

Centro Nacional de Processamento de Alto Desempenho -SP

TREINAMENTO:

INTRODUÇÃO A OMPI

CONTEÚDO

Introdução a Programação Paralela	pag.02
1-Idéias e Conceitos.....	pag.02
2-Comunicação entre Processadores.....	pag.06
3-Criação de um Programa Paralelo.....	pag.08
4-Considerações de Performance.....	pag.10
5-Referências.....	pag.12
Introdução a "Message-Passing"	pag.13
1-O modelo: "Message-Passing".....	pag.13
2-Bibliotecas "Message-Passing".....	pag.14
3-Referências.....	pag.19
Introdução ao MPI	pag.20
1-O que é MPI?.....	pag.20
2-Histórico.....	pag.21
3-Conceitos e Definições.....	pag.23
4-Compilação.....	pag.29
5-Execução.....	pag.30
6-Rotinas Básicas.....	pag.31
6.1-Inicializar um processo MPI.....	pag.31
6.2-Identificar um processo MPI.....	pag.32
6.3-Contar processos MPI.....	pag.33
6.4-Enviar mensagens no MPI.....	pag.34
6.5-Receber mensagens no MPI.....	pag.35
6.6-Finalizar processos no MPI.....	pag.36
7-MPI Message.....	pag.37
7.1-Dado.....	pag.38
7.2-Envelope.....	pag.39
8-Exemplo de um programa básico MPI.....	pag.40
9-Rotinas de Comunicação "Point-to-Point"	pag.41
9.1-Modos de Comunicação.....	pag.42
9.1.1-"Blocking Synchronous Send".....	pag.44
9.1.2-"Blocking Ready Send"	pag.46
9.1.3-"Blocking Buffered Send"	pag.48
9.1.4-"Blocking Standard Send"	pag.50
9.2-"Deadlock".....	pag.53
9.3-Comunicação "Non-blocking".....	pag.54
9.4-Conclusão.....	pag.59
9.5-Rotinas auxiliares.....	pag.60
9.6-Recomendações.....	pag.66
10-Rotinas de Comunicação Coletiva.....	pag.67
10.1-Sincronização.....	pag.68
10.2-"Data Movement".....	pag.69
10.2.1-"Broadcast".....	pag.69
10.2.2-"Gather" e "Scatter".....	pag.71
10.2.3-"Allgather".....	pag.77
10.2.4-"AlltoAll".....	pag.79
10.2.5-Rotinas de Computação Global.....	pag.81
11-Referências.....	pag.85
12-Laboratórios.....	pag.86

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, coordená-las leve mais tempo que executá-las.

Ex: Construção Civil, Indústria 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 do 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 influenciarnameneiradeseescreverumprogramaparalelo.

Memória Compartilhada

- Múltiplos processadores operam de maneira independente, mas compartilhamos 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: *Limitação número de processadores.
- Exemplos: Cray Y-MP, Convex C-2, Cray C-90.

Memória Distribuída

- 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 existem limites no número de processadores;
 - * Cada processador acessa, sem interferência e rapidamente, sua própria memória.
- Desvantagens:
 - * Elevado "overhead" devido à 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 ideia de decomposição funcional e de decomposição de domínio.

1- Decomposição Funcional

- O problema é decomposto em **diferentes** tarefas que serão distribuídas entre múltiplos processadores para execução simultânea;
- 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 imagens e matrizes e para cálculos 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 sabertirar vantagens ao se utilizar uma "Message-PassingLibrary". A programação da comunicação é explícita.
- "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 "DataParallel", são exatamente os mesmos.

*PointtoPoint

*OnetoAllBroadcast

*AlltoAllBroadcast

*OnetoAllPersonalized

*AlltoAllPersonalized

*Shifts

*CollectiveComputations

4-CONSIDERAÇÕES DE PERFORMANCE

Amdahl's Law

A lei de Amdahl determina o potencial de aumento de velocidade de um programa (partido por uma porcentagem paralelizável de um programa) (f):

$$\text{speedup} = 1/(1-f)$$

- Num programa, no qual não ocorra paralelização, $f=0$, 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 paralelizável de processamento, a relação passará a ser modelada por:

$$\text{speedup} = 1/[(P/N)+S]$$

P=Porcentagem paralelizável;

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 entre os processadores, deverá ser, de maneira que o tempo de 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.

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 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

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

- 1- SPParallelProgrammingWorkshop**
IntroductiontoParallelProgramming
MHPCC-MauiHighPerformanceComputingCenter
BlaiseBarney-02Julho1996

- 2- Seminário:OverviewofParallelProcessing**
KevinMoroneyeJeffNucciarone-17Outubro1995

INTRODUÇÃO A "MESSAGE-PASSING"

1-O MODELO: "MESSAGE-PASSING"

O modelo "Message-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 baseada 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 gerenciamento 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 comunicação 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 deliberar 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 de 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).

Blocking 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":

"SystemOverhead"

É trabalho efetuado pelo sistema para transferir um dado para seu processador de destino;

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

"SincronizationOverhead" É tempo gasto na espera de que um evento ocorra em um outro processo;

Ex.: Espera, pelo processo origem, do sinal de OK pelo processador de 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);
- * Utilizar 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- SPParallelProgrammingWorkshop

MessagePassingOverview

MHPCC-MauiHighPerformanceComputingCenter

BlaiseBarney-03Julho1996

2- MPI:TheCompleteReference

TheMITPress-Cambridge,Massachusetts

MarcSnir,SteveOtto,StevenHuss-Lederman,DavidWalker

JackDongarra

INTRODUÇÃO AO MPI

1-O QUE É MPI?

Message Passing Interface

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

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

Utilizado por programas em C e FORTRAN.

A plataforma alvo para o MPI, são ambientes de memória distribuída, 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 como 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 distribuída (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 adotou 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.

Mai de 1994: Disponibilização da versão padrão do MPI (domínio público - **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 Público

UNIFY: Mississippi State University

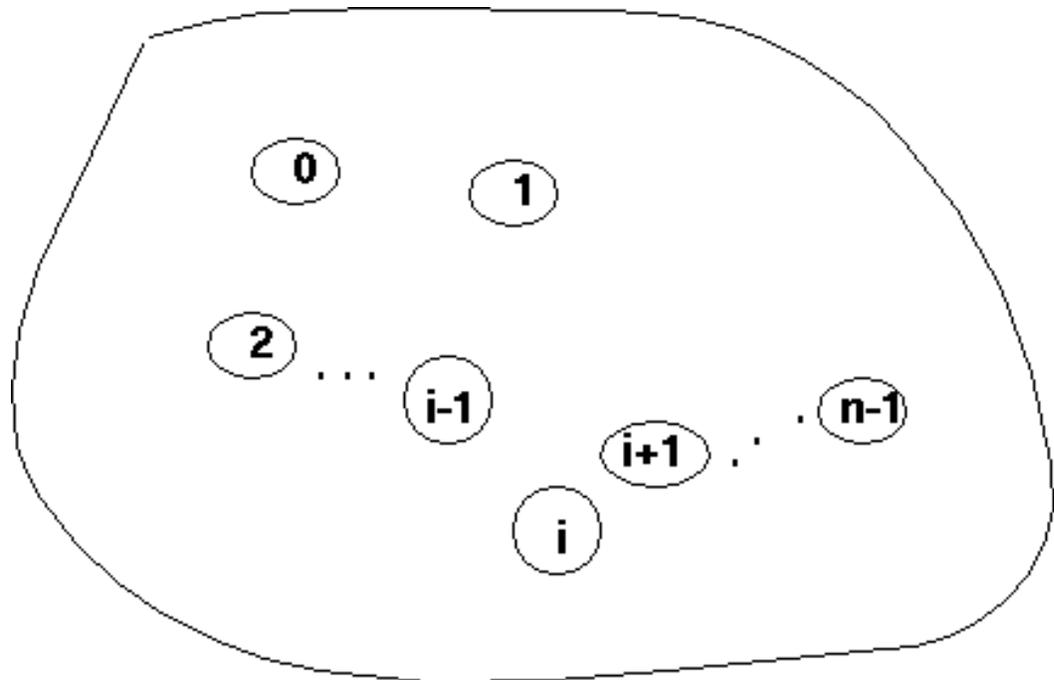
CHIMP: Edinburgh Parallel Computing Center

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 contínua 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 comum "communicator" já pré-estabelecido (MPI_COMM_WORLD).

Communicator

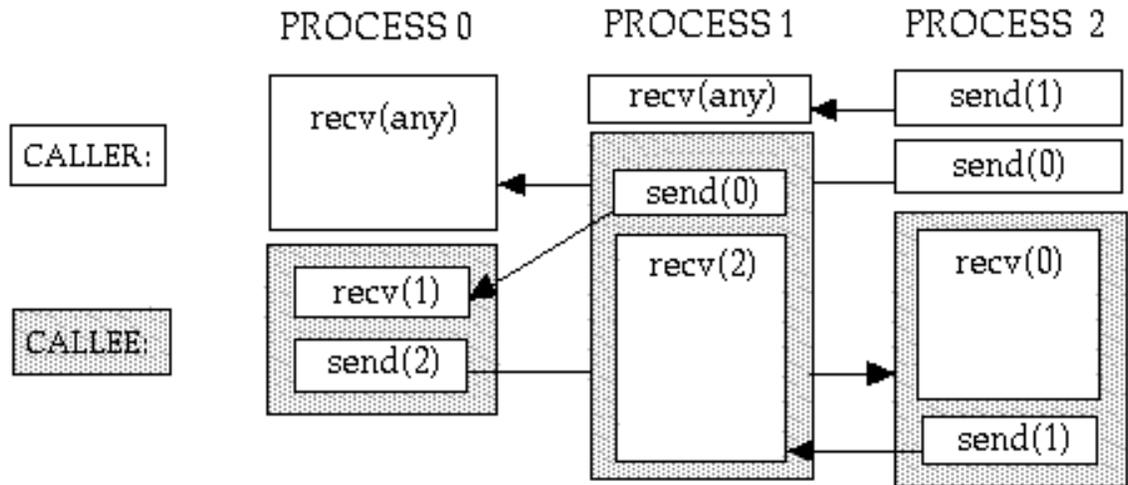
O "communicator" define uma coleção de processos (grupo), que poderão se comunicar entre si (contexto).

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 definidos pelo usuário, numa aplicação MPI.

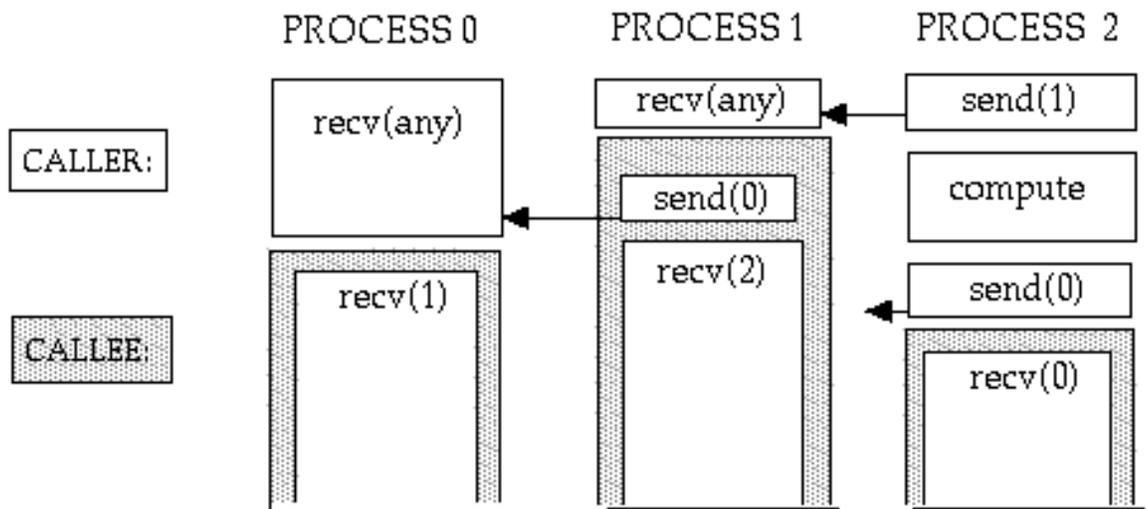
AÇÃODESEJADA



Courtesy David Walker
Oak Ridge Nat. Lab.

RLF.COMMDES 10/16/95

POSSÍVELAÇÃOINDESEJADA



Courtesy David Walker
Oak Ridge Nat. Lab.

RLF.COMMINC 10/16/95

ApplicationBuffer

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

SystemBuffer

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

Dependendo do tipo de operação de **send/receive**, o dado no "application buffer" pode necessitar ser copiado **de/para** o "systembuffer" ("**SendBuffer**" e "**ReceiveBuffer**"). Neste caso teremos comunicação assíncrona.

BlockingCommunication

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 "application buffer" pode ser reutilizado.

Numa rotina de recebimento, o dado tem que ser armazenado no "systembuffer", indicando que o dado pode ser reutilizado.

Non-Blocking Communication

Uma rotina de comunicação é dita **"Non-blocking"**, se a chamadaretornasemesperarqualeventoqueindiqueofim ouosuccessodarotina.

Ex: Nãoesperapela cópiademensagensdo "application buffer" para o "system buffer", ou a indicação do recebimentodeumamensagem.

OBS: Édaresponsabilidadedoprogramador,acertezade queo "application buffer" esteja disponível para ser reutilizado.

Estetipodecomunicaçãoéutilizadoparamelhorara relaçãoentrecomputaçãoecomunicaçãooparaefeitos deganhodeperformance.

StandardSend

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

SynchronousSend

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

BufferedSend

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 "SystemBuffer".

ReadySend

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

StandardReceive

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

ReturnCode

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 *Mississippi 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 um único comando, as tarefas de compilação e linkedição.

FORTRAN77

mpif77-o<executável><fonte>

CStandard

mpicc-o<executável><fonte>

OBS: É possível utilizar todas as opções de compilação dos compiladores CeFORTRAN.

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	Mantém 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-np6teste

mpirun-archsun4-np2-archrs6000-np3teste.%a

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

6-ROTINASBÁSICAS

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

6

6.1-InicializarumprocessoMPI

MPI_INIT

- PrimeirarotinadoMPIaserchamadaporcadaprocesso.
- Estabeleceoambiente necessário para executar o MPI.
- Sincronizatodososprocessosnainicializaçãodeumaaplicação 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-IdentificarprocessodoMPI

MPI_COMM_RANK

- Identificaoprocesso,dentrodeumgrupodeprocessos.
- Valorinteiro,entre0en-1processos.

C

*intMPI_Comm_rank(MPI_Commcomm,int*rank)*

FORTRAN

callMPI_COMM_RANK(comm,rank,mpierr)

comm	MPIcommunicator.
rank	Variável inteira de retorno com o número de identificação do processo.
mpierr	Variávelinteiraderetornocomostatusdarotina.

6.3-ContarprocessosnoMPI

MPI_COMM_SIZE

- Retornaonúmerodeprocessosdentrodeumgrupodeprocessos.

C

*intMPI_Comm_size(MPI_Commcomm,int*size)*

FORTRAN

callMPI_COMM_SIZE(comm,size,mpierr)

comm MPICommunicator.

size Variável inteira de retorno com o número de processosinicializadosduranteumaaplicaçãoMPI .

mpierr Variávelinteiraderetornocomostatusdarotina

6.4-EnviarmenssagensnoMPI

MPI_SEND

- "Blockingsend".
- Arotinaretornaapósodadotersidoenviado.
- Após retorno, libera o "system buffer" e permite acesso ao "applicationbuffer".

C

*intMPI_Send(void*sndbuf,intcount,MPI_Datatype datatype,intdest,int tag,MPI_Commcomm)*

FORTRAN

callMPI_SEND(sndbuf,count,datatype,dest,tag, comm,mpierr)

sndbuf Endereçoinicialdodadosqueseráenviado.Endereçodo "applicationbuffer".

count Número de elementos a serem enviados.

datatype Tipododado.

dest Identificaçãodoprocessodestino.

tag Rótulodamensagem.

comm MPIcommunicator.

mpierr Variávelinteiraderetornocomostatusdarotina.

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,intcount,MPI_Datatype datatype,intsource,
inttag,*status,MPI_Commcomm)*

FORTRAN

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

recvbuf Variável indicando o endereço do "application 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-FinalizarprocessosnoMPI

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-MPIMessage

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 "MPIMessage" contém duas partes: **dado**, na qual se deseja enviar ou receber, e o **envelope** com informações de rotados dados.

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

Ex.:

```
callMPI_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	signedchar
MPI_SHORT	signedshortint
MPI_INT	signedint
MPI_LONG	signedlongint
MPI_UNSIGNED_CHAR	unsignedchar
MPI_UNSIGNED_SHORT	unsignedshortint
MPI_UNSIGNED	unsignedint
MPI_UNSIGNED_LONG	unsignedlongint
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	longdouble
MPI_BYTE	
MPI_PACKED	

Tipos Básicos de Dados para FORTRAN

MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLEPRECISION
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-ExemplodeumProgramaBásicoMPI

```

    programhello
include'mpif.h'
integerme,nt,mpierr,tag,status(MPI_S      TATUS_SIZE)
character(12)message

    callMPI_INIT(mpierr)
callMPI_COMM_SIZE(MPI_COMM_WORLD,nt,mpi      err)
callMPI_COMM_RANK(MPI_COMM_WORLD,me,mpi      err)

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

    callMPI_SEND(message,12,MPI_CHARACTER,i,tag,
&MPI_COMM_WOR      LD,mpierr)

enddo
else

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

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

    callMPI_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-ModosdeComunicação

"BlockingReceive"

Existemquatrotiposde"blockingsend",umaparacadamodode comunicação,masapenasum"blockingreceive"parareceberos dadosdequalquer"blockingsend".

C

```
intMPI_Recv(void*buf,intcount,MPI_Datatypedatatype,intsource  
inttag,MPI_Commcomm,MPI_Statusstatus)
```

FORTRAN

```
callMPI_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 de 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- "BlockingSynchronousSend"

C

```
intMPI_Ssend(void*buf,intcount,MPI_Datatypedatatype,intdest,  
            inttag,MPI_Commcomm)
```

FORTRAN

```
callMPI_SSEND(buf,count,datatype,dest,tag,comm,ierror)
```

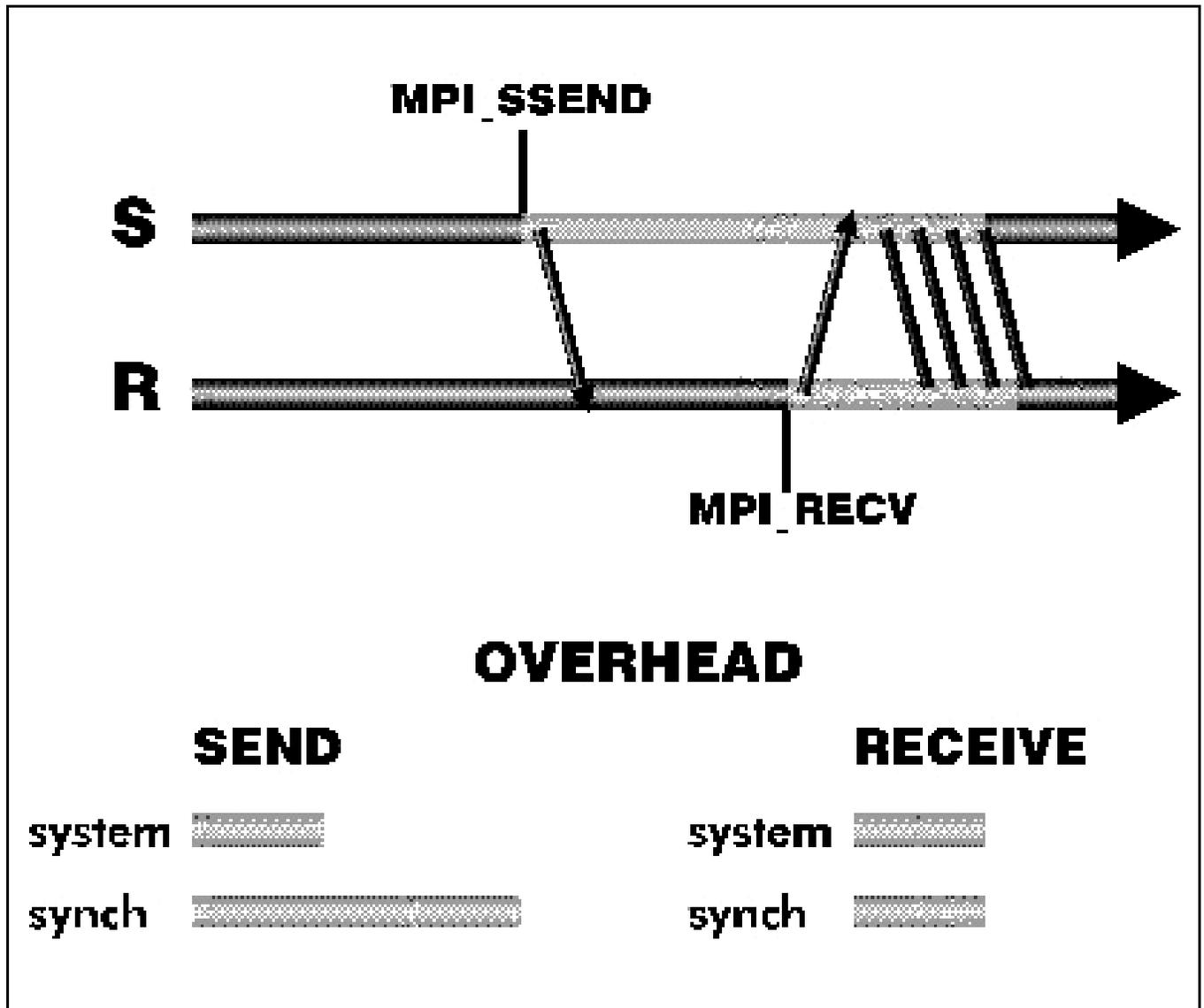
Quando um **MPI_Ssend** é executado, o processo que envia avisa ao processo que recebe que uma mensagem está pronta e **esperando** por um sinal de OK, para que então, seja transferido o dado.

OBS: "**System overhead**" ocorre devido a cópia da mensagem do "sendbuffer" para a rede e de para o "receivebuffer".

"**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 significativo.

BlockingSynchronousSendeBlockingReceive



9.1.2-"BlockingReadySend"

C

```
intMPI_Rsend(void*buf,intcount,MPI_Datatypedatatype,intdest,  
             inttag,Mpi_Commcomm)
```

FORTTRAN

```
callMPI_RSEND(buf,count,datatype,dest,tag,comm,ierror)
```

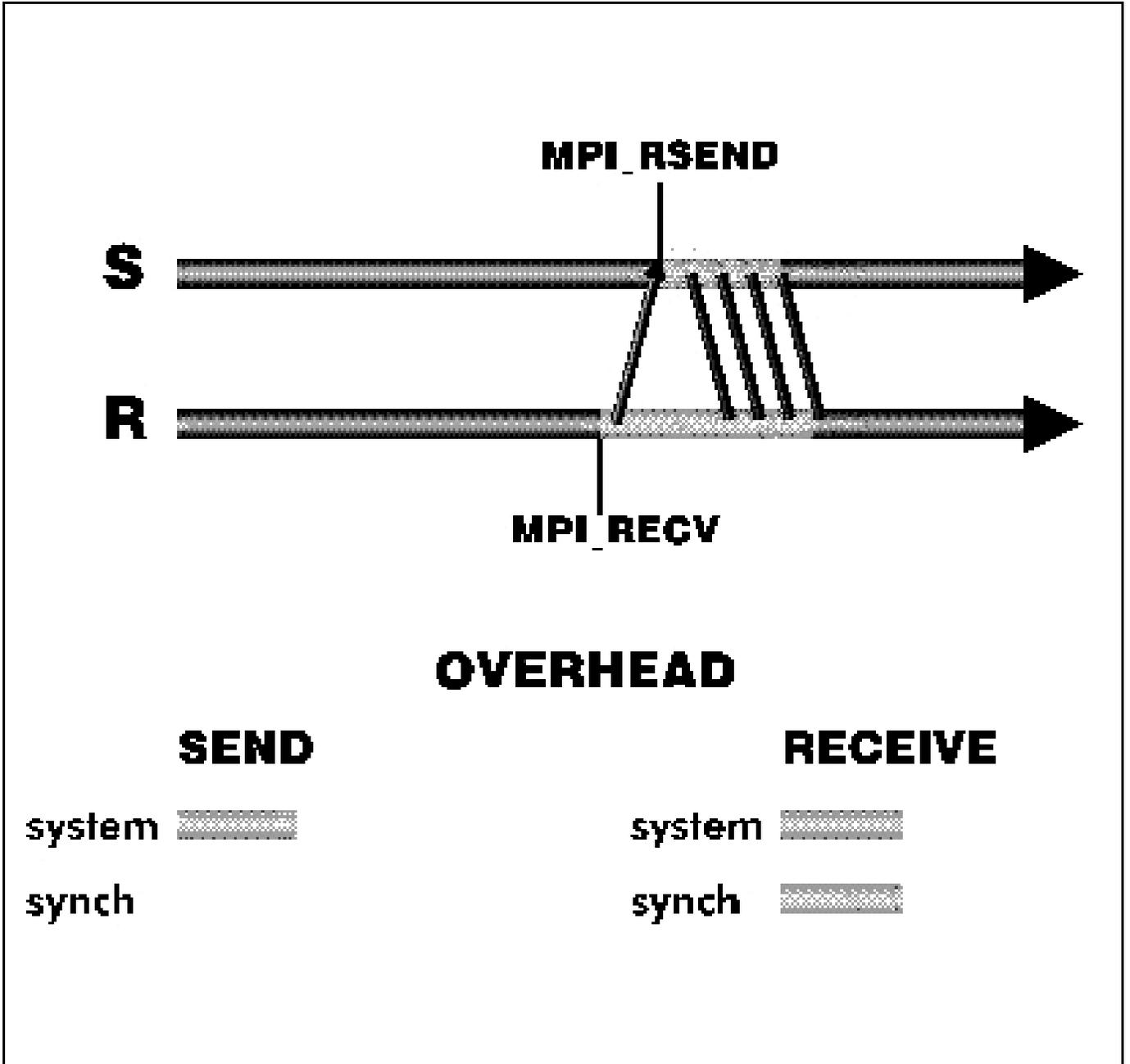
Quando um **MPI_Rsend** é executado a mensagem é enviada imediatamente para a rede. É exigido que um sinal de OK do processo que irá receber, já tenha sido feito.

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**.

BlockingReadySendeBlockingReceive



9.1.3-"BlockingBufferedSend"

C

```
intMPI_Bsend(void*buf,intcount,MPI_Datatypedata_type,intdest,
             inttag,MPI_Commcomm)
```

FORTRAN

```
callMPI_BSEND(buf,count,datatype,dest,tag,comm,ierror)
```

Quando um **MPI_Bsend** é executado a mensagem é copiada do endereço de memória ("Applicationbuffer") 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 "**Systemoverhead**" devido a cópia da mensagem do "Applicationbuffer" para o "buffer" definido pelo usuário.

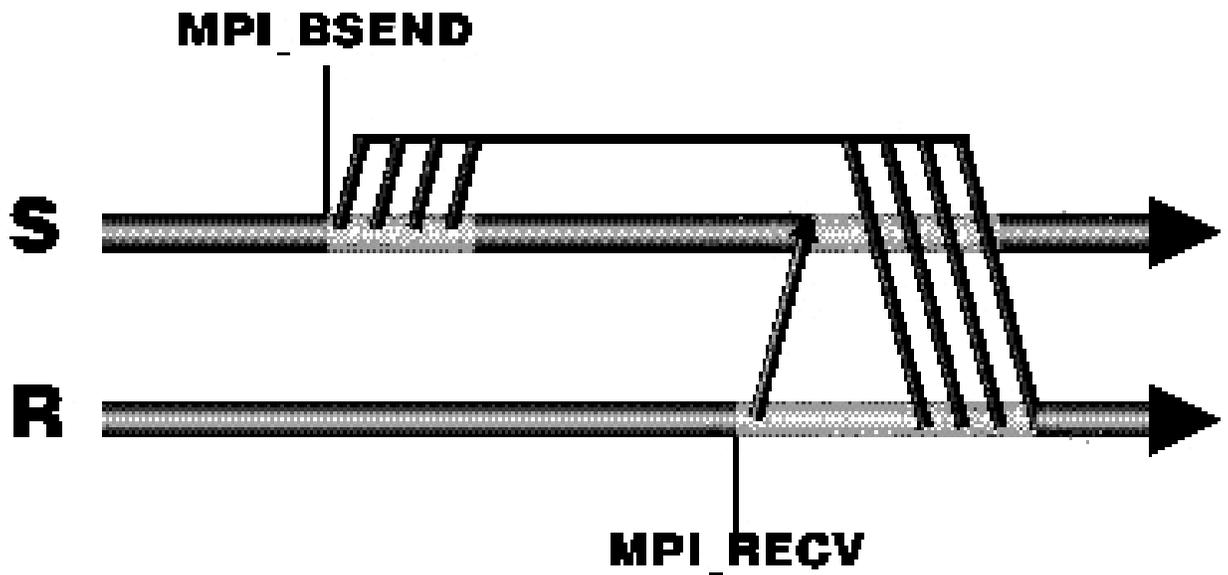
O "**Synchronizationoverhead**", não existe no processo que envia, mas é significativo no processo que recebe, caso seja executado o "receive" antes de um "send".

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

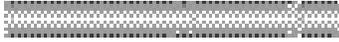
MPI_Buffer_detach

BlockingBufferedSend e BlockingRceive

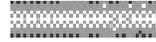


OVERHEAD

SEND

system 
synch 

RECEIVE

system 
synch 

9.1.4- "BlockingStandardSend"

C

```
intMPI_Send(void*buf,intcount,MPI_Datatype datatype,intdest,
            inttag,MPI_Commcomm)
```

FORTRAN

```
callMPI_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 processadores 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 "Systembuffer" do processo que irá receber a mensagem.

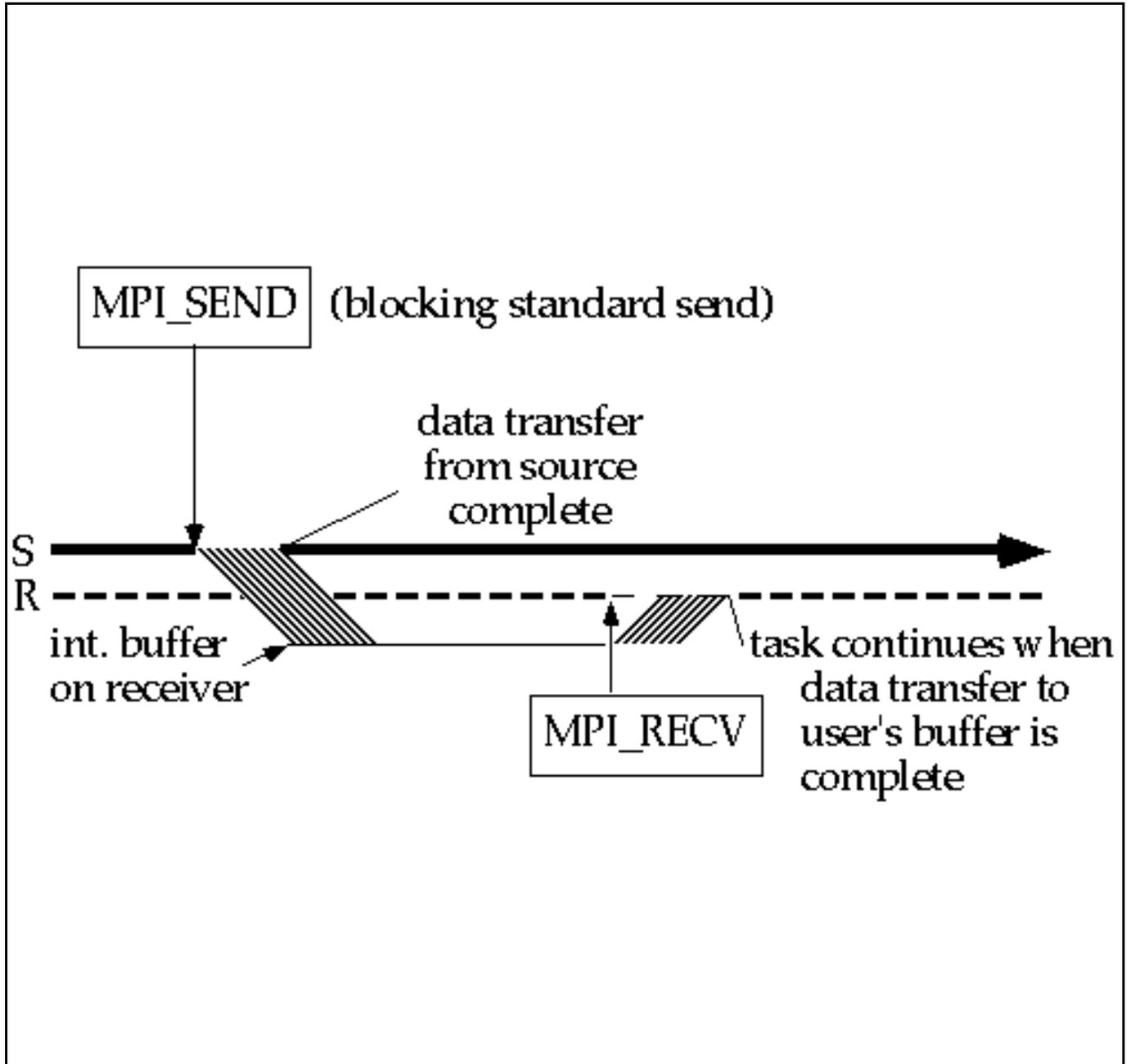
OBS: O **"Synchronizationoverhead"** é reduzido ao preço de se aumentar o **"System overhead"** devido as cópias extras que podem ocorrer, para o buffer.

Message>4K

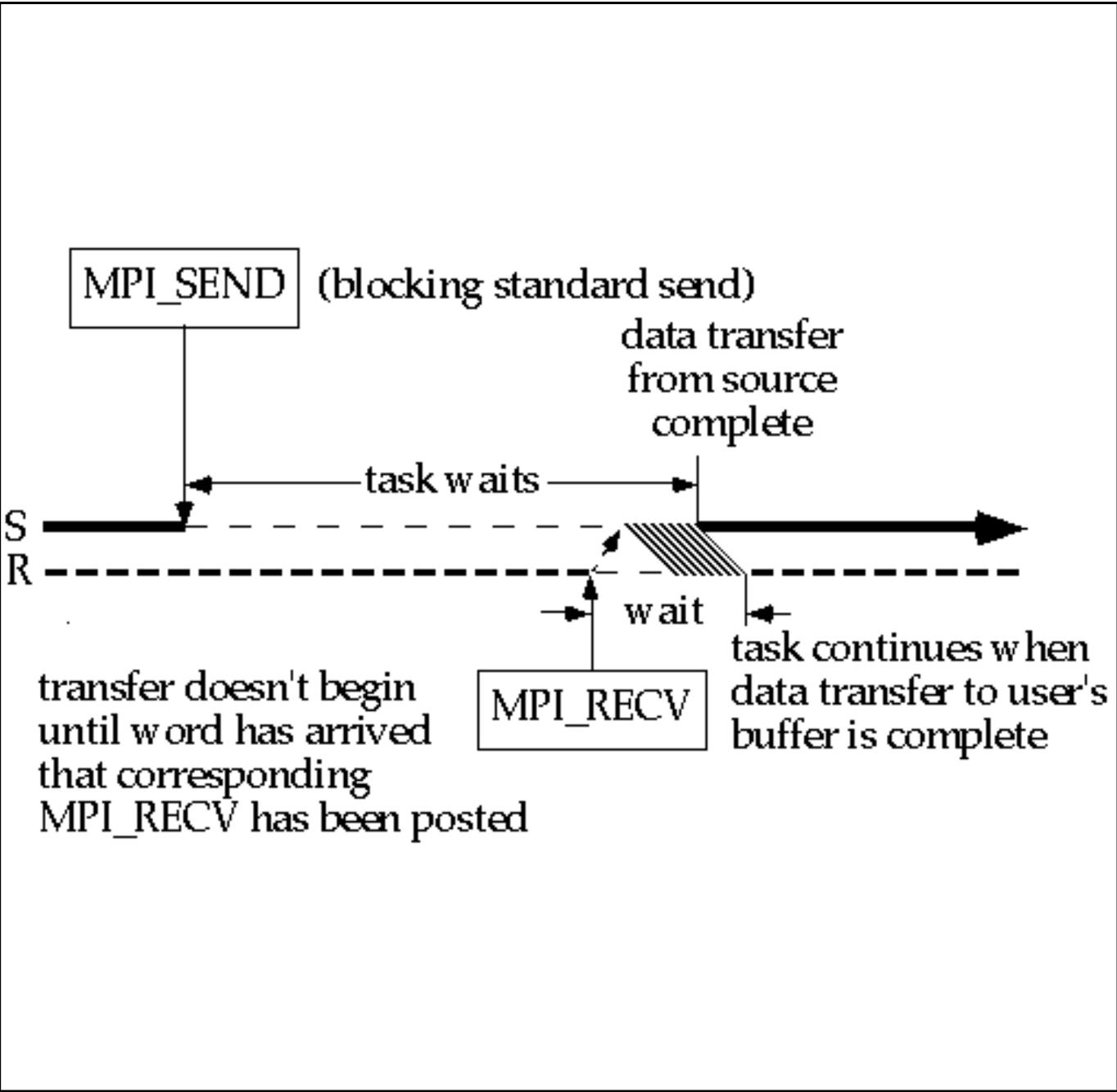
Quando **MPI_Send** é executado a mensagem é transmitida essencialmente igual ao modo "Synchronous".

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

BlockingStandardSendBlockingReceive Message \leq 4K

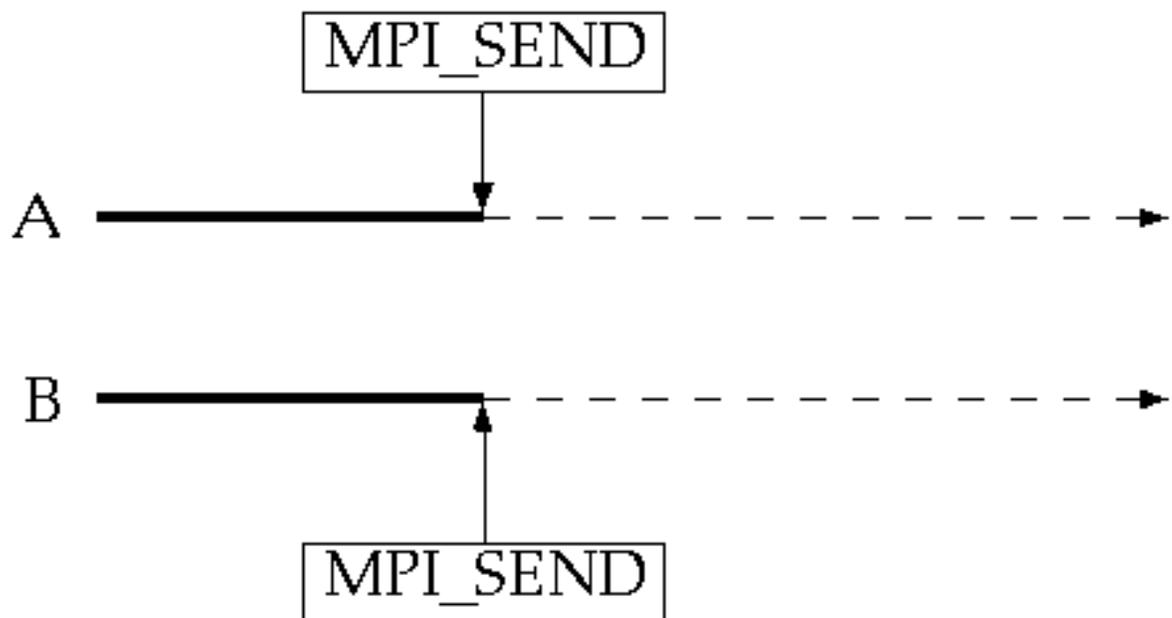


BlockingStandardSendBlockingReceive Message>4K



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, existam 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 "systembuffer" estar pronto para uso.

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

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

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

MPI_Wait

MPI_Test

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

Non-BlockingSynchronousSend

C

```
int MPI_Issend(void*buf,intcount,MPI_Datatypedata type,intdest,int
tag,
MPI_Comm
comm,MPI_Request*request)
```

FORTRAN

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

Non-BlockingReadySend

C

```
int MPI_Irsend(void*buf,intcount,MPI_Datatypedata type,intdest,
inttag,MPI_Commcomm,MPI_Request*request)
```

FORTRAN

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

Non-BlockingBufferedSend

C

```
int MPI_Ibsend(void*buf,intcount,MPI_Datatypedat atype,intdest,
inttag,MPI_Commcomm,MPI_Request*request)
```

FORTRAN

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

Non-BlockingStandardSend

C

*intMPI_Isend(void*buf,intcount,MPI_Datatypesdatatype,intdest,
inttag,MPI_Commcomm,MPI_Request*request)*

FORTTRAN

callMPI_ISEND(buf,count,datatype,dest,tag,comm,request,ierror)

Non-BlockingReceive

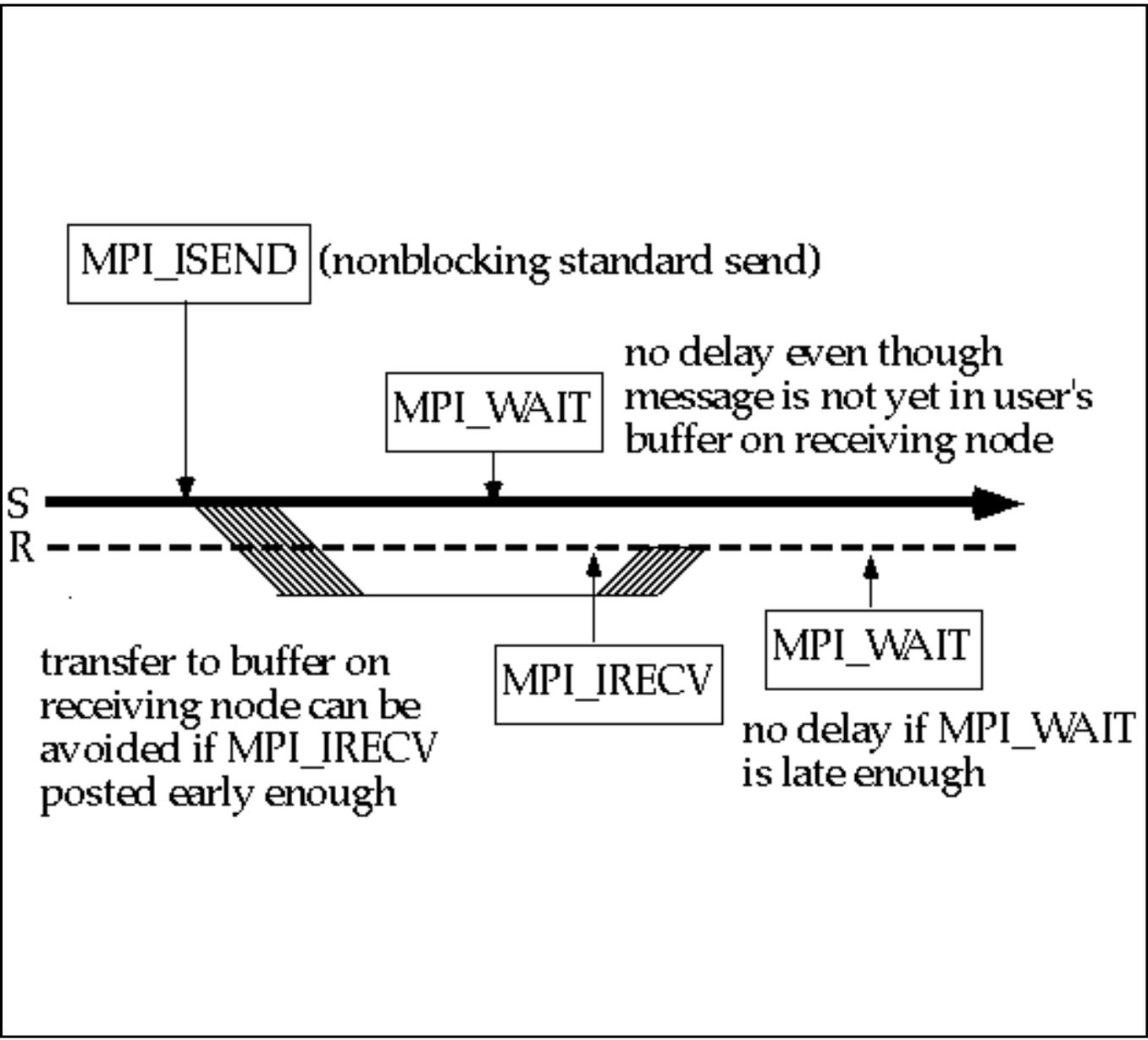
C

*intMPI_Irecv(void*buf,intcount,MPI_Datatypesdatatype,intsource,
inttag,MPI_Commcomm,MPI_Request*request)*

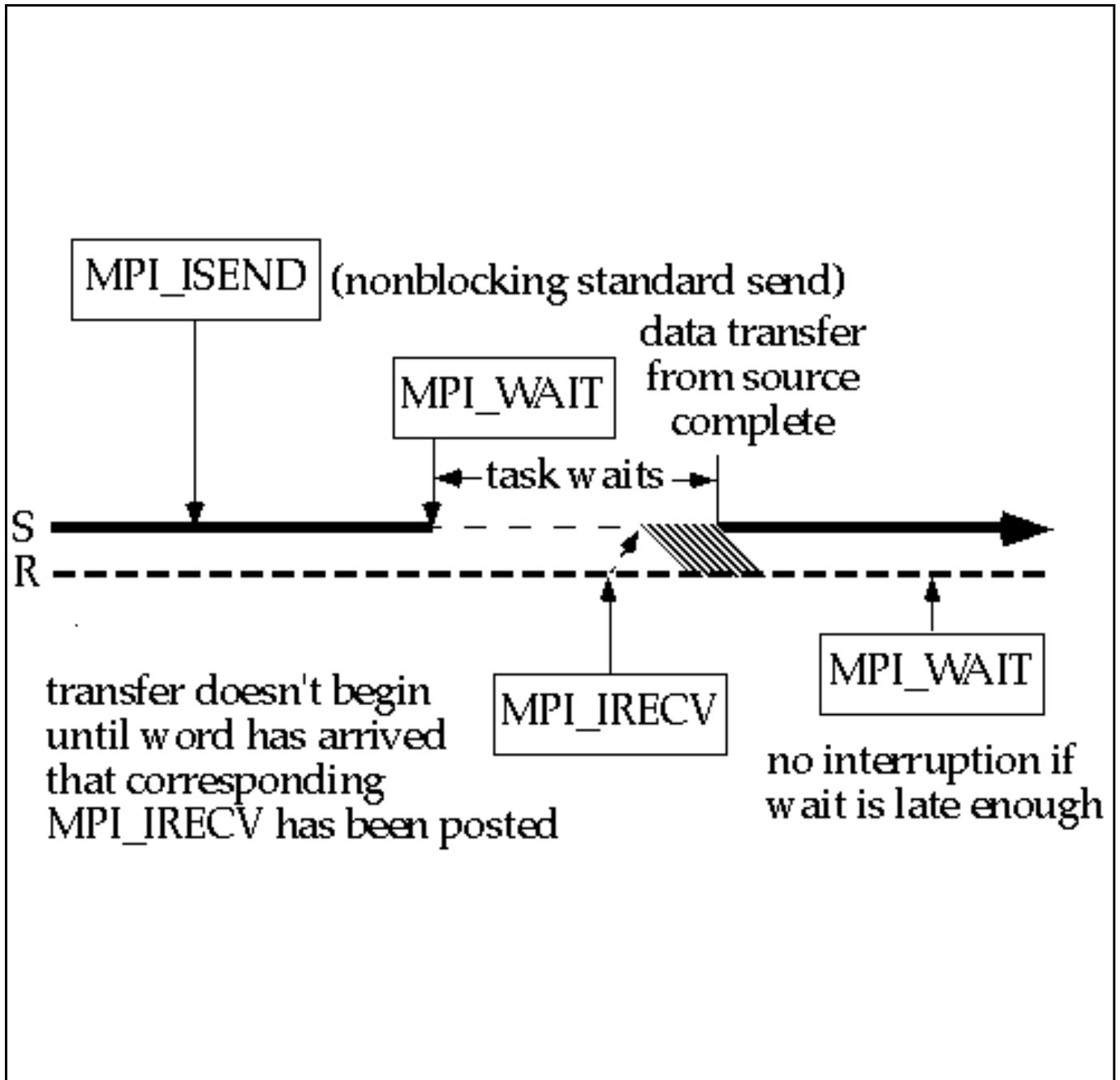
FORTTRAN

callMPI_Irecv(buf,count,datatype,source,tag,comm,request,ierror)

Non-Blocking Standard Send / Non-Blocking Receive Message ≤ 4K



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



9.4-Conclusão

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

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 "BufferedSend".

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

callMPI_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 e libera o espaço para o MPI.

Esta operação bloqueia a execução do programa até todas as mensagens serem 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 uma operação "non-blocking send" foi concluída ou não.

MPI_Type_extent

C

int MPI_Type_extent(MPI_Datatype datatype, int* extent)

FORTTRAN

call MPI_TEST_EXTENT(datatype, extent, ierror)

datatype Tipododado;

extent Variávelinteiracomaextensãoreservada,embytes,
paraotipododado;

ierror Variávelinteiracomostatusdaexecuçãodarotina.

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-blockingStandardSend**" e "**Non-blockingReceive**" (Elimina "deadlock", reduz "synchronizationoverhead" e geralmente possui boa performance);
- "Blocking" será necessário se o usuário necessitar sincronizar processos.
- É mais eficiente utilizar rotinas "blocking", ao invés 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 "systembuffer".
- 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 "synchronizationoverhead" durante a fase 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 coletiva são 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

intMPI_Barrier(MPI_Comm comm)

FORTTRAN

callMPI_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-"DataMovement"

10.2.1-"Broadcast"

C

int **MPI_Bcast**(*void*buffer, intcount, MPI_Datatype datatype, introot, MPI_Commcomm*)

FORTRAN

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

buffer Endereço inicial do dado a ser enviado;

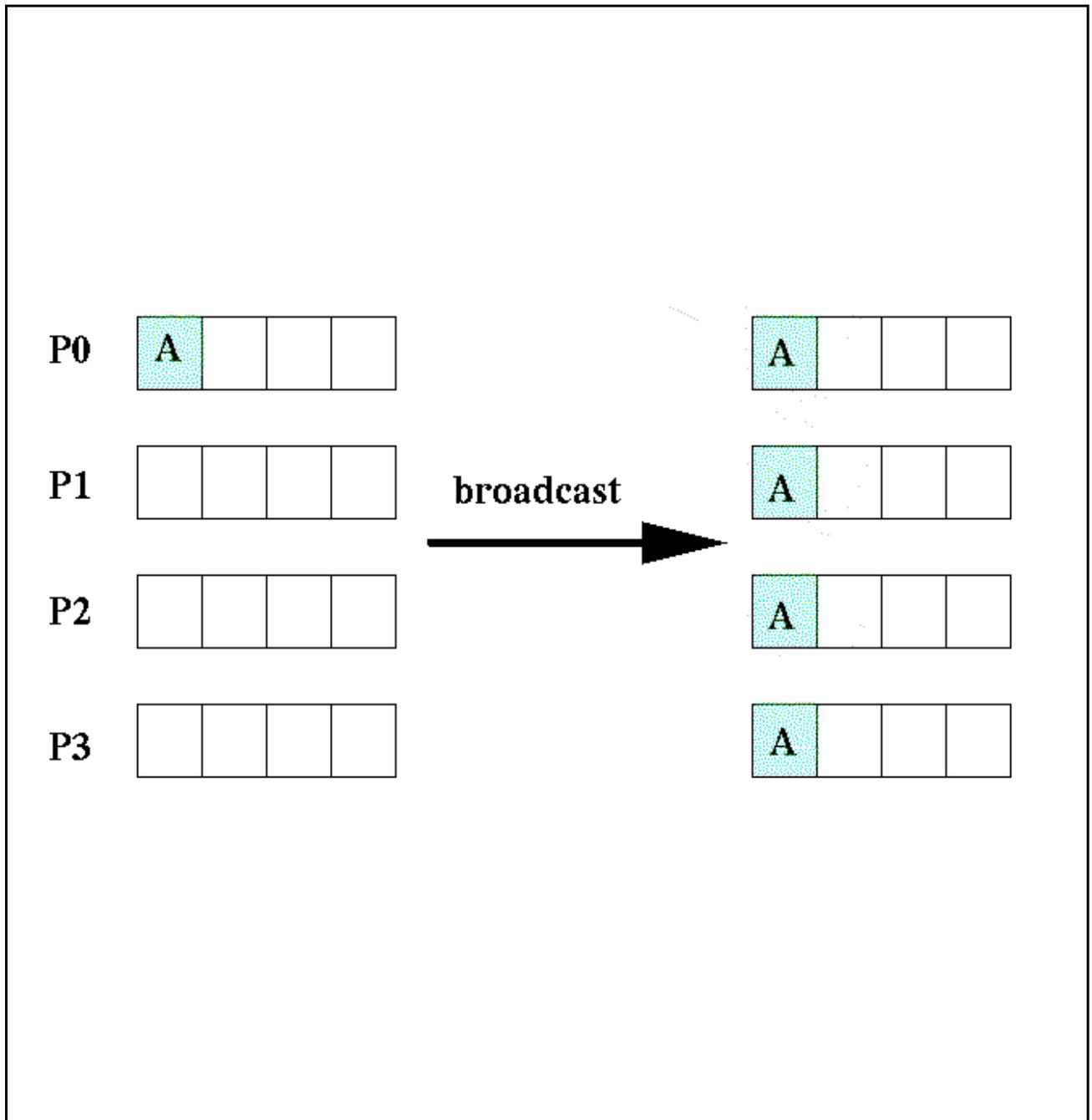
count Inteiro que indica o número de elementos no **buffer**;

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**.

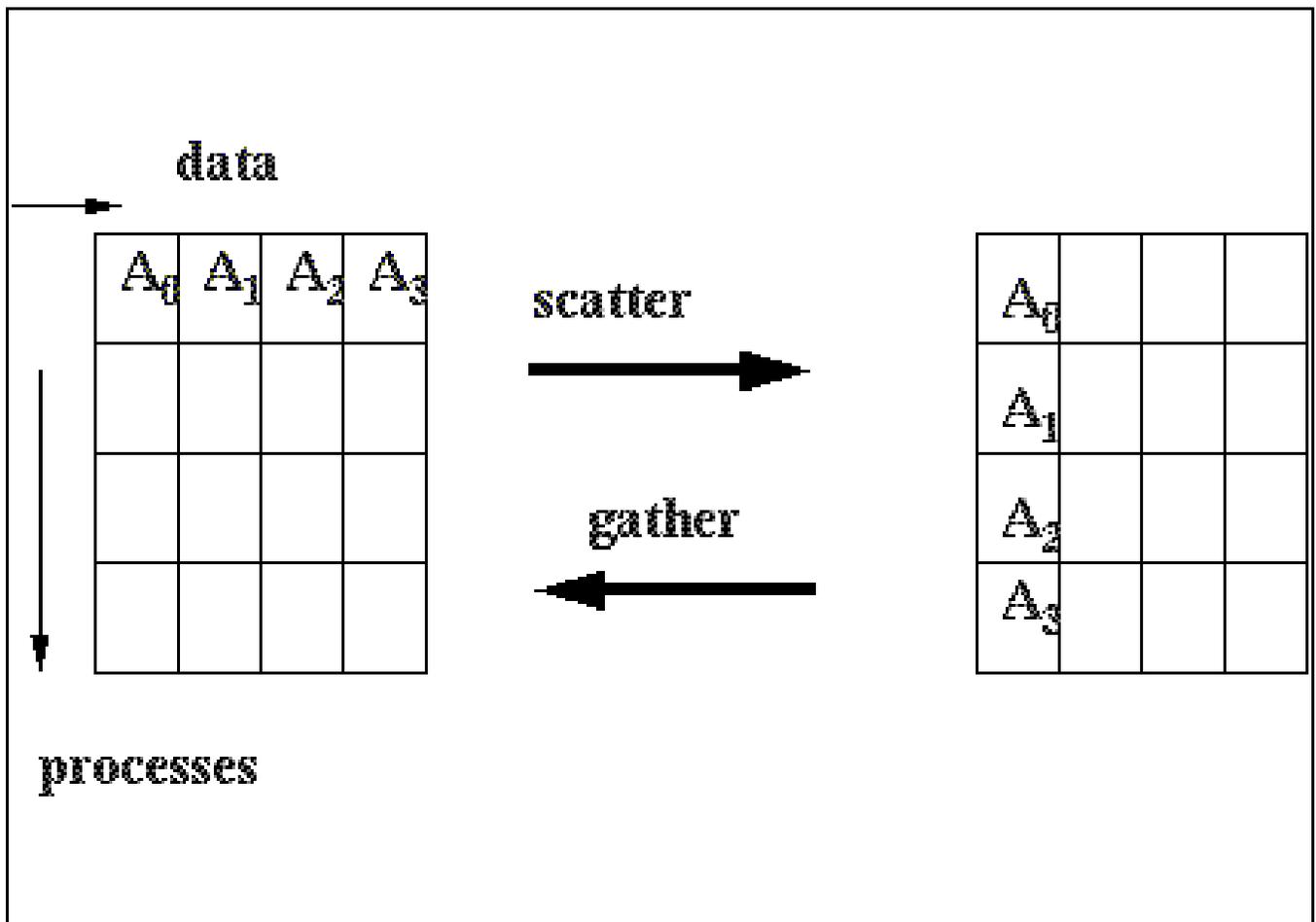
Rotina que permite a um processo enviar dados, de imediato, para todos os processos de um grupo. Todos os processos do grupo, deverão executar um **MPI_Bcast**, com o mesmo **comm** e **root**.



10.2.2-"Gather"e"Scatter"

Seum processonecessitadistribuidadoem n segmentosiguais, ondeo i ésimosegmentoéenviado parao i ésimoprocessonum grupode n processos,utiliza-searotinode **SCATTER**.

Poroutrolado,seumúnico processonecessitacoletar os dados distribuidos em n processos deum grupo.utiliza-se arotinode **GATHER**.



MPI_SCATTER**C**

```
intMPI_Scatter(void*sbuf,intscount,MPI_Datatype stype,void*rbuf,  
              intrcount,MPI_Datatypertype,introot,MPI_Comm comm)
```

FORTRAN

```
callMPI_SCATTER(sbuf,scount,stype,rbuf,rcount,rtype,root,comm,ierr)
```

sbuf Endereço dos dados que serão distribuídos ("sendbuffer");

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**

*intMPI_Gather(void*sbuf,intscount,MPI_Datatype stype,void*rbuf,
intrcount,MPI_Datatype rtype,introot,MPI_Comm com m)*

FORTTRAN

callMPI_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 processado grupo;

rtype Tipo de dado coletado;

root Identificação do processo que irá coletar os dados;

comm Identificação do "communicator".

ierr Variável inteira de retorno com o status da rotina.

Exemplo1

```
DIMENSIONA(25,100),b(100),cpart(25),ctotal(100)
```

```
INTEGERroot
```

```
DATAroot/0/
```

```
DOI=1,25
```

```
  cpart(I)=0.
```

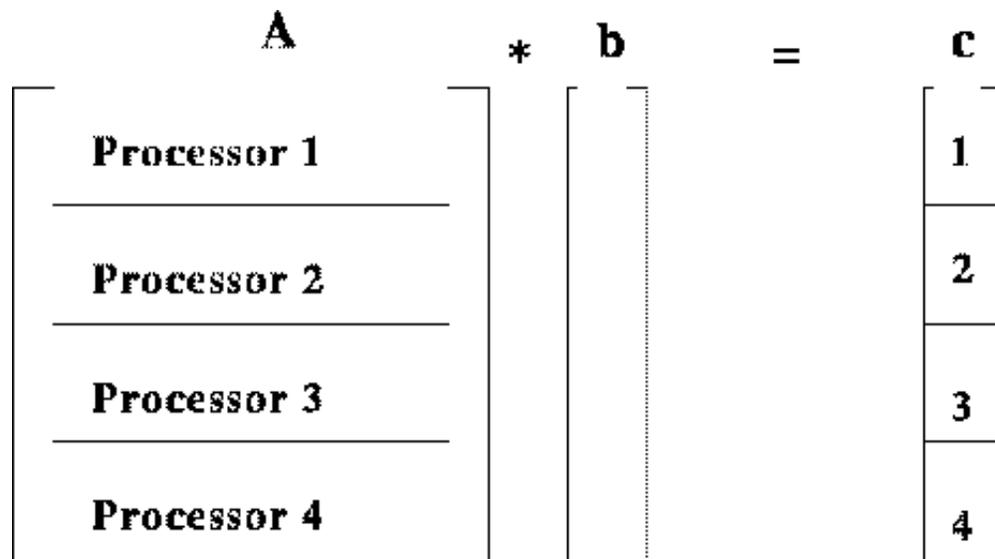
```
  DOK=1,100
```

```
    cpart(I)=cpart(I)+A(I,K)*b(K)
```

```
  ENDDO
```

```
ENDDO
```

```
callMPI_GATHER(cpart,25,MPI_REAL,ctotal,25,MPI_REAL,root,
               MPI_COMM_WORLD,ierr)
```

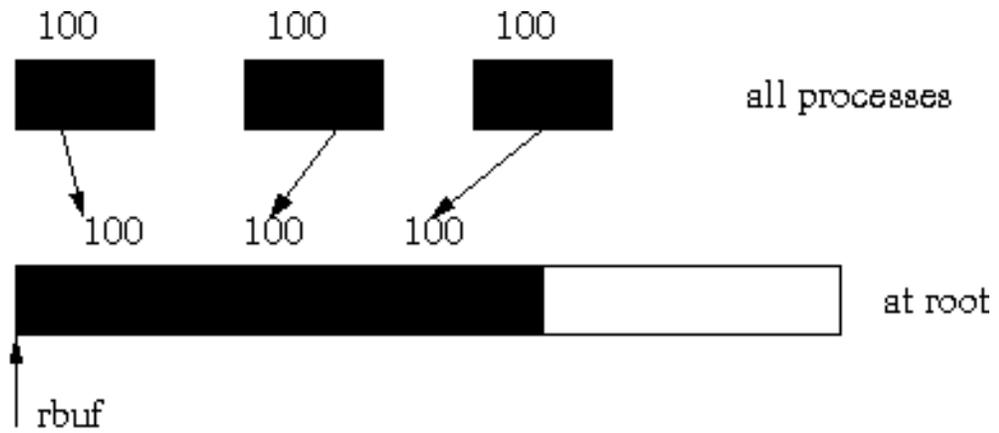


A: Matriz distribuída por linhas;

B: Vetor compartilhado por todos os processos;

C: Vetor atualizado por cada processo independentemente.

Exemplo2

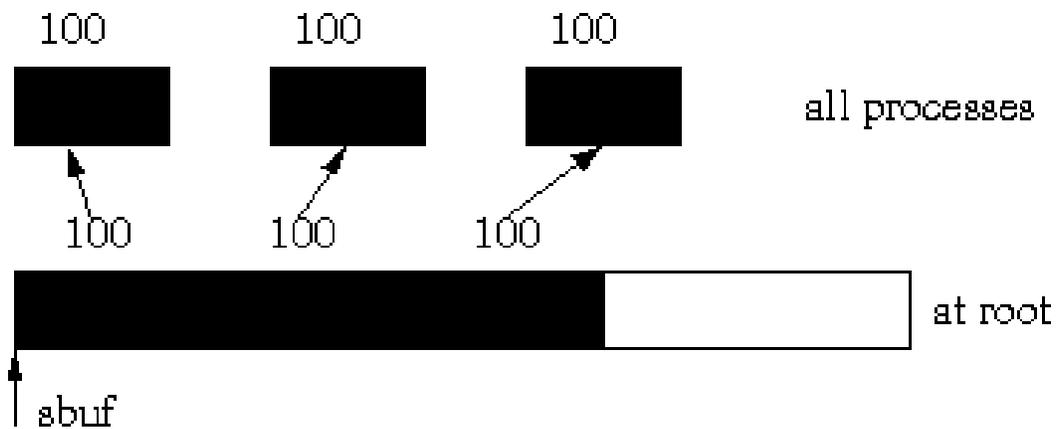


```
reala(100),rbuf(MAX)
```

```
  .      .      .
  .      .      .
  .      .      .
```

```
callmpi_gather(a,100,MPI_REAL,rbuf,100,MPI_REAL,root,  
comm,ierr)
```

Exemplo3



```
realsbuf(MAX),rbuf(100)
```

```
·           ·           ·
·           ·           ·
·           ·           ·
```

```
callmpi_scatter(sbuf,100,MPI_REAL,rbuf,100,MPI_REAL,  
root,comm,ierr)
```

10.2.3-"Allgather"

C

```
intMPI_Allgather(void*sbuf,intscount,MPI_Datatype stype,void*rbuf,intrcount,MPI_Datatype rtype,MPI_Commcomm)
```

FORTRAN

```
callMPI_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 processador do grupo;

rtype Tipo de dado coletado;

comm Identificação do "communicator".

ierr Variável inteira de retorno com o status da rotina.

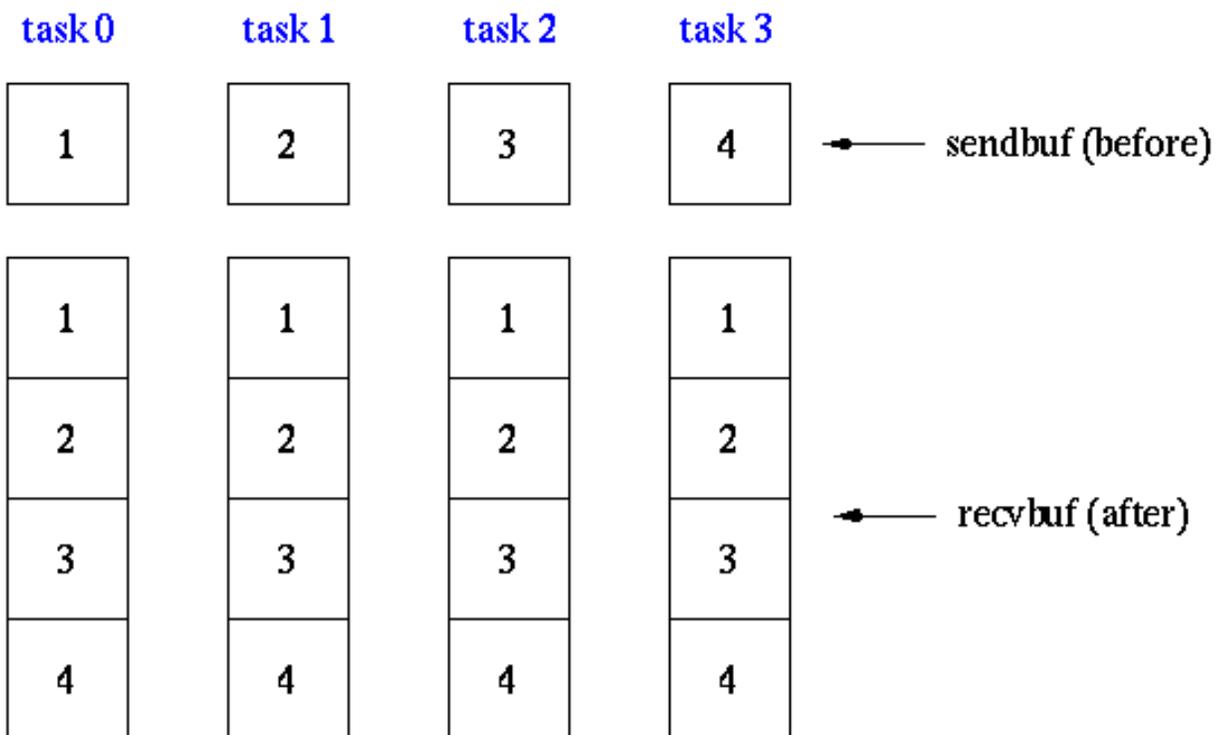
Esta rotina a ser executada faz com que todos os processos coletem os dados de cada processo da aplicação. Seria similar a cada processo efetuarum "broadcast".

MPI_Allgather

```
sendcnt = 1;
```

```
recvcnt = 1;
```

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



10.2.4-"AlltoAll"

C

```
intMPI_Alltoall(void*sbuf,intscount,MPI_Datatype estype,void*rbuf,  
               intrcount,MPI_Datatypertype,MPI_Commcomm)
```

FORTRAN

```
callMPI_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 processador do grupo;

rtype Tipo de dado coletado;

comm Identificação do "communicator".

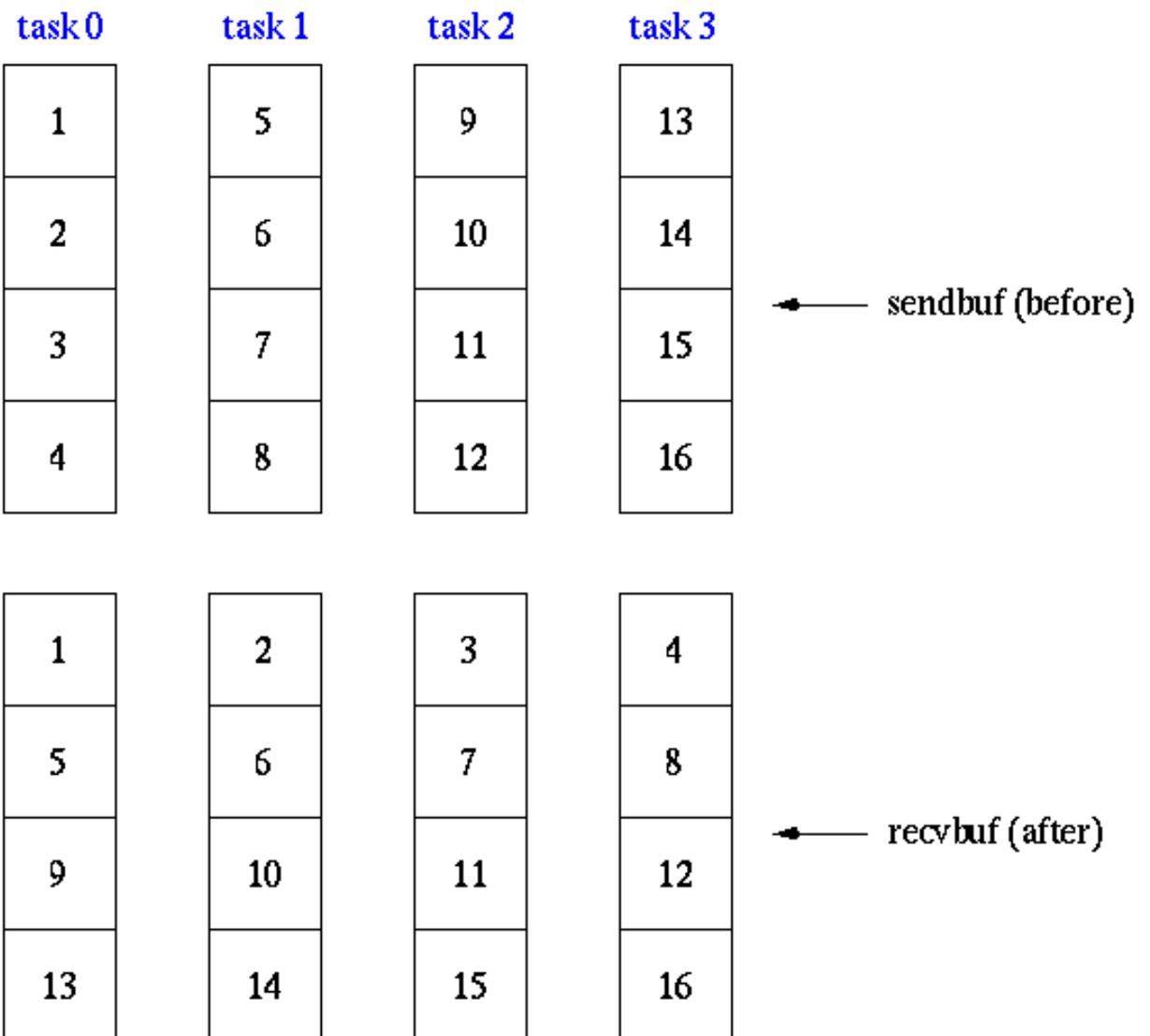
ierr Variável inteira de retorno com o status da rotina.

Esta rotina a 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-RotinasdeComputaçãoGlobal

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 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

MPI_REDUCE

C

```
intMPI_Reduce(void*sbuf,void*rbuf,intcount,MPI_ Datatypestype,  
MPI_Opop,introot,MPI_Commcomm)
```

FORTRAN

```
callMPI_REDUCE(sbuf,rbuf,count,stype,op,root,comm, ierr)
```

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

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

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".

Exemplo1

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)

Exemplo2

Simulação Dinâmica de Florestas

- Cada processo numa simulação dinâmica de florestas, calcula o valor máximo de altura de árvore 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
```

```
·
·   (Gerar relatório com os resultados)
```

```
·
ENDIF
```

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 Tennessee, Knoxville, Tennessee
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 ideia 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 como primeiro exercício do laboratório.

```
%cd./mpi/lab01/ex1
```

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

```
%mpif77hello.f-ohello  
ou  
%mpicchello.c-ohello
```

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

```
%mpirun-np nhello
```

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 em máquina local (máquina logada).

```
%mpirun-np n-nolocalhello
```

Exercício 2

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

```
%cd./mpi/lab01/ex2
```

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

Este programa segue o modelo SPMD (Single Program Multiple Data), ou seja, o mesmo programa é executado em um processo mestre e em 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 ao invés de imprimir a mensagem, devolva-a ao processo mestre. O programa mestre recebe a mensagem devolta 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

CProgramhello.ex1.f

CParallelversionusingMPIcalls

CModifiedfrombasicversionsothatworkerssendback

Camessagetothemaster,whoprintsoutamessage

Cforeachworker

CRLF10/11/95

```

    programhello
    include'mpif.h'
    integerme,nt,mpierr,tag,status(MPI_STATUS_SIZE)
    integerrank
    character(12)message,inmsg
    callMPI_INIT(mpierr)
    callMPI_COMM_SIZE(MPI_COMM_WORLD,nt,mpierr)
    callMPI_COMM_RANK(MPI_COMM_WORLD,me,mpierr)
tag=100
    if(me.eq.0)then
        message='Hello,world'
        doi=1,nt-1
        callMPI_SEND(message,12,MPI_CHARACTER,i,tag,
.MPI_COMM_WORLD,mpierr)
    enddo
    write(6,*)'node',me,':',message

    ===>do...(REGRADODOLOOP)
===>(ROTINAMPI)

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

    ===>(ROTINAMPI)

    endif
    CALLMPI_FINALIZE(mpierr)
end

```

hello.ex1.c

```

/*hello.ex1.c
*ParallelversionusingMPIcalls
*Modifiedfrombasicversionsothatworkerssendback
*amessagetothemaster,whoprintsoutamessage
*foreachworker
*RLF10/11/95
#include<stddef.h>
#include<stdlib.h>
#include"mpi.h"
main(intargc,char**argv)
{
    charmessage[20];
    inti,rank,size,type=99;
    MPI_Statusstatus;
    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...(REGRADODOLOOP)
====>(ROTINAMPI)

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

====>(ROTINAMPI)
    }
    MPI_Finalize();
}

```

Exercício3

1- Caminhe para o diretório como 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 mensagens que estão sendo enviadas ou recebidas.

2- Adicione as rotinas necessárias ao programa em FORTRAN ou em C **substituindo as linhas que possuem as 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

CProgramhello.ex2.f

CParallelversionusingMPIcalls.

CModifiedfrombasicversionsothatworkerssendbackamessagetothe

Cmaster,whoprintsoutamessageforeachworker.Inaddition,the

Cmasternowsendsouttwomessagestoeachworker,withtwodifferent

Ctags,andtheworkerreceivesthemessagesinreverseorder.

CRLF10/11/95

```

    programhello
    include'mpif.h'
    integerme,nt,mpierr,tag,status(MPI_STATUS_SIZE)
    integerrank,tag2
    character(6)message1,message2,inmsg1,inmsg2
    callMPI_INIT(mpierr)
    callMPI_COMM_SIZE(MPI_COMM_WORLD,nt,mpierr)
    callMPI_COMM_RANK(MPI_COMM_WORLD,me,mpierr)
tag=100
tag2=200
    if(me.eq.0)then
        message1='Hello,'
        message2='world'
        doi=1,nt-1
    ===>(ROTINAMPI)
    ===>(ROTINAMPI)
    enddo
    write(6,*)'node',me,':',message1,',',message2
    else
    ===>(ROTINAMPI)
    ===>(ROTINAMPI)
    write(6,*)'node',me,':',inmsg1,',',inmsg2
    endif
    CALLMPI_FINALIZE(mpierr)
end

```

hello.ex2.c

```

/*hello.ex2.c
*ParallelversionusingMPIcalls
*Modifiedfrombasicversionsothatworkerssendback
*amessagetothemaster,whoprintsoutamessage
*foreachworker
*Inaddition,themasternowsendsouttwomessagestoeachworker,with
*twodifferenttags,andeachworkerreceivesthemessages
*inreverseorder.
*RLF10/23/95
*/
#include<stddef.h>
#include<stdlib.h>
#include"mpi.h"
main(intargc,char**argv)
{
    charmessage1[7],message2[7];
    inti,rank,size,type=99;
    inttype2=100;
    MPI_Statusstatus;
    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++){
            ===>(ROTINAMPI)
            ===>(ROTINAMPI)
        }
        printf("node%d:%.7s%.7s\n",rank,message1,message2);
    }
    else{
        ===>(ROTINAMPI)
        ===>(ROTINAMPI)
        printf("node%d:%.7s%.7s\n",rank,message1,message2);
    }
    MPI_Finalize();
}

```

Exercício4

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

```
%cd./mpi/lab01/ex4
```

Neste exercício, o processo mestre inicializa um vetor distribuído em 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*****

CFILE:mpi.ex1.f

CDESCRIPTION:

CInthissimpleexample,themastertaskinitiatesnumtasks-1numberof
Cworkertasks.Itthendistributesanequalportionofanarraytoeach
Cworkertask.Eachworkertaskreceivesitsportionofthearray,and
Cperformsasimplevalueassignmenttoeachofitselements.Thevalue
Cassignedtoeachelementissimplythatelement'sindexinthearray+1.
CEachworkertaskthensendsitsportionofthearraybacktothelmaster
Ctask.Asthelmasterreceivesbackeachportionofthearray,selected
Celementsaredisplayed.

CAUTHOR:BlaiseBarney

CLASTREVISED:6/10/93

CLASTREVISED:1/10/94changedAPItoMPLStacyPendell

C CONVERTEDTOMPI:11/12/94byXiannengShen

C*****

programexample1_master

====>**INCLUDEMPI**

integerstatus(MPI_STATUS_SIZE)

integerARRAYSIZE

parameter(ARRAYSIZE=60000)

parameter(MASTER=0)

integernumtask,numworkers,taskid,dest,index,i,

&arraymsg,indexmsg,source,chunksize,

&int4,real4

real*4data(ARRAYSIZE),result(ARRAYSIZE)

C*****initializations*****

CFindouthowmanytasksareinthispartitionandwhatmytaskidis.Then

Cdefinethenumberofworkertasksandthearraypartitionsizesaschunksize.

CNote:Forthisexample,theMP_PROCSenvironmentvariablesouldbeset

Ctoanodddnumber...toinsureevendistributionofthearraytonumtasks-1

Cworkertasks.

C

*

```

=====>ROTINAMPIDEINICIALIZAÇÃO(ierr)
=====>ROTINAMPIDEIDENTIFICACAORLD(taskid,ierr)
=====>ROTINAMPIPARADETERMINARONUMERODEPROCESSOS
      (numtasks,ierr)

write(*,*)'taskid=',taskid
numworkers=numtasks-1
chunksize=(ARRAYSIZE/numworkers)
arraymsg=1
indexmsg=2
int4=4
real4=4
C*****mastertask*****
if(taskid.eq.MASTER)then
print*, '*****StartingMPIExample1*****'
CInitializethearray
do20i=1,ARRAYSIZE
data(i)=0.0
20continue
CSendeachworkertaskitsportionofthearray
index=1
do30dest=1,numworkers
write(*,*)'Sendingtoworkertask',dest

=====>ROTINAMPIDEENVIODEDADOS(index,dest,ierr)
=====>ROTINAMPIDEENVIODEDADOS(data(index),chunksize,dest,ierr)

index=index+chunksize
30continue
CNowwaittoreceivebacktheresultsfromeachworkertaskandprint
Cafewsamplevalues
do40i=1,numworkers
source=i

=====>ROTINAMPIPARARECEBERDADOS(index,source,status,ierr)
=====>ROTINAMPIPARARECEBERDADOS
      (result(index),chunksize,source,status,ierr)

```

```

print*,'-----'
print*,'MASTER:Sampleresultsfromworkertask',source
print*,'result[' ,index,']=',result(index)
print*,'result[' ,index+100,']=',result(index+100)
print*,'result[' ,index+1000,']=',result(index+1000)
print*,"
40continue
print*,'MASTER:AllDone!'
endif
C*****workertask*****
if(taskid.gt.MASTER)then
CReceivemyportionofarrayfromthemastertask*/

====>ROTINAMPIPARARECEBERDADOS(index,MASTER,status,ierr)
====>ROTINAMPIPARARECEBERDADOS
      (result(index),chunksize,MASTER,status,ierr)

CDoasimplevalueassignmenttoeachofmyarrayelements
do50i=index,index+chunksize
result(i)=i+1
50continue
CSendmyresultsbacktothemaster

====>ROTINAMPIDEENVIODEDADOS(index,MASTER,ierr)
====>ROTINAMPIDEENVIODEDADOS
      (result(index),chunksize,MASTER,ierr)

endif

====>ROTINAMPIDEFINALIZAÇÃO(ierr)

end

```

mpi.ex1.c

```

/*****
**
**
*FILE:mpi.ex1.c
*DESCRIPTION:
*Inthissimpleexample,themastertaskinitiatesnumtasks-1numberof
*workertasks.Itthendistributesanequalportionofanarraytoeach
*workertask.Eachworkertaskreceivesitsportionofthearray,and
*performsasimplevalueassignmenttoeachofitselements.Thevalue
*assignedtoeachelementissimplythatelement'sindexinthearray+1.
*Eachworkertaskthensendsitsportionofthearraybacktothemaster
*task.Asthemasterreceivesbackeachportionofthearray,selected
*elementsaredisplayed.
*AUTHOR:BlaiseBarney
*LASTREVISED:09/14/93forlatestAPIchangesBlaiseBarney
*LASTREVISED:01/10/94changedAPItoMPLStacyPendell
*CONVERTEDTOMPI:11/12/94byXiannengShen
*****/
#include<stdio.h>

====>INCLUDEMPI

#defineARRAYSIZE 60000
#defineMASTER      0      /*taskidoffirstprocess*/

MPI_Statusstatus;
main(intargc,char**argv)
{
int      numtasks,          /*totalnumberofMPIprocessinpartitiion*/
        numworkers,       /*numberofworkertasks*/
        taskid,           /*taskidentifier*/
        dest,             /*destinationtaskidtosendmessage*/
        index,            /*indexintothearray*/
        i,                 /*loopvariable*/
        arraymsg=1,       /*settingamessagetype*/
        indexmsg=2,       /*settingamessagetype*/
        source,           /*origintaskidofmessage*/
        chunksize;        /*forpartitioningthearray*/
float    data[ARRAYSIZE], /*theintialarray*/
        result[ARRAYSIZE]; /*forholdingresultsofarrayoperations*/

```

```

/*****initializations*****/
*Findouthowmanytasksareinthispartitionandwhatmytaskidis.Then
*definethenumberofworkertasksandthearraypartitionsizesaschunksize.
*Note:Forthisexample,theMP_PROCSenvironmentvariableshouldbeset
*toanodddnumber...toinsureevendistributionofthearraytonumtasks-1
*workertasks.
*****/
**/

```

```

====>ROTINAMPIDEINICIALIZAÇÃO(&argc,&argv)
====>ROTINAMPIDEIDENTIFICACAO(&taskid)
====>ROTINAMPIPARADETERMINARONUMERODEPROCESSOS
      (&numtasks)

```

```

numworkers=numtasks-1;
chunksize=(ARRAYSIZE/numworkers);
/*****mastertask*****/
if(taskid==MASTER){
printf("\n*****StartingMPIExample1*****\n");
printf("MASTER:numberofworkertaskswillbe=%d\n",numworkers);
fflush(stdout);

```

```

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

```

```

/*Sendeachworkertaskitsportionofthearray*/
for(dest=1;dest<=numworkers;dest++){
printf("Sendingtoworkertask=%d\n",dest);
fflush(stdout);

```

```

====>ROTINAMPIDEENVIODEDADOS(&index,dest)
====>ROTINAMPIDEENVIODEDADOS(&data[index],chunksize,dest      )

```

```

index=index+chunksize;
}
/*Nowwaittoreceivebacktheresultsfromeachworkertaskandprint*/
/*afewsamplevalues*/
for(i=1;i<=numworkers;i++){
source=i;

```

```

=====>ROTINAMPIPARARECEBERDADOS(&index,source,&status)
=====>ROTINAMPIPARARECEBERDADOS
      (&result[index],chunksize,source,&status)

```

```

printf("-----\n");
printf("MASTER:Sampleresultsfromworkertask=%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:AllDone!\n");
}

```

```

/*****workertask*****/
if(taskid>MASTER){
/*Receivemyportionofarrayfromthemastertask*/
source=MASTER;

```

```

=====>ROTINAMPIPARARECEBERDADOS(&index,source,&status)
=====>ROTINAMPIPARARECEBERDADOS
      (&result[index],chunksize,source,&status)

```

```

/*Doasimplevalueassignmenttoeachofmyarrayelements*/
for(i=index;i<index+chunksize;i++)
result[i]=i+1;

```

```

/*Sendmyresultsbacktothemastertask*/

```

```

=====>ROTINAMPIDEENVIODEDADOS(&index,MASTER)
=====>ROTINAMPIDEENVIODEDADOS
      (&result[index],chunksize,MASTER)
}

```

```

=====>ROTINAMPIDEFINALIZAÇÃO
}

```

Exercício5

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

```
%cd./mpi/lab01/ex5
```

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

Aidei 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 como 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 em um "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 tamanhos de mensagens;
- Em particular compare os resultados da transmissão de dados de 1024 floats e 1025 floats.

Exercício2

1- Caminhe para o diretório como 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 executar por completo. **O que será necessário fazer???**

Exercício3

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

```
%cd./mpi/lab02/ex3
```

Este programa demonstra 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-cnew_sleep.c
```

- 3- Compile o programa principal:

```
%mpxlbrecv.fnew_sleep.o-obrecv
```

ou

```
%mpccbrecv.cnew_sleep.o-obrecv
```

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

Exercício4

1- Caminhe para o diretório como 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 da mensagem recebida, para garantir o 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 o do exercício anterior.

3º LABORATÓRIO

Comunicação Coletiva

Exercício 1

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

```
%cd./mpi/lab03/ex1
```

Neste exercício:

- o processo mestre solicita a entrada de um número, que será automaticamente 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;
- será calculado 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.

ex1.f

```

!-----
!ThisprogramteststheuseofMPICollectiveCommunications
!subroutines.Thestructureoftheprogramis:
!
!--Masternodequeriesforrandomnumberseed
!--Theseedissenttoallnodes(Labproject)
!--Eachnodecalculatesonerandomnumberbasedontheseed
!andtherank
!--Thenodewithhighestrankcalculatesthemeanvalue
!oftherandomnumbers(Labproject)
!--4morerandomnumbersaregeneratedbyeachnode
!--Themaximumvalueandthestandarddeviationofall
!generatedrandomnumbersarecalculated,andthe
!resultsaremadeavailabletoallnodes(Labproject)
!
!AlsoprovidedisaserviceroutineGetStats(rnum,N,data),where
!
!rnum:arrayofrandomnumbers(INPUT)
!N:numberofelementsinrnum(INPUT)
!outd:arrayofsize2containingthemaximumvalueand
!standarddeviation(OUTPUT)
!-----
programCollCom
  include'mpif.h'
  integerstatus(MPI_STATUS_SIZE),ierr
  integer numtasks,taskid
  real*4randnum(5),sum,meanv,rnum(100),rval(2)
  real*8seed
  callMPI_INIT(ierr)
  callMPI_COMM_RANK(MPI_COMM_WORLD,taskid,ierr)
  callMPI_COMM_SIZE(MPI_COMM_WORLD,numtasks,ierr)
  doi=1,100
  rnum(i)=0.0
  enddo
  if(taskid.eq.0)then!Getrandomnumberseed
    write(*,'(a)')Enterrandomnumberseedinf12.5format'
    read(*,'(f12.5)')seed
  endif
!=====

```

```
!Project:Sendseedfromtask0toallnodes.
```

```
!=====
```

```
=====>ROTINAMPI
```

```
write(*,'(a,i3,a,f12.5)')Task',taskid,
&'afterbroadcast;seed=',seed
callsrand(seed+REAL(taskid))
randnum(1)=100*rand()!Eachnodegeneratesarandomnumber
```

```
!=====
```

```
!Project:Havethenodewithhighestrankcalculatethe
!meanvalueoftherandomnumbersandstoreresult
!inthevariable"meanv".
```

```
!=====
```

```
=====>ROTINAMPI
```

```
write(*,'(a,i3,a,f8.3,a,f8.3,a,f8.3)')
&'Task',taskid,'aftermeanvalue;random(1)=',
&randnum(1),'sum=',sum,'mean=',meanv
!Highesttaskwritesoutmeanvalue
if(taskid.eq.(numtasks-1))then
open(unit=10,file='f.data',status='UNKNOWN')
write(10,'(a,f12.5,a,f10.3)')Forseed=',seed,
&'meanvalue=',meanv
endif
!Eachnodegenerates4morerandomnumbers
doii=2,5
randnum(ii)=100*rand()
enddo
```

```
!=====
```

```
!Project:Calculatethemaximumvalueandstandarddeviationof
!allrandomnumbersgenerated,andmakeresultsknown
!toallnodes.
!Method1:
!Method2:
```

```
!=====
```

!-----Method1-----

=====>**ROTINAMPI**

```
if(taskid.eq.0)then
callGetStats(rnum,numtasks*5,rval)
endif
```

=====>**ROTINAMPI**

```
write(*,'(a,i3,a,5f8.3)')'Task',taskid,
&'afterMethod1,rnum(1:5)=',(rnum(I),I=1,5)
if(taskid.eq.(numtasks-1))
&write(10,'(a,2f10.3)')'(Max,S.D.)=',rval
```

!-----Method2-----

=====>**ROTINAMPI**

```
callGetStats(rnum,numtasks*5,rval)
!
write(*,'(a,i3,a,5f8.3)')'Task',taskid,
&'afterMethod2,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)
endif
```

callMPI_FINALIZE(ierr)

end

```

!-----
!TheserviceroutineGetStats(rnum,N,data),where
!
!rnum:arrayofrandomnumbers(INPUT)
!N:numberofelementsinrnum(INPUT)
!outd:arrayofsize2containingthemaximumvalueand
!standarddeviation(OUTPUT)
!
!-----
subroutineGetStats(rnum,N,outd)
real*4rnum(N),outd(2),sum,meanv

sum=0.
outd(1)=0.

doii=1,N
    sum=sum+rnum(ii)
    if(rnum(ii).gt.outd(1))outd(1)=rnum(ii)
enddo

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

doii=1,N
    sdev=sdev+(rnum(ii)-meanv)**2
enddo
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 (Labproject)
--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 (Labproject)
--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 (Labproject)

```

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)/*Getrandomnumberseed*/
{
printf("Enterrandomnumbraspositiveinteger\n");
scanf("%u",&seed);
}

```

```

/*

```

```

=====
Project:Sendseedfromtask0toallnodes.
=====
*/

```

```

*/

```

```

=====>ROTINAMPI

```

```

printf("\nTask%dafterbroadcast;seed=%u",taskid,seed);
srand(seed+taskid);

```

```

randnum[0]=100.*(float)rand()/(float)RAND_MAX;/*Getarandomnumber*/

```

```

/*

```

```

=====
Project:Havethenodewithhighestrankcalculatethe
meanvalueoftherandomnumbersandstoreresult
inthevariable"meanv".
=====
*/

```

```

*/

```

```

=====>ROTINAMPI

```

```

meanv=sum/(float)numtasks;

```

```

printf("\nTask%daftermeanvalue;random[1]=",taskid);
printf("%8.3fsum=%8.3fmean=%8.3f",randnum[1],sum,meanv);

```

```

/*Highesttaskwritesoutmeanvalue*/

```

```

if(taskid==(numtasks-1))

```

```

fprintf(fp,"Forseed=%dmeanvalue=%10.3f\n",seed,meanv);

```

```

/*Generate4morerandomnumbers*/
for(ii=1;ii<5;ii++)
randnum[ii]=100.*(float)rand()/(float)RAND_MAX;
/*
=====
Project:Calculatethemaximumvalueandstandarddeviationof
allrandomnumbersgenerated,andmakesresultsknown
toallnodes.
Method1:
Method2:
=====
*/
/*-----Method1-----*/

=====>ROTINAMPI

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

=====>ROTINAMPI

printf("\nTask%dafterMethod1,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]);
/*-----Method2-----*/

=====>ROTINAMPI

GetStats(rnum,5*numtasks,rval);
printf("\nTask%dafterMethod2,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();
}

```

```

/*-----
ServiceroutineGetStats(rnum,N,data),where

rnum:arrayofrandomnumbers(INPUT)
N:numberofelementsinrnum(INPUT)
outd:arrayofsize2containingthefmaximumvalueand
standarddeviation(OUTPUT)

-----*/

```

```

intGetStats(float*rnum,intN,float*outd)
{
floatsum,meanv,sdev;
inti;

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);
}

```

Exercício2

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

%cd./mpi/lab03/ex2

Aidei do programa e demonstre a execução da rotina **MPI_SCATTER**.

Adicione os seus parâmetros para que ela funcione adequadamente.

2- Compile o programa e execute-o.

scatter.f

```

programscatter
  include'mpif.h'

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

  CFortranstoresthisarrayincolumnmajororder,sothe
  Cscatterwillactuallyscattercolumns,notrows.
  datasendbuf/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/

  callMPI_INIT(ierr)
  callMPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
  callMPI_COMM_SIZE(MPI_COMM_WORLD,numtasks,ierr)

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

    callMPI_SCATTER(.....)

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

  callMPI_FINALIZE(ierr)

end

```

scatter.c

```

#include"mpi.h"
#include<stdio.h>
#defineSIZE4

intmain(argc,argv)
intargc;
char*argv[];{
intnumtasks,rank,sendcount,recvcount,source;
floatsendbuf[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}};
floatrecvbuf[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=%dResults:%f%f%f%f\n",rank,recvbuf[0],
recvbuf[1],recvbuf[2],recvbuf[3]);
}
else
printf("Mustspecify%dprocessors.Terminating.\n",SIZE);

MPI_Finalize();
}

```

Exercício3

1- Caminhe para o diretório como 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 ideia é demonstrar 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.

mpi_prime.f

CFILE:mpi_prime.f

COTHERFILES:make.mpi_prime.f

CDESCRIPTION:

CGeneratesprimenumbers.Alltasksdistributetheworkevenly,taking
 Ceverynthnumber,wherenisthestridecomputedas:(rank*2)+1
 Csothatevennumbersareautomaticallyskipped.Themethodofusing
 Cstrideispreferredovercontiguousblocksofnumbers,sincenumbers
 Cinthehigherrangerequiremoreworktocomputeandmayresultin
 Cloadimbalance.Thisprogramdemonstratesembarrassingparallelism.
 CCollectivecommunicationscallsareusedtoreducetheonlytwodata
 Celementsrequiringcommunications:thenumberofprimesfoundand
 Cthelargestprime.

CAUTHOR:BlaiseBarney11/25/95-adaptedfromversioncontributedby
 CRichardNg&WongSzeCheongduringMHPCCSingaporeWorkshop(8/22/95).

CLASTREVISED:

C-----

CExplanationofconstantsandvariables
 CLIMIT=Increasethistofindmoreprimes
 CFIRST=Rankoffirsttask
 Cntasks=totalnumberoftasksinpartitiion
 Crank=taskidentifier
 Cn=loopvariable
 Cpc=primecounter
 Cpcsum=numberofprimesfoundbyalltasks
 Cfoundone=mostrecentprimefound
 Cmaxprime=largestprimefound
 Cmystart=wheretostartcalculating
 Cstride=calculateeverynthnumber

C-----

programprime

include'mpif.h'

integerLIMIT,FIRST

parameter(LIMIT=2500000)

parameter(FIRST=0)

integerntasks,rank,ierr,n,pc,pcsum,foundone,maxprime,

&mystart,stride

doubleprecisionstart_time,end_time

logicalresult

callMPI_INIT(ierr)

```

callMPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
callMPI_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-thisexerciserequiresanevennumberof'
print*,'processorsevenlydivisibleinto',LIMIT','
print*,'Try4or8.'
    callMPI_FINALIZE(ierr)
stop
endif
CInitializations:mystartmustbeoddnumber.strideismultiplied
Cby2toskipoverevennumbers.
start_time=MPI_WTIME()
mystart=(rank*2)+1
stride=ntasks*2
pc=0
foundone=0
C-----taskwithrank0doesthispart-----
if(rank.eq.FIRST)then
print*,'Using',ntasks,'taskstoscan',LIMIT,'numbers'
CAssumefirstfourprimesarecountedhere
pc=4
don=mystart,LIMIT,stride
callisprime(n,result)
if(result.eqv..true.)then
pc=pc+1
foundone=n
C*****Optional:printeachprimeasitisfound
Cprint*,foundone
C*****
endif
enddo
callMPI_Reduce(.....,1,MPI_INTEGER,.....,MPI_COMM_WORLD,ierr)
callMPI_Reduce(.....,1,MPI_INTEGER,.....,MPI_COMM_WORLD,ierr)
end_time=MPI_WTIME()
print*,'Done.Largestprimeis',maxprime,'Totalprimes',pcsum
print*,'Wallclocktimeelapsed:',end_time-start_time
endif
C-----allothertasksdothispart-----
if(rank.gt.FIRST)then
don=mystart,LIMIT,stride
callisprime(n,result)

```

```

if(result.eqv..true.)then
pc=pc+1
foundone=n
C*****Optional:printeachprimeasitisfound
Cprint*,foundone
C*****
endif
enddo
callMPI_Reduce(...,1,MPI_INTEGER,...,MPI_COMM_WORLD,ierr)
callMPI_Reduce(...,1,MPI_INTEGER,...,MPI_COMM_WORLD,ierr)
endif
    callMPI_FINALIZE(ierr)
end
subroutineisprime(n,result)
integern
logicalresult
integeri,squareroot
real*4realn
if(n.gt.10)then
realn=real(n)
squareroot=int(sqrt(realn))
doi=3,squareroot,2
if(mod(n,i).eq.0)then
result=.false.
return
endif
enddo
result=.true.
return
CAssumefirstfourprimesarecountedelsewhere.Forgeteverythingelse
else
result=.false.
return
endif
end

```

mpi_prime.c

```

/*****
**
**
*FILE:mpi_prime.c
*OTHERFILES:make.mpi_prime.c
*DESCRIPTION:
*Generates primen numbers. All tasks distributethework evenly, taking
*every nth number, wherenisthestride computed as:(rank*2)+1
*sothateven numbers are automatically skipped. The method of using
*stride is preferred over contiguous blocks of numbers, since numbers
*in the higherrange 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
*thelargest prime.
*AUTHOR:Blaise Barney 11/25/95-adapted from version contributed by
*Richard Ng & Wong Sze Cheong during MHPCCS 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 */
intisprime(int n){
inti,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[];{
intntasks,/*totalnumberoftasksinpartition*/
rank,/*taskidentifier*/
n,/*loopvariable*/
pc,/*primecounter*/
pcsum,/*numberofprimesfoundbyalltasks*/
foundone,/*mostrecentprimefound*/
maxprime,/*largestprimefound*/
mystart,/*wheretostartcalculating*/
stride;/*calculateeverynthnumber*/

doublestart_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-thisexerciserequiresanevennumberofprocessors\n");
printf("evenlydivisibleinto%d.Try4or8.\n",LIMIT);
MPI_Finalize();
exit(0);
}

start_time=MPI_Wtime(); /*Initializestarttime*/
mystart=(rank*2)+1; /*Findmystartingpoint-mustbeodddnumber*/
stride=ntasks*2; /*Determinestride,skippingevennumbers*/
pc=0; /*Initializeprimecounter*/
foundone=0; /*Initialize*/
/*****taskwithrank0doesthispart*****/
if(rank==FIRST){
printf("Using%dtaskstoscan%dnumbers\n",ntasks,LIMIT);
pc=4; /*Assumefirstfourprimesarecountedhere*/
for(n=mystart;n<=LIMIT;n=n+stride){
if(isprime(n)){
pc++;
foundone=n;
/*****Optional:printeachprimeasitisfound
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.Largestprimeis%dTotalprimes%d\n",maxprime,pcsum);
printf("Wallclocktimeelapsed:%.2lf\n",end_time-start_time);
}
/*****allothertasksdothispart*****/
if(rank>FIRST){
for(n=mystart;n<=LIMIT;n=n+stride){
if(isprime(n)){
pc++;
foundone=n;
/****Optional:printeachprimeasitisfound
printf("%d\n",foundone);
*****/
}
}

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

}

MPI_Finalize();
}

```

Exercício4

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

```
%cd./mpi/lab03/ex4
```

Este exercício possui duas soluções.

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

- Compile o programa:

```
%mpxfmpi_pi_send.fdboard.f-ompi_pi_send  
ou  
%mpccmpi_pi_send.cdboard.c-ompi_pi_send
```

- Execute o programa:

```
%poe-procs4mpi_pi_send
```

Análise o programa, com relação à 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:

```
%mpxfmpi_pi_opt.fdboard.f-ompi_pi_opt  
ou  
%mpccmpi_pi_opt.cdboard.c-ompi_pi_opt
```

- Execute o programa:

```
%poe-procs4mpi_pi_opt
```

mpi_pi_opt.f

```

C-----
CMPIpiCalculationExample-FortranVersion
CCollectiveCommunicationexamples
CFILE:mpi_pi_opt.f
COTHERFILES:dboard.f
CDESCRIPTION:MPIicalculationexampleprogram.Fortranversion.
CThisprogramcalculatespiusinga"dartboard"algorithm.See
CFoxetal.(1988)SolvingProblemsonConcurrentProcessors,vol.1
Cpage207.Allprocessescontributetothecalculaton,withthe
Cmasteraveragingthevaluesforpi.
C
CSPMDVersion:Conditionalstatementscheckiftheprocessisthe
Cmasteroraworker.
C
CThisversionusesoneMPIroutinetocollectresults
C
CAUTHOR:RoslynLeibensperger.ConverteditoMPI:GeorgeL.Gusciora(1/23/95)
CLASTREVISED:12/14/95BlaiseBarney
C-----
CExplanationofconstantsandvariablesusedinthisprogram:
CDARTS=numberofthrowsatdartboard
CROUNDS=numberoftimes"DARTS"isiterated
CMASTER=taskIDofmastertask
Ctaskid=taskIDofcurrenttask
Cnumtasks=numberoftasks
Chomepi=valueofpicalculatedbycurrenttask
Cpisum=sumoftasks'pivalues
Cpi =averageofpiforthisiteration
Cavepi=averagepivalueforalliterations
Cseednum=seednumber-basedontaskid
C-----
programpi_reduce
include'mpif.h'
integerDARTS,ROUNDS,MASTER
parameter(DARTS=5000)
parameter(ROUNDS=10)
parameter(MASTER=0)
integer taskid,numtasks,i,status(MPI_STATUS_SIZE)
real*4seednum

```

```

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(*,*) 'taskID=', taskid
C Use the taskid 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 empiric 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

```

=====> ROTINAMPI

```

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

```

/*-----
*MPIpiCalculationExample-CVersion
*CollectiveCommunicationexample
*FILE:mpi_pi_opt.c
*OTHERFILES:dboard.c
*DESCRIPTION:MPIpicalculationexampleprogram.CVersion.
*Thisprogramcalculatespiusinga"dartboard"algorithm.See
*Foxetal.(1988)SolvingProblemsonConcurrentProcessors,vol.1
*page207.Allprocessescontribute tothecalculatoin,withthe
*masteraveragingthevaluesforpi.
*SPMDversion:Conditionalstatementscheckifthe process
*isthemasteroraworker.
*ThisversionusesoneMPIroutinetocollectresults
*AUTHOR:RoslynLeibensperger.ConvertedittoMPI:GeorgeL.Gusciora
*LASTREVISED:06/07/96BlaiseBarney
*-----*/

#include"mpi.h"
#include<stdlib.h>
#include<stdio.h>
voidsrandom(unsignedseed);
doubledboard(intdarts);
#defineDARTS5000/*numberofthrowsatdartboard*/
#defineROUNDS10/*numberoftimes"darts"isiterated*/
#defineMASTER0/*taskIDofmastertask*/
intmain(argc,argv)
intargc;
char*argv[];
{
double homepi,/*valueofpicalculatedbycurrenttask*/
    pisum,/*sumoftasks'pivalues*/
    pi, /*averageofpiafter"darts"isthrown*/
    avepi; /*averagepivalueforalliterations*/
int taskid,/*taskID-alsousedasseednumber*/
    numtasks,/*numberoftasks*/
    rc,/*returncode*/
    i;
MPI_Statusstatus;
/*ObtainnumberoftasksandtaskID*/
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("errorinitializingMPIandobtainingtaskIDinformation\n");
else
printf("taskID=%d\n",taskid);
/*SetseedforrandomnumbergeneratorequaltotaskID*/
srandom(taskid);
avepi=0;
for(i=0;i<ROUNDS;i++)
{
/*Alltaskscalculatepiusingdartboardalgorithm*/
homepi=dboard(DARTS);
/*UseMPIRoutinetosumvaluesofhomepiacrossalltasks
*Masterwillstoretheaccumulatedvalueinpisum
*-homepiisthesendbuffer
*-pisumisthereceivebuffer(usedbythereceivingtaskonly)
*-thesizeofthemessageissizeof(double)
*-MASTERisthetaskthatwillreceivetheresultofthereduction
*operation
*-MPI_SUMisapre-definedreductionfunction(double-precision
*floating-pointvectoraddition).Mustbedeclaredextern.
*-MPI_COMM_WORLDisthegroupoftasksthatwillparticipate.
rc=...ROTINAMPI...
if(rc!=0)
printf("%d:failureonmpc_reduce\n",taskid);
/*Mastercomputesaverageforthisiterationandalliterations*/
if(taskid==MASTER)
{
pi=pisum/numtasks;
avepi=((avepi*i)+pi)/(i+1);
printf("After%3dthrows,averagevalueofpi=%10.8f\n",
(DARTS*(i+1)),avepi);
}
}
MPI_Finalize();
return0;
}

```