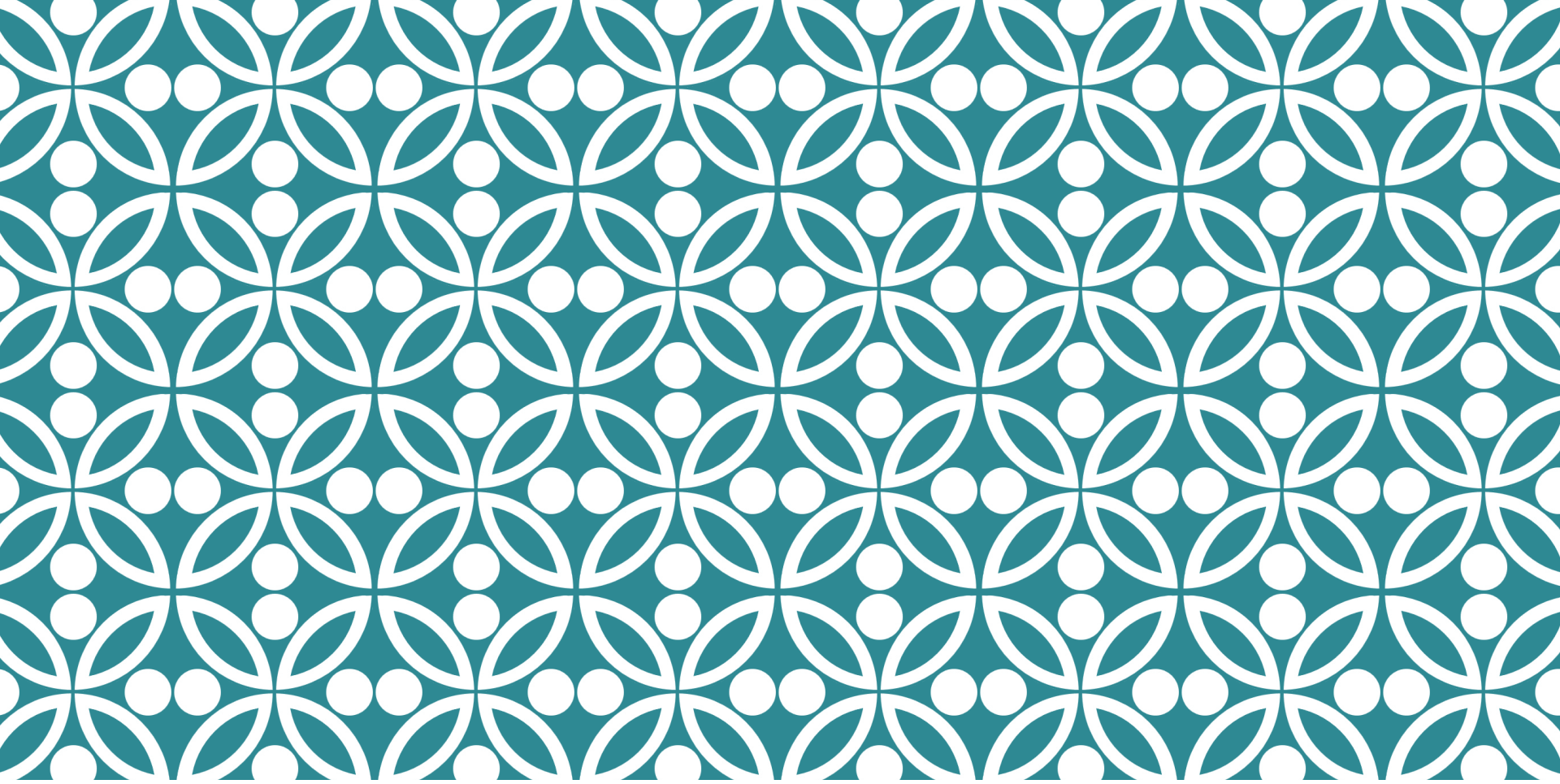


PROGRAMAÇÃO PARALELA
MPI 01 – INTRODUÇÃO

Marco A. Zanata Alves

Slides baseados nos slides de Ricardo Rocha – DCC FCUP (2009/2010)
Programação Paralela e Distribuída

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:

- LAM - http://www.lam-mpi.org
- MPICH - http://www.mcs.anl.gov/mpi/mpich
- CHIMP - http://www.epcc.ed.ac.uk/chimp

Propostas de extensão foram entretanto 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 portável.

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.

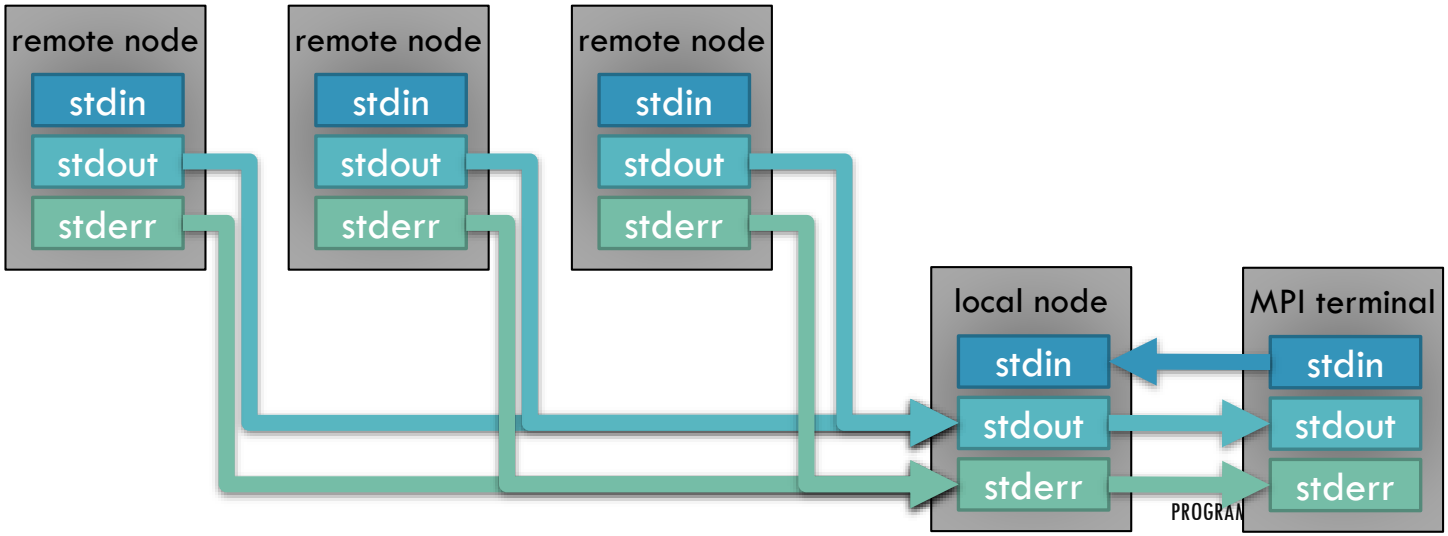
ESTRUTURA BASE DE UM PROGRAMA MPI

```
// 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
...
}
```

STANDARD I/O

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.

O standard output e o standard error são redirecionados em todos os nós para o terminal onde a execução é iniciada.



COMPILAÇÃO E EXECUÇÃO DE PROGRAMAS

De forma a facilitar o processo de compilação, 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)

O comando mpirun permite iniciar a execução distribuída de um dado programa MPI. Para tal é necessário 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, onde deve especificar-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 lance aplicações em diferentes máquinas, é necessário que o usuário possua livre acesso a estas.

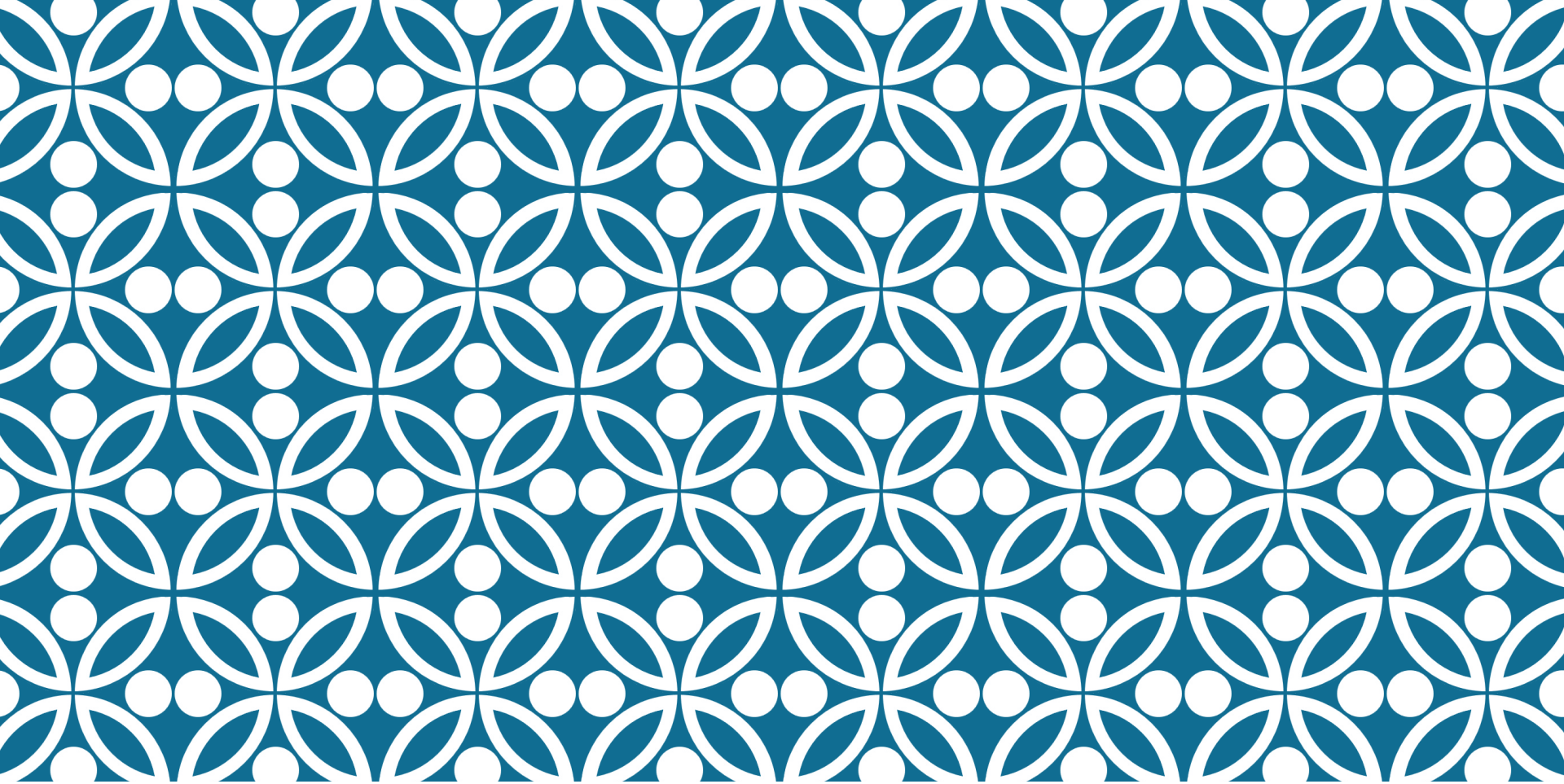
Um esquema para isso pode ser o uso de chaves SSH.

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.

- **Na UFPR podemos considerar que o (network file system) NFS irá solucionar essa questão automaticamente para nós.**

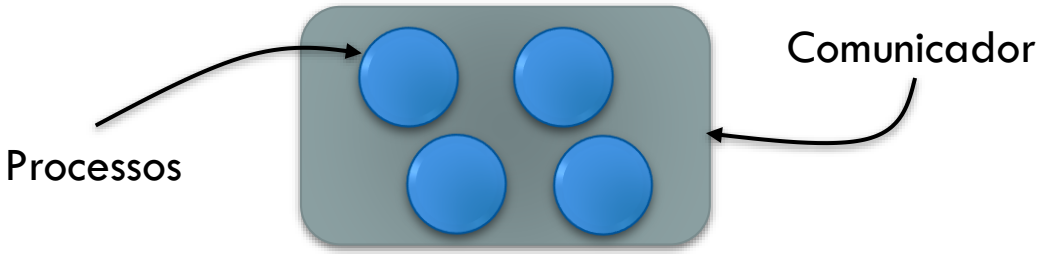


COMUNICADORES

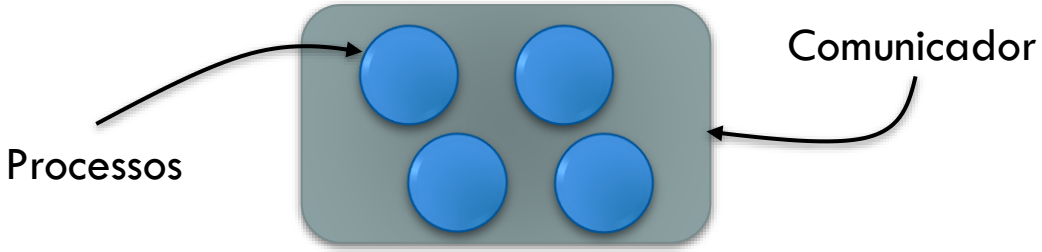
COMUNICADORES

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.

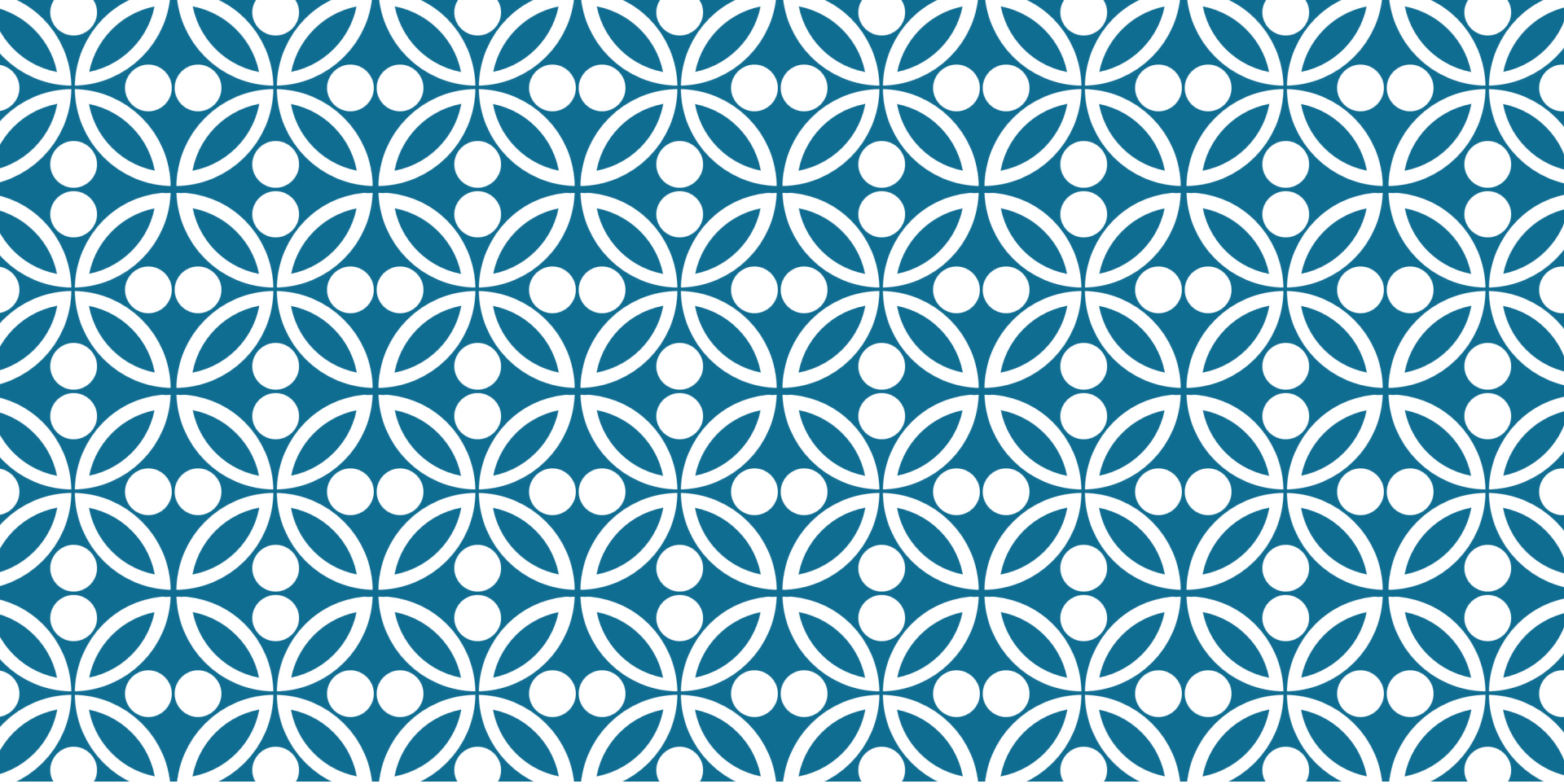
INFORMAÇÃO RELATIVA A UM COMUNICADOR

```
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

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();
}
```

MODOS DE COMUNICAÇÃO

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

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

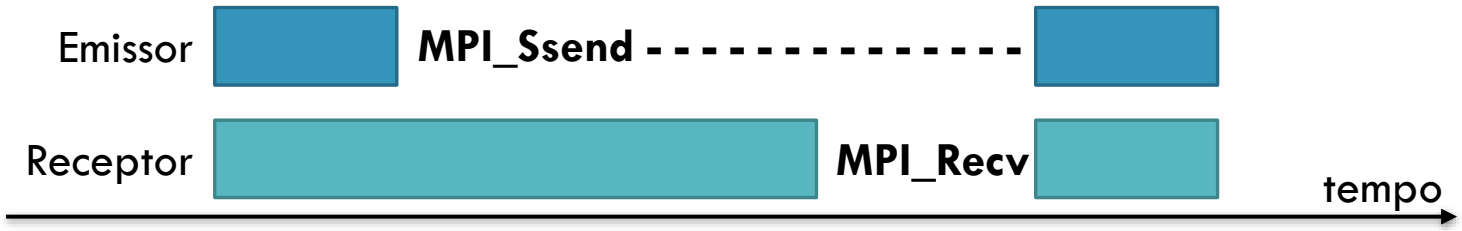
Independentemente do modo de envio, a recepção é sempre feita através da chamada MPI_Recv().

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 1 a chamada a MPI_Send() é emparelhada com a 1 a chamada a MPI_Recv(), a 2 a chamada a MPI_Send() é emparelhada com a 2 a 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.

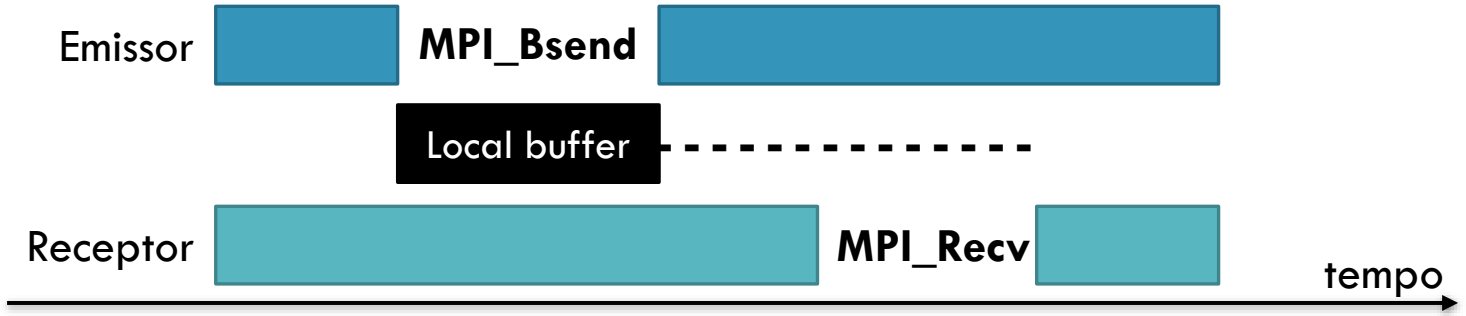
Este tipo de comunicação só deve ser utilizado quando o processo emissor necessita de garantir a recepção antes de continuar a sua execução.

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 pode desde logo continuar a sua execução.

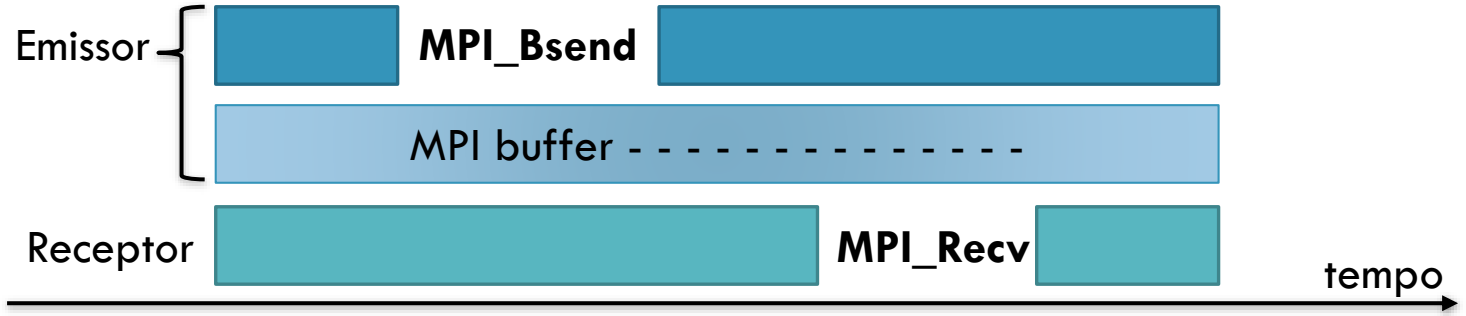


Tem a vantagem de não requerer sincronização, mas o inconveniente de se ter de definir explicitamente um buffer associado ao programa.

BUFFERED VS. STANDARD SEND

```
MPI_Send(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 portável, o programador não deve assumir que o envio termina antes nem depois de começar a recepção.

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)
```

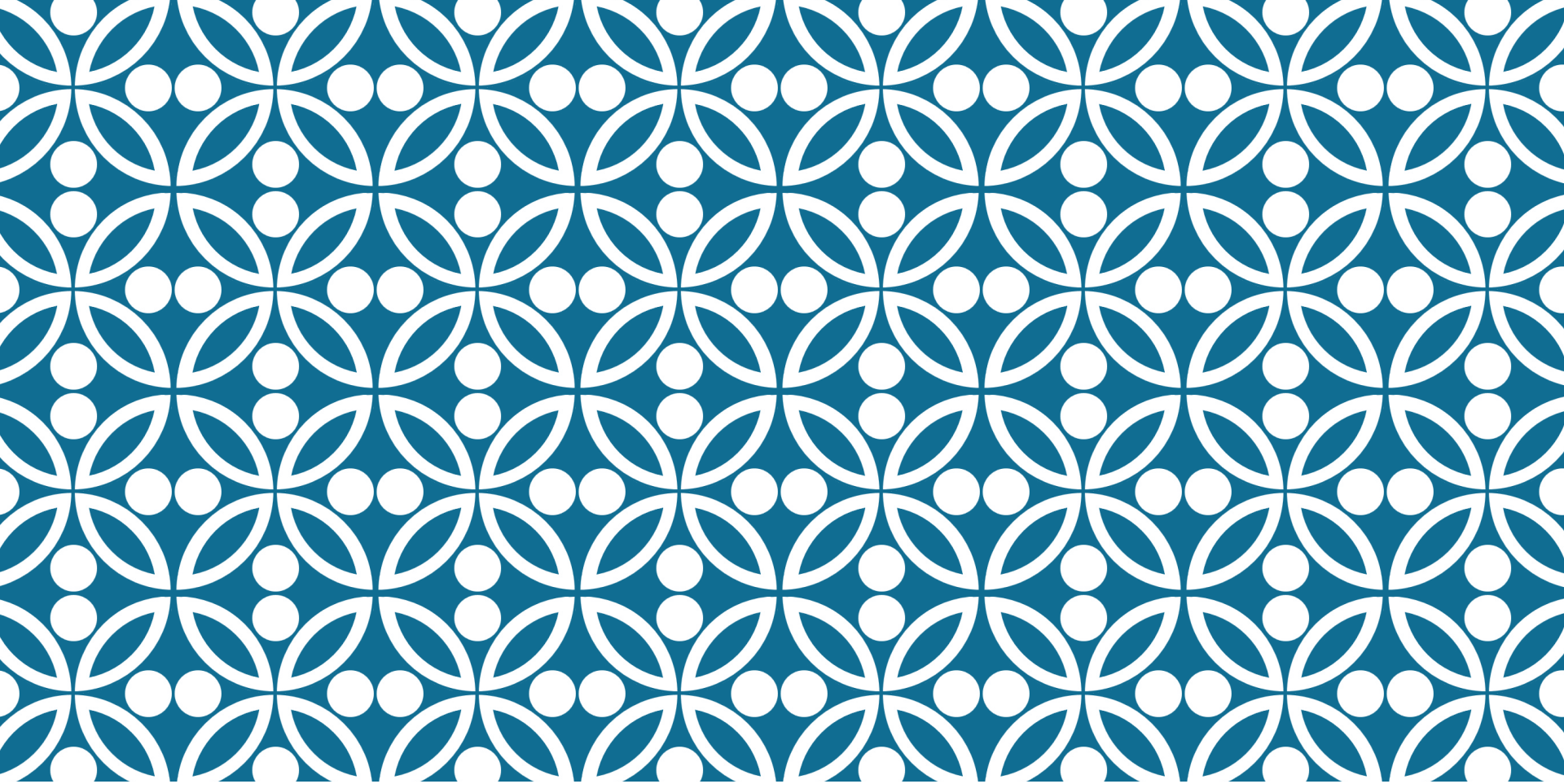
MPI_Buffer_detach() informa o ambiente de execução do MPI que o atual buffer local de mensagens não deve ser mais utilizado. Se existirem mensagens pendentes no buffer, a função só retorna quando todas elas forem entregues.

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.

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();
}
```



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.

senbuf 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

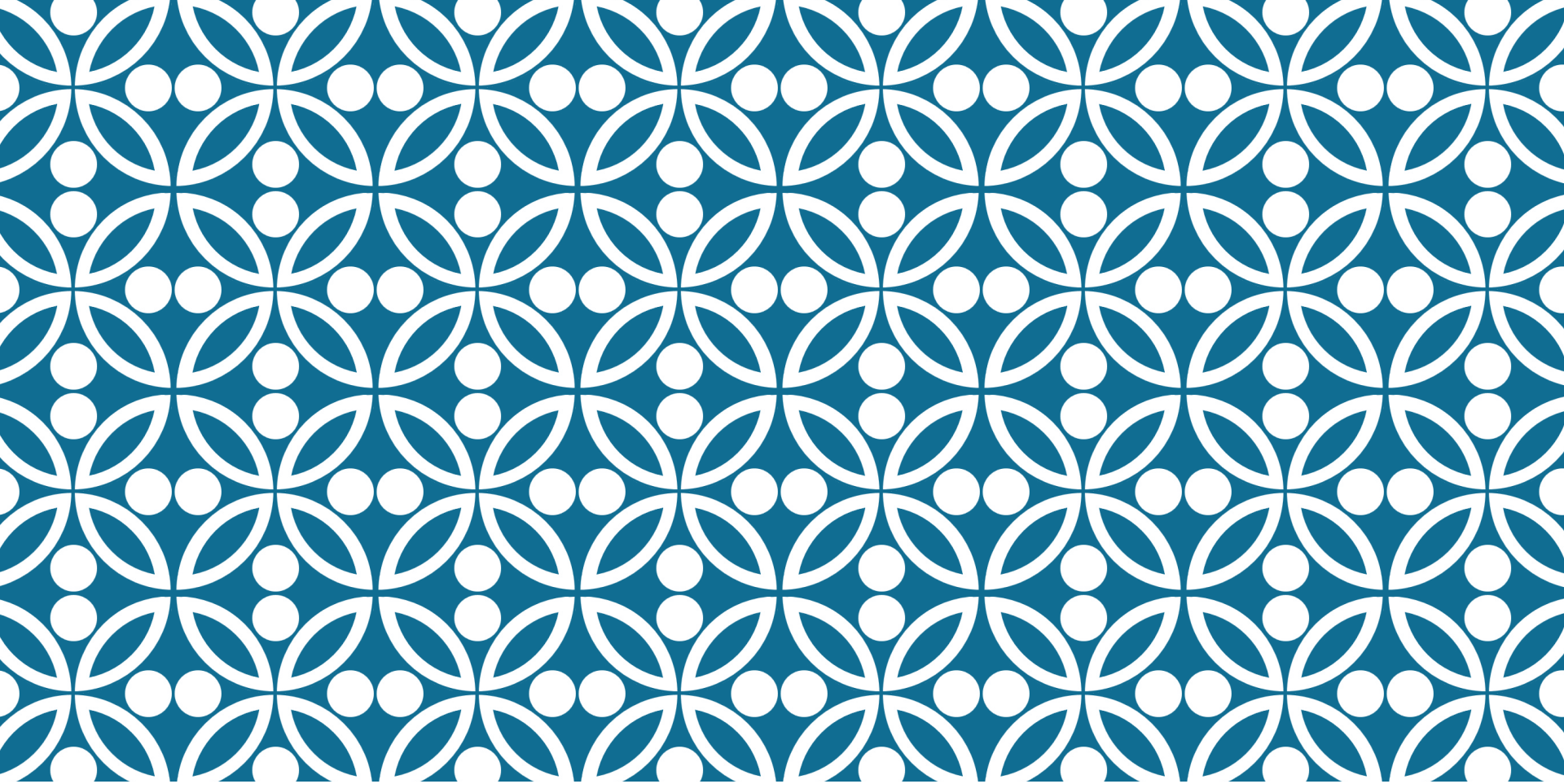
```
MPI_Sendrecv_replace(void *buf, int count,
MPI_Datatype datatype, int dest, int sendtag, int source,
    int recvtag, MPI_Comm comm, MPI_Status *status)
```

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.



COMUNICAÇÕES NÃO BLOQUEANTES

COMUNICAÇÕES NÃO BLOQUEANTES

Uma comunicação diz-se **bloqueante** se a execução é interrompida enquanto a comunicação não sucede. Uma comunicação bloqueante só sucede quando o buffer da mensagem associado à comunicação pode ser reutilizado.

Uma comunicação diz-se **não bloqueante** se a continuação da execução não depende do sucesso da comunicação. O buffer da mensagem associado a uma comunicação não bloqueante não deve, no entanto, ser reutilizado pela aplicação até que a comunicação suceda.

- A ideia das comunicações não bloqueantes é iniciar o envio das mensagens o mais cedo possível, continuar de imediato com a execução, e verificar o mais tarde possível o sucesso das mesmas.
- Uma chamada a uma função não bloqueante apenas anuncia ao ambiente de execução a existência de uma mensagem para ser enviada ou recebida. A função completa de imediato.
- A comunicação fica completa quando num momento posterior o processo toma conhecimento do sucesso da comunicação.

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();
}
```

QUE TIPO DE MENSAGENS DEVO UTILIZAR?

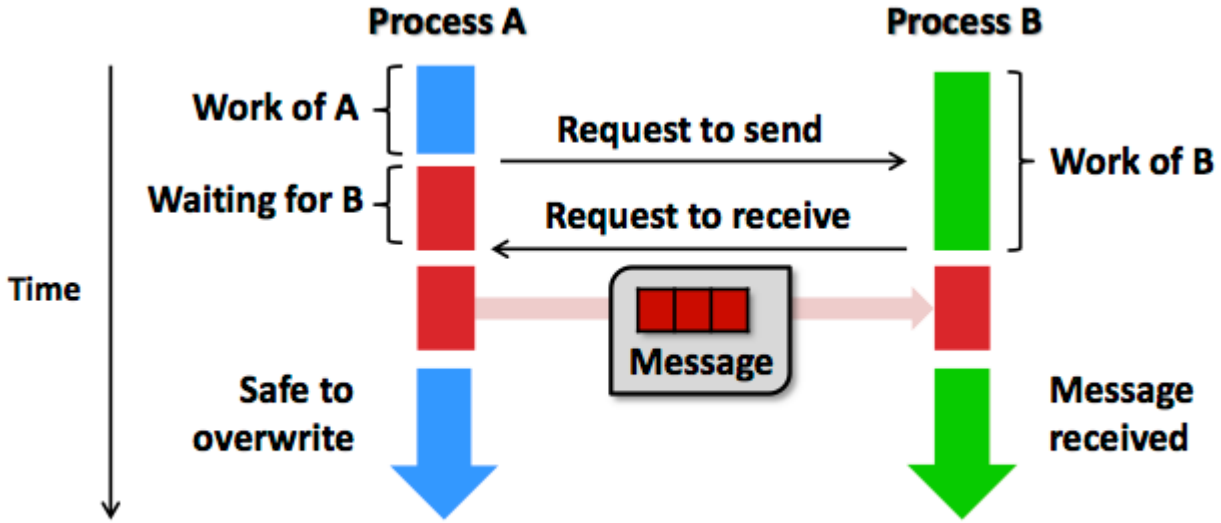
Grande parte dos programadores utiliza as versões standard para envio e recepção de mensagens quando o ambiente de execução possui implementações eficientes dessas funções. No entanto, o uso dessas funções nem sempre garante a portabilidade da aplicação.

Em alternativa, a utilização das versões synchronous e standard não bloqueante é suficiente para construir aplicações robustas e ao mesmo tempo portáveis.

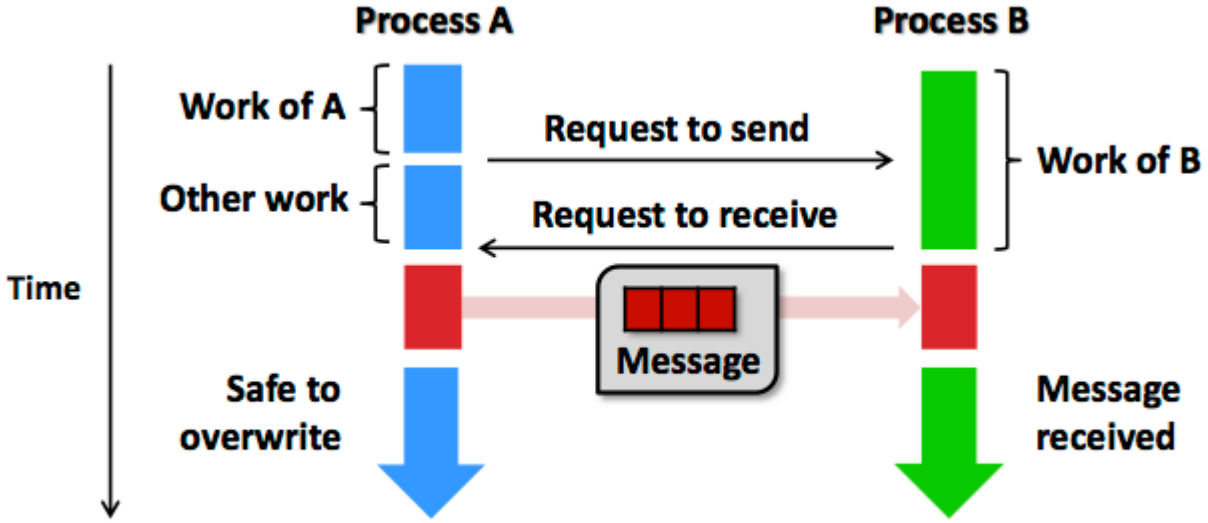
As versões não bloqueantes nem sempre levam a melhores resultados. A sua utilização só deve ser considerada quando existe uma clara e substancial sobreposição da computação.

É perfeitamente possível enviar mensagens com funções não bloqueantes e receber com funções bloqueantes. O contrário é igualmente possível.

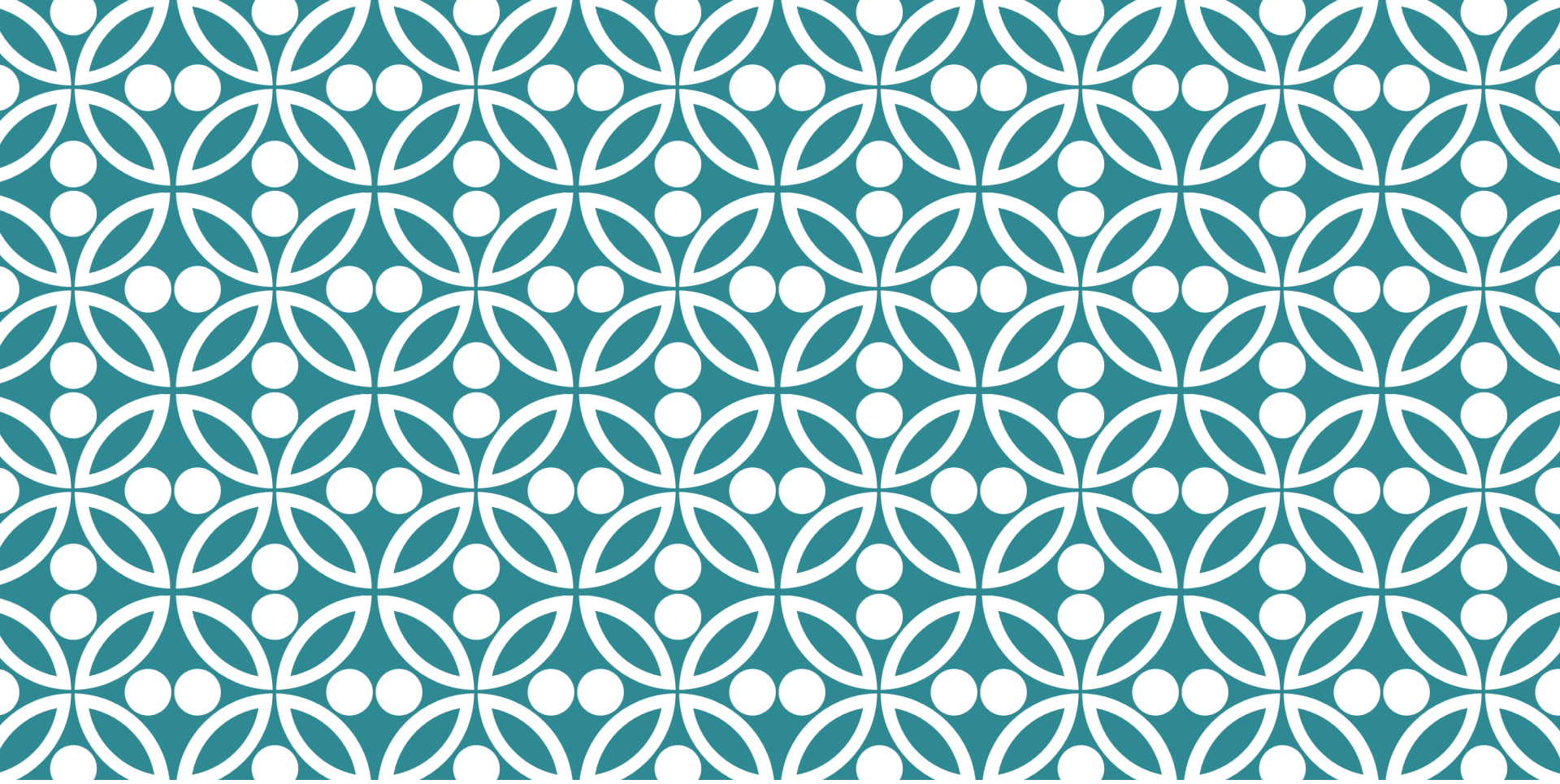
TIPOS DE MENSAGEM



With a blocking send, no other operations can be executed until the communication has completed.



With a nonblocking send, other operations can be executed until the other process responds with a call to a receive function.



PROGRAMAÇÃO PARALELA MPI 02 –DADOS DERIVADOS

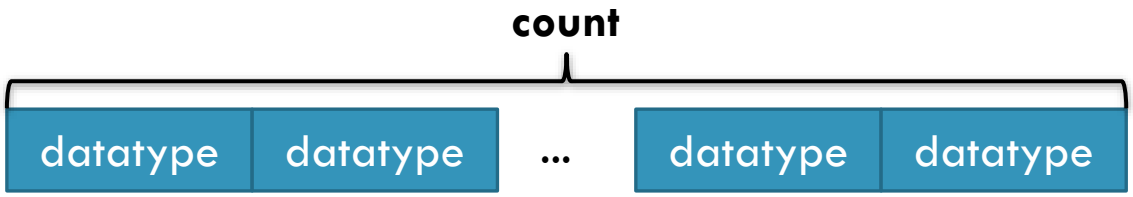
Marco A. Zanata Alves

Slides baseados nos slides de Ricardo Rocha – DCC FCUP (2009/2010)
Programação Paralela e Distribuída

AGRUPAR DADOS PARA COMUNICAÇÃO

Em programação com troca de mensagens, uma heurística natural para maximizar a performance do sistema é minimizar o número de mensagens a trocar.

Por padrão, todas as funções de envio e recepção de mensagens permitem agrupar numa mesma mensagem dados do mesmo tipo guardados em posições contíguas de memória.



Para além desta funcionalidade básica o MPI permite:

- Definir novos tipos de dados que agrupam dados de vários tipos.
- Agrupar dados espalhados na memória
- Empacotar e desempacotar dados para/de um buffer.

TIPOS DERIVADOS

A definição de novos tipos de dados é feita em tempo de execução:

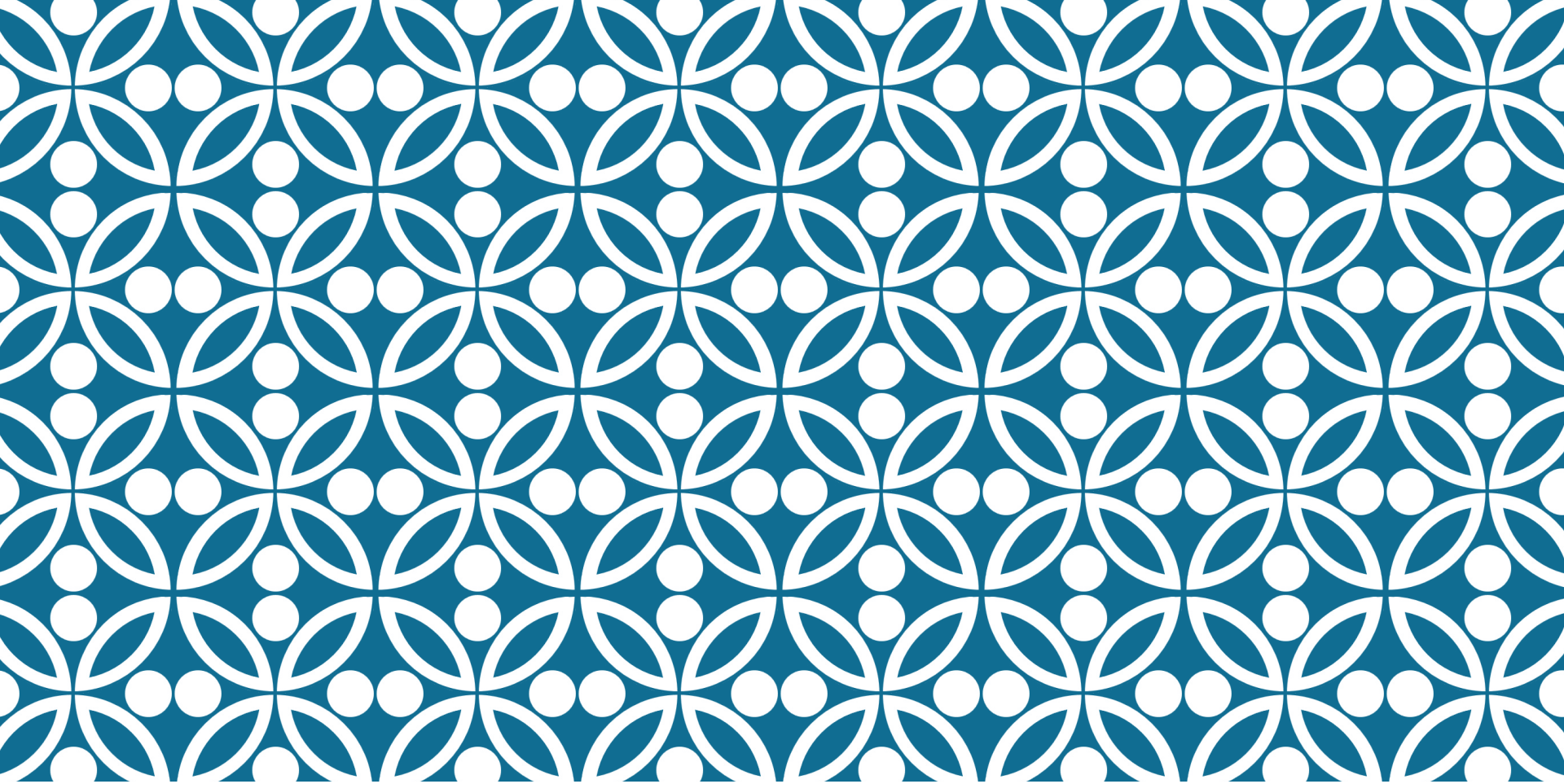
- Inicialmente, os processos devem construir o novo tipo derivado. A construção de tipos derivados é feita a partir dos tipos de dados básicos que o MPI define.
  - `MPI_Type_contiguous, MPI_Type_vector, MPI_Type_struct,`
  - `MPI_Type_indexed, MPI_Type_hvector, MPI_Type_hindexed`
- Em seguida, devem certificar perante o ambiente de execução do MPI a existência do novo tipo derivado.
  - `MPI_Type_Commit`
- Depois, o novo tipo de dados pode ser utilizado normalmente (send, receives, etc.).
- Após a sua utilização, cada processo libera a certificação feita.
  - `MPI_Type_Free(newtype, ierr), MPI_Type_free(type)`

TIPOS DERIVADOS

A construção de tipos derivados é custosa, e só deve ser utilizada quando o número de mensagens a trocar é significativo.

Os novos tipos de dados são utilizados nas funções de envio e recepção de mensagens tal como os outros tipos básicos.

Para tal é necessário que ambos os processos emissor e receptor tenham certificado o novo tipo derivado. Normalmente, isso é feito na parte do código que é comum a ambos os processos.



TYPE CONTIGUOUS

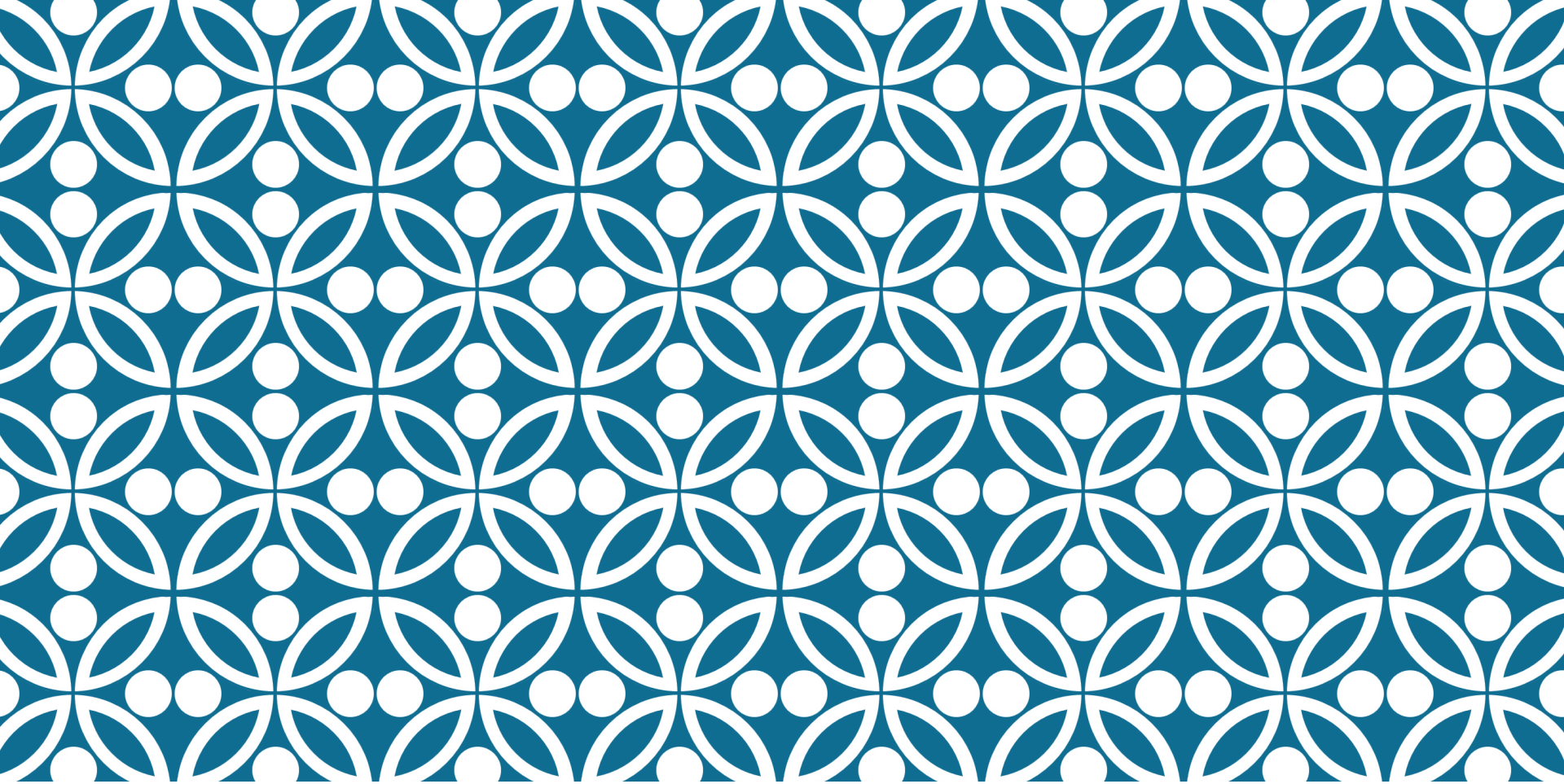
CONSTRUÇÃO DE TIPOS DERIVADOS

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

O tipo mais simples de dados derivado consiste em uma quantidade de itens contíguos de mesmo tipo.

EXEMPLO DE DADOS CONTÍGUOS

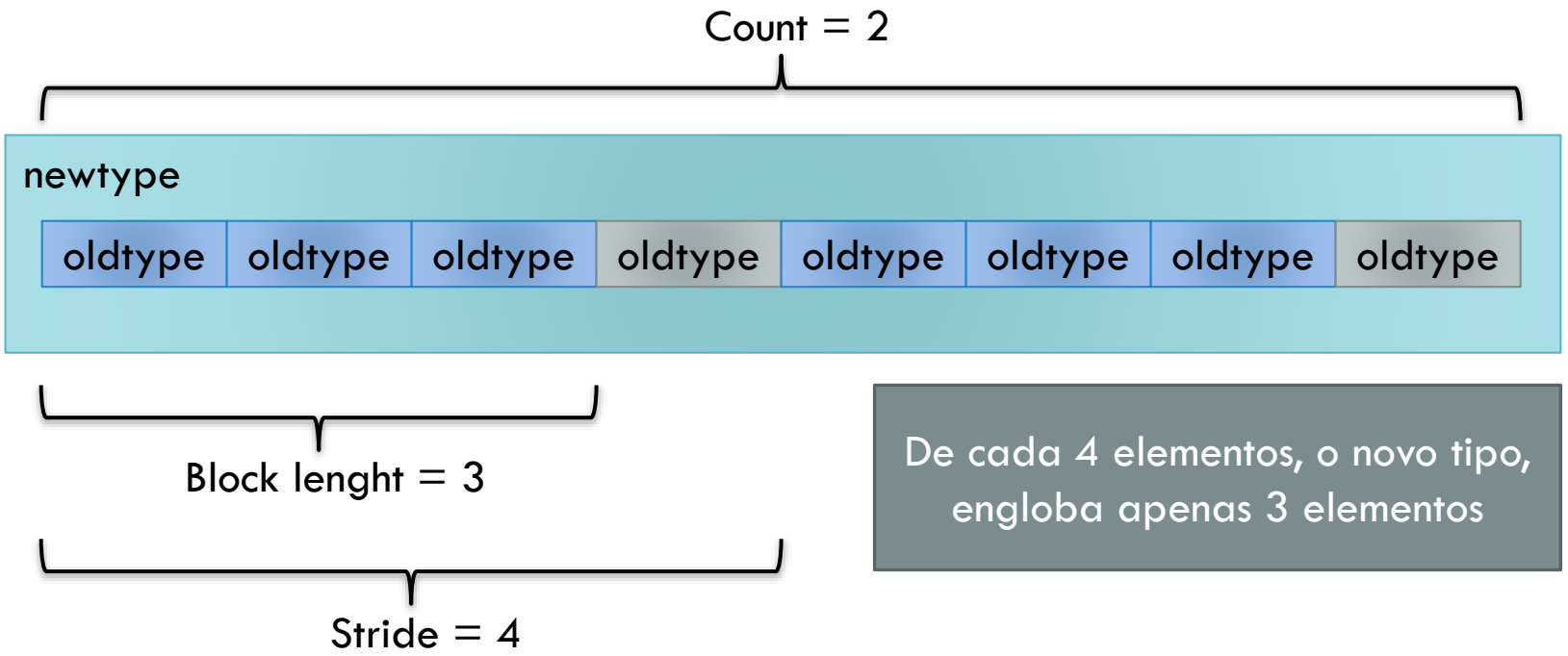
```
int main(int argc, char *argv[]) { /* Run with four processes */
  int rank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  struct { int x; int y; int z; } point;
  MPI_Datatype ptype;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Type_contiguous(3,MPI_INT,&ptype);
  MPI_Type_commit(&ptype);
  if(rank == 3){
    point.x=15; point.y=23; point.z=6;
    MPI_Send(&point,1,ptype,1,52,MPI_COMM_WORLD);
  } else if(rank == 1) {
    MPI_Recv(&point,1,ptype,3,52,MPI_COMM_WORLD,&status);
    printf("P:%d recv coords (%d,%d,%d) \n",rank,point.x,point.y,point.z);
  }
  MPI_Finalize();
}
```



TYPE VECTOR

CONSTRUÇÃO DE TIPOS DERIVADOS

```
MPI_Type_vector(int count, int blocklength, int stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```



De cada 4 elementos, o novo tipo, engloba apenas 3 elementos

CONSTRUÇÃO DE TIPOS DERIVADOS

```
MPI_Type_vector(int count, int blocklength, int stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

MPI_Type_vector() constrói um novo tipo de dados a partir de um vetor de dados.

count é o número de blocos de dados do novo tipo derivado.

blocklength é o número de elementos contíguos de cada bloco.

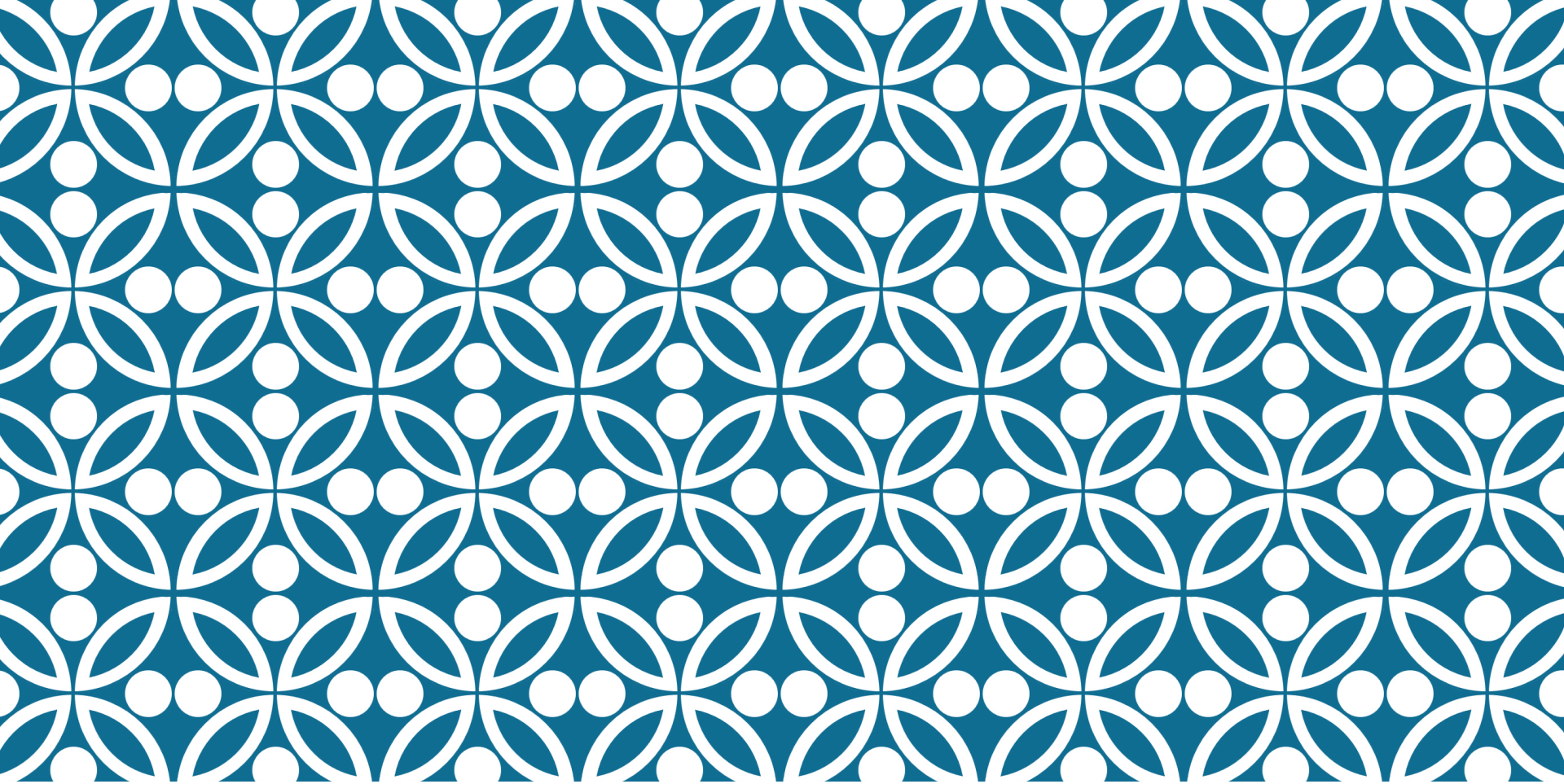
stride é o número de elementos contíguos que separa o início de cada bloco (deslocamento / espalhamento).

oldtype é o tipo de dados dos elementos do vetor.

newtype é o identificador do novo tipo derivado.

EXTRAIR COLUNAS DE MATRIZES (MPI VECTOR.C)

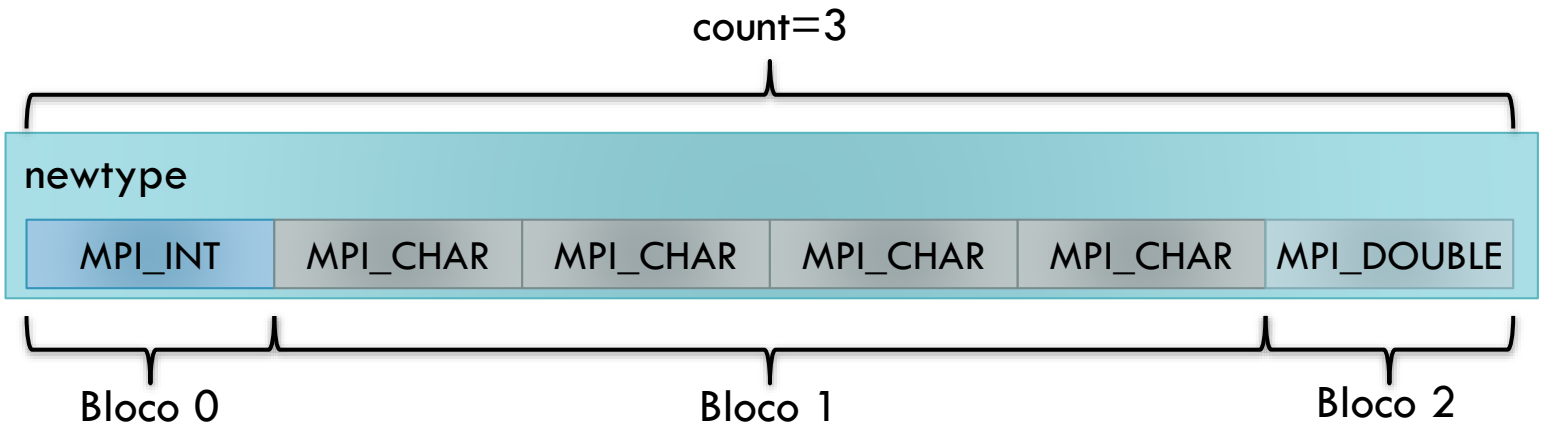
```
int my_matrix[ROWS][COLS];
int my_vector[ROWS];
MPI_Datatype col_matrix;
...
// construir um tipo derivado com ROWS blocos
// de 1 elemento separados por COLS elementos
MPI_Type_vector(ROWS, 1, COLS, MPI_INT, &col_matrix);
MPI_Type_commit(&col_matrix);
...
// enviar a coluna 1 de my_matrix (agrupando o espalhamento)
MPI_Send(&my_matrix[0][1], 1, col_matrix, dest, tag, comm);
...
// receber uma dada coluna na coluna 3 de my_matrix (recebe com espalhamento)
MPI_Recv(&my_matrix[0][3], 1, col_matrix, source, tag, comm, &status);
...
// receber uma dada coluna em my_vector (recebe contíguo)
MPI_Recv(&my_vector[0], ROWS, MPI_INT, source, tag, comm, &status);
...
// libertar o tipo derivado
MPI_Type_free(&col_matrix);
```



TYPE STRUCT

CONSTRUÇÃO DE TIPOS DERIVADOS

```
MPI_Type_struct(int count, int lengths[], MPI_Aint offsets[],
      MPI_Datatype oldtypes[], MPI_Datatype *newtype)
```




```
lengths[3] = {1, 4, 1}
offsets[3] = {0, int_length, int_length + 4 * char_length}
oldtypes[3] = {MPI_INT, MPI_CHAR, MPI_DOUBLE}
```

CONSTRUÇÃO DE TIPOS DERIVADOS

```
MPI_Type_struct(int count, int lengths[], MPI_Aint offsets[],
      MPI_Datatype oldtypes[], MPI_Datatype *newtype)
```

MPI Type struct() constrói um novo tipo de dados a partir de uma estrutura de dados.

count é o número de blocos de dados do novo tipo derivado. Representa igualmente o número de entradas nos vetores **lengths**[], **offsets**[] e **oldtypes**[].

lengths[] é o número de elementos contíguos de cada bloco.

offsets[] é o deslocamento em bytes de cada bloco dentro da estrutura.

oldtypes[] é o tipo de dados dos elementos de cada bloco.

newtype é o identificador do novo tipo derivado.

CONSTRUÇÃO DE TIPOS DERIVADOS

```
typedef struct {
  int var;
  char string[STRING_LENGTH];
  double foo;
} bar;

// 1. Vamos indicar que existem 3 blocos, seus tamanhos e tipos
int count = 3;
int lengths[3] = {1, STRING_LENGTH, 1};
MPI_Dataype oldtypes[3] = {MPI_INT, MPI_CHAR, MPI_DOUBLE};

// 2. Os offsets indicam em que byte cada elemento inicia
MPI_Aint offsets[3] = {0, sizeof(int), sizeof(int) + STRING_LENGTH};

// 3. Declarar o novo tipo, a estrutura e informar os processos
MPI_Datatype barDatatype;
MPI_Type_struct(count, lengths, offsets, types, &barDatatype);
MPI_Type_commit(&barDatatype);
```

ESTRUTURAS DE DADOS (MPI STRUCT.C)

```
struct { int a; char b[10]; double c[2]; } my_struct;
...
MPI_Datatype struct_type;
int blocklengths[3];
blocklengths[0] = 1; blocklengths[1] = 10; blocklengths[2] = 2;

MPI_Aint int_length, char_length, displacements[3];
MPI_Datatype oldtypes[3];
oldtypes[0] = MPI_INT; oldtypes[1] = MPI_CHAR; oldtypes[2] = MPI_DOUBLE;
...
// construir o tipo derivado representando my_struct
MPI_Type_extent(MPI_INT, &int_length);  // Devolve o tamanho do tipo MPI_INT
MPI_Type_extent(MPI_CHAR, &char_length); // Devolve o tamanho do tipo MPI_CHAR
displacements[0] = 0;
displacements[1] = int_length;
displacements[2] = int_length + 10 * char_length;
MPI_Type_struct(3, blocklengths, displacements, oldtypes, &struct_type);
MPI_Type_commit(&struct_type);
...
MPI_Send(&my_struct, 1, struct_type, dest, tag, comm); // enviar my_struct
...
MPI_Recv(&my_struct, 1, struct_type, source, tag, comm, &status); // receber em
my_struct
```

CONSTRUÇÃO DE TIPOS DERIVADOS

Um tipo derivado é um objeto que especifica um conjunto de tipos básicos e os respectivos deslocamentos. Os tipos básicos indicam ao MPI como interpretar os bits, enquanto que os deslocamentos indicam onde se encontram esses bits.

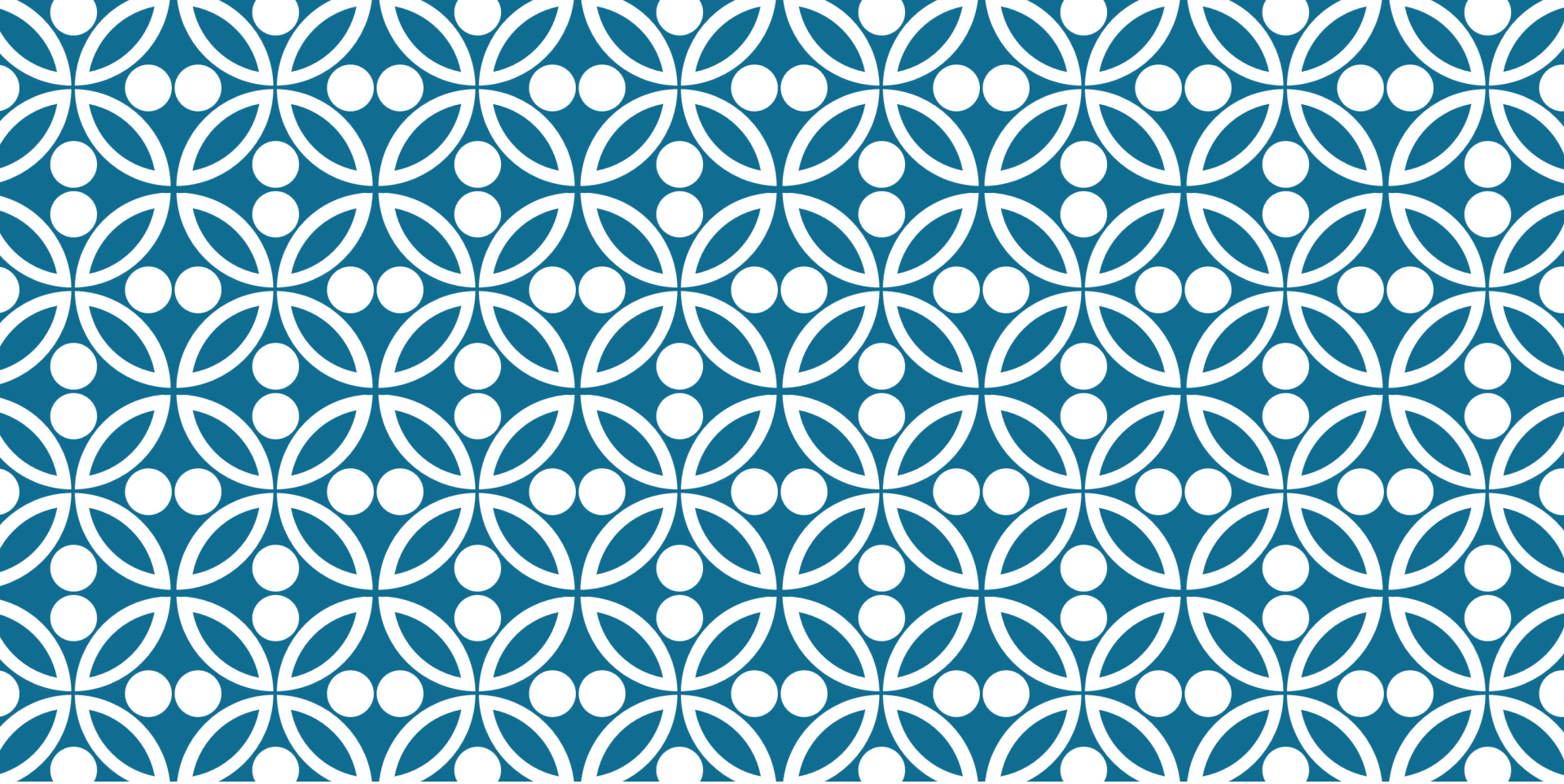
Para instanciar os deslocamentos de um novo tipo derivado deve utilizar-se uma das seguintes funções:

```
MPI_Type_extent(MPI_Datatype datatype, MPI_Aint
                *extent)
```

MPI_Type_extent() devolve em **extent** o tamanho em bytes do tipo de dados datatype.

```
MPI_Address(void *location, MPI_Aint *address)
```

MPI_Address() devolve em **address** o endereço de memória de **location**.



FUNÇÕES AUXILIARIES PARA NOVOS TIPOS DE DADOS

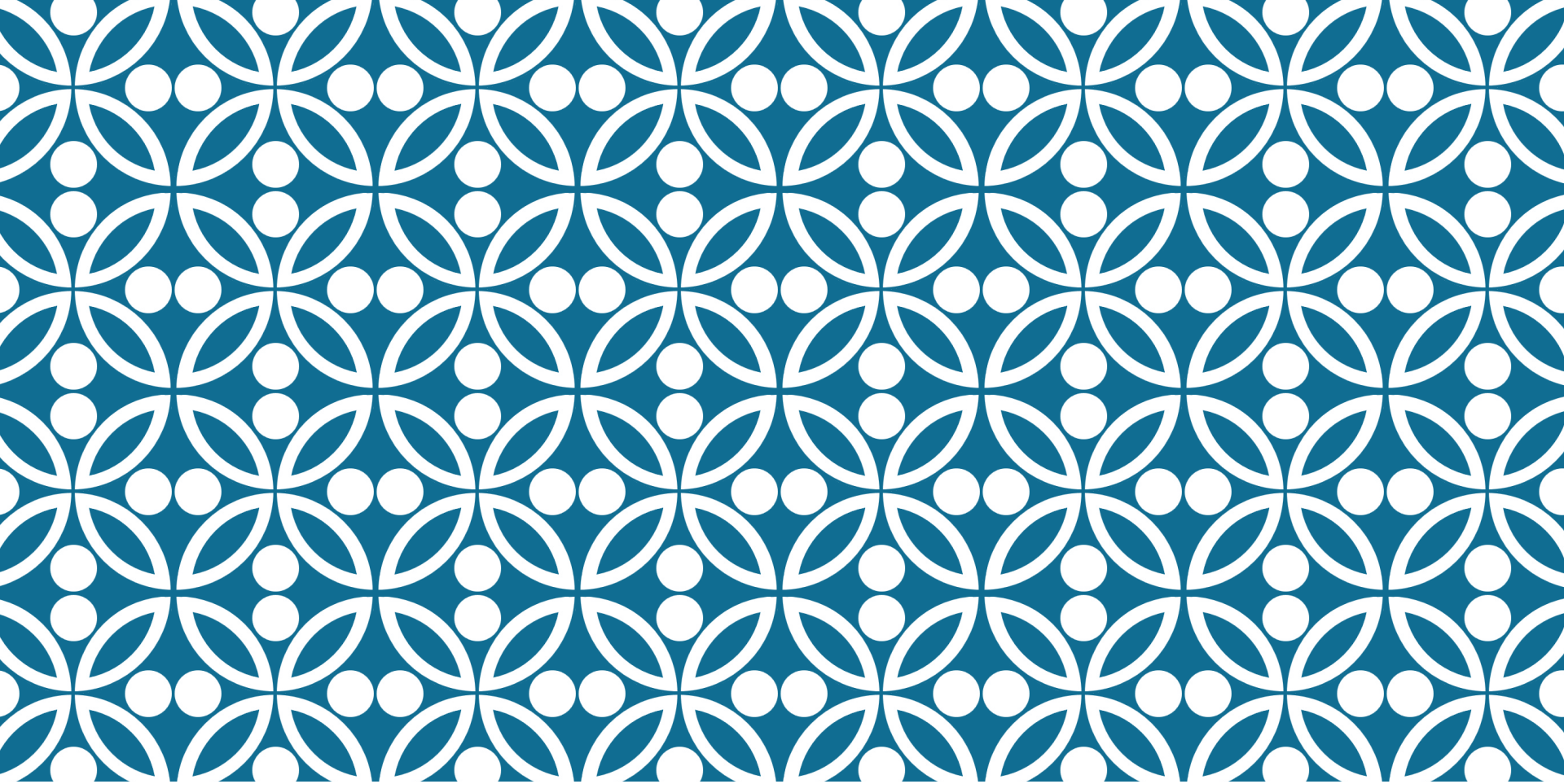
CERTIFICAR E LIBERAR UM TIPO DERIVADO

```
MPI_Type_commit(MPI_Datatype *datatype)
```

MPI_Type_commit() certifica perante o ambiente de execução do MPI a existência de um novo tipo derivado identificado por **datatype**.

```
MPI_Type_free(MPI_Datatype *datatype)
```

MPI_Type_free() libera do ambiente de execução o tipo derivado identificado por **datatype**.



PACOTES DE DADOS

EMPACOTAR DADOS

```
MPI_Pack(void *buf, int count, MPI_Datatype datatype,
void *packbuf, int packsize, int *position, MPI_Comm comm)
```

MPI_Pack() permite empacotar dados não contíguos em posições contiguas de memória.

buf é o endereço inicial dos dados a empacotar.

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

datatype é o tipo de dados a empacotar.

packbuf é o endereço do buffer onde devem ser colocados os dados a empacotar.

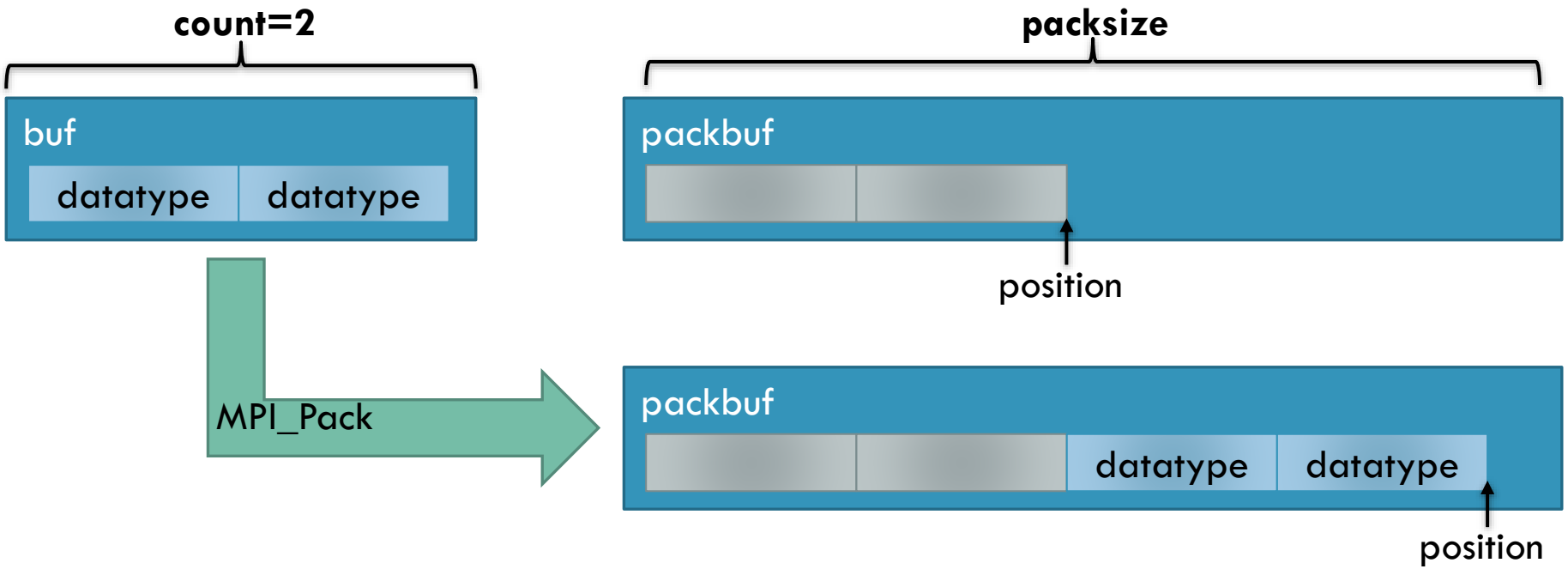
packsize é o tamanho em bytes do buffer de empacotamento.

position é a posição (em bytes) do buffer a partir da qual os dados devem ser empacotados.

comm é o comunicador dos processos envolvidos na comunicação.

EMPACOTAR DADOS

```
MPI_Pack(void *buf, int count, MPI_Datatype datatype,
void *packbuf, int packsize, int *position, MPI_Comm comm)
```



DESEMPACOTAR DADOS

```
MPI_Unpack(void *packbuf, int packsize, int *position,
void *buf, int count, MPI_Datatype datatype, MPI_Comm comm)
```

MPI_Unpack() permite desempacotar dados contíguos em posições não contíguas de memória.

packbuf é o endereço do buffer onde estão os dados a desempacotar.

packsize é o tamanho em bytes do buffer de empacotamento.

position é a posição (em bytes) do buffer a partir da qual estão os dados a desempacotar.

buf é o endereço inicial para onde os dados devem ser desempacotados.

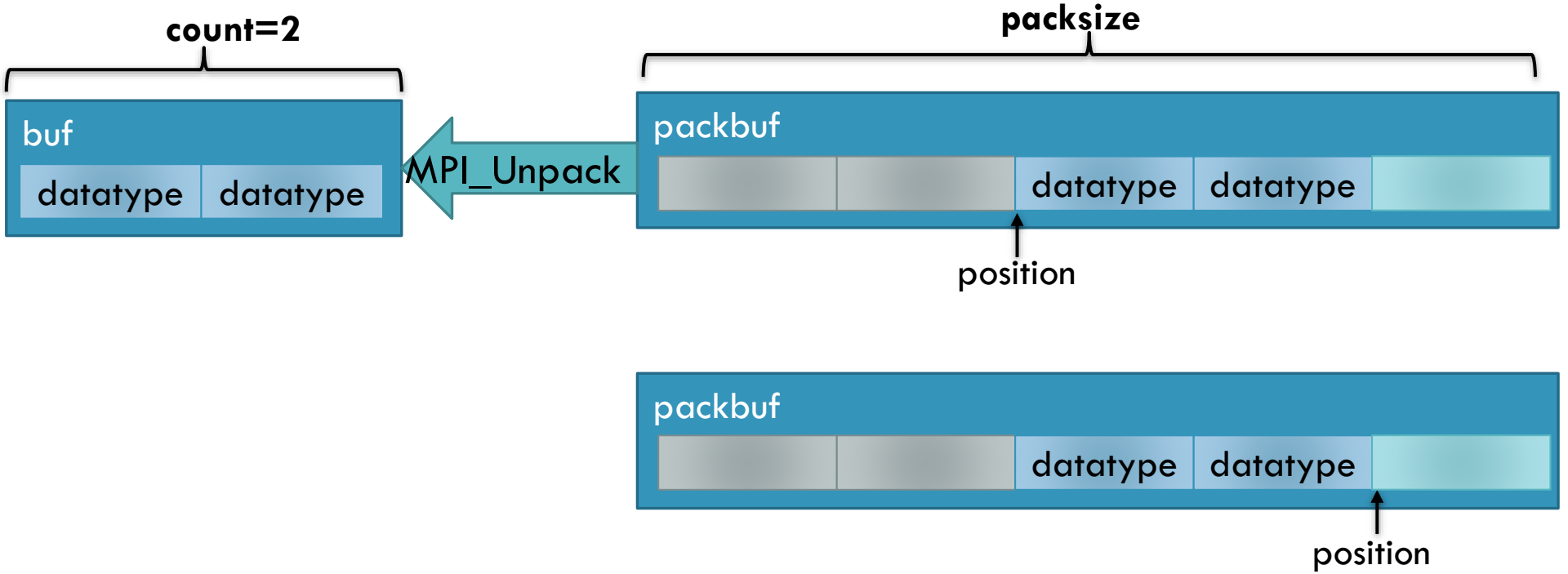
count é o número de elementos do tipo **datatype** a desempacotar.

datatype é o tipo de dados a desempacotar.

comm é o comunicador dos processos envolvidos na comunicação.

DESEMPACOTAR DADOS

```
MPI_Unpack(void *packbuf, int packsize, int *position,
void *buf, int count, MPI_Datatype datatype, MPI_Comm comm)
```



MATRIZ DE TAMANHO VARIÁVEL (MPI PACK.C)

```
// inicialmente ROWS e COLS não são do conhecimento do processo 1
int *my_matrix, ROWS, COLS, pos;
char pack_buf[BUF_SIZE];
...
if (my_rank == 0) {                  // empacotar e enviar ROWS, COLS e my_matrix
  pos = 0;
  MPI_Pack(&ROWS, 1, MPI_INT, pack_buf, BUF_SIZE, &pos, comm);
  MPI_Pack(&COLS, 1, MPI_INT, pack_buf, BUF_SIZE, &pos, comm);
  MPI_Pack(my_matrix, ROWS * COLS, MPI_INT, pack_buf, BUF_SIZE, &pos, comm);
  MPI_Send(pack_buf, pos, MPI_PACKED, 1, tag, comm);
} else if (my_rank == 1) {           // receber e desempacotar ROWS, COLS e my_matrix
  MPI_Recv(&pack_buf, BUF_SIZE, MPI_PACKED, 0, tag, comm, &status);
  pos = 0;
  MPI_Unpack(&pack_buf, BUF_SIZE, &pos, &ROWS, 1, MPI_INT, comm);
  MPI_Unpack(&pack_buf, BUF_SIZE, &pos, &COLS, 1, MPI_INT, comm);
                                     // aloca espaço para representar my_matrix
  my_matrix = (int *) malloc(ROWS * COLS * sizeof(int));
  MPI_Unpack(&pack_buf, BUF_SIZE, &pos, my_matrix, ROWS * COLS, MPI_INT, comm);
}
...
```

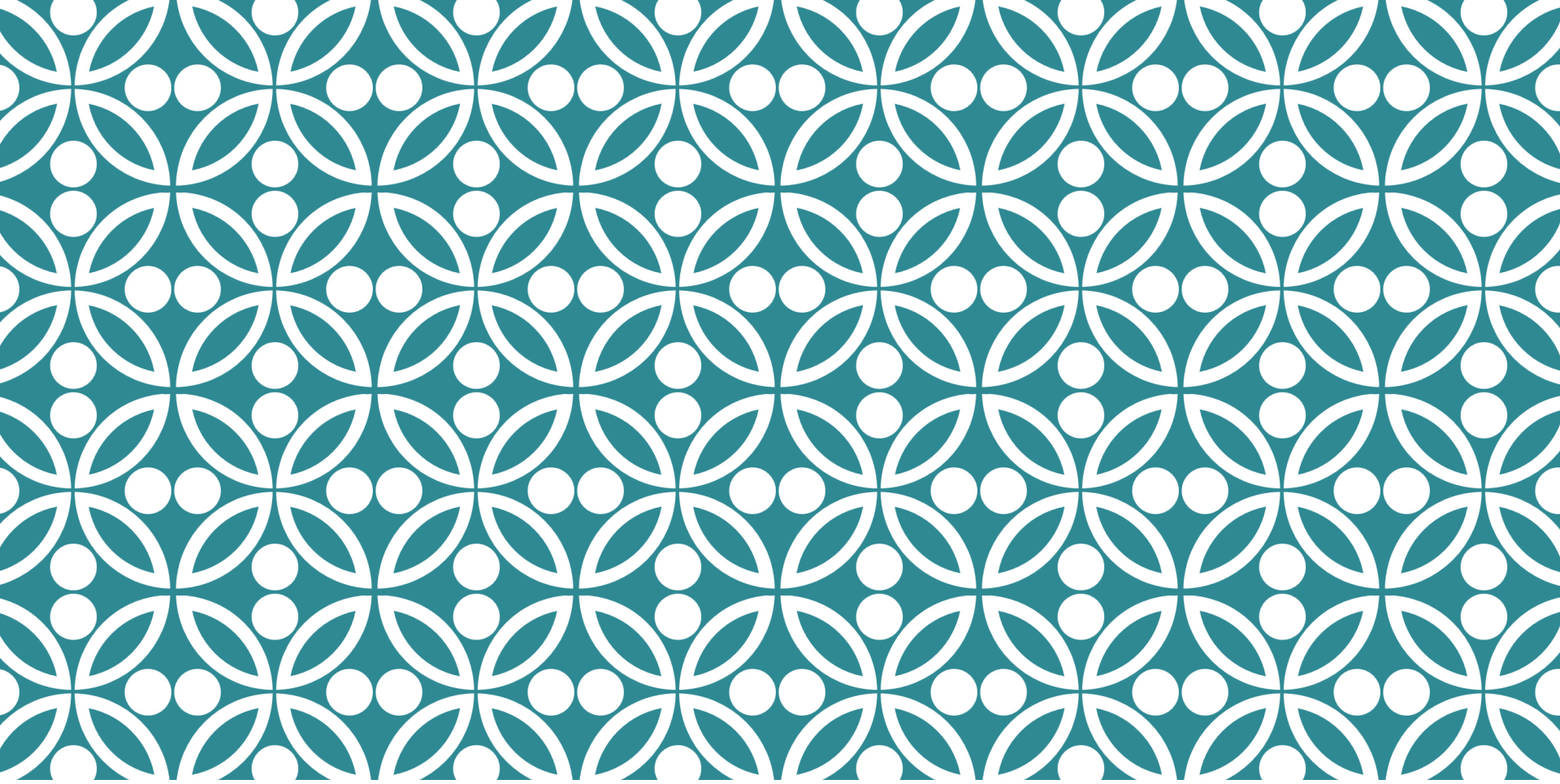
QUE TIPO DE DADOS DEVO UTILIZAR?

Se os dados forem todos do **mesmo tipo e se encontrarem em posições contíguas** de memórias então devemos utilizar contíguos ou o argumento count das funções de envio e recepção de mensagens.

Se os dados forem todos do **mesmo tipo mas não se encontrarem em posições contíguas de memória,** então devemos criar um tipo derivado utilizando as funções MPI_Type_vector() (para dados separados por intervalos regulares) ou MPI_Type_indexed() (para dados separados por intervalos irregulares).

Se os dados forem **heterogêneos e possuírem um determinado padrão constante** então devemos criar um tipo derivado utilizando a função MPI_Type_struct().

Se os dados forem heterogêneos mas **não possuírem padrões regulares** então devemos utilizar as funções MPI_Pack()/MPI_Unpack(). As funções MPI_Pack()/MPI_Unpack() também podem ser utilizadas para trocar **dados heterogêneos** apenas uma vez (ou relativamente **poucas vezes**).



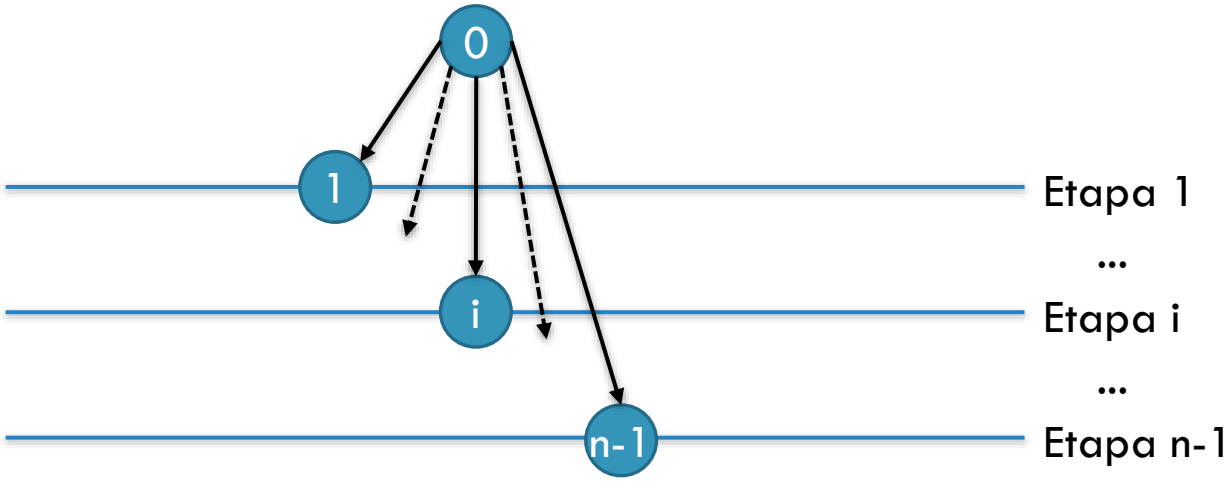
PROGRAMAÇÃO PARALELA
MPI 03 – COMUNICAÇÕES COLETIVAS

Marco A. Zanata Alves

Slides baseados nos slides de Ricardo Rocha – DCC FCUP (2009/2010)
Programação Paralela e Distribuída

COMUNICAÇÕES COLETIVAS

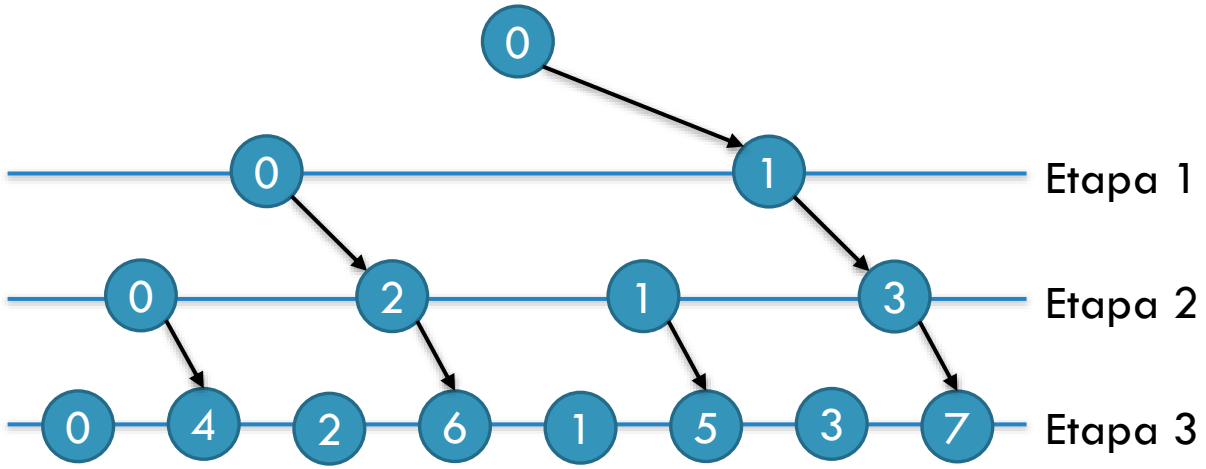
Em programação paralela é habitual que, em determinadas partes do programa, um dado processo distribua o mesmo conjunto de dados para todos os processos (e.g. iniciar dados ou tarefas).




```
...
if (my_rank == 0)
  for (dest = 1; dest < n_procs; dest++)
    MPI_Send(data, count, datatype, dest, tag, MPI_COMM_WORLD);
else
  MPI_Recv(data, count, datatype, 0, tag, MPI_COMM_WORLD, &status);
...
```

COMUNICAÇÕES COLETIVAS

A topologia de comunicação do esquema anterior é inerentemente sequencial pois todas as comunicações são realizadas a partir do processo 0. Se mais processos colaborarem na distribuição da informação podemos reduzir significativamente o tempo total de comunicação.



Se usarmos uma topologia em árvore, tal como na figura acima, podemos distribuir os dados em $\lceil \log_2 N \rceil$ etapas em vez de $N - 1$ como na situação anterior.

COMUNICAÇÕES COLETIVAS

Para implementar a topologia em árvore, cada processo precisa de calcular em cada etapa se é um processo emissor/receptor e qual o destino/origem dos dados a enviar/receber.

- Se $MyRank < 2^{stage-1}$, então envio para $MyRank + 2^{stage-1}$.
- Se $2^{stage-1} \leq MyRank < 2^{stage}$, então recebo de $MyRank - 2^{stage-1}$.

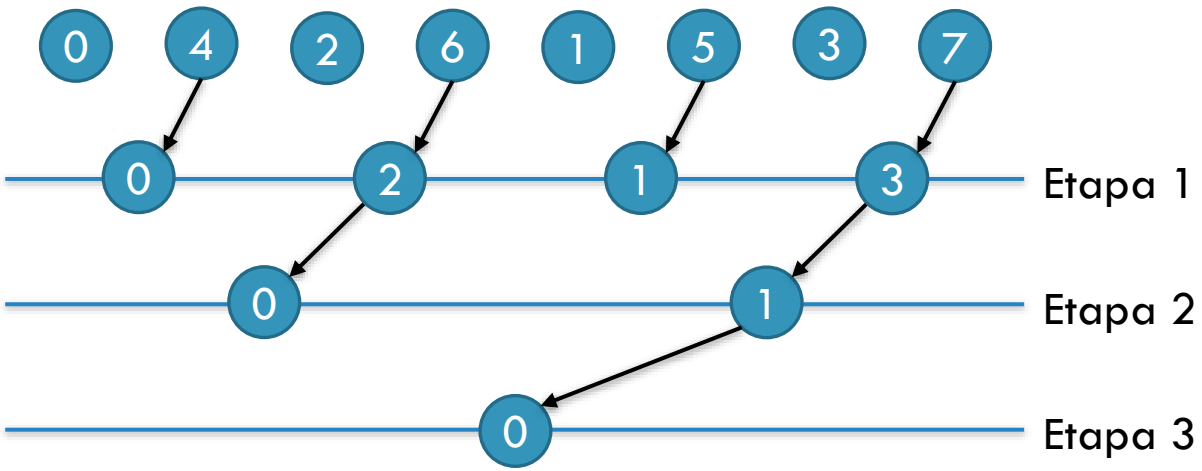
Segue-se uma possível implementação:

```
...
for (stage = 1; stage <= upper_log2(n_procs); stage++)
  if (my_rank < pow(2, stage - 1))
    send_to(my_rank + pow(2, stage - 1));
  else if (my_rank >= pow(2, stage - 1) && my_rank < pow(2, stage))
    receive_from(my_rank - pow(2, stage -1));
...
```

COMUNICAÇÕES COLETIVAS

Em programação paralela é igualmente habitual que, em determinadas partes do programa, um processo (normalmente o processo 0) recolha informação dos outros processos e calcule resumos dessa informação.

Se invertermos a topologia de comunicação em árvore, podemos aplicar a mesma ideia para agrupar dados em $\lceil \log_2 N \rceil$ etapas.



MENSAGENS COLETIVAS

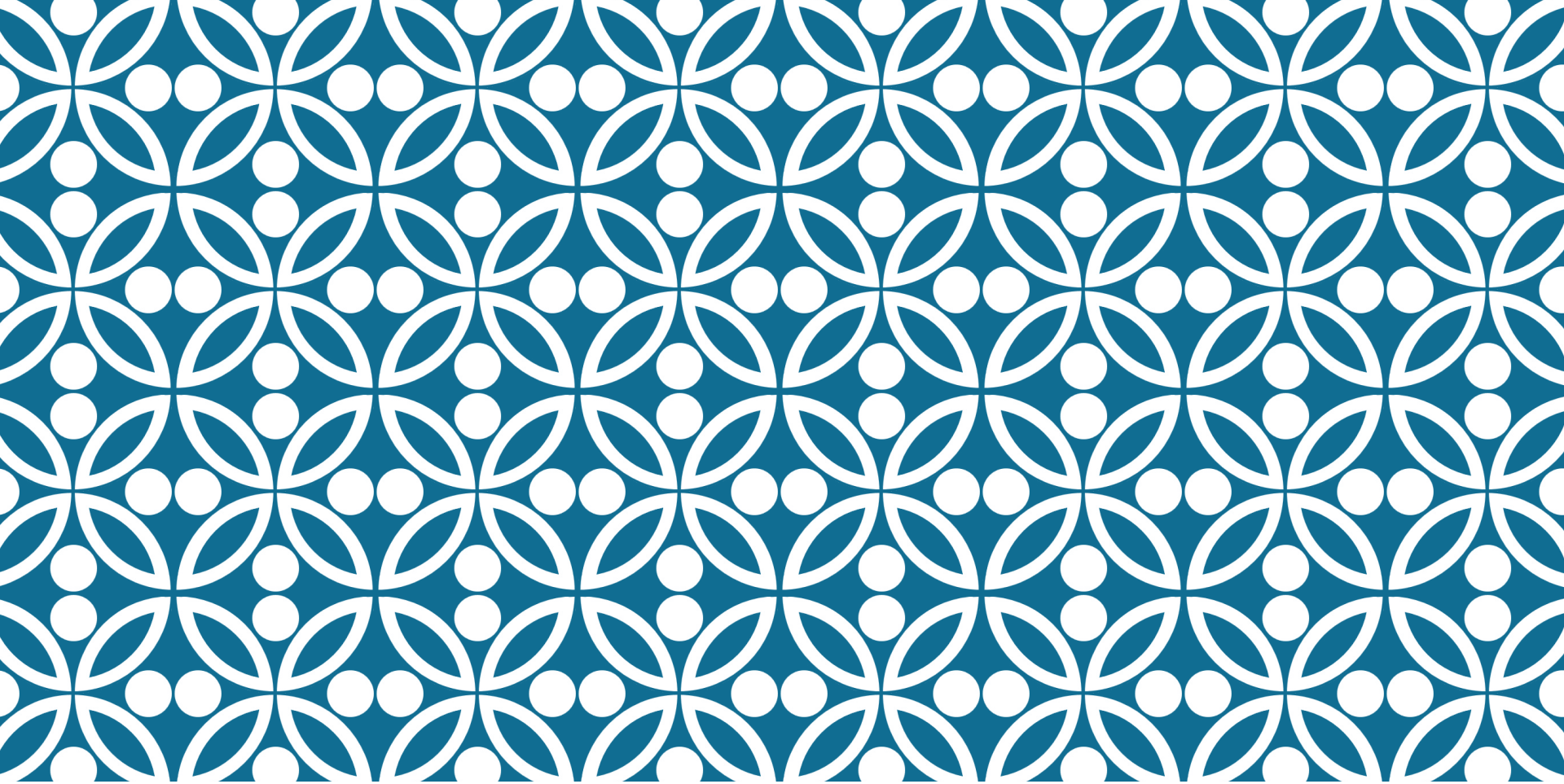
De forma a liberar o programador dos detalhes inerentes à topologia e eficiência das comunicações coletivas, o MPI define um conjunto de funções para lidar especificamente com este tipo de comunicações.

Podemos então classificar as mensagens em:

- **Ponto-a-ponto**: a mensagem é enviada por um processo e recebida por um outro processo (e.g. todo o tipo de mensagens que vimos anteriormente).
- **Coletivas**: podem consistir de várias mensagens ponto-a-ponto concorrentes e envolvendo todos os processos de um comunicador (as mensagens coletivas têm de ser chamadas por todos os processos do comunicador).

As mensagens coletivas são variações ou combinações das seguintes 4 operações primitivas:

- **Broadcast**
- **Reduce**
- **Scatter**
- **Gather**



BROADCAST

BROADCAST

```
MPI_Bcast(void *buf, int count,
MPI_Datatype datatype, int root, MPI_Comm comm)
```

MPI_Bcast() faz chegar uma mensagem de um processo a todos os outros processos no comunicador.

buf é o endereço inicial dos dados a enviar/receber.

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

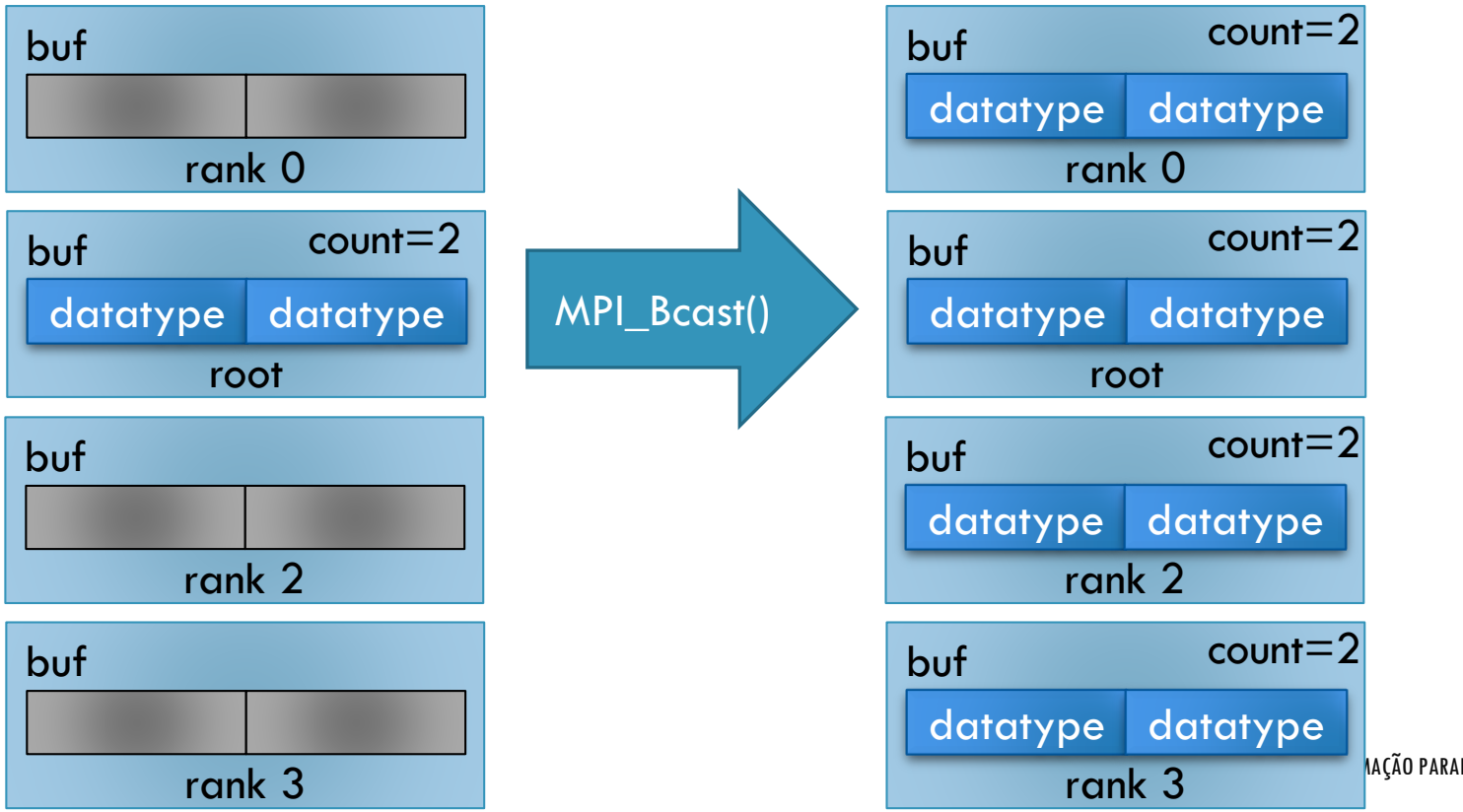
datatype é o tipo de dados a enviar/receber.

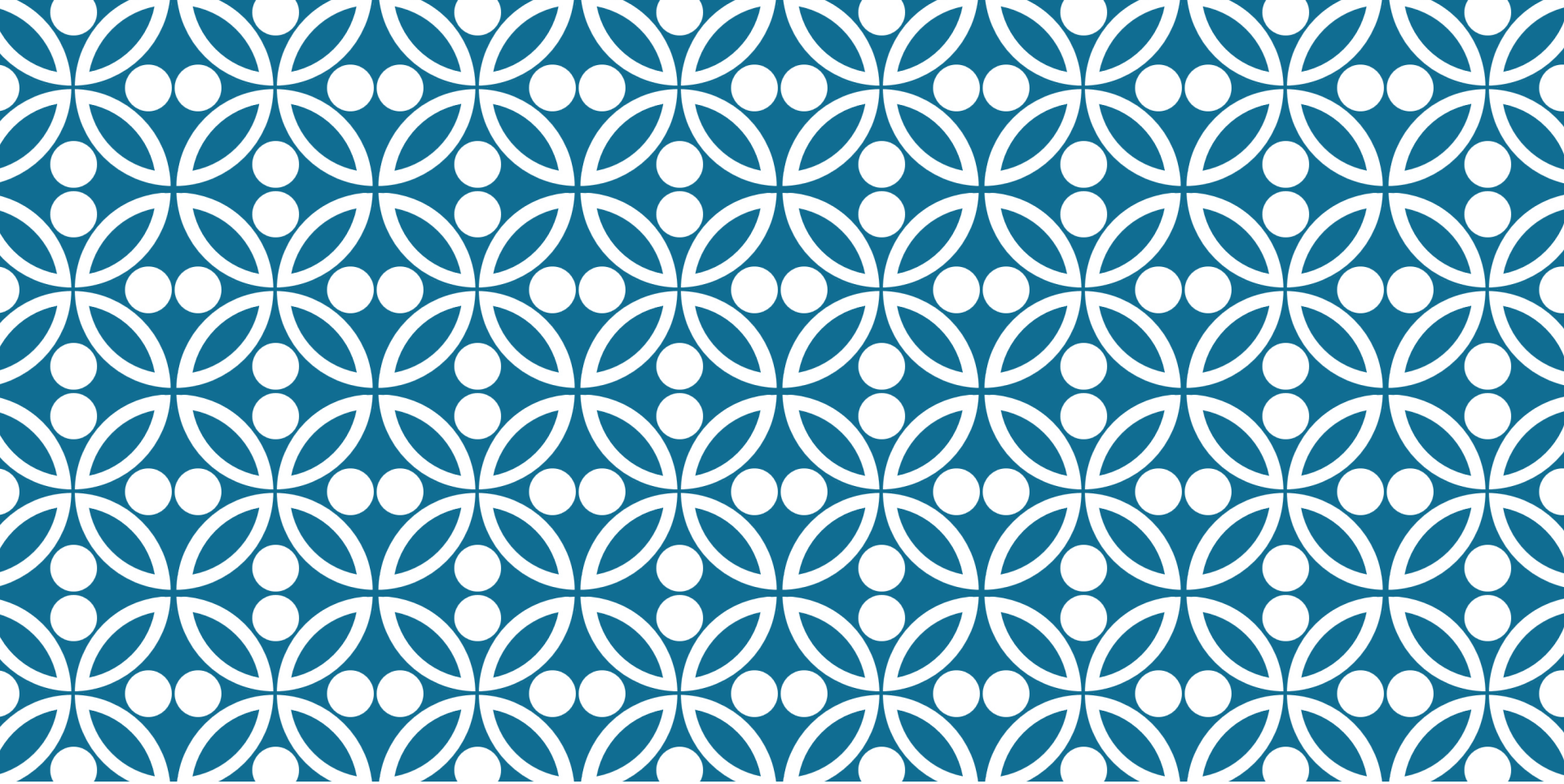
root é a posição do processo, no comunicador comm, que possui à partida a mensagem a enviar.

comm é o comunicador dos processos envolvidos na comunicação.

BROADCAST

```
MPI_Bcast(void *buf, int count,
MPI_Datatype datatype, int root, MPI_Comm comm)
```





REDUCE

REDUCE

```
MPI_Reduce(void *sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

MPI_Reduce() permite realizar operações globais de resumo fazendo chegar mensagens de todos os processos a um único processo no comunicador.

sendbuf é o endereço inicial dos dados a enviar.

recvbuf é o endereço onde devem ser colocados os dados recebidos (só é importante para o processo **root**).

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

datatype é o tipo de dados a enviar.

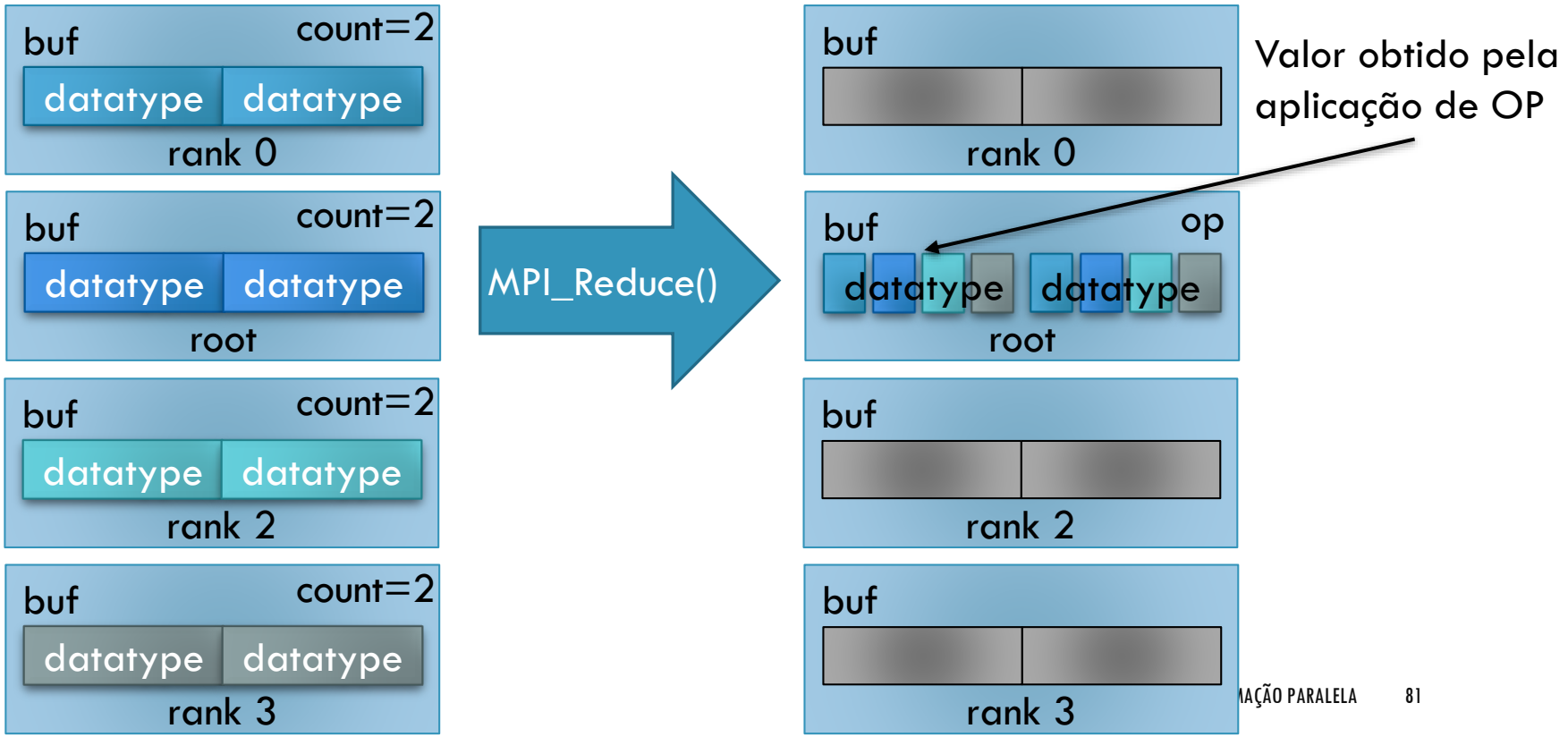
op é a operação de redução a aplicar aos dados recebidos.

root é a posição do processo, no comunicador **comm**, que recebe e resume os dados.

comm é o comunicador dos processos envolvidos na comunicação.

REDUCE

```
MPI_Reduce(void *sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

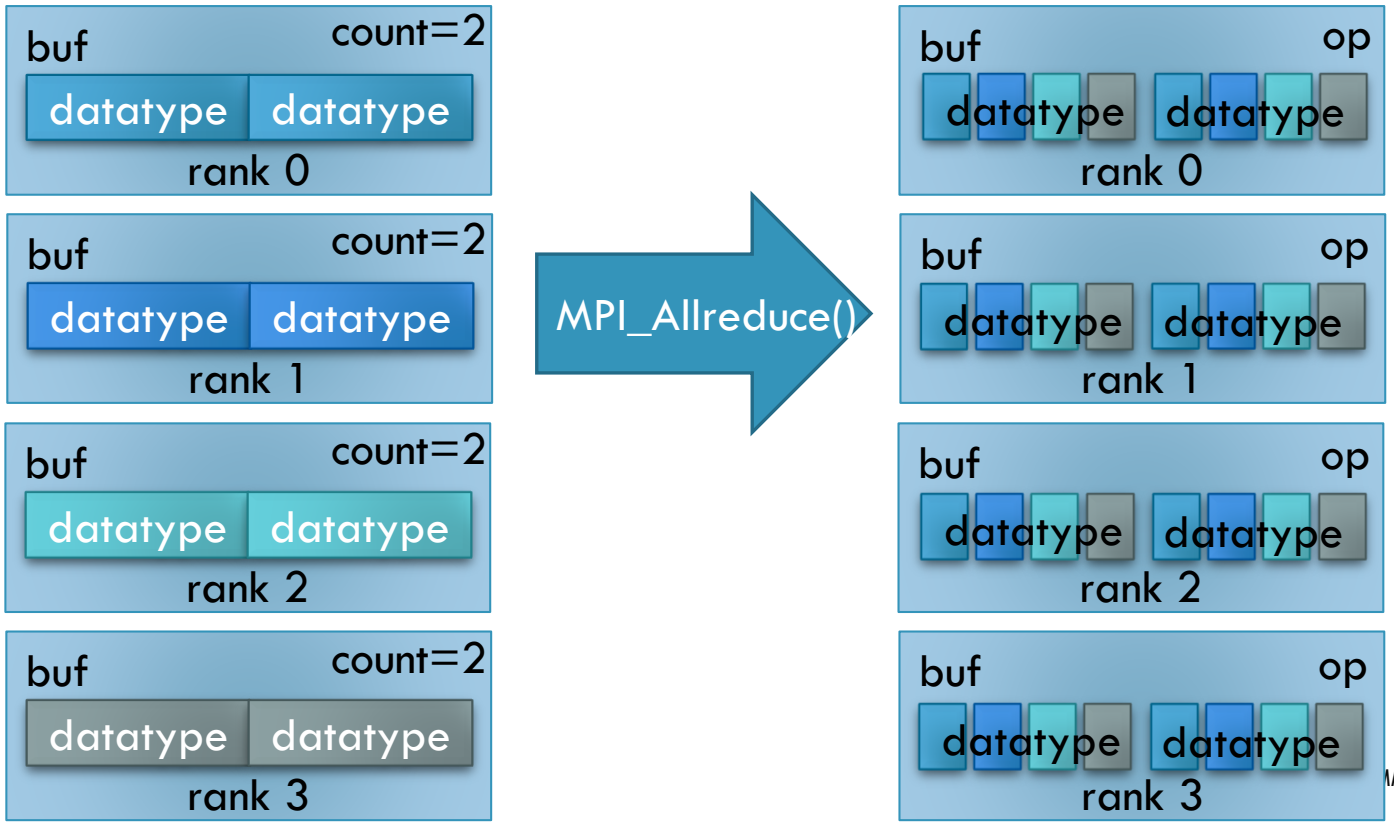


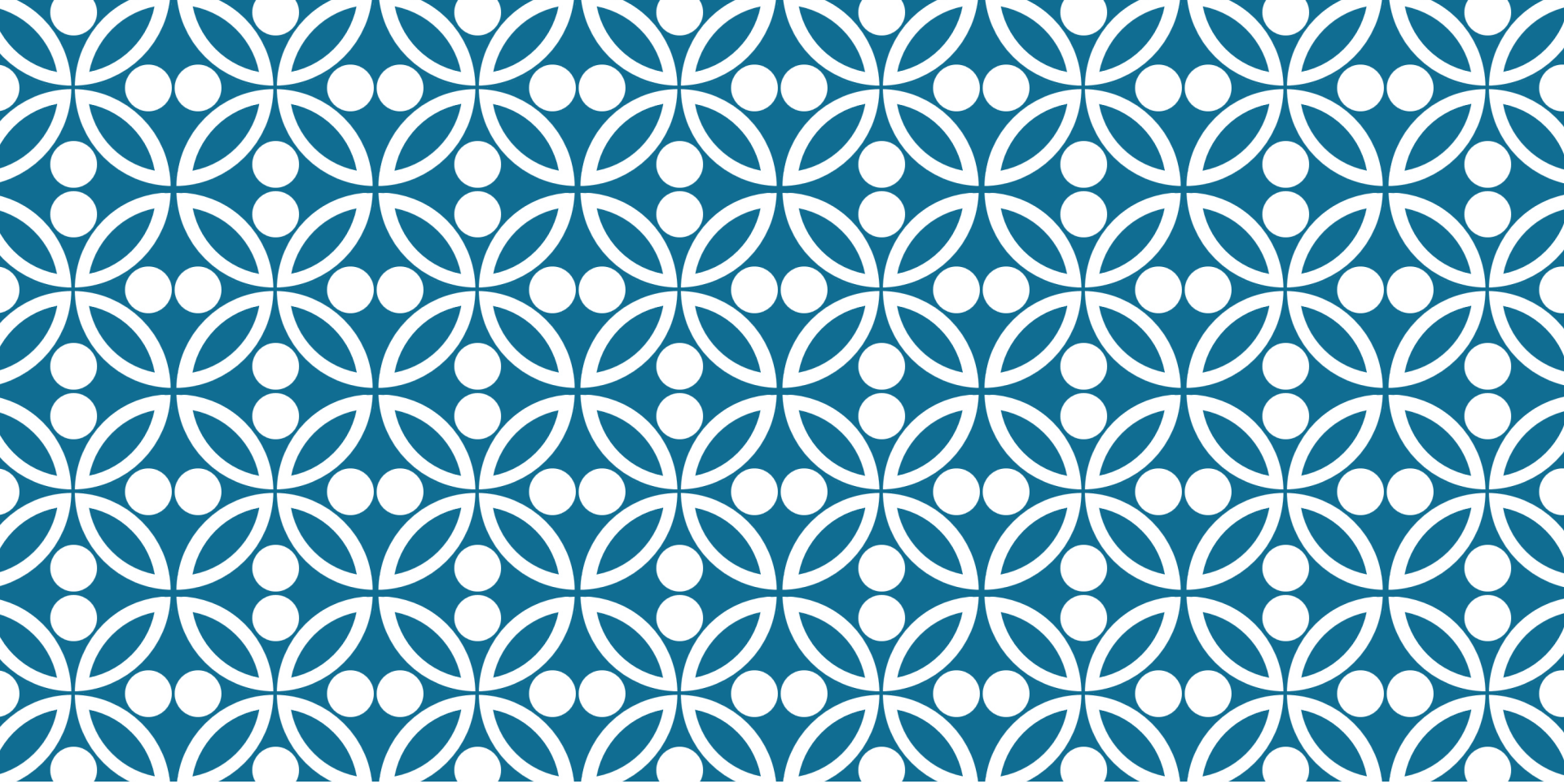
OPERAÇÕES DE REDUÇÃO

Operação	Significado
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Soma
MPI_PROD	Produto
MPI_LAND	E lógico
MPI_BAND	E dos bits
MPI_LOR	OU lógico
MPI_BOR	OU dos bits
MPI_LXOR	OU exclusivo lógico
MPI_BXOR	OU exclusivo dos bits

ALL REDUCE

```
MPI_Allreduce(void *sendbuf, void* recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```





SCATTER

SCATTER

```
MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_Scatter() divide em partes iguais os dados de uma mensagem e distribui ordenadamente cada uma das partes por cada um dos processos no comunicador.

sendbuf é o endereço inicial dos dados a enviar (só é importante para o processo **root**).

sendcount é o número de elementos do tipo **sendtype** a enviar para cada processo (só é importante para o processo **root**).

sendtype é o tipo de dados a enviar (só é importante para o processo **root**).

SCATTER

```
MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

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

recvcount é o número de elementos do tipo **recvtype** a receber por processo (normalmente o mesmo que **sendcount**).

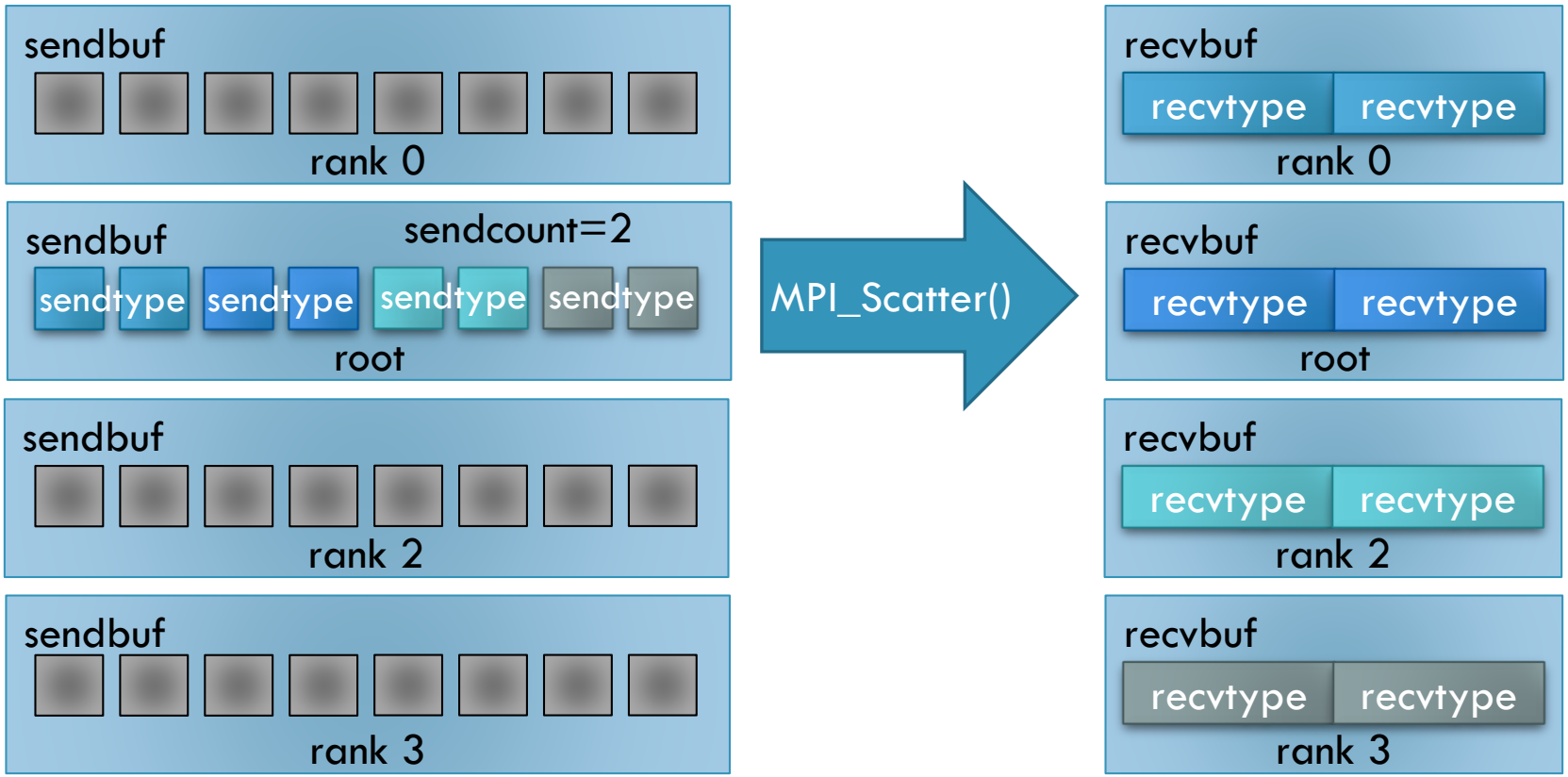
recvtype é o tipo de dados a receber (normalmente o mesmo que **sendtype**).

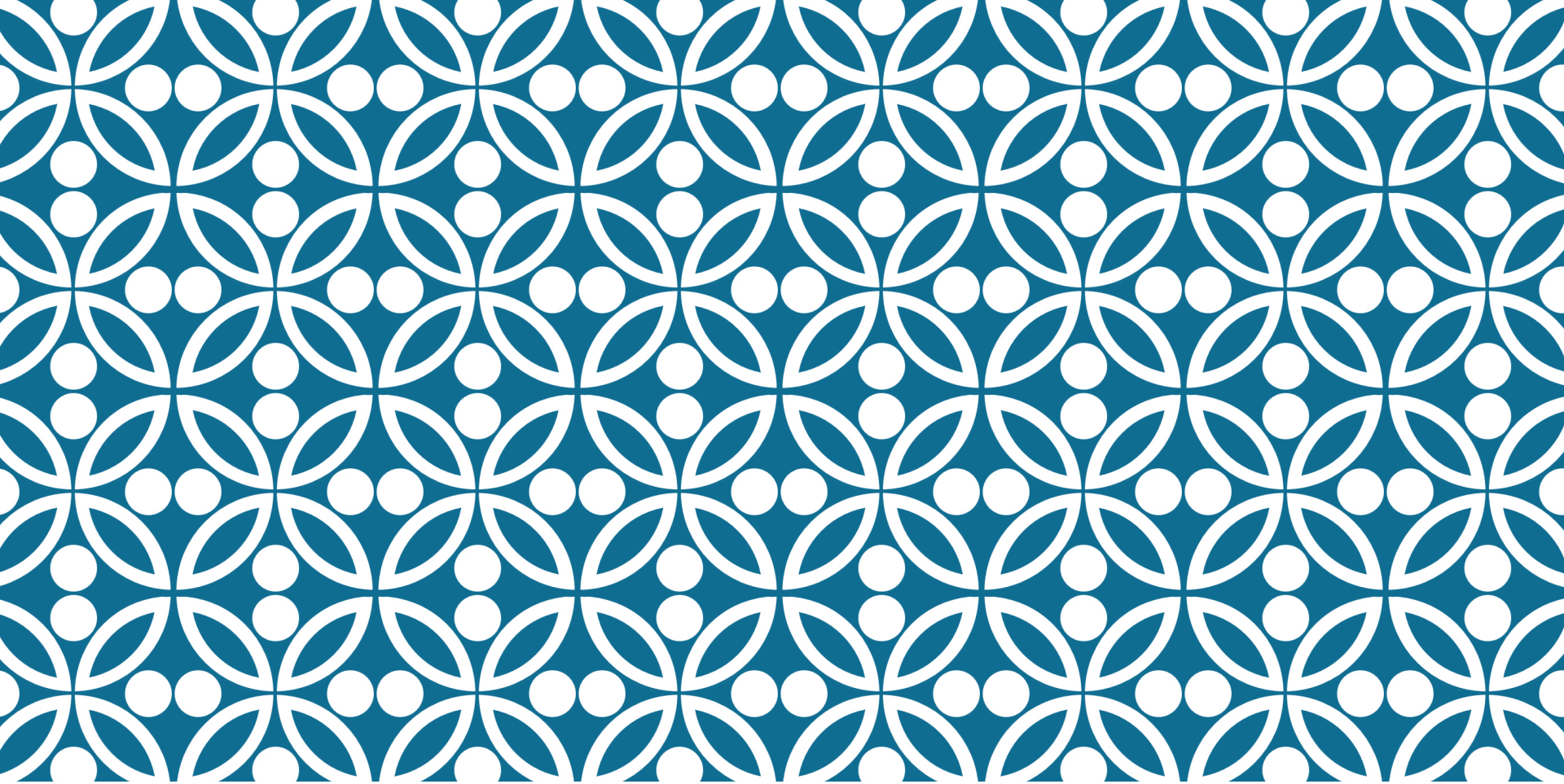
root é a posição do processo, no comunicador **comm**, que possui à partida a mensagem a enviar.

comm é o comunicador dos processos envolvidos na comunicação.

SCATTER

```
MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```





GATHER

GATHER

```
MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_Gather() recolhe ordenadamente num único processo um conjunto de mensagens oriundo de todos os processos no comunicador.

sendbuf é o endereço inicial dos dados a enviar.

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

sendtype é o tipo de dados a enviar.

GATHER

```
MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

recvbuf é o endereço onde devem ser colocados os dados recebidos (só é importante para o processo **root**).

recvcount é o número de elementos do tipo **recvtype** a receber de cada processo (normalmente o mesmo que **sendcount**; só é importante para o processo **root**).

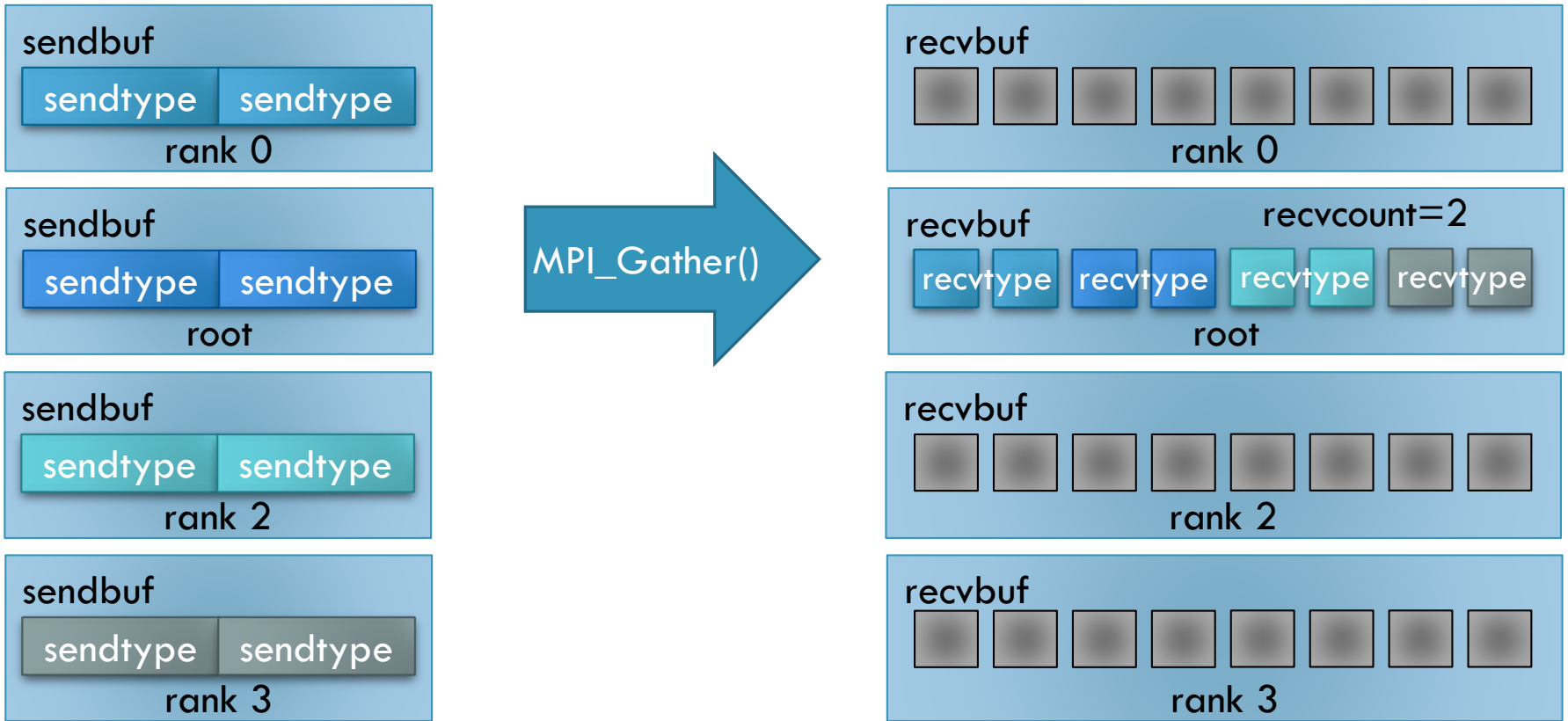
recvtype é o tipo de dados a receber (normalmente o mesmo que **sendtype**; só é importante para o processo **root**).

root é a posição do processo, no comunicador **comm**, que recebe os dados.

comm é o comunicador dos processos envolvidos na comunicação.

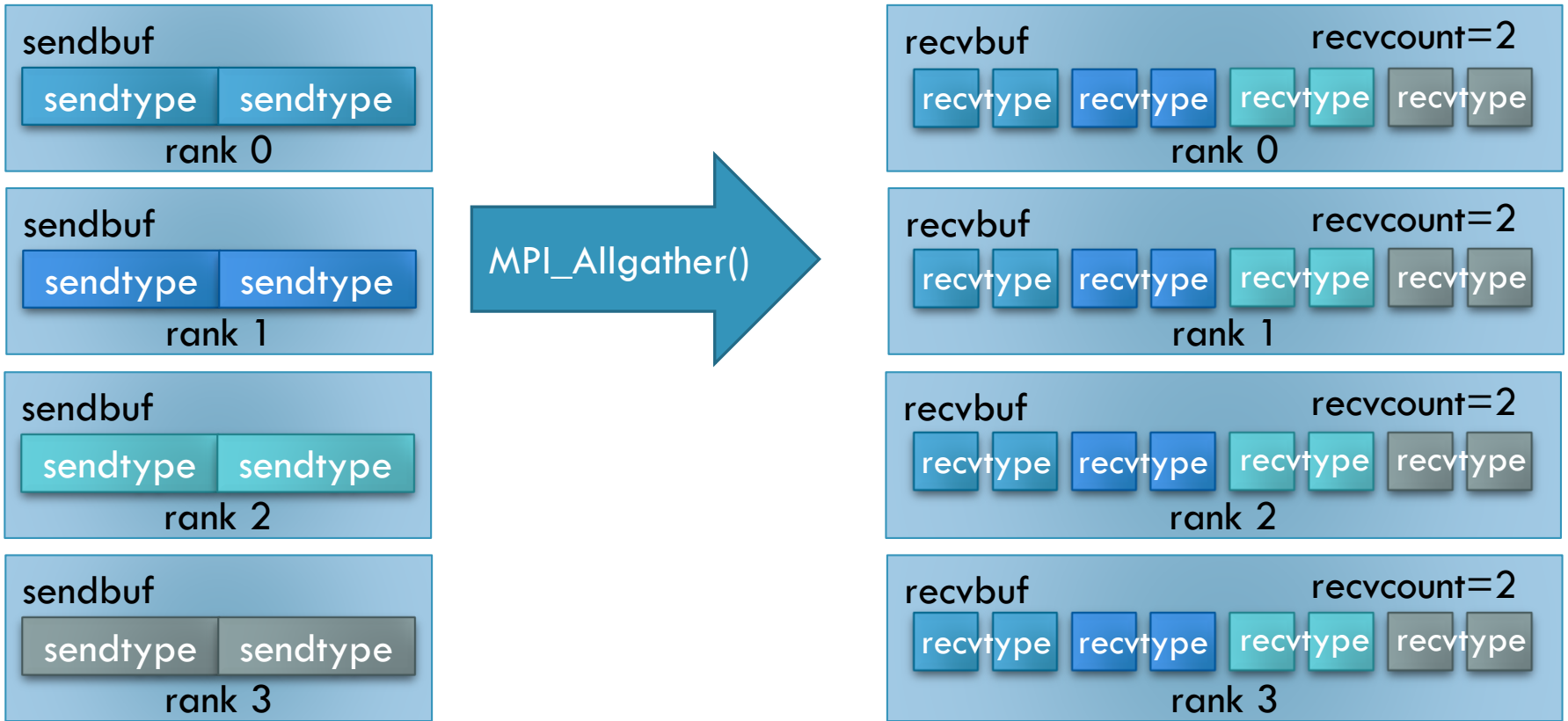
GATHER

```
MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



ALL GATHER

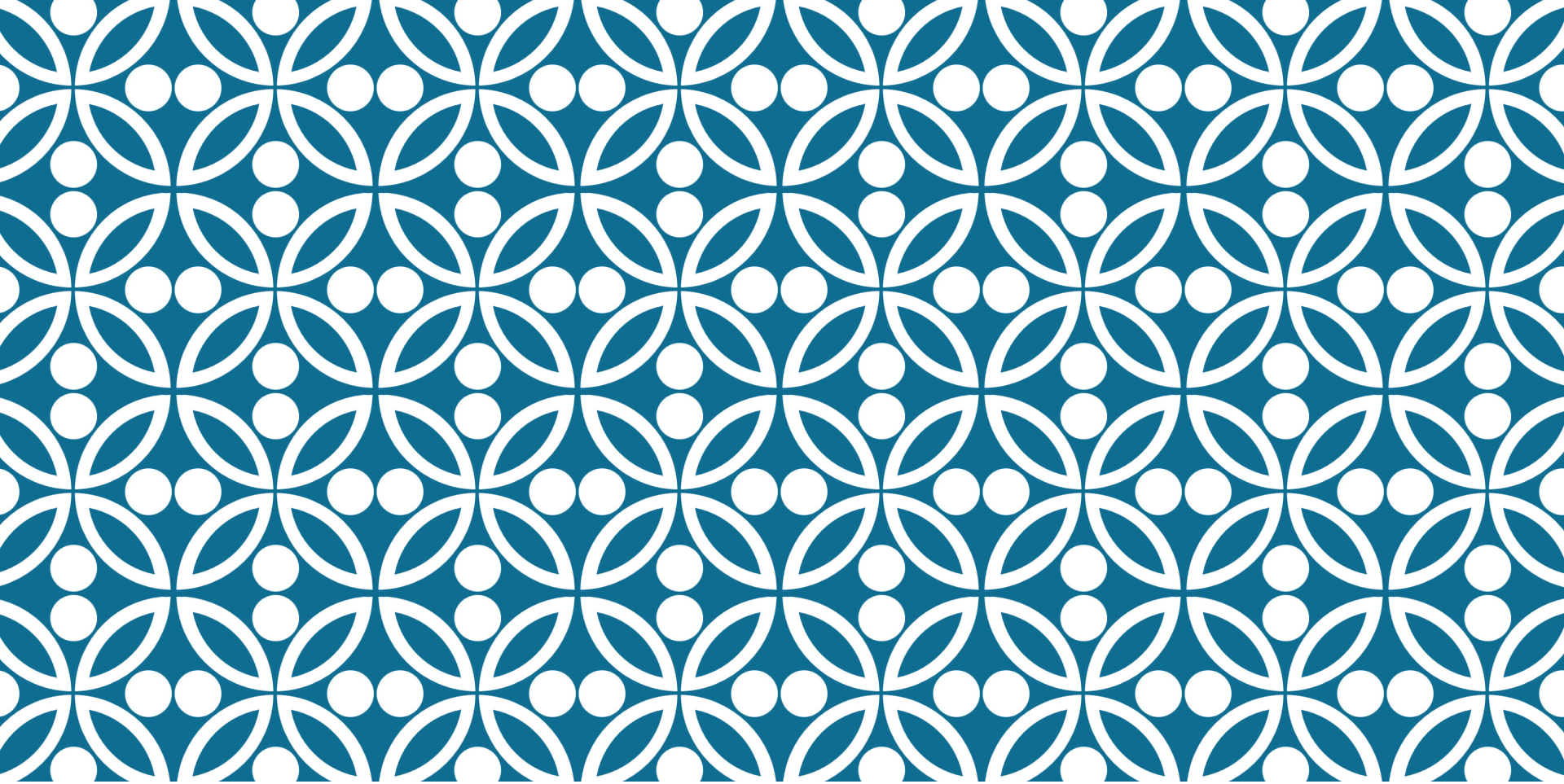
```
MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



OUTRAS VARIAÇÕES

MPI_SCATTERV e MPI_GATHERV estendem funcionalidades, permitindo que um número variável de dados a serem enviados para cada processo, uma vez que ele suporta um vetor de tamanhos.

Também prove flexibilidade para escolher de onde os dados serão tomados do root (acesso stride/esparso), através do argumento displs.



EXEMPLO

MÉDIA DE N NÚMEROS

```
if (world_rank == 0) {
    rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);

// Gather all partial averages down to the root process
float *sub_avgs = NULL;
if (world_rank == 0) {
    sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the total average of all numbers.
if (world_rank == 0) {
    float avg = compute_avg(sub_avgs, world_size);
}
```

PRODUTO ESCALAR

O produto escalar de 2 vetores de dimensão N é definido por:

$$x \cdot y = x_0 y_0 + x_1 y_1 + \cdots + x_{n-1} y_{n-1}$$

Se tivermos P processos, cada um deles pode calcular K(N/P) componentes do produto escalar:

Processo	Componentes
0	$x_0 y_0 + x_1 y_1 + \cdots + x_{k-1} y_{k-1}$
1	$x_k y_k + x_{k+1} y_{k+1} + \cdots + x_{2k-1} y_{2k-1}$
...	...
P-1	$x_{(p-1)k} y_{(p-1)k} + x_{(p-1)k+1} y_{(p-1)k+1} + \cdots + x_{n-1} y_{n-1}$

PRODUTO ESCALAR

Processo	Componentes
0	$x_0y_0 + x_1y_1 + \cdots + x_{k-1}y_{k-1}$
1	$x_ky_k + x_{k+1}y_{k+1} + \cdots + x_{2k-1}y_{2k-1}$
...	...
P-1	$x_{(p-1)k}y_{(p-1)k} + x_{(p-1)k+1}y_{(p-1)k+1} + \cdots + x_{n-1}y_{n-1}$

Segue-se uma possível implementação:

```
int produto_escalar(int x[], int y[], int n) {
  int i, pe = 0;
  for (i = 0; i < n; i++)
    pe = pe + x[i] * y[i];
  return pe;
}
```

PRODUTO ESCALAR (MPI_ESCALAR.C)

```
int *vector_x, *vector_y;
int K, pe, loc_pe, *loc_x, *loc_y;
...
if (my_rank == ROOT) {
  ... // calcular K e iniciar os vectores X e Y
}
// enviar K a todos os processos
MPI_Bcast(&K, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
// alocar espaço para os vectores locais
loc_x = (int *) malloc(K * sizeof(int));
loc_y = (int *) malloc(K * sizeof(int));
// distribuir as componentes dos vectores X e Y
MPI_Scatter(vector_x, K, MPI_INT, loc_x, K, MPI_INT, ROOT, MPI_COMM_WORLD);
MPI_Scatter(vector_y, K, MPI_INT, loc_y, K, MPI_INT, ROOT, MPI_COMM_WORLD);
// calcular o produto escalar
loc_pe = produto_escalar(loc_x, loc_y, K);
MPI_Reduce(&loc_pe, &pe, 1, MPI_INT, MPI_SUM, ROOT,MPI_COMM_WORLD);
// apresentar o resultado
if (my_rank == ROOT)
  printf("Produto Escalar = %d \ n", pe);
...
```

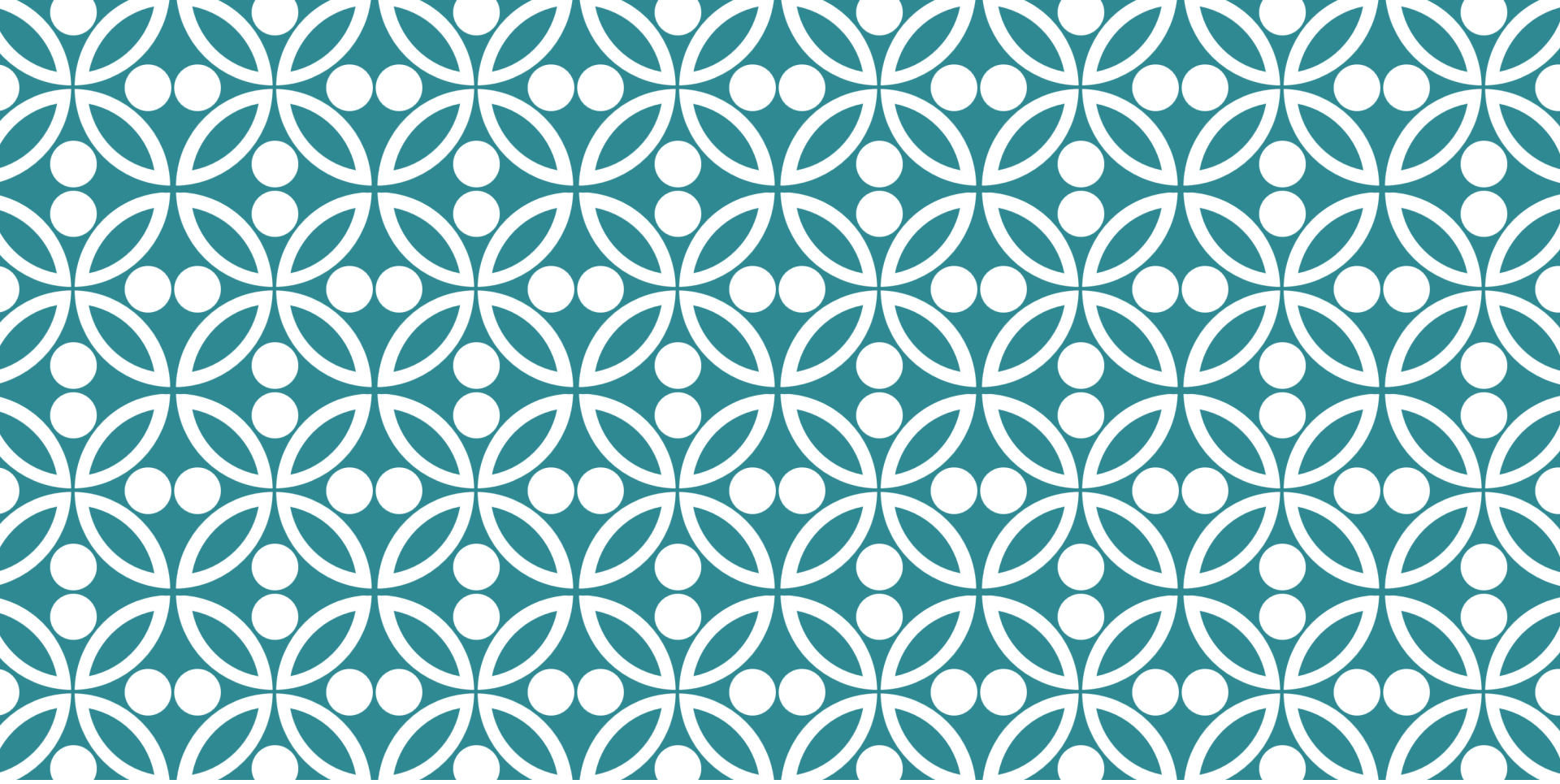
PRODUTO MATRIZ-VECTOR (MPI_PRODUTO.C)

Sejam **matrix[ROWS,COLS]** e **vector[COLS]** respectivamente uma matriz e um vetor coluna. O produto matriz-vetor é um vetor linha **result[ROWS]** em que cada result[i] é o produto escalar da linha i da matriz pelo vetor.

Se tivermos ROWS processos, cada um deles pode calcular um elemento do vetor resultado.

PRODUTO MATRIZ-VECTOR (MPI_PRODUTO.C)

```
int ROWS, COLS, *matrix, *vector, *result;
int pe, *linha;
... // iniciar ROWS, COLS e o vetor
if (my_rank == ROOT) { ... // iniciar a matriz }
// distribuir a matriz
MPI_Scatter(matrix, COLS, MPI_INT, linha, COLS, MPI_INT, ROOT,
MPI_COMM_WORLD);
// calcular o produto matriz-vetor e apresentar o resultado
pe = produto_escalar(linha, vector, COLS);
MPI_Gather(&pe, 1, MPI_INT, result, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
if (my_rank == ROOT) {
  printf("Produto Matriz-Vector: ");
  for (i = 0; i < ROWS; i++)
    printf("%d ", result[i]);
}
...
```



PROGRAMAÇÃO PARALELA
MPI 04 - COMUNICADORES

Marco A. Zanata Alves

Slides baseados nos slides de Ricardo Rocha – DCC FCUP (2009/2010)
Programação Paralela e Distribuída

COMUNICADORES

Um comunicador pode ser descrito como um grupo de processos que podem trocar mensagens entre si. Associado a um comunicador temos:

- **Um grupo**: conjunto ordenado de processos.
- **Um contexto**: estrutura de dados que o identifica de forma única.

Para além do comunicador universal, o ambiente de execução do MPI permite criar novos comunicadores.

O MPI distingue 2 tipos de comunicadores:

- **Intra-comunicadores**: permitem a troca de mensagens e realização de operações coletivas.
- **Inter-comunicadores**: permitem a troca de mensagens entre processos pertencentes a intra-comunicadores disjuntos.

CRIAR GRUPOS

```
MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

MPI_Comm_group() devolve em **group** o grupo de processos do comunicador **comm**.

```
MPI_Group_incl(MPI_Group old_group, int size,
    int ranks[], MPI_Group *new_group)
```

MPI_Group_incl() cria um novo grupo **new_group** a partir de **old_group** constituído pelos **size** processos referenciados em **ranks**[].

```
MPI_Group_excl(MPI_Group old_group, int size,
    int ranks[], MPI_Group *new_group)
```

MPI_Group_excl() cria um novo grupo **new_group** a partir de **old_group** e exclui os **size** processos referenciados em **ranks**[].

CRIAR COMUNICADORES

```
MPI_Comm_create(MPI_Comm old_comm,
     MPI_Group group, MPI_Comm *new_comm)
```

MPI_Comm_create() cria um novo comunicador **new_comm** constituído pelo grupo de processos **group** do comunicador **old_comm**.

MPI_Comm_create() é uma comunicação coletiva, pelo que deve ser chamada por todos os processos, incluindo aqueles que não aderem ao novo comunicador.

No caso de serem criados vários comunicadores, a ordem de criação deve ser a mesma em todos os processos.

LIBERAR GRUPOS E COMUNICADORES

```
MPI_Group_free(MPI_Group *group)
```

MPI_Group_free() libera o grupo **group** do ambiente de execução.

```
MPI_Comm_free(MPI_Comm *comm)
```

MPI_Comm_free() libera o comunicador **comm** do ambiente de execução.

PROCESSOS PARES (MPI_EVEN.C)

```
MPI_Group world_group, even_group;
MPI_Comm even_comm;
...
for (i = 0; i < n_procs; i += 2)
  ranks[i/2] = i;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
MPI_Group_incl(world_group, (n_procs + 1)/2, ranks, &even_group);
MPI_Comm_create(MPI_COMM_WORLD, even_group, &even_comm);
MPI_Group_free(&world_group);
MPI_Group_free(&even_group);
if (my_rank % 2 == 0) {
  MPI_Comm_rank(even_comm, &even_rank);
  printf("Rank: world %d even %d \ n", my_rank, even_rank);
  MPI_Comm_free(&even_comm);
}
...
```

CRIAR COMUNICADORES

```
MPI_Comm_dup(MPI_Comm old_comm, MPI Comm *new_comm)
```

MPI_Comm_dup() cria um novo comunicador **new_comm** idêntico a **old_comm**.

```
MPI_Comm_split(MPI_Comm old_comm, int split_key,
               int rank_key, MPI_Comm *new_comm)
```

MPI_Comm_split() cria um ou mais comunicadores **new_comm** a partir de **old_comm** agrupando em cada novo comunicador os processos com idênticos valores de **split_key** e ordenando-os por **rank_key**.

Os ranks dos processos nos novos comunicadores são atribuídos por ordem crescente do argumento rank_key. Ou seja, o processo com o menor rank_key terá rank 0, o segundo menor rank 1, e assim sucessivamente.

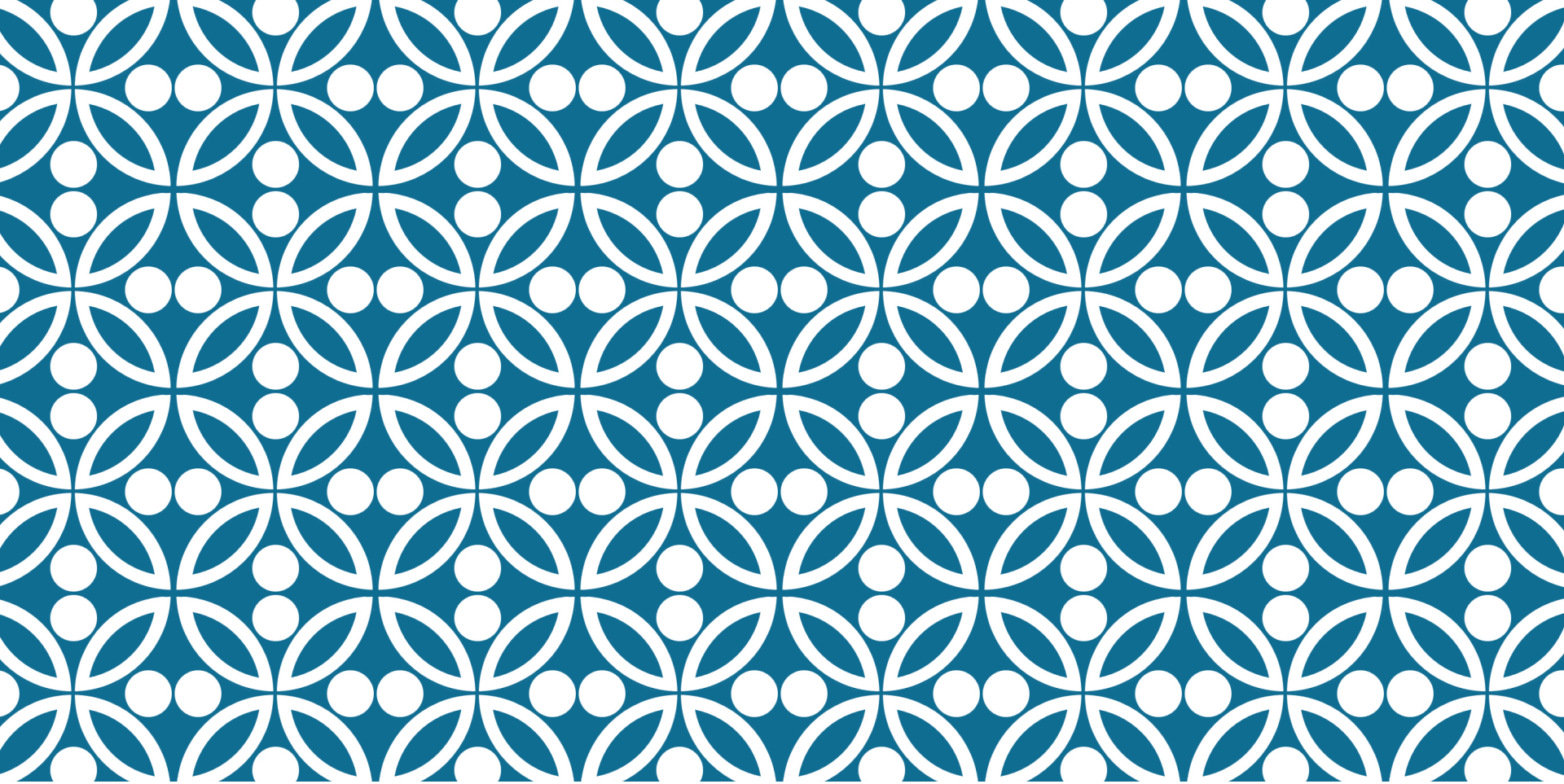
Os processos que não pretendam aderir a nenhum novo comunicador devem indicar em **split_key** a constante **MPI_UNDEFINED**.

PROCESSOS PARES E ÍMPARES (MPI SPLIT.C)

```
MPI_Comm split_comm;
...
MPI_Comm_split(MPI_COMM_WORLD, my_rank % 2, my_rank, &split_comm);
MPI_Comm_rank(split_comm, &split_rank);
printf("Rank: world %d split %d \ n", my_rank, split_rank);
MPI_Comm_free(&split_comm);
...
```

Se executarmos o exemplo com 5 processos obtemos o seguinte output:

Rank: world 0 split 0
Rank: world 1 split 0
Rank: world 2 split 1
Rank: world 3 split 1
Rank: world 4 split 2



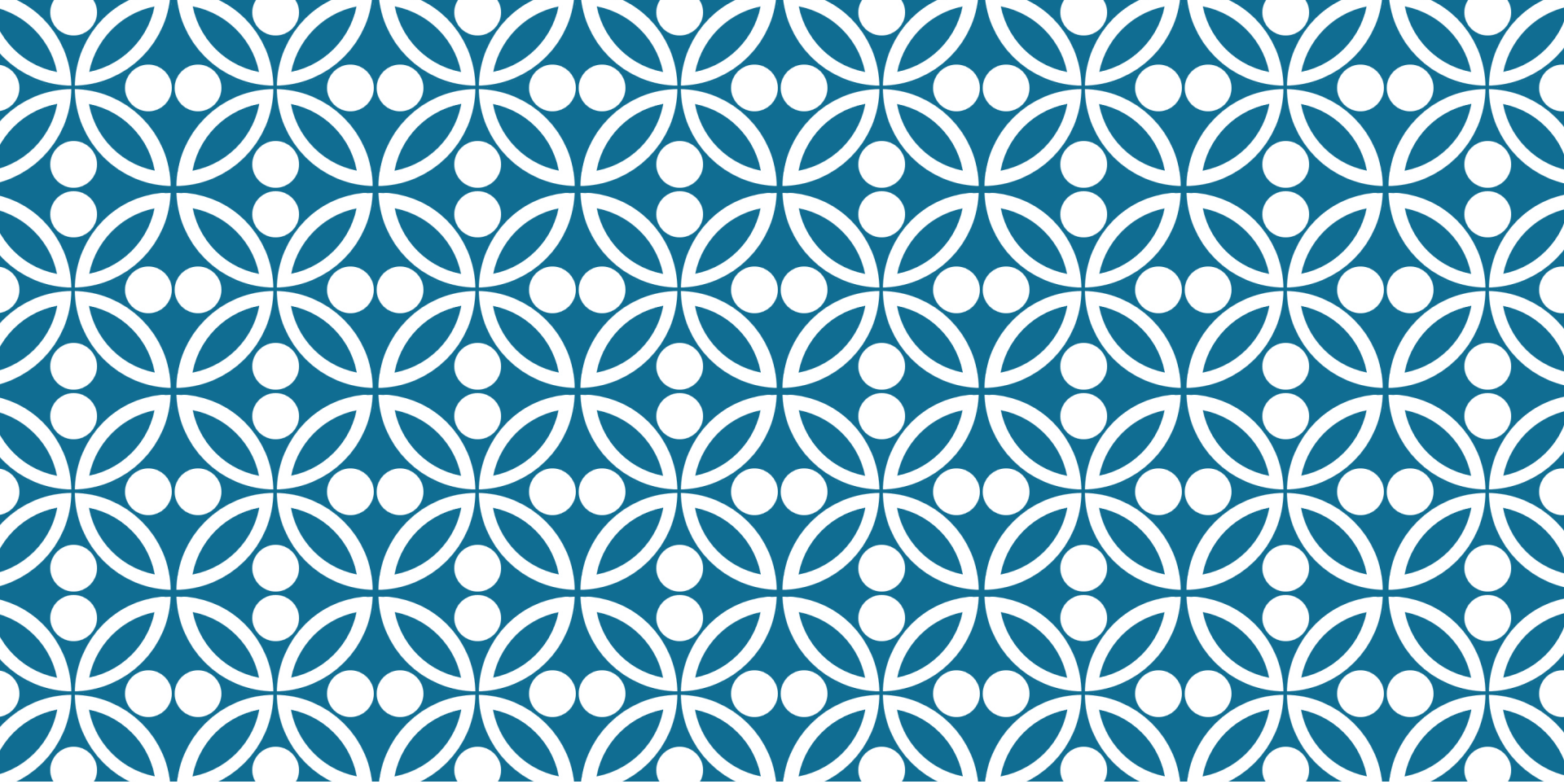
SINCRONIZADOR (BARREIRA)

BARRIER

```
int MPI_Barrier(MPI_Comm comm)
```

MPI_Barrier() sincroniza todos os processos no comunicador **comm**. Cada processo bloqueia até que todos os processos no comunicador chamem **MPI_Barrier()**.

A variação **MPI_Ibarrier()** avisa quando todos os processos passaram pela barreira, mas não força os processos a aguardarem pelos demais.



AVALIANDO O TEMPO DE EXECUÇÃO E SINCRONIZAÇÃO

MEDINDO O TEMPO DE EXECUÇÃO

```
double MPI_Wtime(void)
```

MPI_Wtime() retorna o tempo em segundos que passou desde um determinado ponto arbitrário no passado.

```
double MPI_Wtick(void)
```

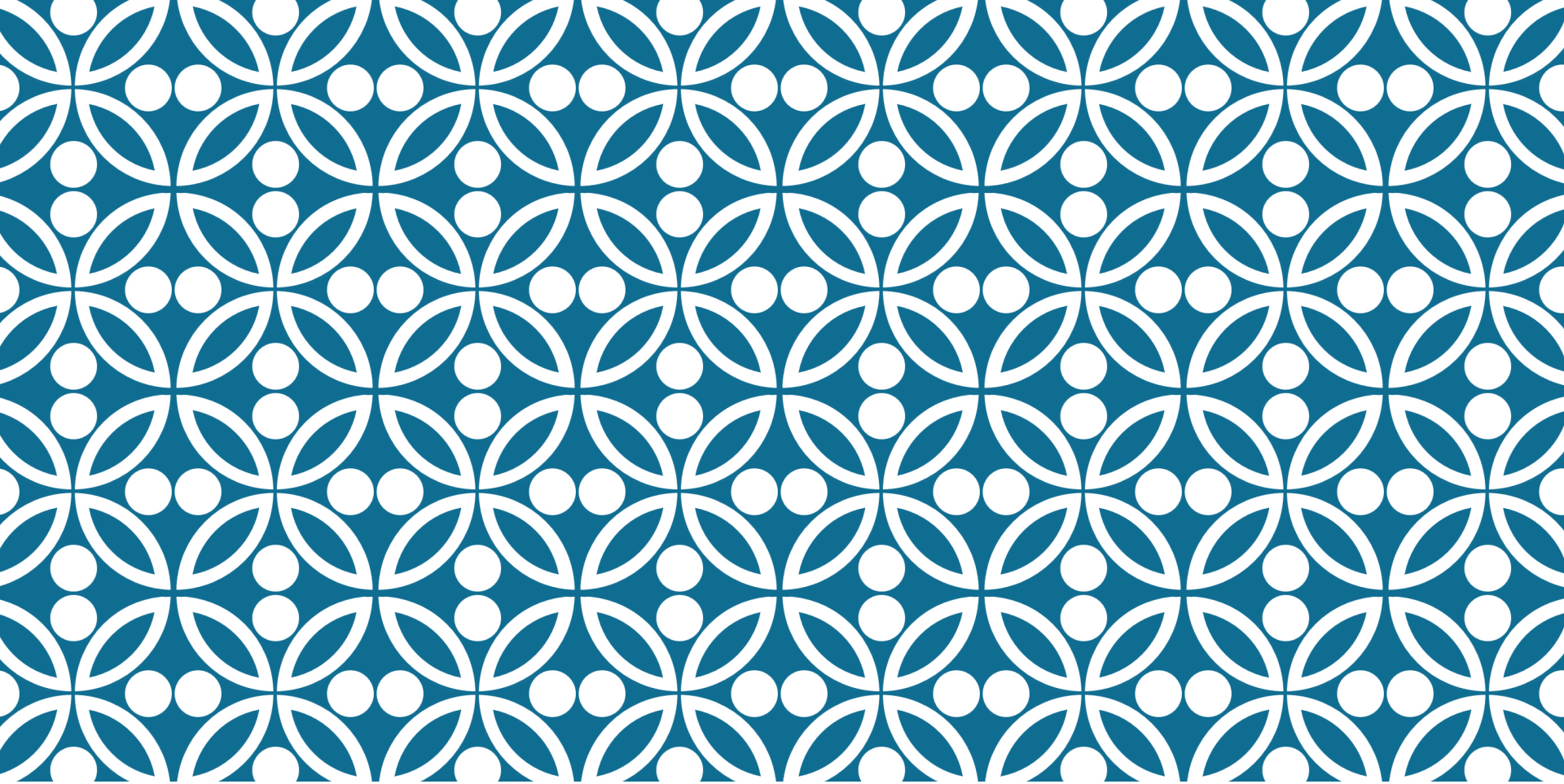
MPI_Wtick() retorna a precisão da função **MPI_Wtime().** Por exemplo, se **MPI_Wtime()** for incrementado a cada microsegundo então **MPI_Wtick()** retorna 0.000001. A precisão depende de como o hardware counter do clock for implementado na máquina.

```
MPI_Barrier(MPI_Comm comm)
```

MPI_Barrier() sincroniza todos os processos no comunicador.

MEDIR O TEMPO DE EXECUÇÃO (MPI_TIME.C)

```
double start, finish;
...
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
...
  // Parte da execução a medir ... (Muitos comandos)
...
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
if (my_rank == 0)
  printf("Tempo de Execução: %f segundos \ n", finish - start);
...
// Os valores devolvidos por MPI_Wtime() são em tempo real, ou seja, todo o
tempo que o processo possa ter estado interrompido pelo sistema é
igualmente contabilizado.
```



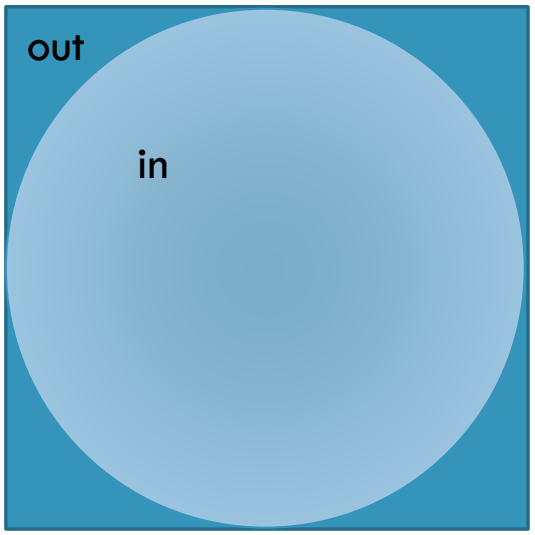
EXEMPLO COMPLETO

CÁLCULO DE π

O valor de π pode ser calculado por aproximação utilizando o método de *Monte Carlo*

A ideia é a seguinte:

- Gerar N pontos aleatórios (x, y).
- Para cada ponto (x, y) verificar se $(x^2 + y^2) < 1$ e em função disso incrementar **in** ou **out**.
- Calcular o valor aproximado de π como $\frac{4*in}{in+out}$



$$\frac{\text{área} \bigcirc}{\text{área} \square} = \frac{\pi}{4}$$

CÁLCULO DE π

Como proceder em paralelo?

Definir um dos processos como servidor de sequências de números aleatórios.

Com os restantes processos definir um novo comunicador de clientes.

Os clientes pedem sucessivamente sequências de números ao servidor, verificam onde caem os pontos resultantes de cada par de números e propagam essa informação aos restantes clientes.

Quando o total de pontos processados for superior a N a computação termina e um dos processos escreve o resultado aproximado do cálculo de π.

CÁLCULO DE π (MPI_PI.C)

```
...
// define novo comunicador para os clientes
MPI_Comm_group(MPI_COMM_WORLD, &world_group);
ranks[0] = SERVER;
MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(MPI_COMM_WORLD, worker_group, &workers_comm);
MPI_Group_free(&worker_group);
MPI_Group_free(&world_group);
// obtém o número de pontos e propaga a todos
if (my_rank == ROOT) scanf("%d", &total_points);
MPI_Bcast(&total_points, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
// inicia a contagem do tempo
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
// calcula o valor aproximado de PI
if (my_rank == SERVER) { ... // servidor } else { ... // cliente }
// termina a contagem do tempo e escreve resultado
MPI_Barrier(MPI_COMM_WORLD);
finish = MPI_Wtime();
if (my_rank == ROOT) {
  printf("PI = %.20f \ n", (4.0 * total_in) / (total_in + total_out));
  printf("Tempo de Execução = %f segundos \ n", finish - start);
}
...
```

CÁLCULO DE π (MPI_PI.C)

```
// servidor
if (my_rank == SERVER) {
  do {
    MPI_Recv(&req_points, 1, MPI_INT, MPI_ANY_SOURCE,
             REQUEST, MPI_COMM_WORLD, &status);
    if (req_points) {
      for (i = 0; i < req_points; i++)
        rands[i] = random();
    MPI_Send(rands, req_points, MPI_INT, status.MPI_SOURCE,
             REPLY, MPI_COMM_WORLD);
     }
  } while (req_points);
}
```

CÁLCULO DE π (MPI_PI.C)

```
// cliente
if (my_rank != SERVER) {
  in = out = 0;
  do {
    req_points = REQ_POINTS;
    MPI_Send(&req_points, 1, MPI_INT, SERVER, REQUEST, MPI_COMM_WORLD);
    MPI_Recv(rands, req_points, MPI_INT, SERVER, REPLY, MPI_COMM_WORLD,
&status);
    for (i = 0; i < req_points; i += 2) {
      x = (((double) rands[i]) / RAND_MAX) * 2 - 1;
      y = (((double) rands[i+1]) / RAND_MAX) * 2 - 1;
      (x * x + y * y < 1.0) ? in++ : out++;
    }
    MPI_Allreduce(&in, &total_in, 1, MPI_INT, MPI_SUM, workers_comm);
    MPI_Allreduce(&out, &total_out, 1, MPI_INT, MPI_SUM, workers_comm);
    req_points = (total_in + total_out < total_points);
    if (req_points == 0 && my_rank == ROOT)
      MPI_Send(&req_points, 1, MPI_INT, SERVER, REQUEST, MPI_COMM_WORLD);
  } while (req_points);
  MPI_Comm_free(&workers_comm);
}
```