Comunicação Interprocessos

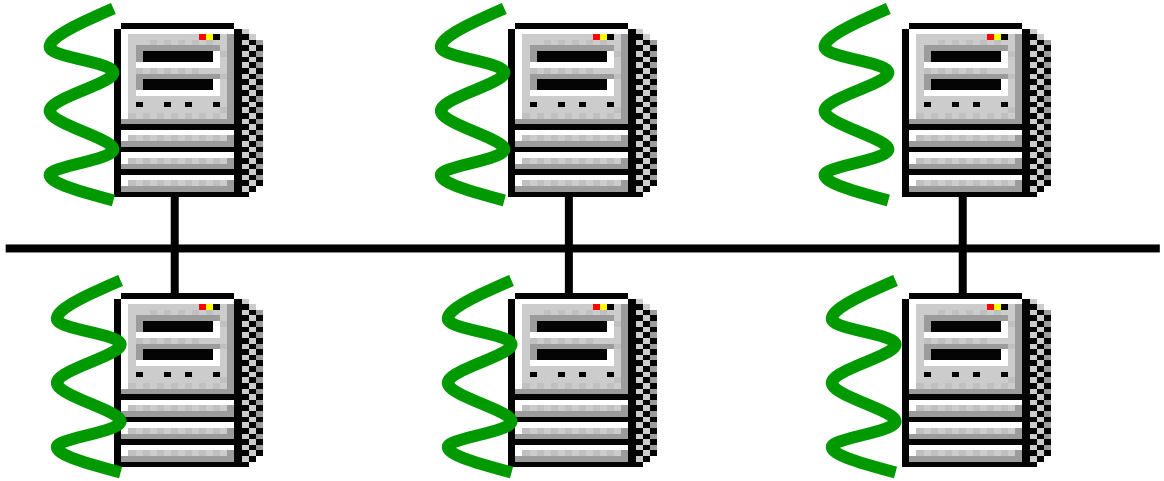
Programação Paralela e Distribuída

Conceito de Programa e Execução

Graduação em Ciência da Computação
Universidade do Vale do Rio dos Sinos

Objetivos

Explorar a concorrência entre-nós



Objetivos

Explorar a concorrência intra-nó

Criação de uma máquina virtual

Estudo de caso: MPI
LAM – Local Area Machine

MPI

Principais versões

- IBM MPI implementação IBM para SP e cluster
- MPICH Argonne National Laboratory/Mississipi State Universaty
- UNIFY Mississipi State Universaty
- CHIMP Edinburgh Parallel Computing Center
- LAM Ohio Supercomputer Center
- PMPIO NASA
- MPIX Mississipi State University NSF Engineerin Research Center

MPI

Por quê MPI?

1. Possui diversas implementações de qualidade livres.
2. Permite programar eficientemente MPP e aglomerados.
3. Especificação feita com cuidado.
4. Manipula de forma eficiente buffers de mensagens.
5. Propõe comunicação assíncrona.
6. Comunicação em grupo é eficiente e determinista.
7. Define como deve ser realizado o mecanismo de profiling externo.
8. A sincronização provida protege software externo.
9. É portável.
10. É um padrão.

MPI

Message Passing Interface

- Comunicação ponto a ponto
- Comunicação em grupo
- Tipos de dados derivados

- 125 rotinas
- Padrão de fato

MPI

Message Passing Interface

- Antes de 1980: Bibliotecas proprietárias
- 1989: Parallel Virtual Machine (PVM) (Oak Ridge National Lab)
- 1992: MPI – Primeiros passos – MPIForum (40 organizações)
- 1994: MPI v. 1.0
- 1997: MPI v. 2.0 – Criação dinâmica de processos / E/S paralelo
- Today: Mais utilizado

`http://www.mpi-forum.org`

MPI

- Uma especificação de biblioteca de comunicação:
  - Implementa um modelo de troca de mensagens
  - Não propõe uma especificação para compiladores
  - Não é focada a um determinado produto
- Projetada para:
  - Computadores paralelos, aglomerados e redes de heterogeneas de computadores
  - Auxiliar projetistas a implementar bibliotecas de comunicação portáveis
- Desenvolvida para auxiliar:
  - Usuários finais (programadores de aplicações)
  - Desenvolvedores de bibliotecas de comunicação
  - Programadores de ferramentas

Termos

- **Máquina virtual**: a execução de uma aplicação MPI se dá em uma máquina virtual, especialmente contruída para executar uma determinada aplicação.
- **Processo**: um processo é um nó virtual de uma arquitetura MPI. Consiste em uma instância do programa MPI lançado. Um processo Unix torna-se um processo MPI no momento em que executa MPI_Init.
- **Task**: muitas vezes empregado como sinônimo de processo.
- **Síncrono/Assíncrono**: serviços podem ser síncronos ou assíncronos, cabe verificar a propriedade oferecida pela primitiva de interesse.

Termos

- **Bloqueante**: um serviço é dito bloqueante quando o retorno de sua chamada ocorrer somente quando o serviço associado tiver sido completado.
- **Não bloquante**: um serviço é dito não bloqueante quando o retorno de sua chamada ocorrer somente quando o procedimento associado a chamada tiver sido completado, não necessáriamente o serviço associado encontra-se completado. O procedimento da chamada tem a função de iniciar a execução do serviço associado.
- **Requisição**: representada por um objeto, uma requisição encontra-se associada a um serviço iniciado através de uma chamada não bloqueante.

Termos

- **Local**: um procedimento é dito local quando sua operação não depender de interação com outros serviços em nós remotos.
- **Não local**: um procedimento é dito não local quando sua operação depender de interação com outros serviços em nós remotos.
- **Coletiva**: um procedimento coletivo implica na interação de todos os processos participantes de um grupo.
- **Pré-definido**: tipo de dado primitivo que o MPI pode manipular.
- **Derivado**: tipo de dado construído a partir de tipos pré-definidos.
- **Portável**: um tipo de dado é dito portável se ele é um tipo pré-definido ou se ele deriva de um tipo pré-definido utilizando construtores de enumeração (vetor, bloco, array, ...)

Memória Distribuída

MPI – LAM

- Programação SPMD
- Máquina virtual
- LAM básico
- Mensagens
- Comunicação de grupo

Memória Distribuída

Programação SPMD

Vários processos são criados para executar um mesmo programa.

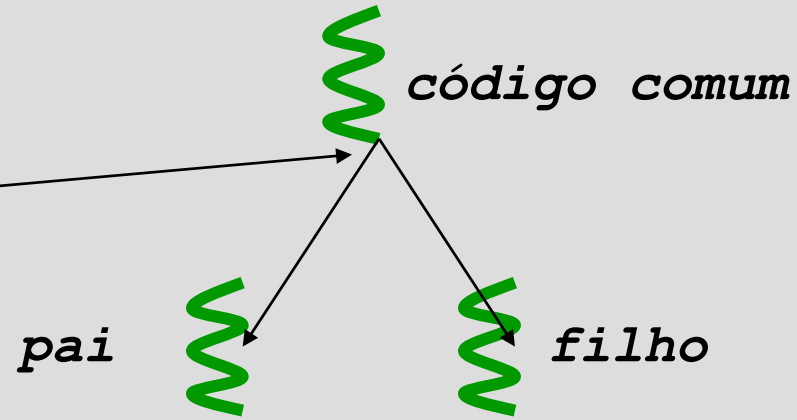
Cada processo possui seu próprio conjunto de dados, o que pode implicar em execuções totalmente diferentes

Memória Distribuída

Programação SPMD

O `fork` no Unix

```
main() {
  int id;
  código comum
  id = fork();
  if( id != 0 )
    { código pai }
  else
    { código filho }
}
```



Memória Distribuída

Programação SPMD

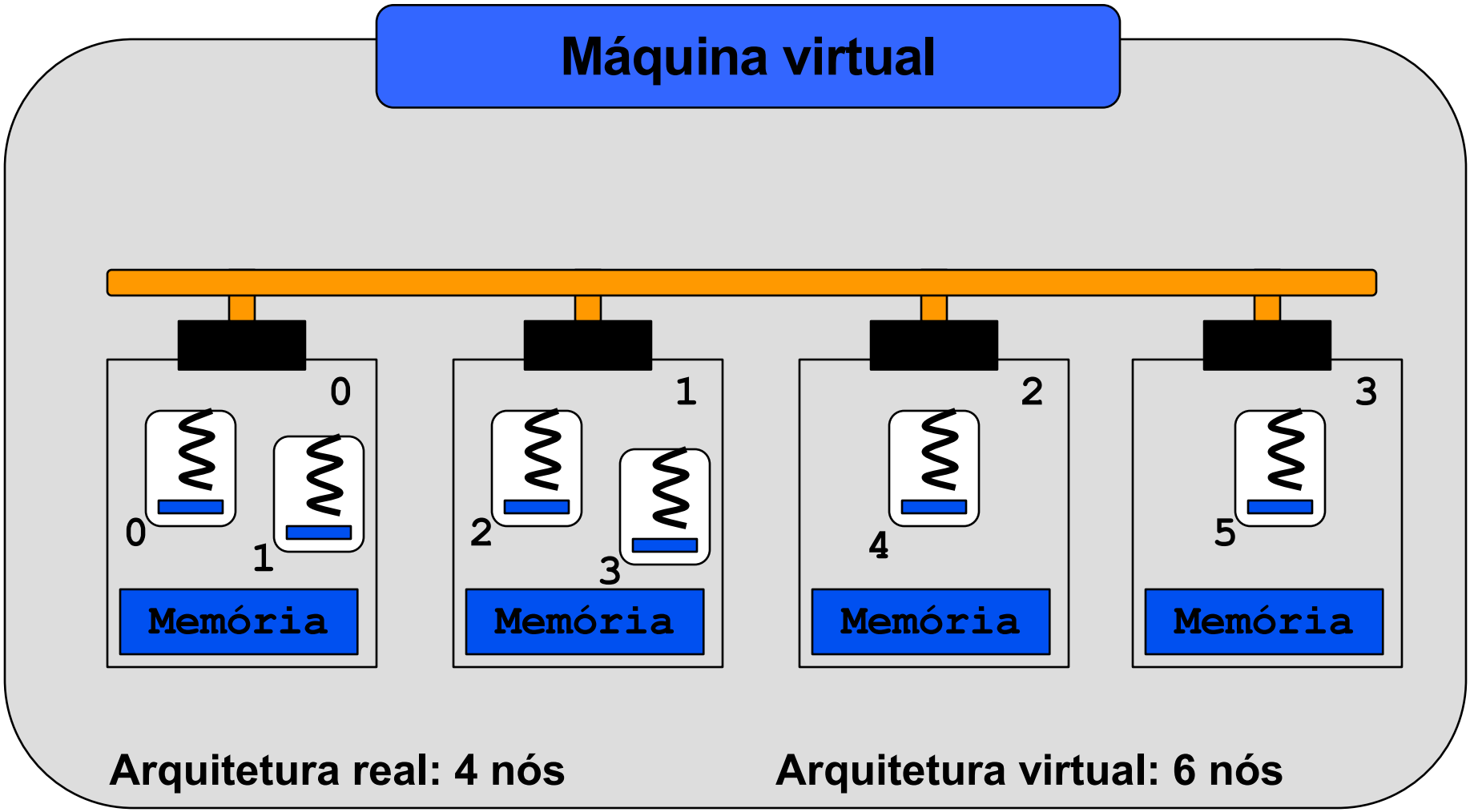
O `fork` no Unix

```
main() {
  int id;
  código comum
  id = fork();
  if( id != 0 )
    { código pai }
  else
    { código filho }
}
```

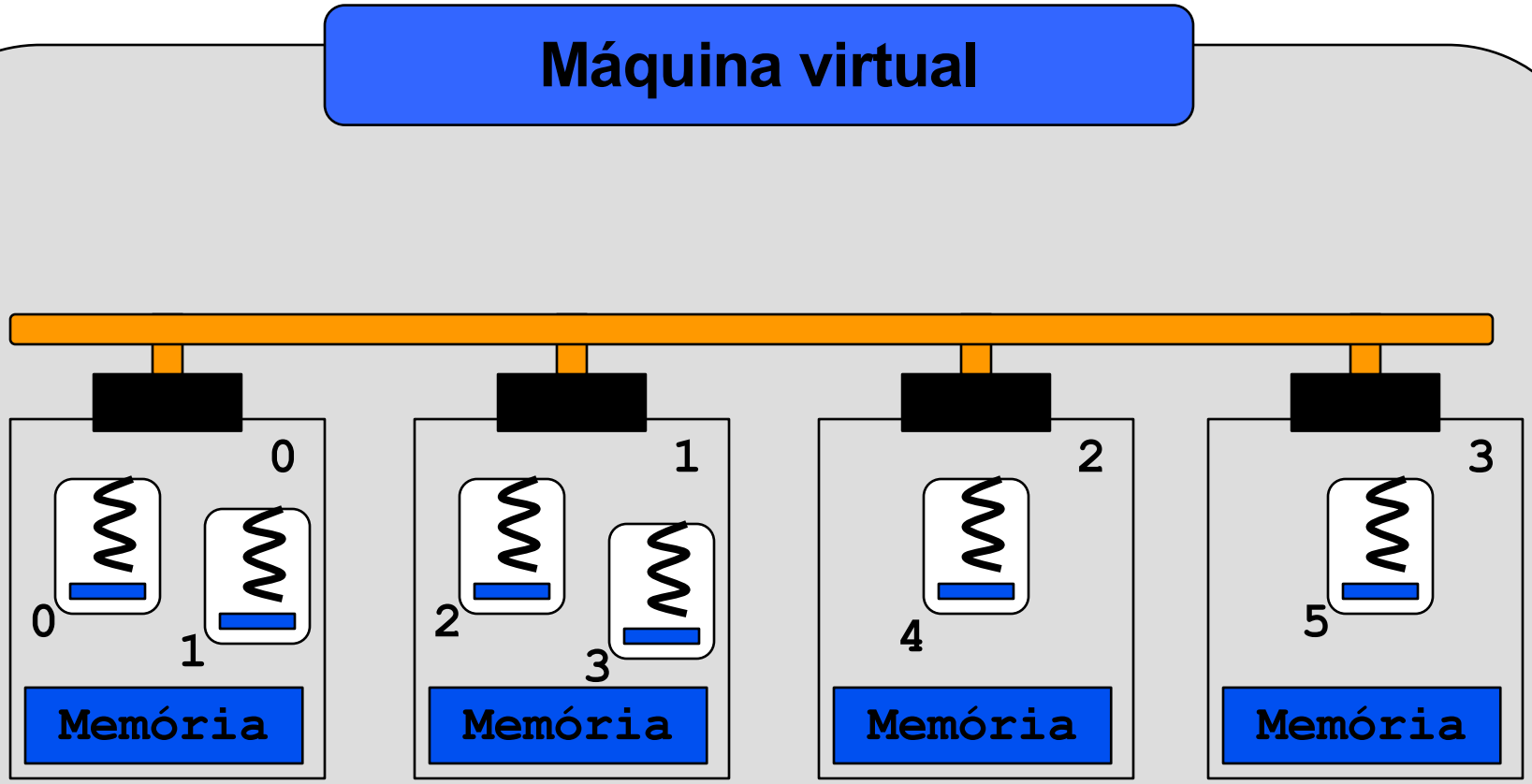
Diferença fundamental:

Em MPI os processos criado não possuem cópias idênticas da área de dados e diversas cópias podem ser criadas simultaneamente

Memória Distribuída



Memória Distribuída



Criada no lançamento da execução

Memória Distribuída

LAM básico

Não é encontrado em todas as distribuições Linux

Processo de make

`http://www.lam-mpi.org/`

Memória Distribuída

Memória Distribuída

LAM básico

Manipulação da máquina virtual

`lamboot`

`wipe`

`mpirun`

Inicialização do suporte à máquina virtual

`lamboot maqs`

Memória Distribuída

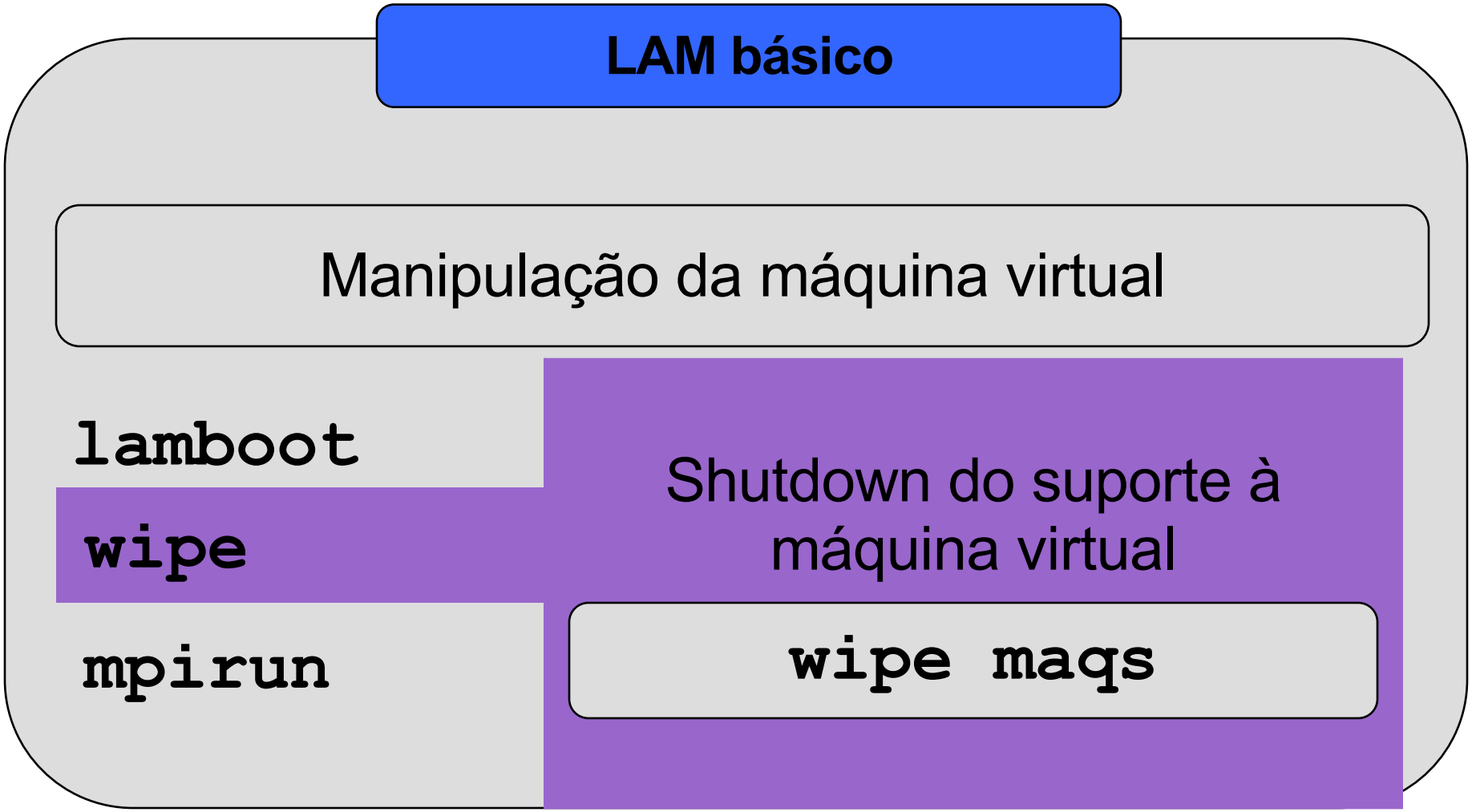
LAM básico

Manipulação da máquina virtual

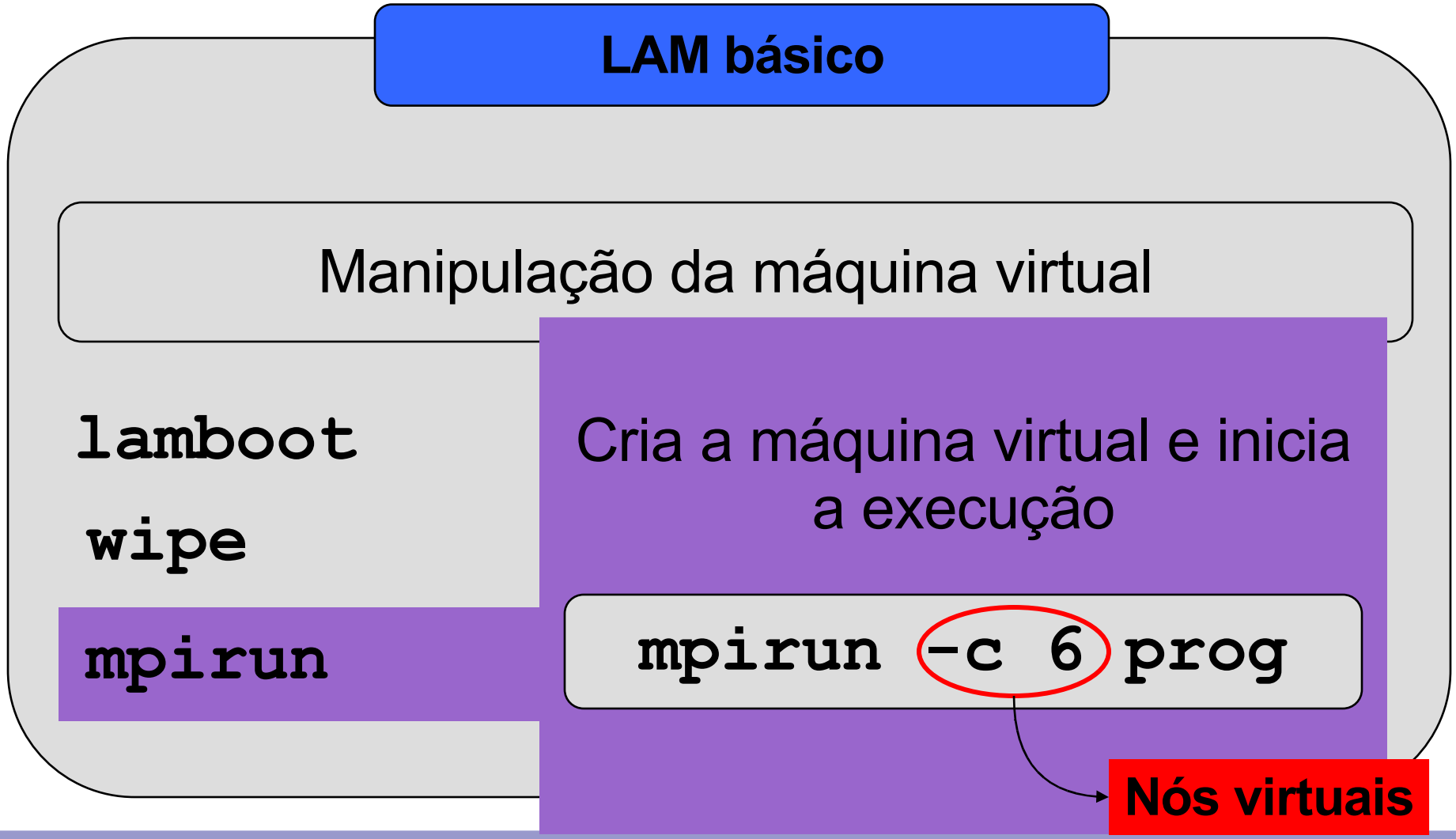
```
lamboot
wipe
mpirun
```

```
$ more maqs
clu0
clu1
clu2
clu3
$
```

Memória Distribuída



Memória Distribuída



Memória Distribuída

LAM básico

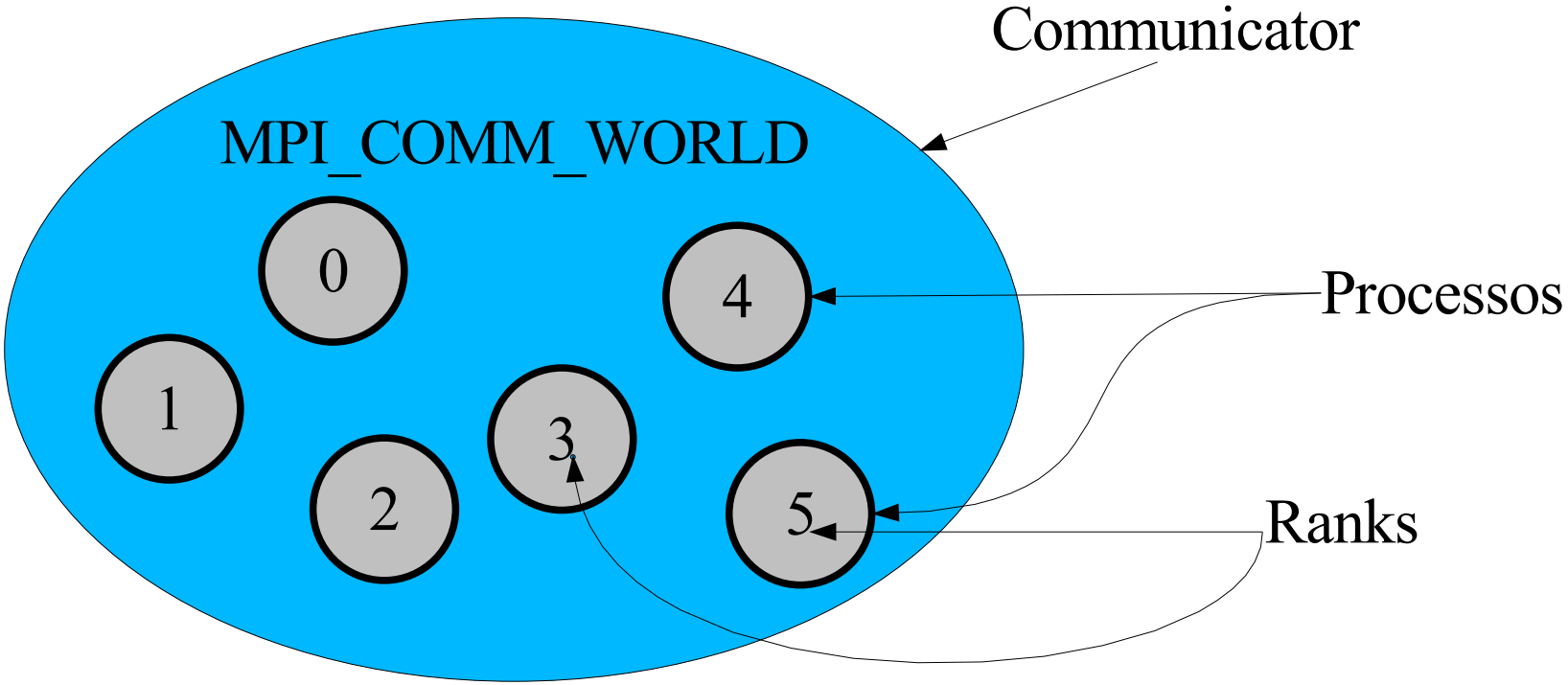
Primeiro programa

```
#include <mpi.h>


 MPI_init( &argc, &argv );
 MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
 MPI_Comm_size( MPI_COMM_WORLD, &size );


 MPI_Finalize();
```

Memória Distribuída

LAM básico

Rank e Communicator



Memória Distribuída

LAM básico

Primeiro programa

```
#include <mpi.h>
int main( int argc, char** argv ) {
 int myrank, size;
 MPI_init( &argc, &argv );
 MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
 MPI_Comm_size( MPI_COMM_WORLD, &size );
 if( myrank == 0 ) { código nó 0 }
 else { código dos outros nós }
 MPI_Finalize();
}
```

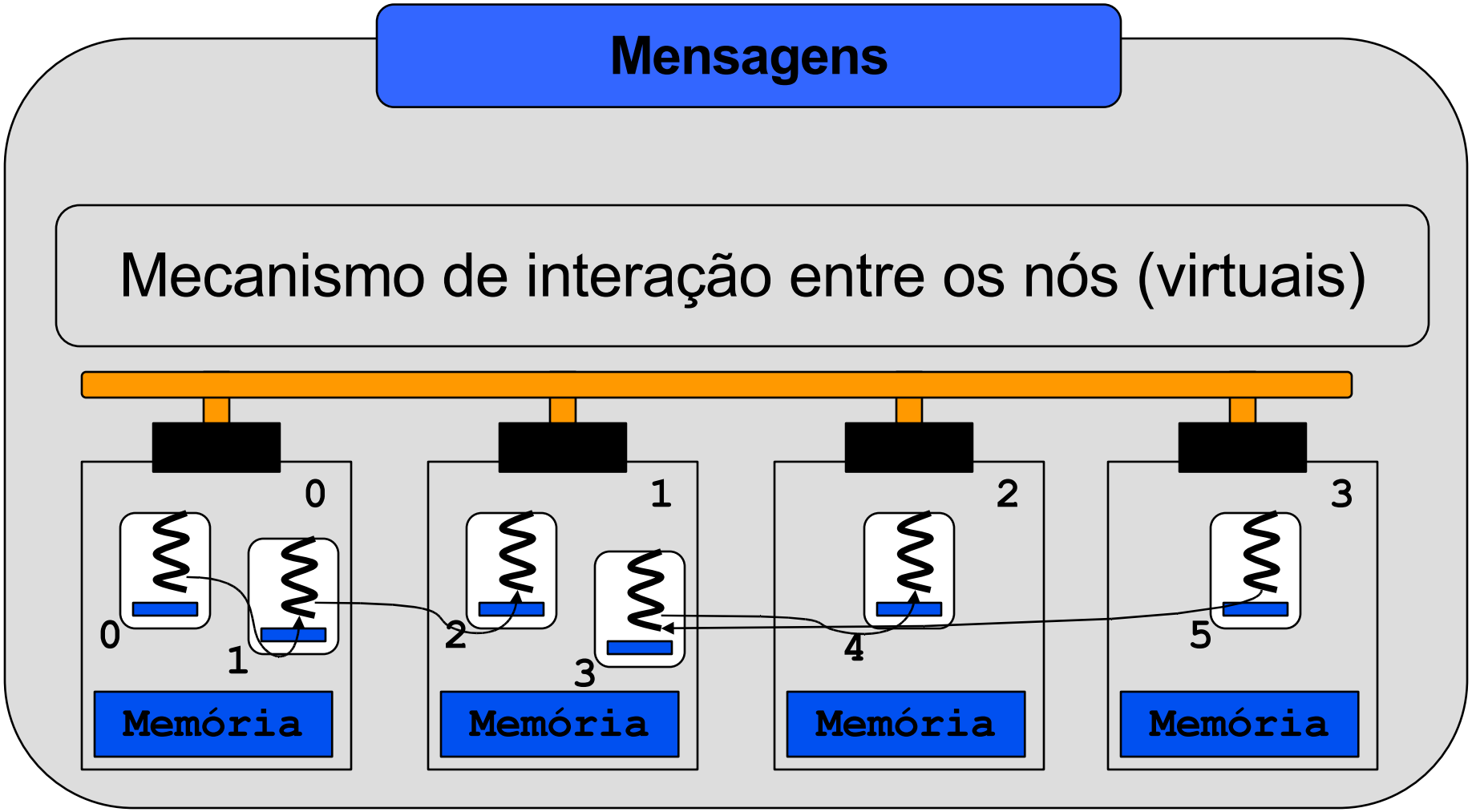
Memória Distribuída

LAM básico

Uma sessão típica

```
$ mpicc prog.c -o prog
$ lamboot maqs
$ mpirun -c 6 proc
$ wipe maqs
```

Memória Distribuída



Memória Distribuída

Mensagens

Formato

Origem	Destino	Tag	Dados

Identificação dos nós virtuais

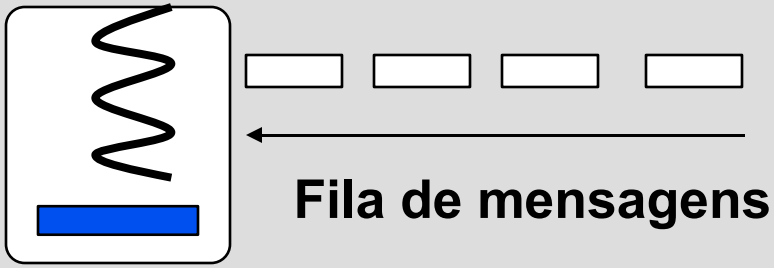
Permite a filtragem de mensagens no destino

Memória Distribuída

Mensagens

Formato

Origem	Destino	Tag	Dados



Fila de mensagens

As mensagens aguardam tratamento em uma fila. Os tags permitem escolher uma para ser tratada.

Memória Distribuída

Mensagens

Serviços básicos

Envio e recebimento bloqueante:

MPI_Send

Retorna quando o envio foi completado localmente.
Não quer dizer que tenha sido recebida!!

MPI_Recv

Retorna quando uma mensagem for recebida.

Memória Distribuída

Mensagens

Serviços básicos

Envio:

```
int MPI_Send( void *buf, int cont,
    MPI_Datatype tipo, int dest, int tag,
    MPI_Comm grupo );
```

Memória Distribuída

Mensagens

Serviços básicos

Envio:

```
int MPI_Send( void *buf, int cont,
    MPI_Datatype tipo, int dest, int tag,
    MPI_Comm grupo );
```

Recepção:

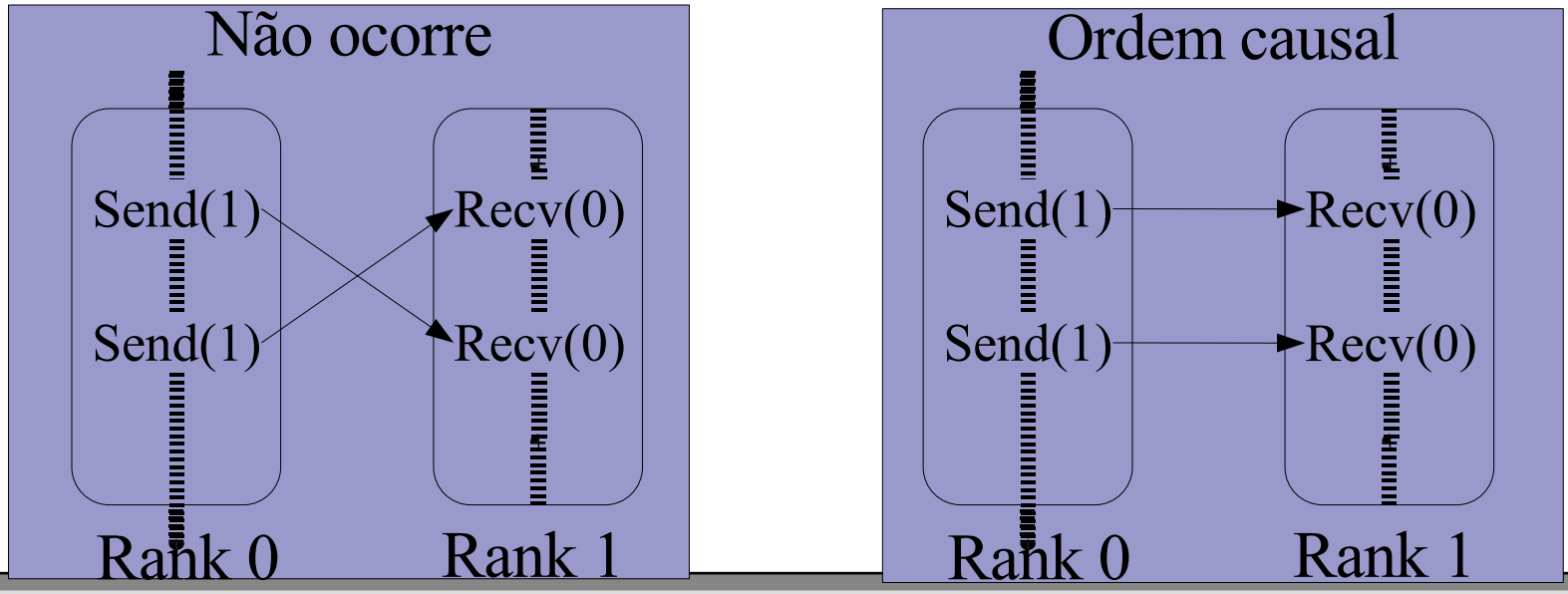
```
int MPI_Recv( void *buf, int cont,
    MPI_Datatype tipo, int origem, int tag,
    MPI_Comm grupo, MPI_Status *st );
```

Memória Distribuída

Mensagens

Envio e Recepção:

- A **ordem** entre mensagens enviadas entre dois processos é respeitada.



Memória Distribuída

Mensagens

MPI_Datatype

- MPI_CHAR
- MPI_DOUBLE
- MPI_FLOAT
- MPI_INT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_SHORT
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_SHORT

Memória Distribuída

Mensagens

Exemplo

Envio:

```
int vet[10];
MPI_Send( vet, 10, MPI_INT, 4, MPI_ANY_TAG,
          MPI_COMM_WORLD );
```

Memória Distribuída

Mensagens

Exemplo

Envio:

```
int vet[10];
MPI_Send( vet, 10, MPI_INT, 4, MPI_ANY_TAG,
          MPI_COMM_WORLD );
```

Recepção:

```
MPI_Status st;
int vet[10];
MPI_Recv( vet, 10, MPI_INT, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &st );
```

Memória Distribuída

Mensagens

Existem igualmente serviços não-bloqueantes

`MPI_Isend` `MPI_Irecv`

Isend : retorna mesmo que o dado não tenha sido enviado.

Irecv : retorna mesmo se não há mensagem.

Memória Distribuída

Mensagens

Exemplo

Envio:

```
MPI_Request req; MPI_Status st;
int vet[10];
MPI_Isend( vet, 10, MPI_INT, 4, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
... // faz outra coisa
MPI_Wait( &req, &st );
```

Recepção:

```
MPI_Request req; MPI_Status st;
int vet[10];
MPI_Irecv( vet, 10, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
           MPI_COMM_WORLD, &st );
... // faz outra coisa
MPI_Wait( &req, &st );
```

Memória Distribuída

Mensagens

Pack/Unpack

- MPI_CHAR
- MPI_DOUBLE
- MPI_FLOAT
- MPI_INT
- MPI_LONG
- MPI_LONG_DOUBLE
- MPI_SHORT
- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_UNSIGNED_SHORT
- MPI_PACKED

Memória Distribuída

Mensagens

Pack/Unpack

Empacotamento:

```
int MPI_Pack( void *dta, int cont,
    MPI_Datatype tipo, void *buff,
    int bsize, int *posic, MPI_Comm grupo );
```

Desempacotamento:

```
int MPI_Unpack( void *buff, int bsize,
    int *posic, void *dta, MPI_Datatype tipo,
    int bsize,  MPI_Comm grupo );
```

Memória Distribuída

Mensagens

Pack/Unpack (Exemplo)

```
// envio
int i, posic = 0;
char c[100], buffer[110];

MPI_Pack( &i,  1, MPI_INT,  buffer, 110, &posic, MPI_COMM_WORLD );
MPI_Pack( c, 100, MPI_CHAR, buffer, 110, &posic, MPI_COMM_WORLD );
MPI_Send( buffer, posic, MPI_PACKED, 1, 0, MPI_COMM_WORLD );

// Recepção
int i, posic = 0;
char c[100], buffer[110];
MPI_Status st;

MPI_Recv( buffer, 110, MPI_PACKED, 1, 0, MPI_COMM_WORLD );
MPI_Unpack( buffer, 110, &posic, &i,  1, MPI_INT, MPI_COMM_WORLD );
MPI_Unpack( buffer, 110, &posic, c, 100, MPI_INT, MPI_COMM_WORLD );
```

Exercício

- Faça um programa MPI que construa um anel entre os nós virtuais, de forma que um o nó 0 envia uma mensagem para o nó 1, o nó 1 para o nó 2, até que o nó Comm_size-1 receba a mensagem e envie a mensagem para o nó 0. O corpo da mensagem é um valor inteiro, cujo valor na primeira mensagem (nó 0 para nó 1) é igual a 0. A cada nó, ao receber o valor, adiciona seu proprio Comm_rank ao valor antes de reenvia-lo.

Memória Distribuída

Modos de Comunicação

- Standard
  - Send/Receive Isend/Irecv
- Bufferizado
  - Bsend
- Síncrono
  - Ssend
- Ready
  - Rrecv

Memória Distribuída

Modos de Comunicação

- Modo standard
  - MPI_Send/MPI_Recv e MPI_Isend/MPI_Irecv.
  - O MPI é responsável por bufferizar as mensagens.
  - A operação de send pode ser iniciada e terminada mesmo se a operação receive não tiver sido postada.
  - A operação send é não-local: é concluída com sucesso somente na ocorrência de um receive.

Memória Distribuída

Modos de Comunicação

- Modo bufferizado
  - MPI_Bsend
  - A operação de send pode ser iniciada mesmo se a operação receive não tiver sido postada.
  - A operação send é local: conclui com sucesso independente da ocorrência de um receive.
  - As mensagens devem ser bufferizadas no processo que executou o send.
    - Bufferização responsabilidade da aplicação: MPI_Buffer_attach/dettach

Memória Distribuída

Modos de Comunicação

- Modo síncrono
  - MPI_Ssend
  - Rendezvous
  - A operação de send pode ser iniciada mesmo se a operação receive não tiver sido postada.
  - A operação send é não-local: conclui com sucesso apenas com a ocorrência de um receive.

Memória Distribuída

Modos de Comunicação

- Modo ready
  - MPI_Rsend
  - A operação de send pode ser iniciada apenas se a operação receive já tiver sido postada.
  - Aumento de desempenho.

Memória Distribuída

Modos de Comunicação

- Outras rotinas
  - MPI_Ibsend
  - MPI_Issend
  - MPI_Irsend
- Discussão
  - O melhor desempenho é obtido com o modo ready, no entanto a programação é complexa. Um desempenho bastante interessante é obtido com o modo síncrono, que permite evitar a bufferização desnecessária de mensagens. O modo standard é o mais flexível, ao custo de legar à implementação do MPI a gestão dos buffers. O modo bufferizado permite otimizar o uso da bufferização, também com custo de programação.

Memória Distribuída

Mensagens

Comunicação de grupo

- Barreira
- Broadcast: um envia, todos recebem
- Reduce: todos enviam, um recebe aplicando uma operação
- Gather: recebe em um array dados distribuídos
- Scatter: distribui dados de um array
- All-to-all: troca de dados entre todos
- Scan: uma redução pré-fixada

Memória Distribuída

Mensagens

Comunicação de grupo

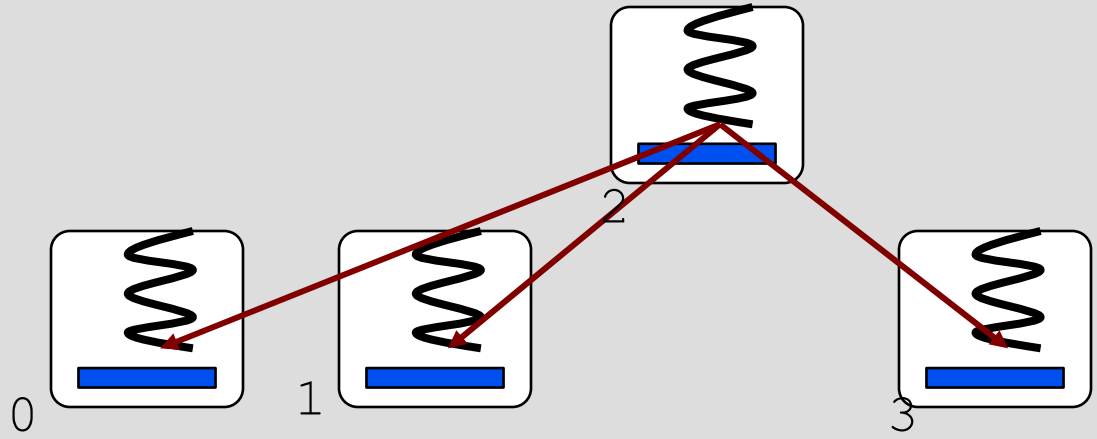
Envio e recepção:

```
MPI_Bcast( void *buf, int cont, MPI_Datatype t,
           int raiz, MPI_Comm grupo);
```

Memória Distribuída

Mensagens

Comunicação de grupo




```
MPI_Bcast( dta, 1, MPI_INT, 2, MPI_COMM_WORLD );
```

Memória Distribuída

Mensagens

Comunicação de grupo

Envio e recepção:

```
MPI_Reduce(void *sendbuf, void *recvbuf,
           int count,
           MPI_Datatype t, MPI_Op op,
           int raiz, MPI_Comm grupo);
```

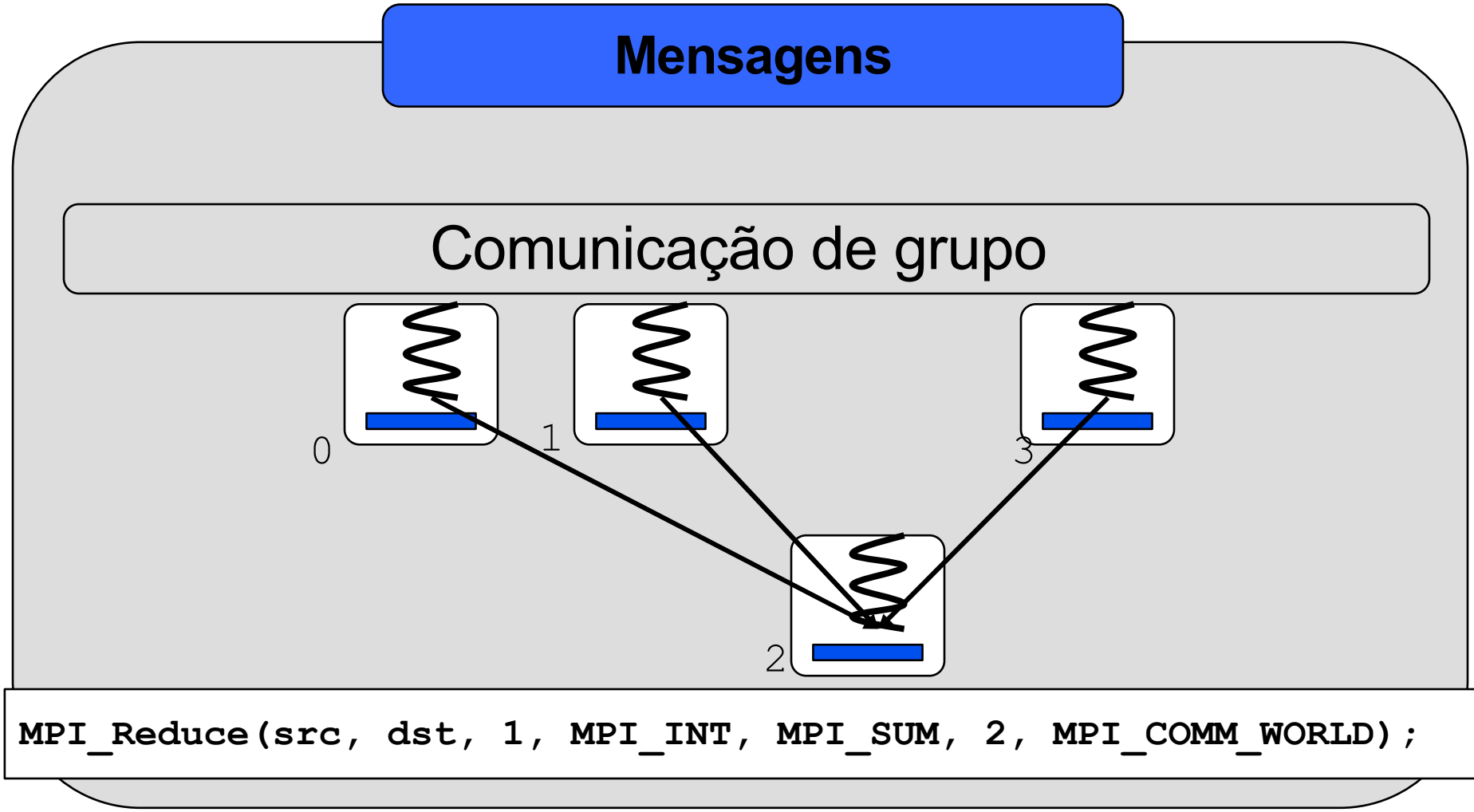
Memória Distribuída

Mensagens

MPI_Op

- MPI_BAND
- MPI_BOR
- MPI_BXOR
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_MAX
- MPI_MAXLOC
- MPI_MIN
- MPI_MINLOC
- MPI_PROD
- MPI_SUM

Memória Distribuída



Memória Distribuída

Mensagens

Comunicação de grupo

Envio e recepção:

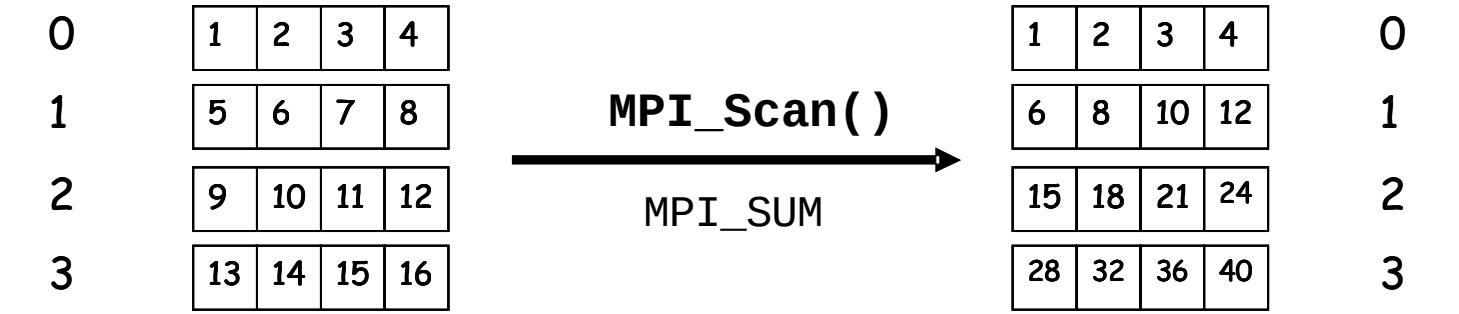
```
MPI_Scan(void *sendbuf, void *recvbuf, int count,
         MPI_Datatype t, MPI_Op op, MPI_Comm grupo);
```

Memória Distribuída

Mensagens

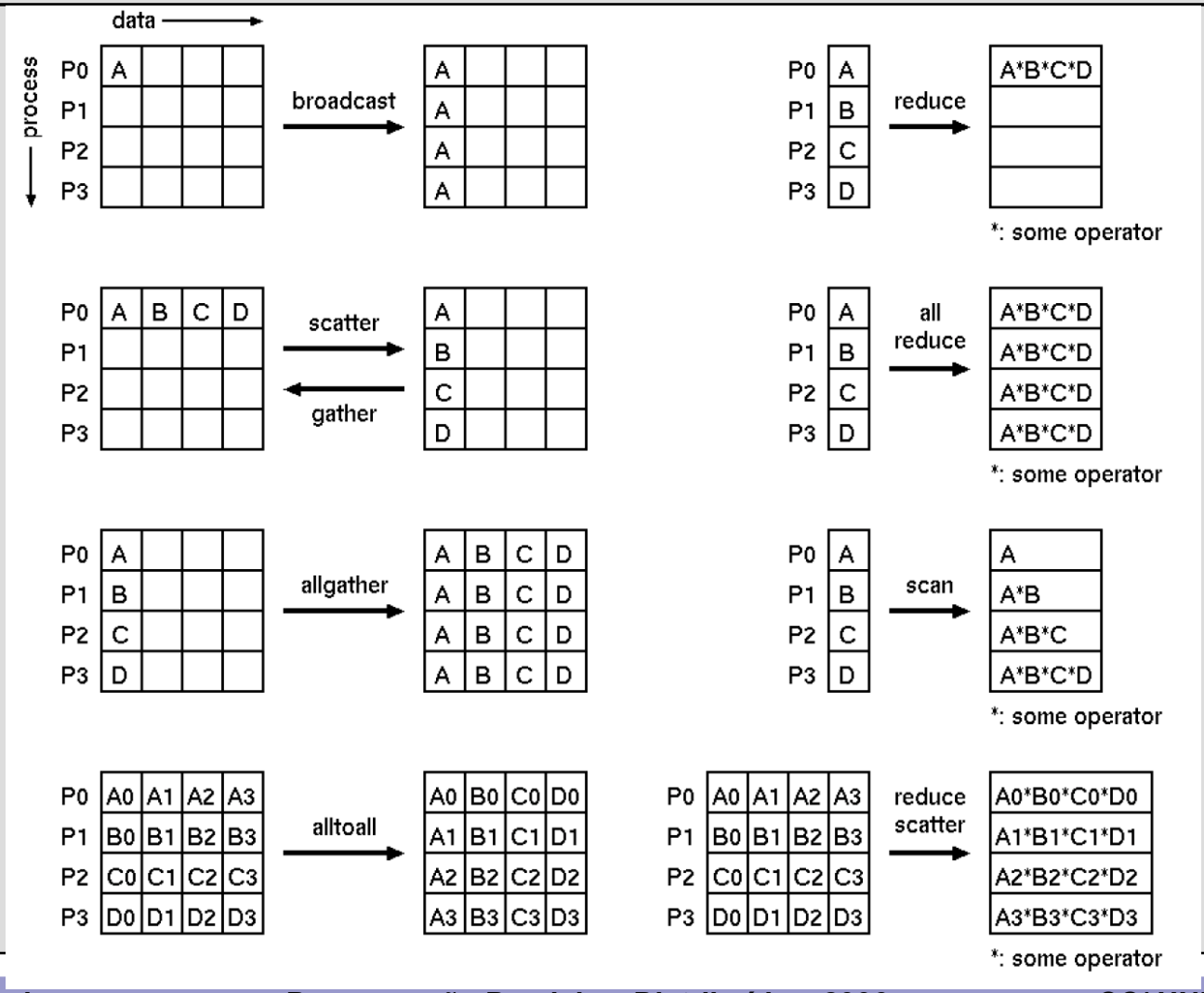
Comunicação de grupo

MPI_Scan



Memória Distribuída

Padrões de Comunicação de grupo



IBM Redbook

Memória Distribuída

Tipos de dados derivados

No envio de mensagens, pode ser necessário enviar tipos de dados definidos pelo programador. Duas casos podem ser observados:

- Uma seqüencia de dados do mesmo tipo
  - *Contiguous derived data types*
- Dados são compostos por diferentes tipos
  - *Structured derived data types*

Memória Distribuída

Tipos de dados derivados

Primitivas principais:

- *MPI_Type_contiguous()* - Constroi um tipo continuo
- *MPI_Type_struct()* - Constroi um tipo estruturado
- *MPI_Type_vector()*
- *MPI_Type_indexed()*

Primitivas auxiliares:

- *MPI_Type_extent()* - Retorna o tamanho de um tipo
- *MPI_Type_commit()* - autoriza o uso

Memória Distribuída

Tipos de dados derivados

Exemplo: Contiguous devired data type

```
#define SIZE 9
int numbers[SIZE] = {2,5,6,8,9,7,1,3,10};
int numbers_slave[SIZE];
MPI_Datatype mytype;
MPI_Init(&argc, &argv);
MPI_Type_contiguous(SIZE/numproc, MPI_INT, &mytype);
MPI_Type_commit(&mytype):
if (myid == 0)
  for (i=1; i<numproc; ++i)
    MPI_Send(&numbers[(SIZE/numproc)*i],1,mytype, MPI_COMM_WORLD);
else
  MPI_Recv(&numbers_slave, 1, mytype, 0, 0, MPI_COMM_WORLD, &Stat);
```

```
MPI_Type_contiguous( int count,
                     MPI_Datatype orig,
                     MPI_Datatype *novo );
```

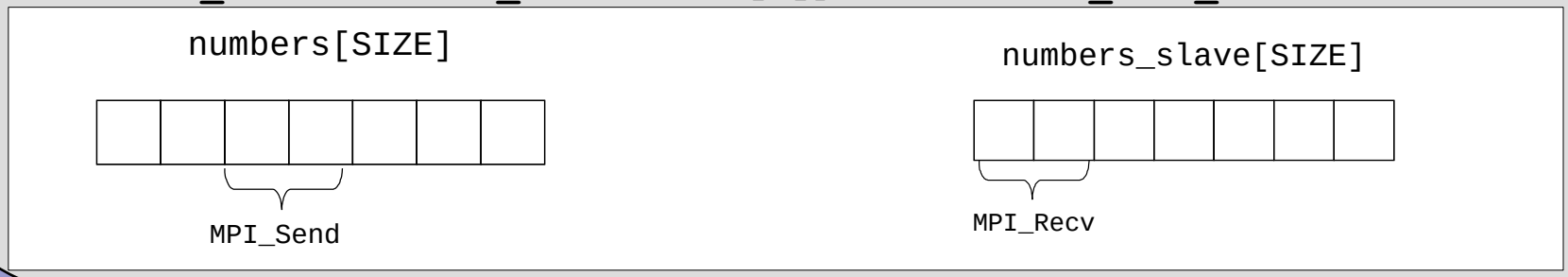
Memória Distribuída

Tipos de dados derivados

Exemplo: Contiguous devired data type

```
#define SIZE 9
int numbers[SIZE] = {2,5,6,8,9,7,1,3,10};
int numbers_slave[SIZE];
MPI_Datatype mytype;
MPI_Init(&argc, &argv);
MPI_Type_contiguous(SIZE/numproc, MPI_INT, &mytype);
MPI_Type_commit(&mytype):
if (myid == 0)
  for (i=1; i<numproc; ++i)
    MPI_Send(&numbers[(SIZE/numproc)*i],1,mytype, MPI_COMM_WORLD);
else
  MPI_Recv(&numbers_slave, 1, mytype, 0, 0, MPI_COMM_WORLD, &Stat);
```

```
MPI_Type_contiguous( int count,
                     MPI_Datatype orig,
                     MPI_Datatype *novo );
```



Memória Distribuída

Tipos de dados derivados

Exemplo: Contiguous devired data type

```
int MPI_Type_struct( int count, int blocklens[], MPI_Aint indices[],
                     MPI_Datatype old_types[], MPI_Datatype *newtype )
```

Entrada:

count	número de blocos (integer)
blocklens	número de elementos em cada bloco (array)
indices	deslocamento para cada bloco no dado (array)
old_types	MPI_Datatype associado a cada bloco (array)

Memória Distribuída

Tipos de dados derivados

Exemplo: Contiguous derived data type

```
typedef struct {int x, y; float z;} message;
message slave_buf[SIZE], master_buf[SIZE];
MPI_Datatype mytype;
MPI_Datatype oldtypes[2] = {MPI_INT, MPI_FLOAT};
int blocklens[2] = {2,1};
MPI_Aint indicies[2], length;
…
MPI_Type_extent(MPI_INT, &length);
indicies[0]=0;
indicies[1]=2*length;
MPI_Type_struct(2, blocklens, indicies, oldtypes, &mytype);
MPI_Type_commit(&mytype);
```

Memória Distribuída

Criação dinâmica de processos

- Criação e término dinâmico de processos
  - Operações em grupo
- Comunicação entre aplicações MPI
  - Cliente / Servidor

Memória Distribuída

Criação dinâmica de processos

- Criação e término dinâmico de processos.
  - MPI_Comm_spawn

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs,
          MPI_Info info, int root, MPI_Comm comm,
          MPI_Comm *intercomm, int array_of_errcd[]);
```

  - MPI_Comm_spawn_multiple

```
int MPI_Comm_spawn_multiple(int count, char array_of_commands[],
           char **array_of_argv[], int array_of_maxprs[],
           MPI_Info array_of_info[], int root, MPI_Comm comm,
           MPI_Comm *intercomm, int array_of_errcodes[]);
```

Memória Distribuída

Criação dinâmica de processos

```
int MPI_Comm_spawn(char *command, char *argv[], int maxprocs,
                   MPI_Info info, int root, MPI_Comm comm,
                   MPI_Comm *intercomm, int array_of_errcd[])
```

- command — comando para executar o programa (válido no root)
- argv — argumentos para o comando (um para cada instância, válido no root)
- maxprocs — número de cópias para serem inicializadas (válido no root)
- info — informações adicionais para criação do processo (dependente de implementação, válido no root))
- root — identificação do processo root
- comm — intercomunicador para os novos processos
- intercomm — intercommunicator entre o grupo original e o novo grupo
- array_of_errcds — código de erro, um por processo

Memória Distribuída

Estabelecimento de comunicação

- Comunicação entre processos que não compartilham comunicador
  - Uso:
    - Aplicação iniciada através de módulos independentes
    - Integração com uma ferrametna de visualização/profiling
    - Um servidor pode aceitar conexões de múltiplos clientes
      - Tanto servidor como cliente podem ser programas paralelos
- É possivel criar um intercomunicador entre dois grupos de processos.
- Operação coletiva, mas assimétrica: um grupo (servidor) indica que aceita conexões de outros grupos (clientes).

Memória Distribuída

Estabelecimento de comunicação

Funções do Servidor

- `int MPI_Open_port(MPI_Info info, char *port_name);`
  - Estabelece um endereço de rede, codificado em port_name string, pelo qual o servidor pode aceitar conecções de clientes. port_name é fornecido pelo sistema, possivelmente utilizando informações de info.
- `int MPI_Close_port(char *port_name);`
  - Fecha uma conexão representada por port_name
- `int MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm);`
  - Estabelece uma comunicação com um cliente. É uma operação coletiva sobre o comunicador. Retorna um intercomunicador que permite a comunicação com o cliente.

Memória Distribuída

Estabelecimento de comunicação

Função do Cliente

- `int MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm);`
  - Estabelece uma comunicação com o servidor especificado em port_name. É uma operação coletiva e retorna um intercommunicador com o grupo que participou do MPI_COMM_ACCEPT

Memória Distribuída

Estabelecimento de comunicação

Publicando um serviço

- `int MPI_Publish_name(char *service_name, MPI_Info info, char *port_name);`
  - Esta rotina publica o par(port_name, service_name) de forma que uma aplicação pode encontrar uma deterninada porta (port_name) utilizando o nome de um serviço conhecido (service_name).
- `int MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name);`
  - Permite procurar a porta associada a um serviço.
- `int MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name);`
  - Retira a publicação de um serviço.

Memória Distribuída

Exemplo de estabelecimento de conexão

Sem serviço de nomes: o servidor imprime o port name e o usuário precisa informar no cliente.

Servidor:

```
char myport[MPI_MAX_PORT_NAME];
MPI_Comm intercomm;
/* ... */
MPI_Open_port(MPI_INFO_NULL, myport);
printf("port name is: %s\n", myport);

MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
/* do something with intercomm */
```

Cliente:

```
MPI_Comm intercomm;
char name[MPI_MAX_PORT_NAME];
printf("enter port name: ");
gets(name);
MPI_Comm_connect(name, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
```

Outros exemplos em:
http://www.mpi-forum.org/docs/mpi-20-html/node106.htm

E/S Paralela

- Múltiplas instâncias de um programa paralelo acessando um mesmo arquivo.
- Alternativas não paralelas:
  - Todos os processos mandam os dados para um processo servidor (como o processo rank 0) e este processo se encarrega de escrever
  - Cada processo opera sua saída de dados em um arquivo próprio e após é feita a união dos resultados

E/S Paralela

- Alternativas não paralelas são simples, mas:
  - Oferecem baixo desempenho (um único processo escreve o arquivo)
  - Baixo índice de interoperabilidade (cada processo escrevendo em próprio arquivo)
- E/S Paralela
  - Melhor desempenho
  - Um único arquivo de saída pode ser melhor operado por outras aplicações colaboradores (ex.: visualização)

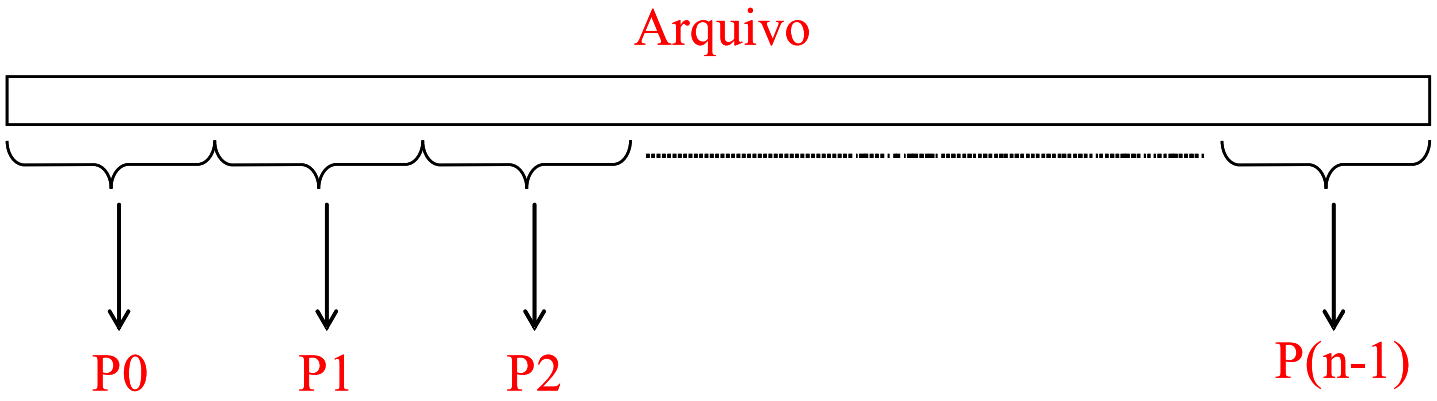
E/S Paralela

E/S Paralela em MPI

- Escrita/Leitura
  - Similares ao Envio/Recebimento de mensagems
- Em essência, operações de E/S paralela reflete a estrutura de MPI:
  - Define operações coletivas (MPI communicators)
  - Define layout de dados não contíguos em memória e em arquivo (MPI datatypes)
  - Necessita de teste de término de operações não bloqueantes (MPI request objects)

E/S Paralela

E/S Paralela em MPI



Cada processo necessita ler um bloco de dados a partir de um arquivo compartilhado

E/S Paralela

E/S Paralela em MPI

Operações:

- **MPI_File_open**
  - **MPI_MODE_CREATE**
  - **MPI_MODE_WRONLY**
  - **MPI_MODE_RDWR**
- **MPI_File_seek**
- **MPI_File_close**
- **MPI_File_write**
- **MPI_File_write_at**

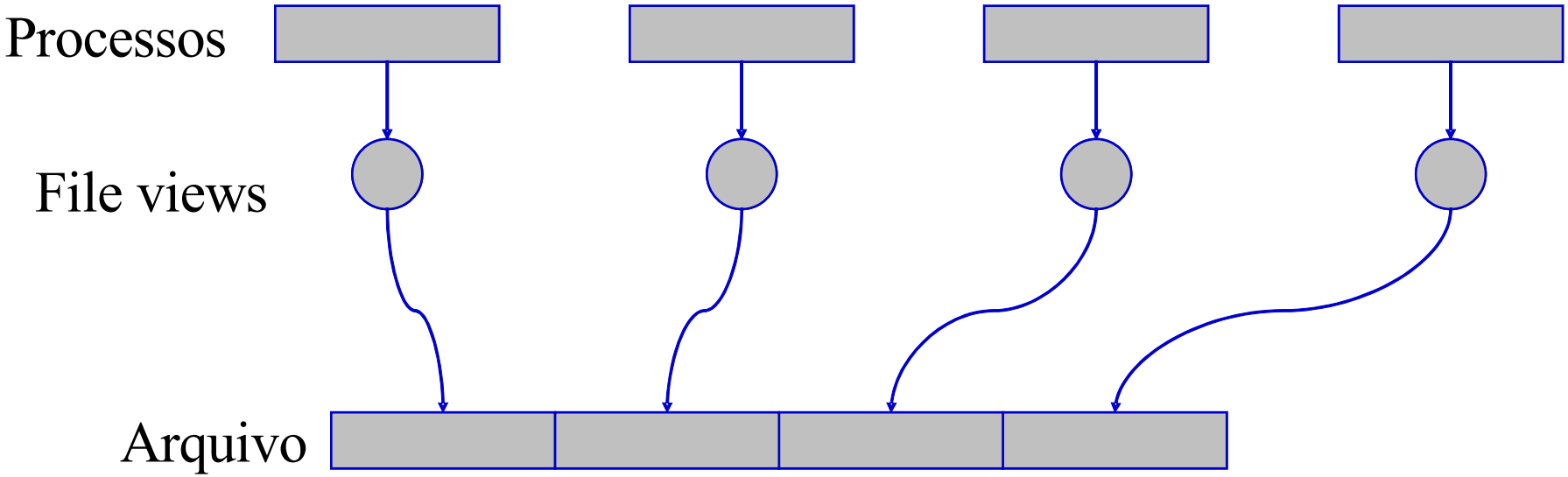
E/S Paralela

E/S Paralela em MPI

```
MPI_File fh;
MPI_Status status;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
                    MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

E/S Paralela

File Views em MPI



`MPI_File_set_view` define regiões visíveis a processos MPI

E/S Paralela

File Views em MPI

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype,
                      MPI_Datatype filetype,
                      char *datarep, MPI_Info info);
```

- **File views** são especificados por uma tupla contendo três informações:
  - displacement: número de bytes a serem disconsiderados no início do arquivo
  - etype: unidade básica de acesso (tipo primitivo ou derivado)
  - filetype: especifica qual porção do arquivo é visível ao processo
- datarep: native, internal ou external32

E/S Paralela

E/S Paralela em MPI

```
MPI_File thefile;

for (i=0; i<BUFSIZE; i++)
    buf[i] = myrank * BUFSIZE + i;
MPI_File_open(MPI_COMM_WORLD, "testfile",
              MPI_MODE_CREATE | MPI_MODE_WRONLY,
              MPI_INFO_NULL, &thefile);
MPI_File_set_view(thefile,myrank*BUFSIZE*sizeof(int),
          MPI_INT, MPI_INT, "native", MPI_INFO_NULL);
MPI_File_write(thefile, buf, BUFSIZE, MPI_INT,
               MPI_STATUS_IGNORE);
MPI_File_close(&thefile);
```