PARALELISMO POR TROCA DE MENSAGENS

Message Passing Interface

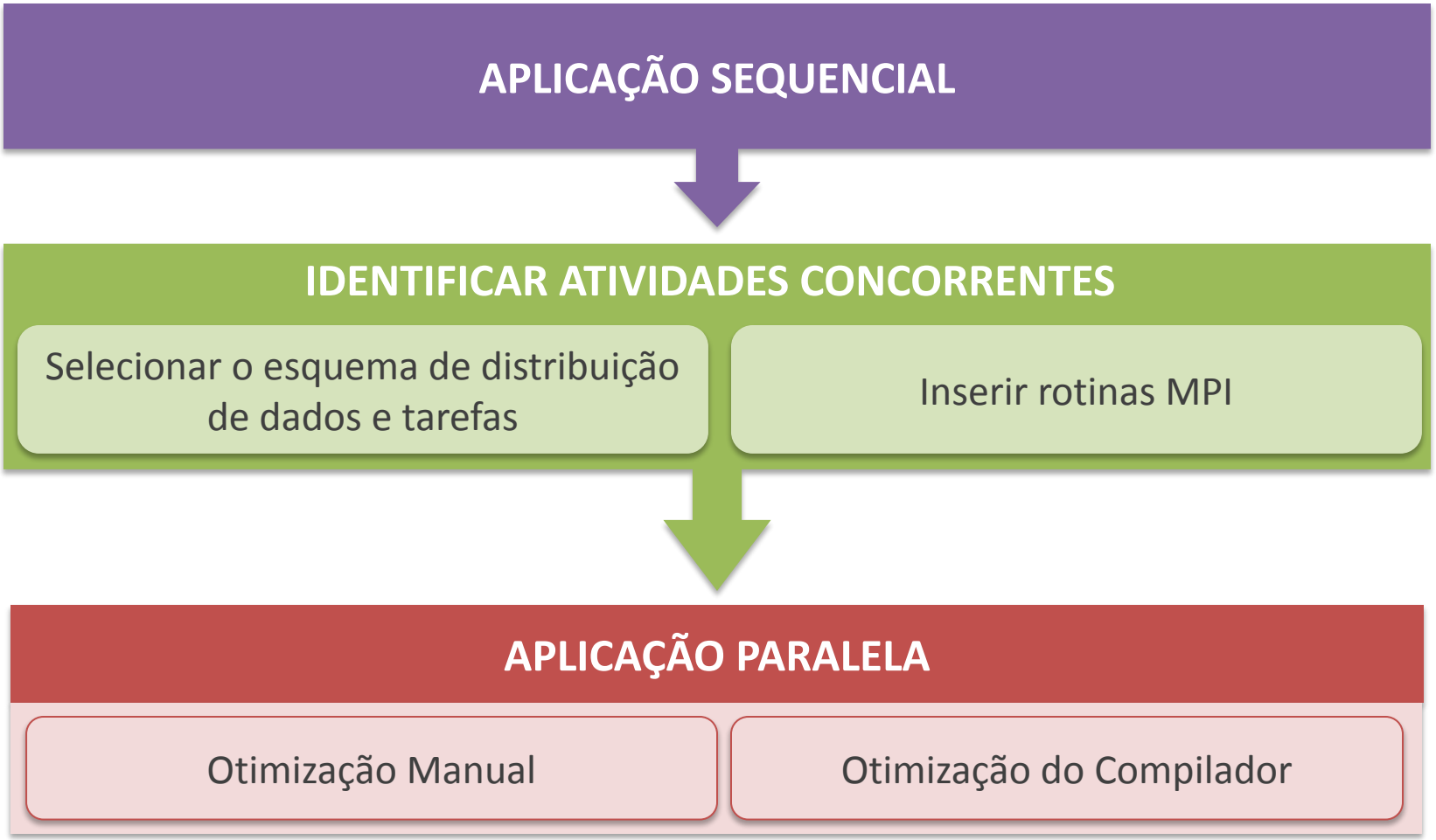
JOSE J. CAMATA

Departamento de Ciência da Computação
Universidade Federal de Juiz de Fora

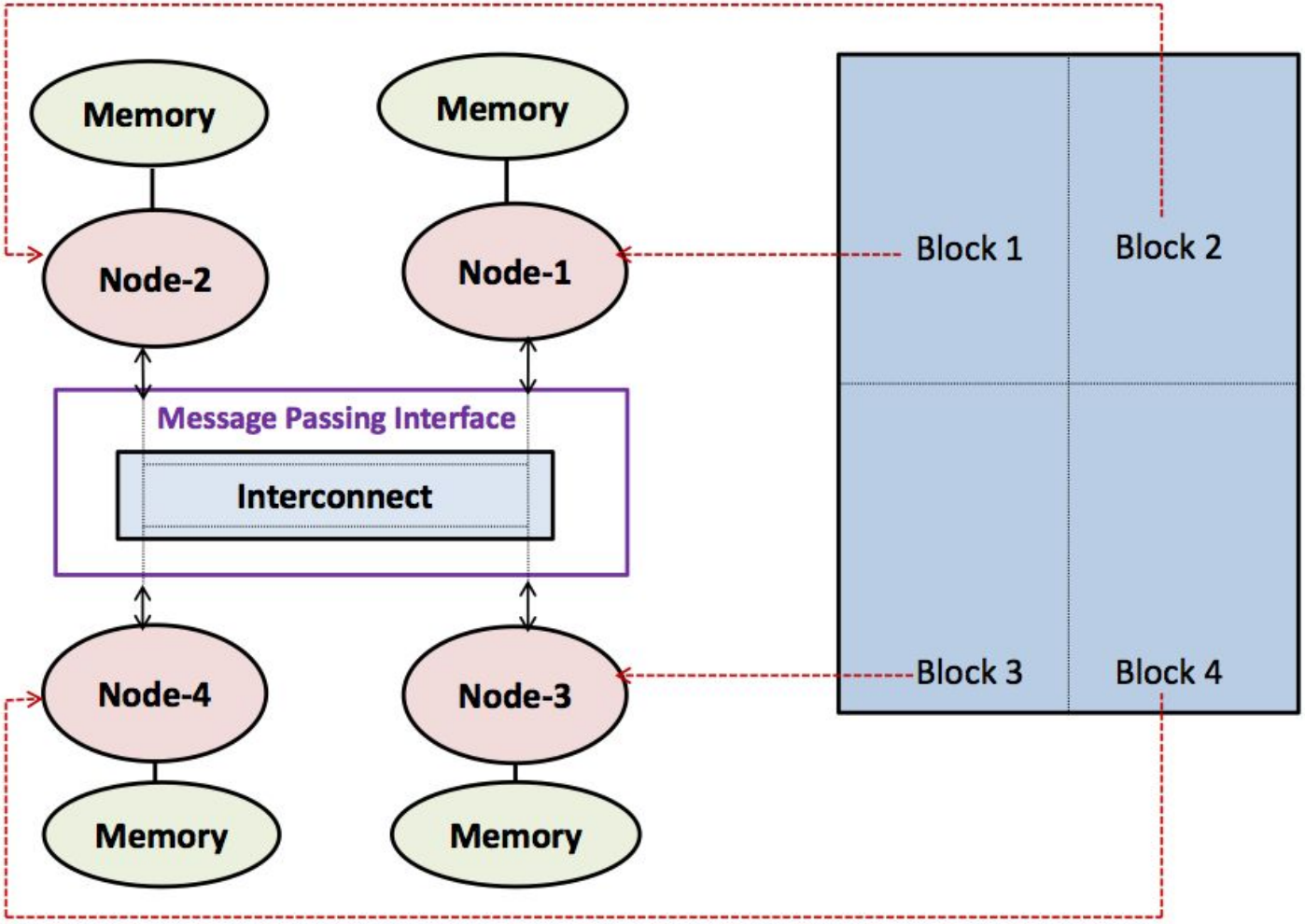
Aspectos Gerais

- Fácil de entender → difícil de implementar:
- Bastante versátil/portátil pois funciona em sistemas de memória distribuída ou compartilhada;
- Basicamente convertemos um problema grande em vários problemas menores e atribuímos cada sub-problema a um processador (ou processo);
- Várias cópias idênticas do programa são executadas simultaneamente;
- Cada cópia atua em sua porção do problema independentemente;
- Partes em comum são “sincronizadas” através de operações de trocas de mensagens utilizando as rotinas do MPI;
- Paralelismo baseado na chamada de rotinas da biblioteca MPI para a troca de informações entre os processos (MPI_BROADCAST, MPI_SEND, MPI_ALLREDUCE, ...)

Paralelização com MPI

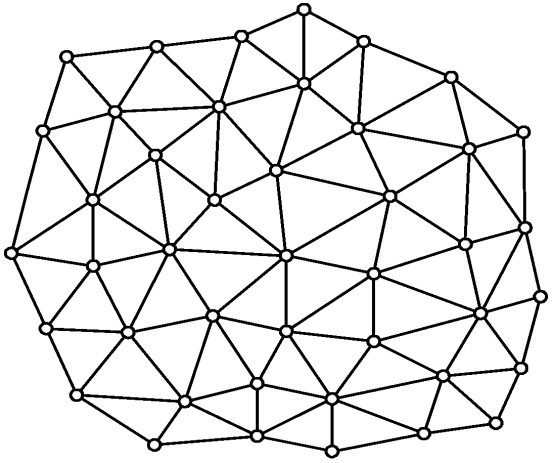


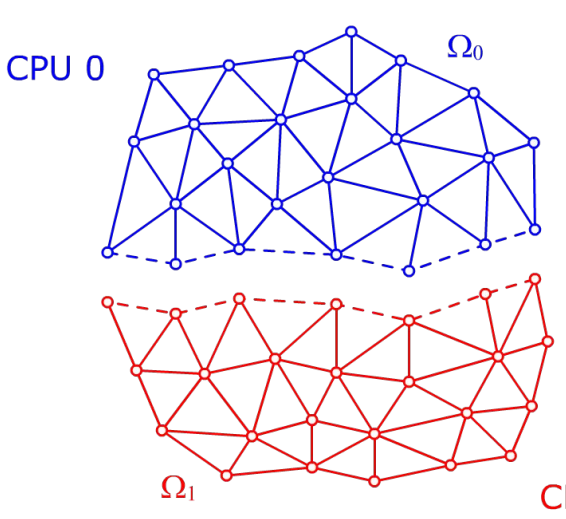
Dividir & Conquistar



Dividir & Conquistar

- A forma mais intuitiva e amplamente difundida de se aplicar o MPI à problemas de mecânica computacional é através da divisão do domínio em diversos subdomínios menores e mais fáceis de serem calculados. Desta forma, cada processador pode atuar de forma colaborativa na solução mais rápida do problema global.





=> Técnicas de particionamento dos dados

MPI: O que é?

- MPI é um padrão/especificação para uma biblioteca de trocas de mensagens
  - Múltiplas implementações: OpenMPI, MPICH, MPT, etc
- Principalmente usado para **sistemas de memória distribuída**
  - Cada processo tem um espaço de endereçamento distinto
  - Processo precisa comunicar com os demais processos
    - Sincronização
    - Trocas de dados
  - Pode também ser usado em sistemas de memória compartilhada e híbridos
- As especificações MPI foram definidas para C, C++ e Fortran

Trocas de Mensagens

- O que é troca de mensagens?
  - Resposta simples: O envio e recebimento de mensagens entre diversos recursos computacionais

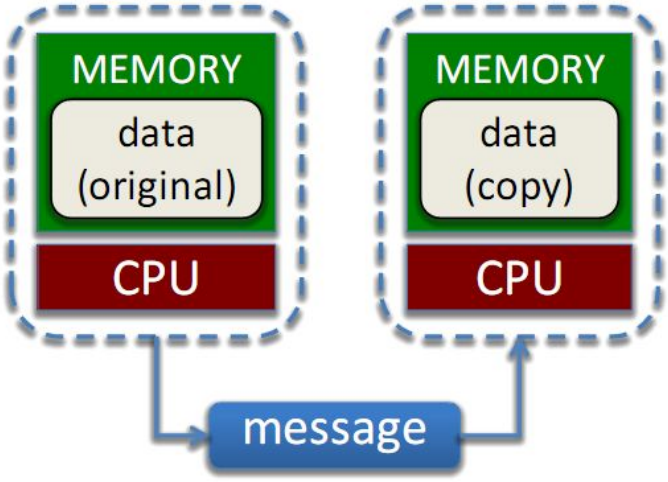
Trocas de Mensagens

- Mensagens podem ser usadas para
  - Enviar dados
  - Executar operações nos dados
  - Sincronização entre tarefas

- Por que necessitamos enviar/receber mensagens?
  - Em clusters, cada nó tem seu próprio espaço de endereço, e não há nenhum outro modo de acessar dados do outro nó, exceto pela rede.

Modelo de Troca de Mensagens

- Tarefas enviam e recebem mensagens
- Transferência de dados requer uma operação cooperativa para ser executada por cada processo
- O programador é responsável por determinar todo o paralelismo
- Message Passing Interface (MPI) foi lançado em 1994 (MPI 2 em 1996)
- MPI tornou-se padrão para troca de mensagens
  - http://www-unix.mcs.anl.gov/mpi/



MPI: Características Básicas

- Gerais
  - Comunicadores combinam um grupo de processos em um mesmo contexto
- Comunicação Ponto-a-Ponto
  - Buffers estruturados e tipos de dados derivados, heterogeneidade.
  - Modos: normal (bloqueante e não bloqueante), síncrona e buferizada.
- Comunicação Coletiva
  - Operações predefinidas ou definidas pelo usuário
  - Grande número de rotinas de transferência de dados
  - Subgrupos definidos diretamente ou por topologia
- Aplicação orientada a topologia
  - Suporte a grids e grafos

MPI: Características Básicas

- Subconjuntos de funcionalidades
  - básico (6 funções)
  - Intermediário
  - Avançado (até 125 funções)

- Objetivo principal do MPI é permitir o desenvolvimento de aplicações científicas voltadas para sistemas paralelos complexos
  - Não apenas para programadores e cientistas da computação

- Algumas bibliotecas desenvolvidas com o MPI
  - PETsc
  - SAMRAI
  - CACTUS
  - FFTW
  - PLAPACK

Construindo programas em MPI

- Quem faz o que?
- Quem lê os dados?
- Cada processo lê o seu dado ou o apenas um lê e distribui?
- O que deve ser comunicado/sincronizado?
- Quando eu devo comunicar/sincronizar os processos?
- Quem imprimirá mensagens na tela?
- Quem gravará os arquivos de saída?
- Os arquivos de saída devem ser unificados?
- *"...Depurar em MPI é a arte de saber escrever mensagens na tela em trechos estratégicos do programa..."*

Por que aprender MPI?

- MPI é um padrão
  - Domínio público
  - Fácil instalação
  - Versões otimizadas disponíveis para a maioria das topologias de comunicação
- Aplicações MPI são portáveis
- MPI é um bom modo de aprender a teoria de computação paralela

Implementações MPI

- Diversas instituições/companhias implementam as especificações MPI
  - MPICH2 (http://www.mcs.anl.gov/research/projects/mpich2/)
  - OpenMPI (http://www.open-mpi.org/)
  - MVAPICH (http://mvapich.cse.ohio-state.edu/)

Estrutura de um programa MPI

```
#include <mpi.h>

Declarações, protótipos, etc

Programa Inicia
...
```

Serial

```
Inicialização do ambiente MPI

Computação e chamadas de rotinas para trocas de mensagens

Finalização do ambiente MPI
```

Paralelo

```
...
Programa Finaliza
```

Serial

Um simples programa MPI

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
     int rank, size;
     // Inicializa MPI
     MPI_Init(&argc, &argv);

     printf("Hello world\n");

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

Exemplo (1)

- Programa serial

```
#include <stdio.h>

int main(int argc, char* argv) {

   printf("Alô mundo!!\n");
   return 0;

}
```

```
Compilando:
$gcc -o exemplo1 exemplo1.c

Executando:
$./exemplo1
Alô mundo!!
```

Exemplo (2)

- Seqüencial para paralelo

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

int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);

   printf("Alô mundo!!\n");

   MPI_Finalize();
   return 0;

}
```

Compilando:

```
$mpicc -o exemplo1 exemplo1.c
```

Executando:

```
$mpirun -np <#PROCS> ./exemplo1
```

Conceito de Comunicadores

- Comunicadores são objetos que são usados para definir uma coleção de processos que podem se comunicar entre eles
- Maioria das rotinas MPI requer um comunicador como argumento
- MPI_COMM_WORLD é o comunicador que inclui todos os processos MPI
- Múltiplos comunicadores podem ser definidos

Execução MPI

- Cada processo roda uma cópia do executável
- Cada processo pega a porção do trabalho de acordo com seu rank
  - *Rank:* identificador do processo em um determinado **comunicador**
- Cada processo ***trabalha independentemente*** dos outros processos, exceto quando há comunicação

Raciocinando em Paralelo

- Programas em MPI normalmente possuem um “mestre”: O rank 0
  - O rank 0 normalmente controla o fluxo principal do programa
  - O rank 0 também deve ser utilizado em processamento – ***“não desperdice nenhum processador!!!”***
- Programas em MPI devem funcionar também em modo serial (somente 1 processador)
- O resultado de uma rodada paralela deve ser igual ao de uma rodada serial (mas nem sempre é idêntico!)

Todo programa MPI...

- Deve incluir o arquivo de cabeçalho mpi.h
- Deve ter uma rotina de inicialização do ambiente(MPI_Init)
- Deve ter uma rotina que finaliza o ambiente MPI (MPI_Finalize)

Rotinas de Gerenciamento de Ambiente

- **`MPI_Init`**: inicializa a ambiente de execução MPI, deve ser chamada antes de qualquer outra rotina MPI e é invocada uma única vez em um programa MPI.
  - `MPI_init(int * argc, char **argv[])`

- **`MPI_Finalize`**: termina a execução do ambiente MPI e deve ser chamada por último.
  - `MPI_Finalize()`

Rotinas de Gerenciamento de Ambiente

- **`MPI_Comm_size`** determina o número de processos que estão associados com um comunicador:
  - `MPI_Comm_size(MPI_Comm comm, int* size)`

- **`MPI_Comm_rank`** determina o número do processo dentro de um comunicador
  - `MPI_Comm_rank(MPI_Comm comm, int* rank)`

Exemplo (3)

- Usando um comunicador e rotinas de gerenciamento de ambiente

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

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

   printf("Alô mundo!! Eu sou o processo %d de %d \n", rank, size);

   MPI_Finalize();
   return 0;

}
```

Trocas de mensagens

- Rotinas podem ser:
  - Sincronização;
  - Comunicação ponto-a-ponto:
    - Bloqueante,
    - Não-Bloqueante,
    - Síncrona,
    - Buferizada,
    - Combinada;
  - Comunicação coletiva.

COMUNICAÇÃO PONTO-A-PONTO

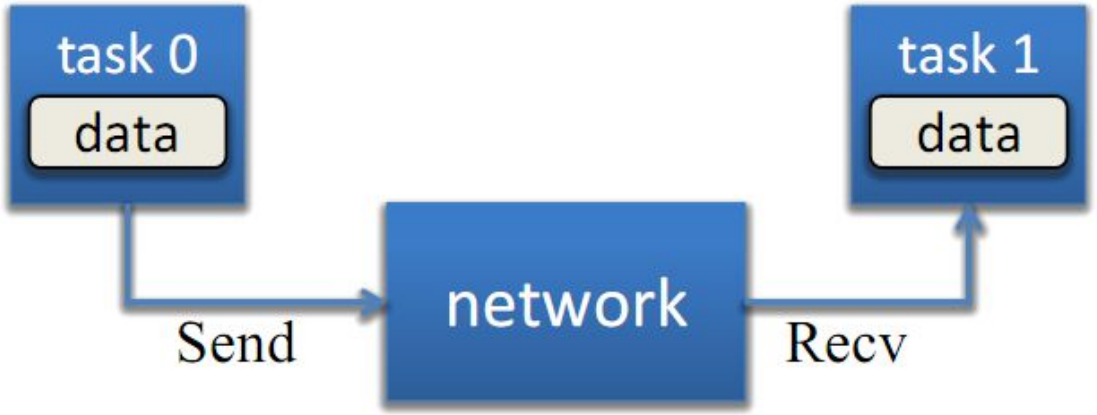
Comunicação Ponto-a-Ponto

- Envolve a troca de mensagens entre dois processos MPI distintos.
- Um processo executa uma operação de envio enquanto o outro processo uma operação casada de recebimento.
- **Importante:** Sempre deve ter um pareamento entre as rotinas de envio e recebimento.
  - Se isso não acontecer, o programa entrará em deadlock.

Comunicação Ponto-a-Ponto

- Princípio: Envio de dados de um ponto (processo) para outro ponto (processo).

- Um processo envia enquanto outro recebe



Tipos de operações de Ponto-a-Ponto

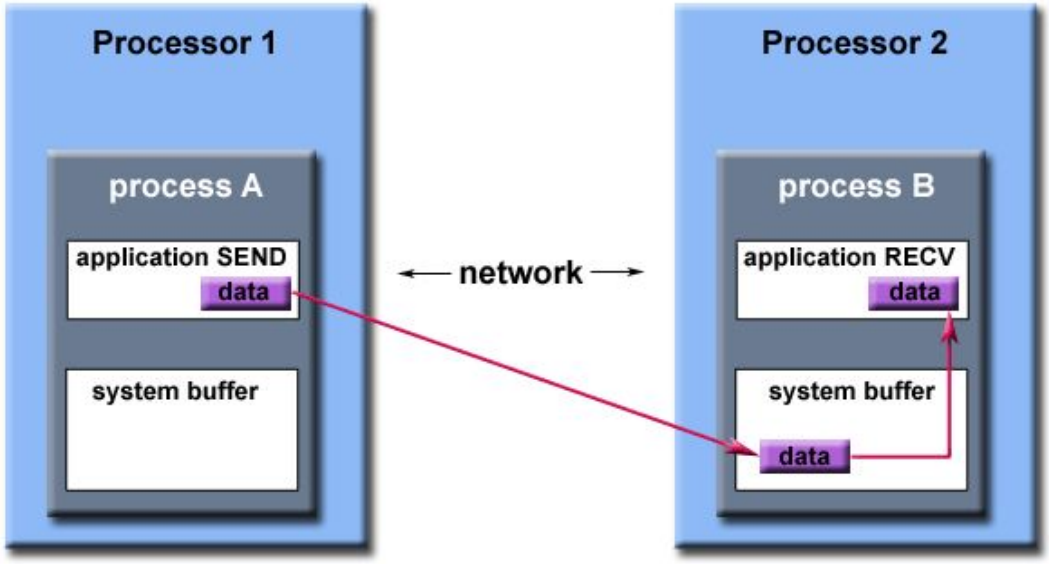
- Operações de ponto-a-ponto normalmente envolvem a passagem de mensagens entre duas, e apenas duas diferentes tarefas MPI.
- Existem diferentes tipos de rotinas de envio e recebimento que são usadas para finalidades diferentes. Por exemplo:
  - Envio síncrono
  - Envio/recebimento bloqueante
  - Envio/recebimento não bloqueante
- Qualquer tipo de rotina de envio pode ser combinado com qualquer tipo de rotina recebimento.
- MPI também fornece várias rotinas associadas às operações de envio/recebimento, tais como aqueles usados para esperar o recebimento de uma mensagem (wait) ou sondagem para saber se a mensagem chegou (probe)

Buffering

- Em um mundo perfeito, cada operação de envio seria perfeitamente sincronizado com a sua operação correspondente de recebimento. Isso raramente é o caso. De alguma forma ou de outra, a implementação MPI deve ser capaz de lidar com o armazenamento de dados quando as duas tarefas estão fora de sincronia.
- Considere os dois casos seguintes:
  - A operação de envio ocorre 5 segundos antes do recebimento está pronto - onde está a mensagem, enquanto a recepção está pendente?
  - Múltiplos envios chegam à mesma tarefa de recebimento, que só pode aceitar um envio de uma vez - o que acontece com as mensagens que estão chegando?

Buffering (cont)

- A implementação MPI (não o padrão MPI) decide o que acontece com os dados nestes tipos de casos. Normalmente, uma área de buffer do sistema são reservados para armazenar dados em trânsito. Por exemplo:



Path of a message buffered at the receiving process

Figura obtida em https://computing.llnl.gov/tutorials/mpi/

Buffering (cont)

- Espaço de buffer do sistema é:
  - Invisível para o programador e gerida exclusivamente pela biblioteca MPI
  - Um recurso finito que pode facilmente esgotar
  - Não muito bem documentado
  - Capaz de existir no lado de envio, no lado receptor, ou em ambos
  - Algo que pode melhorar o desempenho do programa, pois permite operações de envio e recebimento assíncronas.

P2P: Bloqueante vs. Não Bloqueante

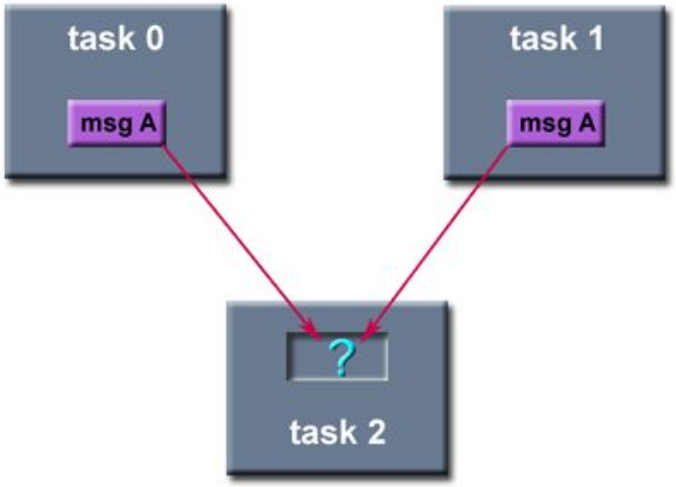
- **Bloqueante**:
  - Um envio bloqueante somente retornará se for seguro modificar o buffer de envio;
    - *Seguro aqui significa que modificações no buffer não afetarão os dados que estão sendo enviados*
  - Seguro não implica que os dados foram realmente recebidos
  - Envio bloqueante pode ser síncrono, ou seja, existe um acordo (handshaking) com a receptor para garantir o envio seguro.
  - Um envio bloqueante pode ser assíncrona se um sistema de buffer é utilizado para armazenar os dados para uma entrega posterior ao receptor.
  - Recebimento bloqueante apenas libera o fluxo de execução após os dados chegarem e estarem prontos para uso pelo programa.

P2P: Bloqueante vs. Não Bloqueante

- **Não Bloqueante**:
  - As rotinas de envio/recebimento retornam imediatamente.
  - Operações não-bloqueantes requisitam que a operações sejam executadas quando possível.
  - Não é seguro modificar o buffer até que a operações requisitadas tenham sido executadas. Existem rotinas para verificar isto.
  - Comunicação não-bloqueante é utilizada principalmente para sobrepor computação com comunicação e explorar possíveis ganhos de desempenho.

Ordem e Equidade

- **Ordem**: MPI garante que uma mensagem não irá ultrapassar outras.
  - Se um remetente envia duas mensagens (Mensagem 1 e Mensagem 2), em sucessão para o mesmo destino, e ambos se encaixam na mesma operação de recebimento, a operação ocorrerá primeiro para Mensagem 1 e então para Mensagem 2.
  - Se o receptor postou dois recebimentos em sucessão e ambos estão esperando a mesma mensagem, a primeira postagem de recebimento receberá a mensagem primeiro.
- **Equidade** : Não garantido pelo MPI. O programador deve evitar deadlocks.
  - Exemplo: tarefa 0 envia uma mensagem para a tarefa 2. No entanto, tarefa 1 envia uma mensagem concorrente a tarefa 2. Apenas um dos envios vai completar.



Rotinas de comunicação ponto a ponto bloqueantes

Envio Bloqueante: MPI_Send

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

Operação de envio básico com bloqueio. A rotina retorna apenas quando o buffer de aplicação na tarefa de envio estiver livre para reutilização.

Argumento	Descrição
buf	Endereço inicial do buffer de envio
count	Número de itens para enviar
Datatype	Tipo de dados MPI
dest	Rank do processo que receberá a mensagem
tag	Identificador da mensagem
Comm	Comunicador MPI onde o envio ocorrerá

Recebimento bloqueante: MPI_Recv

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

Receba uma mensagem e bloqueie até que os dados solicitados estejam disponíveis no buffer de aplicação na tarefa de recebimento.

Argumento	Descrição
buf	Endereço inicial do buffer de recebimento
count	Número de itens para receber
Datatype	Tipo de dados MPI
dest	Rank do processo que enviou a mensagem
tag	Identificador da mensagem
Comm	Comunicador MPI onde o recebimento ocorrerá
Status	Retorna informação da mensagem recebida

Resumo: MPI_Send & MPI_Recv

```
MPI_Send(buf, count, datatype, dest,   tag, comm);
MPI_Recv(buf, count, datatype, source, tag, comm, status);
```

- **Tag** é um inteiro arbitrário não negativo atribuído pelo programador para identificar uma mensagem.
  - Operações de envio e recebimento devem ter tags correspondentes.
  - Para uma operação de recebimento, a tag MPI_ANY_TAG pode ser usado para receber uma mensagem, independentemente da sua etiqueta.
  - O Padrão MPI permite que inteiros entre 0-32767 podem ser usados como marcadores, mas a maioria das implementações permitem uma gama muito maior.
- No objeto **status**, o sistema pode retornar detalhes da mensagem recebida:
  - É possível obter a tag, o processo que enviou, o tamanho de mensagem, etc.
  - Pode-se passar o objeto default MPI_STATUS_IGNORE

Tipos de Dados MPI

Por questão de portabilidade, MPI pré-define seus tipos de dados elementares

Tipos de Dados em C		Tipos de dados Fortran	
MPI_CHAR	signed char	**MPI_CHARACTER**	character(1)
MPI_SHORT	signed short int		
MPI_INT	signed int	**MPI_INTEGER**	integer
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_REAL**	real
MPI_DOUBLE	double	**MPI_DOUBLE_PRECISION**	double precision
MPI_LONG_DOUBLE	long double		
		MPI_COMPLEX	complex
		MPI_DOUBLE_COMPLEX	double complex
		MPI_LOGICAL	logical
MPI_BYTE	8 binary digits	**MPI_BYTE**	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack	**MPI_PACKED**	data packed or unpacked with MPI_Pack()/ MPI_Unpack

Exemplo #1

Enviando um vetor do processo 0 para processo 1

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
   int rank, nprocs;
   MPI_Status status;
   double vector[10];

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

   if(nprocs != 2 )
   {
       printf("Este programa necessita de 2 processos\n");
       MPI_Finalize();
       return 0;
    }
    if(rank == 0)
    {
        for(int i=0; i < 10; i++)
         vector[i] = 0.1*i;
        MPI_Send(vector,10,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
    else if (rank == 1)
    {
        MPI_Recv(vector,10,MPI_DOUBLE,0,1,MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
    return 0;
}
```

código

Outras rotinas

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

Envio síncrono com bloqueio: Envia uma mensagem e bloqueia até que o buffer da aplicação na tarefa de envio esteja livre para reutilização e o processo de destino tenha começado a receber a mensagem.

```
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 )
```

Envia uma mensagem e posta uma operação de recebimento antes de bloquear. Vai bloquear até que o buffer da aplicação de envio esteja livre para reutilização e até que o buffer da aplicação de recebimento contenha a mensagem recebida.

Outras rotinas

```
int MPI_Wait (MPI_Request  *request, MPI_Status   *status)
int MPI_Waitall( int count, MPI_Request array_of_requests[],
                            MPI_Status array_of_statuses[] )
```

MPI_Wait bloqueia até que uma operação de envio ou recebimento não bloqueante especificada tenha sido concluída. Para múltiplas operações não bloqueantes, o programador pode especificar qualquer, todas ou algumas conclusões.

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

Realiza um teste de bloqueio para uma mensagem. Os "wildcards" MPI_ANY_SOURCE e MPI_ANY_TAG podem ser usados para testar uma mensagem de qualquer origem ou com qualquer tag. Em C, a origem real e a marcação serão retornadas na estrutura status como status.MPI_SOURCE e status.MPI_TAG.

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

Dados um status de uma rotina de recebimento, retorna a quantidade de elementos do tipo datatype recebidos.

```
int main(int argc, char *argv[]) {
      int numtasks, rank, dest, source, rc, count, tag=1;
      char inmsg, outmsg='x';
      MPI_Status Stat;
      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      if (rank == 0) {
            dest = 1; // Tarefa zero 0 envia para tarefa 1
            source = 1;
            MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
            MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
      }
      else if (rank == 1) {
            dest = 0;
            source = 0; // Tarefa 1 recebe da tarefa 0
            MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
            MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
      }
      // Testa a variável Stat e imprime mensagem.
      MPI_Get_count(&Stat, MPI_CHAR, &count);
      printf("Task %d: Received %d char(s) from task %d with tag %d \n",
      rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
      MPI_Finalize();
}
```

Comunicação Ponto a Ponto: Rotinas Não-bloqueantes

Envio não-bloqueante

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

Envio não bloqueante: Identifica uma área na memória para servir como buffer de envio. O processamento continua imediatamente sem esperar que a mensagem seja copiada do buffer de aplicação. Um identificador de solicitação (MPI_request) da comunicação é retornado para lidar com o *status* da mensagem pendente. O programa não deve modificar o buffer de aplicação até que chamadas subsequentes para MPI_Wait ou MPI_Test indiquem que o envio não-bloqueante tenha sido concluído.

Recebimento não-bloqueante

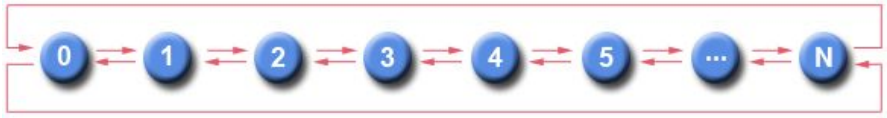
```
int MPI_Irecv( void *buf, int count, MPI_Datatype datatype,
               int source, int tag, MPI_Comm comm,
               MPI_Request *request )
```

Recebimento não bloqueante: Identifica uma área na memória para servir como buffer de recebimento. O processamento continua imediatamente sem esperar efetivamente que a mensagem seja recebida e copiada para o buffer de aplicação. Um identificador da solicitação de comunicação (MPI_Request) é retornado para lidar com o status da mensagem pendente. O programa deve usar chamadas para MPI_Wait ou MPI_Test para determinar quando a operação de recebimento não-bloqueante é concluída e a mensagem solicitada está disponível no buffer de aplicação.

```
int main(int argc, char *argv[]) {
     int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
     MPI_Request reqs[4]; // required variable for non-blocking calls
     MPI_Status stats[4]; // required variable for Waitall routine
     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     // determine left and right neighbors
     prev = rank-1;
     next = rank+1;
     if (rank == 0) prev = numtasks - 1;
     if (rank == (numtasks - 1)) next = 0;
     // post non-blocking receives and sends for neighbors
     MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
     MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

     MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
     MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);
     // do some work while sends/receives progress in background
     // wait for all non-blocking operations to complete
     MPI_Waitall(4, reqs, stats);
     // continue - do more work
     MPI_Finalize();
}
```

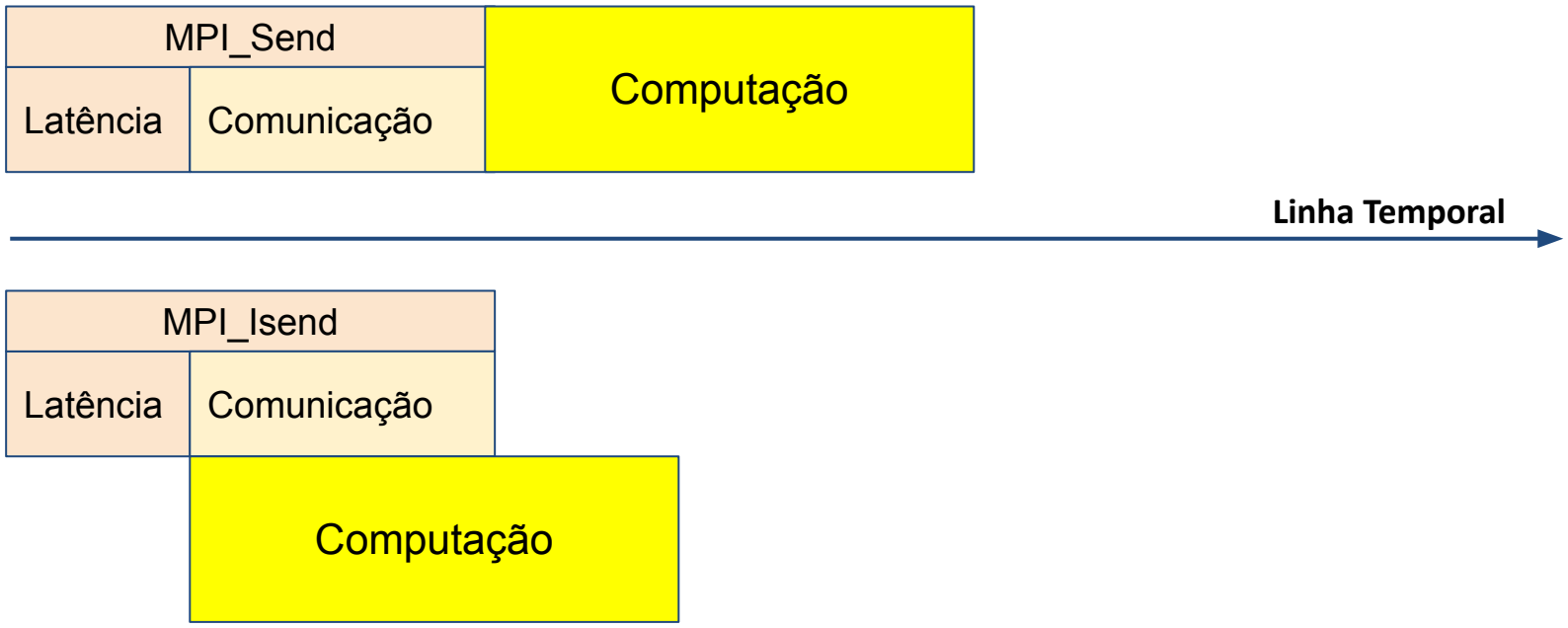
Exemplo:

Comunicação em anel entre vizinhos próximos



Implementação Eficiente

- Uso de comunicação não bloqueante permite sobrepor comunicação com computação



Exercício

1. Considere o código que calcula a integral da função cos(x) em uma intervalo [a,b] usando a regra de integração do ponto médio disponível neste link.
   Faça:
   a. Compile o código serial e execute
   b. Parallelize usando apenas send/recv bloqueantes
   c. Adapte para as rotinas não-bloqueantes.

COMUNICAÇÃO COLETIVA

Comunicação Coletiva

- Definido como uma comunicação entre diversos processos (> 2)
  - Tipos
    - um-para-muitos
    - muitos-para-um
    - muitos-para-muitos
- Uma operação coletiva requer que todos os processos dentro de um mesmo comunicador chame a mesma rotina de comunicação coletiva com argumentos casados.

Básico de Comunicação Coletiva

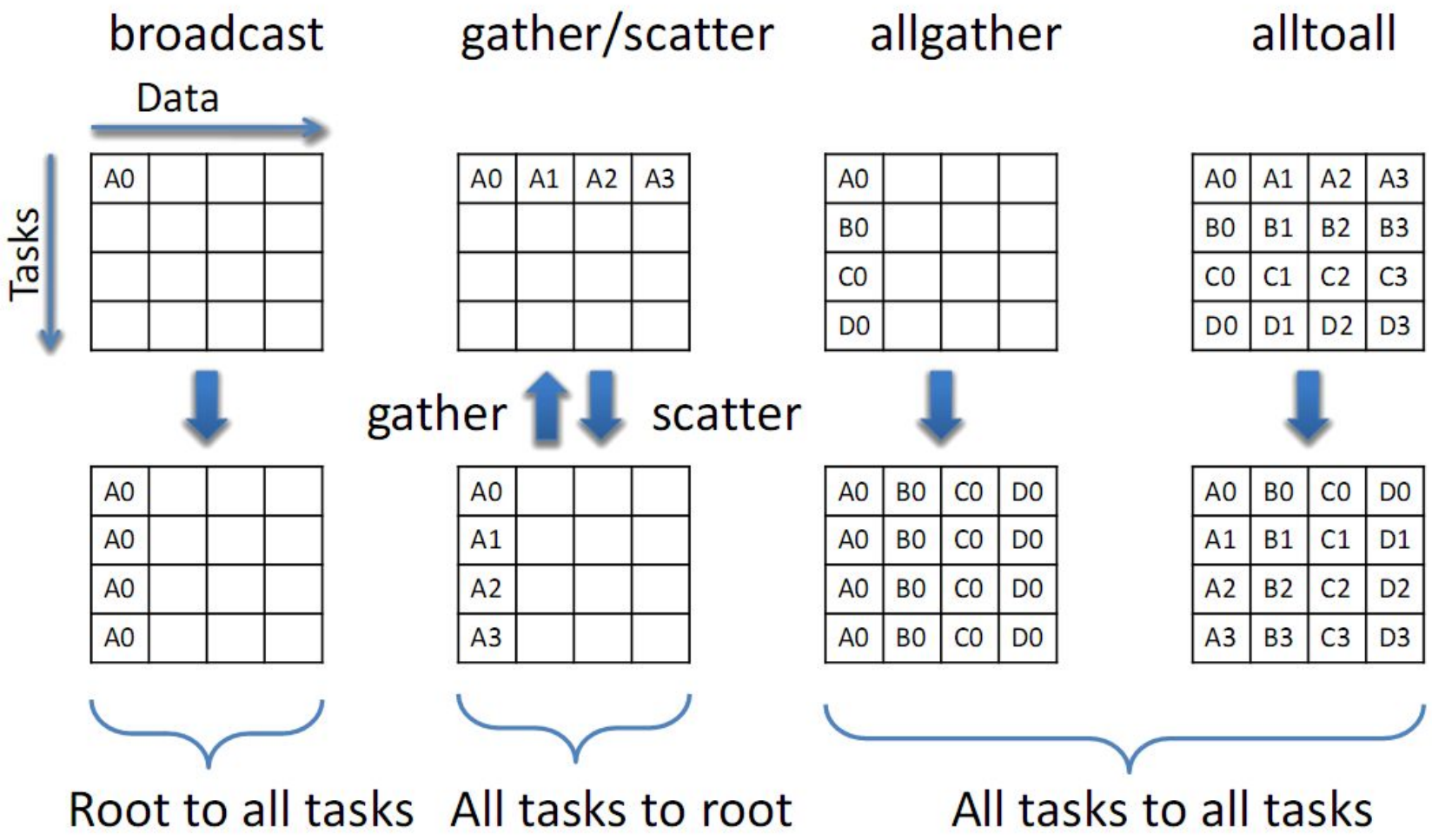
- Envolve todos os processos dentro de um comunicador
- É de responsabilidade do programador assegurar que todos os processos chamarão a mesma comunicação coletiva ao mesmo tempo
- Tipo de operações coletivas
  - Sincronização
  - Movimento de Dados
  - Redução
- Considerações de programação e restrições
  - Comunicação coletiva são operações bloqueantes
  - Operações coletivas em um subconjunto de processos requer um novo comunicador
  - O tamanho da mensagem enviada deve casar exatamente com o tamanho da mensagem recebida

Sincronização de Processos

MPI_Barrier(MPI_comm comm)

- Bloqueia o fluxo de execução até que todos os processos do comunicador comm alcancem esta rotina
- Pode ser usado quando deseja medir o custo de comunicação/computação e depuração.
- Deve-se preocupar com o excesso de sincronização: isto pode reduzir o desempenho de sua aplicação

Movimentação de Dados



Broadcast

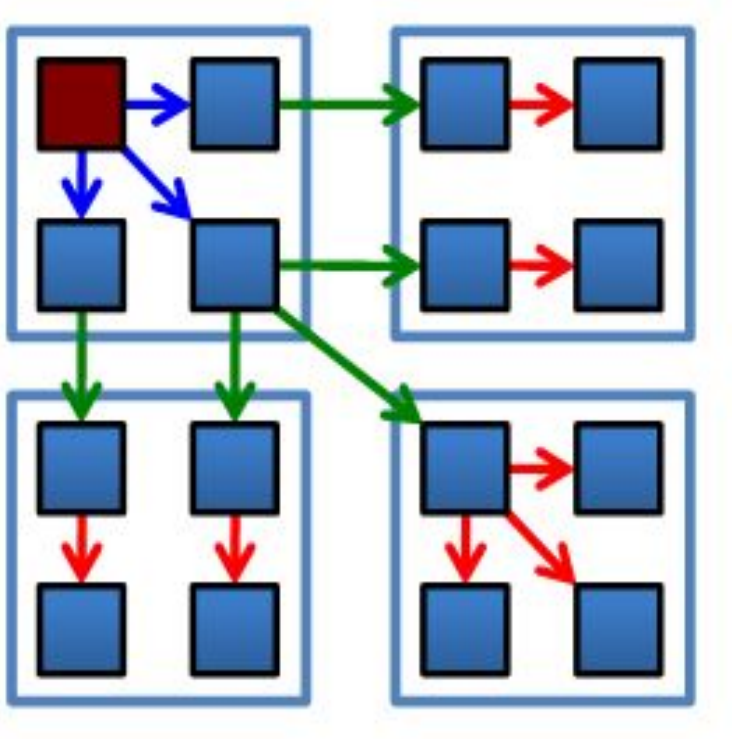
- **Um processo deseja enviar uma mensagem para todos os outros processos**
  - Implementação ingênua:

```
if (rank == 0 ) {
  for (int id=1; id<np; id++) {
    MPI_Send( ..., /* dest= */ id, ... );
  }
} else {
  MPI_Recv( ..., /* source= */ 0, ... );
}
```

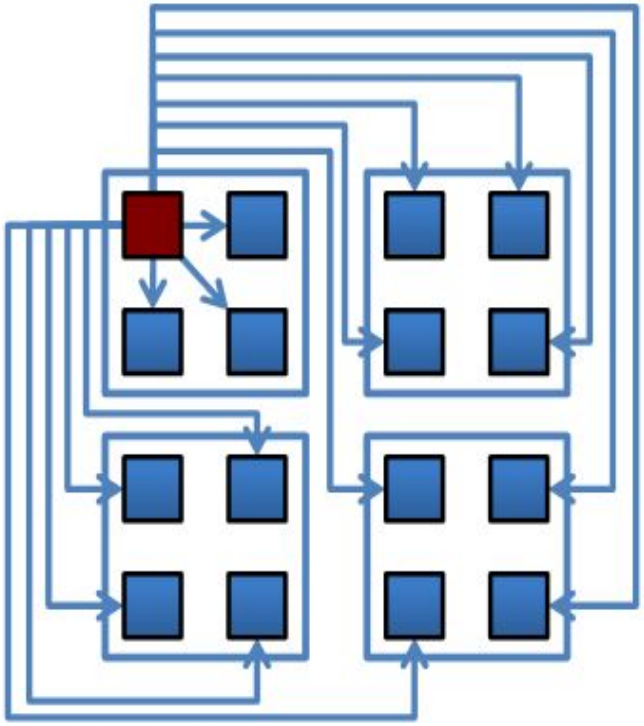
  - Este código envia informação do processo 0 para todos os outros processos, mas
    - Muito primitivo – nenhuma otimização para o hardware
    - Usa comunicação bloqueante

Implementação do Broadcast

Implementação MPI



Implementação ingênua



MPI Broadcast

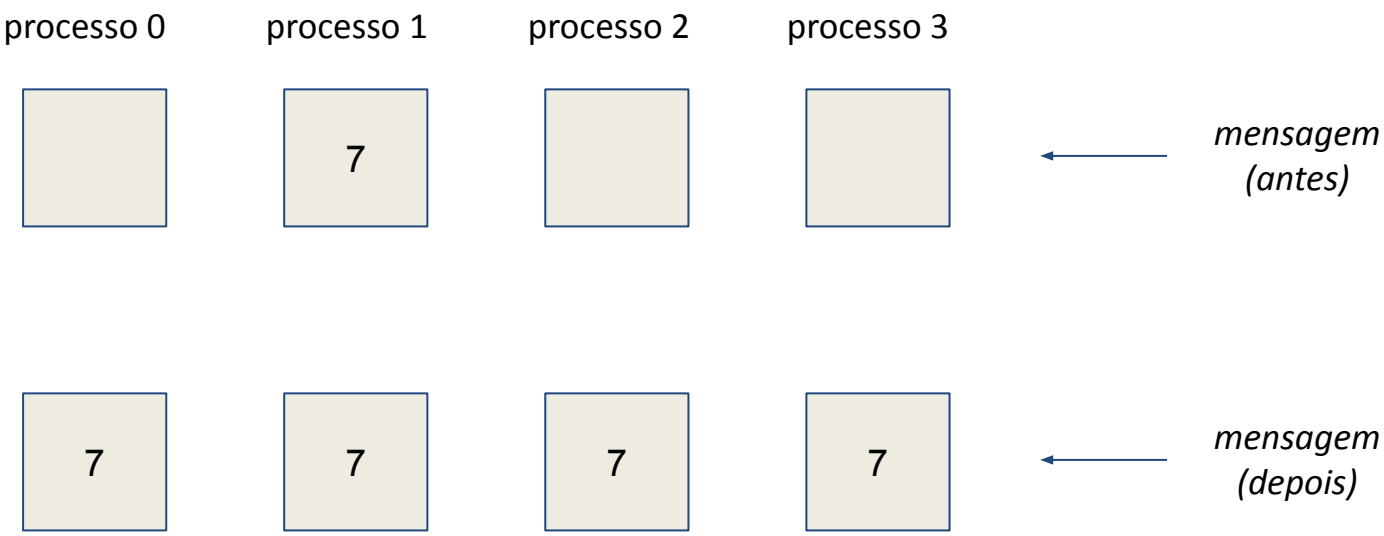
- Envia uma mensagem de um processo com identificador “root” para todos os outros processos em um grupo (no comunicador comm)

MPI_BCAST(buf,count,datatype,root,comm)		
IN/OUT	buf	Início do endereço de memória do buffer
IN	count	Número de elementos no buffer
IN	datatype	Tipo de dados contidos no buffer
IN	root	Identificador do processo que enviará a mensagem
IN	comm	comunicador

MPI_Bcast

Envia uma mensagem para todos outros processos de um comunicador

```
int mensagem = 7
int root     = 1
MPI_Bcast(&mensagem, 1, MPI_INT,root,MPI_COMM_WORLD)
```



MPI_Scatter

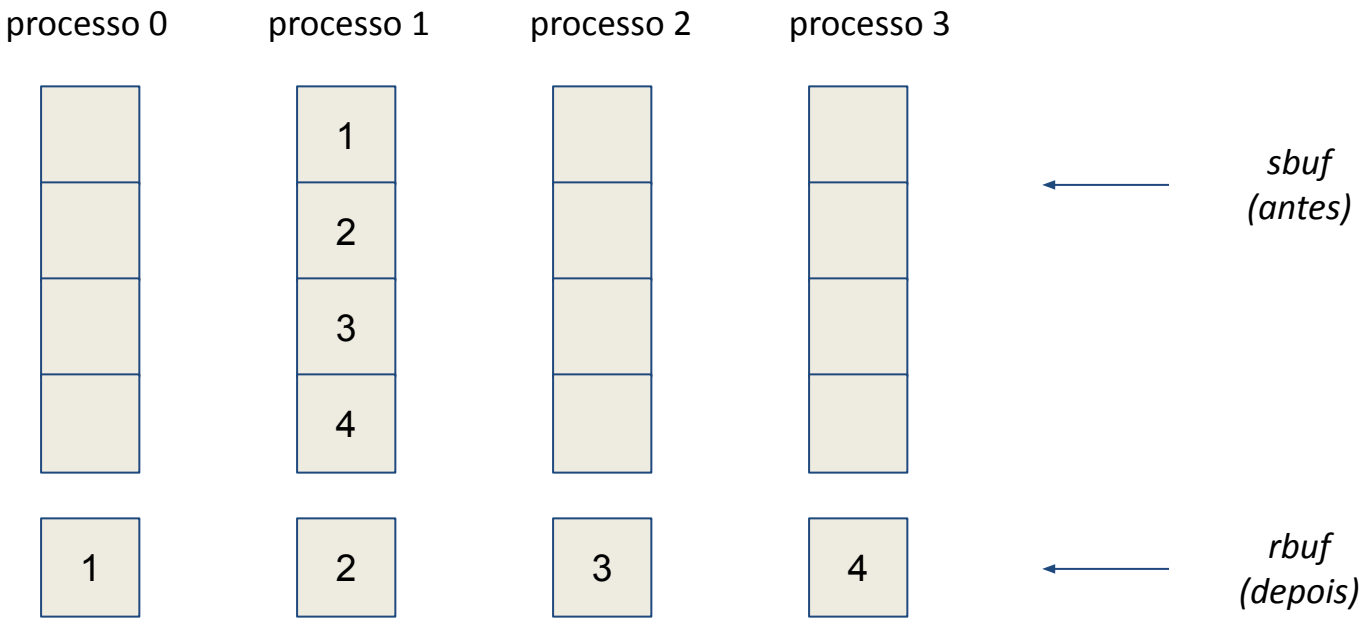
- O processo raiz (root) divide seu buffer de envio em n segmentos e envia
- Cada processo recebe um segmento do processo raiz e coloca em seu buffer de recebimento.

MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN	recvcount	Número de elementos recebidos
IN	Recvtype	Tipo dos dados recebidos
IN	Root	Processo que envia
IN	Comm	comunicador

MPI_Scatter

Distribui dados de um processo para todos os demais por processos de um comunicador

```
int sbuf[4] = {1,2,3,4};
int rbuf;
int root     = 1
MPI_Scatter(sbuf, 1, MPI_INT,&rbuf,1,MPI_INT,root,MPI_COMM_WORLD)
```



Fonte: https://computing.llnl.gov/tutorials/mpi/

```
#define SIZE 4
int main(int argc,char *argv[])
{
      int numtasks, rank, sendcount, recvcount, source;
      float sendbuf[SIZE][SIZE] = { {1.0, 2.0, 3.0, 4.0},
                                    {5.0, 6.0, 7.0, 8.0},
                                    {9.0, 10.0, 11.0, 12.0},
                                    {13.0, 14.0, 15.0, 16.0} };
      float recvbuf[SIZE];
      MPI_Init(&argc,&argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

      if (numtasks == SIZE) {
            source = 1;
            sendcount = SIZE;
            recvcount = SIZE;
            MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,MPI_FLOAT,
                        source,MPI_COMM_WORLD);
            printf("rank= %d Results: %f %f %f %f\n",rank,recvbuf[0],
            recvbuf[1],recvbuf[2],recvbuf[3]);
      } else
            printf("Must specify %d processors. Terminating.\n",SIZE);
      MPI_Finalize();
}
```

MPI_Gather

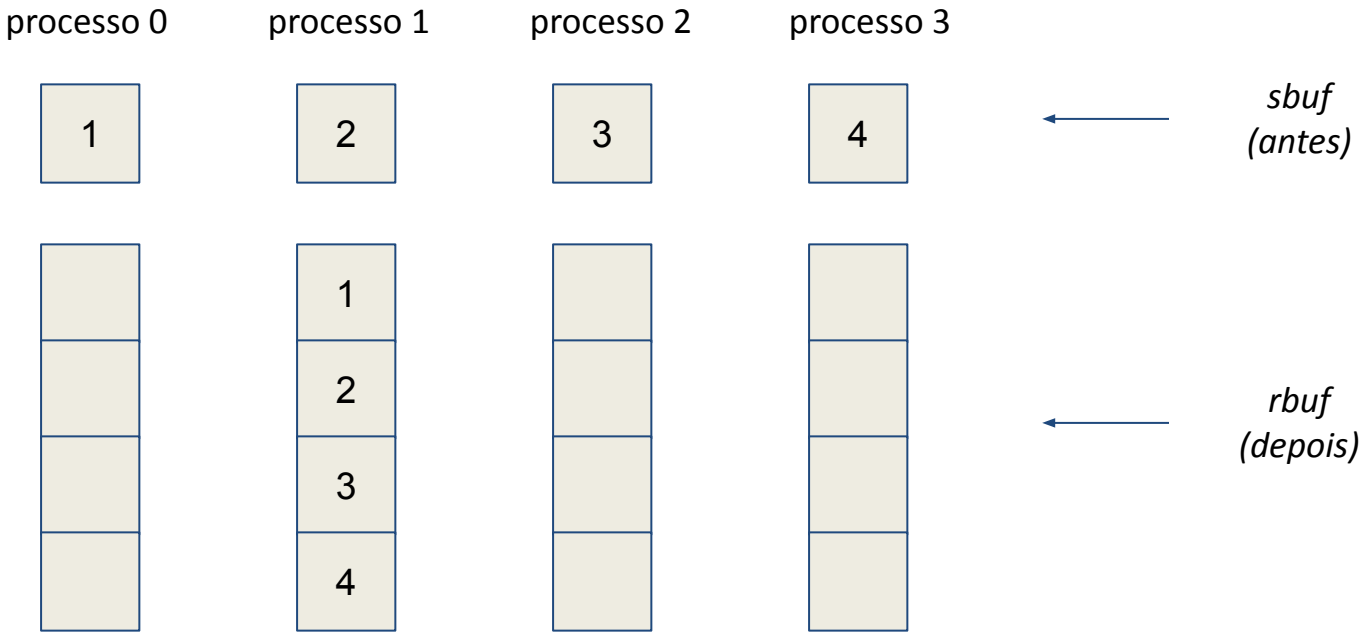
- Cada processo envia o conteúdo do seu buffer de envio para o processo raiz (root)
- O processo raiz armazena os dados em seu buffer de recebimento de acordo com o rank dos processos remetentes
- Reverso do MPI_Scatter

`MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)`		
IN	sendbuf	início do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN	recvcount	Número de elementos recebidos
IN	recvtype	Tipo dos dados recebidos
IN	Root	Processo que envia
IN	Comm	comunicador

MPI_Gather

Recolhe dados de vários processos de comunicador para apenas um processo

```
int sbuf;
int rbuf[4];
int root  = 1
MPI_Gather(&sbuf, 1, MPI_INT,rbuf,1,MPI_INT,root,MPI_COMM_WORLD)
```



Fonte: https://computing.llnl.gov/tutorials/mpi/

```
int main (int argc, char **argv) {
     int myrank, size, recvbufflen = 0;
     int *recvbuffer;
     int sendbuffer[2];
     MPI_Status status;

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

     for(int i=0;i<2;i++)
          sendbuffer[i] = i*2;
     if (myrank==0){
          recvbufflen = 2*size;
          recvbuffer = (int*)malloc(recvbufflen * sizeof(int));
     }

     MPI_Gather(sendbuffer,2,MPI_INT,recvbuffer,2,MPI_INT,0,MPI_COMM_WORLD);
     if(myrank==0){
          for(int i=0;i<recvbufflen;i++)
               printf("recvbuffer[%d]=%d\n",i,recvbuffer[i]);
     }
     MPI_Finalize();
     return 0;
}
```

MPI_Allgather

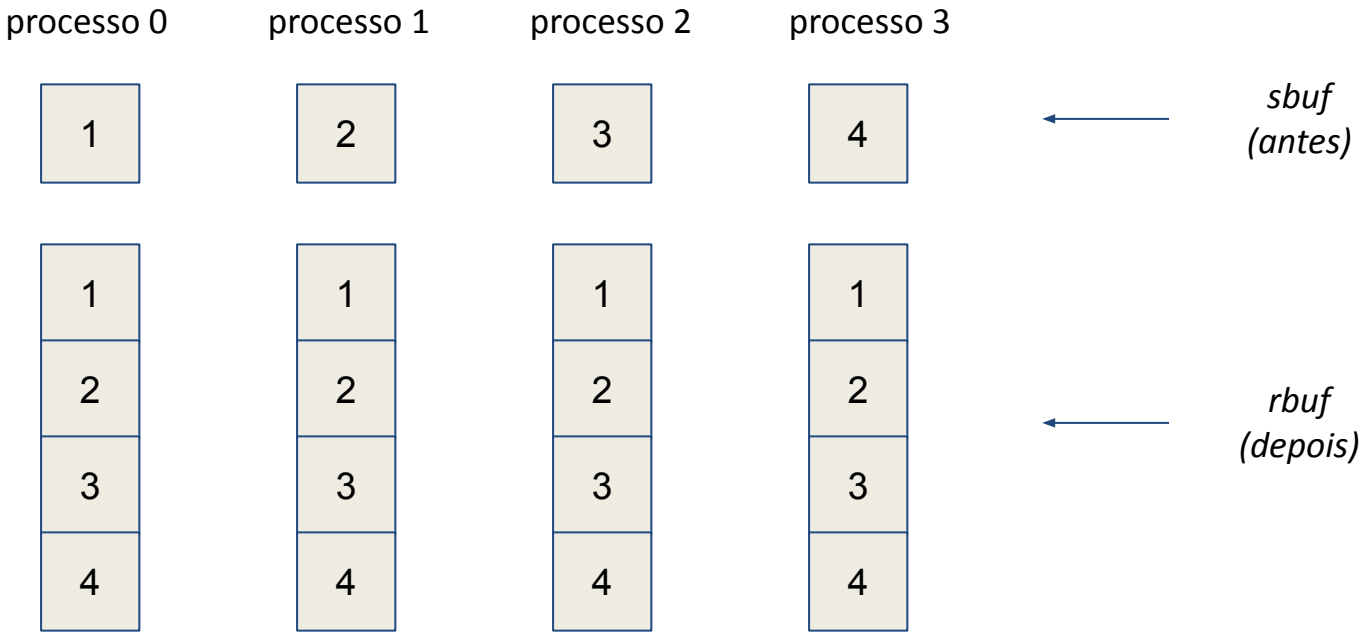
- Um MPI_Gather cujo resultado final abrange todos os processos
- Cada processo de um grupo executa um um-para-um broadcast dentro do grupo

MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN	recvcount	Número de elementos recebidos
IN	recvtype	Tipo dos dados recebidos
IN	comm	comunicador

MPI_Allgather

Recolhe dados de vários processos de comunicador e redistribui para todos

```
int sbuf;
int rbuf[4];
int root  = 1
MPI_AllGather(&sbuf, 1, MPI_INT,rbuf,1,MPI_INT,MPI_COMM_WORLD)
```



Fonte: https://computing.llnl.gov/tutorials/mpi/

```
int main (int argc, char **argv) {
     int myrank, size, recvbufflen = 0;
     int *recvbuffer;
     int sendbuffer[2];
     MPI_Status status;

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

     for(int i=0;i<2;i++)
          sendbuffer[i] = i*2;

     recvbufflen = 2*size;
     recvbuffer = (int*)malloc(recvbufflen * sizeof(int));

     MPI_Allgather(sendbuffer,2,MPI_INT,recvbuffer,2,MPI_INT,MPI_COMM_WORLD);

     for(int i=0;i<recvbufflen;i++)
          printf("recvbuffer[%d]=%d\n",i,recvbuffer[i]);

     free(recvbuffer);
     MPI_Finalize();
     return 0;
}
```

MPI_Alltoall

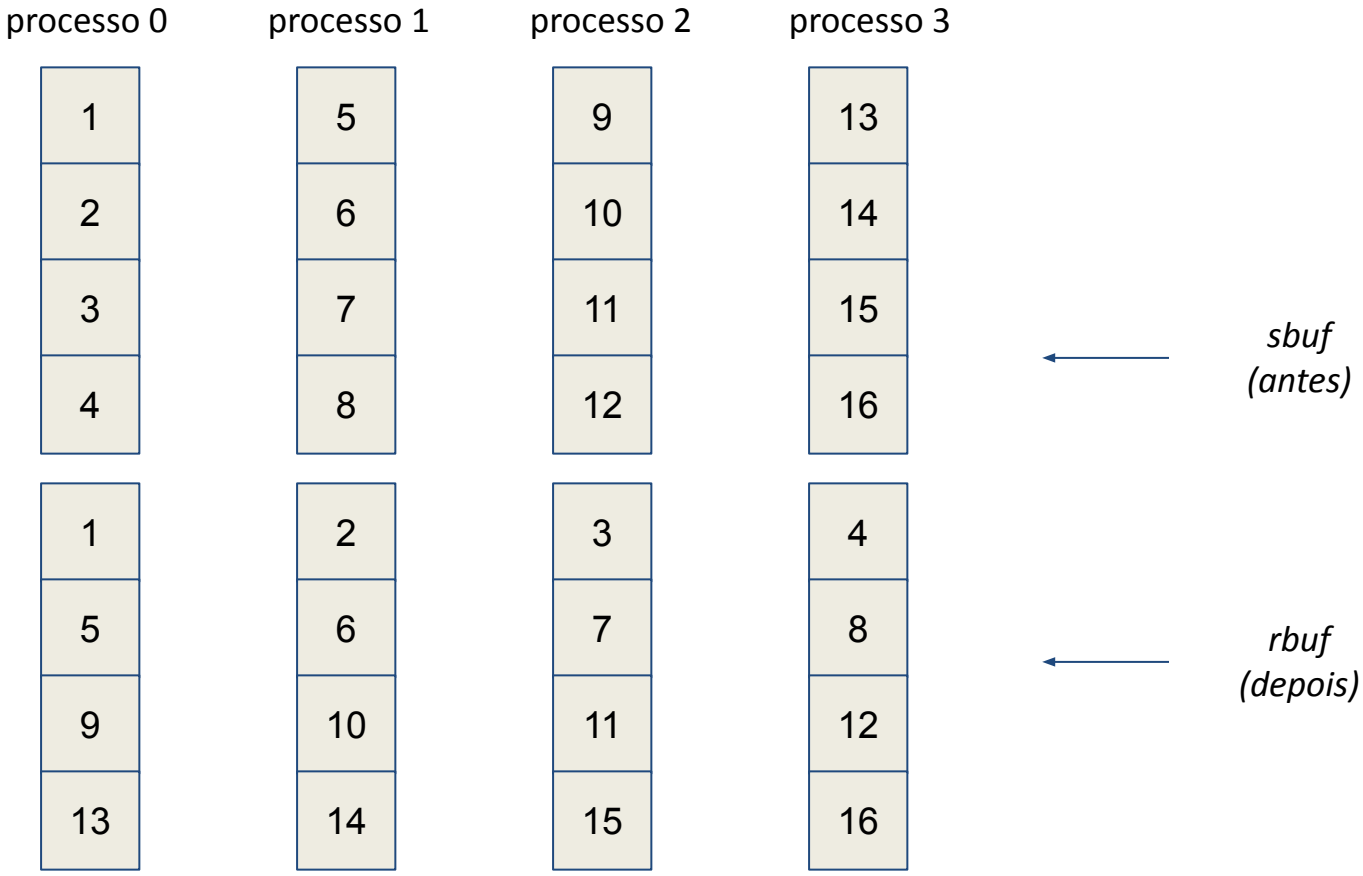
- Cada processo de um grupo executa uma operação de scatter, enviando uma mensagem distinta para os demais processos do grupo em ordem pelo rank.

MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN	recvcount	Número de elementos recebidos
IN	recvtype	Tipo dos dados recebidos
IN	comm	comunicador

MPI_Allgather

Cada processo realiza uma operação de scatter

```
MPI_Alltoall(sbuf,4, MPI_INT,rbuf,4,MPI_INT,MPI_COMM_WORLD)
```



Fonte: https://computing.llnl.gov/tutorials/mpi/

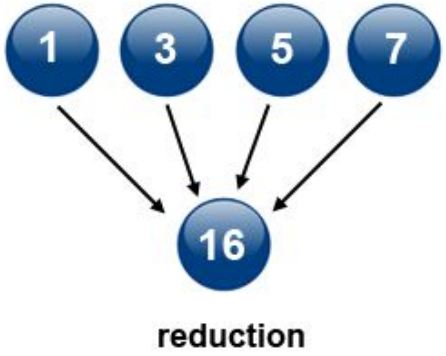
Outras operações de transferência

- **MPI_Gatherv:** Coleta dados variáveis de todos os membros de um grupo para um membro. A função MPI_Gatherv adiciona flexibilidade à função MPI_Gather permitindo uma contagem variável de dados de cada processo.

- **MPI_Scatterv:** Dispersa dados de um membro em todos os membros de um grupo. A função MPI_Scatterv executa o inverso da operação executada pela função MPI_Gatherv .

Operação de Redução

- Aplica uma operação de redução em todos os processos de um grupo e coloca o resultado do buffer de recebimento do processo raiz (root).
- Possíveis operações são MPI_SUM, MPI_MAX, MPI_MIN,...

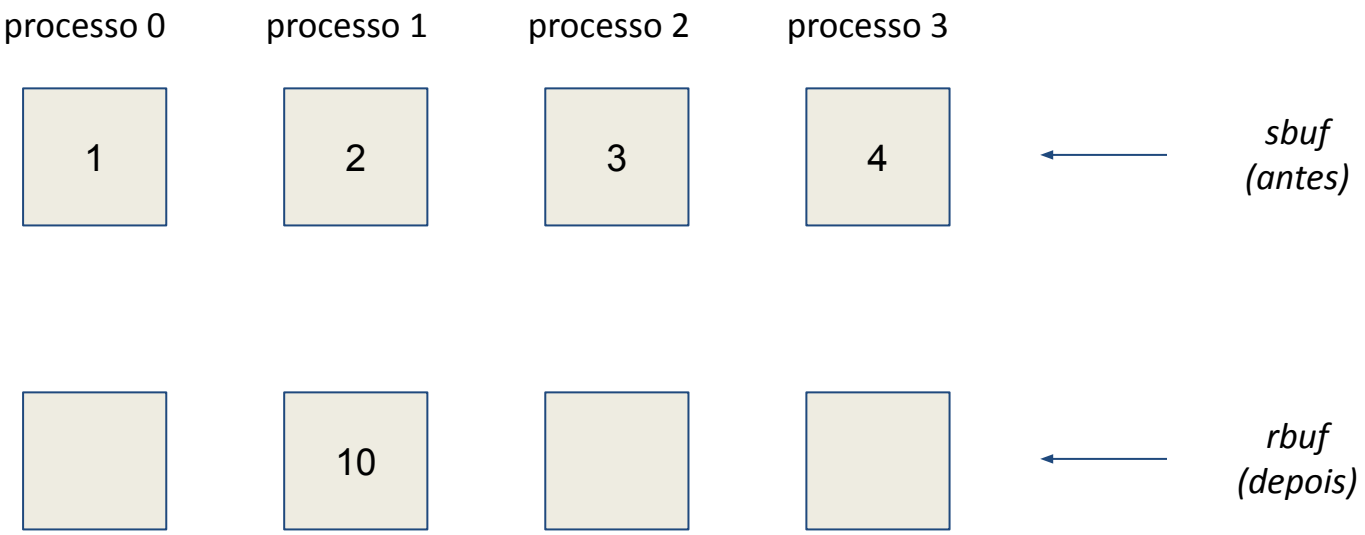


MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
OUT	recvbuf	endereço de memória do buffer de recebimento no processo root
IN	count	número de elementos enviados
IN	datatype	tipo dos dados do buffer de envio
IN	op	operação de redução
IN	root	rank do processo root
IN	comm	comunicador

MPI_Reduce

Opera sobre os dados dos processos colocando o resultado no processo de destino

```
int sbuf;
int rbuf;
int destino = 1
MPI_Reduce(&sbug,&rbuf,1,MPI_INT,MPI_SUM,destino,MPI_COMM_WORLD)
```



```
#define ARRAYSIZE 100

 int main(int argc , char **argv)
{
    int size, rank,i=0,localsum=0,globalsum=0;
    int data[ARRAYSIZE];

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

    int start = floor(rank*ARRAYSIZE/size);
    int end   = floor((rank+1)*ARRAYSIZE/size)-1;

    for(i=start;i<=end;i++)
    {
        data[i] = i*2;
        localsum = localsum + data[i];
    }

    MPI_Reduce(&localsum,&globalsum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);

    if(rank==0)
        printf("The global sum =%d\n",globalsum);

    MPI_Finalize();

}
```

MPI_Allreduce

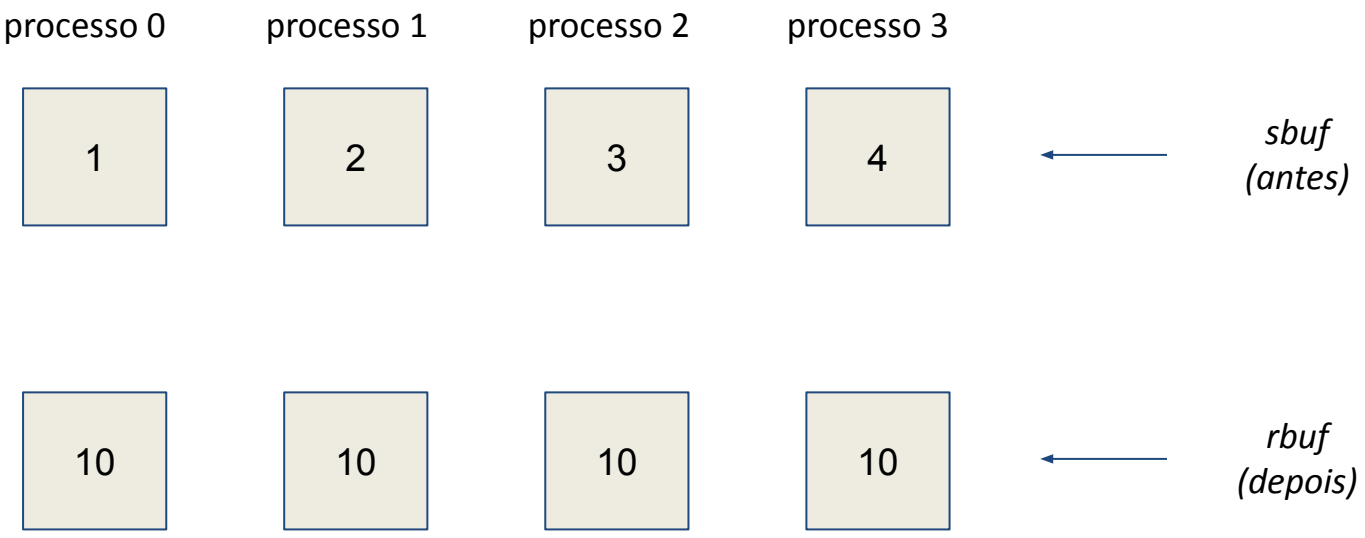
- Aplica uma operação de redução em todos os processos de um grupo e coloca o resultado do buffer de recebimento de todos os processos do grupo.
- Equivalente a um MPI_Reduce seguido pelo MPI_Bcast

MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
OUT	recvbuf	endereço de memória do buffer de recebimento
IN	count	número de elementos enviados
IN	datatype	tipo dos dados do buffer de envio
IN	op	operação de redução
IN	comm	comunicador

MPI_Allreduce

Opera sobre os dados dos processos colocando o resultado no processo de destino

```
int sbuf;
int rbuf;
MPI_Allreduce(&sbug,&rbuf,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD)
```



Lista Completa das Operações de Redução

Nome	Operação
MPI_MAX	máximo
MPI_MIN	mínimo
MPI_SUM	soma
MPI_PROD	produto
MPI_LAND	E lógico
MPI_BAND	E bit-a-bit
MPI_LOR	Ou lógico
MPI_BOR	OU bit-a-bit
MPI_LXOR	Ou-exclusivo
MPI_BXOR	XOR bit-a-bit
MPI_MAXLOC	Qual processo tem o máximo
MPI_MINLOC	Qual processo tem o mínimo

MPI_Scan

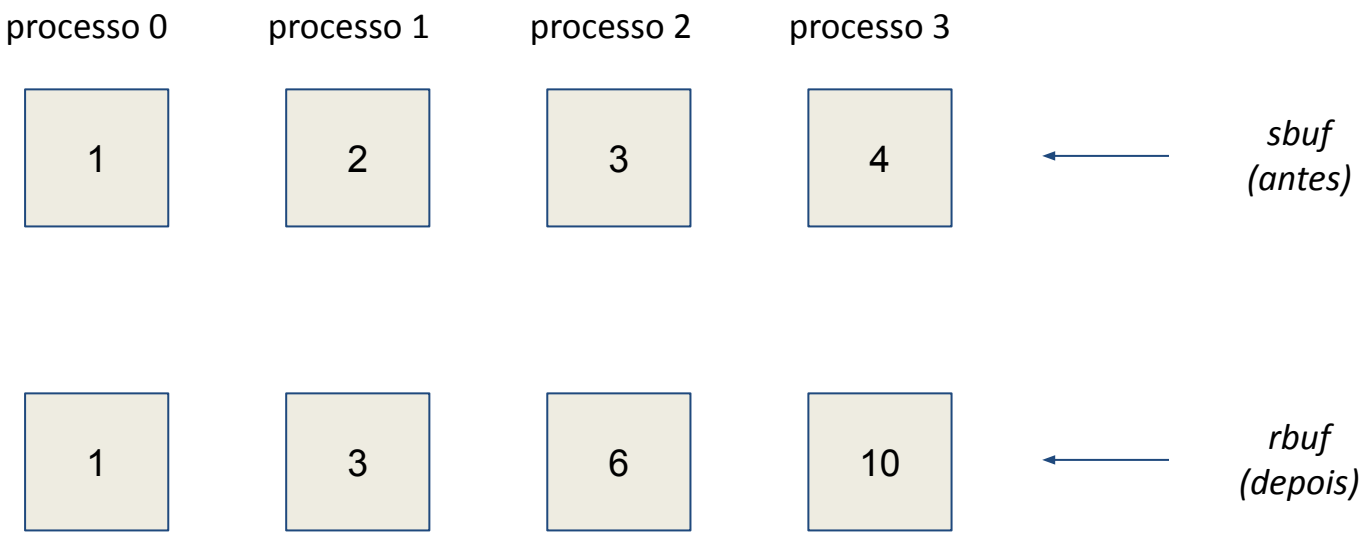
- Aplica uma operação de redução em todos os processos de um grupo e colocando o resultado parcial no buffer de recebimento dos processos

MPI_Scan(sendbuf, recvbuf, count, datatype, op, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
OUT	recvbuf	endereço de memória do buffer de recebimento
IN	count	número de elementos enviados
IN	datatype	tipo dos dados do buffer de envio
IN	op	operação de redução
IN	comm	comunicador

MPI_Scan

Calcula a reduções parciais de dados em uma coleção de processos.

```
int sbuf;
int rbuf;
MPI_Scan(&sbug,&rbuf,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD)
```



Definindo Operações

int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

onde

- function: função definida pelo usuário
- Commute: true se a operação é comutativa.

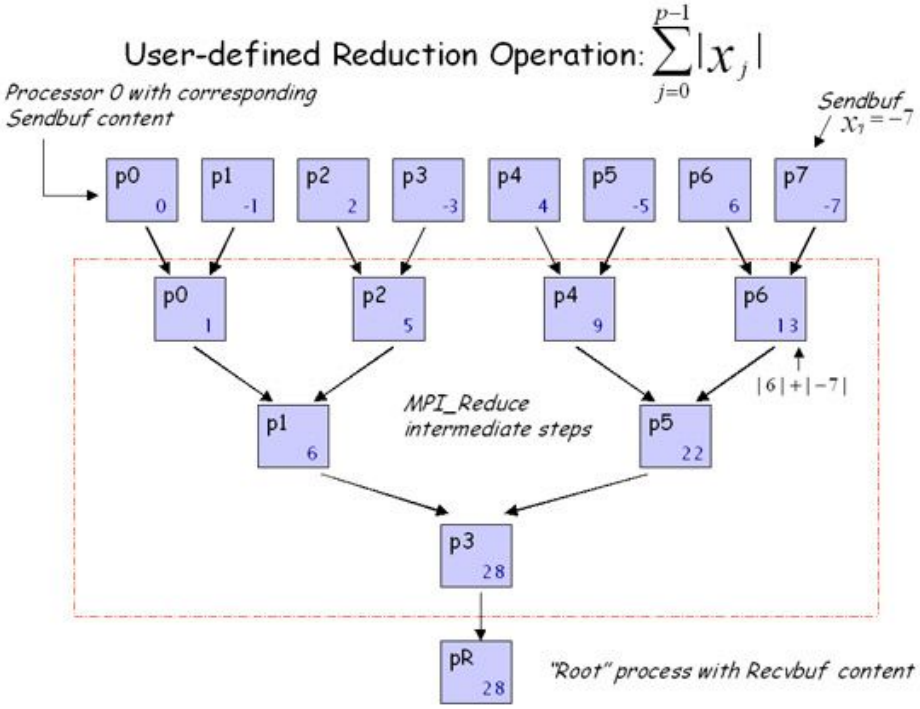
A função do usuário deve ser definida como:

*void myfunc(void *in, void *inout, int *len, MPI_Datatype *datatype)*

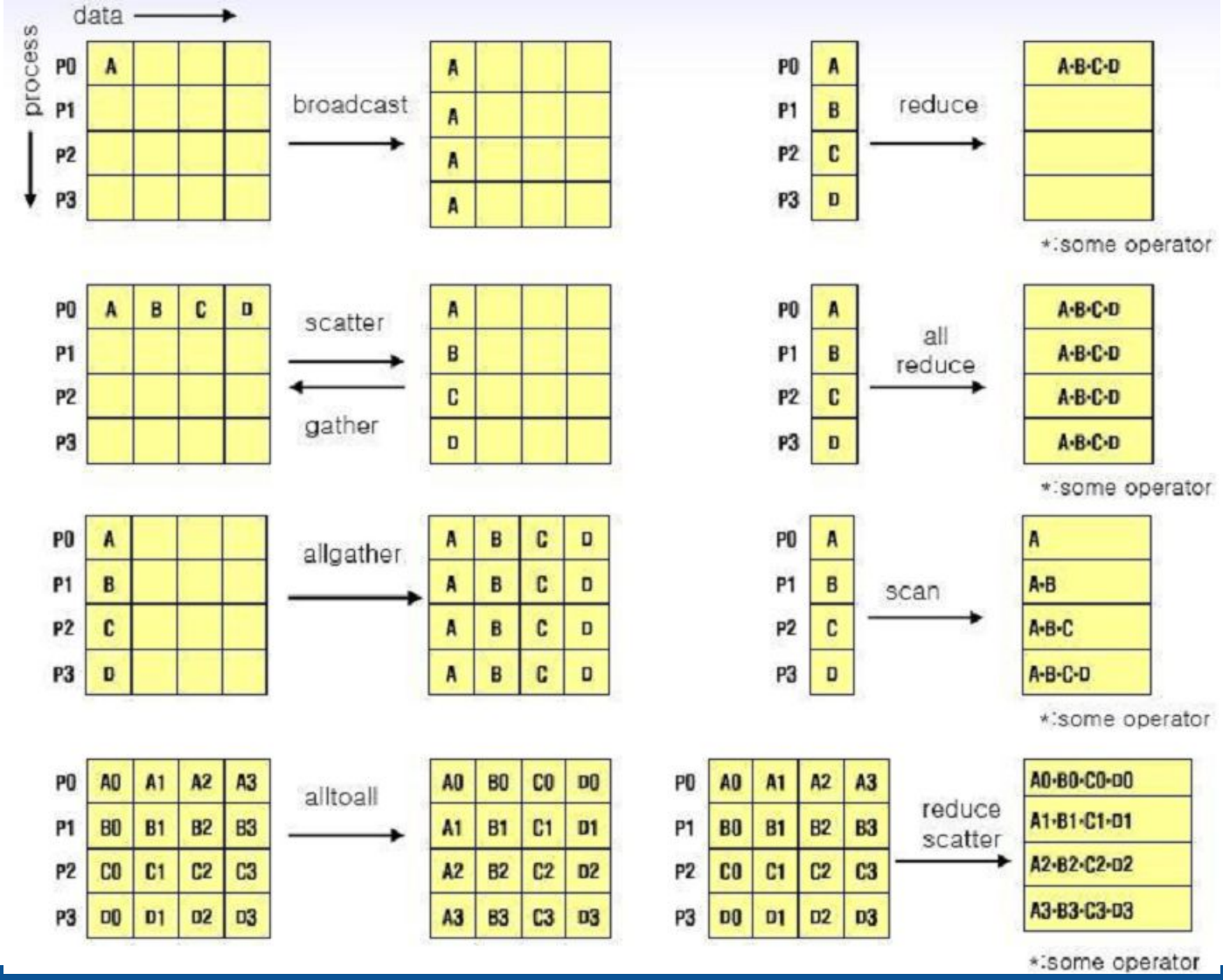
Exemplo: Cálculo da norma 1:

$$N_1(x) = \sum_{j=0}^{p-1} |x_j|$$

```
void onenorm(float *in, float *inout, int *len, MPI_Datatype *type)
{
      int i;
      for (i=0; i<*len; i++) {
          *inout = fabs(*in) + fabs(*inout);
           in++;
           inout++;
      }
}

void main(int argc, char* argv[])
{
      int root=0, p, myid; float sendbuf, recvb
      MPI_Op myop;
      int commute=0;

      MPI_Init(&argc, &argv);
      MPI_Comm_size(MPI_COMM_WORLD, &p);
      MPI_Comm_rank(MPI_COMM_WORLD, &myid);
      MPI_Op_create(onenorm, commute, &myop);

      sendbuf = myid*(-1)^myid;

      MPI_Barrier(MPI_COMM_WORLD);
      MPI_Reduce(&sendbuf, &recvbuf, 1, MPI_FLOAT, myop, root,
      MPI_COMM_WORLD);
      if(myid == root)
              printf("The operation yields %f\n", recvbuf);
      MPI_Finalize();
}
```



Comunicação Coletiva (resumo)



Gerenciando Grupos e Comunicadores

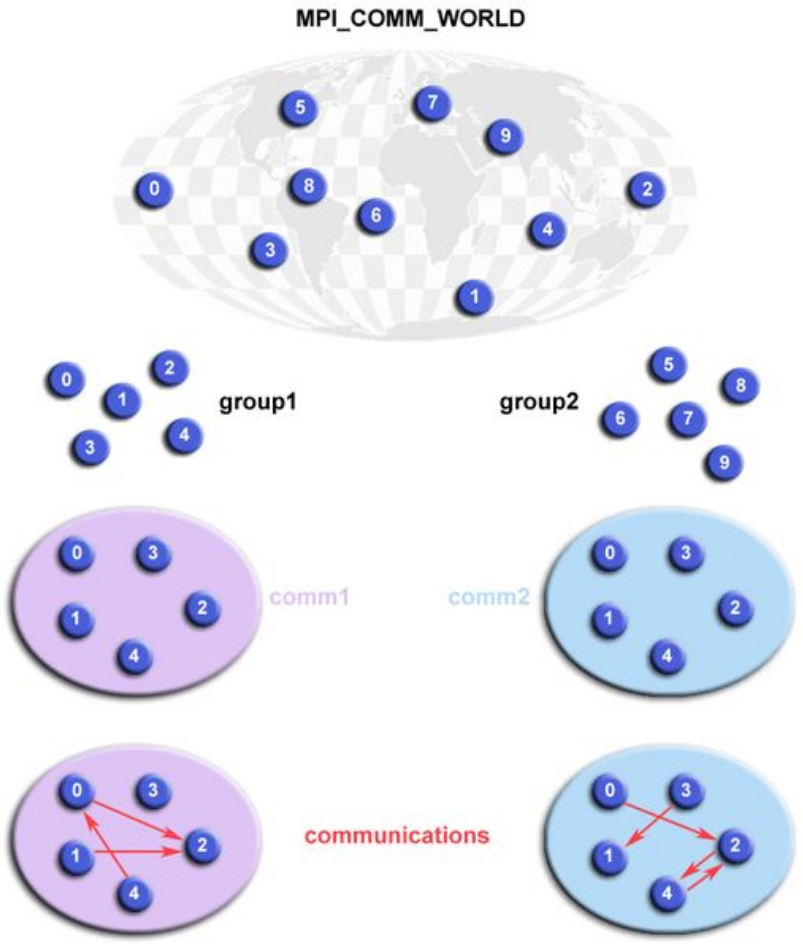
- Um **grupo** é um **conjunto ordenado de processos**. Cada processo em um grupo está identificado com um inteiro único. Os valores de classificação começa em zero e vão até N-1, onde N é o número de processos no grupo. Um grupo está sempre associada a um objeto comunicador.

- **Um comunicador engloba um grupo de processos** que podem comunicar uns com os outros. **Todas as mensagens MPI deve especificar um comunicador**. No sentido mais simples, o comunicador é uma "tag" extra que deve ser incluído com as chamadas MPI. Por exemplo, o identificador para o comunicador que compreende todas as tarefas é MPI_COMM_WORLD.

- Do ponto de vista do programador, um grupo e um comunicador são um só. As rotinas de grupo são principalmente utilizados para especificar quais os processos devem ser utilizados para construir um comunicador

Gerenciando Grupos e Comunicadores

- Principais objetivos do grupo e Communicator:
  - Permitem organizar as tarefas em grupos de trabalho.
  - Permitir operações de comunicações coletiva através de um subconjunto de tarefas relacionadas.
  - Fornecer bases para a implementação de topologias virtuais definida pelo usuário/desenvolvedor.



Gerenciando Grupos e Comunicadores

- Considerações sobre programação e restrições:
  - Grupos / comunicadores são dinâmicos - podem ser criados e destruídos durante a execução do programa.
  - Os processos podem estar em mais do que um grupo/comunicador. Eles terão uma classificação única dentro de cada grupo/comunicador
  - MPI dispõe mais de 40 rotinas relacionadas a grupos, comunicadores e topologias virtuais.
- Uso típico:
  - Extrair o manipulador do grupo global do MPI_COMM_WORLD usando MPI_Comm_group
  - Formar novo grupo como um subconjunto do grupo global usando MPI_Group_incl
  - Criar novo comunicador para o novo grupo usando MPI_Comm_create
  - Determinar nova identificação no novo comunicador usando MPI_Comm_rank
  - Realizar comunicações usando rotina MPI usando o novo comunicador
  - Quando terminar, liberar a memória do novo comunicador e grupo (opcional) usando MPI_Comm_free e MPI_Group_free.

```
#define NPROCS 8
int main(int argc, char *argv[])
{
        int rank, new_rank, sendbuf, recvbuf, numtasks, ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
        MPI_Group orig_group, new_group;
        MPI_Comm new_comm;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        if (numtasks != NPROCS) {
              printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
               MPI_Finalize();
               exit(0);
        }
        sendbuf = rank;
        // Extraindo grupo original
        MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

        // Dividindo os processos entre dois grupos distintos
         if (rank < NPROCS/2) {
            MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
         }
        else {
            MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
        }
        // Criando novo comunicador  e  executando uma operação coletiva *
        MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
        MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
        MPI_Group_rank (new_group, &new_rank);
        printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
        MPI_Finalize();
}
```

Tipos de Dados Derivados

- MPI_Type_contiguous:
  - O construtor simples. Produz um novo tipo de dados, agrupando elementos de um tipo de dados existente.

MPI_Type_contiguous (count,oldtype,&newtype)

Parâmetros:

- *count*
  O número de elementos no novo tipo de dados.
- *oldtype*
  O tipo de dados MPI de cada elemento.
- *newtype* [out]
  No retorno, contém um identificador MPI_Datatype que representa o novo tipo de dados.

MPI_Type_contiguous

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

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

a[4][4]

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

9.0	10.0	11.0	12.0

1 element of rowtype

```
#define SIZE 4
main(int argc, char *argv[])
{
  int numtasks, rank, source=0, dest, tag=1, i;
  float a[SIZE][SIZE] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
                         9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0};
  float b[SIZE];
  MPI_Status stat;
  MPI_Datatype rowtype; // required variable
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
  MPI_Type_commit(&rowtype);
  if (numtasks == SIZE)
  { // task 0 sends one element of rowtype to all tasks
    if (rank == 0) {
     for (i=0; i<numtasks; i++)
        MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
    } // all tasks receive rowtype data from task 0
    MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);

  } else
     printf("Must specify %d processors. Terminating.\n",SIZE);

  MPI_Type_free(&rowtype);
  MPI_Finalize();
}
```

Tipos de Dados Derivados

- MPI_Type_vector:
  - Semelhante ao contiguous, mas permite espaçamento regulares entre os elementos

MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)

MPI_Type_vector

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

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

a[4][4]

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

2.0	6.0	10.0	14.0

1 element of columntype

```
#define SIZE 4
int main(int argc, char *argv[]) {
      int numtasks, rank, source=0, dest, tag=1, i;
      float a[SIZE][SIZE] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
                             9.0, 10.0, 11.0, 12.0,13.0, 14.0, 15.0, 16.0};
      float b[SIZE];
      MPI_Status stat;
      MPI_Datatype columntype; // required variable
      MPI_Init(&argc,&argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
      MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
      MPI_Type_commit(&columntype);
      if (numtasks == SIZE) {
            if (rank == 0) {
                  for (i=0; i<numtasks; i++)
                        MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
            }
            MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
            printf("rank= %d b= %3.1f %3.1f %3.1f %3.1f\n",
            rank,b[0],b[1],b[2],b[3]);
      }
      MPI_Type_free(&columntype);
      MPI_Finalize();
}
```

Tipos de Dados Derivados

- MPI_Type_commit:
  - Commit o novo tipo de dados para o sistema. Obrigatório para todos os tipos derivados de dados construídos pelo usuário.

```
MPI_Type_commit (&datatype)
```

- MPI_Type_free
  - Desaloca o tipo de dados especificado. A utilização desta rotina é especialmente importante para evitar o esgotamento de memória em situações onde o tipo de dados é criado diversas vezes

```
MPI_Type_free (&datatype)
```

Tipos de Dados Derivados

- MPI_Type_create_struct
  - Rotina mais flexível para criar um tipo de dado MPI.
  - Constrói o MPI_Datatype como uma sequência de blocos, sendo que cada bloco é criado replicando um MPI_Datatype existente um certo número de vezes.

```
int MPI_Type_create_struct(int block_count,
                           const int block_lengths[],
                           const MPI_Aint displacements[],
                           MPI_Datatype block_types[],
                           MPI_Datatype* new_datatype);
```

Tipos de Dados Derivados

MPI_Type_create_struct(block_count,block_lengths,displs,block_types,new_datatype);

Parâmetros:

- block_count
  - O número de blocos a serem criados.
- block_lengths
  - Array contendo o comprimento de cada bloco.
- displs
  - Array contendo o deslocamento para cada bloco, expresso em bytes. O deslocamento é a distância entre o início do tipo de dado MPI criado e o início do bloco. É diferente do passo usado em MPI_Type_vector, por exemplo, onde ele expressa a distância entre o início de um bloco para o próximo.
- block_types
  - Array contendo os tipos de dados MPI a serem replicados para cada bloco.
- new_datatype
  - A variável na qual armazenar o tipo de dado criado.

Exemplo: MPI_Type_create_struct

- Considere a seguinte estrutura de dados:

```
struct person_t {
    int age;
    double height;
    char name[10];
};
```

- Escreva um programa mpi onde o processo 0 envie os dados de uma pessoa para o processo 1.

Exemplo: MPI_Type_create_struct (1 / 3)

```
40  struct person_t
41  {
42      int age;
43      double height;
44      char name[10];
45  };
46
47  int main(int argc, char* argv[])
48  {
49      MPI_Init(&argc, &argv);
50
51      // Get the number of processes and check only 2 processes are used
52      int size;
53      MPI_Comm_size(MPI_COMM_WORLD, &size);
54      if(size != 2)
55      {
56          printf("This application is meant to be run with 2 processes.\n");
57          MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
58      }
```

Exemplo: MPI_Type_create_struct (2 / 3)

```
60    // Criar o tipo de dado MPI para a estrutura person_t
61    MPI_Datatype person_type;
62    int lengths[3] = { 1, 1, 10 };
63
64    // Calcular os deslocamentos
65    // Em C, por padrão, o preenchimento pode ser inserido entre os campos.
66    // MPI_Get_address permitirá obter o endereço de cada campo da estrutura
67    // e calcular o deslocamento correspondente em relação ao endereço base
68    // daquela estrutura. Os deslocamentos calculados dessa forma
69    // incluirão qualquer preenchimento, se houver.
70    MPI_Aint displacements[3];
71    struct person_t dummy_person;
72    MPI_Aint base_address;
73    MPI_Get_address(&dummy_person, &base_address);
74    MPI_Get_address(&dummy_person.age, &displacements[0]);
75    MPI_Get_address(&dummy_person.height, &displacements[1]);
76    MPI_Get_address(&dummy_person.name[0], &displacements[2]);
77    displacements[0] = MPI_Aint_diff(displacements[0], base_address);
78    displacements[1] = MPI_Aint_diff(displacements[1], base_address);
79    displacements[2] = MPI_Aint_diff(displacements[2], base_address);
80
81    MPI_Datatype types[3] = { MPI_INT, MPI_DOUBLE, MPI_CHAR };
82    MPI_Type_create_struct(3, lengths, displacements, types, &person_type);
83    MPI_Type_commit(&person_type);
```

Exemplo: MPI_Type_create_struct (3 / 3)

```
84      // Obter o rank do processo
85      enum rank_roles { SENDER, RECEIVER };
86      int my_rank;
87      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
88      switch(my_rank){
89          case SENDER:
90          {
91              // Send the message
92              struct person_t buffer;
93              buffer.age = 20;
94              buffer.height = 1.83;
95              strncpy(buffer.name, "Tom", 9);
96              buffer.name[9] = '\0';
97              printf("MPI process %d sends person:\n\t- age = %d\n\t- height = %f\n\t- name = %s
98              MPI_Send(&buffer, 1, person_type, RECEIVER, 0, MPI_COMM_WORLD);
99              break;
100         }
101         case RECEIVER:
102         {
103             // Receive the message
104             struct person_t received;
105             MPI_Recv(&received, 1, person_type, SENDER, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
106             printf("MPI process %d received person:\n\t- age = %d\n\t- height = %f\n\t- name =
107             break;
108         }
109     }
110     MPI_Finalize();
111     return EXIT_SUCCESS;
112 }
```

Estudo de Caso

Algoritmos Numéricos com Malha Regulares

- Muitas aplicações científicas envolvem a solução de equações diferenciais (EDPs)
- Muitos algoritmos para aproximar a solução de EDPs dependem da formação de um conjunto de equações algébricas
  - Diferenças finitas, elementos finitos, volumes finitos

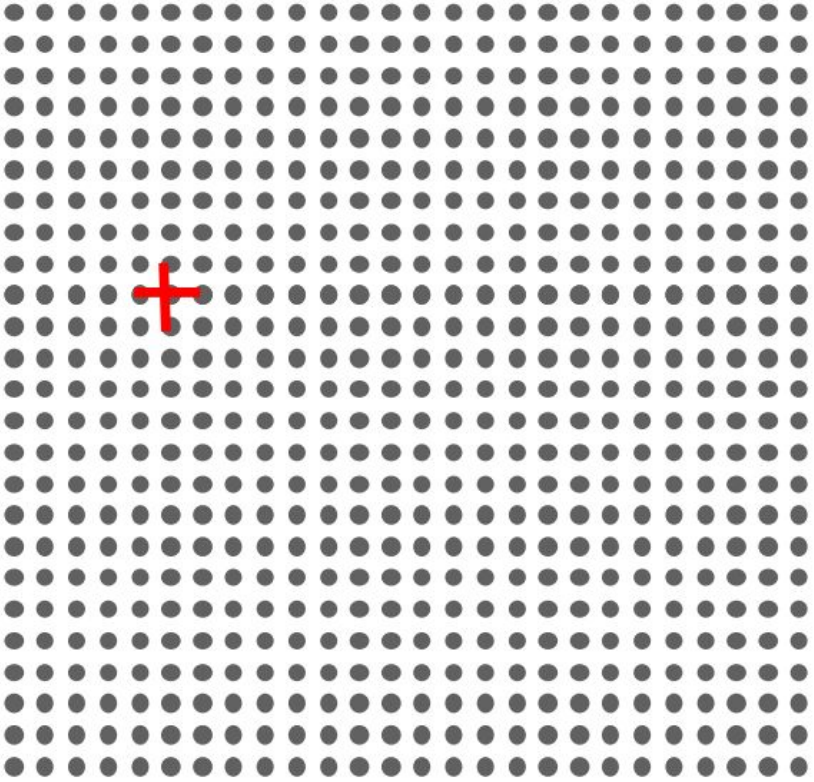
Equação Poisson 2D

- Para aproximar a solução do Problema de Poisson $\nabla^2 \boldsymbol{u} = \mathbf{f}$ em um quadrado unitário, com $\boldsymbol{u}$ definido nos limites do domínio (condições de contorno de Dirichlet), o seguinte esquema de diferença finitas de de 2ª ordem é freqüentemente usado:
  - $(\mathbf{u}(x+h,y) - 2\mathbf{u}(x,y) + \mathbf{u}(x-h,y))/h^2 + (\mathbf{u}(x,y+h) - 2\mathbf{u}(x,y) + \mathbf{u}(x,y-h))/h^2 = f(x,y)$
    - onde, a solução **u** será aproximada em um malha discreta com prontos x=0,h,2h,..., 1 e y=0,h,2h,...,1
    - Para simplificar a notação, $\mathbf{u}(ih,jh) = \mathbf{u}_{ij}$

- Isso é definido em uma malha discreta de pontos (x,y) = (ih,jh), para um espaçamento de malha “h”

Estrutura de Dados

- Cada círculo é um ponto na malha
- A equação de diferenças avaliada em cada ponto envolve os quatro vizinhos
- O “mais” vermelho é chamado de estêncil do método para caso 2D
- Bons algoritmos numéricos formam uma equação matricial Au=f;
- Outros resolvem de forma explícita:
  - $\mathbf{u}_{ij} = (\mathbf{u}_{i-1j} + \mathbf{u}_{i+1j} + \mathbf{u}_{ij-1} + \mathbf{u}_{ij+1})/4$

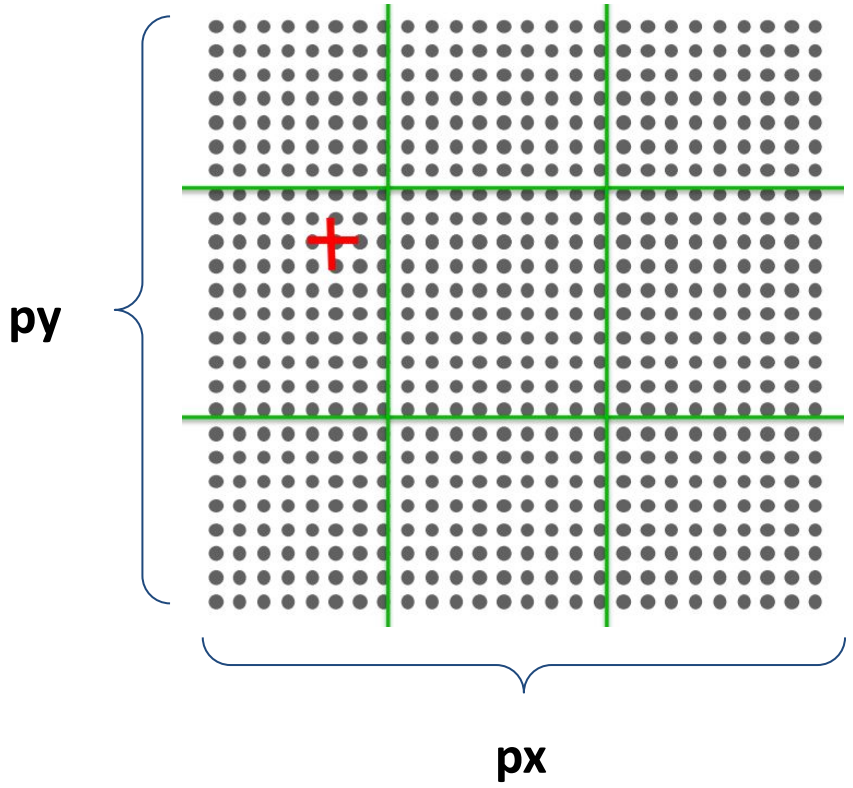


Código Stencil 2D Serial

```
int main(int argc, char **argv)
{
     // Alocação dos dados: u,u_old
     for(int iter=0; iter<niters; ++iter) {
          for(int j=1; j<n+1; ++j) {
               for(int i=1; i<n+1; ++i) {
                    unew[ind(i,j)] = appplyStencil(uold,i,j,n);
               }
          }
          for(int i=0; i<nsources; ++i) {
               unew[ind(sources[i][0],sources[i][1])] += energy; // heat source
          }
          // if(norm(unew - uold) < tolerance) break;
          tmp=anew; unew=uold; uold=tmp; // swap arrays
     }
     // Impressão dos resultados
}
```

Etapas Paralelização

- Especificar a decomposição do domínio por linha de comando (px, py)

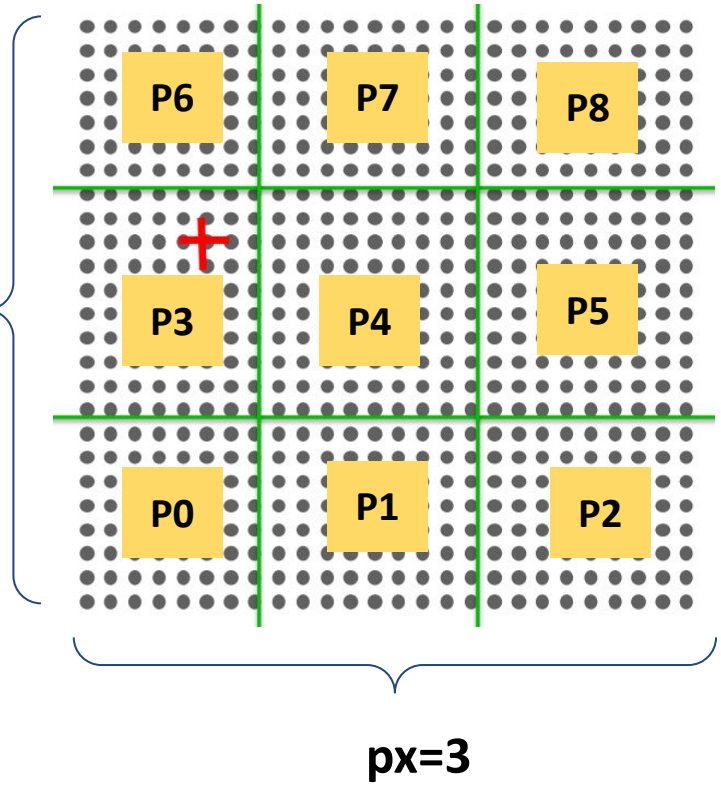


Etapas Paralelização

- Calcular a relação de vizinhança entre os processos

Exemplo:

- Para o processo P4 seus vizinhos são:
  - leste: P5
  - oeste: P3
  - norte: P4
  - sul: P1



Como determinar essa relação de maneira simples?

MPI_Cart_create

- Cria um comunicador a partir das informações de topologia cartesiana fornecidas.
  - Observação: existe uma função conveniente, MPI_Dims_create, que ajuda a organizar os processos MPI em um determinado número de dimensões.

```
int MPI_Cart_create(MPI_Comm old_communicator,
                    int dimension_number,
                    const int* dimensions,
                    const int* periods,
                    int reorder,
                    MPI_Comm* new_communicator);
```

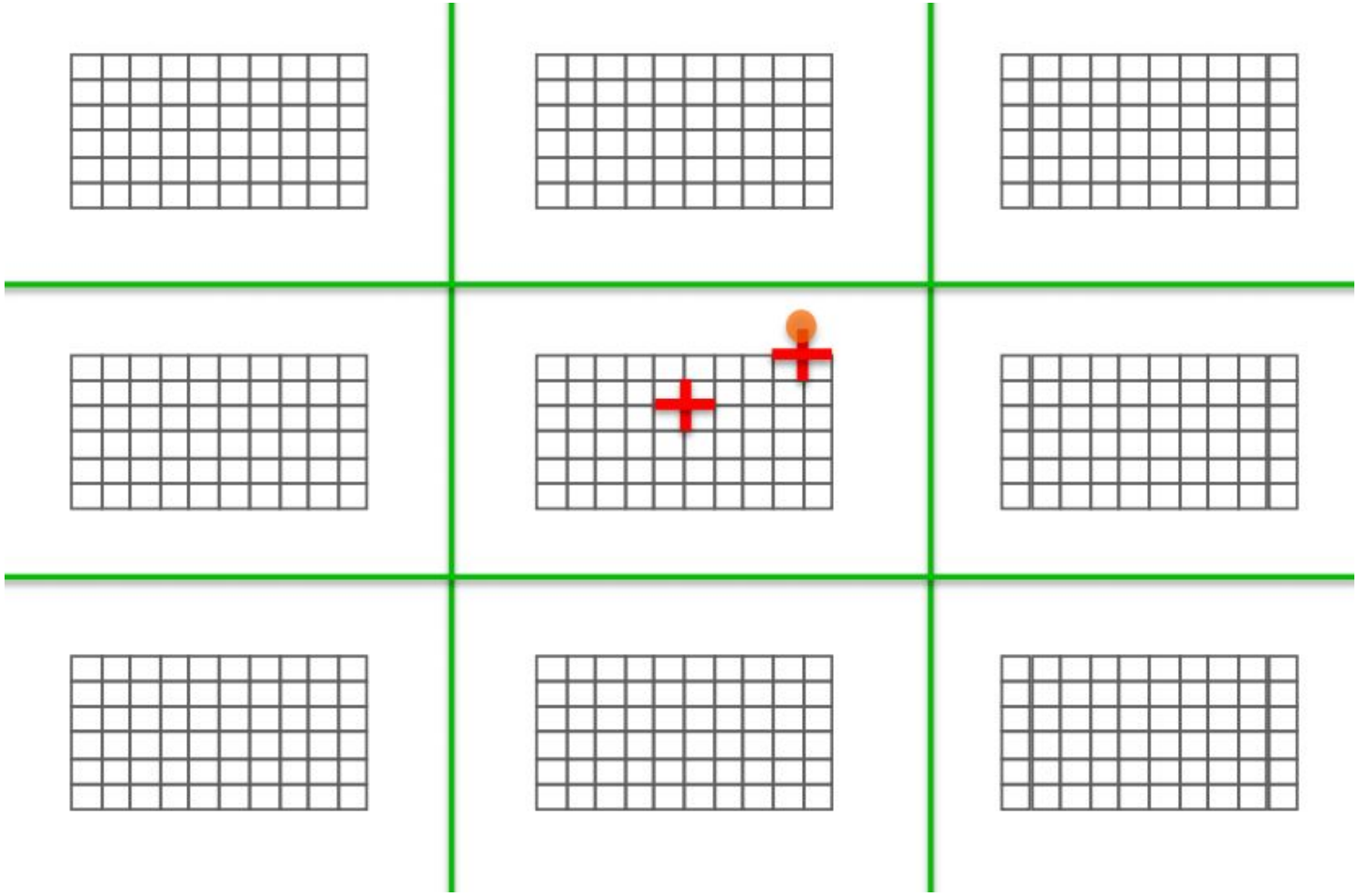
MPI_Cart_create

MPI_Cart_create(old_comm, n_dim, dims,periods, reorder, new_comm);

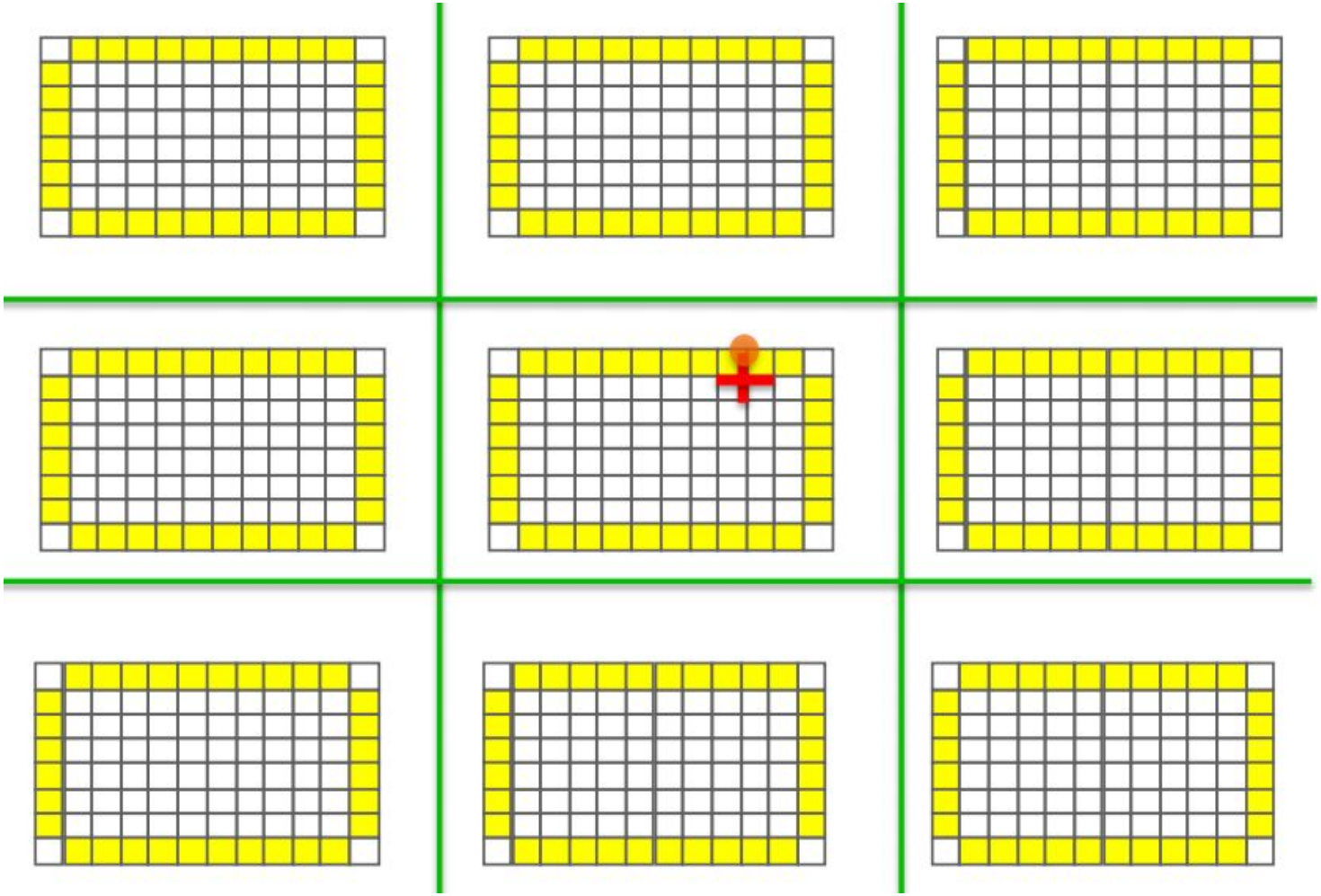
Parâmetros:

- old_comm
  - Comunicador contendo os processos a serem usados na criação do novo comunicador.
- n_dim
  - O número de dimensões na grade cartesiana.
- dims
  - O array contendo o número de processos a serem atribuídos a cada dimensão.
- periods
  - O array contendo a periodicidade para cada dimensão. Indica, para cada dimensão, se ela é periódica (true) ou não periódica (false).
- reorder
  - Indica se os processos devem preservar sua classificação do comunicador antigo para o novo.
- new_communicator
  - Contém o novo comunicador criado. Certos processos podem obter MPI_COMM_NULL no caso de a grade cartesiana fornecida precisar de menos processos do que os contidos no comunicador antigo.

Necessidade de Transferência de Dados

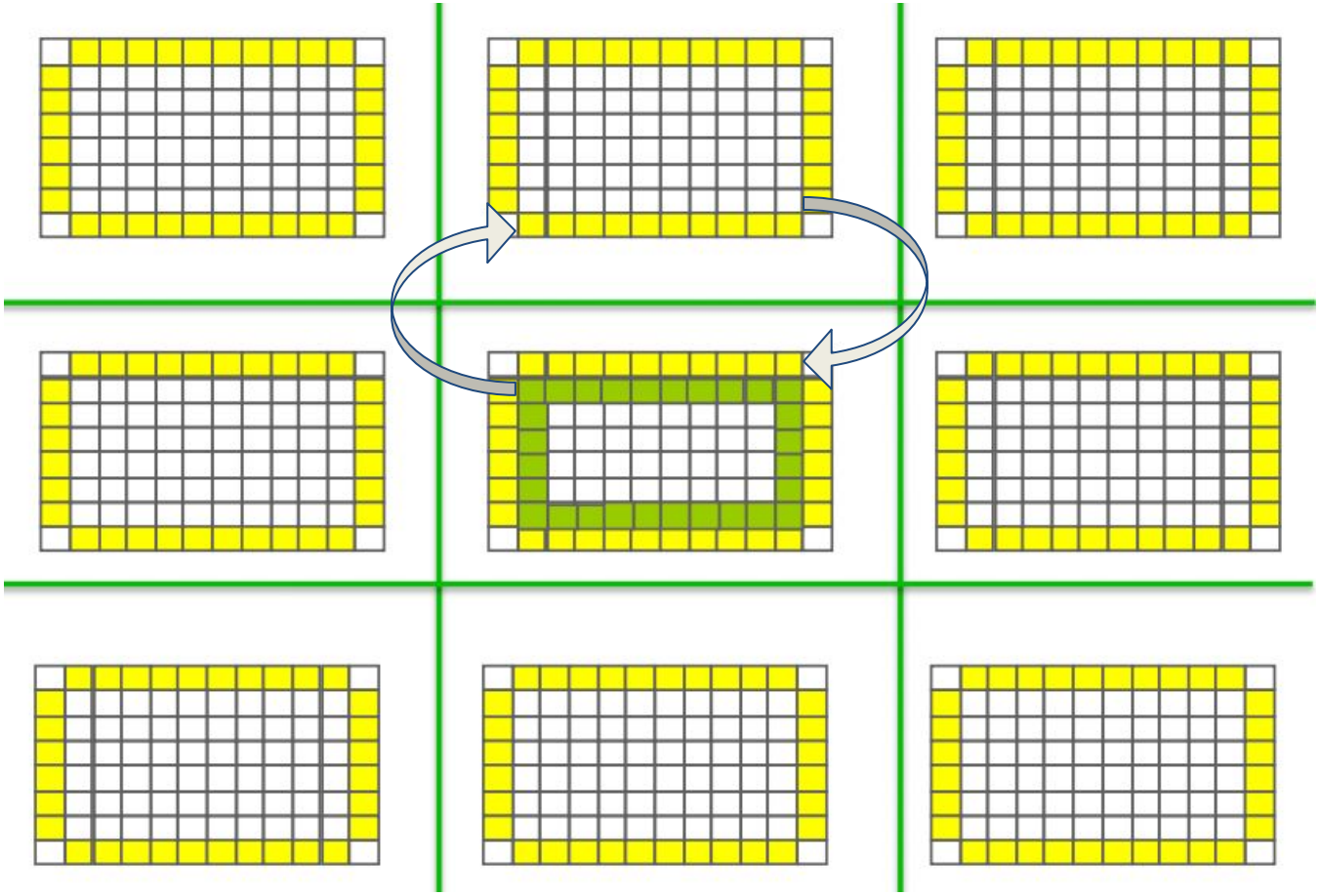


Adicionar zona de vizinhança (halo)



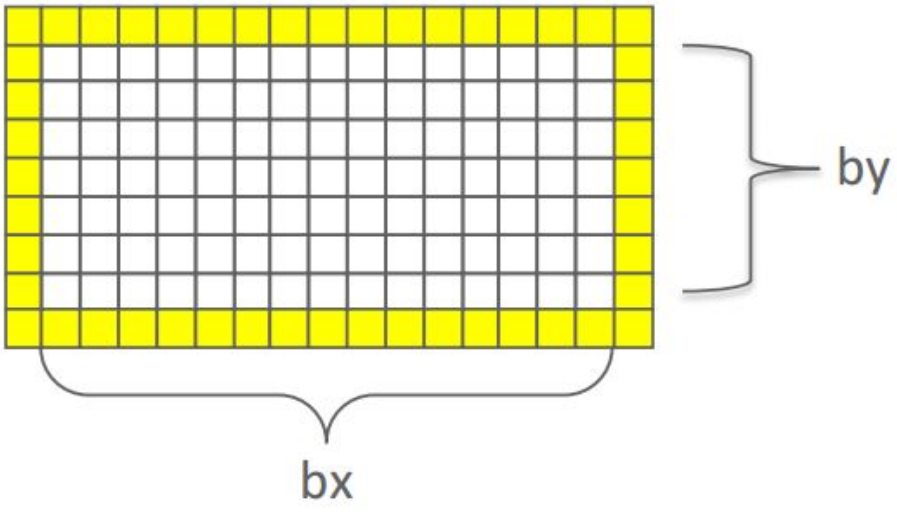
Adicionar zona de vizinhança (halo)

- Devemos provê acesso a dados remotos por meio de uma troca de dados da vizinhança (estêncil de 5 pontos)



Estrutura de Dados Local

- Cada processo tem seu “pedaço” local do array global
  - bx e by são as dimensões do array local
  - Sempre devemos alocar espaço ao redor do array local para dados fantasmas
  - Array deve ser alocado com dimensão (bx+2) x (by+2)



Código Stencil 2D - Paralelo

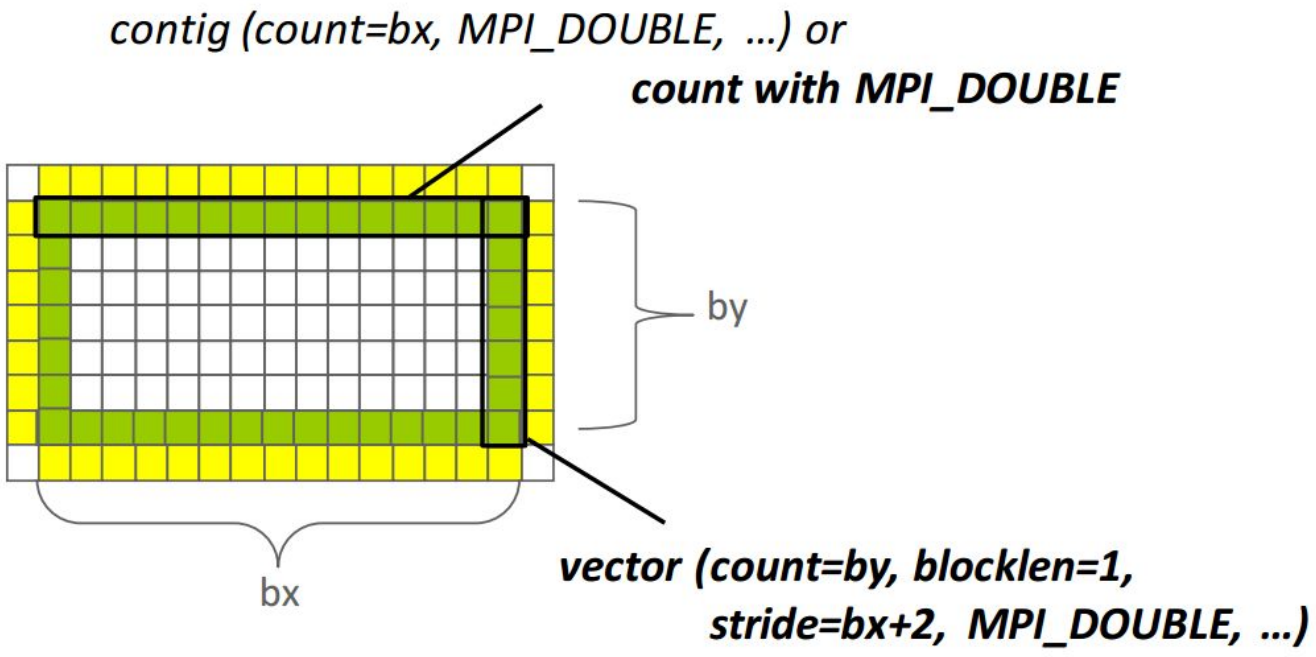
```
int main(int argc, char **argv)
{
    // ALOCAR uold e unew COM DIMENSÕES APROPRIADAS PARA A PARTIÇÃO
    // ALOCAR BUFFER ENVIO/RECEBIMENTO
    for(int iter=0; iter<niters; ++iter)
    {
        // COMUNICAR DADOS DA VIZINHANÇA
        for(int j=1; j<nlocal+1; ++j) {
            for(int i=1; i<nlocal+1; ++i) {
                unew[ind(i,j)] = appplyStencil(uold,i,j,n);
                energy += unew[ind(i,j)]
            }
        }
        // REDUÇÃO PARA CÁLCULO ENERGY
        for(int i=0; i<nsources; ++i)
            unew[ind(sources[i][0],sources[i][1])] += energy; // heat source
        tmp=anew; unew=uold; uold=tmp; // swap arrays
    }
    // IMPRESSÃO DADOS
}
```

Exercício 1

1. Dado o código serial disponível, implemente uma versão paralela usando comunicação não-bloqueante
   a. Use arranjos de reais como buffers de envio e recebimento.
2. Faça uma análise de desempenho, medindo speedup e eficiência da sua implementação.

Exercício 2

- Implemente uma versão, usando dados derivados para troca de dados



Considerações Finais

- O paralelismo é crítico hoje, já que é a única maneira de obter melhoria de desempenho em um hardware moderno
- MPI é um modelo padrão da indústria para programação paralela
  - Existe um grande número de implementações MPI (comerciais e de domínio público)
  - Praticamente todos os sistemas do mundo suportam MPI
- Dá ao usuário controle explícito sobre o gerenciamento de dados
- Amplamente utilizado por muitas aplicações científicas com grande sucesso
- **Sua aplicação pode ser a próxima!**

Considerações Finais

- Antes de saltar de cabeça no MPI, pare e considere:
  - Você está implementando algo que outros já implementaram e já encontram-se disponíveis em bibliotecas?
  - Eu devo reutilizar código ou desenvolver um código do zero?
  - Mapear o ciclo de desenvolvimento e requisitos, certifique-se que o MPI é a resposta certa para você.
- Se o MPI for a resposta
  - Pesquise ferramentas existentes, bibliotecas e funções. Reutilização de código poupa tempo, esforço e frustração.
  - Mapeie os requisitos da aplicação e verifique se há algoritmos paralelos para eles. Evite reinventar a roda.
  - Inclui declarações de depuração "inteligentes" no seu código, dê preferência à alguma forma de rastreamento capaz de acompanhar onde os problemas ocorrem.