

Material Suplementar



# Programação Paralela e Distribuída - Parte 1

**Gabriel P. Silva**

# Programação Paralela e Distribuída

com MPI, OpenMP e OpenACC  
para computação de alto desempenho



 Casa do Código | **alura**

GABRIEL P. SILVA  
CALEBE P. BIANCHINI  
EVALDO B. COSTA

Os códigos fontes utilizados neste material estão disponíveis em:

<https://github.com/Programacao-Paralela-e-Distribuida/MPI>

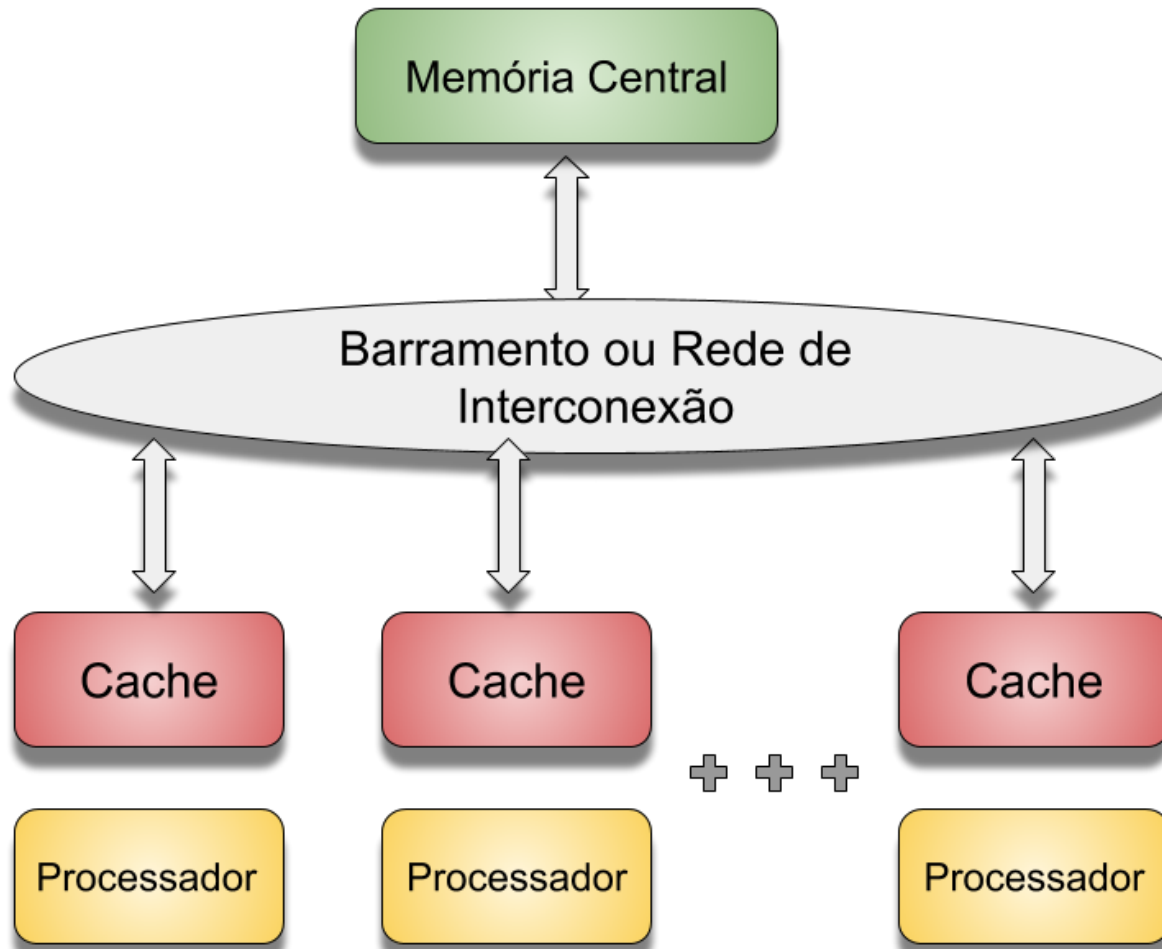
# Modelos de Programação



# Memória Compartilhada

- Todos os processos/threads compartilham uma memória global comum, como consequência, toda comunicação entre as tarefas é feita através de variáveis na memória.
- Mecanismos de sincronização, como semáforos ou regiões críticas, são utilizados para evitar conflitos e garantir a correta ordenação de acesso à memória compartilhada.
- Como o acesso à memória compartilhada pode ser limitado, problemas de escalabilidade podem surgir com o aumento do número de processadores.
- As bibliotecas de programação mais utilizadas: OpenMP, pthreads.

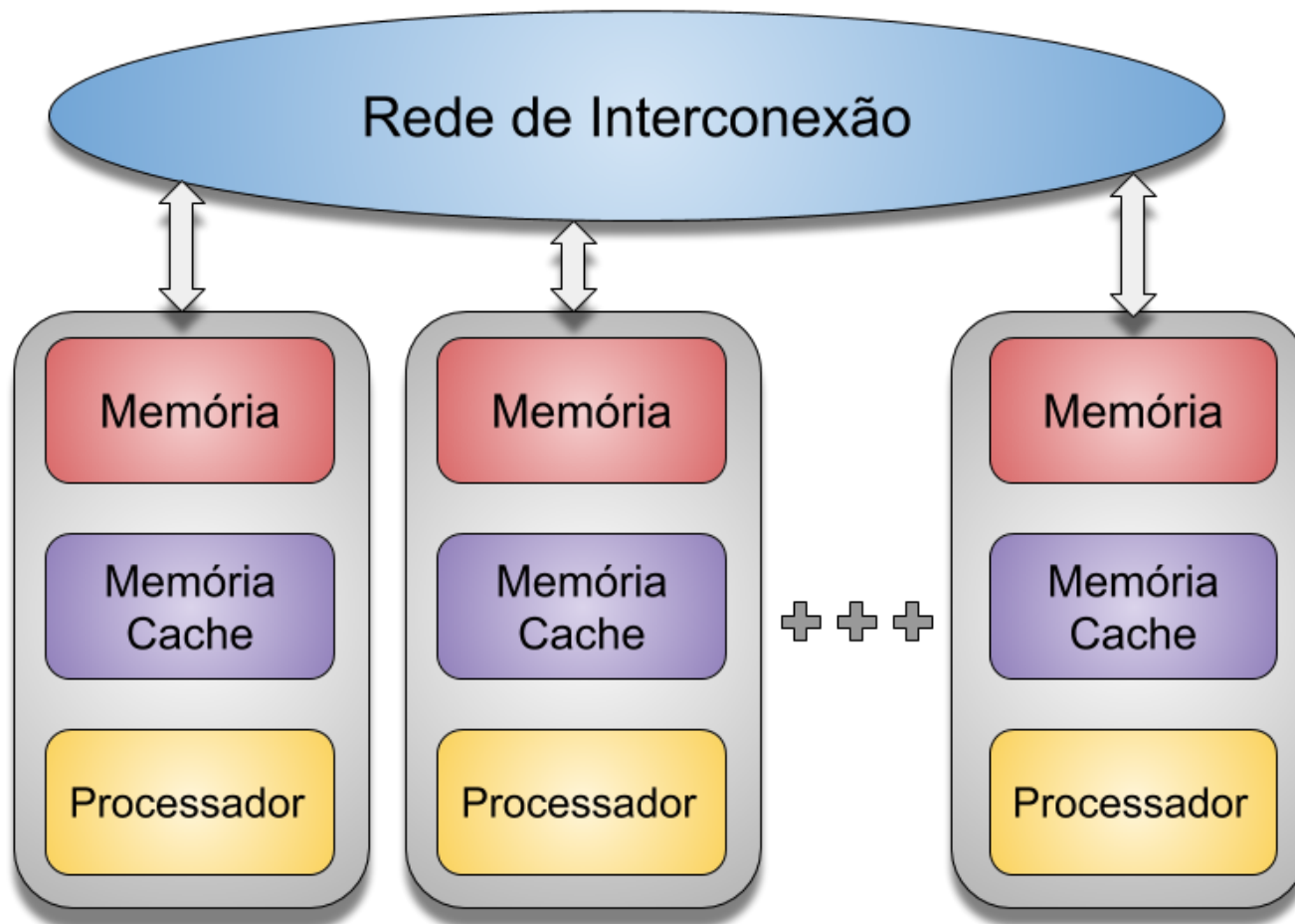
# Memória Compartilhada



# Memória Distribuída

- Os processos/threads não dispõem de um espaço de memória em comum para operações de comunicação ou sincronização, sendo que essas operações são feitas por meio de troca de mensagens.
- Os processos dependem de uma rede de interconexão e de uma biblioteca de comunicação, como PVM e MPI, para realizar o envio e recebimento de mensagens.
- A programação paralela por troca de mensagens é altamente escalável, o que significa que é possível adicionar mais processadores para aumentar o desempenho da aplicação.

# Memória Distribuída

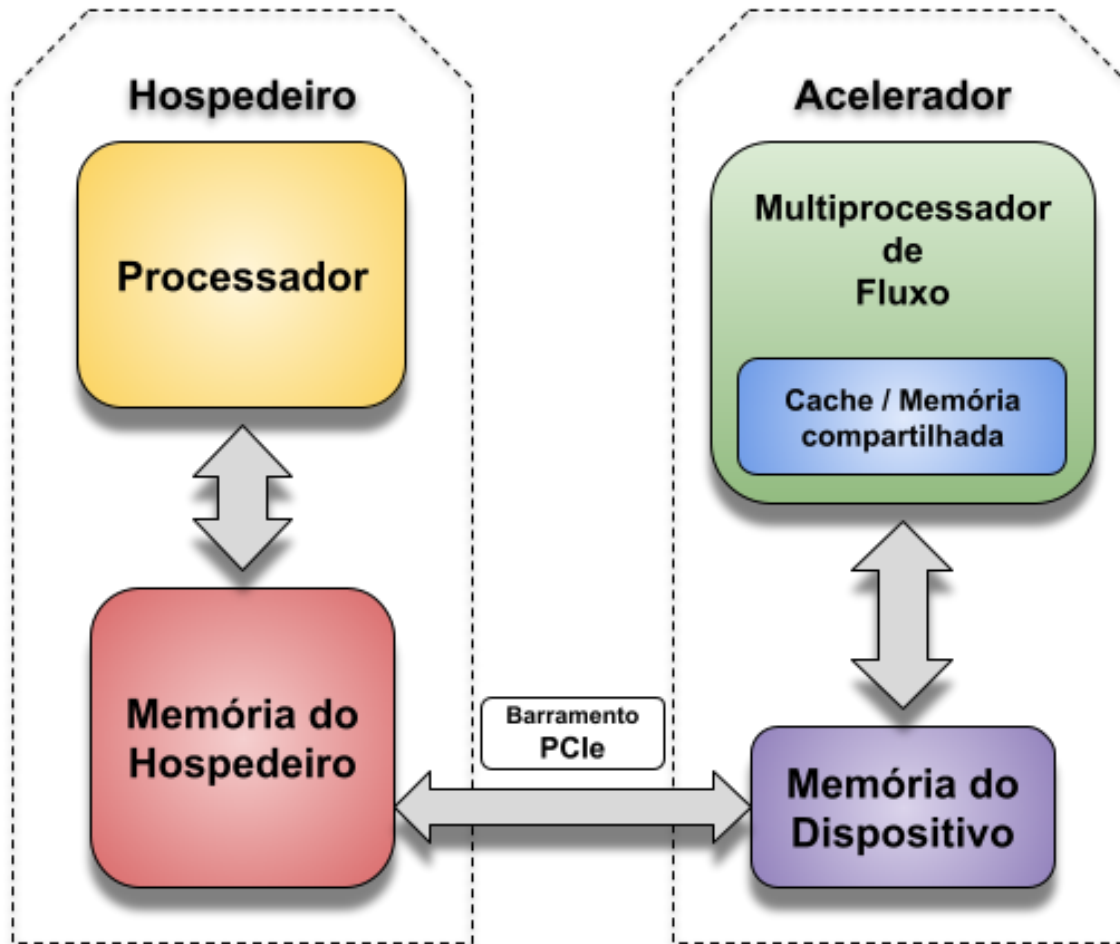




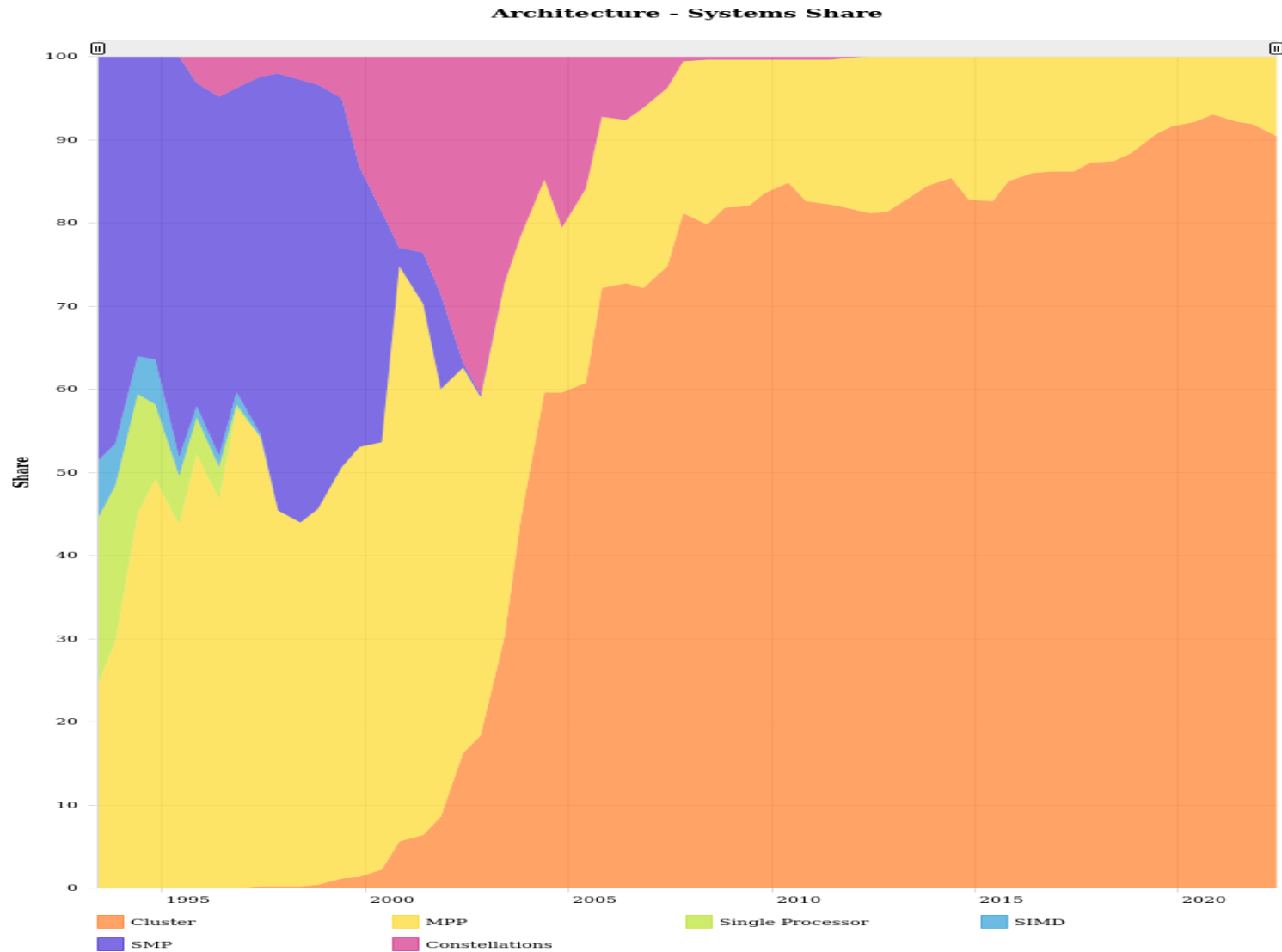
# Aceleradores

- Os laços de computação mais intensiva são transferidos e executados no aceleradores.
- Existem memórias separadas para o hospedeiro e o acelerador.
- A sincronização é feita com rotinas especiais dependentes da biblioteca utilizada.
- As bibliotecas mais comuns são OpenACC, OpenMP, CUDA e OpenCL.
- A programação com CUDA pode ser bem difícil.
- Exemplo de acelerador:  
<https://www.amd.com/en/graphics/instinct-server-accelerators>

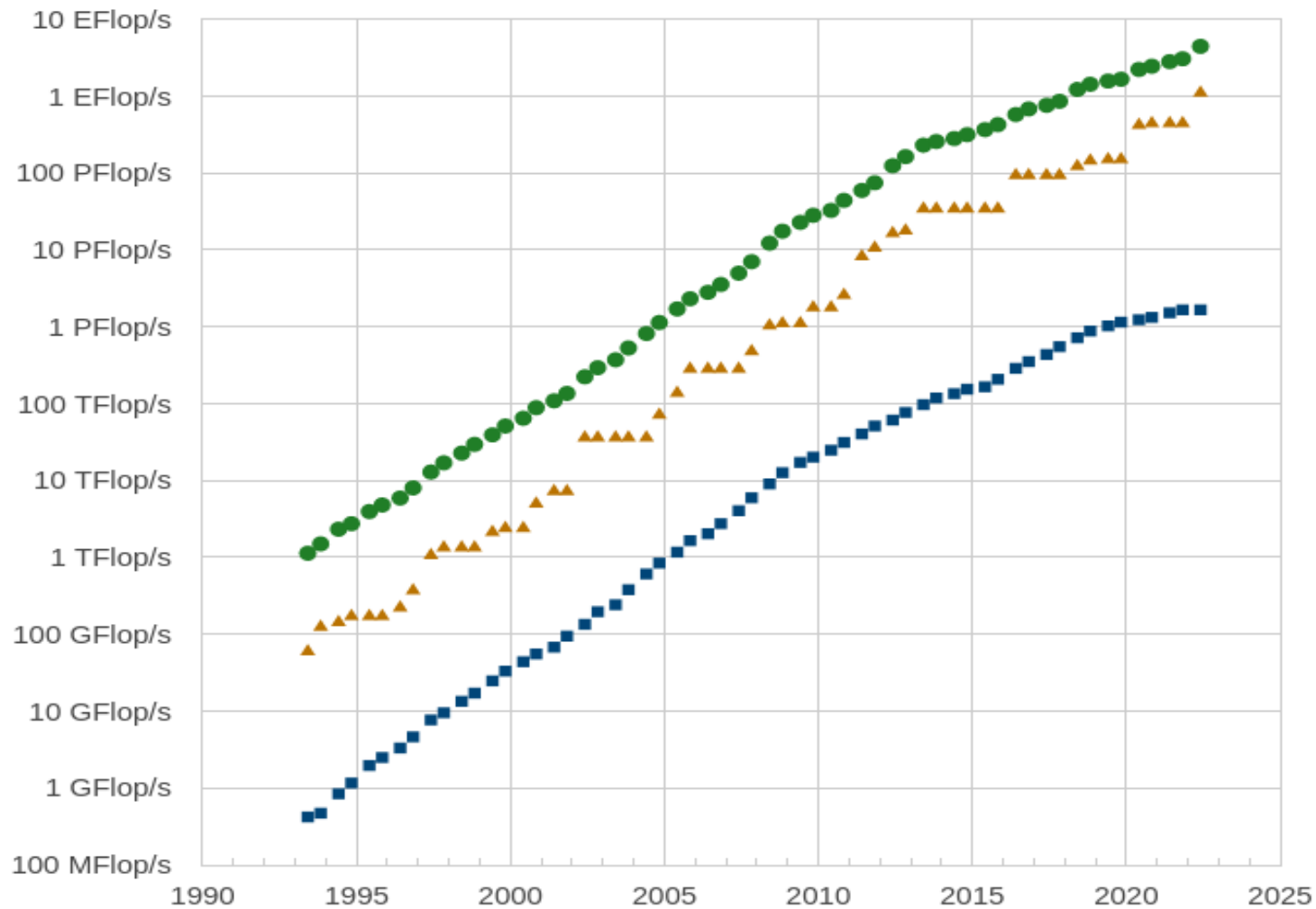
# Aceleradores



<http://www.top500.org>



<http://www.top500.org>



# Método de Barnes-Hut

<https://www.youtube.com/watch?v=D-0GaBQ494E>

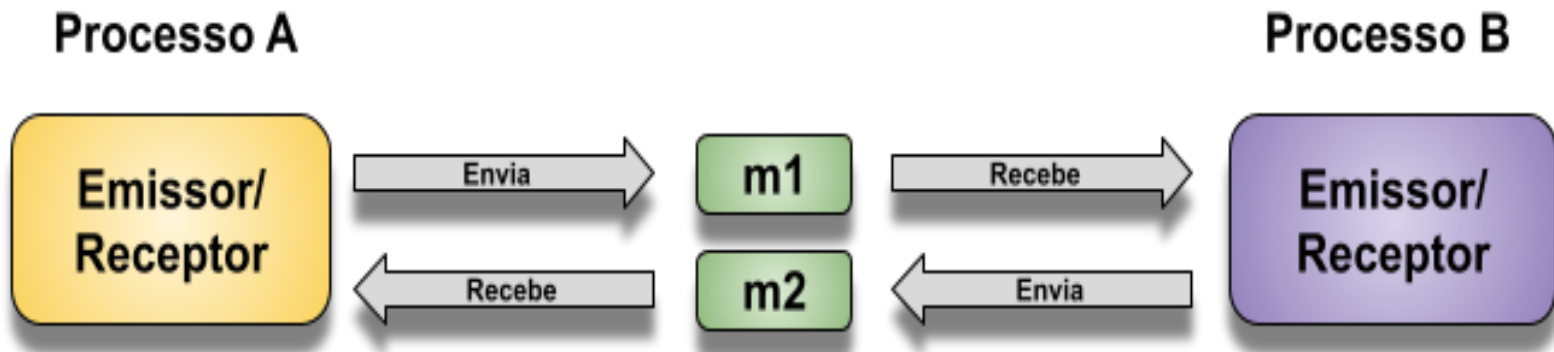
# Conceitos Básicos



# Troca de Mensagens

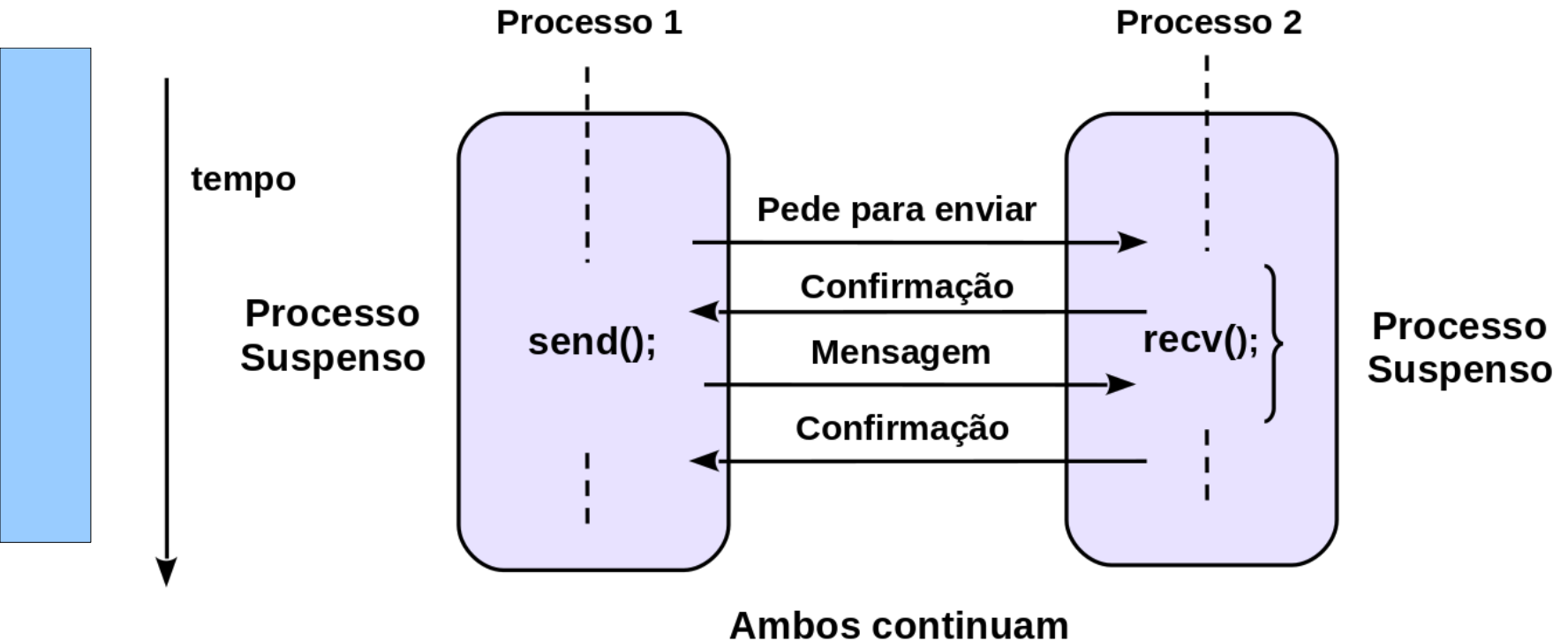
- As tarefas são mapeadas em processos, cada um com sua memória privada, que podem ser criados de forma estática ou dinâmica.
- Os processos são mapeados para processadores interligados por uma rede de comunicação, como Ethernet ou Infiniband.
- A comunicação entre os processos é feita através do envio explícito de mensagens, com os dados e informações necessárias para as tarefas sejam executadas por cada processo.
- A sincronização entre as tarefas pode ser feita de forma implícita pela troca de mensagens ou por operações coletivas de sincronização, tais como as barreiras.

# Troca de Mensagens

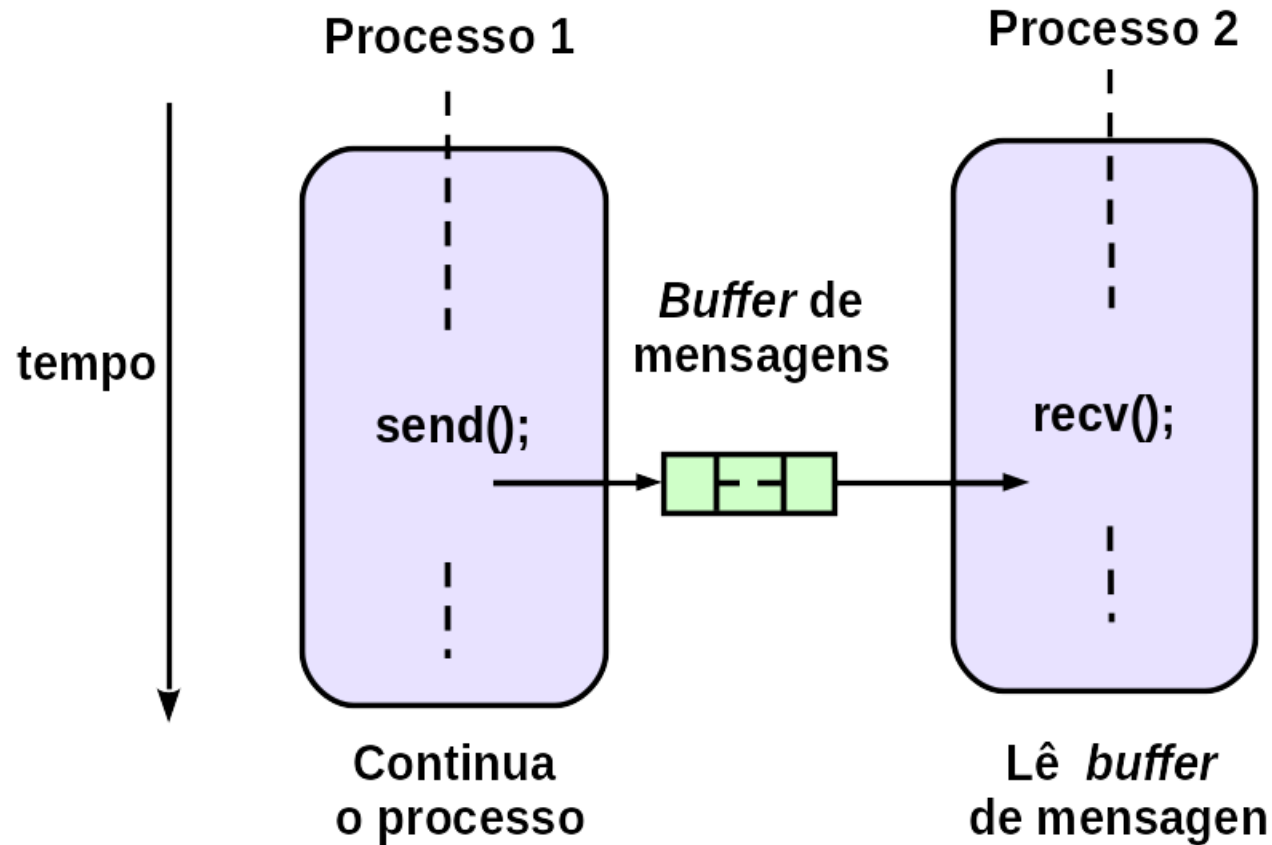




# Comunicação Síncrona



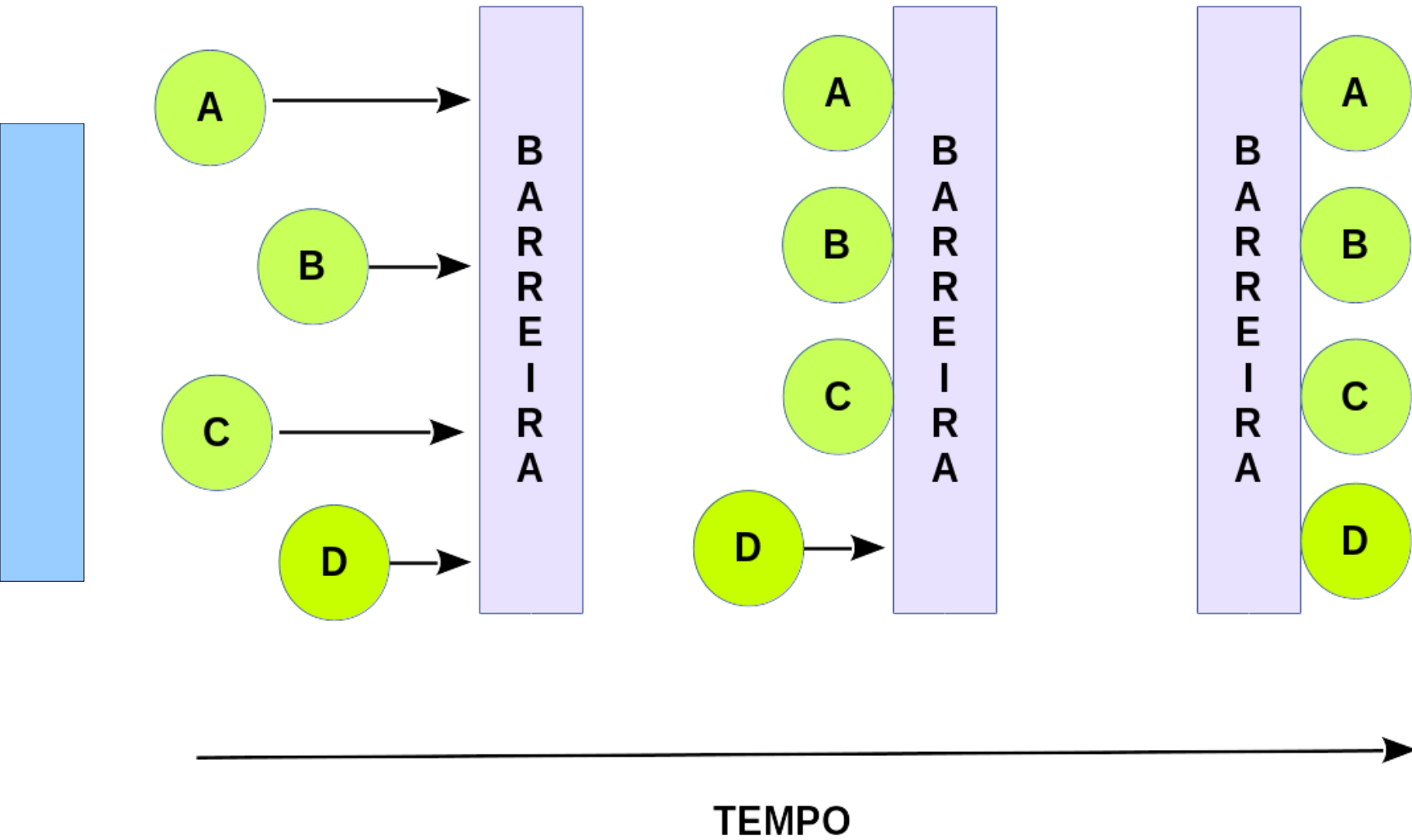
# Comunicação Assíncrona



# Síncrona vs. Assíncrona

- O modo de comunicação síncrono é simples e seguro, permite a sincronização entre processos, contudo elimina a possibilidade de haver superposição entre o processamento da aplicação e a transmissão das mensagens, diminuindo as possibilidades de exploração de paralelismo.
- O modo de comunicação assíncrono é o que permite maior superposição no tempo entre o processamento da aplicação e a transmissão das mensagens, permitindo maior paralelismo, mas elimina a possibilidade de sincronização entre processos com uso das rotinas de comunicação ponto-a-ponto.

# Barreira



# Medidas de Desempenho

- Speed-up (Aceleração):

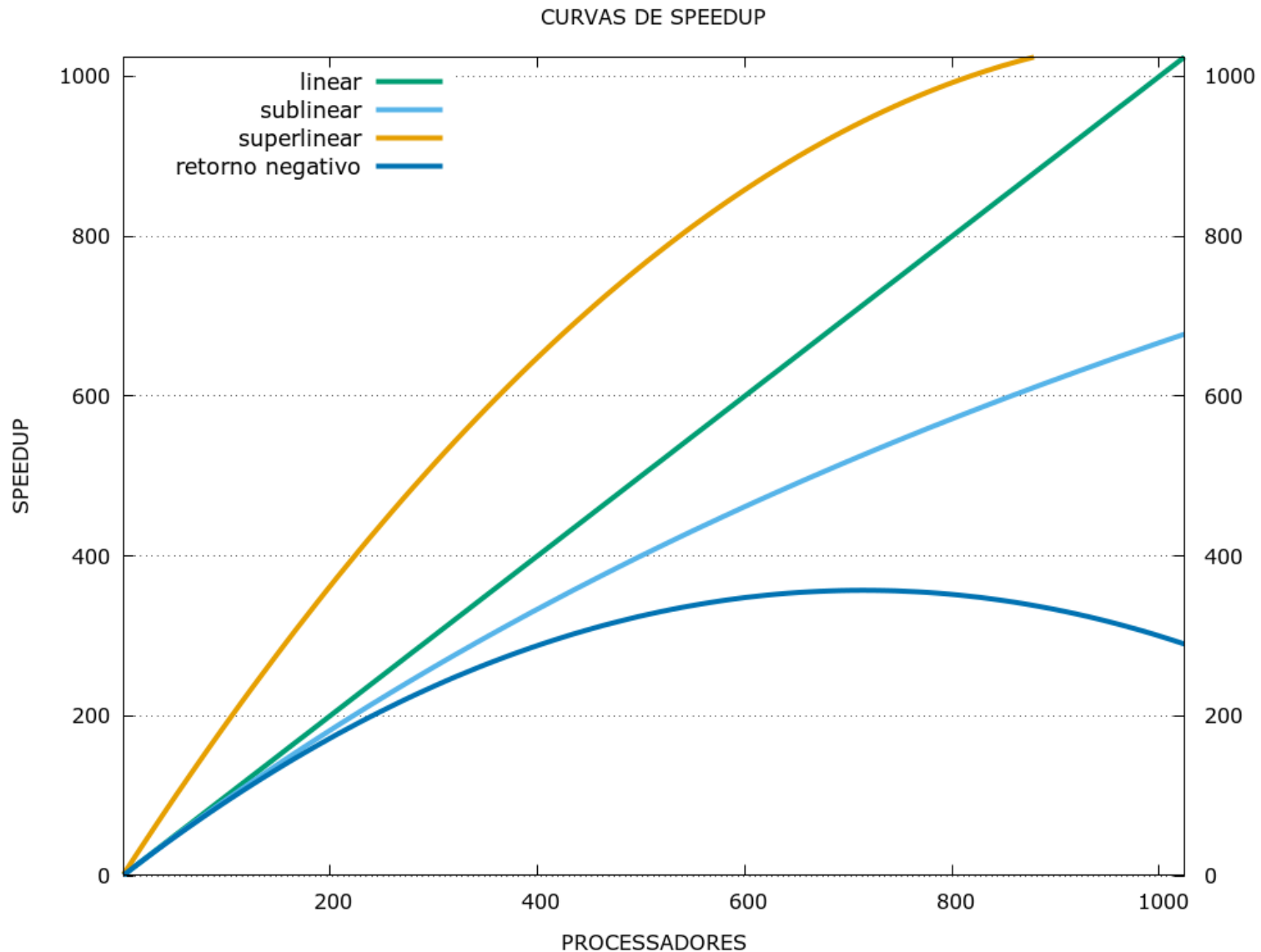
Mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador e o tempo gasto na execução com  $n$  processadores:

$$S(n) = T(1)/T(n)$$

- Eficiência:

$$E(n) = S(n)/n$$

# Curvas de Speed-up



# Criação de um Programa Paralelo

- A criação de um programa paralelo pode ser feita de diversas maneiras, não há uma receita única para essa criação.
- A divisão do problema vai ser feita em processos ou threads que irão trabalhar de forma cooperativa para a resolução do problema.
- O particionamento das tarefas depende do modelo de programação adotado (troca de mensagens ou memória compartilhada) e da eficiência com que as primitivas de comunicação e sincronização desse modelo são suportadas pelo sistema onde o programa será executado.

# Criação de um Programa Paralelo

- As tarefas de comunicação e sincronização não podem ser mais complexas que as tarefas de computação. Se isto ocorrer, a execução em paralelo certamente será mais ineficiente do que a seqüencial.
- Uma vez determinada a forma de particionamento das tarefas, os processos e *threads* devem ser mapeados para os processadores reais que existam no sistema.
- Para que isto seja feito de uma maneira eficiente, deve-se explorar a localidade da rede de interconexão, mantendo os processos relacionados alocados ao mesmo processador ou em processadores “próximos” entre si.



# Dicas de Programação Paralela

- O particionamento das tarefas deve ter como objetivo a redução da serialização no acesso aos recursos compartilhados, e aumentar a sobreposição do processamento com a comunicação, para maximizar o desempenho dos programas paralelos.
- Deve-se evitar ao máximo que os dados sejam movidos de um lado para outro, seja na memória do mesmo computador ou entre computadores diferentes, e criar formas inteligentes para acesso aos dados compartilhados.
- Se em algum momento na execução do seu programa paralelo houver uma parcela muito grande de processadores ociosos, então certamente você não fez uma boa distribuição dessas tarefas.

MPI



# Características do MPI

- É um padrão de troca de mensagens portátil que facilita o desenvolvimento de aplicações paralelas.
- Usa o paradigma de programação paralela por troca de mensagens e pode ser usado tanto em clusters como em sistemas de memória compartilhada.
- É uma biblioteca de funções utilizável com programas escritos em C, C++ ou Fortran.
- O MPI foi fortemente influenciado pelo trabalho no IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex e PARMACS. Outras contribuições importantes vieram do Zipcode, Chimp, PVM, Chameleon e PICTL.

# Características do MPI

- Suporte para comunicação entre processos ponto a ponto ou coletiva.
- Suporte para vários modelos de comunicação, incluindo comunicação síncrona e assíncrona.
- Suporte para vários tipos de topologias de rede virtuais, incluindo anel, árvore e malha.
- Suporte para tipos de dados básicos e estruturados.
- Suporte para operações de comunicação coletiva como difusão, redução e dispersão.
- Portabilidade entre diferentes arquiteturas de computação paralela e sistemas operacionais.

# Características do MPI

- O MPI pode ser combinado com outros modelos de programação.
- Assim, podemos combinar MPI com OpenMP, que apresenta vantagens em alguns casos, como por exemplo:
  - Códigos com escalabilidade MPI limitada, quer seja pelo algoritmo ou pelas rotinas de comunicação coletiva utilizadas.
  - Códigos limitado pelo tamanho de memória, tendo muitos dados replicados em cada processo MPI.
  - Códigos com problemas de desempenho pela ineficiência da implementação da comunicação intra-nó em MPI.
- O MPI também pode ser combinado com OpenACC ou CUDA para uso de aceleradores.

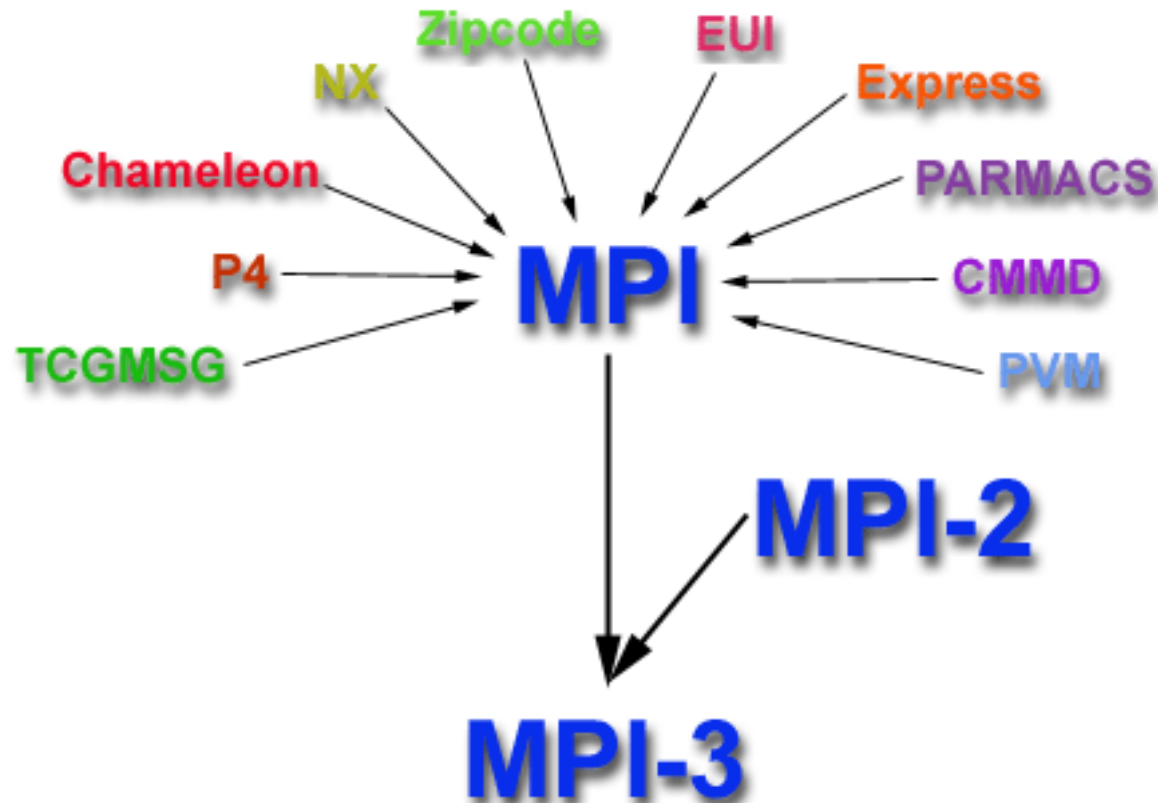
# Linguagens Suportadas

- C, C++ e Fortran: O MPI foi originalmente desenvolvido para ser usado com o C, C++ e Fortran.
- Python: Existem várias bibliotecas MPI para Python, como mpi4py e pyMPI, que permitem que os programas Python se comuniquem usando MPI.
- Java: Existe uma biblioteca MPI para Java chamada MPJ Express, que permite que os programas Java se comuniquem usando MPI.
- Outras linguagens: Além das linguagens mencionadas acima, também existem bibliotecas MPI para outras linguagens como Perl, Ruby, Lua, entre outras.

# Histórico de Versões do MPI

- MPI-1.1: primeira versão funcional lançada em 1998.
- MP1-1.3: documento final que consolida a versão 1.0 do MPI, lançada apenas em 2008.
- MPI-2.1: lançada oficialmente em 2008 como um livro com diversos exemplos e orientações para os usuários.
- MPI-2.2: lançada em 2009. A última versão desta série.
- MPI-3.1: a versão final do padrão 3.0, foi lançada em 2015.
- MPI 4.0: a versão mais recente do padrão foi lançada em 2021.
- Todas as versões estão disponíveis em <https://www.mpi-forum.org/docs/>

# Histórico



<https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf>



# Objetivos

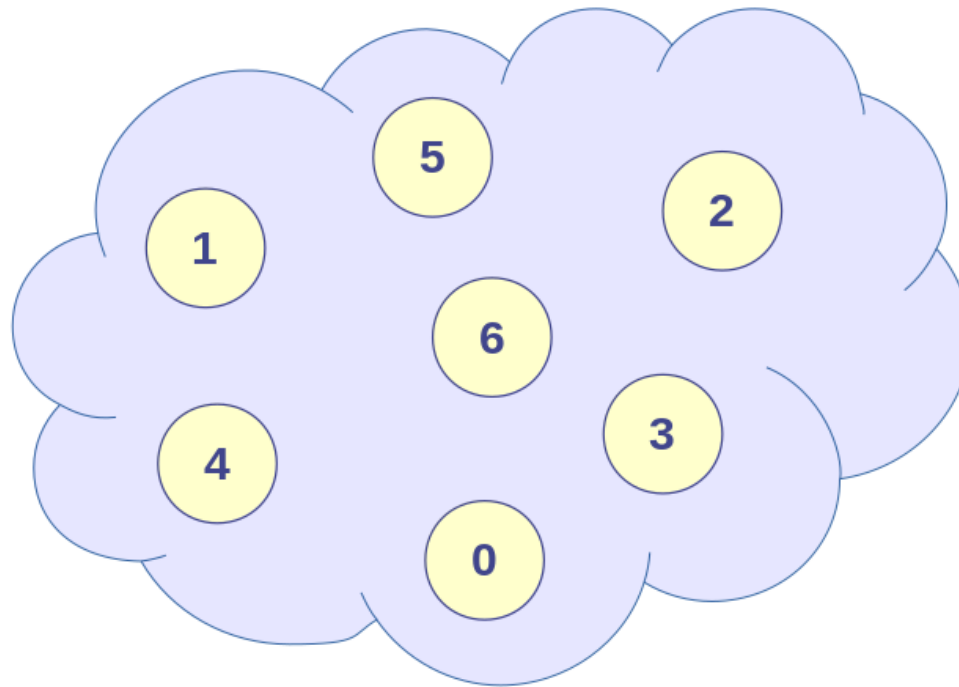
- Um dos objetivos do MPI é oferecer possibilidade de uma implementação eficiente da comunicação:
  - Evitando cópias de memória para memória.
  - Permitindo superposição de comunicação e computação.
- Permitir implementações em ambientes heterogêneos.
- Supõe-se que a interface de comunicação é confiável:
  - Falhas de comunicação devem ser tratadas pelo subsistema de comunicação da plataforma.

# Comunicadores

- A biblioteca MPI trabalha com o conceito de *comunicadores* para definir o universo de processos envolvidos em uma operação de comunicação, através dos atributos de grupo e contexto:
  - Dois processos que pertencem a um mesmo **grupo** e usando um mesmo **contexto** podem se comunicar diretamente.
  - O comunicador padrão recebe o nome de **MPI\_COMM\_WORLD** e contém todos os processos que iniciados na execução de um programa.
  - Cada processo possui um identificador único chamado de **ranque**, que vai de 0 até P-1, onde P é o número de processos em um comunicador.

# Comunicadores

**Comunicador**

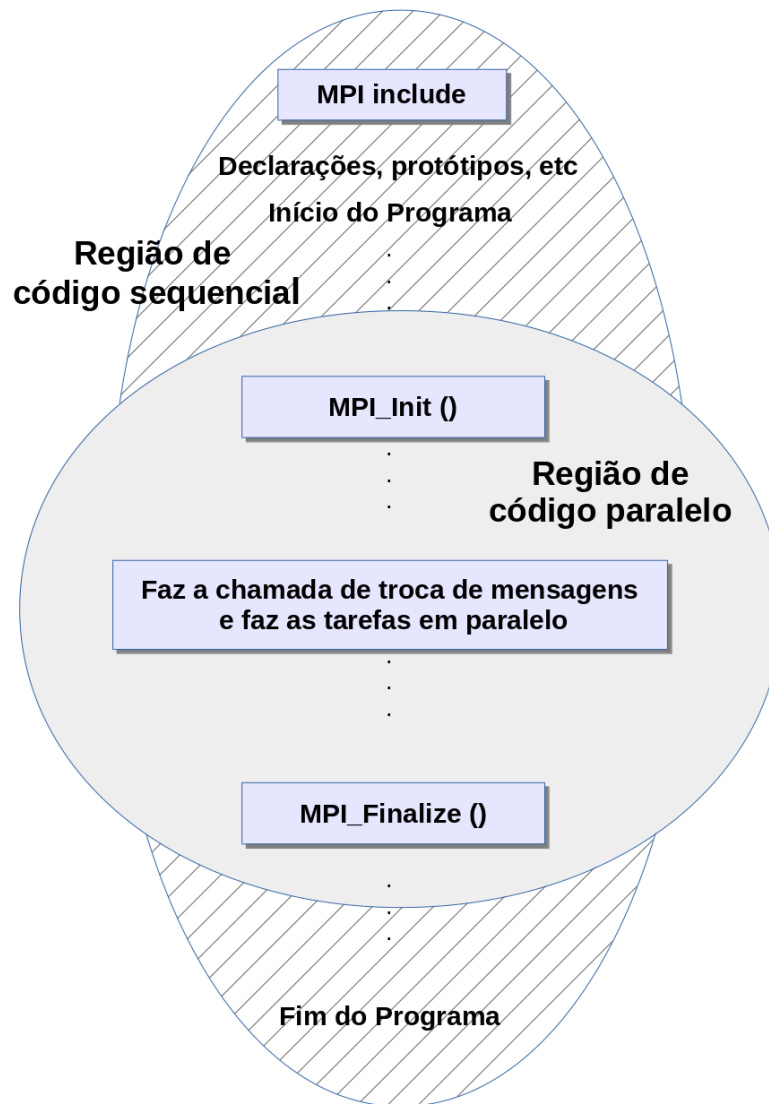


# Iniciando o MPI

- Todo programa em MPI deve conter a seguinte diretiva para o pré-processador:  
`#include "mpi.h"`
- Este arquivo, `mpi.h`, contém as definições, macros e funções de protótipos de funções necessários para a compilação de um programa MPI.
- Antes de qualquer outra função MPI ser chamada, a função `MPI_Init` deve ser chamada pelo menos uma vez.
- Seus argumentos são os ponteiros para os parâmetros do programa principal, `argc` e `argv`.

# Iniciando o MPI

- Esta função permite que o sistema realize as operações de preparação necessárias para que a biblioteca MPI seja utilizada.
- Ao término do programa a função **MPI\_Finalize** deve ser chamada.
- Esta função limpa qualquer pendência deixada pelo MPI, p. ex, recepções pendentes que nunca foram completadas.
- Tipicamente, um programa em MPI deve ter o seguinte leiaute:



# Iniciando o MPI

```

. . .
#include "mpi.h"
. . .
main(int argc, char** argv) {
. . .
/* Nenhuma função MPI pode ser chamada antes deste ponto */
MPI_Init(&argc, &argv);
. . .
MPI_Finalize();
/* Nenhuma função MPI pode ser chamada depois deste ponto*/
. . .
/* main */
. . .

```

# Funções Básicas

- Em algumas situações pode ser necessário verificar se as funções `MPI_Init` e `MPI_Finalize` já foram chamadas.
- A rotina `MPI_Initialized` indica se a função `MPI_Init` foi chamada, retornando um valor lógico verdadeiro (1) ou falso (0).
- A rotina `MPI_Finalized` indica se a função `MPI_Finalize` foi chamada.

```
int MPI_Initialized (int *flag)
```

```
int MPI_Finalized (int *flag)
```



# Funções Básicas

```
#include "mpi.h"

int main(int argc, char *argv[ ]) {
    int iniciado, finalizado;
    ...
    MPI_Initialized(&iniciado);
    if (!iniciado)
        MPI_Init(&argc, &argv);
    /* Realiza o trabalho em paralelo */
    ...
    /* Quando o programa está para terminar */
    MPI_Finalized(&finalizado);
    if (!finalizado)
        MPI_Finalize();
    return(0); }
```

# Quem sou eu?

- O MPI tem a função **MPI\_Comm\_Rank** que retorna o *ranque* de um processo no seu segundo argumento.

- Sua sintaxe é:

```
int MPI_Comm_Rank(MPI_Comm com, int *ranque)
```

- O primeiro argumento é um **comunicador**.  
Essencialmente um **comunicador** é uma coleção de processos que podem enviar mensagens entre si.
- Para os programas básicos, o único comunicador necessário é **MPI\_COMM\_WORLD**, que é pré-definido no MPI e consiste de todos os processos executando quando a execução do programa começa.

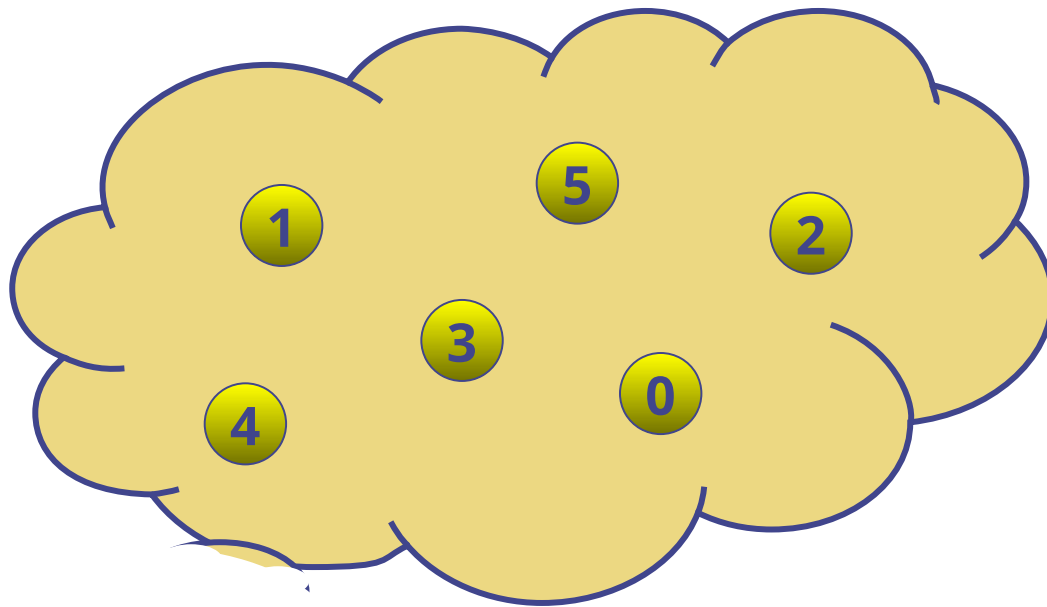
# Quantos processos existem?

- Muitas construções em nossos programas também dependem do número de processos executando o programa.
- O MPI oferece a função `MPI_Comm_size` para determinar este valor.
- Essa função retorna o número de processos em um comunicador no seu segundo argumento.
- Sua sintaxe é:

```
int MPI_Comm_size(MPI_Comm com, int *num_procs)
```

# Ranque de um Processo

`MPI_COMM_WORLD`



# Funções Básicas

- Abortando um programa:

```
int MPI_Abort(MPI_Comm com, int erro)
```

- Identificando a versão do MPI:

```
int MPI_Get_version(int *versao, int *subversao)
```

- Recuperando o nome do computador:

```
int MPI_Get_processor_name (char *nome, int  
*comprimento)
```

# Medindo o tempo de execução

- A função **MPI\_Wtime** retorna (em precisão dupla) o tempo total em segundos decorrido desde um instante determinado no passado.
- Esse instante é dependente de implementação, mas deve sempre o mesmo para uma dada implementação.
- A função **MPI\_Wtick** retorna (em precisão dupla) a resolução em segundos da função **MPI\_Wtime**.
- Um exemplo de uso dessas funções pode ser visto a seguir.

# Funções Básicas

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[ ]) {    /* mpi_wtime.c */
double tempo_inicial, tempo_final, a;
tempo_inicial = MPI_Wtime();
/* Realiza o trabalho em paralelo */
    for (long int i = 0; i < 1000000000000; i++) {
        a = (double) i;
    }
tempo_final = MPI_Wtime();
printf("Foram gastos %3.6f segundos para calcular a =
      %3.0f com precisão de %3.3e segundos\n",tempo_final-
      tempo_inicial, a, MPI_Wtick ());
return(0);
}
```

# Exemplo do Uso de Funções

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) {    /* mpi_funcoes.c */
int meu_rankue, num_procs;
int versao, subversao, aux, ret;
char maquina[MPI_MAX_PROCESSOR_NAME];
    /* Inicia o MPI. Em caso de erro aborta o programa */
    ret = MPI_Init(&argc, &argv);
    if (ret != MPI_SUCCESS) {
        printf("Erro ao iniciar o programa MPI. Abortando.\n");
        MPI_Abort(MPI_COMM_WORLD, ret);
    }
```



# Exemplo do Uso de Funções

```
/* Imprime a versão e subversão da biblioteca MPI */
MPI_Get_version(&versao,&subversao);
printf("Versão do MPI = %d Subversão = %d \n", versao,
subversao);

/* Obtém o ranque e número de processos em execução */
MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);

/* Define o nome do computador onde o processo está
executando */
MPI_Get_processor_name(maquina, &aux);
printf("Número de tarefas = %d Meu ranque = %d
Executando em %s\n", num_procs, meu_ranque, maquina);

/* Finaliza o MPI */
MPI_Finalize();
return(0);
}
```

# Exemplo do Uso de Funções

[https://github.com/gpsilva2003/MPI/mpi\\_funcoes.c](https://github.com/gpsilva2003/MPI/mpi_funcoes.c)



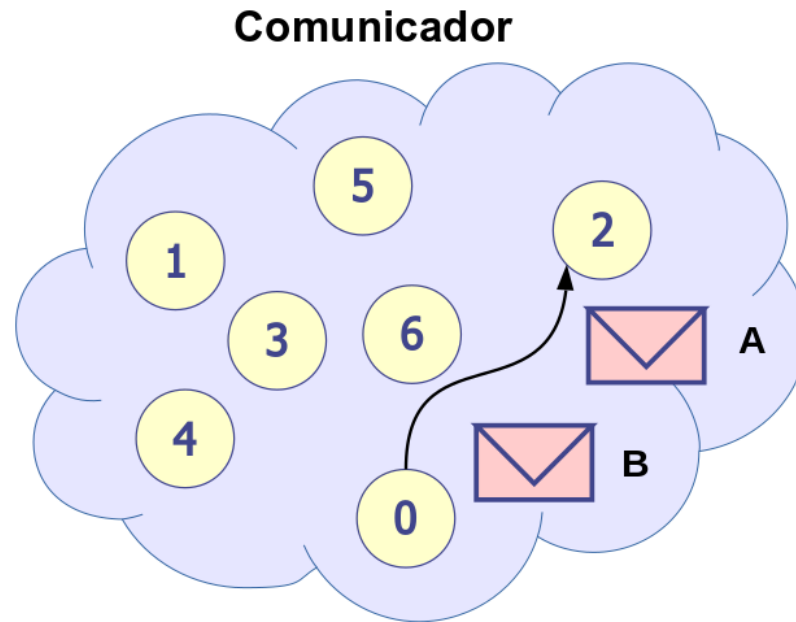


# Comunicação Ponto a Ponto

# Mensagem MPI

- Mensagem = Dados + Envelope
- Para que a mensagem seja comunicada com sucesso, o sistema deve anexar alguma informação aos dados que o programa de aplicação deseja transmitir.
- Essa informação adicional forma o envelope da mensagem, que no MPI contém a seguinte informação:
  - O *ranque* do processo origem.
  - O *ranque* do processo destino.
  - Uma *etiqueta* especificando o tipo da mensagem.
  - Um *comunicador* definindo o domínio de comunicação.

# Preservação da Ordem das Mensagens



- As mensagens não ultrapassam umas às outras.
- Por exemplo, se o processo com ranque 0 enviar duas mensagens sucessivas A e B, e o processo com ranque 2 chamar duas rotinas de recepção que combinam com qualquer uma das mensagens, a ordem das mensagens é preservada, sendo que A será sempre recebida antes de B.

# Comunicação Ponto a Ponto

- Quando dois processos estão se comunicando utilizando `MPI_Send` e `MPI_Recv`, a importância do uso do `comunicador` aumenta quando módulos de um programa foram escritos independentemente um do outro.
- Por exemplo, se você quiser utilizar uma biblioteca para resolver um sistema de equações lineares, você pode criar um `comunicador` para ser utilizado exclusivamente pelo solucionador linear e evitar que suas mensagens seja confundidas com outro programa que utilize as mesmas *etiquetas*.

# Comunicação Ponto a Ponto

- Vamos utilizar por enquanto o comunicador pré-definido **MPI\_COMM\_WORLD**.
- Ele é composto por todos os processos ativos desde que a execução do programa iniciou.
- O mecanismo real de troca de mensagens em nossos programas é executado no MPI pelas funções **MPI\_Send** e **MPI\_Recv**.
- A primeira envia a mensagem para um determinado processo e a segunda recebe a mensagem de um processo.
- Ambas são **bloqueantes**. O envio bloqueante espera até que todos os dados tenham sido copiados dos **buffers** de envio. A recepção bloqueante espera até que o **buffer** de recepção contenha a mensagem.

# Comunicação Ponto a Ponto

- Correspondência entre os tipos MPI e C:

## MPI datatype

MPI\_CHAR

MPI\_SHORT

MPI\_INT

MPI\_LONG

MPI\_UNSIGNED CHAR

MPI\_UNSIGNED SHORT

MPI\_UNSIGNED

MPI\_UNSIGNED LONG

MPI\_FLOAT

MPI\_DOUBLE

MPI\_LONG DOUBLE

MPI\_BYTE

MPI\_PACKED

## C datatype

signed char

signed short int

signed int

signed long int

unsigned char

unsigned short int

unsigned int

unsigned long int

float

double

long double



# Comunicação Ponto a Ponto

- Os dois últimos tipos, **MPI\_BYTE** e **MPI\_PACKED** não correspondem aos tipos padrão em C.
- O tipo **MPI\_BYTE** pode ser usado se você desejar não realizar nenhuma conversão entre tipos de dados diferentes.
- O tipo **MPI\_PACKED** será discutido posteriormente.
- Note que a quantidade de espaço alocado pelo buffer de recepção não precisa ser igual a quantidade de espaço na mensagem recebida.
- O MPI permite que uma mensagem seja recebida enquanto houver espaço suficiente alocado.

# MPI\_Send

```
int MPI_Send(void* mensagem, int cont, MPI_Datatype  
tipo_mpi, int destino, int etiq, MPI_Comm com)
```

- *mensagem*: endereço inicial do dado a ser enviado.
- *cont*: número de dados.
- *tipo\_mpi*: MPI\_CHAR, MPI\_INT, MPI\_FLOAT, MPI\_BYTE, MPI\_LONG, MPI\_UNSIGNED\_CHAR, etc.
- *destino*: *ranque* do processo destino.
- *etiq*: etiqueta da mensagem.
- *com* : comunicador que especifica o contexto da comunicação e os processos participantes do grupo. O *comunicador* padrão é MPI\_COMM\_WORLD.

# Comunicação Ponto a Ponto

- A maioria das funções MPI retorna um código de erro inteiro.
- Contudo, como a maioria dos programadores em C, nós vamos ignorar este código a maior parte das vezes.
- O conteúdo das mensagens são armazenados em um bloco de memória referenciado pelo argumento *mensagem*.
- Os próximos dois argumentos, *cont* e *tipo\_mpi* permitem ao sistema identificar o final da mensagem: eles contêm uma sequência de valores de contagem, cada um contendo um tipo de dados MPI.

# MPI\_Recv

```
int MPI_Recv(void* mensagem, int cont, MPI_Datatype  
tipo_mpi, int origem, int etiq, MPI_Comm com,  
MPI_Status* estado)
```

- *mensagem*: Endereço inicial do buffer de recepção
- *cont*: Número máximo de dados a serem recebidos
- *tipo\_mpi*: MPI\_CHAR, MPI\_INT, MPI\_FLOAT, MPI\_BYTE, MPI\_LONG, MPI\_UNSIGNED\_CHAR, etc.
- *origem*: *ranque* do processo origem ( \* = MPI\_ANY\_SOURCE)
- *etiq*: etiqueta da mensagem ( \* = MPI\_ANY\_TAG)
- *com*: comunicador
- *estado*: Estrutura com três campos: MPI\_SOURCE, MPI\_TAG, MPI\_ERROR.

# Programa Simples em MPI

[https://github.com/gpsilva2003/MPI/mpi\\_simples.c](https://github.com/gpsilva2003/MPI/mpi_simples.c)

# Programa Simples em MPI

```
#include <stdio.h>
```

```
#include <string.h>
```

```
#include "mpi.h"
```

```
main(int argc, char** argv) {
```

```
    int    meu_ranque, num_procs, origem, destino, etiq=0;
```

```
    char  mensagem[100];
```

```
    MPI_Status estado;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_ranque);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```

# Programa Simples em MPI

```
if (meu_ranque != 0) {
    sprintf(msg, "Processo %d está vivo!", meu_ranque);
    destino = 0;
    MPI_Send(mensagem, strlen(mensagem)+1, MPI_CHAR, destino, etiq,
    MPI_COMM_WORLD);
}
else {
    for (origem=1; origem < num_procs; origem++) {
        MPI_Recv(mensagem, 100, MPI_CHAR, origem, etiq,
        MPI_COMM_WORLD, &estado);
        printf("%s\n", mensagem);
    }
}
MPI_Finalize( ); }
```

# Comunicação Ponto a Ponto

- Os argumentos *destino* e *origem* são, respectivamente, o *ranque* dos processos de recepção e de envio.
- O MPI permite que *origem* seja um coringa (\*), neste caso usamos **MPI\_ANY\_SOURCE** neste parâmetro.
- Não há coringa para o destino.
- O *etiq* é um inteiro e, por enquanto, **MPI\_COMM\_WORLD** é nosso único comunicador.
- Existe um coringa, **MPI\_ANY\_TAG**, que MPI\_Recv pode usar como *etiqueta*.
- Não existe coringa para o comunicador.



# Comunicação Ponto a Ponto

- Esses itens podem ser usados pelo receptor para distinguir entre as mensagens entrantes.
- O argumento *origem* pode ser usado para distinguir mensagens recebidas de diferentes processos.
- O *etiq* é especificado pelo usuário para distinguir mensagens de um único processo.
- O MPI garante que inteiros entre 0 e 32767 possam ser usados como *etiquetas*. Muitas implementações permitem valores maiores.
- Um *comunicador* é basicamente uma coleção de processos que podem enviar mensagens uns para os outros.

# Comunicação Ponto a Ponto

- Em outras palavras, para que o processo A possa enviar uma mensagem para o processo B; os argumentos que A usa em `MPI_Send` devem ser idênticos ao que B usa em `MPI_Recv`.
- O último argumento de `MPI_Recv`, *estado*, retorna a informação sobre os dados realmente recebidos.
- Este argumento referencia um registro com dois campos: um para *origem* e outro para *etiq*.
- Então, por exemplo, se a *origem* da recepção era `MPI_ANY_SOURCE`, então o *estado* irá conter o *ranque* do processo que enviou a mensagem.

# Utilizando o “Handle Status”

- Informação sobre a recepção com o uso de coringa é retornada pela função **MPI\_Recv** no “handle status”.

Informação	C
remetente	status.MPI_SOURCE
etiqueta	status.MPI_TAG
erro	status.MPI_ERROR

- Para saber o total de elementos recebidos utilize a rotina:

```
int MPI_Get_count( MPI_Status *status,  
MPI_Datatype datatype, int *count )
```

# Exemplo de Uso do Status

[https://github.com/gpsilva2003/MPI/mpi\\_status.c](https://github.com/gpsilva2003/MPI/mpi_status.c)

# Verificando as mensagens recebidas

- Agora que já vimos como o objeto **MPI\_Status** funciona, podemos utilizá-lo junto com a rotina **MPI\_Probe** para determinar o tamanho de uma mensagem antes de efetivamente recebê-la.
- Isso nos permite dimensionar a variável de recepção adequadamente, ao invés de reservar um espaço exageradamente grande, para todos os possíveis tamanhos de mensagem.
- A função **MPI\_Probe** tem grande utilidade em aplicações do tipo mestre/trabalhador onde há grande troca de mensagens de tamanho variável.

# MPI\_Probe

int MPI\_Probe(int origem, int etiq, MPI\_Comm com,  
MPI\_Status\* estado)

- A função **MPI\_Probe** é muito parecida com a função **MPI\_Recv**.
- Em realidade, a primeira realiza as mesmas funções que a segunda, menos receber a mensagem.
- A função **MPI\_Probe** irá bloquear esperando por uma mensagem com a origem e etiqueta correspondentes. Quando a mensagem estiver disponível, ela irá preencher a estrutura estado com a informação apropriada.
- O usuário pode então utilizar a função **MPI\_Recv** para receber a mensagem verdadeira.

# Exemplo do Uso MPI\_Probe

[https://github.com/gpsilva2003/MPI/mpi\\_probe.c](https://github.com/gpsilva2003/MPI/mpi_probe.c)

# Estudo de Caso – Integral definida método trapézio

- Vamos lembrar que o método do trapézio estima o valor de  $f(x)$  dividindo o intervalo  $[a; b]$  em  $n$  segmentos iguais e calculando a seguinte soma:

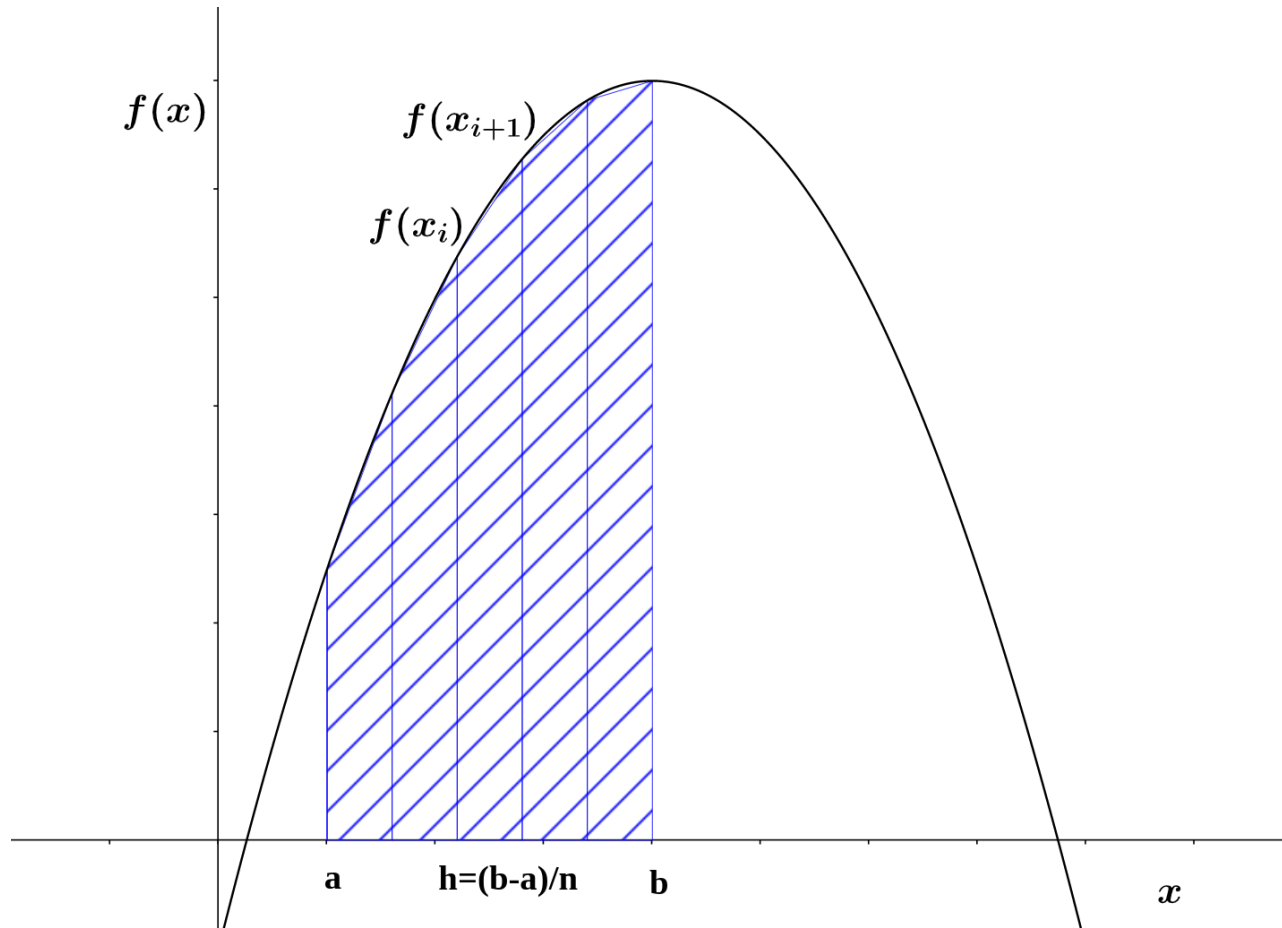
$$h * \left[ \frac{f(x_0)}{2} + \frac{f(x_n)}{2} + \sum_{i=1}^{n-1} f(x_i) \right] \quad (3.1)$$

$$h = \frac{(b-a)}{n} \text{ e } x_i = a + i * h, i = 1, \dots, (n - 1)$$

- Colocando  $f(x)$  em uma rotina, podemos escrever um programa para calcular uma integral utilizando o método do trapézio.



# Estudo de Caso – Integral definida método trapézio



# Estudo de Caso – Integral definida método trapézio

```
/* A função f(x) é pré-definida.
 * Entrada: a, b, n.
 * Saída: estimativa da integral de a até b de f(x).
 */
#include <stdio.h>
float f(float x) {
    float return_val;
    /* Calcula f(x). Armazena resultado em return_val. */
    ...
    return return_val;
} /* f */
main() {
    float integral;      /* Armazena resultado em integral */
    float a, b;          /* Limite esquerdo e direito */
    int n;                /* Número de Trapezóides */
    float h;              /* Largura da base do Trapezóide */
```

# Estudo de Caso – Integral definida método trapézio

```
float x;  
int i;  
    printf("Entre a, b, e n \n");  
    scanf("%f %f %d", &a, &b, &n);  
    h = (b-a)/n;  
    integral = (f(a) + f(b))/2.0;  
    x = a;  
    for (i = 1; i != n-1; i++) {  
        x += h;  
        integral += f(x);  
    }  
    integral *= h;  
    printf("Com n = %d trapezóides, a estimativa \n", n);  
    printf("da integral de %f até %f = %f \n", a, b, integral);  
} /* main */
```

# Estudo de Caso – Integral definida método trapézio

- Uma forma de paralelizar este programa é simplesmente dividir o intervalo  $[a;b]$  entre os processos e cada processo pode fazer a estimativa do valor da integral de  $f(x)$  em seu subintervalo.
- Para calcular o valor total da integral, os valores calculados localmente são adicionados.
- Suponha que há “ $p$ ” processos e “ $n$ ” trapézios e, de modo a simplificar a discussão, também supomos que “ $n$ ” é divisível por “ $p$ ”.
- Então é natural que o primeiro processo calcule a área dos primeiros “ $n/p$ ” trapézios, o segundo processo calcule a área dos próximos “ $n/p$ ” e assim por diante.

# Estudo de Caso – Integral definida método trapézio

- Então, o processo  $q$  irá estimar a integral sobre o intervalo:

$$\left[ a + q \frac{nh}{p}, a + (q + 1) \frac{nh}{p} \right]$$

- Logo cada processo precisa da seguinte informação:
  - O número de processos,  $p$ .
  - Seu *ranque*.
  - O intervalo inteiro de integração,  $[a; b]$ .
  - O número de subintervalos,  $n$ .

# Estudo de Caso – Integral definida método trapézio

- Lembre-se que os dois primeiros itens podem ser encontrados chamando as funções MPI:

`MPI_Comm_size`

`MPI_Comm_rank`

- Os dois últimos itens podem ser fornecidos pelo usuário.
- Para a nossa primeira tentativa de paralelização, vamos dar valores fixos atribuídos no programa.
- Uma maneira direta de calcular a soma de todos os valores locais é fazer cada processo enviar o seu resultado para o processo 0 e este processo fazer a soma final.

# Exemplo do método trapézio

[https://github.com/gpsilva2003/MPI/mpi\\_trapezio.c](https://github.com/gpsilva2003/MPI/mpi_trapezio.c)

# Medidas de Desempenho

- Speed-up (Aceleração):

Mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador e o tempo gasto na execução com  $n$  processadores:

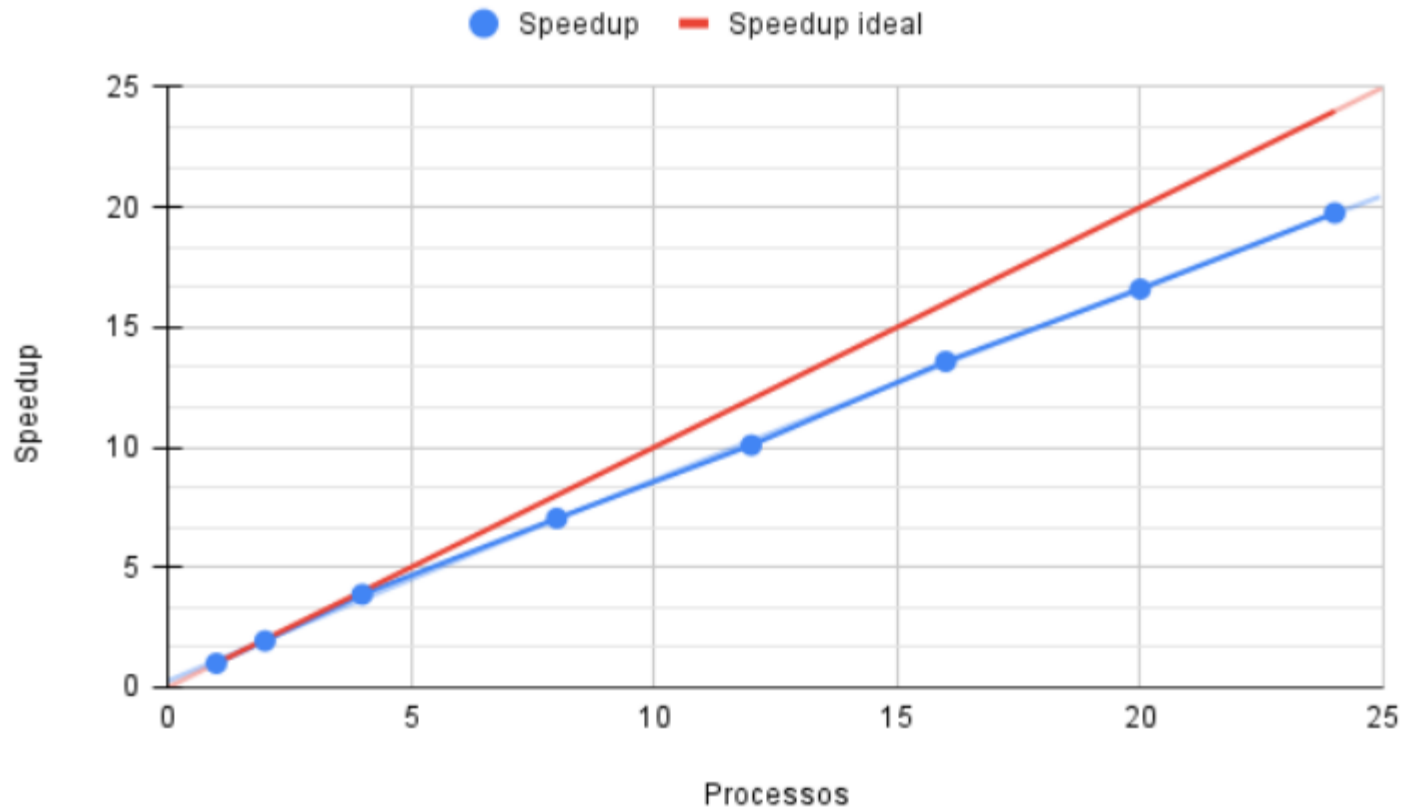
$$S(n) = T(1)/T(n)$$

- Eficiência:

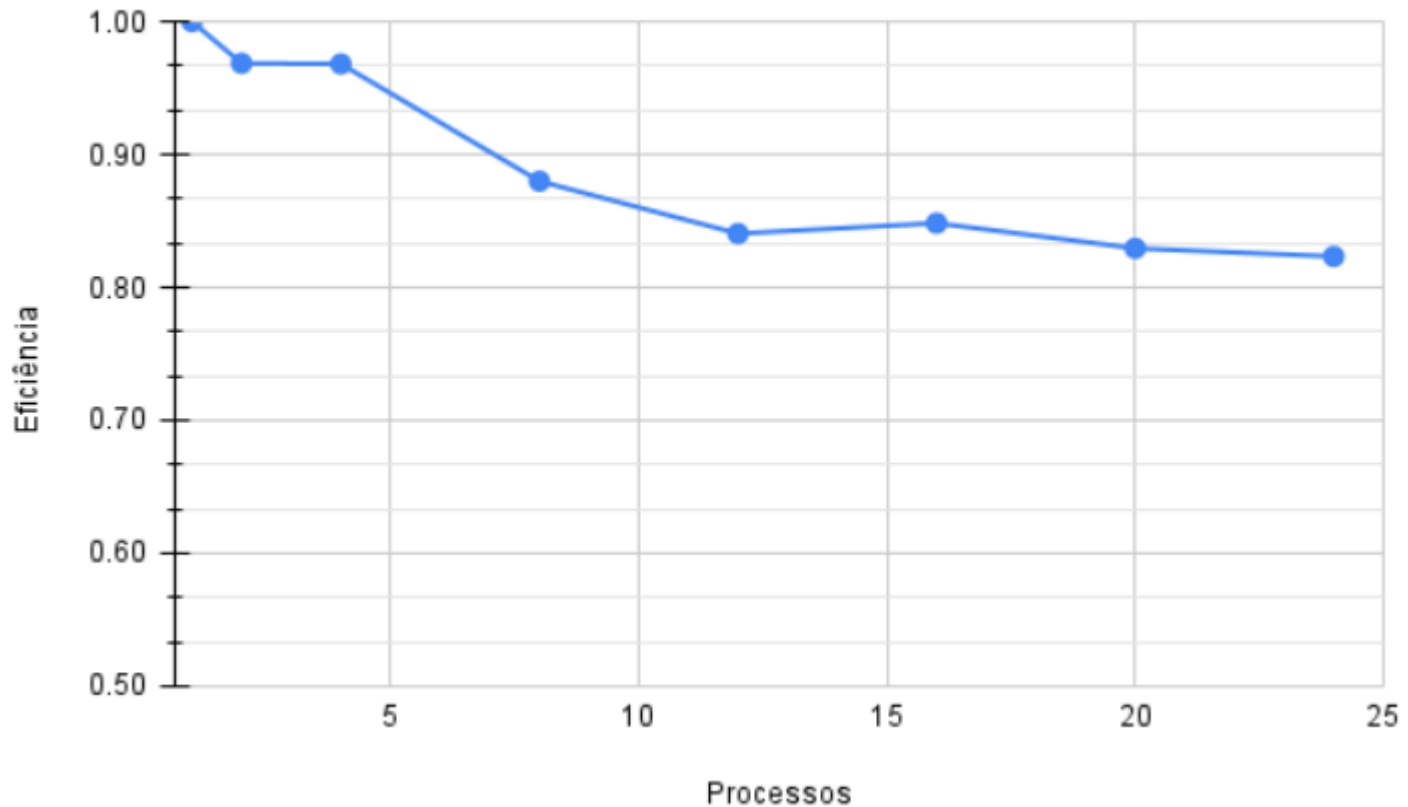
$$E(n) = S(n)/n$$



# Método do Trapézio - Speedup



# Método do Trapézio - Eficiência





# Comunicação Coletiva

# 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.
  - As funções estão disponíveis apenas no modo padrão\*.
- Todos os processos participantes da comunicação coletiva chamam a mesma função com argumentos compatíveis.

\* Nas últimas versões do padrão já existem versões não bloqueantes das rotinas de comunicação coletiva.

# Comunicação Coletiva

- Quando uma operação coletiva possui 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 *Gather* seguido de uma difusão.

- Redução (*reduce*)

Realiza as operações coletivas de soma, máximo, mínimo, etc.

- Redução com difusão (**Allreduce**)

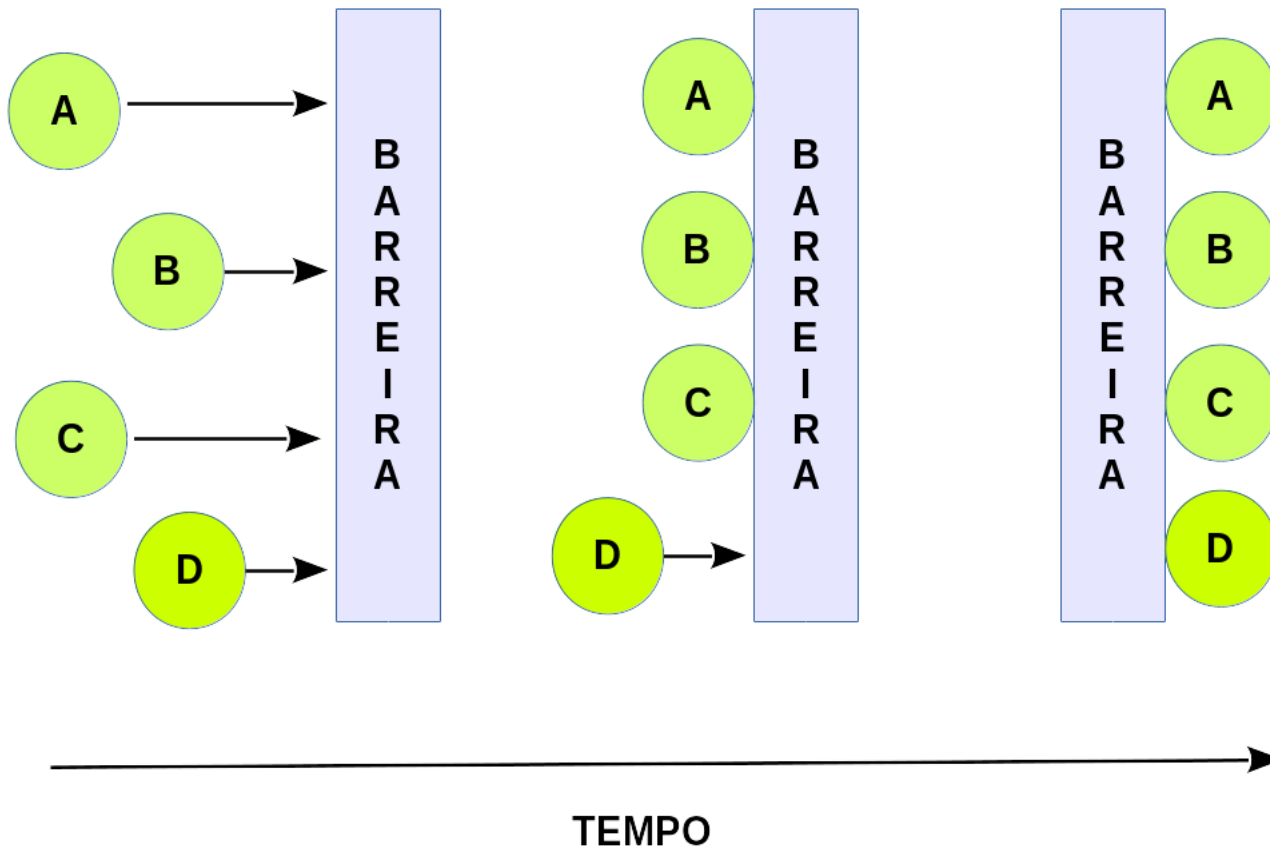
Uma redução seguida de uma difusão.

- Alltoall

Conjunto de *gathers* onde cada processo recebe dados diferentes.\*

\* Não vamos abordar neste nosso estudo

# Barreira



# MPI\_Barrier

```
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 em ***com*** tenham chamado **MPI\_Barrier**.



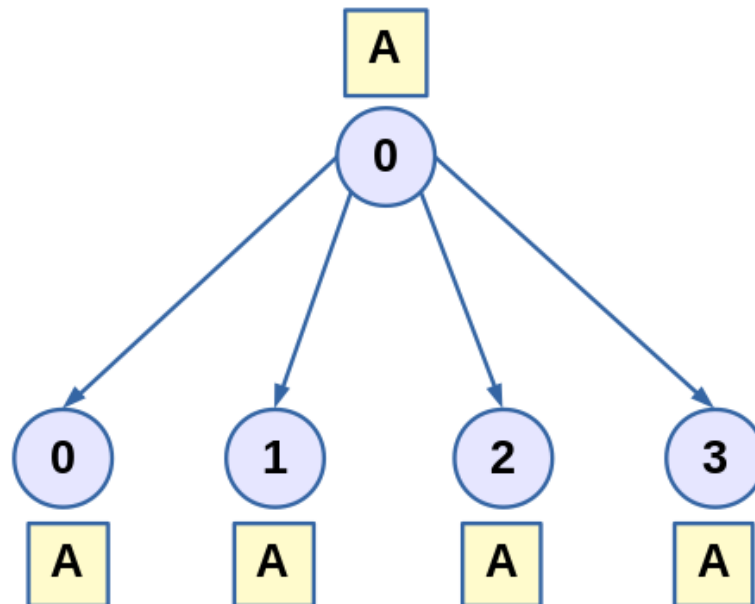
# MPI\_Bcast

- 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)
```

# Difusão

**MPI\_Bcast**



# MPI\_Bcast

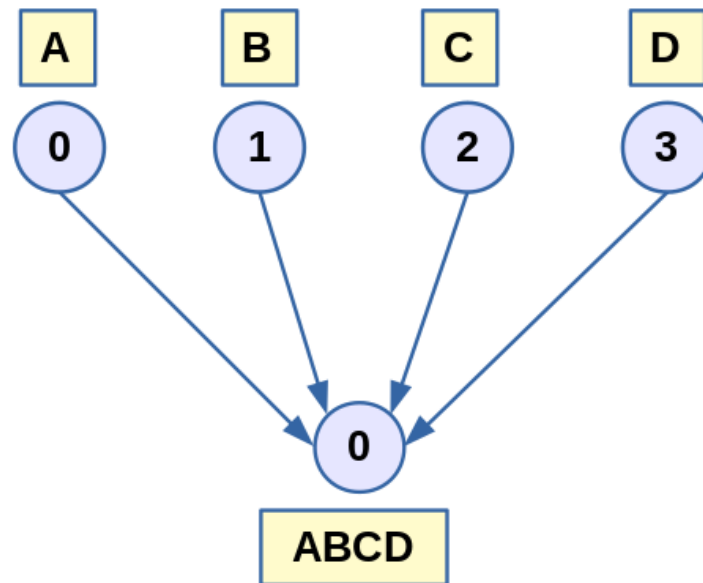
- Ela simplesmente envia uma cópia dos dados de *mensagem* no processo *raiz* para cada processo no comunicador *com*.
- Deve ser chamado por todos os processos no comunicador com os mesmos argumentos para *raiz* e *com*.
- Uma mensagem de broadcast não pode ser recebida com *MPI\_Recv*.
- Os parâmetros *cont* e *tipo\_mpi* têm a mesma função que nas funções *MPI\_Send* e *MPI\_Recv*: especificam o tamanho da mensagem.

# Comunicação Coletiva

- Contudo, ao contrário das funções ponto-a-ponto, o padrão MPI exige que *cont* e *tipo\_mpi* sejam os mesmos para todos os processos no mesmo *comunicador* para uma comunicação coletiva.
- A razão para isto é que um único processo pode receber dados de muitos outros processos, e para poder determinar o total de dados recebidos, seria necessário um vetor inteiro de status de retorno.

# Coleta

**MPI\_Gather**



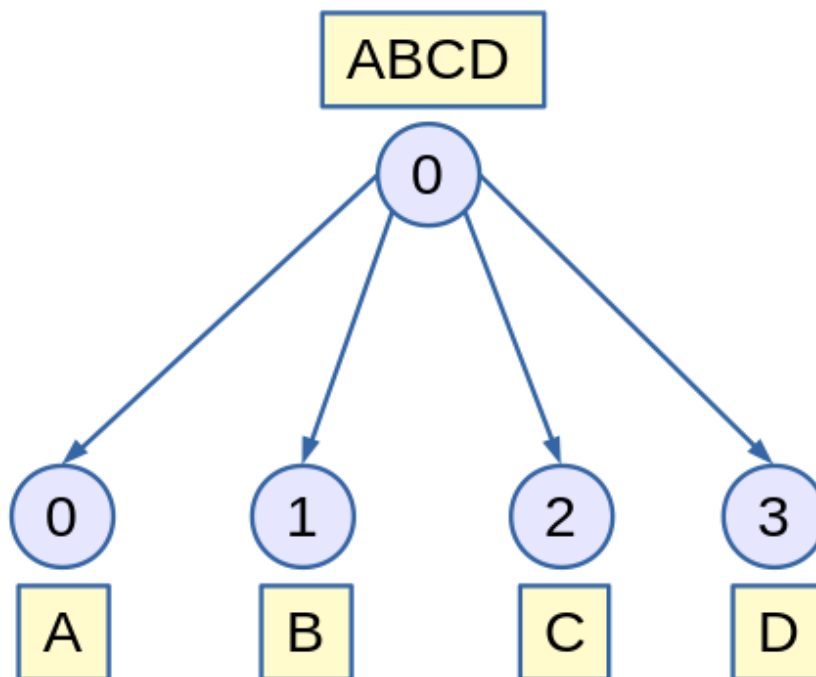
# MPI\_Gather

```
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 em *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 *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*.

# Dispersão

**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)
```

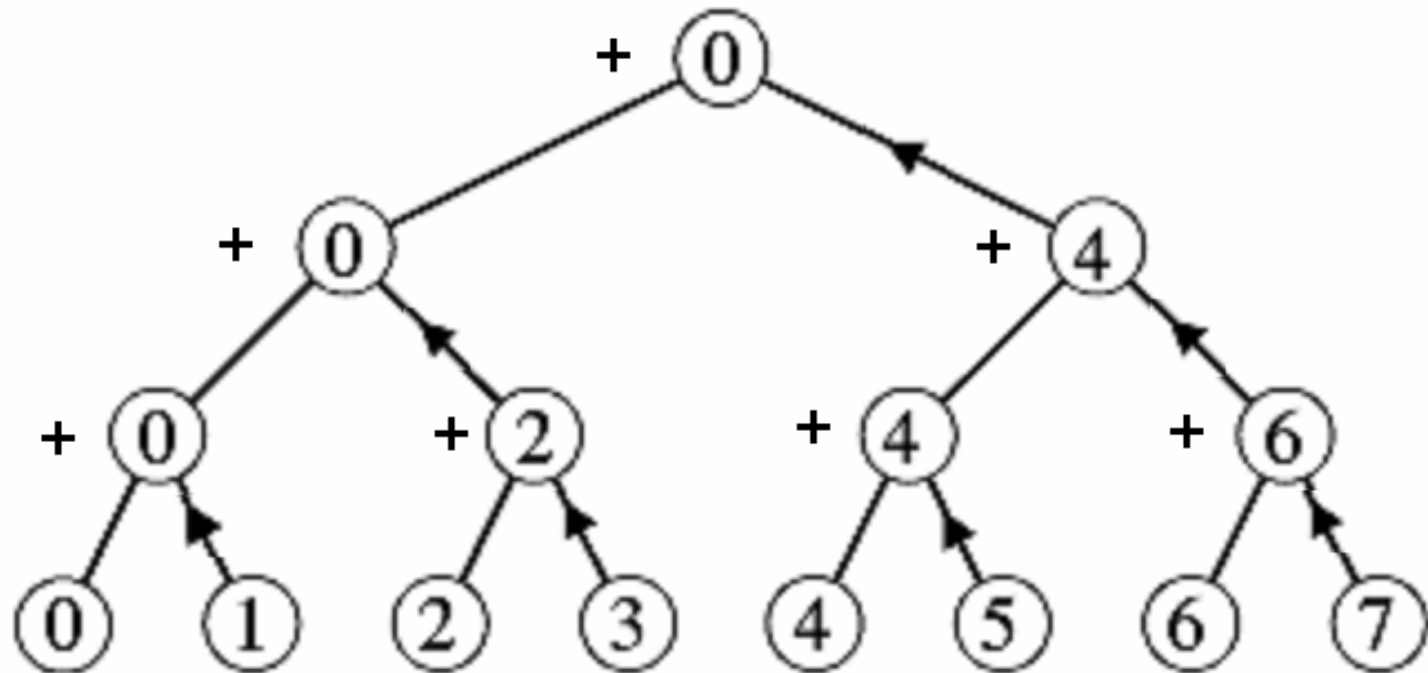
- O processo com o *ranque* igual a *raiz* distribui o conteúdo de *vet\_envia* entre os processos.
- O conteúdo de *vet\_envia* é dividido em *p* segmentos, cada um deles consistindo de *cont\_envia* itens.
- O primeiro segmento vai para o processo com ranque 0, o segundo para o processo com ranque 1, etc.
- O argumento *vet\_envia* é significativo apenas no processo *raiz*.



# Redução

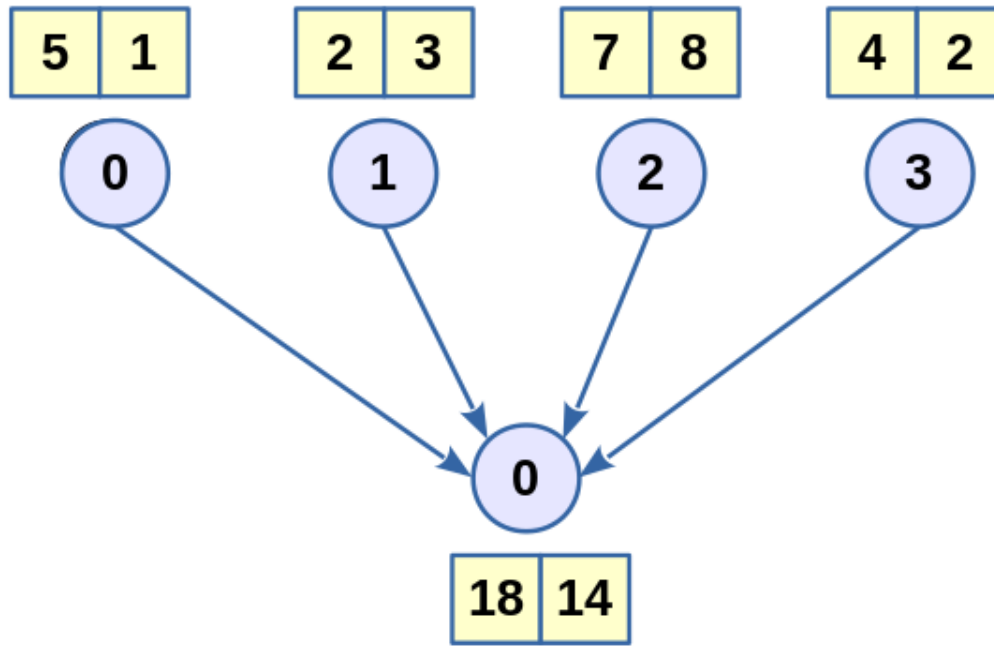
- No programa do método do trapézio, depois da fase de entrada, cada processador executa os mesmos comandos até o final da fase de soma.
- Contudo, este não é caso depois da final da fase de soma, onde as tarefas não são bem balanceadas.
- Podemos utilizar o seguinte procedimento:
  - a) 1 envia resultado para 0, 3 para 2, 5 para 4, 7 para 6.
  - b) 0 soma sua integral com a de 1, 2 soma com a de 3, etc.
  - c) 2 envia para 0, 6 envia para 4.
  - d) 0 soma, 4 soma.
  - e) 4 envia para 0.
  - f) 0 soma.

# Redução



# Redução

**MPI\_Reduce**



**MPI\_SUM**

# Redução

- A soma global que estamos tentando calcular 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.

# MPI\_Reduce

- O argumento *oper* pode ter um dos seguintes valores pré-definidos:

## Nome da Operação Significado

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

# MPI\_Reduce

- Com um exemplo, vamos reescrever as últimas linhas do programa do método do trapézio:

...

```
/* Adiciona as integrais calculadas por cada processo */  
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,  
MPI_SUM, 0, MPI_COMM_WORLD);  
/* Imprime o resultado */
```

...

- Note que cada processo chama a rotina **MPI\_REDUCE** com os mesmos argumentos.
- Em particular, embora *total* tenha apenas significado no processo 0, cada processo deve fornecê-lo como argumento.

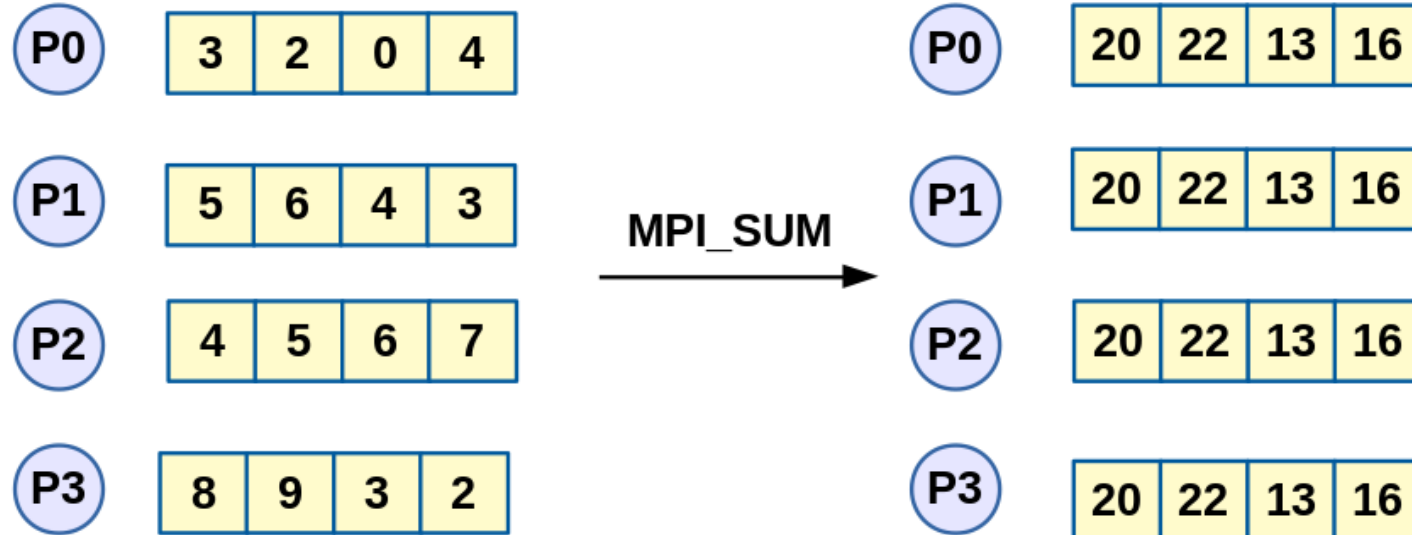
# Exemplo Redução

[https://github.com/gpsilva2003/MPI/mpi\\_reduce.c](https://github.com/gpsilva2003/MPI/mpi_reduce.c)



# Redução com Difusão

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** armazena o resultado da operação de redução *oper* no buffer *vet\_recebe* de cada processo.

# Estudo de caso – Multiplicação Matriz - Vetor

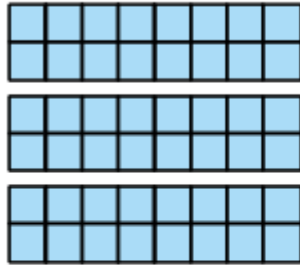
$$A_{m,n} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \quad b_n = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}$$
$$c_m = Ab = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{m-1} \end{bmatrix}$$

$$c_i = a_{i,0}.b_0 + a_{i,1}.b_1 + a_{i,2}.b_2 + \cdots + a_{i,n-1}.b_{n-1}$$

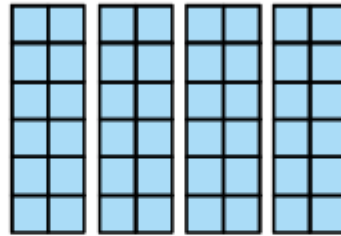
# Estudo de caso – Multiplicação Matriz - Vetor

$$A \times b = c$$
$$\begin{bmatrix} 2 & 1 & 3 & 4 & 0 \\ 5 & -1 & 2 & -2 & 4 \\ 0 & 3 & 4 & 1 & 2 \\ 2 & 3 & 1 & -3 & 0 \end{bmatrix} \times \begin{bmatrix} 3 \\ 1 \\ 4 \\ 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 19 \\ 34 \\ 25 \\ 13 \end{bmatrix}$$

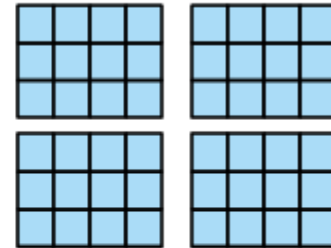
# Estudo de caso – Multiplicação Matriz - Vetor



Decomposição em  
blocos no sentido  
das linhas



Decomposição em  
blocos no sentido  
das colunas



Decomposição em  
blocos  $j \times k$

# Estudo de caso – Multiplicação Matriz - Vetor

- Cada uma dessas formas de decomposição tem as suas vantagens e desvantagens, além de complexidades distintas.
- Por simplicidade, vamos assumir que utilizaremos a primeira alternativa de distribuição de dados, com cada processo possuindo um bloco de linhas da matriz  $A$  e os vetores  $b$  e  $c$  replicados em cada processo.
- Uma análise simples de complexidade, supondo-se  $m = n$ , indica uma complexidade computacional sequencial de  $O(n^2)$ . Quando  $p$  processos são utilizados, a complexidade computacional por processo, sem custos de comunicação, igual a  $O(n^2 / p)$ .

# Estudo de caso – Multiplicação Matriz - Vetor

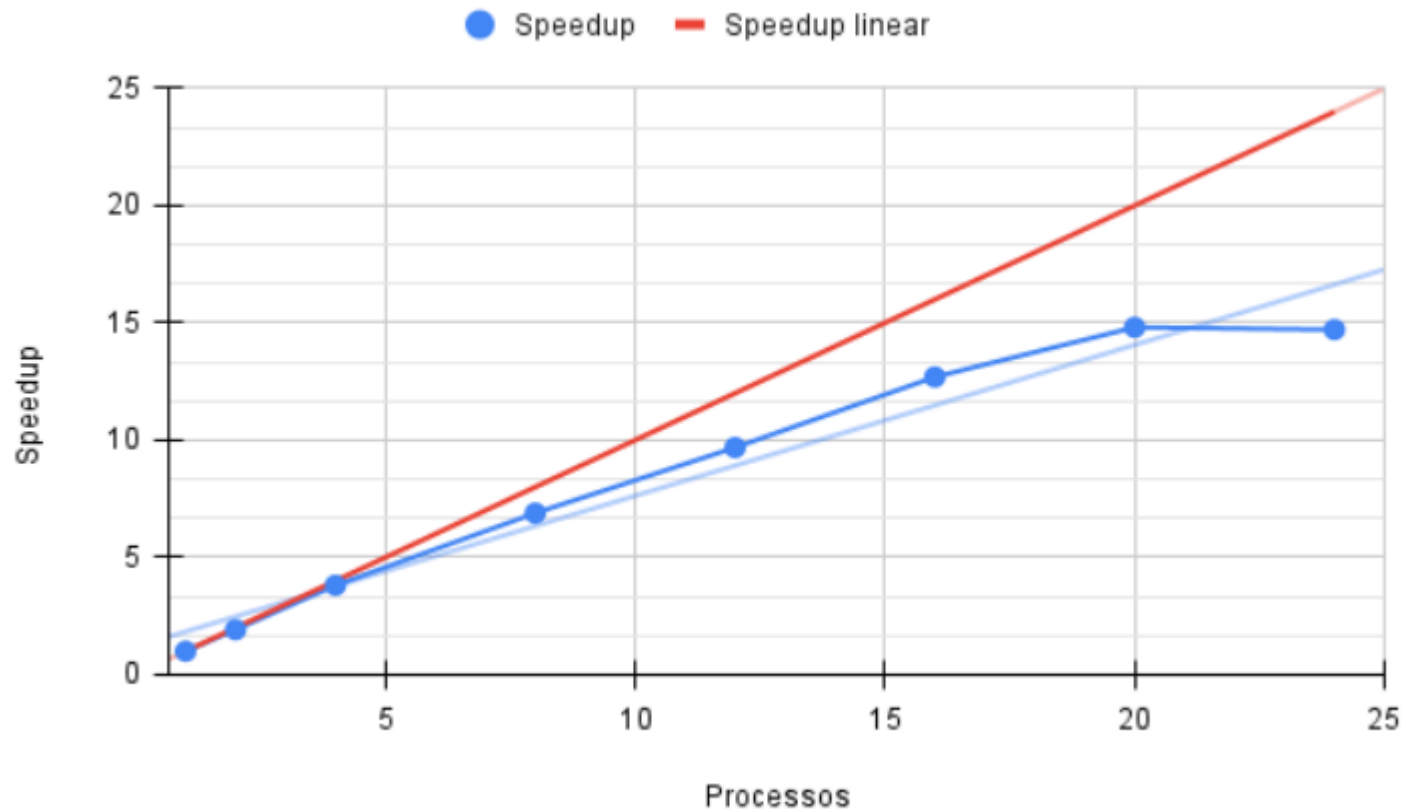
- Para os custos de comunicação, não incluindo o custo do envio da linha  $i$  e do vetor  $b$ , e apenas a coleta e difusão do vetor de resultado  $c$  para todos os processos, temos a seguinte situação.
- Um algoritmo eficiente para a função **MPI\_Allgather** requer que cada processo envie  $\lceil \log_2 p \rceil$  mensagens, com o número total de elementos enviados por processo igual a  $n(p - 1)/p$ .
- Então a complexidade de comunicação é igual a  $O(n + \log_2 p)$ , e a complexidade total desse algoritmo igual a  $O(n^2/p + n + \log_2 p)$  (J et al., 2007).

# Exemplo Multiplicação Matriz – Vetor

[https://github.com/gpsilva2003/MPI/mpi\\_mxv.c](https://github.com/gpsilva2003/MPI/mpi_mxv.c)

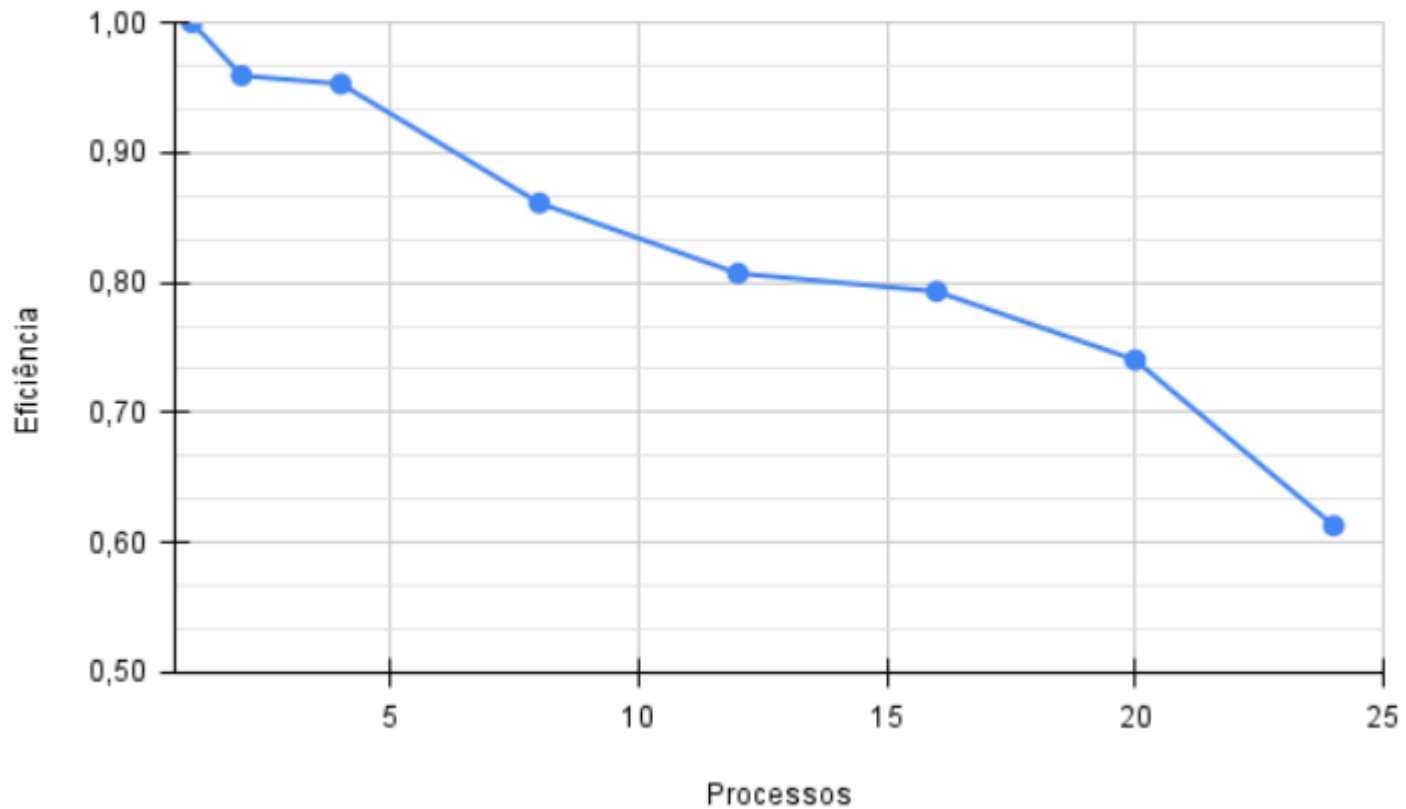


# Multiplicação Matriz – Vetor Speedup



# Multiplicação Matriz – Vetor

## Eficiência



# Comunicação Coletiva - Aplicação

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

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

- Calculada em paralelo, dividindo-se o intervalo de integração entre os processos.

# Cálculo de Pi

```
#include "mpi.h"
#include <math.h>
int main (argc, argv)
int argc;
char argv[ ];
{
    int n, meu_rank, num_procs, i, rc;
    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_rank);
    if (meu_rank == 0)
    { printf ("Entre com o número de intervalos: ");
      scanf("%d", &n);
    }
}
```

# Cálculo de Pi

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n != 0)
{
    h=1.0/(double) n;
    for (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_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_WORLD_COMM);
    if (meu_ranque == 0) printf ("valor aproximado de pi: %.16f \n",
pi);
}
MPI_Finalize( );
}
```



# Modos de Comunicação

# Rotinas Boqueantes e Não-Bloqueantes

- Cada rotina de envio/recepção de mensagens do MPI possui sempre duas versões, uma bloqueante e outra não-bloqueante.
- Nas versões bloqueantes as rotinas de envio/recepção não retornam até que os dados tenham sido totalmente copiados do espaço do usuário para uma área do sistema (no caso do envio) ou estejam completamente disponíveis para uso no espaço do usuário (no caso da recepção).
- Ou seja, do ponto de vista do programador, ao retorna da rotina, a operação desejada foi realizada completamente e as variáveis utilizadas para o envio podem ser reutilizadas para o envio de uma nova mensagem ou os dados recebidos já podem ser utilizados na computação.

# Rotinas Bloqueantes e Não-Bloqueantes

- Nas versões não-bloqueantes, contudo, a computação prossegue de imediato após a chamada das rotinas, sendo necessário, posteriormente, verificar se a operação realmente já terminou ou não, para que as variáveis possam ser utilizadas para o envio de uma nova mensagem, ou os dados recebidos utilizados na computação.
- Podemos dizer, de certa maneira, que as versões não-bloqueantes indicam apenas a intenção de realizar a operação de envio ou recepção da mensagem, sendo necessário verificar posteriormente, se as operações desejadas já foram efetivamente realizadas.
- As rotinas de comunicação não-bloqueantes permitem sobreposição da computação com a comunicação.



# Comunicação Bloqueante

- Na comunicação bloqueante:
  - A rotina de recepção não completa até que o *buffer* de recepção no espaço de usuário esteja cheio (mensagem disponível para uso).
  - A rotina de envio não completa até que o *buffer* de envio no espaço de usuário esteja vazio (*buffer* disponível para reuso).
- Uma rotina de envio bloqueante só vai retornar depois que houver garantia que os dados da aplicação (os seus dados de envio) possam ser reutilizados.
- Por garantia, entenda-se que modificações posteriores não afetarão os dados que estão sendo enviados. Essa garantia não implica que os dados foram efetivamente recebidos por outro processo - podem muito bem estar situados em um buffer de sistema.

# Comunicação Bloqueante

- Um envio bloqueante pode ser **síncrono**, obrigando a realização de um protocolo de confirmação com a rotina de recepção, para assegurar o envio completo da mensagem.
- Um envio bloqueante também pode ser **assíncrono**, desde que um buffer de sistema seja utilizado para armazenar os dados antes de sua distribuição para a respectiva rotina de recepção.
- O sucesso da operação de comunicação depende do tamanho da mensagem e do tamanho do buffer do sistema, sendo que as mensagens maiores que o espaço disponível no buffer do sistema serão enviadas no modo síncrono.
- Um programa correto em MPI não pode depender do uso de um buffer de sistema. Programas assim são chamados inseguros, embora possam executar e produzir resultados corretos na maior parte das vezes.

# Comunicação Não-Bloqueante

- No envio não-bloqueante a computação prossegue e, quando for necessário, verificamos se a operação já terminou ou não.
- As rotinas de comunicação não-bloqueantes retornam (imediatamente) um “handle request”.
- Este *handle* deve ser utilizado para verificar a chegada/envio da mensagem, podendo ser testado um única vez ou usado para ficar-se em espera ocupada.
- Sendo assim, se a programação for feita de maneira adequada, podemos realizar computação e comunicação em paralelo, melhorando o desempenho final do programa.
- Caso **não** estejamos utilizando *buffers*, devemos utilizar as rotinas de envio/recepção não-bloqueantes, para garantir que não haverá “deadlock”.

# Operações Não\_Bloqueantes

int MPI\_Isend(void\* mensagem, int cont, MPI\_Datatype  
tipo\_mpi, int destino, int etiq, MPI\_Comm com,  
MPI\_Request \*pedido)

int MPI\_Irecv(void\* mensagem, int cont, MPI\_Datatype  
tipo\_mpi, int origem, int etiq, MPI\_Comm com,  
MPI\_Request \*pedido)

# Esperando a Mensagem

- Esperando a mensagem chegar:

```
int MPI_Wait(MPI_Request *pedido, MPI_Status  
*estado)
```

- Você também pode testar sem esperar:

```
int MPI_Test(MPI_Request *pedido, int *flag,  
MPI_Status *estado)
```

# Múltipla Espera

- Algumas vezes é desejável esperar por múltiplos “pedidos”:

```
int MPI_Waitall(int cont, MPI_Request  
vetor_de_pedidos[ ], MPI_Status vetor_de_estados[ ])
```

```
int MPI_Waitany(int cont, MPI_Request  
vetor_de_pedidos[ ], int *indice, MPI_Status *estado)
```

```
int MPI_Waitsome(int cont_entra, MPI_Request  
vetor_de_pedidos[ ], int *cont_saida, int  
vetor_de_indices[ ], MPI_Status vetor_de_estados[ ])
```

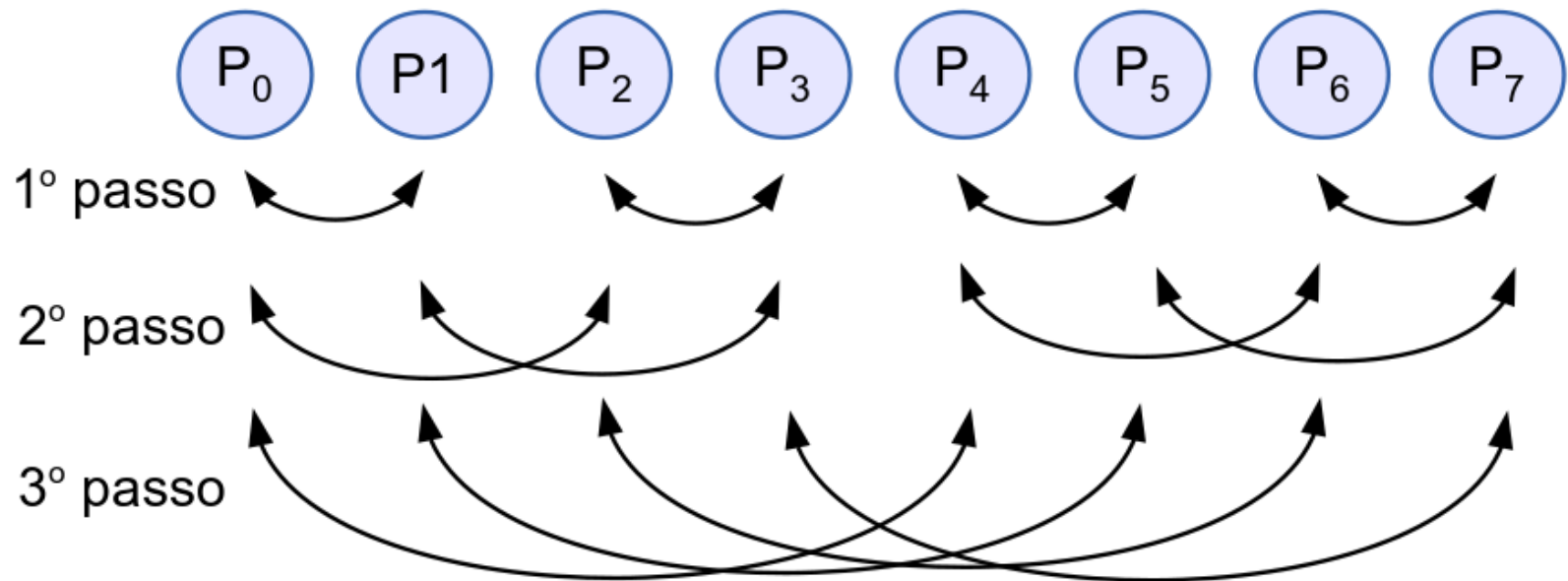
- Existem versões correspondentes de **test** para cada uma das funções acima.

# Exemplo Redução com Difusão

- Os exemplos a seguir apresentam uma implementação da operação de redução com difusão (allreduce) utilizando os diversos modos de comunicação do MPI e as rotinas de envio e recepção não bloqueantes.
- O algoritmo da implementação MPICH do MPI, que utiliza uma técnica de recursive doubling, foi utilizado como base para esses exemplos.
- Por simplificação, vamos assumir que  $P$ , o número de processos, é uma potência de 2 e implementar apenas a operação de redução MPI\_MAX (obter o máximo entre todos os valores).
- O algoritmo requer apenas  $\log_2 P$  passos para a realização do algoritmo de redução e a difusão do resultado para todos os nós,

# Exemplo Redução com Difusão

Processos





# Exemplo Comunicação Não Bloqueante

[https://github.com/gpsilva2003/MPI/mpi\\_isend.c](https://github.com/gpsilva2003/MPI/mpi_isend.c)

# Modos de Comunicação

- O MPI possui quatro modos de comunicação: padrão, bufferizado, pronto e síncrono.
- O modo de comunicação utilizado é determinado pelas rotinas de envio utilizadas, e podem ser utilizadas quaisquer uma das duas rotinas para a recepção: `MPI_Recv` ou `MPI_Irecv`.
- Estes modos existem para assegurar o controle do programador sobre o comportamento do programa, de modo que a evitar a ocorrência de “dealocks” entre os processos participantes da computação, melhorar o desempenho do envio e recepção das mensagens e garantir a sincronização entre os diversos processos.
- O modo de comunicação que utilizamos até agora é o modo padrão, que é bastante prático, mas apresenta alguns inconvenientes, como iremos procurar esclarecer a seguir.

# Modos de Comunicação

- Padrão (standard)
  - O sistema decide se o envio da mensagem vai ser “bufferizada” ou não. Ou seja, a comunicação pode mudar de assíncrona para síncrona, dependendo do tamanho da mensagem, sem que nenhuma notificação seja dada pelo programa. Assim, o seu programa pode ter um comportamento errático, dependendo inclusive do sistema onde for executado.
- Bufferizado (buffered)
  - Neste modo de comunicação, o programador deve fornecer explicitamente um “buffer” para que a mensagem enviada seja armazenada. Este modo garante sempre uma comunicação assíncrona, mas oferece como desvantagem um baixo desempenho no envio das mensagens pelo elevado número de cópias realizadas.


# Modos de Comunicação

- Pronto (ready)
  - A operação de envio só pode ser iniciada após uma operação de recepção correspondente já ter sido iniciada. Ou seja, é necessário o uso de operações de barreira para garantir a ordenação das operações de envio e recepção. Este modo, teoricamente, é o que garante melhor desempenho para o envio e recepção das mensagens.
- Síncrono (synchronous)
  - A operação de envio não se completa até que a operação de recepção correspondente tenha se iniciado. É o modo mais seguro de programação, ou seja, se o seu programa funcionar neste modo irá funcionar no modo padrão sempre. Tem como desvantagem não permitir a sobreposição de computação e comunicação.

# Rotinas de Comunicação Ponto-a-Ponto

Modo	Rotinas	Rotinas
Comunicação	Bloqueantes	Não-Bloqueantes
Síncrono	MPI_Ssend	MPI_ISsend
Pronto	MPI_Rsend	MPI_IRsend
Bufferizado	MPI_Bsend	MPI_IBsend
Padrão	MPI_Send	MPI_Isend
	MPI_Recv	MPI_Irecv

# Modos de Comunicação



```
int MPI_Bsend(void* mensagem, int cont,  
MPI_Datatype tipo_mpi, int dest, int etiq, MPI_Comm  
com)
```

```
int MPI_Ssend(void* mensagem, int cont,  
MPI_Datatype tipo_mpi, int dest, int etiq, MPI_Comm  
com)
```

```
int MPI_Rsend(void* mensagem, int cont,  
MPI_Datatype tipo_mpi, int dest, int etiq, MPI_Comm  
com)
```

# Modos de Comunicação

- Para uma dada rotina de envio de mensagem, diz-se que foi **postada** uma operação de recepção **correspondente**, quando um processo qualquer iniciar uma rotina de recepção com comunicador e etiqueta (“tag”) que satisfaçam ao comunicador e etiqueta utilizados pela rotina de envio.
- Note que é permitido o uso de coringas nas operações de recepção tanto para o remetente (**MPI\_ANY\_SOURCE**) quanto para a etiqueta (**MPI\_ANY\_TAG**), que devem ser considerados nesse caso.
- O **MPI\_Recv** e **MPI\_Irecv** podem receber mensagens enviadas em qualquer modo.

# Modo Bufferizado

- A operação de envio pode ser iniciada havendo ou não uma operação de recepção correspondente iniciada. A operação de envio poderá completar antes de uma recepção **correspondente** ter sido **postada**.
- Existe a necessidade do uso de funções adicionais para alocação e liberação do espaço para armazenamento das mensagens.
- É função do usuário, e não do sistema, gerenciar a alocação dos *buffers*.
- É garantido que as operações de envio e recepção **não** são sincronizadas.



# Modo Bufferizado

```
int MPI_Buffer_attach (void *buffer, int tam_buffer);
```

- Só pode haver um buffer ativo por vez.
- O total de espaço alocado deve ser suficiente para garantir o funcionamento correto do programa.

```
int MPI_Buffer_detach(void *endereco_buffer,  
int * tam_ptr);
```

- Esta rotina retorna um ponteiro para o buffer que está sendo desativado e um ponteiro para o seu tamanho.
- Isso é feito para permitir que uma biblioteca possa substituir e restaurar o buffer.
- O espaço alocado **não** é liberado.

# MPI\_Pack\_Size

```
int MPI_Pack_size(int cont, MPI_Datatype datatype,  
MPI_Comm com, int *tam)
```

```
MPI_Pack_size(20, MPI_INT, com, &tam1);
```

```
MPI_Pack_size(40, MPI_FLOAT, com, &tam2);
```

```
tam_buffer = tam1 + tam2 + 2 * MPI_BSEND_OVERHEAD;
```

- Para efeito de cálculo do espaço necessário para cada envio, a rotina MPI\_Pack\_size deve ser usada.
- A constante MPI\_BSEND\_OVERHEAD especifica o máximo de espaço adicional possível de ser utilizado pela rotina MPI\_Bsend para enviar cada mensagem.
- Essa constante tem um valor específico para cada implementação de MPI.

# Exemplo Comunicação Modo Bufferizado

[https://github.com/gpsilva2003/MPI/mpi\\_bsend.c](https://github.com/gpsilva2003/MPI/mpi_bsend.c)

# Modo Bufferizado

```
char    buffer[MAX_BUF];
int     buffer_size = MAX_BUF;
...
MPI_Buffer_attach(buffer, buffer_size);

for (i = 0; i < p - 1; i++) {
    send_offset = ((meu_ranque - i + p) % p) * blocksize;
    recv_offset = ((meu_ranque - i - 1 + p) % p) * blocksize;
    MPI_Bsend(y + send_offset, blocksize, MPI_FLOAT, sucessor, 0,
com);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT, predecessor, 0,
com, &status);
}
MPI_Buffer_detach(&buffer, &buffer_size);
```

# Modo Síncrono

- A rotina de envio pode ser iniciada havendo ou não uma rotina de recepção correspondente postada.
- Contudo, o envio irá **completar** com sucesso apenas quando uma recepção **correspondente** tiver sido **postada** e a recepção da mensagem enviada pela rotina de envio síncrona for iniciada pela operação de recepção.
- Este modo não requer o uso de bufferização do sistema.
- Pode-se assegurar que o nosso programa está **seguro** se executar corretamente utilizando apenas rotinas de envio no modo síncrono.

# Exemplo Comunicação Modo Síncrono

[https://github.com/gpsilva2003/MPI/mpi\\_ssend.c](https://github.com/gpsilva2003/MPI/mpi_ssend.c)

# Modo Síncrono

```
for (i = 0; i < p - 1; i++) {
```

```
...
```

```
    if ((meu_ranke % 2) == 0){ /* Ranque ímpar envia primeiro */
        MPI_Ssend(y + send_offset, blocksize, MPI_FLOAT, sucessor, 0,
com);
        MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
predecessor, 0, com, &status);
    } else { /* Ranque par recebe primeiro */
        MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT, predecessor, 0,
com, &status);
        MPI_Ssend(y + send_offset, blocksize, MPI_FLOAT, sucessor, 0,
com);
    }
}
```

# Modo Pronto

- O envio pode ser iniciado **apenas** se houver uma rotina de recepção correspondente já iniciada. Caso isto não ocorra, o programa terminará com **erro**.
- Embora seja esperado, não é garantido que a implementação das rotinas em modo pronto seja mais eficiente que a de modo padrão.
- É necessário o uso de funções de sincronização (p.ex. barreiras) para garantir que a recepção em um processo é **postada** antes do envio pelo outro.
- Este é o modo mais **difícil** de programar e só deve ser usado quando o desempenho for importante.



# Exemplo Comunicação Modo Pronto

[https://github.com/gpsilva2003/MPI/mpi\\_rsend.c](https://github.com/gpsilva2003/MPI/mpi_rsend.c)

# Modo Pronto

```
MPI_Request request [p-1];
```

```
for (i = 0; i < p - 1; i++) {  
    recv_offset = ((meu_ranque - i - 1 + p) % p)*blocksize;  
    MPI_Irecv(y + recv_offset, blocksize, MPI_FLOAT, predecessor, i, com,  
    &(request[i]));  
}  
MPI_Barrier(com);  
  
for (i = 0; i < p - 1; i++) {  
    send_offset = ((meu_ranque - i + p) % p)*blocksize;  
    MPI_Rsend(y + send_offset, blocksize, MPI_FLOAT, sucessor, i, com);  
    MPI_Wait(&(request[i]), &status);  
}
```

# Modo Padrão

- Neste modo o MPI decide se as mensagens enviadas serão *bufferizadas* ou enviadas em modo síncrono.
- Uma rotina de envio no modo padrão pode ser iniciada havendo ou não uma rotina de recepção correspondente postada.
- Não se pode assumir que a operação de envio irá terminar antes ou depois da recepção correspondente ser iniciada.
- Como consequência, pode ser que se programa funcione bem em um sistema e em outro não, caso o programa esteja programado de modo **não-seguro**.

# Exemplo Comunicação Modo Padrão

[https://github.com/gpsilva2003/MPI/mpi\\_padrao.c](https://github.com/gpsilva2003/MPI/mpi_padrao.c)

# Modo Padrão

```
sucessor = (meu_ranque + 1) % p;  
predecessor = (meu_ranque - 1 + p) % p;
```

```
for (i = 0; i < p - 1; i++) {  
    send_offset = ((meu_ranque - i + p) % p) * blocksize;  
    recv_offset = ((meu_ranque - i - 1 + p) % p) * blocksize;  
    MPI_Send(y + send_offset, blocksize, MPI_FLOAT, sucessor, 0, com);  
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT, predecessor, 0,  
com, &status);  
}
```

# Fontes de Deadlock

- Envie uma mensagem grande do processo 0 para o processo 1

Se o espaço de armazenamento no destino for insuficiente, a rotina de envio deve esperar até que o usuário providencie espaço de memória suficiente (através da chamada de uma rotina de recepção).

- O que acontece com este código?

**Processo 0**

**Processo 1**

**MPI\_Send(1)**

**MPI\_Send(0)**

**MPI\_Recv(1)**

**MPI\_Recv(0)**

- Isto é chamado de “não-seguro” porque depende da disponibilidade dos *buffers* de sistema.

# Algumas soluções

- Ordenar as operações de envio e recepção adequadamente:

**Processo 0**

**Processo 1**

---

**MPI\_Send(1)**

**MPI\_Recv(0)**

**MPI\_Recv(1)**

**MPI\_Send(0)**

- Fornecer um buffer de recepção ao mesmo tempo que envia a mensagem:

# Mais Soluções

- O usuário fornece explicitamente um *buffer* para envio:

**Processo 0**

**Processo 1**

MPI\_BSend(1)

MPI\_BSend(0)

MPI\_Recv(1)

MPI\_Recv(0)

- Uso de operações não-bloqueantes:

**Processo 0**

**Processo 1**

MPI\_Isend(1)

MPI\_Isend(0)

MPI\_Irecv(1)

MPI\_Irecv(0)

MPI\_Waitall

MPI\_Waitall



# MPI\_Sendrecv

- Permite envio e recepção simultâneos.
- Fornece um buffer de recepção ao mesmo tempo que envia a mensagem.
- Os tipos de dados de envio e recepção podem ser diferentes.
- Pode-se usar MPI\_Sendrecv com um MPI\_Recv ou MPI\_Send comuns (ou MPI\_Irecv, MPI\_Ssend, etc.)

**Processo 0**

**Processo 1**

---

**MPI\_Sendrecv(1)**

**MPI\_Sendrecv(0)**

# MPI\_Sendrecv

```
int MPI_Sendrecv (const void *vet_envia, int cont_envia,  
MPI_Datatype tipo_envia, int dest, int etiq_envia, void  
*vet_recebe, int cont_recebe, MPI_Datatype tipo_recebe,  
int origem, int etiq_recebe, MPI_Comm com, MPI_Status  
*estado)
```

# Exemplo MPI\_Sendrecv

[https://github.com/gpsilva2003/MPI/mpi\\_sendrecv.c](https://github.com/gpsilva2003/MPI/mpi_sendrecv.c)

# Estudo de Caso - Primos

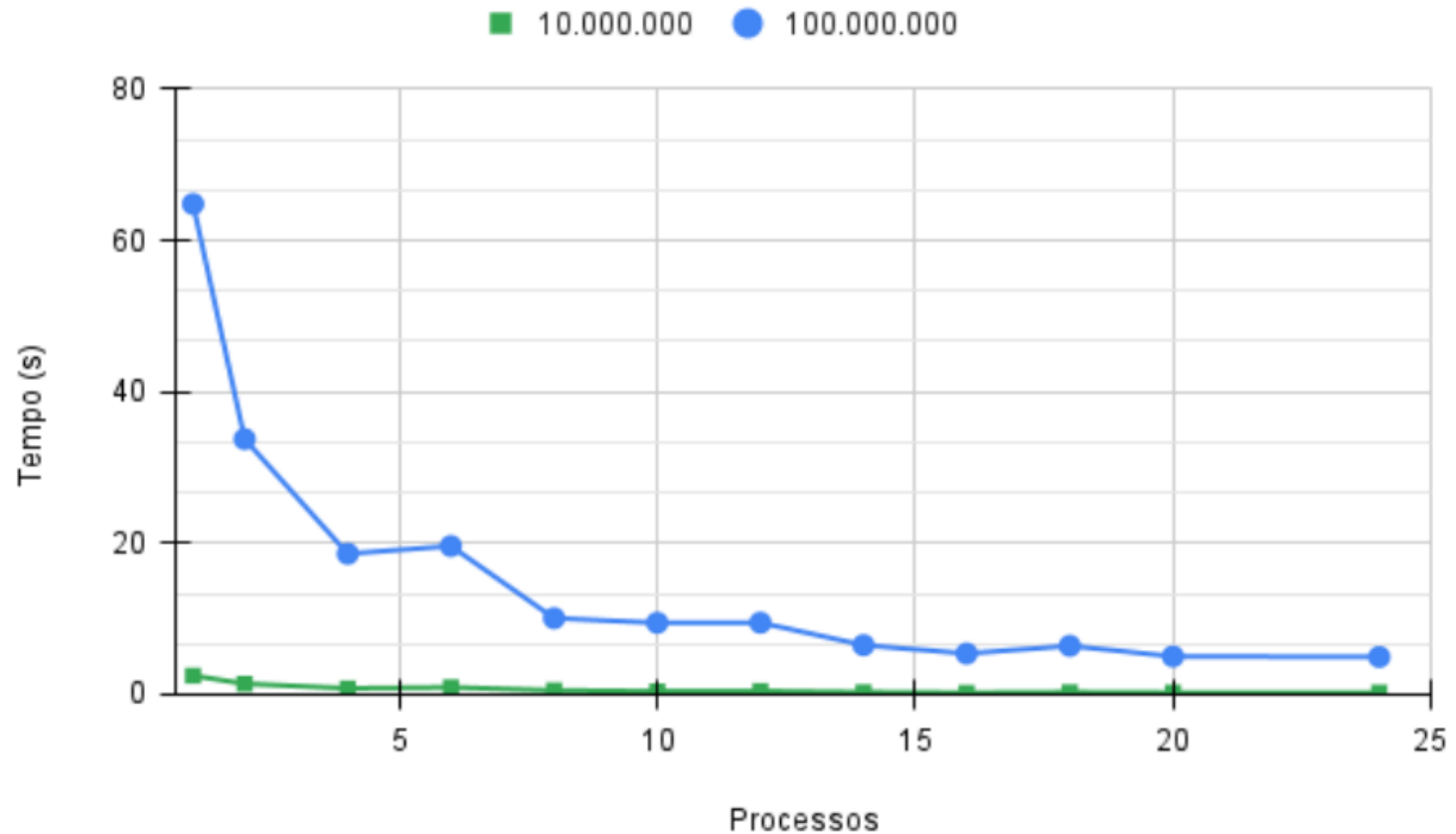
- Nesta seção, nosso estudo de caso será um programa para calcular a quantidade de números primos entre 0 e um determinado valor inteiro N.
- Ele basicamente verifica se N é divisível por algum número ímpar entre 0 e a raiz quadrada de N, sendo que os números pares são descartados de imediato.

```
$ mpicc -O3 -o mpi_primos mpi_primos.c -lm  
$ mpirun -np 4 ./mpi_primos 1000000000
```

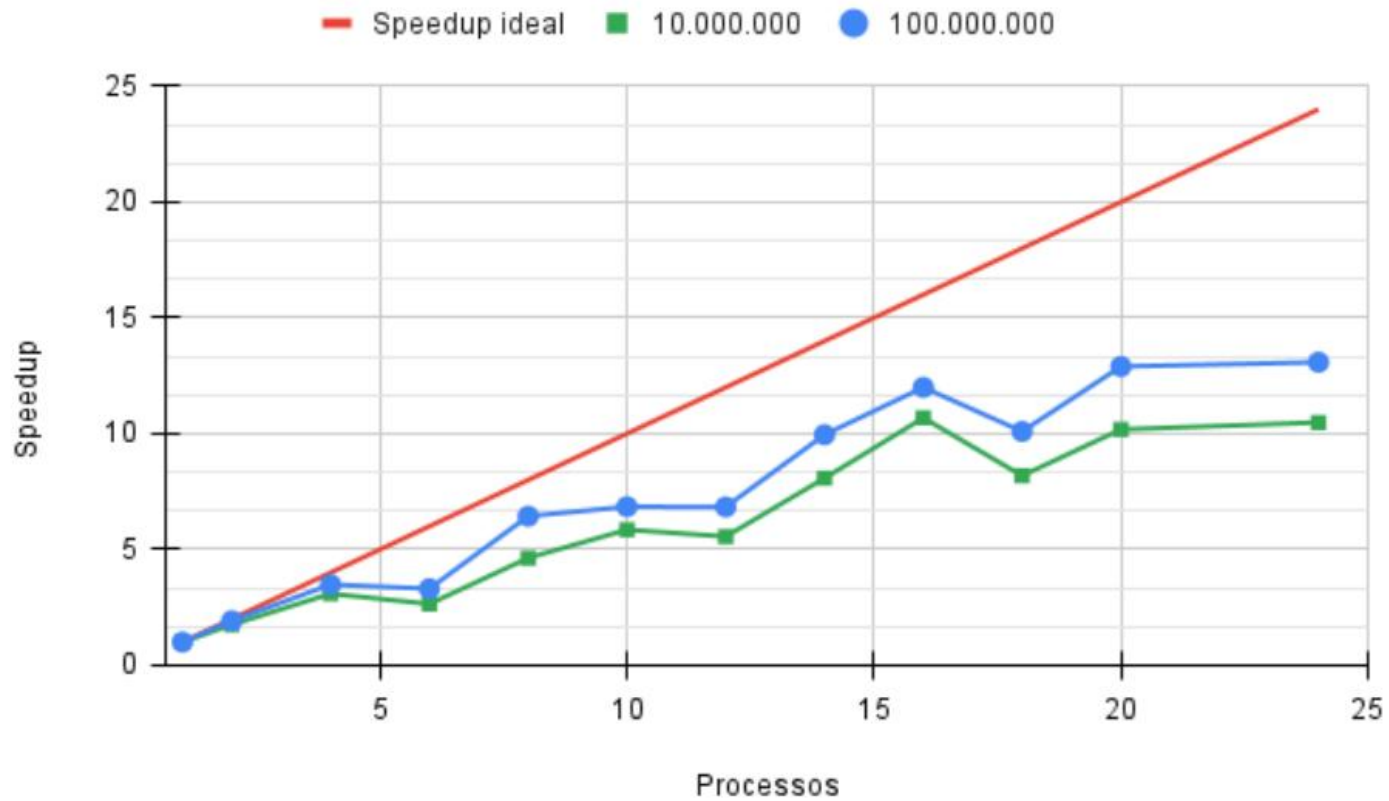
# Exemplo Primos – Naive

[https://github.com/gpsilva2003/MPI/mpi\\_primos.c](https://github.com/gpsilva2003/MPI/mpi_primos.c)

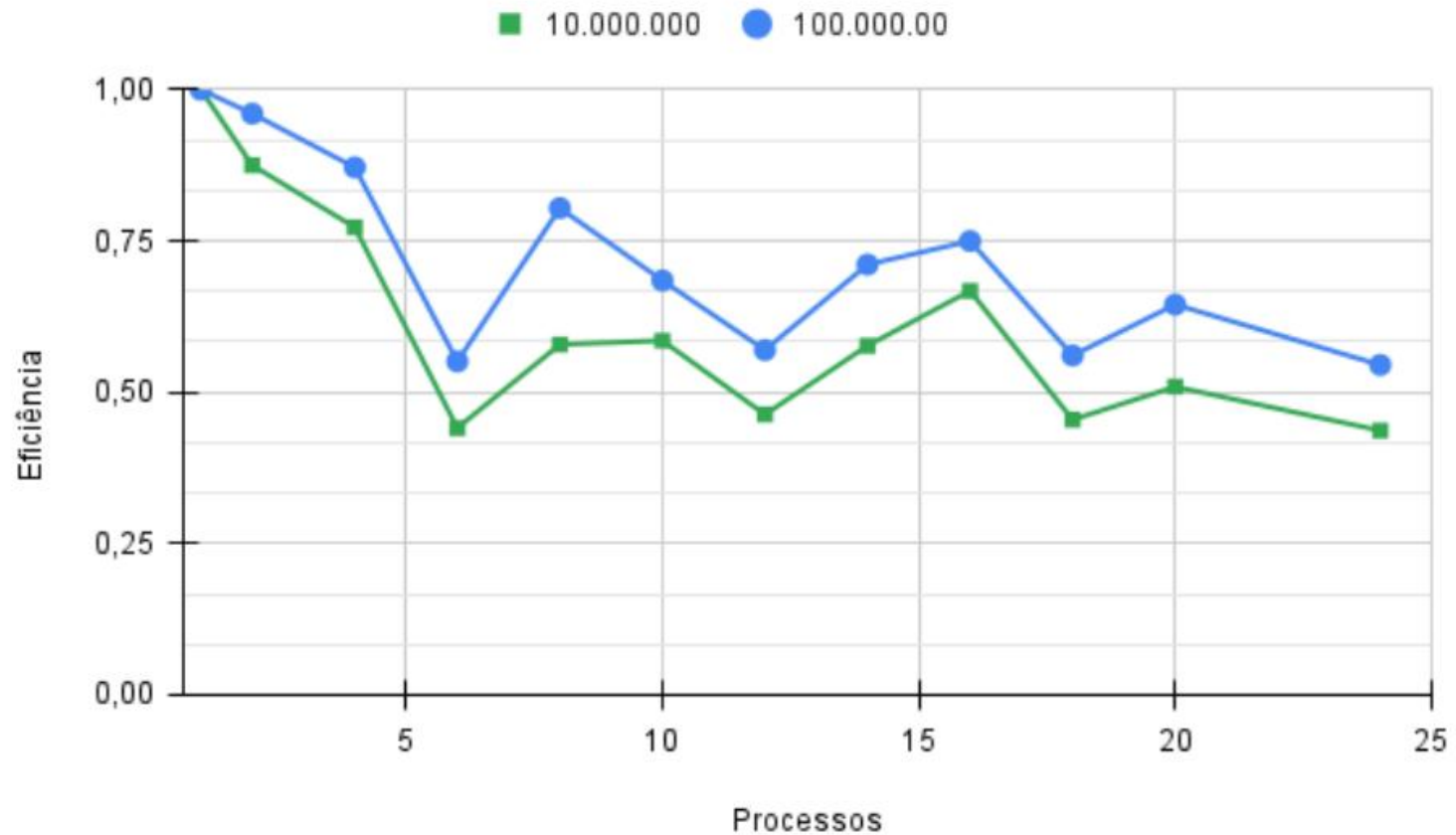
# Primos – Naive – Tempo de Execução



# Primos – Naive - Speedup



# Primos – Naive – Eficiência



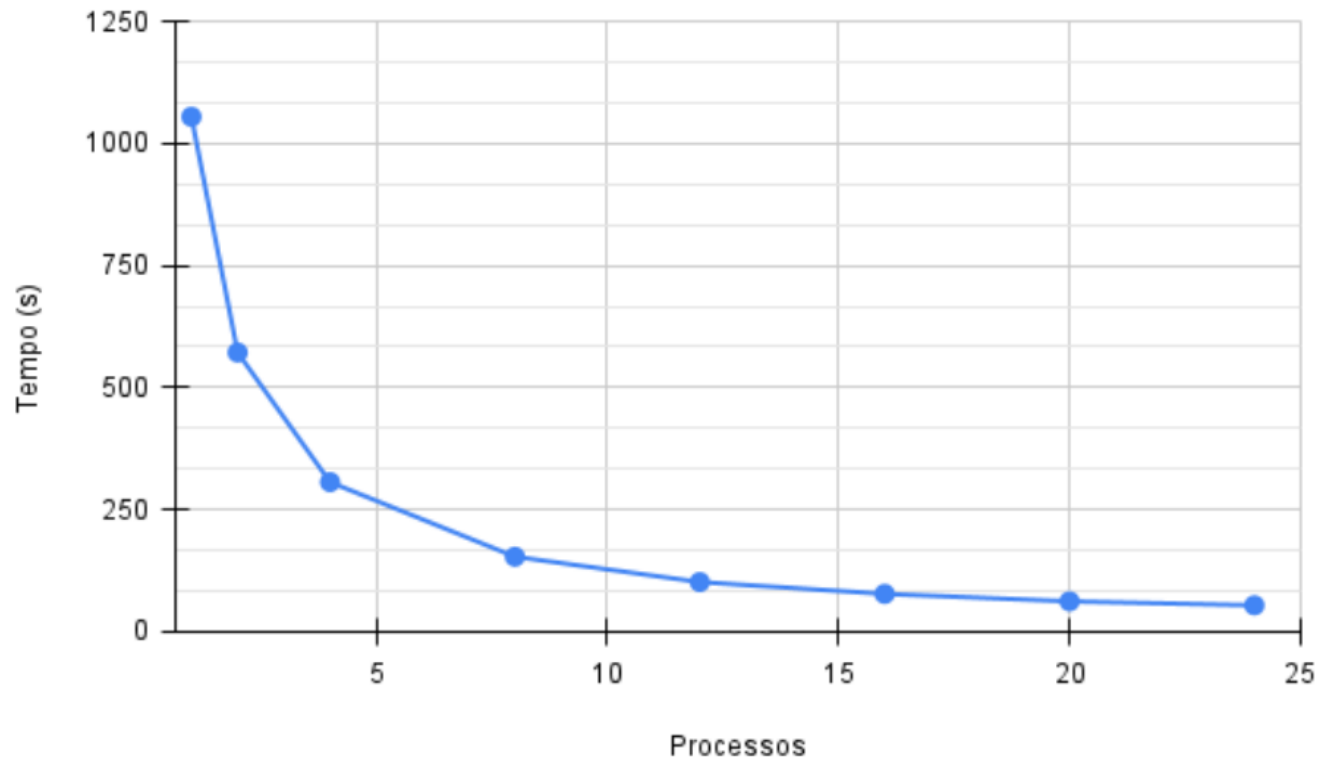


# Exemplo Primos – Saco de Tarefas

[https://github.com/gpsilva2003/MPI/mpi\\_primosbag.c](https://github.com/gpsilva2003/MPI/mpi_primosbag.c)

# Primos - Saco de Tarefas

## Tempo de Execução



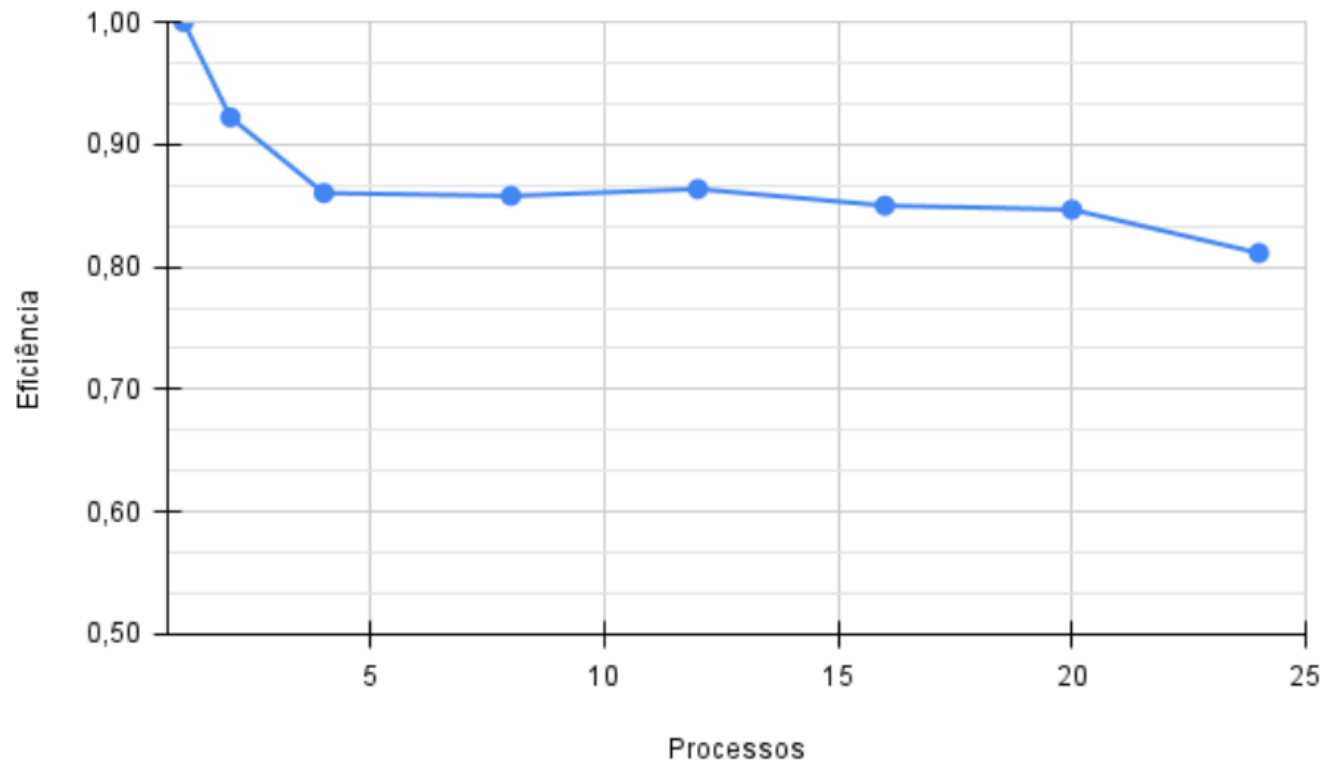
# Primos - Saco de Tarefas

## Speedup



# Primos - Saco de Tarefas

## Eficiência





# OpenMPI

# Instalando o OpenMPI

- Basicamente:

Instalar o OpenMPI do repositório:

```
% sudo apt-get install openmpi-bin
```

```
% sudo apt-get install openmpi-doc
```

Ou

```
% sudo yum install openmpi
```

# Usando o OpenMPI

- Assumimos que o programa será executado em uma rede heterogênea de computadores com UNIX.
- Os programas executáveis, bibliotecas e arquivos de cabeçalho supõem-se que estejam instalados em um diretório público nas máquinas em que você está executando e compilando o seu programa.
- Nos exemplos a seguir assumimos que os arquivos do OpenMPI são armazenados nos seguintes diretórios:

Executáveis: `/usr/bin`

Bibliotecas: `/usr/lib`

Arquivos de Cabeçalho: `/usr/include`

# Usando o OpenMPI

- As seguintes variáveis de ambiente devem estar configuradas:

PATH:            /usr/openmpi/bin

LD\_LIBRARY\_PATH: /usr/openmpi/lib



# Usando o OpenMPI

- Para compilar um arquivo fonte `prog.c`, digite:  
`% mpicc -o prog prog.c`
- Para executar o programa com, digamos, 4 processos, você deve copiar o executável para o seu diretório \$HOME em cada máquina e digitar:  
`% mpirun -n 4 prog`
- A cópia é desnecessária se o diretório estiver montado remotamente (NFS).

# Usando o OpenMPI

- O comando **mpiexec** permite opções mais elaboradas:

```
% mpiexec -n 1 -host paraty : -n 19 slave
```

- Dispara o processo com *ranque* 0 na máquina paraty e outros 19 divididos entre as demais máquinas.

# Usando o OpenMPI

Para executar através de múltiplas máquinas:

```
$ mpirun --hostfile my_hostfile -np 4  
    my_parallel_application
```

Onde my\_hostfile é um arquivo contendo o nome ou IP das máquinas onde deseja executar a aplicação

- Para saber mais opções digite:

```
% mpiexec --help
```

# Referências

- 1) Gabriel P. Silva, Calebe Bianchini e Evaldo B. Costa  
“Programação Paralela – Um Curso Introdutório” Editora Casa do Código, 2022
- 2) Neil MacDonald et alii, Writing Message Passing Programs with MPI, Edinburgh Parallel Computer Centre
- 3) Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 2nd ed., Englewood Cliffs, NJ, Prentice--Hall, 1988.
- 4) Peter S. Pacheco, Parallel Programming with MPI, Morgan Kaufman Pub, 1997.
- 5) Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Version 3.1, 2015 Acesso em <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>

# Obrigado!

Gabriel P. Silva

[gabriel@ic.ufrj.br](mailto:gabriel@ic.ufrj.br)

<http://github.com/gpsilva2003>