

Universidade Estadual de Campinas



Centro Nacional de Processamento de Alto Desempenho  
São Paulo

CENAPAD

**Apostila de Treinamento:**  
**Introdução ao MPI**

Revisão: 2012

# Conteúdo

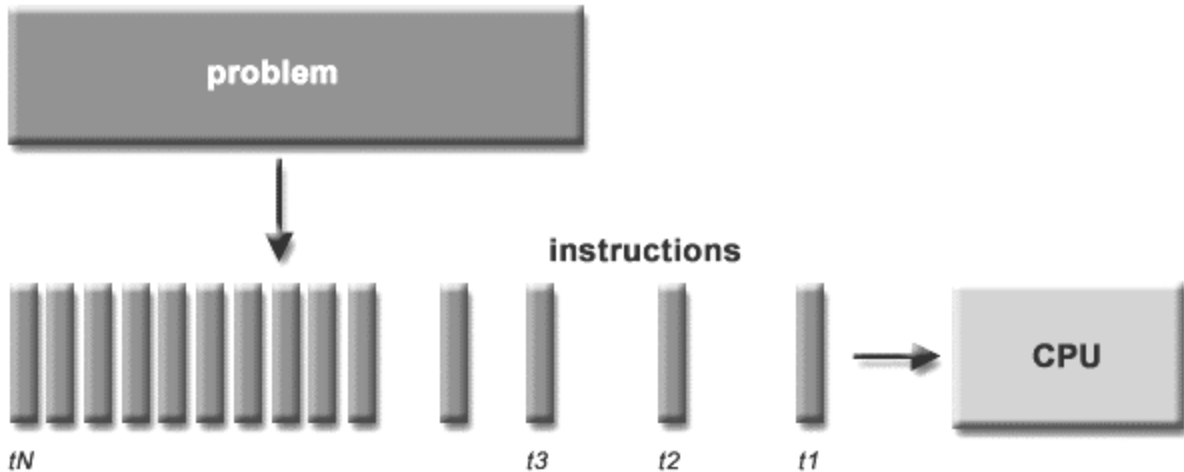
<b>1 – Computação Paralela</b>	<b>pag.04</b>
1.1 – Programação Serial	pag.04
1.2 – Programação Paralela	pag.04
1.3 – Recursos e Necessidades	pag.05
1.4 – Por que usar computação paralela?	pag.06
1.5 – Quem usa e para que usa computação paralela	pag.07
1.6 – Conceitos e Terminologia	pag.08
1.7 – Arquitetura de Memória	pag.13
“Shared Memory” – Memória Compartilhada	pag.13
“Distributed Memory” – Memória Distribuída	pag.15
1.8 – Modelos de Programação Paralela	pag.17
Memória Compartilhada	pag.17
Threads	pag.17
“Message Passing”	pag.18
“Data Parallel” – Paralelismo de Dados	pag.19
<b>2 – Criação de um Programa Paralelo</b>	<b>pag.20</b>
2.1 – Decomposição do Programa	pag.20
<b>3 – Considerações de Performance</b>	<b>pag.22</b>
3.1 – “Amdahl’s Law”	pag.22
3.2 – Balanceamento de Carga – “Load Balancing”	pag.23
<b>4 – Introdução a “Message Passing”</b>	<b>pag.24</b>
4.1 – O Modelo “Message Passing”	pag.24
4.2 – Bibliotecas de “Message Passing”	pag.24
4.3 – Terminologia de Comunicação	pag.25
<b>5 – Introdução ao MPI</b>	<b>pag.28</b>
5.1 – O que é MPI?	pag.28
5.2 – Histórico	pag.28
5.3 – Instalação Básica do MPICH2	pag.29
5.4 – Conceitos e Definições	pag.30
5.5 – Compilação	pag.34
5.6 – Execução	pag.35
<b>6 – Implementação Básica de um Programa com MPI</b>	<b>pag.36</b>
6.1 – Arquivo “headers”	pag.36
6.2 – Iniciar um Processo MPI	pag.37
6.3 – Finalizar um Processo MPI	pag.38
6.4 – Identificar um Processo MPI	pag.39
6.5 – Informar o Número de Processos MPI	pag.40
6.6 – Enviar Mensagens no MPI	pag.41
6.7 – Receber Mensagens no MPI	pag.42
6.8 – “MPI Message”	pag.43
<b>LABORATÓRIO 1 – Rotinas Básicas do MPI</b>	<b>pag.44</b>
<b>Exercício 1</b>	<b>pag.44</b>
<b>Exercício 2</b>	<b>pag.45</b>
<b>Exercício 3</b>	<b>pag.46</b>
<b>Exercício 4</b>	<b>pag.47</b>
<b>Exercício 5</b>	<b>pag.48</b>
<b>Exercício 6</b>	<b>pag.50</b>
<b>7 – Método de Comunicação Point-to-Point</b>	<b>pag.51</b>
7.1 – “Blocking Synchronous Send”	pag.52
7.2 – “Blocking Ready Send”	pag.53
7.3 – “Blocking Buffered Send”	pag.54
7.4 – “Blocking Standard Send”	pag.55
7.5 – Rotina MPI_Buffer_attach	pag.57
7.6 – Rotina MPI_Buffer_detach	pag.57
7.7 – “Deadlock”	pag.58
7.8 – Rotinas de Comunicação “Non-Blocking”	pag.59
7.8.1 – Rotina MPI_Issend – Non-blocking Synchronous	pag.60
7.8.2 – Rotina MPI_Irsend – Non-blocking Ready Send	pag.60

7.8.3 – Rotina MPI_Ibsend – Non-blocking Buffered Send	pag.60
7.8.4 – Rotina MPI_Isend – Non-blocking Standard Send	pag.60
7.8.5 – Rotina MPI_Irecv – Non-blocking Receive	pag.60
7.9 – Rotinas Auxiliares as Rotinas “Non-Blocking”	pag.62
7.9.1 – Rotina MPI_Wait	pag.62
7.9.2 – Rotina MPI_Test	pag.63
7.10 – Conclusões	pag.64
<b>LABORATORIO 2 – Comunicação Point-to-Point</b>	<b>pag.65</b>
Exercício 1	pag.65
Exercício 2	pag.66
Exercício 3	pag.67
Exercício 4	pag.68
<b>8 – Rotinas de Comunicação Coletiva</b>	<b>pag.69</b>
8.1 – “Synchronization”	pag.70
8.1.1 – Rotina MPI_Barrier	pag.70
8.2 – “Data Movement”	pag.71
8.2.1 – Rotina MPI_Broadcast	pag.71
8.2.2 – Rotina MPI_Scatter	pag.73
8.2.3 – Rotina MPI_Gather	pag.74
8.2.4 – Rotina MPI_Allgather	pag.77
8.2.5 – Rotina MPI_Alltoall	pag.78
8.3 – Computação Global	pag.79
8.3.1 – Rotina MPI_Reduce	pag.79
<b>LABORATORIO 3 – Comunicação Coletiva</b>	<b>pag.81</b>
Exercício 1	pag.81
Exercício 2	pag.82
Exercício 3	pag.83
Exercício 4	pag.84
<b>9 – Grupos e “Communicators”</b>	<b>pag.85</b>
9.1 – Identifica um Grupo – Rotina MPI_Comm_group	pag.86
9.2 – Cria um Grupo por Inclusão – Rotina MPI_Group_incl	pag.86
9.3 – Cria um Grupo por Exclusão – Rotina MPI_Group_excl	pag.86
9.4 – Identifica Processo no Grupo – Rotina MPI_Group_rank	pag.87
9.5 – Cria um “Communicator” – Rotina MPI_Comm_create	pag.87
9.6 – Apaga Grupos – Rotina MPI_Group_free	pag.88
9.7 – Apaga “Communicators” – Rotina MPI_Comm_free	pag.88
Exemplo de Uso de Grupos e “Communicators”	pag.89
<b>LABORATÓRIO 4 – Grupo e “Communicator”</b>	<b>pag.90</b>
Exercício 1	pag.90
Exercício 2	pag.91
<b>LABORATÓRIO 5 – Problemas de Execução</b>	<b>pag.92</b>
Exercício 1	pag.92
Exercício 2	pag.93
Exercício 3	pag.94
<b>10 – Referências</b>	<b>pag.95</b>

# 1-Computação Paralela

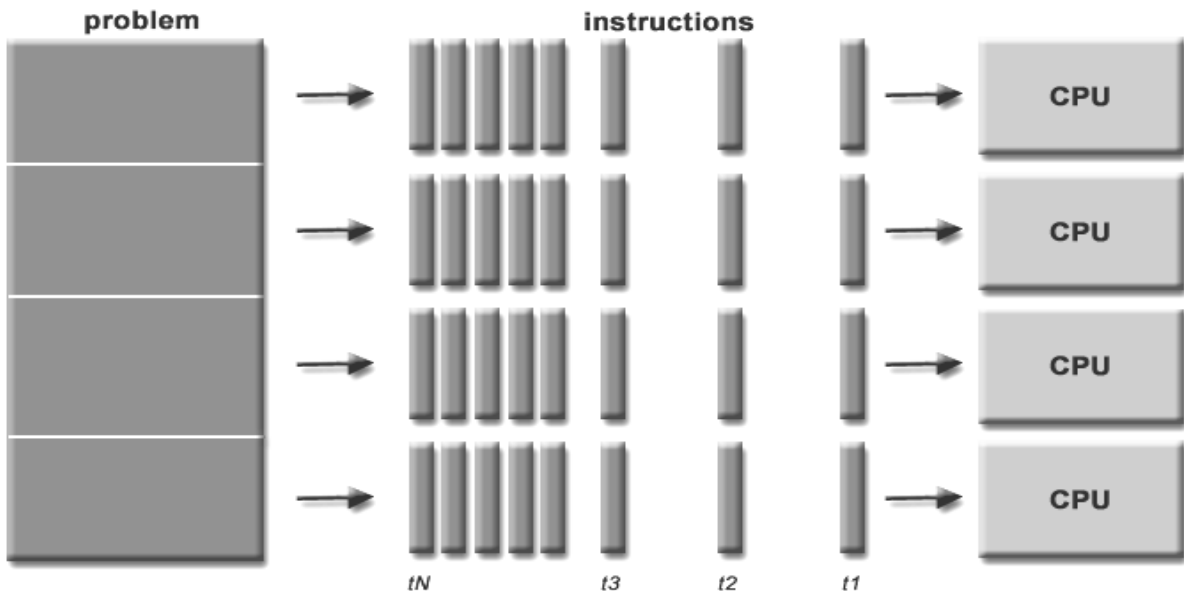
## 1.1- Processamento Serial

- Tradicionalmente os programas são escritos para processamento serial:
  - Execução em um computador com única CPU;
  - Os problemas são divididos em uma série de instruções;
  - Cada instrução é executada uma após a outra;
  - Somente uma instrução é executada por vez em qualquer momento do tempo de execução.



## 1.2 – Processamento Paralelo

- O processamento paralelo, de uma maneira simples, é o uso simultâneo de diversos recursos computacionais para resolver um problema:
  - Execução utilizando diversas CPUs;
  - Os problemas são divididos em diversos “pedaços”, que podem ser resolvidos concorrentemente;
  - Cada “pedaço”, por sua vez, é dividido em uma série de instruções;
  - As instruções de cada “pedaço” serão executadas simultaneamente em diferentes CPUs.



## 1.3 – Recursos e Necessidades

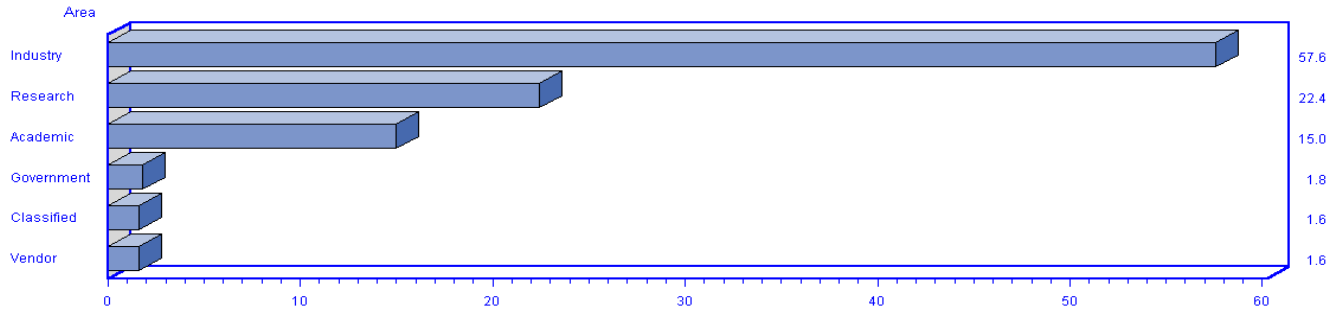
- Os recursos computacionais para o processamento paralelo podem ser:
  - Um unidade de processamento (um nó) com várias CPUs (cores);
  - Um número ilimitado de computadores conectados por uma rede;
  - Uma combinação de ambos os itens acima.
  
- Os problemas computacionais devem possuir características que possibilitem:
  - Serem divididos em “pedaços”, e que possam ser solucionados, de uma maneira **concorrente, simultânea**;
  - Executar diversas instruções de programa a qualquer momento no tempo de execução;
  - Serem solucionados em um menor espaço de tempo que o equivalente serial;
  
- A computação paralela vem sendo considerada como “o objetivo final em computação”, vem sendo motivada pelas simulações numéricas de sistemas complexos e devido aos grandes desafios computacionais científicos:
  - Previsão do tempo;
  - Comportamento do clima no mundo;
  - Reações químicas e nucleares;
  - Estudo do genoma humano;
  - Desenvolvimento de equipamentos mecânicos: Simulação de aviões, carros, máquinas;
  - Desenvolvimento de circuitos eletrônicos: Tecnologia de semicondutores, microchips;
  - Etc.
  
- As aplicações comerciais, também intensificaram num igual e forte esforço, no desenvolvimento de computadores cada vez mais rápidos. As aplicações necessitam de processamento de imensas bases de dados de várias maneiras sofisticadas:
  - Banco de Dados com processamento paralelo (Oracle, DB2) ;
  - “Data Mining”, mineração de dados;
  - Exploração de petróleo;
  - Mecanismos de busca na web;
  - Diagnósticos médicos assistidos por computador;
  - Realidade virtual;
  - Tecnologia em multimídia;

## 1.4 – Por que usar computação paralela?

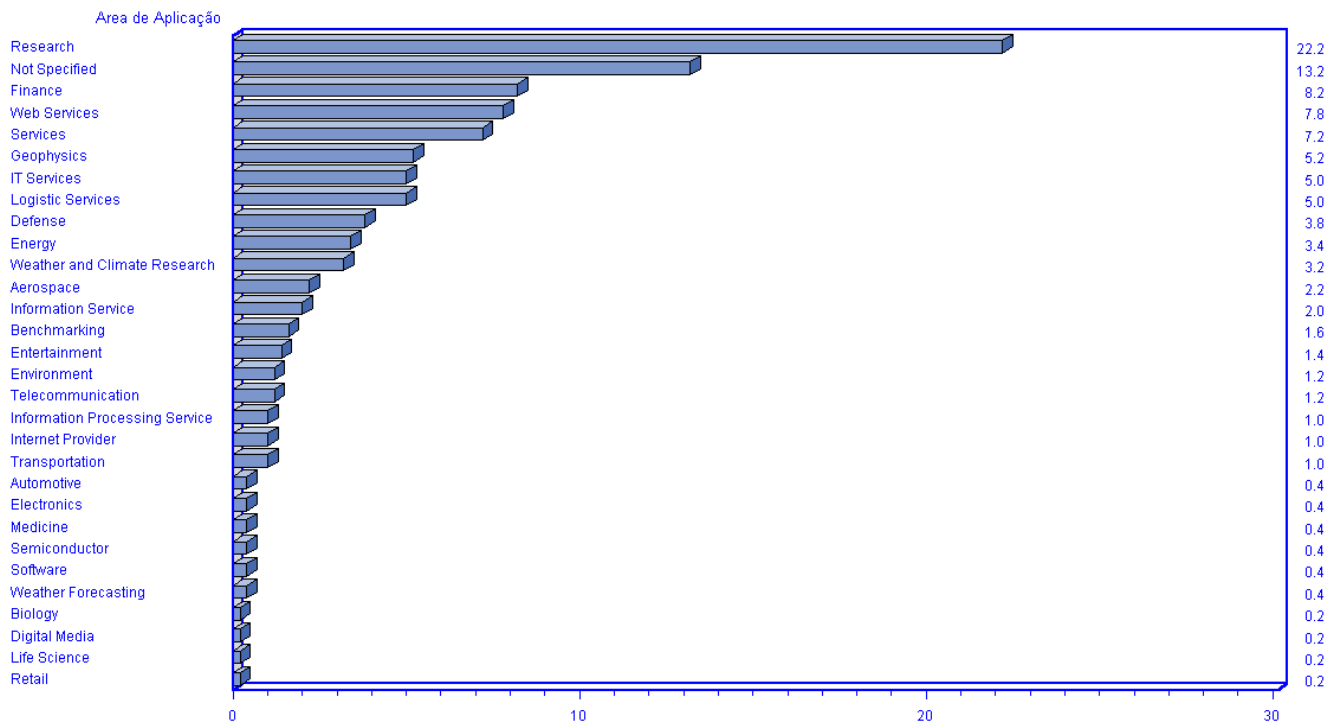
- Principais razões:
  - Reduzir o tempo total de execução;
  - Resolver grandes problemas;
  - **Utilizar concorrência (vários processamentos ao mesmo tempo).**
  
- Outras razões:
  - Tirar vantagens dos recursos disponíveis na rede (Internet), disponibilizados por grandes centros de processamento de dados (Computação em GRID);
  - Baixar os custos de processamento, utilizando estações de trabalho, processadores comuns no mercado, ao invés de se utilizar “supercomputadores”;
  - Recursos de memória – Um simples computador possui memória finita ao invés de se utilizar os recursos de memória de diversos computadores;
  - Limitação do processamento serial, baseados nos limites físicos e práticos da construção de máquinas seriais mais rápidas:
    - Velocidade de transmissão das informações:
      - Limite absoluto: Velocidade da luz – 30cm/nanosegundos.
      - Limite no cobre – 9cm/nanosegundos.
    - Limites de miniaturização: Nível atômico ou molecular
    - Limites econômicos;
  - O paralelismo é uma necessidade e, é o futuro da computação.

# 1.5 – Quem usa e para que usa computação paralela

**Percentual de Uso de Processamento Paralelo**  
Por Area



**Percentual de Uso de Processamento Paralelo**  
Por Area de Aplicação



Fonte: top500.org 11/2011

## 1.6 – Conceitos e Terminologia

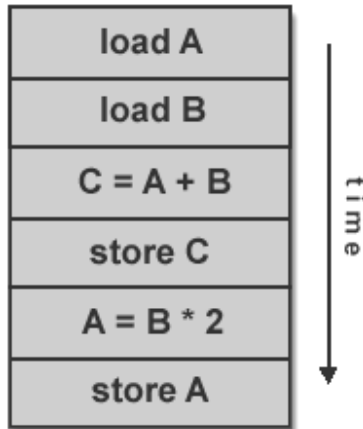
- Existem diversas maneiras de se classificar computadores paralelos. Uma das mais utilizadas, desde 1966, é a classificação de **Flynn**.
- Essa classificação distingue arquiteturas de computadores com múltiplos processadores, combinando duas dimensões independentes: **Instrução** e **Dados**. Cada dimensão só pode ter dois estados: **Simple**s ou **Múltiplos**.
- Existem quatro possibilidades de classificação:

<b>SISD</b> Single Instruction, Single Data	<b>SIMD</b> Single Instruction, Multiple Data
<b>MISD</b> Multiple Instruction, Single Data	<b>MIMD</b> Multiple Instruction, Multiple Data



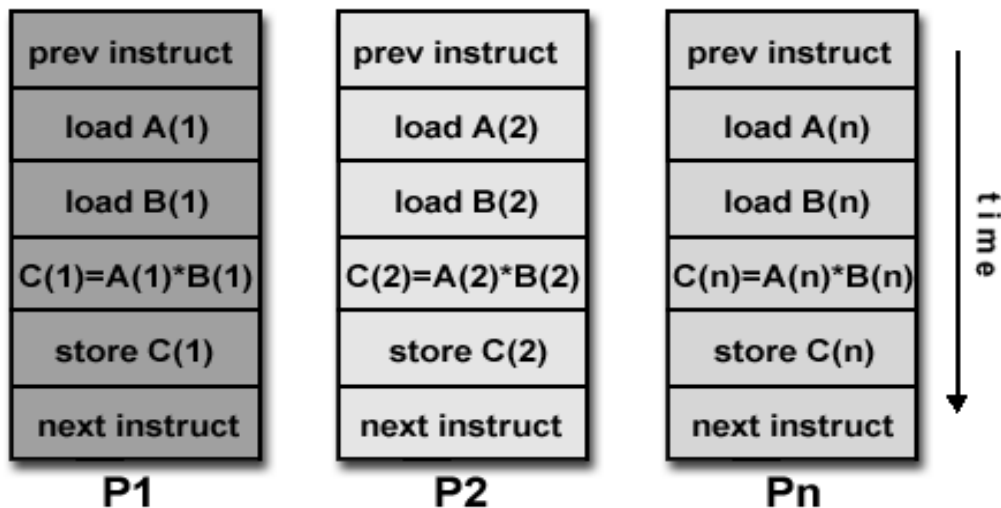
- **SISD – Single Instruction, Single Data**

- Computador serial;
- Somente uma instrução por vez;
- Somente um fluxo de dado é utilizado por ciclos de CPU;
- Exemplos: A maioria dos PCs;



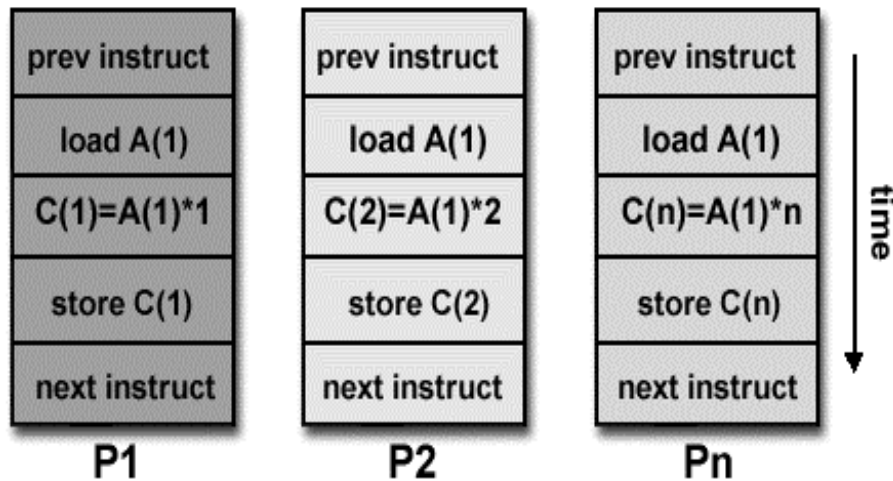
- **SIMD – Single Instruction, Multiple Data**

- Tipo de computador paralelo;
- Todas as CPUs executam a mesma instrução;
- Cada CPU opera em um conjunto de dados diferentes;
- Arquitetura idealizada para aplicações que necessitam processar vetores: Tratamento de imagens;
- Exemplos: IBM 9000, Cray C90, NEC SX-3.



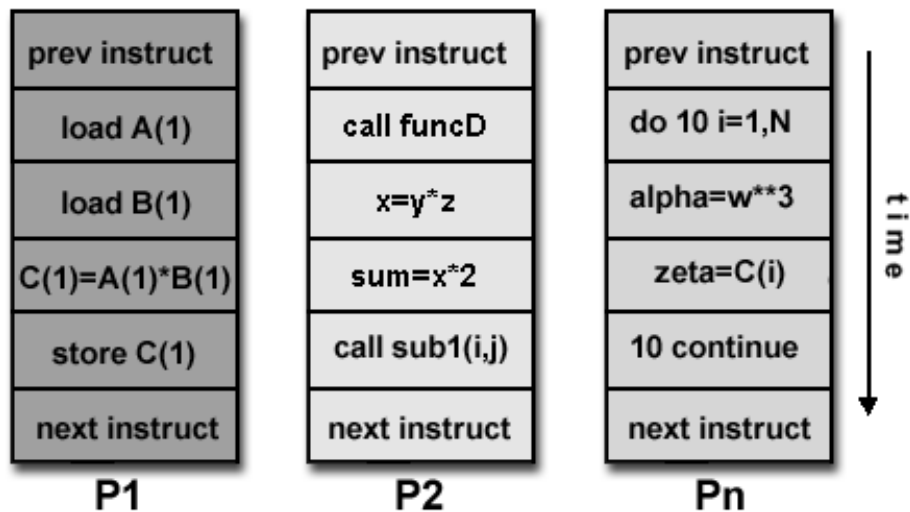
- **MISD – Multiple Instruction, Single Data**

- Um único conjunto de dados é processado por múltiplas CPUs;
- Não existe computador comercial com essa classificação;
- Possíveis aplicações:
  - Múltiplos filtros de frequência, analisando um mesmo sinal;
  - Múltiplos algoritmos de criptografia analisando uma única mensagem;



- **MIMD – Multiple Instruction, Multiple Data**

- Computador paralelo mais comum;
- Cada CPU pode executar um conjunto de instruções diferentes;
- Cada CPU pode processar um conjunto de dados diferentes;
- A execução pode ser síncrona ou assíncrona;
- Exemplos: A maioria dos supercomputadores atuais, redes de computadores (“grid”), clusters



- **Task - Tarefa**

Seção lógica de um “trabalho” computacional. Tipicamente, a **Tarefa** representa um programa com um conjunto de instruções que são executadas pelo processador.

- **Parallel Task – Tarefa Paralela**

Seção lógica de um “trabalho” computacional, que pode ser executado, de maneira confiável, por múltiplos processadores.

- **Serial Execution – Execução Serial**

Execução de um programa seqüencialmente, comando após comando. Normalmente, é a situação de máquinas com um único processador.

- **Parallel Execution – Execução Paralela**

Execução de um programa que possui mais de uma seção lógica (tarefa) e que podem ser executadas ao mesmo tempo.

- **Shared Memory – Memória Compartilhada**

Descreve uma arquitetura de computador aonde todos os processadores possuem acesso direto (“Data Bus”) à uma mesma memória física.

- **Distributed Memory – Memória Distribuída**

Descreve uma arquitetura de computador aonde cada processador utiliza sua própria memória física, mas todos são conectados entre si por meio de uma interface de comunicação, um “switch” (Ex.: InfiniBand), ou uma rede.

- **Synchronization – Sincronização**

É a coordenação das comunicações entre as tarefas paralelas em tempo real. É utilizado em aplicações cujas tarefas paralelas não podem prosseguir enquanto as outras tarefas atinjam o mesmo ponto lógico da execução. Sincronização, usualmente, envolve tempo de espera por pelo menos uma tarefa, o que pode causar um aumento no tempo total de execução da aplicação.

- **Granularity - Granularidade**

Em computação paralela, granularidade é uma medida da razão entre computação e comunicação.

*Coarse (Grosso)* – Grande número de trabalho computacional entre eventos de comunicação;  
*Fine (Fino)* – Pequeno número de trabalho computacional entre eventos de comunicação;

- **Observed Speedup – Aumento da Velocidade de Processamento**

$$\frac{\text{Tempo total da execução serial}}{\text{Tempo total da execução paralela}}$$

- **Parallel Overhead**

Tempo gasto e necessário para coordenar as tarefas paralelas. Os fatores de coordenação podem ser:

- Início de processos paralelos
- Sincronizações durante a execução
- Comunicação de dados entre os processos
- Compiladores paralelos, bibliotecas, ferramentas, sistema operacional, etc.
- Finalização de processos paralelos

- **Massively Parallel**

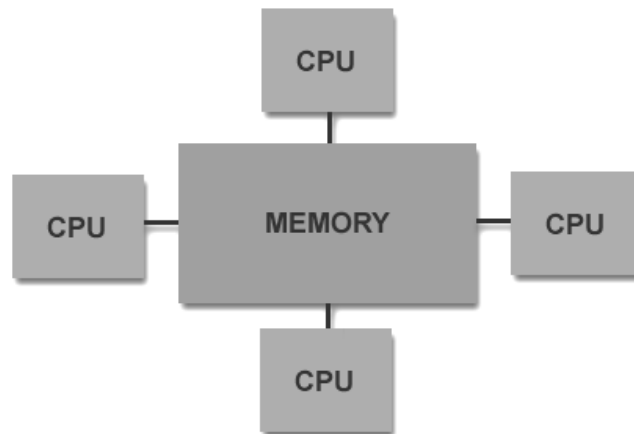
Hardware que constitui o sistema paralelo, possuindo diversos processadores e permitindo o aumento desse sistema (Quantidade atual de processadores nos grandes centros de supercomputação, gira em torno de 6 dígitos).

- **Scalability – Escalabilidade**

Refere-se ao sistema paralelo que demonstra a habilidade de aumentar a velocidade de processamento com a adição de novos processadores.

## 1.7 – Arquitetura de Memória

### Shared Memory – Memória Compartilhada



- Computadores de memória compartilhada variam muito de configuração, mas no geral possuem em comum a habilidade de todos os processadores acessarem toda a memória como um espaço de endereçamento comum ou global;
- Múltiplos processadores podem operar de forma independente, mas compartilham os mesmos recursos de memória;
- Mudanças num endereço de memória por um processador será visível por todos os outros processadores;
- As máquinas com memória compartilhada podem ser divididas em duas classes baseadas no número de acessos e no intervalo de acesso à memória: *UMA* e *NUMA*.

## **Uniform Memory Access (UMA)**

- Máquinas SMP (Symmetric Multiprocessor);
- Processadores idênticos;
- O Número de acessos à memória é dividido igualmente por entre todos os processadores;
- Também chamadas de CC-UMA – Cache Coherent UMA – Se um processador altera um endereço de memória, todos os processadores serão “avisados” da atualização. É uma implementação à nível de hardware.

## **Non-Uniform Memory Access (NUMA)**

- Ambiente paralelo que possui a ligação física entre duas ou mais máquinas SMPs;
- Cada máquina SMP tem acesso direto à memória das outras máquinas SMPs;
- O número de acessos a memória não é igualmente dividido entre todos os processadores do ambiente;
- O acesso a memória de outra SMP é mais lento;
- Se for mantida a coerência de cachê, o ambiente é chamado de CC-NUMA

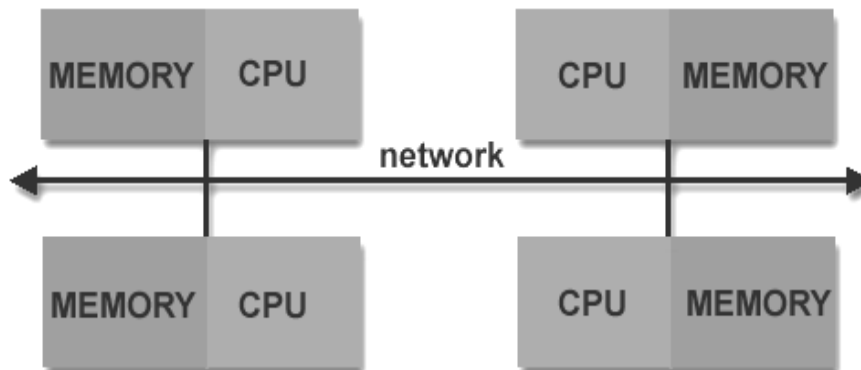
### **• Vantagens:**

- O Endereçamento global permite uma programação mais simples e um processamento mais rápido;
- O compartilhamento de dados entre tarefas é rápido e uniforme devido a proximidade entre memória e CPUs.

### **• Desvantagens:**

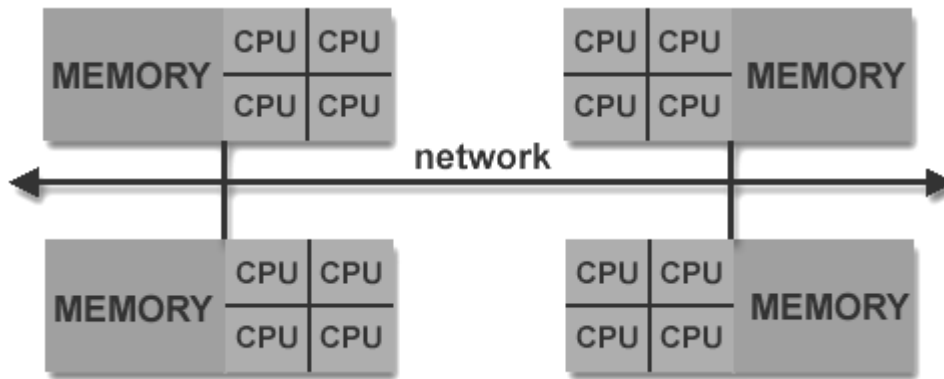
- Escalabilidade. A adição de mais CPUs pode aumentar geometricamente o tráfego de dados na memória compartilhada com as CPUs e com o sistema de gerenciamento da memória cache;
- Alto custo na fabricação de máquinas SMPs com muitos processadores;

## Distributed Memory – Memória Distribuída



- Sistemas de memória distribuída necessitam de uma rede de comunicação que conecta os processadores entre si;
- Cada processador possui sua própria memória local;
- O endereçamento de memória de um processador não endereça a memória de outro processador, ou seja, não existe o conceito de endereçamento global;
- Cada processador opera independentemente. Mudanças em um endereço de memória de um processador, não são informadas para os outros processadores. Não existe o conceito de coerência de cache;
- Quando o processador necessita de um dado de outro processador, será a tarefa paralela em execução do programador, que explicitamente definirá como será a comunicação. A sincronização de tarefas é uma responsabilidade do programador;
- **Vantagens:**
  - Escalabilidade. A memória do sistema aumenta com o aumento do número de processadores;
  - Cada processador tem acesso rápido a sua memória, sem a interferência e o custo em se manter um sistema de coerência de cache;
- **Desvantagens:**
  - O Programador é responsável por todos os detalhes associados a comunicação entre os processadores;

## Híbridos



- Atualmente os computadores mais rápidos no mundo utilizam ambas as arquiteturas de memória;
- Os componentes de memória compartilhada são usualmente máquinas SMPs com coerência de cache;
- Os componentes de memória distribuída são máquinas SMPs conectadas em rede;

Arquitetura	CC-UMA	CC-NUMA	Distributed
<b>Exemplos</b>	SMPs DEC/Compaq SGI Challenge IBM POWER3	SGI Altix HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2
<b>Comunicação</b>	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
<b>Escalabilidade</b>	Dezenas de processadores	Centenas de processadores	Milhares de processadores
<b>Disponibilidade de Software</b>	Milhares	milhares	centenas



## 1.8 – Modelos de Programação Paralela

### Memória Compartilhada

- As tarefas compartilham o mesmo espaço de endereçamento. O acesso é feito em modo assíncrono;
- A grande vantagem desse modelo, é que a comunicação entre as tarefas é feita de maneira implícita, não havendo necessidade de controlar essa comunicação, tornando a programação mais fácil;
- A implementação desse modelo é feita pelos compiladores nativos do ambiente.

### Threads

- Um único processo pode possuir múltiplos e concorrentes sub processos, ou melhor, “processos leves”, que são sub tarefas independentes;
- Todos os “processos leves” compartilham o mesmo espaço de memória do processo principal;
- Não existe proteção de memória entre threads.
- Uma analogia simples que pode ser utilizada para entender threads, seria a de um programa com várias sub-rotinas;

1 - O programa principal **a.out** inicia a execução, carrega e adquire todos os recursos necessários do sistema para a execução;

2 – O programa **a.out** efetua algum trabalho serial e então gera um número de tarefas (threads), que podem ser agendados e executados simultaneamente pelo sistema operacional;

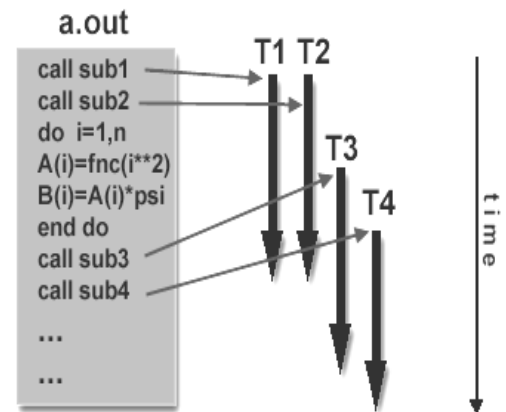
3 – Cada thread possui seus próprios dados, mas podem compartilhar os recursos do programa principal, **a.out**, evitando, assim, a replicação de recursos para cada thread;

4 – Cada thread se beneficia também do acesso ao endereçamento de memória do programa **a.out**;

5 – O trabalho de uma thread pode ser melhor descrito como uma sub rotina do programa principal. Qualquer thread pode executar qualquer sub rotina, ao mesmo tempo, que outras threads;

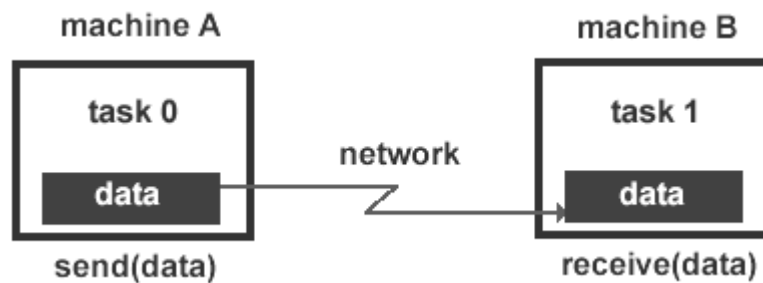
6 – As threads se comunicam entre si através da memória principal (atualizando endereços). Isto requer sincronização nos acessos para garantir que não existam outras threads tentando atualizar o mesmo endereço ao mesmo tempo;

- Threads são normalmente associados a arquiteturas de memória compartilhada;
- Implementações:
  - Posix Threads (Pthreads) – somente em C
  - OpenMP – C/C++ e Fortran



## “Message Passing”

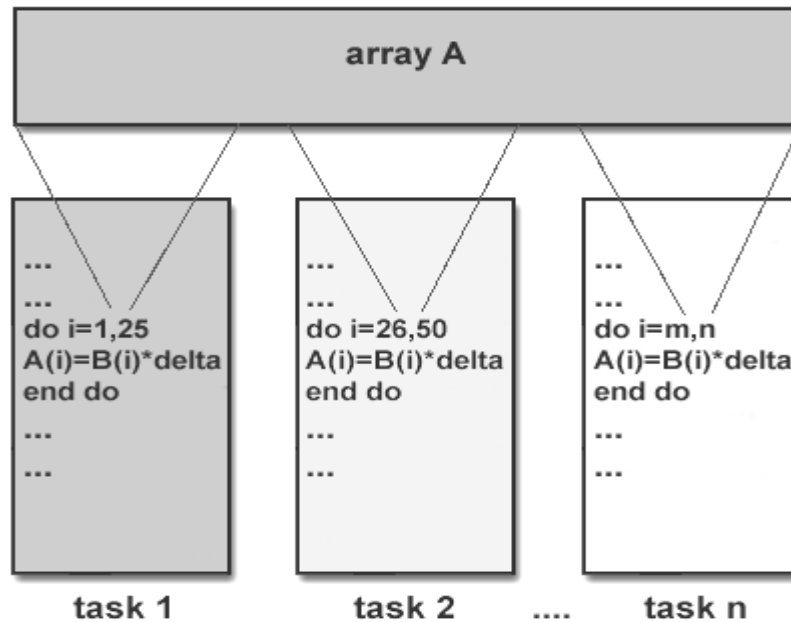
- Modelo normalmente associado a ambiente de memória distribuída;
- Múltiplas tarefas iniciadas e distribuídas pelos processadores do ambiente, utilizando o seu próprio endereçamento de memória;
- As tarefas compartilham dados através de comunicação de envio e recebimento de mensagens (“message-passing”);



- O programador é responsável por determinar todo o paralelismo e a comunicação entre as tarefas;
- A transferência de dados, usualmente requer a cooperação de operações entre os processos. Ex.: Uma operação de envio de mensagem por um processo tem que “casar” com uma operação de recebimento de outro processo;
- Implementações: PVM, Linda, MPI

## “Data Parallel” – Paralelismo de Dados

- O trabalho paralelo é efetuado em um conjunto de dados;
- Os dados devem estar organizados na forma de vetores;
- Cada tarefa trabalha com partições diferentes da estrutura de dados;
- Cada tarefa efetua a mesma operação em sua partição da estrutura de dados;



- Em Memória compartilhada – Todos as tarefas possuem acesso a estrutura de dados na memória global;
- Em Memória distribuída – Partições da estrutura de dados são distribuídas para a memória de cada tarefa em cada processador do ambiente;
- A programação baseada nesse modelo utiliza recursos de paralelismo de dados, que normalmente são sub rotinas de uma biblioteca “Data Parallel” ou diretivas (opções) de um compilador;
- Implementações:
  - Data Parallel Compilers: Fortran90/95, HPF - High Performance Fortran
  - Message Passing: MPI, LAM, MPICH

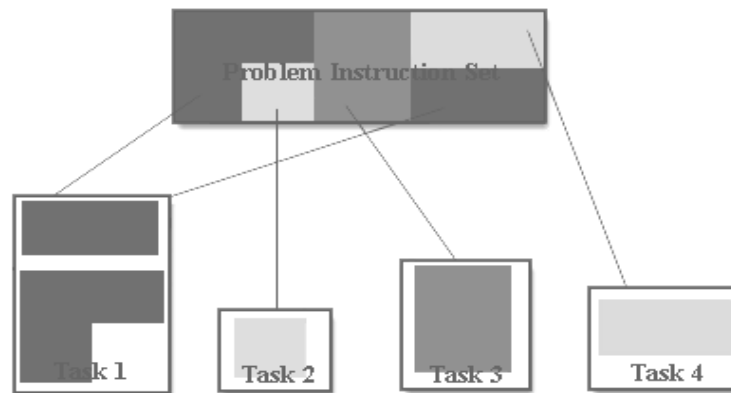
## 2 - Criação de um Programa Paralelo

### 2.1 - Decomposição do Programa

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

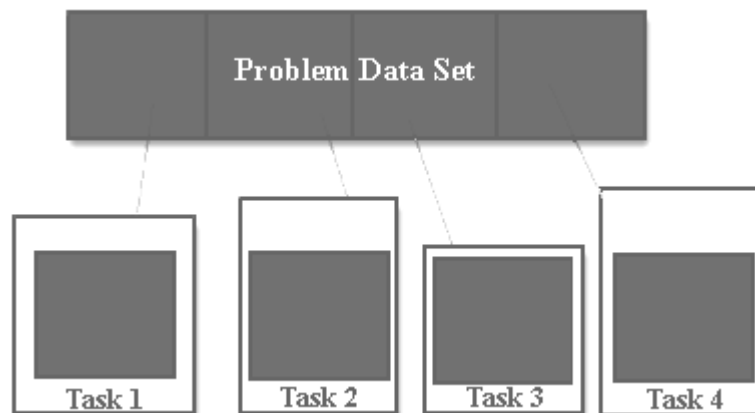
#### Decomposição Funcional

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

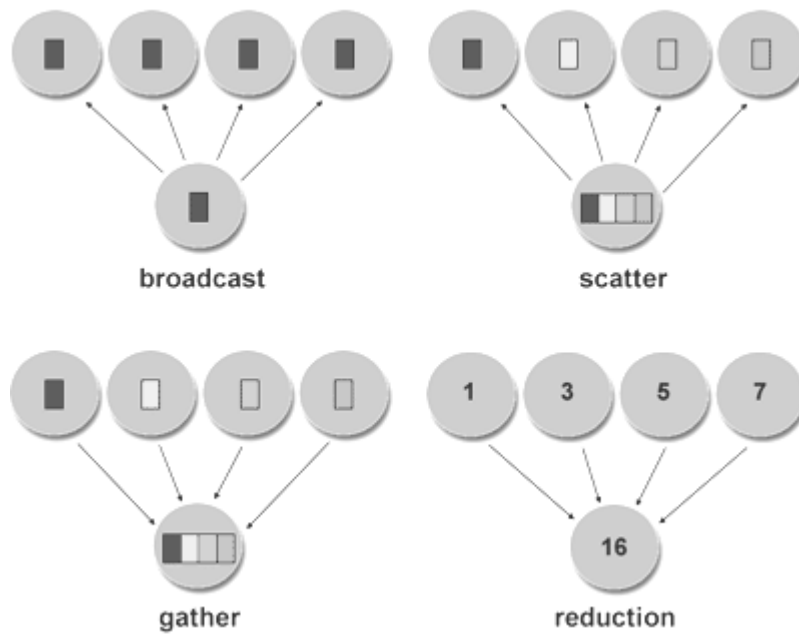


#### Decomposição de Domínio

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



- Os métodos de comunicação para "Message-Passing" e para "Data Parallel", são exatamente os mesmos.
  - Comunicação Point to Point (Send-Receive);
  - Comunicação Coletiva: Broadcast, Scatter, Gather, Collective Computations (Reduction).



### 3 - Considerações de Performance

#### 3.1 - Amdahl's Law

- A lei de Amdahl's determina o potencial de aumento de velocidade a partir da porcentagem de paralelismo, **por decomposição funcional**, de um programa ( **f** ):

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

- 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 do processamento, a relação passará a ser modelada por:

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

**P = Porcentagem paralela;**

**N = Número de processadores;**

**S = Porcentagem serial;**

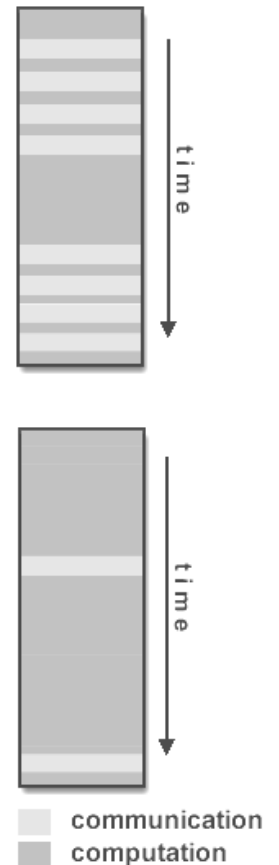
<b>N</b>	<b>P = 0,50</b>	<b>P = 0,90</b>	<b>P = 0,99</b>
<b>10</b>	<b>1,82</b>	<b>5,26</b>	<b>9,17</b>
<b>100</b>	<b>1,98</b>	<b>9,17</b>	<b>50,25</b>
<b>1000</b>	<b>1,99</b>	<b>9,91</b>	<b>90,99</b>
<b>10000</b>	<b>1,99</b>	<b>9,91</b>	<b>99,02</b>

## 3.2 - 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.



## 4 – Introdução a "Message-Passing"

### 4.1 - 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 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).

### 4.2 – Bibliotecas de "Message-Passing"

- A implementação de uma biblioteca de "Message-Passing", baseia-se na construção de rotinas de comunicação entre processos.

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

**Proprietárias** – MPL(IBM), NX(NEC), CMMD(CM), MPI(IBM, CRAY, SGI, Intel), 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, etc;
  - Rotinas de comunicação "Point-to-Point": Enviar e receber mensagens entre dois processos;
  - Rotinas de comunicação Coletiva: "broadcast", "scatter", "gather", "reduce" e sincronizar processos.



## 4.3 - Terminologia de Comunicação

- **Buffering**

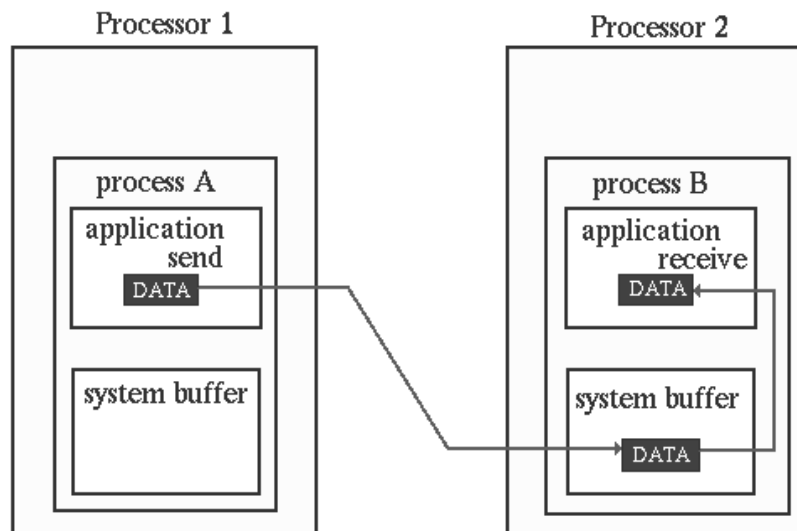
Cópia temporária de mensagens entre endereços de memória efetuada pelo sistema MPI, como parte de seu protocolo de transmissão. A cópia ocorre entre o **buffer da aplicação** (variáveis do programa) e o **buffer do sistema** (definido pela biblioteca MPI);

Em um processamento teórico, toda operação de envio deve estar perfeitamente sincronizada e casada com uma operação de recebimento. Isto é raro! Portanto, toda implementação de MPI deve possibilitar o armazenamento das mensagens no caso dos processos não estarem sincronizados;

Considere os dois casos abaixo:

- Uma operação de envio ocorre 5 segundos antes da operação de recebimento estar ativa. A onde estará a mensagem enquanto o recebimento estiver pendente?
- Múltiplas operações de envio são feitas para o mesmo processo, que só pode aceitar uma mensagem por vez. O que acontece com as outras mensagens enviadas?

A própria biblioteca MPI decide o que fazer com as mensagens nesses casos. Normalmente é reservada uma área de memória para armazenar os dados em trânsito – **buffer do sistema MPI (System Buffer)**.



Path of a message buffered at the receiving process

O espaço do System Buffer é:

- Transparente para o programador e gerenciado pelo MPI;
- Um recurso limitado e facilmente esgotado;
- Não é bem documentado;
- Pode existir no processo que envia, no que recebe e em ambos;
- É um recurso que melhora a performance do programa, pois permite operações de envio e recebimento, assíncronos.

- **Blocking**

Uma rotina de comunicação é **blocking**, quando a finalização da execução da rotina, depende de certos eventos, ou seja, espera por determinada ação, antes de liberar a continuação do processamento:

- Uma rotina **blocking send** somente termina após o **buffer da aplicação** (variável), estiver seguro e liberado para uso. Seguro, significa que qualquer modificação no buffer, não irá afetar o dado transmitido para o processo que recebe. Seguro, não implica que o dado já tenha sido recebido;
- Um **blocking send** pode ser síncrono, o que significa que existe uma comunicação ocorrendo com o processo que recebe, com confirmação da transmissão da mensagem pelo processo que recebe;
- Um **blocking send** pode ser assíncrono, o que significa que o **system buffer** está sendo utilizado para armazenar uma mensagem para ser enviada;
- Um **blocking receive** só retorna a execução após a mensagem ter chegado, e estar pronta para ser utilizada pelo programa.

- **Non-blocking**

Uma rotina de comunicação é **non-blocking**, quando a finalização da execução da rotina, não depende de certos eventos, ou seja, não há espera por uma ação, o processo continua sendo executado normalmente;

- As operações **non-blocking** solicitam à biblioteca MPI que execute a operação quando for possível. O programador não tem idéia de quando isso irá ocorrer;
- Não é seguro modificar o valor do buffer de aplicação, até que se saiba que a operação **non\_blocking** tenha sido executada pela biblioteca MPI;
- As operações **non-blocking** são utilizadas para evitar problemas de comunicação (“**deadlock**”) e melhorar a performance.

- **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).

<b>Bloking</b>	<b>Send</b>	Finaliza, quando o "buffer" de envio está pronto para ser reutilizado;
	<b>Receive</b>	Finaliza, quando o "buffer" de recebimento está pronto para ser reutilizado;
<b>Nonblocking</b>		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.

**"Synchronization 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.

## 5 – Introdução ao MPI

### 5.1 - O que é MPI ?

- **Message Passing Interface**
- Uma biblioteca de "Message-Passing", desenvolvida para ser padrão, inicialmente, 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 se analisar a performance.
- Utilizado por programas codificados em C e FORTRAN.
- A plataforma para uso do MPI, são os ambientes com máquinas paralelas massivas.
- Todo paralelismo é **explícito**: o programador é responsável em identificar o paralelismo e implementar um algoritmo utilizando construções com o MPI.

### 5.2 – Histórico

- **Final 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); 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**). Criação de um **MPI Forum**, que consiste eventualmente de aproximadamente 175 pessoas de 40 organizações, incluindo fabricantes de computadores, empresas de softwares, universidades e cientistas de aplicação.

- **Mai de 1994**

Disponibilização como domínio público, da versão padrão do MPI, atualmente, o padrão **MPI-2**: <http://www.mcs.anl.gov/research/projects/mpich2/>

- **Dezembro de 1995**

Conferência de Supercomputação 95 - Reunião para discussão do **MPI-2** e suas extensões.

## 5.3 – Instalação Básica do MPICH2

- **download**

wget <http://www.mcs.anl.gov/research/projects/mpich2/downloads/tarballs/1.3.2p1/mpich2-1.3.2p1.tar.gz>

- **desempacotar**

tar xzf mpich2.tar.gz ou gunzip -c mpich2.tar.gz | tar xf -

- **configurar a instalação**

Opção para definir o “Process Manager” - Administrador de Processos (**--with-pm=**):

**hydra** Opção “default”, utiliza os “daemons” nativos já instalados no ambiente: ssh, slurm, pbs;

**mpd** Opção tradicional do MPICH2 que instala um “daemon” próprio de controle de processos;

**smpd** Única opção disponível para o ambiente Windows;

**gforker** Opção de controle de processos utilizado em ambientes homogêneos de processadores e ambientes de memória compartilhada.

Opção para definir o método de comunicação (**--with-device=**):

**ch3:nemesis** Método “default”, de alta performance. Utilizado em ambiente híbridos. Usa memória compartilhada entre processos no mesmo “nó” e rede TCP/IP nos processos entre os “nós”;

**ch3:sock** Método para ambientes apenas com memória distribuída, utiliza a rede TCP/IP para comunicação entre todos os processos.

Exemplo do comando “configure” e outras opções, com os compiladores da GNU:

```
./configure --prefix=/usr/local/mpich2
            --with-pm=gforker
            --with-device=ch3:nemesis
            --enable-fc
            --enable-cxx
            --disable-check-compiler-flags
            --enable-fast=O3
            CC=gcc
            CXX=g++
            FC=gfortran
            F77=gfortran
            FCFLAGS=-m64
            FFLAGS=-m64
            CFLAGS=-m64
            CXXFLAGS=-m64
```

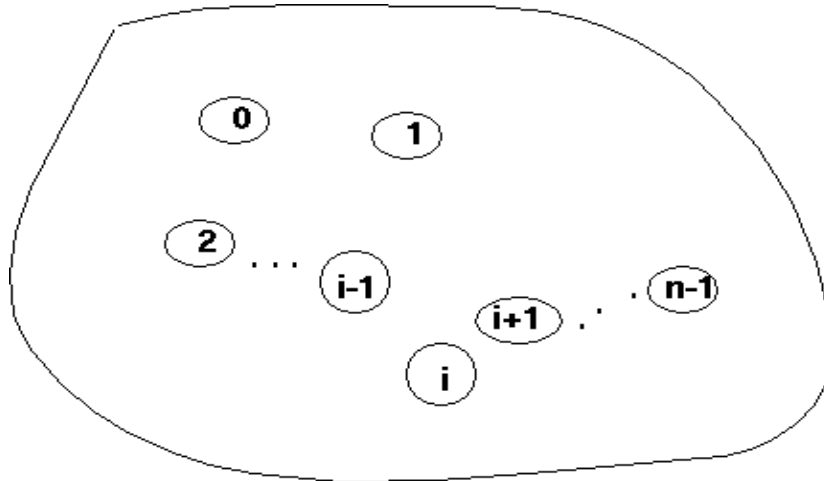
- **instalação**

```
make
make install
```

## 5.4 – Conceitos e Definições

- **Rank**

Todo processo tem uma única identificação, atribuída pelo sistema quando o processo é iniciado. 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 com um "communicator" já pré-estabelecido (**MPI\_COMM\_WORLD**).

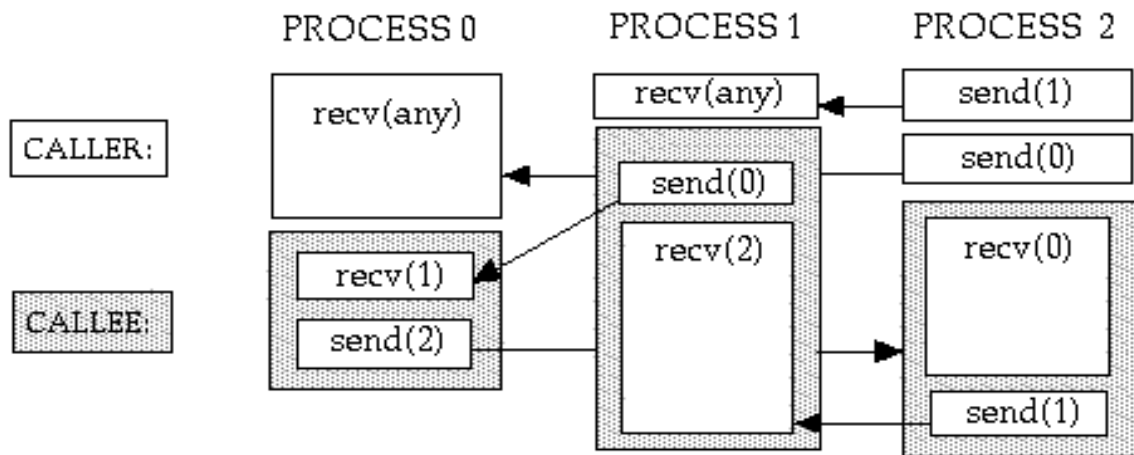
- **Communicator**

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 subrotina da biblioteca, pode por sua vez, usar uma mensagem idêntica a mensagem da aplicação do usuário.

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

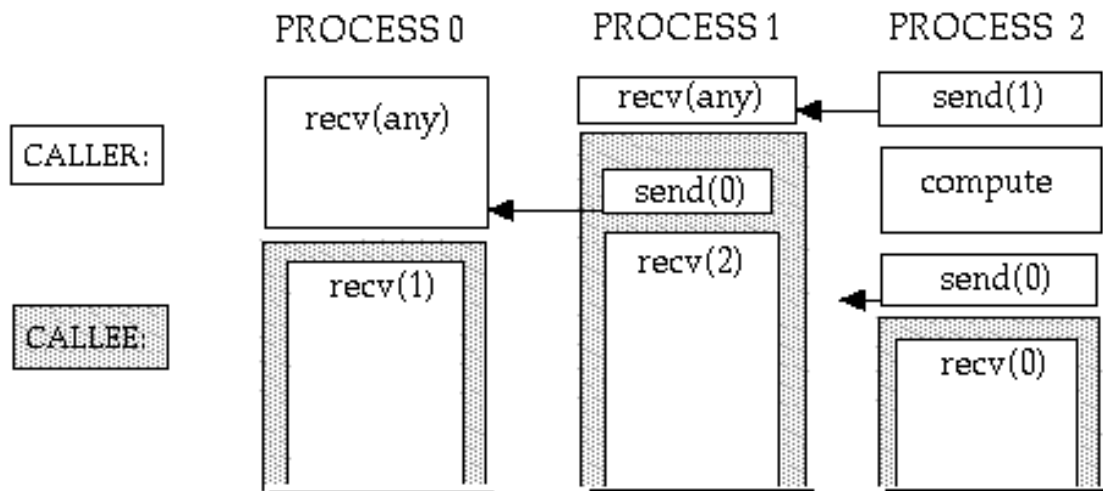
## Ação Desejada



Courtesy David Walker  
Oak Ridge Nat. Lab.

RLF.COMMDES 10/16/95

## Possível ação indesejada



Courtesy David Walker  
Oak Ridge Nat. Lab.

RLF.COMMINC 10/16/95

- **Application Buffer**

É um endereço normal de memória (Ex: endereço de uma 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. Normalmente, em comunicação assíncrona, numa operação de **send/receive**, o dado no "application buffer" pode necessitar ser copiado **de/para** o "system buffer" ("**Send Buffer**" e "**Receive Buffer**").

- **Blocking Communication**

Numa rotina de comunicação **bloking**, a finalização da chamada depende 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 "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 "application buffer" para o "system buffer", ou qualquer indicação do recebimento de uma mensagem. É da responsabilidade do programador, a certeza de que o "application buffer" esteja disponível para ser reutilizado.



- **Standard Send**

Implementação padrão do MPI de envio de mensagens, usada para transmitir dados de um processo para outro.

- **Synchronous Send**

Operação sincronizada entre o processo que envia e o processo que recebe. O processo que envia, bloqueia a execução do programa até que ocorra uma operação de "receive" no processo destino.

- **Buffered Send**

Operação de envio na qual se utiliza um novo "buffer" criado e adaptado ao tamanho dos dados que serão enviados. Isso pode ser necessário para evitar o processo de "buffering", devido ao espaço padrão 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.

## 5.5 - Compilação

A compilação de uma aplicação que utiliza rotinas MPI, depende da implementação **MPI** existente no ambiente.

### Ambiente IBM/AIX com “Parallel Enviroment” instalado

Fortran77

```
mpxlf <fonte.f> -o <executável>
```

Fortran90

```
mpxlf90 <fonte.f90> -o <executável>
```

C Standard

```
mpcc <fonte.c> -o <executável>
```

C++

```
mpCC <fonte.cxx> -o <executável>
```

### Ambiente LINUX com processador e compiladores da INTEL e biblioteca MPI proprietária

Fortran77 ou Fortran90

```
ifort <fonte.f ou .f90> -o <executável> -lmpi
```

C Standard

```
icc <fonte.c> -o <executável> -lmpi
```

C++

```
icpc <fonte.cxx> -o <executável> -lmpi
```

### Qualquer ambiente e compiladores, com biblioteca MPICH2

Fortran77

```
mpif77 <fonte.f> -o <executável>
```

Fortran90

```
mpif90 <fonte.f90> -o <executável>
```

C Standard

```
mpicc <fonte.c> -o <executável>
```

C++

```
mpicxx <fonte.cxx> -o <executável>
```

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

## 5.6 – Execução

A execução de uma aplicação que utiliza rotinas MPI, depende da implementação **MPI** existente no ambiente.

### Ambiente IBM/AIX com “Parallel Environment” instalado

A execução de um programa, com rotinas **MPI**, no ambiente IBM/AIX, é feita através de uma interface de operação, instalada através do “Parallel Environment”, e que configura o ambiente paralelo - POE, “*Parallel Operation Environment*”.

```
poe executável -procs número de processos -hostfile arquivo de máquinas
```

ou

```
export MP_PROCS=número de processos  
export MP_HOSTFILE=arquivo de máquinas
```

```
poe executável
```

### Ambiente LINUX com processador e compiladores da INTEL e biblioteca MPI proprietária

```
ulimit -v unlimited
```

```
mpirun -np número de processos ./executável
```

### Qualquer ambiente e compiladores, com biblioteca MPICH2

```
mpiexec executável -np número de processos -f arquivo de máquinas
```

## 6 – Implementação Básica de um Programa com MPI

Para um grande número de aplicações, um conjunto de apenas **6 subrotinas MPI** serão suficientes para desenvolver uma aplicação com MPI e executar em paralelo. Em C, os nomes das rotinas e parâmetros são “case sensitive”; em Fortran, “case insensitive”.

### 6.1 - Arquivo “headers”

Necessário para todos os programas e/ou subrotinas que efetuam chamadas para a biblioteca MPI. Normalmente é colocado no início do programa. Possui as definições básicas dos parâmetros das subrotinas e tipos de dados MPI.

**C**                    **#include “mpi.h”**

**FORTRAN**        **include “mpif.h”**

```
C

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

int main( argc, argv )
int argc;
char **argv;
{
    MPI_Init( &argc, &argv );
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```

```
Fortran

program mpisimple
    implicit none
    integer ierr
    include 'mpif.h'
    CALL MPI_INIT(ierr)
c    print message to screen
    write(6,*) 'Hello World!'
    CALL MPI_FINALIZE(ierr)
end
```

## 6.2 - Iniciar um processo MPI

### MPI\_INIT

- Primeira rotina MPI a ser utilizada;
- Define e inicializa o ambiente necessário para executar o MPI;
- Deve ser chamada antes de qualquer outra rotina 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 (MPI\_SUCCESS)  
**mpierr<0**, Erro

```
C

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

int main( argc, argv )
int argc;
char **argv;
{
    MPI_Init( &argc, &argv );
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```

```
Fortran

program mpisimple
    implicit none
    integer ierr
    include 'mpif.h'
    CALL MPI_INIT(ierr)
c    print message to screen
    write(6,*) 'Hello World!'
    CALL MPI_FINALIZE(ierr)
end
```

## 6.3 – Finalizar um processo MPI

### MPI\_FINALIZE

- Finaliza o processo MPI, eliminando qualquer sinalização MPI;
- Última rotina MPI a ser executada por uma aplicação MPI;
- **Importante:** Essa rotina só deve ser chamada após se ter certeza da finalização de qualquer comunicação entre os processos.

**C**                    **int MPI\_Finalize()**

**FORTRAN**            **CALL MPI\_FINALIZE (mpierr)**

**mpierr**              Variável inteira de retorno com o status da rotina.

**mpierr=0**, Sucesso (MPI\_SUCCESS)

**mpierr<0**, Erro

```
C

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

int main( argc, argv )
int argc;
char **argv;
{
    MPI_Init( &argc, &argv );
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```

```
Fortran

program mpisimple
    implicit none
    integer ierr
    include 'mpif.h'
    CALL MPI_INIT(ierr)
    c    print message to screen
        write(6,*) 'Hello World!'
        CALL MPI_FINALIZE(ierr)
end
```

## 6.4 - Identificar um processo MPI

### MPI\_COMM\_RANK

- Identifica o processo, dentro de um grupo de processos iniciados pelo “process manager”.
- 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.

**C**

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

int main( argc, argv )
int argc;
char **argv;
{
    int rank;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```

**Fortran**

```
program mpisimple
    implicit none
    integer ierr, rank
    include 'mpif.h'
    CALL MPI_INIT(ierr)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
    c print message to screen
    write(6,*) 'Hello World!'
    CALL MPI_FINALIZE(ierr)
end
```

## 6.5 – Informar o número de processos 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 inicializados durante uma aplicação MPI;

**mpierr**              Variável inteira de retorno com o status da rotina

**C**

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

int main( argc, argv )
int argc;
char **argv;
{
    int rank, nprocs;
    MPI_Init( &argc, &argv );
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf( "Hello world\n" );
    MPI_Finalize();
    return 0;
}
```

**Fortran**

```
program mpisimple
    implicit none
    integer ierr, rank, nprocs
    include 'mpif.h'
    CALL MPI_INIT(ierr)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
    c print message to screen
    write(6,*) 'Hello World!'
    call mpi_finalize(ierr)
end
```



## 6.6 - Enviar mensagens no MPI

### MPI\_SEND

- Rotina padrão de envio de mensagens, com interrupção da execução - "**Blocking send**".
- O processo só retorna a execução após o dado ter sido enviado.
- Após retorno, libera o "system buffer" e permite acesso ao "application buffer".

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

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

<b>sndbuf</b>	Endereço do dado que será enviado. Endereço do "application buffer";
<b>count</b>	Número de elementos contidos no endereço inicial, que serão enviados;
<b>datatype</b>	Tipo dos dados no endereço inicial;
<b>dest</b>	Identificação do processo destino que irá receber os dados;
<b>tag</b>	Rótulo de identificação da mensagem, somente numérico;
<b>comm</b>	"MPI communicator";
<b>mpierr</b>	Variável inteira de retorno com o status da rotina.

```
C
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
    char message[20];
    int i,rank, size, type=99;
    MPI Status status;

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

    if(rank == 0) {
        strcpy(message, "Hello, world");
        for (i=1; i<size; i++)
            MPI_Send(message, 13, MPI_CHAR, i,
                type, MPI_COMM_WORLD);
    }
    else
        ...
        ...
        ...

    printf( "Message from node =%d : %.13s\n",
            rank,message);
    MPI_Finalize();
}
```

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

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

tag = 100
if (me .eq. 0) then
    message = 'Hello, world'
    do i=1,nt-1
        CALL MPI_SEND(message,12,MPI_CHARACTER,i,tag, &
            MPI_COMM_WORLD,mpierr)
    enddo
    write(6,*) 'node',me,':',message
else
    ...
    ...
    write(6,*) 'node',me,':',inmsg
endif
call MPI_FINALIZE(mpierr)
end
```

## 6.7 - Receber mensagens no MPI

### MPI\_RECV

- Rotina padrão de recebimento de mensagens, com interrupção da execução - "**Blocking receive**".
- O processo só retorna a execução 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)**

<b>recvbuf</b>	Endereço aonde será armazenado o dado recebido. Endereço do "application buffer";
<b>count</b>	Número de elementos a serem recebidos, que serão armazenados;
<b>datatype</b>	Tipo do dado recebido;
<b>source</b>	Identificação do processo que enviou os dados, ou <b>MPI_ANY_SOURCE</b> ;
<b>tag</b>	Rótulo de identificação da mensagem, somente numérico, ou <b>MPI_ANY_TAG</b> ;
<b>comm</b>	MPI communicator;
<b>status</b>	Vetor de três elementos, com informações de <b>source, tag e erro</b> ;
<b>mpierr</b>	Variável inteira de retorno com o status da rotina.

**C**

```
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
    char message[20];
    int i,rank, size, type=99;
    MPI_Status status;

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

    if(rank == 0) {
        strcpy(message, "Hello, world");
        for (i=1; i<size; i++)
            MPI_Send(message, 13, MPI_CHAR, i,
                    type, MPI_COMM_WORLD);
    }
    else
        MPI_Recv(message, 13, MPI_CHAR, 0,
                type, MPI_COMM_WORLD, &status);

    printf( "Message from node =%d : %.13s\n",
            rank,message);
    MPI_Finalize();
}
```

**Fortran**

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

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

tag = 100
if (me .eq. 0) then
    message = 'Hello, world'
    do i=1,nt-1
        CALL MPI_SEND(message,12,MPI_CHARACTER,i,tag, &
            MPI_COMM_WORLD,mpierr)
    enddo
    write(6,*) 'node',me,':',message
    else
        CALL MPI_RECV(inmsg,12,MPI_CHARACTER,0,tag, &
            MPI_COMM_WORLD,status,mpierr)
        write(6,*) 'node',me,':',inmsg
    endif
    call MPI_FINALIZE(mpierr)
end
```



# LABORATÓRIO 1 - Rotinas Básicas do MPI

## Exercício 1

- 1 - Caminhe para a pasta do primeiro exercício do laboratório.

```
cd ./lab01/ex1
```

- 2 - Compile o programa **hello.f** ou **hello.c** de acordo com a instalação MPI do ambiente.

```
IBM      : mpixlf hello.f -o hello      ou      mpicc hello.c -o hello
```

```
INTEL    : ifort hello.f -o hello -lmpi  ou      icc hello.c -o hello -lmpi
```

```
MPICH2: mpif77 hello.f -o hello      ou      mpicc hello.c -o hello
```

- 3 - Caso utilize MPICH2, crie um arquivo com uma configuração de máquinas existentes no ambiente.

OBS: Se a máquina ou nó possuir *n* “cores”, repita o nome dessa máquina *n* vezes

Ex.: Ambiente com uma duas máquinas “quad-core”: maq01 e maq02

vi maquinas (Outros editores: **pico** ou **nano**)

```
maq01
maq01
maq01
maq01
maq02
maq02
maq02
maq02
```

- 4 - Execute o programa várias vezes, alterando o número de processos.

```
IBM      : poe ./hello -procs n -hostfile ./arquivo
```

```
INTEL    : ulimit -v unlimited
```

```
mpirun -np n ./hello
```

```
MPICH2: mpiexec ./hello -np n -f ./arquivo
```

*n* = número de processos

*arquivo* = nome do arquivo de máquinas

## Exercício 2

Os programas apresentados nos exercícios dos laboratórios seguem o modelo **SPMD** (“**Single Program**” – “**Multiple Data**”), ou seja, o mesmo código, executa como um processo pai, enviando os dados e coordenando os processos, e como processo filho, recebendo os dados, processando tarefas e devolvendo os resultados ao processo pai.

No caso do próximo exercício, o processo pai envia uma mensagem (“Hello world”) para todos os processos filhos e imprime a sua mensagem na “saída padrão”. Os processos filhos, recebem a mensagem e, também imprimem-na, na “saída padrão”.

- 1 - Caminhe para a pasta com o segundo exercício do laboratório.

```
cd ./lab01/ex2
```

O exercício está em alterar o programa, de maneira que, cada processo filho receba a mensagem enviada pelo processo pai, mas não faz nada com ela, e envia a sua identificação MPI para o processo pai. O processo pai recebe a identificação do processo filho e imprime essa identificação junto com a mensagem “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.**

```
vi hello.ex1.f      ou      vi hello.ex1.c      (Outros editores: pico ou nano)
```

- 3 - Compile o programa:

```
IBM      : mpixlf hello.ex1.f -o hello2      ou      mpicc hello.ex1.c -o hello2
```

```
INTEL    : ifort hello.ex1.f -o hello2 -lmpi  ou      icc hello.ex1.c -o hello2 -lmpi
```

```
MPICH2: mpif77 hello.ex1.f -o hello2      ou      mpicc hello.ex1.c -o hello2
```

- 4 - Execute o programa:

```
IBM      : poe ./hello2 -procs n -hostfile ./arquivo
```

```
INTEL    : mpirun -np n ./hello2
```

```
MPICH2: mpiexec ./hello2 -np n -f ./arquivo
```

*n* = número de processos

*arquivo* = nome do arquivo de máquinas

- 5 - Altere novamente o programa e na rotina de “receive” do processo pai, substitua o parâmetro que identifica a origem da mensagem, pela palavra reservada: **MPI\_ANY\_SOURCE**.

- 6 - Compile e execute novamente. Percebeu alguma diferença no padrão da resposta?

### Exercício 3

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

```
cd ./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 pai envie duas mensagens ("Hello" e "World") para cada processo filho, utilizando-se de duas **tags** diferentes. Os processos filhos deverão receber as mensagens na ordem invertida e irão imprimir o resultado na “saída padrão”.

```
vi hello.ex2.f      ou      vi hello.ex2.c      (Outros editores: pico ou nano)
```

- 3 - Compile o programa:

```
IBM      : mpixlf hello.ex2.f -o hello3      ou      mpicc hello.ex2.c -o hello3
```

```
INTEL    : ifort hello.ex2.f -o hello3 -lmpi ou      icc hello.ex2.c -o hello3 -lmpi
```

```
MPICH2: mpif77 hello.ex2.f -o hello3      ou      mpicc hello.ex2.c -o hello3
```

- 4 - Execute o programa:

```
IBM      : poe ./hello3 -procs n -hostfile /arquivo
```

```
INTEL    : mpirun -np n ./hello3
```

```
MPICH2: mpiexec ./hello3 -np n -f /arquivo
```

*n* = número de processos

*arquivo* = nome do arquivo de máquinas

## Exercício 4

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

```
cd ./lab01/ex4
```

Neste exercício, o processo pai inicia um vetor de 60.000 elementos, e distribui partes desse vetor para vários processos filhos. Cada processo filho recebe a sua parte do vetor principal, efetua um cálculo 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.**

```
vi mpi.ex1.f          ou          vi mpi.ex1.c      (Outros editores: pico ou nano)
```

- 3 - Compile o programa:

```
IBM      : mpixlf mpi.ex1.f -o vetor          ou          mpicc mpi.ex1.c -o vetor
```

```
INTEL    : ifort mpi.ex1.f -o vetor -lmpi     ou          icc mpi.ex1.c -o vetor -lmpi
```

```
MPICH2: mpif77 mpi.ex1.f -o vetor          ou          mpicc mpi.ex1.c -o vetor
```

- 4 - Execute o programa:

```
IBM      : poe ./vetor -procs n -hostfile ./arquivo
```

```
INTEL    : mpirun -np n ./vetor
```

```
MPICH2: mpiexec ./vetor -np n -f ./arquivo
```

*n* = número de processos

*arquivo* = nome do arquivo de máquinas

**OBS:** O número de processos filhos deve ser um múltiplo do tamanho do vetor.

## Exercício 5

O objetivo desse exercício é demonstrar a execução paralela de vários códigos diferentes e comunicando-se entre si, ou seja, vamos trabalhar com o modelo MPMD ("Multiple Program" – "Multiple Data"). Um código irá executar como um processo pai, enviando os dados e coordenando os processos, e um outro código executará como processo filho, recebendo os dados, processando tarefas e devolvendo os resultados ao processo pai.

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

```
cd ./lab01/ex5
```

- 2 - Vamos trabalhar com o código da solução do exercício 4 que está pronto para ser dividido em dois programas. Faça duas cópias do programa em C ou Fortran.

```
cp mpi.ex1.c pai.c ou cp mpi.ex1.f pai.f  
cp mpi.ex1.c filho.c ou cp mpi.ex1.f filho.f
```

- 3 - Edite separadamente cada arquivo, elimine ou mantenha as linhas referentes a execução como processo pai e a execução como processo filho. A parte inicial do programa mpi.ex1, será comum aos programas pai e filho, mas no programa filho, será necessário acrescentar a rotina de finalização do MPI.

```
vi pai.f ou vi pai.c (Outros editores: pico ou nano)  
vi filho.f ou vi filho.c
```

- 4 - Compile cada um dos programas:

```
IBM : mpixlf pai.f -o pai ou mpicc pai.c -o pai  
mpixlf filho.f -o filho ou mpicc filho.c -o filho
```

```
INTEL : ifort pai.f -o pai -lmpi ou icc pai.c -o pai -lmpi  
ifort filho.f -o filho -lmpi ou icc filho.c -o filho -lmpi
```

```
MPICH2: mpif77 pai.f -o pai ou mpicc pai.c -o pai  
mpif77 filho.f -o filho ou mpicc filho.c -o filho
```



5 - Execute o programa:

IBM : É necessário definir um arquivo de execução dos processos. Veja o exemplo abaixo, para a execução de 1 processo **pai** e 6 processos **filho**.

vi executa (Outros editores: **pico** ou **nano**)

```
./pai  
./filho  
./filho  
./filho  
./filho  
./filho  
./filho
```

```
poe -procs 7 -pgmmodel mpmd -cmdfile ./executa -hostfile ./arquivo
```

As opções:	<b>-pgmmodel</b>	Define o modelo de execução (spmd ou mpmd);
	<b>-cmdfile</b>	Indica o arquivo com o nome dos executáveis;
	<b>-hostfile</b>	Indica o arquivo com o nome das máquinas;
	<b>-procs</b>	Indica o número de processos. Tem que está de acordo com o número de executáveis indicado na opção <b>cmdfile</b> .

```
INTEL : mpirun -np n ./pai : -np n ./filho
```

```
MPICH2: mpiexec -f ./arquivo -np n ./pai : -np n ./filho
```

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

**OBS:** O número de processos filhos deve ser um múltiplo do tamanho do vetor.

## Exercício 6

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

```
cd ./lab01/ex6
```

Este programa calcula o valor de **PI**, através de um algoritmo que soluciona uma integral. A idéia do exercício é de perceber erros de lógica na execução de um programa. O programa não está funcionando corretamente; **corrija dois erros** bem simples, um de cálculo e outro de finalização dos resultados.

- 2 - Compile o programa com os erros:

```
IBM      : mpixlf karp.mpi.f -o pi      ou      mpicc karp.mpi.c -o pi
```

```
INTEL    : ifort karp.mpi.f -o pi -lmpi ou      icc karp.mpi.c -o pi -lmpi
```

```
MPICH2: mpif77 karp.mpi.f -o pi      ou      mpicc karp.mpi.c -o pi
```

- 3 - Execute o programa e verifique os erros:

```
IBM      : poe ./pi -procs n -hostfile ./arquivo
```

```
INTEL    : mpirun -np n ./pi
```

```
MPICH2: mpiexec ./pi -np n -f ./arquivo
```

*n = número de processos*

*arquivo = nome do arquivo de máquinas*

- 4 - Edite o programa e corrija os erros:

```
vi karp.mpi.f      ou      vi karp.mpi.c      (Outros editores: pico ou nano)
```

- 5 - Compile e execute novamente o programa para verificar se os problemas foram corrigidos.

## 7 – Métodos de Comunicação "Point-To-Point"

- Numa comunicação "Point-to-Point", um processo envia uma mensagem e um segundo processo, recebe.
- Existem várias opções de programação utilizando-se comunicação "Point-to-Point". Opções que incluem:

Quatro métodos de comunicação: **synchronous, ready, buffered, e standard**

Duas formas de processamento: **blocking e non-blocking**.

- Existem quatro rotinas "blocking send" e quatro rotinas "non-blocking send", correspondentes aos quatro métodos de comunicação.
- A rotina de "receive" não especifica o método de comunicação. Simplesmente, ou a rotina é "blocking" ou, "non-blocking".

## 7.1 - "Blocking Synchronous Send"

C	<code>int MPI_Ssend ( *buf, count, datatype, dest, tag, comm )</code>
FORTTRAN	<code>CALL MPI_SSEND ( buf, count, datatype, dest, tag, comm, ierror )</code>

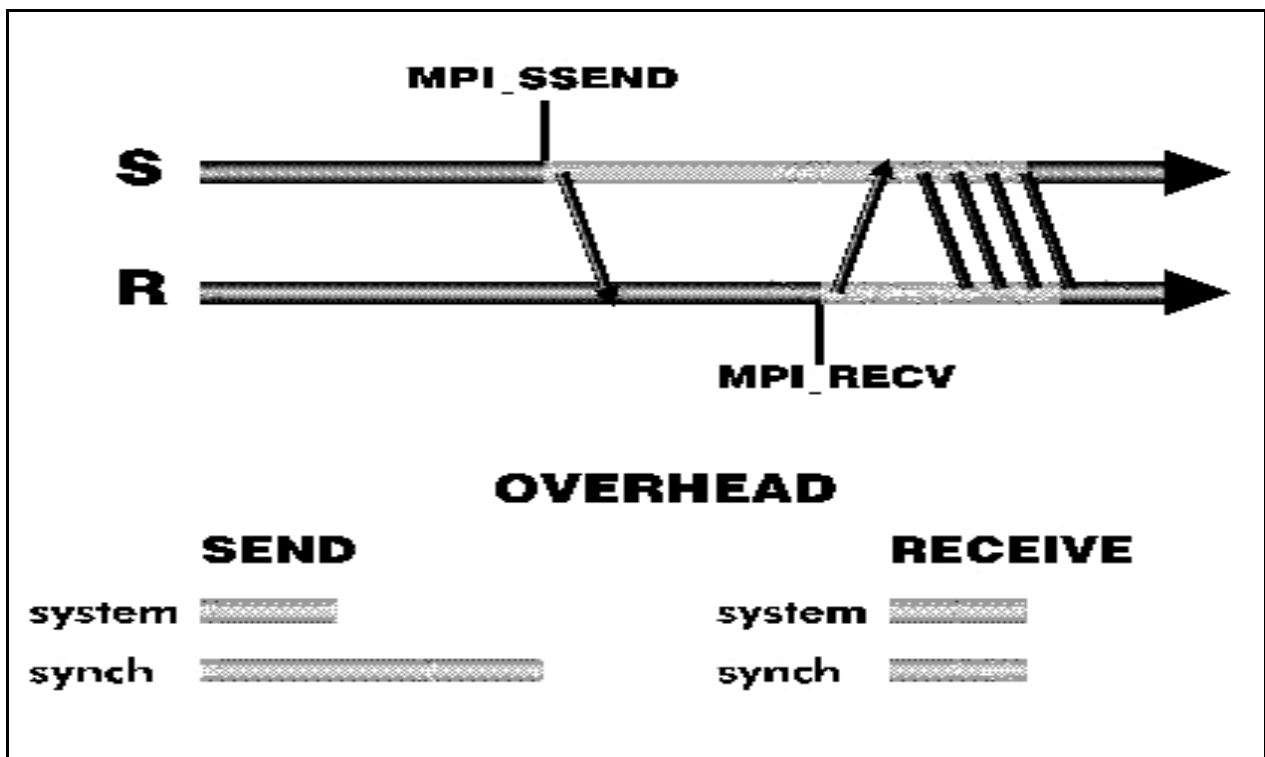
Quando um `MPI_Ssend` é executado, o processo que irá enviar a mensagem, primeiro, envia um sinal ao processo que recebe para verificar se já existe um "receive" ativo, e se não existir, **espera** até que seja executado um "receive", para então, iniciar transferência dos dados.

### Overheads

- System overhead** Ocorre devido a cópia da mensagem do "send buffer" para a rede e da rede para o "receive buffer". Depende do tamanho da mensagem.
- Synchronization overhead** Ocorre devido ao tempo de espera de um dos processos pelo sinal de OK de outro processo.

**OBS:** Neste modo, o "Synchronization overhead", pode ser significativa.

### Blocking Synchronous Send e Blocking Receive



## 7.2 - "Blocking Ready Send"

C	<code>int MPI_Rsend (*buf, count, datatype, dest, tag, comm)</code>
FORTTRAN	<code>CALL MPI_RSEND (buf, count, datatype, dest, tag, comm, ierror)</code>

Quando um **MPI\_Rsend** é executado a mensagem é enviada imediatamente, sem que haja qualquer controle no envio, ou qualquer controle no recebimento – as ações internas do MPI de “handshake”. É essencial que já exista um “receive” ativo no processo que irá receber, do contrário as consequências são imprevisíveis.

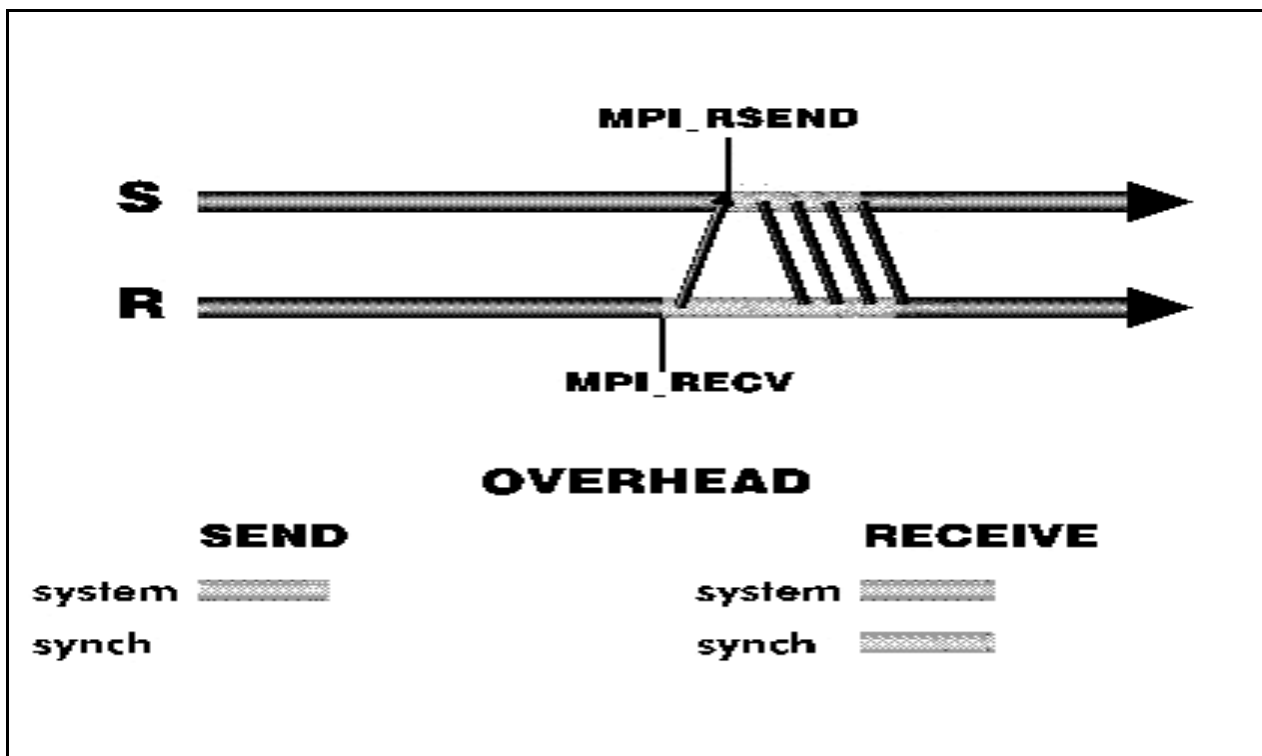
### Overheads

**System overhead** Tempo padrão. Depende do tamanho da mensagem.

**Synchronization overhead** É mínimo ou nada no processo que envia, mas no processo que recebe, pode ser muito alto, dependendo do momento que é executada a rotina de “receive”.

**OBS: Atenção**, este modo somente deverá ser utilizado se o programador tiver certeza que uma rotina de “receive”, em qualquer processo, tenha sido executada, antes do processo que envia executar um **MPI\_Rsend**.

### Blocking Ready Send e Blocking Receive



## 7.3 - "Blocking Buffered Send"

C	<code>int MPI_Bsend (*buf, count, datatype, dest, tag, comm)</code>
FORTTRAN	<code>CALL MPI_BSEND (buf, count, datatype, dest, tag, comm, ierror)</code>

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

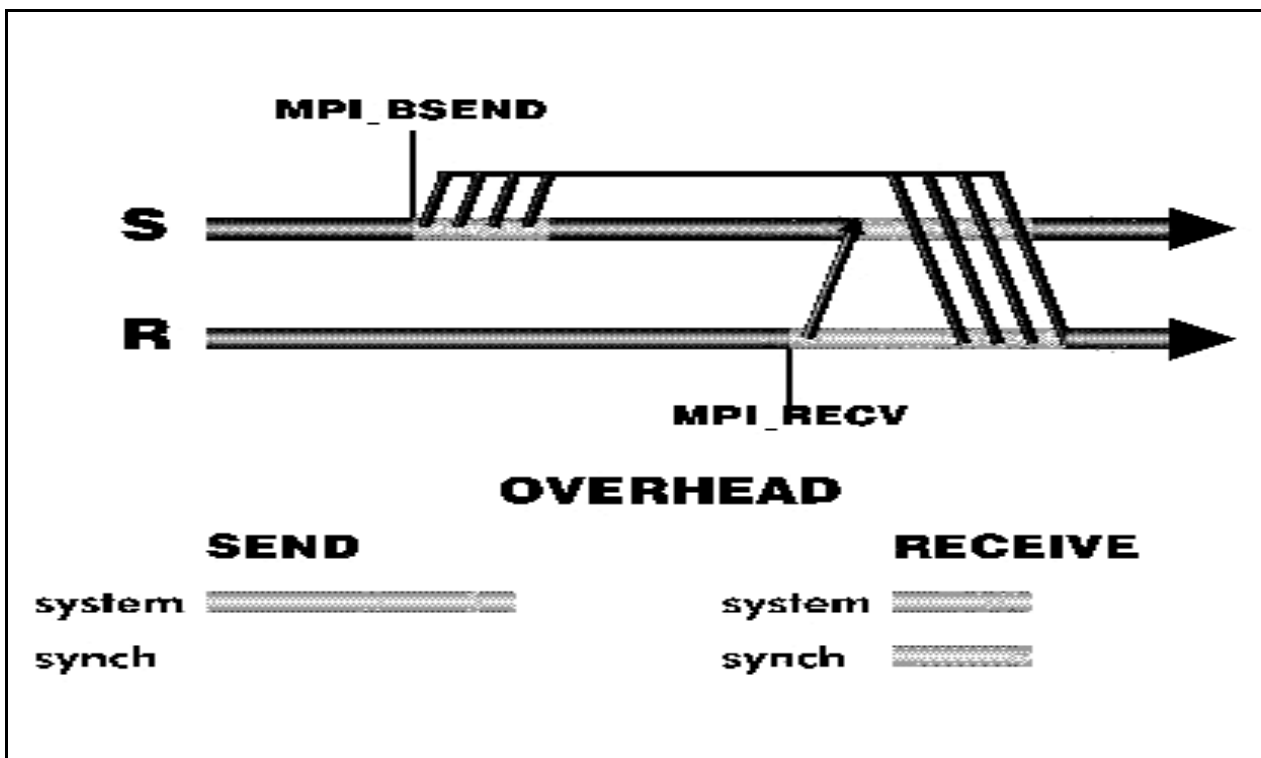
### Overheads

**System overhead** Aumenta em relação ao padrão, pois ocorrem duas transferências de dados, uma devido a cópia da mensagem do "Application buffer" para o "buffer" definido pelo usuário e a outra para o processo que irá receber os dados.

**Synchronization overhead** Não existe no processo que envia, e no processo que recebe. irá depender do momento da execução do "receive".

**OBS: Atenção**, este método, só utiliza o "buffer" definido pela rotina **MPI\_Buffer\_attach**, que deverá ser utilizada antes da execução de um **MPI\_Bsend**.

### Blocking Buffered Send e Blocking Receive



## 7.4 - "Blocking Standard Send"

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

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

O método padrão de envio de mensagens, trabalha utilizando "buffers" do sistema. O tamanho desses "buffers" depende da implementação MPI utilizada, e que pode variar entre 4K, 8K, 16K, 32K ou 64K bytes de memória, e também, de acordo com o número de processos iniciados. Na implementação inicial e básica do MPI, o "buffer" definido é de 4K, portanto, vamos analisar o envio de mensagens com esse padrão de "buffer".

### Message <= 4K

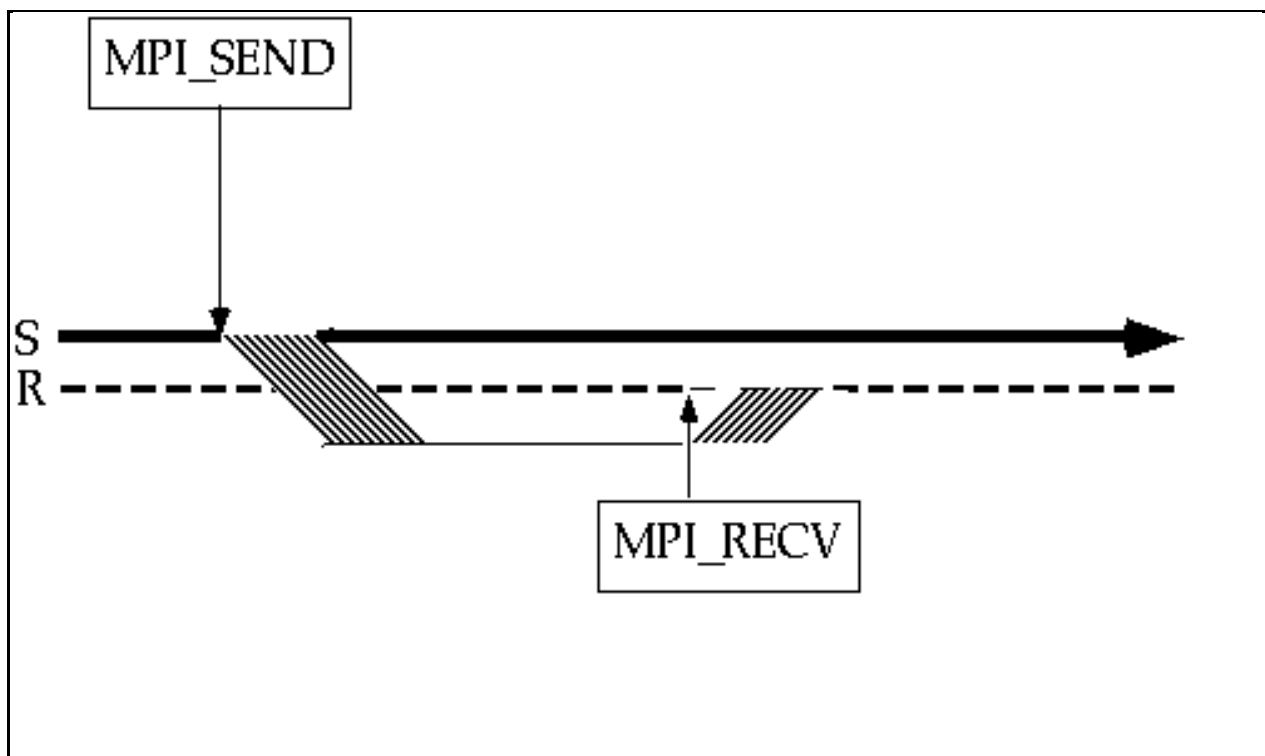
Quando `MPI_Send` é executado, a mensagem é imediatamente transmitida para o processo que irá receber a mensagem, e fica armazenada no "buffer" desse processo até que seja executado um "receive".

### Overheads

**System Overhead** Tempo padrão. Depende do tamanho da mensagem.

**Synchronization overhead** Mínimo, pois não existe a necessidade de sincronização dos processos.

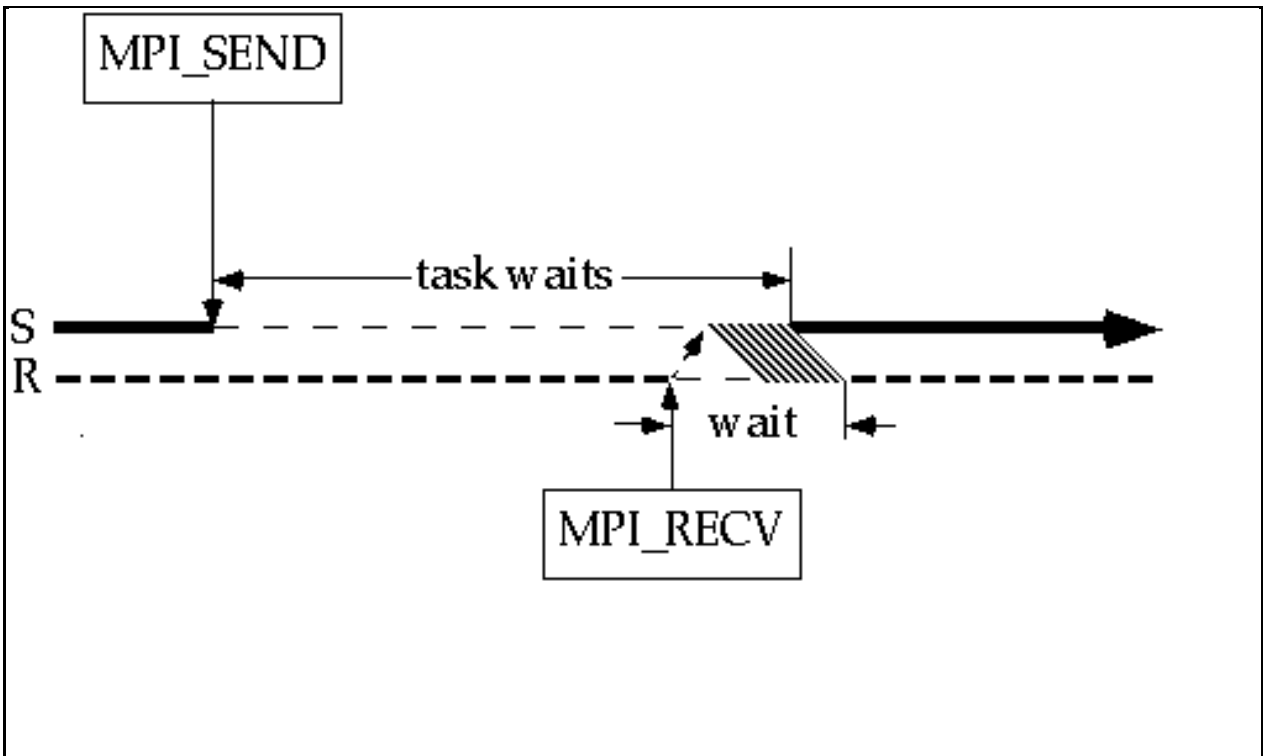
### Blocking Standard Send e Blocking Receive Message <= 4K



## Message > 4K

Quando `MPI_Send` é executado e a mensagem que será enviada é maior que o “buffer” do sistema, o método síncrono é utilizado, pois não é possível garantir o espaço do receive “buffer”. É necessário aguardar a execução de um “receive” para que a mensagem seja transmitida.

### Blocking Standard Send e Blocking Receive Message > 4K





## 7.5 – Rotina MPI\_Buffer\_attach

<b>C</b>	<b>int MPI_Buffer_attach (*buf, size)</b>
<b>FORTTRAN</b>	<b>CALL MPI_BUFFER_ATTACH (buf, size, ierror )</b>

**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.

Rotina MPI que é utilizada em conjunto com a rotina “buffered send” (MPI\_Bsend). Define para o MPI, um "buffer" de tamanho específico, para ser utilizado no envio de mensagens. Somente a rotina “buffered send” poderá utilizar esse endereço definido.

**OBS:** Somente um "buffer" poder ser definido por processo, durante a execução da aplicação. Se for necessário definir um novo “buffer”, deve-se, primeiro, eliminar o “buffer” anterior.

## 7.6 – Rotina MPI\_Buffer\_detach

<b>C</b>	<b>int MPI_Buffer_detach (*buffer, *size )</b>
<b>FORTTRAN</b>	<b>CALL MPI_BUFFER_DETACH (buf, size, ierror)</b>

**buf**            Variável que identifica o endereço do "buffer", que será eliminado;

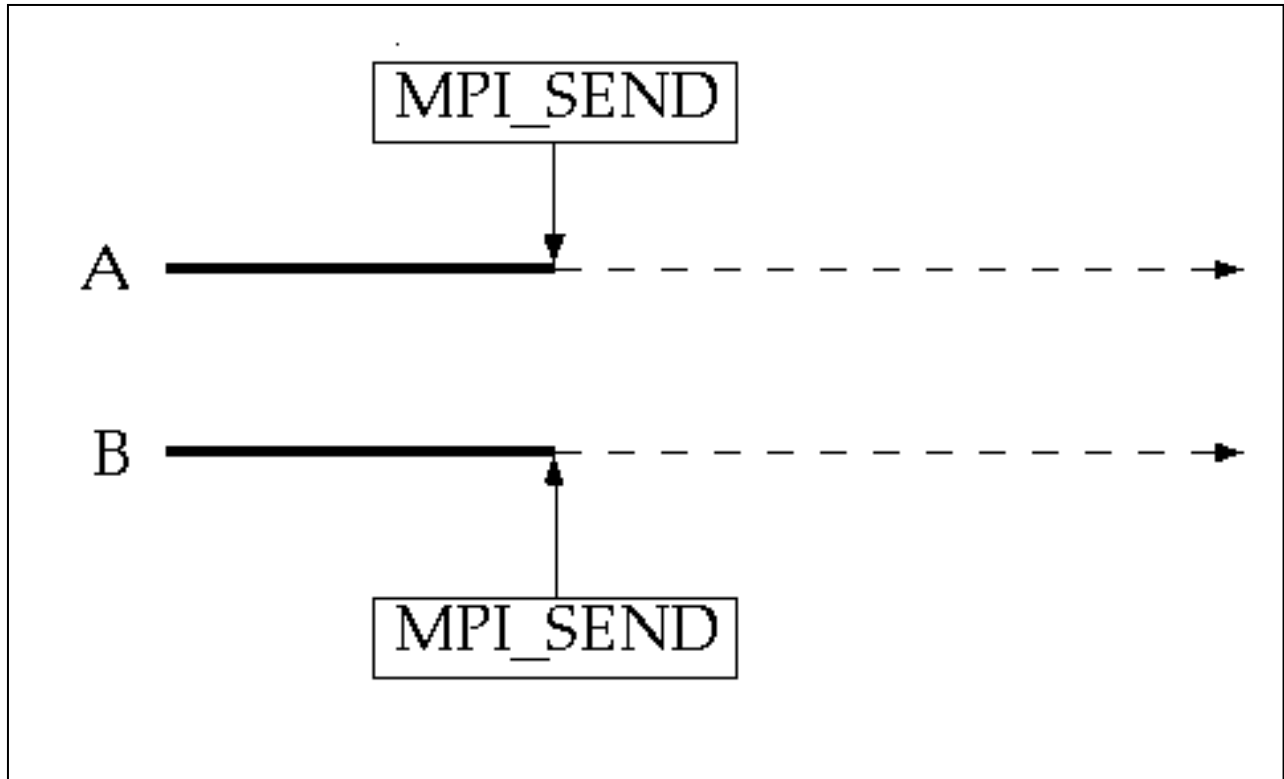
**size**            Variável inteira de retorno com informação do tamanho do "buffer" (em número de bytes) que foi eliminado;

**ierror**          Variável inteira com status da execução da rotina.

Elimina o "buffer" que foi iniciado anteriormente para o MPI. Isso as vezes é necessário para liberar espaço de memória utilizada pelo processo. Esta operação bloqueia a execução do programa até que a transmissão da mensagem tenha sido terminado.

## 7.7 - "Deadlock"

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



- Organize os eventos de envio e recebimento de mensagens na sua aplicação, de maneira que, os eventos de envio de mensagens casem com os eventos de recebimento antes de enviar novas mensagens;

- Utilize as rotinas "non-blocking communication".

## 7.8 – Rotinas de Comunicação "Non-blocking"

- Em uma comunicação "**blocking**", a execução do programa é suspensa até o "buffer" de envio ou recebimento de mensagens estar liberado para uso novamente. Quando se executa um "**blocking send**", a execução do processo para, e dependendo do método utilizado, aguarda por determinados eventos acontecerem:

**Synchronous**     Aguarde sinal de receive do processo que irá receber a mensagem;

**Ready**             Aguarda a liberação do "Application buffer", após envio dos dados para a rede;

**Buffered**          Aguarda a liberação do "Application buffer", após a cópia dos dados para o novo "buffer";

**Standard**         Aguarda a liberação do "Application buffer", após a cópia dos dados para o "receive buffer".

- Em uma comunicação "**non-blocking**", a execução do programa continua imediatamente após ter sido iniciada a comunicação; não existe a espera por qualquer evento, e o programador não tem idéia se a mensagem já foi enviada ou recebida. Em uma comunicação "**non-blocking**", será necessário bloquear a continuação da execução do programa, ou averiguar o status do "system buffer", antes de reutilizá-lo.

- Existem algumas rotinas MPI, específicas para verificar o status do "buffer" de envio e recebimento de mensagens:

**MPI\_Wait**

**MPI\_Test**

- O nome das subrotinas "**non-blocking**", iniciam com a letra **I**, (**MPI\_Ixxxx**), e possuem mais um parâmetro de retorno, com a informação da situação do "buffer".

- Parâmetro de retorno "**request**"; variável inteira utilizada nas rotinas "non-blocking", possui ação transparente, ou seja, é apenas utilizada como parâmetro de ligação entre as rotinas de envio e recebimento de mensagens, com as rotinas de controle do status do "buffer", **MPI\_Wait**, **MPI\_Test**.

### **7.8.1 - Rotina MPI\_Issend - 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)`

### **7.8.2 - Rotina MPI\_Irsend - 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)`

### **7.8.3 - Rotina MPI\_Ibsend - 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)`

### **7.8.4 - Rotina MPI\_Isend - Non-Blocking Standard Send**

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

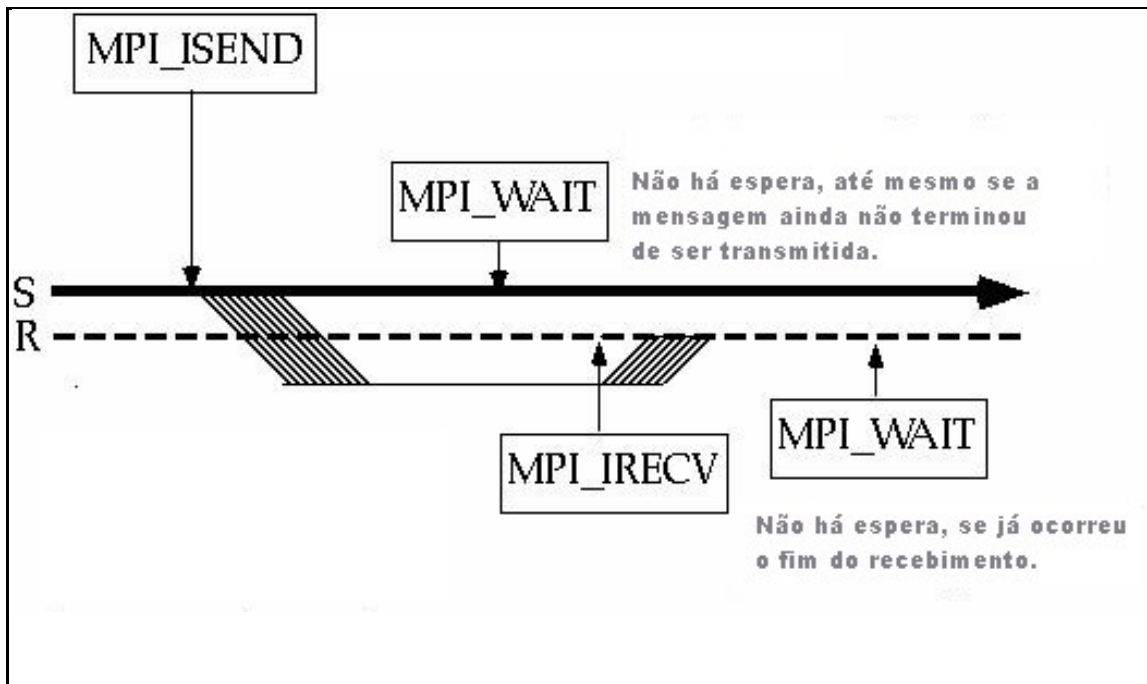
**FORTRAN**        `CALL MPI_ISEND(buf, count, datatype, dest, tag, comm, request, ierror)`

### **7.8.5 - Rotina MPI\_Irecv - Non-Blocking Receive**

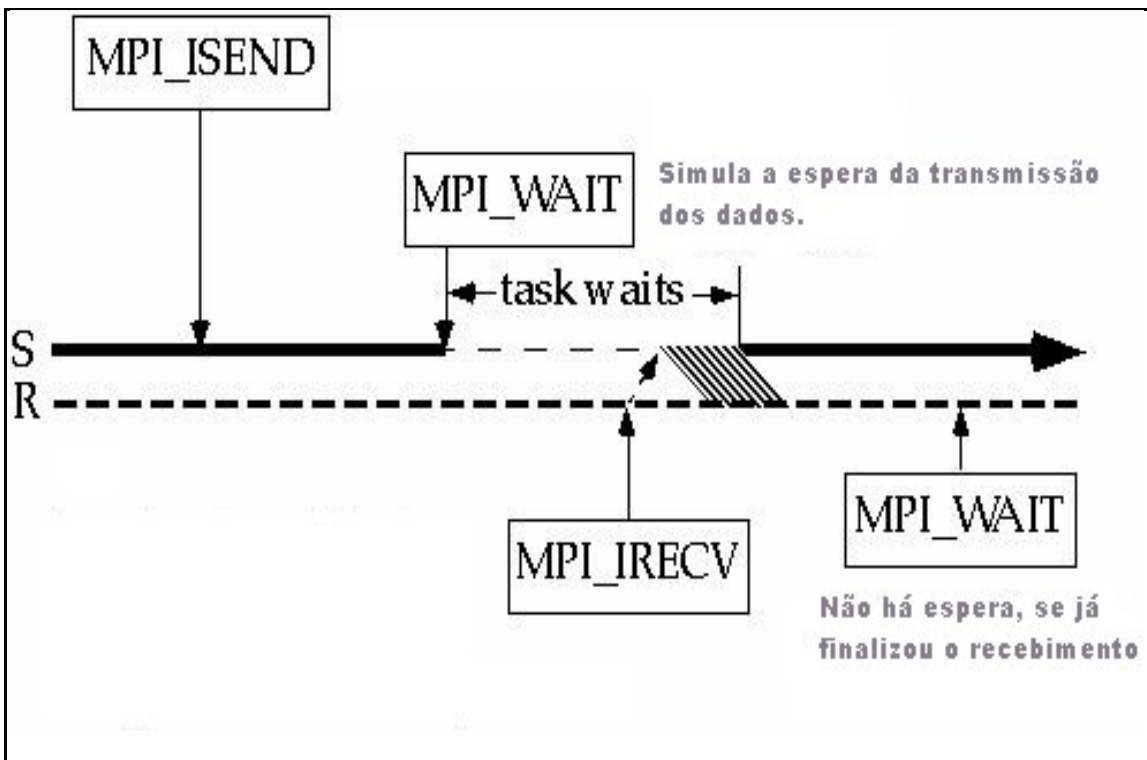
**C**                    `int MPI_Irecv(*buf, count, datatype, source, tag, comm, *request)`

**FORTRAN**        `CALL MPI_IRECV(buf, count, datatype, source, tag, comm, request, ierror)`

**Non-Blocking Standard Send e Non-Blocking Receive  
Message  $\leq 4K$**



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



## 7.9 – Rotinas Auxiliares as Rotinas “Non-blocking”

### 7.9.1 - Rotina MPI\_Wait

**C**                    **int MPI\_Wait (\*request, \*status)**

**FORTRAN**        **CALL MPI\_WAIT (request, status, ierror)**

**request**            Variável inteira “transparente”, que armazena a informação sobre a transferência dos dados pelo processo. Essa informação é fornecida pelas rotinas "non-blocking send" ;

**status**             Vetor com informações da mensagem, enviada ou recebida;

**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** ( de ativo para inativo ):

**active**             “buffer” ativo. Transferência de dados em andamento;

**inactive ou null**    “buffer” inativo. Transferência de dados finalizada.

“blocking” → “deadlock”

```
If( rank == 0 ) Then
  Call MPI_send(buffer1, 1, MPI_integer, 1, 10, &
    MPI_comm_world, error )
  Call MPI_recv(buffer2, 1, MPI_integer, 1, 20, &
    MPI_comm_world, status, error )
Else If( rank == 1 ) Then
  Call MPI_send(buffer2, 1, MPI_integer, 0, 20, &
    MPI_comm_world, error )
  Call MPI_recv(buffer1, 1, MPI_integer, 0, 10, &
    MPI_comm_world, status, error )
End If
```

“non-blocking” → “no deadlock”

```
If( rank == 0 ) Then
  Call MPI_Isend(buffer1, 1, MPI_integer, 1, 10, &
    MPI_comm_world, REQUEST, error )
  Call MPI_Irecv(buffer2, 1, MPI_integer, 1, 20, &
    MPI_comm_world, REQUEST, error )
Else If( rank == 1 ) Then
  Call MPI_Isend(buffer2, 1, MPI_integer, 0, 20, &
    MPI_comm_world, REQUEST, error )
  Call MPI_Irecv(buffer1, 1, MPI_integer, 0, 10, &
    MPI_comm_world, REQUEST, error )
End If
Call MPI_wait( REQUEST, status ) ! ???
Call MPI_finalize( error )
```

## 7.9.2 - Rotina MPI\_Test

**C**                    **int MPI\_Test( \*request, \*flag, \*status )**

**FORTRAN**        **CALL MPI\_TEST( request, flag, status, ierror )**

**request**            Variável inteira “transparente”, que armazena a informação sobre a transferência dos dados pelo processo. Essa informação é fornecida pelas rotinas "non-blocking send" ;

**flag**                Variável lógica que identifica o valor de **request**:

“buffer” inativo    **MPI\_REQUEST\_NULL**                    **flag=true**  
“buffer” ativo     **MPI\_REQUEST\_ACTIVE**                **flag=false**

**status**             Vetor com informações da mensagem;

**ierror**             Variável inteira com o status da execução da rotina.

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

## 7.10 – Conclusões

- O modo "**Synchronous**" é o mais seguro e ao mesmo tempo, o mais portátil (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" no processo destino, deve preceder a execução de um "send" no processo de origem.
- O modo "**Buffered**" permite o controle no tamanho do "buffer".
- O modo "**Standard**" é a implementação básica do MPI. Possui excelente performance.
- 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", no geral.
- As rotinas "**Non-blocking**" necessitam de um maior controle, que pode ser feito por rotinas auxiliares.



# LABORATÓRIO 2 - Comunicação Point-to-Point

## Exercício 1

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

```
cd ./lab02/ex1
```

Este exercício tem como objetivo, demonstrar e comparar a performance entre os quatro métodos de comunicação “blocking send”, a partir da transmissão de um vetor de dados, que pode simular o tamanho de “buffers”, a partir de 4Kbytes.

**OBS:** O programa não mede o tempo de comunicação completo e nem o total do "system overhead", apenas o tempo de transferência de dados das rotinas de “send”.

- 2 - Inicialmente, o programa define um vetor de dados, que utiliza 4Kbytes de memória.

C mlen=1024 (elementos do tipo integer=4 bytes, total de 4Kbytes)

Fortran imlen=1024 (elementos do tipo integer=4 bytes, total de 4Kbytes)

- 3 - Compile o programa

**OBS:** A versão em Fortran depende do compilador instalado e de suas rotinas medição de tempo. No exercício existe um programa alternativo, caso o original não funcione.

```
IBM : mpixf blocksends.f -o bks ou mpcc blocksends.c -o bks
```

```
INTEL : ifort blocksends.f -o bks -lmpi ou icc blocksends.c -o bks -lmpi
```

```
MPICH2 : mpif77 blocksends.f -o bks ou mpicc blocksends.c -o bks
```

- 4 - Execute o programa iniciando apenas **dois processos** para que seja possível realizar a medição de tempos. Execute pelo menos três vezes para se ter uma média dos tempos;

```
IBM : poe ./bks -procs 2 -hostfile ./arquivo
```

```
INTEL : ulimit -v unlimited
```

```
mpirun -np 2 ./bks
```

```
MPICH2 : mpiexec ./bks -np 2 -f ./arquivo
```

*arquivo = nome do arquivo de máquinas*

- 5 - Edite o programa e altere o número de elementos do vetor, principalmente em relação aos possíveis limite de “buffers” do ambiente:

```
vi blocksends.f ou vi blocksends.c (Outros editores: pico ou nano)
```

1024 (<=4K) e 1025 (>4K)	2048 (<=8K) e 2049 (> 8K)	4096 (<=16K) e 4097 (> 16K)
8192 (<=32K) e 8193 (>32K)	12288 (<=48K) e 16289 (>48K)	16384 (<=64K) e 16385 (>64K)

- 6 - Compile e execute novamente o programa. OBS: Normalmente o “send standard” possui excelente performance, mas quando o tamanho do “buffer” é ultrapassado, a sua performance piora bastante.

## Exercício 2

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

```
cd ./mpi/lab02/ex2
```

Este programa tem por objetivo demonstrar a utilização das rotinas "non-blocking" para evitar "deadlock".

- 2 - Compile o programa:

```
IBM      : mpxlf deadlock.f -o dlock      ou      mpcc deadlock.c -o dlock
```

```
INTEL    : ifort deadlock.f -o dlock -lmpi  ou      icc deadlock.c -o dlock -lmpi
```

```
MPICH2   : mpif77 deadlock.f -o dlock      ou      mpicc deadlock.c -o dlock
```

- 4 - Execute o programa, iniciando apenas **dois processos**.

```
IBM      : poe ./dlock -procs 2 -hostfile ./arquivo
```

```
INTEL    : mpirun ./dlock -np 2 ./dlock
```

```
MPICH2   : mpiexec ./dlock -np 2 -f ./arquivo
```

*arquivo = nome do arquivo de máquinas*

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

- 5 - Edite o programa e corrija o problema. **O que será necessário alterar ???**

```
vi deadlock.f    ou    vi deadlock.c      (Outros editores: pico ou nano)
```

- 6 - Compile e execute novamente para verificar se o problema foi corrigido.

### Exercício 3

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

```
cd ./lab02/ex3
```

Este programa tenta demonstrar o tempo perdido em "synchronization overhead" para mensagens maiores de 4Kbytes. A rotina externa (*sleep*), utilizada nos programas, serve para simular tempo de processamento.

- 2 - Compile primeiro a rotina *sleep*, criando apenas o código objeto ( opção **-c**):

```
IBM : xlc -c new_sleep.c
```

```
INTEL: icc -c new_sleep.c
```

```
GNU : gcc -c new_sleep.c
```

- 3 - Compile o programa principal, adicionando o objeto da rotina *sleep*:

```
IBM : mpxlf brecv.f new_sleep.o -o brecv ou mpcc brecv.c new_sleep.o -o brecv
```

```
INTEL : ifort brecv.f new_sleep.o -o brecv -lmpi ou icc brecv.c new_sleep.o -o brecv -lmpi
```

```
MPICH2: mpif77 brecv.f new_sleep.o -o brecv ou mpicc brecv.c new_sleep.o -o brecv
```

- 4 - Execute o programa iniciando apenas **dois processos**. Execute pelo menos três vezes e anote os tempos de execução.

```
IBM : poe ./brecv -procs 2 -hostfile ./arquivo
```

```
INTEL : mpirun ./brecv -np 2 ./brecv
```

```
MPICH2 : mpiexec ./brecv -np 2 -f ./arquivo
```

*arquivo = nome do arquivo de máquinas*

## Exercício 4

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

```
cd ./lab02/ex4
```

- 2 - Edite o programa do exercício anterior:

```
vi brevc.f      ou   vi brevc.c      (Outros editores: pico ou nano)
```

- 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 o recebimento.

- 2 - Compile primeiro a rotina *sleep*, criando apenas o código objeto ( opção **-c** ):

```
IBM   : xlc -c new_sleep.c
```

```
INTEL: icc -c new_sleep.c
```

```
GNU   : gcc -c new_sleep.c
```

- 3 - Compile o programa principal, adicionando o objeto da rotina *sleep*:

```
IBM   : mpixlf brevc.f new_sleep.o -o brevc  ou   mpicc brevc.c new_sleep.o -o brevc
```

```
INTEL : ifort brevc.f new_sleep.o -o brevc -lmpi  ou  icc brevc.c new_sleep.o -o brevc -lmpi
```

```
MPICH2: mpif77 brevc.f new_sleep.o -o brevc  ou   mpicc brevc.c new_sleep.o -o brevc
```

- 4 - Execute o programa iniciando apenas **dois processos**. Execute pelo menos três vezes. Verifique se a execução está correta e se houve alguma melhora no tempo de execução, em relação ao exercício anterior.

```
IBM   : poe ./brevc -procs 2 -hostfile ./arquivo
```

```
INTEL : mpirun ./brevc -np 2 ./brevc
```

```
MPICH2 : mpiexec ./brevc -np 2 -f ./arquivo
```

*arquivo = nome do arquivo de máquinas*

## 8 - Rotinas de Comunicação Coletiva

- Uma comunicação coletiva envolve todos os processos de um grupo de processos;
- O objetivo deste tipo de comunicação é o de manipular um pedaço **comum** de informação por todos os processos de um grupo;
- As rotinas de comunicação coletiva são montadas utilizando-se as rotinas de comunicação "point-to-point" e somente "blocking";
- As rotinas de comunicação coletivas estão divididas em três categorias:

**"synchronization"**  
**"data movement"**  
**"global computation"**

- Envolve comunicação coordenada entre processos de um grupo, identificados pelo parâmetro **"communicator"**;
- Não é necessário rotular as mensagens (utilizar **tags**);
- Todos os processos executam a mesma rotina de comunicação coletiva, que pode funcionar como um "send" ou pode funcionar como um "receive";

## 8.1 - "Synchronization"

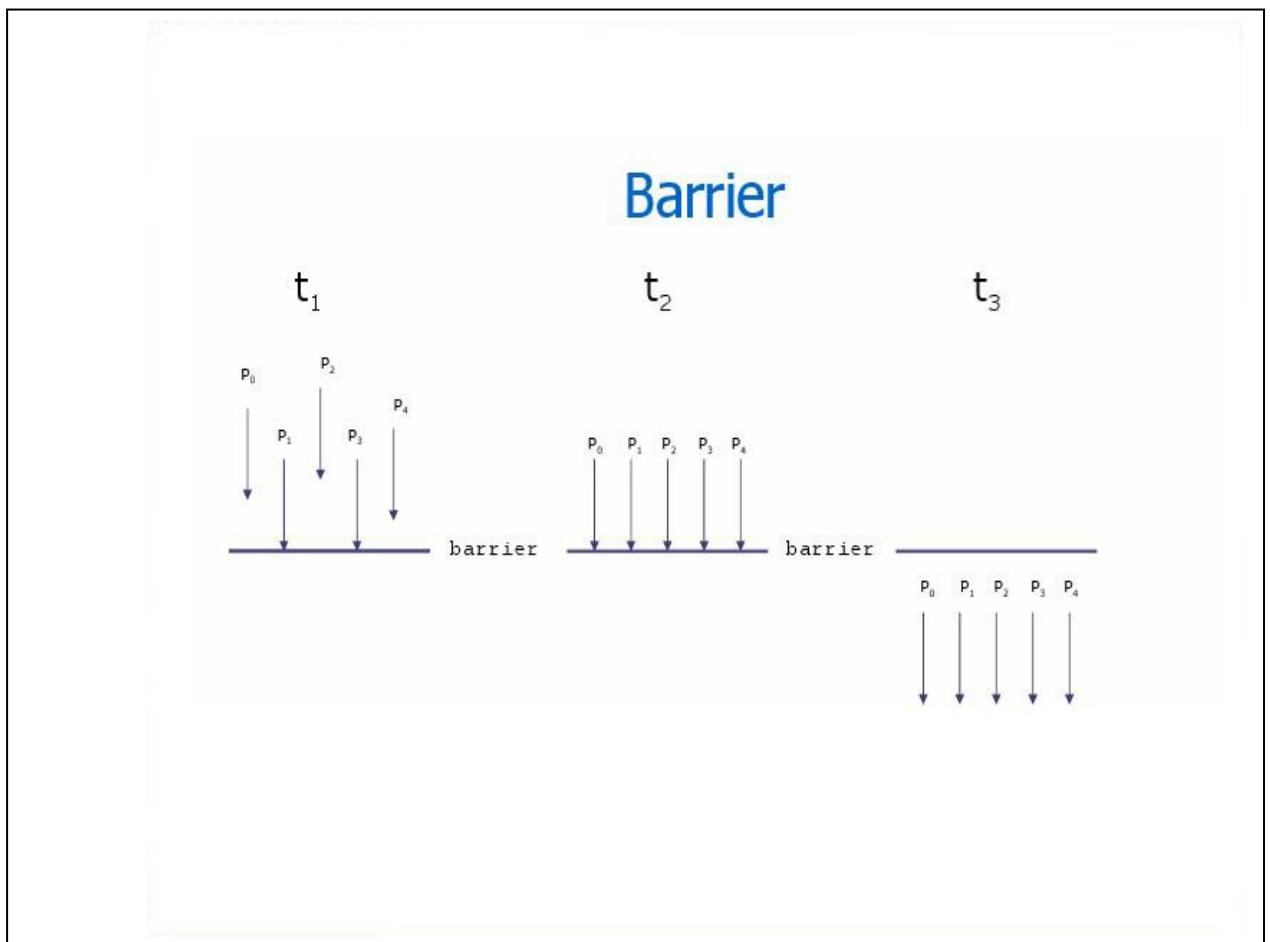
### 8.1.1 - Rotina MPI\_Barrier

C	<code>int MPI_Barrier ( comm )</code>
FORTTRAN	<code>CALL MPI_BARRIER (comm, ierr)</code>

**comm**            Inteiro que determina o "**communicator**", que por sua vez, determina o grupo de processos;

**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**.



## 8.2 - "Data Movement"

### 8.2.1 - Rotina MPI\_Broadcast

C	<code>int MPI_Bcast(*buffer, count, datatype, root, comm)</code>
FORTRAN	<code>CALL MPI_BCAST (buffer, count, datatype, root, comm, ierr)</code>

**buffer**                   Endereço inicial do dado a ser enviado;

**count**                   Variável inteira 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**, enviar a mensagem;

**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**. O processo identificado como **root**, enviará os dados, enquanto que o processo que não possui a identificação **root**, receberá os dados.

```
PROGRAM broad cast
  INCLUDE 'mpif.h'
  INTEGER ierr, myid, nproc, root
  INTEGER status(MPI_STATUS_SIZE)
  REAL A(2)

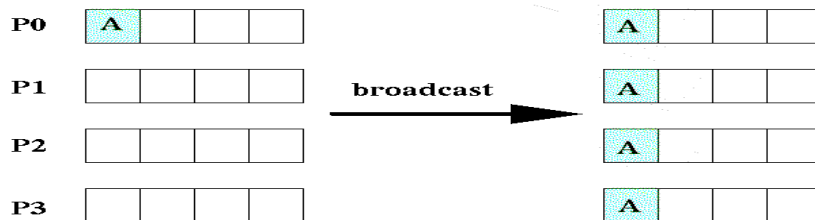
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

  root = 0
  IF( myid .EQ. 0 ) THEN
    a(1) = 2.0
    a(2) = 4.0
  END IF

  CALL MPI_BCAST(a, 2, MPI_REAL, 0, MPI_COMM_WORLD, ierr)

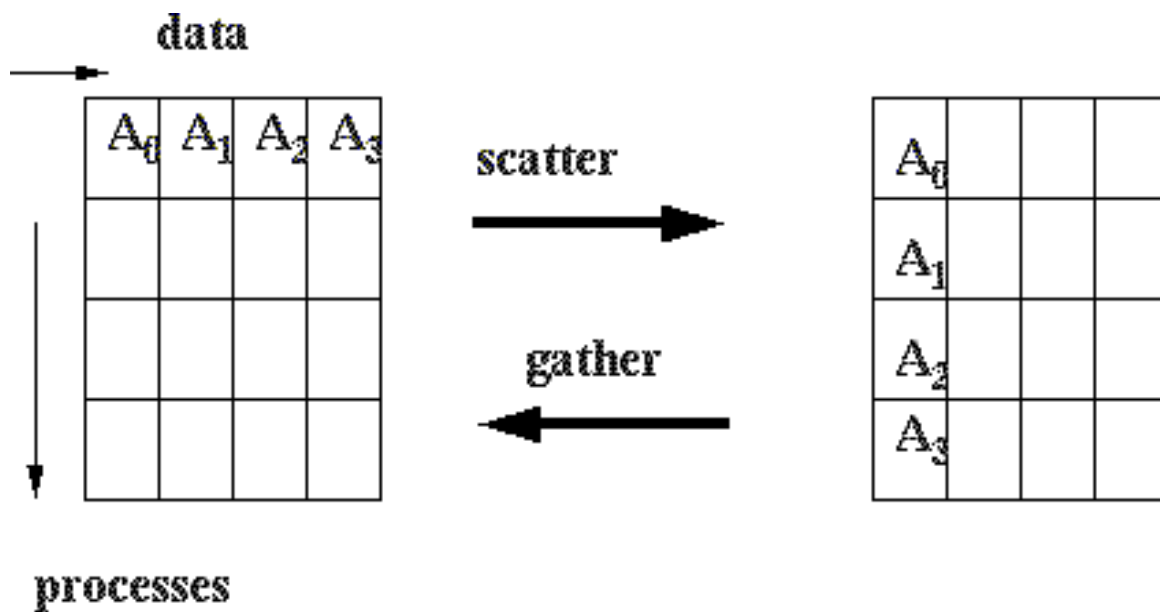
  WRITE(6,*) myid, ': a(1)=', a(1), 'a(2)=', a(2)

  CALL MPI_FINALIZE(ierr)
END
```



## “Scatter” e “Gather”

Se um processo necessita distribuir dados em  $n$  segmentos iguais, onde o  $n$ ésimo segmento é enviado para  $n$ ésimo 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**.





## 8.2.2 – Rotina MPI\_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 pelo processo **root**;

**scount** Número de elementos que serão distribuídos **para cada processo**;

**stype** Tipo de dado que será distribuído;

**rbuf** Endereço aonde os dados serão coletados **por cada processo**;

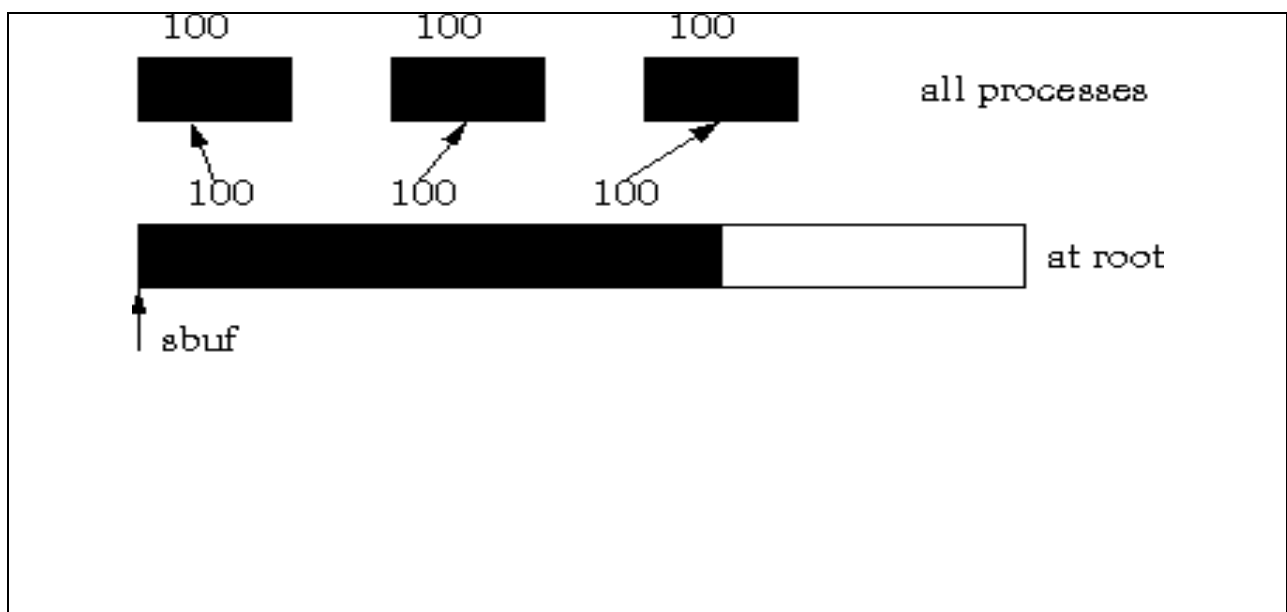
**rcount** Número de elementos que serão coletados **por cada processo** ;

**rtype** Tipo de dado que será coletado;

**root** Identificação do processo que irá distribuir os dados;

**comm** Identificação do "communicator".

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



## 8.2.3 – Rotina MPI\_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 enviados **por cada processo**;

**scount** Número de elementos que serão enviados **por cada processo**;

**stype** Tipo de dado que será enviado;

**rbuf** Endereço aonde os dados serão coletados pelo processo **root**;

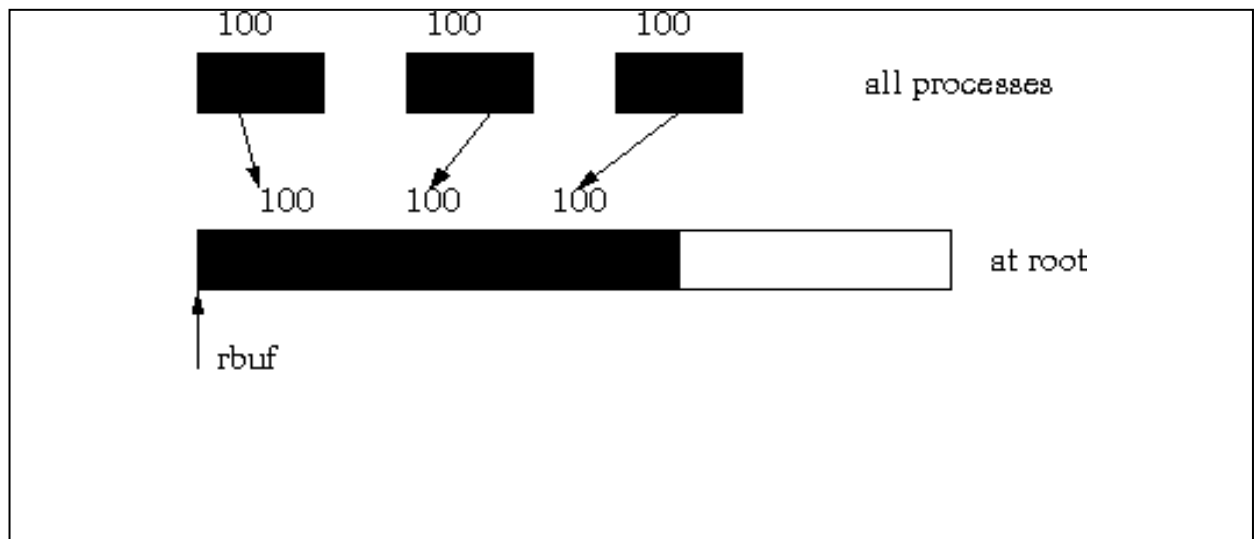
**rcount** Número de elementos que serão coletados;

**rtype** Tipo de dado coletado;

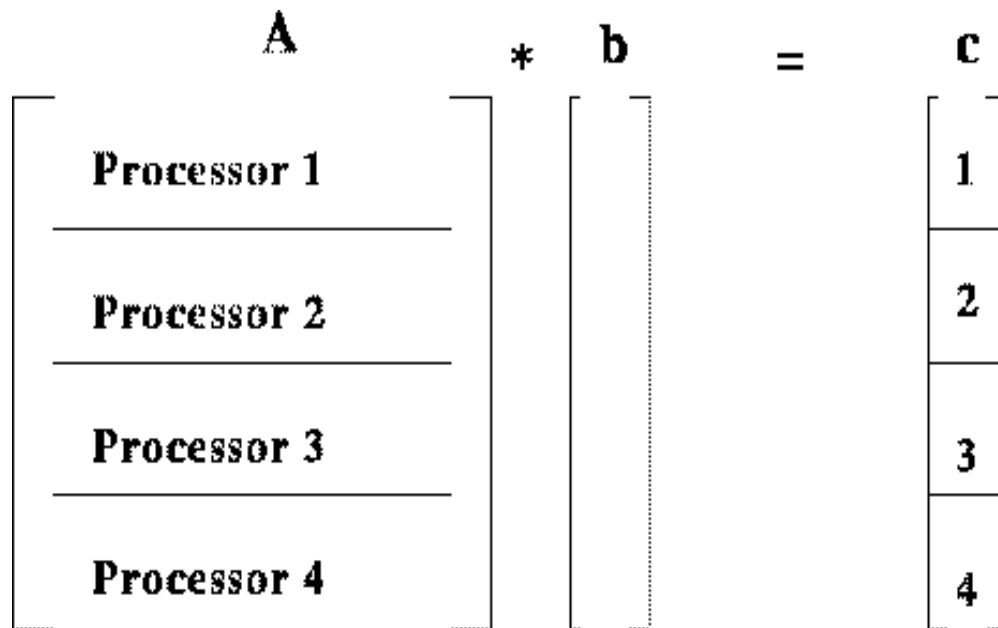
**root** Identificação do processo que ira coletar os dados;

**comm** Identificação do "communicator".

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



## Exemplo



**A: Matriz distribuída por linhas;**

**b: Vetor compartilhado por todos os processos;**

**c: Vetor atualizado por cada processo, independentemente.**

```

REAL A(100,100), b(100), cpart(25), ctotat(100)
INTEGER root, me, erro, tp
DATA root/0/
    ...
    (distribuição dos dados) !!!
    ...

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,ctotat,25,MPI_REAL,root,MPI_COMM_WORLD,ierr)
CALL MPI_Finalize(erro);
    ...
    ...
    ...
END

```

**OBS: Exemplo completo no LABORATÓRIO3 na pasta exemplo:**

```

lab03/exemplo/multiplicacao_matriz_vetor.f
lab03/exemplo/multiplicacao_matriz_vetor.c

```

## 8.2.4 – Rotina MPI\_Allgather

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

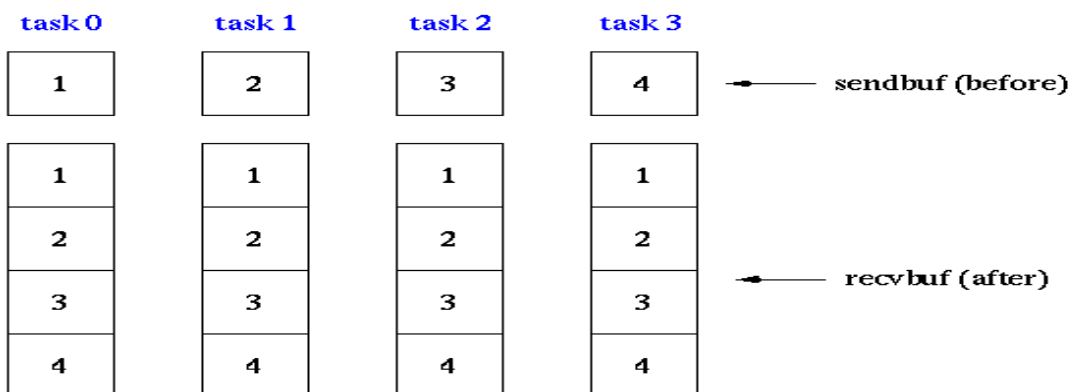
```
FORTRAN  CALL MPI_ALLGATHER(sbuf,scount,stype,rbuf,rcount,rtype,comm,ierr)
```

- sbuf** Endereço inicial dos dados que serão enviados **por todos os processos**;
- scount** Número de elementos que serão enviados **por todos os processos**;
- stype** Tipo de dado que será distribuído;
- rbuf** Endereço aonde os dados serão coletados por **todos os processos**;
- rcount** Número de elementos que serão coletados **por todos os processos**;
- rtype** Tipo de dado coletado;
- comm** Identificação do "communicator".

Essa rotina ao ser executada faz com que todos os processos enviem dados e colem dados de cada processo da aplicação. Seria similar a **cada processo** efetuar um "broadcast".

### MPI\_Allgather

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



## 8.2.5 – Rotina MPI\_Alltoall

C	<code>int MPI_Alltoall( *sbuf, scount, stype, *rbuf, rcount, rtype, comm)</code>
FORTRAN	<code>CALL MPI_ALLTOALL(sbuf,scount,stype,rbuf,rcount,rtype,comm,ierr)</code>

**sbuf** Endereço inicial dos dados que serão distribuídos por **todos os processos**;

**scount** Número de elementos que serão distribuídos **por todos os processos**;

**stype** Tipo de dado que será distribuído;

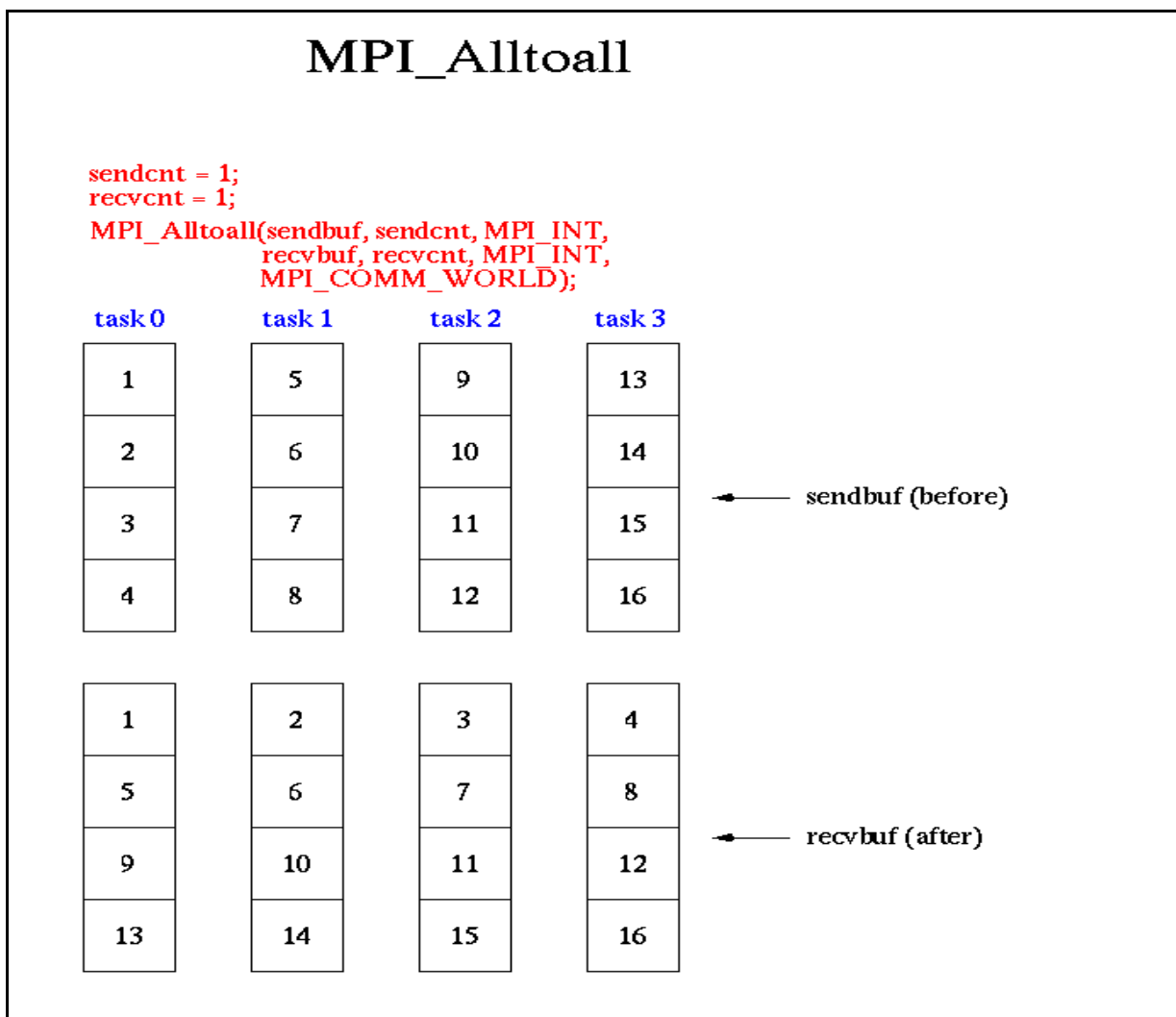
**rbuf** Endereço aonde o dados serão coletados **por todos os processos**;

**rcount** Número de elementos que serão coletados **por todos os processos**;

**rtype** Tipo de dado coletado;

**comm** Identificação do "communicator".

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



## 8.3 - 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**.

### 8.3.1 – Rotina MPI\_Reduce

<b>C</b>	<code>int MPI_Reduce( *sbuf, *rbuf, count, stype, op, root, comm)</code>
<b>FORTRAN</b>	<code>call MPI_REDUCE(sbuf,rbuf,count,stype,op,root,comm,ierr)</code>

**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".

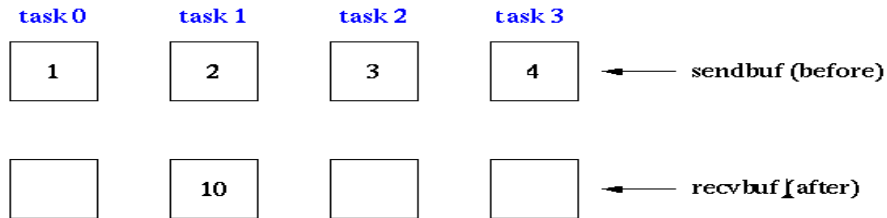
**Tabela com Funções Pré-definidas de Operações de Redução**

<b>FUNÇÃO</b>	<b>RESULTADO</b>	<b>C</b>	<b>FORTRAN</b>
<b>MPI_MAX</b>	<b>valor máximo</b>	<b>integer,float</b>	<b>integer,real,complex</b>
<b>MPI_MIN</b>	<b>valor mínimo</b>	<b>integer,float</b>	<b>integer,real,complex</b>
<b>MPI_SUM</b>	<b>somatório</b>	<b>integer,float</b>	<b>integer,real,complex</b>
<b>MPI_PROD</b>	<b>produto</b>	<b>integer,float</b>	<b>integer,real,complex</b>

## MPI\_Reduce

```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
            dest, MPI_COMM_WORLD);
```

result will be placed in task 1



### C

```
#include <stdio.h>
#include <mpi.h>
void main(int argc, char *argv[])
{
    int rank;
    int source,result,root;

    /* run on 10 processors */

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

    root=7;
    source=rank+1;
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Reduce(&source,&result,1,MPI_INT,
              MPI_PROD,root,MPI_COMM_WORLD);

    if(rank==root)
        printf("P:%d MPI_PROD result is %d\n",rank,result);

    MPI_Finalize();
}
```

### Fortran

```
PROGRAM product
    INCLUDE 'mpif.h'
    INTEGER err, rank, size
    INTEGER source,result,root

    C run on 10 processors

    CALL MPI_INIT(err)
    CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, err)
    CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, err)

    root=7
    source=rank+1
    call MPI_BARRIER(MPI_COMM_WORLD,err)

    call MPI_REDUCE(source,result,1,MPI_INT, &
                   MPI_PROD,root,MPI_COMM_WORLD,err)

    if(rank.eq.root) then
        print *, 'P:',rank, ' MPI PROD result is',result
    end if

    CALL MPI_FINALIZE(err)
END
```



# LABORATÓRIO 3 - Comunicação Coletiva

## Exercício 1

- 1 - Caminhe para o diretório com o primeiro exercício do laboratório. Edite o programa e altere o que for solicitado:

```
cd ./lab03/ex1
vi collective.f    ou    vi collective.c    (Outros editores: pico ou nano)
```

- O processo mestre solicita a entrada de um número, que será a semente para o cálculo de um número randômico, esse número será **enviado para todos os processos**.

**1ª Rotina:** Adicione ao programa, no lugar da "seta", a rotina MPI que envia dados para todos os processos;

- Cada processo calculará um número randômico, baseado no número de entrada informado. O processo com o maior número de identificação **receberá o somatório dos números randômicos** e calculará o valor médio de todos os números randômicos calculados.

**2ª Rotina:** Adicione ao programa, no lugar da "seta", a rotina MPI que efetua uma operação global de redução;

- Cada processo irá calcular, novamente, mais 4 novos números randômicos. Um processo **receberá todos os números randômicos**, irá calcular o valor máximo e o desvio padrão, e os resultados **serão distribuídos para todos os processos**. Existem dois métodos par efetuar essas tarefas, utilizando rotinas de comunicação coletiva diferentes.

**1º Método - 3ª Rotina:** Adicione ao programa, no lugar das "setas", a rotina MPI que coleta os dados de outros processo;

**4ª Rotina:** Adicione ao programa, no lugar das "setas", a rotina MPI que envia os dados para todos os processos;

**2º Método - 5ª Rotina:** Adicione ao programa, no lugar das "setas", a rotina MPI que coleta e envia, simultaneamente, os dados dos processo;

- 2 - Compile o programa:

```
IBM      : mpixlf collective.f -o cole ou      mpicc collective.c -o cole
```

```
INTEL    : ifort collective.f -o cole -lmpi ou    icc collective.c -o cole -lmpi
```

```
MPICH2   : mpif77 collective.f -o cole ou      mpicc collective.c -o cole
```

- 3 - Execute o programa:

```
IBM      : poe ./cole -procs n -hostfile ./arquivo
```

```
INTEL    : ulimit -v unlimited
```

```
mpirun -np n ./cole
```

```
MPICH2   : mpiexec ./cole -np n -f ./arquivo
```

*n = número de processos*

*arquivo = nome do arquivo de máquinas*

## Exercício 2

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

```
cd ./lab03/ex2
```

A idéia do programa é demonstrar a execução da rotina **MPI\_SCATTER**.

- 2 - Edite o programa e adicione os parâmetros da rotina, para que ela funcione adequadamente.

```
vi sactter.f    ou    vi scatter.c           (Outros editores: pico ou nano)
```

- 3 - Compile o programa:

```
IBM    : mpxlf scatter.f -o sca           ou    mpcc scatter.c -o sca
```

```
INTEL  : ifort scatter.f -o sca -lmpi    ou    icc scatter.c -o sca -lmpi
```

```
MPICH2: mpif77 scatter.f -o sca         ou    mpicc scatter.c -o sca
```

- 4 - Execute o programa:

```
IBM    : poe ./sca -procs n -hostfile ./arquivo
```

```
INTEL  : mpirun -np n ./sca
```

```
MPICH2: mpiexec ./sca -np n -f ./arquivo
```

*n = número de processos*

*arquivo = nome do arquivo de máquinas*

### Exercício 3

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

```
cd ./lab03/ex3
```

Este programa calcula o maior número primo dos números primos calculados, até um limite determinado pelo programador (O valor da variável LIMIT). A idéia é demonstrar a utilização, apenas da rotina de operação de redução, **MPI\_REDUCE**.

- 2 - Edite o programa e adicione os parâmetros da rotina, para que ela funcione adequadamente.

```
vi mpi_prime.f   ou   vi mpi_prime.c           (Outros editores: pico ou nano)
```

- 3 - Compile o programa:

```
IBM    : mpxlf mpi_prime.f -o primo    ou mpcc mpi_prime.c -o primo
```

```
INTEL  : ifort mpi_prime.f -o primo -lmpi ou icc mpi_prime.c -o primo -lmpi
```

```
MPICH2: mpif77 mpi_prime.f -o primo    ou mpicc mpi_prime.c -o primo
```

- 4 - Execute o programa:

```
IBM    : poe ./primo -procs n -hostfile ./arquivo
```

```
INTEL  : mpirun -np n ./primo
```

```
MPICH2: mpiexec ./primo -np n -f ./arquivo
```

*n = número de processos*

*arquivo = nome do arquivo de máquinas*

## Exercício 4

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

```
cd ./lab03/ex4
```

Este exercício possui programas que calculam o valor de pi, com o uso de um algoritmo especial:

- 1.1 - Na primeira solução, é utilizada as rotinas de **send** e **receive** para que os processos calculem uma parte do valor de pi. O valor de pi é calculado por uma subrotina externa (**dboard.f**), que deve ser compilada junto com o programa principal. Apenas analise a paralelização do programa com essas rotinas e verifique a execução

- Compile o programa:

```
IBM : mpixlf pi_paralelo_v1.f dboard.f -o piv1 ou mpcc pi_paralelo_v1.c dboard.c -o piv1
```

```
INTEL : ifort pi_paralelo_v1.f dboard.f -o piv1 -lmpi ou icc pi_paralelo_v1.c dboard.c -o piv1 -lmpi
```

```
MPICH2: mpif77 pi_paralelo_v1.f dboard.f -o piv1 ou mpicc pi_paralelo_v1.c dboard.c -o piv1
```

- Execute o programa:

```
IBM : poe ./piv1 -procs n -hostfile ./arquivo
```

```
INTEL : mpirun ./piv1 -np n -machinefile ./arquivo
```

```
MPICH2 : mpiexec ./piv1 -np n -f ./arquivo
```

*n = número de processos*

*arquivo = nome do arquivo de máquinas*

- 1.2 - A segunda solução, é mais eficiente pois utiliza uma rotina MPI de comunicação coletiva. A lógica de paralelização está pronta, apenas codifique as rotinas MPI necessárias.

- Edite o programa e codifique o que for necessário para executar com MPI:

```
vi pi_paralelo_v2.f ou vi pi_paralelo_v2.c (Outros editores: pico ou nano)
```

- Compile o programa:

```
IBM : mpixlf pi_paralelo_v2.f dboard.f -o piv2 ou mpcc pi_paralelo_v2.c dboard.c -o piv2
```

```
INTEL : ifort pi_paralelo_v2.f dboard.f -o piv2 -lmpi ou icc pi_paralelo_v2.c dboard.c -o piv2 -lmpi
```

```
MPICH2: mpif77 pi_paralelo_v2.f dboard.f -o piv2 ou mpicc pi_paralelo_v2.c dboard.c -o piv2
```

- Execute o programa:

```
IBM : poe ./piv2 -procs n -hostfile ./arquivo
```

```
INTEL : mpirun -np n ./piv2
```

```
MPICH2 : mpiexec ./piv2 -np n -f ./arquivo
```

*n = número de processos*

*arquivo = nome do arquivo de máquinas*

## 9 - Grupos e “Communicators”

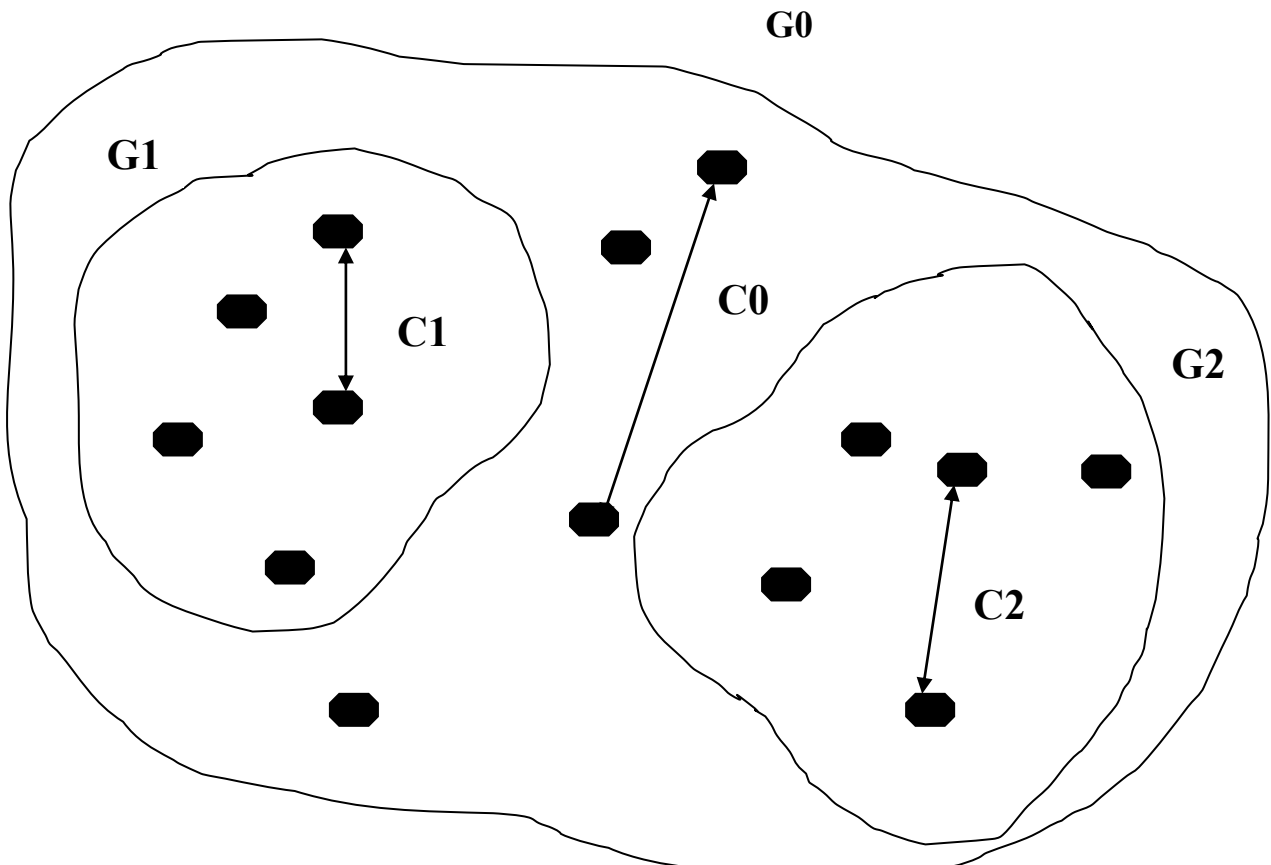
### Grupo

- Grupo é um conjunto ordenado de processos. Cada processo em um grupo pode possuir um novo e único número de identificação;
- O MPI suporta o processamento de grupos, permitindo:
  - 1 Organizar processos em grupos, de acordo com a natureza da aplicação;
  - 2 Permitir operações de Comunicação Coletiva entre alguns processos;
- Um processo pode pertencer a mais de um grupo;

### “Communicator”

- Um grupo utiliza um “communicator” específico, que descreve o universo de comunicação entre os processos;
- Em termos de programação, grupo e communicator são equivalentes;
- Em MPI, grupo e communicator são objetos dinâmico que podem ser criados e destruídos durante a execução de um programa.

### C0 – MPI\_COMM\_WORLD



## 9.1 - Identifica um Grupo - Rotina MPI\_Comm\_Group

C	<code>int MPI_Comm_group(comm , *group )</code>
FORTRAN	<code>CALL MPI_COMM_GROUP (comm, group, ierr)</code>

Identifica o grupo que está associado a um determinado “communicator”. É necessário identificar o grupo para poder criar novos grupos.

**comm** “Communicator”

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

**ierr** Status de execução da rotina.

## 9.2 - Cria um Grupo por Inclusão - Rotina MPI\_Group\_incl

C	<code>int MPI_Group_incl (group, n, *ranks, *newgroup)</code>
FORTRAN	<code>CALL MPI_GROUP_INCL (group, n, ranks, newgroup, ierr)</code>

Cria um novo grupo a partir de um grupo existente, com a **inclusão de processos** identificados.

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

**n** Número de processos que serão incluídos no novo grupo (Também indica o tamanho do conjunto de *ranks*);

**ranks** Vetor com a identificação dos processos do grupo existente, que serão **incluídos**;

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

**ierr** Status de execução da rotina.

## 9.3 - Cria um Grupo por Exclusão - Rotina MPI\_Group\_excl

C	<code>int MPI_Group_excl (group, n, *ranks, *newgroup)</code>
FORTRAN	<code>call MPI_GROUP_EXCL (group, n, ranks, newgroup, ierr)</code>

Cria um novo grupo a partir de um grupo existente, com a **exclusão de processos** que são identificados.

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

**n** Número de processos que serão excluídos do novo grupo (Também indica o tamanho do conjunto *ranks*);

**ranks** Vetor com a identificação dos processos no grupo existente, que serão **excluídos**. Os processos que não forem identificados, farão parte do novo grupo;

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

**ierr** Status de execução da rotina.

## 9.4 - Identifica Processo no Grupo - Rotina MPI\_Group\_rank

```
C          int MPI_Group_rank( group, *rank )
FORTRAN  CALL MPI_GROUP_RANK( group, rank, ierr )
```

Rotina que identifica um processo dentro do novo grupo.

**group** Variável inteira, “transparente”, com a identificação do grupo;

**rank** Variável inteira de retorno com a identificação do processo no grupo. Número contínuo de 0 a n-1;

**ierr** Status de execução da rotina.

## 9.5 - Cria um “Communicator” - Rotina MPI\_Comm\_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.

## 9.6 - Apagar Grupos - Rotina MPI\_Group\_free

<b>C</b>	<b>int MPI_Group_free ( group)</b>
<b>FORTRAN</b>	<b>CALL MPI_GROUP_FREE (group, ierr)</b>

Apaga, somente, a definição de um grupo.

**group** Variável inteira, *transparente*, que identifica o grupo que será apagado a definição.

**ierr** Status de execução da rotina.

## 9.7 - Apagar “Communicators” - Rotina MPI\_Comm\_free

<b>C</b>	<b>int MPI_Comm_free (*comm)</b>
<b>FORTRAN</b>	<b>CALL MPI_COMM_FREE (comm, ierr)</b>

Apaga a definição de um “communicator”.

**comm** Variável inteira, *transparente*, que identifica o “communicator” que será apagado a definição.

**ierr** Status de execução da rotina.



## Exemplo de Uso de Grupos e “Communicators”

```
PROGRAM mainprog
  IMPLICIT NONE

  INCLUDE "mpif.h"

  INTEGER          :: me
  INTEGER          :: ranks = 0
  INTEGER          :: send_buf=1, send_buf2=1, recv_buf=0, recv_buf2=0
  INTEGER          :: count=1, count2=1
  INTEGER          :: COMMSLAVE, MPI_GROUP_WORLD
  INTEGER          :: GRPREM, rstat ! Status variable

  CALL MPI_Init(rstat)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, me, rstat)

! Identificar o grupo principal

  CALL MPI_Comm_group(MPI_COMM_WORLD, MPI_GROUP_WORLD, rstat)

! Criar um novo grupo a partir da exclusão de processos

  CALL MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, GRPREM, rstat)

! Criar um novo “communicator” para o novo grupo

  CALL MPI_Comm_create(MPI_COMM_WORLD, grprem, COMMSLAVE, rstat)

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

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

  print *, "Processo", me, "Grupo=", recv_buf, "Geral=", recv_buf2

  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
```

```
mpirun -np 6 ./a.out
```

```
Processo      0 Grupo=      0 Geral=      6
Processo      1 Grupo=      5 Geral=      0
Processo      3 Grupo=      0 Geral=      0
Processo      4 Grupo=      0 Geral=      0
Processo      5 Grupo=      0 Geral=      0
Processo      2 Grupo=      0 Geral=      0
```

# LABORATÓRIO 4 - Grupo e “Communicator”

## Exercício 1

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

```
cd ./lab04/ex1
```

Neste exercício, serão criados dois grupos identificados como: processos ímpares e processos pares. Cada grupo executará uma operação de redução.

2 - Edite o programa e **adicione, no lugar das "setas", a rotina MPI adequada para esta operação;**

```
vi create.f    ou    vi create.c          (Outros editores: pico ou nano)
```

3 - Compile o programa:

```
IBM : mpixlf create.f -o grupo    ou    mpicc create.c -o grupo
```

```
INTEL : ifort create.f -o grupo -lmpi    ou    icc create.c -o grupo -lmpi
```

```
MPICH2: mpif77 create.f -o grupo    ou    mpicc create.c -o grupo
```

4 - Execute o programa:

```
IBM      : poe ./grupo -procs n -hostfile ./arquivo
```

```
INTEL    : ulimit -v unlimited
```

```
          mpirun -np n ./grupo
```

```
MPICH2   : mpiexec ./grupo -np n -f ./arquivo
```

*n* = número de processos

*arquivo* = nome do arquivo de máquinas

## Exercício 2

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

```
cd ./lab04/ex2
```

Neste exercício serão criados dois grupos utilizando a rotina de inclusão em grupo, que será executada duas vezes para dois vetores de identificação de processos. No final da execução do programa.

2 - Edite o programa e **adicione, no lugar das "setas", a rotina MPI adequada para esta operação;**

```
vi mpi_group.f   ou   vi mpi_group.c           (Outros editores: pico ou nano)
```

3 - Compile o programa:

```
IBM : mpixlf mpi_group.f -o grp   ou   mpicc mpi_group.c -o grp
```

```
INTEL : ifort mpi_group.f -o grp -lmpi   ou   icc mpi_group.c -o grp -lmpi
```

```
MPICH2: mpif77 mpi_group.f -o grp   ou   mpicc mpi_group.c -o grp
```

4 - Execute o programa:

```
IBM      : poe ./grp -procs n -hostfile ./arquivo
```

```
INTEL    : mpirun -np n ./grp
```

```
MPICH2   : mpiexec ./grp -np n -f ./arquivo
```

*n* = número de processos

*arquivo* = nome do arquivo de máquinas

# LABORATÓRIO 5 – Problemas de Execução

## Exercício 1

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

```
cd ./lab05/ex1
```

O programa possui um problema que trava a execução.

2 - Compile o programa:

```
IBM : mpixlf mpi_bug1.f -o bug1 ou mpicc mpi_bug1.c -o bug1
```

```
INTEL : ifort mpi_bug1.f -o bug1 -lmpi ou icc mpi_bug1.c -o bug1 -lmpi
```

```
MPICH2: mpif77 mpi_bug1.f -o bug1 ou mpicc mpi_bug1.c -o bug1
```

3 - Execute o programa, iniciando apenas 2 processos:

```
IBM : poe ./bug1 -procs 2 -hostfile ./arquivo
```

```
INTEL : mpirun -np 2 ./bug1
```

```
MPICH2 : mpiexec ./bug1 -np 2 -f ./arquivo
```

*arquivo = nome do arquivo de máquinas*

4 - Edite o programa e tente corrigir o problema.

```
vi mpi_bug1.f ou vi mpi_bug1.c (Outros editores: pico ou nano)
```

5 - Compile e execute novamente.

## Exercício 2

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

```
cd ./lab05/ex2
```

Os dados recebidos e impressos por um processo, são diferentes dos dados enviados. Aonde está o problema?

2 - Compile o programa:

```
IBM : mpxlf mpi_bug2.f -o bug2 ou mpcc mpi_bug2.c -o bug2
```

```
INTEL : ifort mpi_bug2.f -o bug2 -lmpi ou icc mpi_bug2.c -o bug2 -lmpi
```

```
MPICH2: mpif77 mpi_bug2.f -o bug2 ou mpicc mpi_bug2.c -o bug2
```

3 - Execute o programa, iniciando apenas 2 processos:

```
IBM : poe ./bug2 -procs 2 -hostfile ./arquivo
```

```
INTEL : mpirun -np 2 ./bug2
```

```
MPICH2 : mpiexec ./bug2 -np 2 -f ./arquivo
```

*arquivo = nome do arquivo de máquinas*

4 - Edite o programa e tente corrigir o problema.

```
vi mpi_bug2.f ou vi mpi_bug2.c (Outros editores: pico ou nano)
```

5 - Compile e execute novamente.

### Exercício 3

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

```
cd ./lab05/ex3
```

A execução e cancelada por erros de violação de memória. Aonde está o problema?

2 - Compile o programa:

```
IBM : mpixlf mpi_bug3.f -o bug3 ou mpicc mpi_bug3.c -o bug3
```

```
INTEL : ifort mpi_bug3.f -o bug3 -lmpi ou icc mpi_bug3.c -o bug3 -lmpi
```

```
MPICH2: mpif77 mpi_bug3.f -o bug3 ou mpicc mpi_bug3.c -o bug3
```

3 - Execute o programa:

```
IBM : poe ./bug3 -procs n -hostfile ./arquivo
```

```
INTEL : mpirun -np n ./bug3
```

```
MPICH2 : mpiexec ./bug3 -np n -f ./arquivo
```

*n* = número de processos

*arquivo* = nome do arquivo de máquinas

4 - Edite o programa e tente corrigir o problema.

```
vi mpi_bug3.f ou vi mpi_bug3.c (Outros editores: pico ou nano)
```

5 - Compile e execute novamente.

## 10 - Referências

- 1 - **Message Passing Interface (MPI)**  
MHPCC - Maui High Performance Computing Center  
Blaise Barney - August 29, 1996
- 2 - **SP Parallel Programming Workshop - Message Passing Overview**  
MHPCC - Maui High Performance Computing Center  
Blaise Barney - 03 Julho 1996
- 3 - **Programming Languages and Tools: MPI**  
CTC - Cornell Theory Center  
April, 1996
- 4 - **MPI: A Message-Passing Interface Standard**  
University of Tennessee, Knoxville, Tennessee  
May 5, 1994
- 5 - **MPI: The Complete Reference**  
The MIT Press - Cambridge, Massachusetts  
Marc Snir, Steve Otto, Steven Huss-Lederman,  
David Walker, Jack Dongarra  
1996
- 6 - **Democritos/ICTP course in “Tools for computational physics”**  
Democritos/INFN + SISSA  
Stefano Cozzini [cozzini@democritos.it](mailto:cozzini@democritos.it)  
Fevereiro/2005
- 7 - **Introduction to Parallel Computing**  
Lawrence Livermore National Laboratory – LLNL  
California – USA – 22 Junho 2007  
Blaise Barney