

Programação de Arquiteturas com Memória Distribuída Com MPI

MCZA020-13 - Programação Paralela

Emilio Francesquini

e.francesquini@ufabc.edu.br

2025.Q2

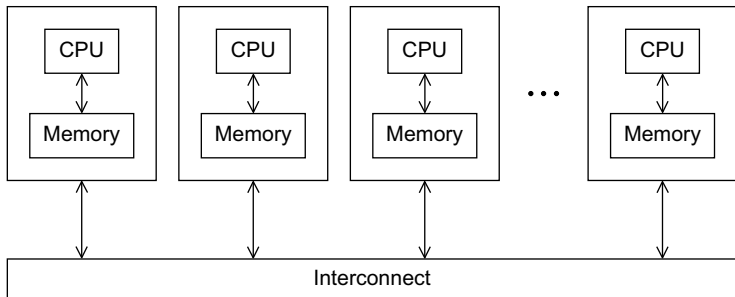
Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

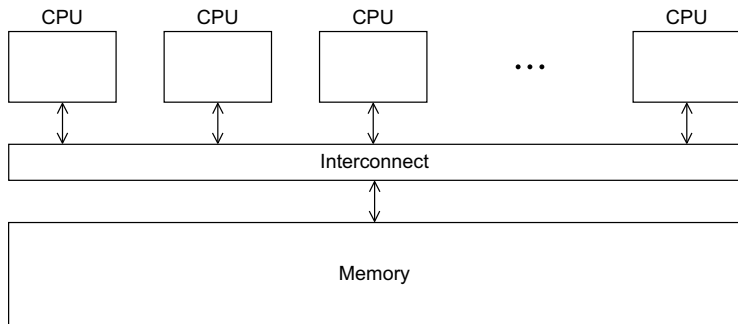


- Estes slides foram preparados para o curso de **Programação Paralela na UFABC**.
- Estes slides são baseados naqueles produzidos por Peter Pacheco como parte do livro *An Introduction to Parallel Programming* disponíveis em:
<https://www.cs.usfca.edu/~peter/ipp/>
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Algumas figuras foram obtidas em: <http://pngimg.com>



MPI





```
1  #include <stdio.h>
2
3  int main(void) {
4      printf("hello, world\n");
5      return 0;
6  }
```

Um clássico!

- Cada linha de execução no MPI é chamada de processo
- Processos são identificados por um número inteiro não negativo
- A identificação de um processo é chamada de **rank** do processo
- Um programa MPI com p processos tem processos com ranks de $0, 1, \dots, p - 1$

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <mpi.h>
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8      char      msg[MAX_STRING];
9      int       comm_sz;
10     int       my_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     ...
```

```
1  ...
2  if (my_rank != 0) {
3      sprintf(msg, "Greetings from process %d of %d!",
4                my_rank, comm_sz);
5      MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 0, 0,
6                MPI_COMM_WORLD);
7  } else {
8      printf("Greetings from process %d of %d!\n",
9             ↪ my_rank, comm_sz);
10     for (int q = 1; q < comm_sz; q++) {
11         MPI_Recv(msg, MAX_STRING, MPI_CHAR, q,
12                  0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13         printf("%s\n", msg);
14     }
15     ...
```

```
1     ...  
2     MPI_Finalize();  
3     return 0;  
4 }
```

```
1 $ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

- **mpicc** - É um *wrapper* para o compilador do sistema (tipicamente o **gcc**)
 - ▶ **mpicc --showme** - Mostra o real comando que será executado
- **-g, -Wall, -o** - Flags com significado padrão entre os compiladores C

```
1 mpirun/mpiexec -np <número de processos> <executável>
```

```
1 $ mpirun -np 1 ./mpi_hello
```

```
2
```

```
3 $ mpirun -np 8 ./mpi_hello
```

- **mpirun** ou **mpiexec** - Assim como o **mpicc** é um wrapper para executar o programa
- Muitos parâmetros opcionais disponíveis
 - ▶ Há opções para escolher quais e quantos processadores/cores, quantidade de threads, processos, ...

```
1  $ mpiexec -np 1 ./mpi_hello
2  Greetings from process 0 of 1!
3  $ mpiexec -np 2 ./mpi_hello
4  Greetings from process 0 of 2!
5  Greetings from process 1 of 2!
6  $ mpiexec -np 4 ./mpi_hello
7  -----
8  There are not enough slots available in the system to
9  satisfy the 4 slots that were requested by the
10 application:
11     ./mpi_hello
12
13 Either request fewer slots for your application, or
14 make more slots available for use.
15 -----
16 $
```

- O ambiente de execução determinou que não há **slots** disponíveis
 - ▶ Neste caso o ambiente detectou 2 processadores e portanto 4 está além do que há disponível
- Slots são configurados pelo administrador do sistema
- É possível colocar mais processos do que slots (**oversubscription**) utilizando o parâmetro **--oversubscribe**

```
1 $ mpiexec --oversubscribe -np 5 ./mpi_hello
2 Greetings from process 0 of 5!
3 Greetings from process 1 of 5!
4 Greetings from process 2 of 5!
5 Greetings from process 3 of 5!
6 Greetings from process 4 of 5!
7 $
```

- E o que acontece se executarmos sem o `mpirun`

```
1 $ ./mpi_hello
```

- E o que acontece se executarmos sem o `mpirun`

```
1 $ ./mpi_hello
```

- Ele simplesmente roda como uma aplicação sequencial

```
1 $ ./mpi_hello
2 Greetings from process 0 of 1!
3 $
```

- São escritos em C
 - ▶ Como qualquer programa em C têm função `main`
 - ▶ Podem usar bibliotecas (mais comuns `stdio.h`, `stdlib.h`, ...)
 - ▶ Precisam incluir `mpi.h`
- Todos os identificadores da biblioteca MPI começam com `MPI_`
 - ▶ Por padrão a primeira letra após o *underscore* é maiúscula
 - ▶ Esta padronização ajuda a evitar confusões

MPI_INIT

- Inicializa o MPI (pode receber parâmetros adicionais)

```
1 int MPI Init(  
2     int* argc_p /* in/out */,  
3     char*** argv_p /* in/out */);
```

- `argc_p` e `argv_p` são ponteiros para os parâmetros recebidos pela `main`

MPI_Finalize

- Indica ao ambiente de execução do MPI que o programa acabou e inicia a liberação dos recursos alocados

```
1 int MPI Finalize(void);
```

```
1  ...
2  #include <mpi.h>
3  ...
4  int main(int argc, char *argv[]) {
5      /* Nenhuma chamada MPI pode ser feita antes da
6         ↳ inicialização */
7      MPI_Init(&argc, &argv);
8      ...
9      MPI_Finalize();
10     /* Nenhuma chamada MPI pode ser feita após a
11        ↳ finalização */
12     return 0;
13 }
```

- Um comunicador (*communicator*) é um conjunto de processos que podem mandar mensagens uns para os outros
- A função **MPI_Init** cria um comunicador padrão que consiste de todos os processos criados quando o programa é executado
- Este comunicador é chamado **MPI_COMM_WORLD**

MPI_Comm_size

- Para obter o número de processos participantes do comunicador

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

MPI_Comm_rank

- Devolve o rank do processo que chamou a função

```
1 int MPI_Comm_rank(  
2     MPI_Comm comm      /* in */,  
3     int*      my_rank  /* out */);
```

- Todas as funções MPI (com exceção de **MPI_Wtime** and **MPI_Wtick**) devolvem como retorno um código de erro
 - ▶ É prudente sempre verificar o retorno
- Valores notáveis
 - ▶ **MPI_SUCCESS** - Indica sucesso na execução
 - ▶ **MPI_ERR_COMM** - Comunicador inválido. É um erro comum enviar o valor **NULL** como comunicador. Isto não é permitido nem mesmo para chamadas como **MPI_Comm_rank**.

- Compilamos **um único** programa
- O processo com rank 0 é diferente (usamos um **if** para fazer isso)
 - ▶ O rank 0 recebe mensagens e as imprime enquanto os outros processos trabalham
- Não há nada especial com o rank 0 para o MPI. Mas é muito conveniente usar o rank 0 e não o rank 1, 2, 3... **Por quê?**
- O uso de condicionais como **if-else** torna o nosso programa SPMD.

```
1  int MPI_Send(  
2      void*      msg_buf_p    /* in */,  
3      int        msg_size     /* in */,  
4      MPI_Datatype msg_type    /* in */,  
5      int        dest         /* in */,  
6      int        tag          /* in */,  
7      MPI_Comm   communicator /* in */);
```

Table 3.1 Some Predefined MPI Datatypes

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	

```
1  int MPI_Recv(  
2      void*      msg_buf_p    /* out */,  
3      int        buf_size     /* in  */,  
4      MPI_Datatype buf_type    /* in  */,  
5      int        source       /* in  */,  
6      int        tag          /* in  */,  
7      MPI_Comm   communicator /* in  */,  
8      MPI_Status  status      /* out */);
```

- Para uma mensagem ser entregue, diversos dos seus atributos precisam *casar* (*message matching*)

```
1  /* o processo com rank p roda a linha abaixo */
2  MPI_Send(send_buf_p, send_buf_sz, send_type, dest,
   ↪  send_tag, send_comm);
3
4  /* o processo com rank q roda a linha abaixo */
5  MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src,
   ↪  recv_tag, recv_comm, &status);
```

- Tomando o código acima como exemplo, uma mensagem enviada por **q** só será recebida por **r** se:
 - ▶ `recv_comm == send_comm` e;
 - ▶ `recv_tag == send_tag` e;
 - ▶ `dest == r` e;
 - ▶ `src == q`

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

MPI_Send

src = *q*



MPI_Recv

dest = *r*

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

- O `MPI_Recv` tem como parâmetros de saída apenas o buffer e o status
- A variável de status do tipo `MPI_Status` contém
 - ▶ O tamanho da mensagem
 - ▶ A origem (!) da mensagem
 - ▶ A tag (!) da mensagem



- O `MPI_Recv` tem como parâmetros de saída apenas o buffer e o status
- A variável de status do tipo `MPI_Status` contém
 - ▶ O tamanho da mensagem
 - ▶ A origem (!) da mensagem
 - ▶ A tag (!) da mensagem



- ▶ Existem parâmetros especiais `MPI_ANY_SOURCE` e `MPI_ANY_TAG` que aceitam mensagens vindas de qualquer fonte e qualquer tag

- `MPI_Status` é uma struct com pelo menos os seguintes campos
 - ▶ `MPI_SOURCE`
 - ▶ `MPI_TAG`
 - ▶ `MPI_ERROR`
- Quantos dados foram recebidos?

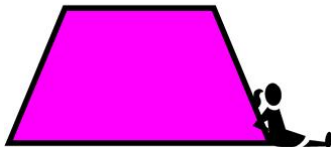
```
1  int MPI_Get_count(  
2      MPI_Status* status_p /* in */,  
3      MPI_Datatype type    /* in */,  
4      int* count_p /* out */);
```

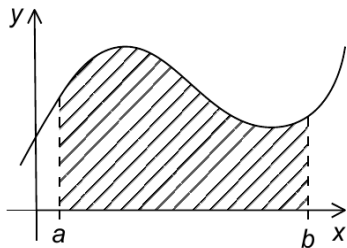
```
1 MPI_Status status;
2 MPI_Recv(recv_buf_p, recv_buf_sz, recv_type,
   ↪ MPI_ANY_SOURCE, MPI_ANY_TAG, recv_comm, &status);
3
4 int src = status.MPI_SOURCE;
5 int tag = status.MPI_TAG;
6 int count;
7 MPI_Get_count(&status, recv_type, &count);
8 ...
```

- O comportamento exato do programa é determinado pela implementação de MPI escolhida
- **MPI_Send** pode se comportar de maneira diferente com relação ao tamanho do *buffer*, *cutoffs* ou *blocking*
- **MPI_Recv** sempre bloqueia até que uma mensagem que case seja recebida
- Estude a implementação que você escolher! Não assumam nada!

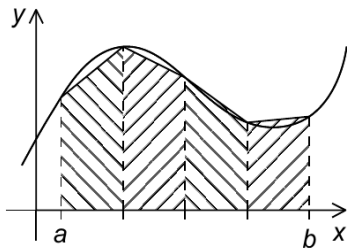
Aproximação trapezoidal em MPI

- A ideia é aproximar o valor da integral de uma função $f(x)$ através da soma das áreas de um número crescente de trapézios
- Quanto maior o número de trapézios
 - ▶ Melhor a aproximação
 - ▶ Maior custo computacional

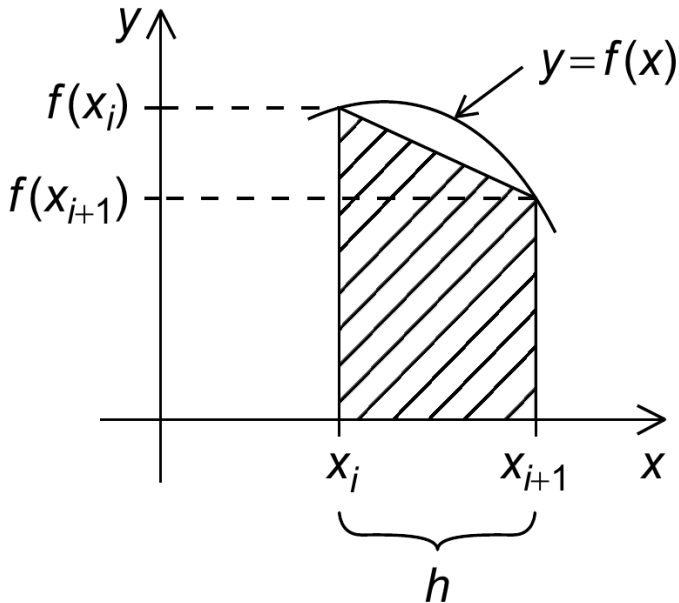




(a)



(b)



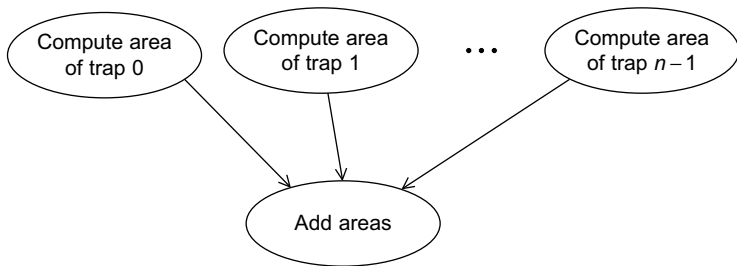
- Área de um trapézio $= \frac{h}{2}[f(x_i) + f(x_{i+1})]$
- $h = \frac{b-a}{n}$
- $x_0 = a, x_1 = a + h, x_2 = a + 2h, \dots, x_{n-1} = a + (n-1)h, x_n = b$
- Soma de todas as áreas dos trapézios
 $= h[f(x_0)/2 + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + f(x_n)/2]$

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

- 1 Particione o problema em tarefas
- 2 Identifique as comunicações entre as tarefas
- 3 Agregue as tarefas em tarefas compostas
- 4 Mapeie a execução das tarefas aos cores

```
Get a, b, n;  
h = (b - a)/n;  
local_n = n/comm_sz;  
local_a = a + my_rank * local_n * h;  
local_b = local_a + local_n * h;  
local_integral = Trap(local_a,local_b,local_n, h);
```

```
if (my_rank != 0)
    Send local_integral to process 0;
else { /* my_rank == 0 */
    total_integral = local_integral;
    for (proc = 1; proc < comm_sz; proc++) {
        Receive local_integral from proc;
        total_integral += local_integral;
    }
}
if (my_rank == 0)
    print result;
```



```
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     ...
```

```
1  ...
2
3  h = (b-a)/n;
4  local_n = n/comm_sz;
5  local_a = a + my_rank*local_n*h;
6  local_b = local_a + local_n*h;
7  local_int = Trap(local_a, local_b, local_n, h);
8
9  ...
```

```
1  ...
2
3  /* Soma as integrais parciais calculadas por cada um dos
4  processos */
5  if (my_rank != 0) {
6      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
7              MPI_COMM_WORLD);
8  } else {
9      total_int = local_int;
10     for (source = 1; source < comm_sz; source++) {
11         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
12                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
13         total_int += local_int;
14     }
15 }
16
17  ...
```

```
1      ...
2
3      /* Imprime o resultado */
4      if (my_rank == 0) {
5          printf("With n = %d trapezoids, our estimate\n",
6              ↪ n);
7          printf("of the integral from %f to %f =
8              ↪ %.15e\n",
9              a, b, total_int);
10     }
11
12     MPI_Finalize();
13     return 0;
14 } /* main */
```

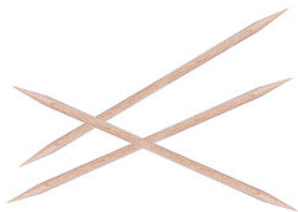
```
1  double Trap(double left, double right, int n_traps,  
   ↪  double base_len) {  
2  
3      double estimate, x;  
4      int i;  
5  
6      estimate = (f(left) + f(right))/2.0;  
7      for (i = 1; i <= n_traps-1; i++) {  
8          x = left + i*base_len;  
9          estimate += f(x);  
10     }  
11     estimate = estimate*base_len;  
12  
13     return estimate;  
14 }
```

Considere o seguinte programa. O que ele imprime?

```
1  #include <stdio.h>
2  #include <mpi.h>
3  int main(void) {
4      int my_rank, comm_sz;
5      MPI_Init(NULL, NULL);
6      MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
7      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
8      printf("Proc %d of %d > Does anyone have a
   ↳ toothpick?\n", my_rank, comm_sz);
9      MPI_Finalize();
10     return 0;
11 }
```

```
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 imprevisível!



- A maior parte das implementações de MPI só permite acesso à **stdin** pelo processo com rank 0 relativo ao comunicador **MPI_COMM_WORLD**
- O processo 0 deve ler a entrada (com **scanf** por exemplo) e enviar o que for necessário aos demais processos

```
1  ...
2  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
3  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
4  Get_data(my_rank, comm_sz, &a, &b, &n);
5  h = (b - a)/n;
6  ...
```

```
1 void Get_input(int my_rank, int comm_sz, double* a_p,  
  ↪ double* b_p, int* n_p) {  
2     int dest;  
3     /* caso seja o processo mestre */  
4     if (my_rank == 0) {  
5         printf("Enter a, b, and n\n");  
6         scanf("%lf %lf %d", a_p, b_p, n_p);  
7         for (dest = 1; dest < comm_sz; dest++) {  
8             MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0,  
  ↪ MPI_COMM_WORLD);  
9             MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0,  
  ↪ MPI_COMM_WORLD);  
10            MPI_Send(n_p, 1, MPI_INT, dest, 0,  
  ↪ MPI_COMM_WORLD);  
11        }  
12    } else { /* my_rank != 0 */  
13        ...
```

```
1  ...
2      } else { /* my_rank != 0 */
3          MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0,
4              ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
5          MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0,
6              ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7          MPI_Recv(n_p, 1, MPI_INT, 0, 0,
8              ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
9      }
10 }
```

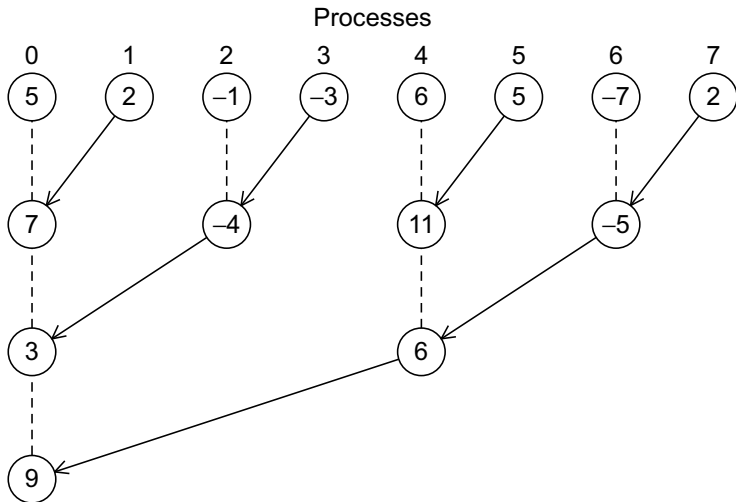
Comunicação coletiva

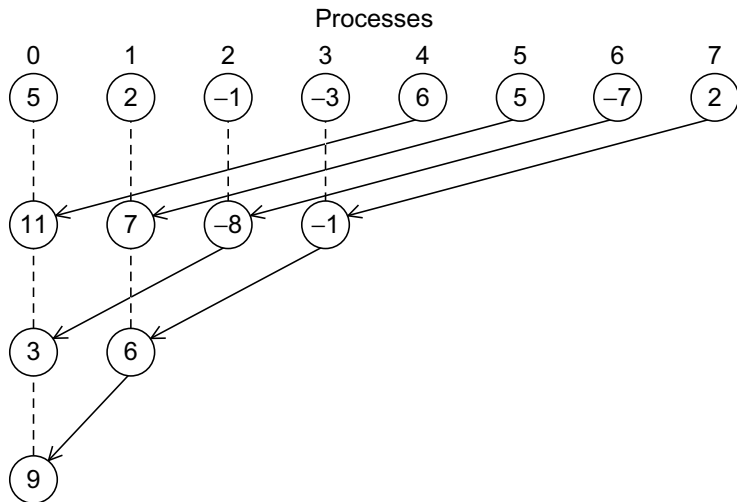
1 Primeira fase

- 1 Processo 1 envia ao 0, 3 ao 2, 5 ao 4, e 7 ao 6
- 2 Processos 0, 2, 4 e 6 acumulam os valores recebidos
- 3 Processos 2 e 6 enviam os valores calculados aos processos 0 e 4 respectivamente
- 4 Processos 0 e 4 acumulam os valores recebidos

2 Segunda fase

- 1 Processo 4 envia o valor calculado ao processo 0
- 2 Processo 0 acumula o valor recebido ao seu valor calculado mais recente





```
1  int MPI Reduce(  
2      void*      input_data_p  /* in */,  
3      void*      output_data_p /* out */,  
4      int        count         /* in */,  
5      MPI_Datatype datatype     /* in */,  
6      MPI_Op      operator      /* in */,  
7      int        dest_process  /* in */,  
8      MPI_Comm    comm         /* in */);
```

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

```
1  double local x[N], sum[N];  
2  ...  
3  MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,  
    ↪ MPI_COMM_WORLD);
```

Table 3.2 Predefined Reduction Operators in MPI

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

- **Todos** os processos do comunicador devem chamar a **mesma** função coletiva
- Por exemplo, um programa que chama **MPI_Reduce** em um processo enquanto outro processo faz uma chamada ao **MPI_Recv** está incorreto e muito provavelmente vai travar (entrar em um impasse) ou até mesmo capotar¹

¹Termo não tão técnico contudo também é uma excelente tradução de *crash*

- Assim como na comunicação ponto a ponto, os argumentos das chamadas às funções coletivas tem que *casar*
- Por exemplo, se um processo passa 0 como **dest_process** e outro passa 1, então a saída da chamada ao **MPI_Reduce** não vai casar causando capotes² ou até mesmo impasses.

²Outro termo muito técnico

- O argumento `output_data_p` é utilizado apenas no `dest_process`
- Contudo, todos os processos precisam completar o parâmetro com algo (dada a sintaxe da linguagem C). Como o parâmetro será ignorado nos processos não destino, pode-se enviar **NULL** sem problemas.

- Comunicações ponto-a-ponto são casadas baseando-se em tags e comunicadores
- Comunicações coletivas não usam tags → Comunicações coletivas são casadas baseando-se, apenas, na ordem em que são chamadas

Table 3.3 Multiple Calls to MPI_Reduce

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

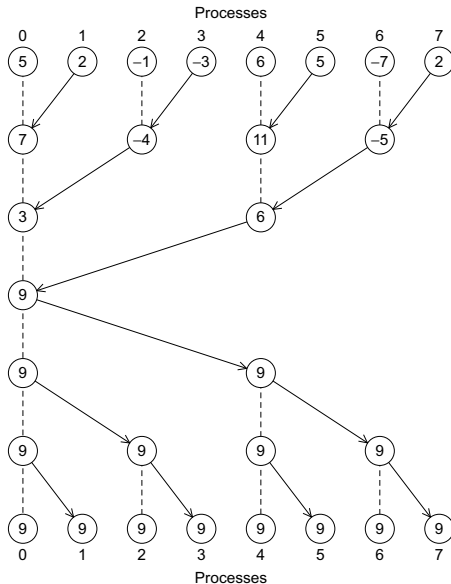
Múltiplas chamadas à MPI_Reduce

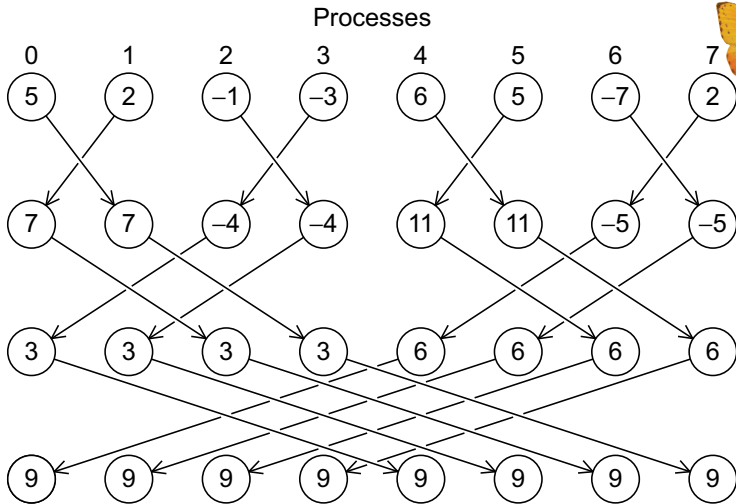
- Suponha que cada processo chame **MPI_Reduce** com o operador **MPI_SUM** e com o processo de destino 0
- À primeira vista pode parecer que após as duas chamadas à **MPI_Reduce**, o valor **b** será 3 e o valor **d** será 6

- Note, contudo, que as localizações da memória e o nome das variáveis são irrelevantes para o casamento das chamadas do **MPI_Reduce**
- A ordem das chamadas vai determinar o casamento, logo o valor em **b** será $1 + 2 + 1 = 4$ e o valor armazenado em **d** será $2 + 1 + 2 = 5$

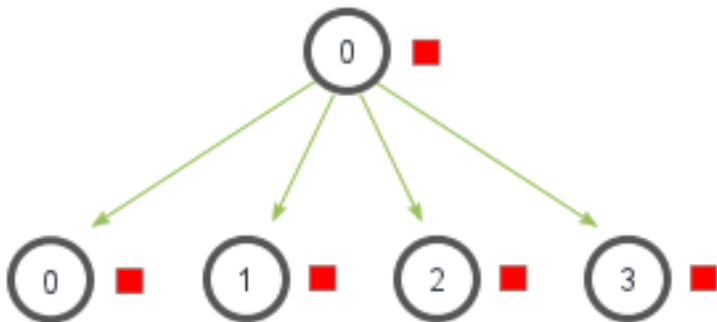
- Útil em uma situação onde todos os processos precisam do resultado de uma redução para continuar a computação

```
1  int MPI_Allreduce(  
2      void*      input_data_p  /* in */,  
3      void*      output_data_p /* out */,  
4      int        count         /* in */,  
5      MPI_Datatype datatype     /* in */,  
6      MPI_Op      operator      /* in */,  
7      MPI_Comm    comm         /* in */);
```





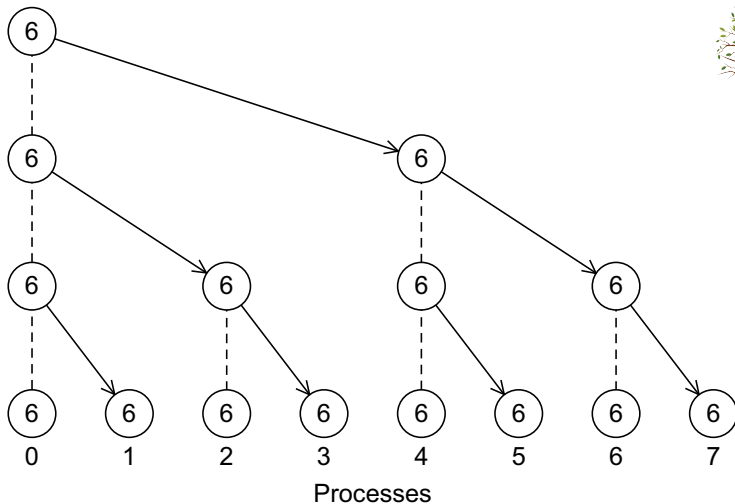
MPI_Bcast



Fonte: <http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>

- Dados que estão em um processo são enviados a todos os processos do mesmo comunicador

```
1  int MPI_Bcast(  
2      void*      data_p      /* in/out */,  
3      int        count      /* in      */,  
4      MPI_Datatype datatype  /* in      */,  
5      int        source_proc /* in      */,  
6      MPI_Comm   comm       /* in      */);
```



```
1 void Get_input(int my_rank, int comm_sz, double* a_p,  
  ↪ double* b_p, int* n_p) {  
2     if (my_rank == 0) {  
3         printf("Enter a, b, and n\n");  
4         scanf("%lf %lf %d", a_p, b_p, n_p);  
5     }  
6     MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
7     MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
8     MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);  
9 }
```

Queremos computar a soma de vetores

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

```
1 void Vector sum(double x[], double y[], double z[], int  
  ↪ n) {  
2     int i;  
3  
4     for (i = 0; i < n; i++)  
5         z[i] = x[i] + y[i];  
6 }
```

Há algumas maneiras bem comuns de particionar os vetores. Abaixo algumas delas.

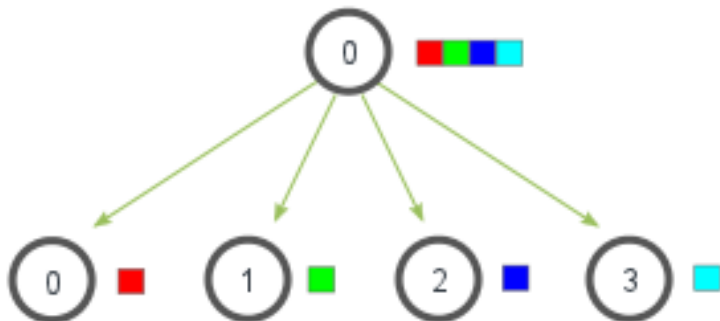
Table 3.4 Different Partitions of a 12-Component Vector among Three Processes

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	+6'	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

- Particionamento em blocos
 - ▶ Atribui-se blocos contendo componentes consecutivos a cada processo
- Particionamento cíclico
 - ▶ Atribui-se os componentes utilizando *round-robin*
- Particionamento bloco-cíclico
 - ▶ Utiliza-se uma atribuição cíclica de blocos

```
1 void Parallel_vector_sum(  
2     double local_x[] /* in */,  
3     double local_y[] /* in */,  
4     double local_z[] /* out */,  
5     int local_n /* in */) {  
6  
7     int local_i;  
8  
9     for (local_i = 0; local_i < local_n; local_i++)  
10         local_z[local_i] = local_x[local_i] +  
11         ↪ local_y[local_i];  
12 }
```

MPI_Scatter



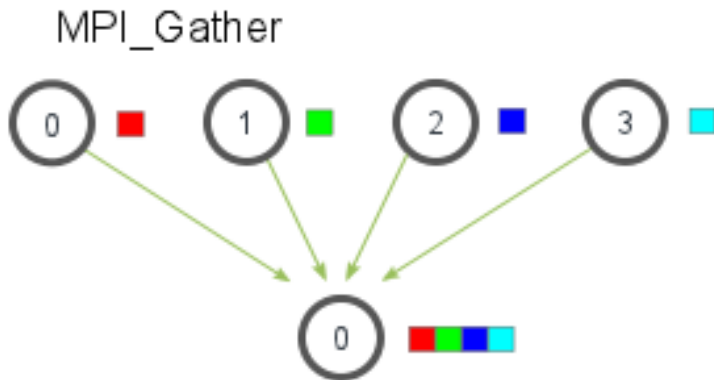
Fonte: <http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>

- A função **MPI_Scatter** pode ser usada, por exemplo, para ler a entrada toda em um processo e distribuir apenas as partes necessárias entre os demais.

```
1  int MPI_Scatter(  
2      void*      send_buf_p /* in */,  
3      int        send_count /* in */,  
4      MPI_Datatype send_type /* in */,  
5      void*      recv_buf_p /* out */,  
6      int        recv_count /* in */,  
7      MPI_Datatype recv_type /* in */,  
8      int        src_proc   /* in */,  
9      MPI_Comm    comm      /* in */);
```

```
1 void Read_vector(  
2     double    local_a[]    /* out */,  
3     int       local_n     /* in  */,  
4     int       n           /* in  */,  
5     char      vec_name[]  /* in  */,  
6     int       my_rank     /* in  */,  
7     MPI_Comm  comm       /* in  */) {  
8  
9     double* a = NULL;  
10    int i;  
11    int local_ok = 1;  
12    char* fname = "Read_vector";  
13  
14    ...
```

```
1      ...
2
3      if (my_rank == 0) {
4          a = malloc(n * sizeof(double));
5          printf("Enter the vector %s\n", vec_name);
6          for (i = 0; i < n; i++)
7              scanf("%lf", &a[i]);
8          MPI_Scatter(a, local_n, MPI_DOUBLE, local_a,
9                    ↪ local_n, MPI_DOUBLE, 0,
10                     comm);
11          free(a);
12      } else {
13          MPI_Scatter(a, local_n, MPI_DOUBLE, local_a,
14                    ↪ local_n, MPI_DOUBLE, 0,
15                     comm);
16      }
17 }
```



Fonte: <http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>

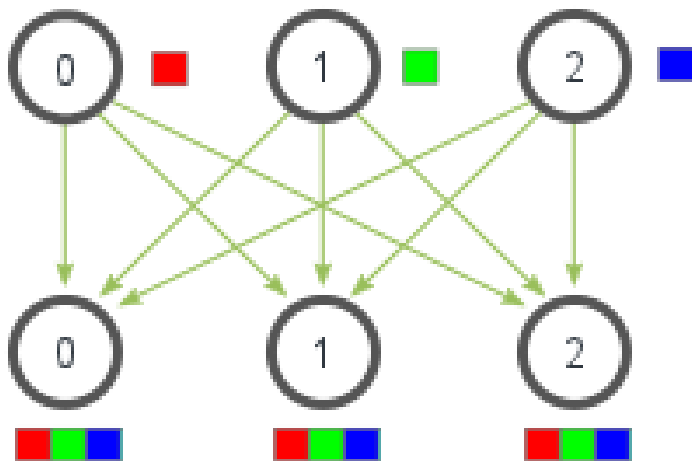
- A função **MPI_Gather** permite coletar todos os componentes de um vetor em processo para que este possa utilizá-los

```
1  int MPI_Gather(  
2      void*      send_buf_p /* in */,  
3      int        send_count /* in */,  
4      MPI_Datatype send_type /* in */,  
5      void*      recv_buf_p /* out */,  
6      int        recv_count /* in */,  
7      MPI_Datatype recv_type /* in */,  
8      int        dest_proc  /* in */,  
9      MPI_Comm   comm       /* in */);
```

```
1 void Print_vector(  
2     double    local_b[] /* in */,  
3     int        local_n  /* in */,  
4     int        n        /* in */,  
5     char       title[]  /* in */,  
6     int        my_rank  /* in */,  
7     MPI_Comm   comm     /* in */) {  
8  
9     double* b = NULL;  
10    int i;  
11  
12    ...
```

```
1      ...
2
3      if (my_rank == 0) {
4          b = malloc(n * sizeof(double));
5          MPI_Gather(local_b, local_n, MPI_DOUBLE, b,
6                  ↪ local_n, MPI_DOUBLE, 0, comm);
7          printf("%s\n", title);
8          for (i = 0; i < n; i++)
9              printf("%f ", b[i]);
10         printf("\n");
11         free(b);
12     } else
13         MPI_Gather(local_b, local_n, MPI_DOUBLE, b,
14                 ↪ local_n, MPI_DOUBLE, 0, comm);
15 }
```

MPI_Allgather



- **MPI_Allgather** concatena o conteúdo de cada um dos processos apontado por **send_buf_p** e armazena em cada um dos processos na posição apontada por **recv_buf_p**
- Como de costume, **recv_count** é a quantidade de dados recebida por cada um dos processos

```
1 int MPI_Allgather(  
2     void*      send_buf_p /* in */,  
3     int        send_count /* in */,  
4     MPI_Datatype send_type /* in */,  
5     void*      recv_buf_p /* out */,  
6     int        recv_count /* in */,  
7     MPI_Datatype recv_type /* in */,  
8     MPI_Comm    comm      /* in */);  
9
```

Multiplicação vetor $\times$ matriz

- Seja
 - ▶ $A = (a_{ij})$, uma matriz $m \times n$
 - ▶ x um vetor com n componentes
 - ▶ $y = Ax$, um vetor com m componentes
- Então, cada um dos componentes y_i de y se dá por
 - ▶ $Y_i = a_{i0}x_0 + a_{i1}x_1 + \dots + a_{i,n-1}x_{n-1}$
 - ▶ Cada Y_i é o produto interno (*dot product*) da i -ésima linha de A por x

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}

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

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

será armazenado na memória como uma sequência

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

```
1 void Mat_vect_mult(  
2     double A[] /* in */,  
3     double x[] /* in */,  
4     double y[] /* out */,  
5     int m /* in */,  
6     int n /* in */) {  
7  
8     int i, j;  
9     for (i = 0; i < m; i++) {  
10         y[i] = 0.0;  
11         for (j = 0; j < n; j++)  
12             y[i] += A[i * n + j] * x[j];  
13     }  
14 }
```

```
1
2 void Mat_vect_mult(
3     double    local_A[] /* in */,
4     double    local_x[] /* in */,
5     double    local_y[] /* out */,
6     int       local_m   /* in */,
7     int       n         /* in */,
8     int       local_n   /* in */,
9     MPI_Comm  comm      /* in */) {
10     double* x;
11     int local_i, j;
12
13     ...
```

```
1      ...
2
3      x = malloc(n * sizeof(double));
4      MPI_Allgather(local_x, local_n, MPI_DOUBLE,
5                    x, local_n, MPI_DOUBLE, comm);
6
7      for (local_i = 0; local_i < local_m; local_i++) {
8          local_y[local_i] = 0.0;
9          for (j = 0; j < n; j++)
10             local_y[local_i] +=
11             ↪ local_A[local_i*n+j]*x[j];
12      }
13      free(x);
14 }
```

Tipos derivados em MPI

- Usados para representar qualquer coleção de dados em memória
 - ▶ Guarda tanto os seus tipos quanto suas posições relativas
- A ideia é a de que se uma função que envie os dados sabe sobre essas informações, ela pode informar o MPI para evitar uma cópia adicional
- Da mesma maneira, uma função que recebe esses dados pode utilizar este mesmo mecanismo

- Mais formalmente, um tipo derivado é composto de uma sequência de dados dos tipos básicos do MPI em conjunto com os seus deslocamentos relativos
- Exemplo com o programa da regra trapezoidal

Variável	Endereço
a	24
b	40
n	48

`{(MPI_DOUBLE, 0), (MPI_DOUBLE, 16), (MPI_INT, 24)}`

- Cria um tipo derivado que consiste de elementos com diferentes tipos básicos do MPI

```
1  int MPI_Type_create_struct(  
2      int          count          /* in */,  
3      int          array_of_blocklengths[] /* in */,  
4      MPI_Aint     array_of_displacements[] /* in */,  
5      MPI_Datatype array_of_types[] /* in */,  
6      MPI_Datatype* new_type_p     /* out */);
```

- Devolve o endereço da memória relativa ao ponteiro `location_p`
- O tipo especial `MPI_Aint` é um inteiro que é grande suficiente para armazenar um ponteiro na máquina

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

- Otimiza a representação interna do tipo pelo MPI para acelerar as comunicações

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

- Quando as comunicações estiverem finalizadas, libera o espaço armazenado pelas estruturas internas do MPI

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

```
1 void Build_mpi_type(  
2     double*      a_p          /* in */,  
3     double*      b_p          /* in */,  
4     int*         n_p          /* in */,  
5     MPI_Datatype* input_mpi_t_p /* out */) {  
6  
7     int array_of_blocklengths[3] = {1, 1, 1};  
8     MPI_Datatype array_of_types[3] = {MPI_DOUBLE,  
9         ↪ MPI_DOUBLE, MPI_INT};  
10    MPI_Aint a_addr, b_addr, n_addr;  
11    MPI_Aint array_of_displacements[3] = {0};  
12    ...
```

```
1      ...
2
3      MPI_Get_address(a_p, &a_addr);
4      MPI_Get_address(b_p, &b_addr);
5      MPI_Get_address(n_p, &n_addr);
6      array_of_displacements[1] = b_addr-a_addr;
7      array_of_displacements[2] = n_addr-a_addr;
8      MPI_Type_create_struct(3, array_of_blocklengths,
9                          array_of_displacements, array_of_types,
10                         input_mpi_t_p);
11      MPI_Type_commit(input_mpi_t_p);
12 }
```

```
1 void Get_input(  
2     int      my_rank    /* in */,  
3     int      comm_sz    /* in */,  
4     double*  a_p        /* out */,  
5     double*  b_p        /* out */,  
6     int*     n_p        /* out */) {  
7     MPI_Datatype input_mpi_t;  
8     Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);  
9  
10    if (my_rank == 0) {  
11        printf("Enter a, b, and n\n");  
12        scanf("%lf %lf %d", a_p, b_p, n_p);  
13    }  
14    MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);  
15    MPI_Type_free(&input_mpi_t);  
16 }
```

Avaliação de desempenho

- A função `MPI_Wtime` devolve o número de segundos desde algum momento arbitrário no passado

```
1  double MPI_Wtime(void);
2
3  double inicio, fim;
4  inicio = MPI_Wtime();
5  /* Faz um monte de coisas interessantes */
6  fim = MPI_Wtime();
7  printf("O código interessante no processador %d levou %e
   ↪ segundos\n", my_rank, finish - start);
```

- Neste caso você não precisa *linkar* com a biblioteca do MPI
- Veja exemplos de como medir o tempo no código disponível na [página da disciplina](#)

- Garante que nenhum processo vai receber o controle de volta (a função bloqueia) até que todos os processos no comunicador tenham alcançado esta chamada

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

O seguinte código pode ser utilizado para medir o tempo de um bloco de código com chamadas MPI e obter uma única medida

```
1  double local_start, local_finish, local_elapsed,  
   ↪  elapsed;  
2  
3  MPI_Barrier(comm);  
4  local_start = MPI_Wtime();  
5  /* Código a ser medido */  
6  local_finish = MPI_Wtime();  
7  local_elapsed = local_finish - local_start;  
8  MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,  
   ↪  MPI_MAX, 0, comm);  
9  if (my_rank == 0)  
10     printf("Tempo decorrido = %e segundos\n", elapsed);
```

Table 3.5 Run-Times of Serial and Parallel Matrix-Vector Multiplication (times are in milliseconds)

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

$$S(n, p) = \frac{T_{\text{sequencial}}(n)}{T_{\text{paralelo}}(n, p)}$$

$$E(n, p) = \frac{S(n, p)}{p} = \frac{\frac{T_{\text{sequencial}}(n)}{T_{\text{paralelo}}(n, p)}}{p} = \frac{T_{\text{sequencial}}(n)}{p \cdot T_{\text{paralelo}}(n, p)}$$

Table 3.6 Speedups of Parallel Matrix-Vector Multiplication

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

Table 3.7 Efficiencies of Parallel Matrix-Vector Multiplication

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

- Um programa é **escalável** se o tamanho do problema pode ser aumentado a uma taxa tal que a eficiência não diminua conforme o número de processos cresce.
- Dizemos que um programa é **fortemente escalável** se ele é capaz de manter a eficiência constante sem que haja necessidade de aumentar o tamanho do problema
- Dizemos que um programa é **fracamente escalável** se ele é capaz de manter a eficiência se o tamanho do problema for aumentado em uma taxa proporcional (=linear) ao aumento do número de processos



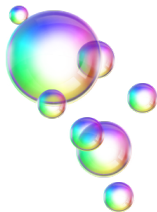
Pergunta

Qual tipo de escalabilidade é apresentada pelo programa de multiplicação matriz/vetor?

Um algoritmo de ordenação paralelo

- n chaves e p (`comm_sz`) processos
- n/p chaves associadas a cada processo
- Não há restrições sobre quais chaves serão associadas a quais processos
- Quando o algoritmo terminar
 - ▶ As chaves associadas a cada processo devem estar armazenadas em ordem
 - ▶ Se $0 \leq q < r < p$, então cada chave associada ao processo q deve ser menor ou igual a todas as chaves associadas ao processo r

```
1 void Bubble_sort(  
2     int a[] /* in/out */,  
3     int n   /* in */) {  
4     int list_length, i, temp;  
5  
6     for (list_length = n; list_length >= 2;  
7         ↪ list_length--)  
8         for (i = 0; i < list_length - 1; i++)  
9             if (a[i] > a[i+1]) {  
10                temp = a[i];  
11                a[i] = a[i+1];  
12                a[i+1] = temp;  
13            }
```



- Funciona em uma sequência de fases
- Fases pares - comparam e trocam iniciando dos índices pares
 - ▶ $(a[0], a[1]), (a[2], a[3]), (a[4], a[5]) \dots$
- Fases ímpares - comparam e trocam iniciando dos índices ímpares
 - ▶ $a[0], (a[1], a[2]), (a[3], a[4]), (a[5], a[6]) \dots$

- Começo: 5, 9, 4, 3
- Fase par: compara e troca (5, 9), (4, 3)
 - ▶ Resultado: 5, 9, 3, 4
- Fase ímpar: compara e troca (9, 3)
 - ▶ Resultado: 5, 3, 9, 4
- Fase par: compara e troca (5, 3), (9, 4)
 - ▶ Resultado: 3, 5, 4, 9
- Fase ímpar: compara e troca (5, 4)
 - ▶ Resultado: 3, 4, 5, 9

```
1 void Odd_even_sort(int a[], int n) {
2     int phase, i, temp;
3     for (phase = 0; phase < n; phase++)
4         if (phase % 2 == 0) { /* Even phase */
5             for (i = 1; i < n; i += 2)
6                 if (a[i - 1] > a[i]) {
7                     temp = a[i];
8                     a[i] = a[i - 1];
9                     a[i - 1] = temp;
10                }
11            } else { /* Odd phase */
12                for (i = 1; i < n - 1; i += 2)
13                    if (a[i] > a[i+1]) {
14                        temp = a[i];
15                        a[i] = a[i+1];
16                        a[i+1] = temp;
17                    }
18            }
19 }
```

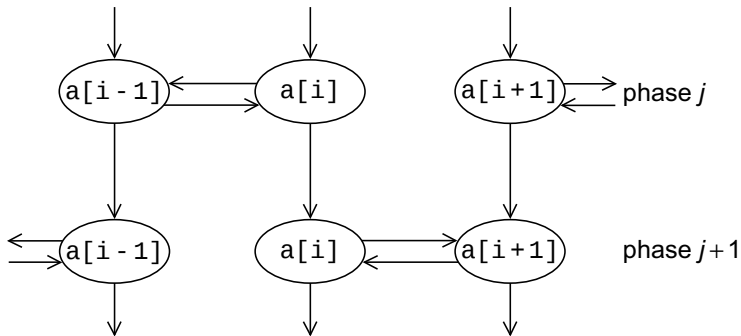


Table 3.8 Parallel Odd-Even Transposition Sort

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

```
Ordena as chaves localmente;
for (fase = 0; fase < comm_sz; fase++) {
    colega = Computa_colega(fase, my_rank);
    if (não estou ocioso) {
        Envia chaves para o colega;
        Recebe chaves do colega;
        if (my_rank < colega)
            Mantem as chaves menores;
        else
            Mantem as chaves maiores;
    }
}
```

```

1  if (phase % 2 == 0) /* Even phase */
2      if (my_rank % 2 != 0) /* Odd rank */
3          partner = my_rank - 1;
4      else /* Even rank */
5          partner = my_rank + 1;
6  else /* Odd phase */
7      if (my_rank % 2 != 0) /* Odd rank */
8          partner = my_rank + 1;
9      else /* Even rank */
10         partner = my_rank - 1;
11  if (partner == - 1 || partner == comm_sz)
12      partner = MPI_PROC_NULL;

```

- O padrão MPI permite ao **MPI_Send** se comportar de duas maneiras distintas
 - ▶ Ele pode simplesmente copiar a mensagem para um buffer do próprio MPI e devolver o controle
 - ▶ Ele pode bloquear a chamada até que a chamada correspondente **MPI_Recv** tenha sido efetuada

- Muitas implementações de MPI tem um limiar (*threshold*) no qual elas trocam de um buffer para a versão bloqueante
- Ou seja, mensagens pequenas tendem a ser *bufferizadas*³ pelo **MPI_Send** e mensagens grandes tendem a bloquear.

³Mais uma tradução de revirar Camões na sua tumba

- Se a chamada à **MPI_Send** executada por cada um dos processos bloqueia, nenhum processo será capaz de iniciar a execução da chamada ao **MPI_Recv**. Nestes casos o programa vai entrar em um **impasse** (*deadlock*).
- Um **impasse** ocorre quando um processo está esperando por algo que nunca acontecerá

- Um programa que se apoia nas características de bufferização do MPI é **inseguro** (*unsafe*)
- O problema é que tal programa pode rodar muitos anos sem demonstrar o problema até que um belo dia ele capota. **Por que?**

- Um método alternativo ao **MPI_Send** definido pelo padrão
- O **s** extra vem de *síncrono* e a função **MPI_Ssend** sempre bloqueia até que o **MPI_Receive** correspondente seja iniciado

```
1  int MPI_Ssend(  
2      void*      msg_buf_p      /* in */,  
3      int        msg_size       /* in */,  
4      MPI_Datatype msg_type     /* in */,  
5      int        dest           /* in */,  
6      int        tag            /* in */,  
7      MPI_Comm   communicator /* in */);
```

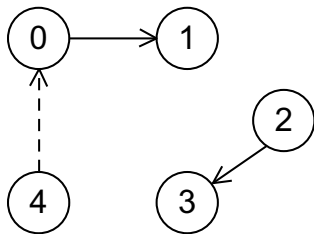
```
1 MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);  
2 MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0,  
  ↪  comm, MPI_STATUS_IGNORE);
```

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

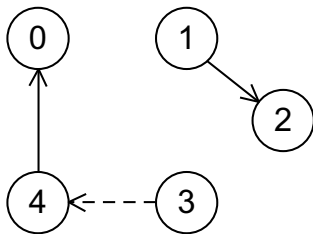
```
1  if (my_rank % 2 == 0) {  
2      MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz,  
    ↪ 0, comm);  
3      MPI_Recv(new msg, size, MPI_INT, (my_rank+comm_sz -  
    ↪ 1) % comm_sz, 0, comm, MPI_STATUS_IGNORE);  
4  } else {  
5      MPI_Recv(new msg, size, MPI_INT, (my_rank+comm_sz -  
    ↪ 1) % comm_sz, 0, comm, MPI_STATUS_IGNORE);  
6      MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz,  
    ↪ 0, comm);  
7  }
```

- Uma alternativa ao escalonamento manual de mensagens
- Faz um envio bloqueante e um recebimento na mesma chamada
- O **dest** e o **source** podem ser os mesmos ou diferentes
- É especialmente útil pois o MPI escalona as comunicações de modo que o programa não entre em um impasse ou capote

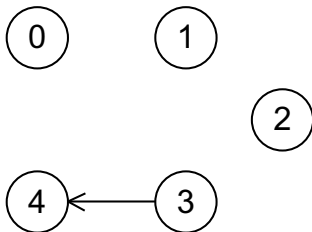
```
1  int MPI_Sendrecv(  
2      void*      send_buf_p    /* in */,  
3      int        send_buf_size /* in */,  
4      MPI_Datatype send_buf_type /* in */,  
5      int        dest          /* in */,  
6      int        send_tag      /* in */,  
7      void*      recv_buf_p    /* out */,  
8      int        recv_buf_size /* in */,  
9      MPI_Datatype recv_buf_type /* in */,  
10     int        source         /* in */,  
11     int        recv_tag       /* in */,  
12     MPI_Comm    communicator  /* in */,  
13     MPI_Status* status_p      /* in */);
```



Time 0



Time 1



Time 2

```

1 void Merge_low(
2     int my_keys[],      /* in/out    */
3     int recv_keys[],   /* in       */
4     int temp_keys[],   /* scratch  */
5     int local_n        /* = n/p, in */) {
6     int m_i, r_i, t_i;
7     m_i = r_i = t_i = 0;
8     while (t_i < local_n) {
9         if (my_keys[m_i] <= recv_keys[r_i]) {
10             temp_keys[t_i] = my_keys[m_i];
11             t_i++; m_i++;
12         } else {
13             temp_keys[t_i] = recv_keys[r_i];
14             t_i++; r_i++;
15         }
16     }
17     memcpy(my_keys, temp_keys, local_n*sizeof(int));
18 }

```

Table 3.9 Run-Times of Parallel Odd-Even Sort (times are in milliseconds)

Processes	Number of Keys (in thousands)				
	<i>200</i>	<i>400</i>	<i>800</i>	<i>1600</i>	<i>3200</i>
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

Conclusão

- MPI ou, *Message Passing Interface* é uma biblioteca de funções que podem ser chamadas a partir de linguagens de programação como C, C++ e Fortran
- Um comunicador é uma coleção de processos que se comunicam por meio de mensagens
- Muitos programas paralelos utilizam a abordagem SPMD (*Single Program Multiple Data*)
- A maior parte dos programas sequenciais é determinística: se executamos o mesmo programa múltiplas vezes com a mesma entrada obtemos a mesma saída
- Programas paralelos, em geral, não têm essa característica
- Comunicações coletivas envolvem todos os processos de um comunicador

- Quando medimos o tempo de execução de um programa paralelo estamos tipicamente interessados no tempo decorrido também chamado de tempo de relógio de parede (*wall clock time*)
- Speedup é a razão entre o tempo de execução de um programa sequencial e o tempo de execução da versão paralela
- Eficiência é speedup dividido pelo número de processos paralelos
- Se é possível aumentar o tamanho do problema (n) de modo que a eficiência não diminua conforme o número de processos (p) aumenta, então dizemos que o programa paralelo é escalável
- Um programa MPI é inseguro se seu comportamento correto é dependente da bufferização do **MPI_Send**