Centro Nacional de Processamento de Alto Desempenho - SP

TREINAMENTO:

INTRODUÇÃO AO MPI

CONTEÚDO

INTRODUÇÃO A PROGRAMAÇÃO PARALELA

1 - IDEIAS E CONCEITOS

O que é paralelismo ?

- É uma estratégia, que em computação, se utiliza para obter resultados mais rápidos de grandes e complexas tarefas.

- Uma grande tarefa pode ser executada serialmente, passo a passo, ou, pode ser dividida para ser executada em paralelo.

- O paralelismo, portanto, é efetuado:

  * Dividindo-se uma tarefa, em várias pequenas tarefas;

  * Distribuindo-se as pequenas tarefas por entre vários "trabalhadores", que irão executá-las simultaneamente;

  * Coordenar os "trabalhadores".

OBS: Evite tarefas, nas quais, coordena-las leve mais tempo que executa-las.

Ex: Contrução Civil, Industria Automobilística

Programação Serial

Tradicionalmente, os programas de um computador são elaborados para serem executados em máquinas seriais:

* Somente um processador;

* Uma instrução executada por vez;

* Tempo de execução irá depender de quão rápido a informação se "movimenta" pelo hardware.

  Limites:

  Velocidade da Luz = 30 cm / nanosegundo
  Velocidade no Cobre = 9 cm / nanosegundo

* As máquinas seriais mais rápidas, executam uma instrução em aproximadamente 9-12 bilionésimos de segundo;

Necessidade de Processamento mais Rápido

Existem várias classes de problemas que necessitam de processamento mais rápido:

* Problemas de modelagem e simulação, baseados em sucessivas aproximações e de cálculos cada vez mais precisos.

* Problemas que dependem de manipulação de imensas bases de dados:

    1 - Processamento de sinal e imagem;
    2 - Visualização de dados;
    3 - Banco de Dados.

* Grandes desafios:

    1 - Modelagem de Clima;
    2 - Turbulência de Fluidos;
    3 - Dispersão de Poluição;
    4 - Engenharia Genética;
    5 - Circulação de Correntes Marítimas;
    6 - Modelagem de Semicondutores;
    7 - Modelagem de Supercondutores;
    8 - Sistema de Combustão.

Computação Paralela

Necessidades para se possuir um ambiente de computação paralela:

* Múltiplos processadores;

* Rede (Interligando os processadores);

* Um ambiente que possibilite criar e manipular um processamento paralelo:
  - Sistema Operacional;
  - Modelos de Programação Paralela.

* Um algoritmo paralelo e um programa paralelo.

Programação Paralela

Necessidades para se programar em paralelo:

* Decomposição do algoritmo, ou dos dados, em "pedaços";

* Distribuição dos "pedaços" por entre processadores diferentes, que trabalhem simultaneamente;

* Coordenação do trabalho e da comunicação entre esses processadores;

* Considerações:
  - Tipo da arquitetura paralela;
  - Tipo da comunicação entre processadores.

2 - COMUNICAÇÃO ENTRE PROCESSADORES

A maneira como os processadores se comunicam é dependente da arquitetura de memória utilizada no ambiente, que por sua vez, irá influenciar na meneira de se escrever um programa paralelo.

Memória Compartilhada

- Multiplos processadores operam de maneira independente, mas compartilham os recursos de uma única memória;

- Somente um processador por vez pode acessar um endereço na memória compartilhada;

- Sincronização é necessária no acesso para leitura e gravação da memória, pelas tarefas paralelas. A "coerência" do ambiente é mantida pelo Sistema Operacional;

- Vantagens: * Fácil de usar;
  * A transferência de dados é rápida.

- Desvantagens: * Limita o número de processadores.

- Examplos: Cray Y-MP, Convex C-2, Cray C-90.

Memória Distribuida

- Múltiplos processadores operam independentemente, sendo que, cada um possui sua própria memória;

- Os dados são compartilhados através de uma rede de comunicação, utilizando-se "Message-Passing";

- O usuário é responsável pela sincronização das tarefas usando "Message-Passing".

- Vantagens:
  * Não existe limites no número de processadores;
  * Cada processador acessa, sem interferência e rapidamente, sua própria memória.

- Desvantagens:
  * Elevado "overhead" devido a comunicação;
  * O usuário é responsável pelo envio e recebimento dos dados.

- Exemplos: nCUBE, Intel Hypercube, IBM SP1, SP2, CM-5

3 - CRIAÇÃO DE UM PROGRAMA PARALELO

Decomposição do Programa

Para se decompor um programa em pequenas tarefas para serem executadas em paralelo, é necessário se ter a idéia da decomposição funcional e da decomposição de domínio.

1 - Decomposição Funcional

- O problema é decomposto em **diferentes** tarefas que serão distribuídas por entre múltiplos processadores para execução simultanea;

- Perfeito para um programa dinâmico e modular. Cada tarefa será um programa diferente.

2 - Decomposição de Domínio

- Os dados são decompostos em grupos que serão distribuídos por entre múltiplos processadores que executarão, simultaneamente, um **mesmo** programa;

- Perfeito para dados estáticos (Resolução de imensas matrizes e para calculos de elementos finitos);

Comunicação

Em um programa, é essencial que se compreenda a comunicação que ocorre entre os processadores:

- O Programador **tem** que entender como ocorrerá a comunicação, para saber tirar vantagens ao se utilizar uma "Message-Passing Library". A programação da comunicação é explicita.

- "Compiladores Paralelos" (Data Parallel Compilers), efetuam toda a comunicação necessária. O programador não necessita entender dos métodos de comunicação.

- Os métodos de comunicação para "Message-Passing" e para "Data Parallel", são exatamente os mesmos.

  * Point to Point
  * One to All Broadcast
  * All to All Broadcast
  * One to All Personalized
  * All to All Personalized
  * Shifts
  * Collective Computations

4 - CONSIDERAÇÕES DE PERFORMANCE

Amdahl's Law

A lei de Amdahl's determina o potencial de aumento de velocidade a partir da porcentagem paralelizavel de um programa (**f**):

speedup = 1 / (1 - f)

- Num programa, no qual não ocorra paralização, **f=0**, logo, **speedup=1** (Não existe aumento na velocidade de processamento);

- Num programa, no qual ocorra paralelização total, **f=1**, logo, **speedup é infinito** (Teoricamente).

Se introduzirmos o número de processadores na porção paralelizavel de processamento, a relação passará a ser modelada por:

speedup = 1 / [(P/N) + S]

P = Porcentagem paralelizavel;
N = Número de processadores;
S = Porcentagem serial;

N	P = 0,50	P = 0,90	P = 0,99
10	1,82	5,26	9,17
100	1,98	9,17	50,25
1000	1,99	9,91	90,99
10000	1,99	9,91	99,02

Balanceamento de Carga ("Load Balancing")

- A distribuição das tarefas por entre os processadores, deverá ser, de maneira que o tempo da execução paralela seja eficiente;

- Se as tarefas não forem distribuidas de maneira balanceada, é possível que ocorra a espera pelo término do processamento de uma única tarefa, para dar prosseguimento ao programa.

Granularidade ("Granularity")

- É a razão entre computação e comunicação:

Fine-Grain
* Tarefas executam um pequeno número de instruções entre ciclos de comunicação;
* Facilita o balanceamento de carga;
* Baixa computação, alta comunicação;
* É posível que ocorra mais comunicação do que computação, diminuindo a performance.

Coarse-Grain
* Tarefas executam um grande número de instruções entre cada ponto de sincronização;
* Difícil de se obter um balanceamento de carga eficiente;
* Alta computação, baixa comunicação;
* Possibilita aumentar a performance.

5 - REFERÊNCIAS

Os dados apresentados nesta primeira parte, foram obtidos dos seguintes documentos:

1 - **SP Parallel Programming Workshop**
Introduction to Parallel Programming
MHPCC - Maui High Performance Computing Center
Blaise Barney - 02 Julho 1996

2 - **Seminário: Overview of Parallel Processinng**
Kevin Morooney e Jeff Nucciarone - 17 Outubro 1995

INTRODUÇÃO A "MESSAGE-PASSING"

1 - O MODELO: "MESSAGE-PASSING"

O modelo "Massage-Passing" é um dos vários modelos computacionais para conceituação de operações de programa. O modelo "Message-Passing" é definido como:

* Conjunto de processos que possuem acesso à memória local;

* Comunicação dos processos baseados no envio e recebimento de mensagens;

* A transferência de dados entre processos requer operações de cooperação entre cada processo (uma operação de envio deve "casar" com uma operação de recebimento).

2 - BIBLIOTECAS "MESSAGE-PASSING"

O conjunto operações de comunicação, formam a base que permite a implementação de uma biblioteca de "Message-Passing":

Domínio público - PICL, PVM, PARMACS, P4, MPICH, etc**;**

Privativas - MPL, NX, CMMD, MPI, etc;

Existem componentes comuns a todas as bibliotecas de "Message-Passing", que incluem:

* Rotinas de gerência de processos (Inicializar, finalizar, determinar o número de processos, identificar processos);

* Rotinas de comunicação "Point-to-Point" (Envio e recebimento de mensagens entre dois processos);

* Rotinas de comunicção de grupos ("broadcast", sincronização de processos).

Terminologia de Comunicação

Buffering Cópia temporária de mensagens entre endereços de memória efetuada pelo sistema como parte de seu protocolo de transmissão. A cópia ocorre entre o "buffer" do usuário (definido pelo processo) e o "buffer" do sistema (definido pela biblioteca);

Blocking Uma rotina de comunicação é "blocking", quando a finalização da execução da rotina, é dependente de certos "eventos" (espera por determinada ação, antes de liberar a continuação do processamento);

Non-blocking Uma rotina de comunicação é "non-blocking", quando a finalização da execução da rotina, não depende de certos "eventos" (não há espera, o processo continua sendo executado normalmente);

Síncrono Comunicação na qual o processo que envia a mensagem, não retorna a execução normal, enquanto não haja um sinal do recebimento da mensagem pelo destinatário;

Assíncrono Comunicação na qual o processo que envia a mensagem, não espera que haja um sinal de recebimento da mensagem pelo destinatário.

Comunicação "Point-to-Point"

Os componentes básicos de qualquer biblioteca de "Message-Passing" são as rotinas de comunicação "Point-to-Point" (transferência de dados entre **dois** processos).

Bloking	**send**	Finaliza, quando o "buffer" de envio está pronto para ser reutilizado;
	receive	Finaliza, quando o "buffer" de recebimento está pronto para ser reutilizado;
Nonblocking		Retorna imediatamente, após envio ou recebimento de uma mensagem.

Comunicação Coletiva

As rotinas de comunicação coletivas, são voltadas para coordenar **grupos** de processos.

Existem, basicamente, três tipos de rotinas de comunicação coletiva:

* Sincronização
* Envio de dados: Broadcast, Scatter/Gather, All to All
* Computação Coletiva: Min, Max, Add, Multiply, etc

"Overhead"

Existem duas fontes de "overhead" em bibliotecas de "message-passing":

"System Overhead" — É o trabalho efetuado pelo sistema para transferir um dado para seu processo de destino;

Ex.: Cópia de dados do "buffer" para a rede.

"Syncronization Overhead" — É tempo gasto na espera de que um evento ocorra em um outro processo;

Ex.: Espera, pelo processo origem, do sinal de OK pelo processo destino.

Passos para se obter performance

* Iniciar pelo programa serial otimizado;

* Controlar o processo de "Granularity" (Aumentar o número de computação em relação a comunicação entre processos);

* Utilize rotinas com comunicação **non-blocking**;

* Evite utilizar rotinas de sincronização de processos;

* Evite, se possivel, "buffering";

* Evite transferência de grande quantidade de dados;

3 - REFERÊNCIAS

Os dados apresentados nesta segunda parte foram obtidos dos seguintes documentos:

1 - **SP Parallel Programming Workshop**
Message Passing Overview
MHPCC - Maui High Performance Computing Center
Blaise Barney - 03 Julho 1996

2 - **MPI: The Complete Reference**
The MIT Press - Cambridge, Massachusetts
Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker
Jack Dongarra

INTRODUÇÃO AO MPI

1 - O QUE É MPI ?

Message **P**assing **I**nterface

Uma biblioteca de "Massage-Passing", desenvolvida para ser padrão em ambientes de memória distribuida, em "Message-Passing" e em computação paralela.

"Message-Passing" portável para qualquer arquitetura, tem aproximadamente 125 funções para programação e ferramentas para se analisar a performance.

Utilizado por programas em C e FORTRAN.

A plataforma alvo para o MPI, são ambientes de memória distribuida, máquinas paralelas massivas, "clusters" de estações de trabalho e redes heterogêneas.

Todo paralelismo é **explícito:** o programador é responsável em identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI.

2 - HISTÓRICO

Fins da década de 80: Memória distribuída, o desenvolvimento da computação paralela, ferramentas para desenvolver programas em ambientes paralelos, problemas com portabilidade, performance, funcionalidade e preço, determinaram a necessidade de se desenvolver um padrão.

Abril de 1992: Encontro de padrões de "Message-Passing" em ambientes de memória distribuida (Centro de Pesquisa em Computação Paralela, Williamsburg, Virginia).

- Discussão das necessidades básicas e essenciais para se estabelecer um padrão de interface "Message-Passing";

- Criado um grupo de trabalho para dar continuidade ao processo de padronização;

Novembro de 1992: Reunião em Minneapolis do grupo de trabalho e apresentação de um primeiro esboço de interface "Message-Passing" (**MPI1**).

- O Grupo adota procedimentos para a criação de um **MPI Forum;**

- **MPIF** consiste eventualmente de aproximadamente 175 pessoas de 40 organizações, incluindo fabricantes de computadores, empresas de softwares, universidades e cientistas de aplicação.

Novembro de 1993: Conferência de Supercomputação 93 -

Apresentação do esboço do padrão MPI.

Maio de 1994: Disponibilização da versão padrão do MPI (domínio publico - **MPI1**)

http://www.mcs.anl.gov/Projects/mpi/standard.html

Dezembro de 1995: Conferência de Supercomputação 95 - Reunião para discussão do **MPI2** e extensões

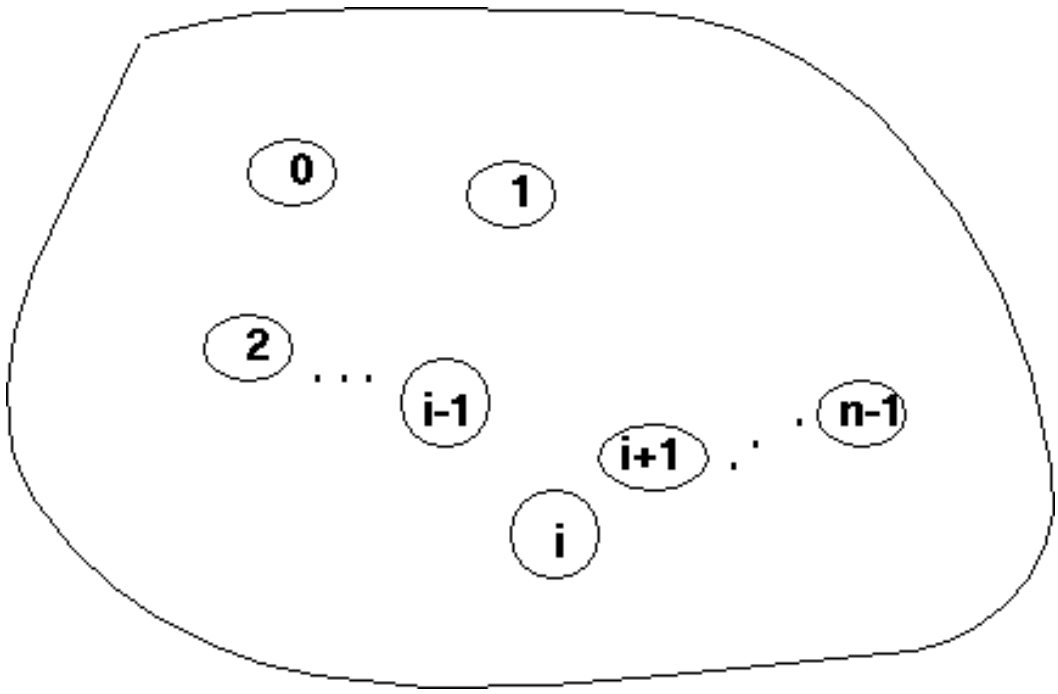
Implementações de MPI

MPI-F: IBM Research
MPICH: ANL/MSU - Domínio Publico
UNIFY: Mississipi State University
CHIMP: Edinburgh Parallel Computing Center
LAM: Ohio Supercomputer Center

3 - CONCEITOS E DEFINIÇÕES

Rank

Todo processo tem uma única identificação, atribuída pelo sistema quando o processo é inicializado. Essa identificação é continua e começa no zero até n-1 processos.



Group

Grupo é um conjunto ordenado de N processos. Todo e qualquer grupo é associado a um "communicator" e, inicialmente, todos os processos são menbros de um grupo com um "communicator" já pré-estabelecido (MPI_COMM_WORLD).

Communicator

O "communicator" define uma coleção de processos (grupo), que poderão se comunicar entre si (contexto).

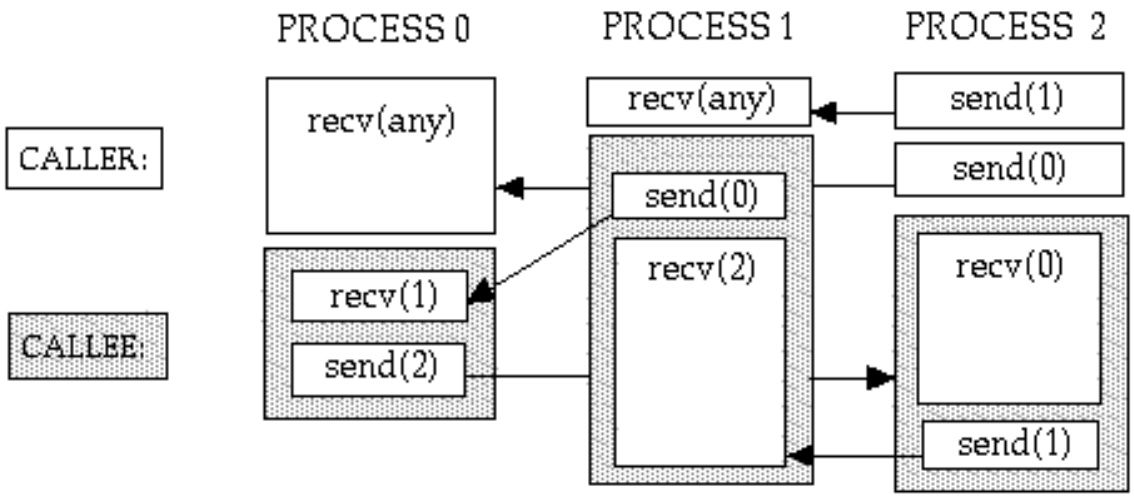
O MPI utiliza essa combinação de grupo e contexto para garantir uma comunicação segura e evitar problemas no envio de mensagens entre os processos.

* É possível que uma aplicação de usuário utilize uma biblioteca de rotinas, que por sua vez, utilize "message-passing".

* Esta rotina pode usar uma mensagem idêntica a mensagem do usuário.

A maioria das rotinas do MPI exigem que seja especificado um "communicator" como argumento. MPI_COMM_WORLD é o comunicador pré-definido que inclui todos os processos definido pelo usuário, numa aplicação MPI.

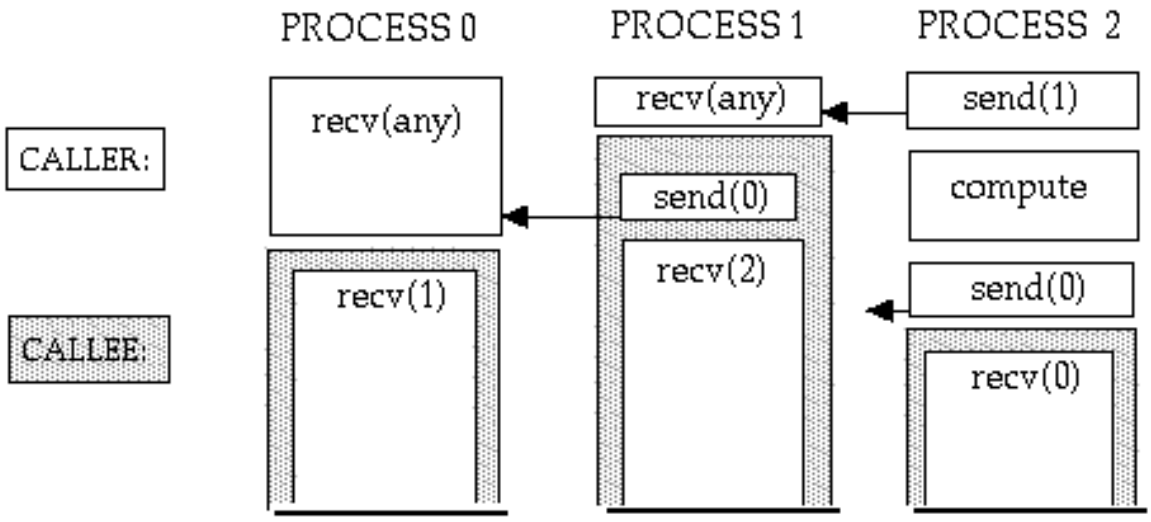
AÇÃO DESEJADA



Courtesy David Walker
Oak Ridge Nat. Lab.

RLF.COMMI.DES 10/16/95

POSSÍVEL AÇÃO INDESEJADA



Courtesy David Walker
Oak Ridge Nat. Lab.

RLF.COMMI.INC 10/16/95

Aplication Buffer

É um endereço normal de memória (Ex: variável) que armazena um dado que o processo necessita enviar ou receber.

System Buffer

É um endereço de memória reservado pelo sistema para armazenar mensagens.

Dependendo do tipo de operação de **send/receive**, o dado no "aplication buffer" pode necessitar ser copiado **de/para** o "system buffer" (**"Send Buffer" e "Receive Buffer"**). Neste caso teremos comunicação assíncrona.

Blocking Comunication

Uma rotina de comunicação é dita **"bloking"**, se a finalização da chamada depender de certos eventos.

Ex: Numa rotina de envio, o dado tem que ter sido enviado com sucesso, ou, ter sido salvo no "system buffer", indicando que o endereço do "aplication buffer" pode ser reutilizado.

Numa rotina de recebimento, o dado tem que ser armazenado no "system buffer", indicando que o dado pode ser utilizado.

Non-Blocking Communication

Uma rotina de comunicação é dita **"Non-blocking"** , se a chamada retorna sem esperar qualquer evento que indique o fim ou o sucesso da rotina.

Ex: Não espera pela cópia de mensagens do "aplication buffer" para o "system buffer", ou a indicação do recebimento de uma mensagem.

OBS: É da responsabilidade do programador, a certeza de que o "aplication buffer" esteja disponível para ser reutilizado.

Este tipo de comunicação é utilizado para melhorar a relação entre computação e comunicação para efeitos de ganho de performance.

Standard Send

Operação básica de envio de mensagens usada para transmitir dados de um processo para outro. Pode ser "blocking" ou "non-blocking".

Synchronous Send

Bloqueia até que ocorra um "receive" correspondente no processo de destino. Pode ser "blocking" ou "non-blocking".

Buffered Send

O programador cria um "buffer" para o dado antes dele ser enviado. Necessidade de se garantir um espaço disponível para um "buffer", na incerteza do espaço do "System Buffer".

Ready Send

Tipo de "send" que pode ser usado se o programador tiver certeza de que exista um "receive" correspondente, já ativo.

Standard Receive

Operação básica de recebimento de mensagens usado para aceitar os dados enviados por qualquer outro processo. Pode ser "blocking" e "non-blocking".

Return Code

Valor inteiro retornado pelo sistema para indicar a finalização da sub-rotina.

4 - COMPILAÇÃO

Atualmente, está instalado no ambiente do CENAPAD-SP duas implementações do MPI:

MPICH Desenvolvido por *Argonne National Laboratory* e *Mississipi State University*. Executa em todas as máquinas interativas do ambiente CENAPAD-SP.

MPI Desenvolvido pela *IBM*. Atualmente, executa em *batch* na máquina SP.

A implementação **MPICH** definiu num único comando, as tarefas de compilação e linkedição.

FORTRAN 77

mpif77 -o <executável> <fonte>

C Standard

mpicc -o <executável> <fonte>

OBS: É possível utilizar todas as opções de compilação dos compiladores C e FORTRAN.

5 - EXECUÇÃO

A implementação **MPICH** desenvolveu um "script" customizado para inicializar os processos.

mpirun <opções> <executável>

opções	**-h**	Mostra todas as opções disponíveis;
	-arch	Especifica a arquitetura;
	-machine	Especifica a máquina;
	-machinefile	Especifica o arquivo de máquinas;
	-np	Especifica o número de processadores;
	-pgleave	Mantêm o arquivo inicializado pelo MPI, com os nomes das máquinas;
	-nolocal	Não executa na máquina local;
	-t	Teste, não executa o programa, apenas imprime o que será executado;
	-dbx	Inicializa o primeiro processo sobre o dbx;

OBS: Já existe um arquivo de máquinas "default", definido durante a instalação do MPICH no diretório:

/usr/local/mpi/mpich/util/machines/machines.rs6000

Ex.:

mpirun -np 6 teste
mpirun -arch sun4 -np 2 -arch rs6000 -np 3 teste.%a

OBS: O símbolo **%a** substitui a arquitetura.

6 - ROTINAS BÁSICAS

Para um grande número de aplicações, um conjunto de apenas **6 subrotinas MPI** serão suficientes para desenvolver uma aplicação no MPI.

6.1 - Inicializar um processo MPI

MPI_INIT

- Primeira rotina do MPI a ser chamada por cada processo.
- Estabelece o ambiente necessário para executar o MPI.
- Sincroniza todos os processos na inicialização de uma aplicação MPI.

C

int MPI_Init (int *argc, char ***argv)

FORTRAN

call MPI_INIT (mpierr)

argc Apontador para um parâmetro da função **main**;

argv Apontador para um parâmetro da função **main**;

mpierr Variável inteira de retorno com o status da rotina.

mpierr=0, Sucesso
mpierr<0, Erro

6.2 - Identificar processo do MPI

MPI_COMM_RANK

- Identifica o processo, dentro de um grupo de processos.
- Valor inteiro, entre 0 e n-1 processos.

C

int MPI_Comm_rank (MPI_Comm comm, int *rank)

FORTRAN

call MPI_COMM_RANK (comm, rank, mpierr)

comm	MPI communicator.
rank	Variável inteira de retorno com o número de identificação do processo.
mpierr	Variável inteira de retorno com o status da rotina.

6.3 - Contar processos no MPI

MPI_COMM_SIZE

- Retorna o número de processos dentro de um grupo de processos.

C

int MPI_Comm_size (MPI_Comm comm, int *size)

FORTRAN

call MPI_COMM_SIZE (comm, size, mpierr)

comm	MPI Communicator.
size	Variável inteira de retorno com o número de processos inicializados durante uma aplicacação MPI.
mpierr	Variável inteira de retorno com o status da rotina

6.4 - Enviar menssagens no MPI

MPI_SEND

- "Blocking send".
- A rotina retorna após o dado ter sido enviado.
- Após retorno, libera o "system buffer" e permite acesso ao "aplication buffer".

C

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

FORTRAN

call MPI_SEND (sndbuf, count, datatype, dest, tag, comm, mpierr)

sndbuf Endereço inicial do dados que será enviado. Endereço do "aplication buffer".

count Número de elementos a serem enviados.

datatype Tipo do dado.

dest Identificação do processo destino.

tag Rótulo da mensagem.

comm MPI communicator.

mpierr Variável inteira de retorno com o status da rotina.

6.5 - Receber mensagens no MPI

MPI_RECV

- "Blocking receive".
- A rotina retorna após o dado ter sido recebido e armazenado.
- Após retorno, libera o "system buffer".

C

int MPI_Recv (void *recvbuf, int count,MPI_Datatype datatype, int source, int tag, *status,MPI_Comm comm)

FORTRAN

call MPI_RECV (recvbuf,count,datatype,source,tag,comm,status,mpierr)

recvbuf Variável indicando o endereço do "aplication buffer".

count Número de elementos a serem recebidos.

datatype Tipo do dado.

source Identificação da fonte. **OBS: MPI_ANY_SOURCE**

tag Rótulo da mensagem. **OBS: MPI_ANY_TAG**

comm MPI communicator.

status Vetor com informações de **source** e **tag**.

mpierr Variável inteira de retorno com o status da rotina.

6.6 - Finalizar processos no MPI

MPI_FINALIZE

- Finaliza o processo para o MPI.
- Última rotina MPI a ser executada por uma aplicação MPI.
- Sincroniza todos os processos na finalização de uma aplicação MPI.

C

int MPI_Finalize()

FORTRAN

call MPI_FINALIZE (mpierr)

mpierr Variável inteira de retorno com o status da rotina.

7 - MPI Message

A passagem de mensagens entre processadores é o modo de comunicação básico num sistema de memória distribuída.

No que consiste essas mensagens ?

Como se descreve uma mensagem do MPI ?

Sempre, uma "MPI Message" contém duas partes: **dado**, na qual se deseja enviar ou receber, e o **envelope** com informações da rota dos dados.

Mensagem=dado(3 parâmetros) + envelope(3 parâmetros)

Ex.:

call MPI_SEND(**sndbuf, count, datatype,** **dest, tag, comm,** mpierr)
DADO **ENVELOPE**

7.1 - DADO

O dado é representado por três argumentos:

1- Endereço onde o dado se localiza;
2- Número de elementos do dado na mensagem;
3- Tipo do dado;

Tipos Básicos de Dados para C

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	

Tipos Básicos de Dados para FORTRAN

MPI_INTEGER	**INTEGER**
MPI_REAL	**REAL**
MPI_DOUBLE_PRECISION	**DOUBLE PRECISION**
MPI_COMPLEX	**COMPLEX**
MPI_LOGICAL	**LOGICAL**
MPI_CHARACTER	**CHARACTER(1)**
MPI_BYTE	
MPI_PACKED	

7.2 - ENVELOPE

Determina como direcionar a mensagem, são basicamente três parâmetros:

1- Identificação do processo que envia ou do processo que recebe;
2- Rótulo ("tag") da mensagem;
3- "Communicator".

8 - Exemplo de um Programa Básico MPI

```
 program hello
include 'mpif.h'
integer me, nt, mpierr, tag, status(MPI_STATUS_SIZE)
character(12) message

call MPI_INIT(mpierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nt, mpierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, mpierr)

tag = 100
if(me .eq. 0) then
  message = 'Hello, world'
  do i=1, nt-1

  call MPI_SEND(message, 12, MPI_CHARACTER, i, tag,
&                       MPI_COMM_WORLD, mpierr)

  enddo
else

  call MPI_RECV(message, 12, MPI_CHARACTER, 0, tag,
&                 MPI_COMM_WORLD, status, mpierr)

endif
print*, 'node', me, ':', message

call MPI_FINALIZE(mpierr)

end
```

9 - COMUNICAÇÃO "POINT-TO-POINT"

Numa comunicação "Point-to-Point", um processo envia uma mensagem e um segundo processo à recebe.

Existem várias opções de programação, utilizando-se comunicação "Point-to-Point", que determinam como o sistema irá trabalhar a mensagem.

Opções que incluem, quatro modos de comunicação: **synchronous, ready, buffered,** e **standard**, e dois modos de processamento: **"blocking"** e **"non-blocking"**.

Existem quatro rotinas "blocking send" e quatro rotinas "non-blocking send", correspondentes aos quatro modos de comunicação.

A rotina de "receive" não especifica o modo de comunicação. Simplesmente, ou a rotina é "blocking" ou, "non-blocking".

9.1 - Modos de Comunicação

"Blocking Receive"

Existem quatro tipos de "blocking send", uma para cada modo de comunicação, mas apenas um "blocking receive" para receber os dados de qualquer "blocking send".

C

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

FORTRAN

call MPI_RECV(buf,count,datatype,source,tag,comm,status,ierror)

Parâmetros Comuns das Rotinas de Send e Receive

buf Endereço do dado a ser enviado, normalmente o nome da variável, do vetor ou da matriz;

count Variável inteira que representa o número de elementos a serem enviados;

datatype Tipo do dado;

source Variável inteira que identifica o processo de origem da mensagem, no contexto do "communicator";

dest Variável inteira que identifica o processo destino, no contexto do "communicator";

tag Variável inteira com o rótulo da mensagem;

comm "Communicator" utilizado;

status Variável inteira com definições da mensagem;

ierror Código de retorno com o status da rotina.

9.1.1 - "Blocking Synchronous Send"

C

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

FORTRAN

call MPI_SSEND(buf, count, datatype, dest, tag, comm, ierror)

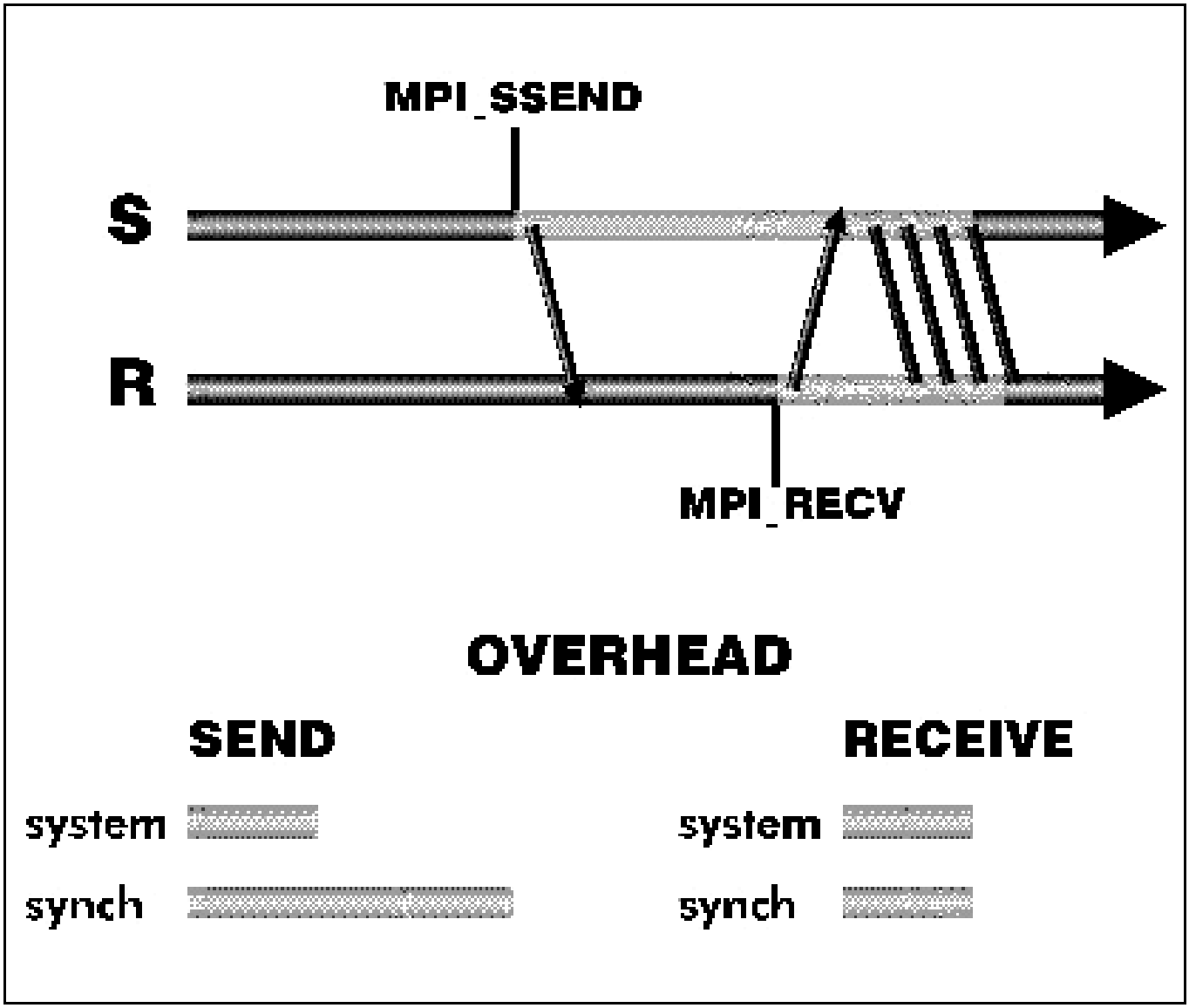
Quando um **MPI_Ssend** é executado, o processo que envia avisa ao processo que recebe que uma mensagem está pronta e **esperando** por um sinal de OK, para que então, seja transferido o dado.

OBS: **"System overhead"** ocorre devido a cópia da mensagem do "send buffer" para a rede e da rede para o "receive buffer".

"Synchronization overhead" ocorre devido ao tempo de espera de um dos processos pelo sinal de OK de outro processo.

Neste modo, o **"Synchronization overhead"**, pode ser significante.

Blocking Synchronous Send e Blocking Receive



9.1.2 - "Blocking Ready Send"

C

int MPI_Rsend(void *buf, int count,MPI_Datatype datatype, int dest, int tag,Mpi_Comm comm)

FORTRAN

call MPI_RSEND(buf, count, datatype, dest, tag, comm, ierror)

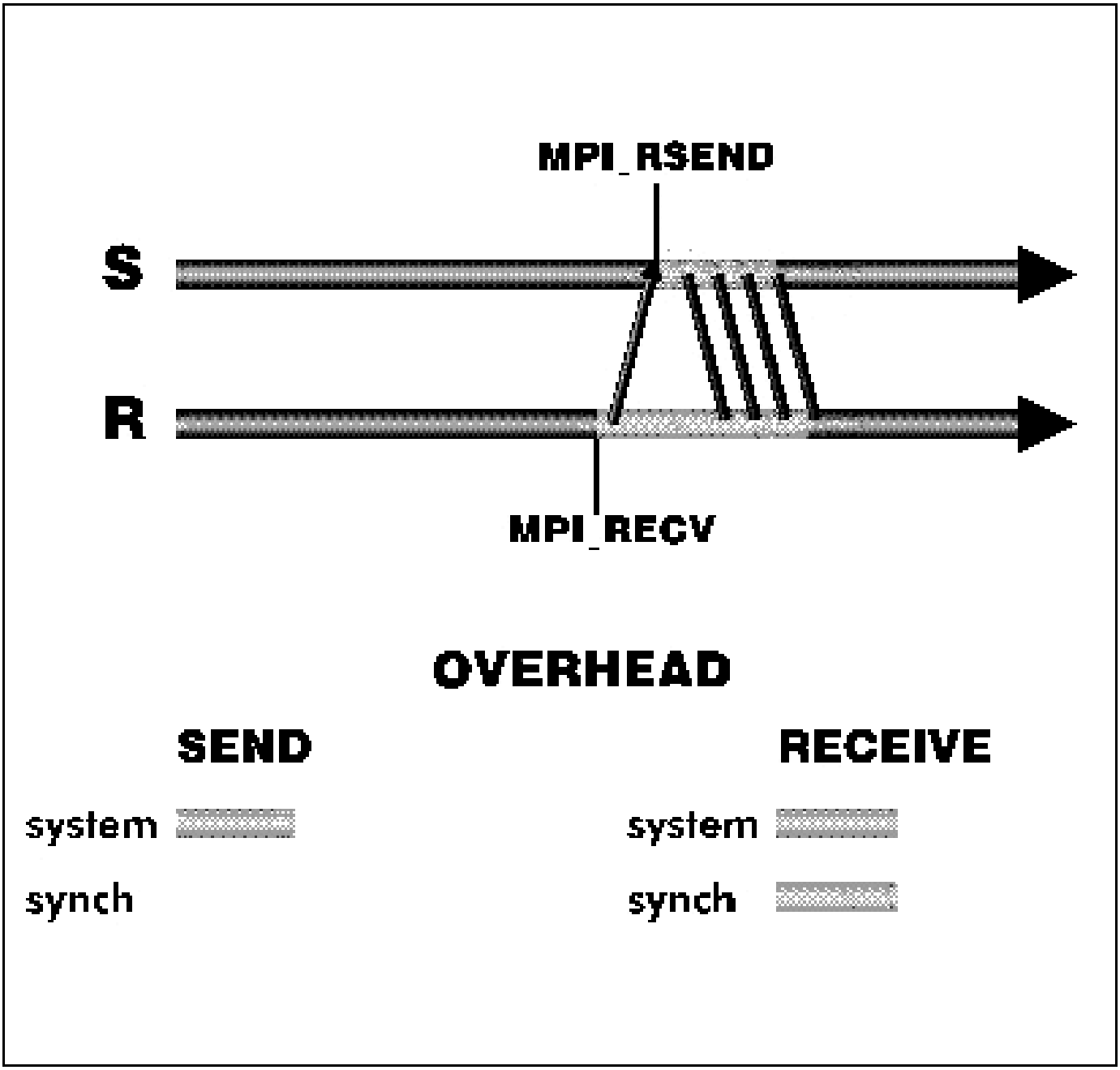
Quando um **MPI_Rsend** é executado a mensagem é enviada imediatamente para a rede. É exigido que um sinal de OK do processo que irá receber, já tenha sido feito.

OBS: Este modo tenta minimizar o "**System overhead**" e o "**Synchronization overhead**" por parte do processo que envia. A única espera ocorre durante a cópia do "send buffer" para a rede.

O processo que recebe pode incorrer num significativo "**Synchronization overhead**". Depende de quão cedo é executada a rotina.

Atenção, este modo somente deverá ser utilizado se o programador tiver certeza que uma **MPI_Recv**, será executado antes de um **MPI_Rsend**.

Blocking Ready Send e Blocking Receive



9.1.3 - "Blocking Buffered Send"

C

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

FORTRAN

call MPI_BSEND(buf, count, datatype, dest, tag, comm, ierror)

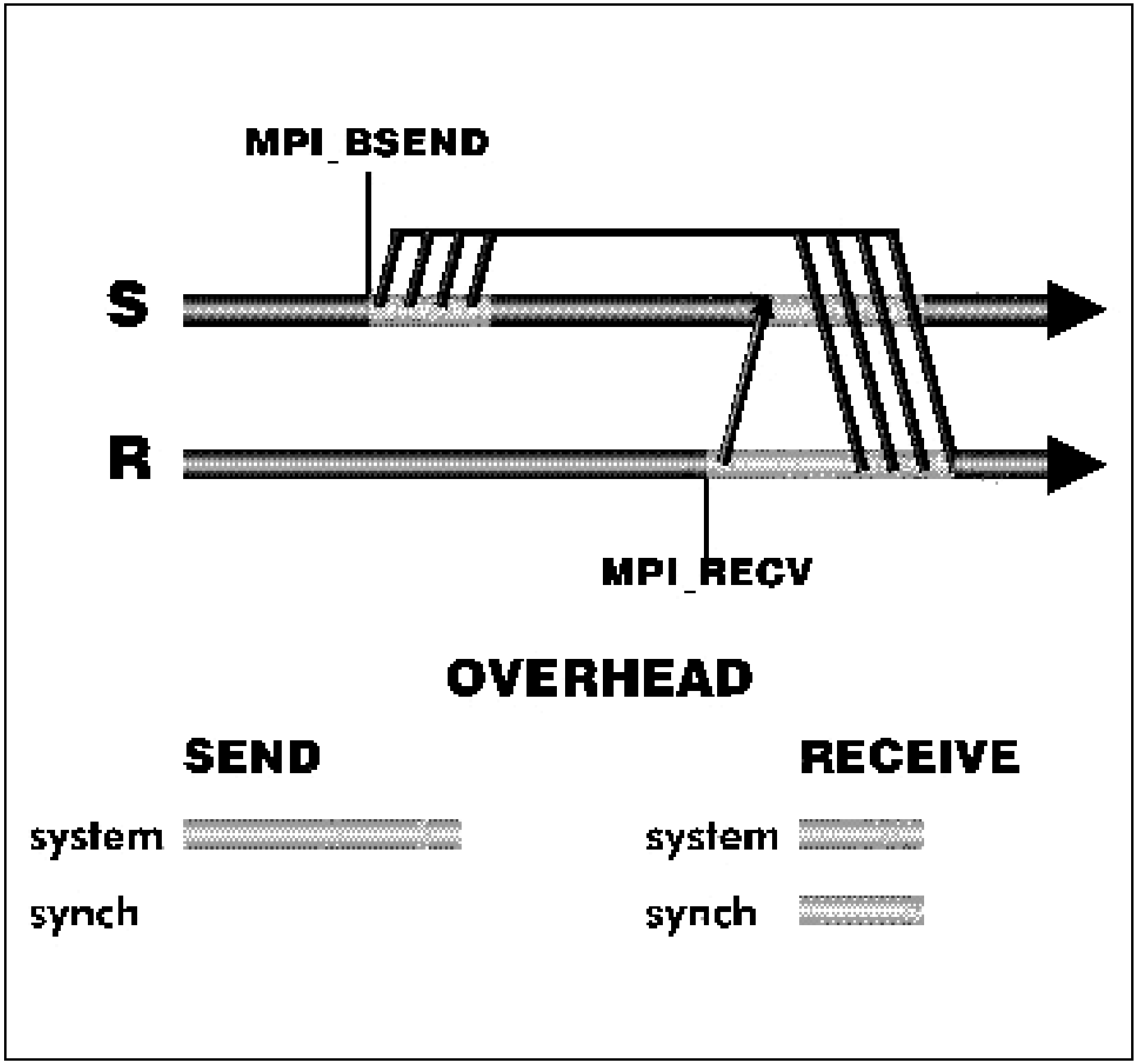
Quando um **MPI_Bsend** é executado a mensagem é copiada do endereço de memória ("Aplication buffer") para um "buffer" definido pelo usuário, e então, retorna a execução normal do programa. É aguardado um sinal de OK do processo que irá receber, para descarregar o "buffer"

OBS: Ocorre **"System overhead"** devido a cópia da mensagem do "Aplication buffer" para o "buffer" definido pelo usuário.

O **"Synchronization overhead"**, não existe no processo que envia, mas é significativo no processo que recebe, caso seja executado o "receive" antes de um "send".

Atenção, neste modo, o usuário é responsável pela definição de um "buffer", de acordo com o tamanho dos dados que serão enviados. Utilizar as rotinas:

MPI_Buffer_attach
MPI_Buffer_detach
Blocking Buffered Send e Blocking Rceive


MPI_BSEND
S
R
MPI_RECV
OVERHEAD
SEND
RECEIVE
system
synch
system
synch

9.1.4 - "Blocking Standard Send"

C

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

FORTRAN

call MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)

Para este modo, será necessário analisar o tamanho da mensagem a ser transmitida, tamanho este, que varia de acordo com o número de processos inicializados. O **"Default"** é um "buffer" de 4Kbytes.

Message <= 4K

Quando **MPI_Send** é executado, a mensagem é imediatamente transmitida para rede, e então, para um "System buffer" do processo que irá receber a mensagem.

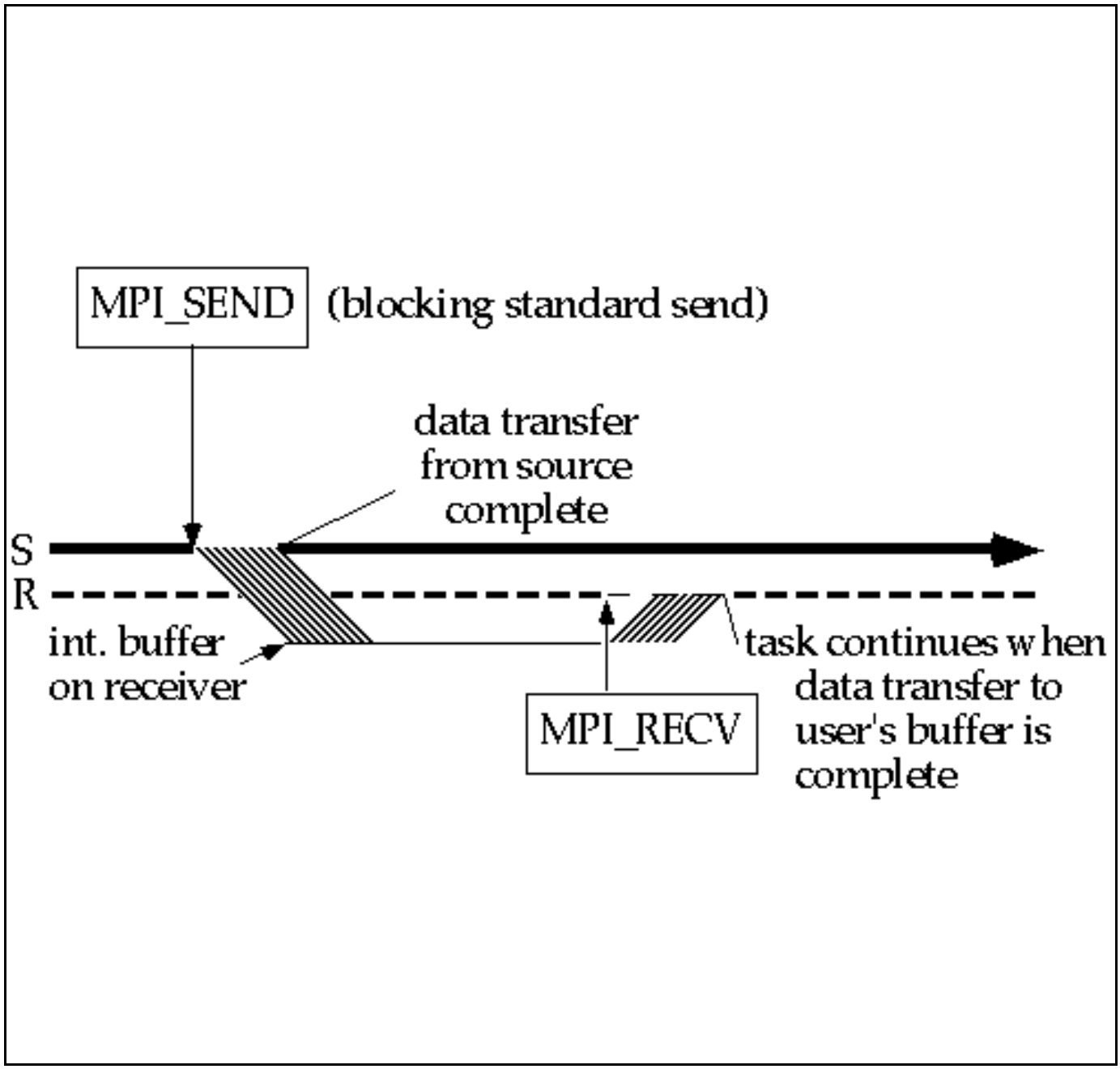
OBS: O **"Synchronization overhead"** é reduzido ao preço de se aumentar o **"System overhead"** devido as cópias extras que podem ocorrer, para o buffer.

Message > 4K

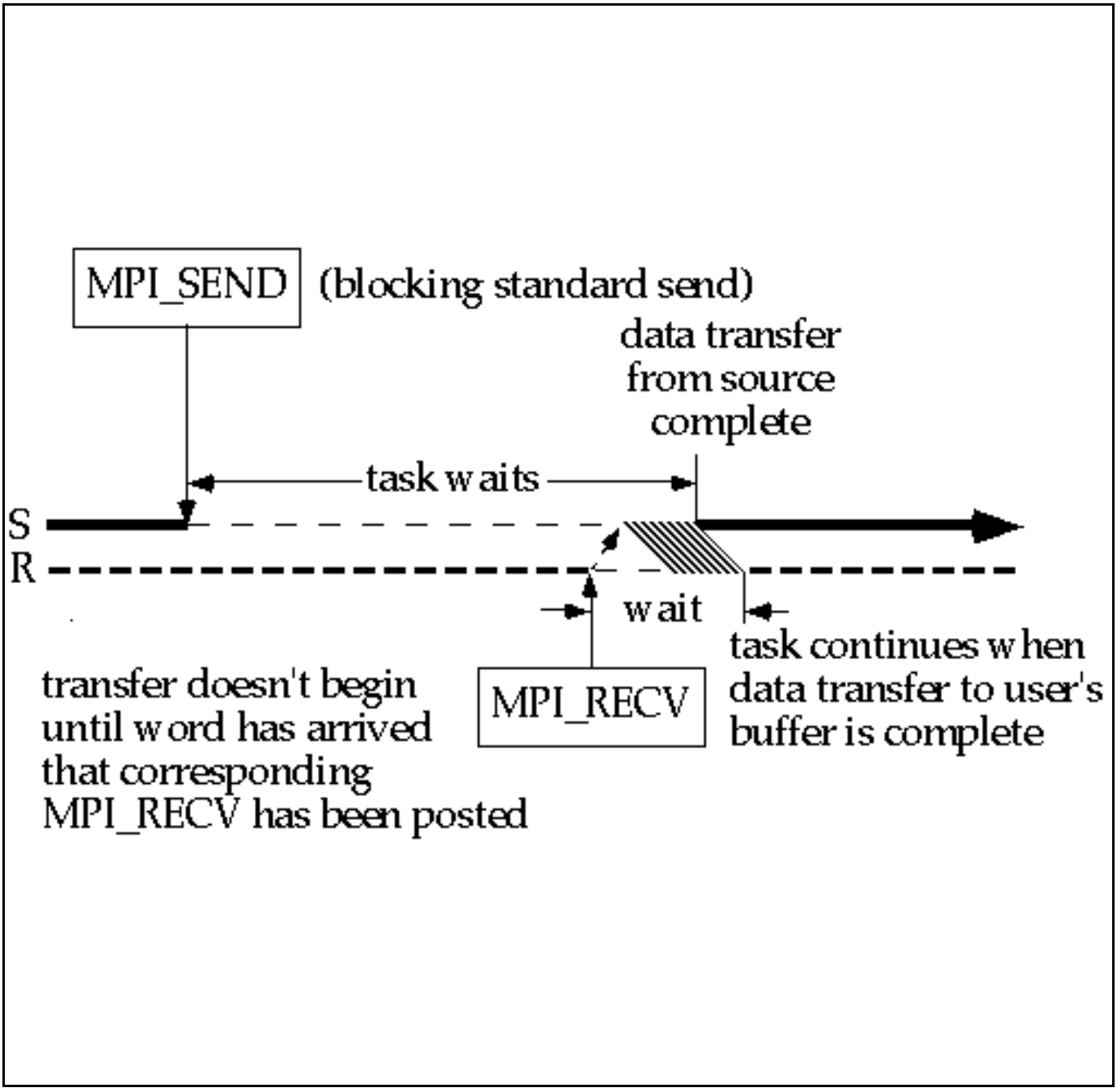
Quando **MPI_Send** é executado a mensagem é transmitida essencialmente igual ao modo "Synchronous".

Atenção: O valor "default" do tamanho do "buffer", diminui a medida que se aumenta o número de processos. (Até 16 ==> 4K)

Blocking Standard Send e Blocking Receive
Message <= 4K

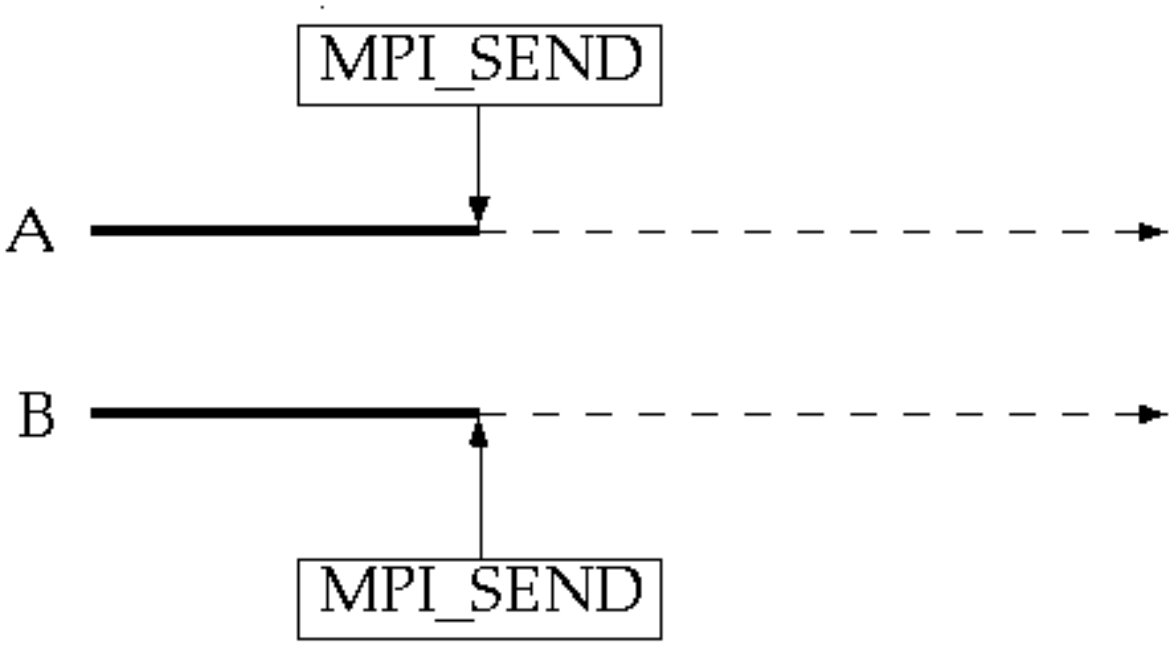


Blocking Standard Send e Blocking Receive
Message > 4K



9.2 - "Deadlock"

Fenômeno comum quando se utiliza "blocking communication". Acontece quando **todos os processos** estão aguardando por eventos que ainda não foram iniciados.



- Arrume sua aplicação, de maneira que, exista casamento entre **Send** e **Recv**;

- Utilize "non-blocking communication";

9.3 - Comunicação "Non-blocking"

Em uma comunicação **"blocking"**, a execução do programa é suspensa até o "system buffer" estar pronto para uso.

Quando se executa um **"blocking send"**, significa que o dado tem que ter sido enviado do "system buffer" para a rede, liberando o "buffer" para ser novamente utilizado.

Em uma comunicação **"non-blocking"**, a execução do programa continua imediatamente após ter sido iniciado a comunicação. O programador não tem idéia se a mensagem já foi enviada ou recebida.

Em uma comunicação **"non-blocking",** é necessário bloquear a continuação da execução do programa, ou averiguar o status do "system buffer", antes de reutilizá-lo.

MPI_Wait
MPI_Test

Todas as sub-rotinas **"non-blocking"**, possuem o prefixo **MPI_I**xxxx, e mais um parâmetro para identificar o status.

Non-Blocking Synchronous Send

C

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

FORTRAN

call MPI_ISSEND(buf,count,datatype,dest,tag,comm,request,ierror)

Non-Blocking Ready Send

C

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

FORTRAN

call MPI_IRSEND(buf,count,datatype,dest,tag,comm,request,ierror)

Non-Blocking Buffered Send

C

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

FORTRAN

call MPI_IBSEND(buf,count,datatype,dest,tag,comm,request,ierror)

Non-Blocking Standard Send

C

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

FORTRAN

call MPI_ISEND(buf,count,datatype,dest,tag,comm,request,ierror)

Non-Blocking Receive

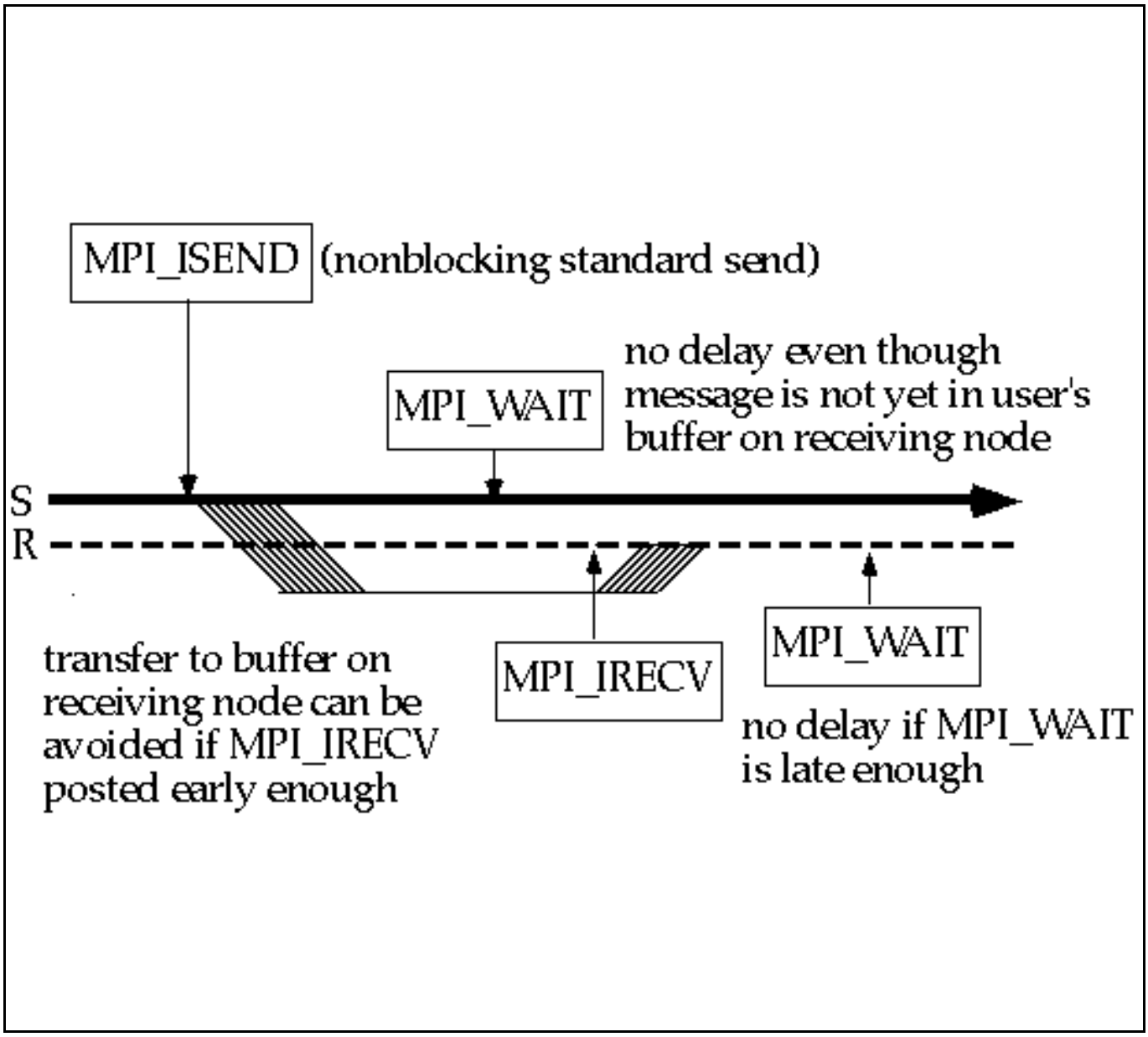
C

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

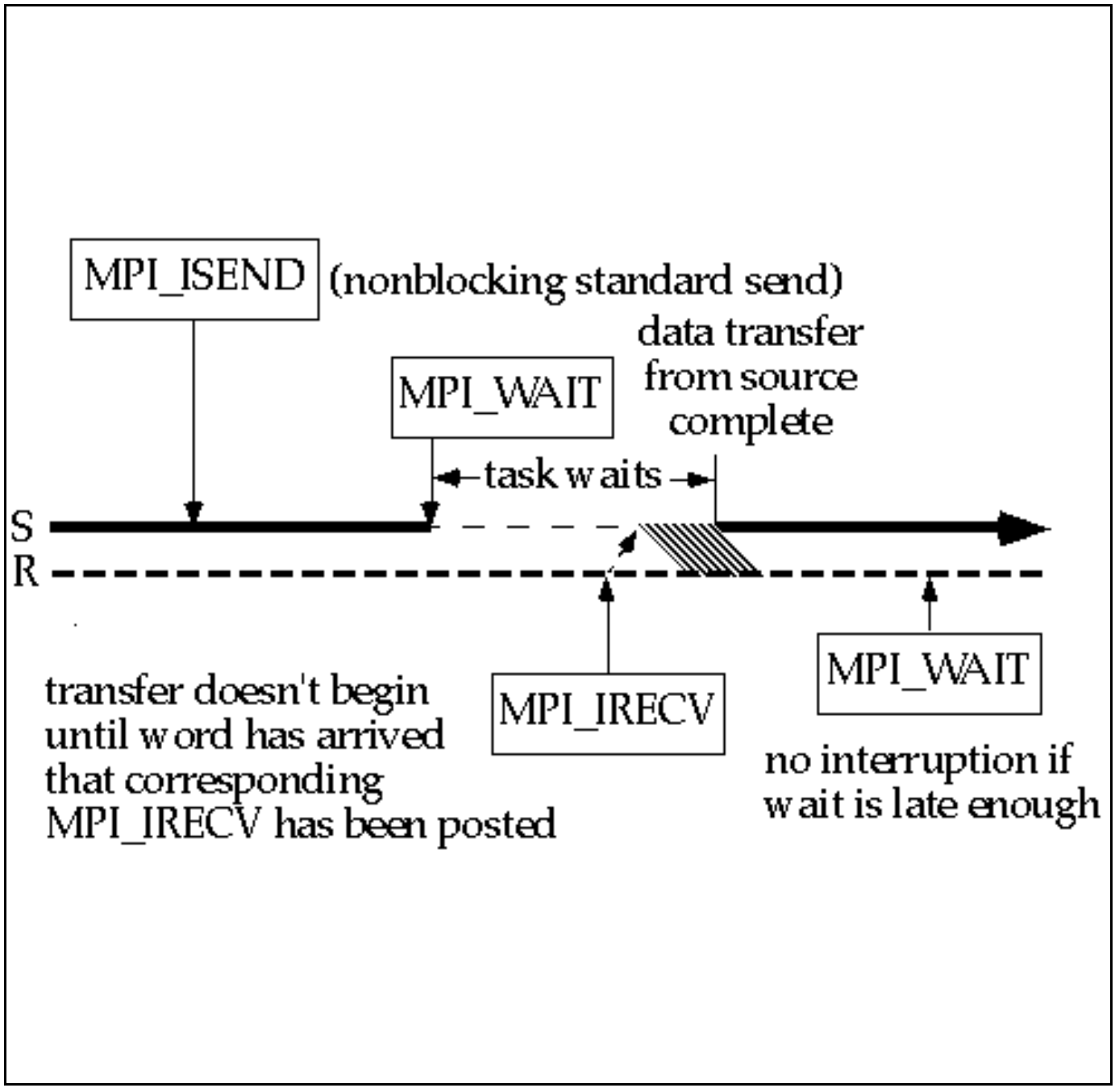
FORTRAN

call MPI_IRECV(buf,count,datatype,source,tag,comm,request,ierror)

Non-Blocking Standard Send e Non-Blocking Receive
Message <= 4K



Non-Blocking Standard Send e Non-Blocking Receive
Message > 4K



9.4 - Conclusão

- O modo **"Synchronous"**, é mais seguro, e ao mesmo tempo, mais portável. (Qualquer tamanho de mensagens, em qualquer arquitetura, em qualquer ordem de execução de "send" e "receive").

- O modo **"Ready"**, possui o menor índice total de "overhead", no entanto, a execução de um "receive" deve preceder a execução de um "send".

- O modo **"Buffered"**, elimina o "Synchronization overhead" e permite controle no tamanho do "buffer".

- O modo **"Standard"**, é a implementação básica do MPI, a biblioteca escolhe a melhor opção.

- As rotinas **"Non-blocking"**, possuem a vantagem de continuar a execução de um programa, mesmo se a mensagem ainda não tiver sido enviada. Elimina o "deadlock" e reduz o "synchronization overhead".

- As rotinas **"Non-blocking"**, necessitam de maior controle, que pode ser feito por rotinas auxiliares.

9.5 - Rotinas Auxiliares

MPI_Buffer_attach

C

int MPI_Buffer_attach (void*buf, int size)

FORTRAN

call MPI_BUFFER_ATTACH (buf, size, ierror)

buf Variável que identifica o endereço do "buffer";

size Variável inteira que determina o tamanho do "buffer" (em número de bytes);

ierror Variável inteira com status da execução da rotina.

Inicializa para o MPI, um "buffer" de tamanho especificado na memória do usuário, para ser utilizado no envio de mensagens.

Só é utilizado no modo de "Blocking" ou "Non-blocking" do "Buffered Send".

Somente um "buffer" poderá ser inicializado por processo durante a execução da aplicação.

MPI_Buffer_detach

C

int MPI_Buffer_detach (void**buffer, int*size)

FORTRAN

call MPI_BUFFER_DETACH (buf, size, ierror)

buf Variável que identifica o endereço do "buffer";

size Variável inteira que determina o tamanho do "buffer" (em número de bytes);

ierror Variável inteira com status da execução da rotina.

Elimina o "buffer" que foi inicializado anteriormente para o MPI.

Esta operação bloqueia a execução do programa até todas as mensagens terem sido transmitidas.

MPI_Wait

C

int MPI_Wait (MPI_Request*request, MPI_Status*status)

FORTRAN

call MPI_WAIT (request, status, ierror)

request Variável inteira que indica se o evento questionado foi executado. Parâmetro fornecido pelas rotinas "non-blocking send" ;

status Variável inteira de retorno com o status do evento questionado;

ierror Variável inteira com o status da execução da rotina.

Esta rotina bloqueia a execução do programa até que seja completada a ação identificada pela variável **request (null, inactive, active)**.

MPI_Waitall

C

int MPI_Waitall (int count,MPI_Request*array_of_request, MPI_Status*array_of_statuses)

FORTRAN

call MPI_WAITALL(count, array_of_request,array_of_ status,ierror)

request Vetor que indica se o evento questionado foi executado por todos os processos. Parâmetro fornecido pelas rotinas "non-blocking send" ;

status Vetor com o status do evento questionado;

ierror Variável inteira com o status da execução da rotina.

Esta rotina bloqueia a execução do processo até que todos os processos na lista **request** tenham executado uma operação de **receive**.

MPI_Test

C

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

FORTRAN

call MPI_TEST(request, flag, status, ierror)

request Variável inteira que identifica o evento questionado. Parâmetro fornecido pelas rotinas "non-blocking send" ;

flag Variável lógica, determina se a operação identificada pelo **request**, foi concluída ou não (**true** ou **false**);

status Variável inteira de retorno com o status do evento questionado;

ierror Variável inteira com o status da execução da rotina.

Esta rotina apenas informa se um operação "non-blocking send" foi concluída ou não.

MPI_Type_extent

C

int MPI_Type_extent(MPI_Datatype datatype,int*extent)

FORTRAN

call MPI_TEST_EXTENT(datatype, extent, ierror)

datatype Tipo do dado;

extent Variável inteira com a extensão reservada, em bytes, para o tipo do dado;

ierror Variável inteira com o status da execução da rotina.

Esta rotina retorna com a extensão do tipo de dado solicitado pelo **datatype**.

9.6 - Recomendações

- Em geral, é razoável iniciar uma programação MPI, utilizando-se de rotinas **"Non-blocking Standard Send"** e **"Non-blocking Receive"** (Elimina "deadlock", reduz "synchronization overhead" e geralmente possui boa performance);

- "Blocking" será necessário se o usuário necessitar sincronizar processos.

- É mais eficiente utilizar rotinas "blocking", ao invês de utilizar uma rotina "non-blocking" seguida de uma rotina "MPI_Wait".

- Se for necessário trabalhar com rotinas "blocking", pode ser vantajoso iniciar com o modo "synchronous", para depois passar para o modo "standard";

- Teste no modo "synchronous", possibilita a certeza de que o programa não necessitará de espaço extra de "system buffer".

- Um próximo passo seria analisar o código e avaliar a performance. Se "non-blocking receives" forem executados bem antes de seus correspondentes "sends", pode ser vantajoso utilizar o modo "ready";

- Se existir um elevado "synchronization overhead" durante a tarefa de envio das mensagens, especialmente com grandes mensagens, o modo "buffered" pode ser mais eficiente.

10 - COMUNICAÇÃO COLETIVA

Comunicação coletiva involve todos os processos em um grupo de processos.

O objetivo deste tipo de comunicação, é o de manipular um pedaço **comum** de informação.

As rotinas de comunicação coletiva foram montadas utilizando-se as rotinas de comunicação "point-to-point"

As rotinas de comunicação coletivas estão divididas em três categorias: **"synchronization", "data movement" e "global computation".**

Características básicas:

- Involve comunicação coordenada entre processos de um grupo, identificados por um **"communicator"**;
- Todas as rotinas efetuam **"block"**, até serem localmente finalizadas;
- A comunicação pode ou não, ser sincronizada;
- Não é necessário rotular as mensagens (**tags**).

10.1 - Sincronização

C

int MPI_Barrier (MPI_Comm comm)

FORTRAN

call MPI_BARRIER (comm, ierr)

comm Inteiro que determina o **"communicator"**;

ierr Inteiro que retorna com o status da execução da rotina.

Aplicações paralelas em ambiente de memória distribuída, as vezes, é necessário que ocorra sincronização implicita ou explicitamente. A rotina **MPI_Barrier**, sincroniza todos os processos de um grupo ("communicator").

Um processo de um grupo que utilize **MPI_Barrier**, para de executar, até que todos os processos do mesmo grupo também executem um **MPI_Barrier**.

10.2 - "Data Movement"

10.2.1 - "Broadcast"

C

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

FORTRAN

call MPI_BCAST (buffer, count, datatype, comm, ierr)

buffer Endereço inicial do dado a ser enviado;

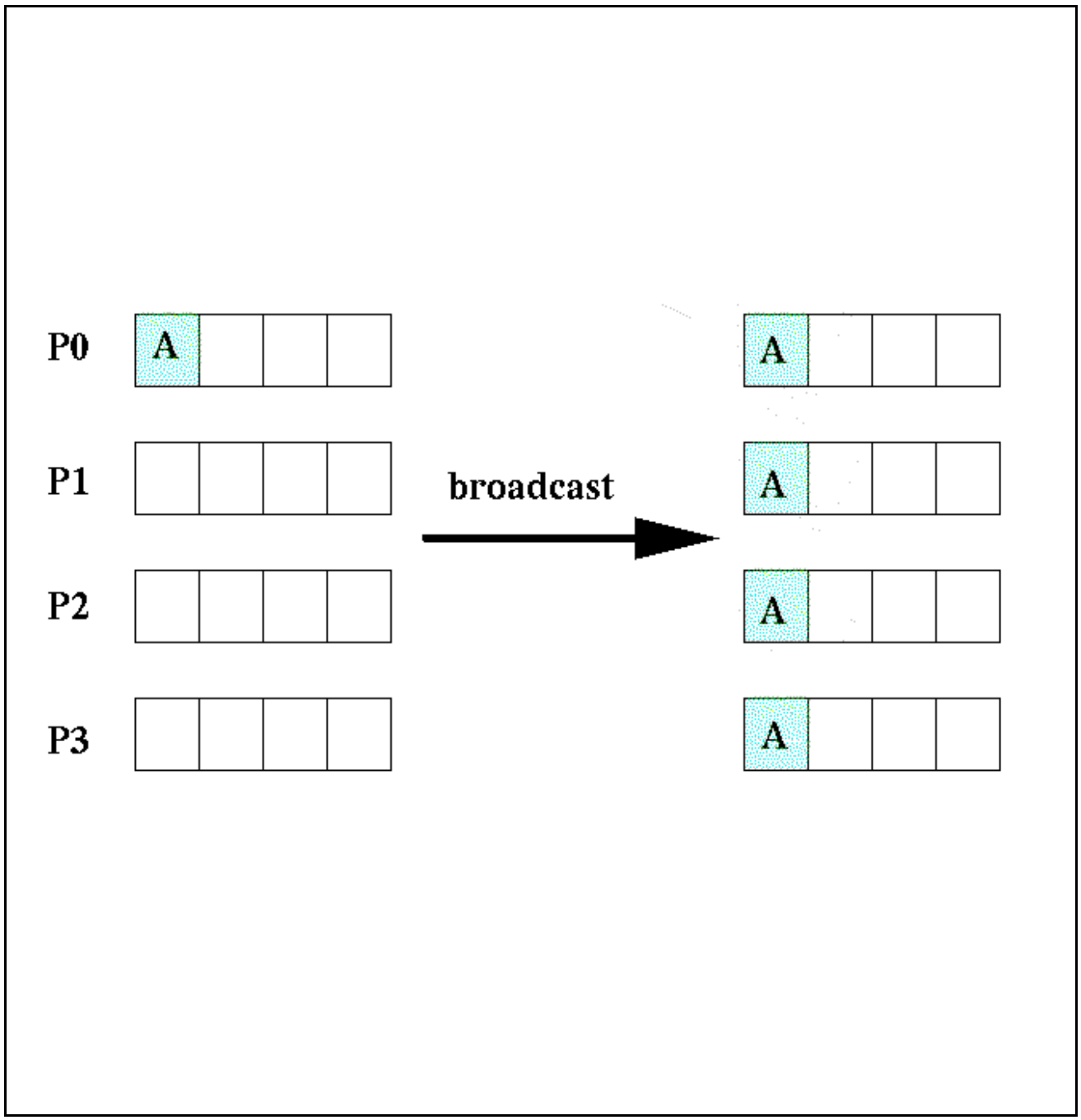
count Inteiro que indica o número de elementos no **buffer**;

datatype Constante MPI que identifica o tipo de dado dos elementos no **buffer**;

root Inteiro com a identificação do processo que irá efetuar um **broadcast;**

comm Identificação do **communicator**.

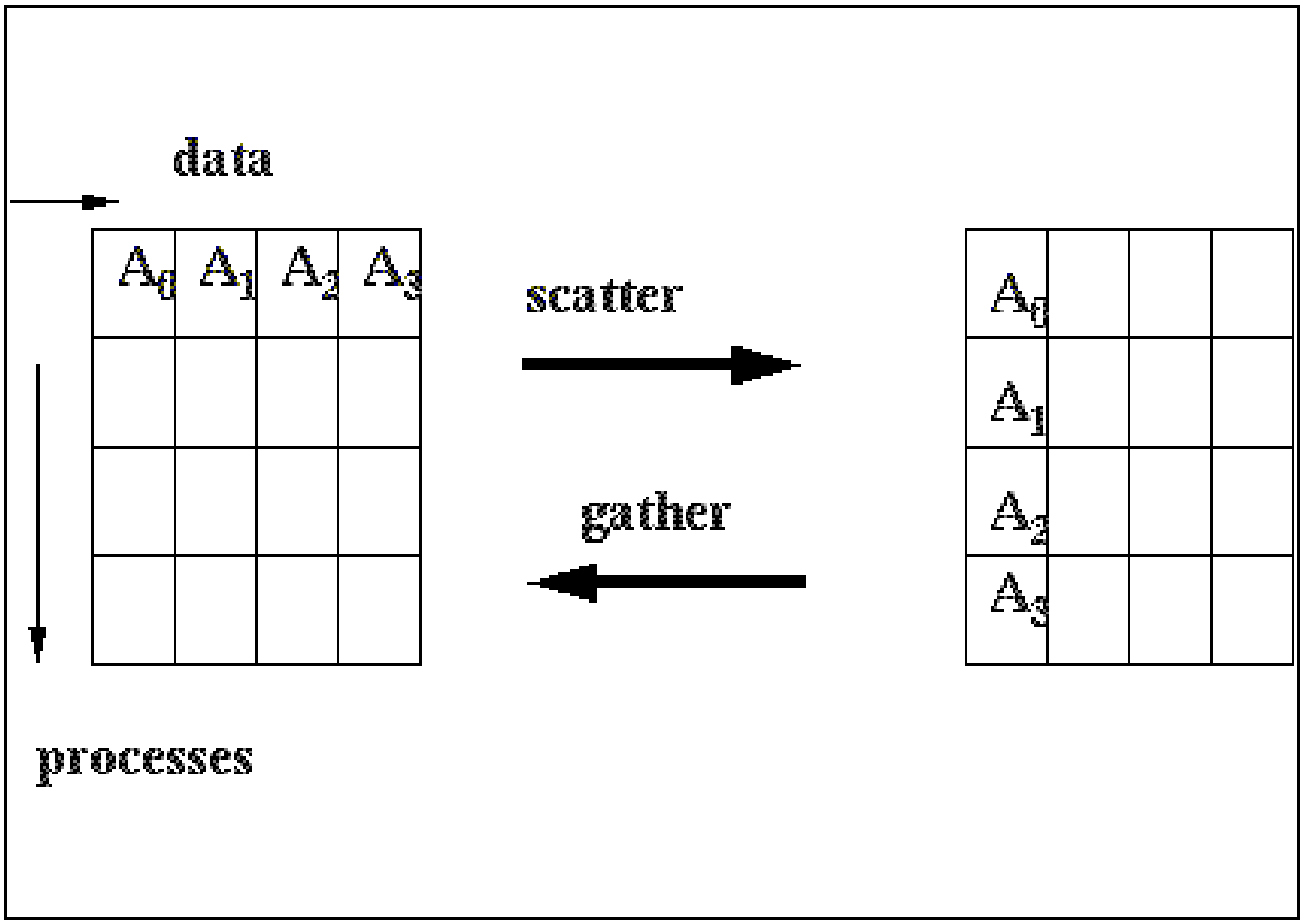
Rotina que permite a um processo enviar dados, de imediato, para todos os processos de um grupo. Todos os processos do grupo, deverão executar um **MPI_Bcast**, com o mesmo **comm** e **root**.


P0
A
P1
P2
P3
broadcast
A
A
A
A

10.2.2 - "Gather" e "Scatter"

Se um processo necessita distribuir dados em *n* segmentos iguais, onde o enésimo segmento é enviado para enésimo processo num grupo de *n* processos, utiliza-se a rotina de **SCATTER**.

Por outro lado, se um único processo necessita coletar os dados distribuidos em *n* processos de um grupo. utiliza-se a rotina de **GATHER**.



MPI_SCATTER

C

int MPI_Scatter(void*sbuf, int scount, MPI_Datatype stype, void*rbuf, int rcount,MPI_Datatype rtype,int root,MPI_Comm comm)

FORTRAN

call MPI_SCATTER(sbuf,scount,stype,rbuf,rcount,rtype,root,comm, ierr)

sbuf Endereço dos dados que serão distribuídos("send buffer");

scount Número de elementos distribuídos para cada processo;

stype Tipo de dado que será distribuído;

rbuf Endereço aonde os dados serão armazenados ("receive buffer");

rcount Número de elementos distribuídos;

rtype Tipo de dado distribuído;

root Identificação do processo que irá distribuir os dados;

comm Identificação do "communicator".

MPI_GATHER

C

int MPI_Gather(void*sbuf, int scount, MPI_Datatype stype, void*rbuf, int rcount,MPI_Datatype rtype,int root,MPI_Comm comm)

FORTRAN

call MPI_GATHER(sbuf,scount,stype,rbuf,rcount,rtype,root,comm,ierr)

sbuf Endereço inicial dos dados a serem coletados("send buffer");

scount Número de elementos a serem coletados;

stype Tipo de dado a ser coletado;

rbuf Endereço aonde o dado será armazenado("receive buffer");

rcount Número de elementos por processo do grupo;

rtype Tipo de dado coletado;

root Identificação do processo que ira coletar os dados;

comm Identificação do "communicator".

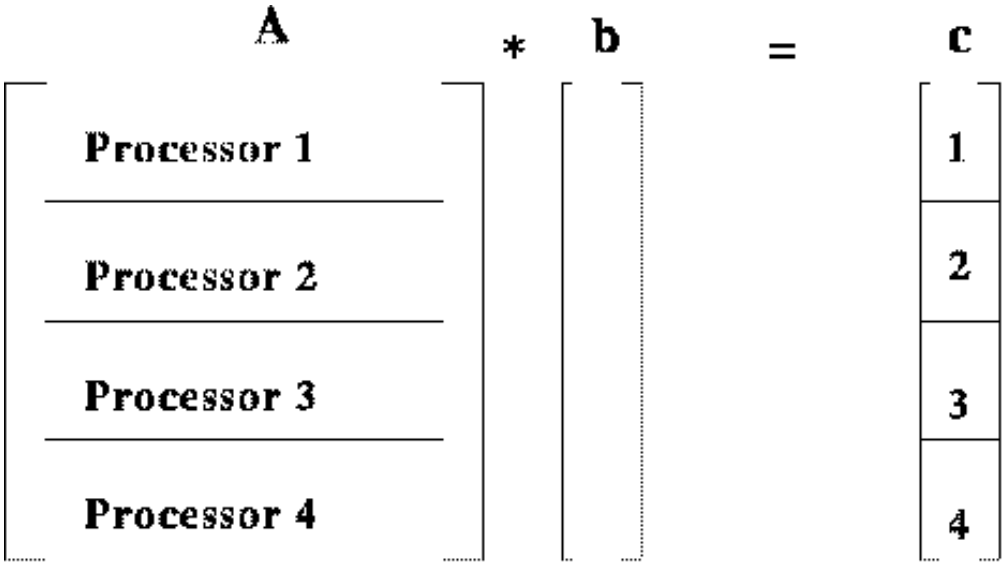
ierr Variável inteira de retorno com o status da rotina.

Exemplo 1

```
DIMENSION A(25,100), b(100), cpart(25), ctotal(100)
INTEGER root
DATA root/0/

DO I=1,25
        cpart(I)=0.
        DO K=1,100
                cpart(I) = cpart(I) + A(I,K)*b(K)
        END DO
END DO
```

call MPI_GATHER(cpart,25,MPI_REAL,ctotal,25,MPI_REAL,root, MPI_COMM_WORLD,ierr)

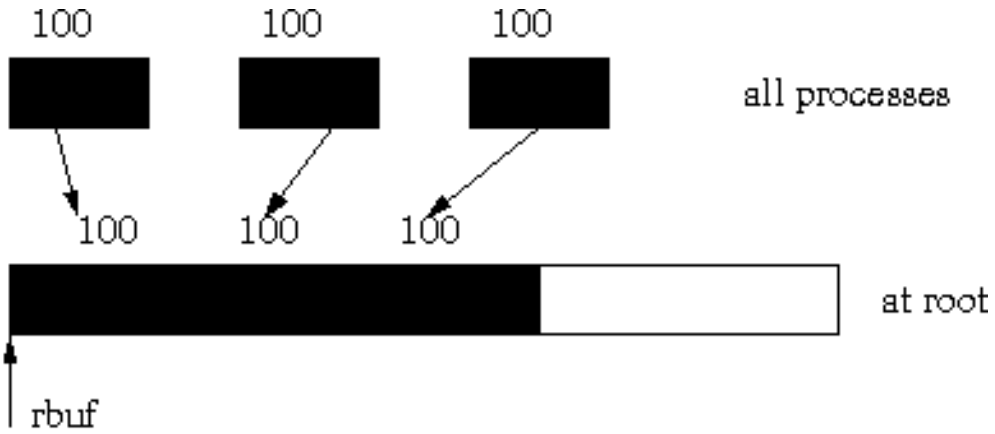


A: Matriz distribuída por linhas;

B: Vetor compartilhado por todos os processos;

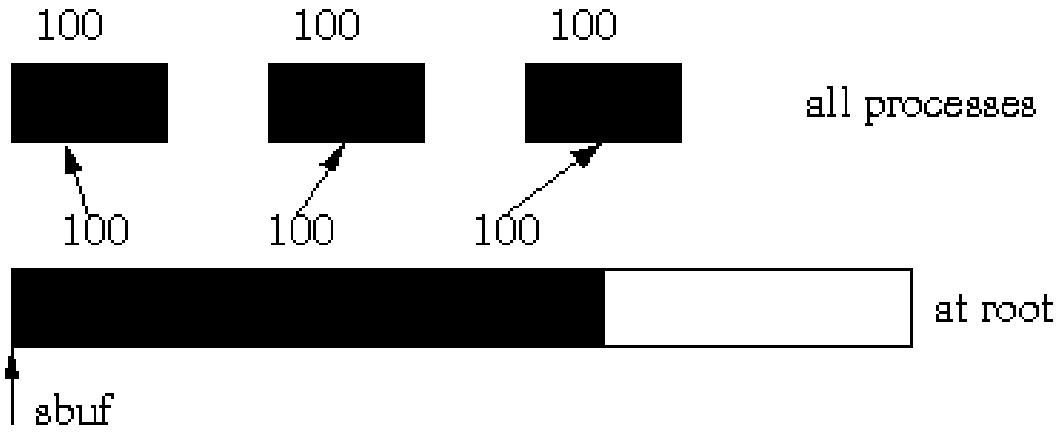
C: Vetor atualizado por cada processo independentemente.

Exemplo 2




```
real a(100), rbuf(MAX)
        .          .          .
        .          .          .
        .          .          .
call mpi_gather(a,100,MPI_REAL,rbuf,100,MPI_REAL,root,
                comm, ierr)
```

Exemplo 3




```
real sbuf(MAX), rbuf(100)
        .           .           .
        .           .           .
        .           .           .
call mpi_scatter(sbuf,100,MPI_REAL,rbuf,100,MPI_REAL,
                 root,comm,ierr)
```

10.2.3 - "Allgather"

C

int MPI_Allgather(void*sbuf, int scount, MPI_Datatype stype, void*rbuf, int rcount,MPI_Datatype rtype,MPI_Comm comm)

FORTRAN

call MPI_ALLGATHER(sbuf,scount,stype,rbuf,rcount,rtype,comm,ierr)

sbuf Endereço inicial dos dados a serem coletados("send buffer");

scount Número de elementos a serem coletados;

stype Tipo de dado a ser coletado;

rbuf Endereço aonde o dado será armazenado("receive buffer");

rcount Número de elementos por processo do grupo;

rtype Tipo de dado coletado;

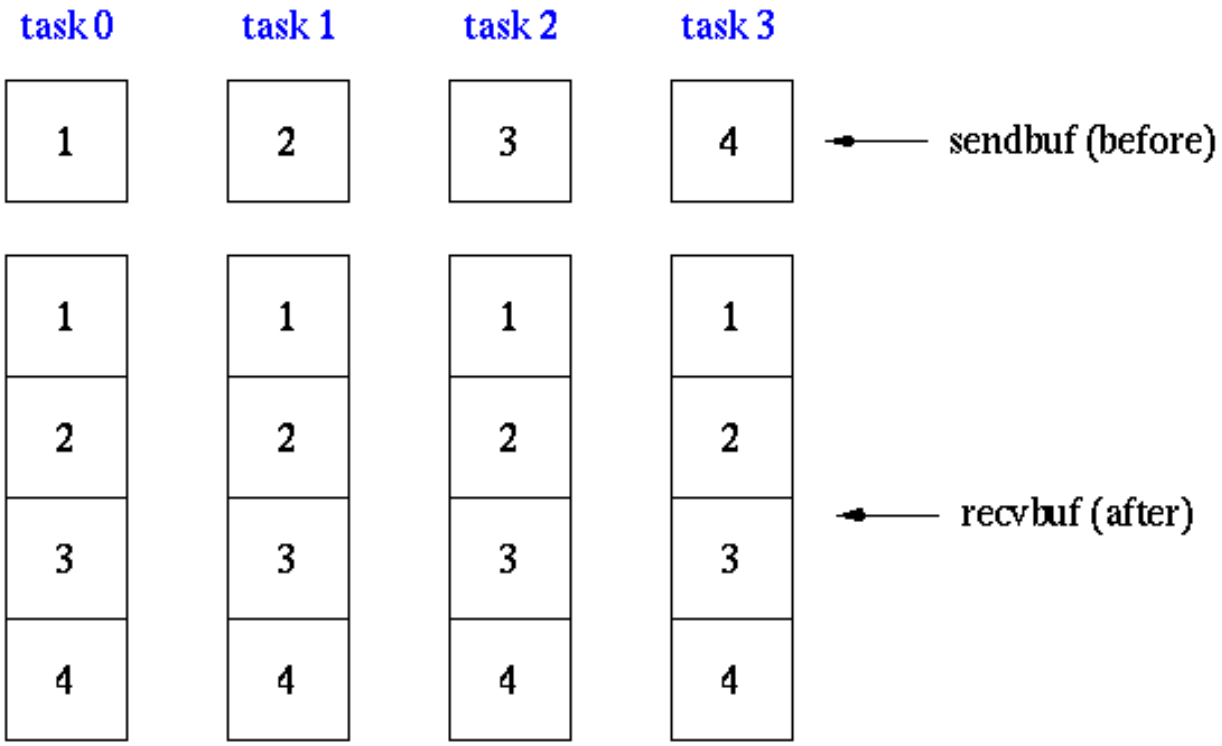
comm Identificação do "communicator".

ierr Variável inteira de retorno com o status da rotina.

Esta rotina ao ser executada faz com que todos os processos coletem os dados de cada processo da aplicação. Seria similar a cada processo efetuar um "brodcast".

MPI_Allgather

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



10.2.4 - "All to All"

C

int MPI_Alltoall(void*sbuf, int scount, MPI_Datatype stype, void*rbuf, int rcount,MPI_Datatype rtype,MPI_Comm comm)

FORTRAN

call MPI_ALLTOALL(sbuf,scount,stype,rbuf,rcount,rtype,comm,ierr)

sbuf Endereço inicial dos dados a serem coletados("send buffer");

scount Número de elementos a serem coletados;

stype Tipo de dado a ser coletado;

rbuf Endereço aonde o dado será armazenado("receive buffer");

rcount Número de elementos por processo do grupo;

rtype Tipo de dado coletado;

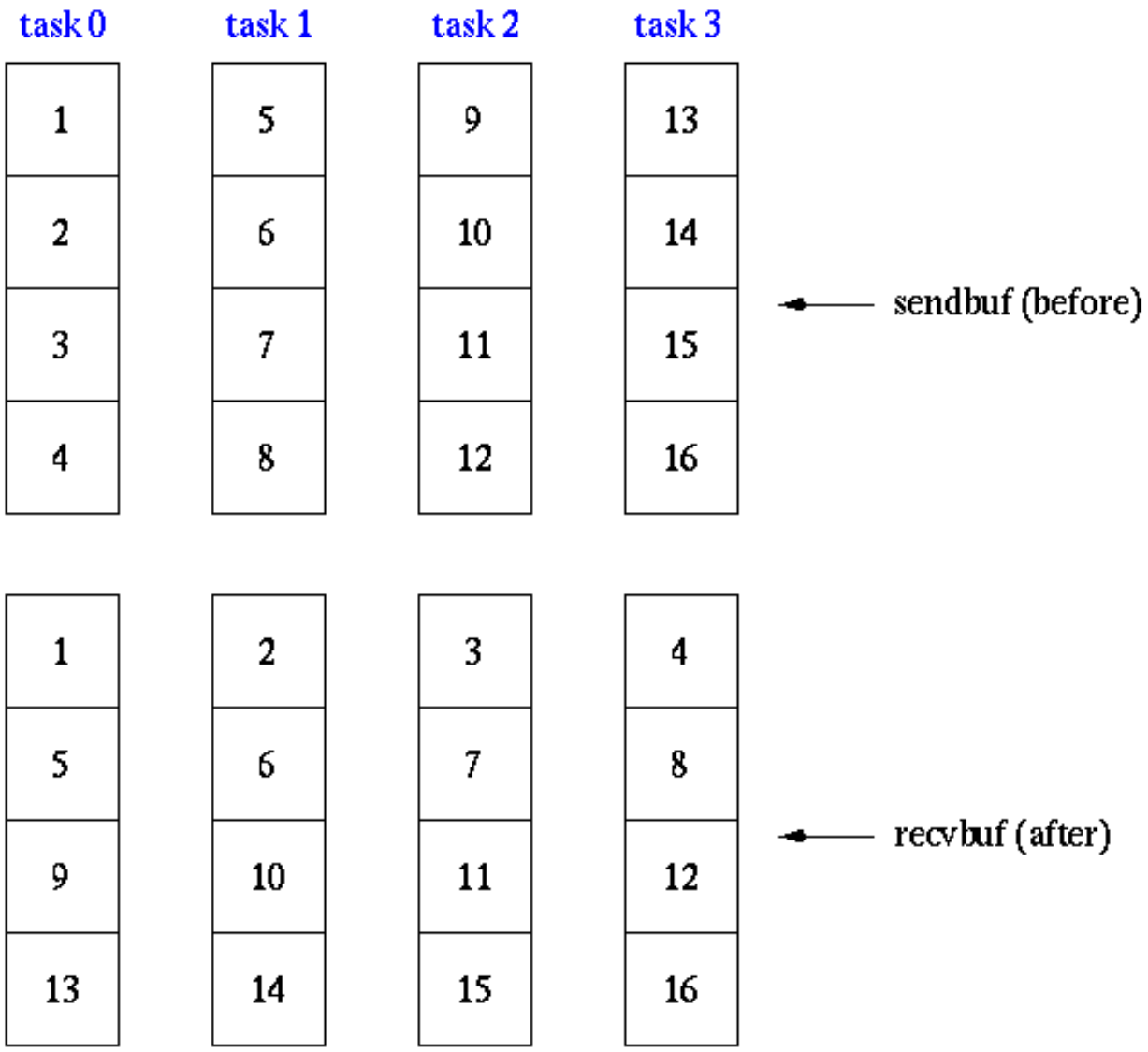
comm Identificação do "communicator".

ierr Variável inteira de retorno com o status da rotina.

Esta rotina ao ser executada faz com que cada processo envie seus dados para todos os outros processos da aplicação. Seria similar a cada processo efetuar um "scatter".

MPI_Alltoall

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



10.2.5 - Rotinas de Computação Global

Uma das ações mais uteis em operações coletivas, são as operações globais de redução ou combinação de operações.

O resultado parcial de cada processo em um grupo é combinado e retornado para um específico processo, utilizando-se algun **tipo de função de operação.**

Tabela com Funções Pré-definidas de Operações de Redução

FUNÇÃO	RESULTADO	C	FORTRAN
MPI_MAX	**valor máximo**	**integer,float**	**integer,real,complex**
MPI_MIN	**valor mínimo**	**integer,float**	**integer,real,complex**
MPI_SUM	**somatório**	**integer,float**	**integer,real,complex**
MPI_PROD	**produto**	**integer,float**	**integer,real,complex**

MPI_REDUCE

C

int MPI_Reduce(void*sbuf, void*rbuf,int count, MPI_Datatype stype, MPI_Op op,int root,MPI_Comm comm)

FORTRAN

call MPI_REDUCE(sbuf,rbuf,count,stype,op,root,comm,ierr)

sbuf Endereço do dado que fará parte de uma operação de redução("send buffer");

rbuf Endereço da variável que armazenará o resultado da redução("receive buffer");

count Número de elementos que farão parte da redução;

stype Tipo dos dados na operação de redução;

op Tipo da operação de redução;

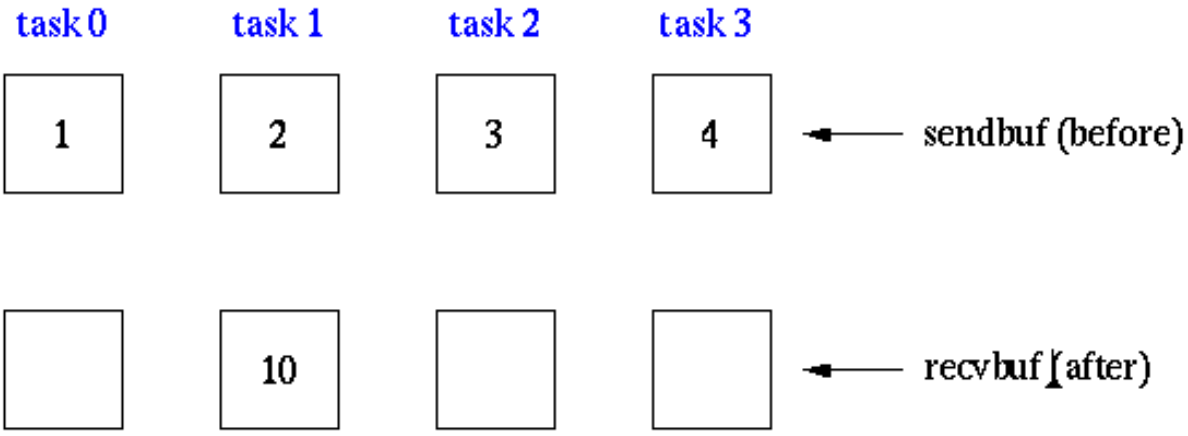
root Identificação do processo que irá receber o resultado da operação de redução;

comm Identificação do "communicator".

Exemplo 1

MPI_Reduce

```
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 2

Simulação Dinâmica de Florestas

- Cada processo numa simulação dinâmica de florestas, calcula o valor máximo de altura de arvore por região;

- O processo principal, que gera o resultado final, necessita saber, a altura máxima global (todas as regiões).

```
INTEGER  maxht, globmax
        .
        .       (cálculos que determinam a altura máxima)
        .
call MPI_REDUCE(maxht, globmax, 1, MPI_INTEGER, MPI_MAX,
                        0, MPI_COMM_WORLD, ierr)

IF (taskid.eq.0)  then
        .
        .       (Gera relatório com os resultados)
        .
END IF
```

11 - REFERÊNCIAS

Os dados apresentados nesta terceira parte, foram obtidos dos seguintes documentos:

1 - Message Passing Interface (MPI)
MHPCC - Maui High Performance Computing Center
Blaise Barney - August 29, 1996

2 - Programming Languages and Tools: MPI
CTC - Cornell Theory Center
April, 1996

3 - MPI: A Message-Passing Interface Standard
University of Tennesse, Knoxville, Tennesse
May 5, 1994

4 - MPI: The Complete Reference
The MIT Press - Cambridge, Massachusetts
Marc Snir, Steve Otto, Steven Huss-Lederman,
David Walker, Jack Dongarra
1996

1º LABORATÓRIO
Rotinas Básicas do MPI

A idéia deste laboratório, será a de manipular as 6 rotinas básicas do MPI, e analisar a performance na execução dos programas.

Exercício 1

1 - Caminhe para o diretório com o primeiro exercício do laboratório.

%cd ./mpi/lab01/ex1

2 - Compile o programa **hello.f** ou **hello.c** com o script de compilação do MPI para o FORTRAN ou para o C.

%mpif77 hello.f -o hello
ou
%mpicc hello.c -o hello

3 - Execute o programa várias vezes, alterando a opção de número de processos inicializados.

%mpirun -np *n* hello

n = número de processos

4 - Crie um arquivo com uma configuração de máquinas desejada. Execute novamente o programa, mas direcionando a execução, para utilizar o arquivo de máquinas criado.

%mpirun -np *n* -machinefile *arquivo* hello

n = número de processos
arquivo = nome do arquivo de máquinas

5 - Execute novamente o programa, mas que não seja criado um processo na máquina local (máquina logada).

%mpirun –np *n* -nolocal hello

Exercício 2

1 - Caminhe para o diretório com o segundo exercício do laboratório.

%cd ./mpi/lab01/ex2

Programa hello.ex1.c e hello.ex1.f

Estes programas seguem o modelo SPMD ("Single Program Multiple Data), ou seja, o mesmo programa executa como um processo mestre e como um processo escravo.

O processo mestre envia uma mensagem ("Hello world") para todos os processos escravos e imprime a mensagem na saída padrão. Os processos escravos recebem a mensagem e, tambêm, imprimem a mensagem na saída padrão.

O exercício está em alterar o programa, de maneira que, cada processo escravo ao invês de imprimir a mensagem, devolva-a ao processo mestre. O programa mestre recebe a mensagem de volta e imprime como "Hello, back".

2 - Adicione as rotinas necessárias ao programa em FORTRAN ou em C **substituindo as linhas que possuem setas por rotinas do MPI, e complete a lógica de alguns comandos**. Compile e execute o programa.

hello.ex1.f

```
C Program hello.ex1.f
C Parallel version using MPI calls
C Modified from basic version so that workers send back
C a message to the master, who prints out a message
C for each worker
C RLF 10/11/95
      program hello
      include 'mpif.h'
      integer me,nt,mpierr,tag,status(MPI_STATUS_SIZE)
      integer rank
      character(12) message, inmsg
      call MPI_INIT(mpierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,nt,mpierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,me,mpierr)
      tag = 100
      if(me .eq. 0) then
        message = 'Hello, world'
        do i=1,nt-1
      call MPI_SEND(message,12,MPI_CHARACTER,i,tag,
     .      MPI_COMM_WORLD,mpierr)
      enddo
      write(6,*)'node',me,':',message

===>    do ...  (REGRA DO LOOP)
===>            ( ROTINA MPI )

        write(6,*) 'node', rank, ':Hello, back'
      enddo
    else
          call MPI_RECV(inmsg,12,MPI_CHARACTER,0,tag,
     .     MPI_COMM_WORLD,status,mpierr)

===>    (ROTINA MPI)

    endif
    CALL MPI_FINALIZE(mpierr)
    end
```

hello.ex1.c

```
 /* hello.ex1.c
 * Parallel version using MPI calls
 * Modified from basic version so that workers send back
 * a message to the master, who prints out a message
 * for each worker
 * RLF 10/11/95
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
          char message[20];
          int i,rank, size, type=99;
          MPI_Status status;
          MPI_Init(&argc, &argv);
          MPI_Comm_size(MPI_COMM_WORLD,&size);
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
          if(rank == 0) {
            strcpy(message, "Hello, world");
            for (i=1; i<size; i++)
            MPI_Send(message, 13, MPI_CHAR, i,
                        type, MPI_COMM_WORLD);
           printf("node %d : %.13s\n", rank, message);

 ===>     for  ...   (REGRA DO LOOP)
 ===>              (ROTINA MPI)

           printf("node %d : Hello back\n", rank);
           }
          }
          else {
           MPI_Recv(message, 20, MPI_CHAR, 0,
                        type, MPI_COMM_WORLD, &status);

===>      (ROTINA MPI)
        }
          MPI_Finalize();
}
```

Exercício 3

1 - Caminhe para o diretório com o terceiro exercício do laboratório.

%cd ./mpi/lab01/ex3

Um programa pode se utilizar do parâmetro **tag** em rotinas de send e receive para distinguir as mensagens que estão sendo enviadas ou recebidas.

2 - Adicione as rotinas necessárias ao programa em FORTRAN ou em C **substituindo as linhas que possuem setas por rotinas do MPI,** de maneira que, o processo mestre envie duas mensagens ("Hello" e "World") para cada processo escravo, utilizando-se de duas **tags** diferentes.

3 - Os processos escravos deverão receber as mensagens na ordem invertida e irão imprimir o resultado na saída padrão. Compile e execute o programa.

hello.ex2.f

```
C Program hello.ex2.f
C Parallel version using MPI calls.
C Modified from basic version so that workers send back a message to the
C master, who prints out a message for each worker.  In addition, the
C master now sends out two messages to each worker, with two different
C tags, and the worker receives the messages in reverse order.
C RLF 10/11/95
      program hello
      include 'mpif.h'
      integer me,nt,mpierr,tag,status(MPI_STATUS_SIZE)
      integer rank, tag2
      character(6) message1, message2, inmsg1, inmsg2
      call MPI_INIT(mpierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,nt,mpierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,me,mpierr)
      tag = 100
      tag2 = 200
      if(me .eq. 0) then
        message1 = 'Hello,'
        message2 = 'world'
        do i=1,nt-1
===>      (ROTINA MPI)
===>      (ROTINA MPI)
      enddo
      write(6,*)'node',me,':',message1,' ',message2
    else
===>      (ROTINA MPI)
===>      (ROTINA MPI)
       write(6,*) 'node', me, ':', inmsg1,' ',inmsg2
    endif
    CALL MPI_FINALIZE(mpierr)
    end
```

hello.ex2.c

```
/ * hello.ex2.c
 * Parallel version using MPI calls
 * Modified from basic version so that workers send back
 * a message to the master, who prints out a message
 * for each worker
 * In addition, the master now sends out two messages to each worker, with
 * two different tags, and each worker receives the messages
 * in reverse order.
 * RLF 10/23/95
*/
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
          char message1[7], message2[7];
          int i,rank, size, type=99;
          int type2=100;
          MPI_Status status;
          MPI_Init(&argc, &argv);
          MPI_Comm_size(MPI_COMM_WORLD,&size);
          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
          if(rank == 0) {
            strcpy(message1, "Hello,");
            strcpy(message2, "world");
            for (i=1; i<size; i++) {
 ===>         (ROTINA MPI)
 ===>         (ROTINA MPI)
        }
        printf("node %d : %.7s %.7s\n", rank, message1, message2);
          }
          else {
 ===>         (ROTINA MPI)
 ===>         (ROTINA MPI)
       printf("node %d : %.7s , %.7s\n", rank, message1, message2);
       }
          MPI_Finalize();
}
```

Exercício 4

1 - Caminhe para o diretório com o quarto exercício do laboratório.

%cd ./mpi/lab01/ex4

Neste exercício, o processo mestre inicializa um vetor e distribiu partes deste vetor para vários processos escravos. Cada processo escravo recebe a sua parte do vetor principal, efetua um cálculo bastante simples e devolve os dados para o processo mestre.

2 - Adicione as rotinas necessárias ao programa em FORTRAN ou em C **substituindo as linhas que possuem setas por rotinas do MPI.** Compile e execute o programa.

mpi.ex1.f

```
C **************************************************************************
C FILE: mpl.ex1.f
C DESCRIPTION:
C   In this simple example, the master task initiates numtasks-1 number of
C   worker tasks.  It then distributes an equal portion of an array to each
C   worker task.  Each worker task receives its portion of the array, and
C   performs a simple value assignment to each of its elements. The value
C   assigned to each element is simply that element's index in the array+1.
C   Each worker task then sends its portion of the array back to the master
C   task.  As the master receives back each portion of the array, selected
C   elements are displayed.
C AUTHOR: Blaise Barney
C LAST REVISED:  6/10/93
C LAST REVISED:  1/10/94 changed API to MPL   Stacy Pendell
C CONVERTED TO MPI: 11/12/94 by     Xianneng Shen
C **************************************************************************
      program example1_master

====>    INCLUDE MPI

      integer status(MPI_STATUS_SIZE)
      integer   ARRAYSIZE
      parameter (ARRAYSIZE = 60000)
      parameter (MASTER = 0)
      integer  numtask, numworkers, taskid, dest, index, i,
     &         arraymsg, indexmsg, source, chunksize,
     &         int4, real4
      real*4   data(ARRAYSIZE), result(ARRAYSIZE)
C ************************ initializations ***********************************
C Find out how many tasks are in this partition and what my task id is.  Then
C define the number of worker tasks and the array partition size as chunksize.
C Note:  For this example, the MP_PROCS environment variable should be set
C to an odd number...to insure even distribution of the array to numtasks-1
C worker tasks.
C
**************************************************************************
*
```

====> ROTINA MPI DE INICIALIZAÇÃO (ierr)
====> ROTINA MPI DE IDENTIFICACAORLD (taskid, ierr)
====> ROTINA MPI PARA DETERMINAR O NUMERO DE PROCESSOS (numtasks, ierr)

```
      write(*,*)'taskid =',taskid
      numworkers = numtasks-1
      chunksize = (ARRAYSIZE / numworkers)
      arraymsg = 1
      indexmsg = 2
      int4 = 4
      real4 = 4
C ***************************** master task ************************************
      if (taskid .eq. MASTER) then
      print *, '************ Starting MPI Example 1 *************'
C     Initialize the array
      do 20 i=1, ARRAYSIZE
        data(i) =  0.0
 20   continue
C     Send each worker task its portion of the array
      index = 1
      do 30 dest=1, numworkers
      write(*,*) 'Sending to worker task', dest
```

====> ROTINA MPI DE ENVIO DE DADOS (index,dest,ierr)
====> ROTINA MPI DE ENVIO DE DADOS (data(index),chunksize,dest,ierr)

```
      index = index + chunksize
 30   continue
C     Now wait to receive back the results from each worker task and print
C     a few sample values
      do 40 i=1, numworkers
        source = i
```

====> ROTINA MPI PARA RECEBER DADOS (index,source,status,ierr)
====> ROTINA MPI PARA RECEBER DADOS (result(index),chunksize,source,status,ierr)

```
      print *, '-------------------------------------------------'
      print *, 'MASTER: Sample results from worker task ', source
      print *, '   result[', index, ']=', result(index)
      print *, '   result[', index+100, ']=', result(index+100)
      print *, '   result[', index+1000, ']=', result(index+1000)
      print *, ' '
 40   continue
     print *, 'MASTER: All Done!'
     endif
C **************************** worker task ************************************
     if (taskid .gt. MASTER) then
C    Receive my portion of array from the master task */
```

====> ROTINA MPI PARA RECEBER DADOS (index,MASTER,status,ierr)
====> ROTINA MPI PARA RECEBER DADOS
(result(index),chunksize,MASTER,status,ierr)

```
C    Do a simple value assignment to each of my array elements
     do 50 i=index, index + chunksize
      result(i) = i + 1
 50   continue
C    Send my results back to the master
```

====> ROTINA MPI DE ENVIO DE DADOS (index,MASTER,ierr)
====> ROTINA MPI DE ENVIO DE DADOS
(result(index),chunksize,MASTER,ierr)

```
     endif
```

====> ROTINA MPI DE FINALIZAÇÃO (ierr)

```
     end
```

mpi.ex1.c

```
/******************************************************************************
**
* FILE: mpl.ex1.c
* DESCRIPTION:
*   In this simple example, the master task initiates numtasks-1 number of
*   worker tasks.  It then distributes an equal portion of an array to each
*   worker task.  Each worker task receives its portion of the array, and
*   performs a simple value assignment to each of its elements. The value
*   assigned to each element is simply that element's index in the array+1.
*   Each worker task then sends its portion of the array back to the master
*   task.  As the master receives back each portion of the array, selected
*   elements are displayed.
* AUTHOR: Blaise Barney
* LAST REVISED:  09/14/93 for latest API changes  Blaise Barney
* LAST REVISED:  01/10/94 changed API to MPL      Stacy Pendell
* CONVERTED TO MPI: 11/12/94 by                   Xianneng Shen
******************************************************************************/
#include <stdio.h>

====>    INCLUDE MPI

#define ARRAYSIZE   60000
#define MASTER                  0       /* taskid of first process */

MPI_Status status;
main(int argc, char **argv)
{
int     numtasks,               /* total number of MPI process in partitiion */
        numworkers,             /* number of worker tasks */
        taskid,                 /* task identifier */
        dest,                   /* destination task id to send message */
        index,                  /* index into the array */
        i,                  /* loop variable */
        arraymsg = 1,           /* setting a message type */
        indexmsg = 2,           /* setting a message type */
        source,                 /* origin task id of message */
        chunksize;              /* for partitioning the array */
float   data[ARRAYSIZE],        /* the intial array */
        result[ARRAYSIZE];      /* for holding results of array operations */
```

```
/************************* initializations ***********************************
* Find out how many tasks are in this partition and what my task id is.  Then
* define the number of worker tasks and the array partition size as chunksize.
* Note:  For this example, the MP_PROCS environment variable should be set
* to an odd number...to insure even distribution of the array to numtasks-1
* worker tasks.
*****************************************************************************
**/
```

====> ROTINA MPI DE INICIALIZAÇÃO (&argc, &argv)
====> ROTINA MPI DE IDENTIFICACAO (&taskid)
====> ROTINA MPI PARA DETERMINAR O NUMERO DE PROCESSOS (&numtasks)

```
numworkers = numtasks-1;
chunksize = (ARRAYSIZE / numworkers);
/***************************** master task ************************************/
if (taskid == MASTER) {
  printf("\n*********** Starting MPI Example 1 ***********\n");
  printf("MASTER: number of worker tasks will be= %d\n",numworkers);
  fflush(stdout);

  /* Initialize the array */
  for(i=0; i<ARRAYSIZE; i++)
    data[i] =  0.0;
  index = 0;

  /* Send each worker task its portion of the array */
  for (dest=1; dest<= numworkers; dest++) {
    printf("Sending to worker task= %d\n",dest);
    fflush(stdout);
```

====> ROTINA MPI DE ENVIO DE DADOS (&index,dest)
====> ROTINA MPI DE ENVIO DE DADOS (&data[index],chunksize,dest)

```
    index = index + chunksize;
    }
  /* Now wait to receive back the results from each worker task and print */
  /* a few sample values */
  for (i=1; i<= numworkers; i++) {
    source = i;
```

```
====>   ROTINA MPI PARA RECEBER DADOS (&index,source,&status)
====>   ROTINA MPI PARA RECEBER DADOS
          (&result[index],chunksize,source,&status)

   printf("-------------------------------------------------\n");
   printf("MASTER: Sample results from worker task = %d\n",source);
   printf("   result[%d]=%f\n", index, result[index]);
   printf("   result[%d]=%f\n", index+100, result[index+100]);
   printf("   result[%d]=%f\n\n", index+1000, result[index+1000]);
   fflush(stdout);
   }

 printf("MASTER: All Done! \n");
 }

/***************************** worker task ************************************/
if (taskid > MASTER) {
 /* Receive my portion of array from the master task */
 source = MASTER;

====>   ROTINA MPI PARA RECEBER DADOS (&index,source,&status)
====>   ROTINA MPI PARA RECEBER DADOS
          (&result[index],chunksize,source,&status)

 /* Do a simple value assignment to each of my array elements */
 for(i=index; i < index + chunksize; i++)
   result[i] = i + 1;

 /* Send my results back to the master task */

====>   ROTINA MPI DE ENVIO DE DADOS (&index,MASTER)
====>   ROTINA MPI DE ENVIO DE DADOS
          (&result[index],chunksize,MASTER)
 }

====>   ROTINA MPI DE FINALIZAÇÃO
}
```

Exercício 5

1 - Caminhe para o diretório com o quinto exercício do laboratório.

%cd ./mpi/lab01/ex5

Este programa calcula o valor de **PI**, através de uma integral de aproximação.

A idéia do exercício é que se entenda o algoritmo paralelo para a implementação da integral e se **corrija dois erros** bem simples de finalização do resultado, pois o programa não está funcionando corretamente.

2 - Compile o programa e execute para verificar os erros e possíveis correções

2º LABORATÓRIO
Comunicação Point-to-Point

Exercício 1

1 - Caminhe para o diretório com o primeiro exercício do laboratório.

%cd ./mpi/lab02/ex1

Este exercício tem como função, demonstrar e comparar a performance dos quatro modos de comunicação, a partir da transmissão de um determinado dado (o tempo mínimo dispendido num "blocking send").

OBS: - O programa não mede o tempo de comunicação completo e nem o total do "system overhead".

- Todos os processos filhos inicializados, executam um "blocking receive", antes do processo pai executar um "blocking send".

2 - Analise o programa e observe a similaridade na sintaxe entre as rotinas de "blocking send", o passo necessário para se executar um "buffered send" e o uso de "non-blocking receives".

3 - Compile o programa e execute-o inicializando apenas **dois processos** para que seja possível realizar a medição:

- Execute várias vezes o programa utilizando diferentes tamanho de mensagens;

- Em particular compare os resultados da transmissão de dados de 1024 floats e 1025 floats.

Exercício 2

1 - Caminhe para o diretório com o segundo exercício do laboratório.

%cd ./mpi/lab02/ex2

Este programa demonstra que a utilização de rotinas "non-blocking" são mais seguras que as rotinas "blocking".

2 - Compile e execute o programa inicializando apenas **dois processos**.

OBS: O programa irá imprimir várias linhas na saída padrão, e então para. Será necessário executar um <***ctrl***> <***c***>, para finalizar o programa.

3 - Corrija o programa para que possa excutar por completo. **O que será necessário fazer ???**

Exercício 3

1 - Caminhe para o diretório com o terceiro exercício do laboratório.

%cd ./mpi/lab02/ex3

Este programa demosntra o tempo perdido em "synchronization overhead" para mensagens maiores de 4Kbytes. Existe uma rotina (sleep) que simula o tempo que poderia ser utilizado em computação.

2 - Compile primeiro a rotina *sleep*:

%cc -c new_sleep.c

3 - Compile o programa principal:

%mpxlf brecv.f new_sleep.o -o brecv

ou

%mpcc brecv.c new_sleep.o -o brecv

4 - Execute o programa inicializando apenas **dois processos**. Anote o tempo de execução.

Exercício 4

1 - Caminhe para o diretório com o quarto exercício do laboratório.

%cd ./mpi/lab02/ex4

2 - Edite o programa do exercício anterior:

- Subistitua o "blocking receive" por um "non-blocking receive" antes da rotina *new_sleep*;

- Adicione a rotina *MPI_Wait* antes da impressão de mensagem recebida, para se garantir do recebimento.

3 - Compile o programa, como foi feito no exercício anterior

4 - Execute o programa, inicializando apenas **dois processos**, e compare o tempo de execução com a do exercício anterior.

3º LABORATÓRIO
Comunicação Coletiva

Exercício 1

1 - Caminhe para o diretório com o primeiro exercício do laboratório.

%cd ./mpi/lab03/ex1

Neste exercício:

- o processo mestre solicita a entrada de um número, que será a semente para o cálculo de um número randômico;

- este número será enviado para todos os processos. **Adicione ao programa, no lugar da "seta", a rotina MPI adequada para esta operação**;

- cada processo calculará um número randômico, baseado no número de netrada informado;

- o processo com o maior número de identificação calculará o valor médio de todos os número randômicos calculados. **Adicione ao programa, no lugar da "seta", a rotina MPI adequada para esta operação**;

- cada processo irá calcular, novamente, mais 4 novos números randômicos;

- será calculado o valor máximo e o desvio padrão de todos os números randômicos e os resultados serão distribuidos para todos os processos. Existem dois métodos par efetuar essas tarefas, utilizando rotinas de comunicação coletiva diferentes. **Adicione ao programa, no lugar das "setas", as rotinas MPI adequadas para cada método**.

2 - Compile o programa e execute.

ex1.f

```
! ----------------------------------------------------------------------
!  This program tests the use of MPI Collective Communications
!  subroutines.  The structure of the program is:
!
!     --  Master node queries for random number seed
!     --  The seed is sent to all nodes (Lab project)
!     --  Each node calculates one random number based on the seed
!         and the rank
!     --  The node with highest rank calculates the mean value
!         of the random numbers (Lab project)
!     --  4 more random numbers are generated by each node
!     --  The maximum value and the standard deviation of all
!         generated random numbers are calculated, and the
!         results are made available to all nodes (Lab project)
!
!  Also provided is a service routine GetStats(rnum,N,data), where
!
!        rnum:  array of random numbers (INPUT)
!        N:     number of elements in rnum (INPUT)
!        outd:  array of size 2 containing the maximum value and
!               standard deviation (OUTPUT)
! ----------------------------------------------------------------------
      program CollCom
      include 'mpif.h'
      integer status(MPI_STATUS_SIZE), ierr
      integer   numtasks, taskid
      real*4 randnum(5), sum, meanv, rnum(100), rval(2)
      real*8 seed
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, taskid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
      do i = 1, 100
        rnum(i) = 0.0
      enddo
      if( taskid .eq. 0 ) then    !  Get random number seed
        write( *, '(a)') ' Enter random number seed in f12.5 format'
        read( *, '(f12.5)') seed
      end if
! ======================================================
```

```
!   Project:  Send seed from task 0 to all nodes.
! ============================================

=====>  ROTINA MPI

      write( *, '(a, i3, a, f12.5)') ' Task', taskid,
     &      ' after broadcast; seed = ', seed
      call srand( seed + REAL(taskid))
      randnum(1) = 100*rand()    !  Each node generates a random number
! ============================================================
!   Project:  Have the node with highest rank calculate the
!             mean value of the random numbers and store result
!             in the variable "meanv".
! ============================================================

=====>  ROTINA MPI

      write(*, '(a, i3, a, f8.3, a, f8.3, a, f8.3)')
     &      ' Task', taskid, ' after mean value; random(1) =',
     &      randnum(1), ' sum =', sum, ' mean =', meanv
                            !  Highest task writes out mean value
      if( taskid .eq. (numtasks-1) ) then
         open( unit=10, file='f.data', status='UNKNOWN' )
         write(10, '(a, f12.5, a, f10.3)') ' For seed = ', seed,
     &         '   mean value = ', meanv
      end if
!  Each node generates 4 more random numbers
      do ii=2, 5
         randnum(ii) = 100*rand()
      end do
! ============================================================
!   Project:  Calculate the maximum value and standard deviation of
!             all random numbers generated, and make results known
!             to all nodes.
!   Method 1:
!   Method 2:
! ============================================================
```

```
!   ------  Method 1   -----------

=====>  ROTINA MPI

      if( taskid .eq. 0 ) then
        call GetStats( rnum, numtasks*5, rval )
      end if

=====>  ROTINA MPI

      write( *, '(a, i3, a, 5f8.3)') ' Task', taskid,
     &      ' after Method 1, rnum(1:5) =', (rnum(I), I=1, 5)
      if( taskid.eq.(numtasks-1) )
     &    write( 10,'(a, 2f10.3)') ' (Max, S.D.) = ', rval

!   ------  Method 2   -----------

=====> ROTINA MPI

      call GetStats( rnum, numtasks*5, rval )
!
      write( *, '(a, i3, a, 5f8.3)') ' Task', taskid,
     &      ' after Method 2, rnum(1:5) =', (rnum(I), I=1, 5)
      if( taskid .eq. (numtasks-1)) then
        write( 10,'(a, 2f10.3)') ' (Max, S.D.) = ', rval
        close( 10 )
      end if

      call MPI_FINALIZE( ierr )

      end
```

```
! ---------------------------------------------------------------------
!   The service routine GetStats(rnum,N,data), where
!
!        rnum:  array of random numbers (INPUT)
!        N:     number of elements in rnum (INPUT)
!        outd:  array of size 2 containing the maximum value and
!               standard deviation (OUTPUT)
!
! ---------------------------------------------------------------------
      subroutine GetStats( rnum, N, outd )
      real*4 rnum(N), outd(2), sum, meanv

      sum = 0.
      outd(1) = 0.

      do ii=1, N
         sum = sum + rnum(ii)
         if( rnum(ii) .gt. outd(1) ) outd(1) = rnum(ii)
      end do

      meanv = sum/REAL(N)
      sdev = 0.

      do ii=1, N
         sdev = sdev + (rnum(ii) - meanv)**2
      end do
      outd(2) = sqrt( (sdev)/N )

      return
      end
```

ex1.c

```
/* --------------------------------------------------------------------
   This program tests the use of MPI Collective Communications
   subroutines.  The structure of the program is:

     --  Master node queries for random number seed
     --  The seed is sent to all nodes (Lab project)
     --  Each node calculates one random number based on the seed
         and the rank
     --  The node with highest rank calculates the mean value
         of the random numbers (Lab project)
     --  4 more random numbers are generated by each node
     --  The maximum value and the standard deviation of all
         generated random numbers are calculated, and the
         results are made available to all nodes (Lab project)

   Also provided is a service routine GetStats(rnum,N,data), where

       rnum:  array of random numbers (INPUT)
       N:     number of elements in rnum (INPUT)
       outd:  array of size 2 containing the maximum value and
              standard deviation (OUTPUT)
 ------------------------------------------------------------------- */
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
int GetStats(float* rnum, int n, float* rval);
void main(int argc, char** argv)
{
  int          numtasks,taskid, ii;
  unsigned int  seed;
  float          randnum[5], sum, meanv, rnum[100], rval[2];
  MPI_Status    status;
  FILE* fp;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &taskid );
  MPI_Comm_size( MPI_COMM_WORLD, &numtasks );
```

```
  fp = fopen( "c.data", "w" );
  if( taskid == 0 )    /*  Get random number seed */
  {
   printf( " Enter random number as positive integer\n" );
   scanf( "%u", &seed);
  }

/*
   ================================================
   Project:  Send seed from task 0 to all nodes.
   ================================================
*/
```

=======> ROTINA MPI

```
  printf( "\n Task %d after broadcast; seed = %u", taskid, seed );
  srand( seed + taskid );

  randnum[0] = 100.*(float)rand()/(float)RAND_MAX; /* Get a random number */

/*
   ============================================================
   Project:  Have the node with highest rank calculate the
             mean value of the random numbers and store result
             in the variable "meanv".
   ============================================================
*/
```

=======> ROTINA MPI

```
  meanv = sum/(float)numtasks;

  printf( "\n Task %d after mean value; random[1] =", taskid );
  printf( "%8.3f sum = %8.3f mean = %8.3f", randnum[1], sum, meanv );

                          /* Highest task writes out mean value */
  if( taskid == (numtasks-1) )
    fprintf( fp, " For seed = %d    mean value = %10.3f\n", seed, meanv );
```

```
                        /*  Generate 4 more random numbers */
  for( ii=1; ii < 5; ii++ )
    randnum[ii] = 100.*(float)rand()/(float)RAND_MAX;
/*
   ============================================================
   Project:  Calculate the maximum value and standard deviation of
             all random numbers generated, and make results known
             to all nodes.
   Method 1:
   Method 2:
============================================================
*/
  /*   ------  Method 1   ----------- */
```

=======> ROTINA MPI

```
  if( taskid == 0 )
    GetStats( rnum, 5*numtasks, rval);
```

=======> ROTINA MPI

```
  printf( "\n Task %d after Method 1, rnum(1:5) =", taskid );
  for( ii=1; ii < 5; ii++ )
    printf( " %8.3f", rnum[ii] );
  if( taskid == (numtasks-1))
    fprintf( fp, " (Max, S.D.) = %10.3f%10.3f\n", rval[0], rval[1]);
  /*   ------  Method 2   -----------  */
```

=======> ROTINA MPI

```
  GetStats( rnum, 5*numtasks, rval);
  printf( "\n Task %d after Method 2, rnum(1:5) =", taskid );
  for( ii=1; ii < 5; ii++ )
    printf( " %8.3f", rnum[ii] );
  printf( "\n" );
  if( taskid == (numtasks-1))
    fprintf( fp," (Max, S.D.) = %10.3f%10.3f\n", rval[0], rval[1]);

  fclose( fp );
  MPI_Finalize();
}
```

```
/* ------------------------------------------------------------------
   Service routine GetStats( rnum, N, data), where

       rnum:  array of random numbers (INPUT)
       N:     number of elements in rnum (INPUT)
       outd:  array of size 2 containing the maximum value and
              standard deviation (OUTPUT)

  ------------------------------------------------------------------ */

int GetStats(float* rnum, int N, float* outd)
{
  float  sum, meanv, sdev;
  int    ii;

  sum = 0.;
  *outd = 0.;

  for( ii = 0 ; ii < N ; ii++ )
  {
    sum += *(rnum + ii);
    if( *(rnum + ii) > *outd )
      *outd = *(rnum + ii);
  }

  meanv = sum/(float)N;
  sdev = 0.;
  for( ii = 0; ii < N; ii++ )
    sdev += (*(rnum + ii) - meanv) * (*(rnum + ii) - meanv);

  *(outd + 1) = sqrt( sdev/(float)N );
}
```

Exercício 2

1 - Caminhe para o diretório com o segundo exercício do laboratório.

%cd ./mpi/lab03/ex2

A idéia do programa é demonstrar a execução da rotina **MPI_SCATTER.**

Adicione os seus parâmetros para que ela funcione adequadamente.

2 - Compile o programa e execute-o.

scatter.f

```
      program scatter
      include 'mpif.h'

      integer SIZE
      parameter(SIZE=4)
      integer numtasks, rank, sendcount, recvcount, source, ierr
      real*4 sendbuf(SIZE,SIZE), recvbuf(SIZE)

C     Fortran stores this array in column major order, so the
C     scatter will actually scatter columns, not rows.
      data sendbuf /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 /

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

      if (numtasks .eq. SIZE) then
        source = 1
        sendcount = SIZE
        recvcount = SIZE

        call MPI_SCATTER( ... , ... , ... , ... , ... , ... , ... , ... )

        print *, 'rank= ',rank,' Results: ',recvbuf
      else
        print *, 'Must specify',SIZE,' processors.  Terminating.'
      endif

      call MPI_FINALIZE(ierr)

      end
```

scatter.c

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

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

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

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;

  MPI_Scatter( ... , ... , ... , ... , ... , ... , ... , ...);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
        recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Exercício 3

1 - Caminhe para o diretório com o primeiro exercício do laboratório.

%cd ./mpi/lab03/ex3

Este programa calcula o maior número primo dos números primos calculados, até um limite determinado pelo programador. A idéia e demonstrar a utilização, apenas da rotina de operação de reducao, **MPI_REDUCE.**

Subistitua, adequadamente, os parâmetros desta rotina, no programa.

2 - Compile o programa e execute-o.

mpi_prime.f

```
C FILE: mpi_prime.f
C OTHER FILES: make.mpi_prime.f
C DESCRIPTION:
C   Generates prime numbers.  All tasks distribute the work evenly, taking
C   every nth number, where n is the stride computed as:  (rank *2) + 1
C   so that even numbers are automatically skipped.  The method of using
C   stride is preferred over contiguous blocks of numbers, since numbers
C   in the higher range require more work to compute and may result in
C   load imbalance.  This program demonstrates embarrassing parallelism.
C   Collective communications calls are used to reduce the only two data
C   elements requiring communications: the number of primes found and
C   the largest prime.
C AUTHOR: Blaise Barney 11/25/95 - adapted from version contributed by
C   Richard Ng & Wong Sze Cheong during MHPCC Singapore Workshop (8/22/95).
C LAST REVISED:
C ----------------------------------------------------------------------------
C Explanation of constants and variables
C   LIMIT                = Increase this to find more primes
C   FIRST                = Rank of first task
C   ntasks               = total number of tasks in partitiion
C   rank                 = task identifier
C   n                    = loop variable
C   pc                   = prime counter
C   pcsum                = number of primes found by all tasks
C   foundone             = most recent prime found
C   maxprime             = largest prime found
C   mystart              = where to start calculating
C   stride               = calculate every nth number
C ----------------------------------------------------------------------------
      program prime
      include 'mpif.h'
      integer LIMIT, FIRST
      parameter(LIMIT=2500000)
      parameter(FIRST=0)
      integer ntasks, rank, ierr, n, pc, pcsum, foundone, maxprime,
     &        mystart, stride
      double precision start_time, end_time
      logical result
      call MPI_INIT(ierr)
```

```
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, ntasks, ierr)
      if ((mod(ntasks,2).ne.0) .or. (mod(LIMIT,ntasks).ne.0)) then
        print *, mod(ntasks,2), mod(LIMIT,ntasks)
        print *,'Sorry -this exercise requires an even number of '
        print *,'processors evenly divisible into ',LIMIT,'.'
        print *,'Try 4 or 8.'
        call MPI_FINALIZE(ierr)
        stop
      endif
C     Initializations: mystart must be odd number. stride is multiplied
C     by 2 to skip over even numbers.
      start_time = MPI_WTIME()
      mystart = (rank*2) + 1
      stride = ntasks*2
      pc = 0
      foundone = 0
C ------------------- task with rank 0 does this part ------------------
      if (rank .eq. FIRST) then
        print *,'Using',ntasks,'tasks to scan',LIMIT,'numbers'
C         Assume first four primes are counted here
        pc = 4
        do n=mystart,LIMIT,stride
          call isprime(n,result)
          if (result .eqv. .true.) then
            pc = pc + 1
            foundone = n
C             ***** Optional: print each prime as it is found
C             print *, foundone
C             *****
          endif
        enddo
      call MPI_Reduce(... , ... ,1,MPI_INTEGER, ... , ... , MPI_COMM_WORLD,ierr)
      call MPI_Reduce(... , ... ,1,MPI_INTEGER, ... , ... , MPI_COMM_WORLD,ierr)
      end_time=MPI_WTIME()
      print *,'Done. Largest prime is ',maxprime,' Total primes ',pcsum
      print *,'Wallclock time elapsed: ' ,end_time-start_time
      endif
C ------------------- all other tasks do this part ---------------------
      if (rank .gt. FIRST) then
        do n=mystart,LIMIT,stride
          call isprime(n,result)
```

```
          if (result .eqv. .true.) then
            pc = pc + 1
            foundone = n
C             ***** Optional: print each prime as it is found
C             print *, foundone
C             *****
          endif
        enddo
      call MPI_Reduce(... , ... ,1,MPI_INTEGER, ... , ... , MPI_COMM_WORLD,ierr)
      call MPI_Reduce(... , ... ,1,MPI_INTEGER, ... , ... , MPI_COMM_WORLD,ierr)
      endif
      call MPI_FINALIZE(ierr)
      end
      subroutine isprime (n,result)
      integer n
      logical result
      integer i, squareroot
      real*4 realn
      if (n .gt. 10) then
        realn = real(n)
        squareroot = int (sqrt(realn))
        do i=3,squareroot,2
          if (mod(n,i) .eq. 0) then
            result = .false.
            return
          endif
        enddo
        result = .true.
        return
C Assume first four primes are counted elsewhere. Forget everything else
      else
        result = .false.
        return
      endif
      end
```

mpi_prime.c

```
/******************************************************************************
**
* FILE: mpi_prime.c
* OTHER FILES: make.mpi_prime.c
* DESCRIPTION:
*   Generates prime numbers.  All tasks distribute the work evenly, taking
*   every nth number, where n is the stride computed as:  (rank *2) + 1
*   so that even numbers are automatically skipped.  The method of using
*   stride is preferred over contiguous blocks of numbers, since numbers
*   in the higher range require more work to compute and may result in
*   load imbalance.  This program demonstrates embarrassing parallelism.
*   Collective communications calls are used to reduce the only two data
*   elements requiring communications: the number of primes found and
*   the largest prime.
* AUTHOR: Blaise Barney 11/25/95 - adapted from version contributed by
*   Richard Ng &  Wong Sze Cheong during MHPCC Singapore Workshop (8/22/95).
* LAST REVISED:
******************************************************************************
**/
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#define LIMIT     2500000     /* Increase this to find more primes */
#define FIRST     0           /* Rank of first task */
int isprime(int n) {
int i,squareroot;
if (n>10) {
  squareroot = (int) sqrt(n);
  for (i=3; i<=squareroot; i=i+2)
    if ((n%i)==0)
      return 0;
  return 1;
  }
/* Assume first four primes are counted elsewhere. Forget everything else */
else
  return 0;
}
int main(argc,argv)
int argc;
```

```
char *argv[]; {
int   ntasks,            /* total number of tasks in partitiion */
      rank,              /* task identifier */
      n,                 /* loop variable */
      pc,                /* prime counter */
      pcsum,             /* number of primes found by all tasks */
      foundone,          /* most recent prime found */
      maxprime,          /* largest prime found */
      mystart,           /* where to start calculating */
      stride;            /* calculate every nth number */

double start_time,end_time;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&ntasks);
if (((ntasks%2) !=0) || ((LIMIT%ntasks) !=0)) {
   printf("Sorry - this exercise requires an even number of processors\n");
   printf("evenly divisible into %d.  Try 4 or 8.\n",LIMIT);
   MPI_Finalize();
   exit(0);
   }

start_time = MPI_Wtime();       /* Initialize start time */
mystart = (rank*2)+1;       /* Find my starting point - must be odd number */
stride = ntasks*2;          /* Determine stride, skipping even numbers */
pc=0;                       /* Initialize prime counter */
foundone = 0;               /* Initialize */
/******************** task with rank 0 does this part ********************/
if (rank == FIRST) {
   printf("Using %d tasks to scan %d numbers\n",ntasks,LIMIT);
   pc = 4;                  /* Assume first four primes are counted here */
   for (n=mystart; n<=LIMIT; n=n+stride) {
      if (isprime(n)) {
         pc++;
         foundone = n;
         /***** Optional: print each prime as it is found
         printf("%d\n",foundone);
         *****/
         }
      }
   MPI_Reduce(&... , &... ,1,MPI_INT, ... , ... ,MPI_COMM_WORLD);
   MPI_Reduce(&... , &... ,1,MPI_INT, ... , ... ,MPI_COMM_WORLD);
```

```
   end_time=MPI_Wtime();
   printf("Done. Largest prime is %d Total primes %d\n",maxprime,pcsum);
   printf("Wallclock time elapsed: %.2lf\n",end_time-start_time);
   }
/********************* all other tasks do this part ************************/
if (rank > FIRST) {
   for (n=mystart; n<=LIMIT; n=n+stride) {
      if (isprime(n)) {
         pc++;
         foundone = n;
         /***** Optional: print each prime as it is found
         printf("%d\n",foundone);
         *****/
         }
      }

   MPI_Reduce(&... , &...,1,MPI_INT, ... , ... ,MPI_COMM_WORLD);
   MPI_Reduce(&..., &... ,1,MPI_INT, ... , ... ,MPI_COMM_WORLD);

   }

MPI_Finalize();
}
```

Exercício 4

1 - Caminhe para o diretório com o quarto exercício do laboratório.

%cd ./mpi/lab03/ex4

Este exercício possui duas soluções.

1.1 - Na primeira solução, e utilizada as rotinas de **send** e **receive** para que os processos calculem o pedaço do valor de pi. Será necessário utilizar uma função com o algoritmo para o calculo de pi

- Compile o programa:

% mpxlf mpi_pi_send.f dboard.f -o mpi_pi_send
ou
% mpcc mpi_pi_send.c dboard.c -o mpi_pi_send

- Execute o programa:

% poe -procs 4 mpi_pi_send

Analise o programa, com relação a utilização das rotinas de **send** e **receive**.

1.2 - Na segunda solução, é utilizada apenas uma rotina, que coleta os resultados de todos os processos.

- **Adicione esta rotina ao programa mpi_pi_opt.f** ou **mpi_pi_opt.c**

- Compile o programa:

% mpxlf mpi_pi_opt.f dboard.f -o mpi_pi_opt
ou
% mpcc mpi_pi_opt.c dboard.c -o mpi_pi_opt

- Execute o programa:

% poe -procs 4 mpi_pi_opt

mpi_pi_opt.f

```
C  ------------------------------------------------------------------
C  MPI pi Calculation Example - Fortran Version
C  Collective Communications examples
C  FILE: mpi_pi_opt.f
C  OTHER FILES: dboard.f
C  DESCRIPTION:  MPI pi calculation example program.  Fortran version.
C  This program calculates pi using a "dartboard" algorithm.  See
C  Fox et al.(1988) Solving Problems on Concurrent Processors, vol.1
C  page 207.  All processes contribute to the calculation, with the
C  master averaging the values for pi.
C
C  SPMD Version:  Conditional statements check if the process is the
C  master or a worker.
C
C  This version uses one MPI routine to collect results
C
C  AUTHOR: Roslyn Leibensperger. Converted to MPI: George L. Gusciora (1/23/95)
C  LAST REVISED: 12/14/95 Blaise Barney
C  ------------------------------------------------------------------
C Explanation of constants and variables used in this program:
C   DARTS         = number of throws at dartboard
C   ROUNDS        = number of times "DARTS" is iterated
C   MASTER        = task ID of master task
C   taskid        = task ID of current task
C   numtasks      = number of tasks
C   homepi        = value of pi calculated by current task
C   pisum         = sum of tasks' pi values
C   pi            = average of pi for this iteration
C   avepi         = average pi value for all iterations
C   seednum       = seed number - based on taskid
C  ------------------------------------------------------------------
      program pi_reduce
      include 'mpif.h'
      integer DARTS, ROUNDS, MASTER
      parameter(DARTS = 5000)
      parameter(ROUNDS = 10)
      parameter(MASTER = 0)
      integer   taskid, numtasks, i, status(MPI_STATUS_SIZE)
      real*4 seednum
```

```
      real*8 homepi, pi, avepi, pisum, dboard
      external  d_vadd
C     Obtain number of tasks and task ID
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, taskid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numtasks, ierr )
      write(*,*)'task ID = ', taskid
C     Use the task id to set the seed number for the random number generator.
      seednum = real(taskid)
      call srand(seednum)
      avepi = 0
      do 40 i = 1, ROUNDS
C     Calculate pi using dartboard algorithm
       homepi = dboard(DARTS)
C     Use mpi routine to sum values of homepi across all tasks
C     Master will store the accumulated value in pisum
C     - homepi is the send buffer
C     - pisum is the receive buffer (used by the receiving task only)
C     - MASTER is the task that will receive the result of the reduction
C       operation
C     - d_vadd is a pre-defined reduction function (double-precision
C       floating-point vector addition)
C     - allgrp is the group of tasks that will participate
```

======> **ROTINA MPI**

```
C     Master computes average for this iteration and all iterations
       if (taskid .eq. MASTER) then
         pi = pisum/numtasks
         avepi = ((avepi*(i-1)) + pi) / i
         write(*,32) DARTS*i, avepi
 32       format('   After',i6,' throws, average value of pi = ',f10.8)
       endif
 40   continue
      call MPI_FINALIZE(ierr)
      end
```

mpi_pi_opt.c

```
/* ------------------------------------------------------------
 * MPI pi Calculation Example - C Version
 * Collective Communication example
 * FILE: mpi_pi_opt.c
 * OTHER FILES: dboard.c
 * DESCRIPTION:  MPI pi calculation example program.  C Version.
 *   This program calculates pi using a "dartboard" algorithm.  See
 *   Fox et al.(1988) Solving Problems on Concurrent Processors, vol.1
 *   page 207.  All processes contribute to the calculation, with the
 *   master averaging the values for pi.
 *   SPMD version:  Conditional statements check if the process
 *   is the master or a worker.
 *   This version uses one MPI routine to collect results
 * AUTHOR: Roslyn Leibensperger. Converted to MPI: George L. Gusciora
 * LAST REVISED: 06/07/96 Blaise Barney
 * ------------------------------------------------------------ */
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
void srandom (unsigned seed);
double dboard (int darts);
#define DARTS 5000      /* number of throws at dartboard */
#define ROUNDS 10       /* number of times "darts" is iterated */
#define MASTER 0        /* task ID of master task */
int main(argc,argv)
int argc;
char *argv[];
{
double  homepi,         /* value of pi calculated by current task */
        pisum,          /* sum of tasks' pi values */
        pi,             /* average of pi after "darts" is thrown */
        avepi;          /* average pi value for all iterations */
int     taskid,         /* task ID - also used as seed number */
        numtasks,       /* number of tasks */
        rc,             /* return code */
        i;
MPI_Status status;
  /* Obtain number of tasks and task ID */
  rc = MPI_Init(&argc,&argv);
```

```
  rc|= MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  rc|= MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
  if (rc != 0)
    printf ("error initializing MPI and obtaining task ID information\n");
  else
    printf ("task ID = %d\n", taskid);
  /* Set seed for random number generator equal to task ID */
  srandom (taskid);
  avepi = 0;
  for (i = 0; i < ROUNDS; i++)
  {
    /* All tasks calculate pi using dartboard algorithm */
    homepi = dboard(DARTS);
    /* Use MPI Routine to sum values of homepi across all tasks
     * Master will store the accumulated value in pisum
     * - homepi is the send buffer
     * - pisum is the receive buffer (used by the receiving task only)
     * - the size of the message is sizeof(double)
     * - MASTER is the task that will receive the result of the reduction
     *   operation
     * - MPI_SUM is a pre-defined reduction function (double-precision
     *   floating-point vector addition).  Must be declared extern.
     * - MPI_COMM_WORLD is the group of tasks that will participate.
    rc = ... ROTINA MPI ...
    if (rc != 0)
      printf("%d: failure on mpc_reduce\n", taskid);
    /* Master computes average for this iteration and all iterations */
    if (taskid == MASTER)
    {
      pi = pisum/numtasks;
      avepi = ((avepi * i) + pi)/(i + 1);
      printf("   After %3d throws, average value of pi = %10.8f\n",
              (DARTS * (i + 1)),avepi);
    }
  }
  MPI_Finalize();
  return 0;
}
```