Modelo para Memória Distribuída

- Modelo de Troca de Mensagens:
  - Processos distintos cooperam na execução de uma computação
  - Cada processo tem o seu espaço de endereçamento
    - Pode ser fisicamente um sistema distribuído, mas não necessariamente
  - Se necessário, um processo envia seus dados ao outro, que os recebe (troca de mensagens)
  - Envio e Recepção são programados explicitamente
  - Os processos podem estar em Sistemas Operacionais distintos, mas não necessariamente

Padrão MPI (*Message Passing Interface*)

- Implementa o modelo de troca de mensagens por chamadas a biblioteca de procedimentos padronizada
- Padrão *de facto*
  - www.mpi-forum.org
- Construído pela necessidade de portabilidade de programas, em um cenário com múltiplas implementações não portáteis e não padronizadas de bibliotecas de troca de mensagens
  - PVM, PARMACS, etc...
- Esforço conjunto de grupos de usuários, acadêmicos e indústria (1992-1998, 2008-2009)

Padrão MPI

- Primeira implementação feita em conjunto com o padrão
  - MPICH, no *Argonne National Labs*
  - Software livre, constantemente suportado
  - Portabilidade para muitas máquinas

- Atualmente, muitas outras implementações
  - LAM (MSU)
  - SCALI MPI
  - Intel MPI
  - PGI MPI

Padrão MPI (cont.)

- Há dois padrões MPI:
  - MPI-1 (1.1, junho de 1995)
  - MPI-2 (2.0, julho de 1997)
- Implementações do padrão 2.0 são recentes; a maioria das implementações é do padrão 1.1
- MPI 2.0 é MPI 1.1 com adições
- Apresentação atual atem-se ao padrão MPI 1.1
  - Mais comumente encontrado

Padrão MPI (cont.)

- Descreve a semântica de cada operação de uma biblioteca de procedimentos
- Semântica para interfaces C e Fortran
- Procedimentos em C são funções que retornam código de erro
- Procedimentos em Fortran são sub-rotinas que retorna código de erro no último argumento

Funções do padrão MPI-I

MPI Function Index

Porque o sucesso de MPI

- Necessidade dos usuários
  - Única forma padronizada de programar máquinas MIMD de memória distribuída, cada vez mais populares (clusters de PCs)
- Interesse da indústria
- Disponibilidade de implementações
  - Atualmente, muitas outras implementações, tanto software livre quanto software proprietário

Fontes de consulta MPI

- Padrão MPI:
  - http://www.mpi-forum.org/docs/docs.html
- MPI-1:
  - William Gropp, Ewing Lusk and Antony Skjellum: "Using MPI: Portable Parallel Programming with the Message Passing Interface", 2nd edition, MIT Press, 1999.
  - Pacheco, P.: "Parallel Programming with MPI", Morgan Kaufmann, 1999.
- MPI-2:
  - William Gropp, Ewing Lusk and Rajeev Thakur: "Using MPI-2: Advanced Features of the Message Passing Interface", MIT Press, 1999.

Compilação de programas MPI

- Padrão MPI não define como compilar e executar programas MPI (depende do Sistema Operacional)

- Para compilar programa MPI no LINUX:
  - Use *mpif90* ou *mpicc* (*script* que invoca o compilador Fotran 90 ou o compilador C/C++ e chaves MPI, o qual faz parte da instalação da biblioteca)
  - Permite as mesmas chaves de compilação do compilador usado

- Em procedimentos que usam MPI, inclua a linha
  - FORTRAN: *include "mpif.h"*
  - C: *#include "mpi.h"*

Execução de programas MPI no LINUX

- Enviar para execução:
  - *mpirun –np <numero de processos> <executável>*
- Chaves para atribuição de processos a processadores:
  - *-machinefile <arquivo>*
    - <*arquivo*> lista os processadores a utilizar, um por linha
      - Processo 0: processador que dispara a execução
      - Demais processos: um para cada processador em <arquivo>
      - Mais processos que processadores: recomeça <arquivo>, *round-robin*
  - *-nolocal*
    - não utiliza processador que dispara a execução

Manejo do ambiente MPI

- Inicio:
  - C: `ierr = MPI_Init(&argc, &argv);`
    - `int ierr`
  - Fortran: `MPI_INIT (ierr)`
    - `integer, intent(out) :: ierr`

- Inscreve o processo na computação MPI

- Deve ser executado antes de qualquer outro procedimento MPI
  - Barreira até que todos os processos estejam inscritos
  - Código de retorno:
    - A interface C/C++ e Fortran de qualquer procedimento MPI retorna `ierr`
      - Retorna `ierr=MPI_SUCCESS` se execução correta

Manejo do ambiente MPI (cont)

- Término ordenado:
  - C: `ierr = MPI_Finalize();`
  - Fortran: `MPI_FINALIZE (ierr)`

- Ultimo procedimento MPI a invocar

- Nem mesmo MPI_INIT pode ser usado após a execução de MPI_FINALIZE
  - Tipicamente, último comando do programa

Manejo do ambiente MPI (cont)

- Término desordenado (erro):
  - C: `ierr = MPI_Abort(comm, cod_erro);`
  - Fortran: `MPI_ABORT (comm, cod_erro, ierr)`
    - `integer, intent(in) :: comm ! comunicador`
    - `integer, intent(in) :: cod_erro`
    - `integer, intent(out) :: ierr`
  - `cod_erro` é código enviado ao sistema operacional

- Não garante saída ordenada de MPI

- "*Best attempt*" para abortar todos os processos do comunicador (tipicamente, *MPI_COMM_WORLD*)

Comunicador

- MPI implementa o conceito de comunicador
  - `ierr = MPI_ABORT (MPI_COMM_WORLD, cod_erro);`
- Um comunicador é um conjunto ordenado de *n* processos, enumerados de *0* a *n-1* (com *n* >= *0*)
- O comunicador cria um contexto (conjunto de processos) no qual ocorrem comunicações
  - O comunicador enumera os processos, permitindo sua identificação e a gerência das mensagens entre processos
- Podem haver múltiplos comunicadores no mesmo programa
  - Comunicações em um comunicador são independentes das comunicações em outro comunicador
- Comunicador pré-definido (*MPI_COMM_WORLD*) com todos os processos da computação no Linux, enumerado por opções do *mpirun*

Comunicadores, “handles” e objetos opacos

- A implementação MPI esconde do usuário a implementação de comunicadores.
- Entretanto, a MPI permite que usuários criem e usem novos comunicadores.
  - Array de comunicadores (típico, mas não obrigatório)
- Dessa forma:
  - A implementação é escondida (objeto opaco)
  - Objetos podem ser referenciados por um tipo específico (*handle*)

Quantos processos na computação MPI?

- `ierr = MPI_Comm_size(comm, &size);`
  - Retorna size, o número de processos MPI no comunicador

- Tipicamente:
  - `ierr = MPI_Comm_size(MPI_COMM_WORLD, size);`

Qual processo eu sou?

- `ierr = MPI_Comm_rank(comm, &id);`
  - Retorna *id*, inteiro de *0* à *size-1*, a identidade deste processo no comunicador

- Tipicamente:
  - `ierr = MPI_Comm_rank(MPI_COMM_WORLD, id);`

Executo em qual processador?

- `ierr = MPI_GET_PROCESSOR_NAME(name, &length);`
  - `char name[MPI_MAX_PROCESSOR_NAME];`
  - Retorna *name*, *character* de comprimento *length*, contendo o nome do processador onde o processo está executando

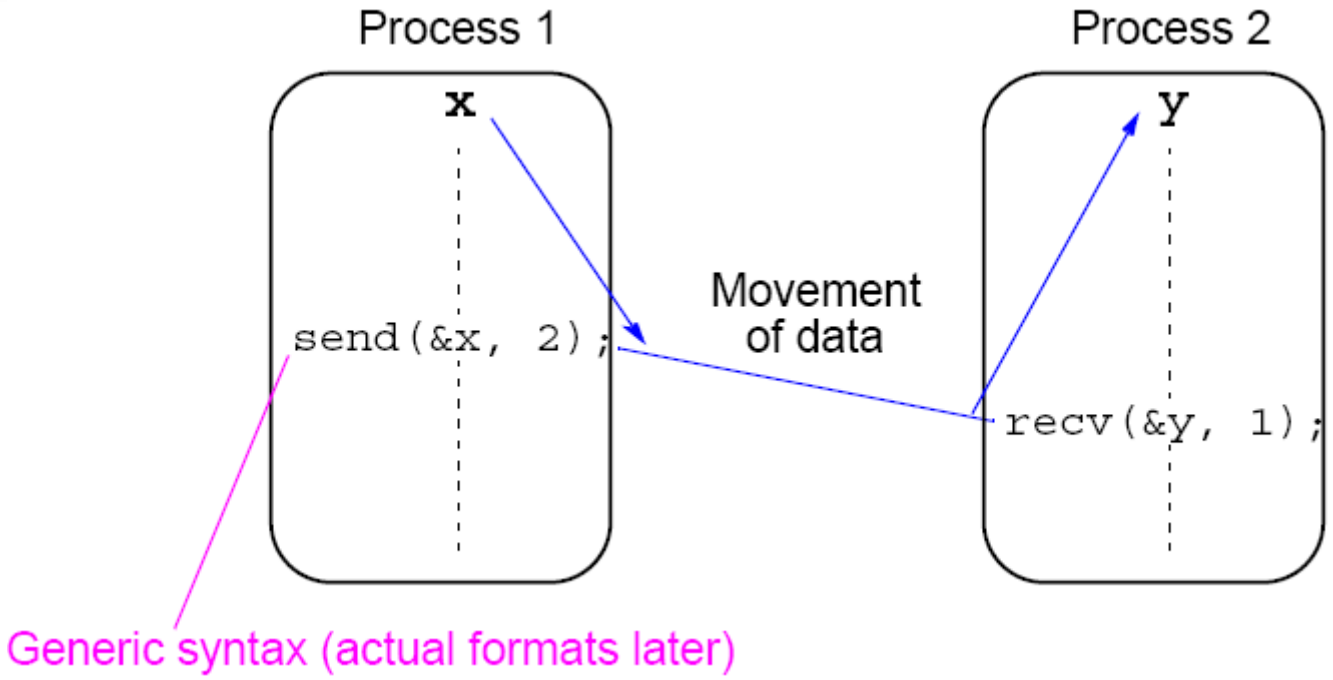
- O comando é independente do comunicador
  - um processo executa em um único processador para qualquer comunicador

Hello (C)

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

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

    int processId; /* rank dos processos */
    int noProcesses; /* Número de processos */
    int nameSize; /* Tamanho do nome */
    char computerName[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &noProcesses);
    MPI_Comm_rank(MPI_COMM_WORLD, &processId);
    MPI_Get_processor_name(computerName, &nameSize);
    printf("Hello from process %d/%d on %s\n", processId,
          noProcesses, computerName);

    MPI_Finalize( );
    return 0; }
```

Ponto-a-ponto (point-to-point) Rotinas: Send e Receive

- Processos trocam mensagem com dados
- Um processo solicita o envio de uma mensagem para outro processo (MPI_SEND)
- Outro processo solicita a recepção de uma mensagem do primeiro (MPI_RECV)



Mensagem composta por: Envelope + Dados

- Envelope da mensagem:
  - Comunicador MPI
  - Identidade (no comunicador MPI) do processo que envia;
  - Identidade (no comunicador MPI) do processo que recebe;
  - *Tag* (inteiro identificador da mensagem)

Mensagem composta por: Envelope + Dados (cont.)

- Dados:
  - Os valores:
    - Endereço da primeira posição de *buffer* na memória
      - O *buffer* é o espaço de posições contíguas de memória que contém a seqüência de valores a enviar, ou o espaço para armazenar os valores recebidos
  - O tamanho de *buffer*:
    - Quantos valores do tipo MPI (não quantos *bytes*)
      - Inteiro não negativo, permitindo mensagens vazias
  - O tipo MPI dos valores comunicados:
    - Mesmo tipo para todos os valores

Tipos de dados MPI em C

MPI datatype	C datatype
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	

Modos de Comunicação Ponto a Ponto

- Os dois principais modos de comunicação ponto a ponto de MPI são:
  - Comunicação síncrona (*synchronous*)
  - Comunicação assíncrona (*buffered*)

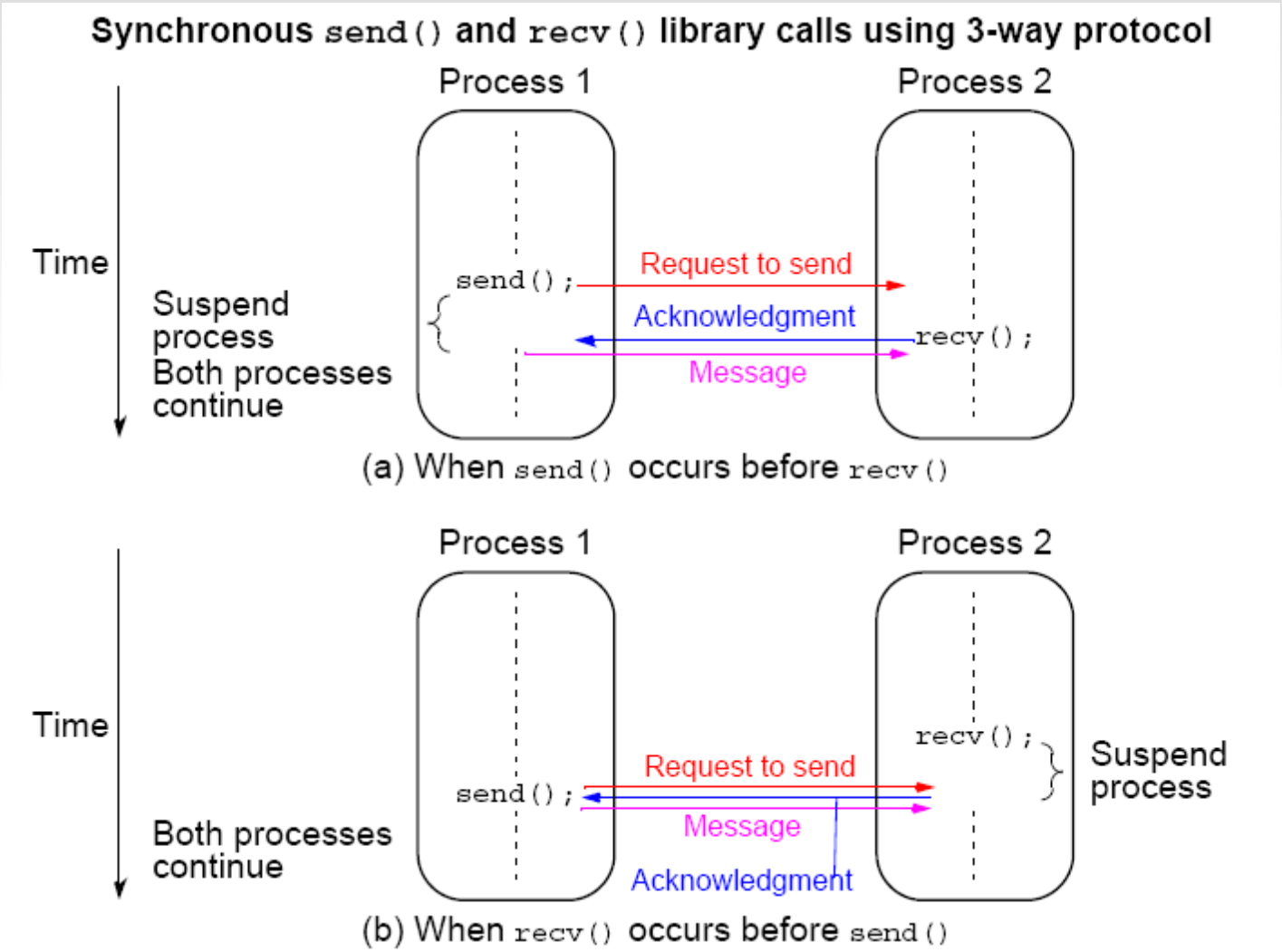
- Os dois modos diferem em:
  - Obrigatoriedade do par SEND-RECV ter sido emitido (sincronização ou não)
  - O que significa a invocação a SEND ou o RECV

Comunicação Síncrona

- MPI copia os dados (e envia a mensagem) diretamente para o *buffer* do receptor
  - SEND termina apenas quando o receptor recebe a mensagem (e avisa o emissor)
  - RECV termina apenas quando os dados estão no próprio *buffer*

- Características:
  - Possibilita sincronização, pois termina após a recepção correspondente terminar;
  - Usa pouca memória (não há *buffers* intermediários), portanto, minimiza cópias

Comunicação Síncrona (cont.)

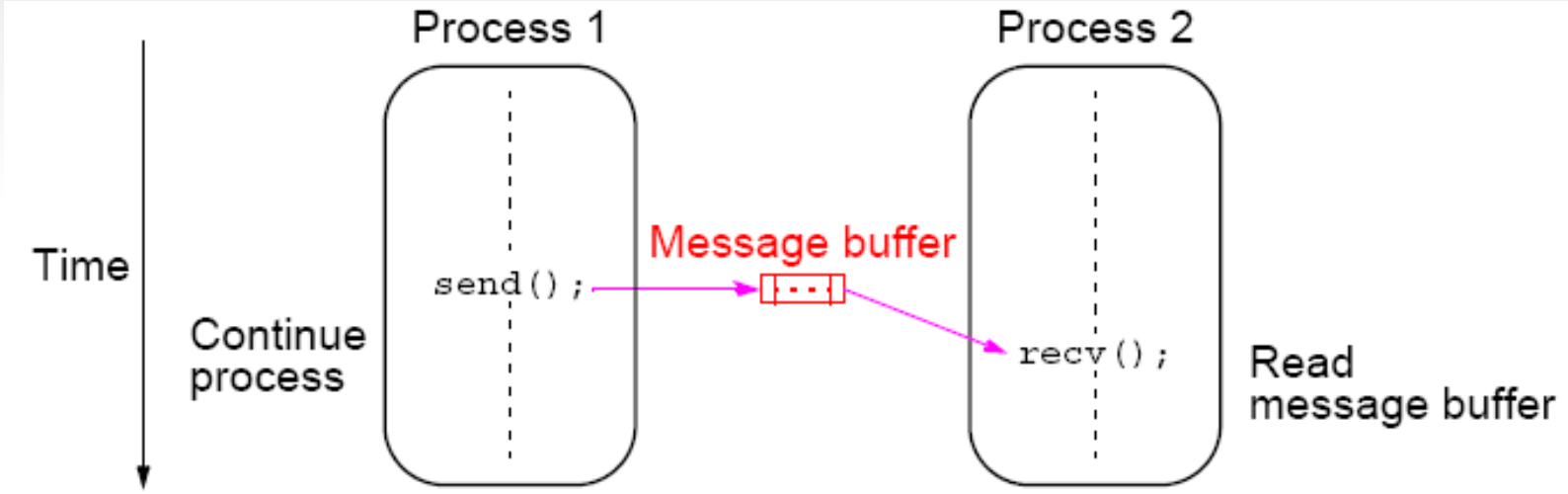


Comunicação Assíncrona

- MPI copia o *buffer* para outro armazém local e envia o novo armazém para o processo destino
  - SEND termina quando dados do seu próprio *buffer* forem copiados para o armazém
  - RECV termina apenas quando os dados estão no próprio *buffer*

- Características:
  - Mensagem local;
  - Não há sincronização (a recepção correspondente não precisa ter começado)
  - Requer mais memória (e cópias) que o modo síncrono

Comunicação Assíncrona (cont.)



Semântica de MPI_SEND e MPI_RECV

- O mesmo MPI_RECV é utilizado para comunicação síncrona e assíncrona
- Diferença apenas no SEND
- MPI possui rotinas específicas para SEND síncrono e para SEND assíncrono;
- MPI_SEND não é nenhuma delas
  - MPI_SEND e MPI_RECV utilizam terceiro modo de comunicação, denominado padrão (*standard*)

Modo Padrão (SEND e RECV)

- Uma implementação de MPI é livre para escolher entre comunicação síncrona e assíncrona na implementação de MPI_SEND
  - Motivo: Para garantir a correção de programas portáteis, não é possível obrigar qualquer sistema a possuir memória suficiente para comunicação assíncrona

- Ao deixar a comunicação síncrona ou assíncrona a critério da implementação, MPI impõe semântica a MPI_SEND e MPI_RECV tal que:
  - Semântica correta em qualquer dos dois modos
  - Permite uma implementação escolher, dinamicamente, um dos modos de comunicação dependendo das circunstâncias
  - Permite que a implementação melhore o desempenho do programa, se há espaço suficiente para comunicação assíncrona
  - Mantém correção de programas portáteis

Comunicação bloqueante e não-bloqueante

- Bloqueante (*Blocking*)
  - Retorna apenas após sua comunicação local tenha sido completada
  - A transferência (envio) pode ainda não ter sido efetuada, sendo o dado salvo em um *buffer*
  - Após o retorno, o *buffer* pode ser re-escrito

- Não-bloqueante (*Non-blocking*)
  - Retorno imediato
  - Assume que o dado a ser comunicado não será modificado pelas ações subseqüentes do programa
    - Premissa a ser assegurada pelo programador

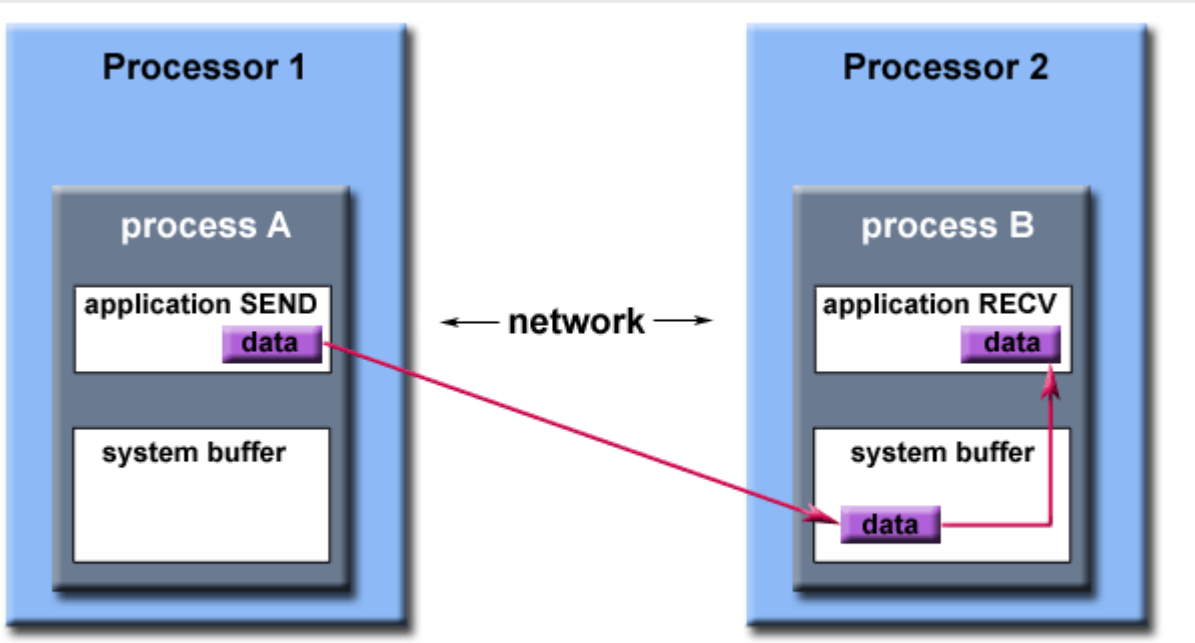
Comunicação padrão é bloqueante

- MPI_SEND e MPI_RECV são bloqueantes

- A semântica do modo padrão de comunicação MPI move a atenção do sincronismo (ou não) da operação para o reuso do *buffer*
  - MPI_SEND termina apenas quando o *buffer* pode ser reusado
    - O término de MPI_SEND não garante que a mensagem chegou ao destino
    - O término de MPI_SEND pode requerer o término do MPI_RECV correspondente
  - MPI_RECV termina apenas quando o *buffer* foi recebido
    - O término de MPI_RECV garante que o MPI_SEND correspondente foi iniciado e enviou os dados
    - O término de MPI_RECV requer o início do MPI_SEND correspondente (mas não garante que o MPI_SEND terminou).

- Para garantir a correção de um programa MPI, é necessário:
  - Garantir que pares MPI_SEND e MPI_RECV correspondentes possam ser executados, ou seja, que nenhum fluxo de execução impeça a execução de um dos dois

Exemplo de Comunicação padrão



Path of a message buffered at the receiving process

Mpi_Send (padrão)

- `ierr = MPI_Send(buf, count, datatype, dest, tag, comm);`

- Onde:
  - `buf` corresponde a primeira posição dos dados a enviar
  - `Count` corresponde a quantidade de dados a enviar (>=0);
    - posições contíguas
  - `datatype` significa o Tipo MPI dos dados a enviar
  - `dest` identifica o Número do processo a receber a mensagem (no comunicador)
  - `tag` é um Identificador (rótulo) da mensagem (>=0 e <= *MPI_TAG_UB*)
  - `comm` é o Comunicador da mensagem

Mpi_Recv (padrão)

- `ierr = MPI_Recv(buf, count, datatype, src, tag, comm, status);`

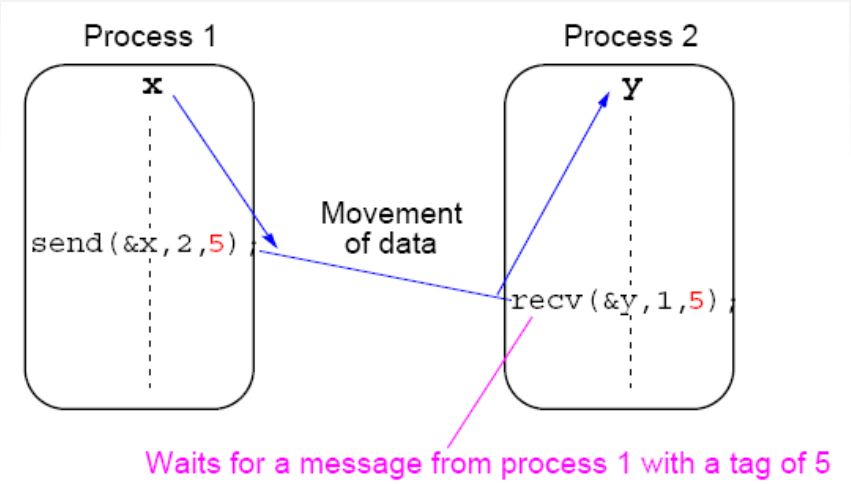
- Onde:
  - `buf` corresponde a primeira posição dos dados a serem recebidos
  - `Count` corresponde a quantidade de dados a serem recebidos (>=0);
    - posições contíguas
  - `datatype` significa o Tipo MPI dos dados a serem recebidos
  - `src` identifica o Número do processo que enviou a mensagem (no comunicador)
  - `tag` é um Identificador (rótulo) da mensagem (>=0 e <= *MPI_TAG_UB*)
  - `comm` é o Comunicador da mensagem
  - `status` retorna informações sobre a mensagem recebida
    - *Array* de tamanho *MPI_STATUS_SIZE*

Detalhes da Recepção

- *src* pode ser *MPI_ANY_SOURCE*
  - Recebe a mensagem de qualquer processo no comunicador
- *tag* pode ser *MPI_ANY_TAG*
  - Recebe mensagem com qualquer *tag*
- *status Array*:
  - *Status[MPI_SOURCE]* contém o número do processo que enviou a mensagem
  - *Status[MPI_TAG]* contém a *tag* da mensagem recebida
  - *Status[MPI_ERROR]* contém um código de erro
  - Contém outras informações, extraídas por funções específicas

Message Tag

- Usado para diferenciar diferentes tipos de mensagem
- O *tag* da mensagem é enviado junto com a mensagem no seu envelope
- Para completar a comunicação um par *MPI_Send* e *MPI_Recv* com o mesmo *tag* deverá existir



Tamanho da Mensagem

- Semântica de `count`:
  - O padrão afirma que `count` é o número de elementos em *buffer*, entretanto:
- Em *SEND*, `count` é a quantidade de dados a enviar;
  - Logo, `size(buf) >= count`;
- Em *RECV*, `count` é o tamanho de `buf`
  - o número de elementos recebidos é `<= count`
- Número de elementos recebidos por RECV:
  - `ierr = MPI_Get_count(status, datatype, cnt);`

Seleção da Mensagem a Receber

- Dentre as mensagens existentes, o MPI seleciona a mensagem correspondente a um dado *MPI_RECV* se e somente se:
  - O comunicador da mensagem é o mesmo do *RECV*
  - A mensagem é enviada para o processo que emite o *RECV*
  - O *tag* da mensagem está de acordo com o *tag* do *RECV*
  - O processo que envia a mensagem está de acordo com o *src* do *RECV*

- Observe que:
  - Nenhuma informação sobre os dados é utilizada para escolher a mensagem a receber
  - Um processo pode enviar mensagem para ele mesmo

Tipos de dados transmitidos

- O tipo MPI de dados do SEND e do RECV devem ser idênticos
- Se o SEND e o RECV estão em máquinas com representações diferentes para um mesmo tipo MPI, então o MPI converte automaticamente a representação de uma máquina para outra.
  - (embora esta seja uma característica do projeto MPI, talvez não seja passível de verificação)
- Exceção: o tipo MPI_BYTE transfere binários de uma máquina para outra.

Exemplo

- Enviar (*send*) um inteiro *x* do processo *0* para o processo *1*

```
/* find rank */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank==0) {
                int x;
                MPI_Send(&x, 1, MPI_INT, 1, msgtag,
                          MPI_COMM_WORLD);
                 }
else if (myrank==1) {
                     int x;
                     MPI_Recv(&x, 1, MPI_INT,
                               0, msgtag, MPI_COMM_WORLD,
                               status);
                     }
```

Calculando máximos e mínimos

- Decomposição de domínio particiona os dados
  - Cada processo realiza todas as funções sobre alguns dados
  - A decomposição de domínio é explícita
- O balanceamento da carga é explícito e atrelado à decomposição do domínio (owner's compute)
- Recodificação das estruturas de dados evitando replicação de memória
  - Alteração de, pelo menos, os limites dos laços
  - Escolha entre usar índices globais e índices locais
    - Índices globais requerem domínio contíguo
    - Índices locais requerem mapemento global – local
- MPI permite execução em um mesmo sistema operacional (memória compartilhada) ou em sistemas operacionais distintos (memória distribuída).

```
/* PROGRAMA SERIAL */
#include <stdio.h>
#include <math.h>
#define tam   100000000

int main(int argc, char *argv[]) {
  float *vec, max, min;
  int i;
  vec = (float *) malloc(tam*sizeof(float));
  if (!vec) {
     printf("Impossivel alocar\n"); return(0);
  }

  for(i=0; i<tam; i++) {
    vec[i] = ((float)i - (float)tam/2.0); vec[i] *= vec[i];
  }
  for(i=0; i<tam; i++) {
    vec[i] = sqrt(vec[i]);
  }
  max = min = vec[0];
  for(i=1; i<tam; i++) {
    if (vec[i]>max) max=vec[i];
    if (vec[i]<min) min=vec[i];
  }
  printf("Max=%f, Min=%f\n", max, min);
  return(0);
}
```

MPI Versão 0

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#define tam   100000000

int main(int argc, char *argv[]) {
float      *vec, max, min, *maxTot, *minTot;
int        i, pri, ult, tamLocal, ierr, numProc,
           esteProc, iproc;
MPI_STATUS status;

MPI_Init(&argc,&argv);
MPI_COMM_SIZE(MPI_COMM_WORLD, numProc);
MPI_COMM_RANK(MPI_COMM_WORLD, esteProc);
```

```
maxTot = (float *) malloc(numProc*sizeof(float));
if (!maxTot) {
  printf("Impossivel alocar MaxTot\n"); return(0);
}

minTot = (float *) malloc(numProc*sizeof(float));
if (!minTot) {
  printf("Impossivel alocar MinTot\n"); return(0);
}
```

```
tamLocal = tam/numProc;
pri = esteProc*tamLocal;

if (esteProc==numProc-1) {
  ult = tam - pri;}
else
  ult = floor((float)tam/numProc);

vec = (float *) malloc(ult*sizeof(float));
if (!vec) {
  printf("Impossivel alocar\n"); return(0);
}

for(i=0; i<ult; i++) {
  vec[i] = ((float)(i+pri) - (float)tam/2.0);
  vec[i] *= vec[i];
}
for(i=0; i<ult; i++) {
  vec[i] = sqrt(vec[i]);
}
```

```
maxTot[esteProc] = vec[0];
minTot[esteProc] = vec[0];
for(i=0; i<ult; i++) {
  if (vec[i]> maxTot[esteProc])
    maxTot[esteProc]=vec[i];
  if (vec[i]< minTot[esteProc])
    minTot[esteProc]=vec[i];
}

if (esteProc!=0) { /* Processos escravos */
  MPI_Send(&maxTot[esteProc], 1, MPI_FLOAT,
           0, 12, MPI_COMM_WORLD);
  MPI_Send(&minTot[esteProc], 1, MPI_FLOAT,
           0, 13, MPI_COMM_WORLD);
}
else { /* Processo Mestre */
  for(iproc=1; iproc<numProc; iproc++) {
    MPI_Recv(&maxTot[iproc], 1, MPI_FLOAT,
             iproc, 12, MPI_COMM_WORLD, &status);
    MPI_Recv(&minTot[iproc], 1, MPI_FLOAT,
             Iproc, 13, MPI_COMM_WORLD, &status);
  }
}
```

```
if (esteProc==0) { /* Processo Mestre */
   max = maxTot[0]; min = minTot[0];
   for(i=1; i<numProc; i++) {
      if (maxTot[i]>max) max=maxTot[i];
      if (minTot[i]<min) min=minTot[i];
   }
printf("Max=%f, Min=%f\n", max, min);
MPI_Finalize();
return(0);
}
```

Speed-up e Eficiência

Processos	Tempo CPU (s)	Speed-up	Eficiência
1	167,87	1,00	100,00%
2	84,78	1,98	99,00%
3	56,90	2,95	98,33%
4	43,37	3,87	96,75%
5	35,26	4,76	95,20%
6	30,19	5,56	92,16%

Decomposição de Domínio – Forma 1

- Arredonda “para baixo” a porção de cada processador
- Concentra resto no último processo

```
tamLocal = tam/numProc;
pri = esteProc*tamLocal;
if (esteProc == numProc-1)
   ult = tam – pri;
else
   ult = floor((float)tam/numProc);
```

Decomposição de Domínio – Forma 2

- Arredonda “para cima” a porção de cada processador
- Retira excesso do último processo

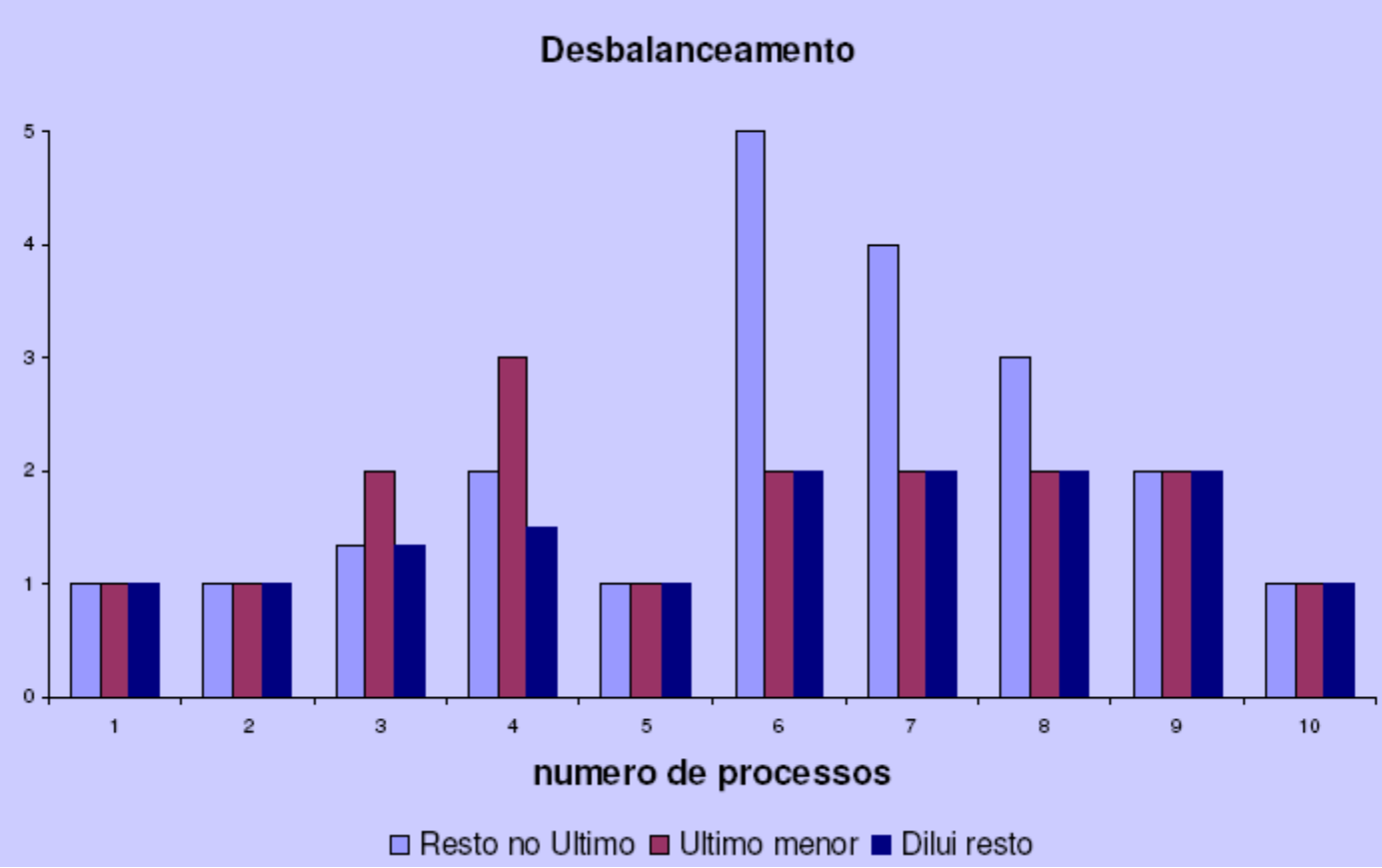
```
tamLocal = ceil(tam/(float)numProc);
pri = esteProc*tamLocal;
if (tam < ((esteProc+1)*tamLocal))
   ult = tam – pri;
else
   ult = (esteProc+1)*tamLocal;
```

Decomposição de Domínio – Forma 3

- Arredonda “para baixo” a porção de cada processador
- Dilui resto em alguns processos, um elemento por processo

```
tamLocal = tam/numProc;
resto = tam – tamLocal*nProc;
if (esteProc < resto) {
   tamLocal = tamLocal + 1;
   resto = 0;
}
pri = esteProc*tamLocal + resto;
ult = (esteProc+1)*tamLocal + resto - pri;
```

Desbalanceamento medido



Comunicações Coletivas

- Uma comunicação coletiva envolve múltiplos processos
- Todos os processos do comunicador emitem a mesma operação
- O término indica que o processo pode acessar ou alterar o *buffer* de comunicação

BARREIRA

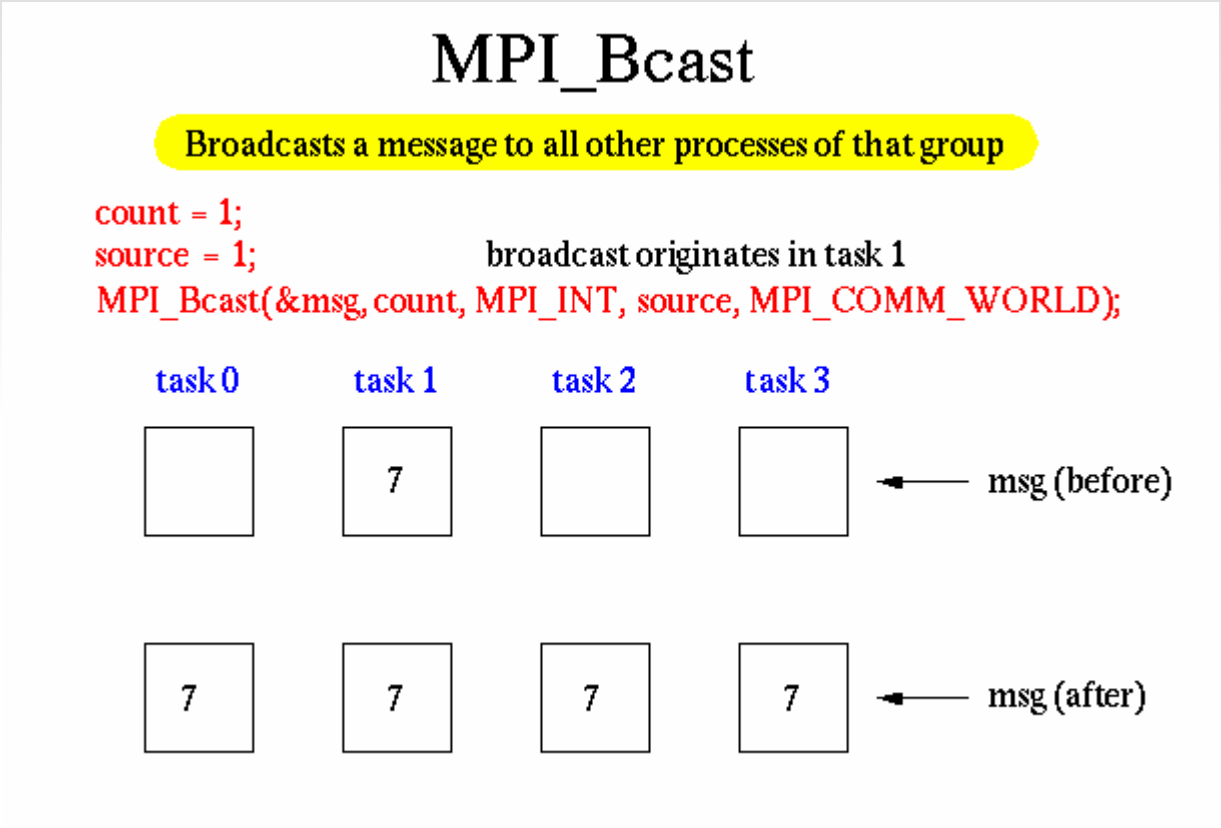
- `ierr = MPI_Barrier(comm);`
- Todos os processos que invocam *MPI_BARRIER* são bloqueados até que todos os processos do comunicador invoquem a barreira
- Sincroniza todos os processos do comunicador
- Cuidado: garanta que todos os processos do comunicador invoquem a barreira

BROADCAST

- `ierr = MPI_Bcast(buf, count, datatype, root, comm);`

- Processo *root* envia *bfr* de tamanho *count* e tipo *datatype* para todos os demais processos do comunicador.

Exemplo de Broadcast



REDUÇÃO

- `MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm);`
- Aplica a operação *op* aos *sendbuf* de todos os processos do comunicador, elemento a elemento
- Coloca o resultado no *recvbuf* do processo *root*
- Todos os *sendbuf* e o *recvbuf* tem tamanho *count* e mesmo tipo MPI

REDUÇÃO (cont.)

- Algumas operações MPI pré-definidas:
  - Lista completa na página 114 do padrão
  - Usuário pode definir outras operações (páginas 120-122 do padrão)

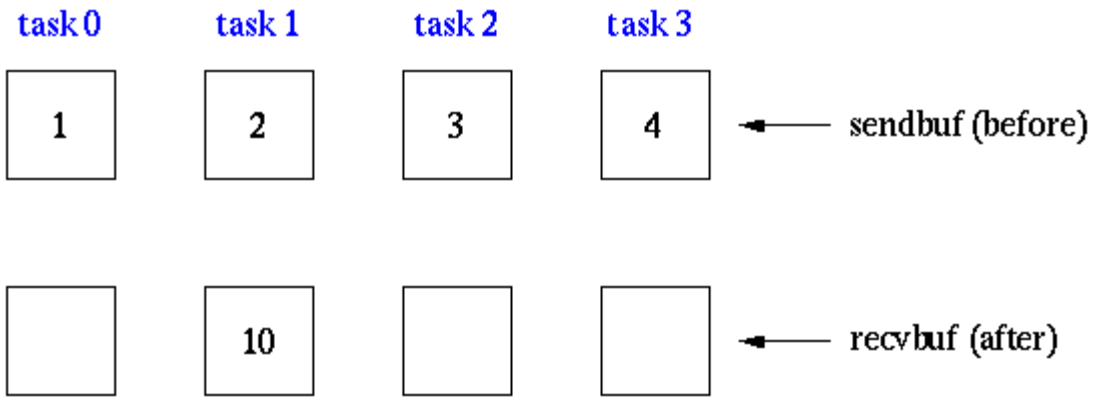
- *MPI_PROD* Produto
- *MPI_SUM* Soma
- *MPI_MIN* Mínimo
- *MPI_MAX* Máximo

Exemplo de redução

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);
```



Exemplo: Calculando PI em C

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
int done = 0, n, myid, numprocs, i, rc;
double PI25DT = 3.14159265358979323846264 3;
double mypi, pi, h, sum, x, a;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
while (!done) {
      if (myid == 0) {
         printf("Enter the number of intervals: (0 quits) ");
         scanf("%d",&n);
      }
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n == 0) break;
```

Exemplo: Calculando PI em C (cont.)

```
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
        }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
       printf("pi is approximately %.16f, Error is .16f\n",
              pi, fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
}
```