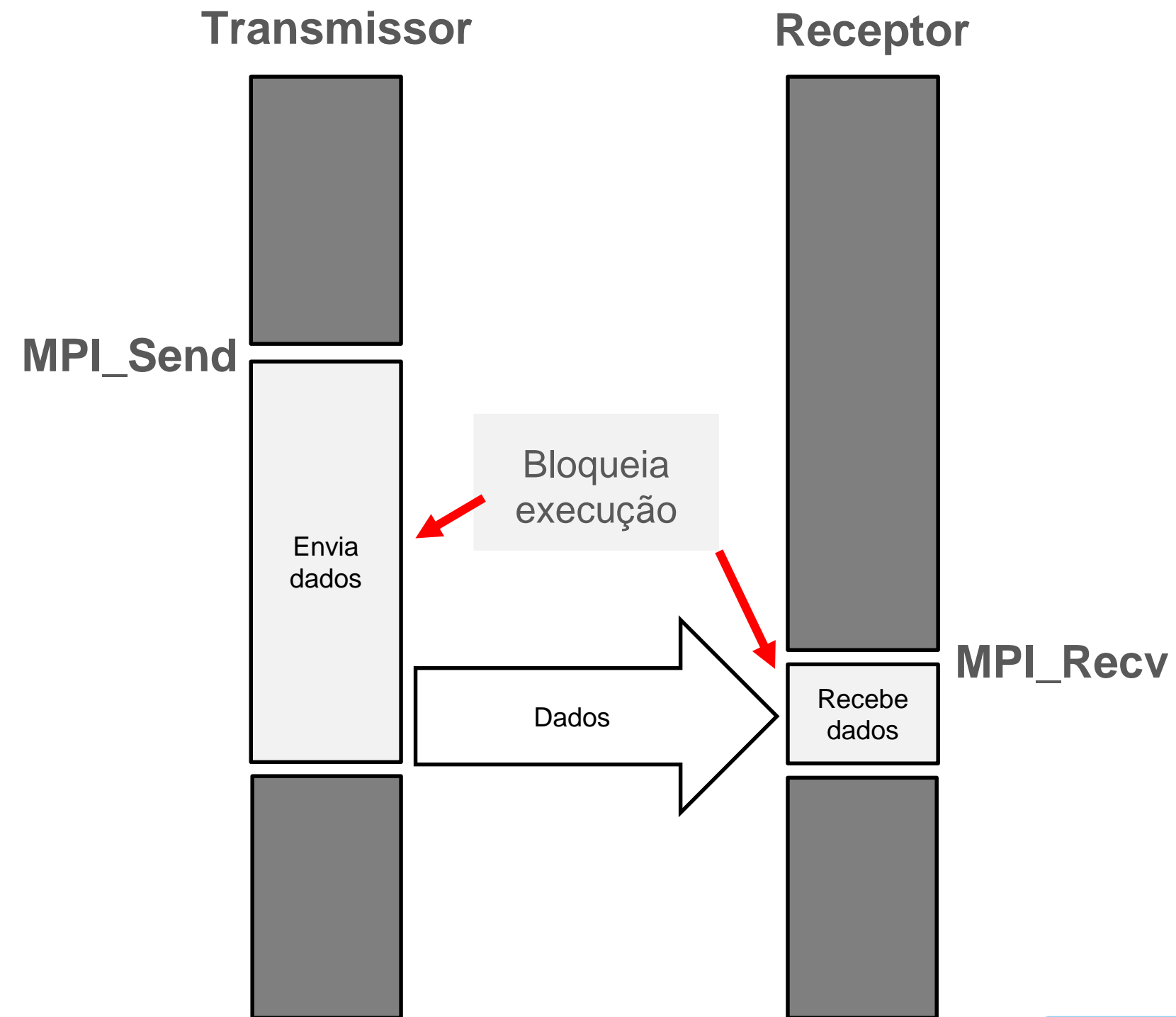
MPI_Send & MPI_Recv

- MPI_Send tem dois modos de funcionamento
 - 1) **MPI_Send com bloqueio**
 - 2) **MPI_Send com buffer**
- **MPI_Recv** sempre funciona com **bloqueio**



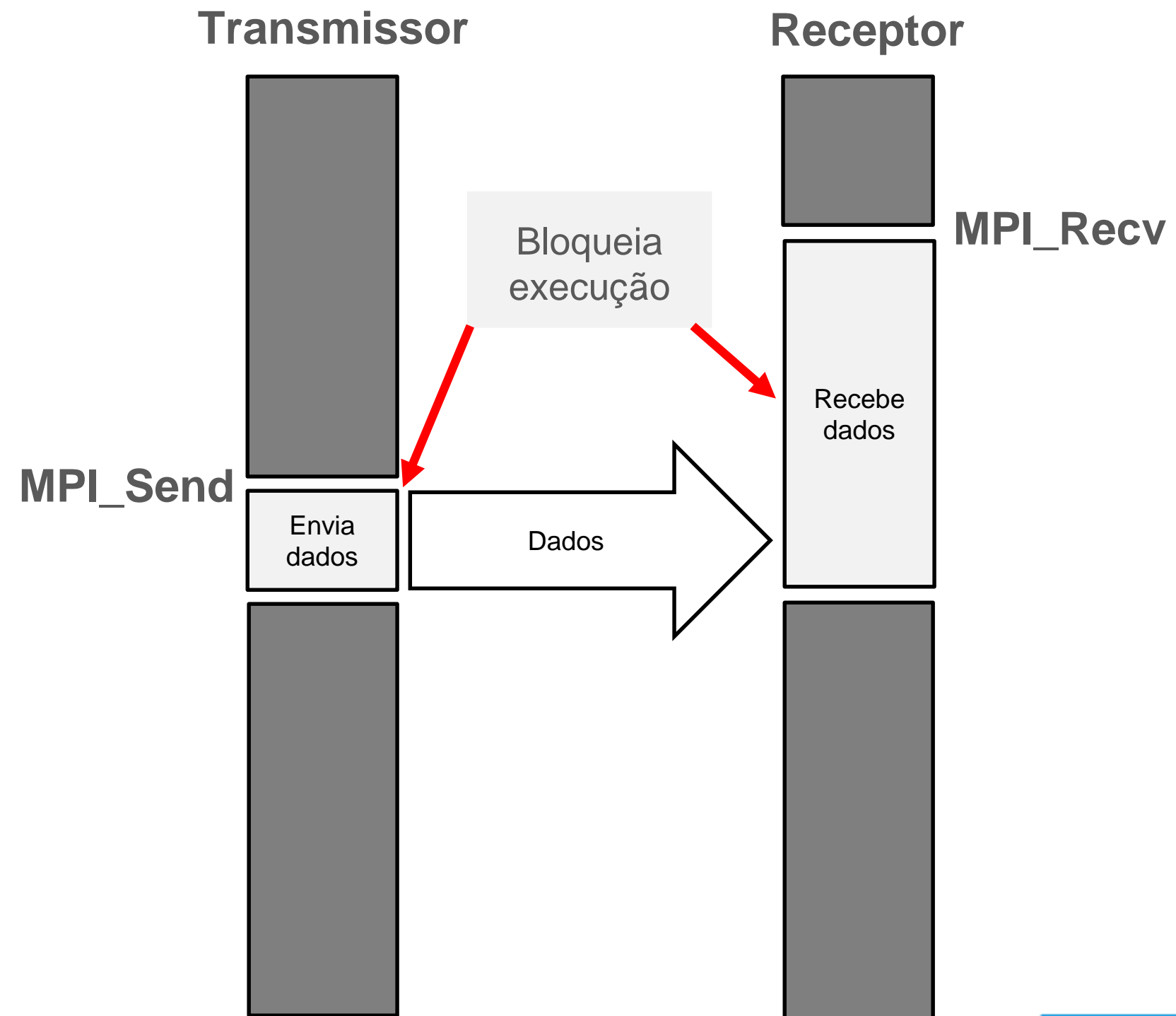
MPI_Send com bloqueio

Exemplo 1



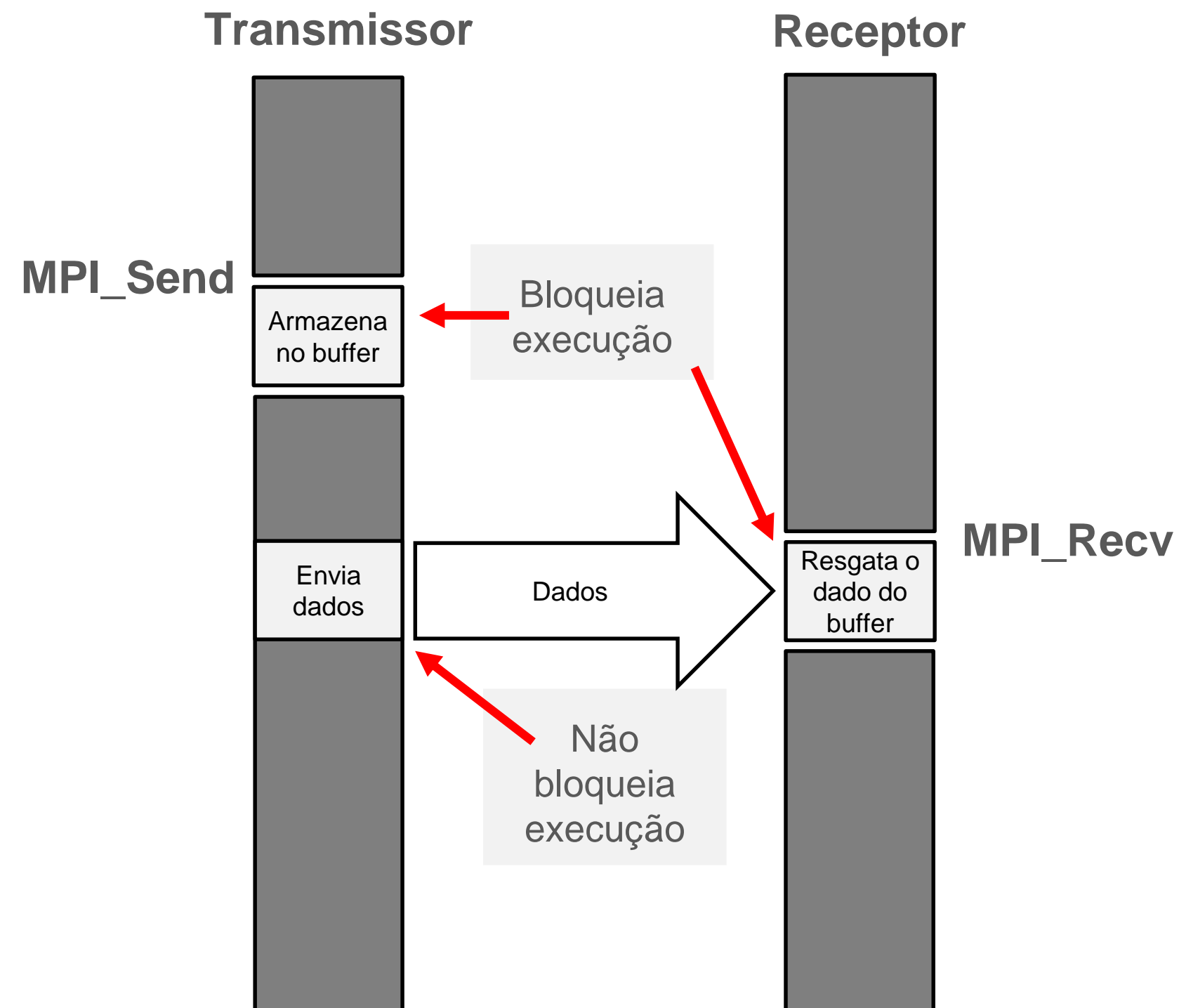
MPI_Send com bloqueio

Exemplo 2



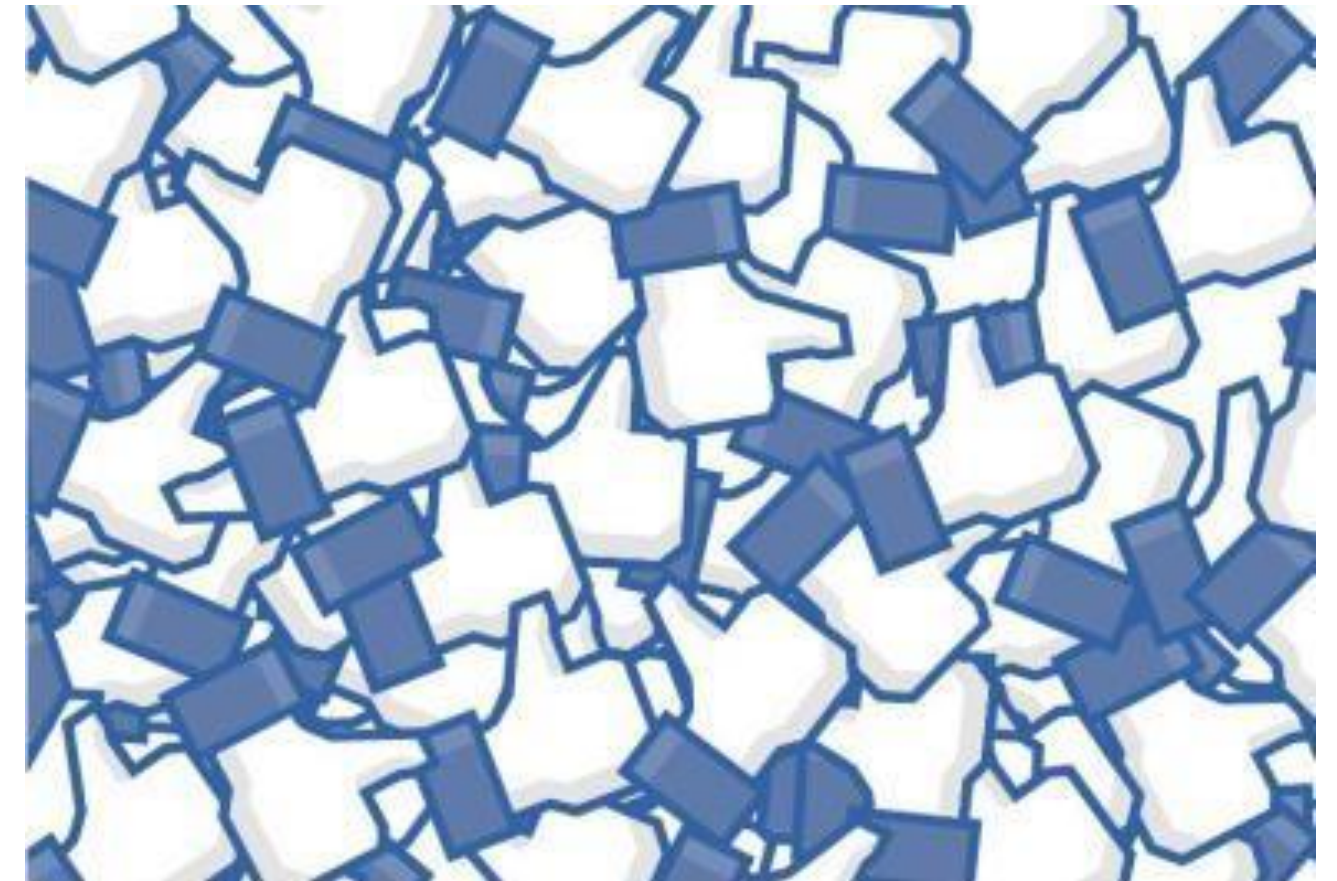
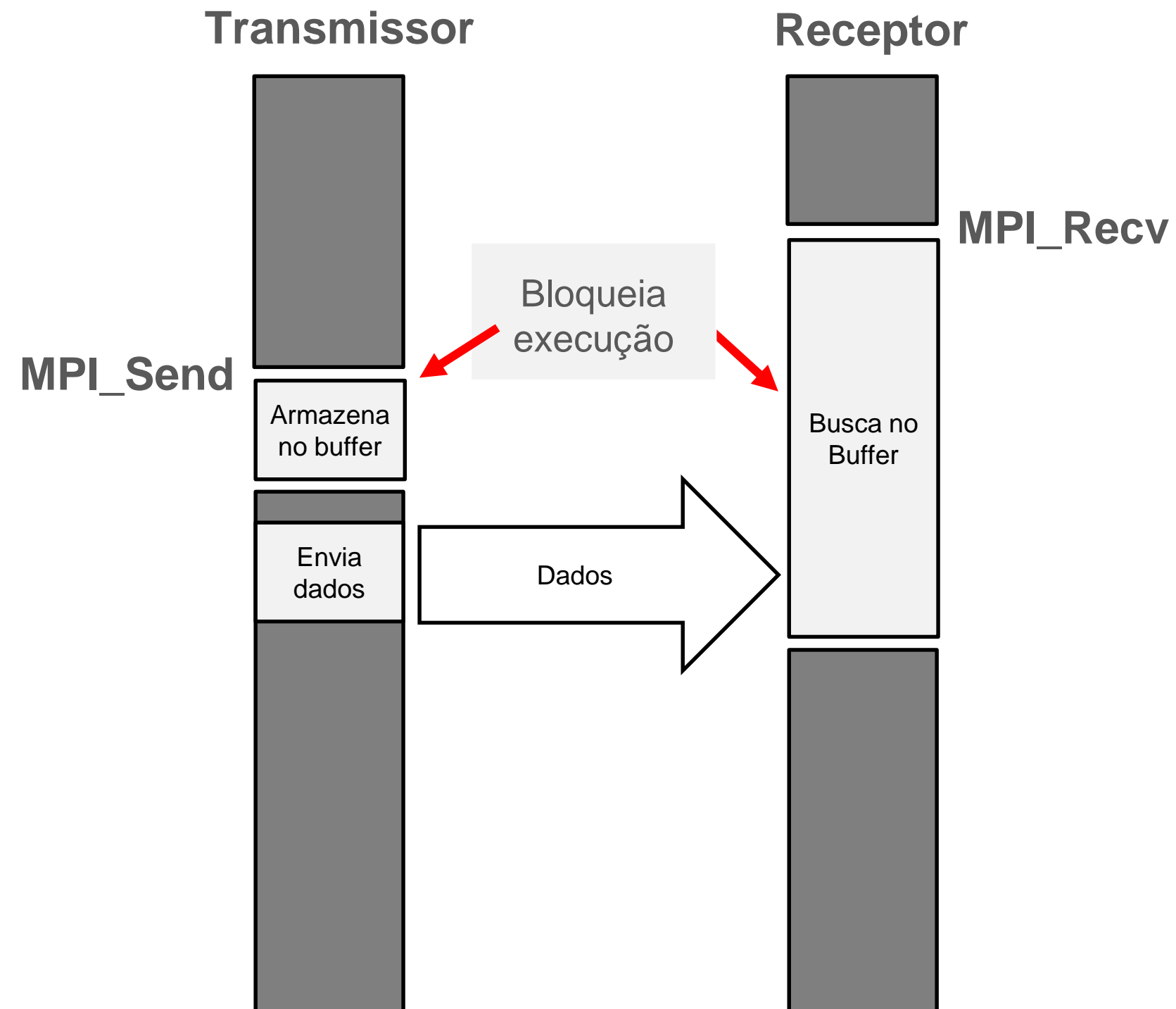
MPI_Send com buffer

Exemplo 1



MPI_Send com buffer

Exemplo 2



Segurança em programas MPI (1)

- Muitas implementações do MPI definem um limite no qual o sistema alterna do buffer para o bloqueio
 - **Mensagens** relativamente **pequenas** serão armazenadas em **buffer** pelo MPI_Send.
 - **Mensagens maiores** causarão um **bloqueio**



Segurança em programas MPI (2)

- Um programa que depende somente do **MPI_Send com Buffer** é considerado **inseguro** (unsafe).
- Esse programa pode ser executado sem problemas para vários conjuntos de entradas, mas pode travar com outros conjuntos.



Segurança em programas MPI (3)

```
1  
2  
3 ...  
4 int a[10], b[10], myrank;  
5 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);  
6  
7 if (myrank == 0) {  
8     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);  
9     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);  
10 }  
11 else if (myrank == 1) {  
12     MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);  
13     MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);  
14 }
```



- Se MPI_Send com buffer: Programa funciona
- Se MPI_Send Bloqueante: Programa Trava
- **Motivo: tag**



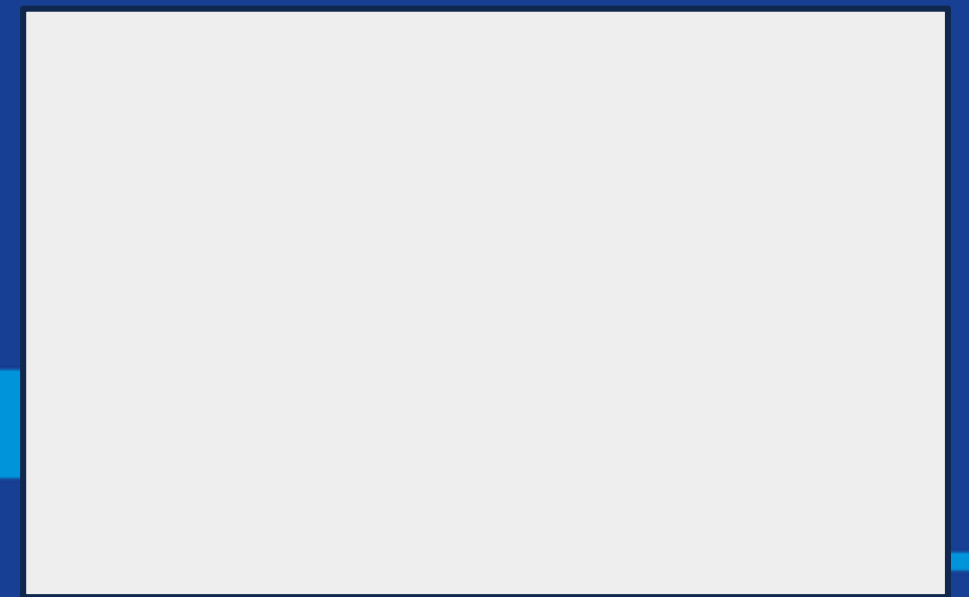
“Garanta um par Send-Recv”

Dica de ouro, platina, diamante, mestre, grão-mestre e desafiante

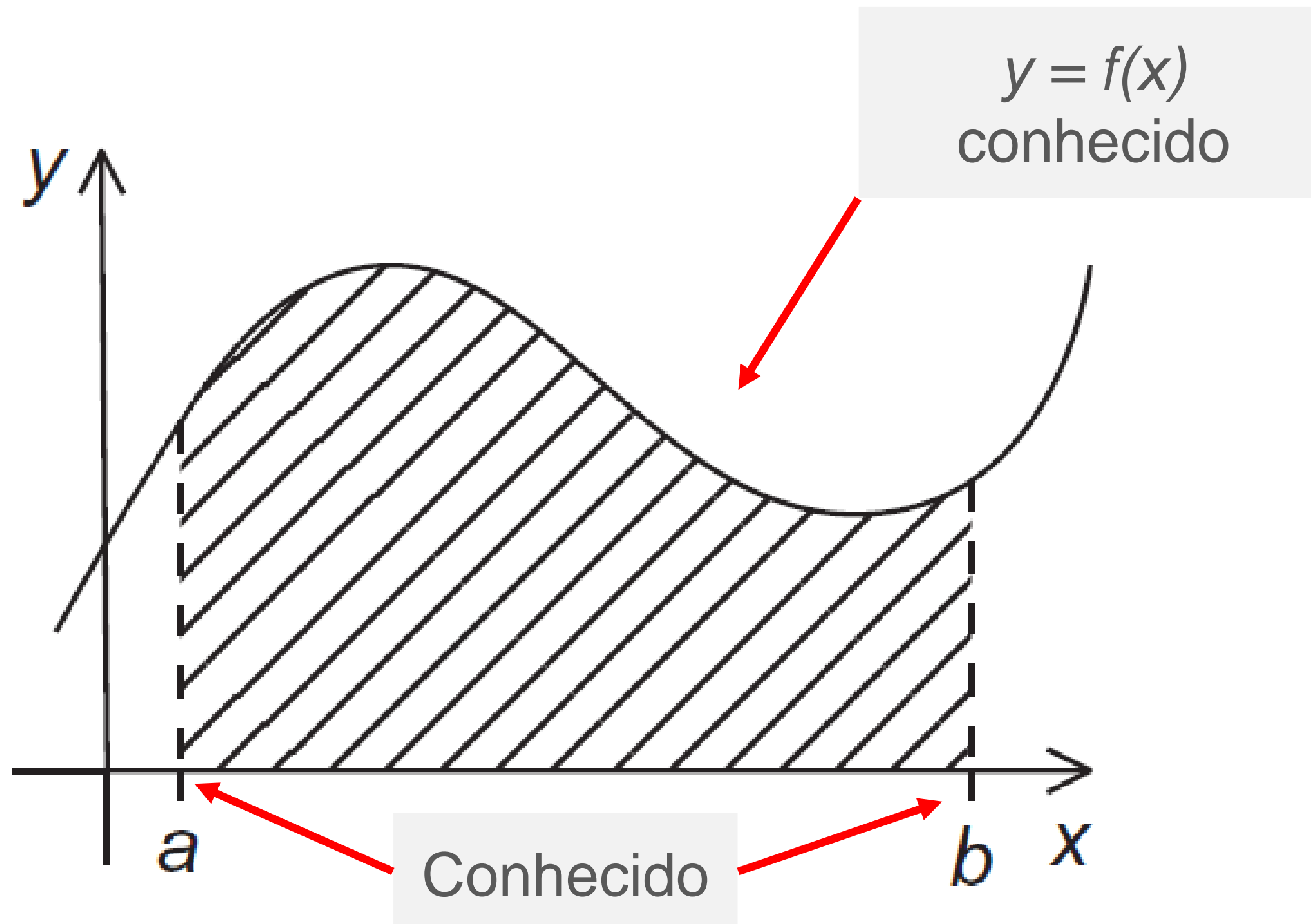


Cálculo Integral de uma função definida

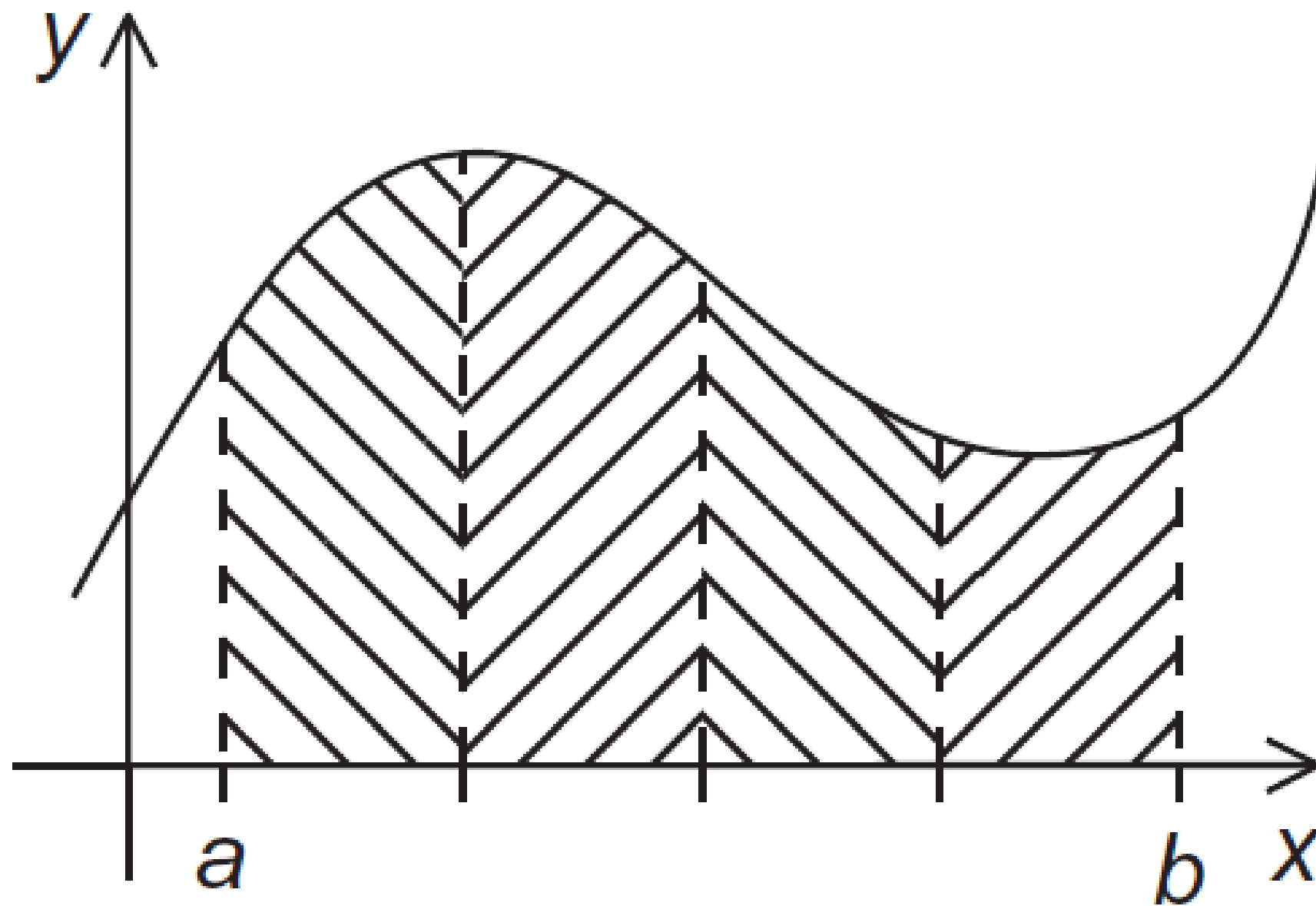
Regra dos Trapézios



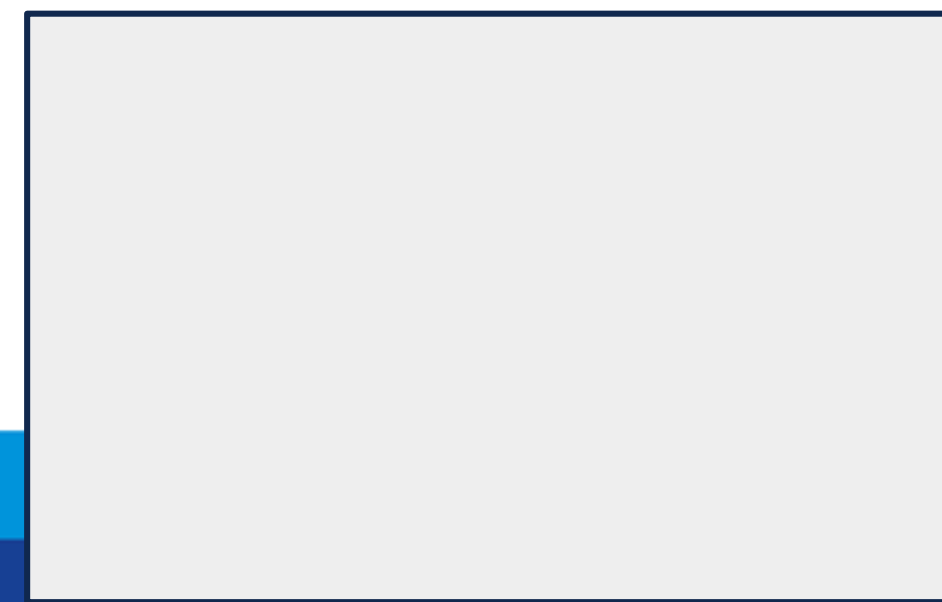
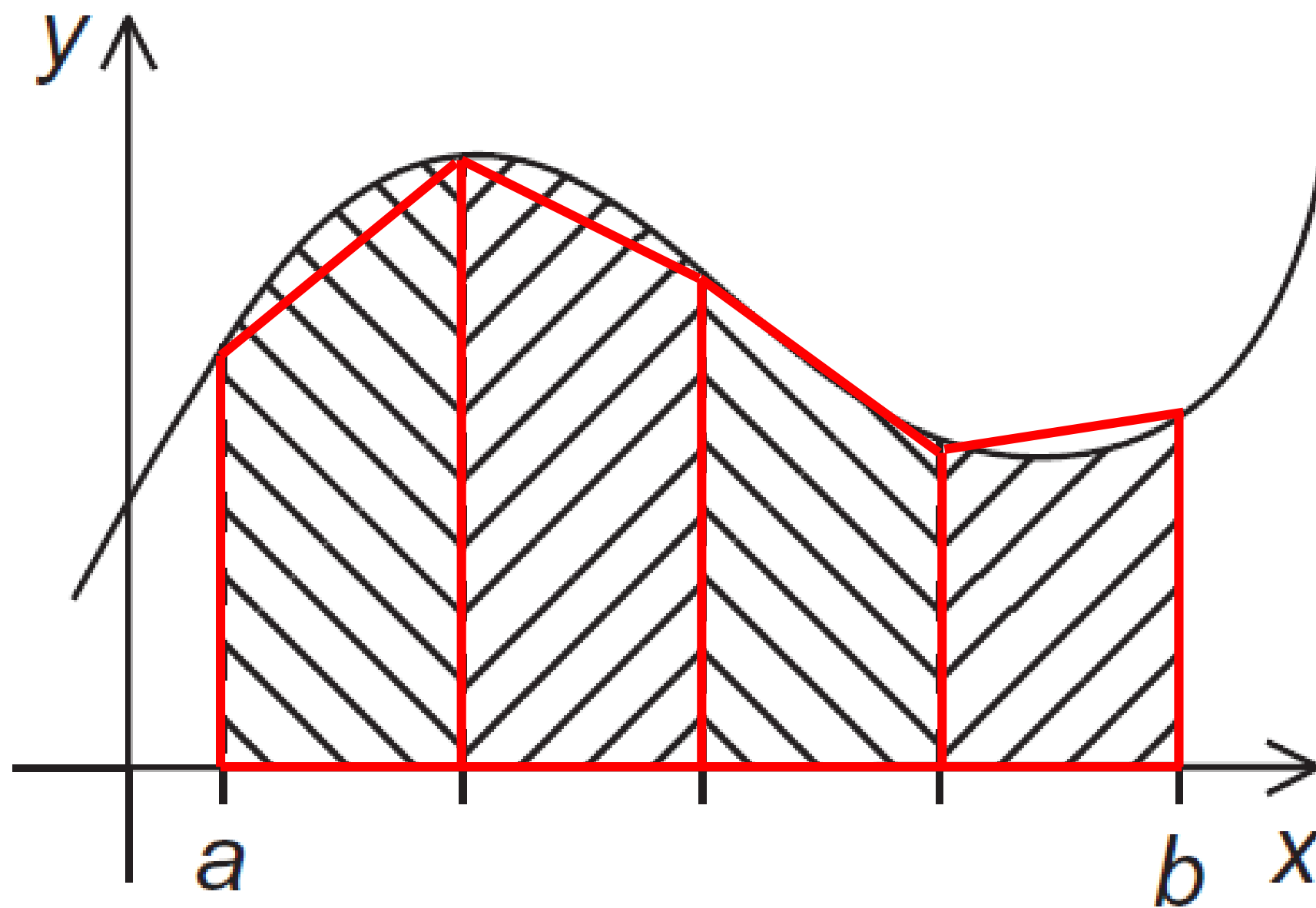
Regra dos Trapézios



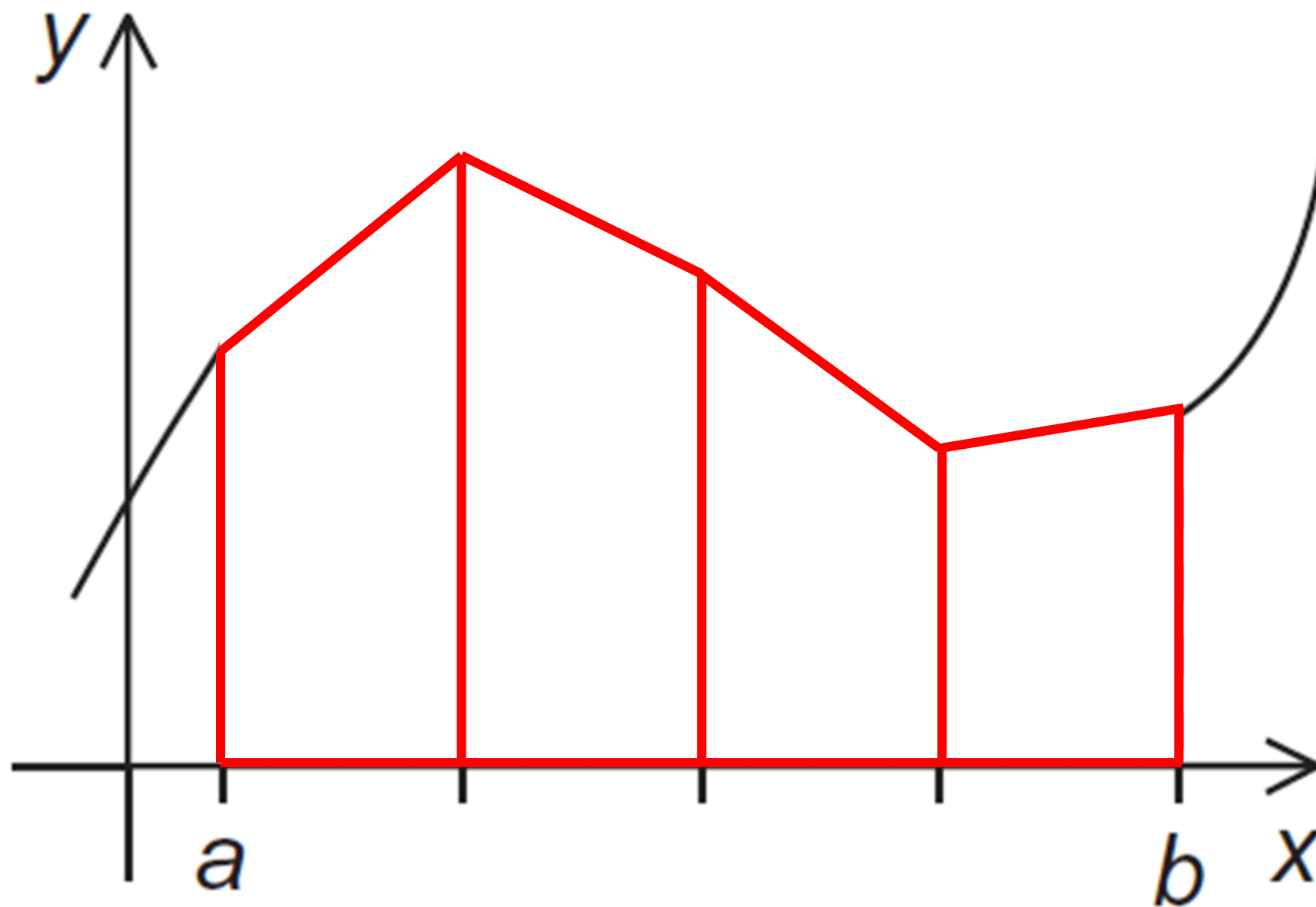
Regra dos Trapézios



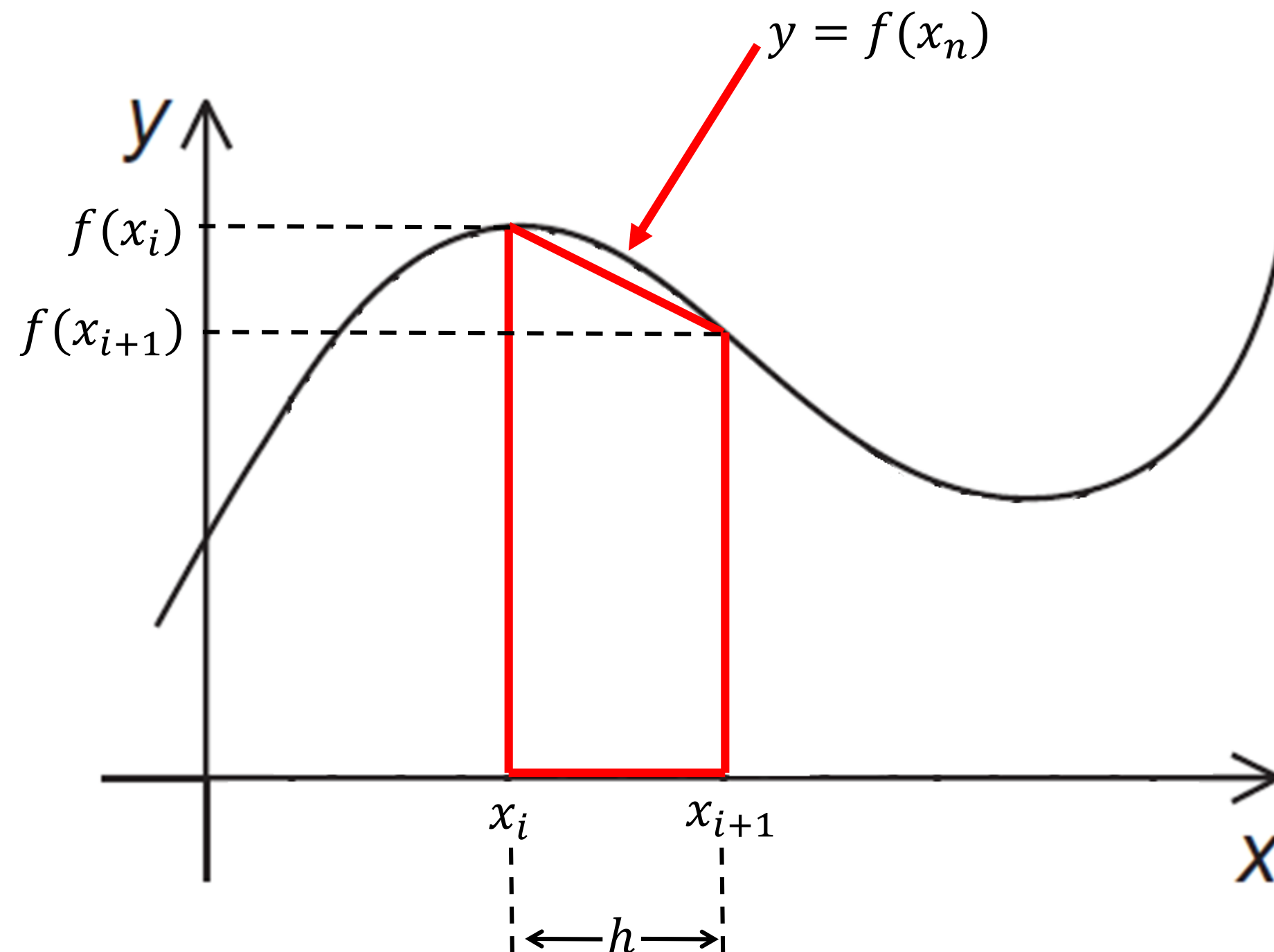
Regra dos Trapézios



Regra dos Trapézios



Regra dos Trapézios

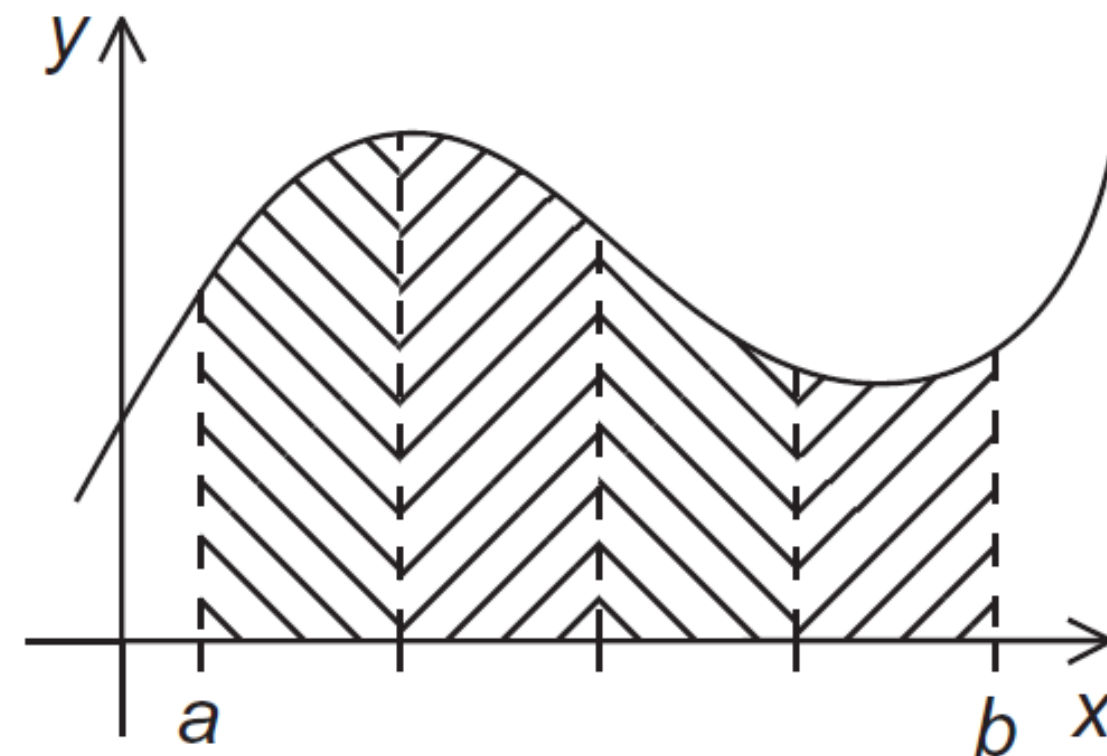


Regra dos Trapézios

$$\text{Área de um trapézio} = \frac{h}{2} [f(x_i) + f(x_{i+1})]$$

$$h = \frac{b - a}{n} \leftarrow \text{Nº de trapézios}$$

$$\text{Area de trapézio} = \frac{h}{2} (B + b)$$



$$x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h, \quad \dots \quad x_{n-1} = a + (n-1)h, \quad x_n = b$$

Solução 1



Regra dos Trapézios

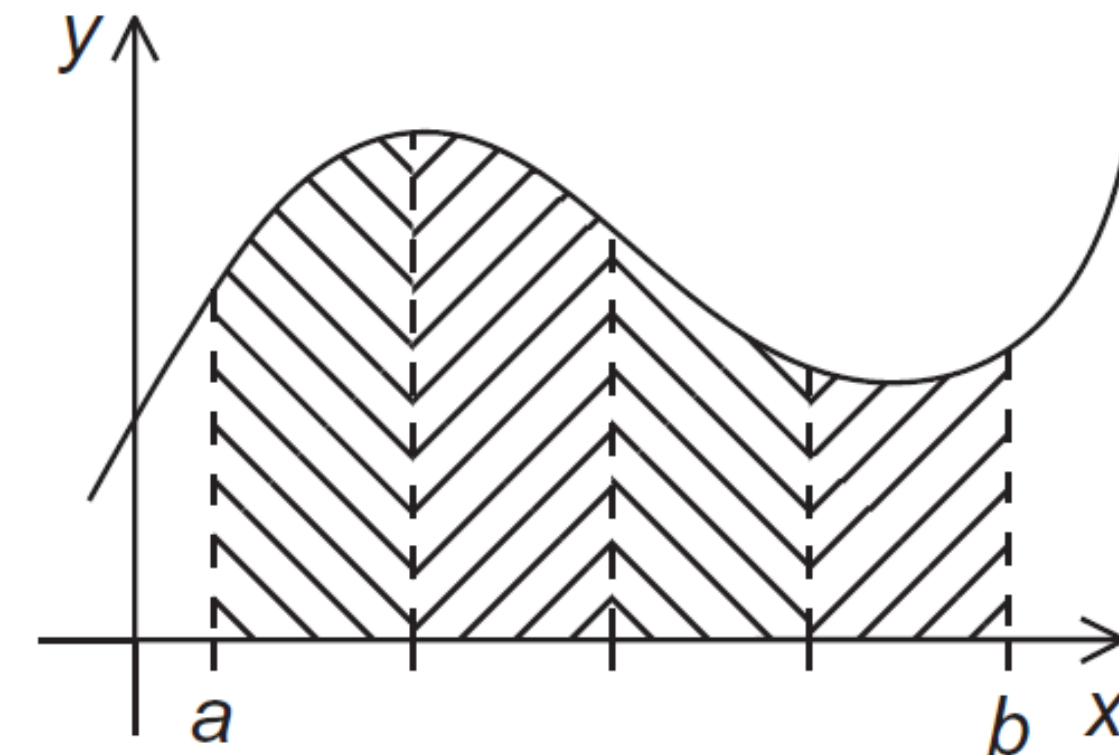
Solução 1

Considere que $f(x_n) = y_n$

$$\text{Área total} = \text{Trapézio}_1 + \text{Trapézio}_2 + \cdots + \text{Trapézio}_{n-1} + \text{Trapézio}_n$$

$$\text{Área total} = \frac{h}{2}[y_0 + y_1] + \frac{h}{2}[y_1 + y_2] + \frac{h}{2}[y_2 + y_3] + \cdots + \frac{h}{2}[y_{n-2} + y_{n-1}] + \frac{h}{2}[y_{n-1} + y_n]$$

$$\text{Area de trapézio} = \frac{h}{2}(B + b)$$



```
1 int main(void) {  
2  
3     h = (b-a)/n;  
4     area_total = (f(a)+f(a+h))*h/2;  
5     for (int i=1; i<n; i++) {  
6         x_i = a+i*h;  
7         area_total += (f(x_i) + f(x_i+h))*h/2;  
8     }  
9 }
```

Solução 2

Regra dos Trapézios

Solução 2

Considere que $f(x_n) = y_n$

$$\text{Área total} = \text{Trapézio}_1 + \text{Trapézio}_2 + \cdots + \text{Trapézio}_{n-1} + \text{Trapézio}_n$$

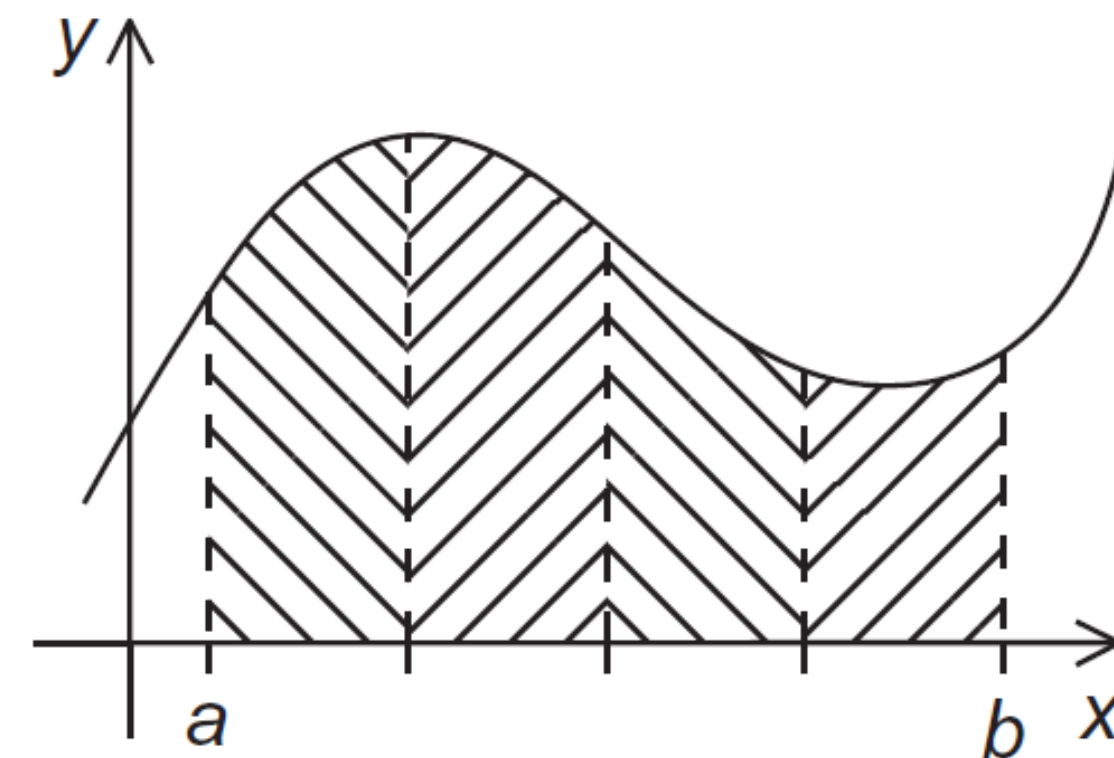
$$\text{Área total} = \frac{h}{2}[y_0 + y_1] + \frac{h}{2}[y_1 + y_2] + \frac{h}{2}[y_2 + y_3] + \cdots + \frac{h}{2}[y_{n-2} + y_{n-1}] + \frac{h}{2}[y_{n-1} + y_n]$$

$$\text{Área total} = \frac{h}{2}[y_0 + y_1 + y_1 + y_2 + y_2 + y_3 + \cdots + y_{n-2} + y_{n-1} + y_{n-1} + y_n]$$

$$\text{Área total} = \frac{h}{2}[y_0 + 2y_1 + 2y_2 + 2y_3 + \cdots + 2y_{n-1} + y_n]$$

$$\text{Área total} = h \left[\frac{y_0}{2} + y_1 + y_2 + y_3 + \cdots + y_{n-1} + \frac{y_n}{2} \right]$$

$$\text{Area de trapézio} = \frac{h}{2}(B + b)$$

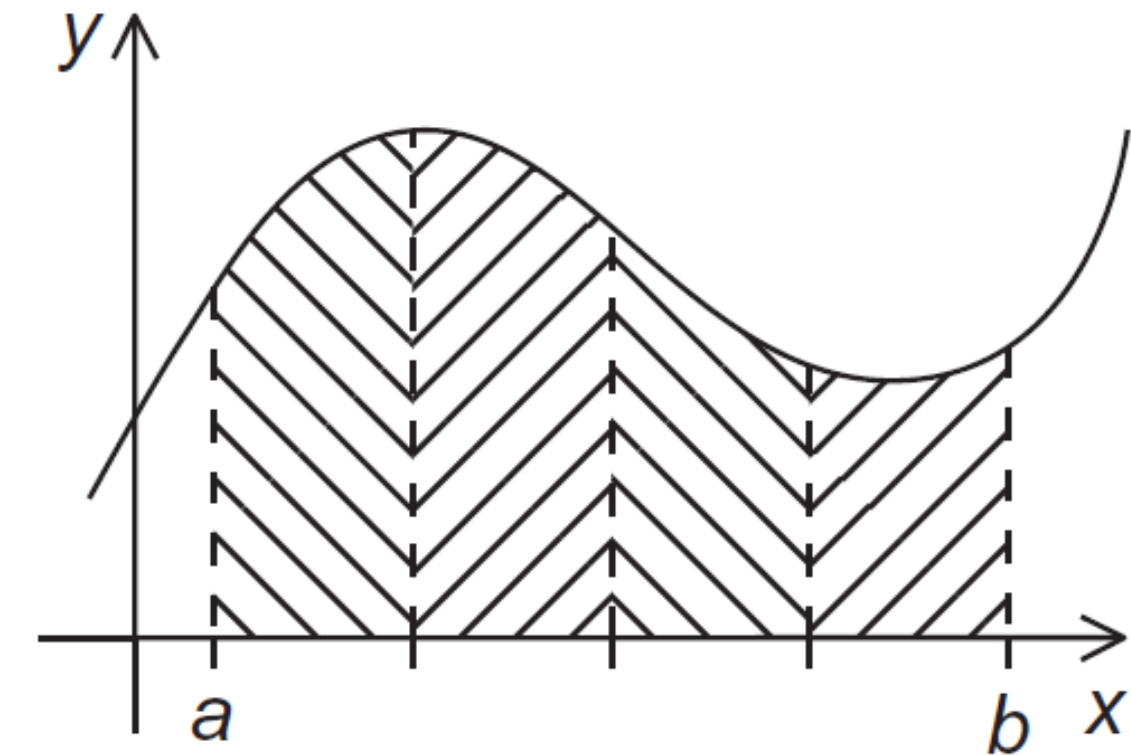


Regra dos Trapézios

Solução 2

$$\text{Área total} = h \left[\frac{y_0}{2} + y_1 + y_2 + y_3 + \cdots + y_{n-1} + \frac{y_n}{2} \right]$$

$$\text{Area de trapézio} = \frac{h}{2} (B + b)$$



```
1 int main(void) {  
2  
3     h = (b-a)/n;  
4     area_total = (f(a)+f(b))/2;  
5     for (int i=1; i<n; i++) {  
6         x_i = a+i*h;  
7         area_total += f(x_i);  
8     }  
9     area_total += h*area_total  
10 }
```

Lado a lado

Solução 1 & 2

Solução 1

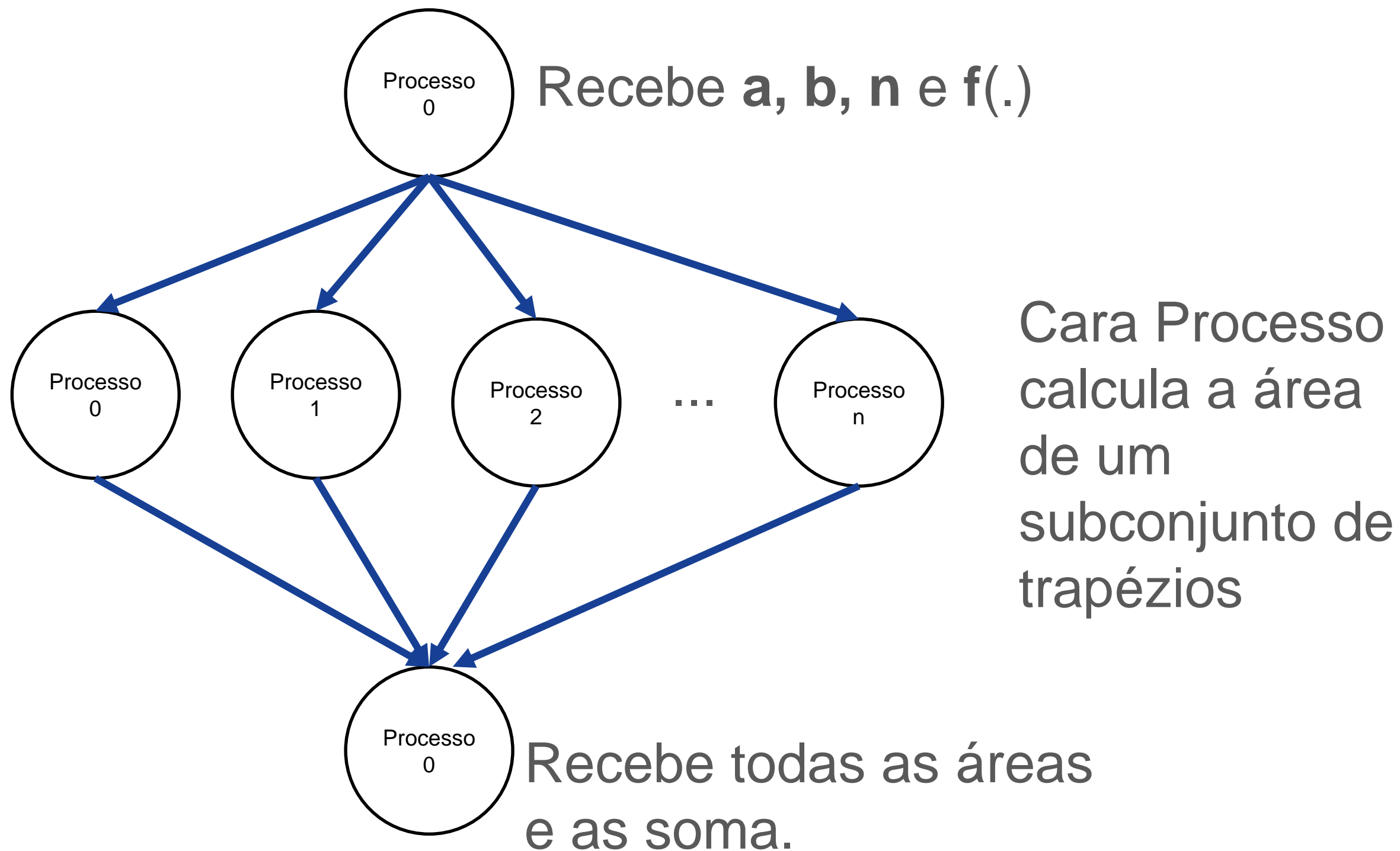
```
1 int main(void) {  
2  
3     h = (b-a)/n;  
4     area_total = (f(a)+f(a+h))*h/2;  
5     for (int i=1; i<n; i++) {  
6         x_i = a+i*h;  
7         area_total += (f(x_i) + f(x_i+h))*h/2;  
8     }  
9 }
```

Solução 2

```
1 int main(void) {  
2  
3     h = (b-a)/n;  
4     area_total = (f(a)+f(b))/2;  
5     for (int i=1; i<n; i++) {  
6         x_i = a+i*h;  
7         area_total += f(x_i);  
8     }  
9     area_total += h*area_total  
10 }
```

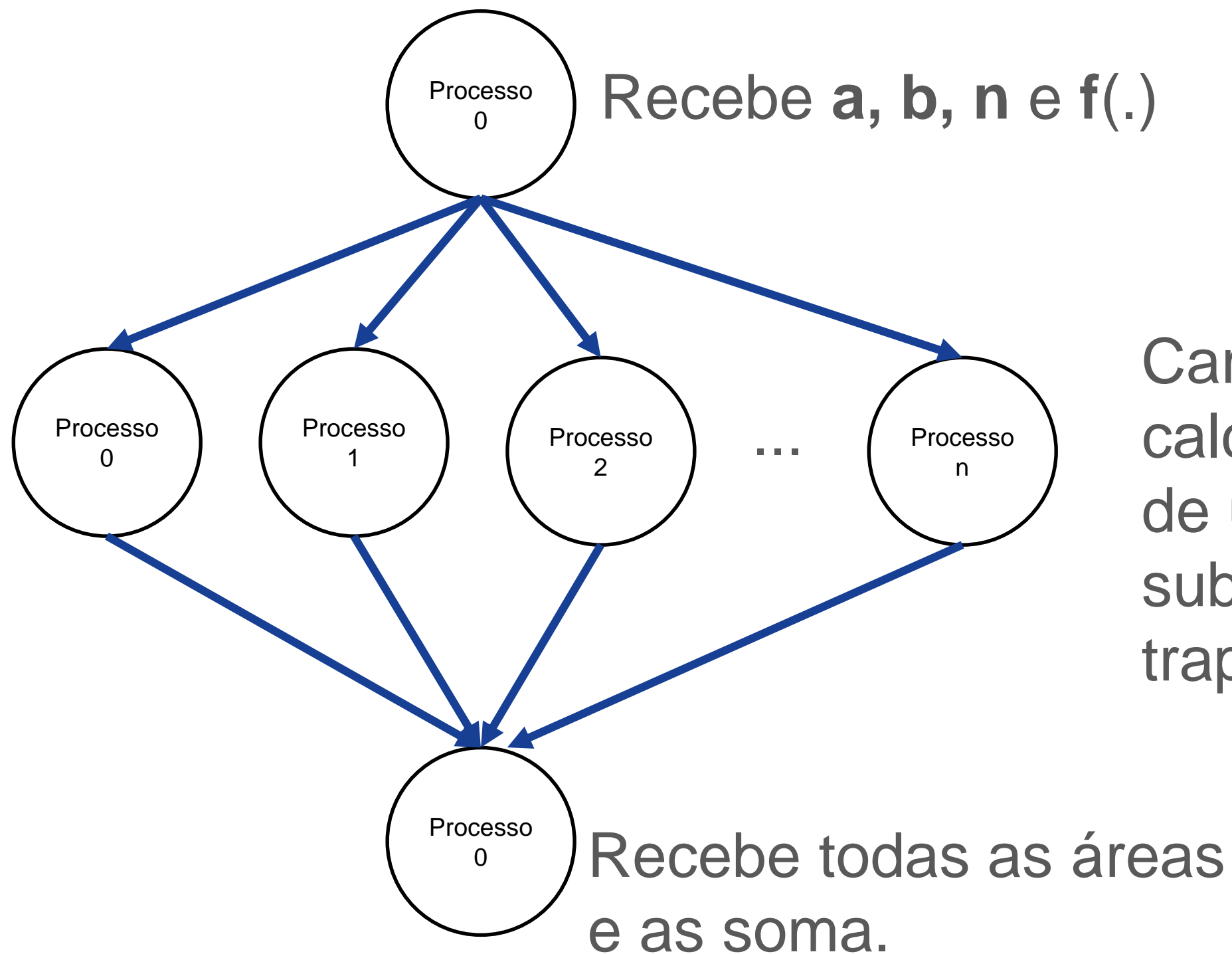
Qual delas executa menos operações?

Regra do Trapézio - Paralelo



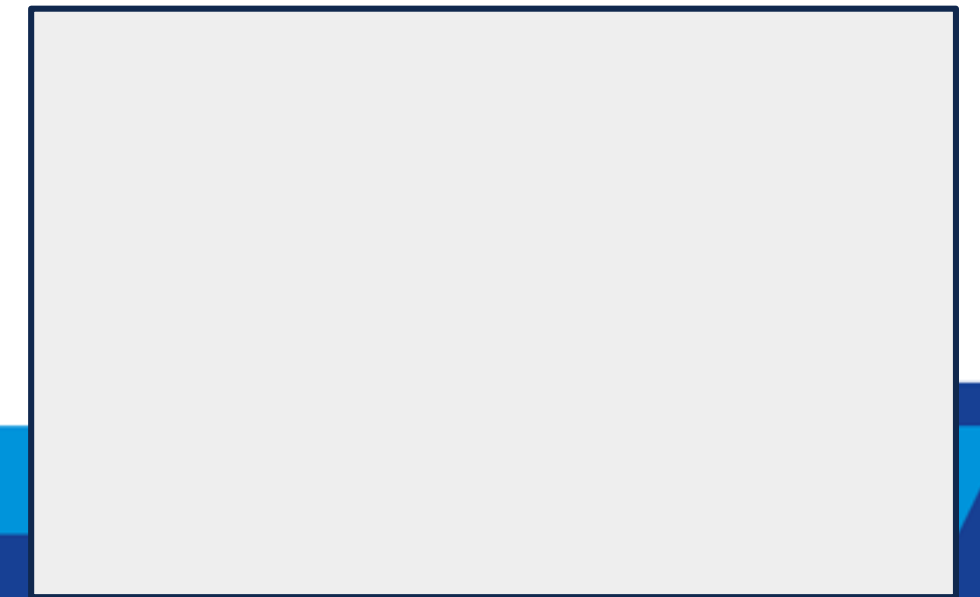
- Mestre não precisa informar quais ou quantos trapézios cada processo irá calcular a área.
- Os próprios processo calculam isso internamente.

Regra do Trapézio - Paralelo



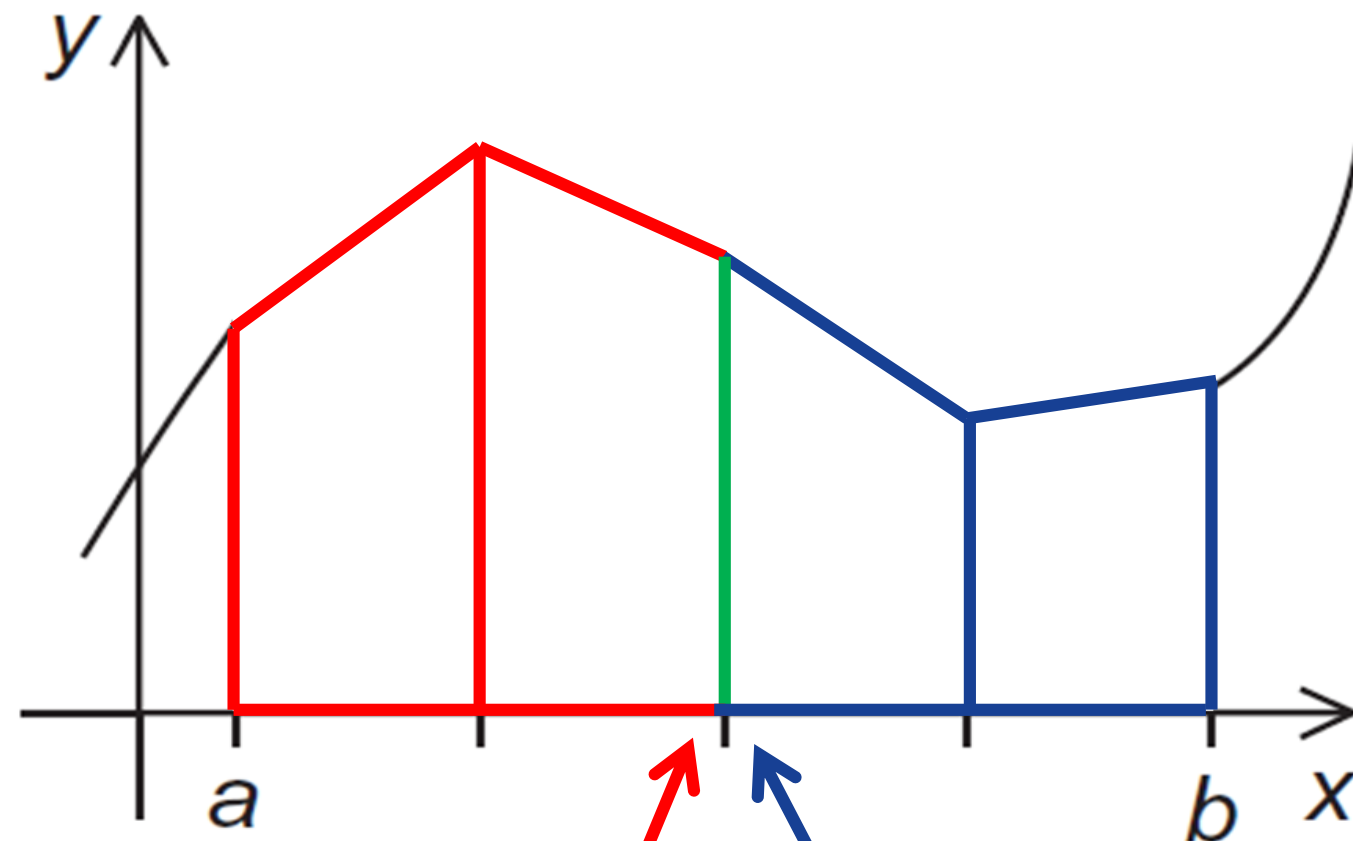
Cada Processo
calcula a área
de um
subconjunto de
trapézios

- Em cada processo:
a: local_a
b: local_b
n: local_n



Regra dos Trapézios Paralelo VISUAL

Considere 2 processos e 4 trapézios para o problema como todo.



Processo 0: local_a
Processo 0: local_b
Processo 0: local_n=2

Processo 1: local_b
Processo 1: local_a
Processo 1: local_n

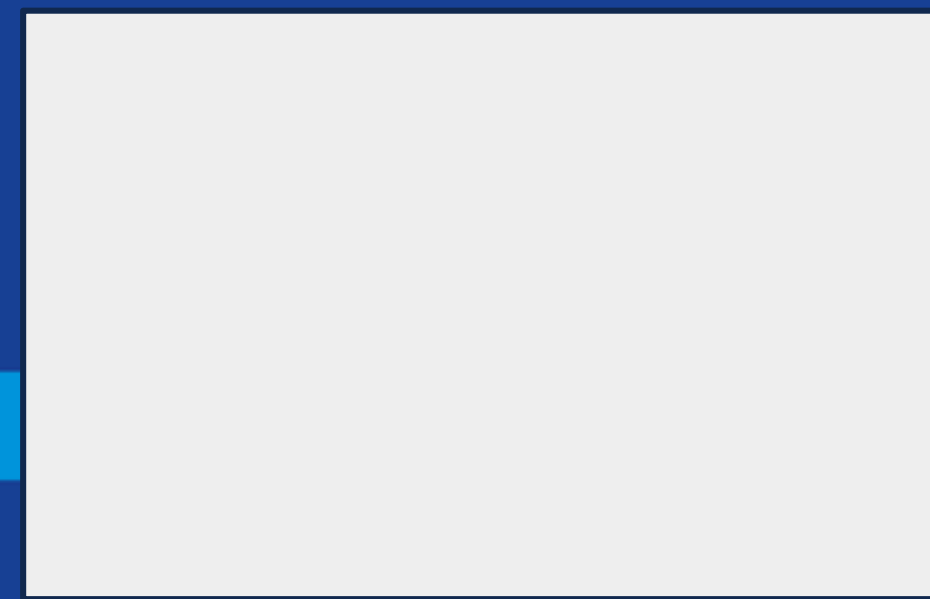
Pseudo-algoritmo Paralelo

Cada processo irá executar ...

```
1  Get a, b, n; ← Número de Trapézios
2  h = (b-a)/n;
3  local_n = n/comm_sz; ← Número de processos
4  local_a = a + my_rank*local_n*h;
5  local_b = local_a + local_n*h;
6  local_integral = Trap(local_a, local_b, local_n, h); ← Calcula área de um trapézio
7  if (my_rank != 0)
8      Send local_integral to process 0;
9  else /* my_rank == 0 */
10     total_integral = local_integral;
11     for (proc = 1; proc < comm_sz; proc++) {
12         Receive local_integral from proc;
13         total_integral += local_integral;
14     }
15 }
16 if (my_rank == 0)
17     print result;
```

Lidando com

Entrada e Saída



Saída

- Qualquer processo pode imprimir na tela.
- Após um **printf**, é aconselhável utilizar um **fflush(stdout)**.
- Printf trabalha com um buffer. Somente imprime na tela quando o buffer está cheio.
- **fflush(stdout)** força a impressão na tela mesmo com o buffer parcialmente preenchido.
- Não-determinística



Entrada

- Maioria das implementações MPI permitem apenas que o processo 0 no `MPI_COMM_WORLD` acesse o stdin.
- O Processo 0 deve ler os dados (scanf) e enviá-los aos demais processos.

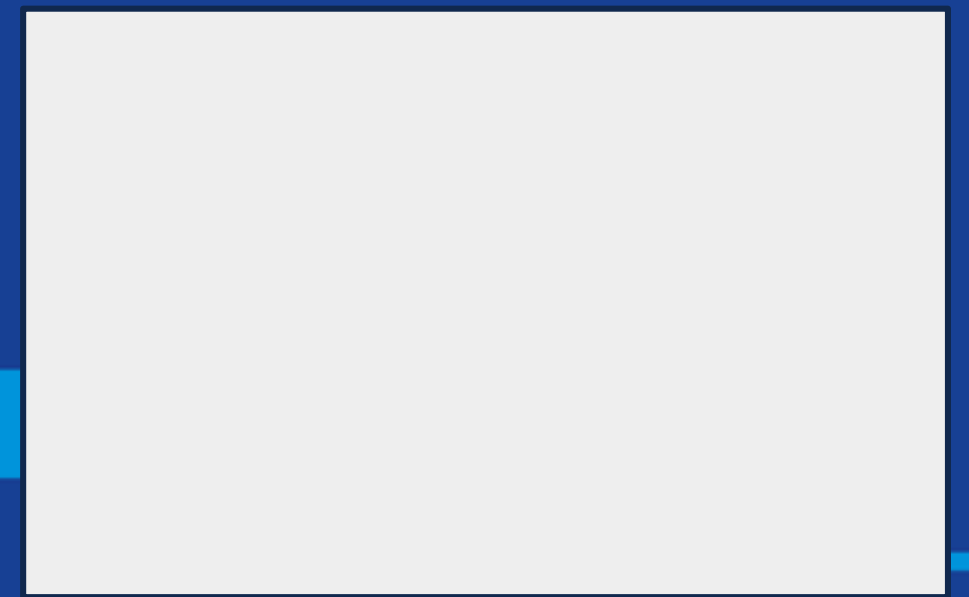


Entrada

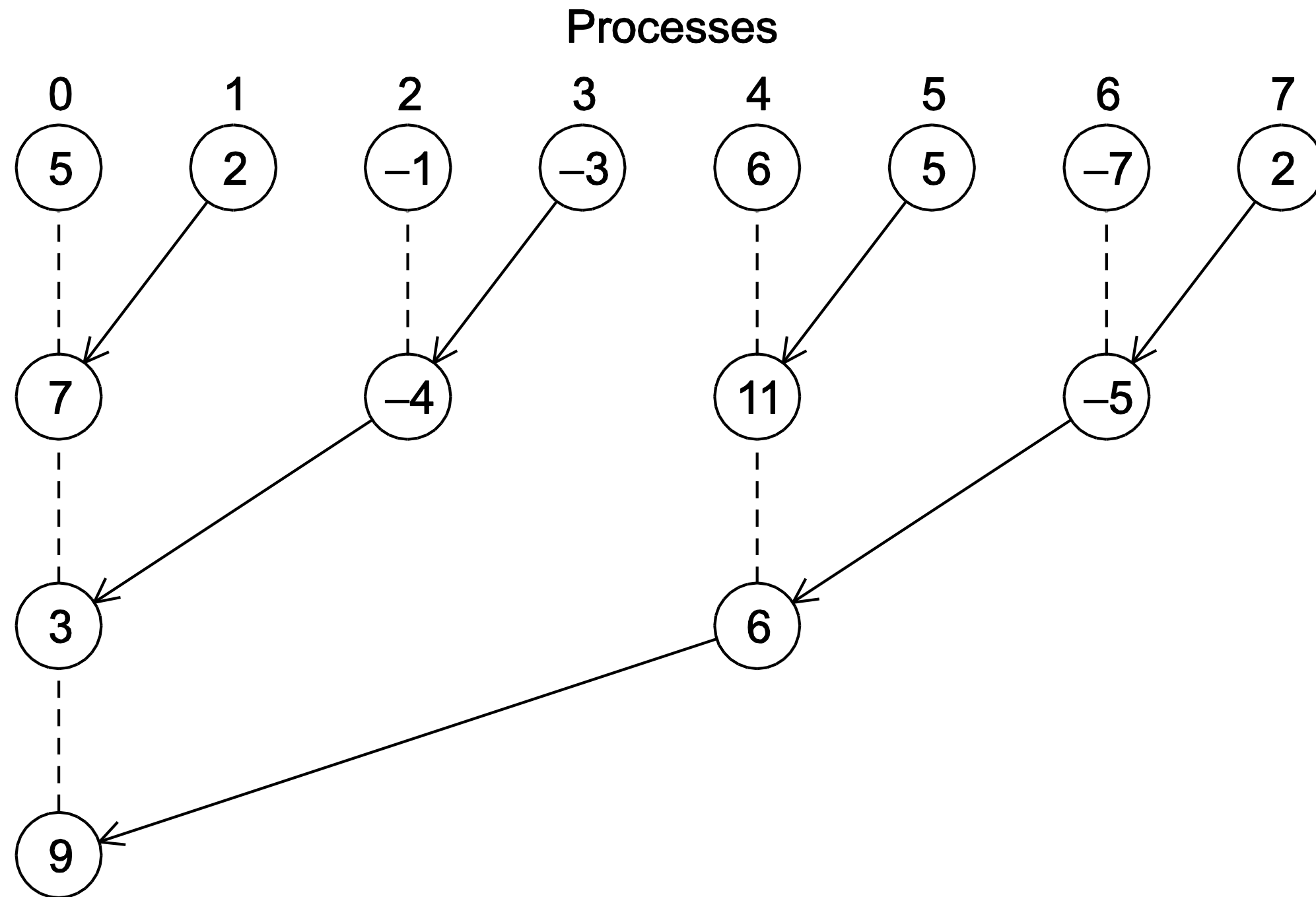
```
void Get_input(  
    int      my_rank    /* in */,  
    int      comm_sz    /* in */,  
    double*  a_p        /* out */,  
    double*  b_p        /* out */,  
    int*     n_p        /* out */) {  
    int dest;  
  
    if (my_rank == 0) {  
        printf("Enter a, b, and n\n");  
        scanf("%lf %lf %d", a_p, b_p, n_p);  
        for (dest = 1; dest < comm_sz; dest++) {  
            MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);  
            MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);  
            MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);  
        }  
    } else { /* my_rank != 0 */  
        MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,  
                MPI_STATUS_IGNORE);  
        MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,  
                MPI_STATUS_IGNORE);  
        MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,  
                MPI_STATUS_IGNORE);  
    }  
} /* Get_input */
```

Roger
That!

Comunicações Coletivas

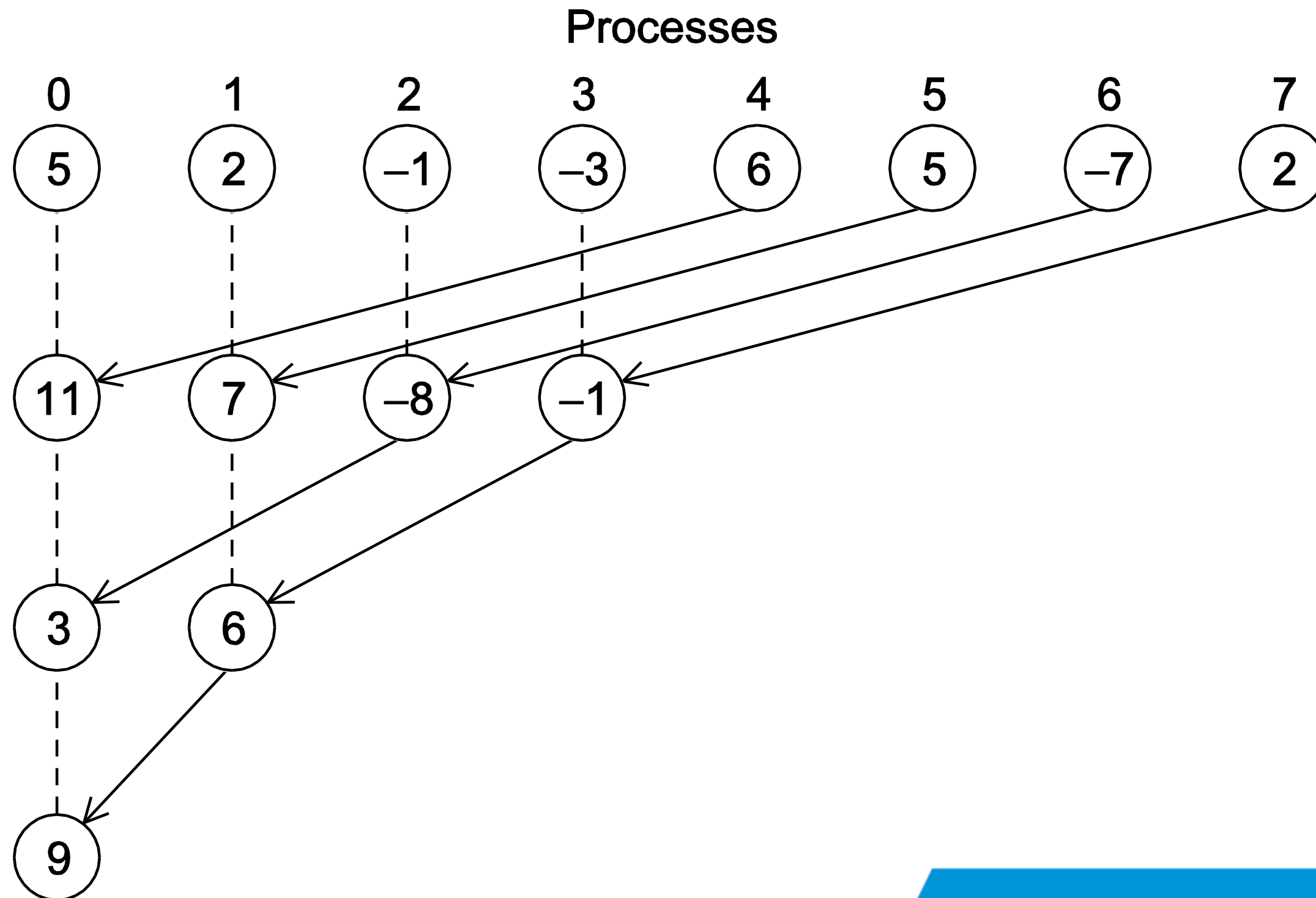


Relembrando...



Soma global em
estrutura de árvore

Uma alternativa...



Uma alternativa de
soma global em
estrutura de árvore

MPI_Reduce (1)

```
int MPI_Reduce(  
    void*      input_data_p    /* in  */,  
    void*      output_data_p   /* out */,  
    int        count           /* in  */,  
    MPI_Datatype datatype      /* in  */,  
    MPI_Op      operator       /* in  */,  
    int        dest_process    /* in  */,  
    MPI_Comm     comm          /* in  */);
```

Vetor de elementos
do tipo **datatype**
que queremos
reduzir

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

MPI_Reduce (2)

```
int MPI_Reduce(  
    void*      input_data_p    /* in  */,  
    void*      output_data_p   /* out */,  
    int        count           /* in  */,  
    MPI_Datatype datatype      /* in  */,  
    MPI_Op      operator       /* in  */,  
    int        dest_process    /* in  */,  
    MPI_Comm     comm          /* in  */);
```

Relevante somente para o **root**.

Vetor que contém o resultado reduzido e tem tamanho de **sizeof(datatype)*count**

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```


MPI_Reduce (3)

```
int MPI_Reduce(  
    void*      input_data_p    /* in  */,  
    void*      output_data_p   /* out */,  
    int        count           /* in  */,  
    MPI_Datatype datatype      /* in  */,  
    MPI_Op      operator       /* in  */,  
    int        dest_process    /* in  */,  
    MPI_Comm     comm          /* in  */);
```

Número de elementos
do vetor que queremos
reduzir

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

MPI_Reduce (4)

```
int MPI_Reduce(  
    void*      input_data_p    /* in  */,  
    void*      output_data_p  /* out */,  
    int        count          /* in  */,  
    MPI_Datatype datatype      /* in  */,  
    MPI_Op      operator       /* in  */,  
    int        dest_process    /* in  */,  
    MPI_Comm    comm           /* in  */);
```

Tipo de redução:
- É tabelado

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

MPI_Reduce (4.1)

Tipos de redução

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Opções



MPI_Reduce (5)

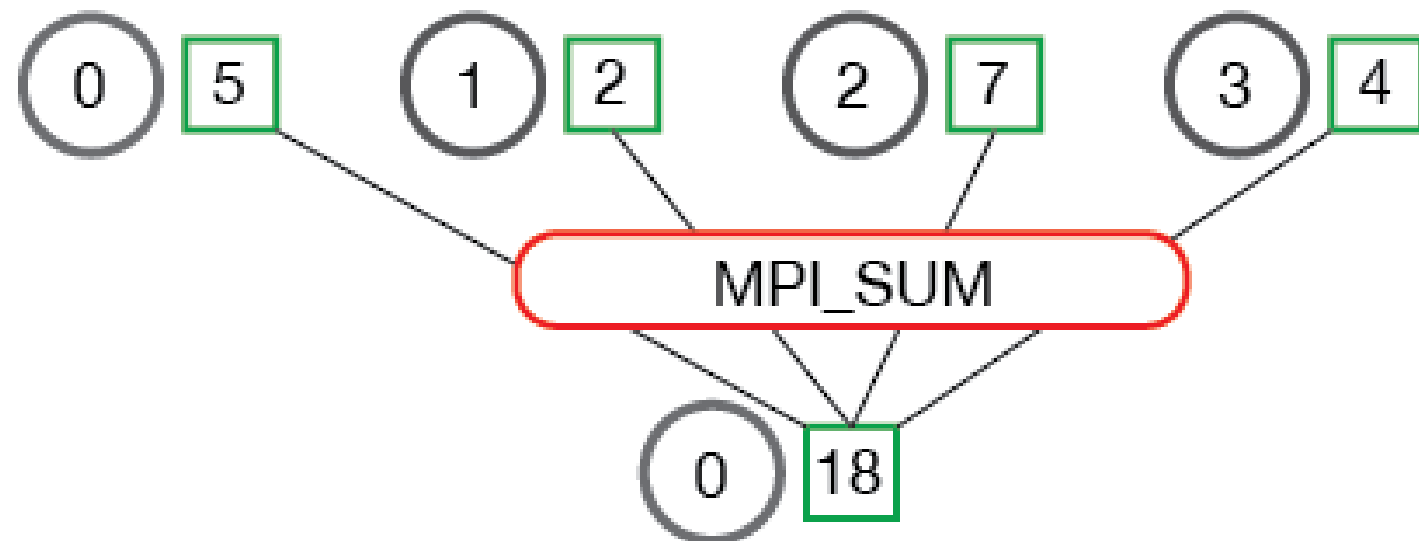
```
int MPI_Reduce(  
    void*      input_data_p    /* in  */,  
    void*      output_data_p   /* out */,  
    int        count           /* in  */,  
    MPI_Datatype datatype       /* in  */,  
    MPI_Op      operator        /* in  */,  
    int        dest_process    /* in  */,  
    MPI_Comm     comm          /* in  */);
```

Número do processo
que será o **root**
Comunicador

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

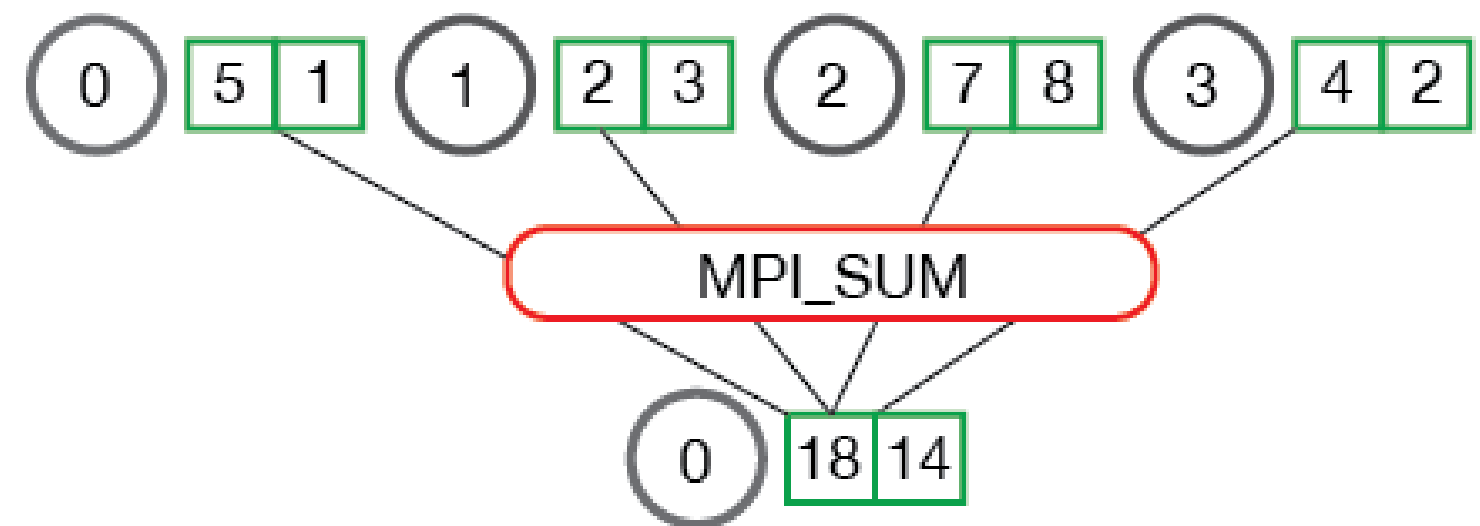
MPI_Reduce (6)

MPI_Reduce



Um elemento

MPI_Reduce



Vetor de elementos

Considerações (1)

Comunicação coletiva

- **Todos** os processos no mesmo comunicador **devem** chamar a mesma função coletiva
 - Por exemplo, um programa tenta corresponder a chamada de MPI_Reduce em um processo com a chamada de MPI_Recv em um outro processo. É provável que o programa trave.
- Os argumentos passados por cada processo em comunicação coletiva devem ser compatíveis.
 - Por exemplo, se um processo passa 0 como destino e outro processo passa 1, é provável que o programa trave.



Considerações (2)

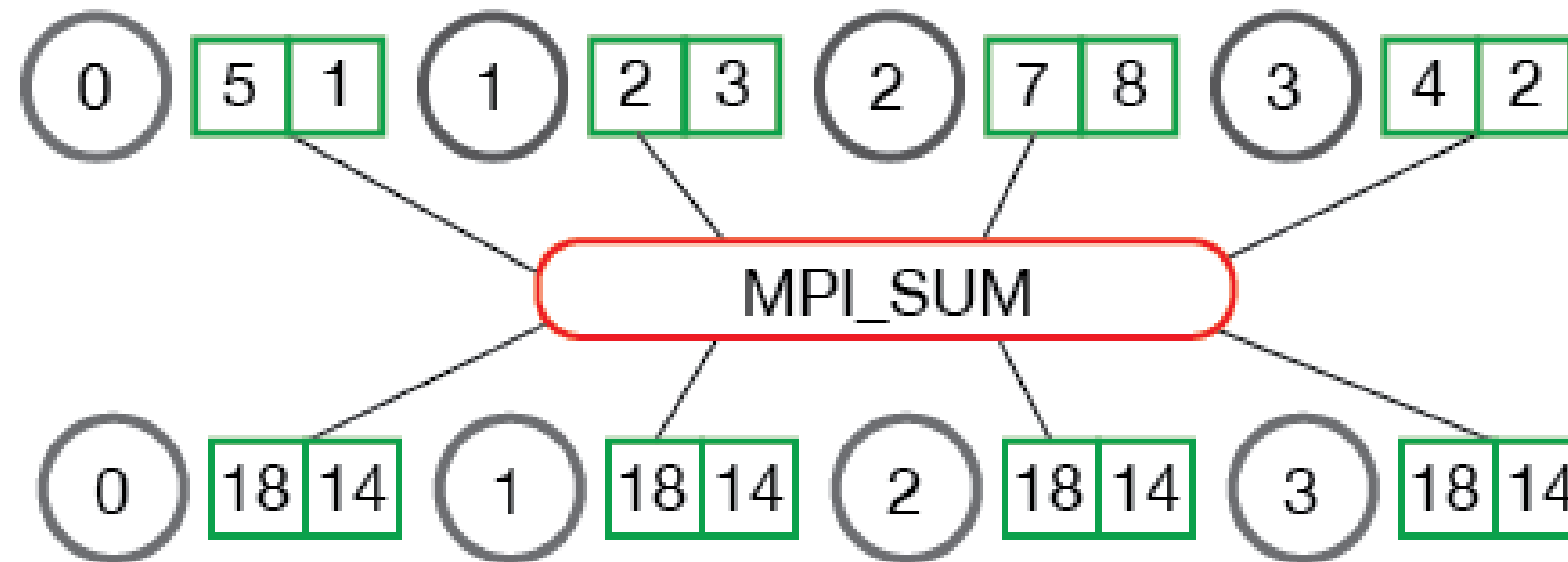
Comunicação coletiva

- O argumento **output_data_p** é somente usado no **root**
 - Entretanto, todos os processo ainda devem passa-lo como arguemnento.
- Comunicação coletiva não usa tag.
 - Já comunicação P2P é baseado tanto no comunicador quanto a tag.



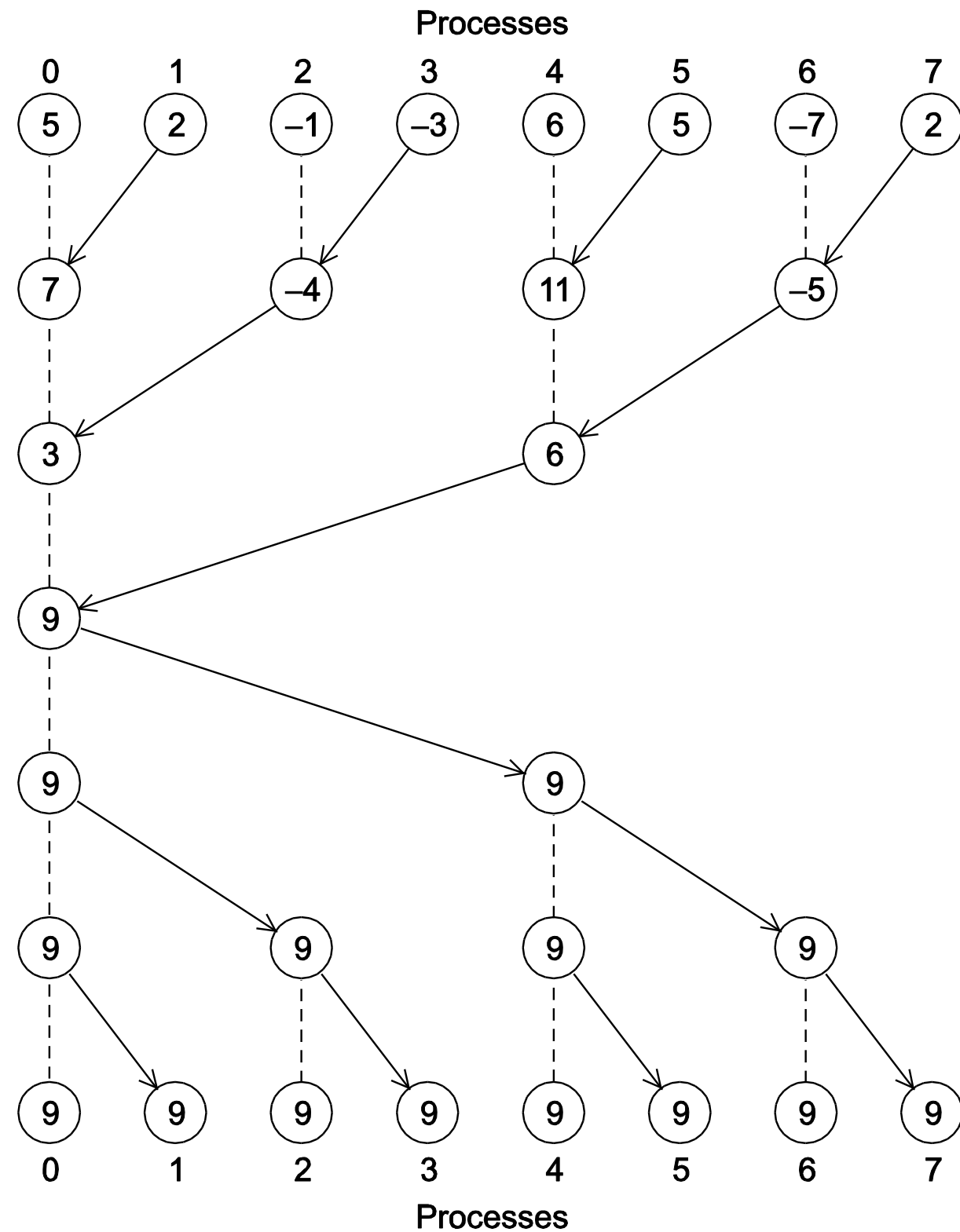
MPI_Allreduce (1)

MPI_Allreduce



```
int MPI_Allreduce(  
    void*      input_data_p    /* in */,  
    void*      output_data_p   /* out */,  
    int        count           /* in */,  
    MPI_Datatype datatype      /* in */,  
    MPI_Op      operator       /* in */,  
    MPI_Comm    comm           /* in */);
```

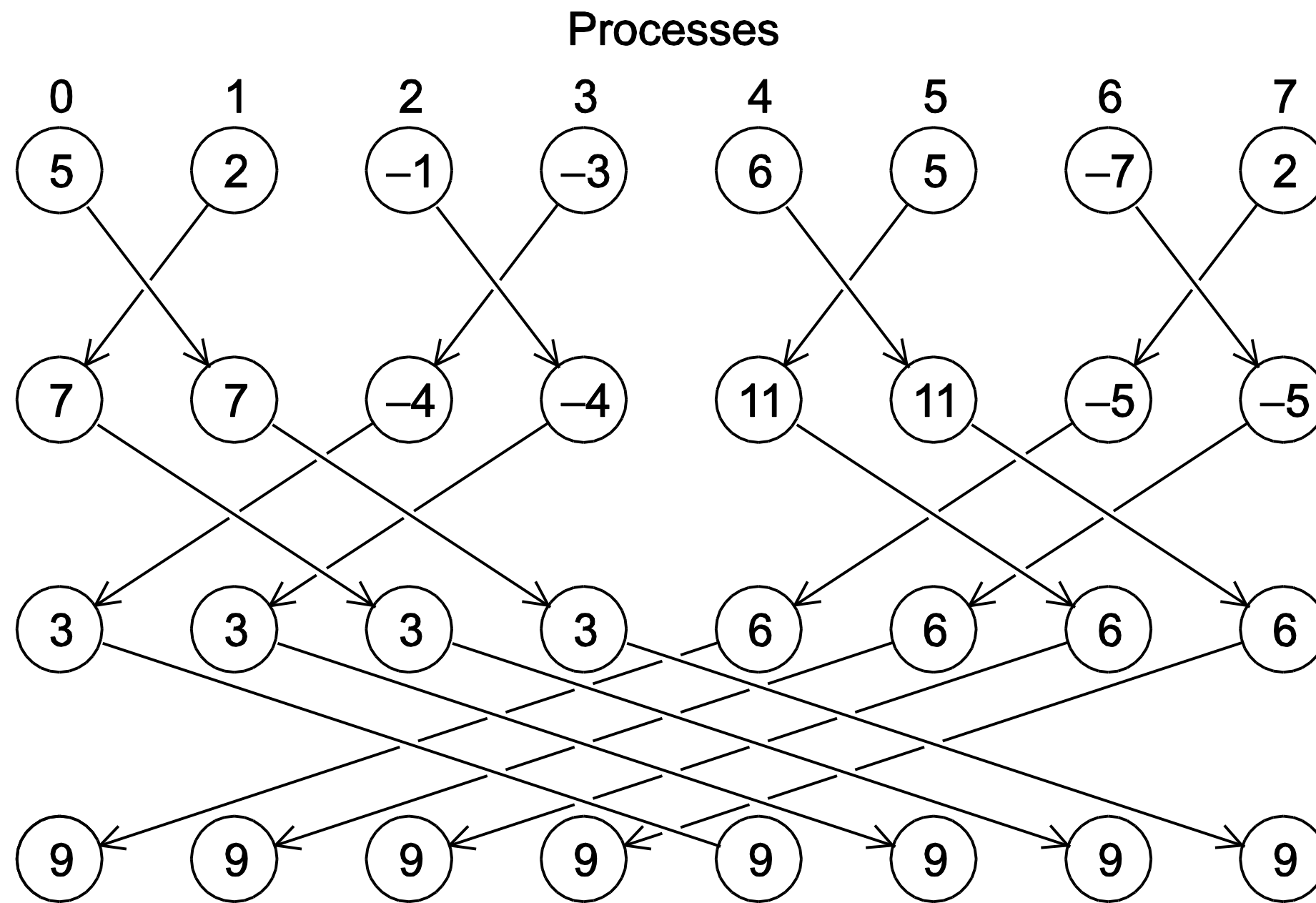

MPI_Allreduce (2)



É equivalente a uma soma global seguida de uma distribuição dos resultados



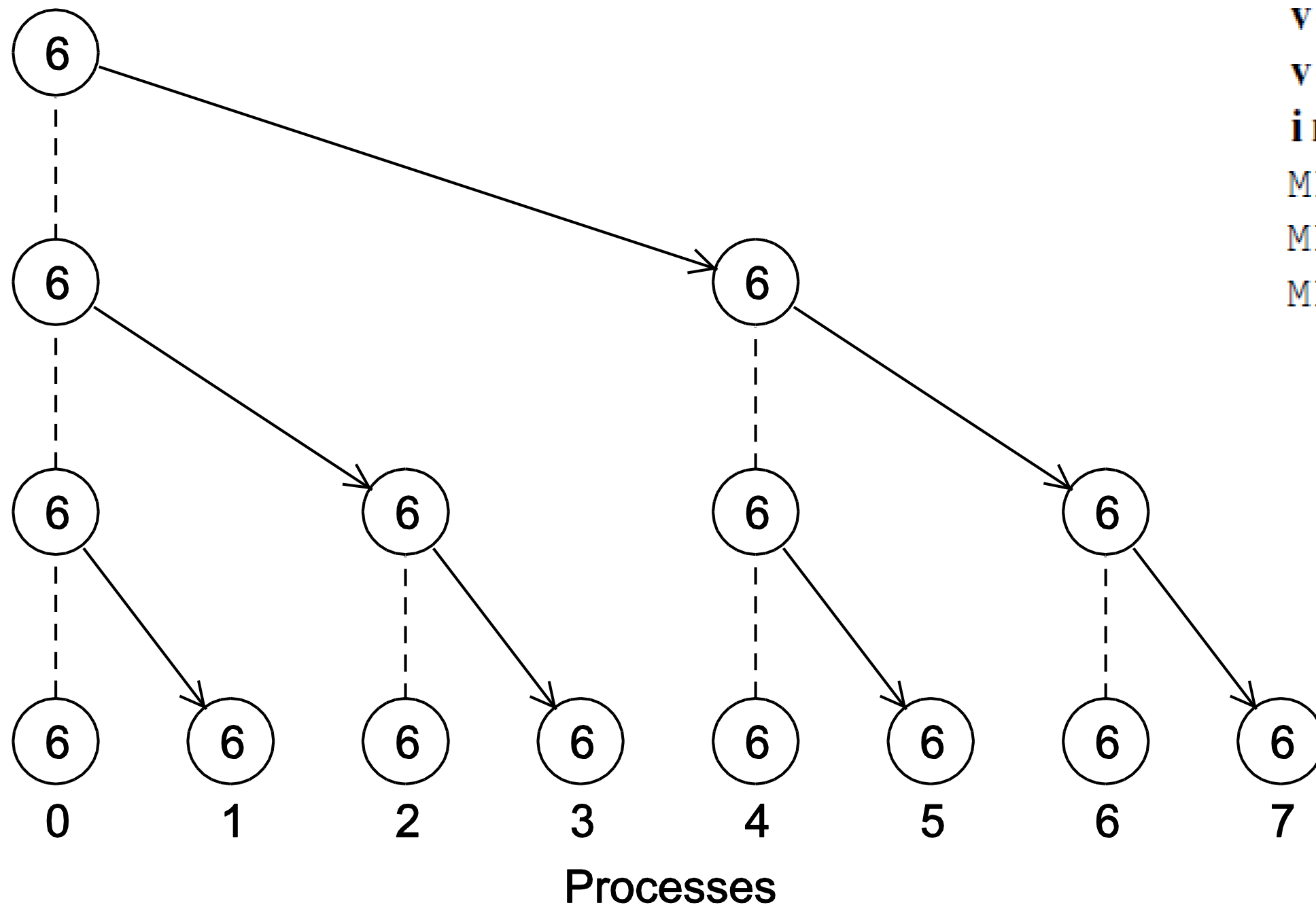
MPI_Allreduce (3)



Soma global em estrutura de borboleta



MPI_Bcast (1)



```
int MPI_Allreduce(  
    void* input_data_p /* in */,  
    void* output_data_p /* out */,  
    int count /* in */,  
    MPI_Datatype datatype /* in */,  
    MPI_Op operator /* in */,  
    MPI_Comm comm /* in */);
```

MPI_Bcast (2)

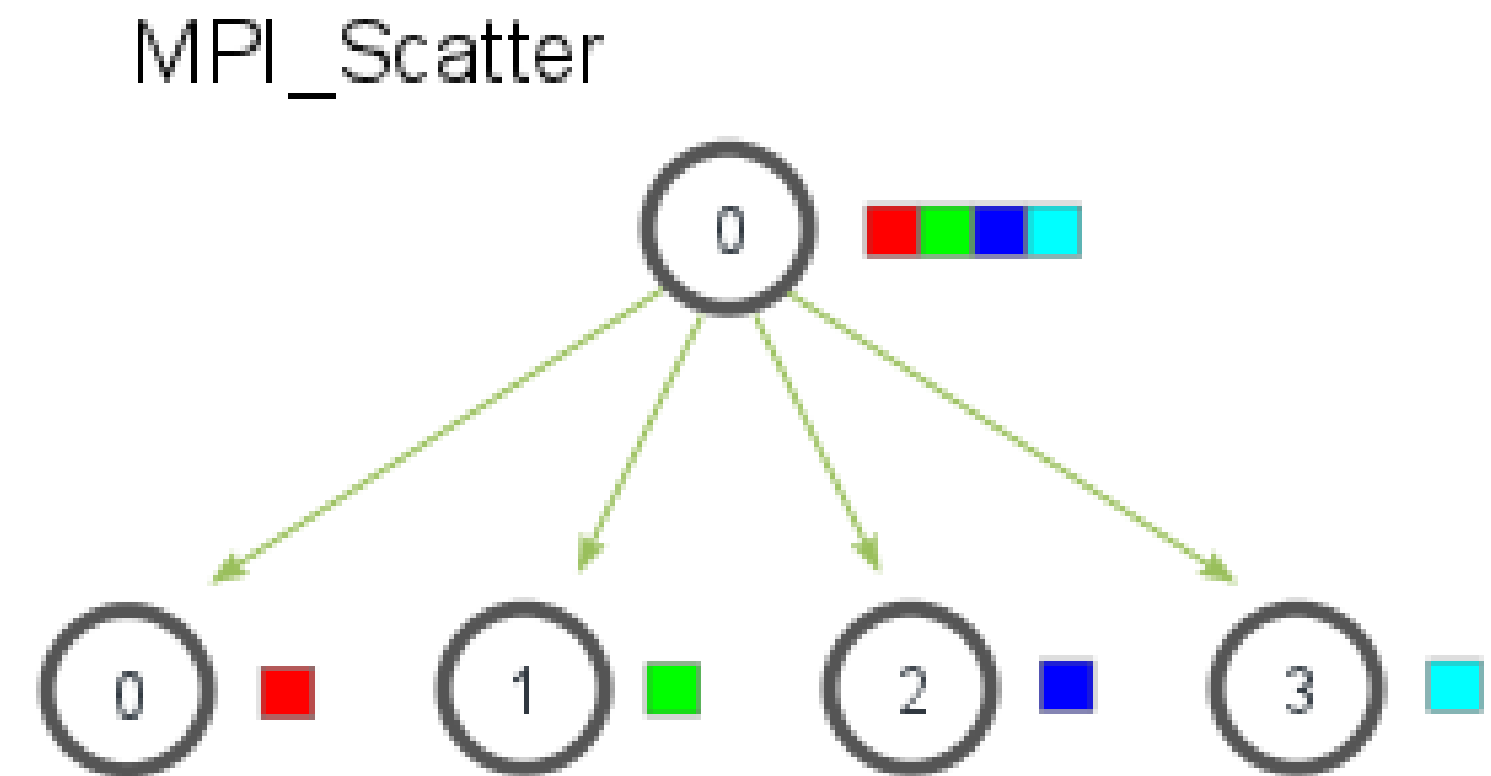
Exemplo – Processo 0 envia 3 dados para todos os demais processos

```
void Get_input(  
    int      my_rank    /* in  */,  
    int      comm_sz    /* in  */,  
    double*  a_p        /* out */,  
    double*  b_p        /* out */,  
    int*     n_p        /* out */) {  
  
    if (my_rank == 0) {  
        printf("Enter a, b, and n\n");  
        scanf("%lf %lf %d", a_p, b_p, n_p);  
    }  
  
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);  
} /* Get_input */
```

Nível de dificuldade:
Goiabinha!

MPI_Scatter (1)

```
int MPI_Scatter(  
    void*      send_buf_p    /* in */,  
    int       send_count    /* in */,  
    MPI_Datatype send_type   /* in */,  
    void*      recv_buf_p    /* out */,  
    int       recv_count    /* in */,  
    MPI_Datatype recv_type   /* in */,  
    int       src_proc       /* in */,  
    MPI_Comm   comm          /* in */);
```



MPI_Scatter (2)

```
int MPI_Scatter(  
    void*      send_buf_p  /* in  */,  
    int        send_count  /* in  */,  
    MPI_Datatype send_type  /* in  */,  
    void*      recv_buf_p  /* out */,  
    int        recv_count  /* in  */,  
    MPI_Datatype recv_type  /* in  */,  
    int        src_proc    /* in  */,  
    MPI_Comm    comm       /* in  */);
```

← Vetor de dados que reside no **root**



MPI_Scatter (3)

```
int MPI_Scatter(  
    void*      send_buf_p  /* in  */,  
    int        send_count  /* in  */,  
    MPI_Datatype send_type  /* in  */,  
    void*      recv_buf_p  /* out */,  
    int        recv_count  /* in  */,  
    MPI_Datatype recv_type  /* in  */,  
    int        src_proc    /* in  */,  
    MPI_Comm    comm       /* in  */);
```

Quantos elementos do **root** será enviado para aos processos.

Normalmente utilizado como número de elementos de **send_buf_p** dividido pelo número de processos.

- **send_count=1**, então **send_buf_p[0]** vai p/ processo₀, **array[1]** p/ processo₁, etc.
- **send_count=2**, então **send_buf_p[0-1]** vai p/ processo₀, **send_buf_p[2-3]** p/ processo₁, etc.

MPI_Scatter (4)

```
int MPI_Scatter(  
    void*      send_buf_p  /* in  */,  
    int        send_count  /* in  */,  
    MPI_Datatype send_type  /* in  */,  
    void*      recv_buf_p  /* out */,  
    int        recv_count  /* in  */,  
    MPI_Datatype recv_type  /* in  */,  
    int        src_proc     /* in  */,  
    MPI_Comm    comm        /* in  */);
```

Vetor de dados que
receberá os dados

Nº de elementos que
receberá. Utilize igual
ao **send_count**

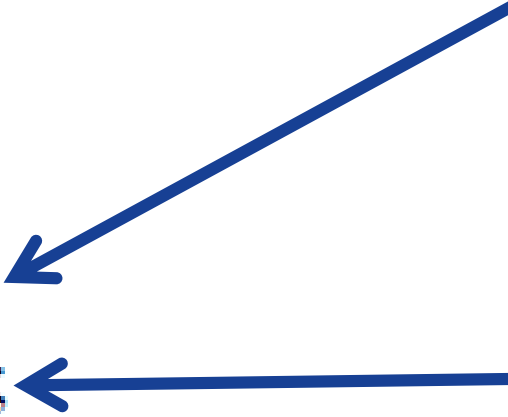


MPI_Scatter (5)

```
int MPI_Scatter(  
    void*      send_buf_p  /* in */,  
    int        send_count  /* in */,  
    MPI_Datatype send_type  /* in */,  
    void*      recv_buf_p  /* out */,  
    int        recv_count  /* in */,  
    MPI_Datatype recv_type  /* in */,  
    int        src_proc     /* in */,  
    MPI_Comm    comm        /* in */);
```

Número do processo
root

Comunicador



MPI_Scatter (6)

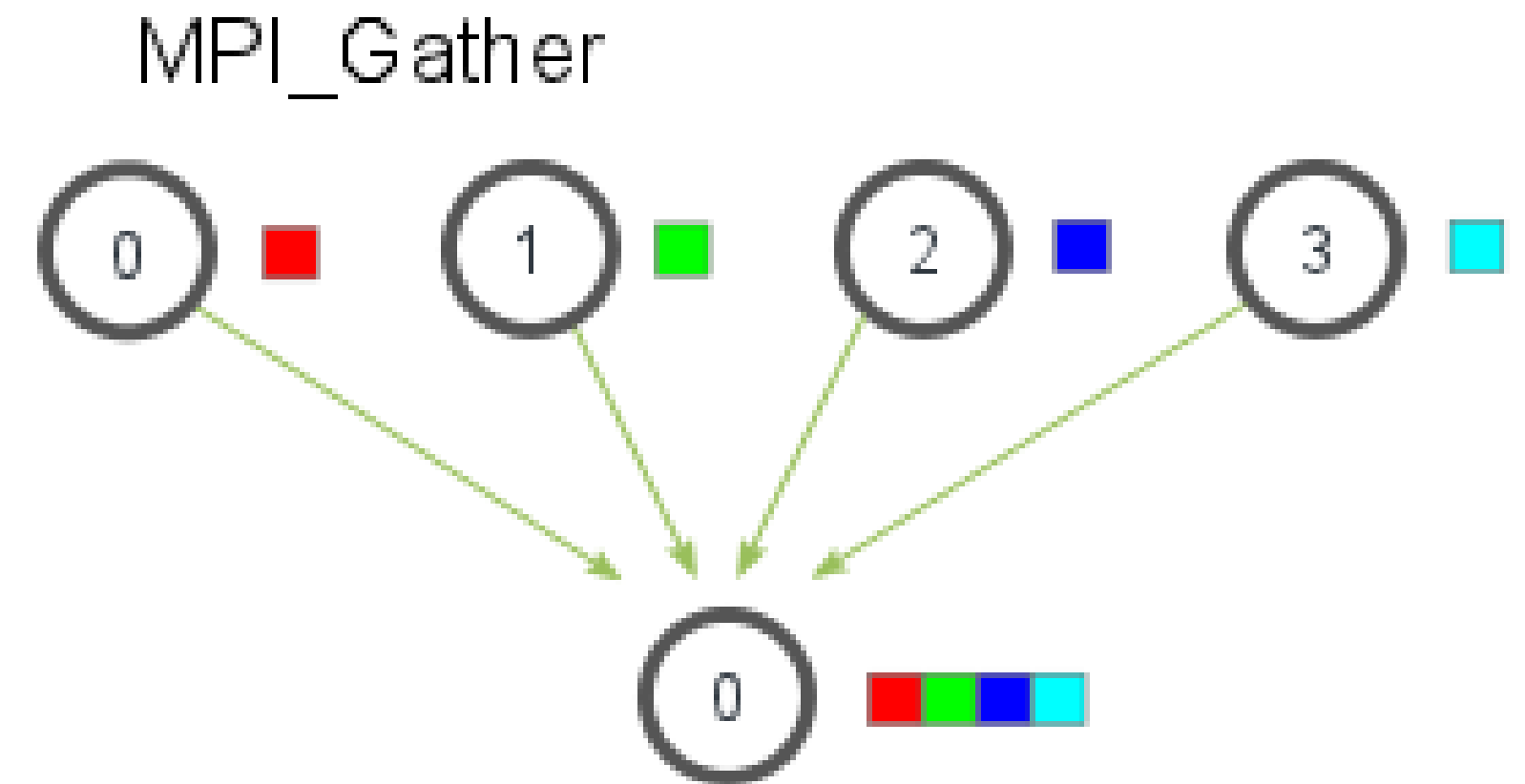
```
void Read_vector(  
    double    local_a[]    /* out */,  
    int       local_n      /* in  */,  
    int       n            /* in  */,  
    char      vec_name[]   /* in  */,  
    int       my_rank      /* in  */,  
    MPI_Comm  comm         /* in  */) {  
  
    double* a = NULL;  
    int i;  
  
    if (my_rank == 0) {  
        a = malloc(n*sizeof(double));  
        printf("Enter the vector %s\n", vec_name);  
        for (i = 0; i < n; i++)  
            scanf("%lf", &a[i]);  
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,  
                    0, comm);  
        free(a);  
    } else {  
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,  
                    0, comm);  
    }  
} /* Read_vector */
```

- Envia local_n elementos do vetor **a** do tipo **double**.
- Cada processo armazenará estes números em **local_a**
- **local_n** igual a n dividido por número de processos



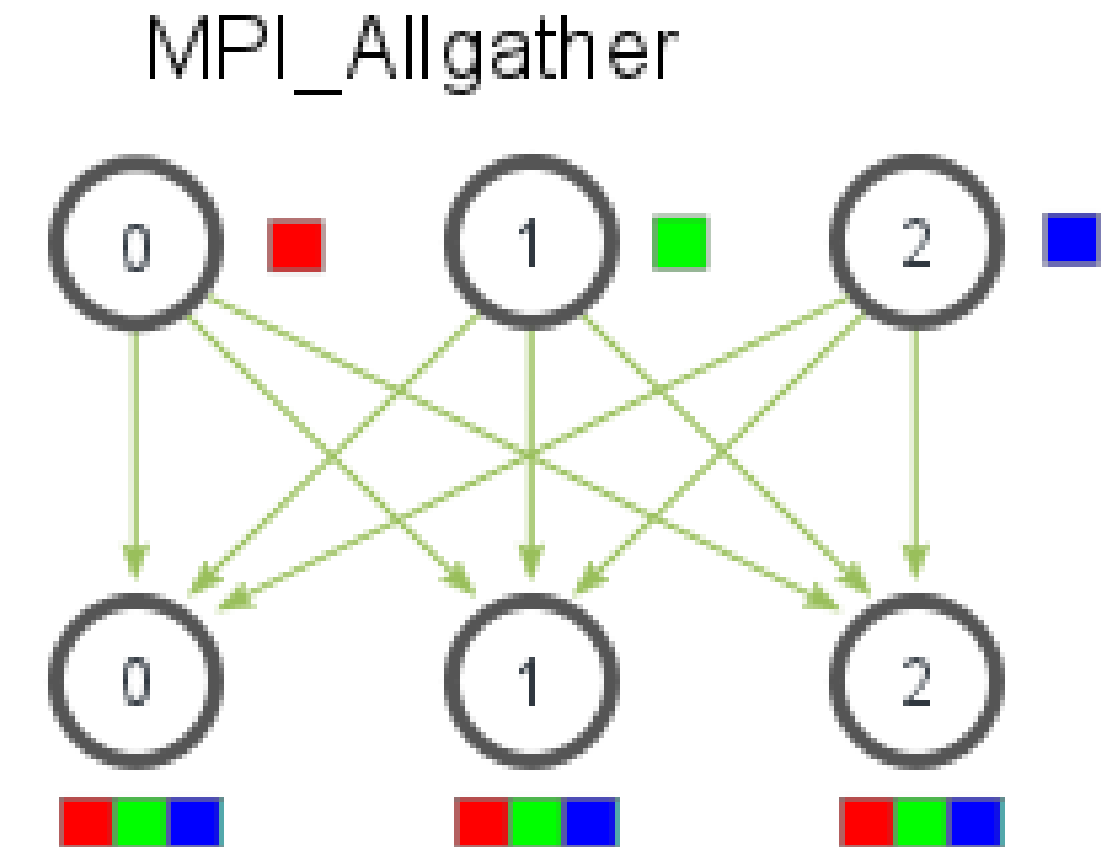
MPI_Gather

```
int MPI_Gather(  
    void* send_buf_p    /* in */,  
    int send_count     /* in */,  
    MPI_Datatype send_type /* in */,  
    void* recv_buf_p   /* out */,  
    int recv_count     /* in */,  
    MPI_Datatype recv_type /* in */,  
    int dest_proc      /* in */,  
    MPI_Comm comm      /* in */ );
```



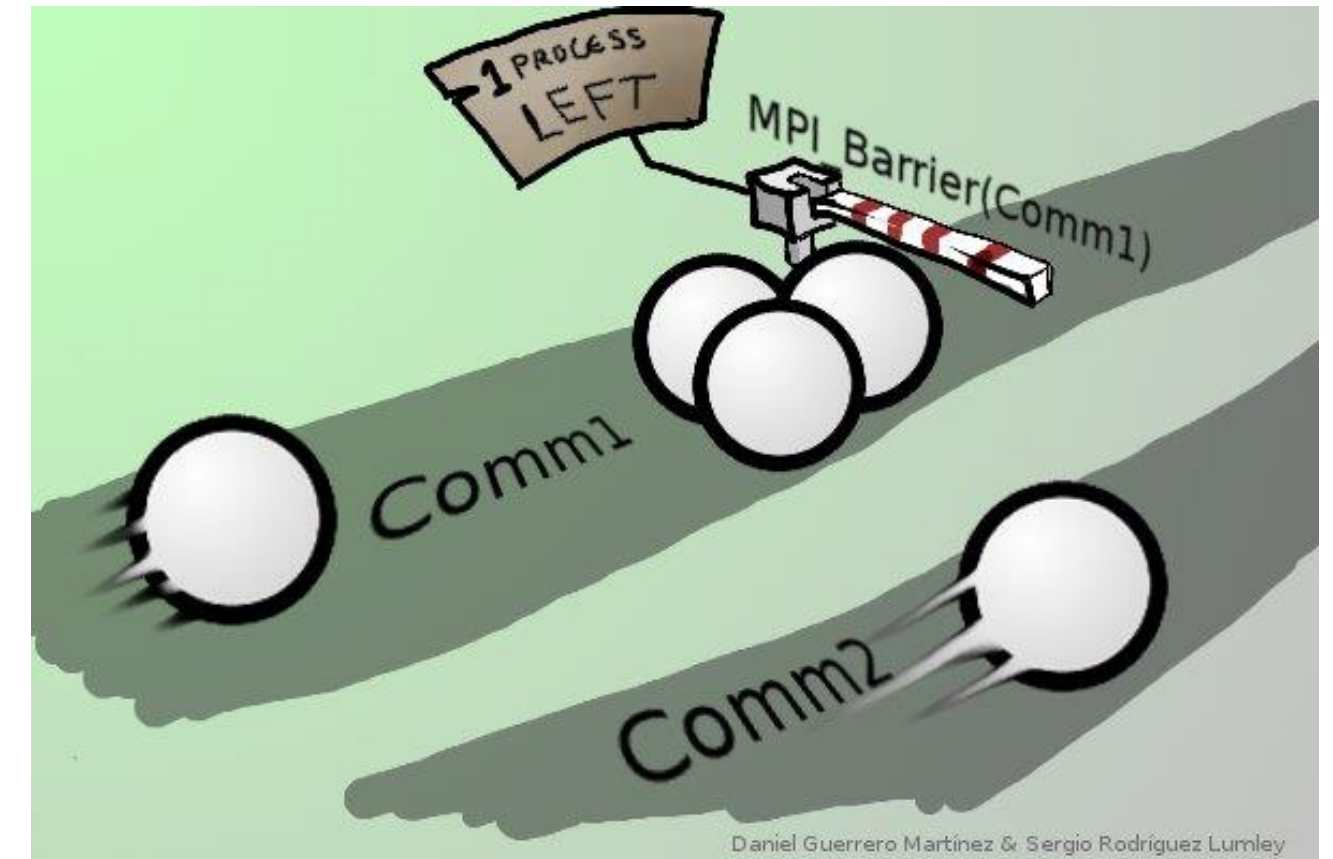
MPI_Allgather

```
int MPI_Allgather(  
    void*      send_buf_p    /* in */ ,  
    int        send_count    /* in */ ,  
    MPI_Datatype send_type    /* in */ ,  
    void*      recv_buf_p    /* out */ ,  
    int        recv_count     /* in */ ,  
    MPI_Datatype recv_type    /* in */ ,  
    MPI_Comm    comm          /* in */ );
```



MPI_Barrier (1)

- Cada processo que chama a barreira tem sua execução parada.
- Os processos parados somente continuam sua execução se **todos** os processos do comunicador alcançarem (executarem) a barreira.



```
int MPI_Barrier(MPI_Comm comm /* in */);
```



MPI_Barrier (2)

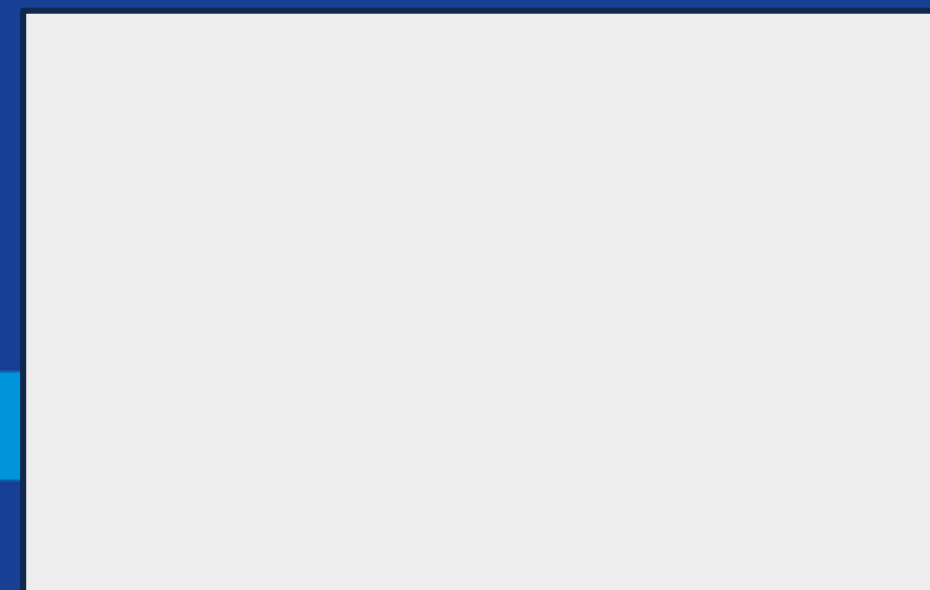
Barreira
antes
da
redução

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
→ MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
→ MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
```

Odd-even transposition sort

Algoritmo de ordenação ímpar-par



Odd-even transposition sort

- Baseado no algoritmo de ordenação em bolha (bubble sort)
- Sequência de fases
- Fases pares, compara e troca:

$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \dots$

- Fases ímpares, compara e troca

$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \dots$



Odd-even transposition sort

Exemplo

- Início do vetor: 5, 9, 4, 3
- Fase par: compara-troca (5,9) e (4,3)
obtemos a lista 5, 9, 3, 4
- Fase ímpar: compara-troca (9,3)
obtemos a lista 5, 3, 9, 4
- Fase ímpar: compara-troca (5,3) e (9,4)
obtemos a lista 3, 5, 4, 9
- Fase par: compara-troca (5,4)
obtemos a lista 3, 4, 5, 9

Tamanho do vetor n
 $n/2$ fases pares
 $n/2$ fases ímpares



Odd-even transposition sort

Serial

```
void Odd_even_sort(  
    int a[] /* in/out */,  
    int n /* in */) {  
    int phase, i, temp;  
  
    for (phase = 0; phase < n; phase++)  
        if (phase % 2 == 0) { /* Even phase */  
            for (i = 1; i < n; i += 2)  
                if (a[i-1] > a[i]) {  
                    temp = a[i];  
                    a[i] = a[i-1];  
                    a[i-1] = temp;  
                }  
        } else { /* Odd phase */  
            for (i = 1; i < n-1; i += 2)  
                if (a[i] > a[i+1]) {  
                    temp = a[i];  
                    a[i] = a[i+1];  
                    a[i+1] = temp;  
                }  
        }  
    } /* Odd_even_sort */  
}
```

Fase par:
comparo i c/ $i-1$



Fase ímpar
comparo i c/ $i+1$



Odd-even transposition sort

Paralelo

Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

