PARALELISMO POR TROCA DE MENSAGENS

Message Passing Interface

JOSE J. CAMATA
ALVARO L. G. A. COUTINHO

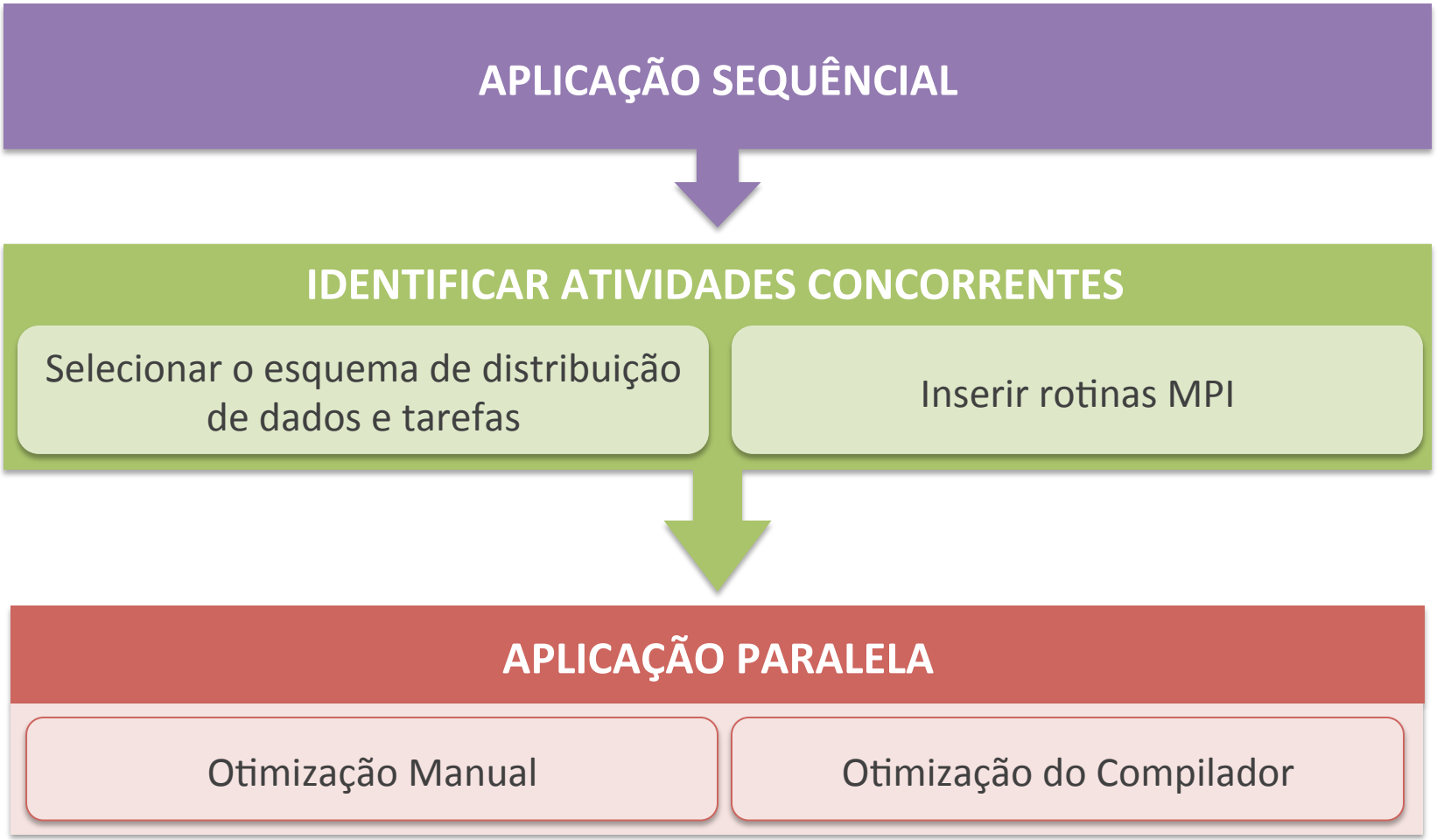
{camata, alvaro}@nacad.ufrj.br

Núcleo de Atendimento em Computação de Alto Desempenho
e Departamento de Engenharia Civil
Universidade Federal do Rio de Janeiro, Brasil
www.nacad.ufrj.br

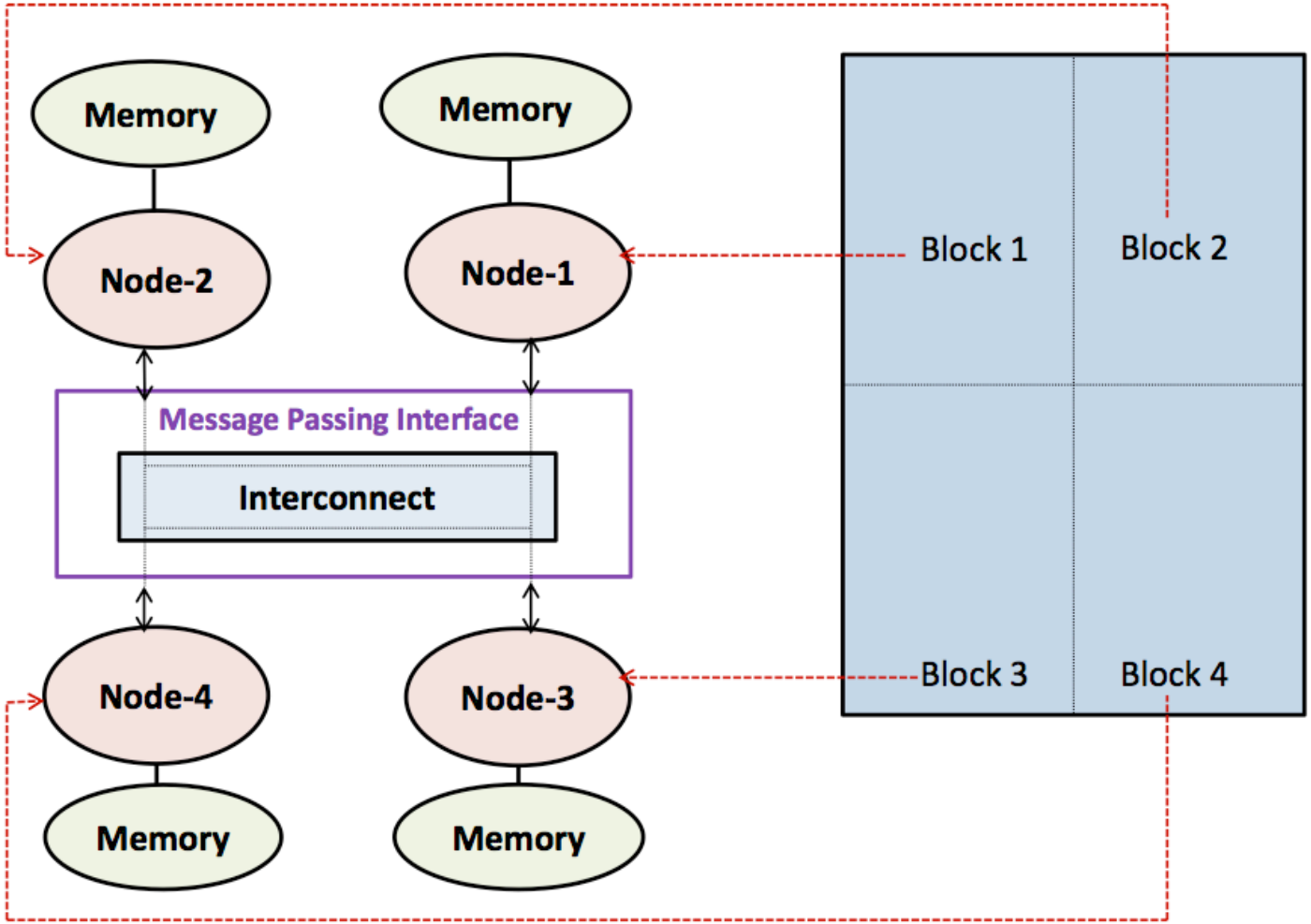
Aspectos Gerais

- 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

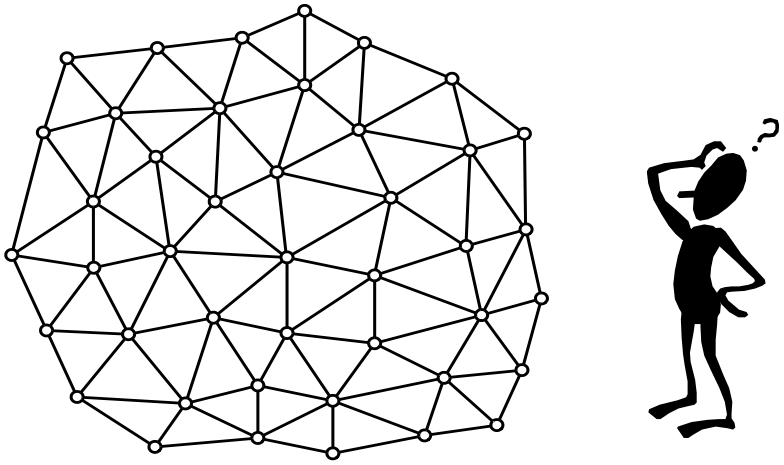


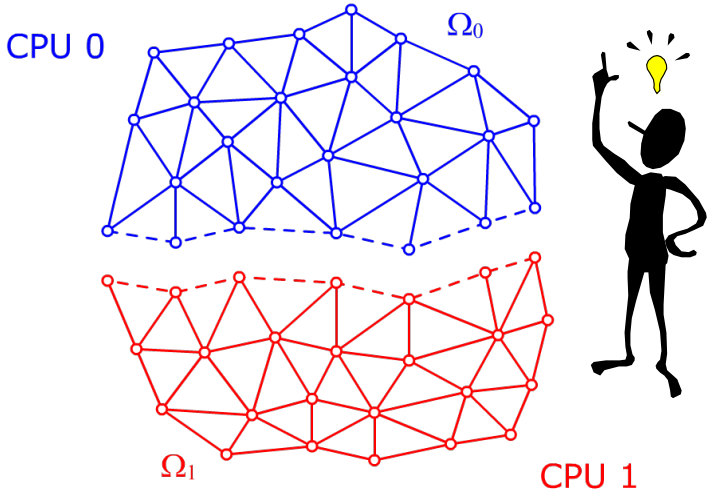
Dividir & Conquistar



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.

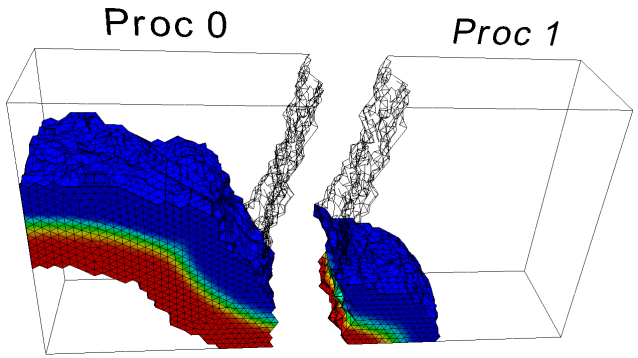




=> Técnicas de particionamento dos dados

Peculiaridades do MPI

- Enquanto alguns algoritmos podem ser diretamente implementados no modelo de paralelismo proposto pelo MPI, outros são particularmente desafiantes, p. ex.:
  - métodos baseados em avanço de fronteira ou algoritmos no qual o método segue a fronteira de solução são bons exemplos no qual o paralelismo com o MPI requer cuidados especiais.
  - Avalie o seguinte exemplo:



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

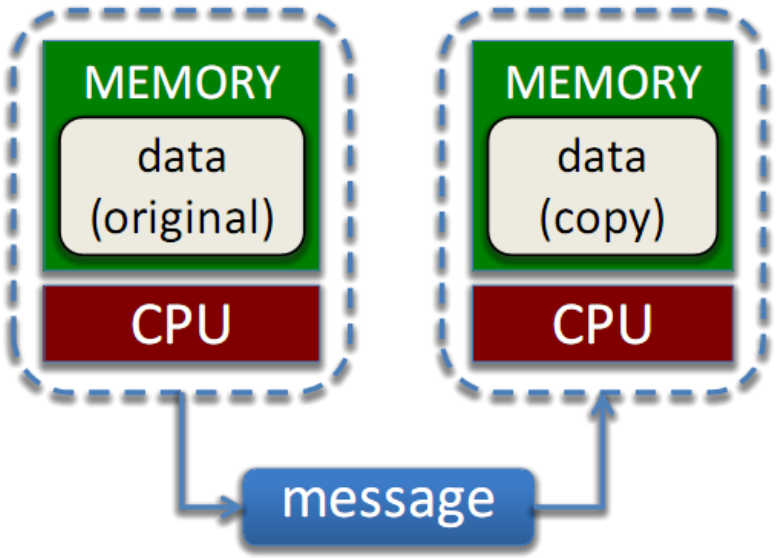
Trocas de Mensagens

- 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 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 rank 0 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..."*

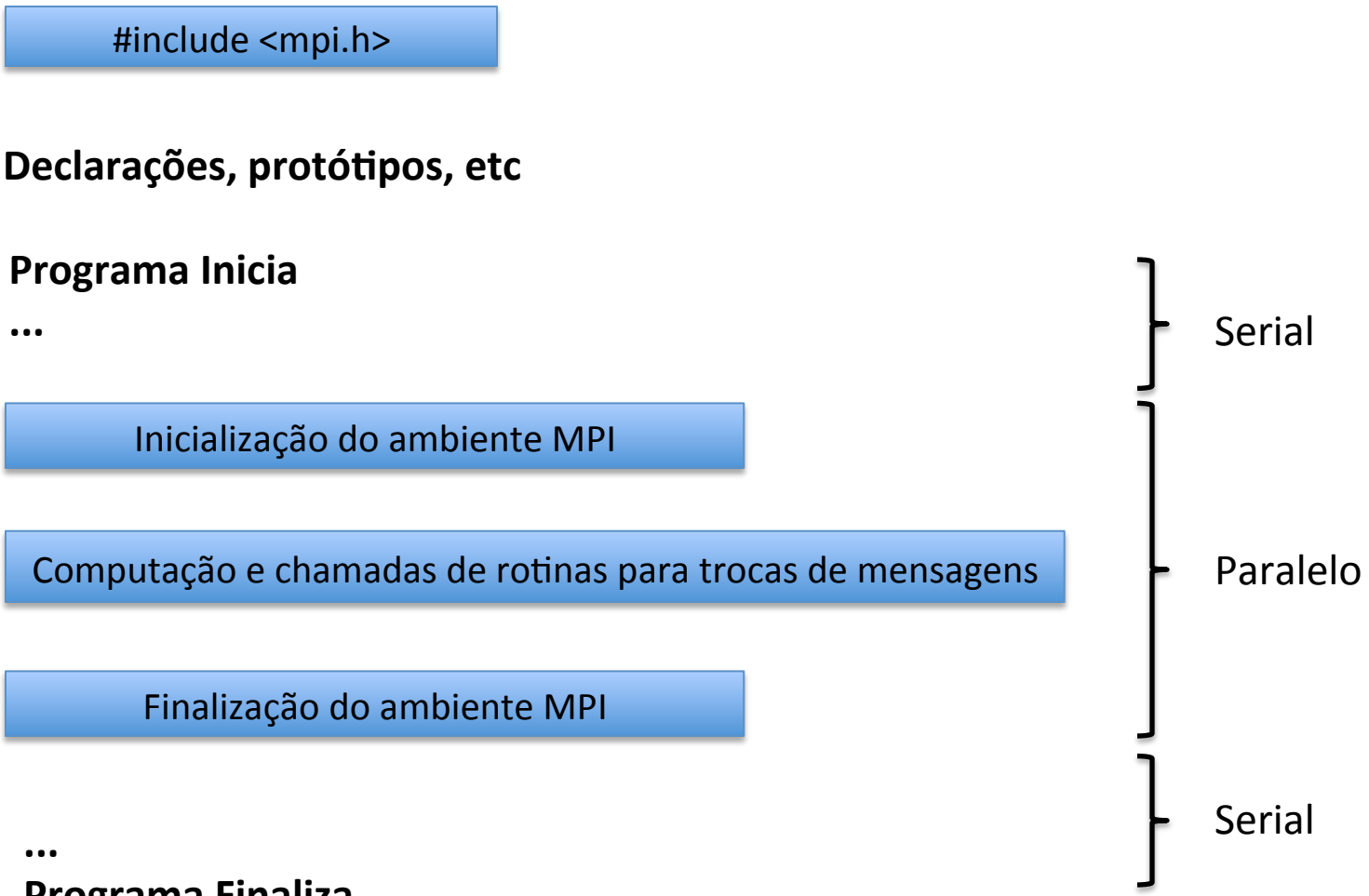
Por que aprender MPI?

- 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/)

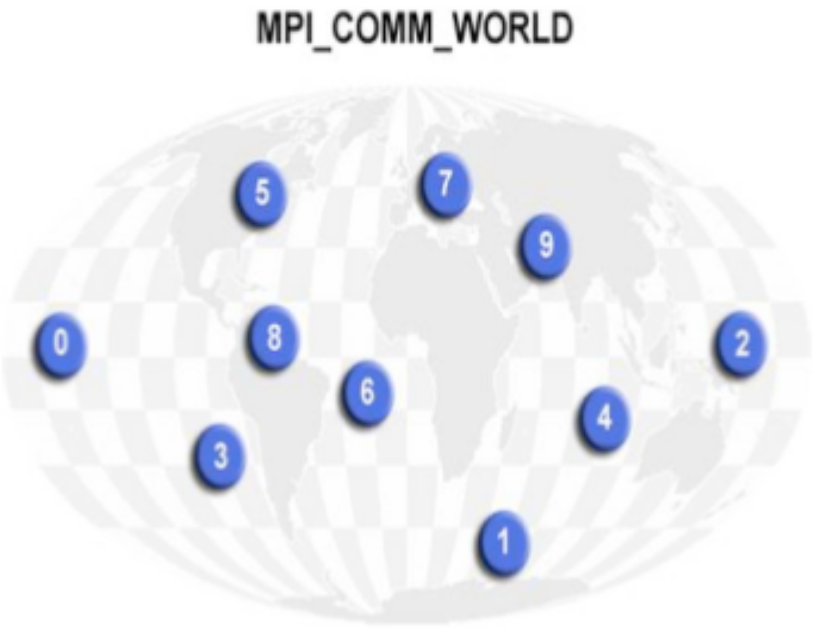
Estrutura de um programa MPI



Um simples programa MPI

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

int main( argc, argv )
int argc;
char **argv;
{
MPI_Init( &argc, &argv );
printf( "Hello world\n" );
MPI_Finalize();
return 0;
}
```

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!)

- DICA DE PROGRAMADOR: Utilize macros de pré-processamento para produzir versões 100% seriais de seu programa.

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)

C	Fortran
`#include "mpi.h"`	`include 'mpif.h'`

`MPI_Xxx(. . .);`	`CALL MPI_XXX(. . ., ierr)`
	
	`Call mpi_xxx(. . ., ierr)`

`MPI_Init(NULL, NULL)`
 	
`MPI_Init(&argc, &argv)`
	`MPI_INIT(ierr)`

`MPI_Finalize()`	`MPI_FINALIZE(ierr)`

Rotinas de Gerenciamento de Ambiente (1)

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

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

Rotinas de Gerenciamento de Ambiente (2)

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

- **`MPI_Wtime`** é uma rotina utilizada na medição de desepenho. Retorna o clock time em segundos.
  - **`MPI_Wtime()`**

Exemplo (1)

- Programa seqüencial

```
#include <stdio.h>

int main(int argc, char* argv)
{

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

}
```

```
Compilando:
$icc –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
```

Exemplo (3)

- Usando um comunicador

```
#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

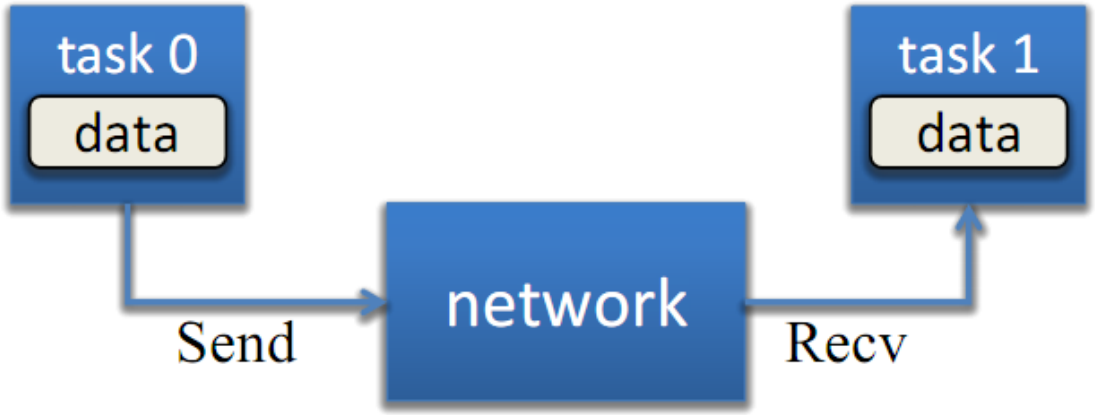
Comunicação Ponto-a-Ponto

- 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
  - Envio “ready”
  - Envio buferizado

- Qualquer tipo de rotina de envio pode ser combinado com qualquer tipo de rotina recebimento.

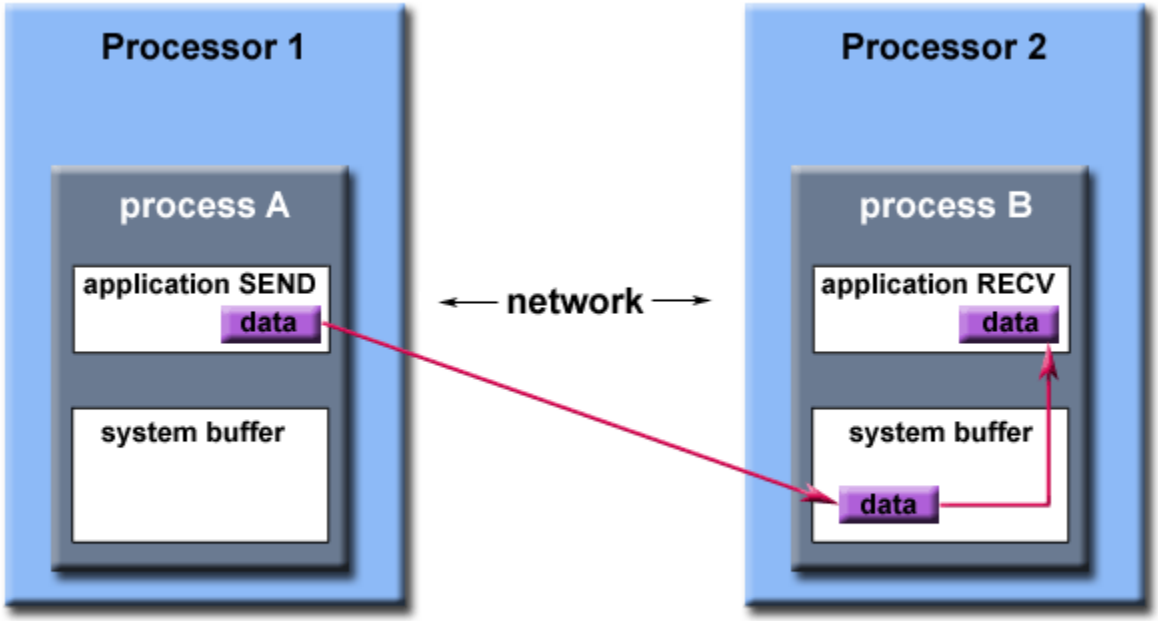
- MPI também fornece várias rotinas associadas as operações de envio/ recebimento, tais como aqueles usados para esperar o recebimento de uma mensagem (wait) ou sondagem para saber se a mensagem chegou (probe)

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 "retorna" após os dados chegarem e estarem prontos para uso pelo programa.

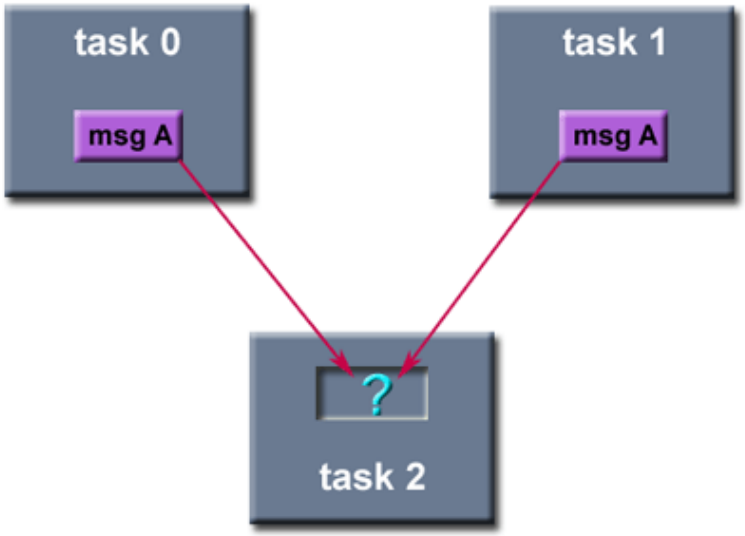
P2P: Bloqueante vs. Não Bloqueante

- 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 as mensagens não irá ultrapassar outras.
  - Se um remetente envia duas mensagens (Messagem 1 e Mensagem 2), em sucessão para o mesmo destino, e ambos se encaixam na mesma operação de recebimento, a operação de 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.



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

Envio Bloqueante: MPI_Send

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

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

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

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 do que esta.

- 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

Exercício #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);
    }
    MPI_Finalize();
    return 0;
}
```

- Enviando um vetor de processo 0 para 1

Rotinas bloqueantes/não-bloqueantes

Descrição	Sintaxe (c)
Envio bloqueante	MPI_Send(buf, count, datatype, dest, tag, comm)
Recebimento bloq.	MPI_Recv(buf, count, datatype, source, tag, comm, status)
Envio não-bloqueante	MPI_Isend(buf, count, datatype, dest, tag, comm, request)
Recebimento não-bloq.	MPI_Irecv(buf, count, datatype, source, tag, comm, request)

- Objeto MPI_Request são usados pelas rotinas não bloqueantes

- MPI_Wait()/MPI_Waitall()
  - Funções bloqueantes
  - Pausa a execução do programa até que as operações iniciadas pelo Isend/ Irecv tenham sido completadas

Exemplos: Envio/Recebimento

- Envio e recebimento bloqueante

```
IF (rank==0) THEN
  CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
ENDIF
```

- Envio não-bloqueante e recebimento bloqueante

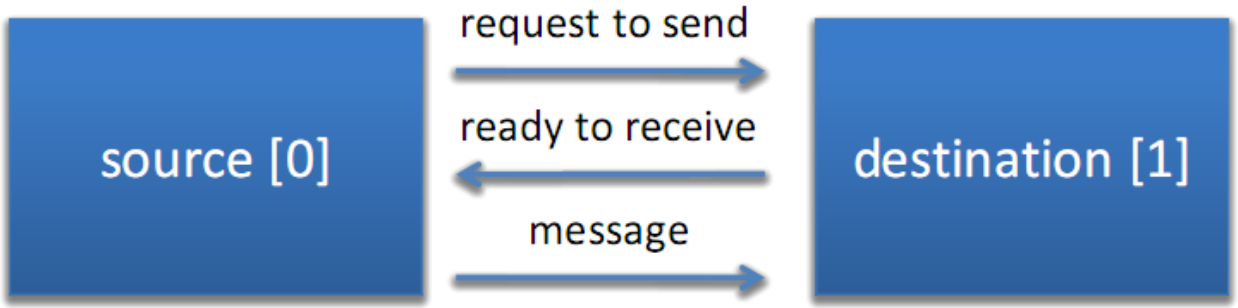
```
IF (rank==0) THEN
  CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req,ierr)
ELSEIF (rank==1) THEN
  CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
ENDIF
CALL MPI_WAIT(req, wait_status)
```

Comunicação P2P: MPI_SendRecv

```
MPI_SendRecv(/*send arguments*/
        sendbuf, sendcount, sendtype,
        dest, sendtag,
        /*receive arguments*/
        recvbuf, recvcount, recvtype, source,
        recvtag, comm, status);
```

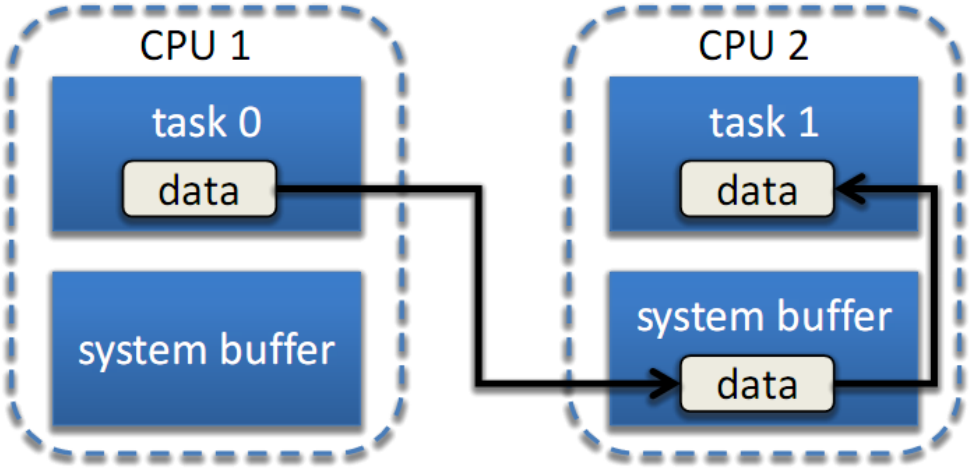
- União dos comandos MPI_Send e MPI_Recv
- Executa uma operação de envio e recebimento bloqueante
- Estágios de envio e recebimento usam o mesmo comunicador, mas tem tags distintos
- Útil para padrões de comunicação onde cada nó envia e recebe mensagens

Comunicação Síncrona



- **MPI_Ssend & MPI_Srecv**
  - Processo 0 espera até o processo 1 ficar pronto
  - *Handshaking* ocorre entre as tarefas de envio e recebimento para confirmar um envio seguro.
  - Envio/recebimento bloqueante
  - Precisa ser capaz de lidar com o armazenamento de dados quando várias tarefas estão fora de sincronia.

Comunicação Buferizada



- **MPI_Bsend & MPI_Brecv**
  - O conteúdo de uma mensagem é copiada para dentro de um bloco de memória do sistema
  - Processo 0 continua executando outras tarefas; quando o processo 1 estiver pronto para receber, o sistema simplesmente copia a mensagem para a localização de memória controlada pelo processo 1

Exemplo de Deadlock

- O seguinte código tem um deadlock

```
IF (rank==0) THEN
     CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
     CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
ELSEIF (rank==1) THEN
     CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
     CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
ENDIF
```

- Solução:

```
IF (rank==0) THEN
     CALL MPI_SEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,ierr)
     CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,status,ierr)
ELSEIF (rank==1) THEN
     CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,status,ierr)
     CALL MPI_SEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,ierr)
ENDIF
```

Soluções alternativas

- Solução usando sendrecv

```
IF (rank==0) THEN
     CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 1, sendtag,
                       recvbuf, count, MPI_REAL, 1, recvtag,
                       MPI_COMM_WORLD, status, ierr)
ELSEIF (rank==1) THEN
     CALL MPI_SENDRECV(sendbuf, count, MPI_REAL, 0, sendtag,
                       recvbuf, count, MPI_REAL, 0, recvtag,
                       MPI_COMM_WORLD, status, ierr)
ENDIF
```

- Outra solução possível (usando chamadas não-bloqueantes)

```
IF (rank==0) THEN
     CALL MPI_ISEND(sendbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req1,ierr)
     CALL MPI_IRECV(recvbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req2,ierr)
ELSEIF (rank==1) THEN
     CALL MPI_ISEND(sendbuf,count,MPI_REAL,0,tag,MPI_COMM_WORLD,req1,ierr)
     CALL MPI_IRECV(recvbuf,count,MPI_REAL,1,tag,MPI_COMM_WORLD,req2,ierr)
ENDIF
CALL MPI_WAIT(req1, wait_status, ierr)
CALL MPI_WAIT(req2, wait_status, ierr)
```

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

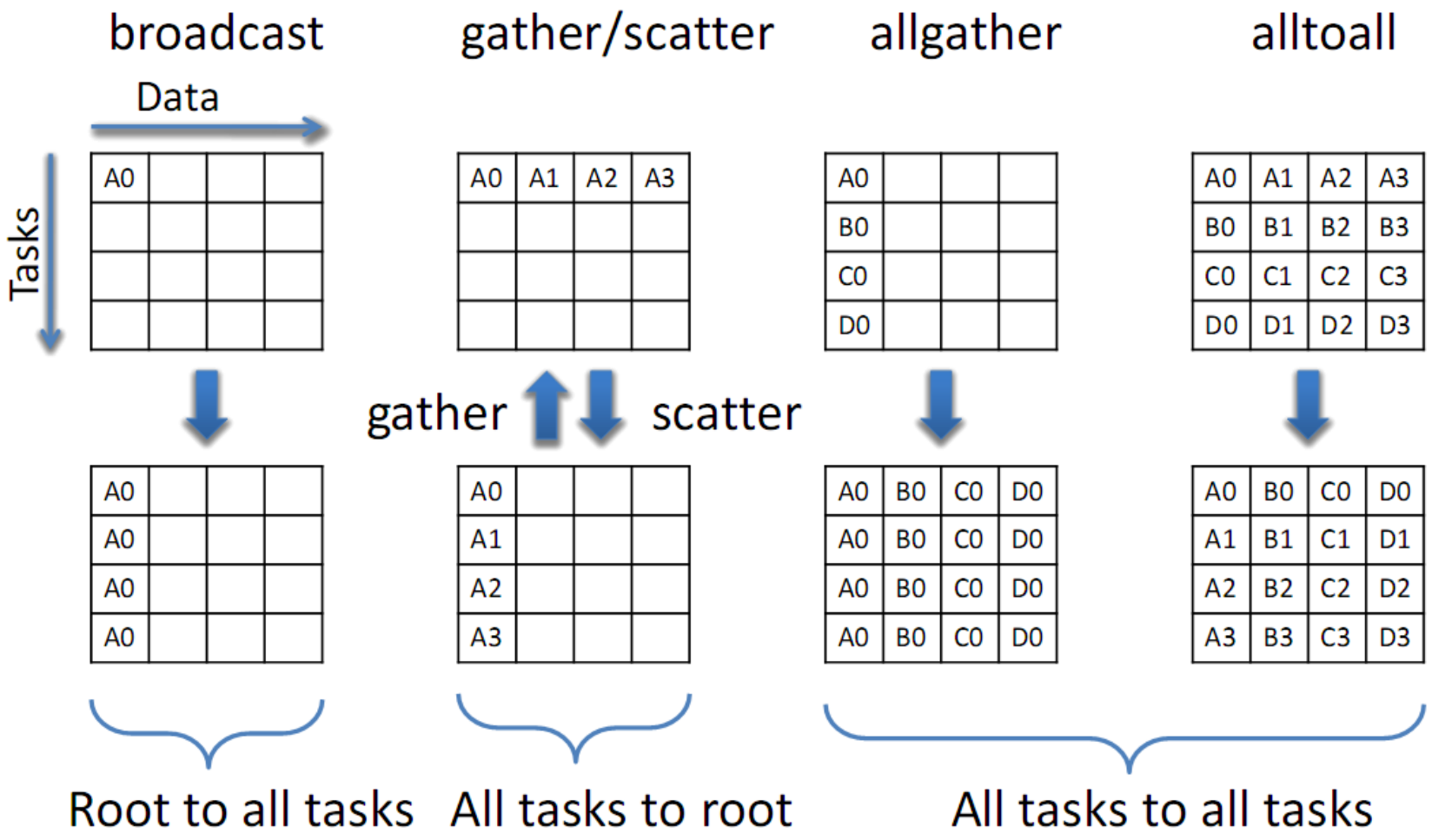
- Envolve todos os processos dentro de um comunicador
- É de responsabilidade do programador assegurar que todos os processos chamarão a mesma comunicação coletiva ao mesmo tempo
- Tipo de operações coletivas
  - Sincronização
  - Movimento de Dados
  - Redução
- Considerações de programação e restrições
  - 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

Sincronizando 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

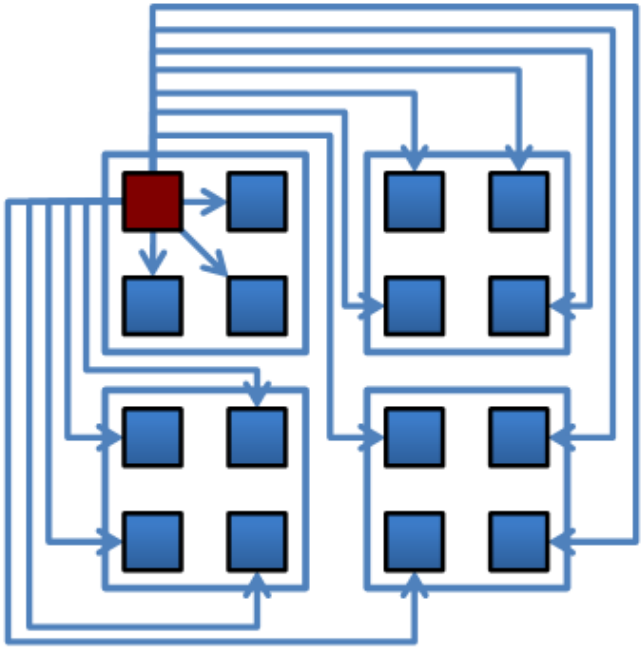
- **Um processo deseja enviar uma mensagem para todos os outros processos**
  - Implementação ingênua:

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

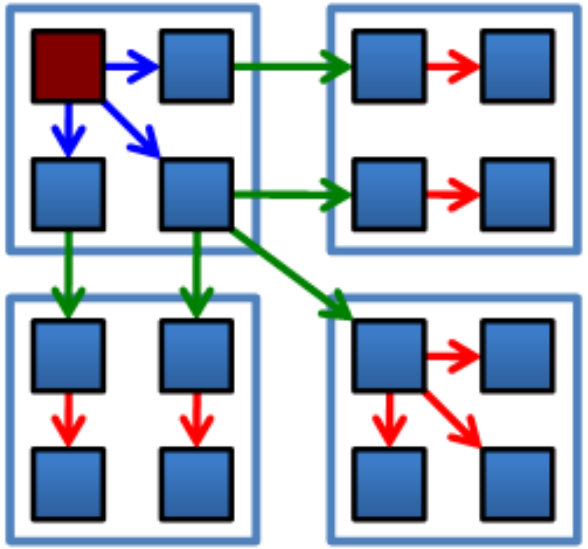
  - Este código envia informação do processo 0 para todos os outros processos, mas
    - Muito primitivo – nenhuma otimização para o hardware
    - Usa comunicação bloqueante

Implementação do Broadcast

Implementação ingênua



Implementação MPI



MPI Broadcast

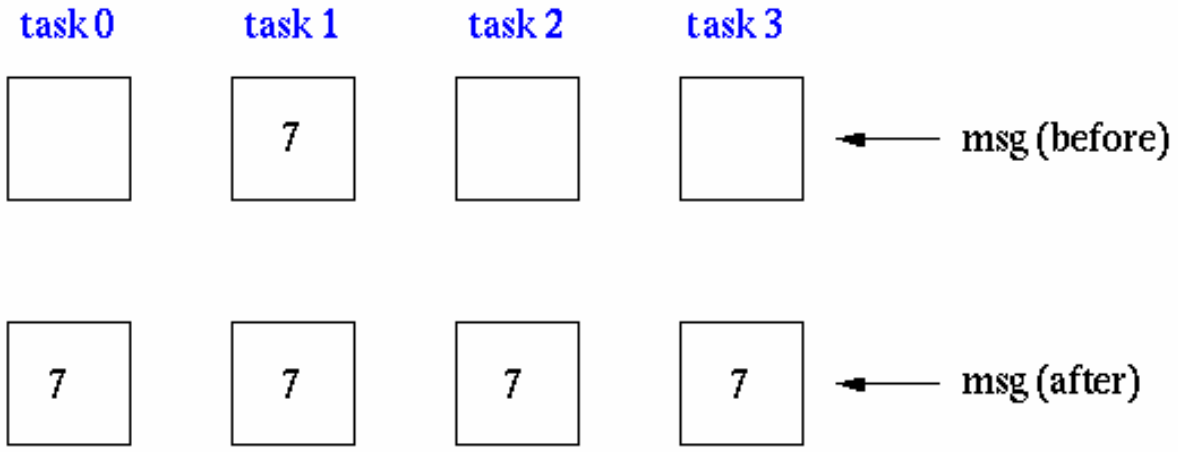
MPI_BCAST(buf,count,datatype,root,comm)		
IN/OUT	buf	Inicio do endereço de memória do buffer
IN	count	Número de elementos no buffer
IN	datatype	Tipo de dados contidos no buffer
IN	root	Identificador do processo que enviará a mensagem
IN	comm	comunicador

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

MPI_Bcast

Broadcasts a message to all other processes of that group

```
count = 1;
source = 1;                    broadcast originates in task 1
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```



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

MPI_Scatter

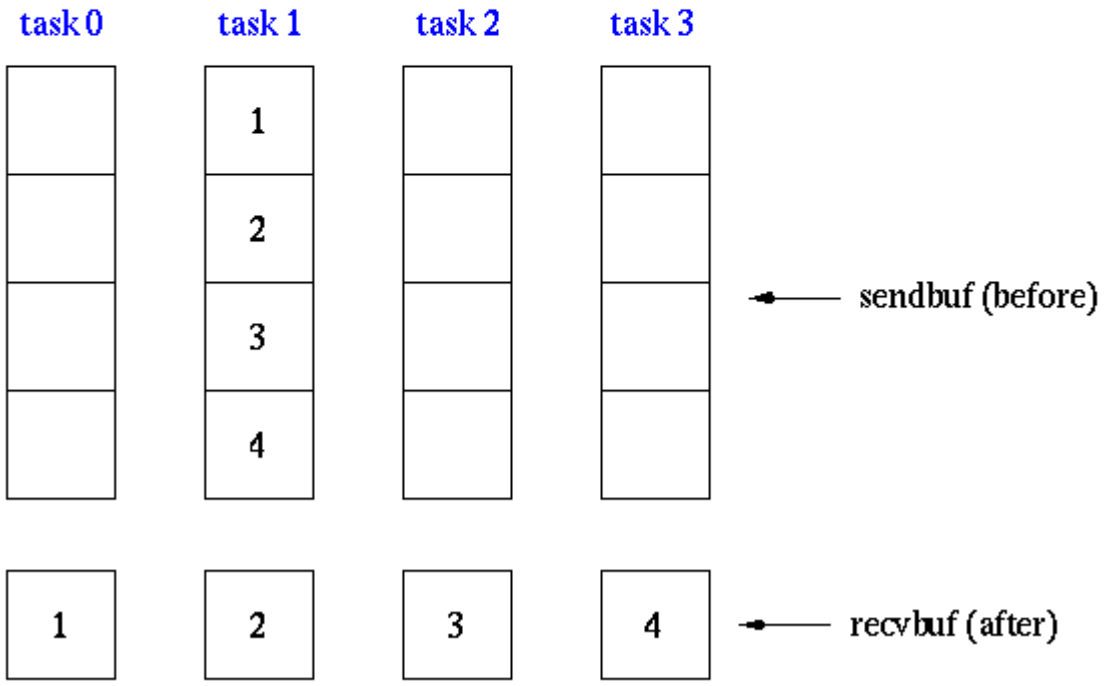
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN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

- 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

Sends data from one task to all other tasks in a group

```
sendcnt = 1;
recvcnt = 1;
src = 1;                task 1 contains the message to be scattered
MPI_Scatter(sendbuf, sendcnt, MPI_INT,
            recvbuf, recvcnt, MPI_INT,
            src, MPI_COMM_WORLD);
```



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

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

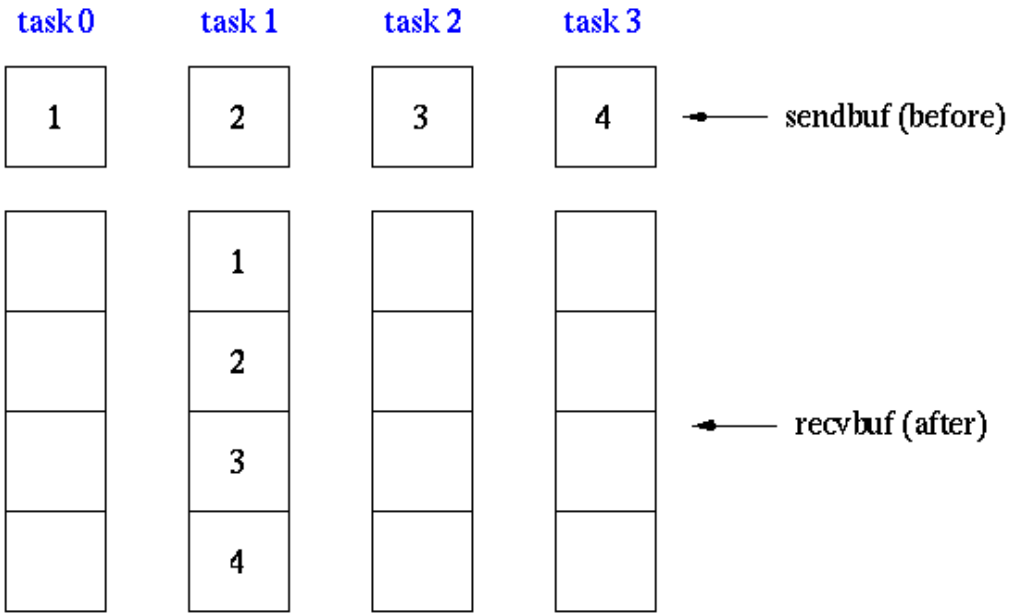
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN	recvcount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

- 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

Gathers together values from a group of processes

```
sendcnt = 1;
recvcnt = 1;
src = 1;                messages will be gathered in task 1
MPI_Gather(sendbuf, sendcnt, MPI_INT,
           recvbuf, recvcnt, MPI_INT,
           src, MPI_COMM_WORLD);
```



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

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

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

        /* Inicializa o buffer de envio*/
        for(i=0;i<2;i++)
            sendbuffer[i] = i*2;
        /*  Somente o rank 0 aloca o buffer de recebimento */
        if (myrank==0){
                recvbufflen = 2*size;
                recvbuffer = (int*)malloc(recvbufflen * sizeof(int));
        }

        /* Operação MPI_Gather  */

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

MPI_Allgather

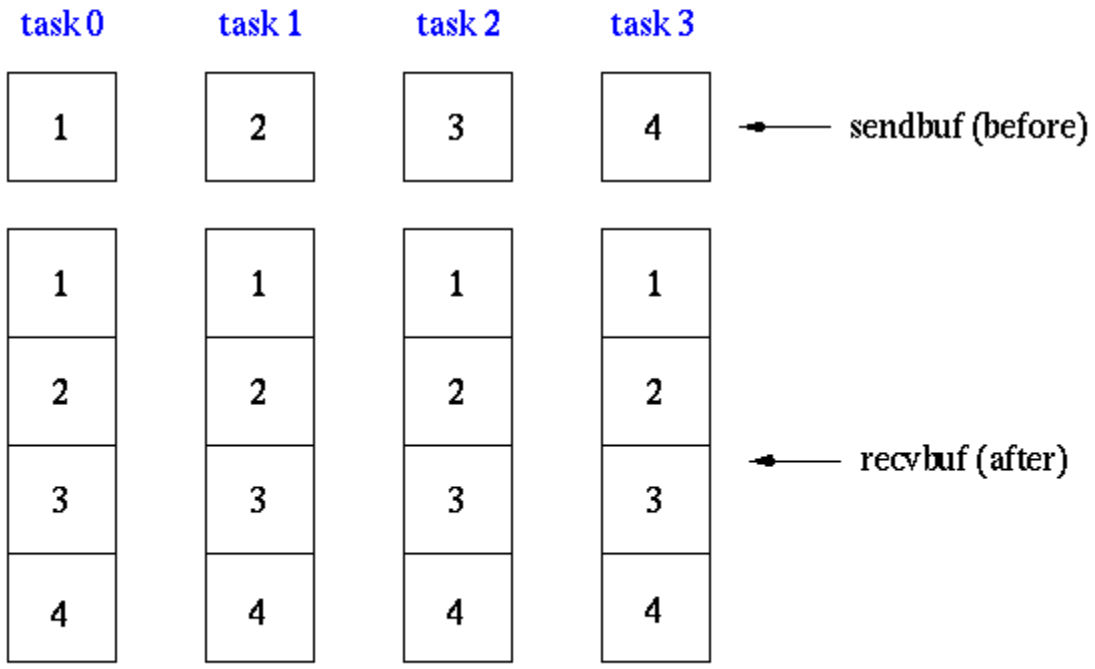
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN	recvcount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

- 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

Gathers together values from a group of processes and distributes to all

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



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

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

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

        /* Inicializa o buffer de envio*/
        for(i=0;i<2;i++)
            sendbuffer[i] = i*2;

        // Todos os processos alocam o buffer de recebimento */

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

        /* Operação MPI_Gather  */

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

MPI_Alltoall

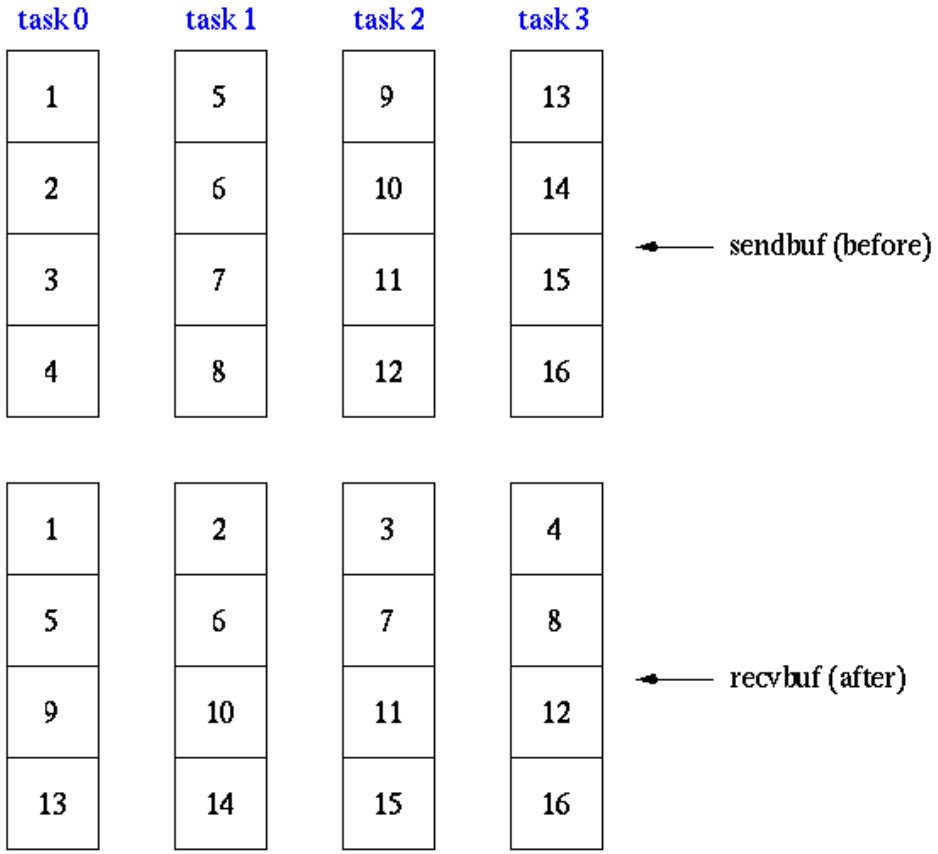
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
IN	sendcount	número de elementos enviados para cada processo
IN	sendtype	tipo dos dado enviado
out	recvbuf	endereço de memória do buffer de recebimento
IN	recvcount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

- 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

Sends data from all to all processes. Each process performs a scatter operation.

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



Operação de Redução

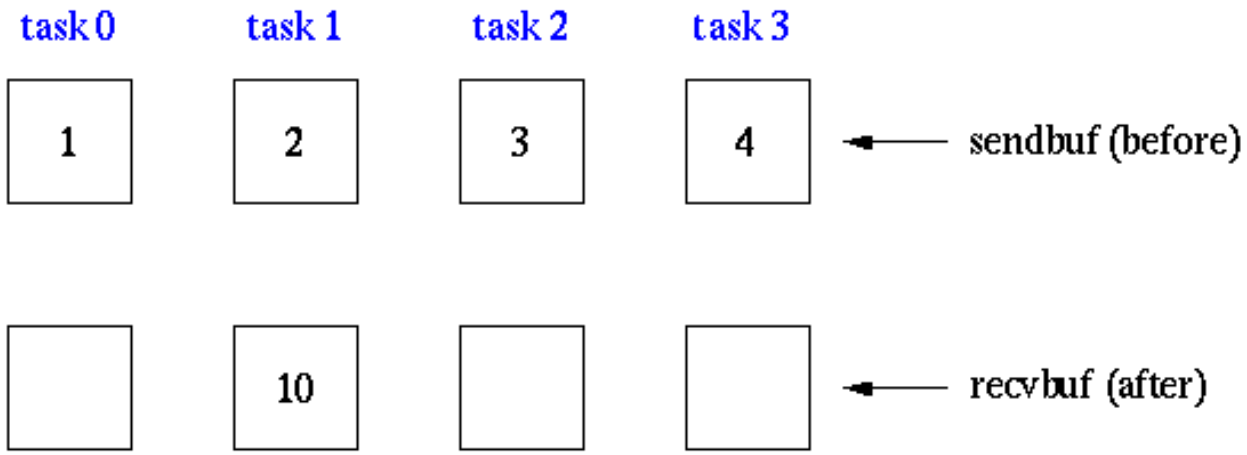
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
OUT	recvbuf	endereço de memória do buffer de recebimento no processo root
IN	count	número de elementos enviados
IN	datatype	tipo dos dados do buffer de envio
IN	op	operação de redução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

- 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

Perform and associate reduction operation across all tasks in the group and place the result in one task

```
count = 1;
dest = 1;                    result will be placed in task 1
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
           dest, MPI_COMM_WORLD);
```



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

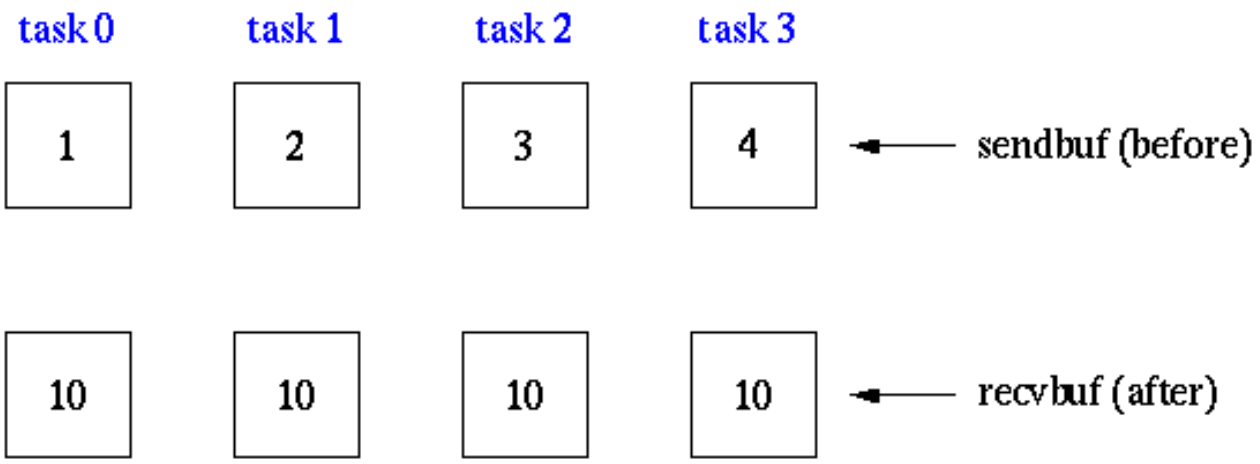
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm)		
IN	sendbuf	inicio do endereço de memória do buffer de envio
OUT	recvbuf	endereço de memória do buffer de recebimento
IN	count	número de elementos enviados
IN	datatype	tipo dos dados do buffer de envio
IN	op	operação de redução
IN	comm	comunicador

- 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

Perform and associate reduction operation across all tasks in the group and place the result in all tasks

```
count = 1;
MPI_Allreduce(sendbuf, recvbuf, count, 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

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.

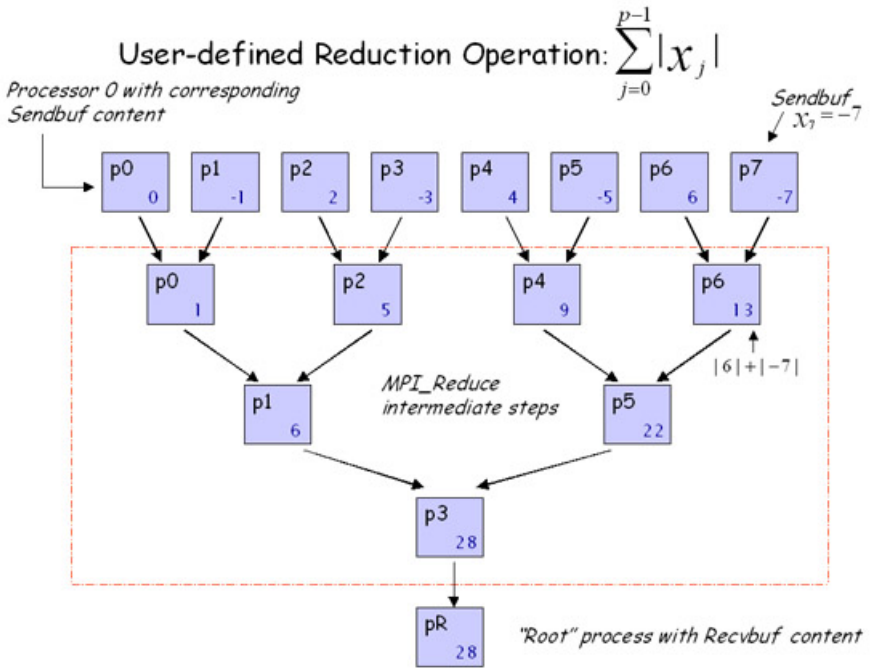
A função do usuário dever ser definida como:

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

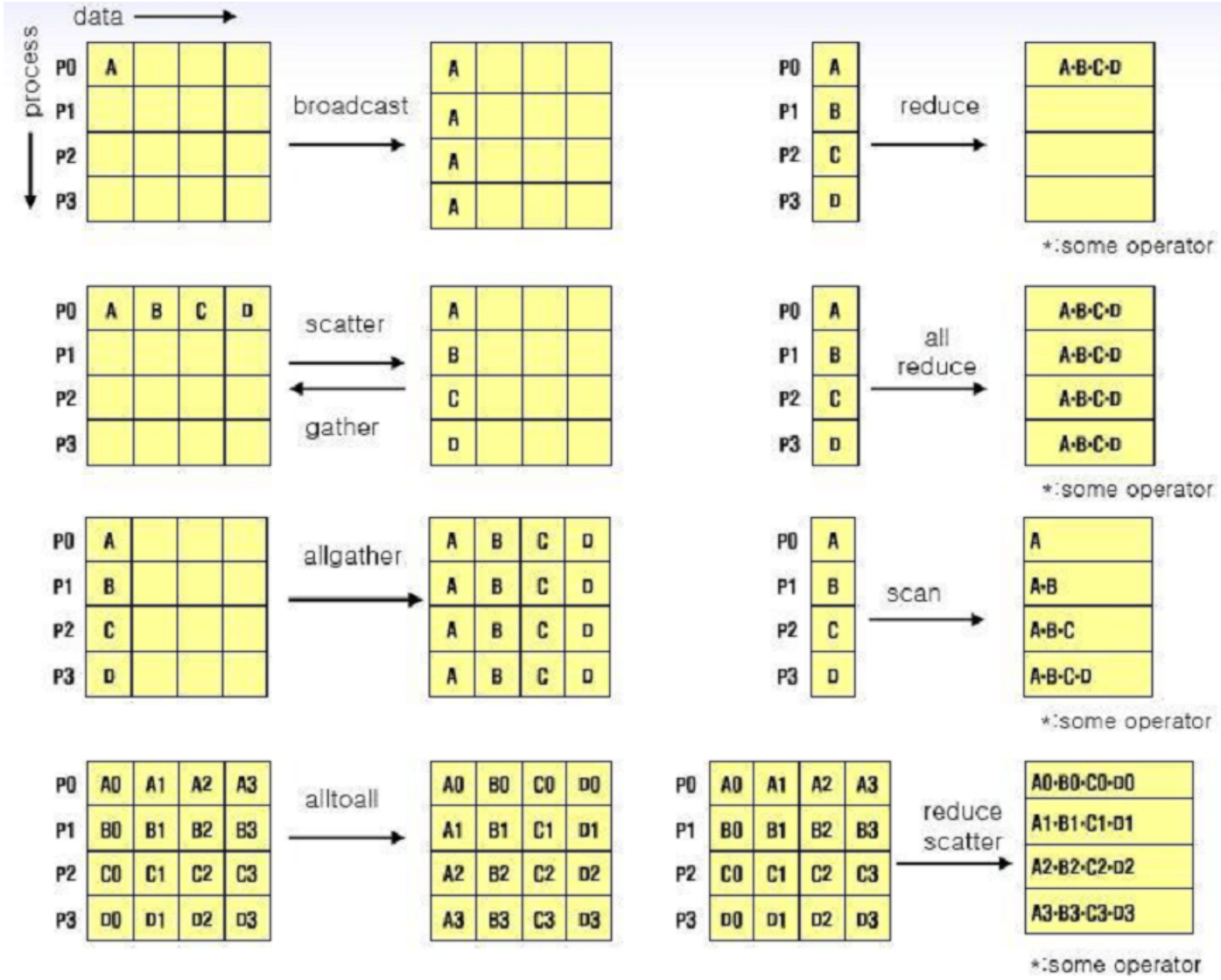
Exemplo: Calculo da norma 1:

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

```
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, recvbuf;
      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

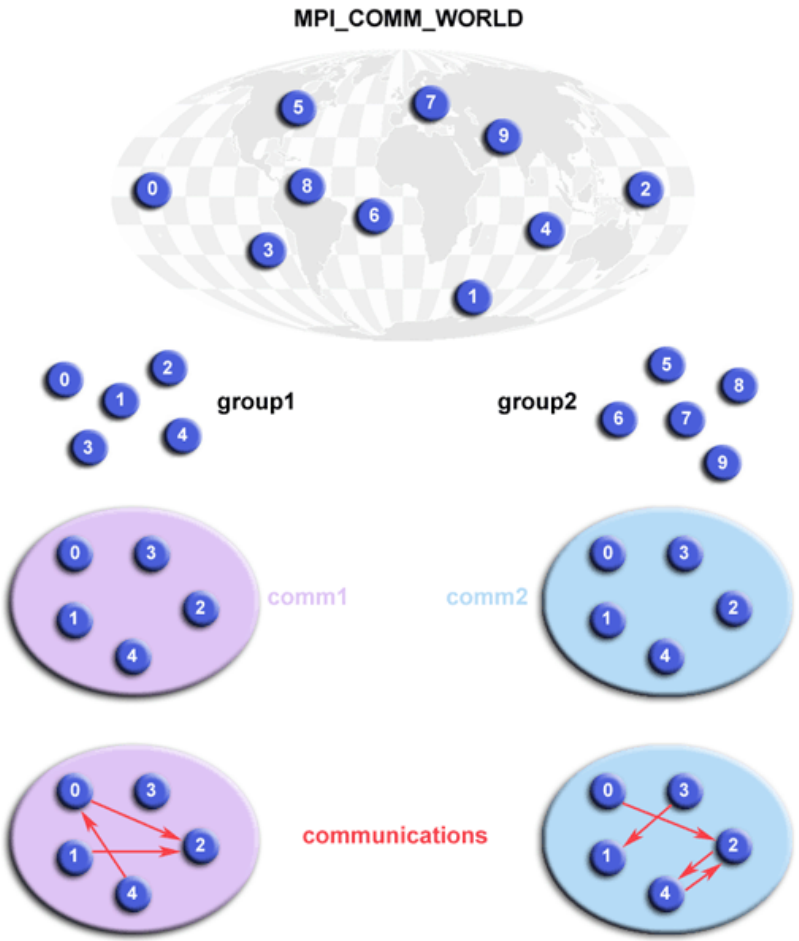
- Um **grupo** é um **conjunto ordenado de processos**. Cada processo em um grupo está identificado com um inteiro único. Os valores de classificação começa em zero e vão até N-1, onde N é o número de processos no grupo. Um grupo está sempre associada a um objeto comunicador.

- **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.

```c
#include "mpi.h“
 #include <stdio.h>
#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();
}
```

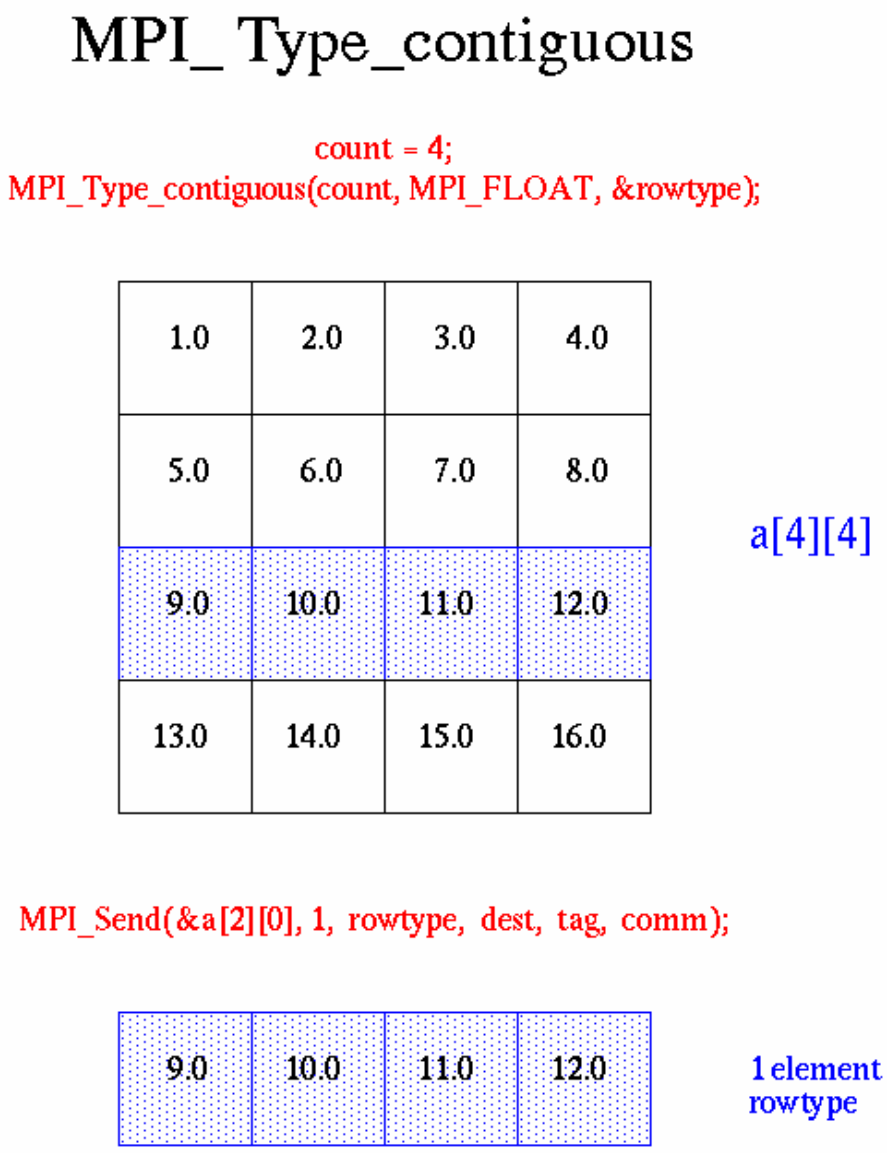
Tipos de Dados Derivados

- 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)

```c
#include "mpi.h"
#include <stdio.h>
#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();
}
```

MPI_Type_contiguous



Tipos de Dados Derivados

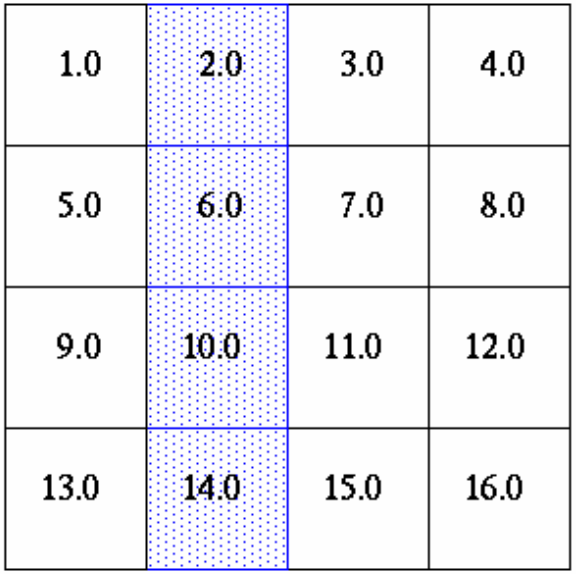
- MPI_Type_vector:
  - Semelhante ao contiguous, mas permite espaçamento regulares entre os elementos

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

```c
#include "mpi.h"
#include <stdio.h>
#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 coltype; // 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, MPI_FLOAT, &coltype);
  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[0][i], 1, coltype, 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();
}
```

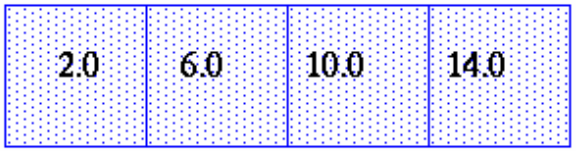
MPI_ Type_vector

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



a[4][4]

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



1 element of columntype

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)

Considerações Finais

- Antes de saltar de cabeça no MPI, pare e considere:
  - Você está implementando algo que outros já implementaram e encontra-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.