MPI - *Message Passing Interface*[1]

João Marcelo Uchôa de Alencar
joao.marcelo@ufc.br
UFC-Quixadá

[1] https://computing.llnl.gov/tutorials/mpi/

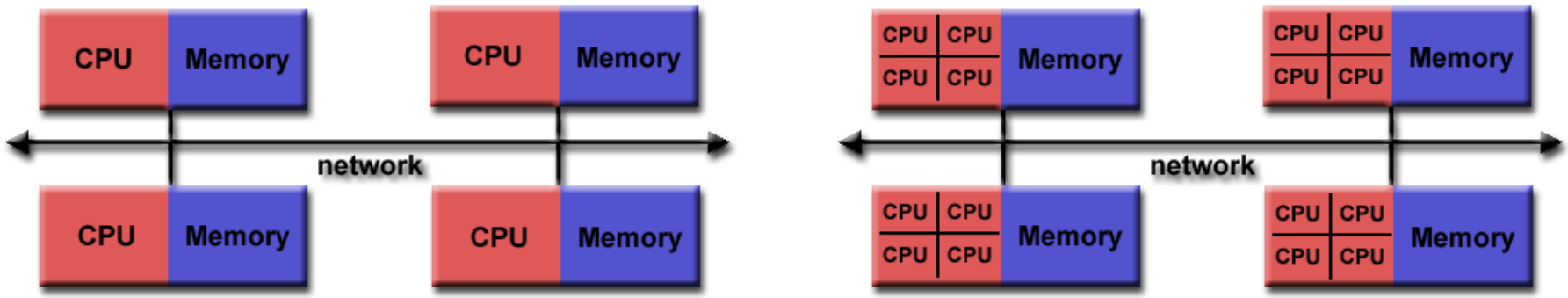
Introdução

M*essage* **P***assing* **I***nterface*

- É um padrão para uma biblioteca de troca de mensagens;
- definido pelos maiores *players* da indústria e academia;
- portável, eficiente e flexível;
- define uma interface que cada fabricante ou grupo decide como implementar;
- modelo de programação de troca de mensagens;

Não é uma biblioteca, mas uma especificação de o que deve estar presente na biblioteca.

Modelo de Programação



Não importa a arquitetura subjacente, o MPI adota uma abstração de memória distribuída.

- Foi desenvolvido pensando em *clusters* de processadores independentes;
- mas pode ser usado em máquinas de vários núcleos.

Razões para o Uso de MPI

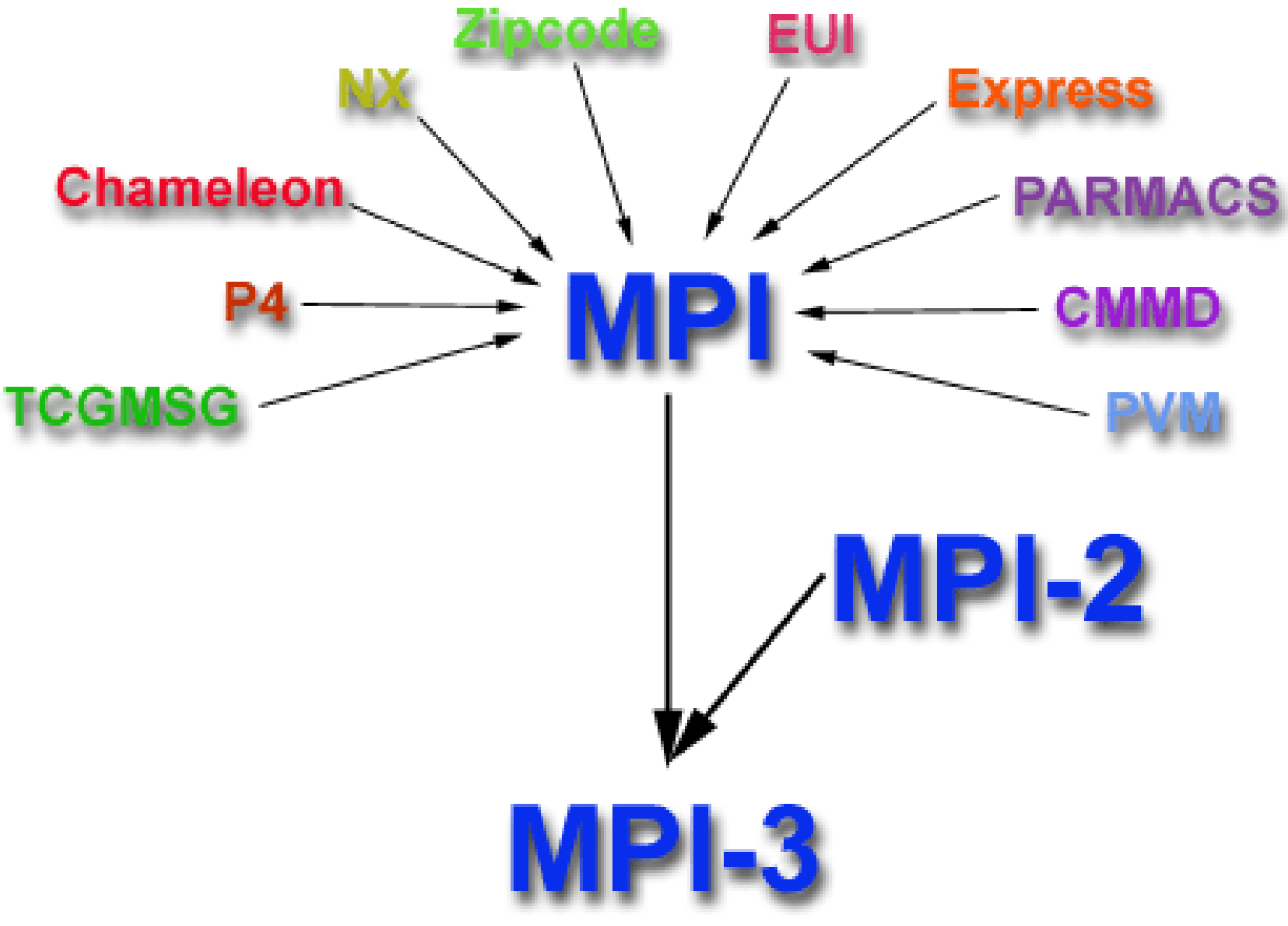
- Padronização;
- portabilidade;
- otimizações;
- funcionalidades;
- disponibilidade ampla.

História e Evolução

Processo de padronização iniciado em 1992.

- 1980-1990: cada fabricante de supercomputadores tinha sua solução;
- abril de 1992: começam os trabalhos do comitê;
- novembro de 1992: esboço da versão 1.0 é liberado;
- novembro de 1993: esboço mais completo apresentado;
- maio de 1994: versão final lançada;
- 1998: versão 2.0 lançada;
- 2012: versão 3.0 lançada.

História e Evolução



Múltiplas Implementações

Apesar da padronização do MPI, várias implementações existem.

- OpenMPI;
- MVAPICH;
- LAM/MPI;
- Intel MPI;
- dentro outros...

O importante é saber qual versão do padrão a implementação que você escolher utiliza. Nós vamos utilizar a OpenMPI, pois está disponível no Ubuntu.

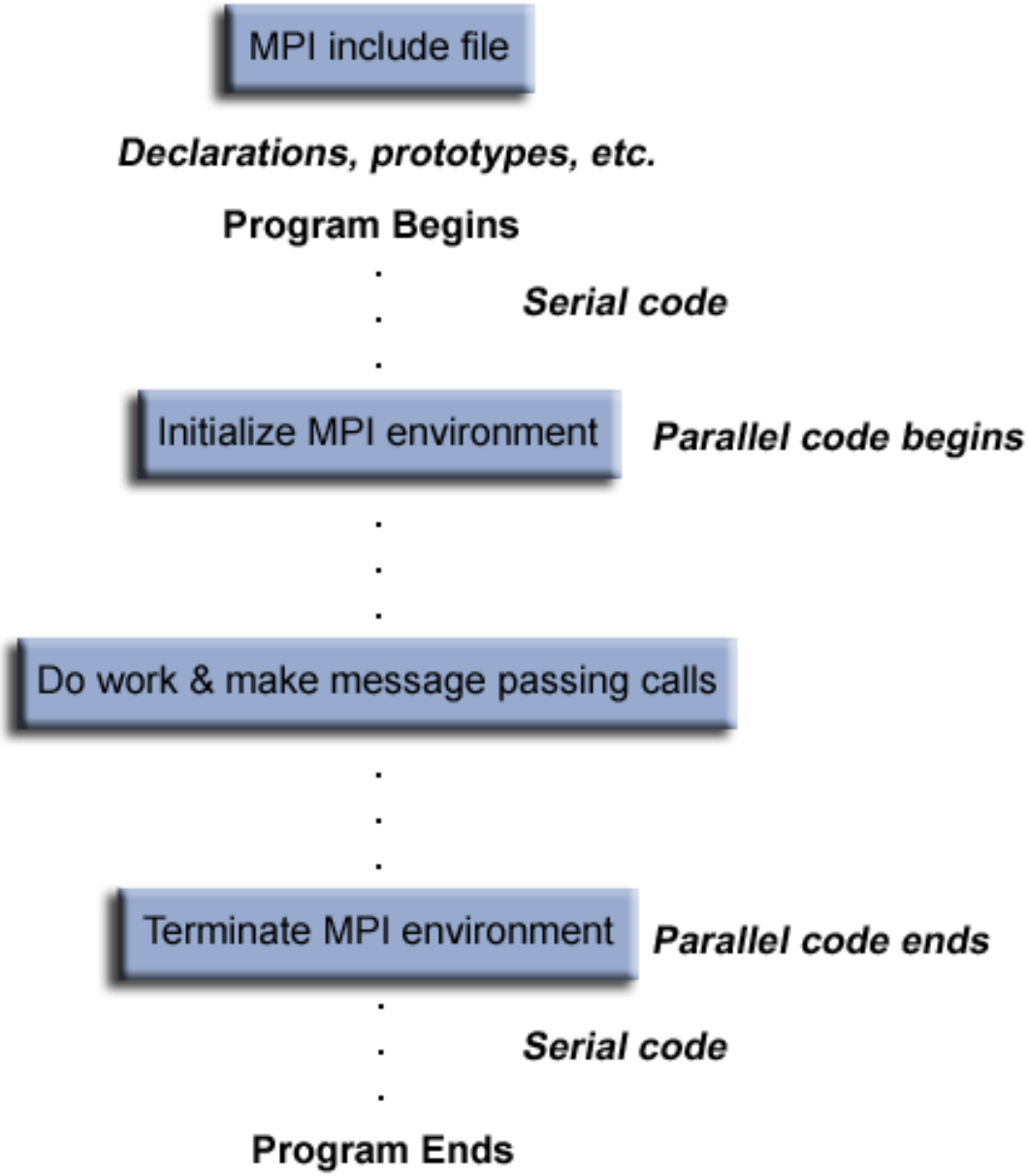
Instalação da OpenMPI no Ubuntu 18.04

```
$ sudo apt install libopenmpi-dev
```

Instalando em apenas uma máquina, se nenhum parâmetro for informado, o número de processos criados será igual ao número de processadores ou núcleos disponíveis. Para testar:

```
$ mpirun /bin/hostname
```

Estrutura Geral de um Programa MPI



Cabeçalho e Formato das Chamadas

Cabeçalho

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

Formato das Chamadas

Formato:	rc = MPI_Xxxxx(parâmetro, ...)
Exemplo:	rc = MPI_Bsend(&buf, count, type, dest, tag, comm)
Código Erro:	Retorna rc.MPI_SUCCESS se correta.

Comunicadores e Grupos

- O MPI usa *objetos* chamados comunicadores e grupos para definir quais processos podem se comunicar entre si;
- na maioria das rotinas MPI, o programador precisa informar qual comunicador é utilizado;
- por padrão, o comunicador **MPI_COMM_WORLD** inclui todos os processos;
- dentro um comunicador, cada processo tem um número inteiro identificador, chamado **rank**, usado para selecionar o destino e a origem na troca de mensagens;

Tratamento de Erros

- A maioria das rotinas retorna um código de erro;
- porém, de acordo com o padrão, o comportamento no caso de erro é abortar;
- há uma maneira de mudar esse comportamento, mas na prática não é comum ser aplicado;
- cada implementação exibe mensagens diferentes para os erros possíveis.

Rotinas para Gestão do Ambiente

```
/* Inicializa o ambiente MPI. */
MPI_Init(&argc, &argv)
/* Retorna a quantidade de processos
   em um comunicador. */
MPI_Comm_size(comm, &size)
/* Retorna o identificador do processo. */
MPI_Comm_rank (comm,&rank)
/* Abortar todos os processos. */
MPI_Abort(comm, errorcode)
/* Retorna o tempo decorrido desde o
   início da execução. */
MPI_Wtime()
/* Finaliza o ambiente MPI. */
MPI_Finalize()
```

Olá Mundo em MPI

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

int main(int argc, char *argv[]) {
   int  numtasks, rank, len, rc;
   char hostname[MPI_MAX_PROCESSOR_NAME];

   MPI_Init(&argc,&argv);

   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   MPI_Get_processor_name(hostname, &len);
   printf ("Número de processos: %d Meu rank: %d Executando em %s\n",
            numtasks, rank, hostname);

   // Realizar trabalho....

   MPI_Finalize();
}
```

Olá Mundo em MPI

```
$ mpicc OlaMundo.c -o OlaMundo
$ mpirun ./OlaMundo
$ mpirun -n 4 ./OlaMundo
```

Comunicação Ponto a Ponto

Envolve troca de mensagem entre apenas dois processos.

- Envio síncrono;
- envio bloqueante, recebimento bloqueante;
- envio não bloqueante, recebimento não bloqueante;
- envio *bufferizado*;
- dentro outras modalidades...

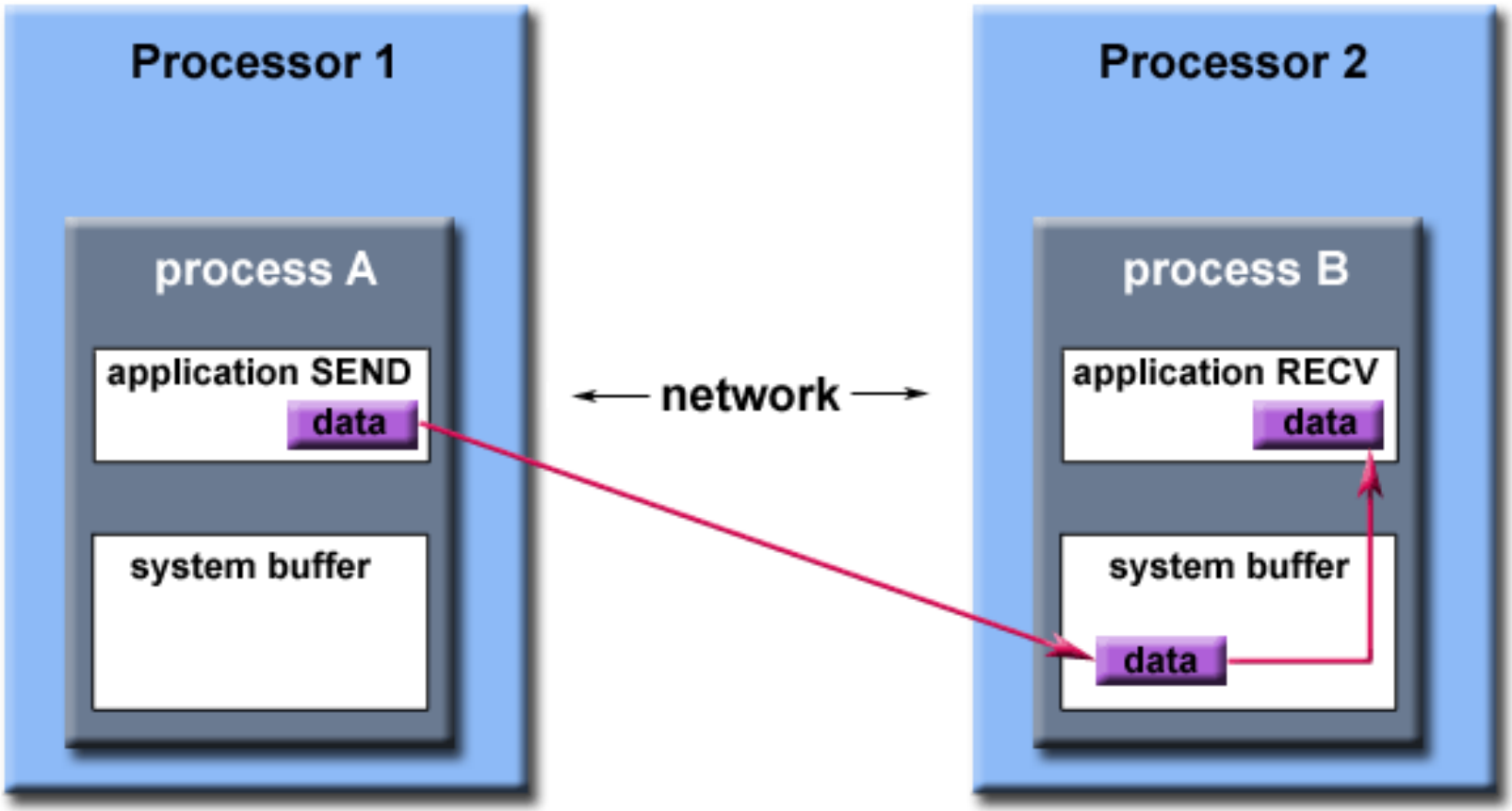
Cada tipo de envio pode ser pareado com qualquer tipo de recebimento. Também existem rotinas para verificar o *status* da transferência.

Buffers

- Processos não podem ser sincronizados de forma exata quando em execução em processadores diferentes;
- é preciso armazenar os dados da mensagem quando os processos comunicantes estão fora de sintonia;
- considere os cenários:
  - Uma operação de envio inicia 5 segundos antes da operação de recebimento;
  - vários envios chegam a um processo que consegue emitir um recebimento por vez.
- o que fazer com a mensagem enquanto os processos se acomodam?

As implementações MPI podem utilizar *buffers* do sistema para armazenar mensagens em trânsito.

Buffers



Path of a message buffered at the receiving process

Buffers

Os *buffers* de sistema são:

- Invisíveis ao programador, gerenciados pela biblioteca;
- tem tamanho finito, podem ser esgotados;
- seu funcionamento varia por implementação, nem sempre bem documentado;
- podem existir no lado do envio, do recebimento, ou em ambos;
- permitem a assicronia.

As variáveis no programa que guardam os dados a serem transmitidos, ou nas quais os dados recebidos são armazenados, é o *buffer* da aplicação.

Chamadas Bloqueantes

- Um envio bloqueante só retorna quando o *buffer* de aplicação pode ser reutilizado. Não significa que os dados foram enviados;
- um envio bloqueante síncrono garante que o receptor iniciou a transferência;
- um envio bloqueante assíncrono transfere os dados para o *buffer* de sistema do emissor e retorna;
- um recebimento bloqueante só retorna quando os dados já estão prontos para serem usados.

Chamadas Não Bloqueantes

- Envios e recebimentos não bloqueantes retornam de imediato, sem nenhuma garantia de cópia do *buffer* de aplicação para o *buffer* do sistema;
- são mais uma requisição para uma operação do que a operação em si;
- não é seguro utilizar as variáveis usadas no envio/recebimento enquanto a operação não for de fato realizada;
- sobrepor computação e comunicação.

Em outras palavras, a sincronia considera a transferência entre dois processos, o bloqueio considera a possibilidade de reutilizar os *buffers*.

Bloqueantes *versus* Não Bloqueantes

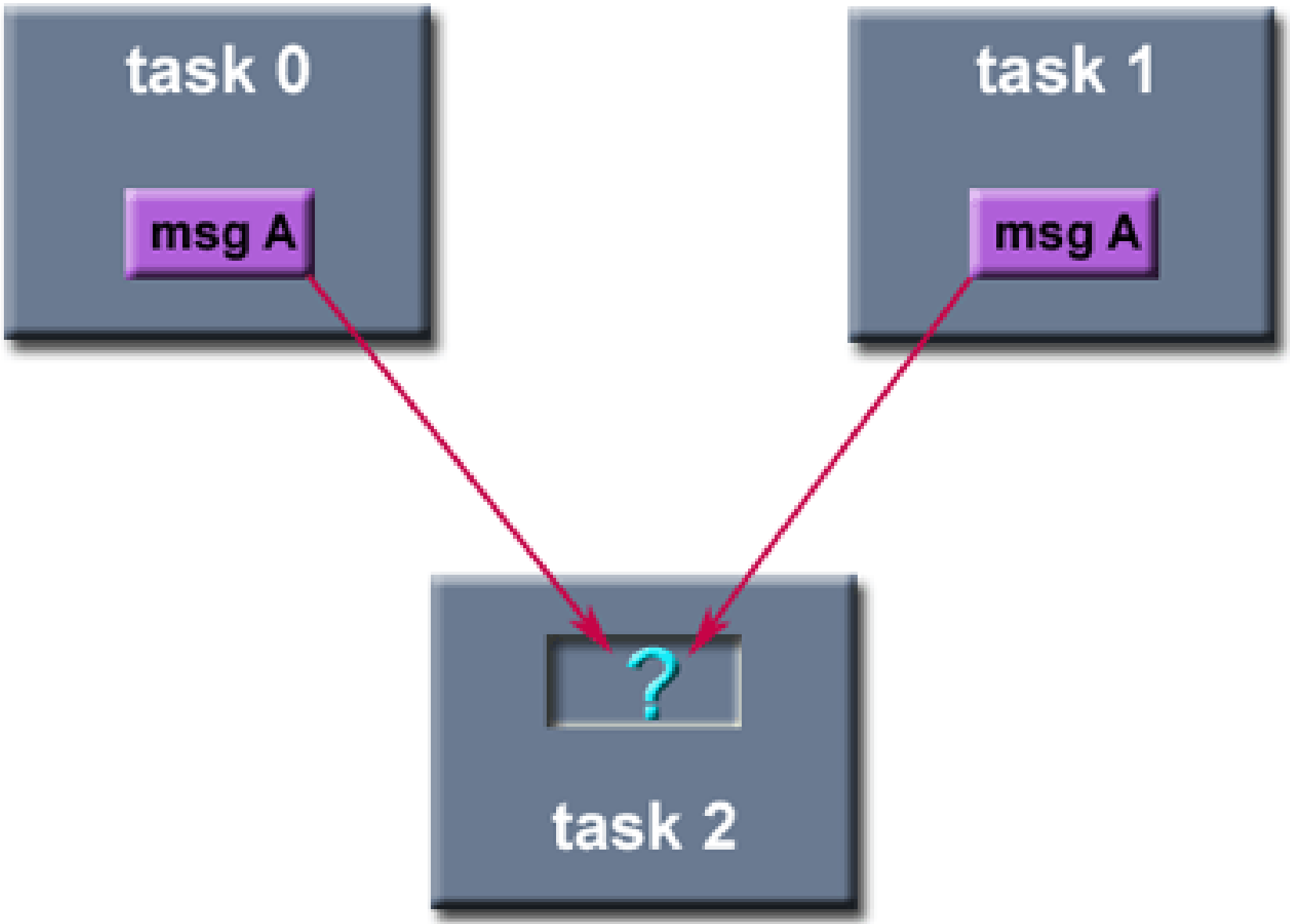
```
// Bloqueante
myvar = 0;
for (i = 0; i  < ntasks; i++) {
   task = i;
   MPI_Send(&myvar, ... ...., task, ...);
   myvar = myvar + 2;
   /* Fazer alguma coisa... */
}

// Não Bloqueante
myvar = 0;
for (i = 0; i  < ntasks; i++) {
   task = i;
   MPI_Isend(&myvar, ... ...., task, ...);
   myvar = myvar + 2;
   /* Fazer alguma coisa... */
   MPI_Wait(...);
}
```

Ordem de Entrega

- O MPI garante que as mensagens não irão se sobreescrever;
- se o emissor enviar mensagens 0 e 1, nesta ordem, para um mesmo receptor, elas chegarão na mesma ordem, 0 e depois 1;
- se o receptor invocar dois recebimentos sucessivamente, para a mesma mensagem, o segundo só ocorre após o primeiro;
- se além dos processos do MPI, *threads* forem utilizados, a ordem não é garantida.

Justiça na Entrega



Envio Bloqueante

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

- *smessage*: *buffer* de aplicação;
- *count*: número de elementos do *buffer*;
- *datatype*: tipo de dados;
- *dest*: *rank* do destinatário;
- *tag*: rótulo para distinguir mensagens;
- *comm*: comunicador.

A versão síncrona é a *MPI_Ssend*.

Recebimento Bloqueante

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

- *rmessage*: *buffer* de aplicação;
- *count*: número de elementos do *buffer*;
- *datatype*: tipo de dados;
- *source*: *rank* do emissor;
- *tag*: rótulo para distinguir mensagens;
- *comm*: comunicador;
- *status*: contém informações de como a transferência ocorreu.

Tipos de Dados MPI

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

Status do Recebimento

- *source* = *MPI_ANY_SOURCE*, permite receber de qualquer origem;
- *tag* = *MPI_ANY_TAG*, permite receber mensagens com qualquer *tag*;
- *status.MPI_SOURCE* permite descobrir qual a origem;
- *status.MPI_TAG* permite descobrir a *tag*;

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

MPI_Get_count permite saber quantos *bytes* foram transferidos.

Exemplo de Envio e Recebimento Bloqueantes

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

int main (int argc, char *argv[]) {
   int rank, size;
   int value = 0;
   MPI_Status status;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   if (rank == 0) {
       value++;
       MPI_Send(&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
       printf("Processo %d: %d.\n", rank, value);
   } else if (rank == 1) {
       MPI_Recv(&value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
       value++;
       printf("Processo %d: %d.\n", rank, value);
   }

   MPI_Finalize();
   return 0;
}
```

Exemplo: Soma de Dois Vetores

Inicializando o ambiente.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size, vetorTamanho;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    vetorTamanho = atoi(argv[1]);
```

Exemplo: Soma de Dois Vetores - *Rank* 0

```
// Cria os vetores
srand(time(0));
float *vetorA = (float *) malloc(vetorTamanho * sizeof(float));
float *vetorB = (float *) malloc(vetorTamanho * sizeof(float));
float *vetorC = (float *) malloc(vetorTamanho * sizeof(float));
float valorA = (rand() % 100) / 3.0;
float valorB = (rand() % 100) / 3.0;
printf("rank %d: vou preencher vetorA com %.2f e vetor B com  %.2f.\n", rank, valorA, valorB);
for (int i = 0; i < vetorTamanho; i++) {
   vetorA[i] = valorA;
   vetorB[i] = valorB;
   vetorC[i] = 0.0;
}

// Envia metade dos vetores para o rank 1
int trabalhoLocal = vetorTamanho / 2;
MPI_Send(vetorA + trabalhoLocal, trabalhoLocal, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
MPI_Send(vetorB + trabalhoLocal, trabalhoLocal, MPI_FLOAT, 1, 1, MPI_COMM_WORLD);

// Calcula sua parte
for (int i = 0; i < trabalhoLocal; i++)
   vetorC[i] = vetorA[i] + vetorB[i];

// Recebe a parte do rank 1
MPI_Status status;
MPI_Recv(vetorC + trabalhoLocal, trabalhoLocal, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, &status);

// Imprime o resultado da soma
printf("Vetor Resultado: \n");
for (int i = 0; i < vetorTamanho; i++)
   printf("%.2f ", vetorC[i]);
printf("\n");
```

Exemplo: Soma de Dois Vetores - *Rank* 1

```
// Aloca espaço para os vetores
int trabalhoLocal = vetorTamanho / 2;
float *vetorALocal = (float *) malloc(trabalhoLocal * sizeof(float));
float *vetorBLocal = (float *) malloc(trabalhoLocal * sizeof(float));
float *vetorCLocal = (float *) malloc(trabalhoLocal * sizeof(float));

// Recebe os vetores
MPI_Status status;
MPI_Recv(vetorALocal, trabalhoLocal, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status);
MPI_Recv(vetorBLocal, trabalhoLocal, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &status);

// Faz seu trabalho
for (int i = 0; i < trabalhoLocal; i++)
   vetorCLocal[i] = vetorALocal[i] + vetorBLocal[i];

// Envia para rank 0
MPI_Send(vetorCLocal, trabalhoLocal, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
```

Deadlocks

O que acontece com o trecho de código a seguir?

```
MPI_Comm_rank(comm, &my_rank);
if (my_rank == 0) {
   MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);
   MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
} else if (my_rank == 1) {
   MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Lembrando que *MPI_Send*() e *MPI_Recv*() são bloqueantes e assíncronas.

Deadlocks

A seguinte alteração resolve o problema?

```
MPI_Comm_rank(comm, &my_rank);
if (my_rank == 0) {
   MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);
} else if (my_rank == 1) {
   MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
   MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
```

Funciona independente da implementação?

Deadlocks

Solução independente da implementação:

```
MPI_Comm_rank(comm, &my_rank);
if (my_rank == 0) {
   MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &status);
} else if (my_rank == 1) {
   MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Não necessita de *buffers* de sistema.

Organização de Processos em Anel

```
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

left = (rank - 1 + p) % p;
right = (rank + 1) %p;

sprintf(send_buffer1, "N:%d\n", rank);
sprintf(send_buffer2, "N:%d\n", rank);

MPI_Send(send_buffer1, size, MPI_CHAR, left, tag, MPI_COMM_WORLD);
MPI_Recv(recv_buffer1, size, MPI_CHAR, right, tag, MPI_COMM_WORLD, &status);

MPI_Send(send_buffer2, size, MPI_CHAR, right, tag, MPI_COMM_WORLD);
MPI_Recv(recv_buffer2, size, MPI_CHAR, left, tag, MPI_COMM_WORLD, &status);

printf("------------------------\n");
printf("Processo %s", send_buffer1);
printf("Vizinho Direita %s", recv_buffer1);
printf("Vizinho Esquerda %s", recv_buffer2);
printf("------------------------\n");

MPI_Finalize();
```

Evitando *Deadlocks*

O envio e recebimento simultâneo é muito comum, mas com risco de *deadlocks*. A solução é a chamada *Sendrecv*.

```
int 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);
```

Usando essa chamada, o programador não precisa se preocupar com a ordem das operações.

Organização de Processos em Anel com *Sendrecv*

```
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

left = (rank - 1 + p) % p;
right = (rank + 1) %p;

sprintf(send_buffer1, "N:%d\n", rank);
sprintf(send_buffer2, "N:%d\n", rank);

MPI_Sendrecv(send_buffer1, size, MPI_CHAR, left, tag,
      recv_buffer1, size, MPI_CHAR, right, tag, MPI_COMM_WORLD, &status);
MPI_Sendrecv(send_buffer2, size, MPI_CHAR, right, tag,
      recv_buffer2, size, MPI_CHAR, left, tag, MPI_COMM_WORLD, &status);

printf("------------------------\n");
printf("Processo %s", send_buffer1);
printf("Vizinho Direita %s", recv_buffer1);
printf("Vizinho Esquerda %s", recv_buffer2);
printf("------------------------\n");

MPI_Finalize();
```

Organização de Processos em Anel - Não Bloqueante

```
left = (rank - 1 + p) % p;
right = (rank + 1) %p;

sprintf(send_buffer1, "N:%d\n", rank);
sprintf(send_buffer2, "N:%d\n", rank);

MPI_Isend(send_buffer1, size, MPI_CHAR, left, tag,
      MPI_COMM_WORLD, &send_request1);
MPI_Irecv(recv_buffer1, size, MPI_CHAR, right, tag,
      MPI_COMM_WORLD, &recv_request1);

MPI_Isend(send_buffer2, size, MPI_CHAR, right, tag,
      MPI_COMM_WORLD, &send_request2);
MPI_Irecv(recv_buffer2, size, MPI_CHAR, left, tag,
      MPI_COMM_WORLD, &recv_request2);

MPI_Wait(&send_request1, &status);
MPI_Wait(&recv_request1, &status);
MPI_Wait(&send_request2, &status);
MPI_Wait(&recv_request2, &status);
```

Operação *Broadcast*

Em um *broadcast*, um único processo de um grupo de processo envia o mesmo bloco de dados para todos os outros processos.

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

- Todos os processos precisam especificar o mesmo processo raiz;
- devem usar também o mesmo comunicador.

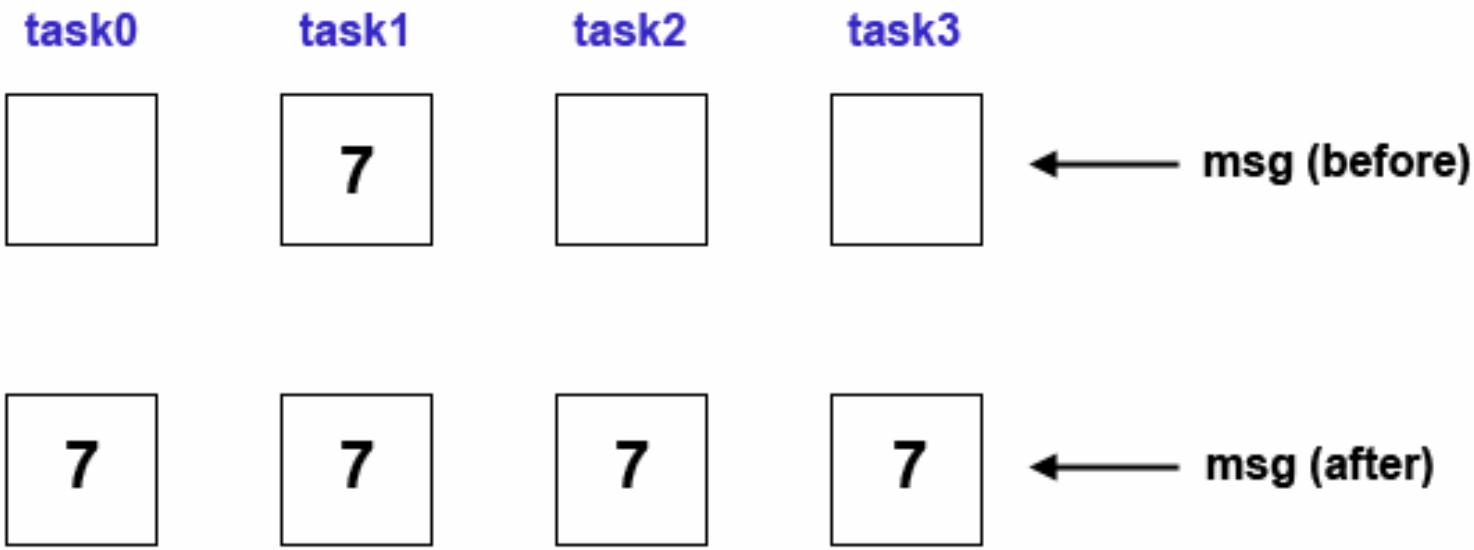
Operação *Broadcast*

MPI_Bcast

Broadcasts a message from one task to all other tasks in communicator

```
count = 1;
source = 1;
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

task1 contains the message to be broadcast



Operação *Broadcast*

- Não há *tags*: os processos receptores não conseguem distinguir entre *broadcasts* diferentes apenas pela *tag*;
- no caso de chamadas sucessivas, a ordem das operações é preservada como se houvesse uma barreira entre elas.
- como todas operações coletivas, a *broadcast* é bloqueante;
- entrentanto, não necessariamente são síncronas, portanto a não garante que os processos estejam sincronizados;
- tudo depende da existência ou não de *buffers* de sistema e seu tamanho.

Operação *Broadcast*

O que acontece no exemplo a seguir?

```
if (my_rank == 0) {
   MPI_Bcast(&x, 1, MPI_INT, 0, comm);
   MPI_Bcast(&y, 1, MPI_INT, 0, comm);
   local_work();
} else if (my_rank == 1) {
   local_work();
   MPI_Bcast(&y, 1, MPI_INT, 0, comm);
   MPI_Bcast(&x, 1, MPI_INT, 0, comm);
} else if (my_rank == 2) {
   local_work();
   MPI_Bcast(&x, 1, MPI_INT, 0, comm);
   MPI_Bcast(&y, 1, MPI_INT, 0, comm);
}
```

Operação Redução

Cada processo participante fornece um bloco de dados que é combinado com outros blocos usando uma operação de redução binária.

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

- A operação precisa ser associativa;
- as operações fornecidas por padrão também são comutativas.

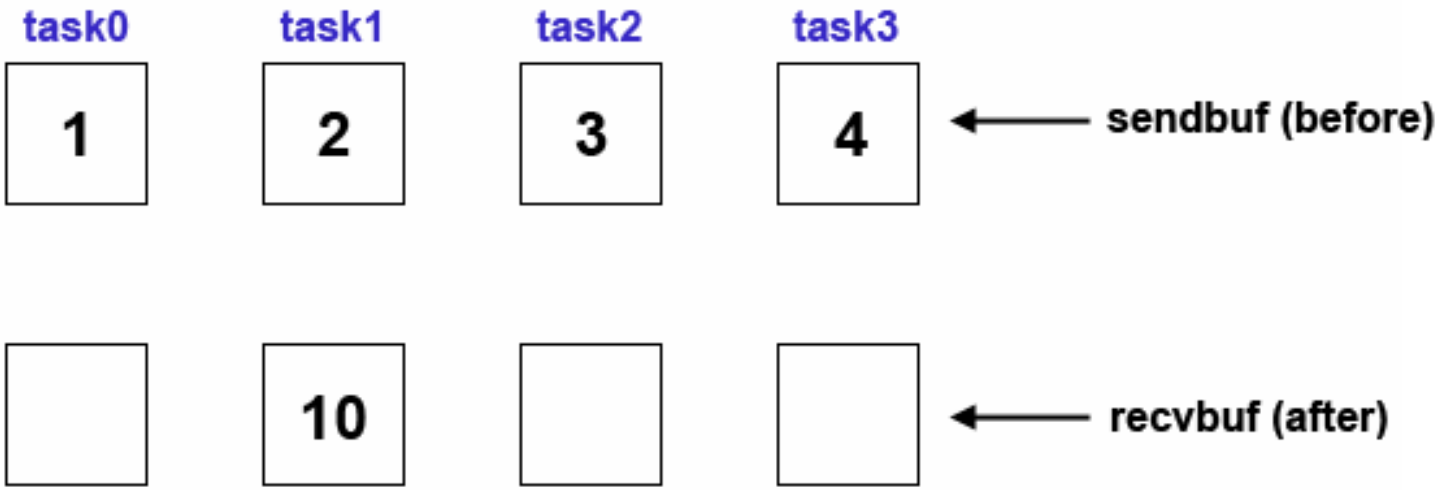
Operação Redução

MPI_Reduce

Perform reduction across all tasks in communicator and store result in 1 task

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```

task1 will contain result



Operação Redução

Representação	Operação
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Soma
MPI_PROD	Produto
MPI_LAND	E lógico
MPI_BAND	E lógico *bits*
MPI_LOR	OU lógico
MPI_BOR	OU lógico *bits*
MPI_LXOR	OU exclusivo lógico
MPI_BXOR	OU exclusivo lógico *bits*
MPI_MAXLOC	Valor máximo e índice correspondente
MPI_MINLOC	Valor mínimo e índice correspondente

Operação *Gather*

Cada processo fornece um bloco de dados para o processo raiz. Para p processos, o bloco de dados coletado na raiz é p vezes maios que os blocos individuais em cada processo.

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

- Os dados são organizados no *recvbuf* na ordem dos *ranks* dos processos;

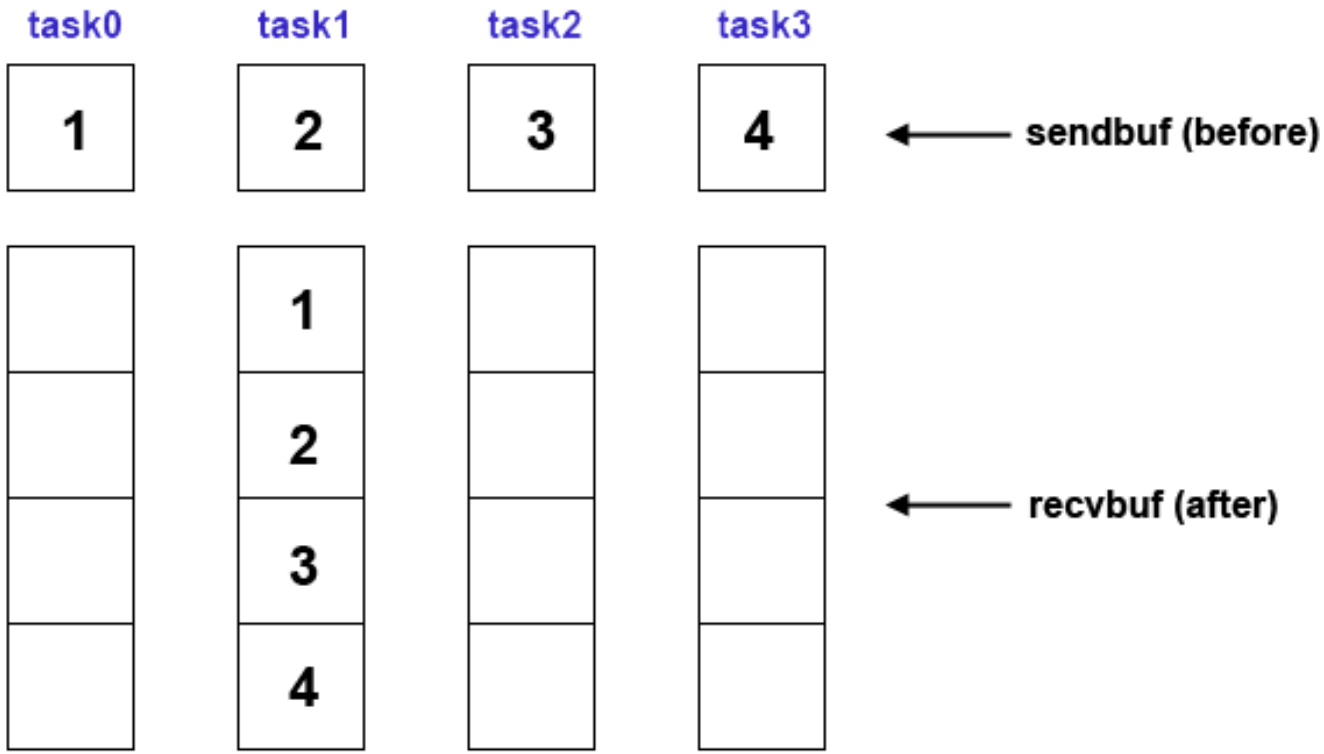
Operação *Gather*

MPI_Gather

Gathers data from all tasks in communicator to a single task

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT
           recvbuf, recvcnt, MPI_INT
           src, MPI_COMM_WORLD);
```

message will be gathered into task1



Operação *Gather*

Variação da *gather* que permite receber um número diferente de elementos de cada processo.

```
int MPI_Gatherv(void *sendbuf, int sendcount,
                MPI_Datatype sendtype,
                void *recvbuf, int *recvcounts,
                int *displs,
                MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

- O parâmetro *sendcount* no processo i deve ser igual ao valor de $recvcounts[i]$ no processo raiz;
- *displs* especifica qual posição em *recvbuf* do processo raiz o bloco de dados do processo i é armazenado.

Operação *Scatter*

Um processo raiz fornece um bloco de dados diferentes para cada processo participante.

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

- Em *sendbuf*, os blocos são ordenados em ordem do *rank* do processo que irá recebê-lo;
- o processo raiz também precisa fornecer um *recvbuf*;
- também existe a versão *Scatterv*.

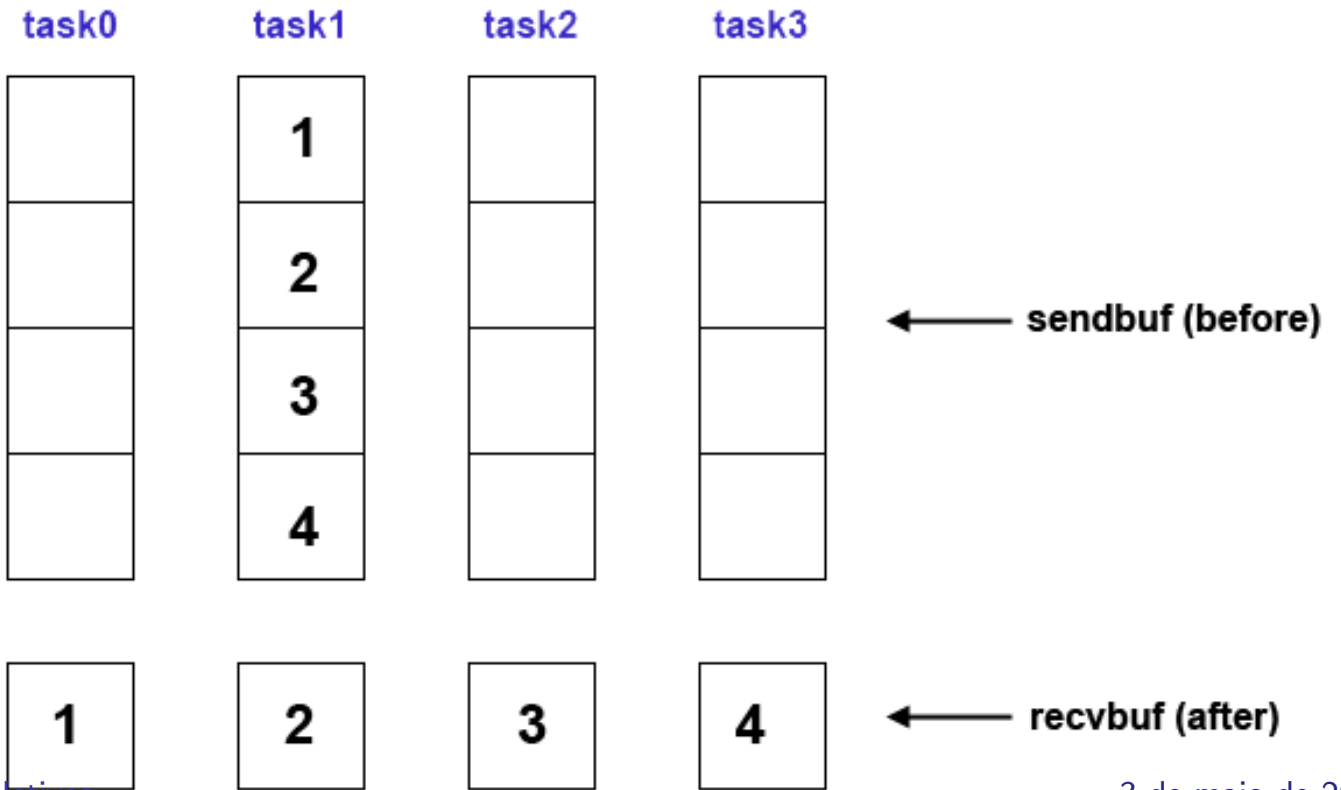
Operação *Scatter*

MPI_Scatter

Sends data from one task to all other tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
src = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT
            recvbuf, recvcnt, MPI_INT
            src, MPI_COMM_WORLD);
```

task1 contains the data to be scattered



Operação *Multi-broadcast*

Cada processo fornece um mesmo bloco de dados que é copiado para todos os outros.

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

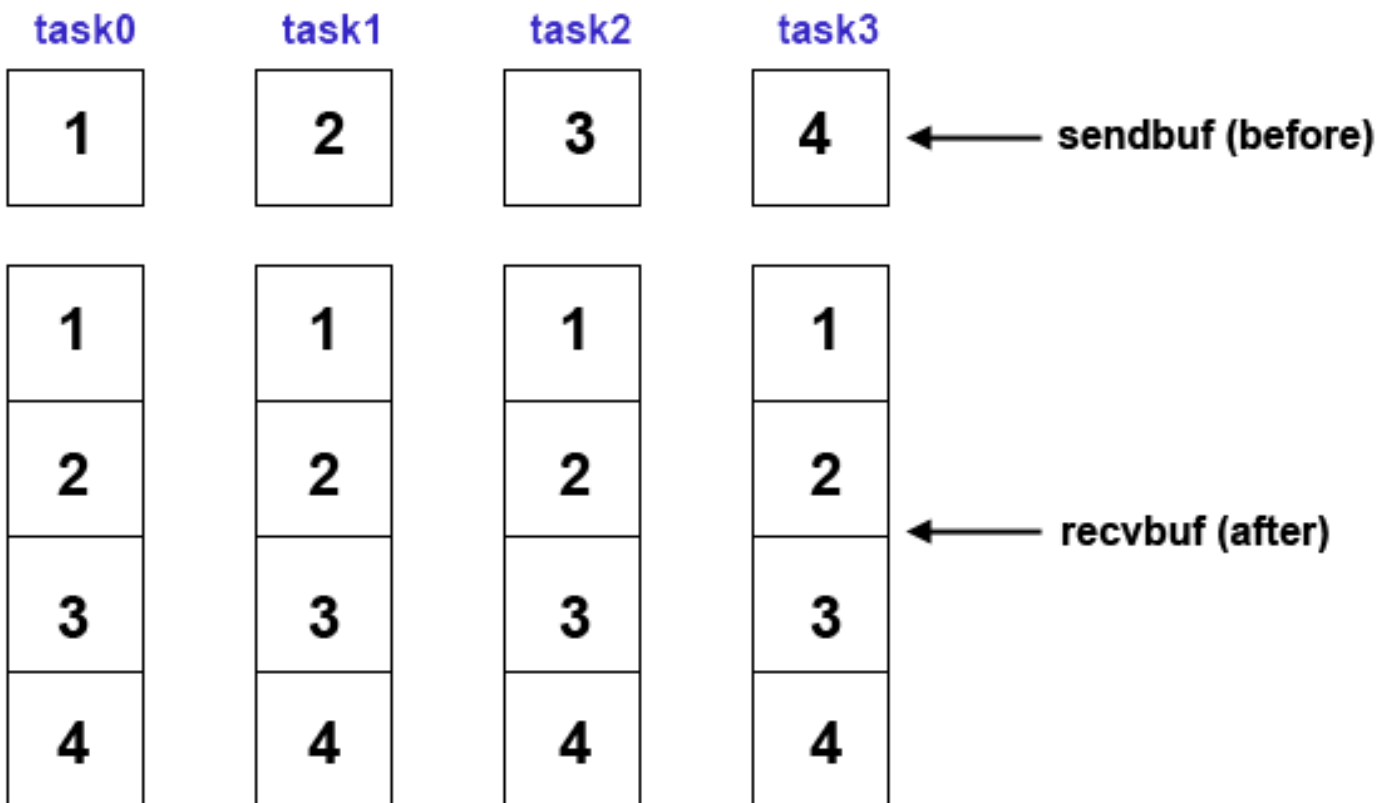
- Todos os blocos serão armazenados em *recvbuf*, existente em cada processo;
- em *recvbuf*, os blocos estão na ordem dos *ranks*;
- existem uma verssão *MPI_Allgatherv*.

Operação *Multi-broadcast*

MPI_Allgather

Gathers data from all tasks and then distributes to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT
              recvbuf, recvcnt, MPI_INT
              MPI_COMM_WORLD);
```



Operação *Multi-accumulation*

Cada processo realiza uma acumulação com blocos de dados dos outros processos.

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

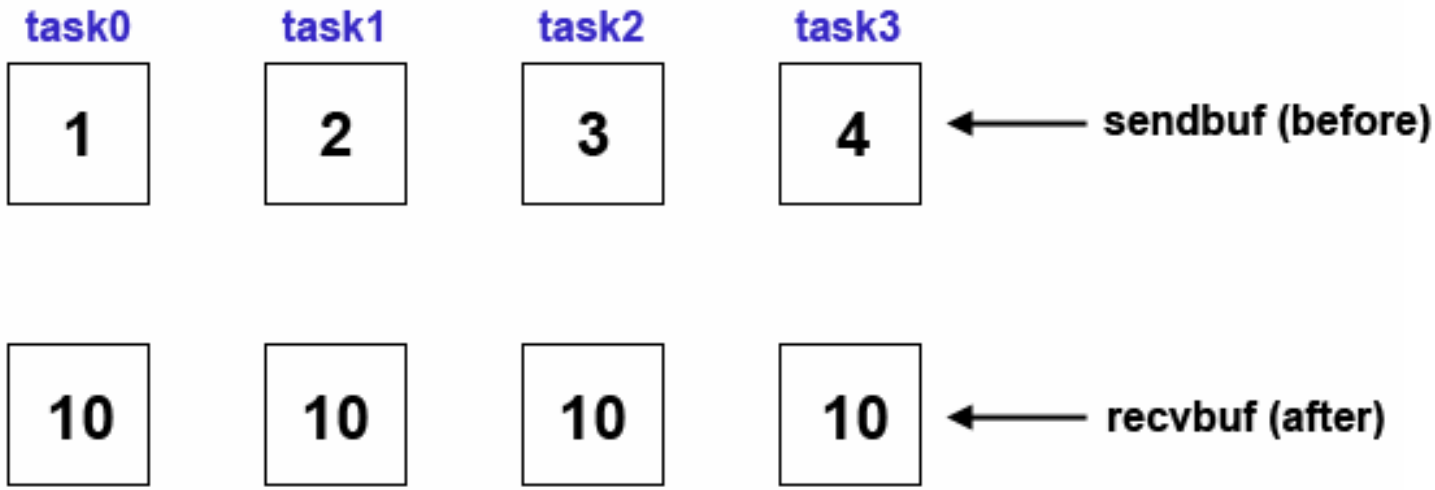
- Cada processo fornece o mesmo bloco de dados para todas as acumulações;
- apenas a operação, todos os processos tem o mesmo resultado.

Operação *Multi-accumulation*

MPI_Allreduce

Perform reduction and store result across all tasks in communicator

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, MPI_INT,
              MPI_SUM, MPI_COMM_WORLD);
```



Operação *Total Exchange*

Cada processo fornece um bloco de dados diferente que é copiado para todos os outros processos.

```
int MPI_Alltoall (void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm)
```

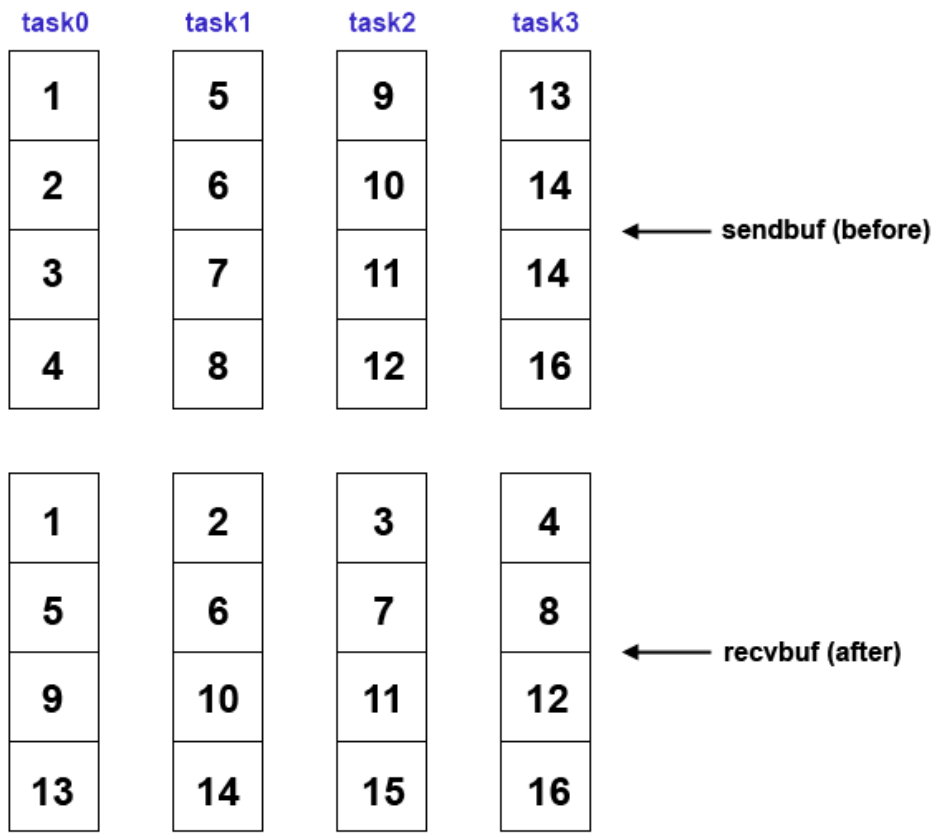
- Em cada processo, *recvbuf* deve ter espaço para todos os blocos;
- os blocos são armazenados em *recvbuf* na ordem dos *ranks*;
- também existe a versão *MPI_Alltoallv*.

Operação *Total Exchange*

MPI_Alltoall

Scatter data from all tasks to all tasks in communicator

```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT
             recvbuf, recvcnt, MPI_INT
             MPI_COMM_WORLD);
```



Deadlocks com Comunicação Coletiva

```
switch (my_rank) {
    case 0: MPI_Bcast(buf1, count, type, 0, comm);
            MPI_Bcast(buf2, count, type, 1, comm);
            break;
    case 1: MPI_Bcast(buf2, count, type, 1, comm);
            MPI_Bcast(buf1, count, type, 0, comm);
}
```

Um **deadlock** pode ocorre se não existirem *buffers* do sistema. Como a operação coletiva gera mais dados, se os *buffers* existirem mas forem pequenos, também há chance de ocorrer deadlock.

Deadlocks com Comunicação Coletiva

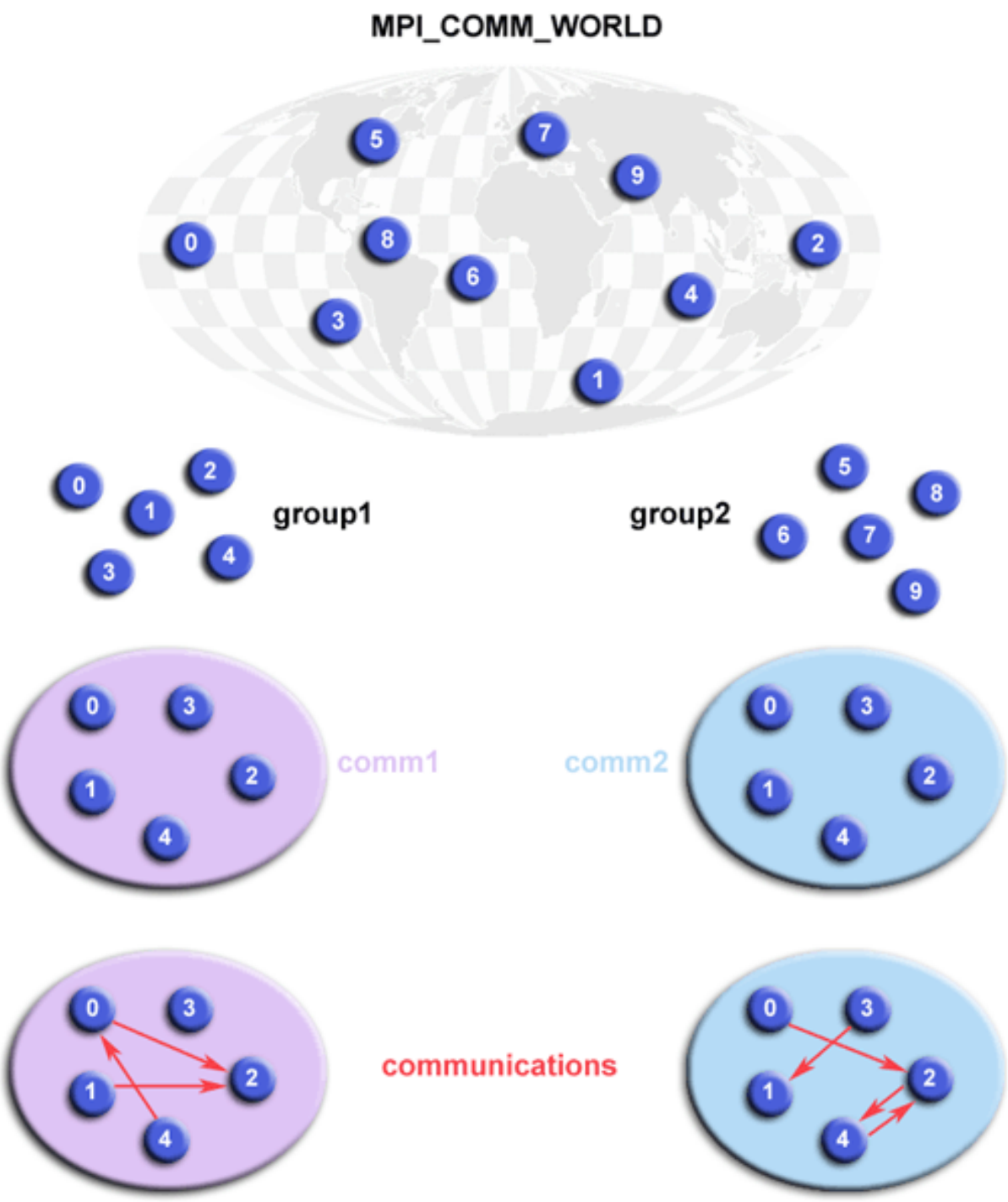
- Operações coletivas são sempre **bloqueantes**;
- portanto, na ausência de *buffers* adequados, se tornam **síncronas**;
- se a quantidade de dados for muito grande, então podemos considerar que as operações coletivas funcionam como uma barreira.

Também podemos criar uma barreira:

```
MPI_Barrier (MPI_Comm comm)
```

Grupos de Processos e Comunicadores

- Um grupo é um conjunto ordenado de processos.
  - Em um grupo, cada processo tem identificador único;
  - *ranks* variam de 0 até $N - 1$, sendo N o número de processos no grupo;
  - no MPI, o grupo está presente como uma estrutura na memória, acessível por um *handle*;
  - um grupo está sempre associado a um objeto **comunicador**.
- um comunicador engloba um grupo de processos que podem se comunicar;
- todas mensagens MPI precisam especificar um comunicador;
- do ponto de vista do programador, comunicadores e grupos são a mesma coisa;
- as rotinas de criação de grupo servem principalmente para dizer quais processos estão acessíveis através de determinado comunicador.


MPI_COMM_WORLD
group1
group2
comm1
comm2
communications

Principais Objetivos de Grupos e Comunicadores

- Permitem organizar processos, baseados em suas funcionalidades, de acordo com grupos;
- restringem comunicações coletivas apenas para determinados processos, otimizando a rede;
- são usados para criar topologias (mais adiante);
- garantem segurança na comunicação.

Considerações e Restrições

- Grupos e comunicadores são dinâmicos, criados e removidos durante a execução;
- processos podem pertencer a mais de um grupo ou comunicador, com um *rank* único em cada;
- existem mais de 40 operações para definir grupos, comunicadores e topologias;

Padrão de Utilização

1. Comece do comunicador global, *MPI_COMM_WORLD*, extraindo seu identificador de grupo usando a chamada *MPI_Comm_group*;
2. crie um novo grupo que é um subconjunto do grupo de *MPI_COMM_WORLD* usando *MPI_Group_incl*;
3. crie um novo comunicador para o grupo criado usando *MPI_Comm_create*;
4. determine os novos *ranks* usando *MPI_Comm_rank*;
5. realize trocas de mensagens;
6. quando finalizar, libere as estruturas com *MPI_Comm_free* e *MPI_Group_free*.

Recuperando o Grupo de um Comunicador

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

- *comm* é um comunicador já existente, por exemplo, MPI_COMM_WORLD;
- *group* é um ponteiro para uma variável *MPI_Group* que serve como *handle* do grupo.

Serve como ponto de partida para recuperar o grupo associado a MPI_COMM_WORLD e realizar novas operações para definir novos grupos.

União de Grupos

```
int MPI_Group_union(MPI_Group group1,
                    MPI_Group group2,
                    MPI_Group *new_group)
```

- Os *ranks* no novo grupo são configurados de forma que os processos em *group*1 mantenham seus *ranks*;
- os processos em *group*2 recebem novos *ranks* respeitando pelo menos a ordem antiga.

Intersecção de Grupos

```
int MPI_Group_intersection(MPI_Group group1,
                           MPI_Group group2,
                           MPI_Group *new_group)
```

Os processos no novo grupo recebem *ranks* sucessivos começando de 0.

Diferença de Grupos

```
int MPI_Group_difference(MPI_Group group1,
                         MPI_Group group2,
                         MPI_Group *new_group)
```

A ordem dos processos do primeiro grupo é mantida nos *ranks*.

Recuperação de Subgrupo

```
int MPI_Group_incl(MPI_Group group,
                   int p,
                   int *ranks,
                   MPI_Group *new_group)
```

- *ranks* é um vetor com p posições;
- um novo grupo é criado com p processos, com *ranks* de 0 a $p - 1$;
- é suposto que p é menor do que a quantidade de processos no grupo original.

Exclusão de Processos

```
int MPI_Group_excl(MPI_Group group,
                   int p,
                   int *ranks,
                   MPI_Group *new_group)
```

- um novo grupo sem os processos representados em *ranks*;
- novamente, só faz sentido se p for menor que a quantidade de processos anterior.

Funções Diversas para Gestão de Grupos

```
# Recuperar tamanho do grupo
int MPI_Group_size(MPI_Group group, int *size)

# Rank do processo que invoca no grupo
int MPI_Group_rank(MPI_Group, int *rank)

# Compara dois grupos
int MPI_Group_compare(MPI_Group group1,
                      MPI_Group group2,
                      int *res)

# Libera o grupo
int MPI_Group_free(MPI_Group *group)
```

Operações em Comunicadores

```
int MPI_Comm_create(MPI_Comm comm,
                    MPI_Group group,
                    MPI_Comm *new_comm)
```

- *comm* é um comunicador existente;
- *group* tem que ser um grupo subconjunto do grupo do comunicador indicado em *comm*;
- todos os processos em *comm* devem invocar para a criação correta.

Operações em Comunicadores

```
# Tamanho do comunicador
int MPI_Comm_size(MPI_Comm comm, int *size)

# Rank do Processo no Comunicador
int MPI_Comm_rank(MPI_Comm comm, int *rank)

# Compara comunicadores
int MPI_Comm_compare(MPI_Comm comm1,
                     MPI_Comm comm2,
                     int *res)

# Duplicar um comunicador
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *new_comm)

# Libera um comunicador
int MPI_Comm_free(MPI_Comm *comm)
```

Topologia de Processos

- Descrevem um mapeamento ou reordenamento de processos em uma forma geométrica virtual;
- as duas principais topologias suportadas são: cartesiano (*mesh*) e grafo;
- elas são virtuais, não precisam corresponder a topologia de rede de interconexão, mas se existir relação, pode levar a ganho de desempenho;
- são criadas usando comunicadores e grupos;
- são criadas pelo desenvolvedor.

Vantagens das Topologias

- Conveniência:
  - Alguns algoritmos casam com a representação dos processos em uma estrutura geométrica;
  - por exemplo, um *mesh* de duas dimensões pode ser adequado para um algoritmo de tratamento de imagens.
- eficiência na comunicação:
  - Algumas arquiteturas podem ter maior latência na comunicação ponto-a-ponto entre nós *distantes*;
  - o casamento das topologias virtuais e reais podem trazer ganhos expressivos de desempenho;
  - entretanto, há detalhes que dependem da implementação.

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Chamadas para Gerência de Topologias

```
int MPI_Cart_create(MPI_Comm comm,
                    int ndims, int *dims,
                    int *periods, int reorder,
                    MPI_Comm *new_comm)
```

- *ndims* é o número de dimensões;
- *dims* é um vetor que indica quantos processos existem em cada dimensão;
- *periods* indica se a dimensão tem uma ligação entre o primeiro e último processo;
- *reorder* indica se os *ranks* são preservados ou podem ser alterados.

Chamadas para Gerência de Topologias

```
int ndims = 2;
int dims[2] = {3, 4};
int periods[2] = {0, 0};
int reorder = 0;
int MPI_Cart_create(comm, ndims, dims,
                    periods, reorder, &new_comm)
```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

Chamadas para Gerências de Topologias

```
# Recuperar o rank informando as coordenadas
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

# Recuperar as coordenadas informando o rank
int MPI_Cart_coords(MPI_Comm comm, int rank,
                    int ndims, int *coords)
```

Encontrando os Vizinhos

```
int MPI_Cart_shift(MPI_Comm comm,
                   int dir, int displ,
                   int *rank_source,
                   int *rank_dest)
```

- *dir* é a dimensão a ser investigada;
- *displ* é a distância até o vizinho;
- *rank_dest* é o *rank* do vizinho;
- *rank_source* é um *rank* do outro vizinho, na direção contrária.

Encontrando os Vizinhos

```
int coords[2], dims[2], periods[2], source, dest,
    my_rank, reorder;
int dims[2] = {3, 4};
periods[0] = periods[1] = 1;
reorder = 0;
float a, b;

MPI_Comm comm_2d;
MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
                reorder, &comm_2d);
MPI_Cart_coords(comm_2d, my_rank, 2, coords);
MPI_Cart_shift(comm_2d, 0, coords[1], &source, &dest);
a = my_rank;
MPI_Sendrecv(&a, 1, MPI_FLOAT, dest, 0, &b, 1, MPI_FLOAT,
             source, 0, comm_2d, &status);
```

Encontrando os Vizinhos

0 (0,0) 0|0	1 (0,1) 9|5	2 (0,2) 6|10	3 (0,3) 3|3
4 (1,0) 4|4	5 (1,1) 1|9	6 (1,2) 10|2	7 (1,3) 7|7
8 (2,0) 8|8	9 (2,1) 5|1	10 (2,2) 2|6	11 (2,3) 11|11

Medindo Tempo de Execução

```
double start = MPI_Wtime();
trabalho_a_ser_medido();
double end = MPI_Wtime();
printf("%.2f\n", end - start);
```

Funcionalidades Extras do MPI-2

- **Processos Dinâmicos**: rotinas para criar novos processos após o início do programa;
- **comunicação unilateral**: rotinas para memória compartilhada e acumulações;
- **comunicações coletivas estendidas**: suporte para comunicações coletivas entre comunicadores;
- **interfaces externas**: instrumentos para auxiliar a criação de depuradores e analisadores de desempenho;
- **suporte a novas linguagens**: C++ e Fortran90.
- **E/S**: entrada e saída paralela.

Funcionalidades Extras do MPI-3

- **Comunicação Coletiva não Bloqueante**: aprimora desempenho da comunicação;
- **melhorias na comunicação unilateral**: suporte a novos modelos de memória;
- **comunicações coletivas em topologias**: suporte ara comunicações coletivas entre vizinhos;
- **suporte a Fortran2008**: expansão a partir do Fortran90;
- **ferramenta MPIT**: melhoria no suporte a depuração.
- **melhoria no suporte a threads**: resolve problemas de concorrência do MPI-2.

Fim

FIM.