



# PROGRAMAÇÃO PARALELA MPI 01 – INTRODUÇÃO

Marco A. Zanata Alves

# PROGRAMAÇÃO COM MEMÓRIA DISTRIBUÍDA

**As aplicações** são vistas como um conjunto de programas que são executados de forma independente em diferentes processadores de diferentes computadores. A semântica da aplicação é mantida através da troca de informação entre os vários programas.

# PROGRAMAÇÃO COM MEMÓRIA DISTRIBUÍDA

As **aplicações** são vistas como um conjunto de programas que são executados de forma independente em diferentes processadores de diferentes computadores. A semântica da aplicação é mantida através da troca de informação entre os vários programas.

A **sincronização** e o modo de funcionamento da aplicação é da **responsabilidade do programador**. No entanto, o programador não quer desperdiçar muito tempo com os aspectos relacionados com a comunicação propriamente dita.

# PROGRAMAÇÃO COM MEMÓRIA DISTRIBUÍDA

**As aplicações** são vistas como um conjunto de programas que são executados de forma independente em diferentes processadores de diferentes computadores. A semântica da aplicação é mantida através da troca de informação entre os vários programas.

**A sincronização** e o modo de funcionamento da aplicação é da responsabilidade do programador. No entanto, o programador não quer desperdiçar muito tempo com os aspectos relacionados com a comunicação propriamente dita.

**A comunicação** é implementada por diferentes bibliotecas que cuidam dos detalhes. Essas bibliotecas permitem executar programas remotamente, monitorizar o seu estado, e trocar informação entre os diferentes programas, sem que o programador precise de saber explicitamente como isso é conseguido.

# MESSAGE-PASSING INTERFACE (MPI)

O que **não** é o MPI:

O **MPI não é** um modelo revolucionário de programar máquinas paralelas. Pelo contrário, ele é um modelo de programação paralela baseado na troca de mensagens que pretendeu recolher as melhores funcionalidades dos sistemas existentes, aperfeiçoá-las e torná-las um standard.

O **MPI não é** uma linguagem de programação. É um conjunto de rotinas (biblioteca) definido inicialmente para ser usado em programas C ou Fortran.

O **MPI não é** a implementação. É apenas a especificação!

# PRINCIPAIS OBJETIVOS:

Aumentar a portabilidade dos programas.

Aumentar e melhorar a funcionalidade.

Conseguir implementações eficientes numa vasta gama de arquiteturas.

Suportar ambientes heterogêneos.

# UM POUCO DE HISTÓRIA

O MPI nasceu em 1992 da cooperação entre universidades, empresas e utilizadores dos Estados Unidos e Europa (MPI Forum – <http://www.mpi-forum.org>) e foi publicado em Abril de 1994.

Principais implementações:

- MPICH - <http://www.mcs.anl.gov/mpi/mpich>
- OpenMPI – <http://www.openmpi.org>


Propostas de extensão foram estudadas e desenvolvidas:

- MPI-2
- MPI-IO

# SINGLE PROGRAM MULTIPLE DATA (SPMD)

SPMD é um modelo de programação em que os vários programas que constituem a aplicação são incorporados num único executável.

Cada processo executa uma cópia desse executável. Utilizando condições de teste sobre o ranking dos processos, diferentes processos executam diferentes partes do programa.

```
...  
if (my_rank == 0) {    // similar ao thread_id   
// código tarefa 0  
} ...  
...  
} else if (my_rank == N) {  
// código tarefa N  
}  
...
```

O MPI não impõe qualquer restrição quanto ao modelo de programação (isso depende do suporte oferecido por cada implementação particular). Sendo assim, o modelo SPMD é aquele que oferece a aproximação mais portátil.



# INICIAR E TERMINAR O AMBIENTE DE EXECUÇÃO DO MPI

```
MPI_Init(int *argc, char ***argv)
```

MPI\_Init() inicia o ambiente de execução do MPI.

```
MPI_Finalize(void)
```

MPI\_Finalize() termina o ambiente de execução do MPI.

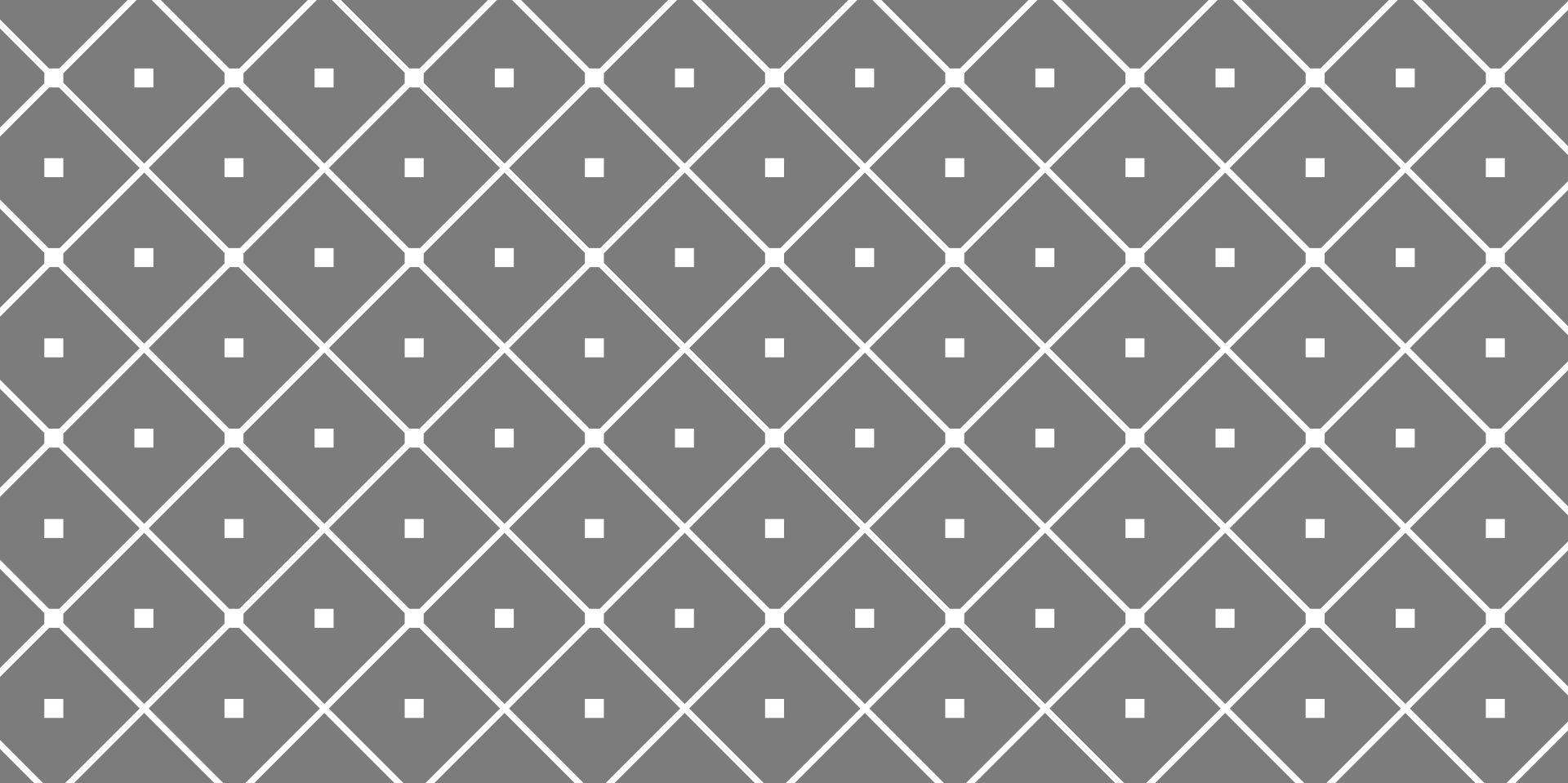
Todas as funções MPI retornam 0 se OK, valor positivo se ERRO.

A especificação não esclarece o que pode ser feito antes da chamada MPI\_Init() ou após a chamada MPI\_Finalize().

Nas implementações MPICH é instruído que sejam feitas a menor quantidade de ações possível. Em particular evitar mudanças no estado externo do programa, como abertura de arquivos, leitura ou escrita do standard input ou output.

# ESTRUTURA BASE DE UM PROGRAMA MPI

```
// incluir a biblioteca de funções MPI
#include <mpi.h>
...
main(int argc, char **argv) {
    ...
    // nenhuma chamada a funções MPI antes deste ponto
    MPI_Init(&argc, &argv);
    ...
    MPI_Finalize();
    // nenhuma chamada a funções MPI depois deste ponto
    ...
}
```



# EXECUTANDO O PROGRAMA COM MPI

# COMPILAÇÃO E EXECUÇÃO DE PROGRAMAS

As implementações MPI disponibilizam um conjunto de scripts para tratar dos caminhos dos headers e libraries necessários à compilação.

- `mpicc` (script de compilação para programas MPI escritos em C)
- `mpic++` (script de compilação para programas MPI escritos em C++)
- `mpif77` (script de compilação para programas MPI escritos em Fortran)

# COMPILAÇÃO E EXECUÇÃO DE PROGRAMAS

O comando `mpirun` permite iniciar a execução distribuída de um dado programa MPI.

Precisamos indicar a seguinte informação:

- A topologia do conjunto de máquinas a executar.
- O número de unidades de execução a lançar por máquina ou por CPU.
- Número de processos a serem lançados.

# EXECUTANDO APLICAÇÕES MPI

Podemos utilizar um arquivo de hosts a serem utilizados

Especifica-se o **nome** das máquinas a utilizar e o **número de CPUs** por máquina, se mais do que 1 (cpu=2).

```
# cluster com 4 máquinas e 6 CPUs
node1
node2
node3 cpu=2
node4 cpu=2
```

Para a execução usamos:

```
mpirun --hostfile <hosts_file> -np <# processos> <binary>
```

# ACESSANDO MÁQUINAS REMOTAS

Para que o MPI crie os processos em diferentes máquinas, é necessário que o usuário possua livre acesso a estas.

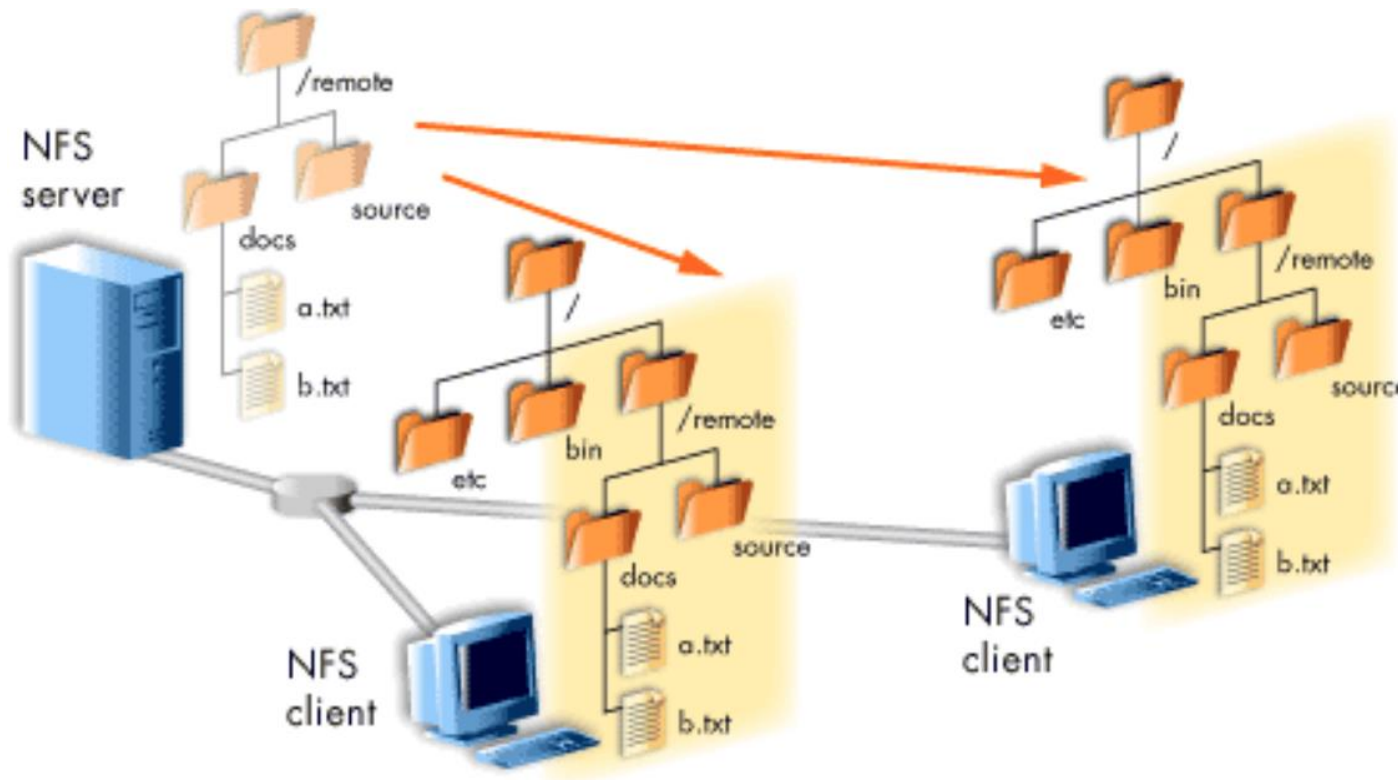
Um esquema para isso pode ser o uso de chaves SSH.

- `ssh-keygen` → Cria chaves públicas
- `ssh-copy-id` → Copia para o servidor especificado as chaves públicas

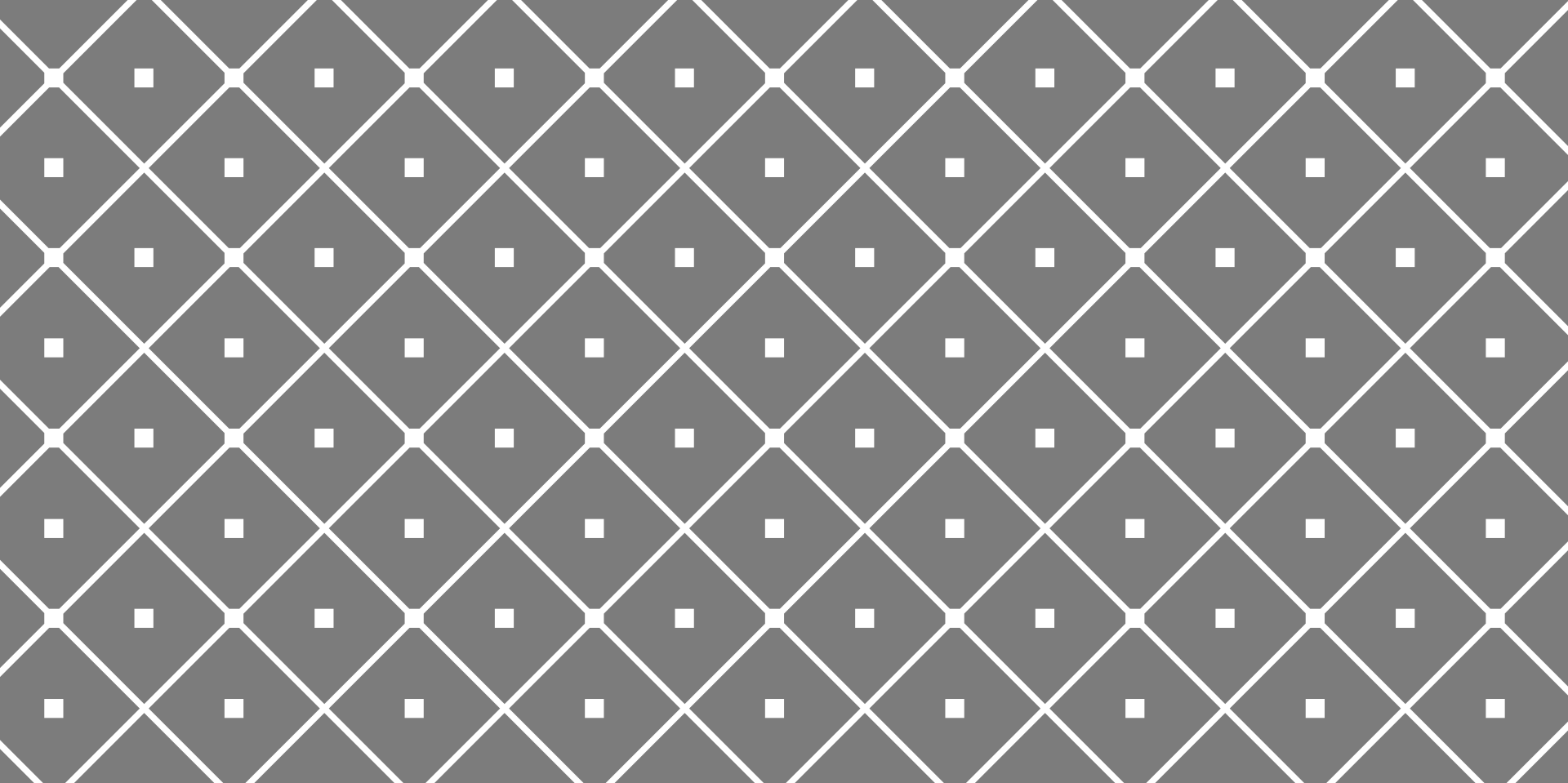
**As máquinas remotas devem possuir também cópia dos binários e demais arquivos a serem utilizados.**

# NETWORK FILE SYSTEM

Em ambientes (ex. UFPR) com NFS (network file system), nossos arquivos já estarão em todas as máquinas automaticamente.



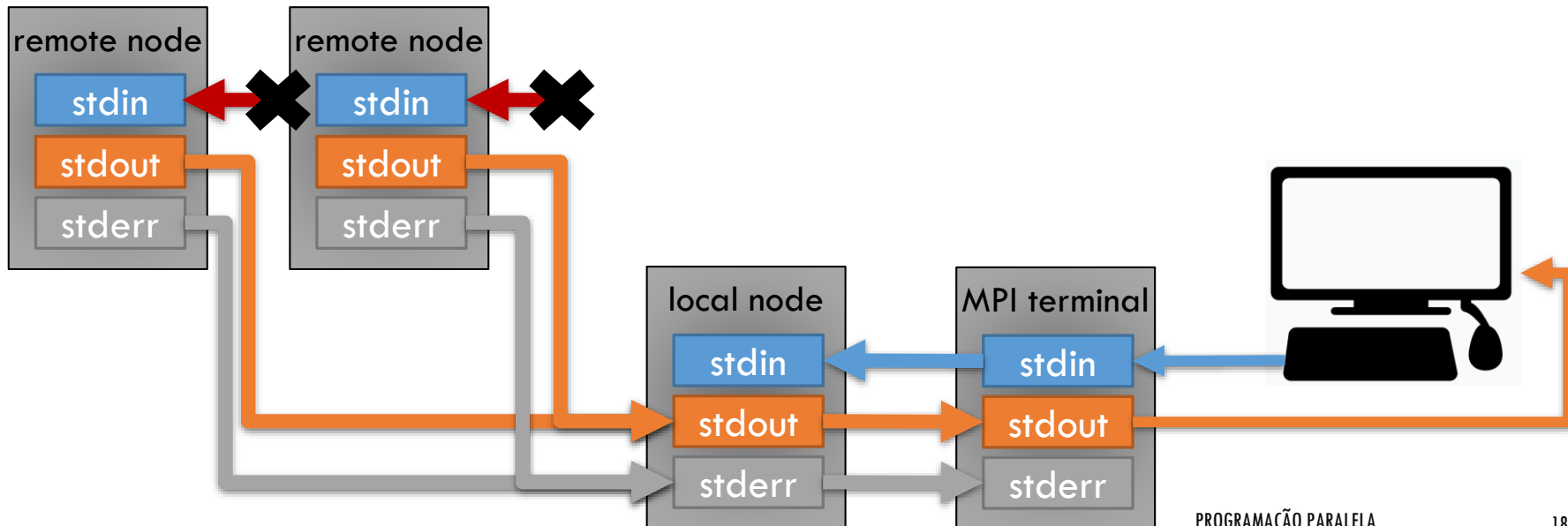


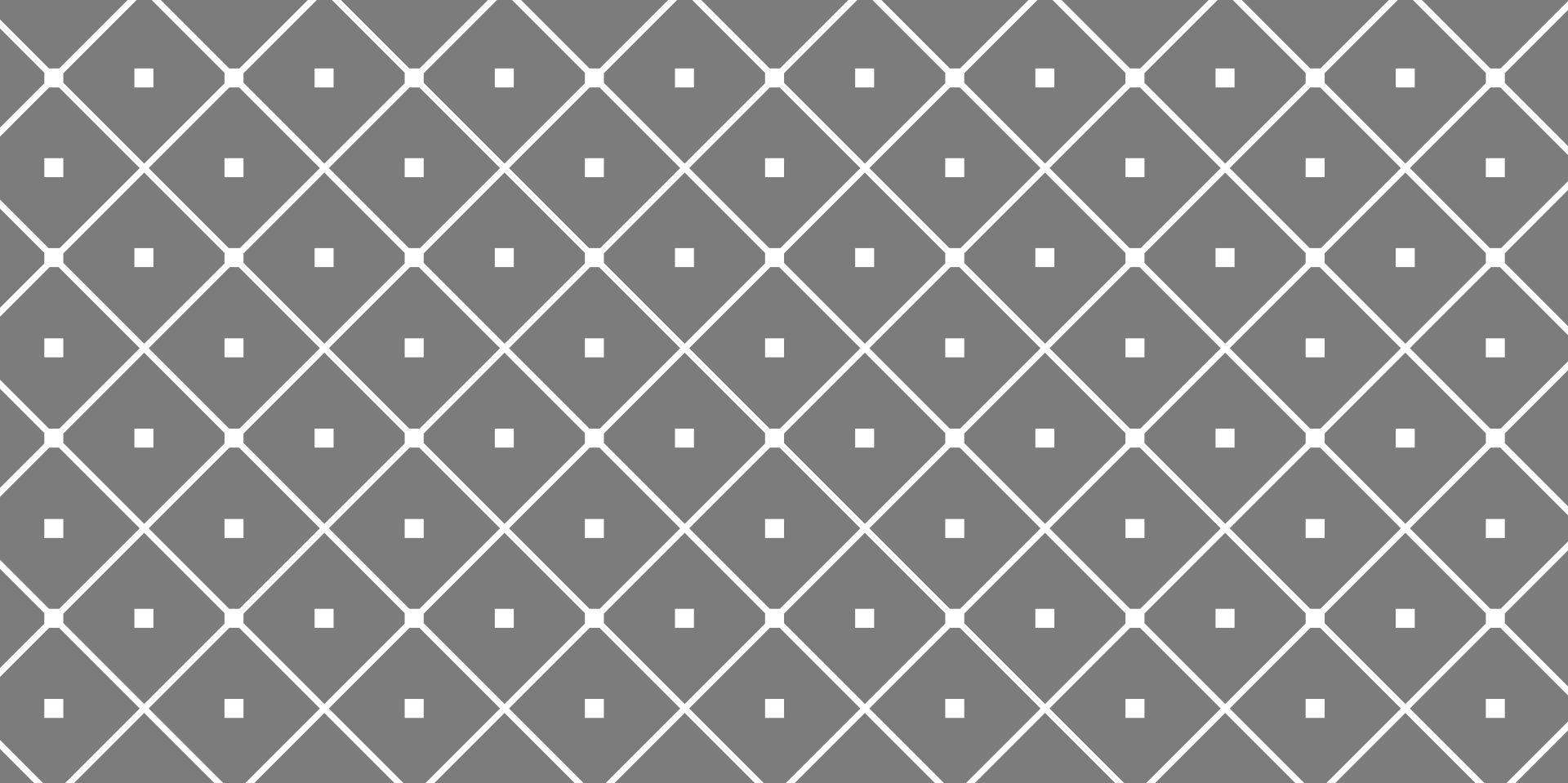


# COMPORTAMENTO DO I/O

# STANDARD I/O

- **standard input** é redirecionado para `/dev/null` em todos os nós remotos.
- O nó local (aquele onde o utilizador invoca o comando que inicia a execução) herda o standard input do terminal onde a execução é iniciada.
- **standard output** e o **standard error** são redirecionados em todos os nós para o terminal onde a execução é iniciada.



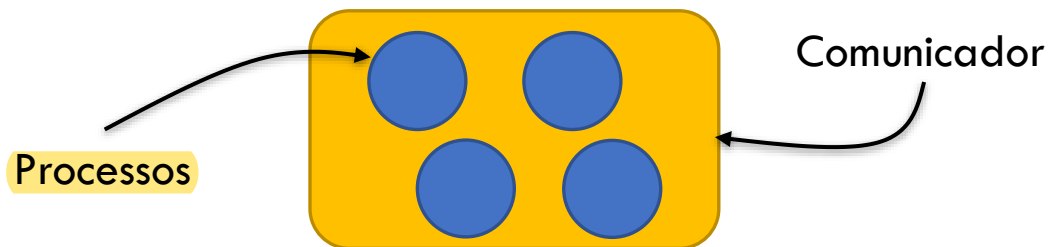


# COMUNICADORES

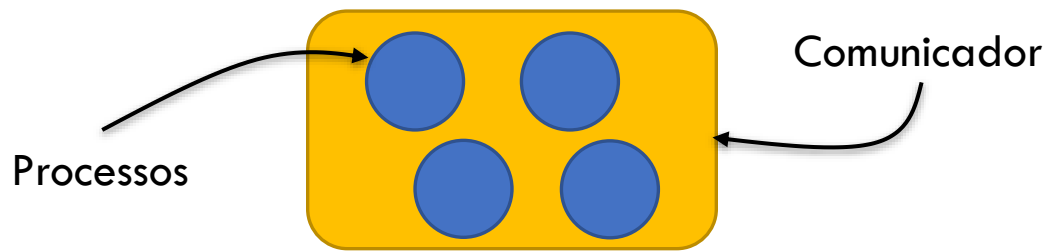
# COMUNICADORES

Uma aplicação MPI vê o seu ambiente de execução paralelo como um conjunto de grupos de processos.

O comunicador é a estrutura de dados MPI que abstrai o conceito de grupo e define quais os processos que podem trocar mensagens entre si. Todas as funções de comunicação têm um argumento relativo ao comunicador.



# COMUNICADORES



Por padrão, o ambiente de execução do MPI define um comunicador universal (**MPI\_COMM\_WORLD**) que engloba todos os processos em execução.

Todos os processos possuem um identificador único (rank) que determina a sua posição (de 0 a N-1) no comunicador. Se um processo pertencer a mais do que um comunicador ele pode ter rankings diferentes em cada um deles.

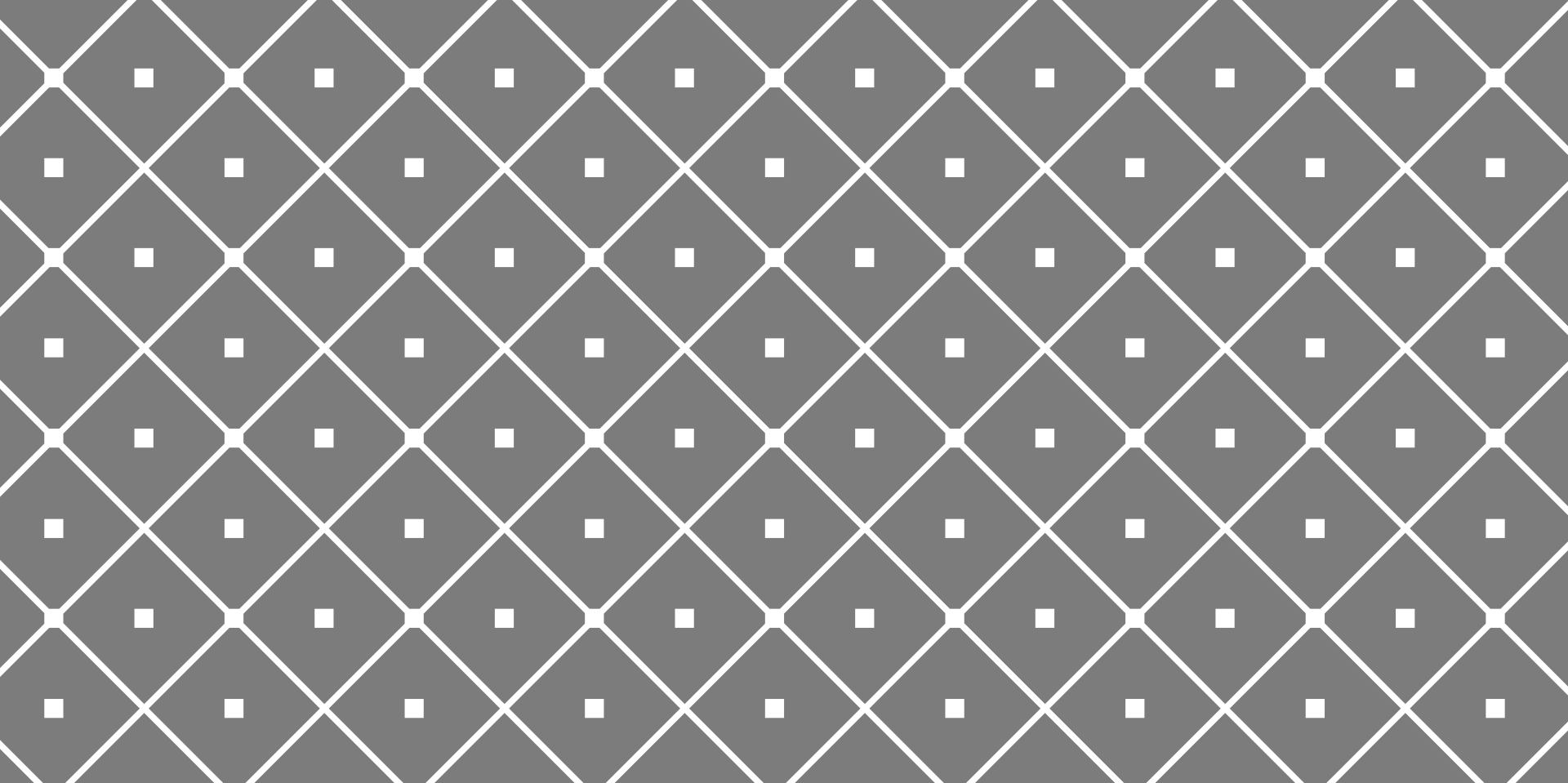
# INFORMAÇÃO RELATIVA A UM COMUNICADOR

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

`MPI_Comm_rank()` devolve em `rank` a posição do processo corrente no comunicador `comm`.

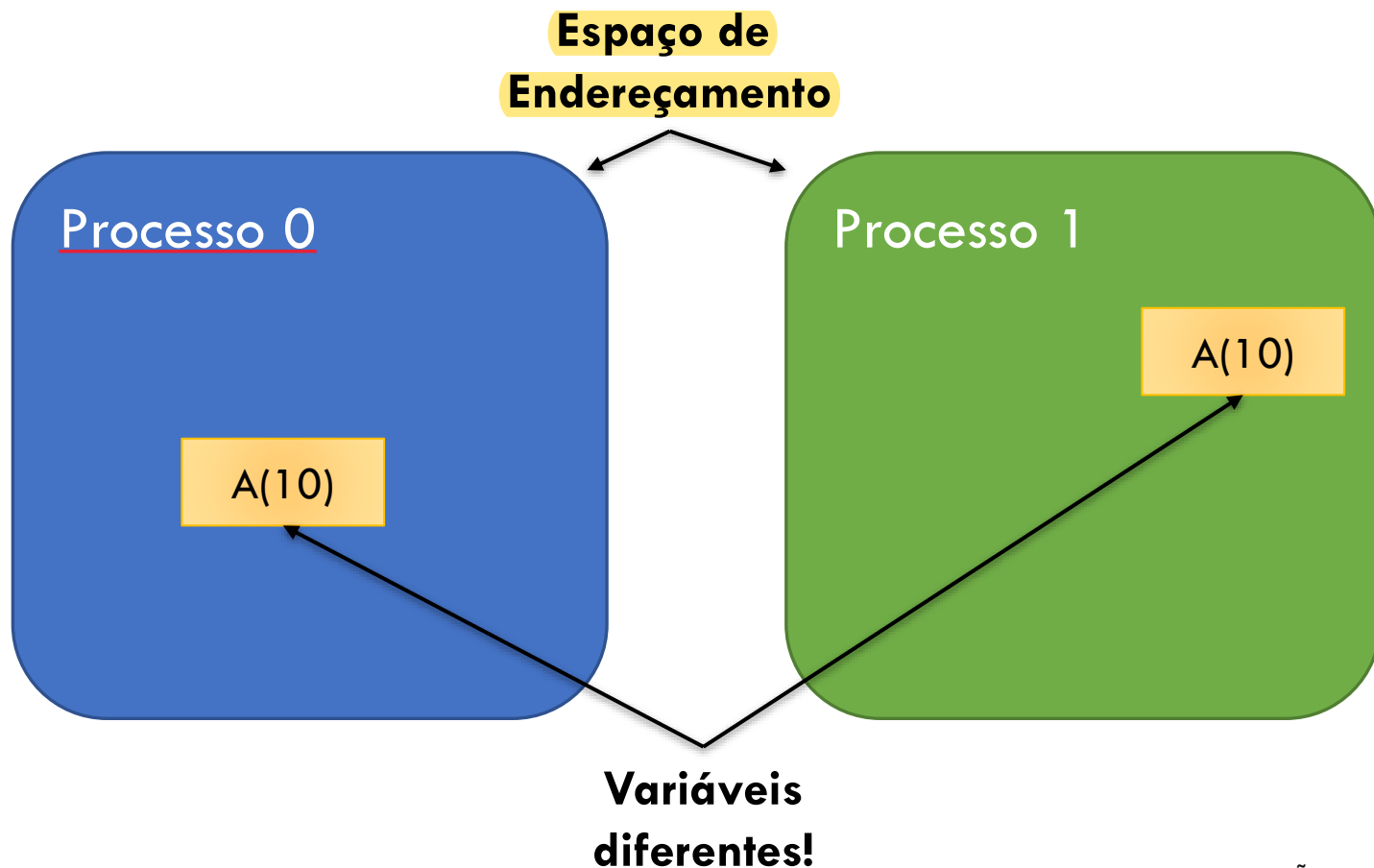
```
MPI_Comm_size(MPI_Comm comm, int *size)
```

`MPI_Comm_size()` devolve em `size` o total de processos no comunicador `comm`.



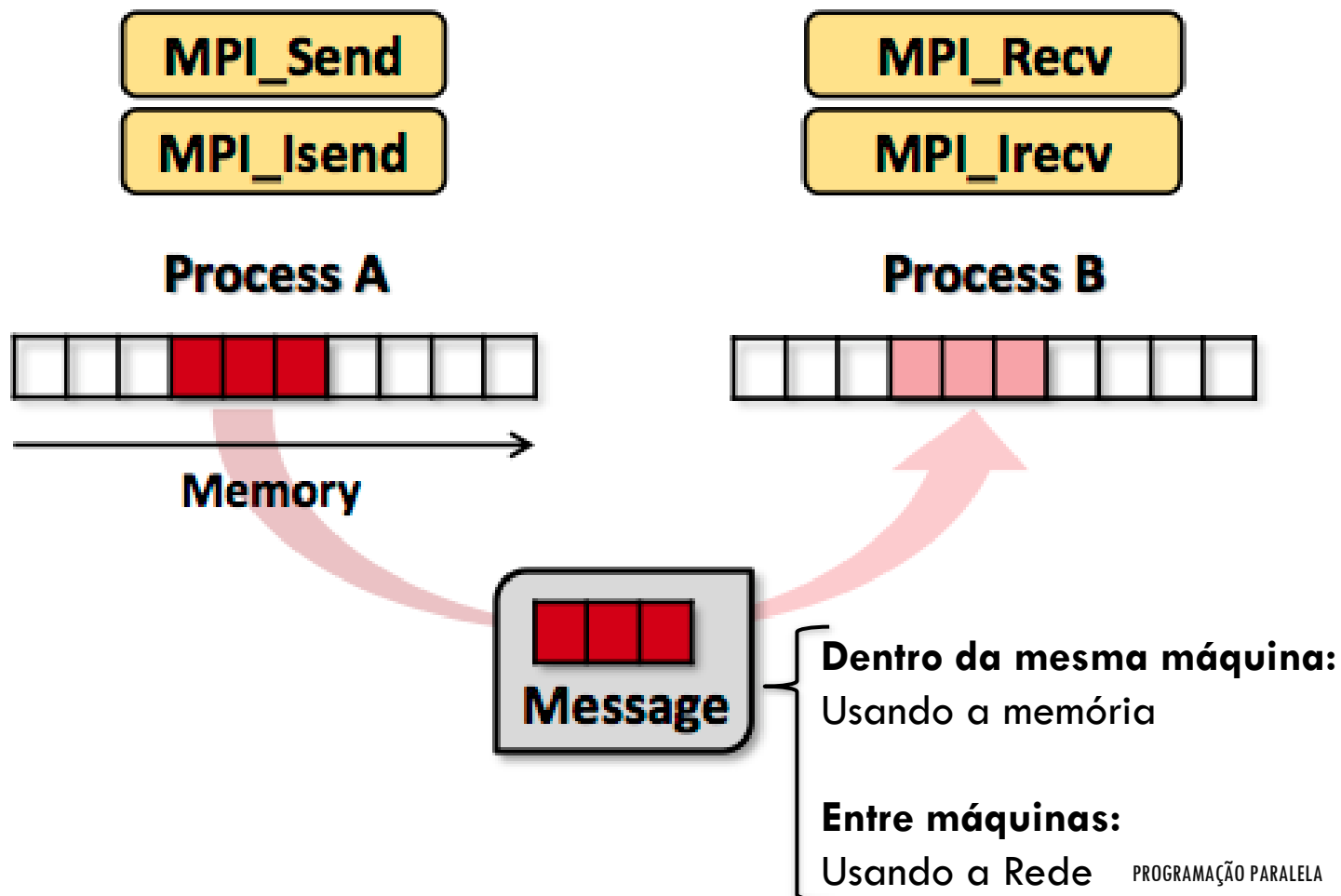
# MENSAGENS

# COMUNICAÇÃO EM MEMÓRIA DISTRIBUÍDA





# COMUNICAÇÃO EM MEMÓRIA DISTRIBUÍDA



# MENSAGENS MPI

Na sua essência, as mensagens não são nada mais do que pacotes de informação trocados entre processos.

Para efetuar uma troca de mensagens, o ambiente de execução necessita de conhecer no mínimo a seguinte informação:

- Processo que envia.
- Processo que recebe.
- Localização dos dados na origem.
- Localização dos dados no destino.
- Tamanho dos dados.
- Tipo dos dados.

O tipo dos dados é um dos itens mais relevantes nas mensagens MPI. Daí, uma mensagem MPI ser normalmente designada como uma sequência de tipo de dados.

# TIPOS DE DADOS BÁSICOS

MPI	C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_PACKED	

# ENVIO STANDARD DE MENSAGENS

```
MPI_Send(void *buf, int count, MPI_Datatype datatype,  
          int dest, int tag, MPI_Comm comm)
```

**MPI\_Send()** é a funcionalidade básica para envio de mensagens.

**buf** é o endereço inicial dos dados a enviar.

**count** é o número de elementos do tipo **datatype** a enviar.

**datatype** é o tipo de dados a enviar.

**dest** é a posição do processo, no comunicador **comm**, a quem se destina a mensagem.

**tag** é uma marca que identifica a mensagem a enviar. As mensagens podem possuir idênticas ou diferentes marcas por forma a que o processo que as envia/recebe as possa agrupar/diferenciar em classes.

**comm** é o comunicador dos processos envolvidos na comunicação.

# RECEPÇÃO STANDARD DE MENSAGENS

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype,  
         int source, int tag, MPI_Comm comm, MPI_Status *status)
```

**MPI\_Recv()** é a funcionalidade básica para recepção de mensagens.

**buf** é o endereço onde devem ser colocados os dados recebidos.

**count** é o número máximo de elementos do tipo **datatype** a receber (tem de ser maior ou igual ao número de elementos enviados).

**datatype** é o tipo de dados a receber (não necessita de corresponder aos dados que foram enviados).

# RECEPÇÃO STANDARD DE MENSAGENS

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype,  
         int source, int tag, MPI_Comm comm, MPI_Status *status)
```

**source** é a posição do processo, no comunicador **comm**, de quem se pretende receber a mensagem. Pode ser **MPI\_ANY\_SOURCE** para receber de qualquer processo no comunicador **comm**.

**tag** é a marca que identifica a mensagem que se pretende receber. Pode ser **MPI\_ANY\_TAG** para receber qualquer mensagem.

**comm** é o comunicador dos processos envolvidos na comunicação.

**status** devolve informação sobre o processo emissor (**status.MPI\_SOURCE**) e a marca da mensagem recebida (**status.MPI\_TAG**). Se essa informação for desprezável indique **MPI\_STATUS\_IGNORE**.

# INFORMAÇÃO RELATIVA À RECEPÇÃO

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

**MPI\_Get\_count()** devolve em **count** o número de elementos do tipo **datatype** recebidos na mensagem associada com **status**.

```
MPI_Probe(int source, int tag, MPI_Comm comm,  
          MPI_status *status)
```

**MPI\_Probe()** sincroniza a recepção da próxima mensagem, retornando em **status** informação sobre a mesma sem contudo proceder à sua recepção.

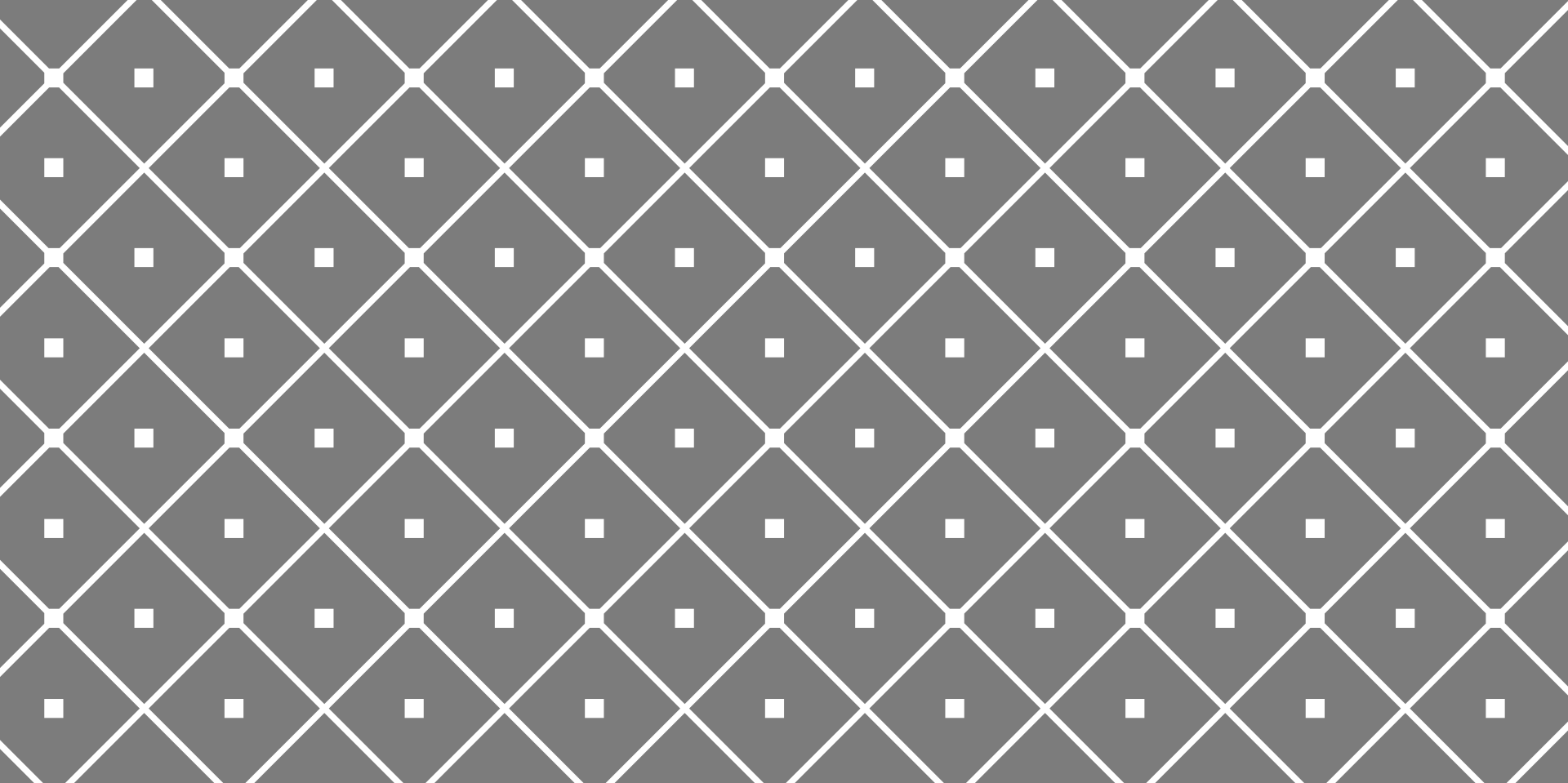
A recepção deverá ser posteriormente feita com **MPI\_Recv()**.

É útil em situações em que não é possível conhecer antecipadamente o tamanho da mensagem e assim evitar que esta exceda o buffer de recepção.

# I'M ALIVE! (MPI ALIVE.C)

```
#include <mpi.h>
#define STD_TAG 0
main(int argc, char **argv) {
    int i, my_rank, n_procs;  char msg[100];  MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    if (my_rank != 0) {
        sprintf(msg, "I'm alive!");
        MPI_Send(msg, strlen(msg) + 1, MPI_CHAR, 0, STD_TAG, MPI_COMM_WORLD);
    } else {
        for (i = 1; i < n_procs; i++) {
            MPI_Recv(msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
            printf("Proc %d: %s \n", status.MPI_SOURCE, msg);
        }
    }
    MPI_Finalize();
}
```





# TIPOS DE COMUNICAÇÃO

# COMUNICAÇÃO BLOQUEANTE VS. NÃO BLOQUEANTE

Durante uma comunicação usamos um buffer de envio.

**Comunicações bloqueantes:** Só executam a próxima instrução quando o **buffer** puder ser usado novamente.

- Significa que o buffer está disponível porque o MPI copiou para outro lugar ou porque a mensagem já foi entregue ao destino.

**Comunicação não-bloqueante:** Executam a próxima instrução, mesmo que o **buffer** ainda não possa ser reutilizado.

- Se o programador usar ou modificar o buffer de envio, não há garantias de corretude do código.

Nos dois casos a comunicação fica completa quando num momento posterior o processo destinatário recebe os dados.

**Não existe garantia que após desbloquear os dados tenham sido entregues.**

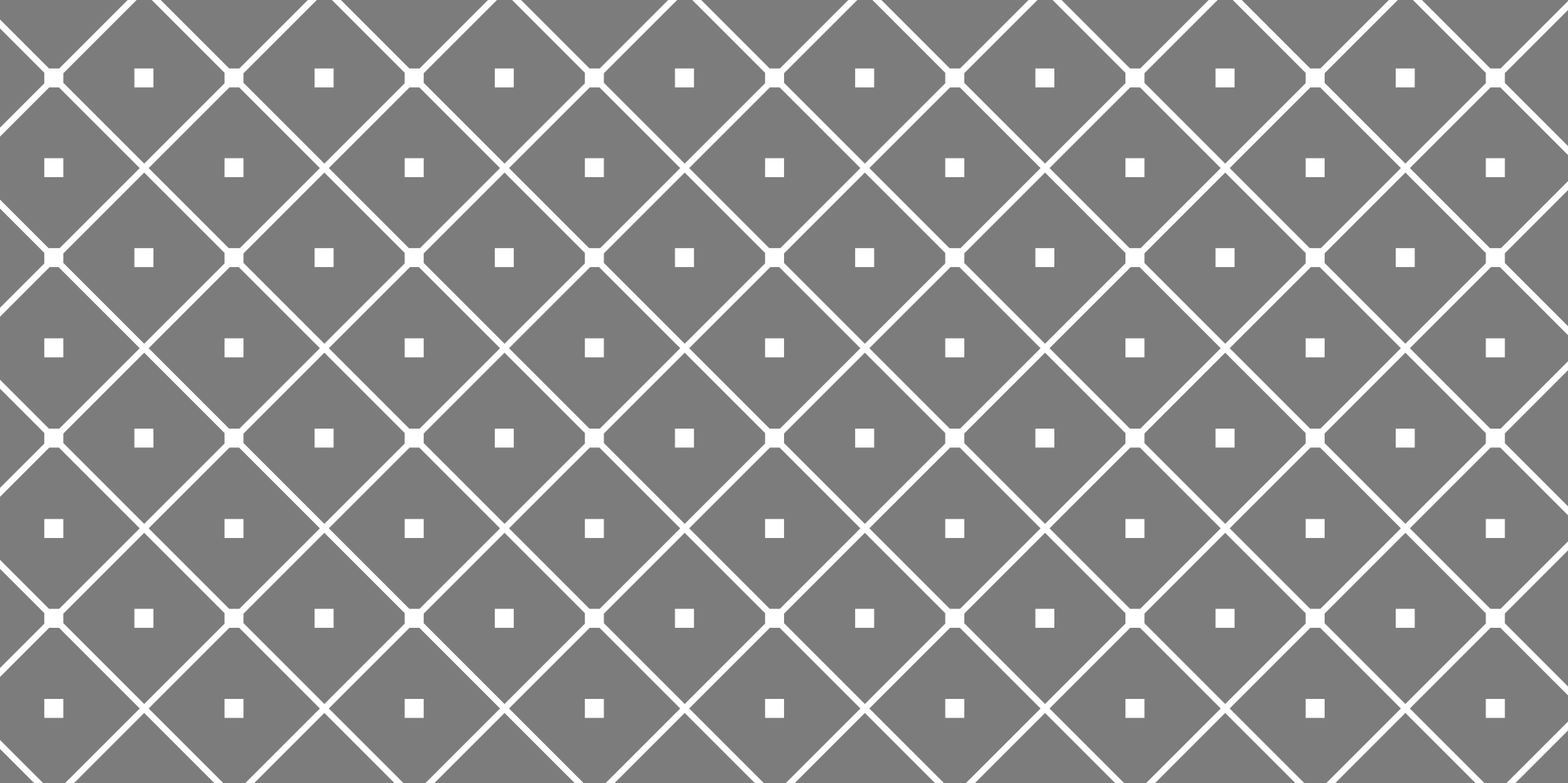
# COMUNICAÇÃO SÍNCRONA VS. ASSÍNCRONA

**Comunicações síncronas:** Não executam a próxima instrução até que seja garantido que os dados foram **entregues**.

**Comunicações assíncronas:** Não garantem o momento que os dados serão **entregues**. Dependendo da política de bloqueante/não bloqueante irá executar a próxima instrução.

# MODOS DE COMUNICAÇÃO

	Síncrono	Assíncrono
Bloqueante	★ <b>Ssend (Synchronous)</b> Rsend (Ready) <div>★ <b>Send (Sync/Buffered)</b></div>	Bsend (Buffered)
Não Bloqueante	ISsend (Immediate+Synchronous) IRsend (Immediate + Ready)	★ <b>Isend (Immediate)</b> IBsend (Immediate+Buffered)



# COMUNICAÇÃO BLOQUEANTE

# MODOS DE COMUNICAÇÃO

Em MPI existem diferentes modos de comunicação para envio de mensagens:

- **Synchronous:** `MPI_Ssend()`
- **Buffered:** `MPI_Bsend()`
- **Standard:** `MPI_Send()`

Independentemente do modo de envio, a recepção é sempre feita através da chamada `MPI_Recv()`.

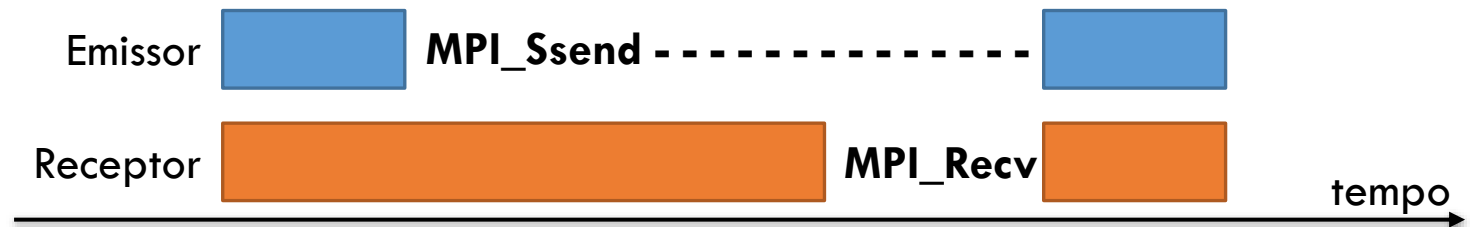
# MODOS DE COMUNICAÇÃO

Em qualquer um dos modos a ordem das mensagens é sempre preservada:

- O processo A envia N mensagens para o processo B fazendo N chamadas a qualquer uma das funções `MPI_Send()`.
- O processo B faz N chamadas a `MPI_Recv()` para receber as N mensagens.
- O ambiente de execução garante que a 1ª chamada a `MPI_Send()` é emparelhada com a 1ª chamada a `MPI_Recv()`, a 2ª chamada a `MPI_Send()` é emparelhada com a 2ª chamada a `MPI_Recv()`, e assim sucessivamente.

# SYNCHRONOUS SEND

```
MPI_Ssend(void *buf, int count, MPI_Datatype datatype,  
           int dest, int tag, MPI_Comm comm)
```



Só quando o processo receptor confirmar que está pronto a receber é que o envio acontece. Até lá o processo emissor fica à espera.

Este tipo de comunicação só deve ser utilizado quando o processo emissor necessita de garantir a recepção antes de continuar a sua execução.

Este método de comunicação pode ser útil para certas situações. No entanto, ele pode atrasar bastante a aplicação, pois enquanto o processo receptor não recebe a mensagem, o processo emissor poderia estar a executar trabalho útil.



# BUFFERED SEND

```
MPI_Bsend(void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm)
```

A mensagem é copiada para um buffer local do programa e só depois enviada. O processo emissor não fica dependente da sincronização com o processo receptor, e após a cópia dos dados para o buffer pode continuar a sua execução.



Tem a vantagem de não requerer sincronização, mas o inconveniente de se ter de definir explicitamente um buffer associado ao programa.

# BUFFERED SEND

`MPI_Buffer_attach(void *buf, int size)`

**MPI\_Buffer\_attach()** informa o ambiente de execução do MPI que o espaço de tamanho **size** bytes a partir do endereço **buf** pode ser usado para buffering local de mensagens.

`MPI_Buffer_detach(void **buf, int *size)`

**MPI\_Buffer\_detach()** informa o ambiente de execução do MPI que o atual buffer local de mensagens não deve ser mais utilizado. Se existirem mensagens pendentes no buffer, a função só retorna quando todas elas forem entregues.

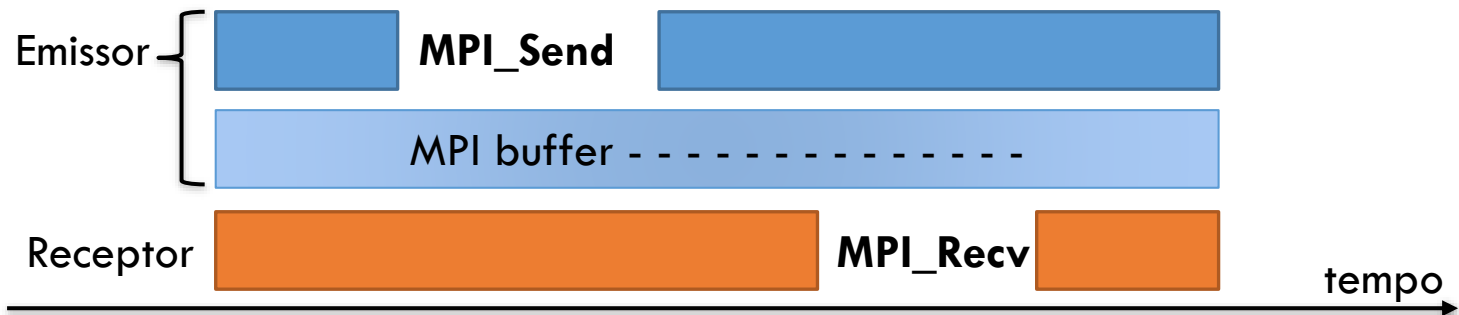
Em cada instante da execução só pode existir um único buffer associado a cada processo.

A função `MPI_Buffer_detach()` não liberta a memória associada ao buffer, para tal é necessário invocar explicitamente a função `free()` do sistema.

# BUFFERED VS. STANDARD SEND

```
MPI_Send(void *buf, int count, MPI_Datatype datatype,  
         int dest, int tag, MPI_Comm comm)
```

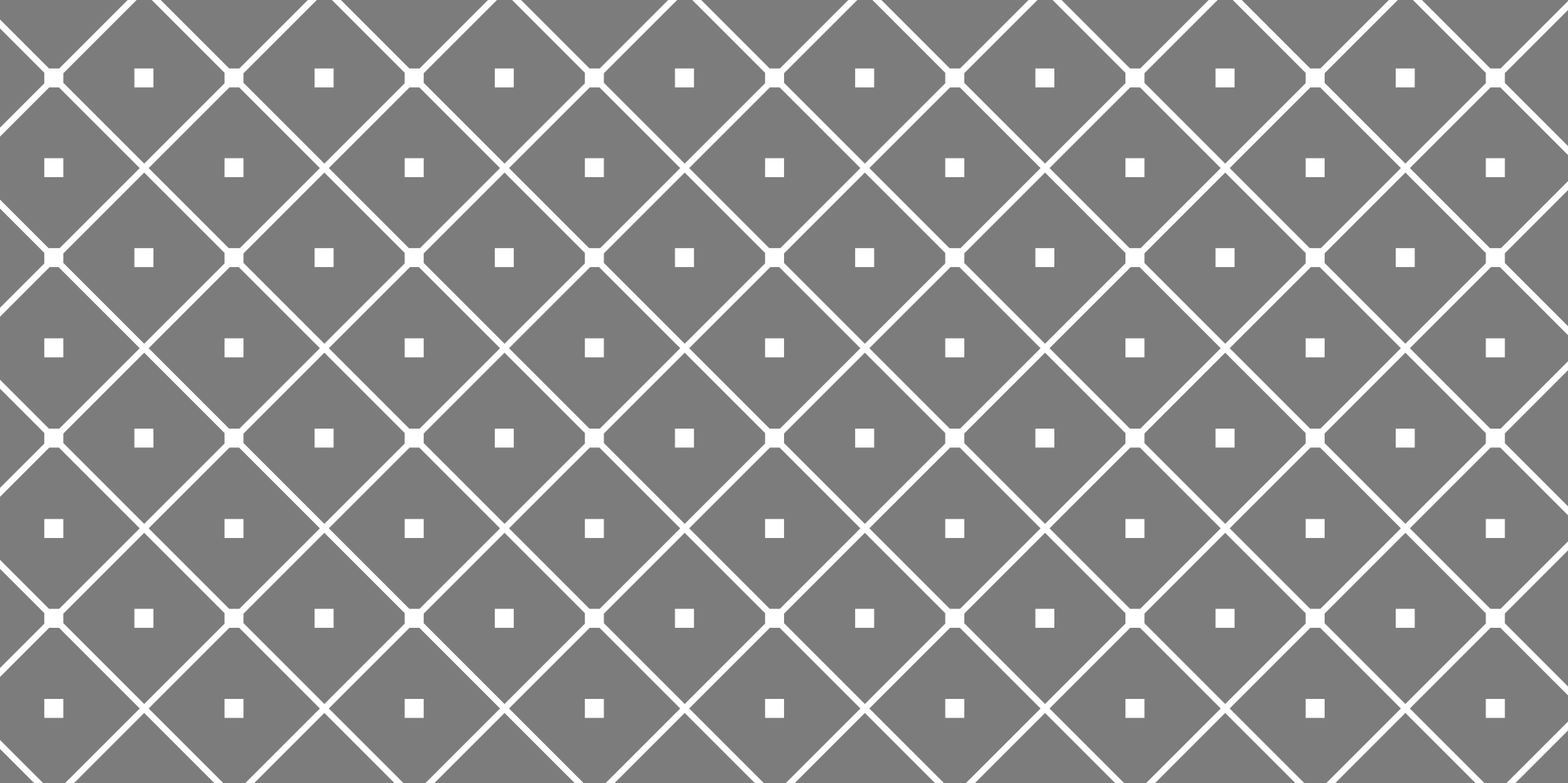
Termina assim que a mensagem é enviada, o que não significa que tenha sido entregue ao processo receptor. A mensagem pode ficar pendente no ambiente de execução durante algum tempo (depende da implementação do MPI).



Tipicamente, as implementações fazem buffering de mensagens pequenas e sincronizam nas grandes. Para escrever código portátil, o programador não deve assumir que o envio termina antes nem depois de começar a recepção.

# WELCOME! (MPI\_WELCOME.C)

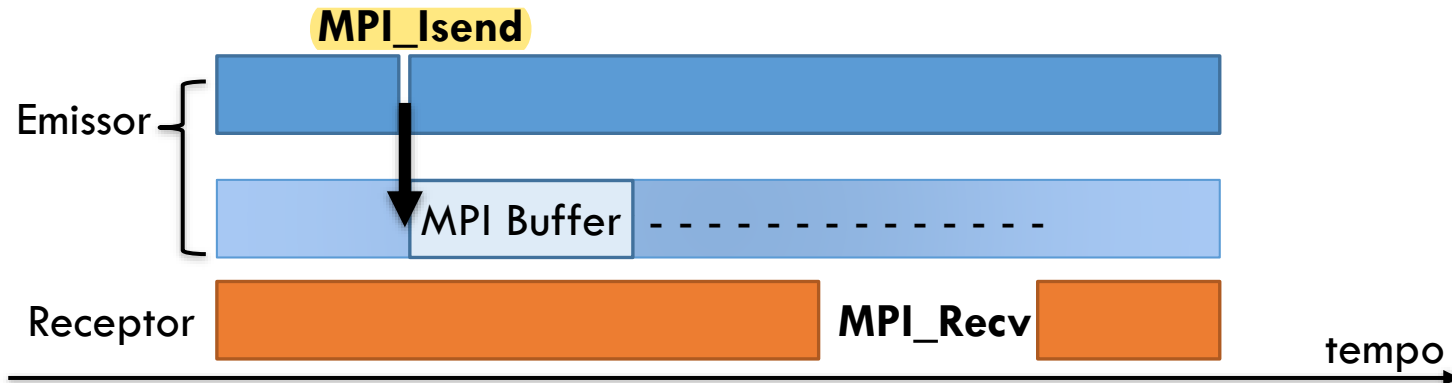
```
main(int argc, char **argv) {
    int buf_size; char *local_buf; ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    buf_size = BUF_SIZE; local_buf = (char *) malloc(buf_size);
    MPI_Buffer_attach(local_buf, buf_size);
    sprintf(msg, "Welcome!");
    for (i = 0; i < n_procs; i++)
        if (my_rank != i)
            MPI_Bsend(msg, strlen(msg) + 1, MPI_CHAR, i, STD_TAG, MPI_COMM_WORLD);
    for (i = 0; i < n_procs; i++)
        if (my_rank != i) {
            sprintf(msg, "Argh!");
            MPI_Recv(msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            printf("Proc %d->%d: %s \n", status.MPI_SOURCE, my_rank, msg);
        }
    MPI_Buffer_detach(&local_buf, &buf_size);
    free(local_buf);
    MPI_Finalize();
}
```



# COMUNICAÇÃO NÃO BLOQUEANTES

# ENVIO E RECEPÇÃO NÃO BLOQUEANTE DE MENSAGENS

```
MPI_Isend(void *buf, int count, MPI_Datatype datatype,  
          int dest, int tag, MPI_Comm comm, MPI_Request *req)
```



```
MPI_Irecv(void *buf, int count, MPI_Datatype datatype,  
          int source, int tag, MPI_Comm comm, MPI_Request *req)
```

Ambas as funções devolvem em **req** o identificador que permite a posterior verificação do sucesso da comunicação.

# VERIFICAR O SUCESSO DE UMA COMUNICAÇÃO NÃO BLOQUEANTE

```
MPI_Iprobe(int source, int tag, MPI_Comm comm,  
           int *flag, MPI_status *status)
```

**MPI\_Iprobe()** testa a chegada de uma mensagem associada com **source**, **tag** e **comm** sem contudo proceder à sua recepção.

Retorna em **flag** o valor lógico que indica a chegada de alguma mensagem, e em caso positivo retorna em **status** informação sobre a mesma.

A recepção deverá ser posteriormente feita com uma função de recepção.

# VERIFICAR O SUCESSO DE UMA COMUNICAÇÃO NÃO BLOQUEANTE

```
MPI_Wait(MPI_Request *req, MPI_Status *status)
```

**MPI\_Wait()** bloqueia até que a comunicação identificada por **req** suceda. Retorna em **status** informação relativa à mensagem.

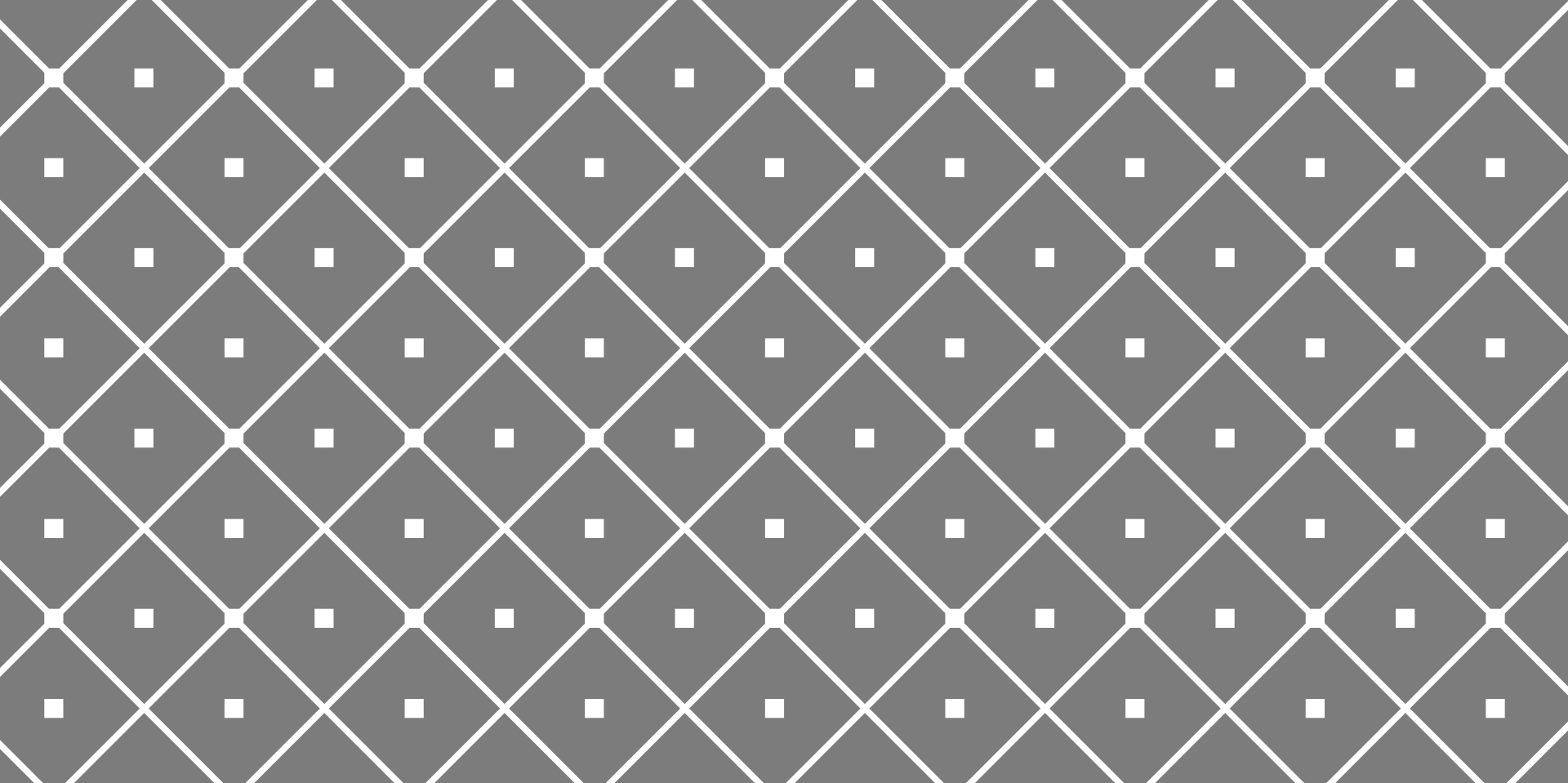
```
MPI_Test(MPI_Request *req, int *flag, MPI_Status  
          *status)
```

**MPI\_Test()** testa se a comunicação identificada por **req** sucedeu. Retorna em **flag** o valor lógico que indica o sucesso da comunicação, e em caso positivo retorna em **status** informação relativa à mensagem.



# HELLO! (MPI\_HELLO.C)

```
main(int argc, char **argv) {
    char recv_msg[100]; MPI_Request req[100];
    ...
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    sprintf(msg, "Hello!");
    for (i = 0; i < n_procs; i++)
        if (my_rank != i) {
            MPI_Irecv(recv_msg, 100, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &(req[i]));
            MPI_Isend(msg, strlen(msg)+1, MPI_CHAR, i, STD_TAG, MPI_COMM_WORLD,
&(req[i+n_procs]));
        }
    for (i = 0; i < n_procs; i++)
        if (my_rank != i) {
            sprintf(recv_msg, "Argh!");
            MPI_Wait(&(req[i + n_procs]), &status);
            MPI_Wait(&(req[i]), &status);
            printf("Proc %d->%d: %s \n", status.MPI_SOURCE, my_rank, recv_msg);
        }
    MPI_Finalize();
}
```



# ENVIO E RECEPÇÃO SIMULTÂNEA DE MENSAGENS

# ENVIO E RECEPÇÃO SIMULTÂNEA DE MENSAGENS

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
             int dest, int sendtag, void *recvbuf, int recvcount,  
             MPI_Datatype recvtype, int source, int recvtag,  
             MPI_Comm comm, MPI_Status *status)
```

**MPI\_Sendrecv()** permite o envio e recepção simultânea de mensagens. É útil para quando se pretende utilizar comunicações circulares sobre um conjunto de processos, pois permite evitar o problema de ordenar corretamente as comunicações de modo a não ocorrerem situações de deadlock.

**sendbuf** é o endereço inicial dos dados a enviar.

**sendcount** é o número de elementos do tipo **sendtype** a enviar.

**sendtype** é o tipo de dados a enviar.

**dest** é a posição do processo no comunicador **comm** a quem se destina a mensagem.

**sendtag** é uma marca que identifica a mensagem a enviar.

# ENVIO E RECEPÇÃO SIMULTÂNEA DE MENSAGENS

```
MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,  
             int dest, int sendtag, void *recvbuf, int recvcount,  
             MPI_Datatype recvtype, int source, int recvtag,  
             MPI_Comm comm, MPI_Status *status)
```

**recvbuf** é o endereço onde devem ser colocados os dados recebidos.

**recvcount** é o número máximo de elementos do tipo **recvtype** a receber.

**recvtype** é o tipo de dados a receber.

**source** é a posição do processo no comunicador **comm** de quem se pretende receber a mensagem.

**recvtag** é a marca que identifica a mensagem que se pretende receber.

**comm** é o comunicador dos processos envolvidos na comunicação.

**status** devolve informação sobre o processo emissor.

# ENVIO E RECEPÇÃO SIMULTÂNEA DE MENSAGENS

Os buffers de envio **sendbuf** e de recepção **recvbuf** devem ser necessariamente diferentes.

As marcas **sendtag** e **recvtag**, os tamanhos **sendcount** e **recvcount**, e os tipos de dados **sendtype** e **recvtype** podem ser diferentes.

Uma mensagem enviada por uma comunicação **MPI\_Sendrecv()** pode ser recebida por qualquer outra comunicação usual de recepção de mensagens.

Uma mensagem recebida por uma comunicação **MPI\_Sendrecv()** pode ter sido enviada por qualquer outra comunicação usual de envio de mensagens.

# ENVIO E RECEPÇÃO SIMULTÂNEA DE MENSAGENS

```
MPI_Sendrecv_replace(void *buf, int count,  
MPI_Datatype datatype, int dest, int sendtag, int source,  
int recvtag, MPI_Comm comm, MPI_Status *status)
```

**MPI\_Sendrecv\_replace()** permite o envio e recepção simultânea de mensagens utilizando o mesmo buffer para o envio e para a recepção. No final da comunicação, a mensagem a enviar é substituída pela mensagem recebida.

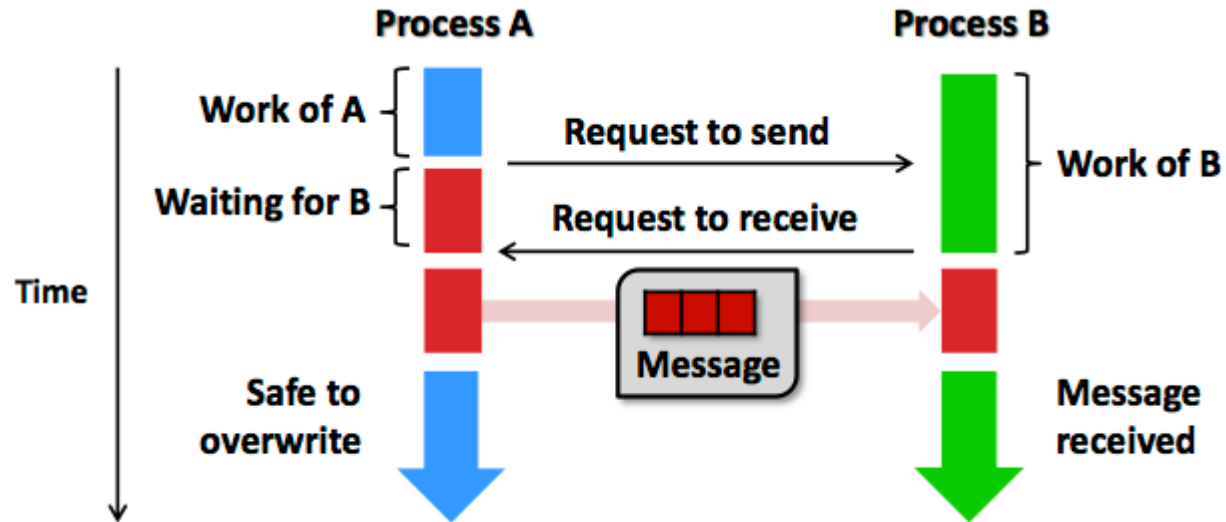
O buffer **buf**, o tamanho **count** e o tipo de dados **datatype** são utilizados para definir tanto a mensagem a enviar como a mensagem a receber.

Uma mensagem enviada por uma comunicação **MPI\_Sendrecv\_replace()** pode ser recebida por qualquer outra comunicação usual de recepção de mensagens.

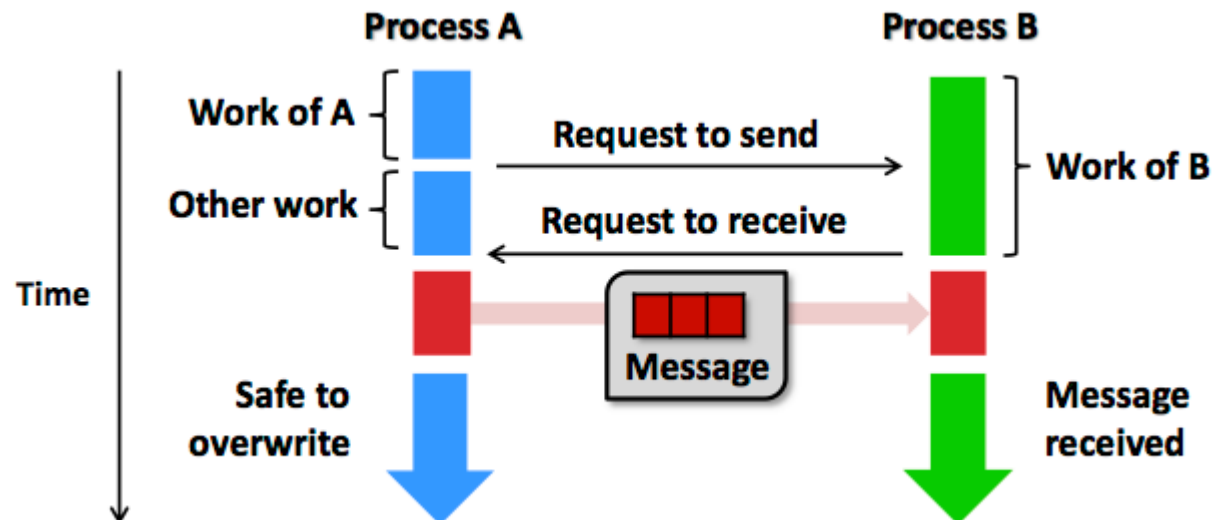
Uma mensagem recebida por uma comunicação **MPI\_Sendrecv\_replace()** pode ter sido enviada por qualquer outra comunicação usual de envio de mensagens.

# PRINCIPAIS TIPOS DE MENSAGEM

**Ssend**  
Síncrono  
Bloqueante



**Lsend**  
Assíncrono  
Não Bloqueante



# TIPOS DE RECEBIMENTO

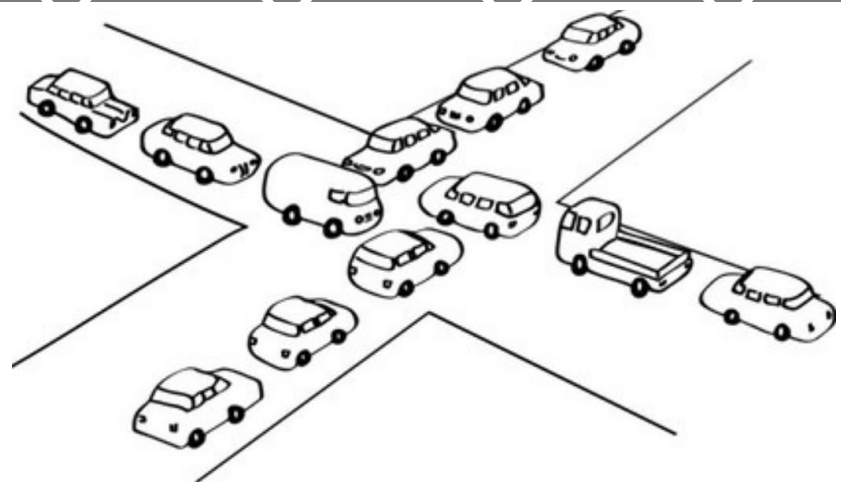
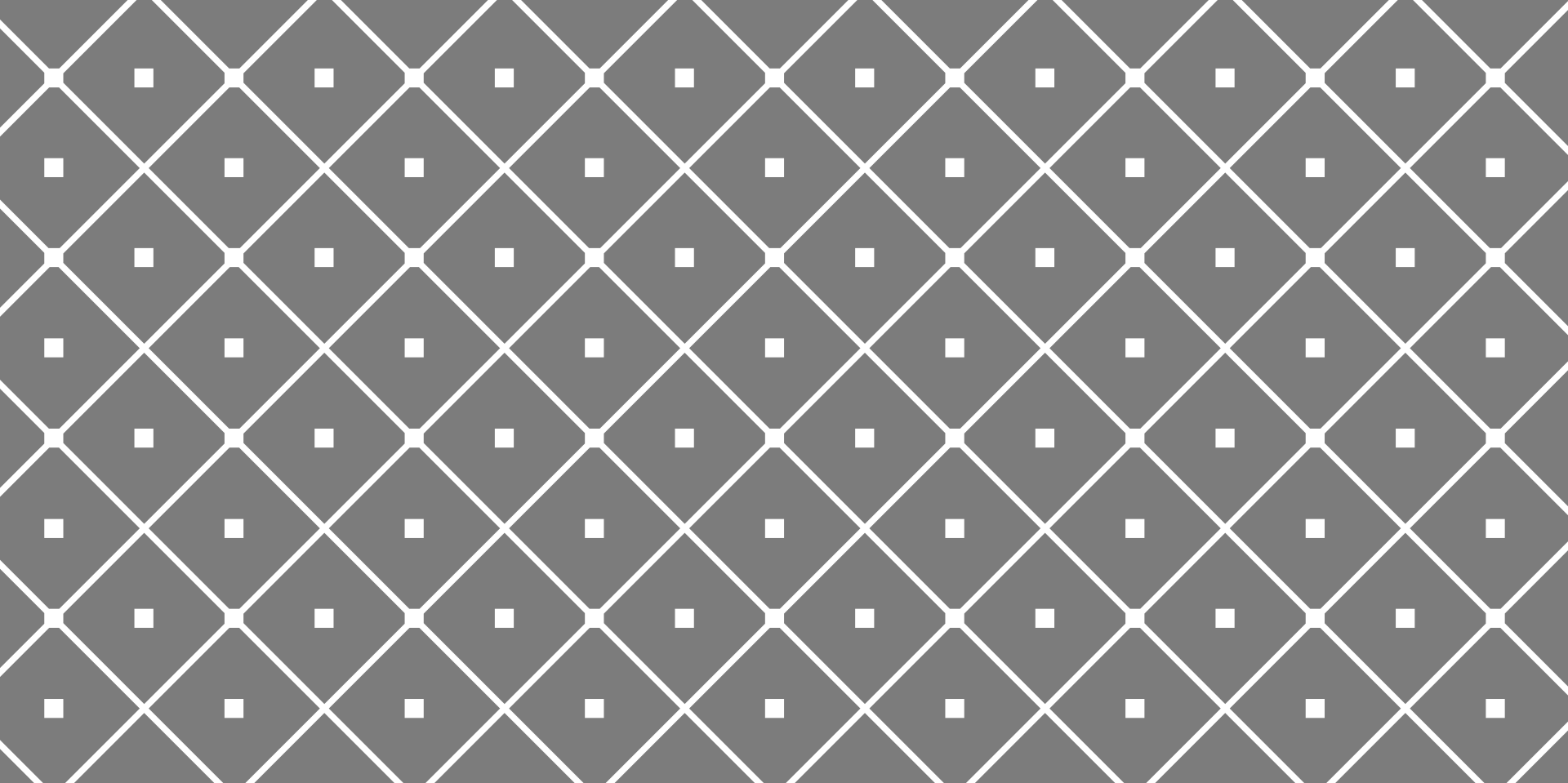
De maneira parecida com o SEND o RECV tem modos de operação:

**MPI\_Recv – Bloqueante e Síncrono:** só executa a próxima instrução quando o buffer estiver com os dados.

**MPI\_Irecv – Não Bloqueante e Não-Síncrono:** executará a próxima instrução em seguida, porém não devemos utilizar os dados do buffer até que asseguremos que a recepção foi sucedida.







# DEADLOCK

# DEADLOCKS

Suponha que dois processos precisem trocar dados

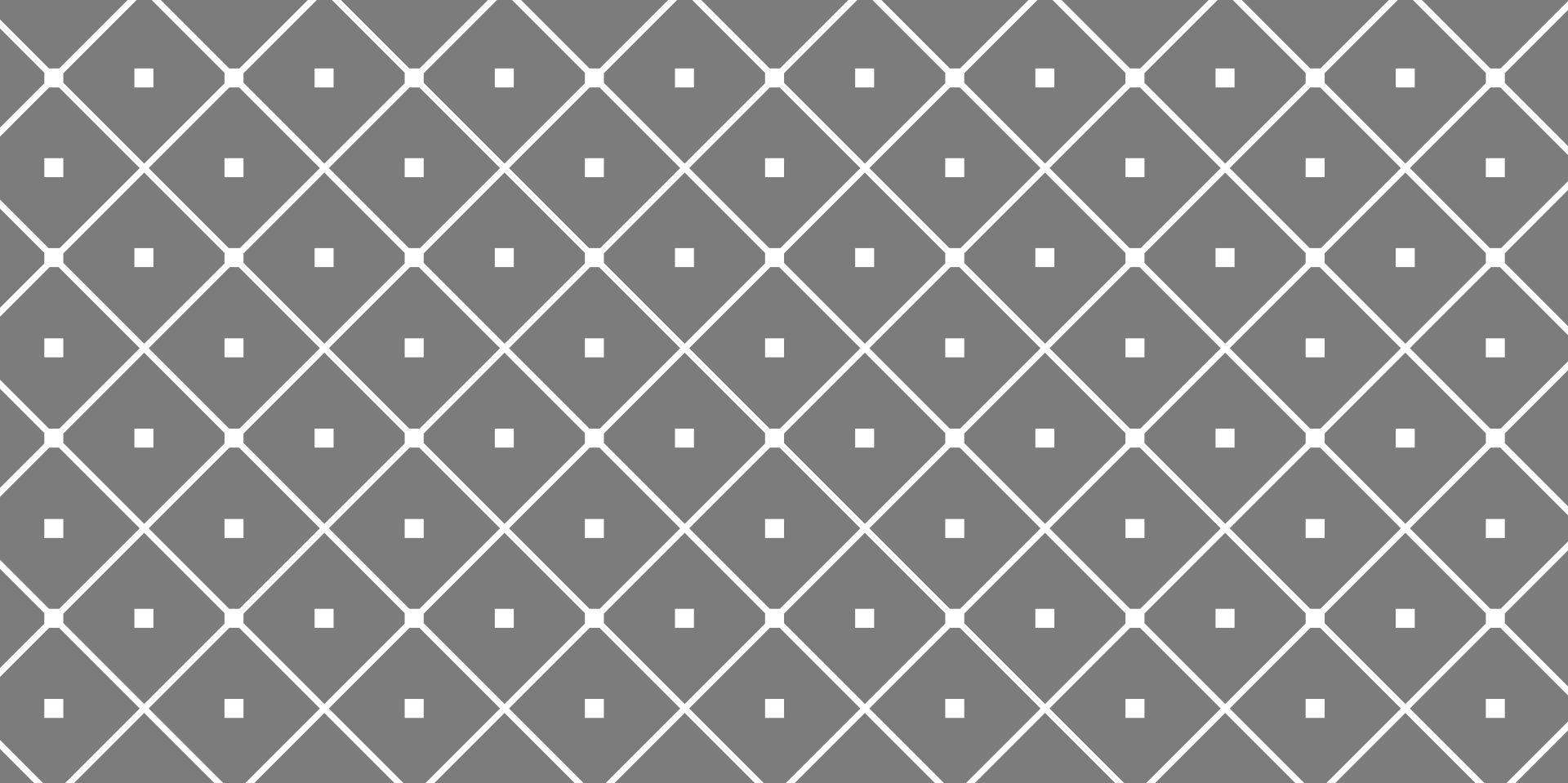
Considere o seguinte pseudocódigo, que pretende trocar dados entre os processos 0 e 1:

```
sync_send(destinatário = outro);  
receive(origem = outro);
```

Imagine que os dois processos executam esse código.

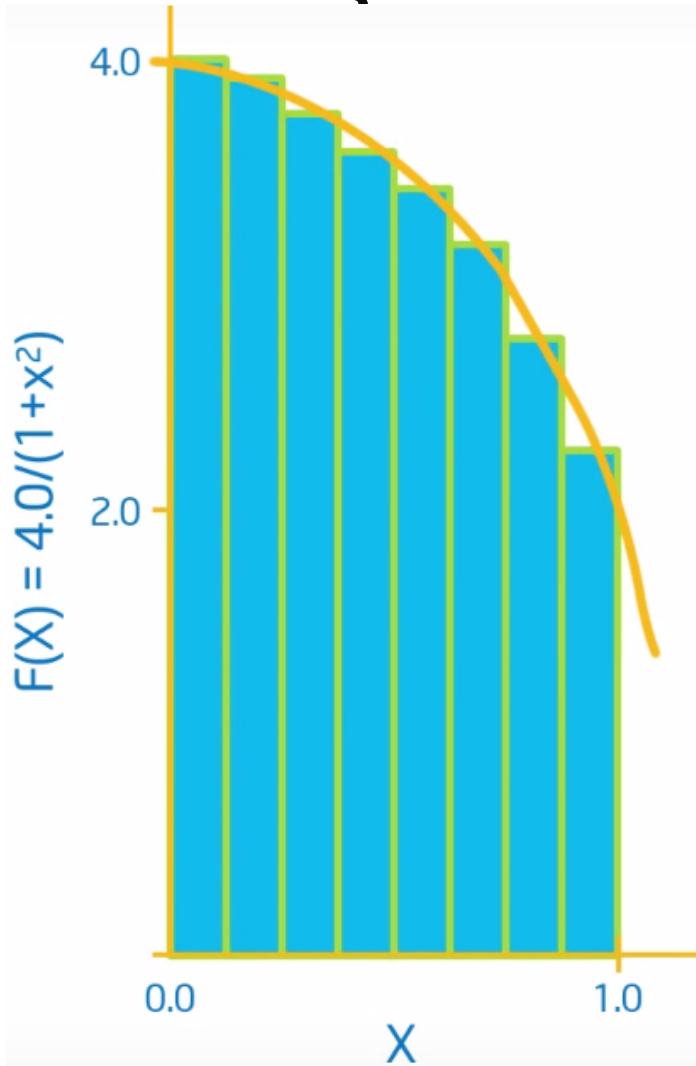
Ambos emitem a chamada de envio... e não podem continuar, porque ambos estão esperando que o outro emita a chamada de recepção correspondente à chamada de envio.

Isso é conhecido como deadlock.



# EXERCÍCIO

# EXERCÍCIOS: INTEGRAÇÃO NUMÉRICA



Matematicamente, sabemos que:

$$\int_0^1 \frac{4.0}{1+x^2} dx = \pi$$

Podemos aproximar essa integral como a soma de retângulos:

$$\sum_{i=0}^n F(x_i) \Delta x \cong \pi$$

Onde cada retângulo tem largura  $\Delta x$  e altura  $F(x_i)$  no meio do intervalo  $i$ .

# EXERCÍCIOS: PROGRAMA PI SERIAL

```
static long num_steps = 100000;
double step;
int main () {
    int i; double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0;i< num_steps; i++){
        x = (i + 0.5) * step; // Largura do retângulo
        sum = sum + 4.0 / (1.0 + x*x); // Sum += Área do retângulo
    }
    pi = step * sum;
}
```

# EXERCÍCIO

Crie uma versão paralela usando MPI do programa pi usando troca de mensagens para agregar a computação final.