

MO644/MC900

Programação Paralela usando MPI

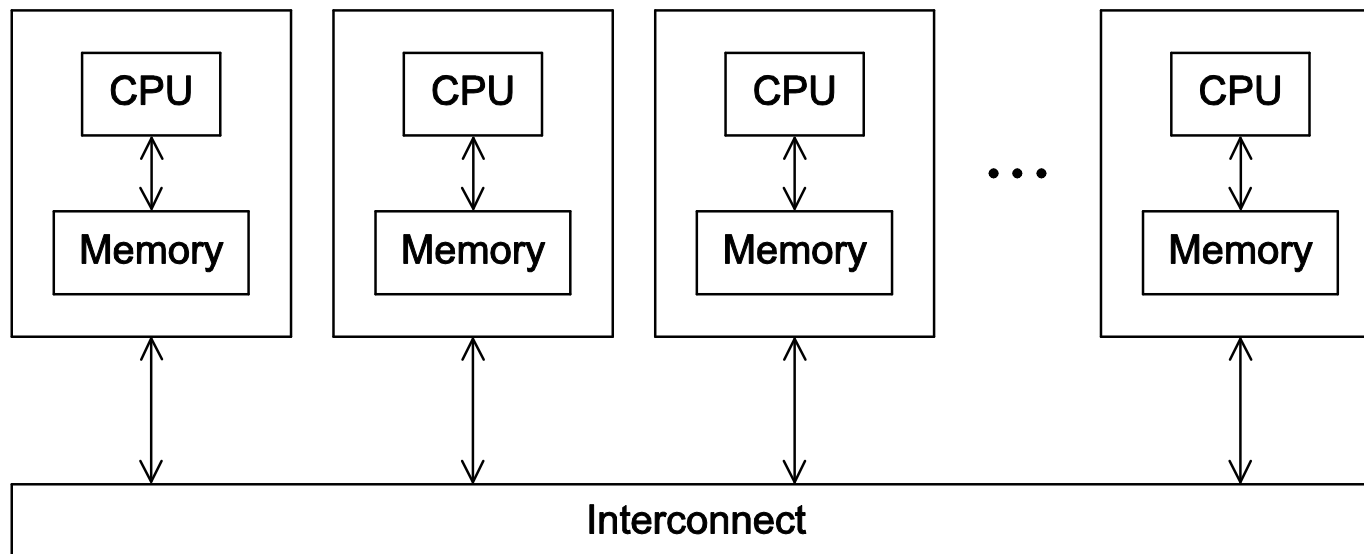
Prof. Guido Araujo

www.ic.unicamp.br/~guido

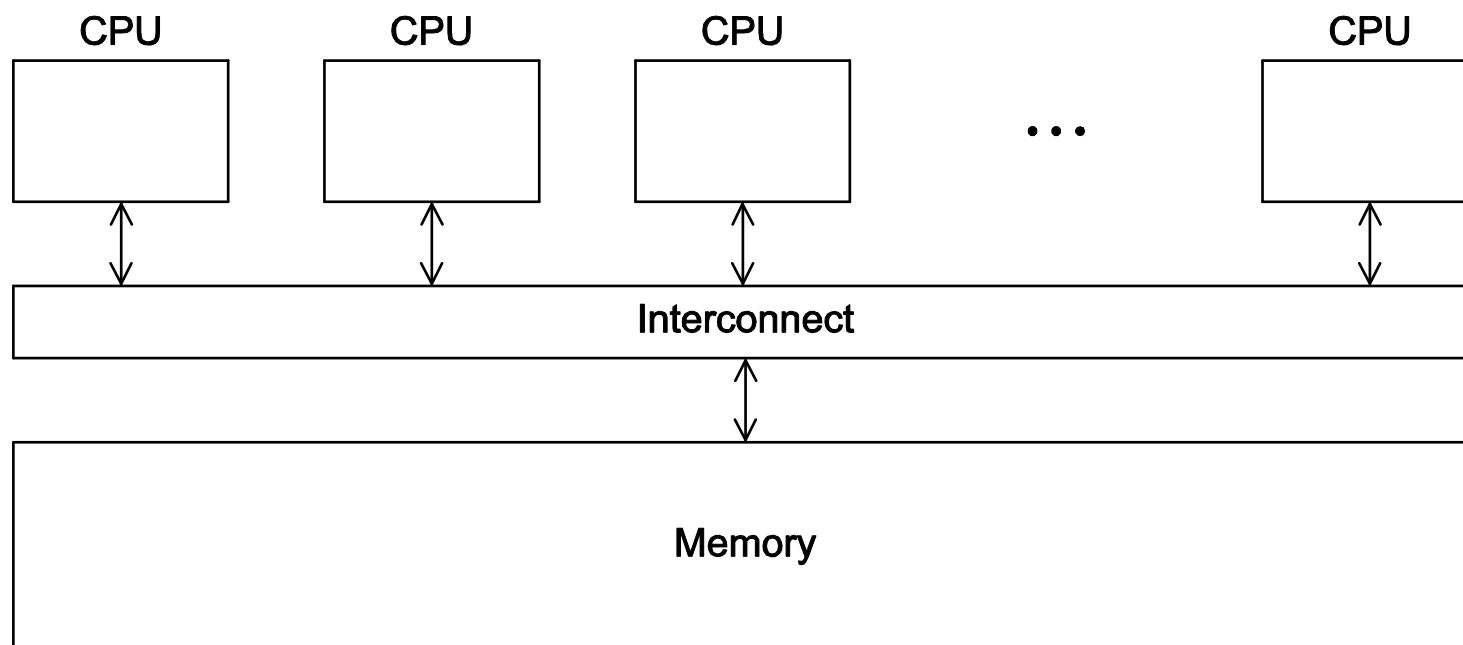
Roadmap

- Escrevendo seu primeiro programa MPI.
- Usando funções básicas de MPI.
- A Regra Trapezoidal em MPI.
- Comunicação coletiva.
- Tipos de dados derivados de MPI.
- Avaliação de desempenho de programas MPI.
- Ordenação paralela.
- Segurança em programas MPI.

Sistema de memória distribuída



Sistema de memória compartilhada



Hello World!

```
#include <stdio.h>

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

    return 0;
}
```

(um clássico)



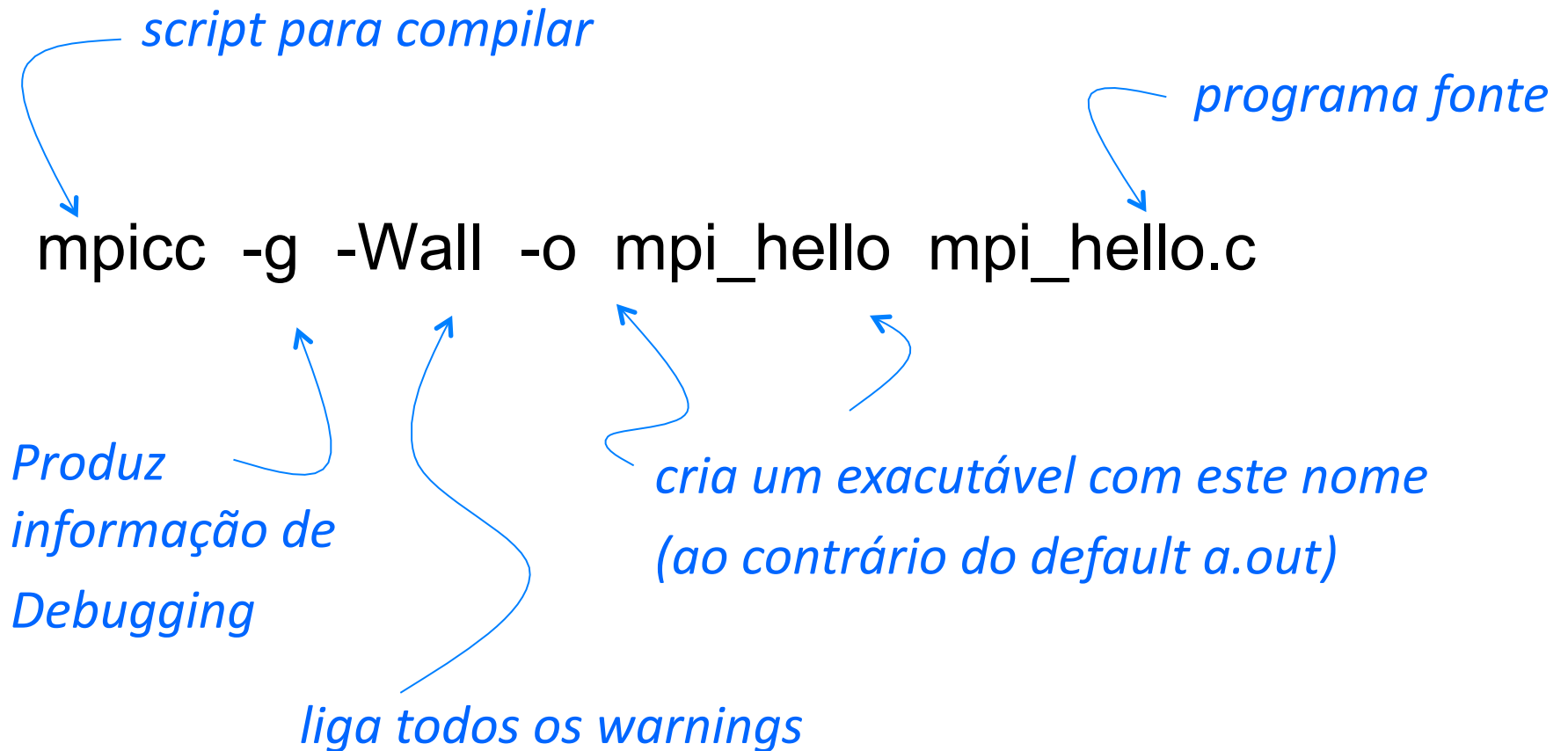
Identificando processos MPI

- É uma prática comum identificar processos através de números inteiros não negativos.
- p processos são numerados $0, 1, 2, \dots, p-1$

Nosso primeiro programa MPI

```
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 */
```

Compilando



Executando

`mpirun -n <number of processes> <executable>`

`mpirun -n 1 ./mpi_hello`

 *execute com 1 processo*

`mpirun -n 4 ./mpi_hello`

 *execute com 4 processos*

Execução

```
mpiexec -n 1 ./mpi_hello
```

Greetings from process 0 of 1 !

```
mpiexec -n 4 ./mpi_hello
```

Greetings from process 0 of 4 !

Greetings from process 1 of 4 !

Greetings from process 2 of 4 !

Greetings from process 3 of 4 !

Programas MPI

- Escritos em C.
 - Tem um main.
 - Usa `stdio.h`, `string.h`, etc.
- É preciso incluir `mpi.h`
- Identificadores de MPI iniciam com “MPI_”.
- Primeira letra após underscore é maiúscula.
 - Usado para nomes de funções e tipos definidos em MPI.
 - Ajuda a evitar confusão de nomes.

MPI Components

- MPI_Init
 - Informa o MPI para fazer o setup inicial

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

- MPI_Finalize
 - Informa o MPI que acabamos de modo que ele pode limpar tudo o que foi alocado.

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

Esqueleto de um programa MPI

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
    . . .
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    . . .
    MPI_Finalize();
    /* No MPI calls after this */
    . . .
    return 0;
}
```

Comunicadores

- É uma coleção de processos que pode enviar mensagens entre si.
- MPI_Init define um comunicador que contém todos os processos a serem utilizados no programa.
- Chamado **MPI_COMM_WORLD**.

Comunicadores



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

retorna número de processos no comunicador

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

*retorna meu rank
(o processo fazendo esta chamada)*

SPMD

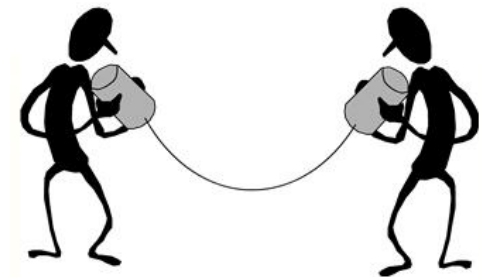
- Single-Program Multiple-Data
- Compilamos um único programa.
- Processo 0 faz algo diferente.
 - Recebe mensagens e as imprime enquanto os outros processos fazem o trabalho.
- A sentença **if-else** torna nosso programa SPMD.

Comunicação

```
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 */);
```

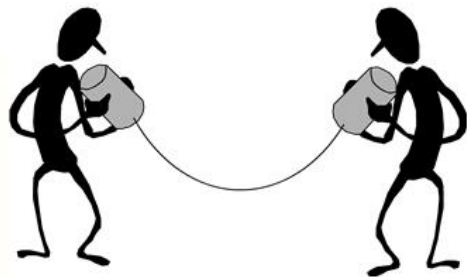


Tipos de dados

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	

Comunicação

```
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 */);
```



Condições mínimas para recepção de uma mensagem

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,  
send_comm);
```

\geq

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,  
recv_comm, &status);
```



Não assuma, verifique sua instalação!!

Casamento de mensagens

q `MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
send_comm);`

MPI_Send

dest = r



MPI_Recv

src = q

r `MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
recv_comm, &status);`

q

Recebendo mensagens

- Um receptor pode receber uma mensagem sem saber:
 - a quantidade de dados em uma mensagem,
 - o remetente (*sender*) da mensagem,
 - ou o rótulo (*tag*) da mensagem.
- Exemplo 1:
 - Um receptor recebe mensagens de vários outros processos que terminam em tempos diferentes
 - *MPI_ANY_SOURCE* pode ser usado
- Exemplo 2:
 - Um receptor recebe mensagens com tags diferentes.
 - *MPI_ANY_TAG* pode ser usado



Como saber o que está faltando então?

status_p argument

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,  
recv_comm, &status);
```

MPI_Status*



MPI_Status* status;

status.MPI_SOURCE

status.MPI_TAG

MPI_SOURCE

MPI_TAG

MPI_ERROR

Use **MPI_STATUS_IGNORE**
se não precisar do status.

Quantos dados estou recebendo?

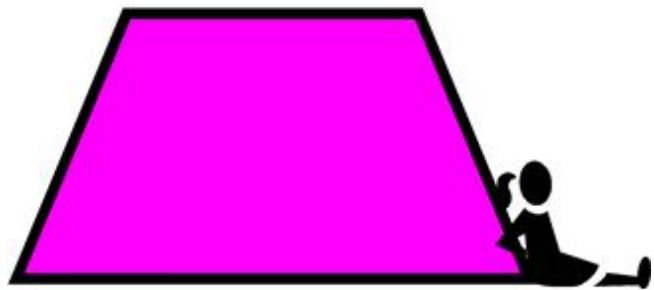
```
int MPI_Get_count(  
    MPI_Status* status_p /* in */,  
    MPI_Datatype type /* in */,  
    int* count_p /* out */);
```



Questões envolvendo *send* e *receive*

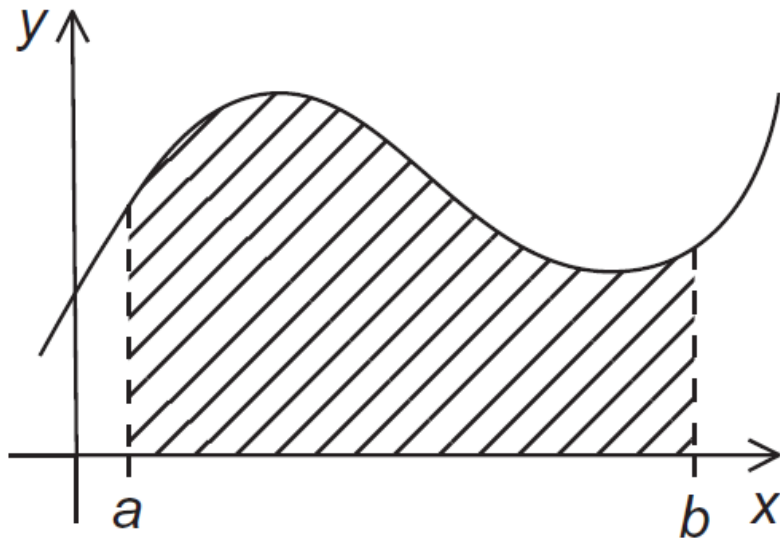
- MPI_Send: bloqueante vs não-bloqueante
 - Tamanho da mensagem define o comportamento
 - Abaixo de um limiar armazena buffer e retorna
 - Acima de um limiar bloqueia até enviar
 - Existem funções especiais para forçar
- MPI_Recv: sempre bloqueia.
- Ordem das mensagens
 - De um mesmo processo são sempre *nonovertaking*.
 - De processos diferentes não existe ordem
- Comportamento exato depende da implementação de MPI
 - Conheça a sua!!



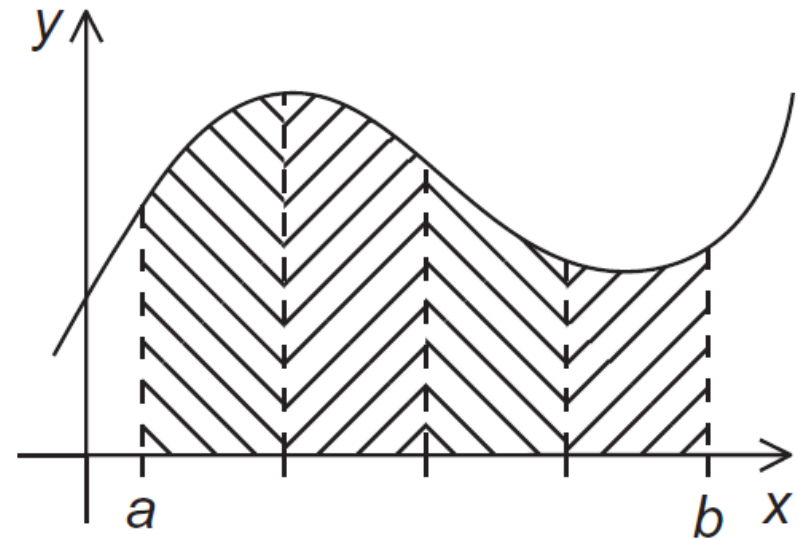


REGRA DO TRAPÉZIO EM MPI

A Regra do Trapézio

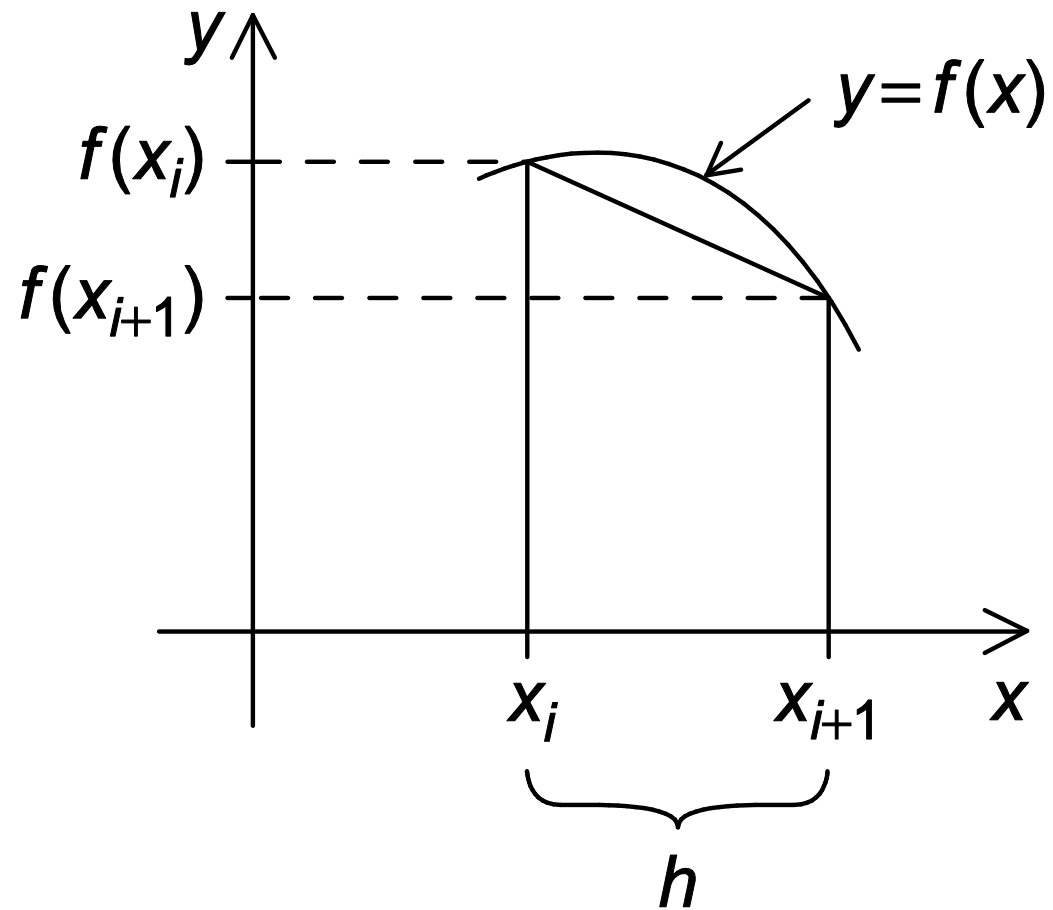


(a)



(b)

Um trapézio



A Regra do Trapézio

$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

$$h = \frac{b - a}{n}$$

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

$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + f(x_n)/2]$$

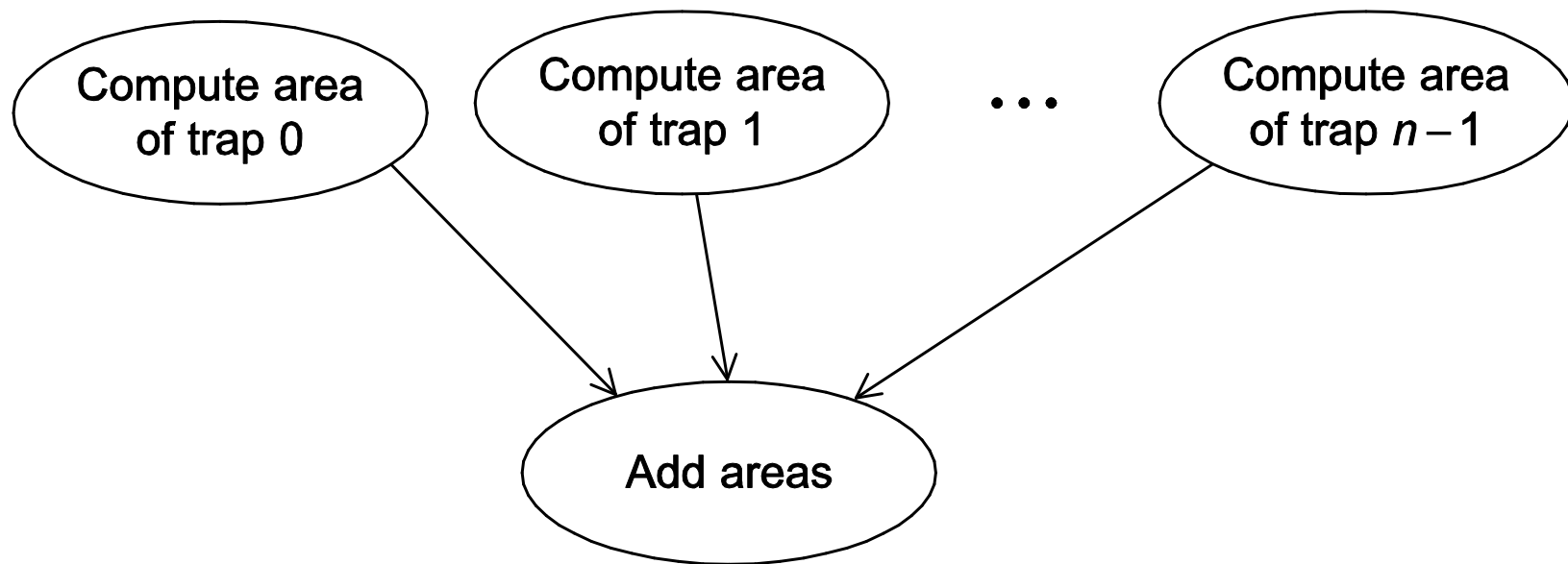
Pseudo-código para o programa

```
/* Input: a, b, n */  
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 0; i <= n-1; i++) {  
    x_i = a + i*h;  
    approx += f(x_i);  
}  
approx = h*approx;
```

Paralelizando a Regra do Trapézio

1. Divida a solução do problema em tarefas.
2. Identifique canais de comunicação entre as tarefas.
3. Agregue tarefas em tarefas compostas.
4. Mapeie tarefas compostas aos núcelos.

Tarefas e comunicações para a Regra do Trapézio



Pseudo-código paralelo

```
1  Get a, b, n;
2  h = (b-a)/n;
3  local_n = n/comm_sz;
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);
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;
```

Primeira versão (1)

```
1  int main(void) {
2      int my_rank, comm_sz, n = 1024, local_n;
3      double a = 0.0, b = 3.0, h, local_a, local_b;
4      double local_int, total_int;
5      int source;
6
7      MPI_Init(NULL, NULL);
8      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11     h = (b-a)/n;          /* h is the same for all processes */
12     local_n = n/comm_sz; /* So is the number of trapezoids */
13
14     local_a = a + my_rank*local_n*h;
15     local_b = local_a + local_n*h;
16     local_int = Trap(local_a, local_b, local_n, h);
17
18     if (my_rank != 0) {
19         MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20                 MPI_COMM_WORLD);
```

Primeira versão (2)

```
21     } else {
22         total_int = local_int;
23         for (source = 1; source < comm_sz; source++) {
24             MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26             total_int += local_int;
27         }
28     }
29
30     if (my_rank == 0) {
31         printf("With n = %d trapezoids, our estimate\n", n);
32         printf("of the integral from %f to %f = %.15e\n",
33               a, b, total_int);
34     }
35     MPI_Finalize();
36     return 0;
37 } /*  main  */
```

Primeira versão (3)

```
1 double Trap(  
2     double left_endpt /* in */,  
3     double right_endpt /* in */,  
4     int trap_count /* in */,  
5     double base_len /* in */) {  
6     double estimate, x;  
7     int i;  
8  
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;  
10    for (i = 1; i <= trap_count-1; i++) {  
11        x = left_endpt + i*base_len;  
12        estimate += f(x);  
13    }  
14    estimate = estimate*base_len;  
15  
16    return estimate;  
17 } /* Trap */
```

Lidando com I/O

```
#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);

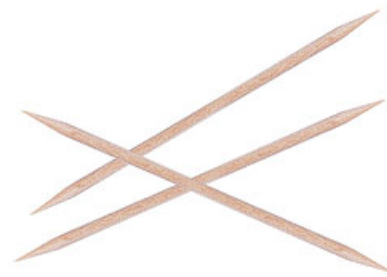
    MPI_Finalize();
    return 0;
} /* main */
```

*Cada processo apenas
imprime uma mensagem.*

Executando 6 processes

```
Proc 0 of 6 > Does anyone have a toothpick?  
Proc 1 of 6 > Does anyone have a toothpick?  
Proc 2 of 6 > Does anyone have a toothpick?  
Proc 4 of 6 > Does anyone have a toothpick?  
Proc 3 of 6 > Does anyone have a toothpick?  
Proc 5 of 6 > Does anyone have a toothpick?
```

saída é imprevisível



Input

- A maioria das implementações de MPI somente permite o processo 0 em MPI_COMM_WORLD acessar a `stdin`.
- O processo 0 deve ler os dados (`scanf`) e enviar para os outros processos.

```
. . .  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
Get_data(my_rank, comm_sz, &a, &b, &n);  
  
h = (b-a)/n;  
. . .
```

Função para leitura da *user input*

```
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 */
```

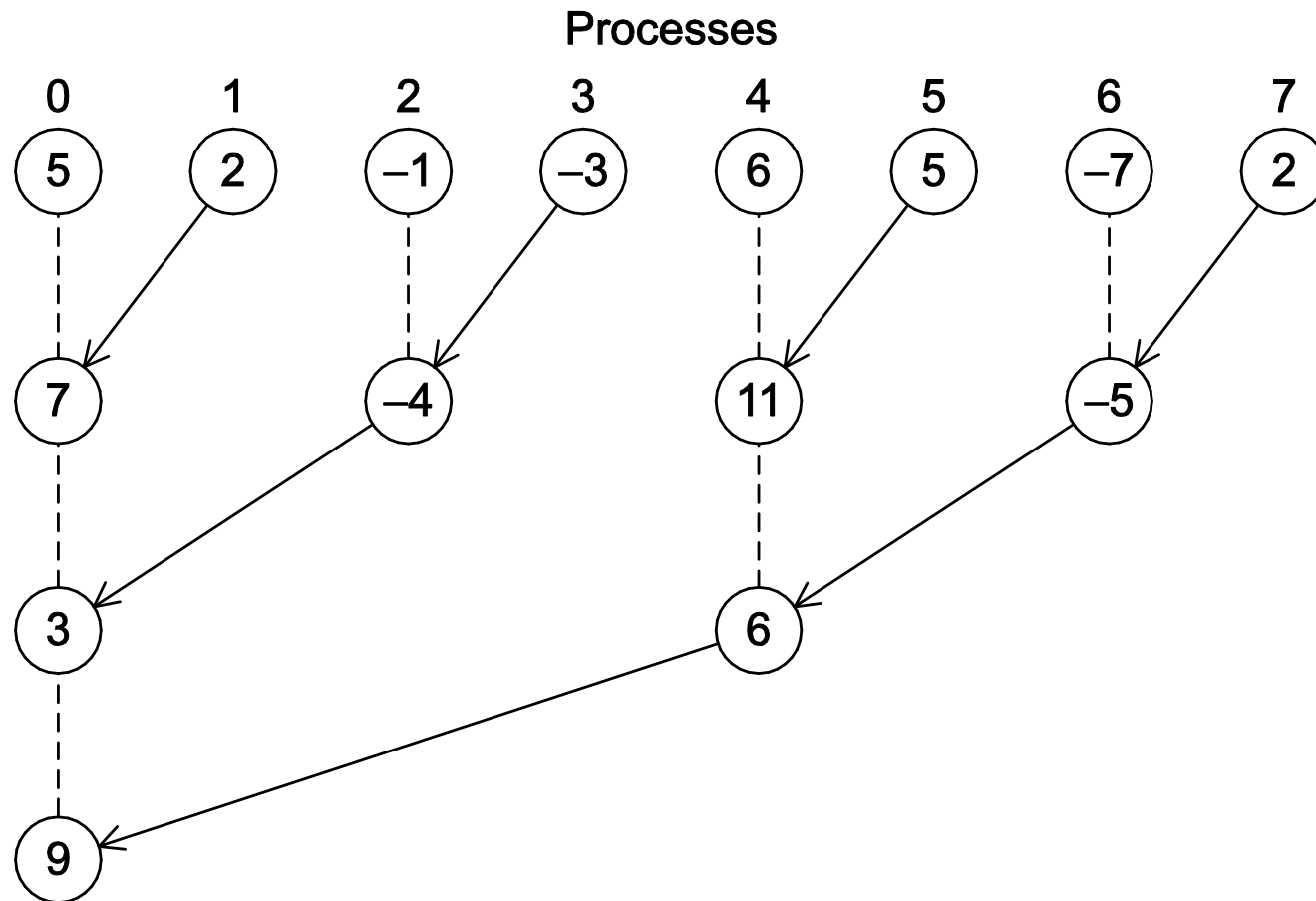



COMUNICAÇÃO COLETIVA

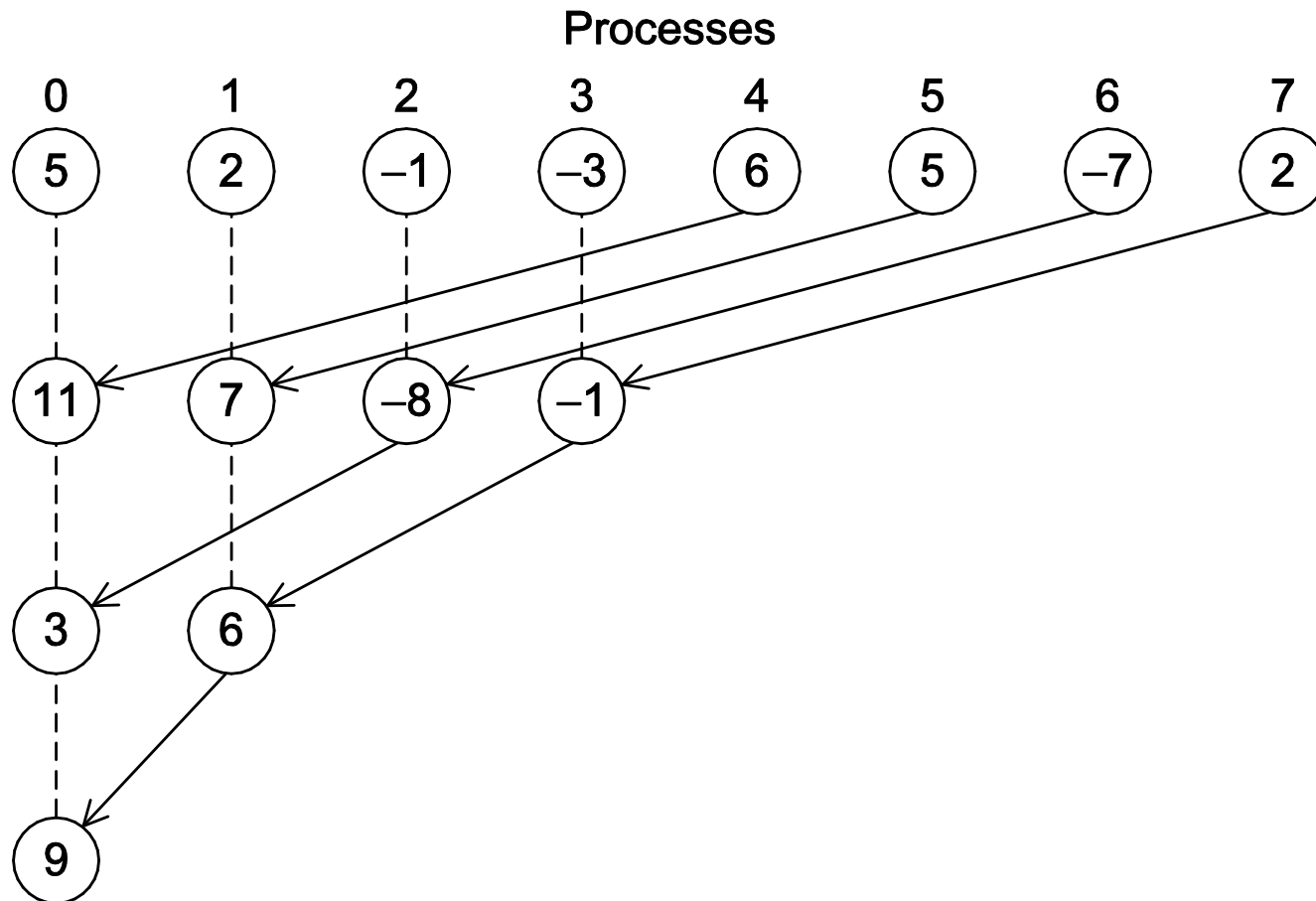
Comunicação usando estrutura em Árvore

1. Na primeira fase:
 - (a) Processo 1 envia para 0, 3 envia para 2, 5 envia para 4, and 7 envia para 6.
 - (b) Processos 0, 2, 4, and 6 somam os valores recebidos.
 - (c) Processes 2 and 6 enviam seus novos valores respectivamente para os processos 0 e 4,.
 - (d) Processos 0 e 4 somam os valores nos seus novos valores
2.
 - (a) Processo 4 envia seu novo valor para o processo 0.
 - (b) Processo 0 soma o valor recebido no seu novo valor.

Soma global usando estrutura em árvore



Soma global usando estrutura em árvore alternativa



MPI_Reduce

```
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 */);
```

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

```
double local_x[N], sum[N];  
...  
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

Predefined reduction operators in MPI

Operation Value	Meaning
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical and
<code>MPI_BAND</code>	Bitwise and
<code>MPI_LOR</code>	Logical or
<code>MPI BOR</code>	Bitwise or
<code>MPI_LXOR</code>	Logical exclusive or
<code>MPI_BXOR</code>	Bitwise exclusive or
<code>MPI_MAXLOC</code>	Maximum and location of maximum
<code>MPI_MINLOC</code>	Minimum and location of minimum

Comunicação

Coletiva vs. Ponto-a-Ponto

- Todos os processos no comunicador precisam chamar a mesma função coletiva.
- Por exemplo, um programa que tenta corresponder uma chamada para `MPI_Reduce` em um processo com uma chamada para `MPI_Recv` em outro processo é errônea, e, com toda a probabilidade, o programa irá travar ou quebrar.

Comunicação

Coletiva vs. Ponto-a-Ponto

- Os argumentos passados por cada processo a uma comunicação MPI coletiva devem ser "compatíveis".
- Por exemplo, se um processo passa em 0, tal como o `dest_process` e outro passa em 1, então o resultado de uma chamada para `MPI_Reduce` é errônea, e, mais uma vez, o programa fica susceptível a travar ou falhar.

Comunicação

Coletiva vs. Ponto-a-Ponto

- O argumento `output_data_p` somente é usado em `dest_process`.
- No entanto, todos os processos ainda precisam fornecer um argumento `output_data_p`, mesmo que seja somente `NULL`.

Comunicação

Coletiva vs. Ponto-a-Ponto

- Comunicações Ponto-a-Ponto são casadas tomando como base *tags* and *communicators*.
- Comunicações Coletiva não usam tags.
- Elas são casadas somente na base do comunicador e da ordem na qual elas são chamadas.

Exemplo (1)

Time	Process 0	Process 1	Process 2
0	a = 1; c = 2 1	a = 1; c = 2 4	a = 1; c = 2 3
1	MPI_Reduce (&a, &b, ...) 1	MPI_Reduce (&c, &d, ...) 4	MPI_Reduce (&a, &b, ...) 3
2	MPI_Reduce (&c, &d, ...) 2	MPI_Reduce (&a, &b, ...) 2	MPI_Reduce (&c, &d, ...) 6

Múltiplas chamadas para MPI_Reduce



Incorreto!!!

Exemplo (1)

Time	Process 0	Process 1	Process 2
0	a = 1; c = 2 1	a = 1; c = 2 3	a = 1; c = 2 4
1	MPI_Reduce (&a, &b, ...) 1	MPI_Reduce (&c, &d, ...) 3	MPI_Reduce (&a, &b, ...) 4
2	MPI_Reduce (&c, &d, ...) 2	MPI_Reduce (&a, &b, ...) 3	MPI_Reduce (&c, &d, ...) 5

Os nomes das variáveis de redução não são importantes, ordem das chamadas é o que importa!!



Correto!!!

Exemplo (2)

- Suponha que cada processo chame `MPI_Reduce` com operador `MPI_SUM`, e processo destino 0.
- Inicialmente, pode parecer que depois de duas chamadas para `MPI_Reduce`, o valor de b será 3, e o valor de d será 6.

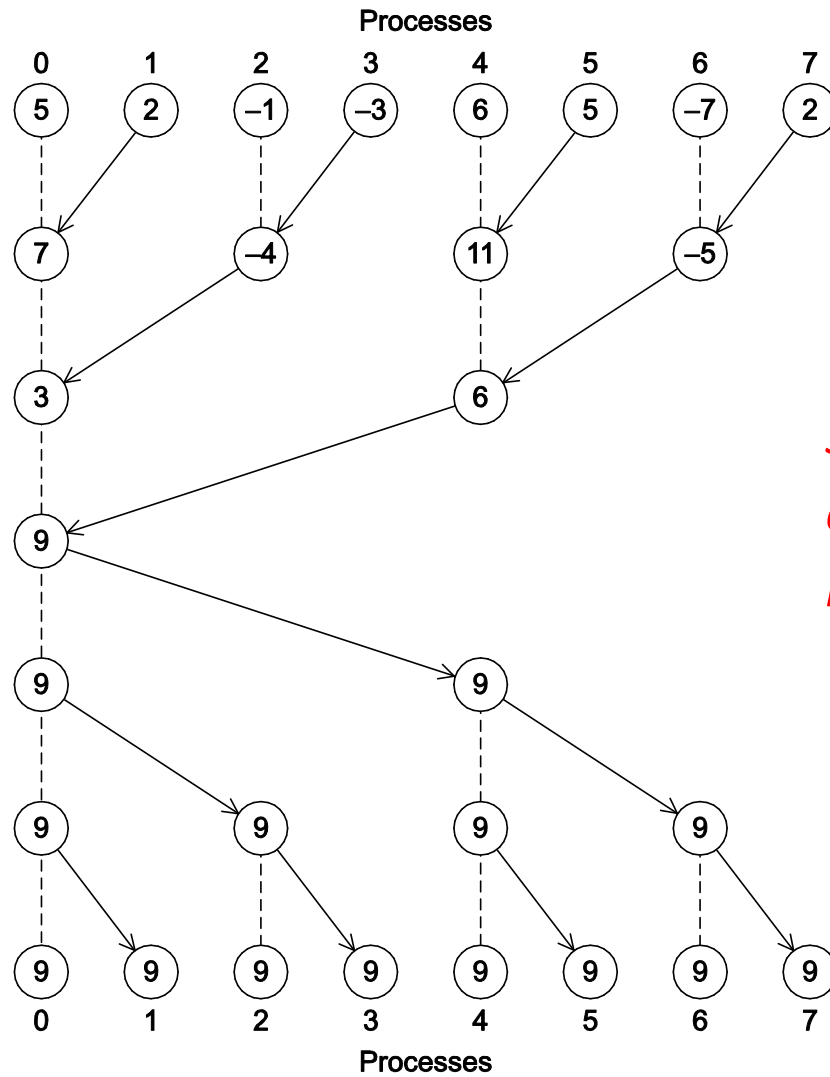
Exemplo (3)

- No entanto, os nomes dos locais de memória são irrelevantes para o caso das chamadas a `MPI_Reduce`.
- A ordem das chamadas irá determinar o casamento de modo que os valor armazenado em b será $1+2+1 = 4$, e o valor armazenado em d será $2+1+2 = 5$.

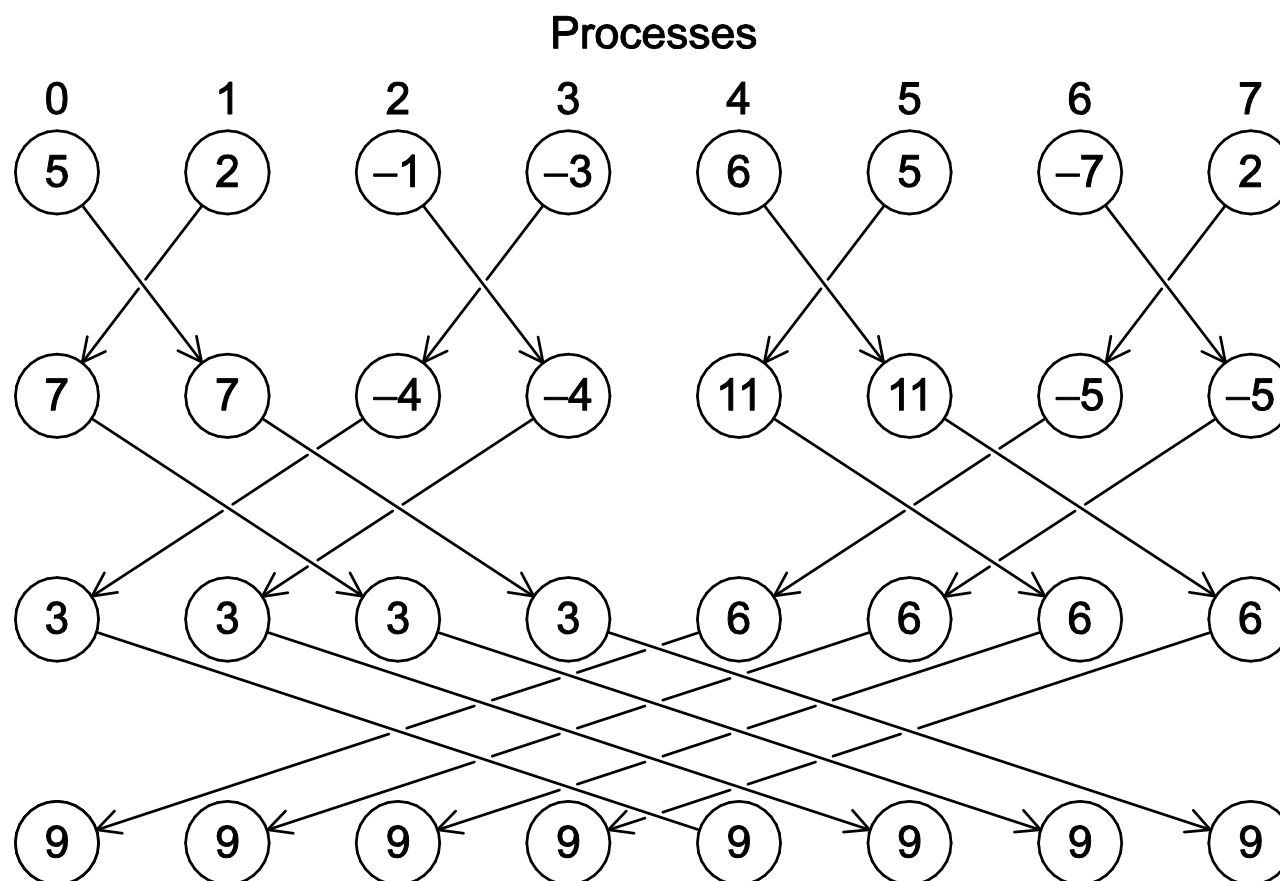
MPI_Allreduce

- Útil em uma situação na qual todos os processos precisam do resultado de uma soma global para completar uma computação maior.

```
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 */);
```



*Soma global seguida
de distribuição de
resultados.*

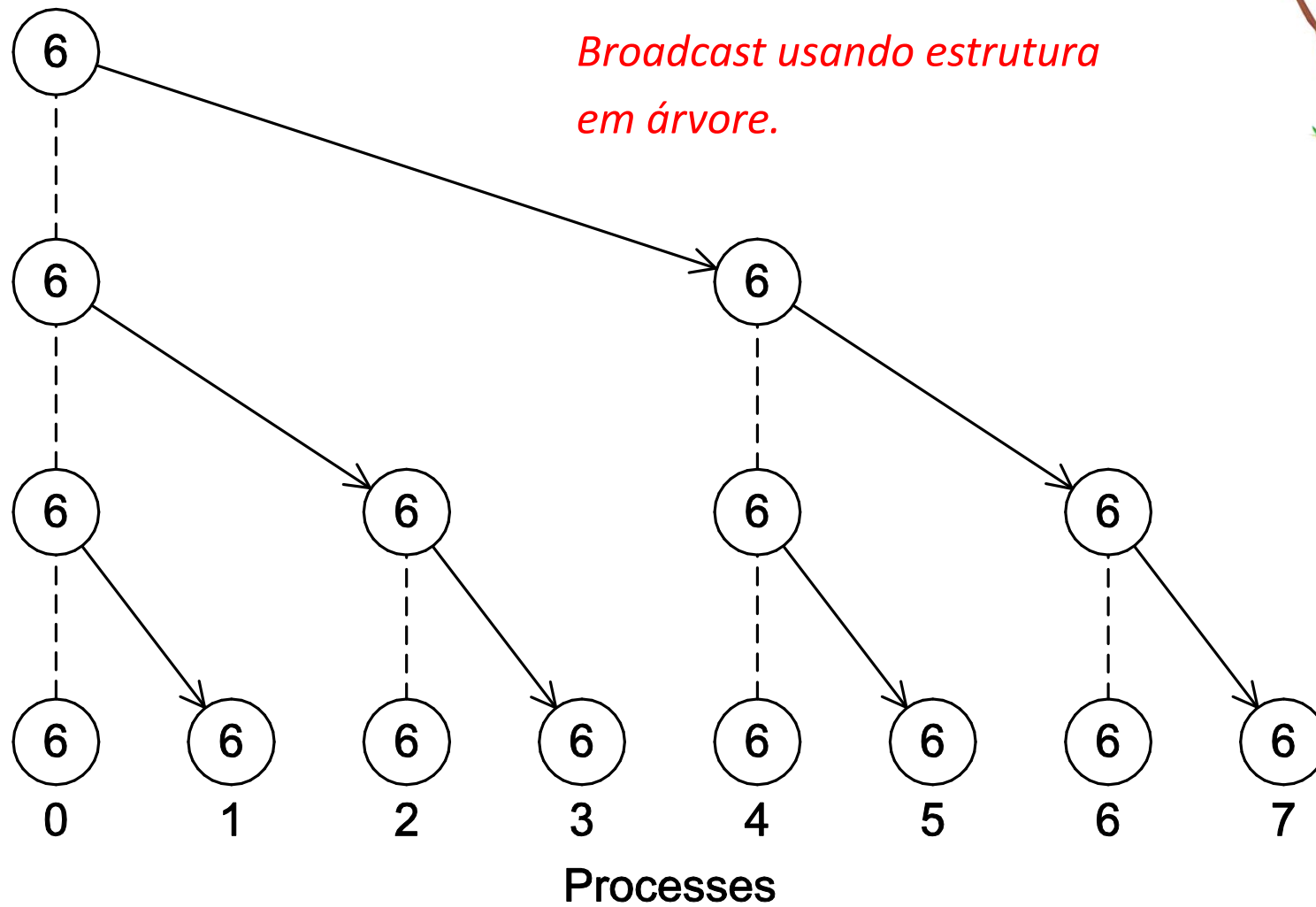


Soma global usando uma estrutura de borboleta.

Broadcast

- Dados pertencentes a um único processo são enviados a todos os processos do comunicador.

```
int MPI_Bcast(  
    void*          data_p          /* in/out */ ,  
    int           count           /* in */ ,  
    MPI_Datatype   datatype       /* in */ ,  
    int           source_proc     /* in */ ,  
    MPI_Comm       comm          /* in */ );
```



Uma versão de *Get_input* que usa *MPI_Bcast*

```
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 */
```

Distribuição de dados

$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \dots, x_{n-1}) + (y_0, y_1, \dots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \dots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

Compute a soma de um vetor

Implementação serial da adição de vetores

```
void Vector_sum(double x[], double y[], double z[], int n) {  
    int i;  
  
    for (i = 0; i < n; i++)  
        z[i] = x[i] + y[i];  
} /* Vector_sum */
```

Partições diferentes de vetor com 12 componentes entre 3 processos

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Opções de partição

- Particionamento por blocos
 - Atribua blocos de componentes consecutivas a cada processo.
- Particionamento cíclico
 - Atribua componentes usando *round-robin*.
- Particionamento cíclico
 - Use uma distribuição cíclica de blocos de componentes.

Implementação paralela da adição de vetores

```
void Parallel_vector_sum(  
    double local_x[] /* in */,  
    double local_y[] /* in */,  
    double local_z[] /* out */,  
    int local_n /* in */) {  
    int local_i;  
  
    for (local_i = 0; local_i < local_n; local_i++)  
        local_z[local_i] = local_x[local_i] + local_y[local_i];  
} /* Parallel_vector_sum */
```

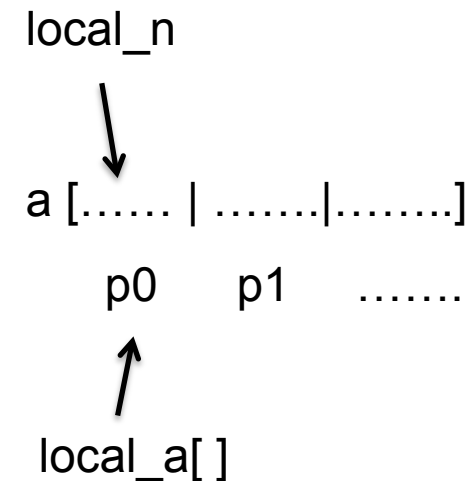
Scatter

- **MPI_Scatter** pode ser usado em uma função que lê um vetor inteiro no processo 0, mas somente envia as componentes necessárias para cada um dos outros processos.

```
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  */);
```

Leitura e distribuição de um vetor

```
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 */
```



Gather

- Coleta todas as componentes do vetor no processo 0, e então o processo 0 pode processar todas as componentes.

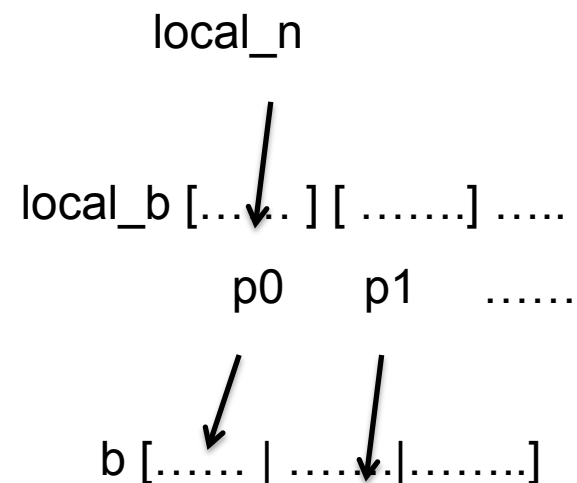
```
int MPI_Gather(  
    void*          send_buf_p  /* in   */,  
    int           send_count  /* in   */,  
    MPI_Datatype   send_type   /* in   */,  
    void*          recv_buf_p /* out  */,  
    int           recv_count  /* in   */,  
    MPI_Datatype   recv_type   /* in   */,  
    int    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

Imprimir um vetor distribuído (1)

```
void Print_vector(  
    double    local_b[]    /* in */,  
    int      local_n      /* in */,  
    int      n             /* in */,  
    char     title[]      /* in */,  
    int      my_rank      /* in */,  
    MPI_Comm comm         /* in */) {  
  
    double* b = NULL;  
    int i;
```

Imprimir um vetor distribuído (2)

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



Allgather

- Concatena os conteúdos dos `send_buf_p` de cada processo e armazena nos `recv_buf_p` de cada processo.
- Usualmente, `recv_count` é a quantidade de dados recebida por cada um dos processs.

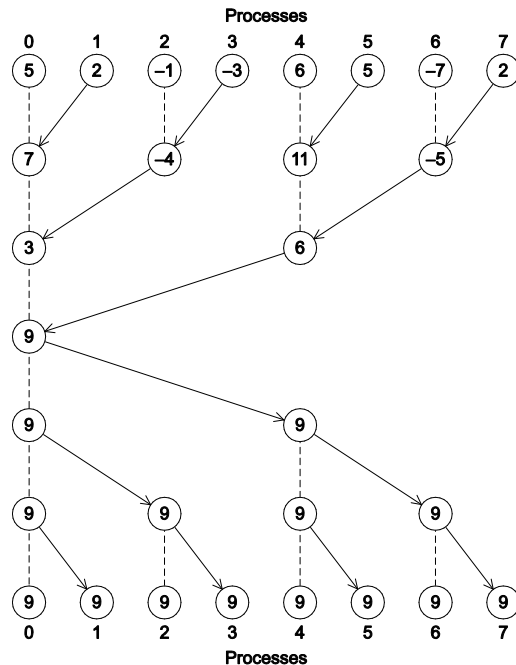
```
int MPI_Allgather(  
    void*      send_buf_p  /* in */,  
    int        send_count  /* in */,  
    MPI_Datatype send_type  /* in */,  
    void*      recv_buf_p  /* out */,  
    int        recv_count  /* in */,  
    MPI_Datatype recv_type  /* in */,  
    MPI_Comm    comm       /* in */);
```

Allgather

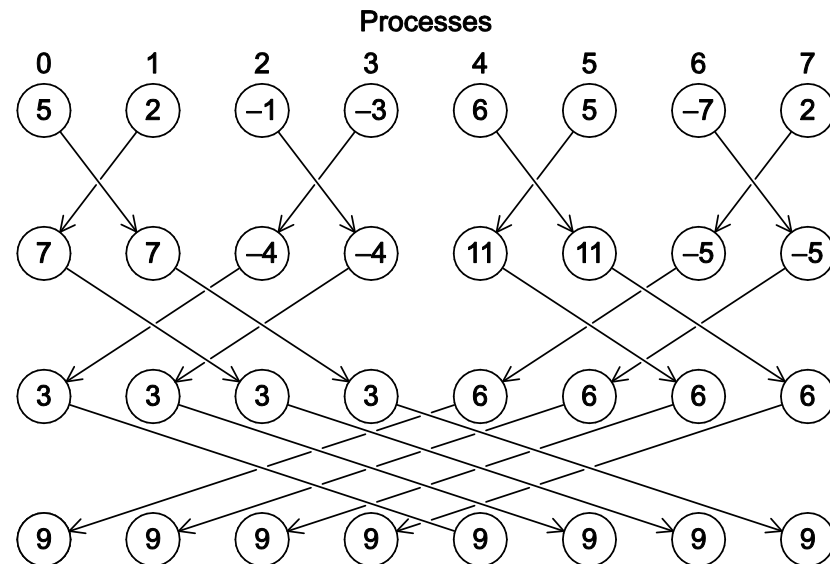
- Como implementar?

Poderia ser um Gather seguido de Broadcast

```
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 */);
```



Mas uma única butterfly faz melhor



Multiplicação matrix-vetor

$A = (a_{ij})$ is an $m \times n$ matrix

\mathbf{x} is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

componente i -ésima de y

*Produto interno da i -ésima linha
de A por x .*

Multiplicação matrix-vetor

a_{00}	a_{01}	\cdots	$a_{0,n-1}$
a_{10}	a_{11}	\cdots	$a_{1,n-1}$
\vdots	\vdots		\vdots
a_{i0}	a_{i1}	\cdots	$a_{i,n-1}$
\vdots	\vdots		\vdots
$a_{m-1,0}$	$a_{m-1,1}$	\cdots	$a_{m-1,n-1}$

x_0
x_1
\vdots
x_{n-1}

 $=$

y_0
y_1
\vdots
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
\vdots
y_{m-1}

Multiplique uma matrix por um vetor

```
/* For each row of A */  
for (i = 0; i < m; i++) {  
    /* Form dot product of ith row with x */  
    v[i] = 0.0;  
    for (j = 0; j < n; j++)  
        y[i] += A[i][j]*x[j];  
}
```

Pseudo-código serial

Arrays em C

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

Armazenado como

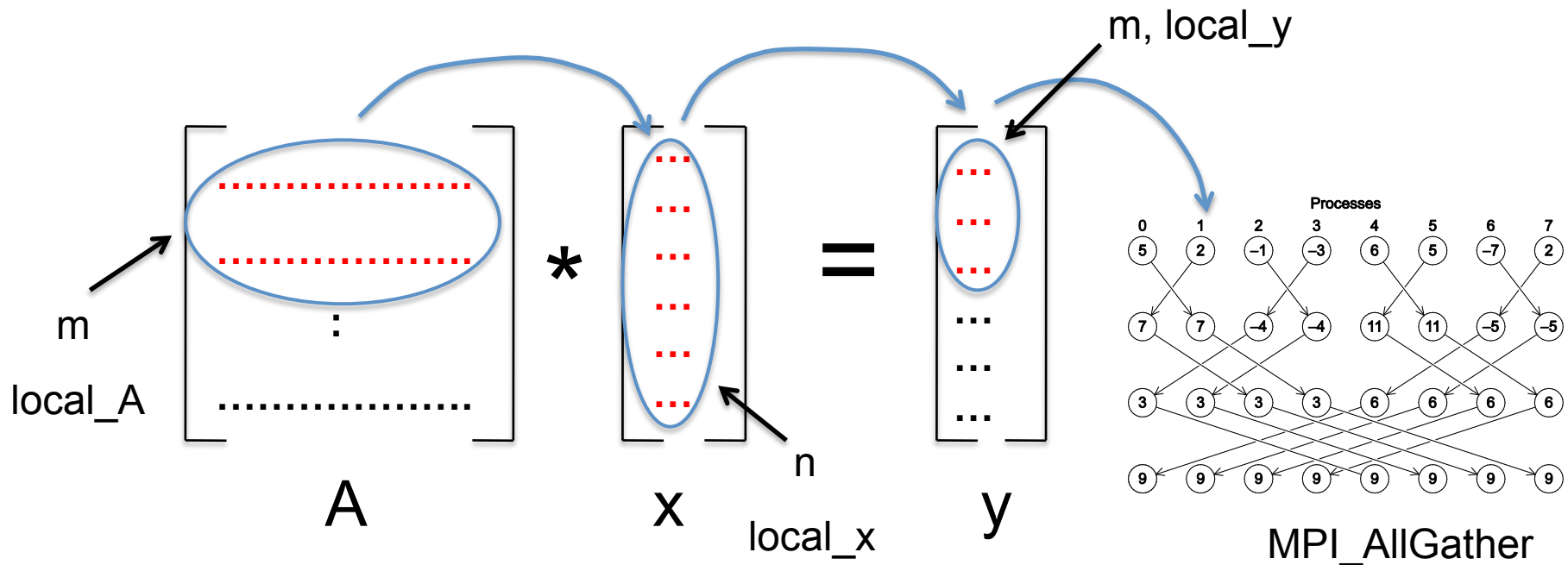
0 1 2 3 4 5 6 7 8 9 10 11

Multiplicação serial matriz-vetor

```
void Mat_vect_mult(  
    double A[] /* in */,  
    double x[] /* in */,  
    double y[] /* out */,  
    int m /* in */,  
    int n /* in */) {  
    int i, j;  
  
    for (i = 0; i < m; i++) {  
        y[i] = 0.0;  
        for (j = 0; j < n; j++)  
            y[i] += A[i*n+j]*x[j];  
    }  
} /* Mat_vect_mult */
```

Assuma que o **x** da próxima execução é o **y** da última

Calculando y e espalhando x



Assuma que o x da próxima execução é o y da última

Uma função MPI para multiplicação de matrix-vector (1)

```
void Mat_vect_mult(  
    double    local_A[]    /* in  */,  
    double    local_x[]    /* in  */,  
    double    local_y[]    /* out */,  
    int        local_m      /* in  */,  
    int        n            /* in  */,  
    int        local_n      /* in  */,  
    MPI_Comm   comm        /* in  */) {  
    double* x;  
    int local_i, j;  
    int local_ok = 1;
```

Uma função MPI para multiplicação de matrix-vector (2)

```
x = malloc(n*sizeof(double));
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
              x, local_n, MPI_DOUBLE, comm);

for (local_i = 0; local_i < local_m; local_i++) {
    local_y[local_i] = 0.0;
    for (j = 0; j < n; j++)
        local_y[local_i] += local_A[local_i*n+j]*x[j];
}
free(x);
} /* Mat_vect_mult */
```




TIPOS DE DADOS DE MPI

Motivação

Qual o mais rápido? Quanto?

```
double x[1000];  
...  
if (my_rank == 0)  
    for (i = 0; i < 1000; i++)  
        MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);  
else /* my_rank == 1 */  
    for (i = 0; i < 1000; i++)  
        MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);
```

```
if (my_rank == 0)  
    MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);  
else /* my_rank == 1 */  
    MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```



50x – 100x mais rápido !!

Tipos de dados derivados

- Usados para representar qualquer coleção de tipos de dados, através da ordenação dos tipos dos itens e de sua localização relativa na memória.
- A idéia é que se uma função que envia dados sabe esta informação sobre uma coleção de itens, ela pode coletar os itens da memória antes que eles sejam enviados .
- Deste modo, uma função que recebe dados, pode distribuir os itens nos seus destinos corretos da memória, quando eles são recebidos.

Tipos de dados derivados

- Formalmente, consistem de uma sequência de tipos de dados básicos de MPI em conjunto com o deslocamento de cada um destes.
- Exemplo: Regra do Trapézio:

Variable	Address
a	24
b	40
n	48

$\{(MPI_DOUBLE, 0), (MPI_DOUBLE, 16), (MPI_INT, 24)\}$

MPI_Type_create_struct

- Constrói um tipo de dado que consiste em elementos individuais cada um possuindo um tipo de dados básico diferente.

```
int MPI_Type_create_struct(  
    int          count          /* in  */,  
    int          array_of_blocklengths[] /* in  */,  
    MPI_Aint     array_of_displacements[] /* in  */,  
    MPI_Datatype array_of_types[] /* in  */,  
    MPI_Datatype* new_type_p    /* out */);
```

MPI_Get_address

- Retorna o endereço do local de memória apontado por `location_p`.
- O tipo especial `MPI_Aint` é um tipo inteiro grande o suficiente para armazenar um endereço no sistema.

```
int MPI_Get_address(  
    void*      location_p  /* in */,  
    MPI_Aint*  address_p   /* out */);
```

MPI_Type_commit

- Permite uma implementação MPI otimizar sua representação interna do tipo de dados para usar em funções de comunicação.

```
int MPI_Type_commit(MPI_Datatype* new_mpi_t_p /* in/out */);
```

MPI_Type_free

- Libera todo o armazenamento adicional depois que não existe mais uso para o tipo de dado derivado.

```
int MPI_Type_free(MPI_Datatype* old_mpi_t_p /* in/out */);
```


Cria uma função de entrada para um tipo derivado de dado (1)

```
void Build_mpi_type(  
    double*      a_p          /* in */,  
    double*      b_p          /* in */,  
    int*         n_p          /* in */,  
    MPI_Datatype* input_mpi_t_p /* out */) {  
  
    int array_of_blocklengths[3] = {1, 1, 1};  
    MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};  
    MPI_Aint a_addr, b_addr, n_addr;  
    MPI_Aint array_of_displacements[3] = {0};
```

Cria uma função de entrada para um tipo derivado de dado (2)

```
MPI_Get_address(a_p, &a_addr);
MPI_Get_address(b_p, &b_addr);
MPI_Get_address(n_p, &n_addr);
array_of_displacements[1] = b_addr-a_addr;
array_of_displacements[2] = n_addr-a_addr;
MPI_Type_create_struct(3, array_of_blocklengths,
                      array_of_displacements, array_of_types,
                      input_mpi_t_p);
MPI_Type_commit(input_mpi_t_p);
} /* Build_mpi_type */
```

Cria uma função de entrada para um tipo derivado de dado (3)

```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
               int* n_p) {
    MPI_Datatype input_mpi_t;

    Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

    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, input_mpi_t, 0, MPI_COMM_WORLD);

    MPI_Type_free(&input_mpi_t);
} /* Get_input */
```

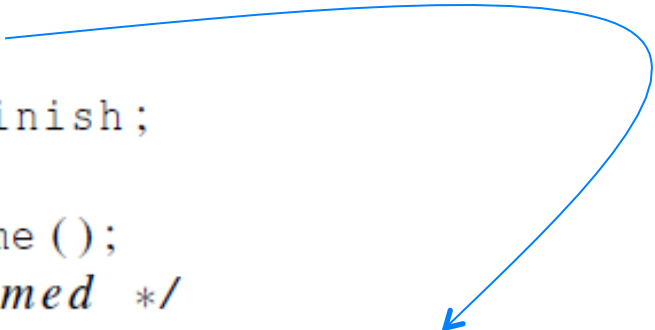


AVALIAÇÃO DE DESEMPENHO

Elapsed parallel time

- Retorna o tempo passado (em segundos) desde um certo momento no passado.

```
double MPI_Wtime(void);  
    double start, finish;  
    . . .  
    start = MPI_Wtime();  
    /* Code to be timed */  
    . . .  
    finish = MPI_Wtime();  
    printf("Proc %d > Elapsed time = %e seconds\n"  
          my_rank, finish-start);
```



Elapsed serial time

- Neste caso você não precisa fazer o link da biblioteca MPI.
- Returns time in microseconds elapsed from some point in the past.

```
#include "timer.h"  
.  
.  
.  
double now;  
.  
.  
.  
GET_TIME(now);
```



Elapsed serial time

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

MPI_Barrier

- Garante que nenhum processo vai retornar desta chamada até que todos os processos no comunicador também a chamem.

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



MPI_Barrier

```
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);
```

Como juntar os tempos locais?



Número de operações

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}$$

Quem é este?

$T_{\text{allgather}}$

```
for (i = 0; i < m; i++) {  
    y[i] = 0.0;  
    for (j = 0; j < n; j++)  
        y[i] += A[i*n+j]*x[j];  
}
```

Se matrix quadrada n x n

$$T_{\text{serial}}(n) \approx an^2$$

$$T_{\text{parallel}}(n) \approx n^2/p$$

Desempenho com n e p

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{allgather}}$$
$$T_{\text{serial}}(n)/p \uparrow$$

Fixando n e variando p = 2, 4 (pequeno)

O que ocorrerá?

$$T_{\text{serial}}(8192) = 1.9 \times T_{\text{parallel}}(8192, 2)$$

$$T_{\text{parallel}}(8192, 2) = 2.0 \times T_{\text{parallel}}(8192, 4)$$

$$T_{\text{serial}}(16, 384) = 2.0 \times T_{\text{parallel}}(16, 384, 2)$$

$$T_{\text{parallel}}(16, 384, 2) = 2.0 \times T_{\text{parallel}}(16, 384, 4)$$

Por que?

Desempenho com n e p

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{allgather}}$$

$\uparrow T_{\text{serial}}(n)/p$

Fixando $p = 2, 4$ (pequeno) e variando n

O que ocorrerá?

$$T_{\text{serial}}(4096) = 4.0 \times T_{\text{serial}}(2048)$$

$$T_{\text{parallel}}(4096, 2) = 3.9 \times T_{\text{parallel}}(2048, 2)$$

$$T_{\text{parallel}}(4096, 4) = 3.5 \times T_{\text{parallel}}(2048, 4)$$

$$T_{\text{serial}}(8192) = 4.2 \times T_{\text{serial}}(4096)$$

$$T_{\text{parallel}}(8192, 2) = 4.2 \times T_{\text{parallel}}(4096, 2)$$

$$T_{\text{parallel}}(8192, 4) = 3.9 \times T_{\text{parallel}}(4096, 4)$$

Por que?

Desempenho com n e p

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{allgather}}$$

$\downarrow T_{\text{serial}}(n)/p \uparrow$

Fazendo n pequeno e p grande

O que ocorrerá?

$$T_{\text{parallel}}(1024, 8) = 1.0 \times T_{\text{parallel}}(1024, 16)$$

$$T_{\text{parallel}}(2048, 16) = 1.5 \times T_{\text{parallel}}(1024, 16)$$

Por que?

Run-times da multiplicação serial e paralela de matriz-vetor

$$T_{\text{parallel}}(n, p) = \boxed{T_{\text{serial}}(n)/p} + T_{\text{allgather}}$$

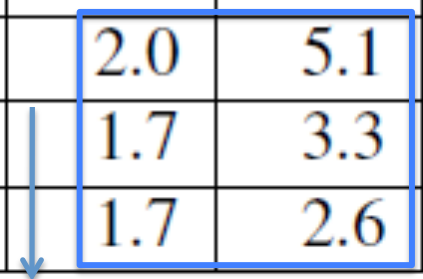
comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71



Run-times da multiplicação serial e paralela de matriz-vetor

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{allgather}}$$

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71



Speedup

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

Eficiência

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)}$$

Speedups da multiplicação serial e paralela de matriz-vetor

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Afasta-se de p

Aproxima-se de p

Eficiências da multiplicação paralela de vetor-matriz

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Escalabilidade

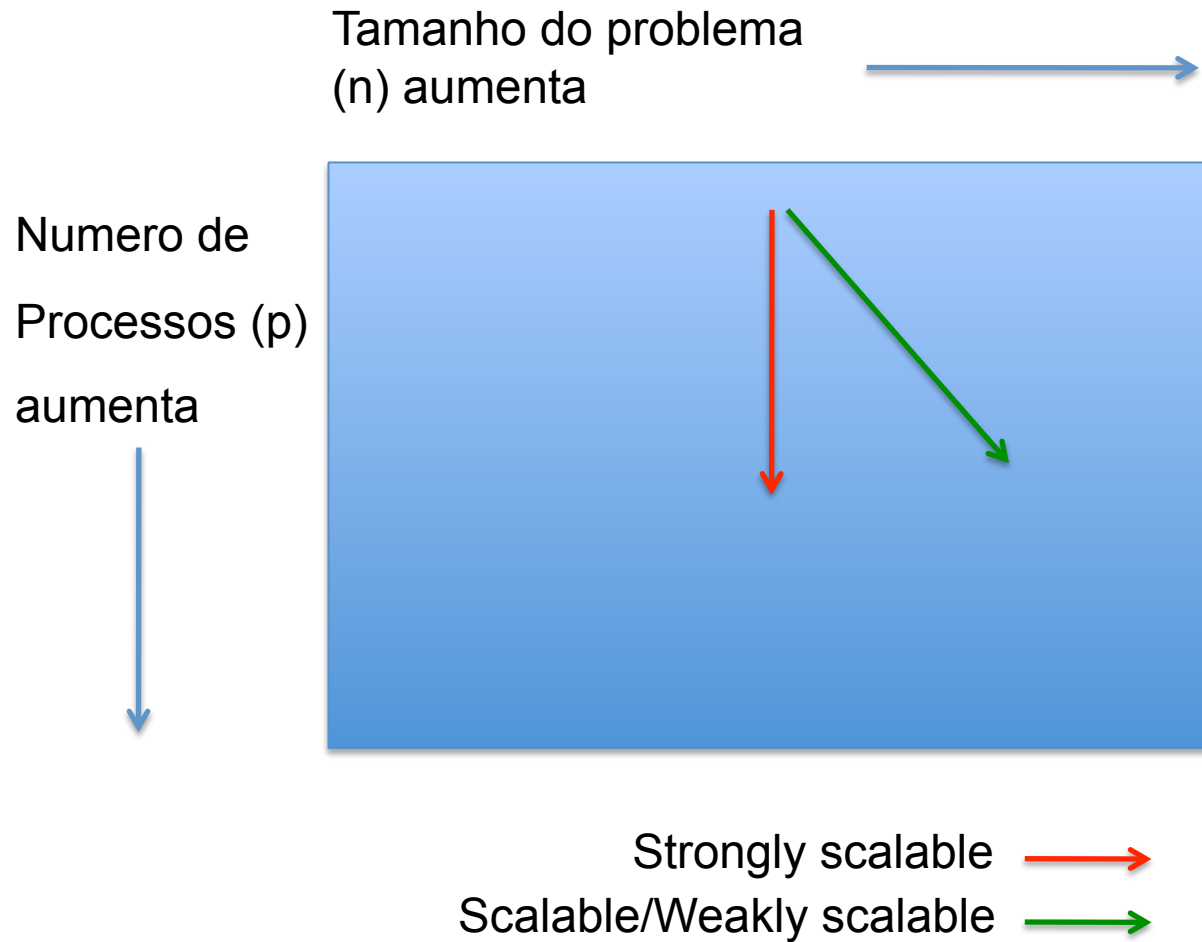
- Um programa é **escalável** se o **tamanho total do problema** pode ser aumentado em uma taxa tal que a eficiência não diminui com o aumento no número processadores:



Escalabilidade

- Programas que conseguem manter uma eficiência constante sem aumentar o **tamanho total do problema** são, às vezes, chamados **strongly scalable**: $E = [t_1 / (t_1 / p)] / p = 1$
- Programas que conseguem manter uma eficiência constante se o **tamanho do problema por nó** aumenta na mesma taxa que o número de processos são, às vezes, chamados de **weakly scalable**: $E = [p * t_1 / (p * t_1 / p)] / p = 1$

Scalable vs. Strong Scalable



ALGORITMO DE ORDENAÇÃO PARALELA

Sorting

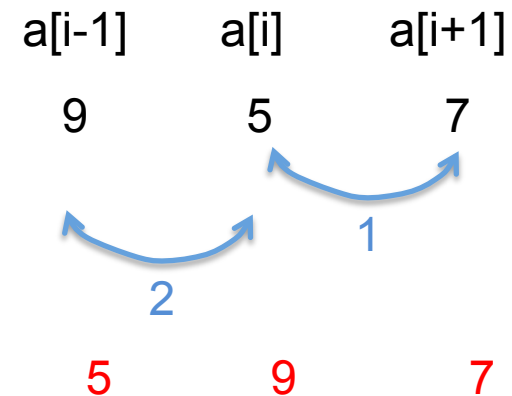
- n chaves e $p = \text{comm sz processes}$.
- n/p chaves atribuídas a cada processo.
- Nenhuma restrição a quais chaves são atribuídas a quais processos.
- Quando o algoritmo termina:
 - As chaves atribuídas a cada processo devem ser ordenadas em (digamos) ordem crescente.
 - Se $0 \leq q < r < p$, então cada chave atribuída ao processo q deve ser menor ou igual a cada chave atribuída ao processo r .

Bubble sort serial



```
void Bubble_sort(  
    int a[] /* in/out */,  
    int n /* in */) {  
    int list_length, i, temp;  
  
    for (list_length = n; list_length >= 2; list_length--)  
        for (i = 0; i < list_length-1; i++)  
            if (a[i] > a[i+1]) {  
                temp = a[i];  
                a[i] = a[i+1];  
                a[i+1] = temp;  
            }  
    }  
    /* Bubble_sort */
```

Funciona ?



Odd-even transposition sort

- Uma sequência de fases.
- Nas fases pares *compare-swap*:

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

- Nas fases ímpares *compare-swap*:

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

Exemplo

Início: 5, 9, 4, 3

Fase par: *compare-swap* (5,9) e (4,3)
resultando na lista 5, 9, 3, 4

Fase ímpar: *compare-swap* (9,3)
resultando na lista 5, 3, 9, 4

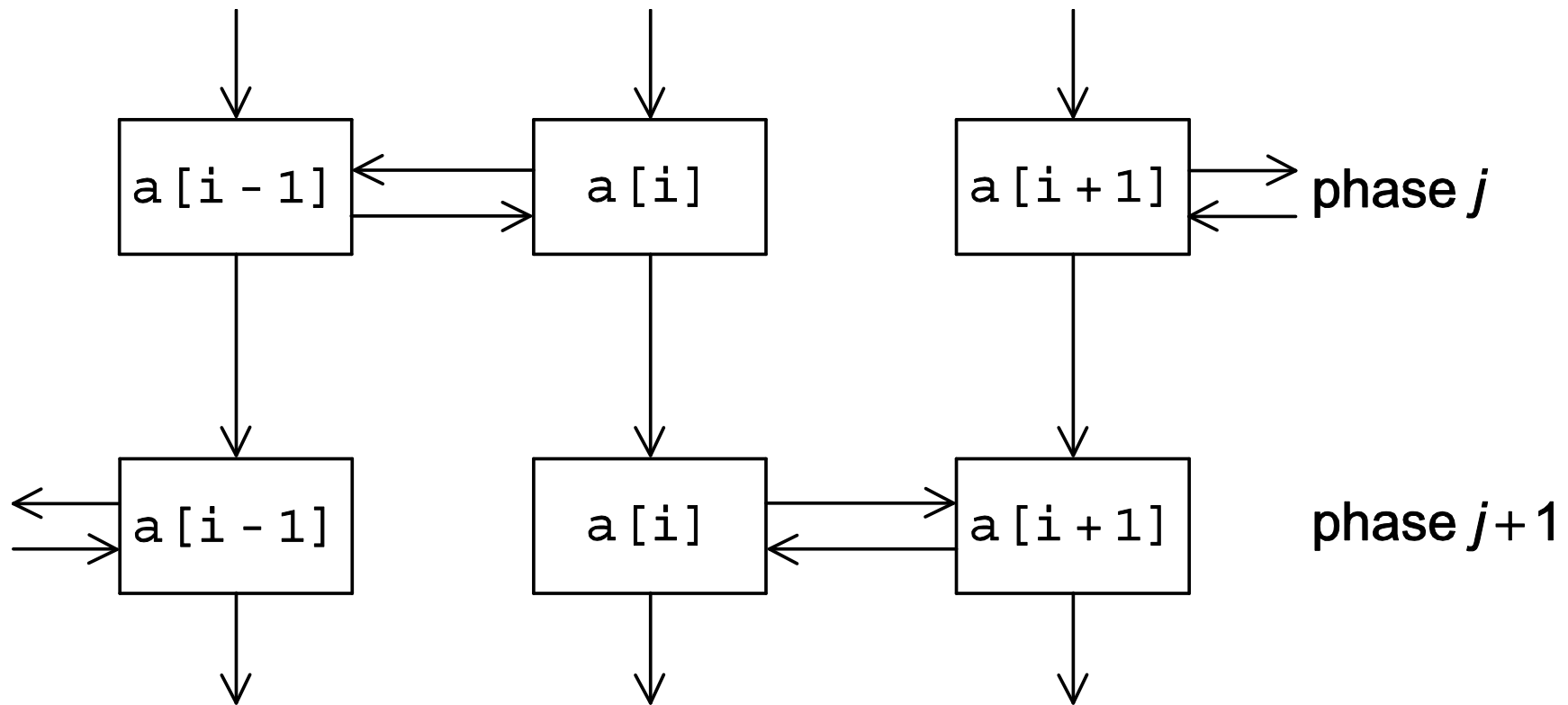
Fase par: *compare-swap* (5,3) and (9,4)
resultando na lista 3, 5, 4, 9

Fase ímpar: *compare-swap* (5,4)
resultando na lista 3, 4, 5, 9

Serial odd-even transposition sort

```
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 */
```

Comunicações entre tarefas da ordenação *odd-even*



Tasks determining $a[i]$ are labeled with $a[i]$.

Parallel odd-even transposition sort

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

Pseudo-código

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

Compute_partner

```
if (phase % 2 == 0)           /* Even phase */
    if (my_rank % 2 != 0)      /* Odd rank */
        partner = my_rank - 1;
    else                       /* Even rank */
        partner = my_rank + 1;
else                          /* Odd phase */
    if (my_rank % 2 != 0)      /* Odd rank */
        partner = my_rank + 1;
    else                       /* Even rank */
        partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
    partner = MPI_PROC_NULL;
```


Segurança em programas MPI

- O padrão MPI permite `MPI_Send` se comportar de duas formas diferentes:
 - ele pode simplesmente copiar a mensagem em um *buffer* gerenciado pelo MPI e retornar,
 - ou ele pode bloquear até que uma chamada de casamento para `MPI_Recv` inicie.

Segurança em programas MPI

- Muitas implementações de MPI definem um limiar no qual o sistema muda de *buffering* para *blocking*.
- Mensagens relativamente pequenas serão armazenadas por `MPI_Send`, que continua.
- Mensagens maiores irão fazer o programa bloquear.

Segurança em programas MPI

- Assuma que dois processos executando o mesmo código.
- O que vai ocorrer em uma mensagem pequena?

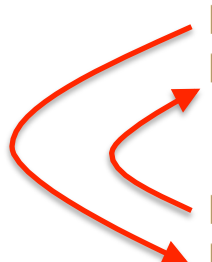
```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,  
         MPI_STATUS_IGNORE);
```

Funciona!!

```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,  
         MPI_STATUS_IGNORE);
```

Segurança em programas MPI

- Assuma que dois processos executando o mesmo código.
- E com uma mensagem grande?

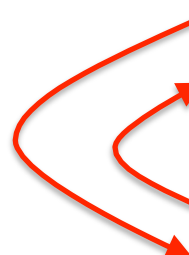


```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,  
         MPI_STATUS_IGNORE);  
  
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,  
         MPI_STATUS_IGNORE);
```

Deadlock!!

Segurança em programas MPI

- Se as chamadas de `MPI_Send` executadas por cada processo bloquearem, nenhum processo será capaz de iniciar a chamada a `MPI_Recv` e o programa irá travar ou entrar em **deadlock**.
- Cada processo irá ficar em espera (bloqueante) por um evento que nunca chegará.



```
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,  
MPI_STATUS_IGNORE);  
  
MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,  
MPI_STATUS_IGNORE);
```

Segurança em programas MPI

- Um programa que precisa de MPI para prover *buffering*, e resolver este tipo de problema, é dito ser **unsafe**.
- Tal programa pode executar sem problemas para vários conjuntos de entradas, mas pode também travar ou quebrar para outros conjuntos.
- Para conjuntos de entradas grandes ele bloqueia...

MPI_Ssend

- Uma alternativa para [MPI_Send](#) disponível na biblioteca de MPI.
- O “s” extra vem de *synchronous*, e MPI_Ssend irá bloquear até que a recepção equivalente inicie.

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

Segurança em programas MPI

- Outra situação em que pode ocorrer problema
 - Comunicação feita em anel.
- `my_rank` envia para `(my_rank + 1) % comm_sz`

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
0, comm, MPI_STATUS_IGNORE.
```


Restruturando a comunicação

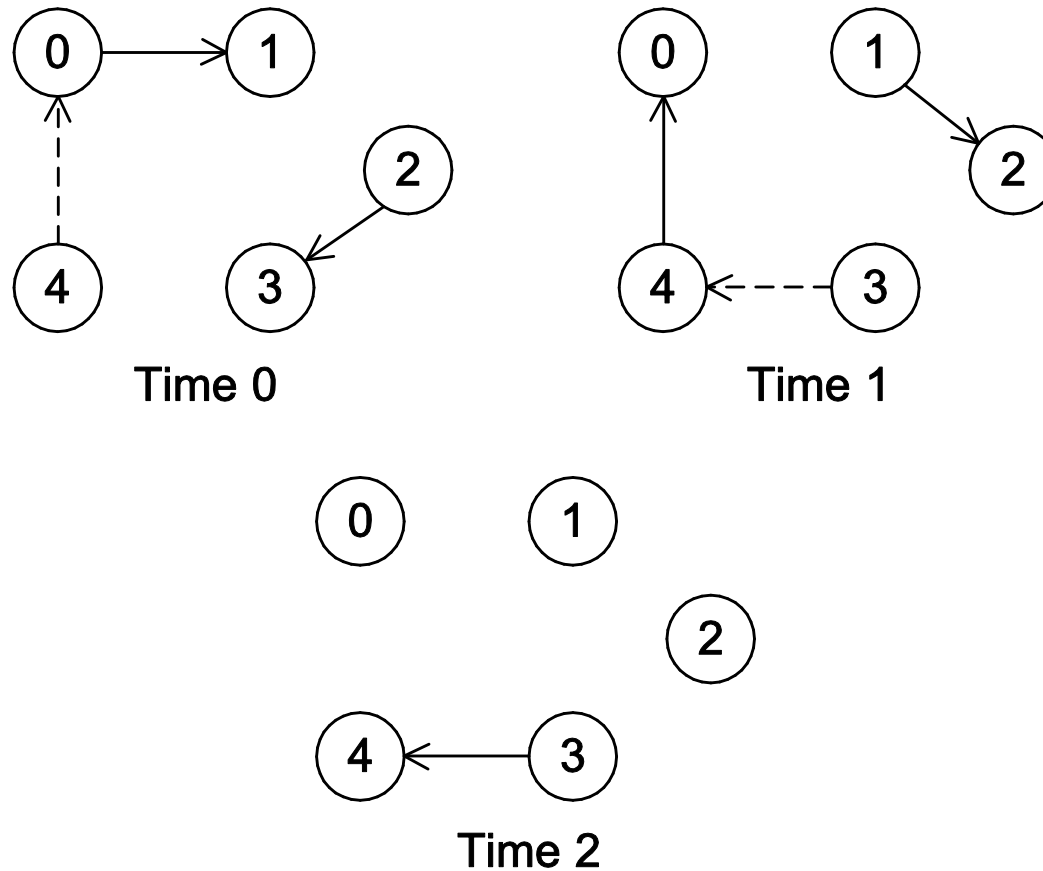
```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
0, comm, MPI_STATUS_IGNORE.
```



O que fazer então?

```
if (my_rank % 2 == 0) {  
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
0, comm, MPI_STATUS_IGNORE.  
} else {  
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
0, comm, MPI_STATUS_IGNORE.  
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
}
```

Comunicação segura com 5 processos



MPI_Sendrecv

- Uma alternativa para escalonar a comunicação nós mesmos.
- Executa um envio bloqueante e uma recepção em uma única chamada.
- O destino e a fonte podem ser os mesmos ou diferentes.
- Especialmente útil pois MPI escalona a comunicação de modo que os programas não travam ou quebram.

MPI_Sendrecv

```
int MPI_Sendrecv(  
    void*          send_buf_p      /* in   */,  
    int           send_buf_size   /* in   */,  
    MPI_Datatype   send_buf_type   /* in   */,  
    int           dest             /* in   */,  
    int           send_tag         /* in   */,  
    void*          recv_buf_p      /* out  */,  
    int           recv_buf_size    /* in   */,  
    MPI_Datatype   recv_buf_type   /* in   */,  
    int           source           /* in   */,  
    int           recv_tag         /* in   */,  
    MPI_Comm       communicator    /* in   */,  
    MPI_Status*    status_p        /* in   */);
```

Última tarefa: organizar as listas em cada processo

```
void Merge_low(  
    int  my_keys[],      /* in/out    */  
    int  recv_keys[],    /* in       */  
    int  temp_keys[],    /* scratch  */  
    int  local_n         /* = n/p, in */) {  
    int  m_i, r_i, t_i;  
  
    m_i = r_i = t_i = 0;  
    while (t_i < local_n) {  
        if (my_keys[m_i] <= recv_keys[r_i]) {  
            temp_keys[t_i] = my_keys[m_i];  
            t_i++; m_i++;  
        } else {  
            temp_keys[t_i] = recv_keys[r_i];  
            t_i++; r_i++;  
        }  
    }  
  
    for (m_i = 0; m_i < local_n; m_i++)  
        my_keys[m_i] = temp_keys[m_i];  
} /* Merge_low */
```

O que faz este código?

Run-times of parallel odd-even sort

Processes	Number of Keys (in thousands)				
	200	400	800	1600	3200
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

(times are in milliseconds)

Comentários finais (1)

- MPI (Message-Passing Interface) que podem ser chamadas por programas C, C++, ou Fortran.
- Um comunicador é um conjunto de processos que pode enviar mensagens entre si.
- Muitos programas paralelos usam uma abordagem Single-Program Multiple Data (SPMD).

Comentários finais (2)

- A maioria dos programas seriais são determinísticos: se executarmos o mesmo programa com os mesmos dados obteremos as mesmas respostas.
- Programas paralelos usualmente não têm esta propriedade.
- Comunicação coletiva envolve todos os processos de um comunicador.

Comentários finais (3)

- Quando medimos o tempo de programas paralelos, usualmente estamos interessados no “elapsed time” ou “wall clock time”.
- *Speedup* é a razão entre o tempo de execução serial pelo tempo de execução paralela.
- Eficiência é o *speedup* dividido pelo número de processos em paralelo.

Comentários finais (4)

- Se, para um dado programa, for possível aumentar o tamanho do problema (n) de modo que a eficiência não diminua com o aumento de p então o programa é dito escalável.
- Um programa MPI é inseguro se o seu comportamento correto depende do fato de que a sua entrada precisa ser *buffered* por `MPI_Send`.