


Escola Supercomputador Santos Dumont
2024

Programação com MPI

Gabriel P. Silva
(gabriel@ic.ufrj.br)

Ementa

- 
1. Modelos de Programação
 2. Conceitos Básicos
 3. Apresentação do MPI
 4. Comunicadores e Funções Básicas
 5. Comunicação Ponto a Ponto
 6. Comunicação Coletiva
 7. Tipos Derivados de Dados

Recursos

[https://www.researchgate.net/
profile/Gabriel-Silva-130](https://www.researchgate.net/profile/Gabriel-Silva-130)

<https://programacao-paralela-e-distribuida.github.io/>



[https://learn.microsoft.com/en-us/
windows/wsl/install](https://learn.microsoft.com/en-us/windows/wsl/install)

<https://www.cygwin.com/>

Referências

- 1) Gabriel P. Silva, Calebe Bianchini e Evaldo B. Costa
“Programação Paralela – Um Curso Introdutório”, Editora Casa do Código, 2022.
- 2) 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>
- 3) Peter S. Pacheco, “Parallel Programming with MPI”, Morgan Kaufman Pub, 1997.
- 4) Brian W. Kernighan and Dennis M. Ritchie, “The C Programming Language”, 2nd ed., Englewood Cliffs, NJ, Prentice--Hall, 1988.
- 5) Michael Quinn, “Parallel Programming in C with MPI and OpenMP”, McGraw-Hill Science/Engineering/Math, 2003.

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

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

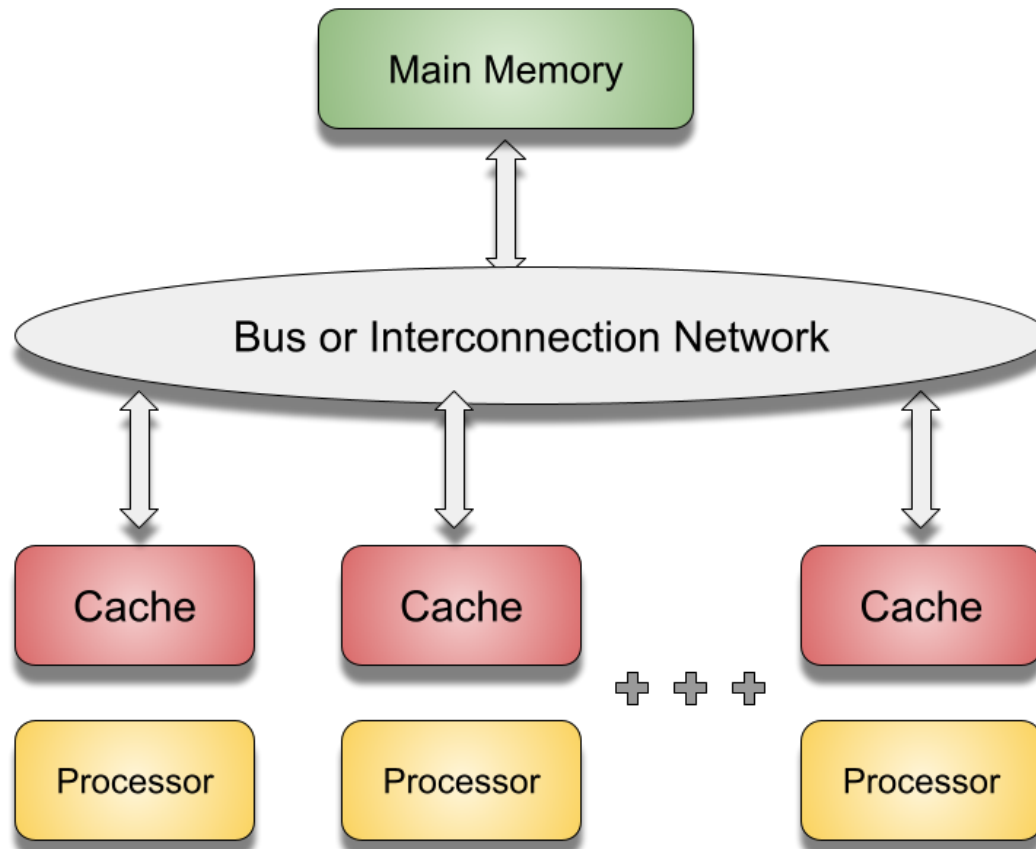
Modelos de Programação



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 do 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.

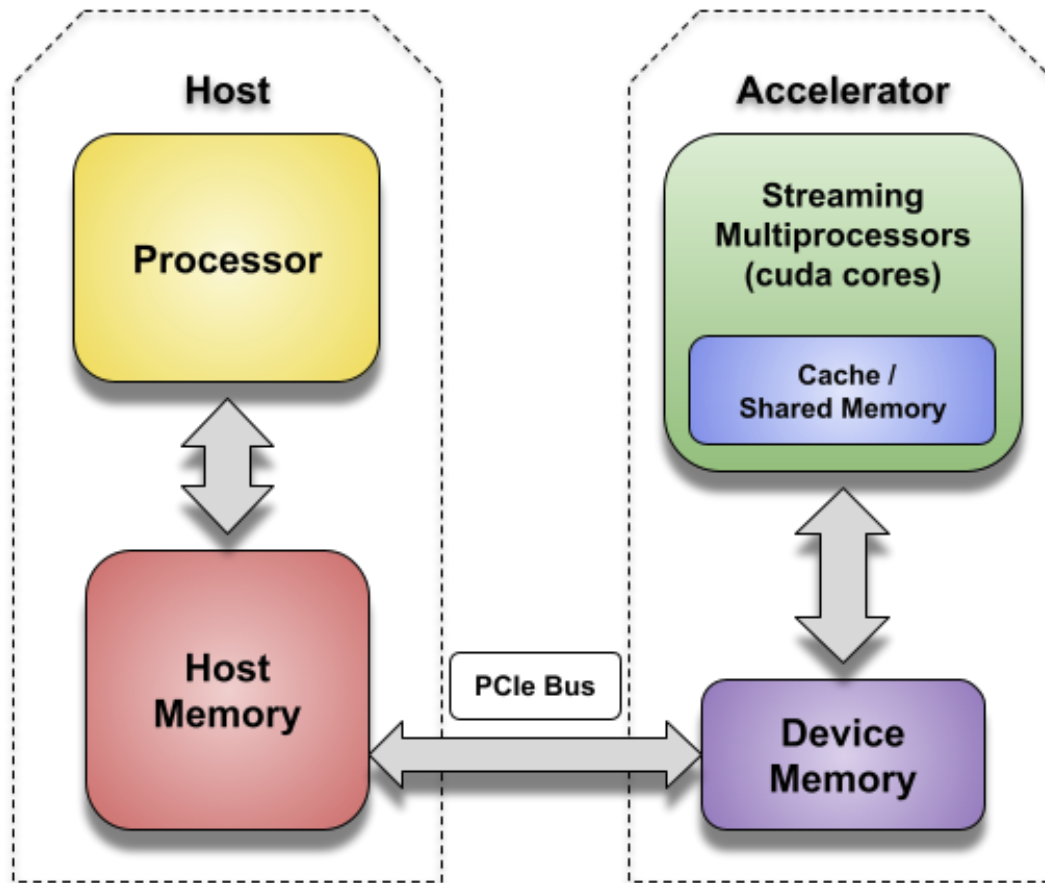
Memória Compartilhada



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.
- Exemplos de aceleradores:
 - <https://www.amd.com/en/graphics/instinct-server-accelerators>
 - <https://www.nvidia.com/en-us/data-center/data-center-gpus/>

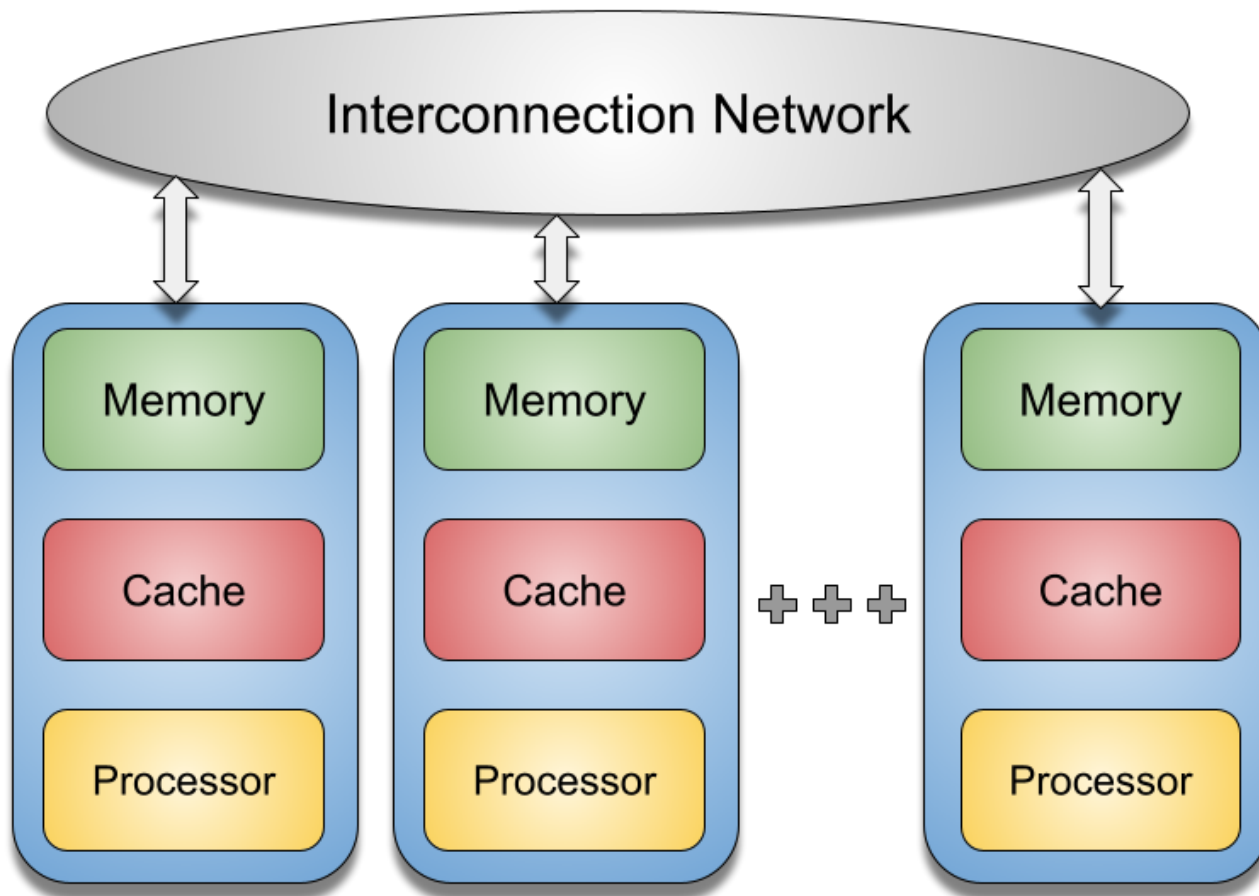
Aceleradores



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



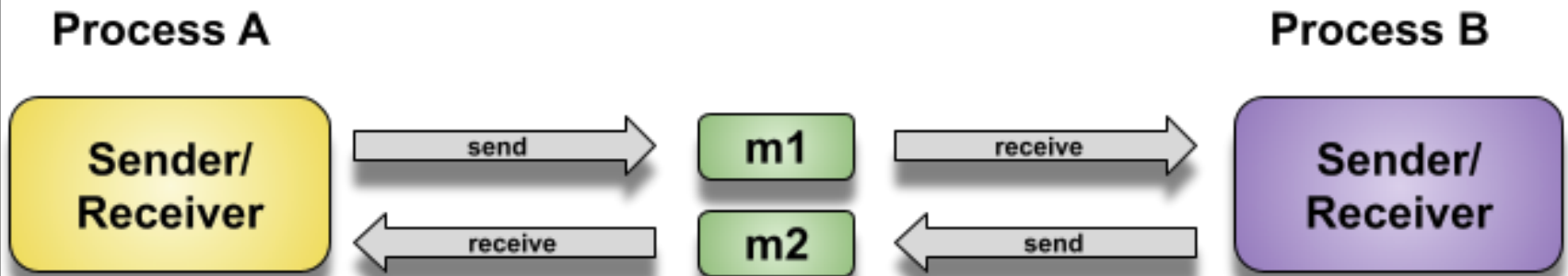
Conceitos Básicos



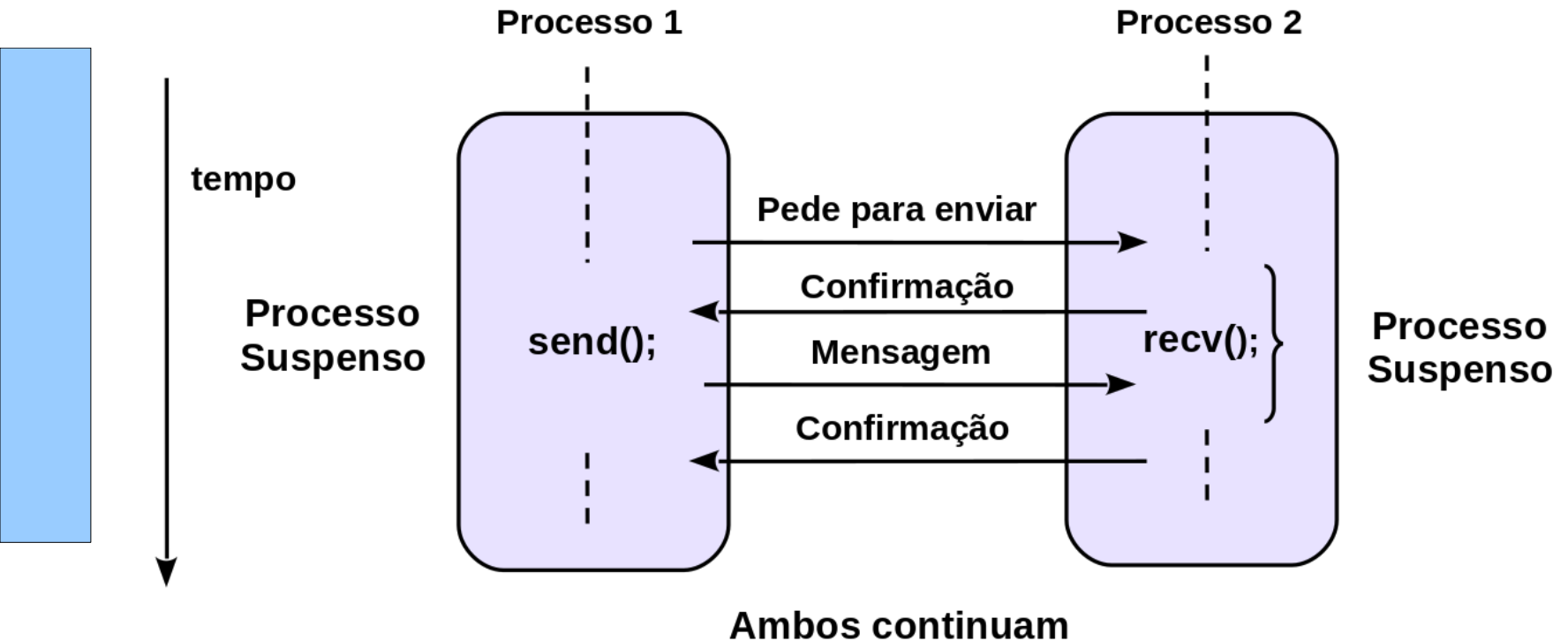
Troca de Mensagens

- 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 que 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**.

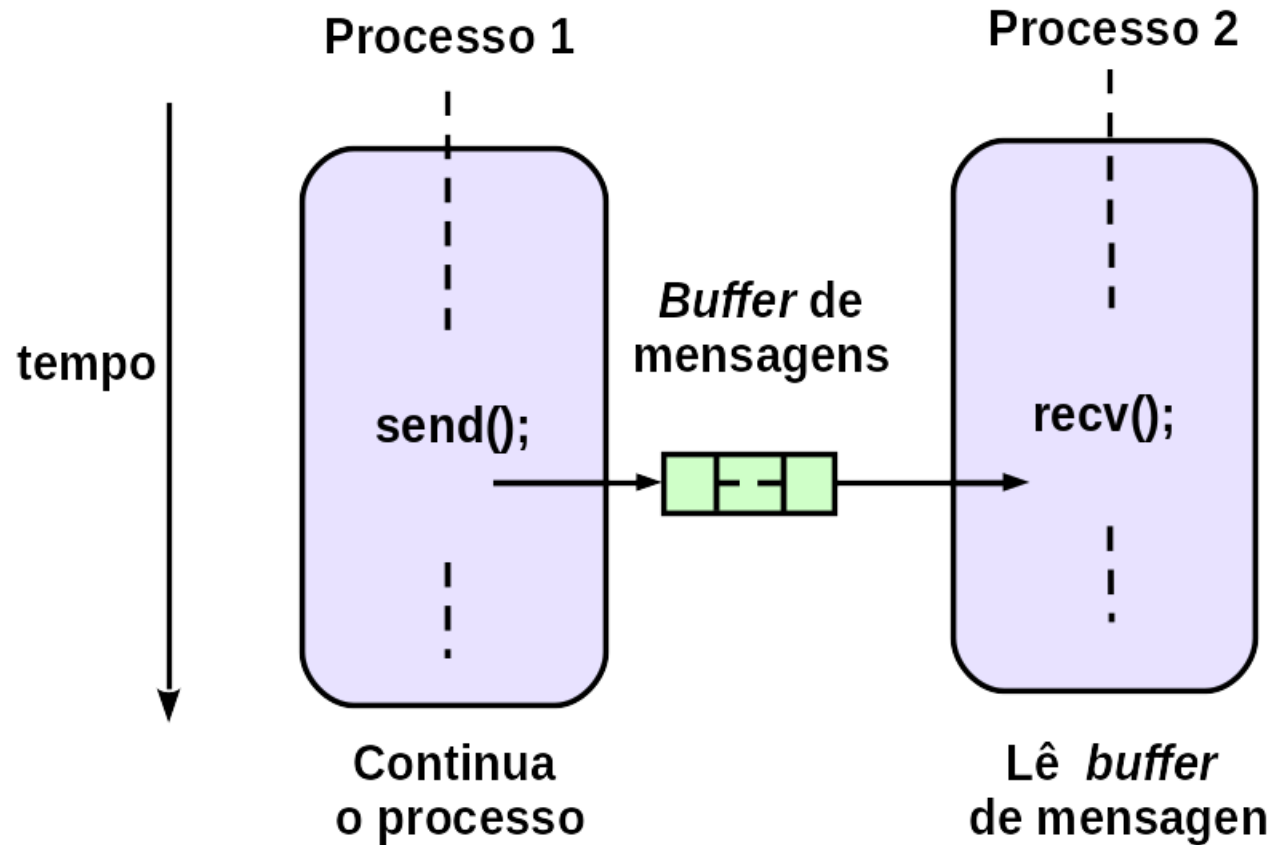
Troca de Mensagens



Comunicação Síncrona



Comunicação Assíncrona



Comunicação 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.

Medidas de Desempenho

- Speed-up (Aceleração):

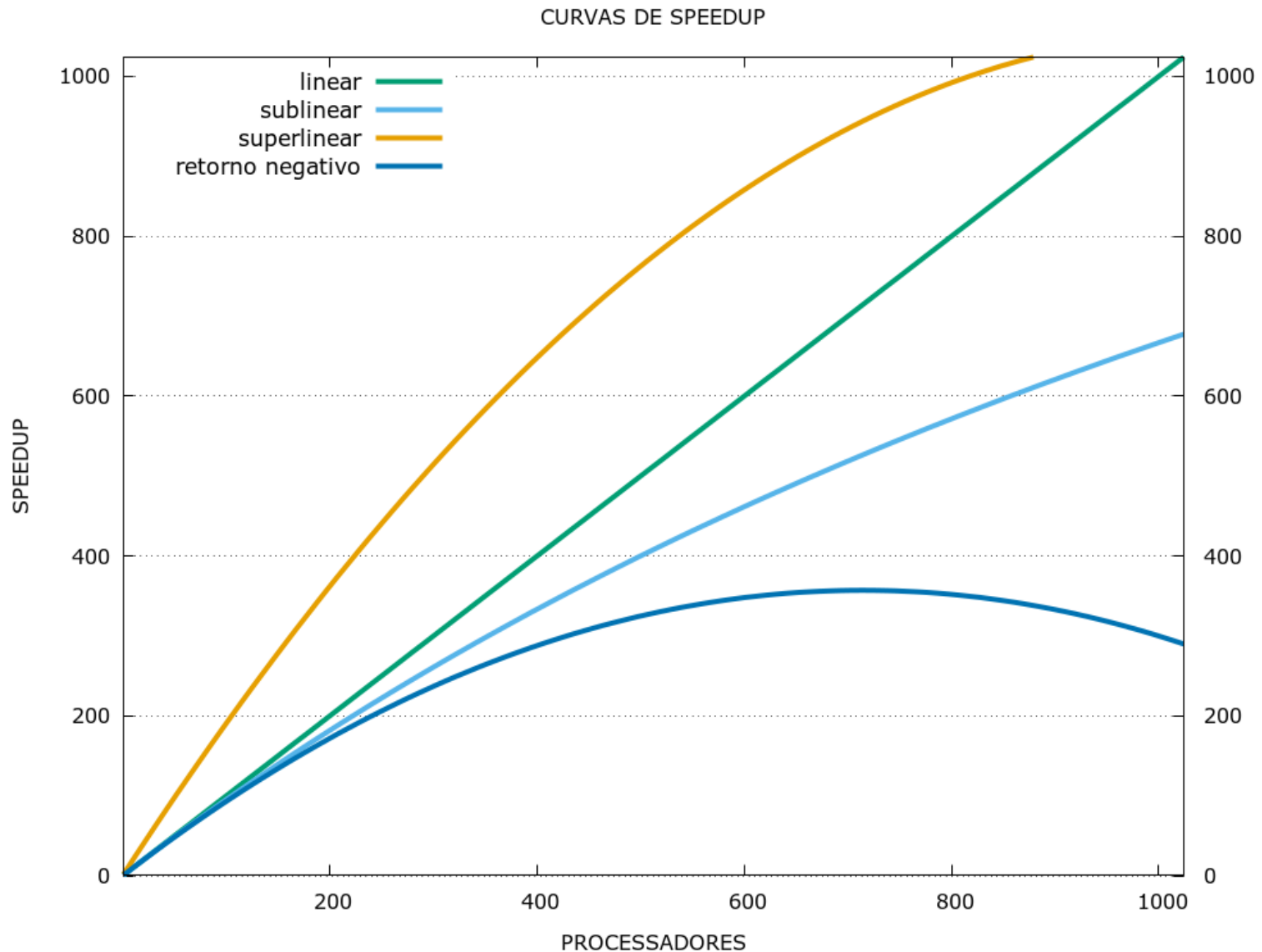
Mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador e o tempo gasto na execução com n processadores:

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

- 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. Pode ser feita a partir do zero ou de uma versão sequencial do programa.
- Deve-se dividir o problema original em conjunto de tarefas que vão executar em paralelo e forma cooperativa para a solução do problema.
- O particionamento das tarefas depende do modelo de programação adotado (troca de mensagens, memória compartilhada ou acelerador) e dos custos de comunicação e sincronização do modelo no 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 que se comunicam alocados ao mesmo processador ou em processadores “próximos” entre si.
- Essa alocação pode ser feita de forma automática pelo ambiente de execução ou manualmente pelo programador.

Criação de um Programa Paralelo

- 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 isso é uma indicação de que você certamente não realizou uma boa distribuição dessas tarefas.

Apresentação do MPI



Histórico

- 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 PICL.

Histórico

- MPI-1.1: primeira versão funcional lançada em 1998.
- MPI-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: essa versão foi lançada em 2021.
- MPI 4.1: em 2023 a versão mais recente do padrão foi liberada.
- Todas as versões estão disponíveis em <https://www.mpi-forum.org/docs/>

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.

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 ou 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.

Distribuições

- OpenMPI
 - <https://www.open-mpi.org/software/ompi/v5.0/>
- MPICH
 - <https://www.mpich.org>
- MVAPICH
 - <https://mvapich.cse.ohio-state.edu>
- Proprietárias
 - **IBM:** <https://www.ibm.com/products/spectrum-mpi>
 - **Microsoft:** <https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi>
 - **Intel:** <https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html>
 - **Cray:** <https://cpe.ext.hpe.com/docs/mpt/mpich/index.html>

Instalação

- WSL e Ubuntu

- \$ sudo apt install openmpi-bin

- \$ sudo apt install libopenmpi-dev

- Fedora

- \$ sudo dnf install openmpi openmpi-devel

- Instalação Manual

- Baixar o código fonte e seguir as instruções de <http://www.open-mpi.org/software/ompi>

- Obs: você vai precisar de um compilador C/C++ instalado também.

Execução

- As seguintes variáveis de ambiente devem estar configuradas:

PATH: /usr/openmpi/bin

LD_LIBRARY_PATH: /usr/openmpi/lib

- 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).

Comunicadores e Funções Básicas

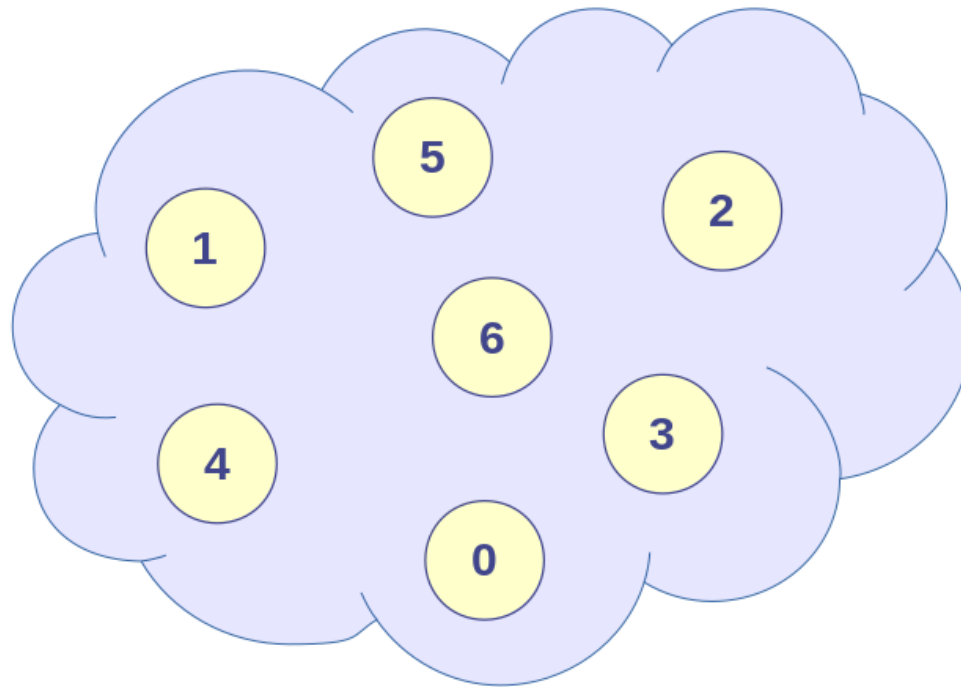


Comunicadores

- Um *comunicador* define 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.
 - Grupo: conjunto ordenado de processos. A ordem de um processo no grupo é chamada de **ranque**.
 - Contexto: um rótulo definido pelo sistema.
 - 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

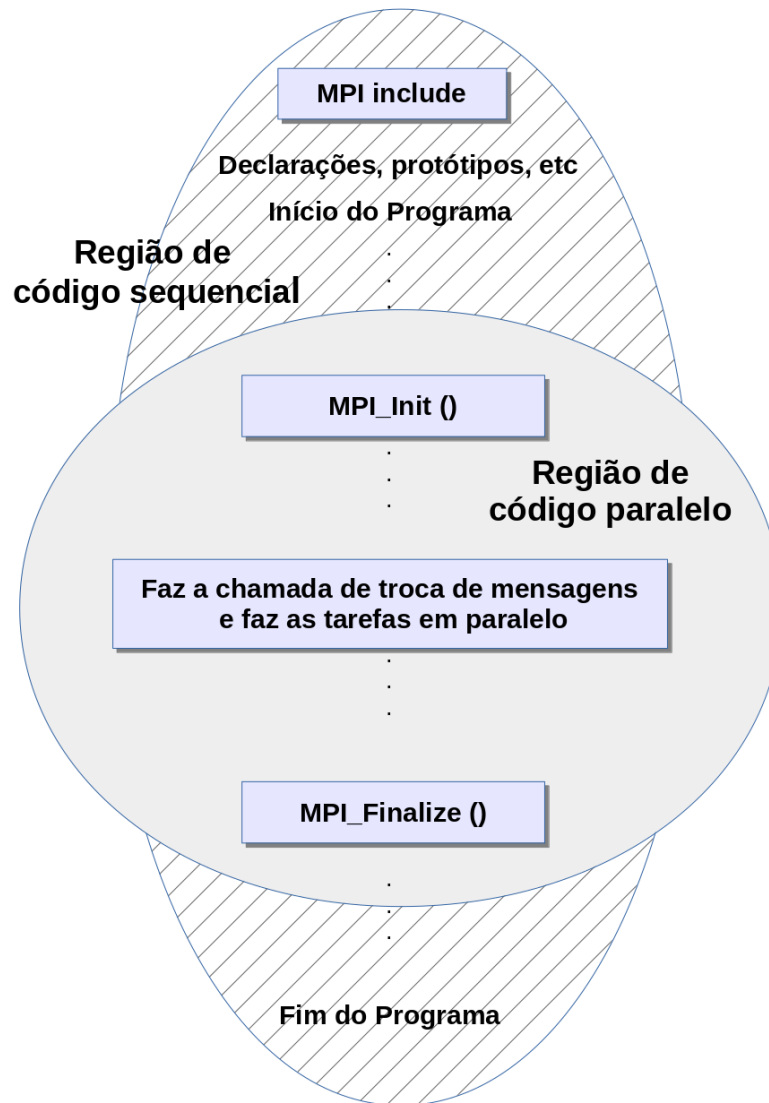


Iniciando o MPI

- 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*/  
...  
exit(0)
```


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**, que é essencialmente 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.

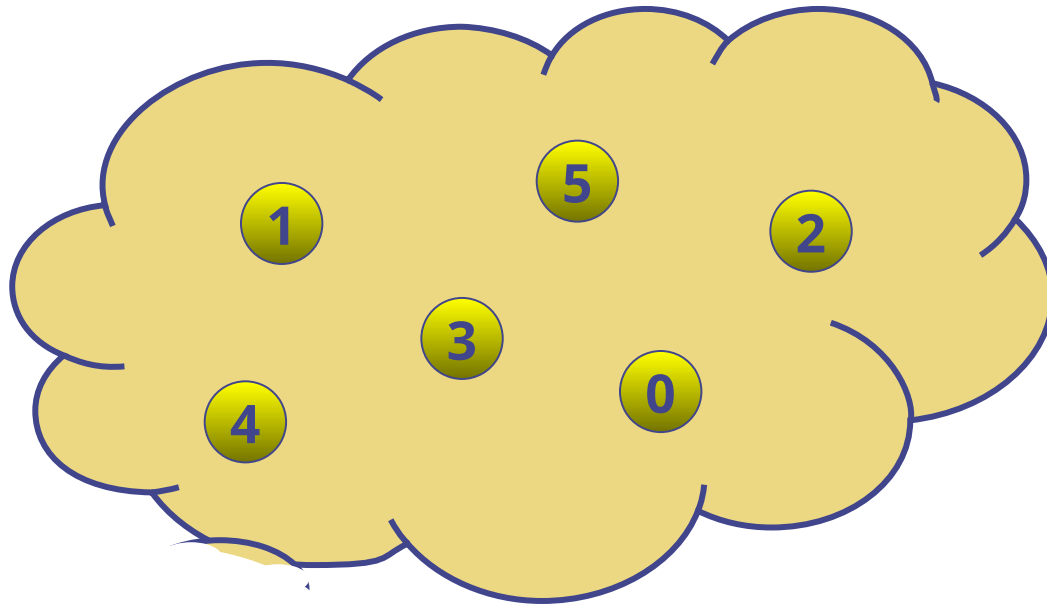
Quantos processos existem?

- 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.

Exemplo do Uso de Funções

https://github.com/gpsilva2003/MPI/mpi_funcoes.c



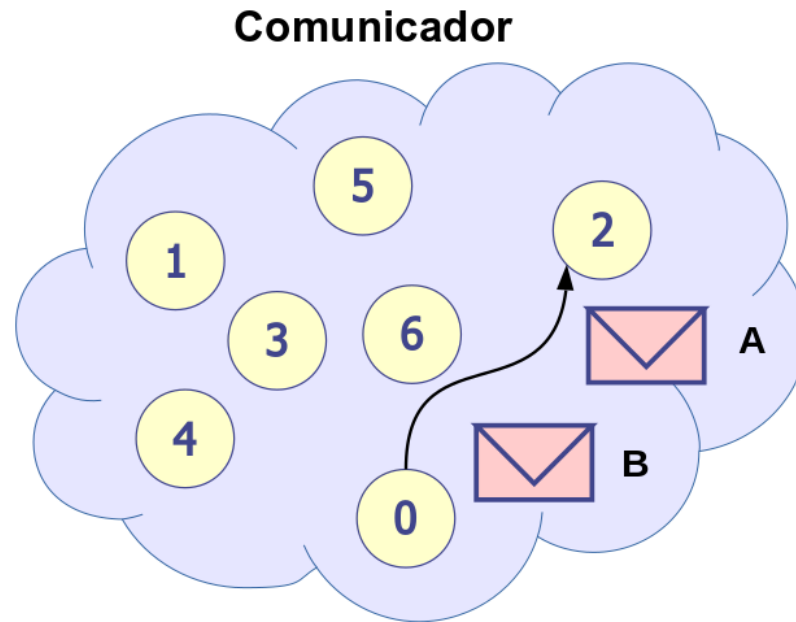


Comunicação Ponto a Ponto

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

- 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 ao 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.

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

Comunicação Ponto a Ponto

- 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.
- Se a etiqueta da recepção era **MPI_ANY_TAG**, então o *estado* irá conter a etiqueta da mensagem recebida.

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

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

Estudo de Caso – Integral definida método trapézio

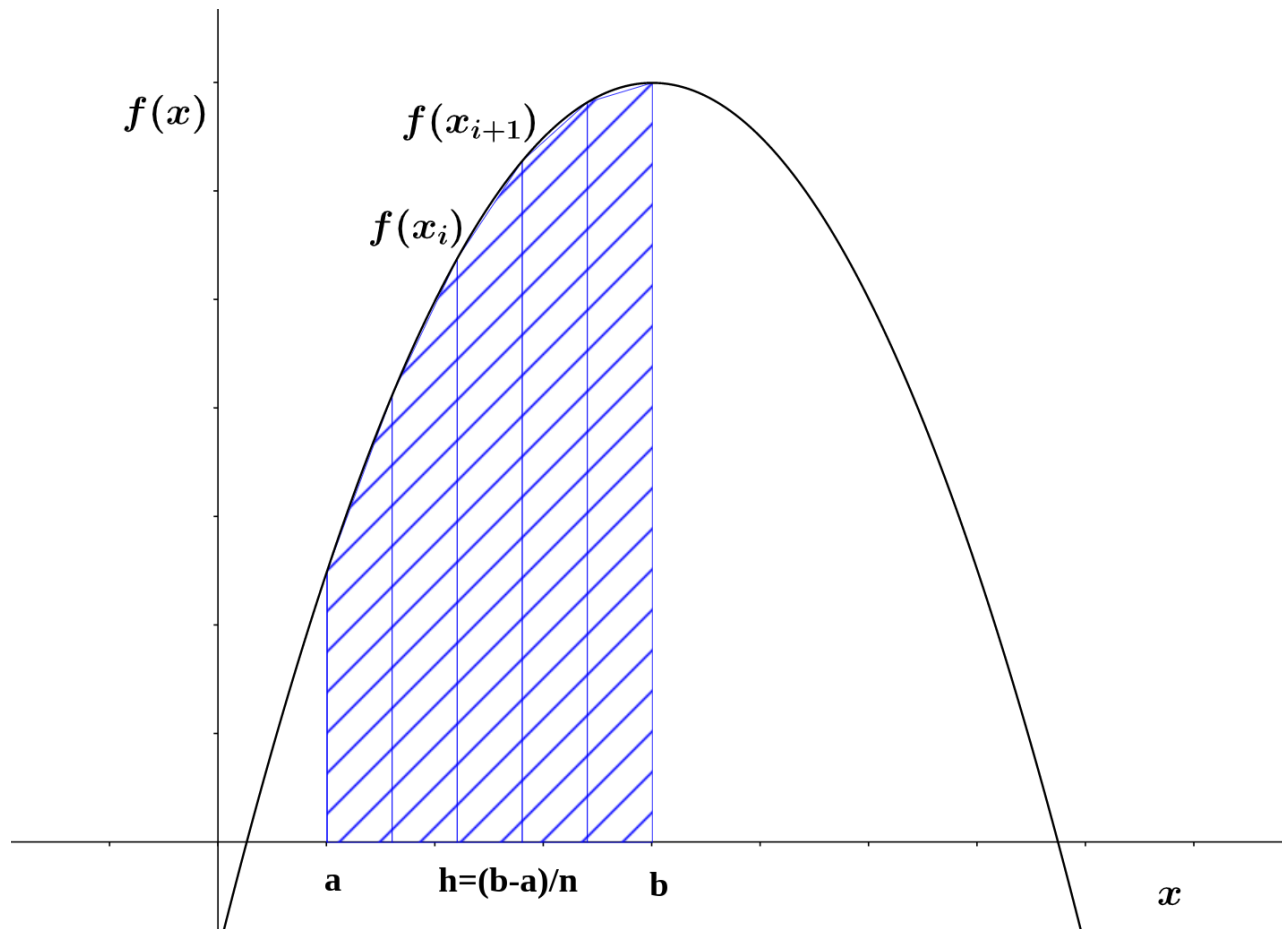
- Vamos lembrar que o método do trapézio estima o valor de $f(x)$ dividindo o intervalo $[a; b]$ em n segmentos iguais e calculando a seguinte soma:

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

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

- 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

Medidas de Desempenho

- Speed-up (Aceleração):

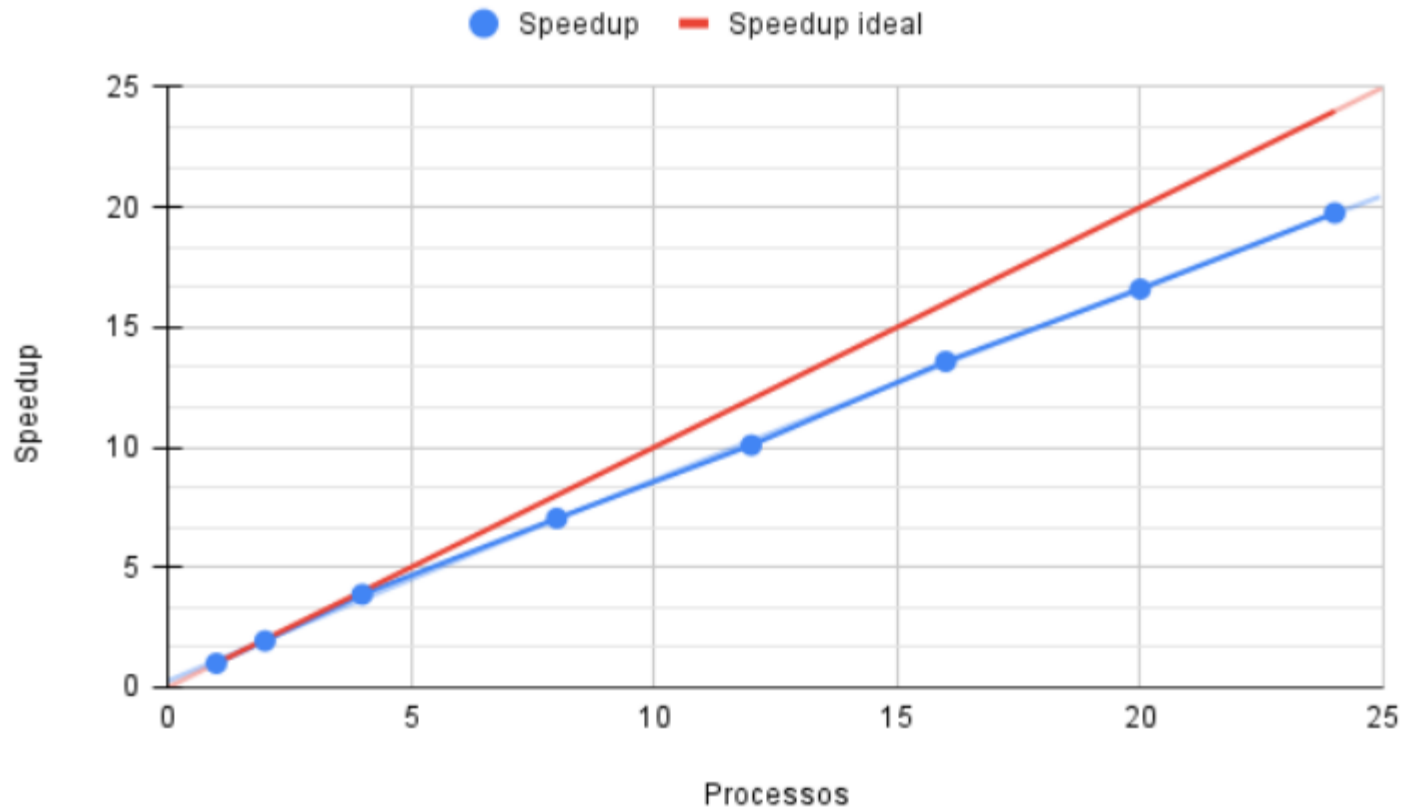
Mede a razão entre o tempo gasto para execução de um algoritmo ou aplicação em um único processador e o tempo gasto na execução com n processadores:

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

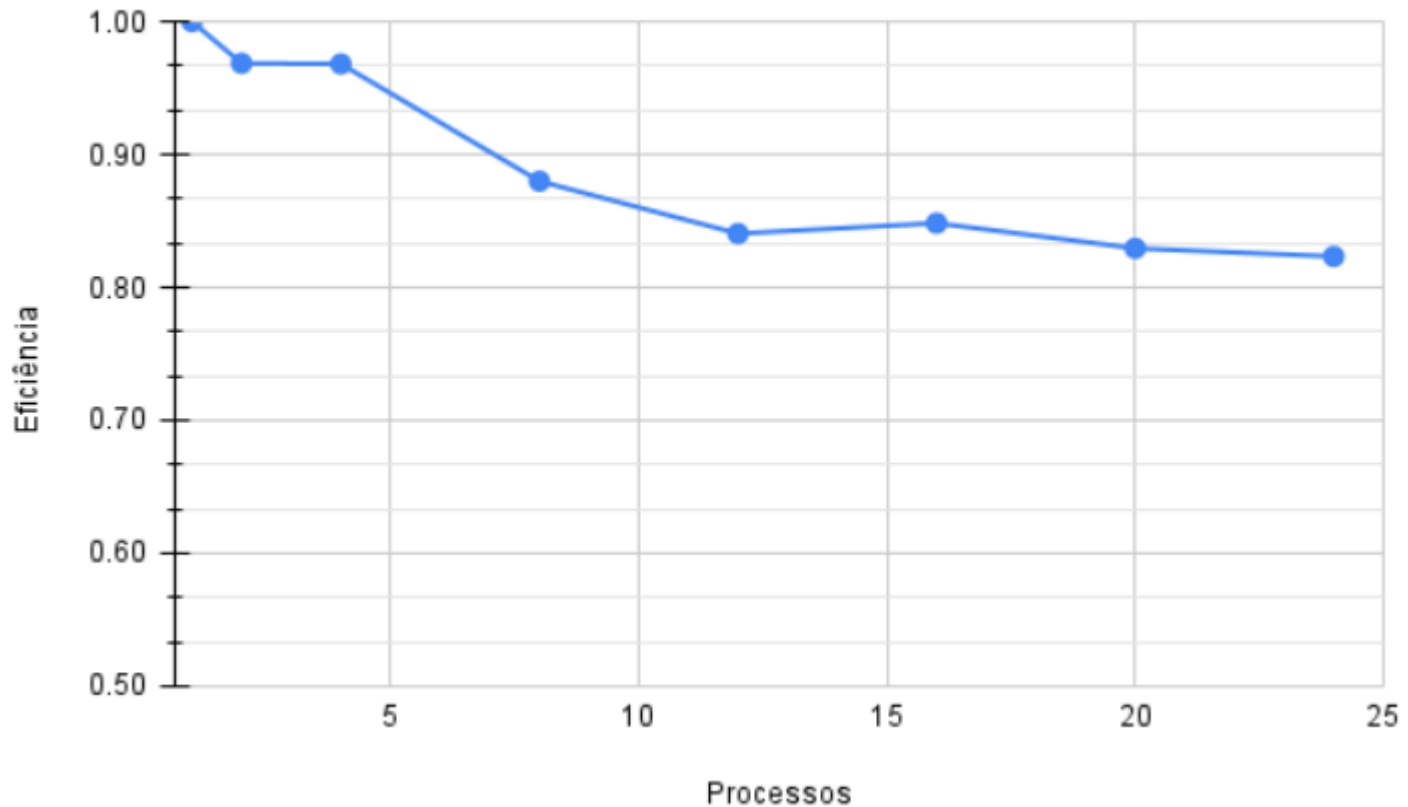
- Eficiência:

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

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**)

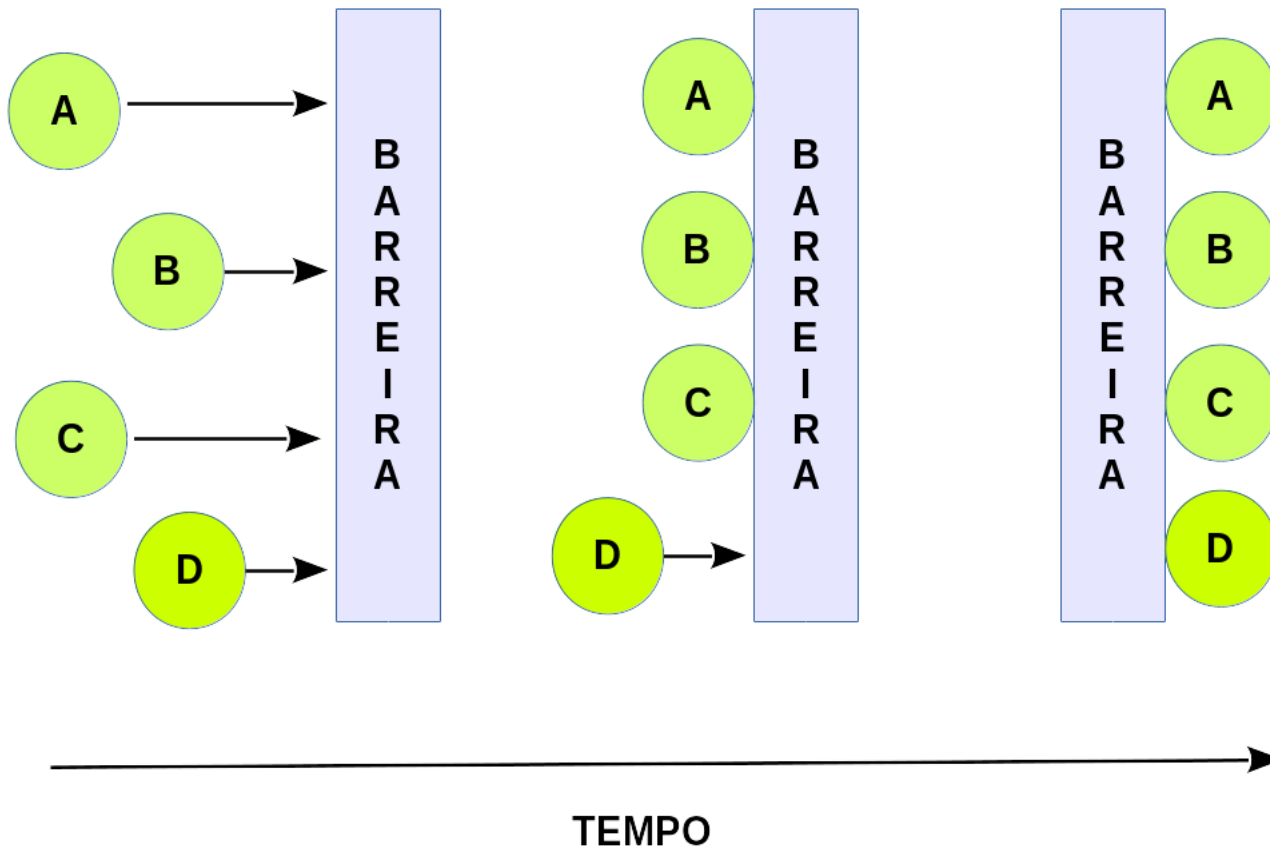
Uma redução seguida de uma difusão.

- Alltoall

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

* Não vamos abordar neste nosso estudo

Barreira



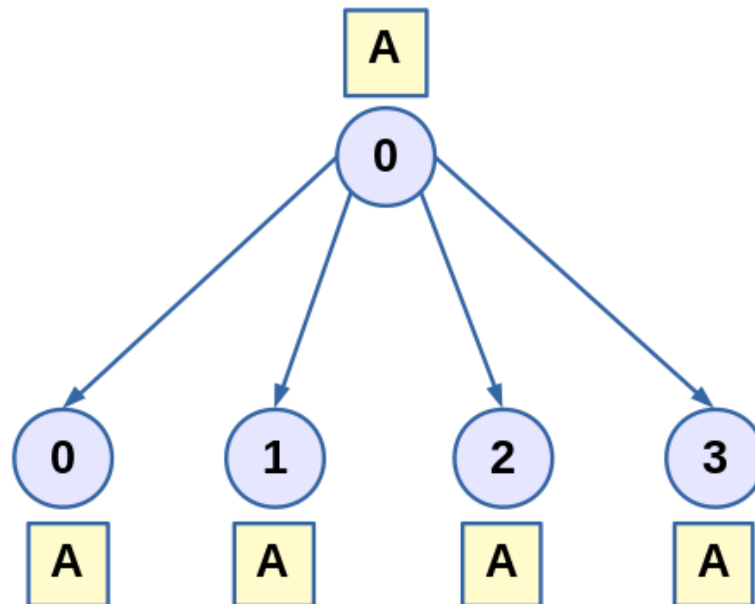
MPI_Barrier

```
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**.

Difusão

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:

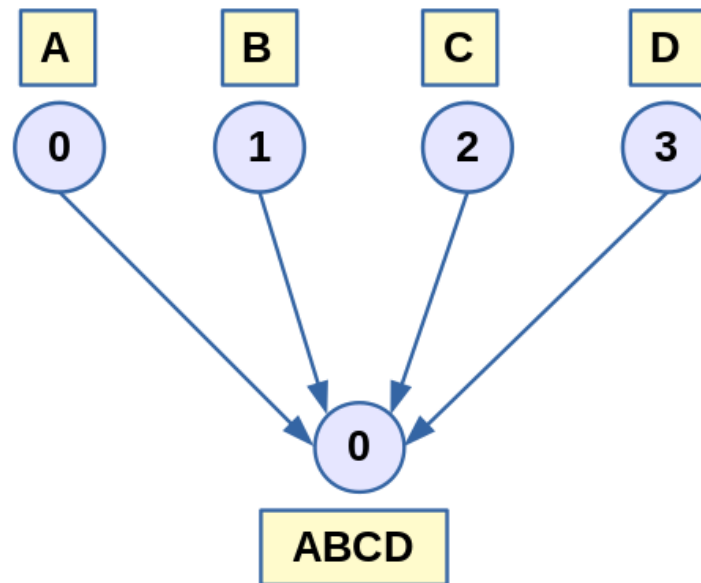
```
int MPI_Bcast (void* mensagem, int cont,  
MPI_Datatype tipo_mpi, int raiz, MPI_Comm com)
```

MPI_Bcast

- Ela simplesmente envia uma cópia dos dados de *mensagem* no processo *raiz* para cada processo no comunicador *com*.
- 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.

Coleta

MPI_Gather



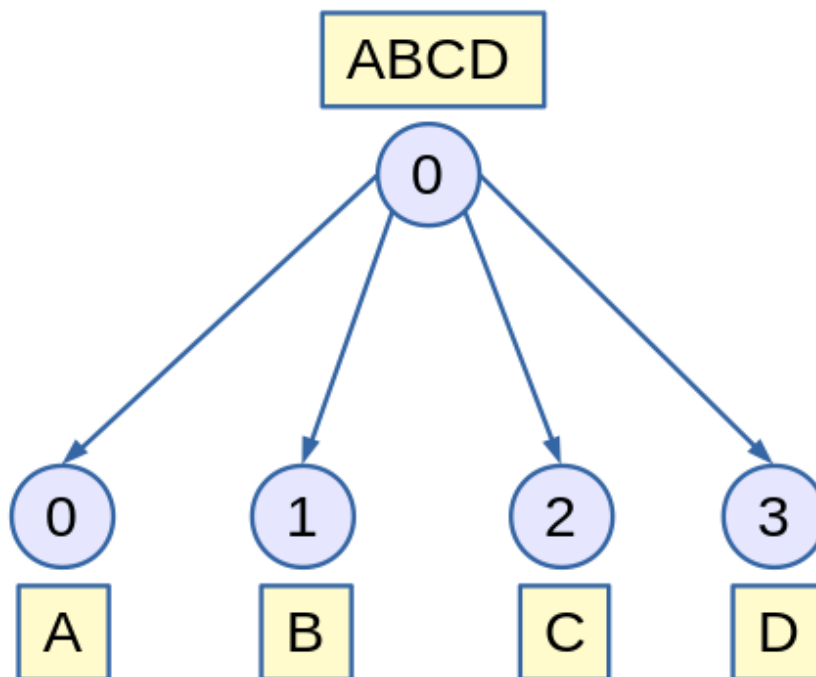
MPI_Gather

```
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

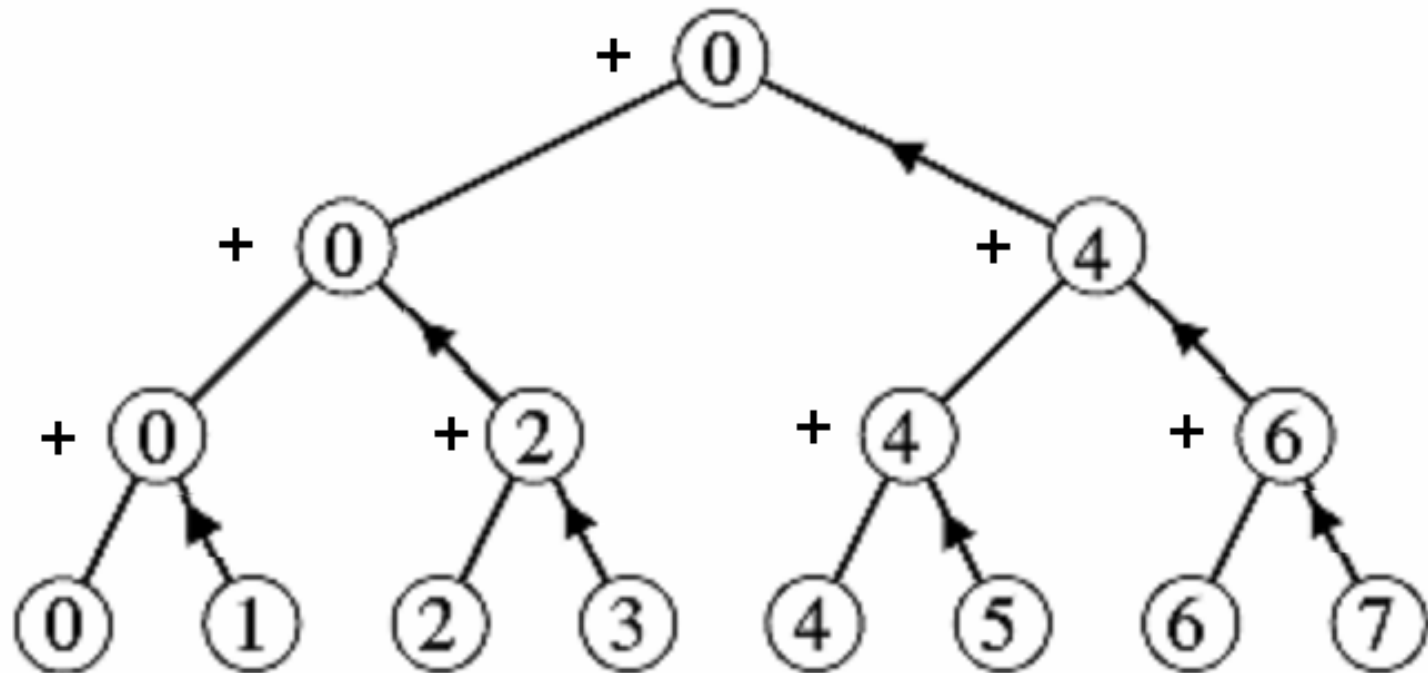
```
int MPI_Scatter(void* vet_envia, int cont_envia,  
MPI_Datatype tipo_envia, void* vet_recebe, int  
cont_recebe, MPI_Datatype tipo_recebe, int raiz,  
MPI_Comm com)
```

- 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

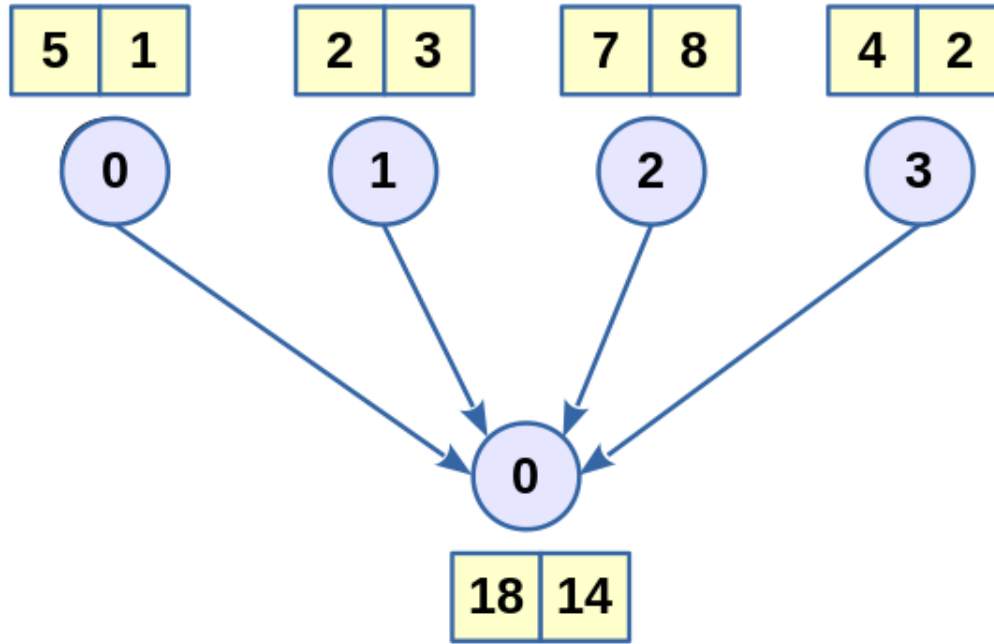
- No programa do método do trapézio, depois da fase de entrada, cada processador executa os mesmos comandos até o final da fase de soma.
- Contudo, este não é caso depois da final da fase de soma, onde as tarefas não são bem balanceadas.
- Podemos utilizar o seguinte procedimento:
 - a) 1 envia resultado 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 */
```

...

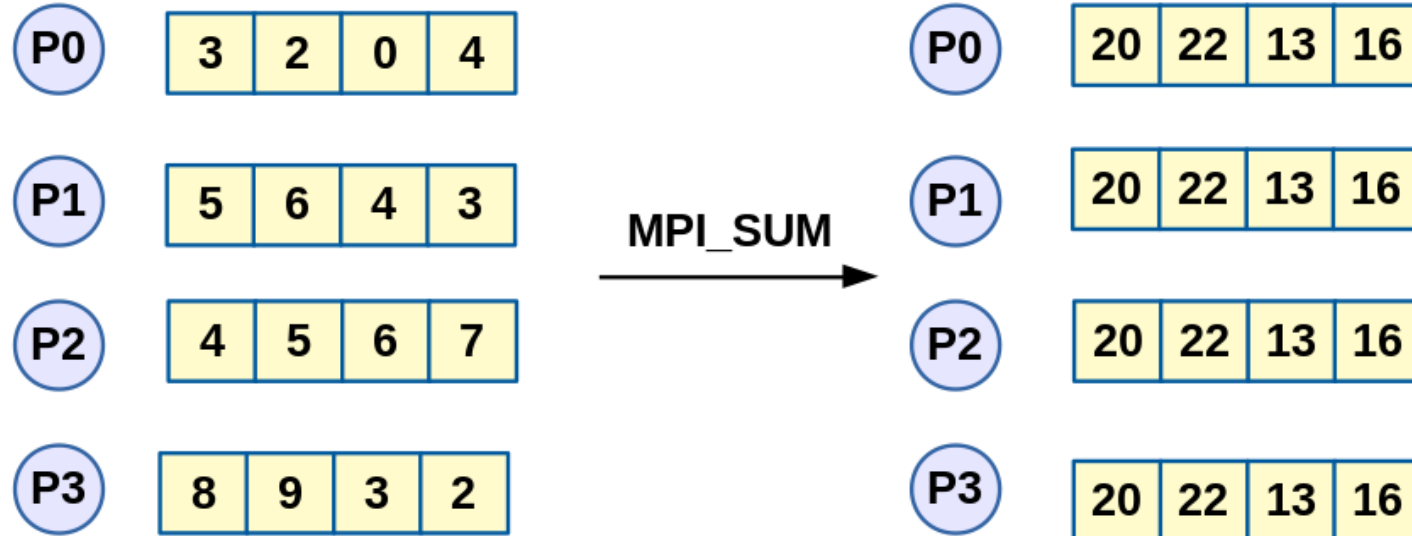
- Note que cada processo chama a rotina **MPI_REDUCE** com os mesmos argumentos.
- Em particular, embora *total* tenha apenas significado no processo 0, cada processo deve fornecê-lo como argumento.

Exemplo Redução

https://github.com/gpsilva2003/MPI/mpi_reduce.c

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

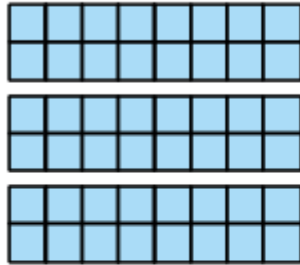
$$A_{m,n} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ a_{2,0} & a_{2,1} & \cdots & a_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \quad b_n = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots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

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

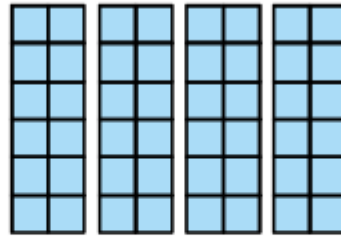
Estudo de caso – Multiplicação Matriz - Vetor

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

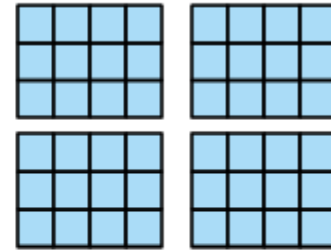
Estudo de caso – Multiplicação Matriz - Vetor



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 $O(n^2 / p)$.

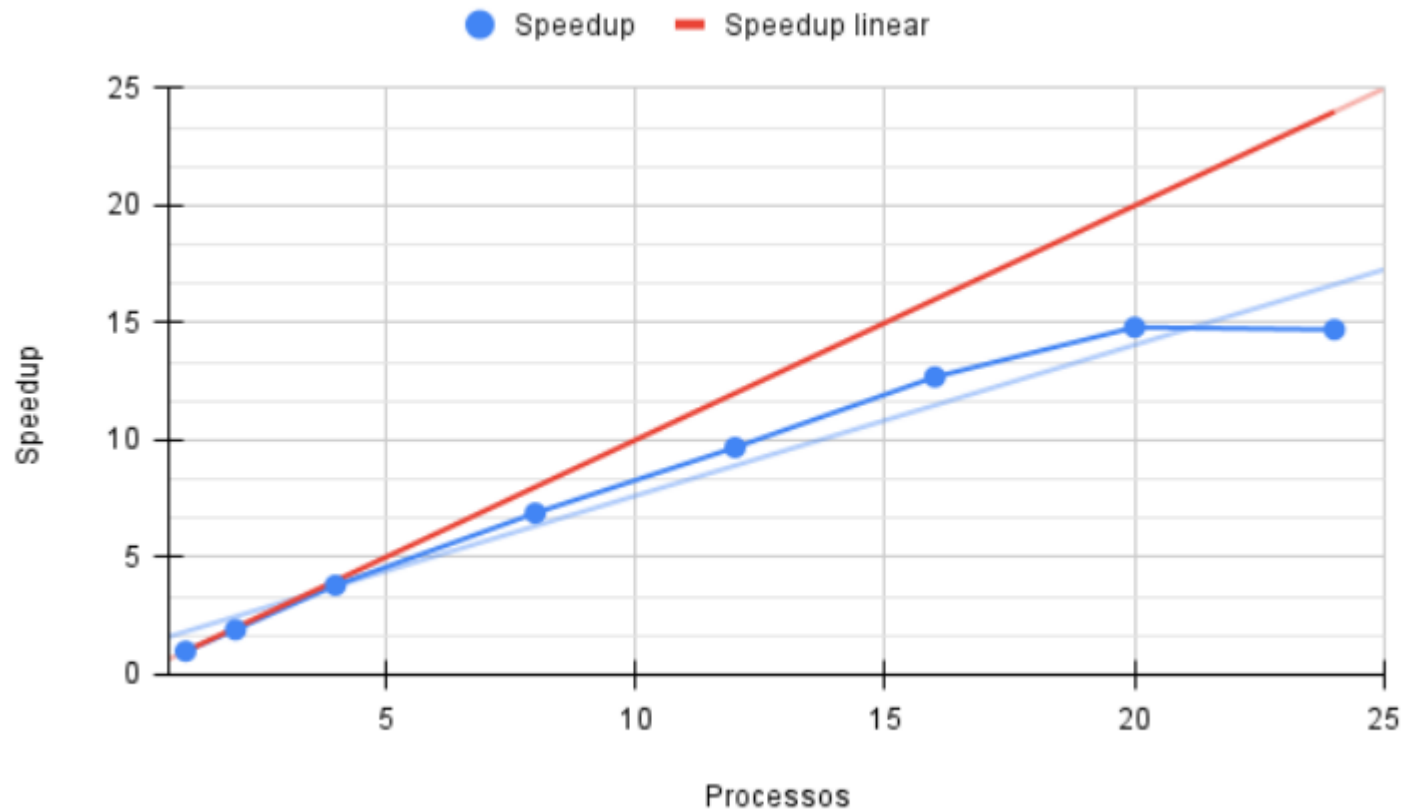
Estudo de caso – Multiplicação Matriz - Vetor

- 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

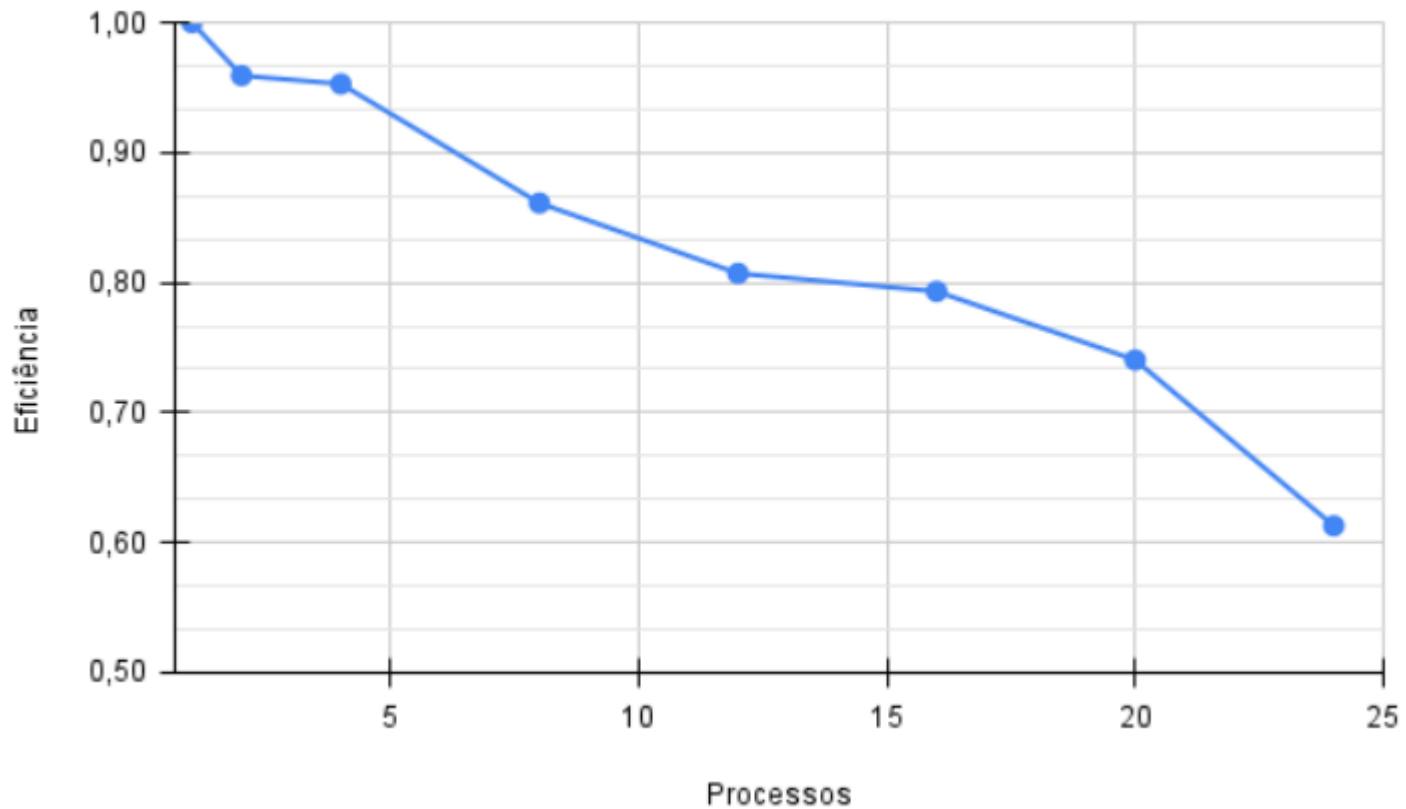
https://github.com/gpsilva2003/MPI/mpi_mxv.c

Multiplicação Matriz – Vetor Speedup



Multiplicação Matriz – Vetor

Eficiência





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 retornar 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 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)
```

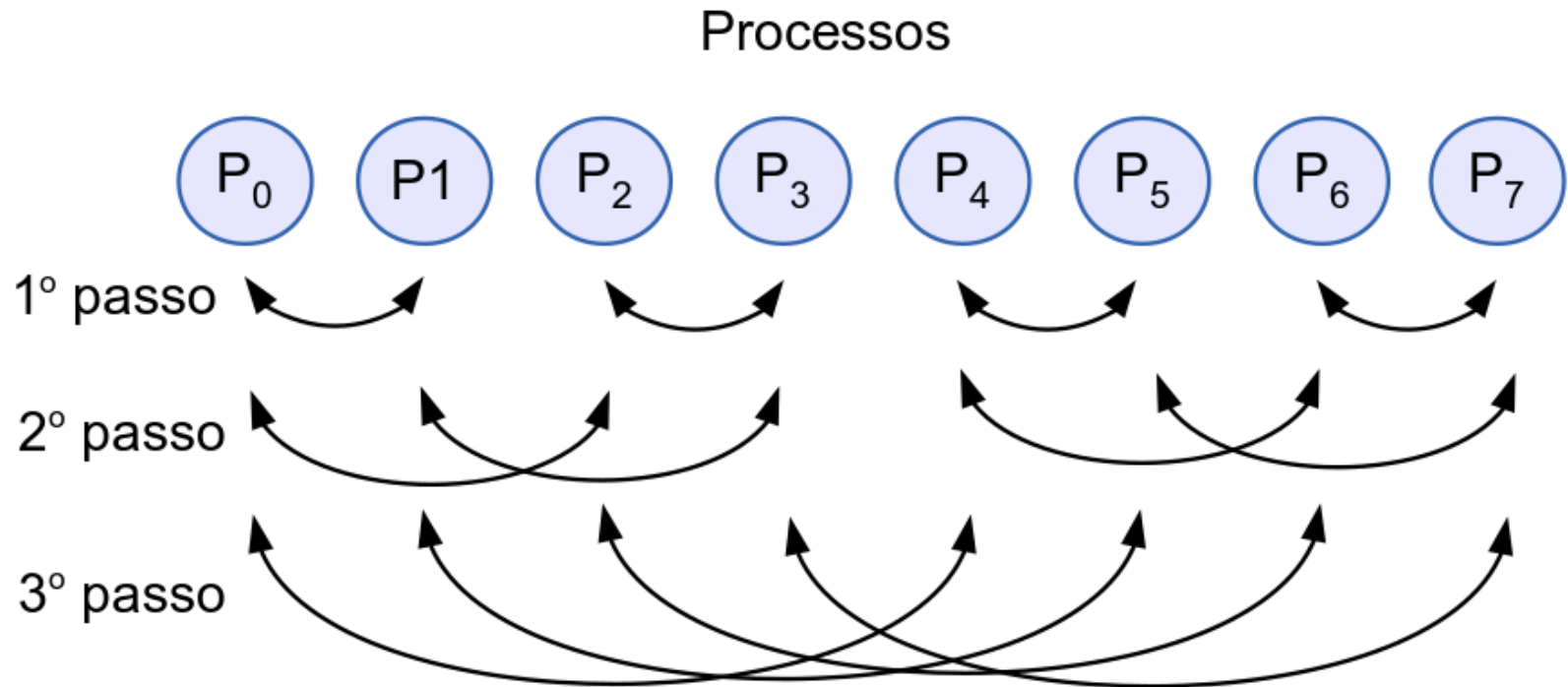
```
int MPI_Waitsome(int cont_entra, MPI_Request  
vetor_de_pedidos[ ], int *cont_saida, int  
vetor_de_indices[ ], MPI_Status vetor_de_estados[ ])
```

- 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



Exemplo Comunicação Não Bloqueante

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)
```


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

Exemplo Comunicação Modo Síncrono

https://github.com/gpsilva2003/MPI/mpi_ssend.c

Exemplo Comunicação Modo Pronto

https://github.com/gpsilva2003/MPI/mpi_rsend.c

Exemplo Comunicação Modo Padrão

https://github.com/gpsilva2003/MPI/mpi_padrao.c

Estudo de Caso - Primos

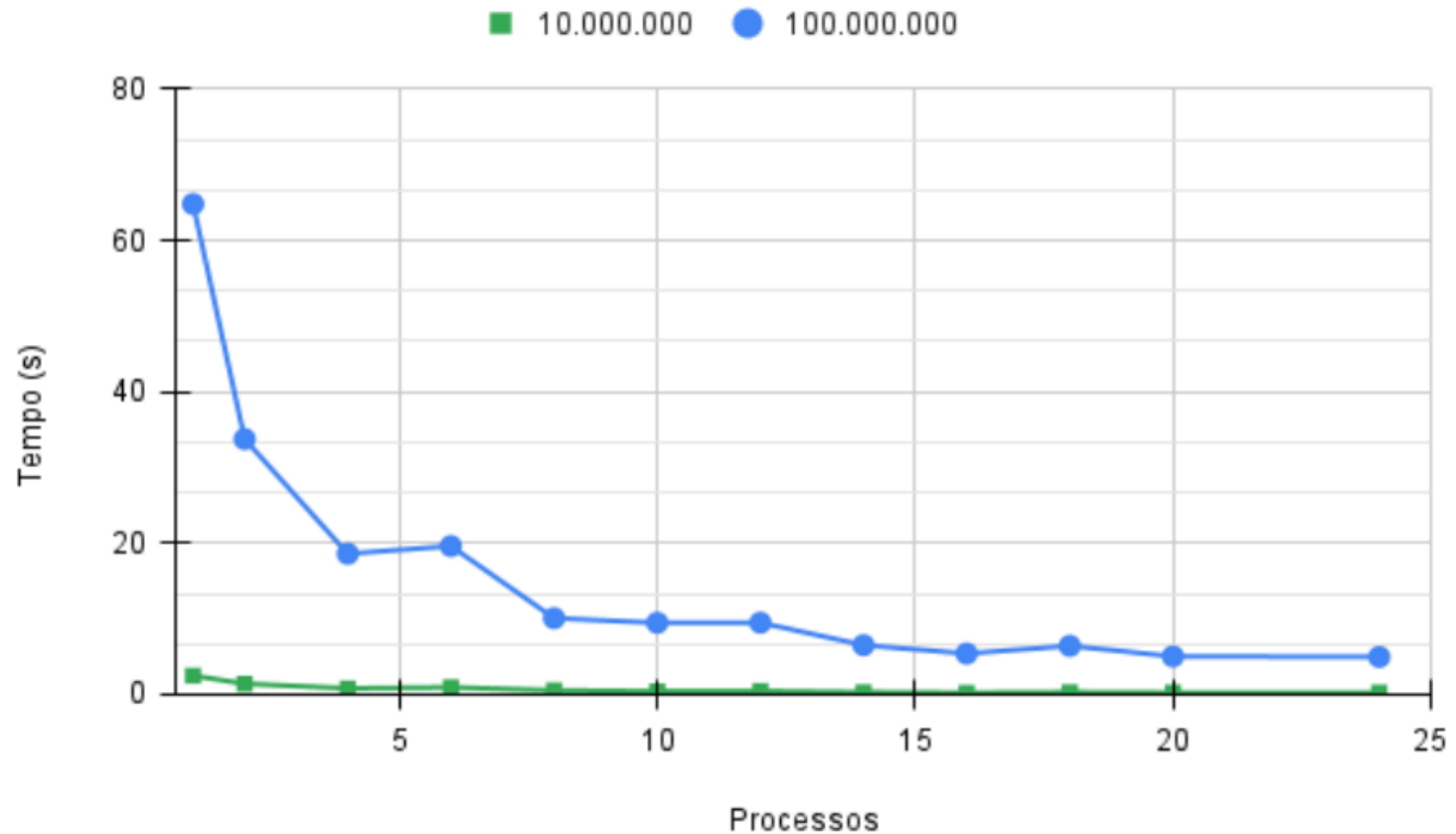
- Nesta seção, nosso estudo de caso será um programa para calcular a quantidade de números primos entre 0 e um determinado valor inteiro N.
- Ele basicamente verifica se N é divisível por algum número ímpar entre 0 e a raiz quadrada de N, sendo que os números pares são descartados de imediato.

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

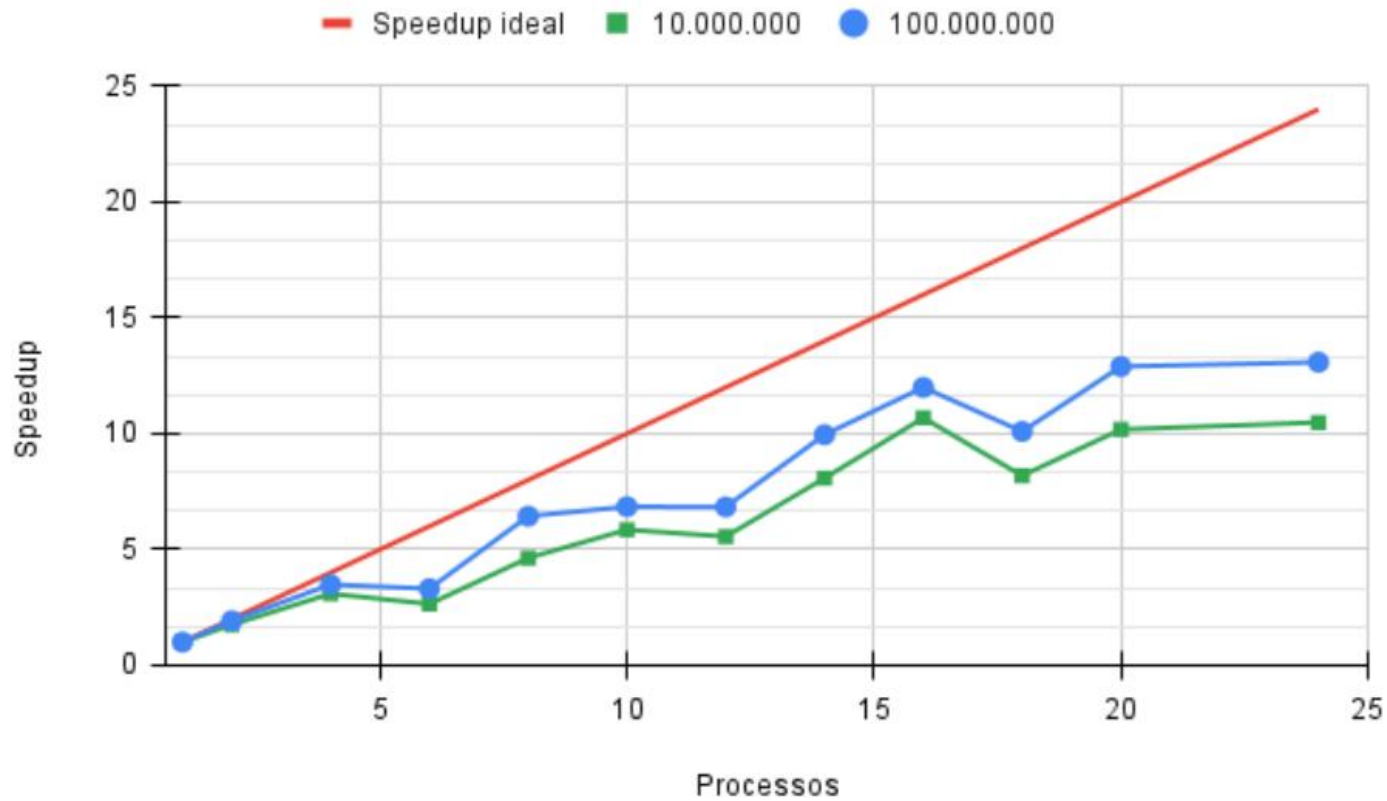
Exemplo Primos – Naive

https://github.com/gpsilva2003/MPI/mpi_primos.c

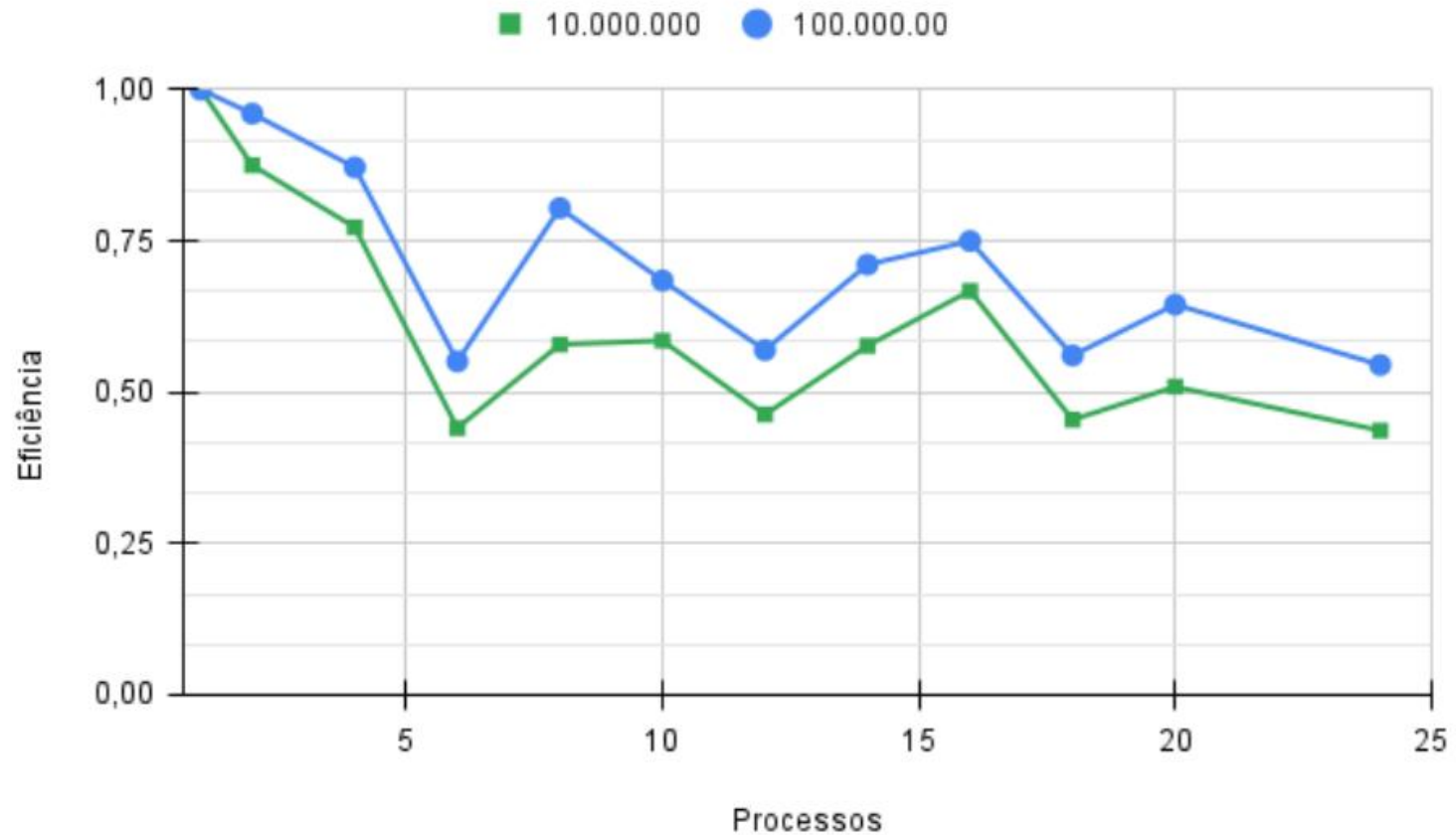
Primos – Naive – Tempo de Execução



Primos – Naive - Speedup



Primos – Naive – Eficiência

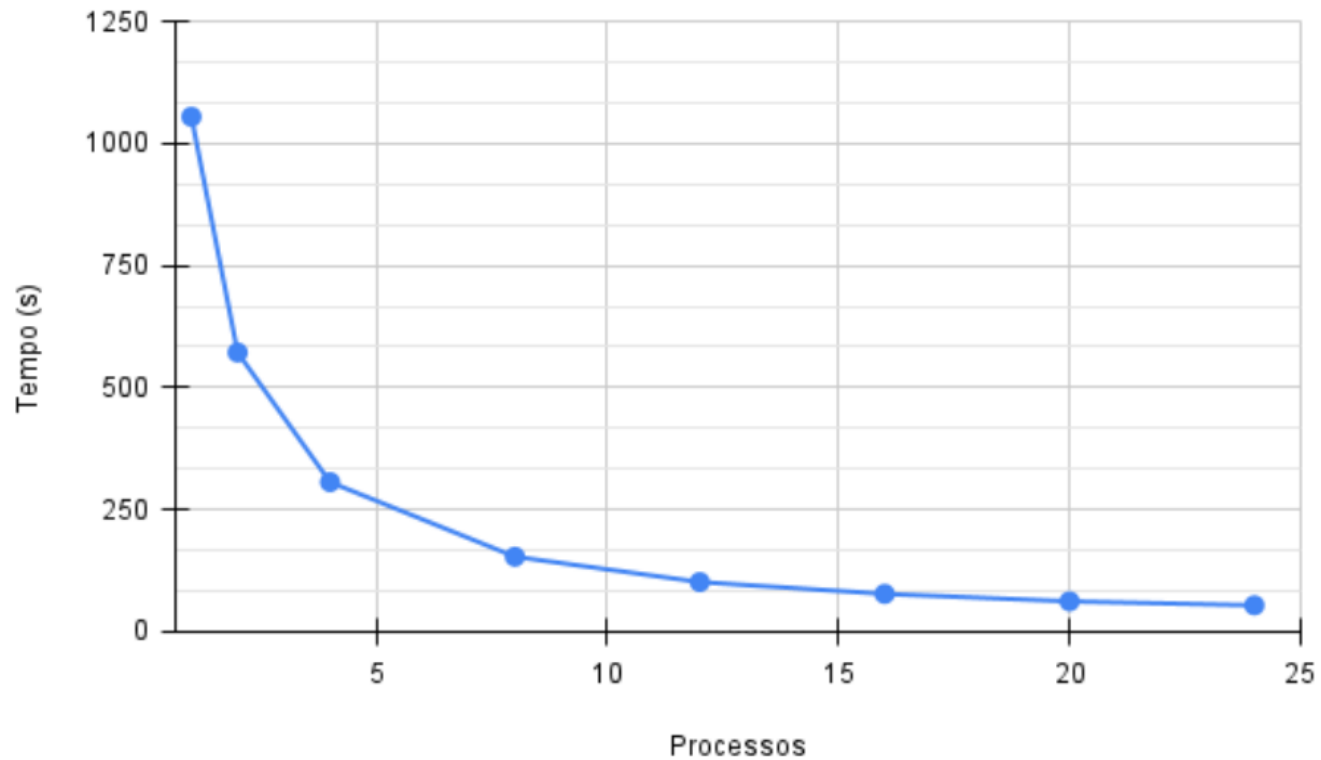


Exemplo Primos – Saco de Tarefas

https://github.com/gpsilva2003/MPI/mpi_primosbag.c

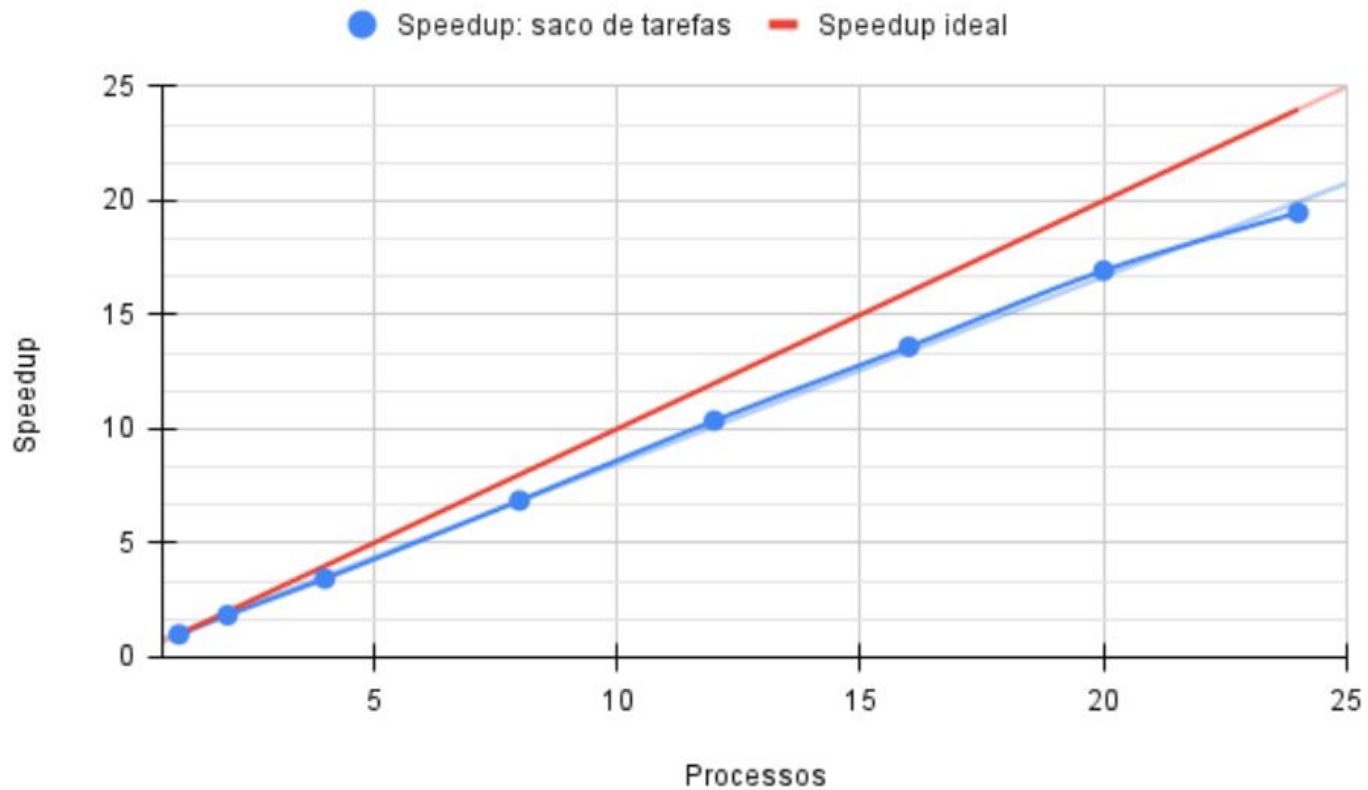
Primos - Saco de Tarefas

Tempo de Execução



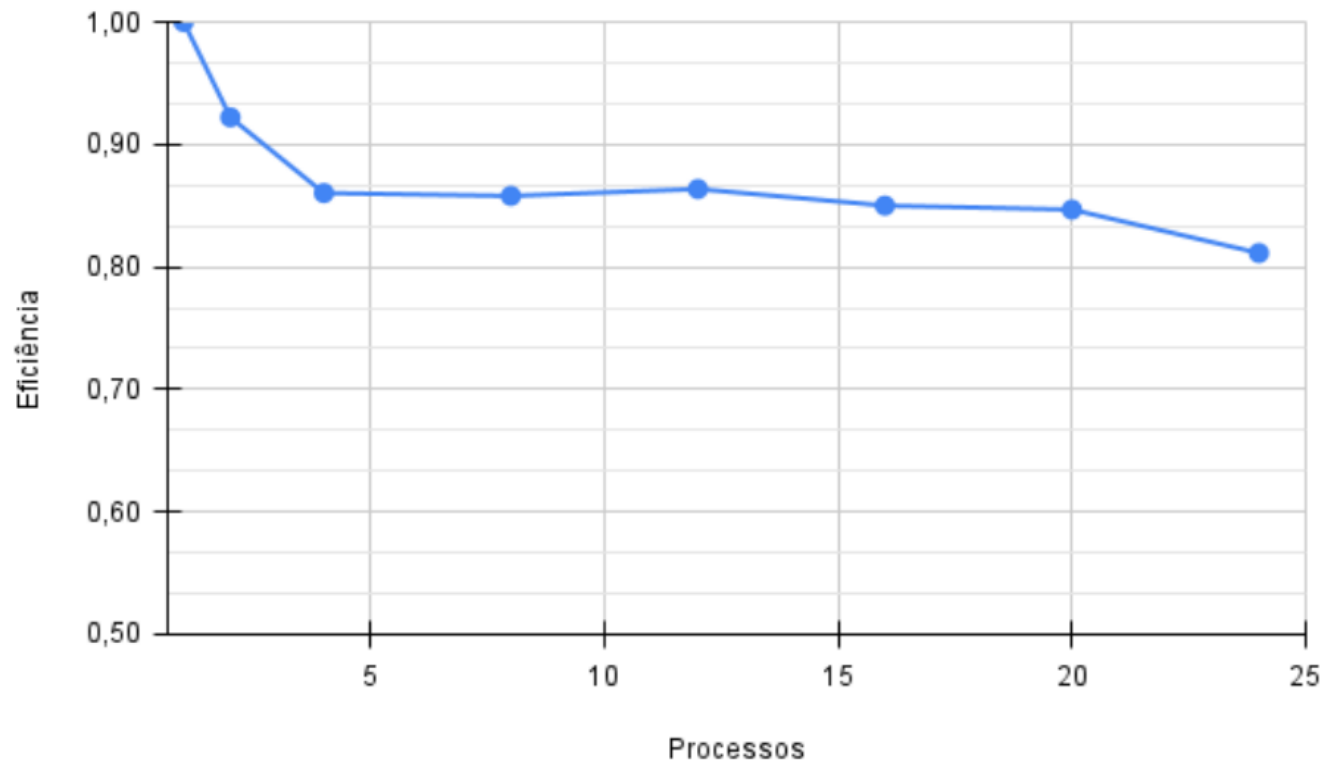
Primos - Saco de Tarefas

Speedup



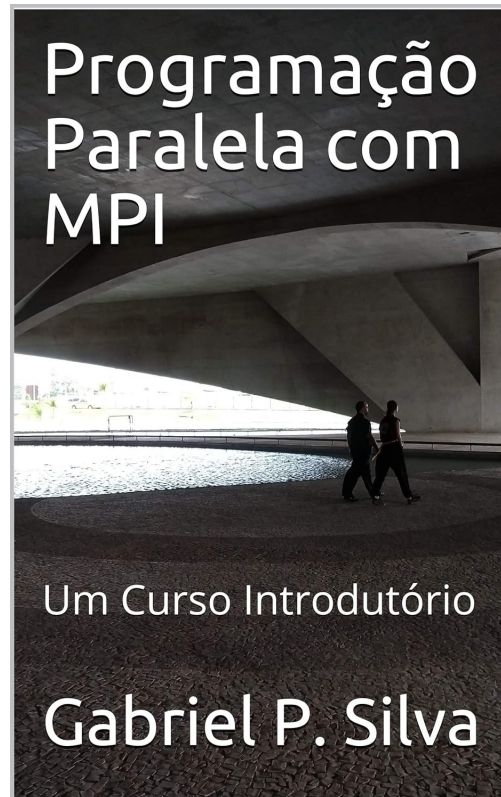
Primos - Saco de Tarefas

Eficiência



Agrupamento de Dados

Referência



<https://www.amazon.com.br/Programação-Paralela-com-MPI-Introductório-ebook/dp/B079WNMF52>

Agrupando Dados

- MPI provê três mecanismos para agrupamento de vários itens em uma única mensagem:
 - O parâmetro *count* para várias rotinas de comunicação.
 - As rotinas MPI Pack/MPI Unpack.
 - Tipos de dados derivados;
- O parâmetro *count*:
 - Lembre-se que as rotinas *MPI_Send*, *MPI_Receive*, *MPI_Bcast* e *MPI_Reduce* têm todas um argumento *count* e um *datatype*.
 - Este dois parâmetros permitem ao usuário agrupar itens de dados tendo os *mesmos* tipos básicos em uma única mensagem.

Agrupando Dados

- De modo a utilizá-los, os itens de dados agrupados devem estar armazenados em posições **contíguas** de memória.
- Já que a linguagem 'C' garante que os elementos de um arranjo (matriz ou vetor) estão armazenados em posições contíguas de memória, se desejarmos enviar os elementos de um arranjo, ou um subconjunto deste, podemos fazê-lo com uma única mensagem.
- Isso permite, por exemplo, enviar a segunda metade de um vetor com 100 valores de ponto flutuante de um processo para outro.

Agrupando Dados

- Infelizmente isto não resolve o problema do envio de variáveis de uma estrutura em linguagem C, como por exemplo.

```
struct ponto {  
    float a;  
    float b;  
    int n;  
};
```

- Pois a linguagem C não garante que elas estejam armazenadas em posições contíguas de memória.
- Além disso, esse tipo de dados não está definido entre os tipos de dados possíveis de serem utilizados pelo MPI.

Agrupando Dados

- O problema aqui é que o MPI é uma biblioteca de funções pré-existentes. Isto é, as funções MPI são escritas sem conhecimento dos tipos de dados que você define no seu programa.
- O MPI fornece algumas soluções para esse problema, uma delas é o empacotamento de dados e a outra a criação de tipos de dados MPI em tempo de execução, no que é chamado de **tipo de dados composto ou derivado**.
- Vejamos a seguir como isso pode ser feito.

Construtores de Tipos de Dados

Construtores de Dados

- Os tipos de dados derivados evitam que tenhamos que fazer operações de empacotamento e desempacotamento de dados.
- Neste caso, o usuário especifica o leiaute dos dados a serem enviados ou recebidos e a biblioteca de comunicação acessa um *buffer* não-contíguo.
- O MPI fornece funções para criar tipos derivados, como ``MPI_Type_contiguous``, ``MPI_Type_vector``, ``MPI_Type_create_struct``, entre outras.

Construtores de Dados

- O construtor de dados mais simples na realidade constrói um tipo derivado cujos elementos são entradas contíguas em um vetor.
- O segundo constrói um tipo cujos elementos são entradas igualmente espaçadas de um vetor.
- O terceiro constrói um tipo cujos elementos são entradas arbitrárias de um vetor.
- Note que antes que qualquer tipo derivado possa ser utilizado para comunicação ele deve ser concluído com uma chamada para `MPI_Type_commit`.
- Detalhes da sintaxe dos construtores de tipo adicionais são mostrados a seguir.

MPI_Type_contiguous

```
int MPI_Type_contiguous(int count, MPI_Datatype  
oldtype, MPI_Datatype* newtype)
```

- **MPI_Type_contiguous** cria um tipo derivado de dados consistindo de *count* elementos do tipo *oldtype*. Os elementos pertencem a posições de memória contíguas.

MPI_Type_contiguous

```
count = 4;  
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);
```

9.0	10.0	11.0	12.0
-----	------	------	------

1 element of
rowtype

MPI_Type_vector

```
int MPI_Type_vector(int count, int block_length, int  
stride, MPI_Datatype element_type, MPI_Datatype*  
newtype)
```

- **MPI_Type_vector** cria um tipo derivado que consiste de *count* elementos. Cada elemento contém *block_length* entradas do tipo *element_type*. Stride é o número de elementos de tipo *element_type* entre sucessivos elementos de *newtype*.

MPI_Type_vector

```
count = 4; blocklength = 1; stride = 4;  
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT,  
                &column_type);
```

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

```
MPI_Send(&a[0][1], 1, column_type, dest, tag, comm);
```

2.0	6.0	10.0	14.0
-----	-----	------	------

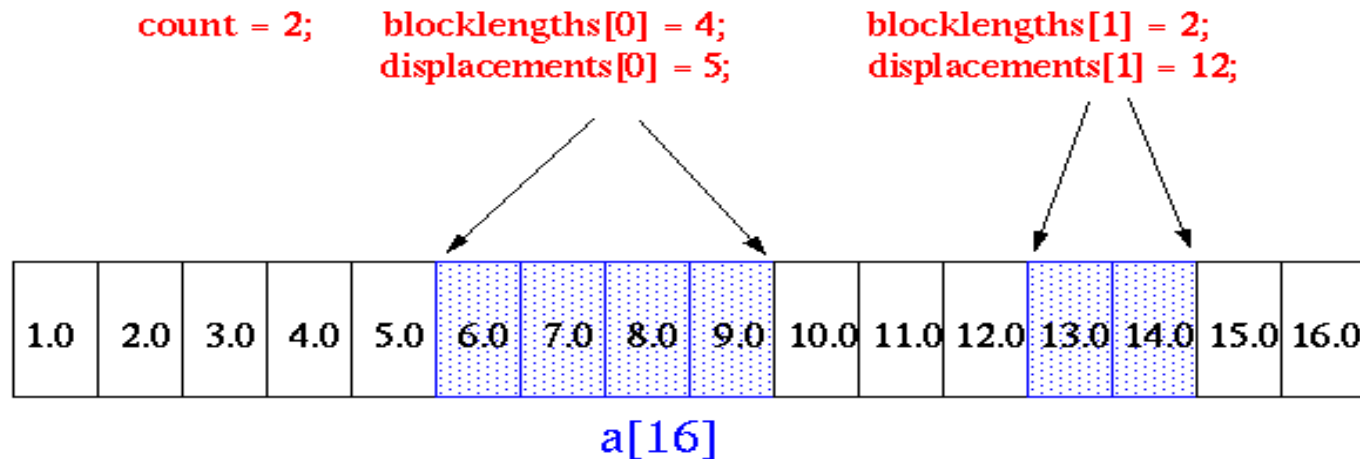
1 element of
column_type

MPI_Type_indexed

```
int MPI_Type_indexed(int count, int*  
array_of_block_lengths, int* array_of_displacements,  
MPI_Datatype element_type, MPI_Datatype*  
newtype)
```

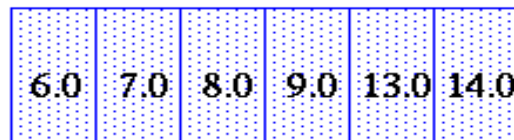
- **MPI_Type_indexed** cria um tipo derivado de dados consistindo de *count* elementos.
- O i-ésimo elemento ($i = 0; 1; \dots; \text{count} - 1$), consiste de um vetor de *block_lengths[i]* entradas do tipo *element_type*, e é deslocado *array_of_displacements[i]* unidades do tipo *element_type* do começo do novo tipo.

MPI_Type_indexed



```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



1 element of
indextype

MPI_Type_create_struct

- Resumindo, nós podemos criar tipos de dados derivados genéricos chamando `MPI_Type_create_struct`, cuja sintaxe é:

```
int MPI_Type_create_struct(int count, int  
vetor_de_comprimento_de_blocos[],  
const MPI_Aint vetor_de_deslocamentos[], const  
MPI_Datatype vetor_de_tipos[],  
MPI_Datatype *novotipo)
```

MPI_Type_create_struct

- O argumento *count* é o número de elementos no tipo derivado. É também o tamanho dos três vetores: *vetor_de_comprimento_de_blocos*, *vetor_de_deslocamentos* e *vetor_de_tipos*.
- O vetor *vetor_de_comprimento_de_blocos* contém o número de entradas em cada elemento do tipo.
- Então, se um elemento do tipo é um vetor de *m* valores, então a entrada correspondente em *vetor_de_comprimento_de_blocos* é *m*.
- O vetor *vetor_de_deslocamentos* contém o deslocamento de cada elemento do início da mensagem, e o vetor *vetor_de_tipos* contém o tipo de dados MPI de cada entrada.

MPI_Type_create_struct

- O argumento *tiponovo* retorna um ponteiro para o tipo de dados MPI criado pela chamada `MPI_Type_create_struct`.
- Note também que *tiponovo* e as entradas em *vetor_de_tipos* todas tem tipo de dados MPI.
- Então `MPI_Type_create_struct` pode ser chamada recursivamente para criar tipos de dados derivados mais complexos 😊 😊 .

MPI_Type_get_extent

- Retorna o limite inferior e a extensão de um tipo de dados, cuja sintaxe é:

```
int MPI_Type_get_extent(MPI_Datatype datatype,  
MPI_Aint *lb, MPI_Aint *extent)
```

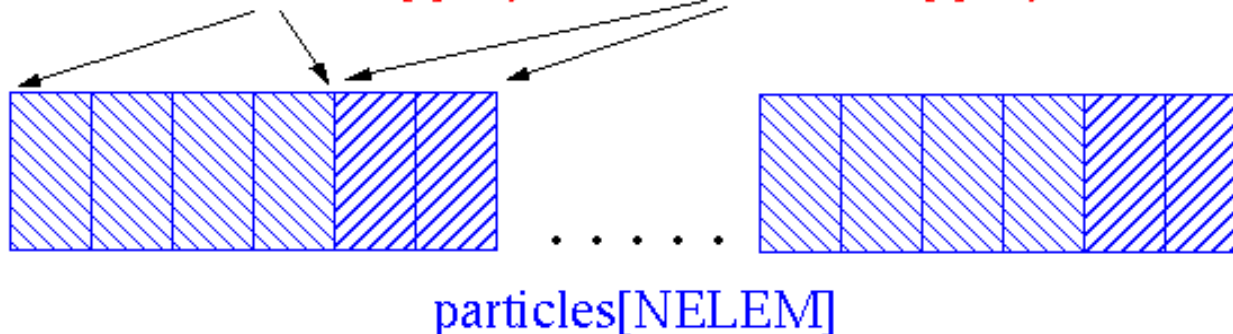
- datatype – O tipo de dados
- lb – o limite inferior, segundo a definição em <https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node83.htm>
- extent - O tamanho do tipo de dados.

MPI_Type_create_struct

```
typedef struct { float x,y,z,velocity; int n,type; } Particle;  
Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);
```

```
count = 2; oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT  
offsets[0] = 0; offsets[1] = 4 * extent;  
blockcounts[0] = 4; blockcounts[1] = 2;
```



```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);
```

```
MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

MPI_Type_create_struct

https://github.com/gpsilva2003/MPI/blob/main/src/mpi_type_create.c

Pack/Unpack

Pack/Unpack

- O MPI oferece funções de pack/unpack para o envio de dados não-contíguos.
- O usuário empacota os dados explicitamente em um buffer antes de enviá-los e desempacota de um buffer contíguo depois de recebê-los.
- As rotinas de pack/unpack são oferecidas para prover compatibilidade com bibliotecas anteriores e são uma opção para o envio de dados não contíguos.
- Em realidade, uma mensagem pode ser recebida em diversas partes, onde a operação de recepção feita em uma parte depende do conteúdo de uma parte anterior.

Pack/Unpack

- Um outro uso é que as mensagens de saída podem ser explicitamente “bufferizadas” em um espaço fornecido pelo usuário, sobrepondo-se a política de “bufferização” do sistema.
- Finalmente, a disponibilidade das operações de pack/unpack facilita o desenvolvimento de bibliotecas de comunicação construídas com o uso do MPI.

MPI_Pack

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype  
datatype, void *outbuf, int outsize, int *position,  
MPI_Comm com)
```

- *inbuf*: início do buffer de entrada
- *incount*: número de itens de entrada (inteiro)
- *datatype*: tipo de dados de cada entrada (handle)
- *outbuf*: início do buffer de saída
- *outsize*: tamanho do buffer de saída (bytes)
- *position*: posição atual no buffer (bytes)
- *com*: comunicador para a mensagem empacotada (handle)

MPI_Pack

- Empacota a mensagem no *buffer* de envio especificado por *inbuf*, *incount*, *datatype* no espaço do buffer especificado por *outbuf* e *outsize*.
- O *buffer input* pode ser qualquer *buffer* de comunicação permitido em MPI_SEND.
- O *buffer* de saída é uma área de armazenamento contiguo que contém *outsize* bytes, iniciando no endereço *outbuf* (o comprimento é contado em bytes, não elementos, como se houvesse um *buffer* de comunicação para a mensagem de tipo MPI_PACKED).

MPI_Pack

- O valor de entrada de *position* é a primeira posição no buffer de saída para ser usada para o empacotamento.
- A variável *position* é incrementada do tamanho da mensagem empacotada, e o valor de saída de *position* é a primeira posição no buffer de saída seguinte às posições ocupadas pela mensagem empacotada.
- O argumento *com* é o comunicador que será usado subsequentemente para o envio da mensagem empacotada.

MPI_Unpack

```
int MPI_Unpack(void* inbuf, int insize, int *position,  
void *outbuf, int outcount, MPI_Datatype datatype,  
MPI_Comm com)
```

- *inbuf*: início do buffer de entrada.
- *insize*: tamanho do buffer de entrada (bytes).
- *position*: posição atual (bytes).
- *outbuf*: início do buffer de saída
- *outcount*: número de itens a serem desempacotados.
- *datatype*: tipo de dados de cada item de saída (handle)
- *com*: comunicador para a mensagem empacotada (handle)

MPI_Unpack

- Desempacota uma mensagem para um buffer de recepção especificado por *outbuf*, *outcount*, *datatype* a partir do espaço do *buffer* especificado por *inbuf* e *insize*.
- O *buffer* de saída pode ser qualquer *buffer* de comunicação permitido em MPI_RECV.
- O *buffer* de entrada é uma área de armazenamento contiguo contendo *insize* bytes, iniciando no endereço *inbuf*.
- O valor de entrada de *position* é a primeira posição no *buffer* de entrada ocupada pela mensagem empacotada.

MPI_Unpack

- A variável *position* é incrementada pelo tamanho da mensagem recebida, de modo que o valor de saída de *position* é a primeira posição livre no buffer de entrada, depois das posições ocupadas pela mensagem que foi desempacotada.
- O parâmetro *com* é o comunicador utilizado para receber a mensagem empacotada.

MPI_Unpack

- Note a diferença entre MPI_RECV e MPI_UNPACK: em MPI_RECV o argumento *count* especifica o número máximo de itens que podem ser recebidos.
- O número real de itens recebidos é determinado pelo comprimento da mensagem que chega.
- Em MPI_UNPACK o argumento *count* especifica o número real de itens que são desempacotados. O tamanho da mensagem correspondente é o incremento em *position*.
- Note que em sistemas heterogêneos, este número não pode ser determinado a priori.

MPI_Pack_Size

- As seguintes chamadas permitem ao usuário descobrir quanto espaço é necessário para empacotar uma mensagem e, então, gerenciar a alocação de espaço para os *buffers*.

```
int MPI_Pack_size(int incout, MPI_Datatype  
datatype, MPI_Comm com, int *size)
```

- *incout*: valor de contagem de entrada (inteiro)
- *datatype*: tipo de dados (handle)
- *com*: comunicador (handle)
- *size*: limite superior no tamanho da mensagem empacotada em bytes (inteiro)

MPI_Pack_Size

- Uma chamada para MPI_PACK_SIZE retorna em *size* um limite superior para o incremento em *position* que é efetuado por uma chamada a MPI_PACK
- A chamada retorna um limite superior, ao invés de um valor exato, já que o total de espaço necessário para empacotar a mensagem pode depender do contexto (p.ex., a primeira mensagem empacotada em uma unidade de empacotamento pode ocupar mais espaço).

Obrigado!

Gabriel P. Silva

gabriel@ic.ufrj.br

<http://github.com/gpsilva2003>

<https://www.ic.ufrj.br/~gabriel>