



Escola Supercomputador Santos Dumont - 2026

# Programação com MPI

## Ementa

- Rotinas de Comunicação Coletivas
- Barreira: MPI\_Barrier()
- Difusão: MPI\_Broadcast()
- Coleta: MPI\_Gather()
- Redução: MPI\_Reduce()
- Redução com difusão: MPI\_Allreduce()
- Coleta com difusão: MPI\_Allgather()

# Escola de Computação Santos Dumont - 2026

## Bibliografia



<https://www.casadocodigo.com.br/products/livro-programacao-paralela>



Escola Supercomputador Santos Dumont - 2026

# Comunicação Coletiva

## Google Colab

- Os exemplos, notebooks e slides do minicurso estão disponíveis no seguinte endereço:
- <https://github.com/Programacao-Paralela-e-Distribuida/MPI/>

## Comunicação coletiva

- As operações de comunicação coletiva são mais restritivas que as comunicações ponto a ponto:
  - A quantidade de dados enviados deve casar exatamente com a quantidade de dados especificada pelo receptor.
  - Apenas a versão bloqueante das funções está disponível.
  - O argumento tag não existe.
  - Todos os processos participantes da comunicação coletiva chamam a mesma função com argumentos compatíveis.
  - As funções estão disponíveis apenas no modo padrão\*.  
\* Nas últimas versões do padrão MPI já existem versões não bloqueantes das rotinas de comunicação coletiva.

## Comunicação Coletiva

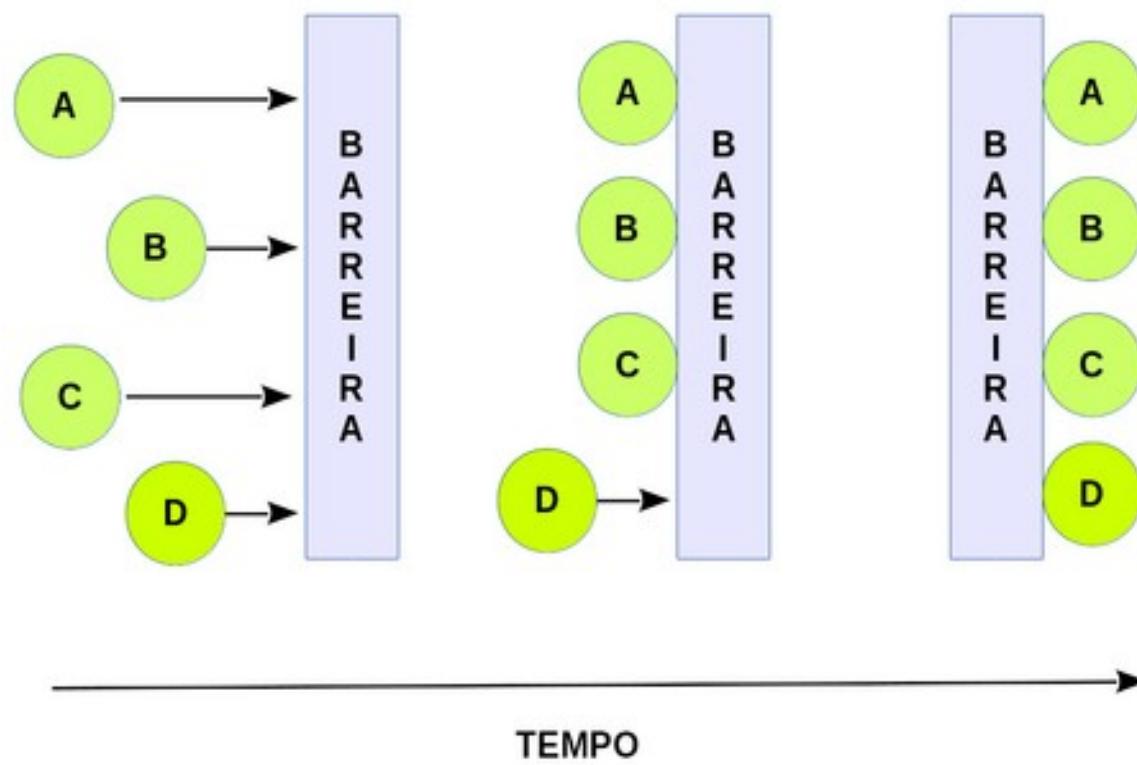
- Quando, em uma operação coletiva, houver um único processo de origem ou um único processo de destino, este processo é chamado de raiz.
- **Barreira**: bloqueia todos os processos até que todos processos do grupo chamem a função.
- **Difusão (broadcast)**: envia a mesma mensagem para todos os processos.
- **Coleta (gather)**: os dados são recolhidos de todos os processos em um único processo.
- **Dispersão (scatter)**: os dados são distribuídos de um processo para os demais.

## Comunicação Coletiva

- **Coleta com difusão (Allgather)**: um coleta (*gather*) seguida de uma difusão.
- **Redução (reduce)**: realiza as operações coletivas de soma, máximo, mínimo, etc.
- **Redução com difusão (Allreduce)**: uma redução seguida de uma difusão.
- **Alltoall**: conjunto de coletas (*gathers*) onde cada processo recebe dados diferentes.\*

\* Não vamos abordar neste nosso estudo

## Barreira



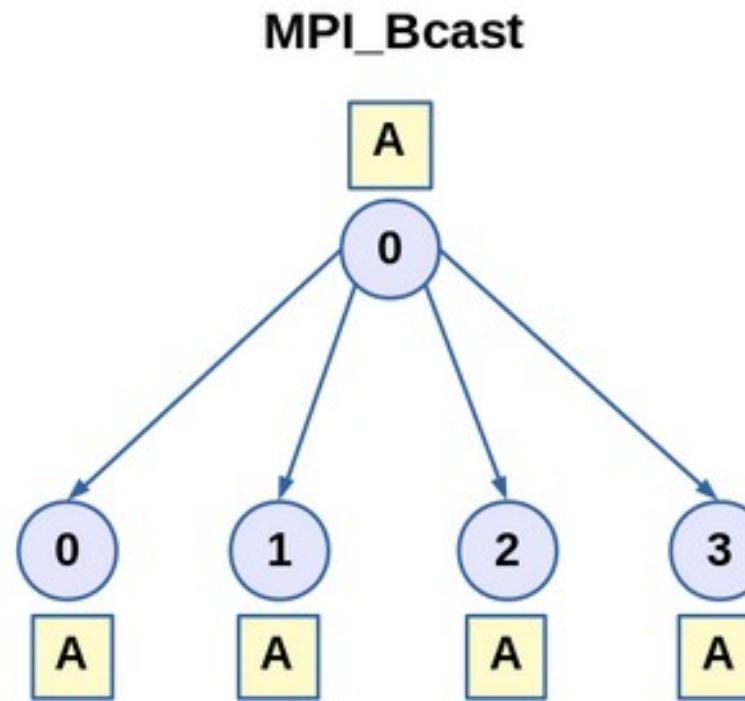
## MPI\_Barrier()

- Sintaxe:

```
int MPI_Barrier(MPI_Comm com)
```

- A função **MPI\_Barrier()** fornece um mecanismo para sincronizar todos os processos no comunicador **com**.
- Cada processo bloqueia (i.e., pára) até todos os processos no comunicador **com** tenham chamado **MPI\_Barrier()**.

## MPI\_Bcast()



## MPI\_Bcast()

- Um padrão de comunicação que envolva todos os processos em um comunicador é chamada de comunicação coletiva.
- Uma difusão (**broadcast**) é uma comunicação coletiva na qual um único processo envia os mesmos dados para cada processo.
- A função MPI para difusão é:  
`int MPI_Bcast (void* mensagem, int cont, MPI_Datatype tipo_mpi, int raiz, MPI_Comm com)`

## MPI\_Bcast()

- Ela simplesmente envia uma cópia dos dados de mensagem no processo **raiz** para cada processo no comunicador **com**.
- Para que a operação funcione corretamente, ela deve ser chamado por **todos** os processos no comunicador com os mesmos argumentos para **raiz** e **com**.
- Uma mensagem de difusão **não** pode ser recebida com **MPI\_Recv()**, pois a operação é exclusivamente coletiva.
- Os parâmetros **cont** e **tipo\_mpi** têm a mesma função que nas funções **MPI\_Send()** e **MPI\_Recv()**, definindo o tamanho e o tipo da mensagem.
- Contudo, ao contrário das funções ponto-a-ponto, o padrão MPI exige que os valores de **cont** e **tipo\_mpi** sejam **iguais para todos os processos** dentro do mesmo comunicador em uma comunicação coletiva.

## Exemplo

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) { /* mpi_bcast.c */
    int valor, meu_ranque, num_procs, raiz = 0;
    MPI_Comm comm=MPI_COMM_WORLD;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(comm, &meu_ranque);
    MPI_Comm_size(comm, &num_procs);
    /* Cada processo tem um valor inicial diferente */
    valor = meu_ranque;
    printf("O processo com ranque %d tem inicialmente o valor: %d\n", meu_ranque,
    valor);
    /* O valor a ser enviado é lido pelo processo raiz */
    if (meu_ranque == raiz) {
        printf("Entre um valor: \n");
        scanf("%d", &valor);
        MPI_Barrier(comm);
    }
}
```

## Exemplo

```
else
    MPI_Barrier(comm);
printf("Passei da barreira. Eu sou o %d de %d processos \n", meu_ranque, num
_procs);
/* A rotina de difusão deve ser chamada por todos processos */
/* Apenas o processo raiz envia, mas todos recebem o valor enviado*/
MPI_Bcast(&valor, 1, MPI_INT, raiz, comm);
/* O valor agora é o mesmo em todos os processos */
printf("O processo com ranque %d recebeu o valor: %d\n",meu_ranque, valor);
/* Termina a execução */
MPI_Finalize();
return 0;
}
```

## Exemplo

```
$ mpirun -np 3 ./teste
```

O processo com ranque 0 tem inicialmente o valor: 0

Entre um valor:

O processo com ranque 1 tem inicialmente o valor: 1

O processo com ranque 2 tem inicialmente o valor: 2

33

Passei da barreira. Eu sou o 0 de 3 processos

O processo com ranque 0 recebeu o valor: 33

Passei da barreira. Eu sou o 1 de 3 processos

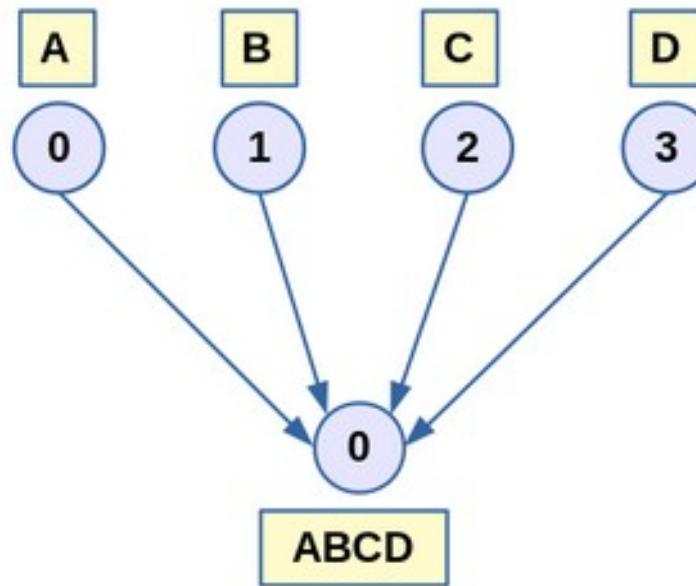
O processo com ranque 1 recebeu o valor: 33

Passei da barreira. Eu sou o 2 de 3 processos

O processo com ranque 2 recebeu o valor: 33

## MPI\_Gather()

**MPI\_Gather**



## MPI\_Gather()

```
int MPI_Gather(void* vet_envia, int cont_envia, MPI_Datatype tipo_envia,  
void* vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe, int raiz,  
MPI_Comm com)
```

- Cada processo no comunicador `com` envia o conteúdo de `vet_envia` para o processo com ranque igual a `raiz`.
- O processo `raiz` concatena os dados que são recebidos em `vet_recebe` em uma ordem que é definida pelo `ranque` de cada processo.
- Os argumentos que terminam com `recebe` são significativos apenas no processo com ranque igual a `raiz`.
- O argumento `cont_recebe` indica o número de itens enviados por **cada processo**, não número total de itens recebidos pelo processo `raiz` e, normalmente, é igual a `cont_envia`.

## Exemplo

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define TAM_VET 10

int main(int argc, char *argv[]) { /* mpi_gather.c */
    int meu_ranque, num_procs, raiz = 2;
    int *vet_recebe, vet_envia[TAM_VET];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* O processo raiz aloca o espaço de memória para o vetor de recepção */
    if (meu_ranque == raiz)
        vet_recebe = (int *) malloc(num_procs*TAM_VET*sizeof(int));
    /* Cada processo atribui valor inicial ao vetor de envio */
    for (int i = 0; i < TAM_VET; i++)
        vet_envia[i] = meu_ranque;
    /* O vetor é recebido pelo processo raiz com as partes recebidas
       de todos processos, incluindo o processo raiz */
    MPI_Gather (vet_envia, TAM_VET, MPI_INT, vet_recebe, TAM_VET, MPI_INT, raiz, MPI_COMM_WORLD);
}
```

## Exemplo

```
/* O processo raiz imprime o vetor recebido */
if (meu_ranque == raiz) {
    printf("Processo = %d, recebeu", meu_ranque);
    for (int i = 0; i < (TAM_VET*num_procs); i++) {
        printf(" %d", vet_recebe[i]);
    }
    printf("\n");
}
/* Termina a execução */
MPI_Finalize();
return(0);
}
```

## Exemplo

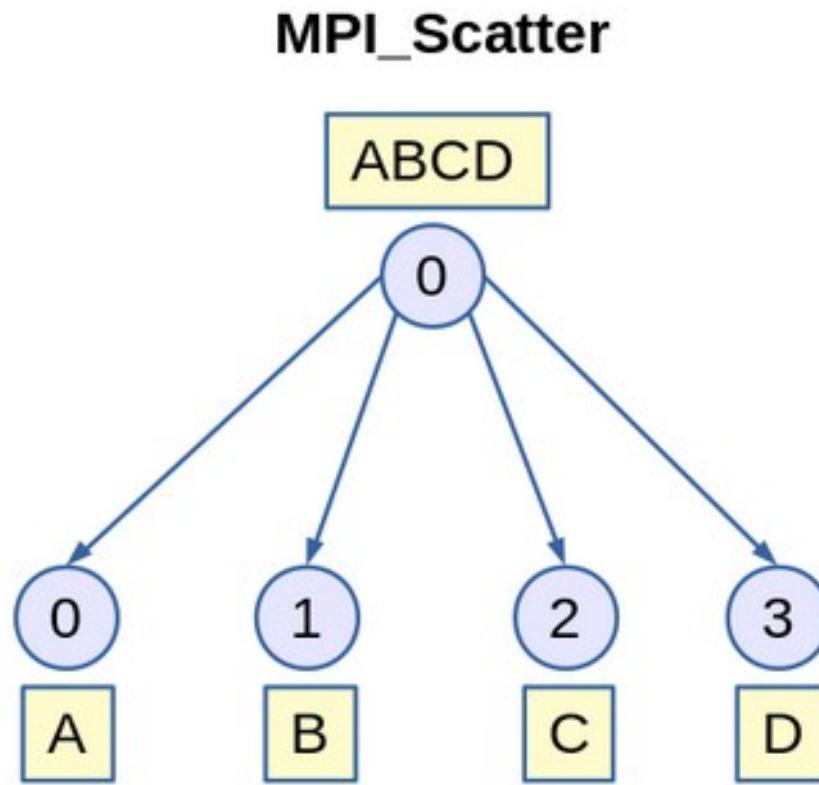
- A execução do programa com os parâmetros abaixo terá o seguinte resultado:

```
$ mpicc -o teste mpi_gather.c
```

```
$ mpirun -np 4 ./teste
```

```
Processo = 2, recebeu 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3  
3 3 3 3
```

## MPI\_Scatter()



## MPI\_Scatter()

```
int MPI_Scatter(void* vet_envia, int cont_envia, MPI_Datatype tipo_envia, void*  
vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe, int raiz, MPI_Comm com)
```

- O processo com o ranque igual a **raiz** distribui o conteúdo de **vet\_envia** entre os **p** processos.
- O conteúdo de **vet\_envia** é dividido em **p** segmentos, cada um deles consistindo de **cont\_envia** itens.
- O primeiro segmento vai para o processo com ranque **0**, o segundo para o processo com ranque **1**, etc.
- Os argumentos que terminam com **envia** são significativos apenas no processo **raiz**, pois ele é o único que possui o conteúdo completo a ser distribuído.

## Exemplo

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define TAM_VET 10

int main(int argc, char *argv[]) { /* mpi_scatter.c */
    int i, meu_ranque, num_procs, raiz = 0;
    int *vet_envia, vet_recebe[TAM_VET];
    MPI_Comm com=MPI_COMM_WORLD;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(com, &meu_ranque);
    MPI_Comm_size(com, &num_procs);
    /* O processo raiz aloca o espaço de memória e inicia o vetor */
    if (meu_ranque == raiz) {
        vet_envia = (int*) malloc (num_procs*TAM_VET*sizeof(int));
        for (i = 0; i < (TAM_VET*num_procs); i++)
            vet_envia[i] = i;
    }
}
```

## Exemplo

```
/* O vetor é distribuído em partes iguais entre os processos,
   incluindo o processo raiz */
MPI_Scatter(vet_envia, TAM_VET, MPI_INT, vet_recebe, TAM_VET, MPI_INT, raiz,
com);
/* Cada processo imprime a parte que recebeu */
printf("Processo = %d, recebeu", meu_ranque);
for (i = 0; i < TAM_VET; i++)
    printf(" %d", vet_recebe[i]);
printf("\n");
/* Termina a execução */
MPI_Finalize();
return(0);
}
```

## Exemplo

```
$ mpicc -o teste mpi_scatter.c
```

```
$ mpirun -np 4 ./bin/mpi_scatter
```

```
Processo = 0, recebeu 0 1 2 3 4 5 6 7 8 9
```

```
Processo = 1, recebeu 10 11 12 13 14 15 16 17 18 19
```

```
Processo = 2, recebeu 20 21 22 23 24 25 26 27 28 29
```

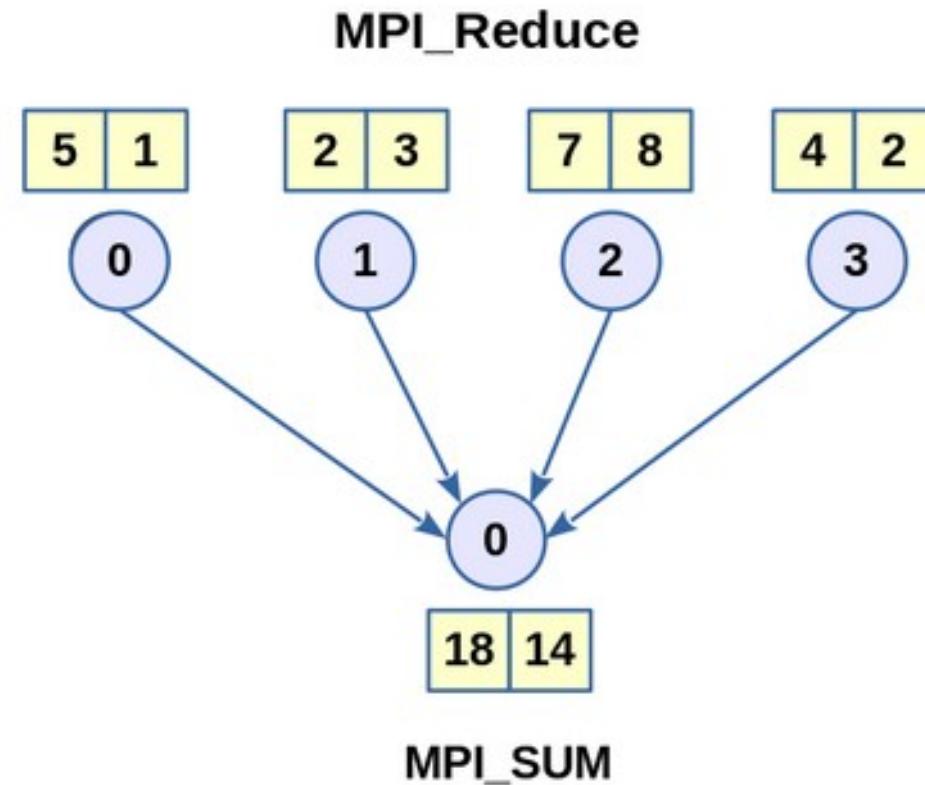
```
Processo = 3, recebeu 30 31 32 33 34 35 36 37 38 39
```



Escola Supercomputador Santos Dumont - 2026

# Comunicação Coletiva

## MPI\_Reduce()



## MPI\_Reduce()

- Realiza, por exemplo, uma soma global, que é um exemplo de uma classe geral de operações de comunicação coletivas chamada operações de **redução**.
- Em uma operação global de redução, todos os processos em um comunicador contribuem com dados que são combinados em operações binárias.
- Operações binárias típicas são a adição, máximo, mínimo, e lógico, etc.
- É possível definir operações adicionais além das mostradas para a função **MPI\_Reduce()**.

## MPI\_Reduce()

```
int MPI_Reduce(void* operando, void* resultado, int cont, MPI_Datatype  
tipo_mpi, MPI_Op oper, int raiz, MPI_Comm com)
```

- A operação `MPI_Reduce()` combina os operandos armazenados em `*operando` usando a operação `oper` e armazena o resultado em `*resultado` no processo `raiz`.
- Tanto `operando` como `resultado` referem-se a `cont` posições de memória com o tipo `tipo_mpi`.
- `MPI_Reduce()` deve ser chamada por todos os processos no comunicador `com` e os valores de `cont`, `tipo_mpi` e `oper` devem ser os mesmos em cada processo.

## Valores do argumento oper

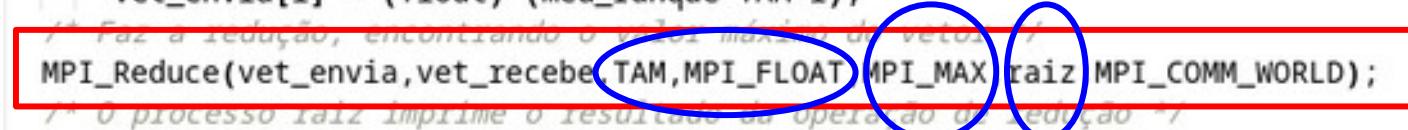
MPI_MAX		Máximo
MPI_MIN		Mínimo
MPI_SUM		Soma
MPI_PROD		Produto
MPI_LAND		“E” lógico
MPI_BAND		“E” bit a bit
MPI_LOR		“Ou” lógico
MPI_BOR		“Ou” bit a bit
MPI_LXOR		“Ou Exclusivo” lógico
MPI_BXOR		“Ou Exclusivo” bit a bit
MPI_MAXLOC		Máximo e Posição do Máximo
MPI_MINLOC		Mínimo e Posição do Mínimo

## Exemplo

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define TAM 5

int main(int argc, char *argv[]) { /* mpi_reduce.c */
    int meu_ranque, num_procs, i, raiz = 0;
    float vet_envia [TAM]; /* Vetor a ser enviado */
    float vet_recebe [TAM]; /* Vetor a ser recebido */

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    /* Preenche o vetor com valores que dependem do ranque */
    for (i = 0; i < TAM; i++)
        vet_envia[i] = (float) (meu_ranque*TAM+i);
    /* Faz a redução, encontrando o valor máximo do vetor */
    MPI_Reduce(vet_envia,vet_recebe,TAM,MPI_FLOAT,MPI_MAX,raiz,MPI_COMM_WORLD);
    /* O processo raiz imprime o resultado da operação de redução */
    if (meu_ranque == raiz) {
        for (i = 0; i < TAM; i++)
            printf("vet_recebe[%d] = %3.1f ", i,vet_recebe[i]);
        printf("\n\n");
    }
    MPI_Finalize();
    return(0);
}
```



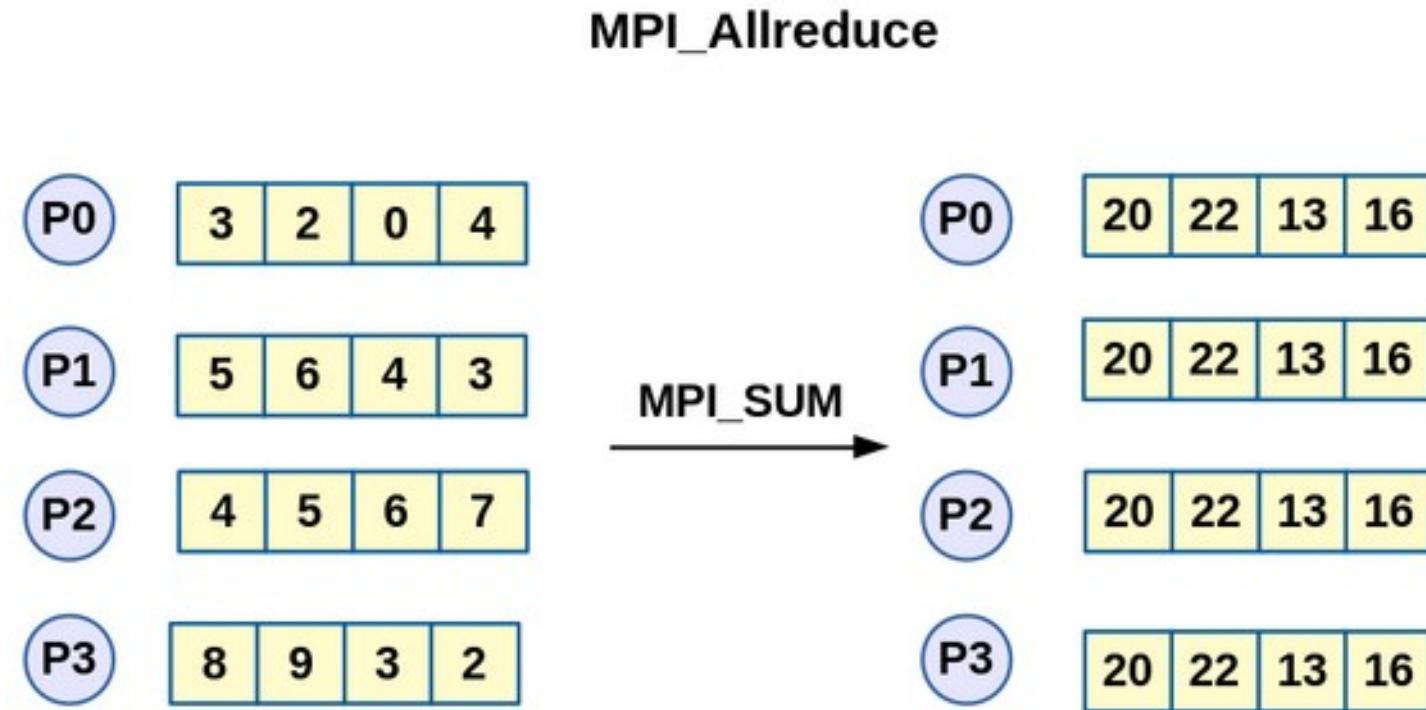
## Exemplo MPI\_Reduce()

```
$ mpicc -o teste mpi_reduce.c
```

```
$ mpirun -np 4 ./bin/mpi_reduce
```

```
vet_recebe[0] = 15.0 vet_recebe[1] = 16.0 vet_recebe[2] = 17.0 vet_recebe[3] = 18.0 vet_recebe[4]  
= 19.0
```

## MPI\_Allreduce()



## MPI\_Allreduce()

```
int MPI_Allreduce (void* vet_envia, void* vet_recebe, int cont, MPI_Datatype  
tipo_mpi, MPI_Op oper, MPI_Comm com)
```

- **MPI\_Allreduce()** tem um resultado similar à operação **MPI\_Reduce()**, com a diferença que o resultado da operação de redução **oper** é armazenado no vetor **vet\_recebe** de cada processo envolvido na chamada e não apenas no processo **raiz**.
- Aliás, como todos recebem o resultado da redução, não há necessidade do parâmetro **raiz** no protótipo da função.

## Cálculo de Pi

- O valor de  $\pi$  pode ser obtido pela integração numérica :

$$\int_0^1 \frac{4}{1+x^2}$$

- Que pode ser calculada em paralelo, dividindo-se o intervalo de integração entre vários processos, de modo similar ao método do trapézio.

## Cálculo de Pi

```
#include<stdio.h>
#include <math.h>
#include "mpi.h"
int main (int argc, char *argv[]){
    int meu_ranque, num_procs, n=10000000;
    double mypi, pi, h, x, sum = 0.0;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
    h=1.0/(double) n;
    for (int i = meu_ranque +1; i <= n; i += num_procs){
        x = h * ((double) i - 0.5);
        sum += (4.0/(1.0 + x*x));
    }
    mypi = h* sum;
    MPI_Allreduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    printf ("valor aproximado de pi: %.16f \n", pi);
    MPI_Finalize( );
}
```

Não tem raiz

## Exemplo

```
$ mpicc -o teste mpi_calcpi.c  
$ mpirun -np 4 ./teste  
valor aproximado de pi: 3.1415926535896865  
valor aproximado de pi: 3.1415926535896865  
valor aproximado de pi: 3.1415926535896865  
valor aproximado de pi: 3.1415926535896865
```