INTRODUÇÃO AO MPI

CENAPAD-SP

Índice

INTRODUÇÃO A PROGRAMAÇÃO PARALELA

1 - IDEIAS E CONCEITOS

O que é paralelismo ?

É uma estratégia para obter resultados mais rápidos, de grandes e complexas tarefas.

O paralelismo é efetuado:

- Dividindo-se uma tarefa, em várias pequenas tarefas;
- Distribuindo-se as pequenas tarefas por entre vários processadores, que irão executá-las simultaneamente;
- Coordenar esses processadores.

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.

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:

Processamento de sinal e imagem;
Visualização de dados;
Banco de Dados.

Grandes desafios computacionais:

Modelagem de Clima;
Turbulência de Fluidos;
Dispersão de Poluição;
Engenharia Genética;
Circulação de Correntes Marítimas;
Modelagem de Semicondutores;
Sistema de Combustão.

Computação Paralela

Necessidades para se possuir um ambiente de computação paralela:

Múltiplos processadores;

Memória centralizada ou distribuída;

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;

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 maneira de se escrever um programa paralelo.

Memória Compartilhada

Múltiplos 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;

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

Vantagens: É fácil de usar e a transferência de dados é rápida.

Desvantagens: Limita o número de processadores (≅ 64 processadores)

Exemplos: 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 interface de comunicação (rede ou switch), utilizando-se "Message-Passing";

- O usuário é responsável pela sincronização das tarefas;

- Vantagens: Não existem limites para número de processadores e cada processador acessa, sem interferência e rapidamente, sua própria memória;

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

- Exemplos: nCUBE, Intel Hypercube, IBM SP, CM-5

3 - CRIAÇÃO DE UM PROGRAMA PARALELO

Decomposição do Programa

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

Decomposição Funcional

- O **problema** é decomposto em **diferentes** tarefas, gerando diversos programas, que serão distribuídos por entre múltiplos processadores, para execução simultânea;

Decomposição de Domínio

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

Comunicação do Programa

Em programação paralela, é essencial que se compreenda a comunicação que ocorre entre os processadores:

"Message-Passing Library" - A programação da comunicação é explicita, ou seja, o programador é responsável pela comunicação entre os processos;

"Compiladores Paralelos" (Data Parallel Compilers) - O programador não necessita entender dos métodos de comunicação, que é feita pelo compilador

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
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 de paralelismo de um programa (**f**):

speedup = 1 / (1 - f)

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

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

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

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

P = Porcentagem paralela;
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, deve ser de uma maneira que o tempo da execução paralela seja eficiente;

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

"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 possível 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

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 (iniciar, finalizar, determinar o número de processos, identificar processos);

Rotinas de comunicação "Point-to-Point" (Enviar e receber mensagens entre dois processos);

Rotinas de comunicação de grupos ("broadcast", sincronizar 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 o 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 possível, "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 distribuída, 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 os ambientes de memória distribuída, máquinas paralelas massivas, "clusters" de estações de trabalho.

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

“Workshop” de padrões de "Message-Passing" em ambientes de memória distribuída (Centro de Pesquisa em Computação Paralela, Williamsburg, Virginia);

Discussão das necessidades básicas e essenciais para se estabelecer um padrão “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, como domínio público, da versão padrão do MPI (**MPI1** http://www.mcs.anl.gov/Projects/mpi/standard.html

Dezembro de 1995

Conferência de Supercomputação 95 - Reunião para discussão do **MPI2** e suas extensões.

Implementações de MPI

MPI-F: IBM Research

MPICH: ANL/MSU - Domínio Publico

UNIFY: Mississipi State University

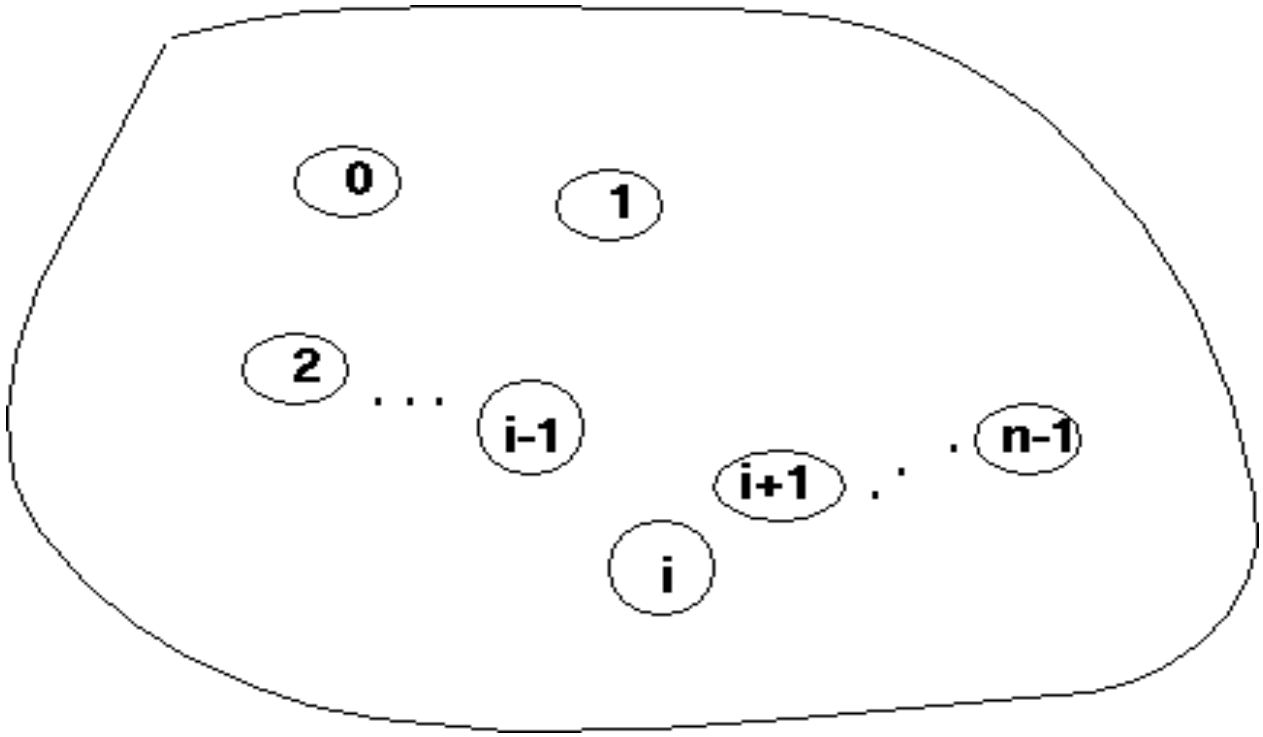
CHIMP: Edinburgh Parallel Computing Center

LAM: Ohio Supercomputer Center

3 - CONCEITOS E DEFINIÇÕES

Rank

Todo processo tem uma única identificação, atribuída pelo sistema quando o processo é iniciado. 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 membros de um grupo com um "communicator" já pré-estabelecido (MPI_COMM_WORLD).

Communicator

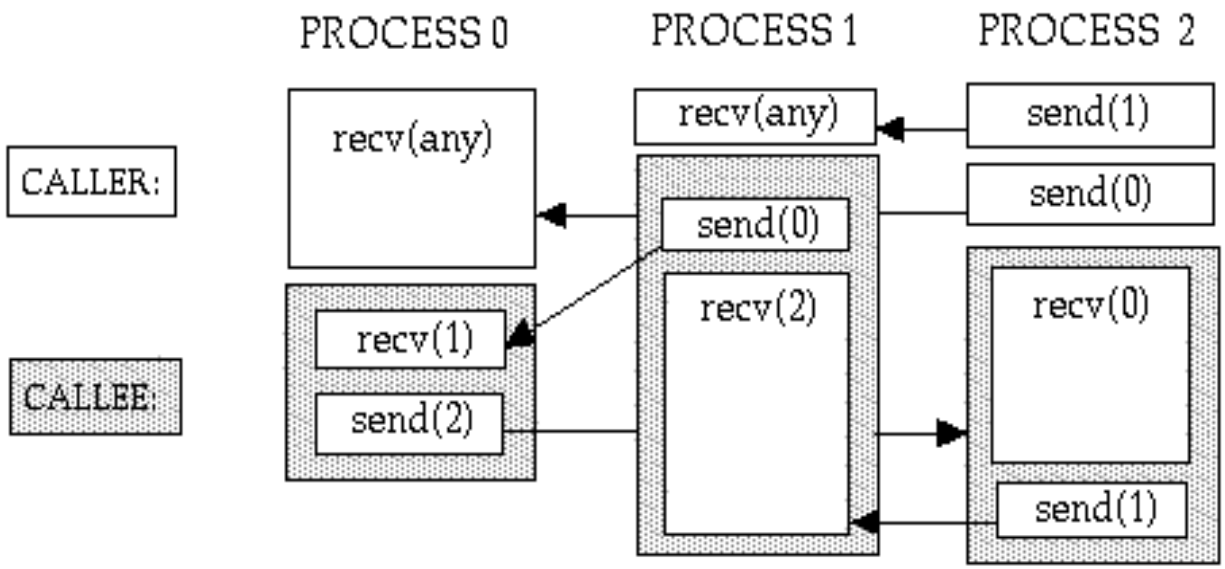
O "communicator" define uma coleção de processos (grupo), que poderão se comunicar entre si (contexto). O MPI utiliza essa combinação de grupo e contexto para garantir uma comunicação segura e evitar problemas no envio de mensagens entre os processos.

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

Essa rotina pode usar uma mensagem idêntica a mensagem do usuário.

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

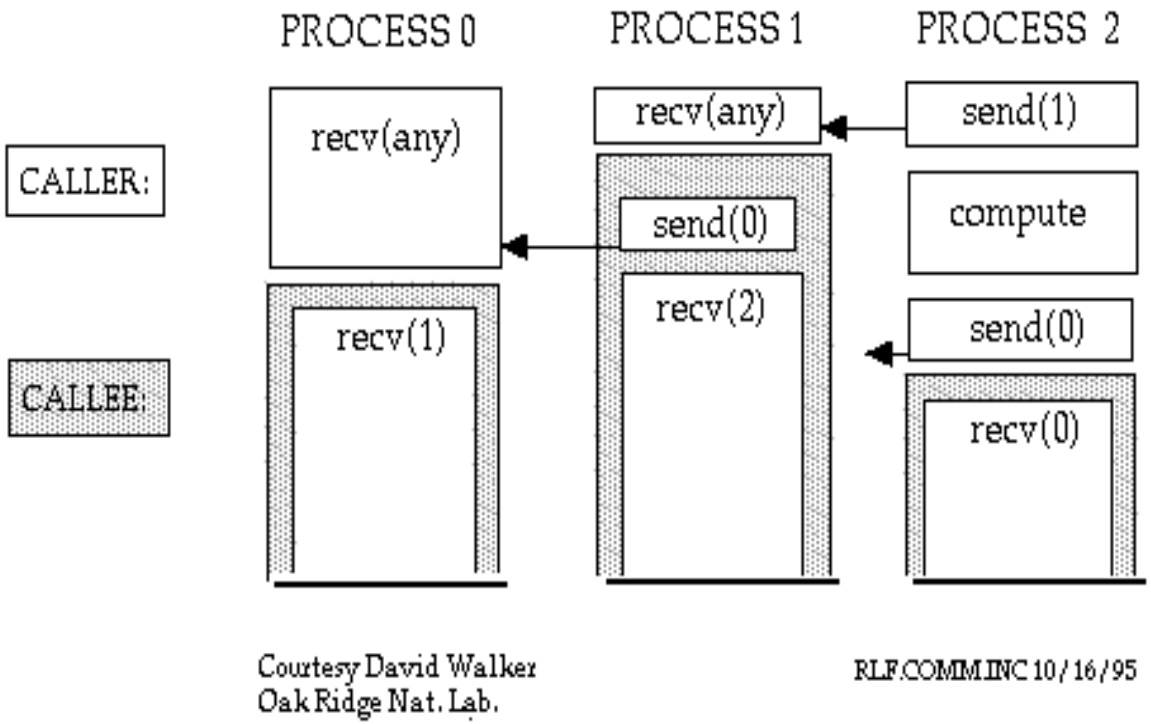
AÇÃO DESEJADA



Courtesy David Walker
Oak Ridge Nat. Lab.

RLF.COMMDES 10/16/95

POSSÍVEL AÇÃO INDESEJADA



Aplication Buffer

É um endereço normal de memória (Ex: variável) aonde se 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.

Synchronous Send

Bloqueia até que ocorra um "receive" correspondente no processo de destino.

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 NO AMBIENTE SGI DO CENAPAD

A implementação do **MPI** definiu num único comando, as tarefas de compilação e linkedição:

FORTRAN 90

ifort <fonte> -o <executável> -lmpi

C Standard

icc <fonte> -o <executável> -lmpi

C++

icpc <fonte> -o <executável> -lmpi

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

5 – EXECUÇÃO

mpirun -np <número de processos> <arquivo executável>

A execução de um programa, com rotinas **MPI,** no ambiente **SGI/INTEL**, é feita através de um programa que configura e estabele o ambiente paralelo, ou seja, o **mpirun.**

6 - ROTINAS BÁSICAS

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

Arquivo de "defaults"

Necessário para todos os programas ou rotinas que efetuam chamadas para a biblioteca MPI. Normalmente é colocado no início do programa.

C **#include "mpi.h"**

FORTRAN **include "mpif.h"**

6.1 - ***Inicializar um processo MPI***

MPI_INIT

- Primeira rotina MPI utilizada.
- Define e inicia o ambiente necessário para executar o MPI.
- Sincroniza todos os processos no inicio de uma aplicação MPI.

C ***int MPI_Init (*argc, *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 (comm, *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 (comm, *size)***

FORTRAN ***call MPI_COMM_SIZE (comm, size, mpierr)***

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

6.4 - ***Enviar menssagens no MPI***

MPI_SEND

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

C ***int MPI_Send (*sndbuf, count, datatype, dest, tag, comm)***

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

sndbuf	Endereço inicial do dado 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(*recvbuf, count, datatype, source, tag, comm.,*status)***

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 método de comunicação básica 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 no 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:

Identificação do processo que envia ou do processo que recebe;

Rótulo ("tag") da mensagem;

"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 a 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(*buf, count, datatype, source, tag, comm, 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 Vetor com informações sobre a mensagem;

ierror Código de retorno com o status da rotina.

9.1.1 - "Blocking Synchronous Send"

C ***int MPI_Ssend(*buf, count, datatype, dest, tag, comm)***

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

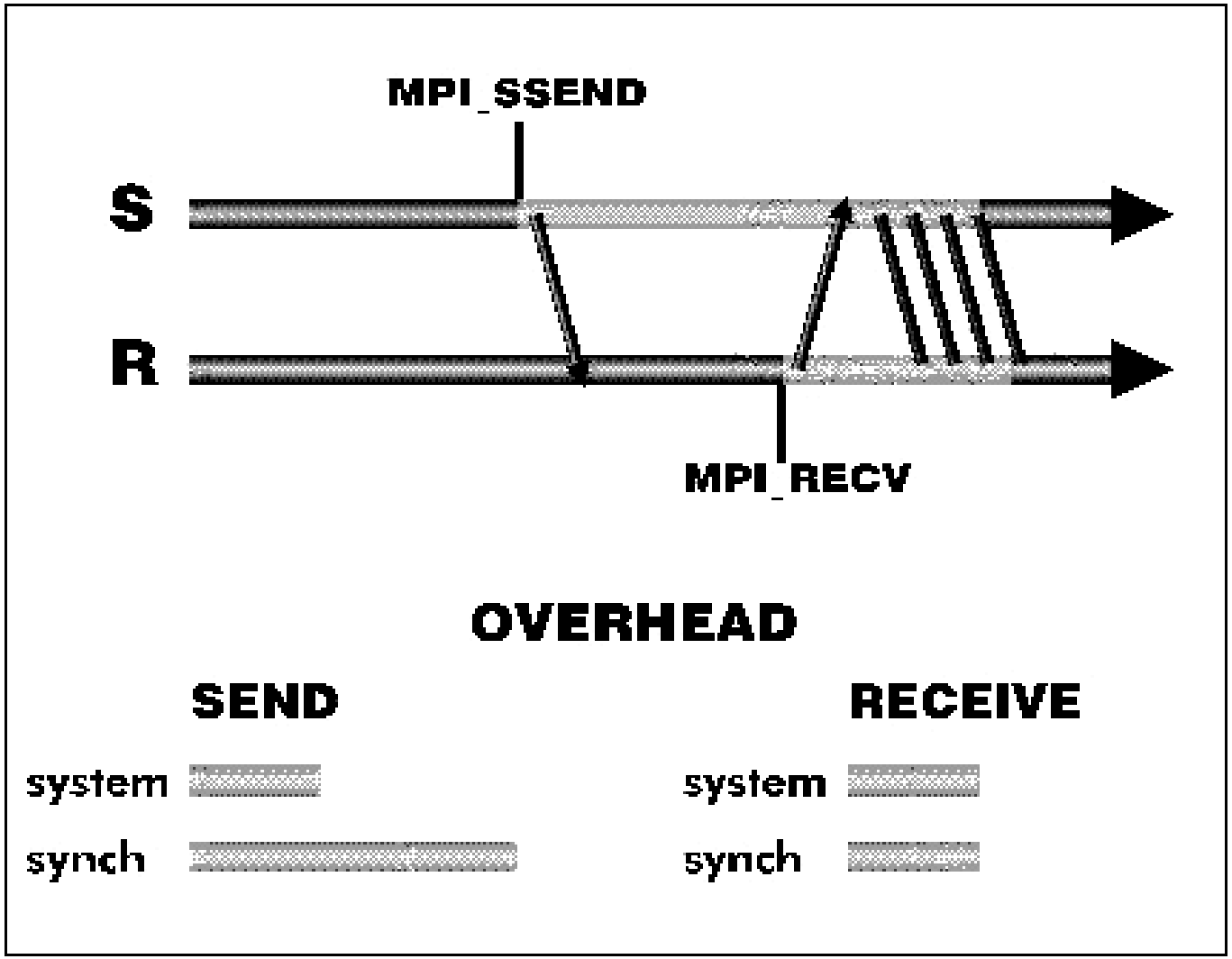
Quando um **MPI_Ssend** é executado, o processo que envia avisa ao processo que recebe que uma mensagem está pronta e **esperando** por um sinal de OK, para que então, seja transferido o dado.

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(*buf, count, datatype, dest, tag, comm)***

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

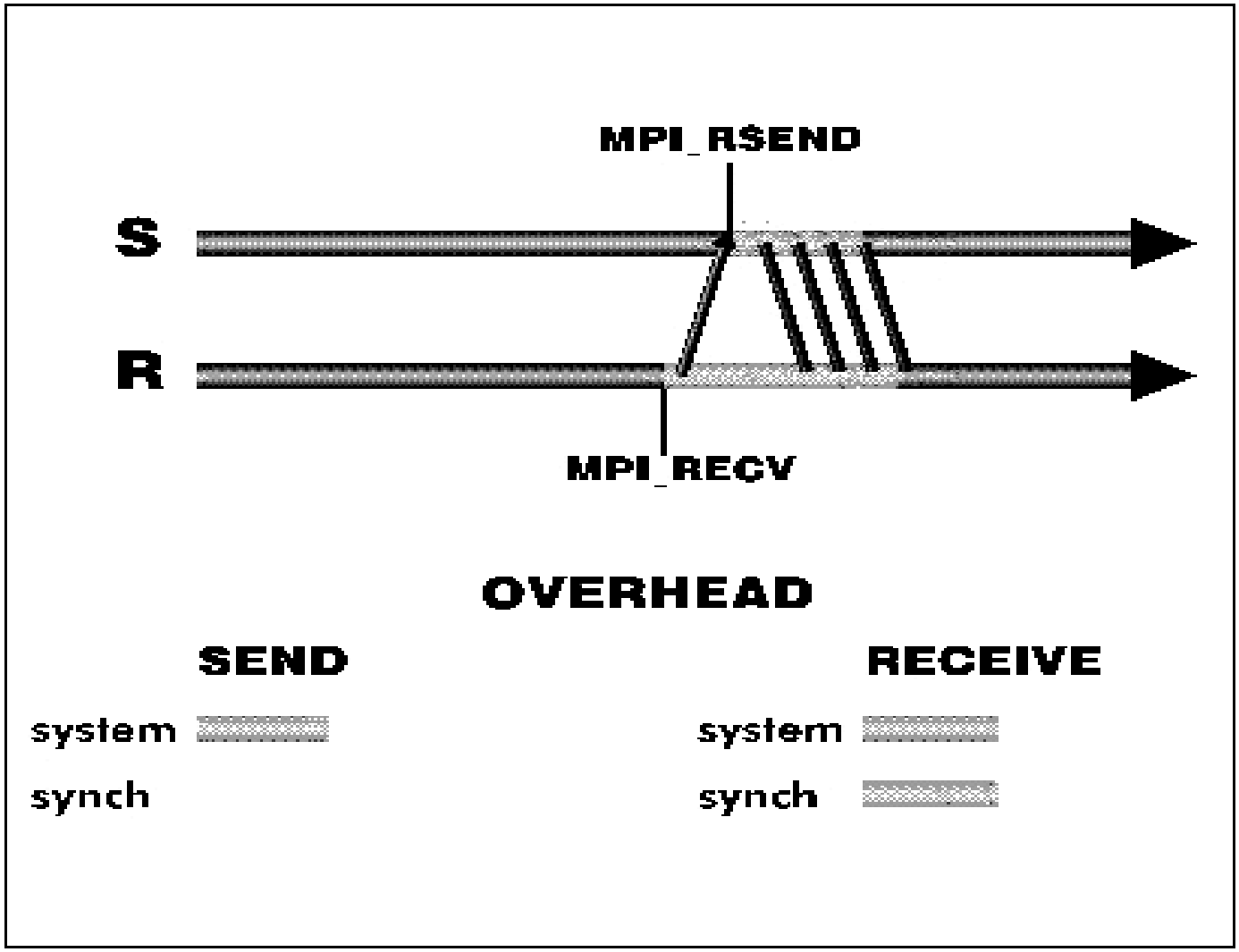
Quando um **MPI_Rsend** é executado a mensagem é enviada imediatamente para a rede. É exigido que um sinal de OK do processo que irá receber, já tenha sido feito.

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(*buf, count, datatype, dest, tag, 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".

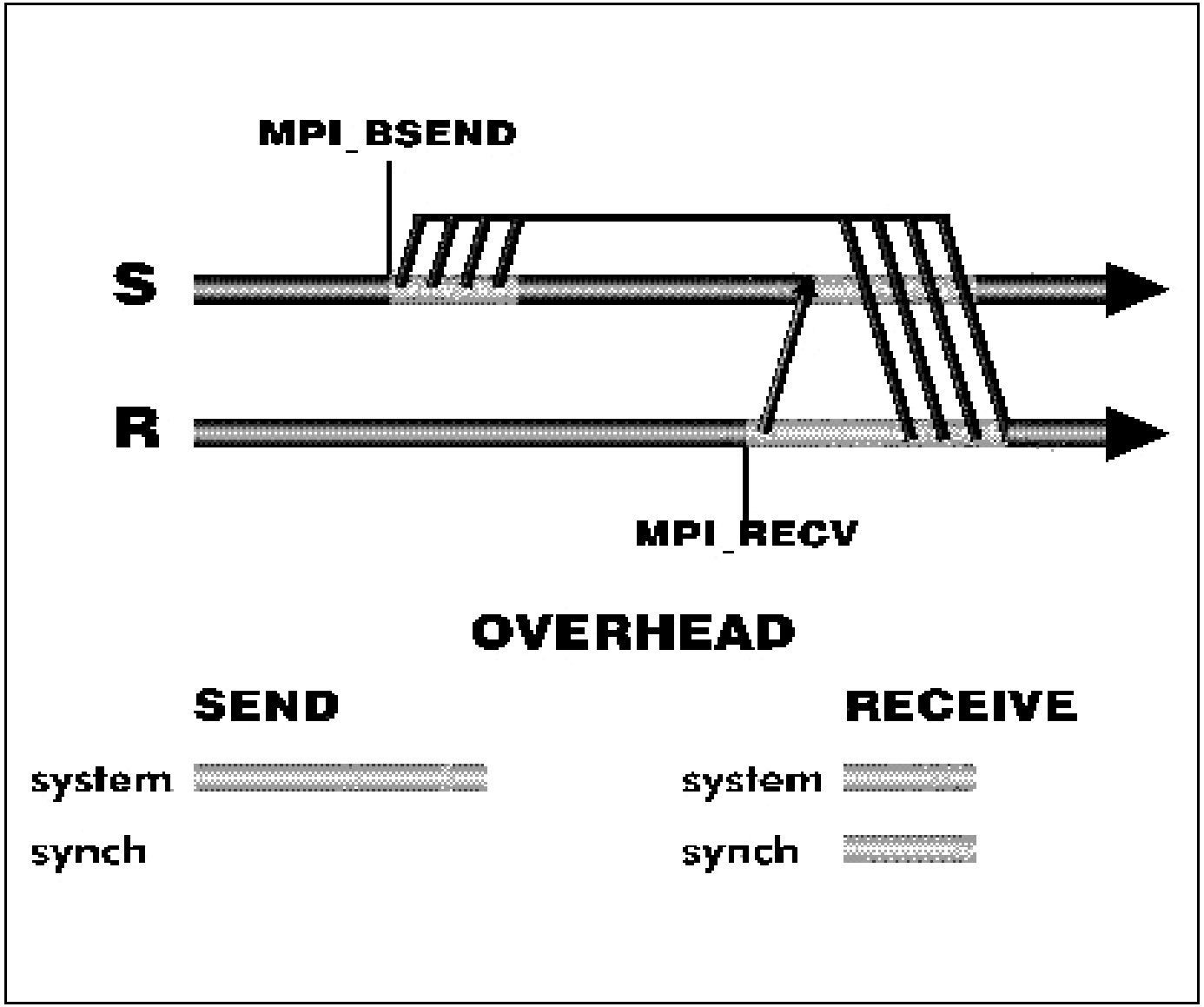
OBS: Ocorre **"System overhead"** devido a cópia da mensagem do "Aplication buffer" para o "buffer" definido pelo usuário.

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



9.1.4 - "Blocking Standard Send"

C ***int MPI_Send(*buf, count, datatype, dest, tag, comm)***

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

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

Message <= 4K

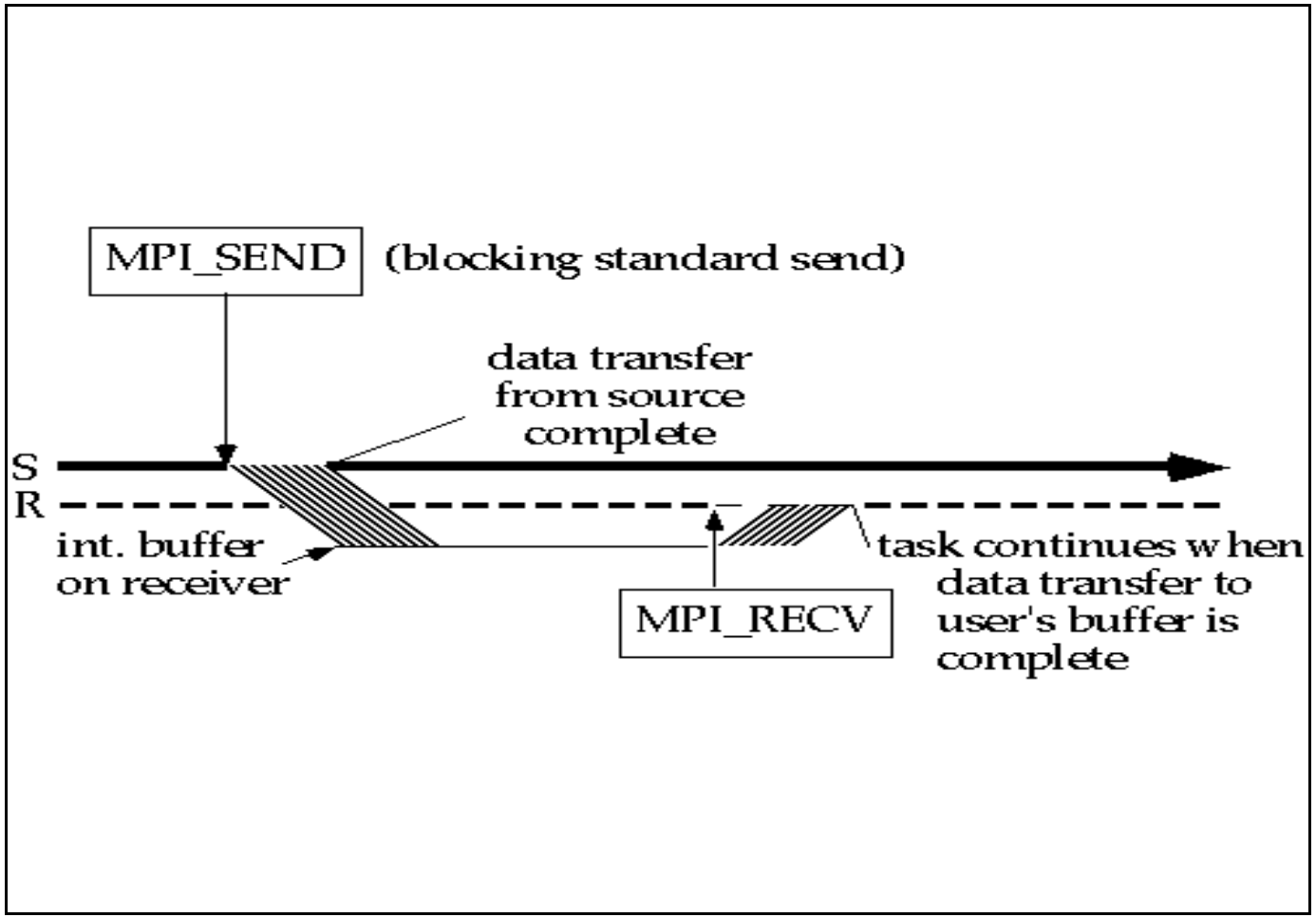
Quando **MPI_Send** é executado, a mensagem é imediatamente transmitida para rede, e então, para um "System buffer" do processo que irá receber a mensagem.

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

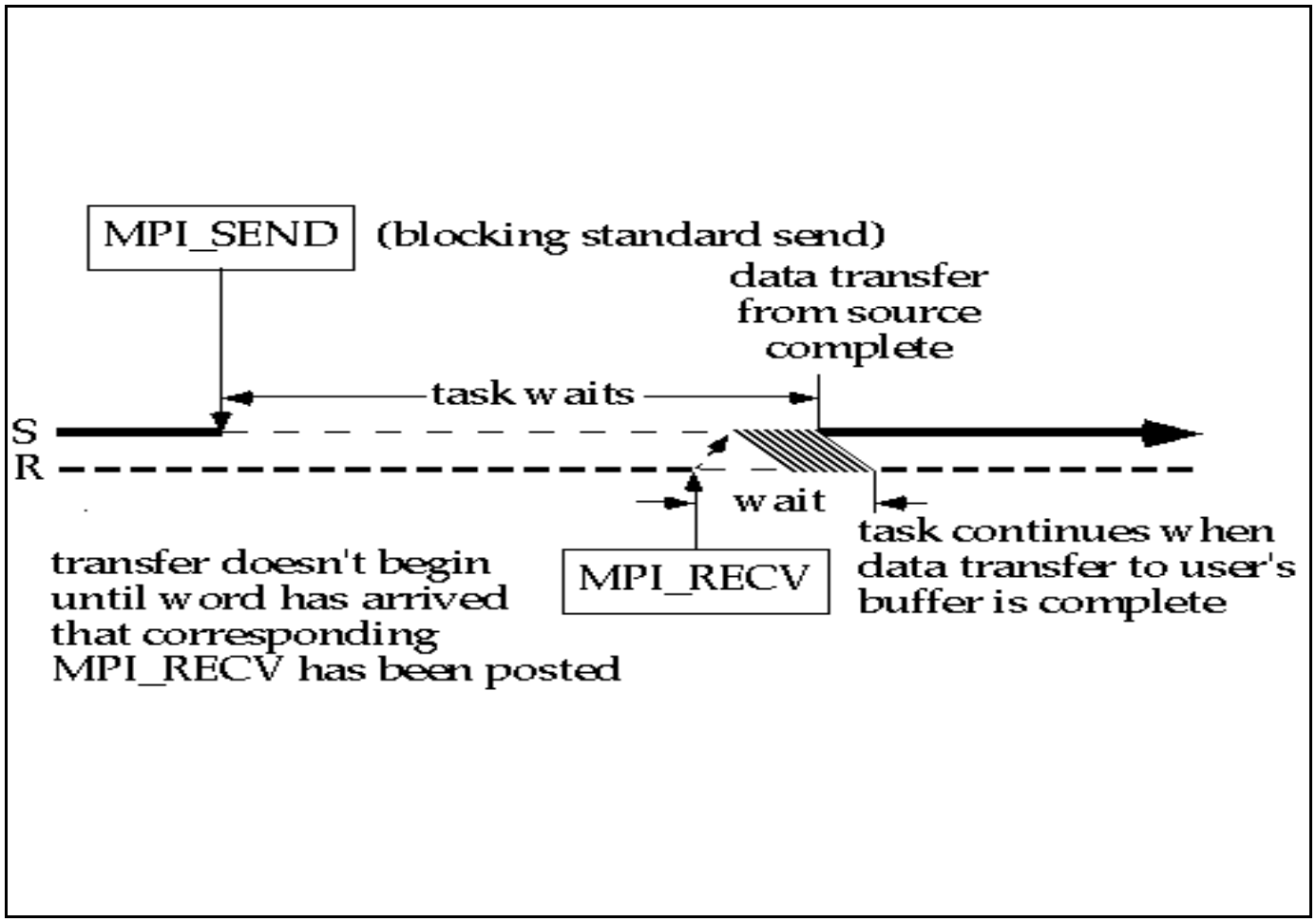
Message > 4K

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

Blocking Standard Send e Blocking Receive
Message <= 4K

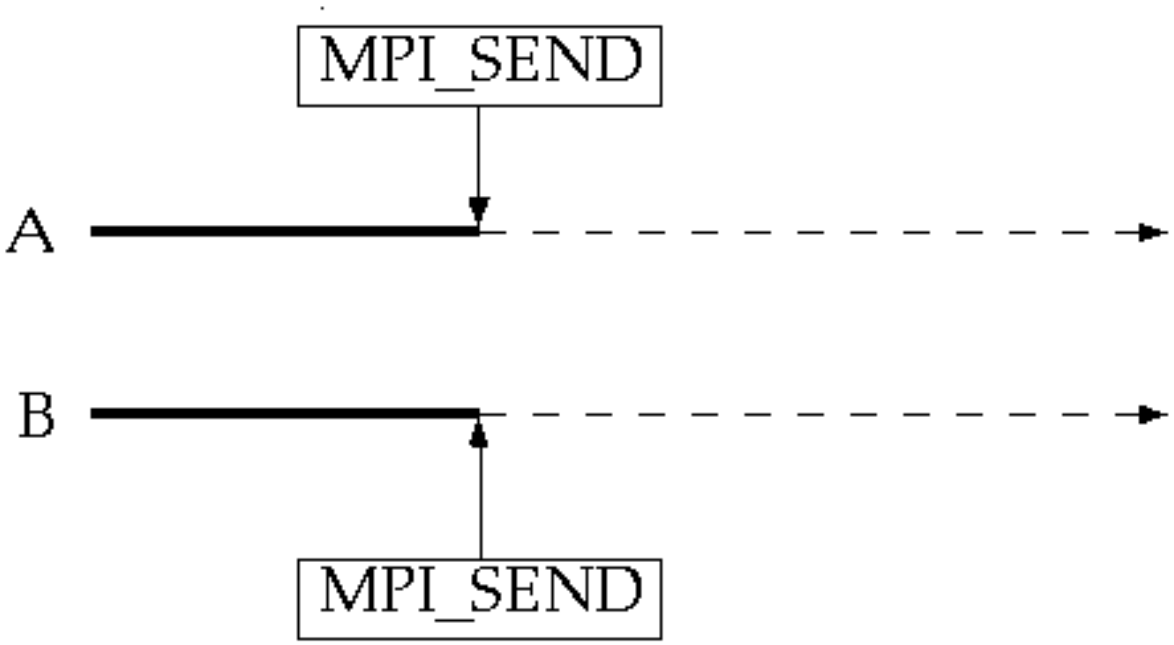


Blocking Standard Send e Blocking Receive
Message > 4K



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 um **Send** e um **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.

9.3.1 - Non-Blocking Synchronous Send

C *int MPI_Issend(*buf, count, datatype, dest, tag, comm, *request)*

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

9.3.2 - Non-Blocking Ready Send

C *int MPI_Irsend(*buf, count, datatype, dest, tag, comm,*request)*

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

9.3.3 - Non-Blocking Buffered Send

C *int MPI_Ibsend(*buf, count, datatype, dest, tag, comm, *request)*

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

9.3.4 - Non-Blocking Standard Send

C *int MPI_Isend(*buf, count, datatype, dest, tag, comm, *request)*

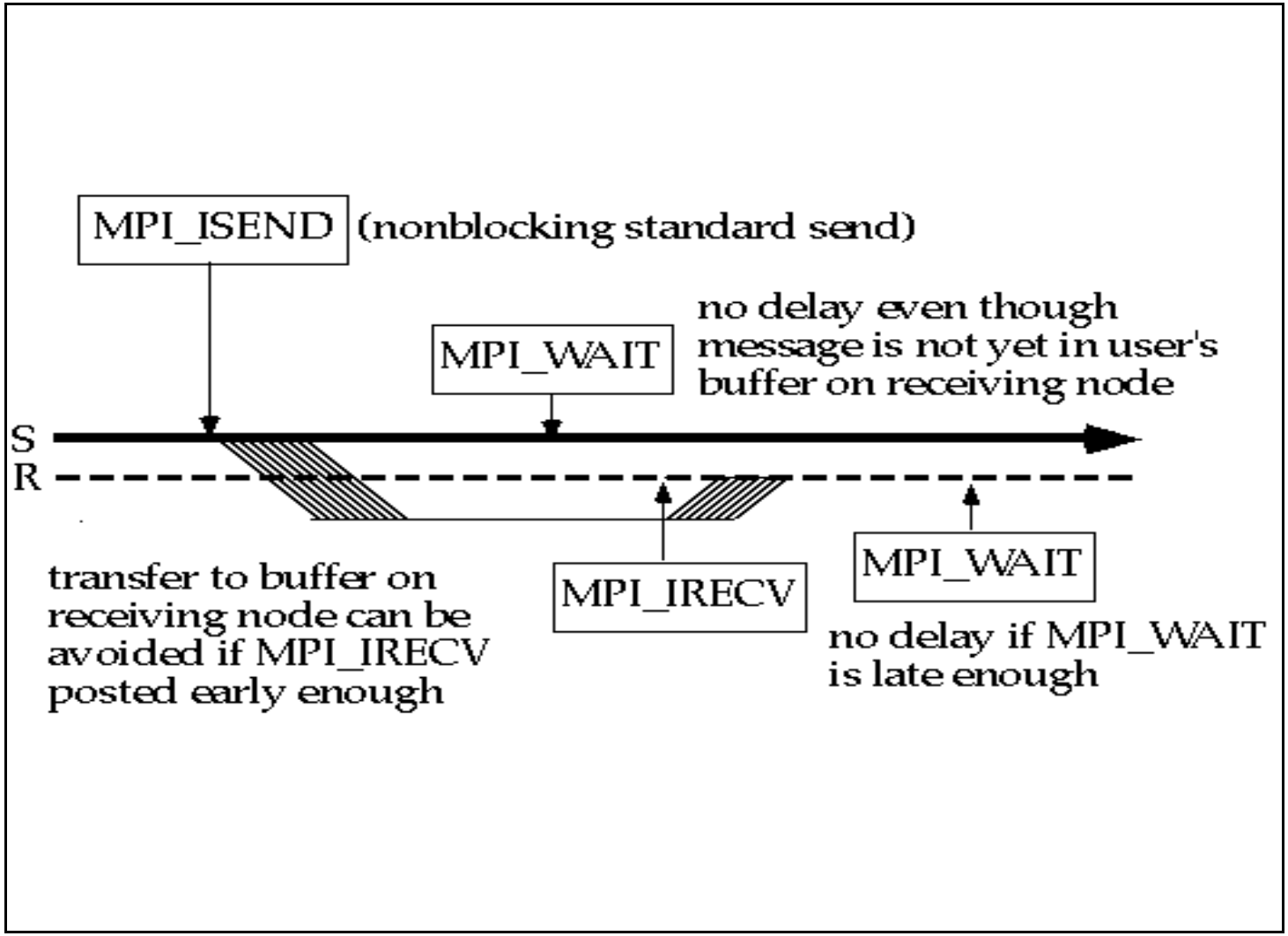
FORTRAN *call MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierror)*

9.3.5 - Non-Blocking Receive

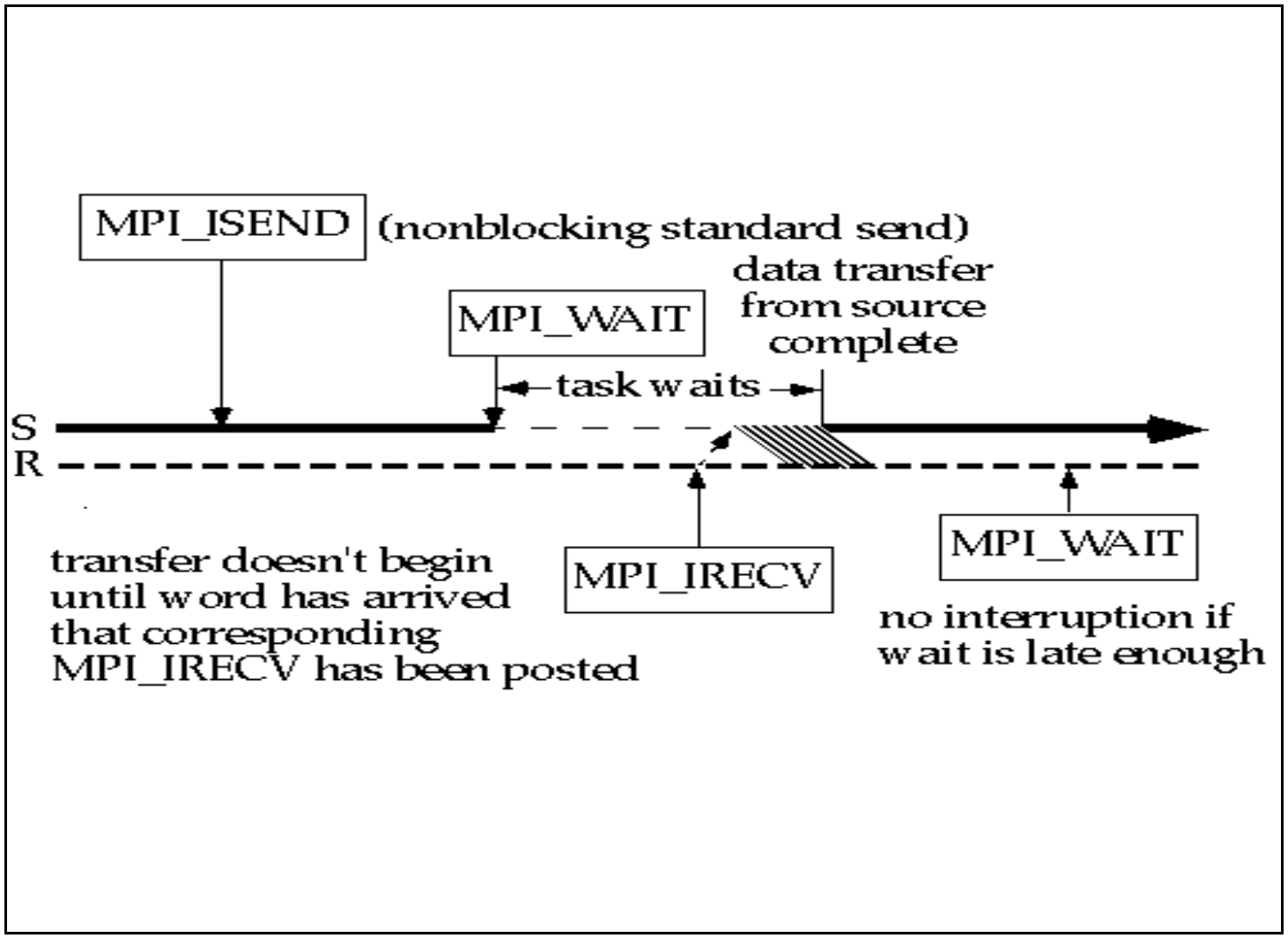
C *int MPI_Irecv(*buf, count, datatype, source, tag, comm, *request)*

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 - Observações

O modo **"Synchronous"** é o mais seguro e ao mesmo tempo, o 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.

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 "system overhead".

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

9.5 - Rotinas Auxiliares

9.5.1 - MPI_Buffer_attach

C ***int MPI_Buffer_attach (*buf, 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.

Define para o MPI, um "buffer" de tamanho específico, para ser utilizado no envio de mensagens. Só é utilizado no modo de "Blocking" ou "Non-blocking" do "Buffered Send".

OBS: Somente um "buffer" poderá ser iniciado por processo, durante a execução da aplicação.

9.5.2 - MPI_Buffer_detach

C ***int MPI_Buffer_detach (*buffer, *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 iniciado anteriormente para o MPI.

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

9.5.3 - MPI_Wait

C ***int MPI_Wait (*request, *status)***

FORTRAN ***call MPI_WAIT (request, status, ierror)***

request Variável inteira “transparente” que questiona o processo de determinadas ações. Parâmetro fornecido pelas rotinas "non-blocking send" ;

status Vetor com informações da mensagem;

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)**.

9.5.4 - MPI_Test

C ***int MPI_Test(*request, *flag, *status)***

FORTRAN ***call MPI_TEST(request, flag, status, ierror)***

request Variável inteira “transparente” que questiona o processo. Parâmetro fornecido pelas rotinas "non-blocking send" ;

flag Variável lógica que identifica o valor de **request**. **MPI_REQUEST_NULL** determina **flag=true** e **MPI_REQUEST_ACTIVE** determina **flag=false**;

status Vetor com informações da mensagem;

ierror Variável inteira com o status da execução da rotina.

Essa rotina apenas informa se uma operação "non-blocking send" foi concluída ou não.

9.5.5 - MPI_Type_size

C ***int MPI_Type_size(datatype,*size)***

FORTRAN ***call MPI_TYPE_SIZE(datatype, size, ierror)***

datatype Tipo do dado;

size Variável inteira com o tamanho, em bytes, para o tipo do dado;

ierror Variável inteira com o status da execução da rotina.

Esta rotina retorna com o tamanho, em bytes, reservado para um tipo de dado.

9.6 – Recomendações

Em geral, é razoável iniciar uma programação MPI utilizando-se de rotinas **"blocking Standard Send"** e **"blocking Receive",** por serem a implementação básica do MPI;

As rotinas “blocking" são necessárias quando se necessita sincronizar processos.

É mais eficiente utilizar rotinas "blocking", quando se utiliza 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";

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 envolve 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".**

Envolve comunicação coordenada entre processos de um grupo, identificados por um **"communicator"**;

Todas as rotinas efetuam **"block"**, até serem localmente finalizadas;

Não é necessário rotular as mensagens (**tags**).

10.1 – Sincronização

BARRIER

C ***int MPI_Barrier (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 implícita 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"

BROADCAST

C ***int MPI_Bcast(*buffer, count, datatype, root, comm)***

FORTRAN ***call MPI_BCAST (buffer, count, datatype, root, comm, ierr)***

buffer Endereço inicial do dado a ser enviado;

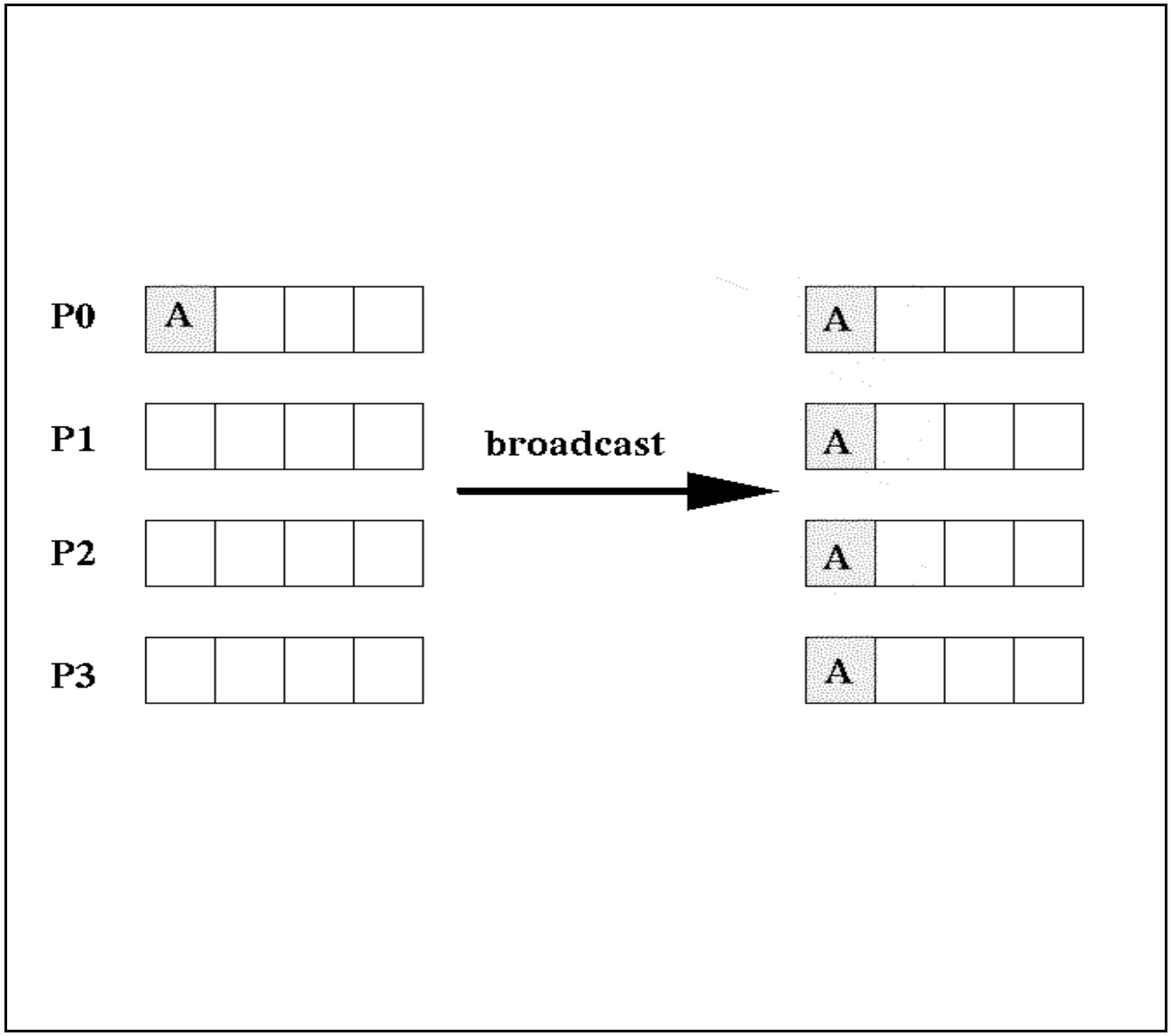
count Inteiro que indica o número de elementos no **buffer**;

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;**

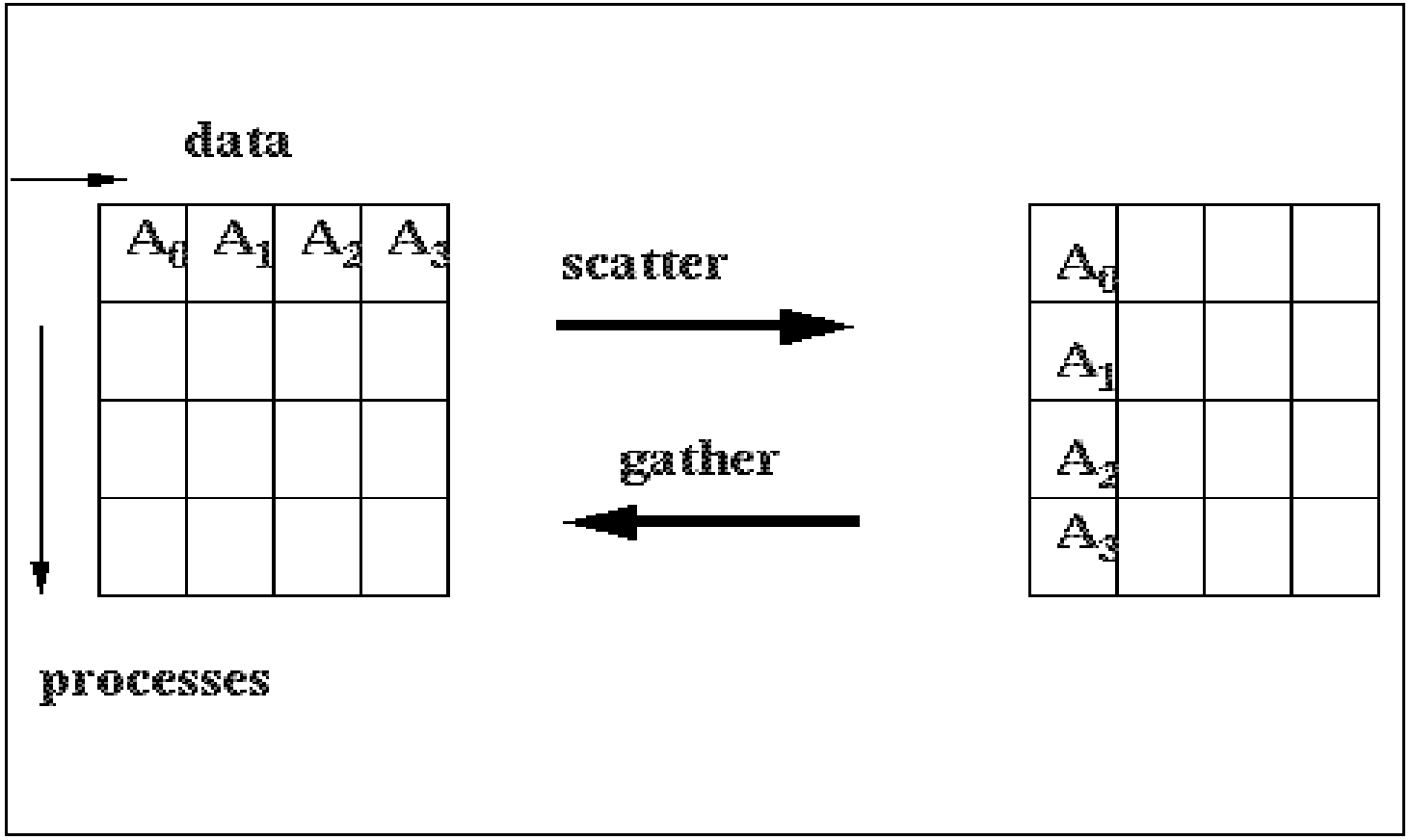
comm Identificação do **communicator**.

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

"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 distribuídos em *n* processos de um grupo. Utiliza a rotina de **GATHER**.



SCATTER

C ***int MPI_Scatter(*sbuf, scount, stype, *rbuf, rcount, rtype, root, 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 que serão distribuídos para cada processo;

stype Tipo de dado que será distribuído;

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

rcount Número de elementos que serão coletados;

rtype Tipo de dado que será coletado;

root Identificação do processo que irá distribuir os dados;

comm Identificação do "communicator".

GATHER

C ***int MPI_Gather(*sbuf, scount, stype, *rbuf, rcount, rtype, root, comm)***

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

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

scount Número de elementos que serão distribuídos para cada processo;

stype Tipo de dado que será distribuído;

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

rcount Número de elementos que serão coletados;

rtype Tipo de dado coletado;

root Identificação do processo que ira coletar os dados;

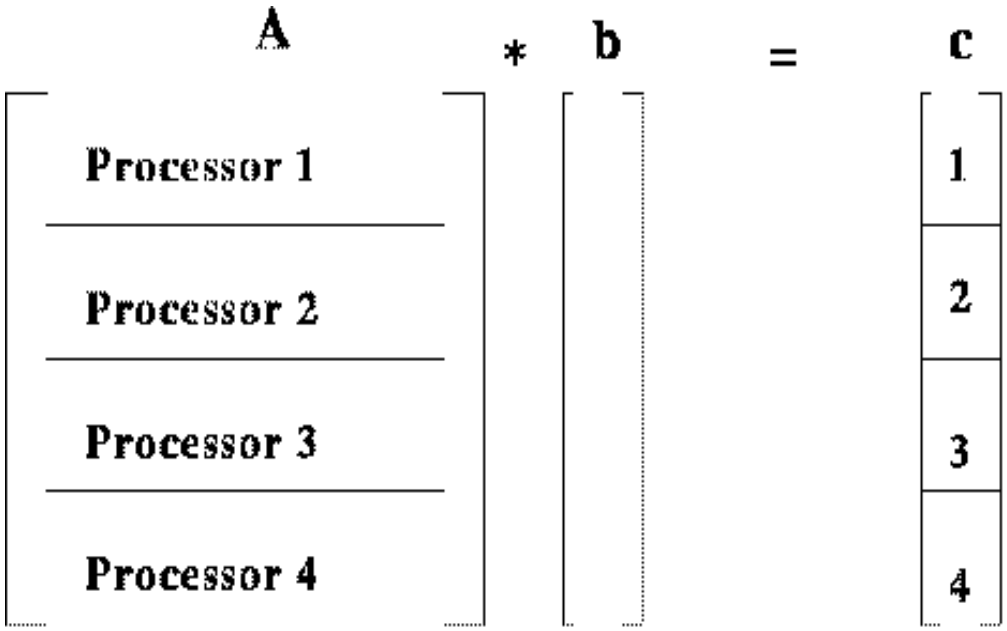
comm Identificação do "communicator".

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)

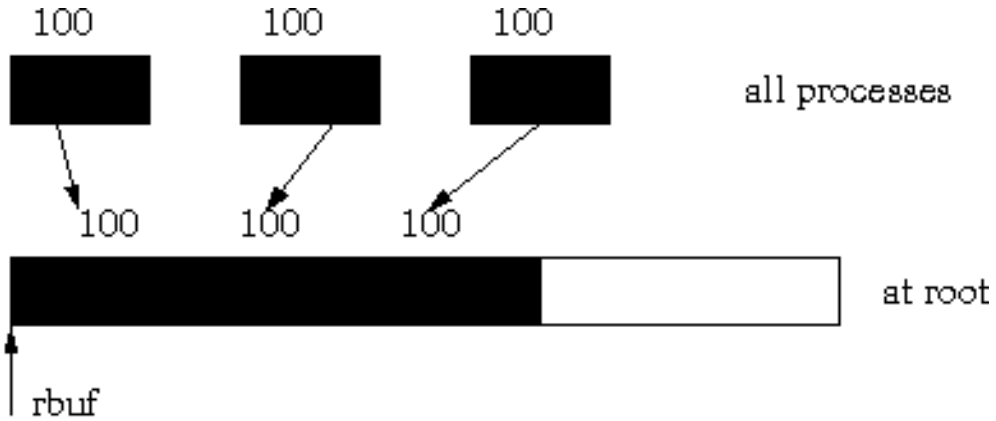


A: Matriz distribuída por linhas;

b: Vetor compartilhado por todos os processos;

c: Vetor atualizado por cada processo, independentemente.

Exemplo 2



real a(100), rbuf(MAX)

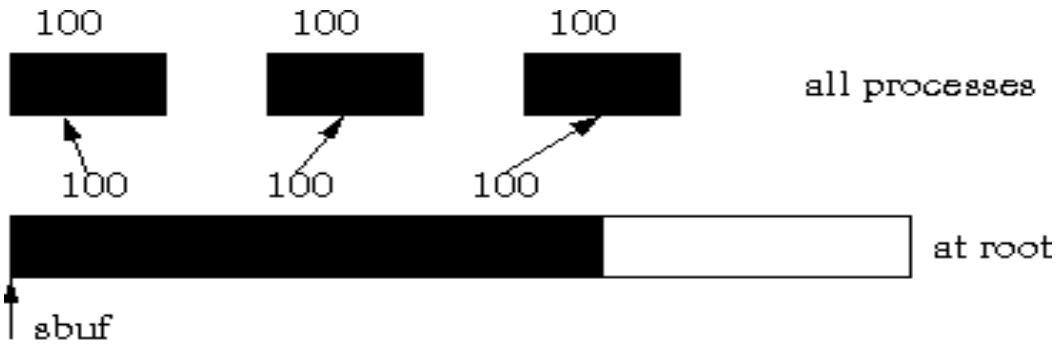
. . .

. . .

. . .

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)
```

ALLGATHER

C ***int MPI_Allgather(*sbuf, scount, stype, *rbuf, rcount, rtype, comm)***

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

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

scount Número de elementos que serão distribuídos;

stype Tipo de dado que será distribuído;

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

rcount Número de elementos que serão coletados;

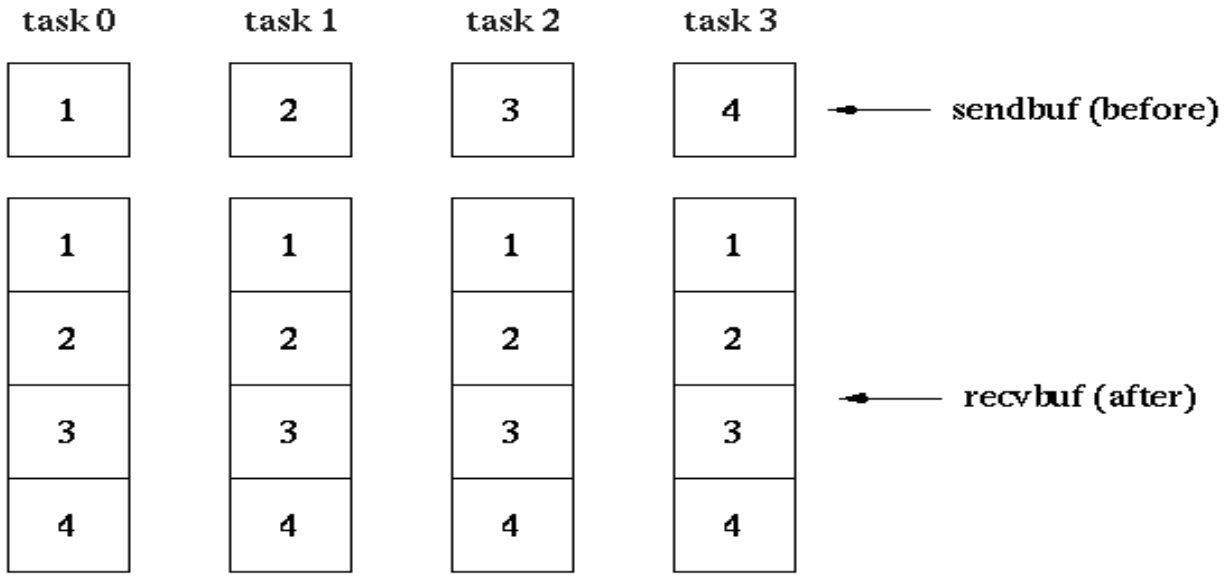
rtype Tipo de dado coletado;

comm Identificação do "communicator".

Essa 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);
```



ALL TO ALL

C ***int MPI_Alltoall(*sbuf, scount, stype, *rbuf, rcount, rtype, comm)***

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

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

scount Número de elementos que serão distribuídos;

stype Tipo de dado que será distribuído;

rbuf Endereço aonde o dados serão coletados ("receive buffer");

rcount Número de elementos que serão coletados;

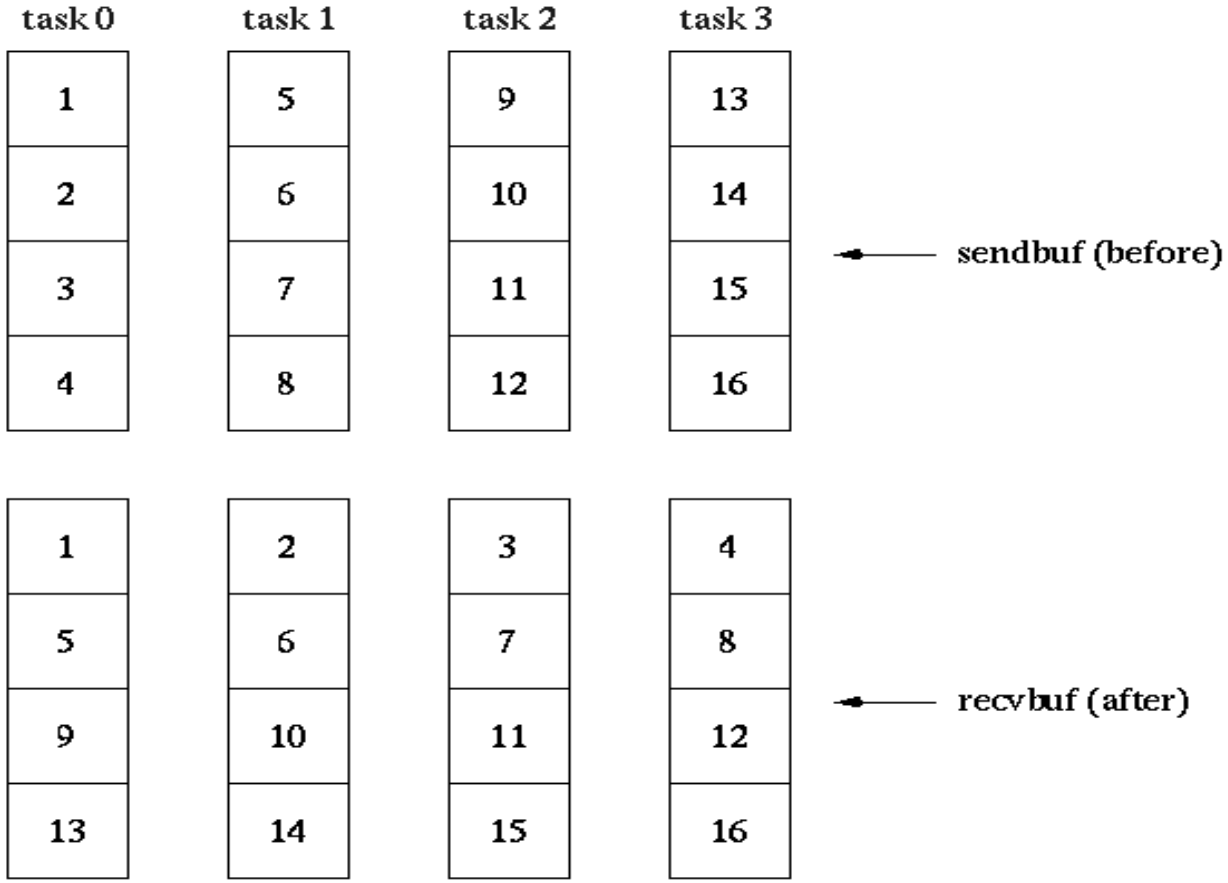
rtype Tipo de dado coletado;

comm Identificação do "communicator".

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);
```



10.3 - Rotinas de Computação Global

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

O resultado parcial de um processo, em um grupo, é combinado e retornado para um específico processo utilizando-se algum **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**

REDUCE

C ***int MPI_Reduce(*sbuf, *rbuf, count, stype, op, root, 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 coletará o resultado da redução ("receive buffer");

count Número de elementos que farão parte da redução;

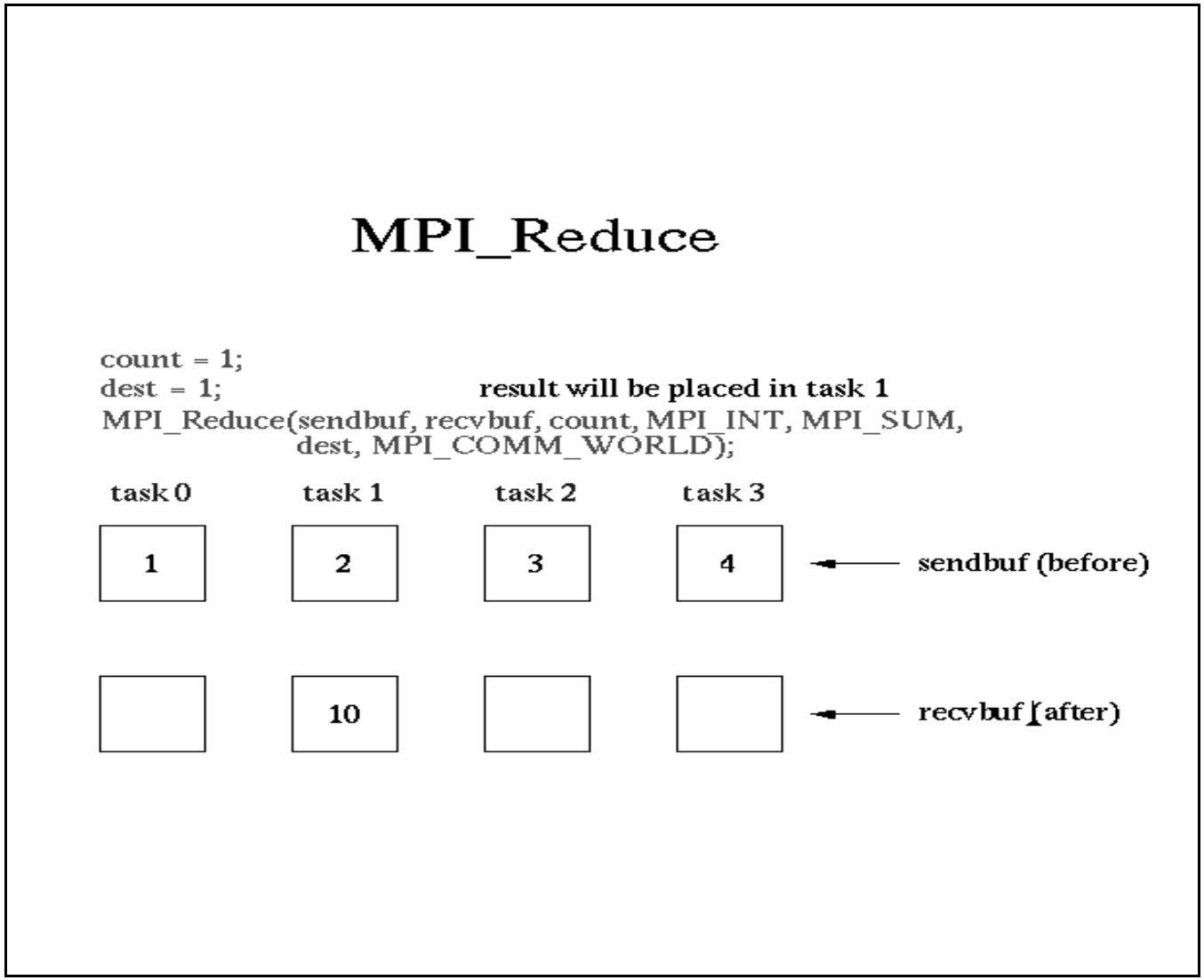
stype Tipo dos dados na operação de redução;

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);
task 0
task 1
task 2
task 3
1
2
3
4
sendbuf (before)
10
recvbuf (after)

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-GRUPOS

Grupo é um conjunto ordenado de processos. Cada processo num grupo, possui um único número de identificação;

O MPI suporta o processamento de grupos, permitindo:

- Organizar processos em grupos, de acordo com a natureza da aplicação;
- Permitir operações de Comunicação Coletiva entre alguns processos;

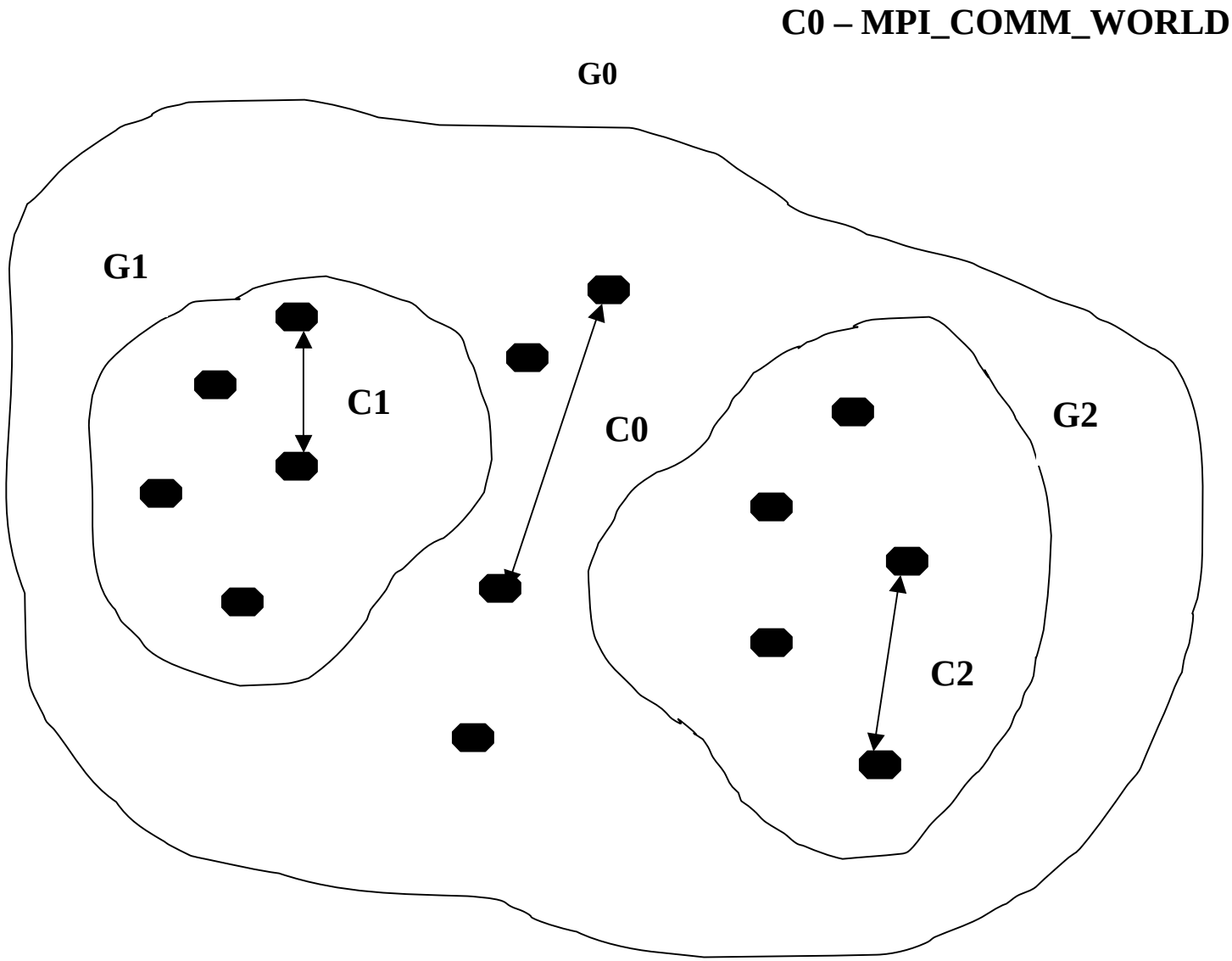
Um processo pode pertencer a mais de um grupo;

Um grupo utiliza um “communicator” específico que descreve o universo de comunicação entre os processos;

Em MPI, grupo é um objeto dinâmico que pode ser criado e destruído durante a execução de um programa.

O MPI possui cerca de 40 rotinas para manipulação e administração de grupos.

GRUPOS e “COMMUNICATOR”



GROUP

C ***int MPI_Comm_group(comm , *group)***

FORTRAN ***call MPI_COMM_GROUP (comm,group, ierr)***

Determina o grupo associado a um determinado “communicator”.

comm “Communicator”

group Variável inteira, “transparente”, de retorno com a identificação do grupo.

ierr Status de execução da rotina.

GROUP INCLUDE

C ***int MPI_Group_incl (group, n, *ranks, *newgroup)***

FORTRAN ***call MPI_GROUP_INCL (group, n, ranks, newgroup, ierr)***

Cria um novo grupo a partir de um grupo existente e somente com os processos identificados.

group Variável inteira, *transparente*, que identifica o grupo que já existe.

n Número de elementos do novo grupo (Também indica o tamanho do conjunto de **r*anks)***.

ranks Vetor com a identificação dos processos no grupo existente que serão incluídos.

newgroup Variável inteira, *transparente*, que irá armazenar a identificação do novo grupo.

ierr Status de execução da rotina.

GROUP EXCLUDE

C ***int MPI_Group_excl (group, n, *ranks, *newgroup)***

FORTRAN ***call MPI_GROUP_EXCL (group, n, ranks, newgroup, ierr)***

Cria um novo grupo a partir de um grupo existente e somente com os processos identificados.

group Variável inteira, *transparente*, que identifica o grupo que já existe.

n Número de elementos do novo grupo (Também indica o tamanho do conjunto de **r*anks***).

ranks Vetor com a identificação dos processos no grupo existente que serão excluídos

newgroup Variável inteira, *transparente*, que irá armazenar a identificação do novo grupo.

ierr Status de execução da rotina.

COMMUNICATOR CREATE

C ***int MPI_Comm_create (comm., group, *newcomm)***

FORTRAN ***call MPI_COMM_CREATE (comm, group, newcomm, ierr)***

Cria um novo "communicator" a partir do "communicator" existente para o novo grupo.

comm "Communicator" no qual pertenciam os processos.

group Variável inteira, *transparente*, que identifica o novo grupo.

newcomm Variável inteira, *transparente*, que irá armazenar a identificação do novo communicator.

ierr Status de execução da rotina.

GROUP FREE

C ***int MPI_Group_free (group)***

FORTRAN ***call MPI_GROUP_FREE (group, ierr)***

Apaga a definição de um grupo.

COMM FREE

C ***int MPI_Comm_free (*comm)***

FORTRAN ***call MPI_COMM_FREE (comm., ierr)***

Apaga a definição de um “communicator”.

Exemplo de GRUPO

```
PROGRAM mainprog
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER                 :: me
  INTEGER                 :: ranks = 0
  INTEGER                 :: send_buf, send_buf2, recv_buf, recv_buf2
  INTEGER                 :: count, count2
  INTEGER                 :: commslave, PI_GROUP_WORLD
  INTEGER                 :: grprem, rstat    ! Status variable

  CALL MPI_Init(rstat)

  CALL MPI_Comm_group(MPI_COMM_WORLD, MPI_GROUP_WORLD, rstat)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, me, rstat)
  CALL MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, grprem, rstat)
  CALL MPI_Comm_create(MPI_COMM_WORLD, grprem, commslave, rstat)

  IF (me /= 0) THEN
    CALL MPI_Reduce(send_buf, recv_buf, count, MPI_INTEGER, &
                    MPI_SUM, 1, commslave, rstat)
  END IF

  CALL MPI_Reduce(send_buf2, recv_buf2, count2, MPI_INTEGER, &
                  MPI_SUM, 0, MPI_COMM_WORLD, rstat)

  IF (commslave /= MPI_COMM_NULL) THEN
    CALL MPI_Comm_free(commslave, rstat)
  END IF
  CALL MPI_Group_free(MPI_GROUP_WORLD, rstat)
  CALL MPI_Group_free(grprem, rstat)
  CALL MPI_Finalize(rstat)
END PROGRAM mainprog
```

12 - REFERÊNCIAS

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

Exercício 1

1 - Caminhe para o diretório do 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.

ifort hello.f -o hello -lmpi
ou
icc hello.c -o hello -lmpi

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

mpirun –np *n* hello ***(n = número de processos)***

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

Esses 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, também, imprimem a mensagem na "saída padrão".

O exercício está em alterar o programa, de maneira que, cada processo escravo imprima a mensagem e 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.

Exercício 3

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

%cd ./mpi/lab01/ex3

Um programa pode utilizar a idéia 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.

Exercício 4

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

%cd ./mpi/lab01/ex4

Neste exercício, o processo mestre inicia um vetor e distribui partes desse 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.

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 implementado, apenas, para a integral, e **corrija dois erros** bem simples de finalização dos resultados, 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 de processamento de um "blocking send").

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

- Todos os processos filhos iniciados, 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 iniciando apenas **dois processos** para que seja possível realizar a medição:

- Execute várias vezes o programa utilizando diferentes tamanhos 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 tenta demonstrar, que a utilização de rotinas "non-blocking" são mais seguras que as rotinas "blocking".

2 - Compile e execute o programa iniciando 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 se possa executa-lo por completo. **O que será necessário alterar ???**

Exercício 3

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

%cd ./mpi/lab02/ex3

Este programa tenta demonstrar 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 iniciando 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:

- Substitua 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, iniciando 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**

- 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 entrada informado;

- o processo com o maior número de identificação calculará o valor médio de todos os números 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;

- serão calculados o valor máximo e o desvio padrão de todos os números randômicos, e os resultados serão distribuídos para todos os processos. Existem dois métodos para 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.

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.

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 redução, **MPI_REDUCE.**

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

2 - Compile o programa e execute-o.

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:

ifort mpi_pi_send.f dboard.f -o mpi_pi_send -lmpi
ou
icc mpi_pi_send.c dboard.c -o mpi_pi_send -lmpi

- Execute o programa:

mpirun –np 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:

ifort mpi_pi_opt.f dboard.f -o mpi_pi_opt -lmpi
ou
icc mpi_pi_opt.c dboard.c -o mpi_pi_opt -lmpi

- Execute o programa:

mpirun –np 4 mpi_pi_opt

4º LABORATÓRIO

Grupo de Processos

Exercício 1

1 - Caminhe para o diretório com o primeiro exercício do laboratório: **%cd ./mpi/lab04/ex1**

Neste exercício, serão criados dois grupos identificados como: processos ímpares e processos pares. **Adicione ao programa, no lugar da "seta", a rotina MPI adequada para esta operação**;