

**Curso de Informática – DCC-IM / UFRJ**

---

# **MPI**

---

Um curso prático

**Gabriel P. Silva**

---

# MPI

- ◆ É um padrão de troca de mensagens portátil que facilita o desenvolvimento de aplicações paralelas.
- ◆ Usa o paradigma de programação paralela por troca de mensagens e pode ser usado em clusters ou em redes de estações de trabalho.
- ◆ É uma biblioteca de funções utilizável com programas escritos em C, C++ ou Fortran.
- ◆ A biblioteca MPI, no entanto, só possui funções para tratamento de mensagens, não oferecendo suporte para criação ou eliminação de processos como o PVM.

# MPI

- ◆ **O projeto do MPI procurou utilizar as melhores facilidades de um grande número de sistemas de troca de mensagem existentes, ao invés de selecionar um deles e utiliza-lo como padrão.**
- ◆ **Logo, o MPI foi fortemente influenciado pelo trabalho no IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex e PARMACS.**
- ◆ **Outras contribuições importantes vieram do Zipcode, Chimp, PVM, Chameleon e PICTL.**

# MPI

- ◆ O esforço de padronização do MPI envolveu cerca de 60 pessoas de diversas organizações dos Estados Unidos e Europa.
- ◆ A maioria dos fabricantes de computadores paralelos estiveram envolvidos no MPI, junto com pesquisadores das universidades, laboratórios do governo e empresas.
- ◆ O processo de padronização começou com o “Workshop on Standards for Message Passing in a Distributed Memory Environment”, patrocinado pelo “Center for Research on Parallel Computing”, que foi realizado em abril de 1992.

# Objetivos do MPI

- ◆ **Um dos objetivos do MPI é oferecer possibilidade de uma implementação eficiente da comunicação:**
  - Evitando cópias de memória para memória;
  - Permitindo superposição de comunicação e computação.
- ◆ **Permitir implementações em ambientes heterogêneos.**
- ◆ **Supõe que a interface de comunicação é confiável:**
  - Falhas de comunicação devem ser tratadas pelo subsistema de comunicação da plataforma.

# Objetivos do MPI

- ◆ Pode ser implementado em plataformas de diferentes fornecedores.
- ◆ Interface familiar para os usuários de PVM.
- ◆ Permitir o uso de programas escritos em C e Fortran.
- ◆ O MPI, versão 1.0, não possui uma função semelhante ao *pvm\_spawn* para criação de "tasks" ou processos em outros nós de processamento da máquina virtual. Tal operação só foi implementada na versão 2.0.

# Objetivos do MPI

- ◆ A biblioteca MPI trabalha com o conceito de **comunicadores** para definir o universo de processos envolvidos em uma operação de comunicação, através dos atributos de grupo e contexto:
  - Dois processos que pertencem a um mesmo **grupo** e usando um mesmo **contexto** podem se comunicar diretamente.

# Funções Básicas

- ◆ Todo programa em MPI deve conter a seguinte diretiva para o pré-processador:  
`#include "mpi.h"`
- ◆ Este arquivo, **mpi.h**, contém as definições, macros e funções de protótipos de funções necessários para a compilação de um programa MPI.
- ◆ Antes de qualquer outra função MPI ser chamada, a função **MPI\_Init** deve ser chamada pelo menos uma vez.
- ◆ Seus argumentos são os ponteiros para os parâmetros do programa principal, **argc** e **argv**.

# Funções Básicas

- ◆ Esta função permite que o sistema realize as operações de preparação necessárias para que a biblioteca MPI seja utilizada.
- ◆ Ao término do programa a função **MPI\_Finalize** deve ser chamada.
- ◆ Esta função limpa qualquer pendência deixada pelo MPI, p. ex, recepções pendentes que nunca foram completadas.
- ◆ Tipicamente, um programa em MPI pode ter o seguinte leiaute:

# Funções Básicas

```
...
#include "mpi.h"
...
main(int argc, char** argv) {
...
/* Nenhuma função MPI pode ser chamada antes deste
   ponto */
MPI_Init(&argc, &argv);
...
MPI_Finalize();
/* Nenhuma função MPI pode ser chamada depois deste
   ponto*/
...
/* main */
...

```

# Programa Simples em MPI

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

```
main(int argc, char** argv)
{
```

```
    int    meu_rank, np, origem, destino, tag=0;
    char msg[100];
    MPI_Status status;
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &np);
```

# Programa Simples em MPI

```
if (meu_rank != 0) {
    sprintf(msg, "Processo %d está vivo!", meu_rank);
    destino = 0;
    MPI_Send(msg, strlen(msg)
+1, MPI_CHAR, destino, tag, MPI_COMM_WORLD);
}
Else {
    for (origem=1; origem<np; origem++) {
        MPI_Recv(msg, 100, MPI_CHAR, origem, tag,
MPI_COMM_WORLD, &status);
        printf("%s\n", msg);
    }
}
MPI_Finalize( ); }
```

# Comunicadores

- ◆ **Comunicadores são utilizados para definir o universo de processos envolvidos em uma operação de comunicação através dos seguintes atributos:**
  - **Grupo:**
    - ◆ Conjunto ordenado de processos. A ordem de um processo no grupo é chamada de *rank*.
  - **Contexto:**
    - ◆ *tag* definido pelo sistema.
- ◆ **Dois processos pertencentes a um mesmo grupo e usando um mesmo contexto podem se comunicar.**

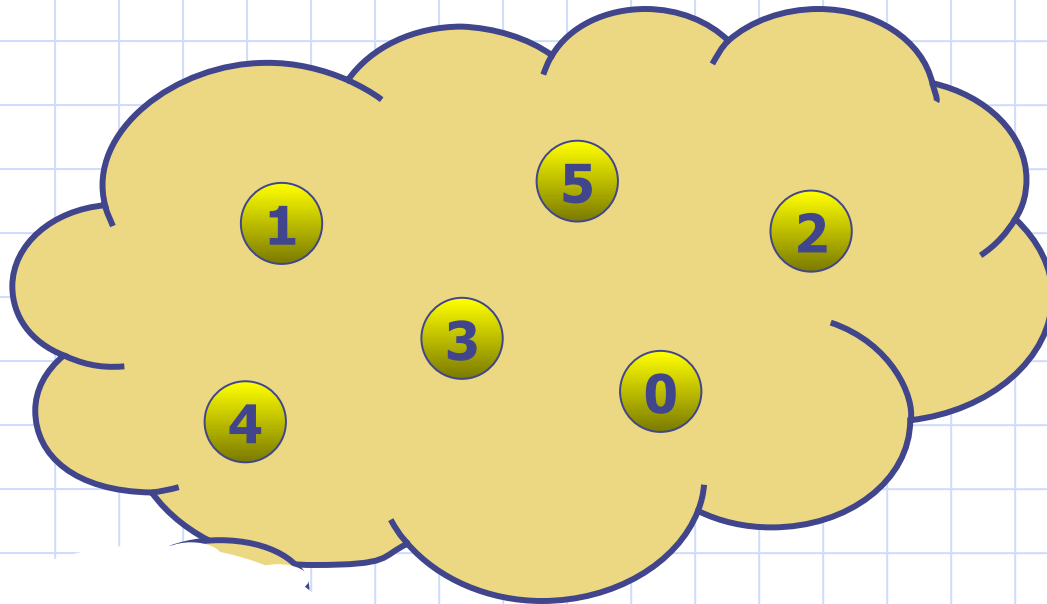
# Contactando outros Processos

- ◆ O MPI tem a função **MPI\_Comm\_Rank** que retorna o *rank* de um processo no seu segundo argumento.
- ◆ Sua sintaxe é:  

```
int MPI_Comm_Rank(MPI_Comm comm, int rank)
```
- ◆ O primeiro argumento é um **comunicador**. Essencialmente um **comunicador** é uma coleção de processos que podem enviar mensagens entre si.
- ◆ Para os programas básicos, o único comunicador necessário é **MPI\_COMM\_WORLD**, que é pré-definido no MPI e consiste de todos os processos executando quando a execução do programa começa.

# Comunicador MPI\_COMM\_WORLD

MPI\_COMM\_WORLD



# Contactando outros Processos

- ◆ Muitas construções em nossos programas também dependem do número de processos executando o programa.
- ◆ O MPI oferece a função **MPI\_Comm\_size** para determinar este valor.
- ◆ Essa função retorna o número de processos em um comunicador no seu segundo argumento.
- ◆ Sua sintaxe é:

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

# Contactando outros Processos

- ◆ Esses itens podem ser usados pelo receptor para distinguir entre as mensagens entrantes.
- ◆ O argumento *source* pode ser usado para distinguir mensagens recebidas de diferentes processos.
- ◆ O *tag* é especificado pelo usuário para distinguir mensagens de um único processo.
- ◆ O MPI garante que inteiros entre 0 e 32767 possam ser usados como *tags*. Muitas implementações permitem valores maiores.
- ◆ Um *comunicador* é basicamente uma coleção de processos que podem enviar mensagens uns para os outros.



# **Comunicação Ponto a Ponto**

# Mensagem MPI

- ◆ Mensagem = Dados + Envelope
- ◆ Para que a mensagem seja comunicada com sucesso, o sistema deve anexar alguma informação aos dados que o programa de aplicação deseja transmitir.
- ◆ Essa informação adicional forma o envelope da mensagem, que no MPI contém a seguinte informação:
  - O **rank** do processo origem.
  - O **rank** do processo destino.
  - Um **tag** (etiqueta especificando o tipo da mensagem).
  - Um **comunicador** (domínio de comunicação).

# Comunicação Ponto a Ponto

- ◆ **As funções de envio e recepção de mensagens, utilizadas na comunicação ponto a ponto, podem ser feitas em 4 modos:**
  - **Padrão (standard)**
    - ◆ O sistema decide se a mensagem vai ser “bufferizada”.
  - **Bufferizada (buffered)**
    - ◆ A aplicação deve prover explicitamente um “buffer” para a mensagem enviada.
  - **Síncrona (synchronous)**
    - ◆ A operação de envio não se completa até que a operação de recepção tenha iniciado.
  - **Pronta (ready)**
    - ◆ A operação de envio só pode ser iniciada após a operação de recepção já ter iniciado.

# Comunicação Ponto a Ponto

- ◆ Quando dois processos estão se comunicando utilizando **MPI\_Send** e **MPI\_Recv**, sua importância aumenta quando módulos de um programa foram escritos independentemente um do outro.
- ◆ Por exemplo, se você quiser utilizar uma biblioteca para resolver um sistema de equações lineares, você pode criar um **comunicador** para ser utilizado exclusivamente pelo solucionador linear e evitar que suas mensagens seja confundidas com outro programa que utilize os mesmos *tags*.

# Comunicação Ponto a Ponto

- ◆ Estaremos utilizando por enquanto o **comunicador** pré-definido **MPI\_COMM\_WORLD**.
- ◆ Ele consiste de todos os processos ativos no programa desde quando a sua execução iniciou.
- ◆ O mecanismo real de troca de mensagens em nossos programas é executado no MPI pelas funções **MPI\_Send** e **MPI\_Recv**.
- ◆ A primeira envia a mensagem para um determinado processo e a segunda recebe a mensagem de um processo.
- ◆ Ambas são **bloqueantes**. A recepção bloqueante espera até que o **buffer** de recepção contenha a mensagem.

# MPI\_Send

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

- ◆ **message**: endereço inicial do dado a ser enviado.
- ◆ **count**: número de dados.
- ◆ **datatype**: MPI\_CHAR, MPI\_INT, MPI\_FLOAT, MPI\_BYTE, MPI\_LONG, MPI\_UNSIGNED\_CHAR, etc.
- ◆ **dest**: *rank* do processo destino.
- ◆ **tag**: etiqueta da mensagem.
- ◆ **comm** : comunicador que especifica o contexto da comunicação e os processos participantes do grupo. O **comunicador** padrão é **MPI\_COMM\_WORLD**.

# MPI\_Recv

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

- ◆ **message**: Endereço inicial do buffer de recepção
- ◆ **count**: Número máximo de dados a serem recebidos
- ◆ **datatype**: MPI\_CHAR, MPI\_INT, MPI\_FLOAT, MPI\_BYTE, MPI\_LONG, MPI\_UNSIGNED\_CHAR, etc.
- ◆ **Source**: *rank* do processo origem ( \* = MPI\_ANY\_SOURCE)
- ◆ **tag**: etiqueta da mensagem ( \* = MPI\_ANY\_TAG)
- ◆ **comm**: comunicador
- ◆ **status**: Estrutura com três campos: MPI\_SOURCE\_TAG, MPI\_TAG, MPI\_ERROR.

# Comunicação Ponto a Ponto

- ◆ A maioria das funções MPI retorna um código de erro inteiro.
- ◆ Contudo, como a maioria dos programadores em C, nós vamos ignorar este código a maior parte das vezes.
- ◆ O conteúdo das mensagens são armazenados em um bloco de memória referenciado pelo argumento *message*.
- ◆ Os próximos dois argumentos, *count* e *datatype* permitem ao sistema identificar o final da mensagem: eles contêm uma seqüência de valores de contagem, cada um contendo um tipo de dados MPI.

# Comunicação Ponto a Ponto

## ◆ Correspondência entre os tipos MPI e C:

<b>MPI datatype</b>	<b>C datatype</b>
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

# Comunicação Ponto a Ponto

- ◆ Os dois últimos tipos, **MPI\_BYTE** e **MPI\_PACKED** não correspondem ao tipos padrão em C.
- ◆ O tipo **MPI\_BYTE** pode ser usado se você desejar não realizar nenhuma conversão entre tipos de dados diferentes.
- ◆ O tipo **MPI\_PACKED** será discutido posteriormente.
- ◆ Note que a quantidade de espaço alocado pelo buffer de recepção não precisa ser igual a quantidade de espaço na mensagem recebida.
- ◆ O MPI permite que uma mensagem seja recebida enquanto houver espaço suficiente alocado.

# Comunicação Ponto a Ponto

- ◆ Os argumentos *dest* e *source* são, respectivamente, o *rank* dos processos de recepção e de envio.
- ◆ O MPI permite que *source* seja um coringa (\*), neste caso usamos **MPI\_ANY\_SOURCE** neste parâmetro.
- ◆ Não há coringa para o destino.
- ◆ O *tag* é um inteiro e, por enquanto, **MPI\_COMM\_WORLD** é nosso único comunicador.
- ◆ Existe um coringa, **MPI\_ANY\_TAG**, que MPI\_Recv pode usar como *tag*.
- ◆ Não existe coringa para o comunicador.

# Comunicação Ponto a Ponto

- ◆ Em outras palavras, para que o processo A possa enviar uma mensagem para o processo B; os argumentos que A usa em **MPI\_Send** devem ser idênticos ao que B usa em **MPI\_Recv**.
- ◆ O último argumento de **MPI\_Recv**, *status*, retorna a informação sobre os dados realmente recebidos.
- ◆ Este argumento referencia um registro com dois campos: um para *source* e outro para *tag*.
- ◆ Então, por exemplo, se o *source* da recepção era **MPI\_ANY\_SOURCE**, então o *status* irá conter o *rank* do processo que enviou a mensagem.

# Utilizando o "Handle Status"

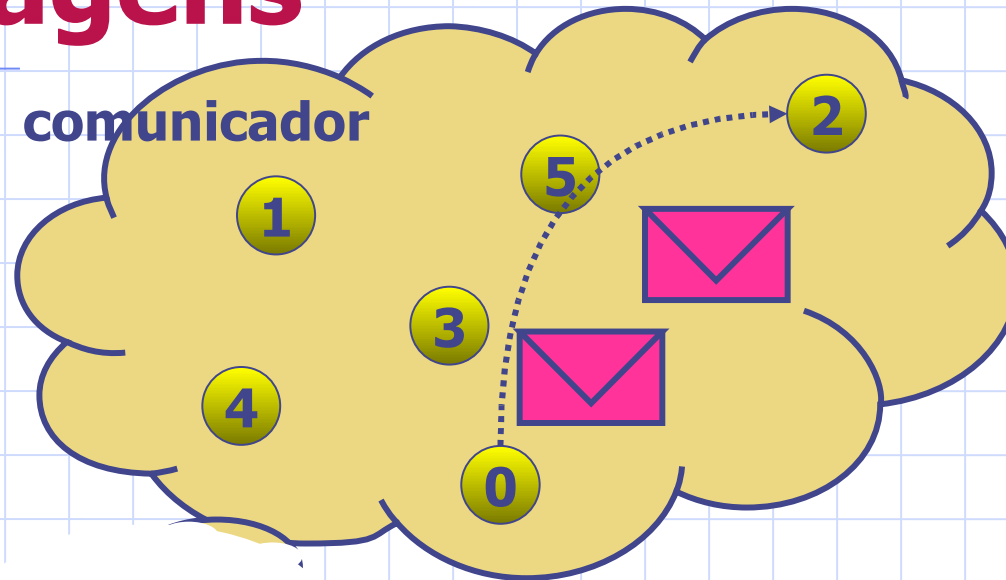
- ◆ Informação sobre a recepção com o uso de coringa é retornada pela função `MPI_RECV` no "handle status".

Informação	C
remetente	status:

- ◆ Para saber o total de elementos recebidos utilize a rotina:

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

# Preservação da Ordem das Mensagens



- ◆ As mensagens não ultrapassam umas às outras.
- ◆ Exemplo: Processo 0 envia duas mensagens.  
Processo 2 chama duas rotinas de recepção  
que combinam com qualquer das duas  
mensagens.  
A ordem é preservada.

# Integral Definida

## Método Trapézio

- ◆ Vamos lembrar que o método do trapézio estima o valor de  $f(x)$  dividindo o intervalo  $[a; b]$  em  $n$  segmentos iguais e calculando a seguinte soma:

$$h \left[ f(x_0)/2 + f(x_n)/2 + \sum_{i=1}^{n-1} f(x_i) \right].$$

Here,  $h = (b - a)/n$ , and  $x_i = a + ih$ ,  $i = 0, \dots, n$ .

- ◆ Colocando  $f(x)$  em uma rotina, podemos escrever um programa para calcular uma integral utilizando o método do trapézio.

# Integral Definida

## Método Trapézio

```
/* A função f(x) é pré-definida.
 * Entrada: a, b, n.
 * Saída: estimativa da integral de a até b de f(x).
 */
#include <stdio.h>
float f(float x) {
float return_val;
/* Calcula f(x). Armazena resultado em return_val. */
...
return return_val;
} /* f */
main() {
float integral; /* Armazena resultado em integral */
float a, b; /* Limite esquerdo e direito */
int n; /* Número de Trapezóides */
float h; /* Largura da base do Trapezóide */
```

# Integral Definida

## Método Trapézio

```
float x;
int i;
    printf("Entre a, b, e n \n");
    scanf("%f %f %d", &a, &b, &n);
    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    x = a;
    for (i = 1; i != n-1; i++) {
        x += h;
        integral += f(x);
    }
    integral *= h;
    printf("Com n = %d trapezóides, a estimativa \n", n);
    printf("da integral de %f até %f = %f \n", a, b, integral);
} /* main */
```

# Integral Definida

## Método Trapézio

- ◆ Uma forma de paralelizar este programa é simplesmente dividir o intervalo  $[a;b]$  entre os processos e cada processo pode fazer a estimativa do valor da integral de  $f(x)$  em seu subintervalo.
- ◆ Para calcular o valor total da integral, os valores calculados localmente são adicionados.
- ◆ Suponha que há “ $p$ ” processos e “ $n$ ” trapézios e, de modo a simplificar a discussão, também supomos que “ $n$ ” é divisível por “ $p$ ”.
- ◆ Então é natural que o primeiro processo calcule a área dos primeiros “ $n/p$ ” trapézios, o segundo processo calcule a área dos próximos “ $n/p$ ” e assim por diante.

# Integral Definida

## Método Trapézio

◆ Então, o processo  $q$  irá estimar a integral sobre o intervalo:

$$\left[ a + q \frac{nh}{p}, a + (q + 1) \frac{nh}{p} \right]$$

◆ Logo cada processo precisa da seguinte informação:

- O número de processos,  $p$ .
- Seu *rank*.
- O intervalo inteiro de integração,  $[a; b]$ .
- O número de subintervalos,  $n$ .

# Integral Definida

## Método Trapézio

- ◆ **Lembre-se que os dois primeiros itens podem ser encontrados chamando as funções MPI:**
  - **MPI\_Comm\_size**
  - **MPI\_Comm\_rank.**
- ◆ **Os dois últimos itens podem ser entrados pelo usuário.**
- ◆ **Para a nossa primeira tentativa de paralelização, vamos dar valores fixos atribuídos no programa.**
- ◆ **Uma maneira direta de calcular a soma de todos os valores locais é fazer cada processo enviar o seu resultado para o processo 0 e este processo fazer a soma final.**

# Método Trapézio Versão Paralela

- ◆ A seguir podemos ver o código paralelizado com uso de MPI:
- ◆ trapezio paralelo

# Entrada de Dados

- ◆ Um problema óbvio com nosso programa é a falta de generalidade: os dados  $a$ ;  $b$  e  $n$  são fixos.
- ◆ O usuário deve poder entrar esses valores durante a execução do programa.
- ◆ A maioria dos processadores paralelos permite que qualquer processo leia da entrada padrão e escreva na saída padrão.
- ◆ Contudo, haverá problemas se diversos processos tentarem realizar simultaneamente operações de E/S, já que a ordem destas operações não pode ser predita "a priori".

# Entrada de Dados

- ◆ Muitos sistemas paralelos permitam que diversos processos realizem operações de E/S.
- ◆ A função **MPI\_Attr\_get** pode determinar o *rank* dos processos que podem realizar as funções usuais de E/S.
- ◆ Contudo, vamos supor, por simplificação, que apenas o processador 0 realize as operações de E/S.
- ◆ Isto é facilmente alcançável com uma função de E/S que usa as funções **MPI\_Send** e **MPI\_Recv**.

# Entrada de Dados

**/\* Função Get\_data**

**\* Lê a entrada do usuário a, b e N.**

**\* Parâmetros de entrada:**

**\* 1. int my\_rank: rank do processo atual.**

**\* 2. int p: número de processos.**

**\* Parâmetros de Saída :**

**\* 1. float\* a\_ptr: ponteiro para o limite esquerdo a.**

**\* 2. float\* b\_ptr: ponteiro para o limite direito b.**

**\* 3. int\* n\_ptr: ponteiro para o número de trapezóides.**

**\* Algoritmo:**

**\* 1. Processo 0 solicita ao usuário para entrada e lê esses valores.**

**\* 2. Processo 0 envia os valores de entrada para outros processos.**

**\*/**

# Entrada de Dados

```
void Get_data(int my_rank, int p, float* a_ptr, float* b_ptr, int*
    n_ptr) {
    int source = 0;    /* Todas as variáveis locais usadas por */
    int dest, tag;    /* MPI_Send e MPI_Recv */
    MPI_Status status;
    if (my_rank == 0) {
        printf("Entre a, b, e n \n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
        for (dest = 1; dest != p; dest++)-{
            tag = 30;
            MPI_Send(a_ptr, 1, MPI_FLOAT, dest, tag,
                MPI_COMM_WORLD);
            tag = 31;
            MPI_Send(b_ptr, 1, MPI_FLOAT, dest, tag,
                MPI_COMM_WORLD);
```

# Entrada de Datos

```
    tag = 32;
    MPI_Send(n_ptr, 1, MPI_INT, dest, tag,
MPI_COMM_WORLD); }
} else {
    tag = 30;
    MPI_Recv(a_ptr, 1, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &status);
    tag = 31;
    MPI_Recv(b_ptr, 1, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &status);
    tag = 32;
    MPI_Recv(n_ptr, 1, MPI_INT, source, tag,
MPI_COMM_WORLD, &status);
}
} /* Get_data */
```



# **Comunicação Coletiva**

# Comunicação Coletiva

- ◆ As operações de comunicação coletiva são mais restritivas que as comunicações ponto a ponto:
  - A quantidade de dados enviados deve casar exatamente com a quantidade de dados especificada pelo receptor.
  - Apenas a versão **bloqueante** das funções está disponível.
  - O argumento *tag* não existe.
  - As funções estão disponíveis apenas no modo "padrão".
- ◆ Todos os processos participantes da comunicação coletiva chamam a mesma função com argumentos compatíveis.

# Comunicação Coletiva

◆ Quando uma operação coletiva possui um único processo de origem ou um único processo de destino, este processo é chamado de **raiz**.

## ◆ Barreira

- Bloqueia todos os processos até que todos os processos do grupo chamem a função.

## ◆ Difusão (***broadcast***)

- Mensagem para todos os processos.

## ◆ Gather (coletar)

- Dados recolhidos de todos os processos em um único processo.

## ◆ Scatter (dispersar)

- Dados distribuídos de um processo para todos.

# Comunicação Coletiva

## ◆ Allgather

- **Gather** seguida de uma difusão.

## ◆ Alltoall

- Conjunto de **gathers** onde cada processo recebe dados diferentes.

## ◆ Redução (**reduce**)

- Operações de soma, máximo, mínimo, etc.

## ◆ Allreduce

- Uma redução seguida de uma difusão.

## ◆ Reducescatter

- Um redução seguida de um **scatter**.

# Comunicação Coletiva

## ◆ Exemplos de funções MPI para realizar comunicação coletiva são:

- **MPI\_BARRIER:** Bloqueia o processo até que todos os processos associados ao **comunicador** chamem essa função.
- **MPI\_BCAST:** Faz a difusão de uma mensagem do processo **raiz** para todos os processos associados ao **comunicador**.
- **MPI\_GATHER:** Cada processo, incluindo o **raiz**, manda uma mensagem para o processo **raiz**, que ao recebê-la, armazena-as na ordem de chegada.
- **MPI\_SCATTER:** Executa uma operação inversa ao **MPI\_GATHER**.

# Comunicação Coletiva

- **MPI\_ALLGATHER:** Operação de *gather* em que todos os processos, e não apenas na **raiz**, recebem as mensagens.
- **MPI\_ALLTOALL:** Extensão da operação *allgather*, onde cada processo envia um dado diferente para cada um dos receptores. O *j-ésimo* dado enviado pelo processo *i* é recebido pelo processo *j* e armazenado no *i-ésimo* bloco do seu buffer de recepção.
- **MPI\_REDUCE:** Combina todos os elementos presentes no *buffer* de cada processo do grupo usando a operação definida como parâmetro e coloca o valor resultante no *buffer* do processo especificado
- **MPI\_ALLREDUCE**
  - ♦ **ALLREDUCE = REDUCE + BROADCAST**
- **MPI\_REDUCE\_SCATTER**
  - ♦ **REDUCE\_SCATTER = REDUCE + SCATTER**

# Difusão

- ◆ Um padrão de comunicação que envolva todos os processos em um **comunicador** é chamada de comunicação coletiva.
- ◆ Uma difusão (***broadcast***) é uma comunicação coletiva na qual um único processo envia os mesmos dados para cada processo.
- ◆ A função MPI para difusão é:

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

# Difusão

- ◆ Ela simplesmente envia uma cópia dos dados de *message* no processo *root* para cada processo no comunicador *comm*.
- ◆ Deve ser chamado por todos os processos no comunicador com os mesmos argumentos para *root* e *comm*.
- ◆ Uma mensagem de broadcast não pode ser recebida com **MPI\_Recv**.
- ◆ Os parâmetros *count* e *datatype* têm a mesma função que nas funções **MPI\_Send** e **MPI\_Recv**: especificam o tamanho da mensagem.

# Difusão

- ◆ Contudo, ao contrário das funções ponto-a-ponto, o padrão MPI exige que *count* e *datatype* sejam os mesmos para todos os processos no mesmo **comunicador** para uma comunicação coletiva.
- ◆ A razão para isto é um único processo pode receber dados de muitos outros processos, e para poder determinar o total de dados recebidos, seria necessário um vetor inteiro de status de retorno.

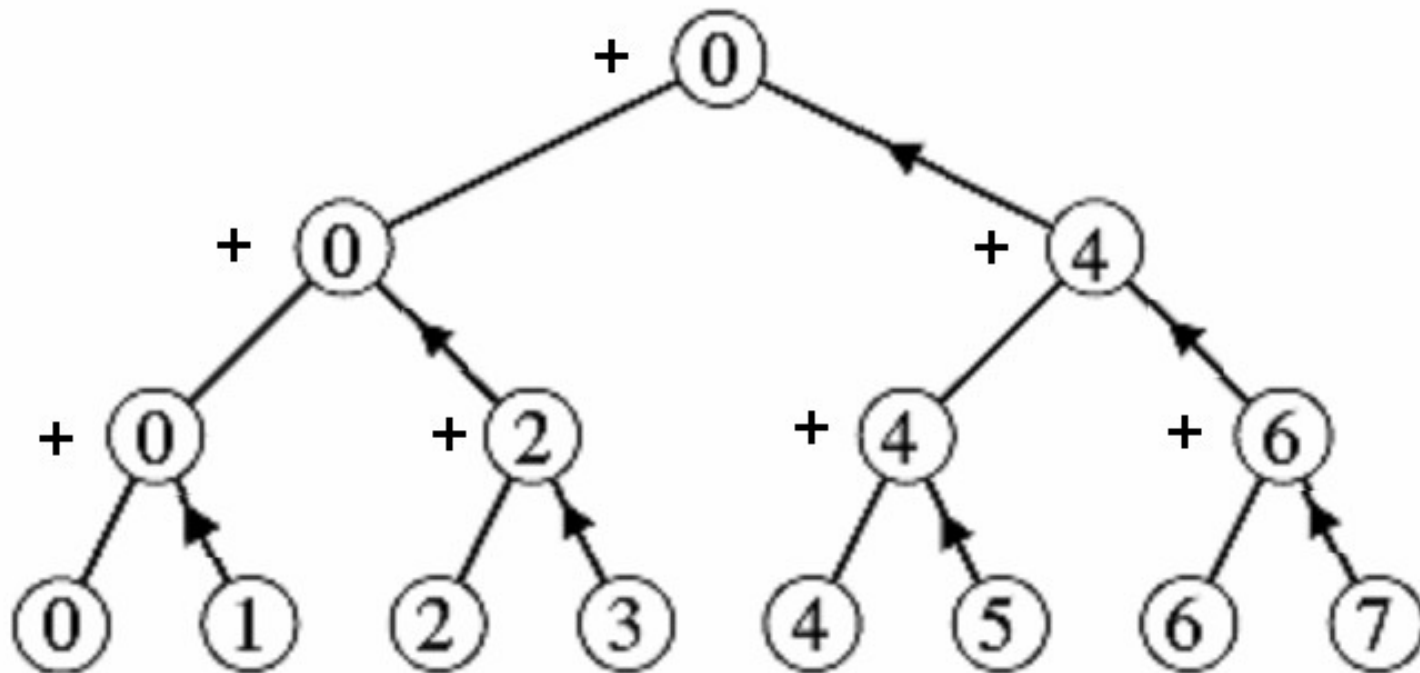
# Difusão - GetData

```
void Get_data2(int my_rank, float* a_ptr, float* b_ptr, int* n_ptr)
{
  int root = 0; /* Argumentos para MPI_Bcast */
  int count = 1;
  if (my_rank == 0)
  {
    printf("Entre a, b, e n \n");
    scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
  }
  MPI_Bcast(a_ptr, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
  MPI_Bcast(b_ptr, 1, MPI_FLOAT, root, MPI_COMM_WORLD);
  MPI_Bcast(n_ptr, 1, MPI_INT, root, MPI_COMM_WORLD);
} /* Get_data2 */
```

# Redução

- ◆ No programa do método do trapézio, depois da fase de entrada, cada processador executa os mesmos comandos até o final da fase de soma.
- ◆ Contudo, este não é caso depois da final da fase de soma, onde as tarefas não são bem balanceadas.
- ◆ Podemos utilizar o seguinte procedimento:
  - a) 1 envia resultado para 0, 3 para 2, 5 para 4, 7 para 6.
  - b) 0 soma sua integral com a de 1, 2 soma com a de 3, etc.
  - c) 2 envia para 0, 6 envia para 4.
  - d) 0 soma, 4 soma.
  - e) 4 envia para 0.
  - f) 0 soma.

# Redução



# Redução

- ◆ A soma global que estamos tentando calcular é um exemplo de uma classe geral de operações de comunicação coletivas chamada operações de redução.
- ◆ Em uma operação global de redução, todos os processos em um comunicador contribuem com dados que são combinados em operações binárias.
- ◆ Operações binárias típicas são a adição, máximo, mínimo, e lógico, etc.
- ◆ É possível definir operações adicionais além das mostradas para a função **MPI\_Reduce**.

# Redução

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

- ◆ A operação **MPI\_Reduce** combina os operandos armazenados em *\*operand* usando a operação *op* e armazena o resultado em *\*result* no processo *root*.
- ◆ Tanto *operand* como *result* referem-se a *count* posições de memória com o tipo *datatype*.
- ◆ **MPI\_Reduce** deve ser chamada por todos os processos no comunicador *comm* e os valores de *count*, *datatype* e *op* devem ser os mesmos em cada processo.

# Redução

- ◆ O argumento *op* pode ter um dos seguintes valores pré-definidos:

## Nome da Operação Significado

<b>MPI_MAX</b>	<b>Máximo</b>
<b>MPI_MIN</b>	<b>Mínimo</b>
<b>MPI_SUM</b>	<b>Soma</b>
<b>MPI_PROD</b>	<b>Produto</b>
<b>MPI_LAND</b>	<b>"E" lógico</b>
<b>MPI_BAND</b>	<b>"E" bit a bit</b>
<b>MPI_LOR</b>	<b>"Ou" lógico</b>
<b>MPI_BOR</b>	<b>"Ou" bit a bit</b>
<b>MPI_LXOR</b>	<b>"Ou Exclusivo" lógico</b>
<b>MPI_BXOR</b>	<b>"Ou Exclusivo" bit a bit</b>
<b>MPI_MAXLOC</b>	<b>Máximo e Posição do Máximo</b>
<b>MPI_MINLOC</b>	<b>Mínimo e Posição do Mínimo</b>

# Redução

- ◆ Com um exemplo, vamos reescrever as últimas linhas do programa do método do trapézio:

...

```
/* Adiciona as integrais calculadas por cada processo */
```

```
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
```

```
MPI_SUM, 0, MPI_COMM_WORLD);
```

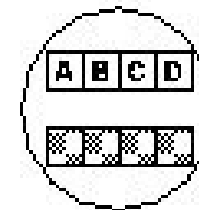
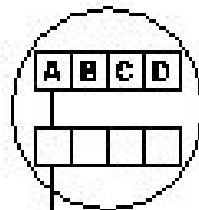
```
/* Imprime o resultado */
```

...

- ◆ Note que cada processo chama a rotina **MPI\_REDUCE** com os mesmos argumentos.
- ◆ Em particular, embora *total* tenha apenas significado no processo 0, cada processo deve fornecê-lo como argumento.

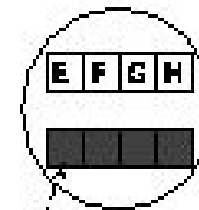
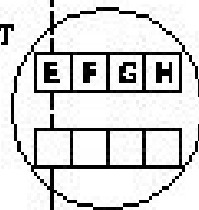
RANK

0

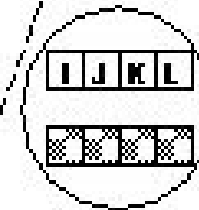
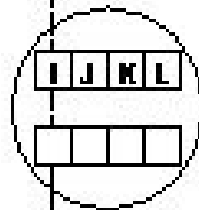


1

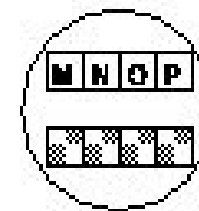
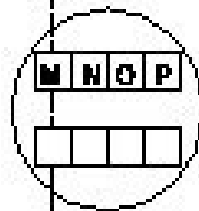
ROOT



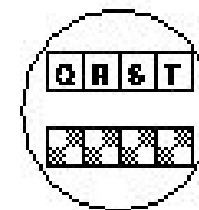
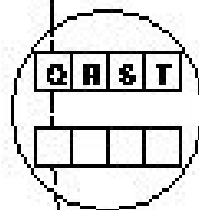
2



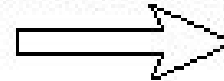
3



4



MPI\_REDUCE



AcEoloMcQ

# Barreira

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

- ◆ A função **MPI\_Barrier** fornece um mecanismo para sincronizar todos os processos no comunicador *comm*.
- ◆ Cada processo bloqueia (i.e., pára) até todos os processos em *comm* tenham chamado **MPI\_Barrier**.

# Gather

```
int MPI_Gather(void* send_buf, int send_count,
MPI_Datatype send_type, void* recv_buf, int
recv_count, MPI_Datatype recv_type, int root,
MPI_comm comm)
```

- ◆ Cada processo em *comm* envia o conteúdo de *send\_buf* para o processo com *rank* igual a *root*.
- ◆ O processo *root* concatena os dados recebidos na ordem definida pelo *rank* em *recv\_buf*.
- ◆ Os argumentos *recv* são significativos apenas no processo com *rank* igual a *root*.
- ◆ O argumento *recv\_count* indica o número de itens recebidos de cada processo, não número total recebido.

# Scatter

```
int MPI_Scatter(void* send_buf, int send_count,  
MPI_Datatype send_type, void* recv_buf, int  
recv_count, MPI_Datatype recv_type, int root,  
MPI_Comm comm)
```

- ◆ O processo com o *rank* igual a *root* distribui o conteúdo de *send\_buf* entre os processos.
- ◆ O conteúdo de *send\_buf* é dividido em *p* segmentos cada um consistindo de *send\_count* itens.
- ◆ O primeiro segmento vai para o processo 0, o segundo para o processo 1, etc.
- ◆ O argumento *send\_buf* é significativo apenas no processo *root*.

# Allgather

```
int MPI_Allgather(void* send_buf, int send_count,  
MPI_Datatype send_type, void* recv_buf, int  
recv_count, MPI_Datatype recv_type, MPI_comm  
comm)
```

- ◆ **MPI\_Allgather** junta o conteúdo de *send\_buf* em cada processo.
- ◆ Seu efeito é equivalente a uma seqüência de **p** chamadas a **MPI\_Gather**, cada qual com um processo diferente agindo como processo raiz.

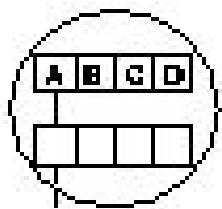
# Allreduce

```
int MPI_Allreduce (void* operand, void* result, int  
count, MPI_Datatype datatype, MPI_Op op,  
MPI_Comm comm)
```

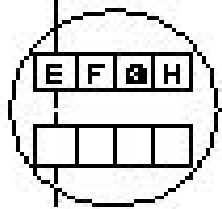
- ◆ **MPI\_Allreduce** armazena o resultado da operação de redução *op* no **buffer *result*** de cada processo.

**RANK**

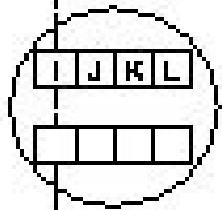
0



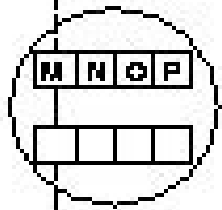
1



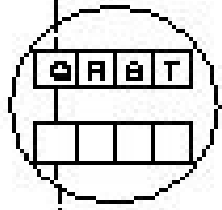
2



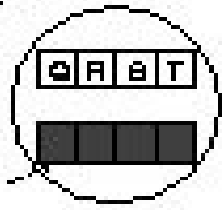
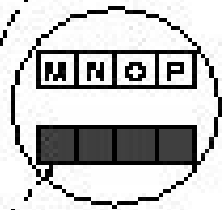
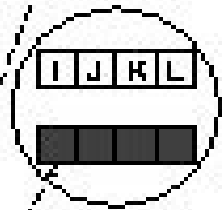
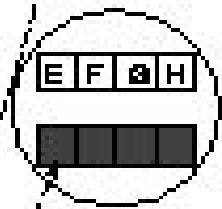
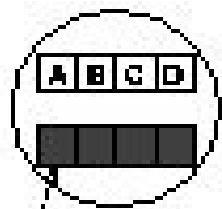
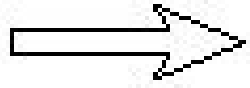
3



4



**MPI\_ALLREDUCE**



**AoEoIoMoQ**



# Comunicação em Árvore

- ◆ O que acontece quando o programa do método do trapézio executa com, digamos, oito processos?
- ◆ Todos os processos começam a execução mais ou menos ao mesmo tempo, mas, depois que o processo 0 pega os parâmetros de entrada, ele realiza uma distribuição seqüencial dos parâmetros para os demais processos.
- ◆ No final do programa o mesmo problema ocorre, quando o processo 0 faz todo o trabalho de coleta dos dados e soma dos valores locais obtidos por cada processo.

# Comunicação em Árvore

- ◆ Isto é altamente indesejável, pois se apenas um processo está realizando todo o trabalho, poderíamos utilizar apenas uma máquina com um processador.
- ◆ Uma solução é utilizar uma estrutura em árvore para dividir o trabalho melhor entre os processos, com o processo 0 como raiz da árvore.
- ◆ Como pode ser visto na figura a seguir, este esquema de distribuição pode reduzir esta etapa de 7 para 3 estágios.
- ◆ Genericamente, se houver  $p$  processos, este procedimento permite distribuir os dados em  $\text{ceil}[\log_2(p)]$ , ao invés de  $p-1$  estágios.

# Comunicação em Árvore

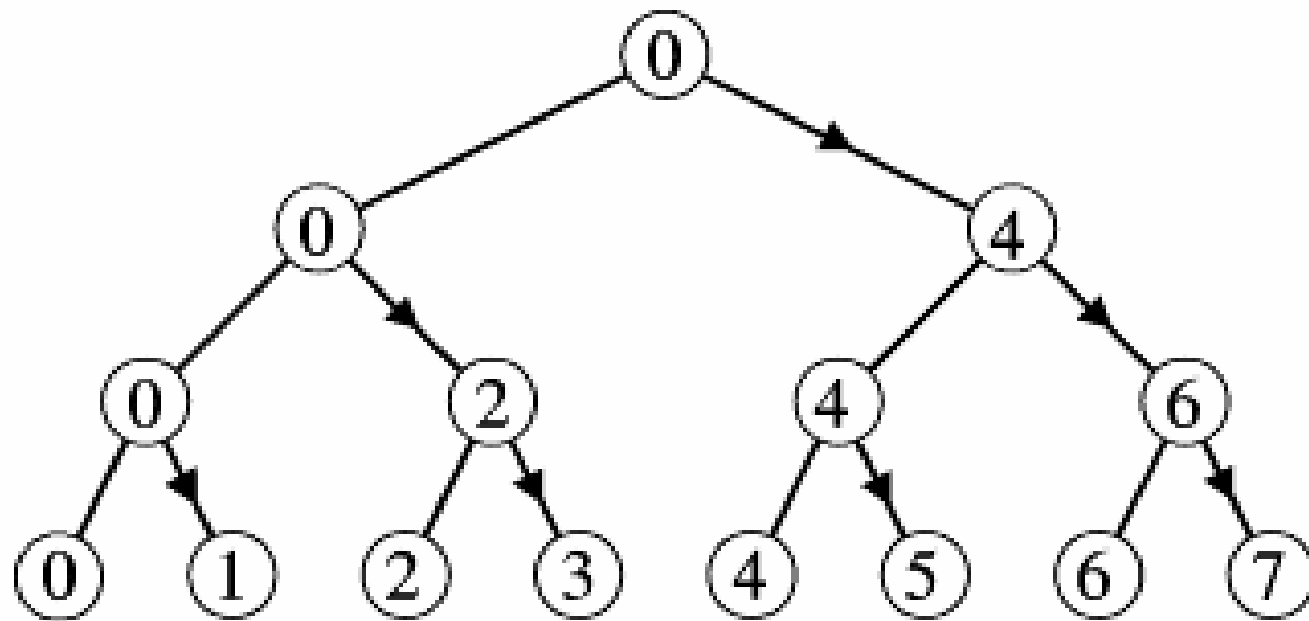
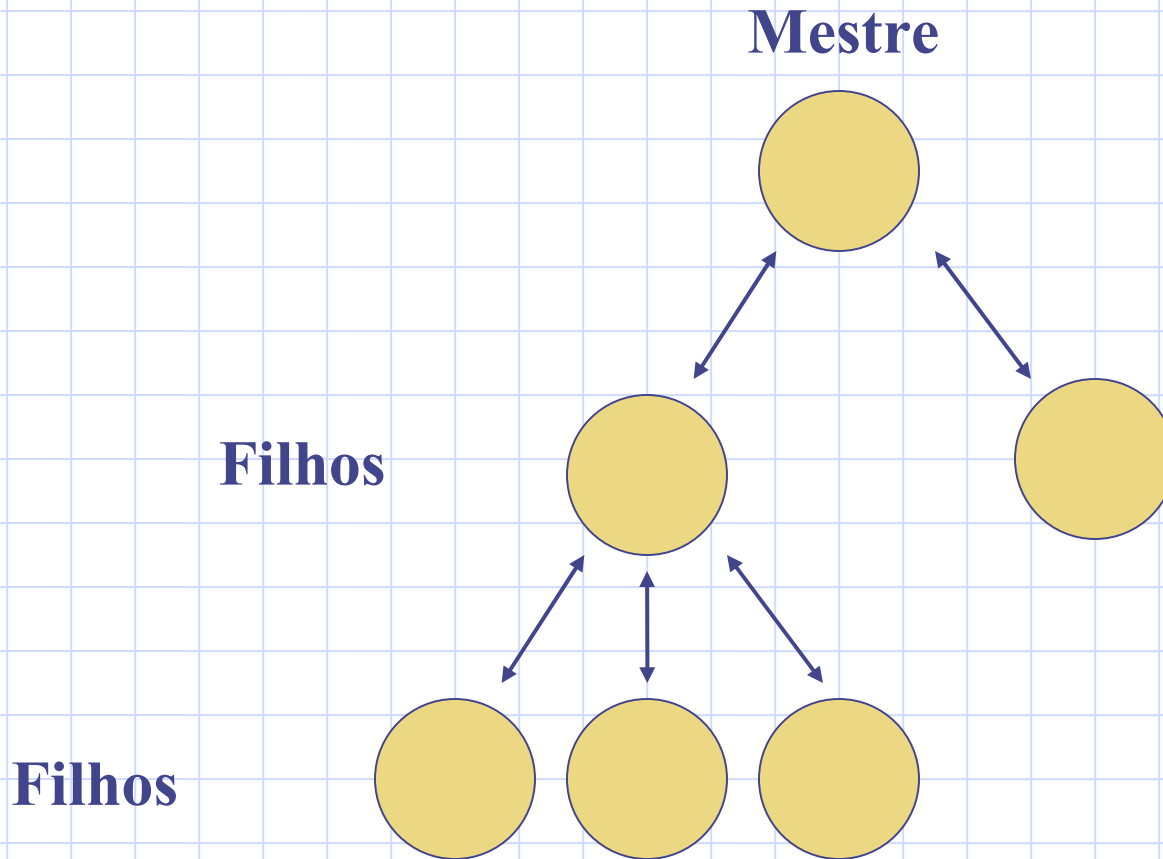


Figure 1: Processors configured as a tree

# Comunicação em Árvore



# Comunicação Coletiva- Aplicação

◆ O valor de  $\pi$  pode ser obtido pela integração numérica :

$$\int_0^1 \frac{4}{1+x^2}$$

◆ Calculada em paralelo, dividindo-se o intervalo de integração entre os processos.

# Cálculo de Pi

```
#include "mpi.h"
#include <math.h>
int main (argc, argv)
int argc;
char argv[ ];
{ int n, myid, numprocs, i, rc;
  double mypi, pi, h, x, sum = 0.0;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0)
  { printf ("Entre com o número de intervalos: ");
    scanf("%d", &n);
  }
}
```

# Cálculo de Pi

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n != 0)
{
    h=1.0/(double) n;
    for (i=myid +1; i <= n; i+=numprocs)
    {
        x = h * ((double) i - 0.5);
        sum += (4.0/(1.0 + x*x));
    }
    mpi = h* sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
0, MPI_WORLD_COMM);
    if (myid == 0) printf ("valor aproximado de pi: %.
16f \n", pi);
}
MPI_Finalize( );
}
```



# **Agrupando Datos (Constructores de Tipos de Datos)**

# Agrupando Dados

◆ MPI provê três mecanismos para agrupamento de vários itens em uma única mensagem:

- O parâmetro *count* para várias rotinas de comunicação;
- Tipos de dados derivados;
- As rotinas MPI Pack/MPI Unpack.

◆ O parâmetro *count*:

- Lembre-se que as rotinas **MPI\_Send**, **MPI\_Receive**, **MPI\_Bcast** e **MPI\_Reduce** têm todas um argumento *count* e um *datatype*.
- Este dois parâmetros permitem ao usuário agrupar itens de dados tendo os **mesmos** tipos básicos em uma única mensagem.

# Agrupando Dados

- ◆ De modo a utilizá-los, os itens de dados agrupados devem estar armazenados em posições **contíguas** de memória.
- ◆ Já que a linguagem C garante que os elementos de um arranjo (matriz ou vetor) estão armazenados em posições contíguas de memória, se desejarmos enviar os elementos de um arranjo, ou um subconjunto deste, podemos fazê-lo com uma única mensagem.
- ◆ Suponha que desejamos enviar a segunda metade de um vetor com 100 valores de ponto flutuante do processo 0 para o processo 1.

# Agrupando Dados

```
float vector[100];
int tag, count, dest, source;
MPI_Status status;
int p;
int my_rank;
...
if (my_rank == 0) --
    /* Inicia o vetor e envia */
    ...
    tag = 47;
    count = 50;
```

# Agrupando Datos

```
dest = 1;
MPI_Send(vector + 50, count, MPI_FLOAT,
dest, tag, MPI_COMM_WORLD);
else -- /* my_rank == 1 */
tag = 47;
count = 50;
source = 0;
MPI_Recv(vector+50, count, MPI_FLOAT,
source, tag, MPI_COMM_WORLD, &status);
```

# Agrupando Dados

◆ Infelizmente isto não resolve o problema do programa do método do trapézio, que tem as variáveis

```
float a;
```

```
float b;
```

```
int n;
```

◆ Pois a linguagem C não garante que elas estejam armazenadas em posições contíguas de memória.

# Agrupando Dados

- ◆ Parece que uma outra opção seria armazenar  $a$ ;  $b$ ; e  $n$  em uma estrutura com três campos – dois *floats* e um *int* – e usar o argumento *datatype* em **MPI\_Bcast**.
- ◆ Mas isto não resolveria porque o tipo de dado criado não está definido nos tipos de dados possíveis de serem utilizados no MPI.
- ◆ O problema aqui é que o MPI é uma biblioteca de funções pré-existentes.
- ◆ Isto é, as funções MPI são escritas sem conhecimento dos tipos de dados que você define no seu programa.

# Agrupando Dados

- ◆ O MPI provê uma solução parcial para este problema, permitindo ao usuário criar tipos de dados MPI em tempo de execução.
- ◆ De modo a criar um tipo de dados MPI, você deve especificar o leiaute dos dados no tipo: os tipos dos elementos e sua posição relativa na memória, que chamamos de **tipos de dados compostos ou derivados**.
- ◆ De modo a ver como isto funciona, vamos construir uma função que irá criar um tipo derivado de dados.

# Agrupando Dados

```
void Build_derived_type(INDATA_TYPE* indata, MPI_Datatype*  
    message_type_ptr){  
    int block_lengths[3];  
    MPI_Aint displacements[3];  
    MPI_Aint addresses[4];  
    MPI_Datatype typelist[3];  
    /* Constrói um tipo de dados derivado que consite de um int e  
       dois floats */  
    /* Primeiro especifica os tipos */  
    typelist[0] = MPI_FLOAT;  
    typelist[1] = MPI_FLOAT;  
    typelist[2] = MPI_INT;  
    /* Especifica o número de elementos de cada tipo */  
    block_lengths[0] = block_lengths[1] = block_lengths[2] = 1;
```

# Agrupando Dados

```
/* Calcula os deslocamentos dos membros relativos a indata */  
MPI_Address(indata, &addresses[0]);  
MPI_Address(&(indata->a), &addresses[1]);  
MPI_Address(&(indata->b), &addresses[2]);  
MPI_Address(&(indata->n), &addresses[3]);  
displacements[0] = addresses[1] - addresses[0];  
displacements[1] = addresses[2] - addresses[0];  
displacements[2] = addresses[3] - addresses[0];  
/* Cria o tipo derivado */  
MPI_Type_struct(3, block_lengths, displacements, typelist,  
message_type_ptr);  
/* Finaliza de modo que ele possa ser usado */  
MPI_Type_commit(message_type_ptr);  
}/* Constrói tipo derivado */
```

# Agrupando Dados

- ◆ A primeira das três sentenças especificam os tipos dos membros do tipo derivado e a próxima sentença especifica o número de elementos de cada tipo.
- ◆ As próximas 4 sentenças calculam os endereços dos três membros de *indata*.
- ◆ As próximas três sentenças usam o endereço calculado para determinar os deslocamentos dos três membros relativos ao endereço do primeiro – ao qual é dado o deslocamento 0.

# Agrupando Dados

- ◆ Com esta informação, sabemos os tipos, tamanhos e posições relativas dos membros de uma variável tendo o tipo C `INDATA_TYPE`, e portanto podemos definir um tipo de dados derivado que corresponda ao tipo C.
- ◆ Isto é feito chamando as funções **`MPI_Type_struct`** e **`MPI_Type_commit`**.
- ◆ O novo tipo de dados MPI criado pode ser usado em qualquer função de comunicação.
- ◆ Para poder utilizá-lo, nós simplesmente usamos o endereço inicial de uma variável do tipo `INDATA_TYPE` como primeiro argumento e o tipo derivado no argumento ***`datatype`***.

# Agrupando Dados

```
void Get_data3(INDATA_TYPE* indata, int my_rank){
MPI_Datatype message_type; /* Argumentos para */
int root = 0;             /* MPI_Bcast */
int count = 1;
    if (my_rank == 0){
        printf("Entre a, b, e n /n");
        scanf("%f %f %d", &(indata->a), &(indata->b),
            &(indata->n));
        Build_derived_type(indata, &message_type);
        MPI_Bcast(indata, count, message_type, root,
MPI_COMM_WORLD);
    }
}
/* Get_data3 */
```

# Agrupando Dados

- ◆ Note que calculamos os endereços dos membros de *indata* com **MPI\_Address** ao invés do operador & em C.
- ◆ A razão para isto é que o ANSI C não exige que um ponteiro seja um inteiro (embora comumente isto aconteça).
- ◆ Note também que o tipo do vetor de deslocamentos é **MPI\_Aint** --- não **int**.
- ◆ Este é um tipo especial em MPI que permite que endereços longos sejam armazenados em um inteiro.

# MPI\_Type\_struct

◆ Resumindo, nós podemos criar tipos de dados derivados genéricos chamando **MPI\_Type\_struct**, cuja sintaxe é:

```
int MPI_Type_Struct(int count, int*  
array_of_block_lengths, MPI_Aint*  
array_of_displacements, MPI_Datatype*  
array_of_types, MPI_Datatype* newtype)
```

# MPI\_Type\_struct

- ◆ O argumento *count* é o número de elementos no tipo derivado. É também o tamanho dos três vetores: *array\_of\_block\_lengths*, *array\_of\_displacements* e *array\_of\_types*.
- ◆ O vetor *array\_of\_block\_lengths* contém o número de entradas em cada elemento do tipo.
- ◆ Então, se um elemento do tipo é um vetor de *m* valores, então a entrada correspondente em *array\_of\_block\_lengths* é *m*.
- ◆ O vetor *array\_of\_displacements* contém o deslocamento de cada elemento do início da mensagem, e o vetor *array\_of\_types* contém o tipo de dados MPI de cada entrada.

# MPI\_Type\_struct

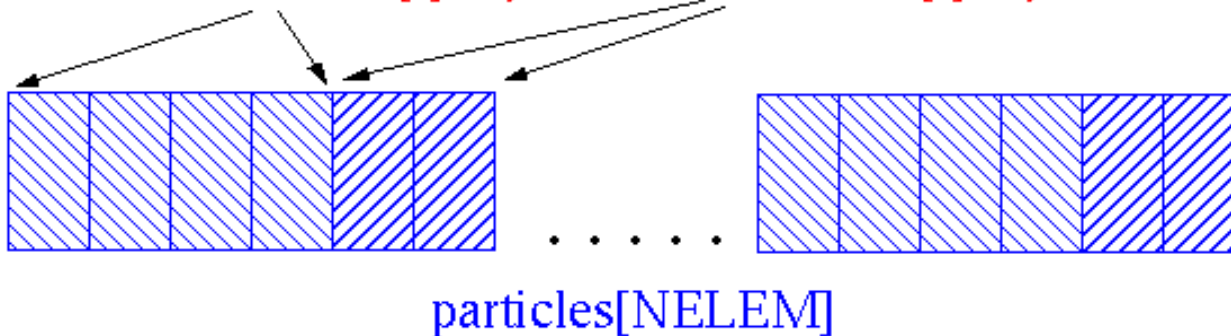
- ◆ O argumento *newtype* retorna um ponteiro para o tipo de dados MPI criado pela chamada **MPI\_Type\_struct**.
- ◆ Note também que *newtype* e as entradas em *array\_of\_types* todas tem tipo de dados MPI.
- ◆ Então **MPI\_Type\_struct** pode ser chamada recursivamente para criar tipos de dados derivados mais complexos.

# MPI\_Type\_struct

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;  
Particle particles[NELEM];
```

```
MPI_Type_extent(MPI_FLOAT, &extent);
```

```
count = 2; oldtypes[0] = MPI_FLOAT; oldtypes[1] = MPI_INT  
offsets[0] = 0; offsets[1] = 4 * extent;  
blockcounts[0] = 4; blockcounts[1] = 2;
```



```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);
```

```
MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

# Outros Construtores

- ◆ **MPI\_Type\_struct** é o construtor de tipos de dados mais geral no MPI, e, como consequência, o usuário deve fornecer uma descrição de cada elemento do tipo.
- ◆ Se os dados a serem transmitidos consistem de um subconjunto de entradas de um vetor, não precisamos fornecer informações tão detalhadas, já que todos os elementos tem o mesmo tipo básico.
- ◆ MPI provê três construtores de tipos de dados derivados para lidar com esta situação:  
**MPI\_Type\_Contiguous, MPI\_Type\_vector e MPI\_Type\_indexed.**

# Outros Construtores

- ◆ O primeiro construtor constrói um tipo derivado cujos elementos são entradas contíguas em um vetor.
- ◆ O segundo constrói um tipo cujos elementos são entradas igualmente espaçadas de um vetor.
- ◆ O terceiro constrói um tipo cujos elementos são entradas arbitrárias de um vetor.
- ◆ Note que antes que qualquer tipo derivado possa ser utilizado para comunicação ele deve ser concluído com uma chamada para **MPI\_Type\_commit**.
- ◆ Detalhes da sintaxe dos construtores de tipo adicionais são mostrados a seguir.

# MPI\_Type\_contiguous

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

- ◆ **MPI\_Type\_contiguous** cria um tipo derivado de dados consistindo de *count* elementos do tipo *oldtype*. Os elementos pertencem a posições de memória contíguas.

# 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

# MPI\_Type\_vector

```
int MPI_Type_vector(int count, int block_length,  
int stride, MPI_Datatype element_type,  
MPI_Datatype* newtype)
```

- ◆ **MPI\_Type\_vector** cria um tipo derivado que consiste de *count* elementos. Cada elemento contém *block\_length* entradas do tipo *element\_type*. Stride é o número de elementos de tipo *element\_type* entre sucessivos elementos de *newtype*.

# MPI\_Type\_vector

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

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, column_type, dest, tag, comm);
```

2.0	6.0	10.0	14.0
-----	-----	------	------

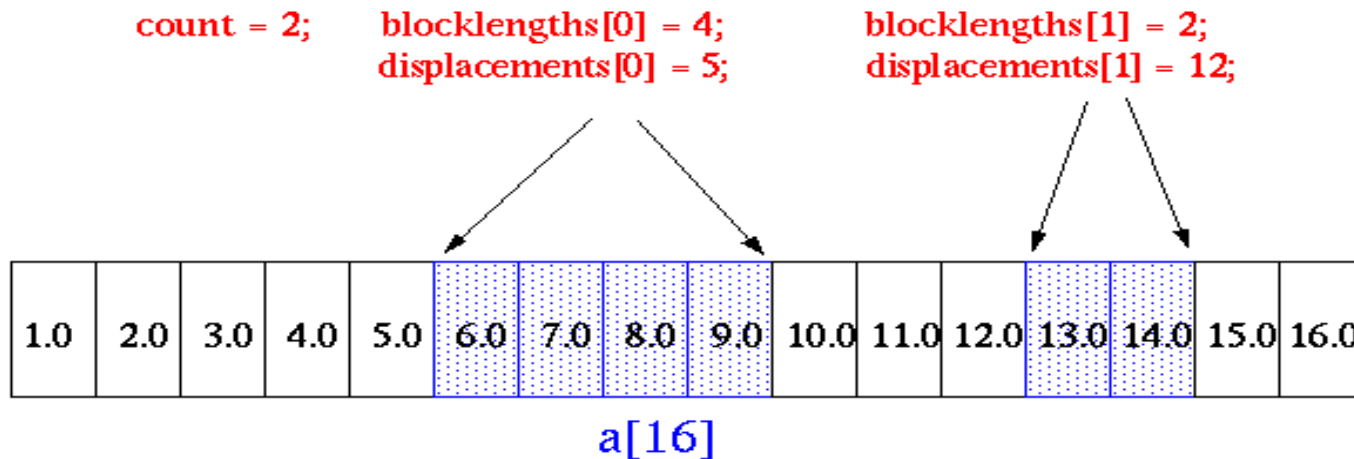
1 element of  
column\_type

# MPI\_Type\_indexed

```
int MPI_Type_indexed(int count, int*  
array_of_block_lengths, int*  
array_of_displacements, MPI_Datatype  
element_type, MPI_Datatype* newtype)
```

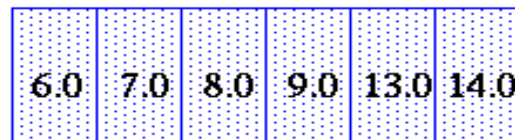
- ◆ **MPI\_Type\_indexed** cria um tipo derivado de dados consistindo de *count* elementos.
- ◆ O *i*-ésimo elemento ( $i = 0; 1; \dots; \text{count} - 1$ ), consiste de um vetor de *block\_lengths[i]* entradas do tipo *element\_type*, e é deslocado *array\_of\_displacements[i]* unidades do tipo *element\_type* do começo do novo tipo.

# MPI\_Type\_indexed



```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



1 element of  
indextype



**Pack/Unpack**

# Pack/Unpack

- ◆ Algumas bibliotecas de comunicação oferecem funções de pack/unpack para o envio de dados não-contíguos.
- ◆ O usuário empacota os dados explicitamente em um buffer antes de enviá-los e desempacota de um buffer contíguo depois de recebe-los.
- ◆ Tipos de dados derivados evitam que tenhamos que fazer estas operações de empacotamento e desempacotamento.
- ◆ Neste caso, o usuário especifica o leiaute dos dados a serem enviados ou recebidos e a biblioteca de comunicação acessa um *buffer* não-contíguo.

# Pack/Unpack

- ◆ **As rotinas de pack/unpack são oferecidas para prover compatibilidade com bibliotecas anteriores e são uma opção para o envio de dados não contíguos.**
- ◆ **Em realidade, uma mensagem pode ser recebida em diversas partes, onde a operação de recepção feita em uma parte depende do conteúdo de uma parte anterior.**
- ◆ **Um outro uso é que as mensagens de saída podem ser explicitamente "bufferizadas" em um espaço fornecido pelo usuário, sobrepondo-se a política de "bufferização" do sistema.**

# Pack/Unpack

- ◆ Finalmente, a disponibilidade das operações de pack/unpack facilita o desenvolvimento de bibliotecas de comunicação construídas com o uso do MPI.

# MPI\_Pack

```
int MPI_Pack(void* inbuf, int incount, MPI  
Datatype datatype, void *outbuf, int outsize, int  
*position, MPI Comm comm)
```

- ◆ ***inbuf***: início do buffer de entrada
- ◆ ***incount***: número de itens de entrada (inteiro)
- ◆ ***datatype***: tipo de dados de cada entrada (handle)
- ◆ ***outbuf***: início do buffer de saída
- ◆ ***outsize***: tamanho do buffer de saída (bytes)
- ◆ ***position***: posição atual no buffer (bytes)
- ◆ ***comm***: comunicador para a mensagem empacotada (handle)

# MPI\_Pack

- ◆ Empacota a mensagem no *buffer* de envio especificado por *inbuf*, *incount*, *datatype* no espaço do buffer especificado por *outbuf* e *outside*.
- ◆ O *buffer input* pode ser qualquer *buffer* de comunicação permitido em MPI\_SEND.
- ◆ O *buffer* de saída é uma área de armazenamento contíguo que contém *outside* bytes, iniciando no endereço *outbuf* (o comprimento é contado em bytes, não elementos, como se houvesse um *buffer* de comunicação para a mensagem de tipo MPI\_PACKED).

# MPI\_Pack

- ◆ O valor de entrada de *position* é a primeira posição no buffer de saída para ser usada para o empacotamento.
- ◆ A variável *position* é incrementada do tamanho da mensagem empacotada, e o valor de saída de *position* é a primeira posição no buffer de saída seguinte às posições ocupadas pela mensagem empacotada.
- ◆ O argumento *comm* é o comunicador que será usado subsequente para o envio da mensagem empacotada.

# MPI\_Unpack

```
int MPI_Unpack(void* inbuf, int insize, int  
*position, void *outbuf, int outcount, MPI  
Datatype datatype, MPI Comm comm)
```

- ◆ ***inbuf***: início do buffer de entrada.
- ◆ ***insize***: tamanho do buffer de entrada (bytes).
- ◆ ***position***: posição atual (bytes).
- ◆ ***outbuf***: início do buffer de saída
- ◆ ***outcount***: número de itens a serem desempacotados.
- ◆ ***datatype***: tipo de dados de cada item de saída (handle)
- ◆ ***comm***: comunicador para a mensagem empacotada (handle)

# MPI\_Unpack

- ◆ Desempacota uma mensagem para um buffer de recepção especificado por *outbuf*, *outcount*, *datatype* a partir do espaço do *buffer* especificado por *inbuf* e *insize*.
- ◆ O *buffer* de saída pode ser qualquer *buffer* de comunicação permitido em MPI\_RECV.
- ◆ O *buffer* de entrada é a área de armazenamento contíguo contendo *insize* bytes, iniciando no endereço *inbuf*.
- ◆ O valor de entrada de *position* é a primeira posição no *buffer* de entrada ocupada pela mensagem empacotada.

# MPI\_Unpack

- ◆ A variável *position* é incrementada pelo tamanho da mensagem empacotada, de modo que o valor de saída de *position* é a primeira posição no buffer de entrada depois das posições ocupadas pela mensagem que foi desempacotada.
- ◆ O parâmetro *comm* é o comunicador utilizado para receber a mensagem empacotada.

# MPI\_Unpack

- ◆ Note a diferença entre MPI\_RECV e MPI\_UNPACK: em MPI\_RECV o argumento *count* especifica o número máximo de itens que podem ser recebidos.
- ◆ O número real de itens recebidos é determinado pelo comprimento da mensagem que chega.
- ◆ Em MPI\_UNPACK o argumento *count* especifica o número real de itens que são desempacotados. O tamanho da mensagem correspondente é o incremento em *position*.
- ◆ Note que em sistemas heterogêneos, este número não pode ser determinado a priori.

# Pack/Unpack

- ◆ Para entender o comportamento de *pack* e *unpack*, é conveniente pensar nos dados que são parte de uma mensagem como sendo uma seqüência obtida pela concatenação de valores sucessivos enviados naquela mensagem.
- ◆ A operação de *pack* armazena esta seqüência no espaço do *buffer*, como se estivéssemos enviando a mensagem para aquele *buffer*.
- ◆ A operação de *unpack* recupera esta seqüência do espaço do *buffer*, como se estivéssemos recebendo esta mensagem daquele *buffer*.
- ◆ Diversas mensagens podem ser empacotadas sucessivamente em apenas uma unidade de empacotamento.

# Pack/Unpack

- ◆ Isto é efetuado por sucessivas chamadas para `MPI_PACK`, onde a primeira chamada usa `position = 0` e cada chamada sucessiva usa o valor de *position* que foi saída da chamada anterior e os mesmos valores de *outbuf*, *outcount* e *comm*.
- ◆ Esta unidade de empacotamento agora contém a informação equivalente que seria armazenada em uma mensagem equivalente a concatenação de diversos *buffers* de várias chamadas a rotina *send*.

# Pack/Unpack

- ◆ Um unidade de empacotamento pode ser enviada usando-se o tipo de dados **MPI\_PACKED**. Qualquer rotina de comunicação coletiva ou ponto a ponto pode ser usada para transferir a seqüência de bytes que forma a unidade de empacotamento de um processo para outro.
- ◆ Esta unidade de empacotamento pode ser agora recebida usando qualquer operação de recepção, com **QUALQUER** tipo de dados: as regras para os tipos de dados são relaxadas para as mensagens enviadas com o tipo **MPI\_PACKED**.

# Pack/Unpack

- ◆ Uma mensagem enviada com qualquer tipo (incluindo **MPI\_PACKED**) pode ser recebida usando o tipo **MPI\_PACKED**. Esta mensagem pode então ser desempacotada com chamadas a **MPI\_UNPACK**.
- ◆ Uma unidade de empacotamento (ou uma mensagem criada com tipos de dados “convencionais”) pode ser desempacotada em diversas mensagens sucessivas.

# Pack/Unpack

- ◆ Isto é realizado por diversas chamadas sucessivas relacionadas a MPI\_UNPACK, onde a primeira chamada é feita com *position* = 0, e cada chamada sucessiva recebe o valor de *position* que foi fornecido como resultado na chamada anterior e os mesmos valores para *inbuf*, *insize* and *comm*.

# MPI\_Pack\_Size

- ◆ As seguintes chamadas permitem ao usuário descobrir quanto espaço é necessário para empacotar uma mensagem e, então, gerenciar a alocação de espaço para os *buffers*.

```
int MPI_Pack_size(int incount, MPI Datatype  
datatype, MPI Comm comm, int *size)
```

- ◆ ***incount***: valor de contagem de entrada (inteiro)
- ◆ ***datatype***: tipo de dados (handle)
- ◆ ***comm***: comunicador (handle)
- ◆ ***size***: limite superior no tamanho da mensagem empacotada em bytes (inteiro)

# MPI\_Pack\_Size

- ◆ Uma chamada para `MPI_PACK_SIZE` retorna em *size* um limite superior para o incremento em *position* que é efetuado por uma chamada a `MPI_PACK`
- ◆ A chamada retorna um limite superior, ao invés de um valor exato, já que o total de espaço necessário para empacotar a mensagem pode depender do contexto (p.ex., a primeira mensagem empacotada em uma unidade de empacotamento pode ocupar mais espaço).

# Pack/Unpack

```
void Get_data4( float* a_ptr , float* b_ptr,
               int*  n_ptr, int  my_rank)
{
    char buffer[100]; /* Buffer para armazenar dados */
    int  position;    /* Início dos dados no buffer */

    if (my_rank == 0){
        printf("Entre a, b e n \n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);

        /* Empacota os dados no buffer. position = 0 */
        /* indicando o início do buffer */
        position = 0;
        /* Position é entrada/saída */
    }
}
```

# Pack/Unpack

```
MPI_Pack(a_ptr, 1, MPI_FLOAT, buffer, 100, &position,  
MPI_COMM_WORLD);
```

```
/* Position foi incrementada: ela referencia a primeira  
posição livre no buffer. */
```

```
MPI_Pack(b_ptr, 1, MPI_FLOAT, buffer, 100, &position,  
MPI_COMM_WORLD);
```

```
/* Position foi incrementada novamente. */
```

```
MPI_Pack(n_ptr, 1, MPI_INT, buffer, 100, &position,  
MPI_COMM_WORLD);
```

```
/* Position foi incrementada novamente. */
```

```
/* Difunde o conteúdo do buffer */
```

```
MPI_Bcast(buffer, 100, MPI_PACKED, 0,  
MPI_COMM_WORLD);
```

# Pack/Unpack

```
} else {  
    MPI_Bcast(buffer, 100, MPI_PACKED, 0,  
MPI_COMM_WORLD);  
    /* Desempacota o conteúdo do buffer */  
    position = 0;  
    MPI_Unpack(buffer, 100, &position, a_ptr, 1, MPI_FLOAT,  
MPI_COMM_WORLD);  
    /* Position foi incrementada novamente: aponta para o  
início de b */  
    MPI_Unpack(buffer, 100, &position, b_ptr, 1, MPI_FLOAT,  
MPI_COMM_WORLD);  
    MPI_Unpack(buffer, 100, &position, n_ptr, 1, MPI_INT,  
MPI_COMM_WORLD);  
}  
} /* Get_data4 */
```



# Constructores de Grupos

# Construtores de Grupos

- ◆ Um **grupo** é um conjunto ordenado de processos. Cada processo em um grupo está associado a um único valor inteiro.
- ◆ Os valores dos *ranks* começam em 0 e vão até N-1, onde N é o número de processos no grupo.
- ◆ No MPI um grupo é acessível ao programador apenas através do seu "handle".
- ◆ Um **comunicador** é constituído por um grupo de processos que pode se comunicar entre si.
- ◆ Todas as mensagens no MPI devem especificar um comunicador.

# Construtores de Grupos

- ◆ De uma forma simples, um comunicador é um “tag” extra que deve ser incluído nas chamadas MPI.
- ◆ Também como os grupos, os comunicadores só podem ser acessados pelo programador com o uso de “handles”. Por exemplo, o *handle* para o comunicador que engloba todas as tarefas é `MPI_COMM_WORLD`.
- ◆ Do ponto de vista do programador um grupo e um comunicador são a mesma coisa, sendo que **as rotinas de manipulação de grupos são usadas primariamente para especificar os processos a serem utilizados na criação dos comunicadores.**

# Construtores de Grupos

- ◆ **O objetivo principal dos grupos e comunicadores é:**
  - ◆ **Permitir organizar as tarefas em grupos de tarefas de acordo com as suas funções.**
  - ◆ **Permitir operações de comunicação coletivas entre um subconjunto de tarefas relacionadas.**
  - ◆ **Fornecer a base para a implementação de topologias virtuais implementadas pelo usuário.**
  - ◆ **Permitir comunicações seguras.**

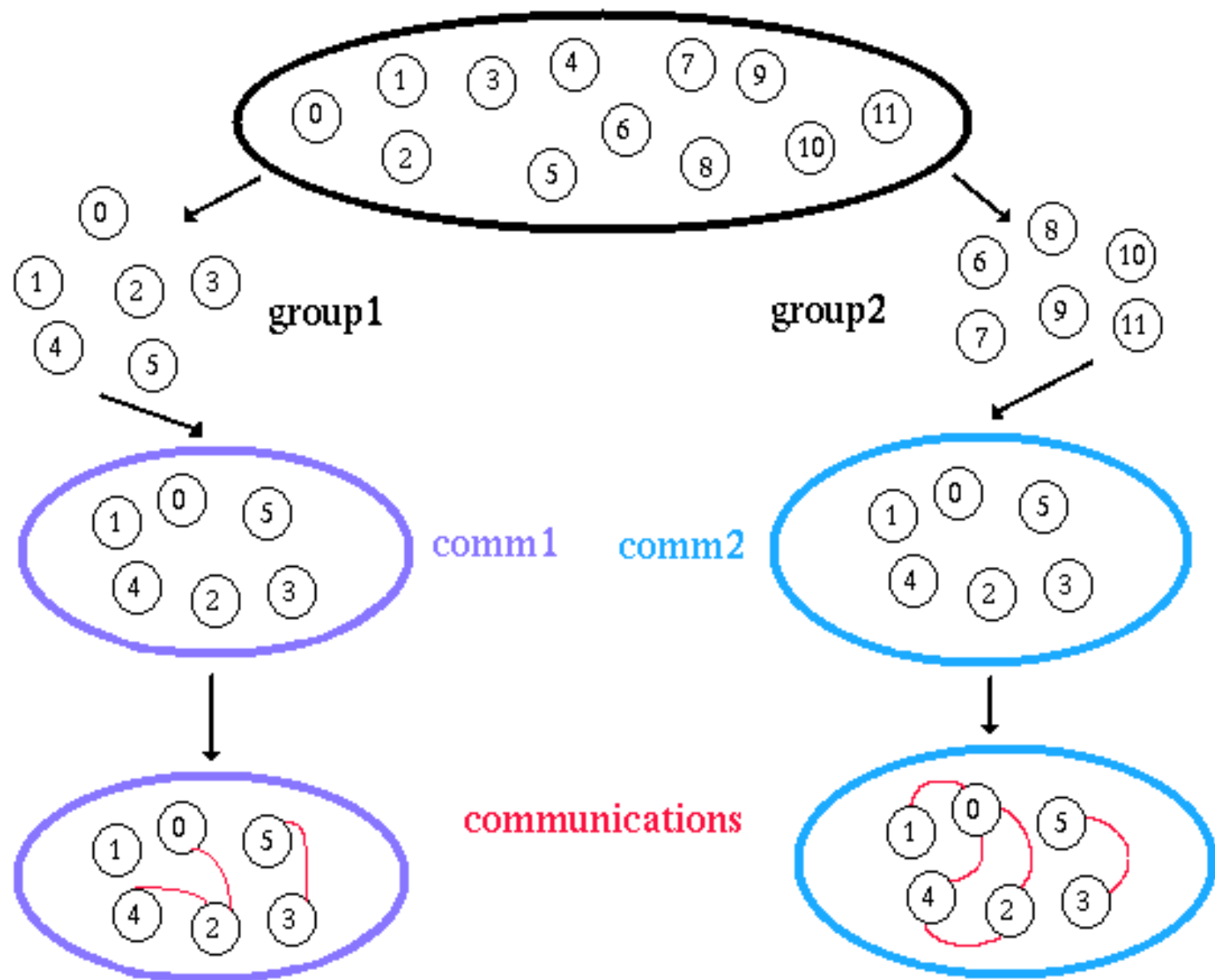
# Construtores de Grupos

- ◆ Os grupos e comunicadores são dinâmicos – eles podem ser criados e destruídos durante a execução do programa.
- ◆ Processos podem estar em mais de um grupo/comunicador. Eles terão um *rank* único e distinto em cada um dos grupos/comunicadores.
- ◆ O MPI oferece diversas rotinas relacionadas a grupos, comunicadores e topologias virtuais.

# Construtores de Grupos

- ◆ **Uso típico:**
- ◆ **Extrair o *handle* do grupo global MPI\_COMM\_WORLD usando MPI\_Comm\_group;**
- 3. Formar um novo grupo como um subconjunto do grupo global usando MPI\_Group\_incl;**
- 4. Criar um novo comunicador para um grupo novo usando MPI\_Comm\_create;**
- 5. Determinar o novo rank em um novo comunicador com MPI\_Comm\_rank;**
- 6. Realizar as comunicações usando qualquer rotina MPI de troca de mensagens;**
- 7. Quanto terminar, liberar o novo comunicador e grupo (opcionalmente) com MPI\_Comm\_free e MPI\_Group\_free.**

# MPI\_COMM\_WORLD



# Construtores de Grupos

- ◆ **Construtores de grupos são utilizados para criar subconjuntos e superconjuntos de grupos. Estes construtores constroem novos grupos a partir de grupos já existentes.**
- ◆ **São operações locais e grupos distintos podem ser definidos em processos diferentes; um processo pode também definir um grupo em que não ele não esteja incluído.**
- ◆ **O MPI não oferece um mecanismo para criar um grupo do zero, mas apenas de um outro grupo já pré-definido.**

# Construtores de Grupos

- ◆ O grupo base, a partir do qual todos os outros grupos são definidos é o grupo associado ao comunicador inicial **MPI\_COMM\_WORLD** (acessível através da função **MPI\_COMM\_GROUP**).
- ◆ Não há necessidade para um duplicador de grupos, já que um grupo, uma vez criado pode ter diversas referências a ele fazendo cópias do *handle*.
- ◆ Os construtores a seguir suprem a necessidade de superconjuntos e subconjuntos para os grupos já existentes.

# Construtores de Grupos

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

◆ A função `MPI_COMM_GROUP` retorna em *group* um *handle* para o grupo de *comm*.

- *comm*: comunicador (*handle*)
- *group*: grupo correspondente a *comm* (*handle*)

# Construtores de Grupos

```
int MPI_Group_incl(MPI_Group group, int n, int  
*ranks, MPI_Group *newgroup)
```

- ◆ A função `MPI_Group_incl` cria um grupo *newgroup* que consiste de **n** processos cujos *ranks* no grupo *group* estão definidos no vetor **ranks[0], ..., ranks[n-1]**;
- ◆ O processo com rank **i** em *newgroup* é o processo com o *rank* igual a **ranks[i]** no grupo *group*.
- ◆ Cada um dos **n** elementos do vetor **ranks** deve ser um *rank* válido em *group* e todos os elementos devem ser distintos, ou então o programa estará errado.

# Construtores de Grupos

- ◆ Se  $n=0$ , então *newgroup* é **MPI\_GROUP\_EMPTY**.
- ◆ Esta função pode, por exemplo, ser utilizada para criar um subgrupo a partir de um grupo existente ou ainda reordenar os elementos de um grupo.

# Construtores de Grupos

```
int MPI_Group_incl(MPI_Group group, int n, int  
*ranks, MPI_Group *newgroup)
```

- ***group***: grupo de origem (handle);
- ***number***: número de elementos no vetor ***ranks*** (e o tamanho de ***newgroup***) (inteiro);
- ***ranks***: vetor com os ***ranks*** dos processos em ***group*** que irão aparecer em ***newgroup*** (vetor de inteiros);
- ***newgroup***: novo grupo derivado do anterior, cujos processos terão o rank de acordo com a ordem definida em ***ranks*** (handle).

# Construtores de Grupos

```
int MPI_Group_excl(MPI_Group group, int n, int  
*ranks, MPI_Group *newgroup)
```

- ◆ A função `MPI_GROUP_EXCL` cria um grupo de processos *newgroup* que é obtido removendo-se de *group* aqueles processos com *ranks iguais a* `ranks[0]`, ..., `ranks[n-1]`.
- ◆ A ordenação de processos em *newgroup* é idêntica à ordenação em *group*. Cada um dos `n` elementos de `ranks` deve ter um rank válido em *group* e todos os elementos devem ser distintos; caso contrário, o programa está errado.
- ◆ Se `n=0`, então *newgroup* é idêntico a *group*.

# Construtores de Grupos

```
int MPI_Group_excl(MPI_Group group, int n, int  
*ranks, MPI_Group *newgroup)
```

- ***group***: grupo (handle);
- ***n***: número de elementos no vetor ***ranks*** (inteiro);
- ***ranks***: vetor de inteiros ***ranks*** pertencentes a ***group*** que não aparecem em ***newgroup***;
- ***newgroup***: novo grupo derivado do anterior; preservando a ordem definida pelo grupo ***group***. (handle).

# Construtores de Grupos

As operações de conjunto são definidas como:

- **União:** Todos os elementos do primeiro grupo (grupo1), seguidos por todos dos elementos do segundo grupo (grupo2) que não estejam no primeiro.
- **Interseção:** todos os elementos do primeiro grupo que estão também no segundo grupo, ordenado como no primeiro grupo.
- **Diferença:** todos os elementos do primeiro grupo que não estão no segundo grupo, ordenados como no primeiro grupo.

# Constructores de Grupos

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

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

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

# Construtores de Grupos

- ◆ Note que para essas operações a ordem dos processos no grupo de saída é determinada primariamente pela ordem do primeiro grupo (se possível) e então, se necessário, pela ordem no segundo grupo.
- ◆ Nem a operação de união nem a de intersecção são comutativas, mas ambas são associativas.
- ◆ O novo grupo pode estar vazio, isto é, igual a **MPI\_GROUP\_EMPTY**.

# Construtores de Grupos

◆ Outras funções de grupo são:

```
int MPI_Group_range_incl(MPI_Group group, int  
n, int ranges[ ][3], MPI_Group *newgroup)
```

```
int MPI_Group_range_excl(MPI_Group group, int  
n, int ranges[ ][3], MPI_Group *newgroup)
```

```
int MPI_Group_compare(MPI_Group  
group1, MPI_Group group2, int *result);
```

◆ Veja o manual para detalhes.

# Comunicadores

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

- ◆ ***comm***: comunicador (handle)
  - ◆ ***group***: grupo, que é um subconjunto do grupo de *comm* (handle)
  - ◆ ***comm\_out***: novo comunicador (handle)
- 
- Esta rotina cria um novo comunicador a partir dos processos contidos em ***group***.
  - A ordem dos processos no novo comunicador será a mesma ordem do grupo de origem, mas iniciando-se com o *rank* igual a 0.

# Comunicadores

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

- ◆ Esta operação coletiva faz com o que o handle passe a ter o valor `MPI_COMM_NULL`.
- ◆ Qualquer operação pendente que use este comunicador irá completar normalmente. O objeto será desalocado apenas se não houver outras referências ativas para ele.
- ◆ Esta chamada se aplica tanto a intra- com o inter-comunicadores.

# Exemplo

```
int main(argc,argv)
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); }
```

# Exemplo

```
sendbuf = rank;
```

```
/* Extract the original group handle */
```

```
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

```
/* Divide tasks into two distinct groups based upon rank */
```

```
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);
```

```
}
```

# Exemplo

```
/* Create new new communicator and then perform collective  
communications */
```

```
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();
```

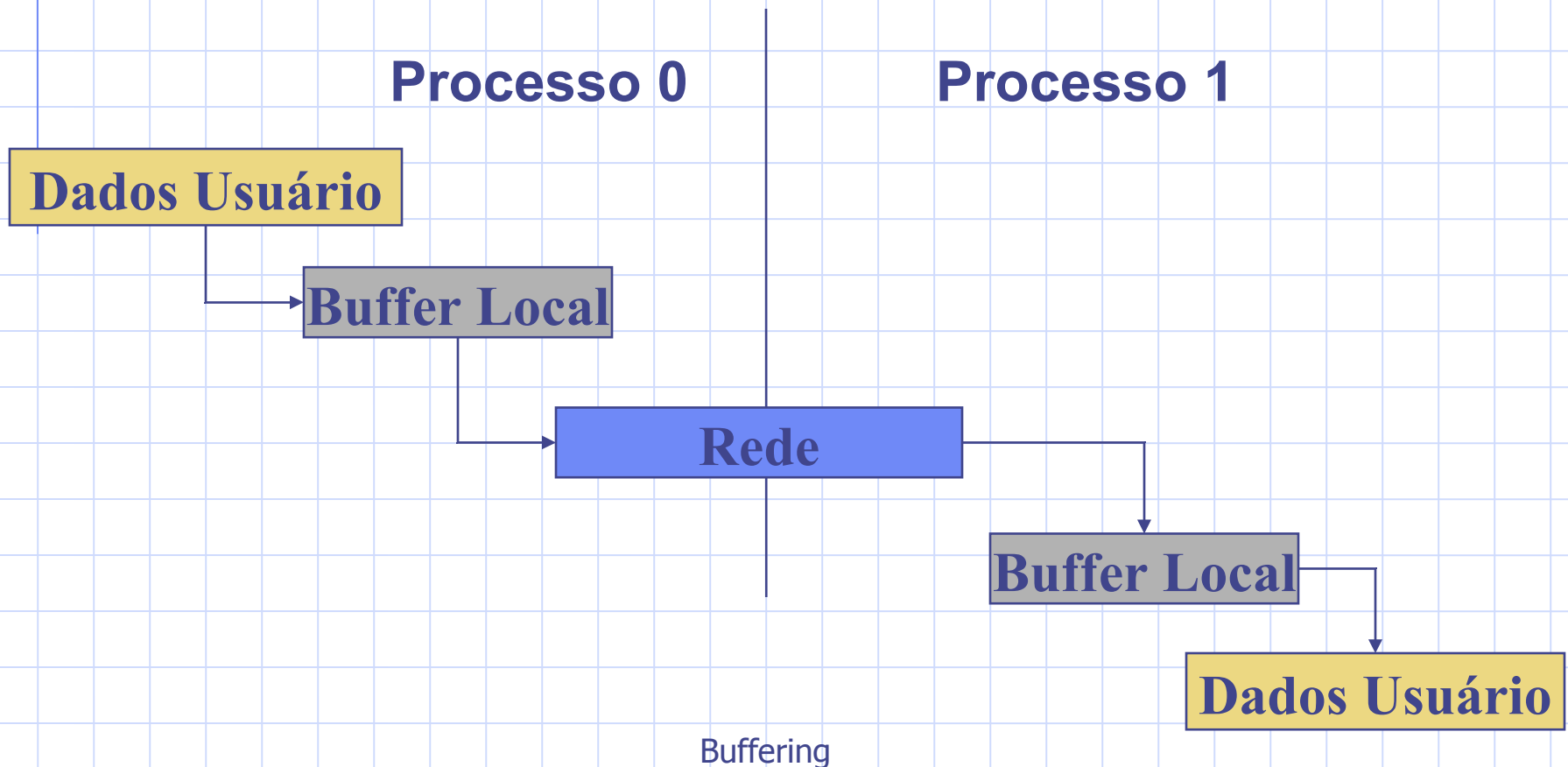
```
}
```



# **Modos de Comunicação**

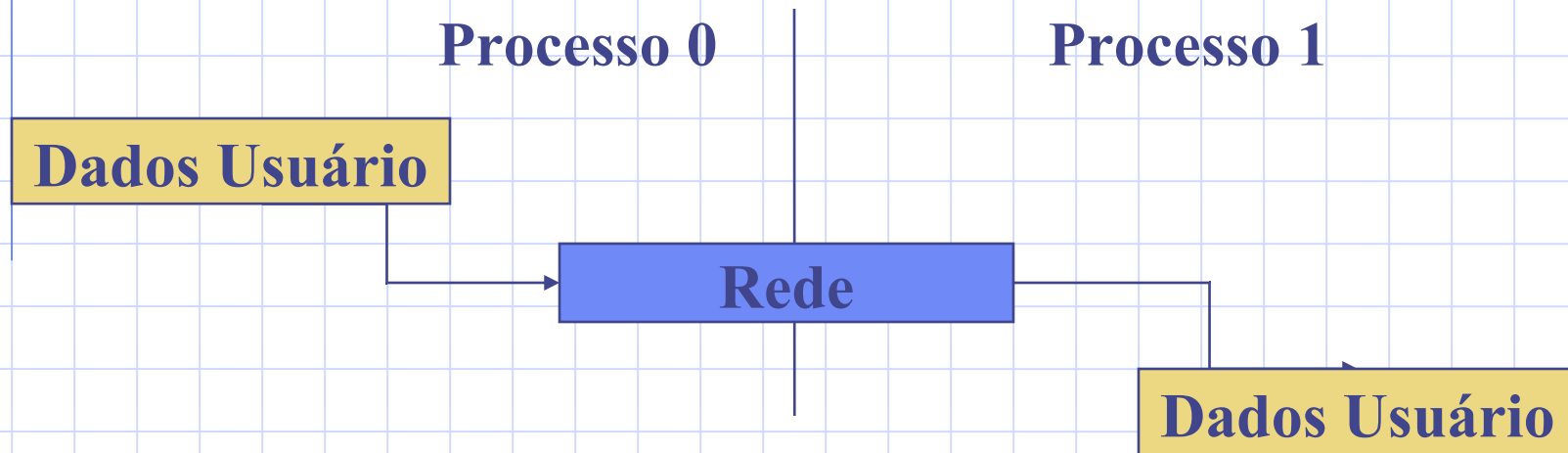
# Enviando uma Mensagem

- ◆ Quando você envia uma mensagem, para onde ela vai? Uma possibilidade é:



# Melhorando o Desempenho

◆ É melhor evitar cópias:



- Isto exige modificações na forma de enviar e receber as mensagens para que a operação possa completar com sucesso.

# Comunicação Bloqueante

- ◆ **No modo de comunicação bloqueante:**
  - O `MPI_Recv` não completa até que o *buffer* de recepção esteja cheio (mensagem disponível para uso).
  - O `MPI_Send` não completa até que o *buffer* de envio esteja vazio (*buffer* disponível para reuso).
- ◆ **O sucesso da operação de comunicação depende do tamanho da mensagem e do tamanho do buffer do sistema.**

# Modo Não-Bloqueante

- ◆ Caso **não** estejamos utilizando *buffers*, devemos utilizar as rotinas de envio/recepção no modo não-bloqueante, para garantir que não haverá “deadlock”.
- ◆ No modo bloqueante, o programa pára até que o *buffer* de mensagem possa ser utilizado com segurança.
- ◆ No modo não-bloqueante a computação prossegue e, quando for necessário, verificamos se a operação já terminou ou não.

# Rotinas de Comunicação

- ◆ Rotinas não-bloqueantes separam comunicação da computação.

<b>Modo</b>	<b>Rotinas</b>	<b>Rotinas</b>
<b>Comunicação</b>	<b>Bloqueantes</b>	<b>Não-Bloqueantes</b>
<b>Síncrono</b>	<b>MPI_Ssend</b>	<b>MPI_ISSend</b>
<b>Pronto</b>	<b>MPI_Rsend</b>	<b>MPI_IRsend</b>
<b>Bufferizado</b>	<b>MPI_Bsend</b>	<b>MPI_IBsend</b>
<b>Padrão</b>	<b>MPI_Send</b>	<b>MPI_Isend</b>
	<b>MPI_Recv</b>	<b>MPI_Irecv</b>

# Operações Não-Bloqueantes

- ◆ A única diferença nas operações não-bloqueantes é que estas retornam (imediatamente) um “handle request”.
- ◆ Este *handle* pode ser testado ou usado para ficar-se em espera pela chegada da mensagem.
- ◆ Sendo assim, se a programação for feita de maneira adequada, podemos realizar computação e comunicação em paralelo, melhorando o desempenho final do programa.

# Operações Não\_Bloqueantes

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

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

# Esperando a Mensagem

## ◆ Esperando a mensagem chegar:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

## ◆ Você também pode testar sem esperar:

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

# Múltipla Espera

- ◆ Algumas vezes é desejável esperar por múltiplos “requests”:

```
int MPI_Waitall(int count, MPI_Request  
*array_of_requests, MPI_Status *array_of_statuses)
```

```
int MPI_Waitany(int count, MPI_Request  
*array_of_requests, int *index, MPI_Status *status)
```

```
int MPI_Waitsome(int incount, MPI_Request  
*array_of_requests, int *outcount, int  
*array_of_indices, MPI_Status *array_of_statuses)
```

- ◆ Existem versões correspondentes de **test** para cada uma dessas acima.

# Exemplo

◆ Exemplo de programa com operações de envio e recepção não-bloqueantes:

<http://equipe.nce.ufrj.br/gabriel/progpar/nonblock.htm>

# Modos de Comunicação

- ◆ O MPI define quatro modos de comunicação:
  - Modo síncrono (o mais seguro)
  - Modo pronto (menor sobrecarga para o sistema)
  - Modo "bufferizado" (desacopla o emissor do receptor)
  - Modo padrão (solução de compromisso)
- ◆ O modo de comunicação é selecionado de acordo com a rotina de **envio** utilizada.

# Modos de Comunicação

**int MPI\_Bsend(void\* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)**

**int MPI\_Ssend(void\* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)**

**int MPI\_Rsend(void\* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)**

# Modos de Comunicação

- ◆ Para uma dada rotina de envio de mensagem, diz-se que foi **postada** uma operação de recepção correspondente, quando um processo qualquer iniciar uma rotina de recepção com comunicador e etiqueta ("tag") que satisfaçam ao comunicador e etiqueta utilizados pela rotina de envio.
- ◆ Note que é permitido o uso de coringas nas operações de recepção tanto para o remetente (**MPI\_ANY\_SOURCE**) quanto para a etiqueta (**MPI\_ANY\_TAG**), que devem ser considerados nesse caso.
- ◆ **MPI\_Recv** recebe mensagens enviadas em qualquer modo.

# Modo Bufferizado

- ◆ A operação de envio pode ser iniciada havendo ou não uma operação de recepção correspondente iniciada. A operação de envio poderá completar antes de uma recepção correspondente ter sido postada.
- ◆ Existe a necessidade do uso de funções adicionais para alocação e liberação do espaço para armazenamento das mensagens.
- ◆ É função do usuário, e não do sistema, gerenciar a alocação dos *buffers*.
- ◆ É garantido que as operações de envio e recepção **não** são sincronizadas.

# Modos Bufferizado

```
int MPI_Buffer_attach (void *buffer, int  
buffer_size);
```

- ◆ Só pode haver um buffer ativo por vez.
- ◆ O total de espaço alocado deve ser suficiente para garantir o funcionamento correto do programa.

```
int MPI_Buffer_detach(void *buffer_address,  
int * size_ptr);
```

- ◆ Esta rotina retorna um ponteiro para o buffer previamente alocado e um ponteiro para o seu tamanho.
- ◆ O espaço alocado **não** é liberado.

# Modos Bufferizado

```
char    buffer[MAX_BUF];
int     buffer_size = MAX_BUF;
...
MPI_Buffer_attach(buffer, buffer_size);

for (i = 0; i < p - 1; i++) {
    send_offset = ((my_rank - i + p) % p)*blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p)*blocksize;
    MPI_Bsend(y + send_offset, blocksize, MPI_FLOAT,
successor, 0, ring_comm);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
predecessor, 0, ring_comm, &status); }

MPI_Buffer_detach(&buffer, &buffer_size);
```

# Modo Síncrono

- ◆ A rotina de envio pode ser iniciada havendo ou não uma recepção correspondente.
- ◆ Contudo, o envio irá completar com sucesso apenas quando uma recepção correspondente tiver sido postada e a recepção da mensagem enviada pela rotina de envio síncrona for iniciada pela operação de recepção.
- ◆ Este modo não requer o uso de bufferização do sistema.
- ◆ Pode-se assegurar que o nosso programa está **seguro** se executar corretamente utilizando apenas rotinas de envio no modo síncrono.

# Modo Síncrono

```
for (i = 0; i < p - 1; i++) {  
    ...  
    if ((my_rank % 2) == 0){ /* Even ranks send first */  
        MPI_Ssend(y + send_offset, blocksize, MPI_FLOAT,  
            successor, 0, ring_comm);  
        MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,  
            predecessor, 0, ring_comm, &status);  
    } else { /* Odd ranks receive first */  
        MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,  
            predecessor, 0, ring_comm, &status);  
        MPI_Ssend(y + send_offset, blocksize, MPI_FLOAT,  
            successor, 0, ring_comm);  
    }  
}
```

# Modo Pronto

- ◆ O envio pode ser iniciado **apenas** se houver uma rotina de recepção correspondente já iniciada. Caso isto não ocorra, o programa terminará com **erro**.
- ◆ Embora seja esperado, não é garantido que a implementação das rotinas em modo pronto seja mais eficiente que a de modo padrão.
- ◆ É necessário o uso de funções de sincronização (p.ex. barreiras) para garantir que a recepção em um processo é postada antes da recepção no outro.
- ◆ Este é o modo mais difícil de programar e só deve ser usado quando o desempenho for importante.

# Modo Pronto

```
MPI_Request request [p-1];
for (i = 0; i < p - 1; i++) {
    recv_offset =
        ((my_rank - i - 1 + p) % p)*blocksize;
    MPI_Irecv(y + recv_offset, blocksize, MPI_FLOAT,
        predecessor, i, ring_comm, &(amp;request[i])); }
MPI_Barrier(ring_comm);
for (i = 0; i < p - 1; i++) {
    send_offset = ((my_rank - i + p) % p)*blocksize;
    MPI_Rsend(y + send_offset, blocksize, MPI_FLOAT,
        successor, i, ring_comm);
    MPI_Wait(&(request[i]), &status); }
```

# Modo Padrão

- ◆ Neste modo o MPI decide se as mensagens enviadas serão *bufferizadas* ou enviadas em modo síncrono.
- ◆ Uma rotina de envio no modo padrão pode ser iniciada havendo ou não uma rotina de recepção correspondente postada.
- ◆ Não se pode assumir que a operação de envio irá terminar antes ou depois da recepção correspondente ser iniciada.
- ◆ Como consequência, pode ser que se programa funcione bem em um sistema e em outro não, caso o programa não esteja programado de modo **seguro**.

# Modo Padrão

```
successor = (my_rank + 1) % p;
predecessor = (my_rank - 1 + p) % p;

for (i = 0; i < p - 1; i++) {
    send_offset = ((my_rank - i + p) % p)*blocksize;
    recv_offset =
        ((my_rank - i - 1 + p) % p)*blocksize;
    MPI_Send(y + send_offset, blocksize, MPI_FLOAT,
            successor, 0, ring_comm);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
            predecessor, 0, ring_comm, &status);
}
```

# Modos de Comunicação

## Vantagens

## Desvantagens

Síncrono

Mais seguro e, portanto, mais portátil. A ordem SEND/RECV não é crítica. Total de espaço gasto é irrelevante.

Pode haver substancial *overhead* de sincronização.

Pronto

O overhead total é o mais baixo. RECV deve preceder o SEND.  
O protocolo SEND/RECV não é necessário.

Bufferizado

Desacopla o SEND do RECV. A Overhead adicional do sistema para copiar o buffer.  
ordem SEND/RECV é irrelevante. O programador pode controlar o tamanho do buffer.

Padrão

Bom em muitos casos.

Seu programa pode não ser adequado.

# Fontes de Deadlock

- ◆ Envie uma mensagem grande do processo 0 para o processo 1
  - Se o espaço de armazenamento no destino for insuficiente, a rotina de envio deve esperar até que o usuário providencie espaço de memória suficiente (através da chamada de uma rotina de recepção).
- ◆ O que acontece com este código?

**Processo 0**

**Processo 1**

---

MPI\_Send(1)

MPI\_Send(0)

MPI\_Recv(1)

MPI\_Recv(0)

- Isto é chamado de "inseguro" porque depende da disponibilidade dos *buffers* de sistema.

# Algumas soluções

- ◆ Ordenar as operações de envio e recepção adequadamente:

**Processo 0**

**Processo 1**

---

MPI\_Send(1)

MPI\_Recv(0)

MPI\_Recv(1)

MPI\_Send(0)

- Fornecer um buffer de recepção ao mesmo tempo que envia a mensagem:

**Processo 0**

**Processo 1**

---

MPI\_Sendrecv(1)

MPI\_Sendrecv(0)

# Mais Soluções

- ◆ O usuário fornece explicitamente um *buffer* para envio:

**Processo 0**

**Processo 1**

---

MPI\_BSend(1)

MPI\_BSend(0)

MPI\_Recv(1)

MPI\_Recv(0)

- Uso de operações não-bloqueantes:

**Processo 0**

**Processo 1**

---

MPI\_Isend(1)

MPI\_Isend(0)

MPI\_Irecv(1)

MPI\_Irecv(0)

MPI\_Waitall

MPI\_Waitall

# MPI\_Sendrecv

- ◆ Permite envio e recepção simultâneos.
- ◆ Os tipos de dados de envio e recepção podem ser diferentes.
- ◆ Pode-se usar MPI\_Sendrecv com um MPI\_Recv ou MPI\_Send comuns (ou MPI\_Irecv, MPI\_Ssend, etc.)

**Processo 0**

**Processo 1**

---

MPI\_Sendrecv(1)

MPI\_Sendrecv(0)



# **MPICH2**

# MPI-2

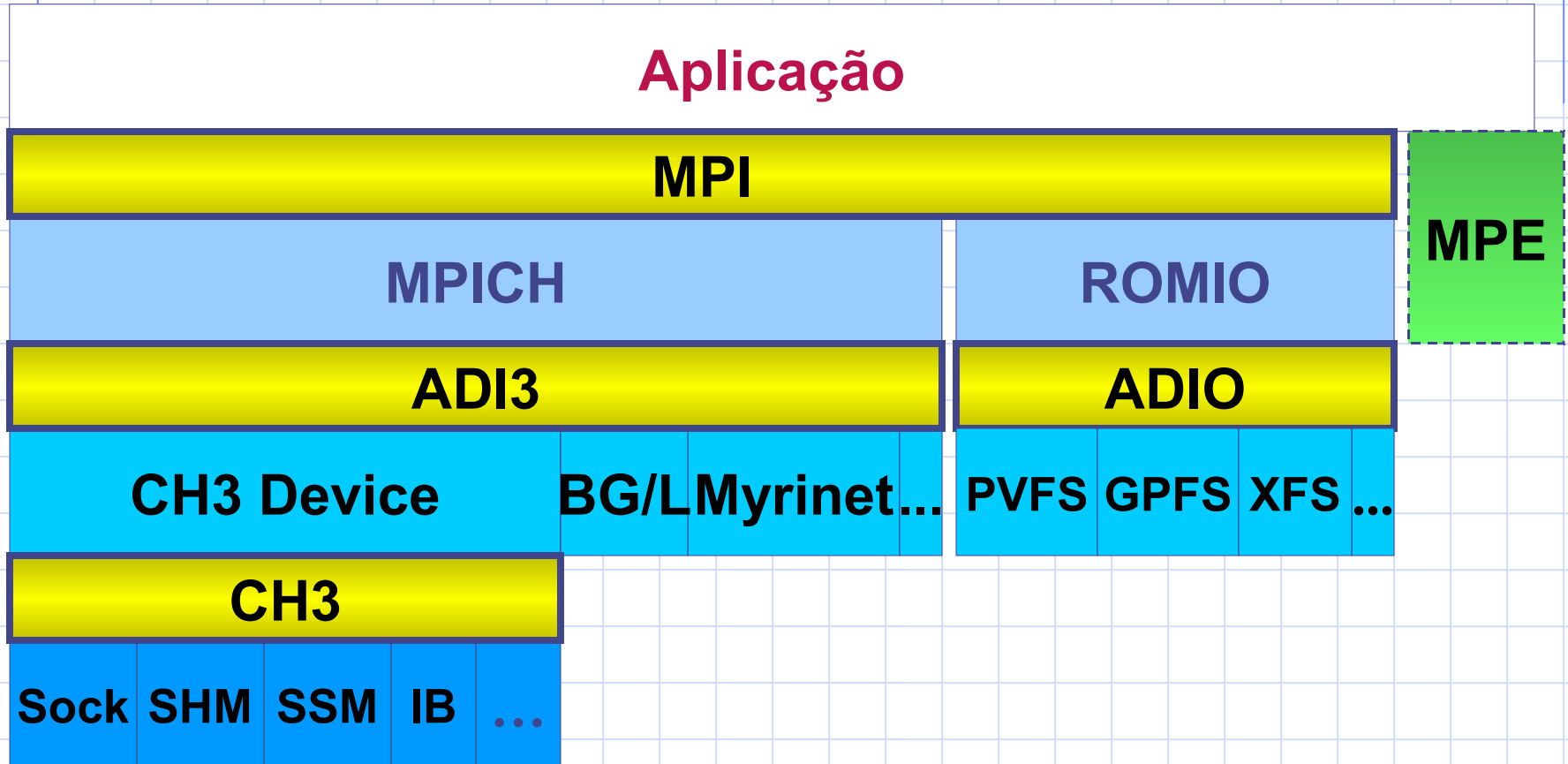
## ◆ Gerenciamento Dinâmico de Processos

- **Facilidade fornecida por duas novas classes de funções:**
  - ◆ **Spawning:** criação de novos conjuntos de processos
  - ◆ **Connecting:** estabelecimento de comunicação entre processos que foram disparados separadamente
- **Usam o conceito de “intercomunicadores”, comunicadores contendo dois grupos de processos ao invés de apenas um.**

## ◆ Suporte a E/S Paralela

- **Mais funcionalidade do que a provida pelas funções de E/S do Unix.**

# Estrutura do MPICH2



# Interfaces

- ◆ **MPI – “Message Passing Interface”**
  - Especificação MPI como definida pelo Fórum MPI.
- ◆ **ADI3 – “Abstract Device Interface”**
  - Especificação de um subconjunto do MPI, que provê a funcionalidade para realizar a comunicação.
- ◆ **CH3 – “Channel interface”**
  - Especificação que simplifica substancialmente a interface de comunicação, constituída aproximadamente por 20 funções.
- ◆ **ADIO – “Abstract Device Interface for I/O”**

# Implementações

## ◆ MPICH

- Implementa a comunicação nos termos definidos pela especificação ADI3.

## ◆ MPE: Extensões úteis ao MPI:

- Biblioteca de "profiling".
- Ferramentas de visualização
- Interface gráfica.

# Gerenciamento de Processos

- ◆ O MPICH2 separa o gerenciamento de processo da comunicação;
- ◆ O gerenciador de processos é responsável por:
  - Iniciar novos processos quando *mpiexec* é executado ou *MPI\_Comm\_spawn()* é chamada;
  - Informar aos processos a respeito de seu grupo:
    - ◆ *rank*, tamanho, identificação.
  - Fornecendo uma base de dados simples para registro e obtenção de informações sobre processos;
  - Sincronização (barreira) de processos de um mesmo grupo.

# Gerenciamento de Processos

## ◆ MPD – Multi Purpose Daemon (version 2)

- MPD executa em cada nó, conectando-se aos demais MPDs através de um anel;
- Implementado com Python;
- Só funciona nas plataformas UNIX .

## ◆ WinMPD – Microsoft Windows MPD

- Implementação Windows do MPD da versão MPICH1.

## ◆ SMPD – Super MPD

- Executa tanto em UNIX como Microsoft Windows;
- Permite que as tarefas executem nos dois ambientes.
- Substituirá o WinMPD.

# Gerenciamento de Processos

## ◆ remshell

- Usa shell remoto (rsh/ssh) para iniciar os processos;

## ◆ forker

- Quando todos os processos executam em uma única máquina;
- Ideal para SMP ou testes.

# Construindo o MPICH2

## ◆ Basicamente:

- Pegar a última versão do mpich2 e executar os seguintes comandos:

```
% tar -zxvf mpich2.tar.gz
```

```
% cd mpich2
```

```
% ./configure
```

```
% make
```

```
% make install
```

## ◆ Algumas opções disponíveis:

- Escolher o gerenciador de processos.
- Definir o diretório de instalação do MPICH2.
- Veja o arquivo README.

# Usando o MPICH2

- ◆ **Assumimos que o programa será executado em uma rede heterogênea de computadores com UNIX.**
- ◆ **Os programas executáveis, bibliotecas e arquivos de cabeçalho supõem-se que estejam instalados em um diretório público nas máquinas em que você está executando e compilando o seu programa.**
- ◆ **Nos exemplos a seguir assumimos que os arquivos MPICH2 são armazenados nos seguintes arquivos:**
  - Executáveis: /usr/local/bin**
  - Bibliotecas: /usr/local/lib**
  - Arquivos de Cabeçalho: /usr/local/include**

# Usando o MPICH2

- ◆ Você deve garantir que o diretório de executáveis esteja no seu PATH:

```
export PATH=/usr/local/bin:$PATH      ou  
setenv PATH /usr/local/bin:$PATH
```

- ◆ Você deve colocar um arquivo de nome **.mpd.conf** em seu diretório HOME contendo a linha:

```
secretword=<secretword>
```

- ◆ Onde **<secretword>** é uma cadeia de caracteres conhecida apenas por você e não deve ser a sua senha UNIX. Faça este arquivo com leitura e escrita apenas por você.

# Usando o MPICH2

◆ Você pode criá-lo usando o seguinte comando:

```
% cd $HOME
```

```
% touch .mpd.conf
```

```
% chmod 600 .mpd.conf
```

```
% echo "secretword=mr45-j9z" >> .mpd.conf
```

◆ Claro, use uma senha diferente de mr45-j9z

# Usando o MPICH2

- ◆ Você também deverá criar um arquivo de nome **mpd.hosts** com o nome de cada máquina que você vai estar utilizando, um em cada linha diferente.
- ◆ Para disparar o *daemon* que irá gerenciar os processos mpi em cada máquina (ou parte delas) execute o comando:

```
% mpdboot -n <número de máquinas>
```

# Usando o MPICH2

- ◆ **Você poderá verificar se o processo foi bem sucedido dando o comando:**

```
% mpdtrace
```

- ◆ **Para testar o conjunto de máquinas que você criou, você poderá verificar o tempo que uma mensagem leva para circular em todas elas digitando:**

```
% mpdringtest 100
```

- ◆ **Onde 100 é o número de vezes que a mensagem circulou por todas as máquinas.**

# Usando o MPICH2

◆ Para compilar um arquivo fonte **prog.c**, digite:

```
% mpicc -o prog prog.c
```

◆ Para executar o programa com, digamos, 4 processos, você deve copiar o executável para o seu diretório \$HOME em cada máquina e digitar:

```
% mpirun -n 4 prog
```

◆ A cópia é desnecessária se o diretório estiver montado remotamente (NFS).

# Usando o MPICH2

- ◆ O comando **mpiexec** permite opções mais elaboradas:

```
%mpiexec -n 1 -host paraty : -n 19 slave
```

- ◆ Dispara o processo com *rank 0* na máquina **paraty** e outros **19** divididos entre as demais máquinas.

- ◆ Para saber mais opções digite:

```
%mpiexec -help
```

- ◆ Para terminar a execução de todos os daemons digite:

```
%mpdallexit
```

# Instalação do MPD Multiusuário

- ◆ **Instalar com setuid root, permitindo executar programas de diferentes usuários.**
- ◆ **Geralmente iniciado de /etc/rc.d ou /etc/init.d**
- ◆ **Arquivo de configuração é /etc/mpd.conf**
  - **Contém chave compartilhada, justamente como o arquivo de usuário ~/.mpd.conf**
  - **Também pode conter o host/porta de um daemon mpd que já faz parte do anel.**

# Referências

- ◆ [1] Neil MacDonald et alli, Writing Message Passing Programs with MPI, Edinburgh Parallel Computer Centre
- ◆ [2] Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, 2nd ed., Englewood Cliffs, NJ, Prentice--Hall, 1988.
- ◆ [3] Peter S. Pacheco, Parallel Programming with MPI, Morgan Kaufman Pub, 1997.
- ◆ [4] Message Passing Interface Forum, MPI: A Message Passing Interface Standard , International Journal of Supercomputer Applications, vol. 8, n. 3/4, 1994.