Computação Concorrente (DCC/UFRJ)

Aula 12: Programação concorrente com memória distribuída usando MPI

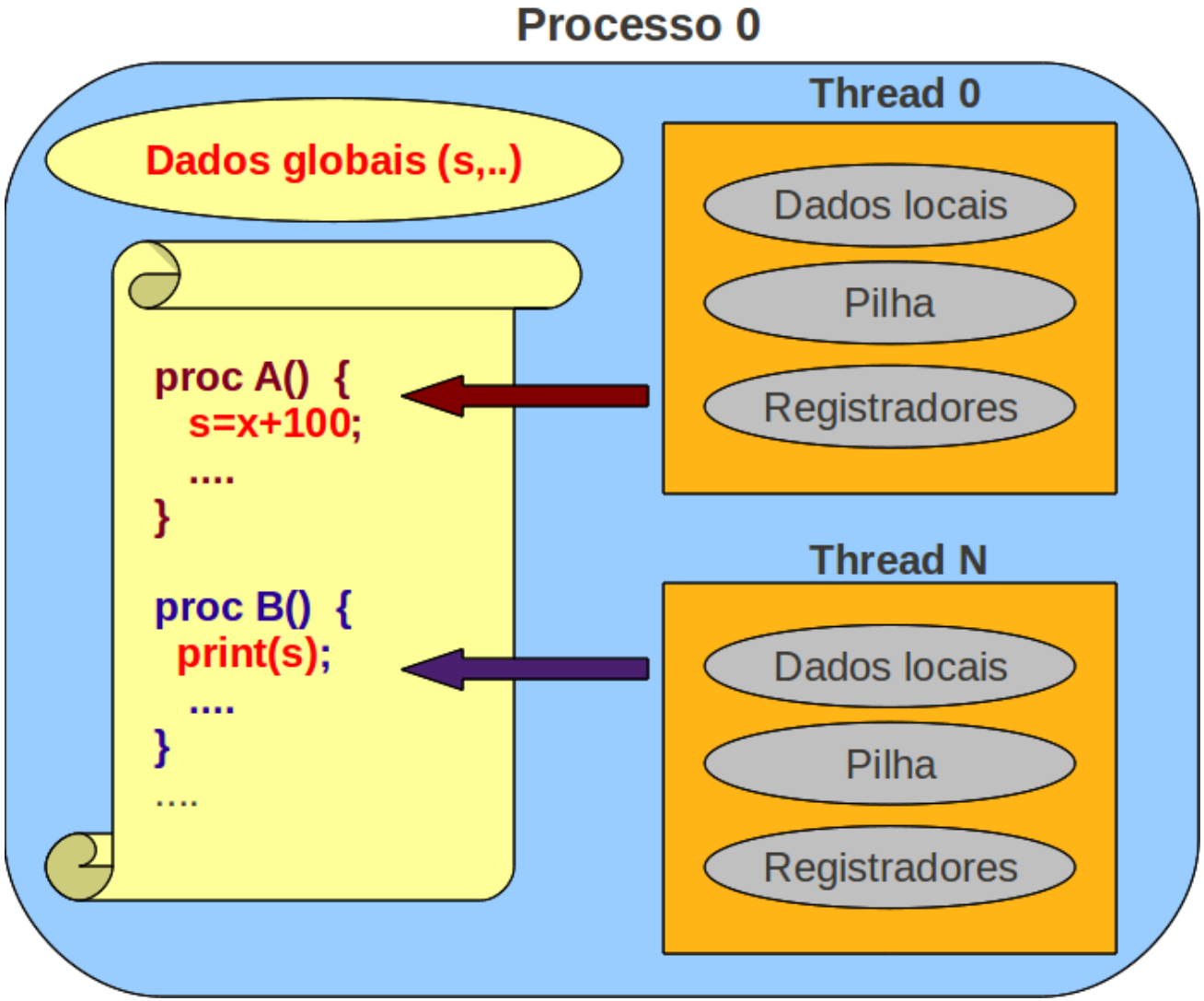
Prof. Silvana Rossetto

26 de janeiro de 2016

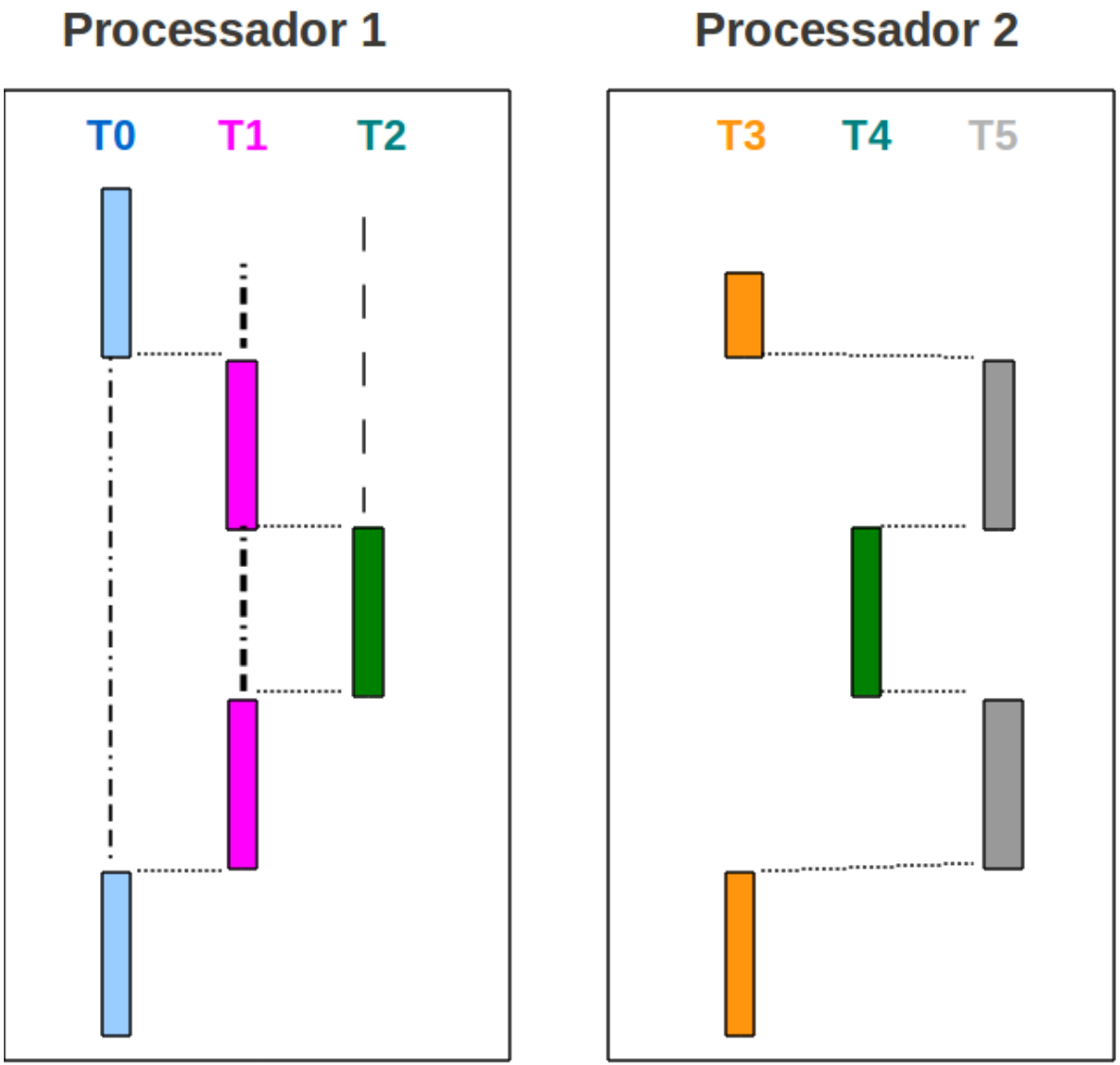
Caminhos para construção de programas concorrentes

1. Aplicação com **várias threads** (multitarefa preemptiva, memória compartilhada)
2. Aplicação com **vários processos** (memória distribuída)

Aplicações com várias threads

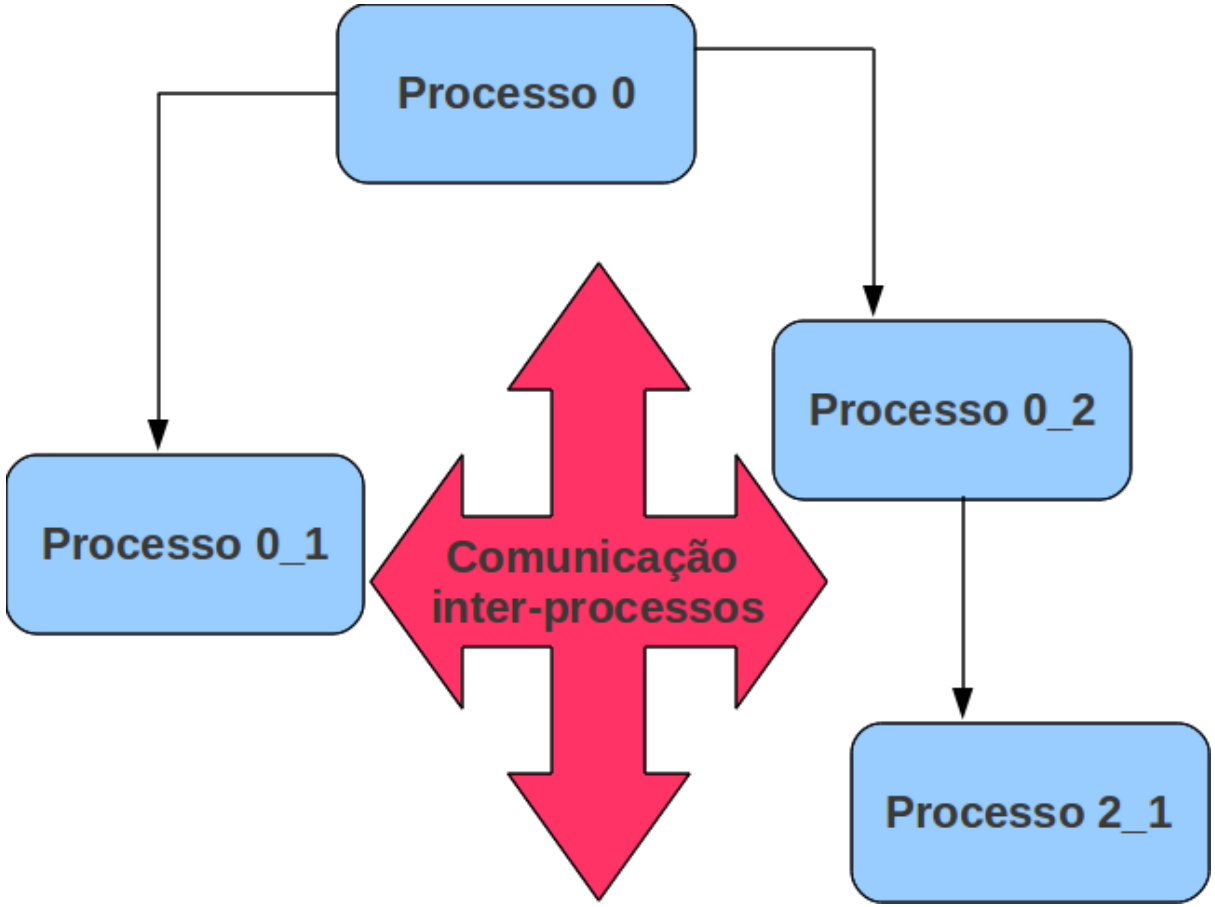


Alternância **implícita** entre threads

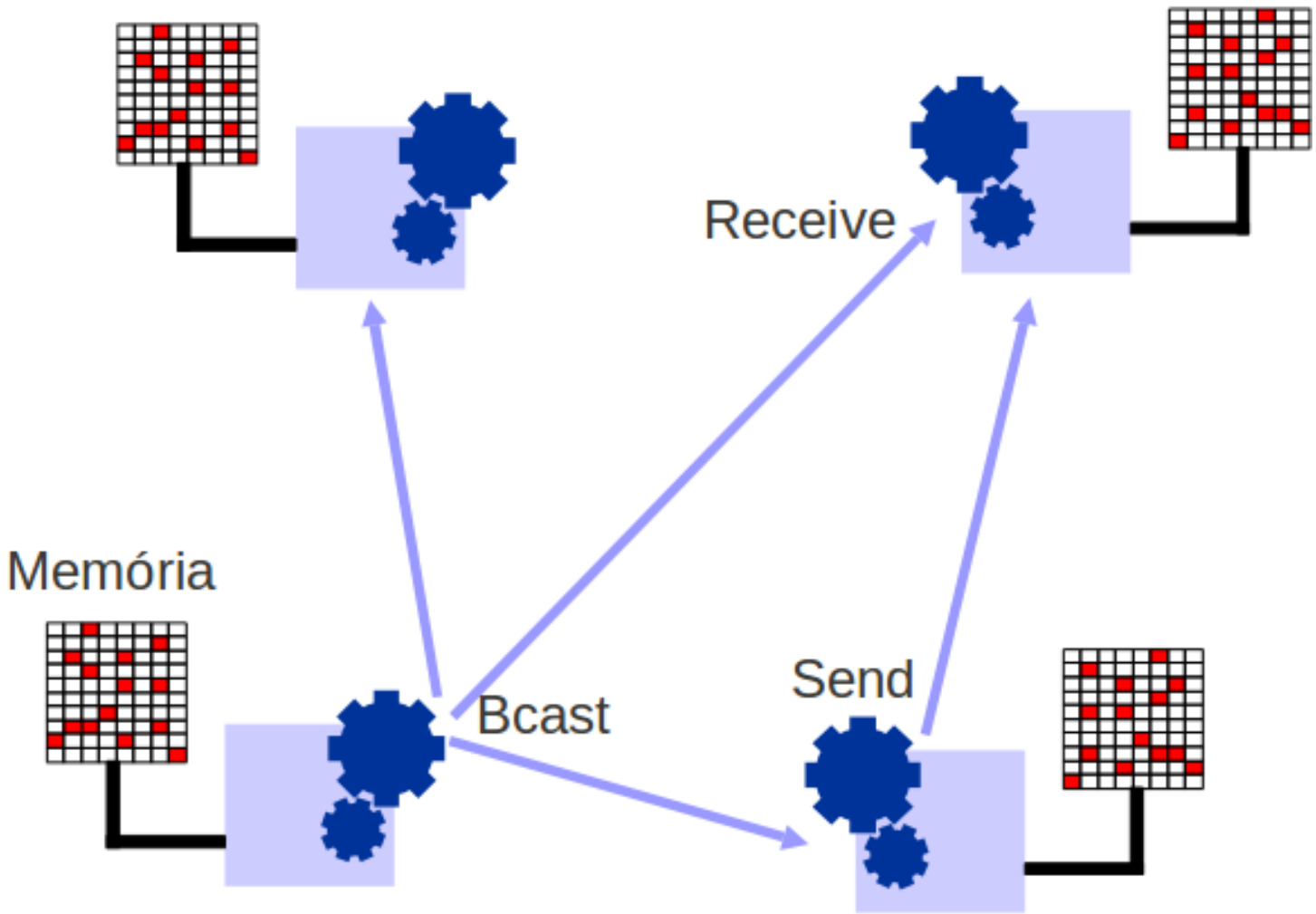


Threads de uma aplicação

Aplicações com vários processos



Primitivas para troca de mensagens entre processos



Interação entre processos

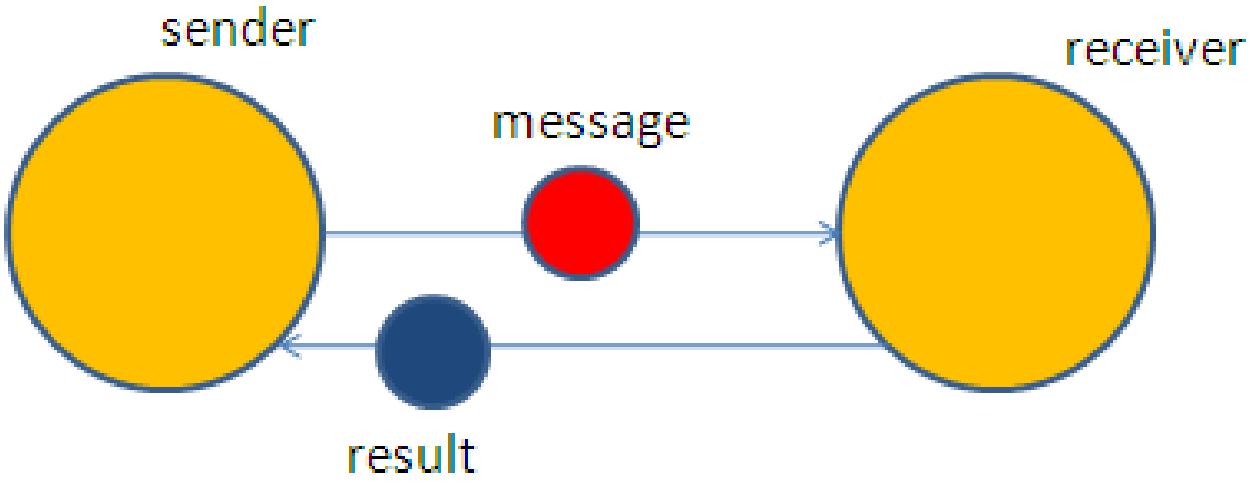
Requisitos básicos incorporados nas primitivas de comunicação

- **Comunicação**: permitir a troca de informações entre processos
- **Sincronização**: garantir acesso exclusivo aos recursos compartilhados

Abordagem com troca de mensagens

Uso de **primitivas de troca de mensagens**: podem ser usadas em ambientes de memória compartilhada ou máquinas distribuídas

Sistemas de troca de mensagens



1

Primitivas de troca de mensagens

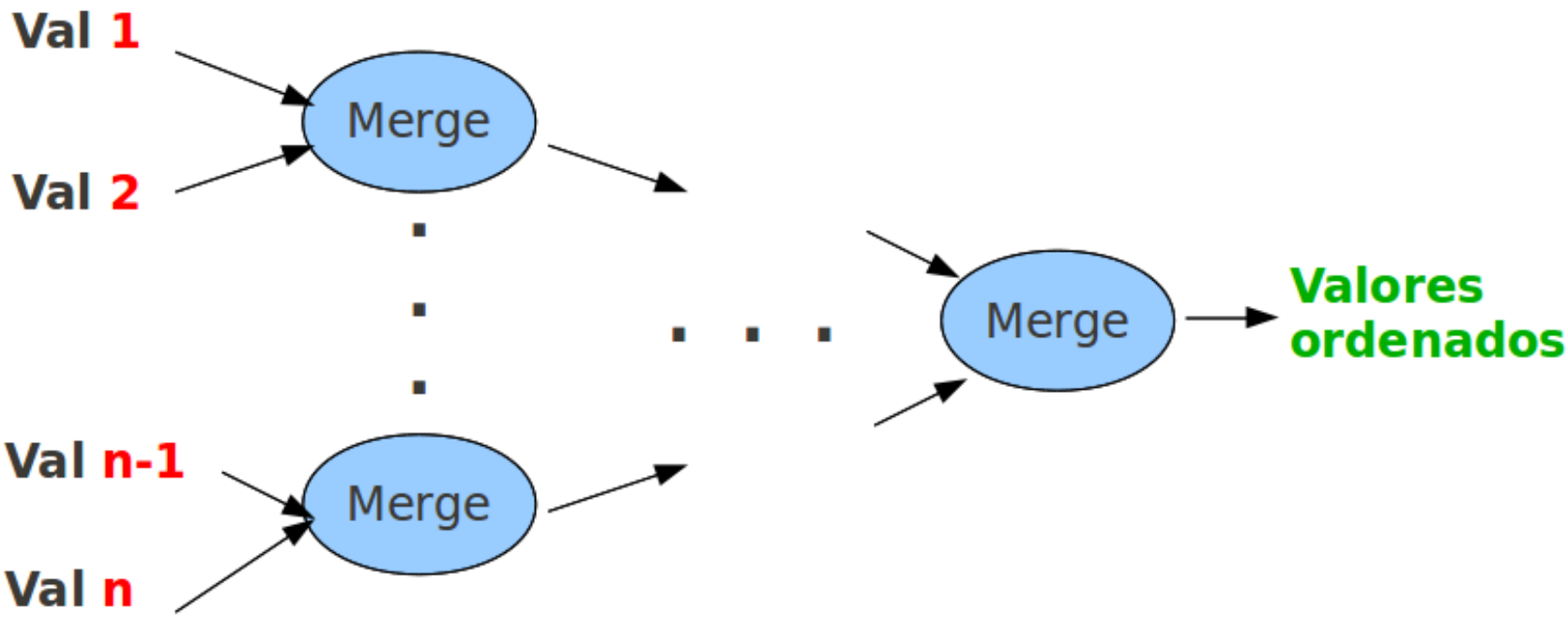
- **send (dst, msg)**
- **receive (src, msg)**

[1] *Fonte: http://ajlopez.wordpress.com*

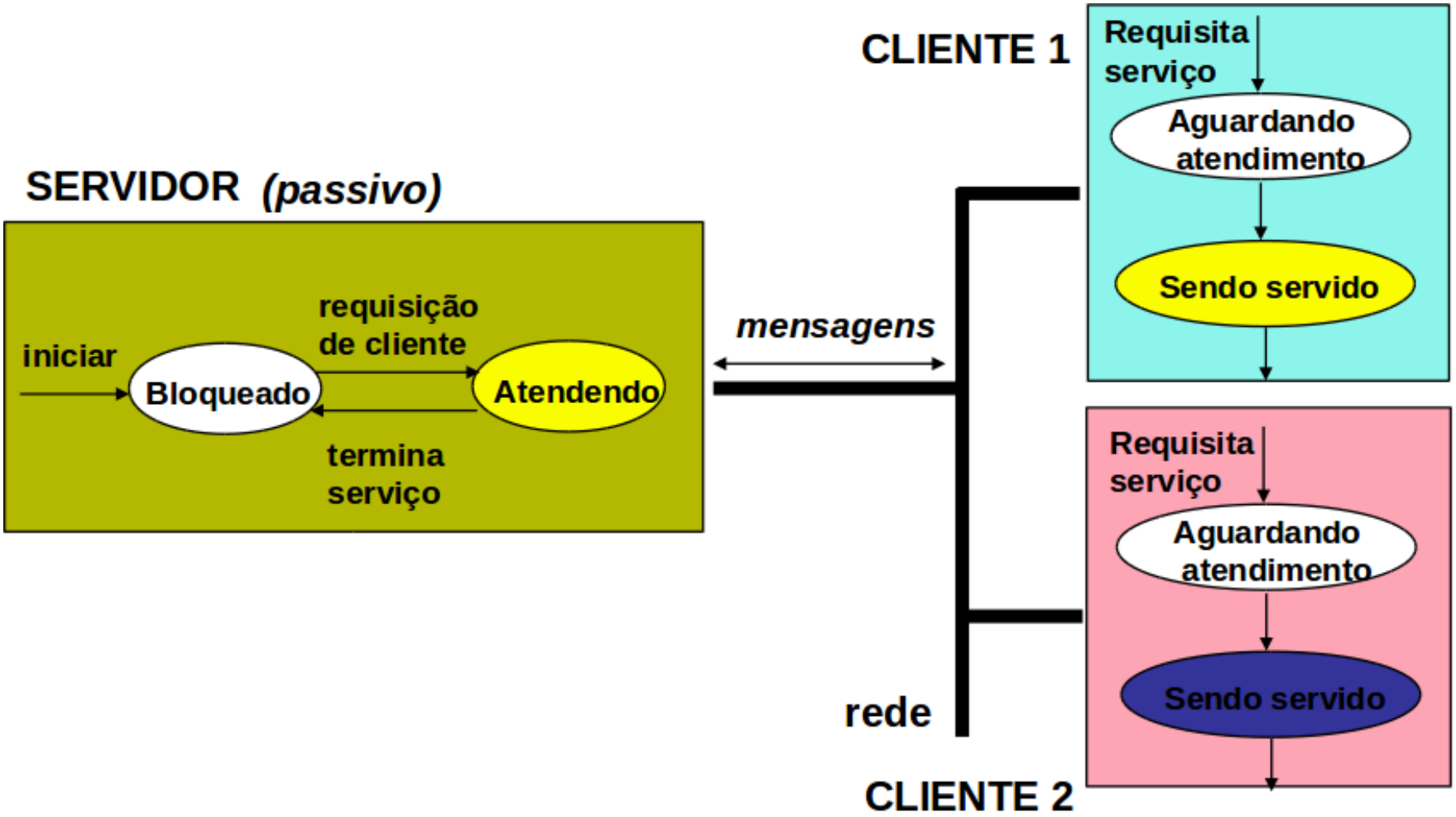
Modelos de interação entre processos

- **Filtros**: processos recebem mensagens de um ou mais canais de entrada e enviam mensagens para um ou mais canais de saída
- **Cliente/Servidor**: processos servidores manipulam requisições de processos clientes
- **Ponto-a-Ponto**: processos interagem aos pares de forma:
  - centralizada (todo processo comunica-se apenas com um processo central)
  - simétrica (todos os processos podem comunicar-se com todos os outros)
  - circular (cada processo comunica-se com um vizinho a esquerda e outro a direita, formando um círculo)

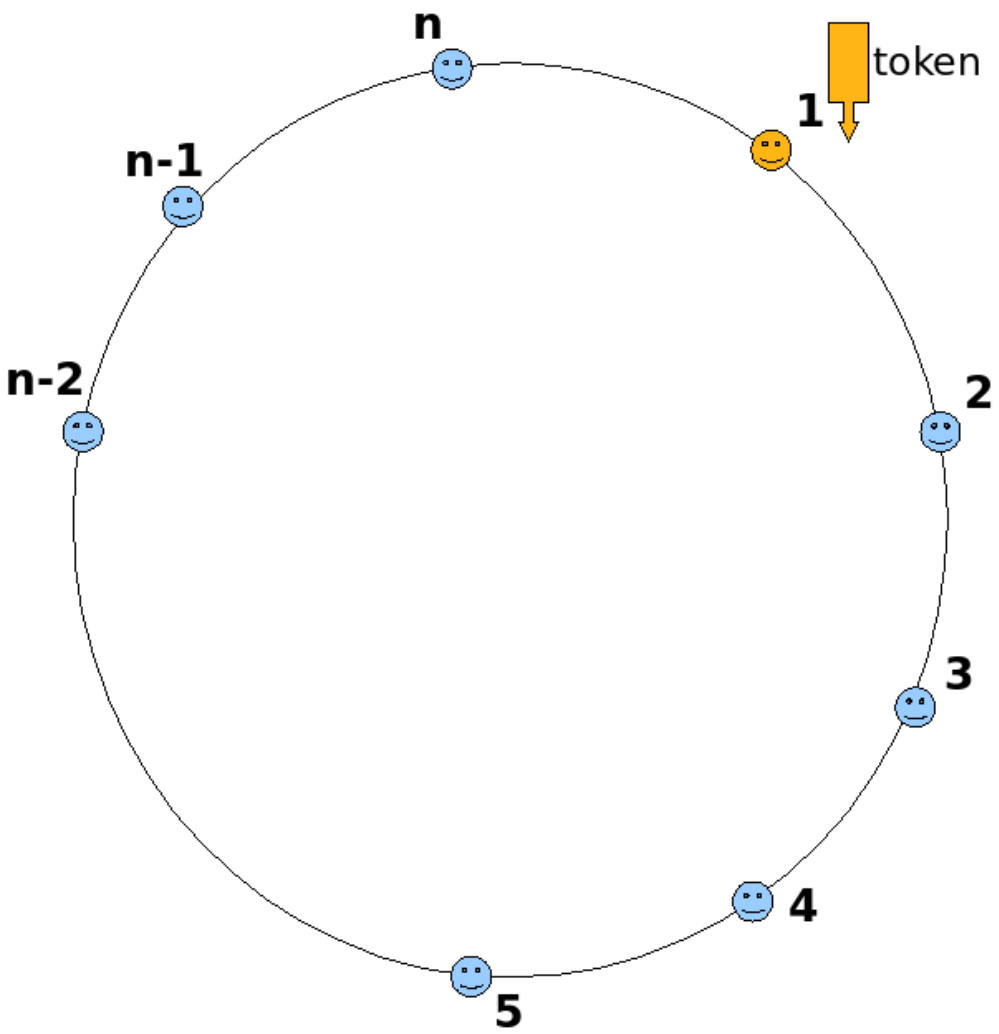
Exemplo filtro



Exemplo cliente/servidor



Exemplo ponto-a-ponto



Passagem de mensagem assíncrona e síncrona

A passagem de mensagem pode ser **assíncrona ou síncrona**

- **Assíncrona**: as mensagens ainda não recebidas são armazenadas em uma fila ou canal de mensagens (o processo que enviou a mensagem pode proceder assincronamente com relação ao processo que irá receber a mensagem)
- **Síncrona**: o processo que envia a mensagem fica bloqueado até que a mensagem seja recebida

Vantagens da passagem de mensagem síncrona

- **Não é preciso dimensionar o tamanho das filas** ou canais de mensagens (cada processo terá no máximo uma mensagem no seu canal de comunicação)
- Quando um processo termina de enviar uma mensagem ele **sabe que o receptor foi notificado** de alguma forma
- É mais **fácil descobrir quando um processo não está mais respondendo**

Desvantagens da passagem de mensagem síncrona

- **Redução da concorrência** (o processo que envia a mensagem fica bloqueado até que o processo destino esteja pronto para recebê-la)
- **Possibilidade maior de ocorrência de deadlock** (garantir que sempre que um processo envia uma mensagem, outro processo esteja esperando uma mensagem desse processo e não tentando enviar)

Bloqueio no recebimento

Comportamento do receptor

Quando a **primitiva receive é executada** há também duas possibilidades:

- se a mensagem já foi enviada, a mensagem é recebida e a execução segue
- se a mensagem ainda não foi enviada:
  1. a **thread pode ser bloqueada até a mensagem chegar** (MAIS USUAL), ou
  2. a **thread continua executando e a tentativa de recebimento é abandonada** (MENOS USUAL)

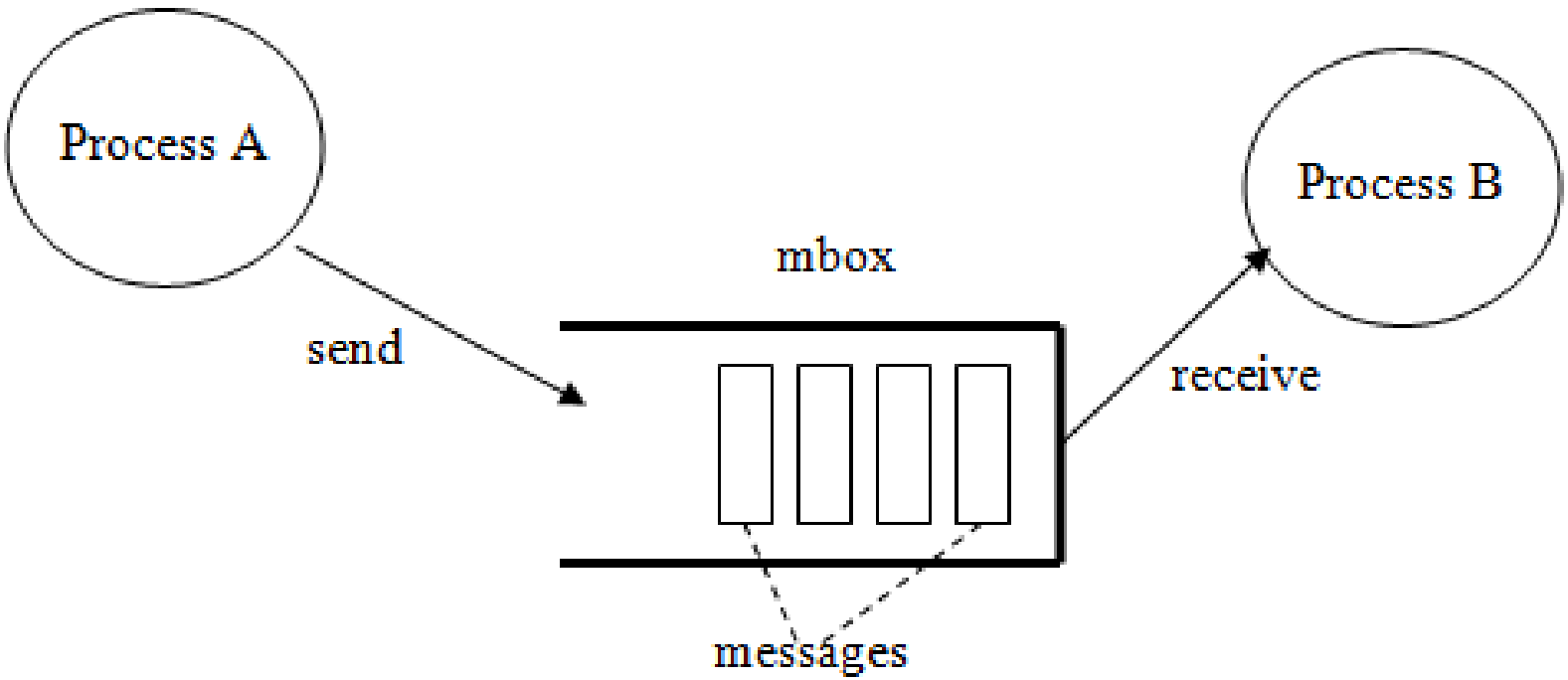
Combinações mais comuns de comportamentos

1. **envio bloqueante, recebimento bloqueante**: emissor e receptor são bloqueados até a mensagem ser entregue (*rendezvous*)
   - caracteriza o tipo mais forte de sincronização
2. **envio não-bloqueante, recebimento bloqueante**: o receptor é bloqueado até a mensagem chegar
   - é a combinação mais usual e permite que uma thread envie mensagens para vários destinos em sequência
3. **envio não-bloqueante, recebimento não-bloqueante**: nenhuma parte fica bloqueada

Endereçamento indireto

Canais de mensagens

Mensagens enviadas para **canais de mensagens** (vantagem: desacoplamento de emissor e receptor)



Produtor/Consumidor usando mensagens

```
void produtor() {
   message pmsg;
   while(true) {
      receive(pode_produzir, pmsg);
      pmsg = produz_item();
      send(pode_consumir, pmsg);
   }
}

void consumidor() {
   message cmsg, msg_vazia;
   while(true) {
      receive(pode_consumir, cmsg);
      send(pode_produzir, msg_vazia);
      consome(cmsg);
   }
}
```

```
void main() {
   int N //tamanho do buffer
   message msg_vazia;
   createMailbox(pode_produzir);
   createMailbox(pode_consumir);

   for(int i=1; i<=N; i++)
      send(pode_produzir, msg_vazia);
   //inicia os processos
}
```

MPI (*Message Passing Interface*)

Definição de uma biblioteca de funções que permite diferentes implementações (ex. MPICH, OpenMPI, MPILAM)

Características do MPI

- Os processos podem usar **operações de comunicação ponto-a-ponto** para enviar e receber mensagens de um processo para outro
- **Grupos de processos** podem usar operações de comunicação coletivas para executar operações globais (ex. soma e *broadcast*)
- O mecanismo chamado **communicator** permite definir subgrupos de processos que trocam informações específicas

Características do MPI

- Nas rotinas de recepção de mensagens pode-se especificar que **a mensagem esperada deve vir de um determinado processo, ou de qualquer processo**
- **Tags** associadas às mensagens podem ser usadas para prover uma forma de distinção entre elas e seu valor é definido pelo processo que a envia
- O processo receptor pode **filtrar a mensagem pela informação da sua tag ou não filtrar e receber qualquer uma das mensagens**

Características do MPI

- O **número de processos** é normalmente definido no início da computação
- O MPI garante apenas que duas mensagens, enviadas pela mesma origem para o mesmo destino, chegarão na ordem em que foram enviadas
- A ordem de chegada de mensagens enviadas por origens diferentes para um mesmo destino não é definida

Single-Program Multiple-Data (SPMD)

- Escrevemos, compilamos e executamos um único programa
- Tipicamente, um dos processos (id=0) faz alguma coisa diferente (distribui as tarefas, coleta os resultados, etc)
- A construção *if-else* é usada para fazer o programa SPMD

Iniciar o ambiente de execução com MPI

MPI_Init: faz todas as inicializações necessárias (deve ser chamada em todo programa MPI, deve preceder todas as demais chamadas MPI, e deve ser chamada apenas uma vez no programa)

```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

Finalizar o ambiente de execução com MPI

MPI_Finalize: finaliza liberando a memória (deve ser chamada em todo programa MPI, e deve ser a última chamada MPI)

```
int MPI_Finalize(void);
```

Iniciar e finalizar o ambiente de execução com MPI

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
  int ret;
  ret = MPI_Init(&argc, &argv);
  if (ret == MPI_SUCCESS) {
    printf ("MPI iniciou corretamente\n");
  }
  MPI_Finalize();
  return 1;
}
```

Iniciar e finalizar o ambiente de execução com MPI

Exercício

1. Abra o arquivo **intro0.c** (mostra como iniciar e finalizar o ambiente para uso do MPI)
2. Compile o programa fazendo: **mpicc -o intro0 intro0.c**
3. Execute o programa com um único processo fazendo: **mpirun ./intro0**
4. Execute o programa com 2 processos fazendo: **mpirun -np 2 ./intro0**
5. Execute o programa com 3 processos passando argumentos: **mpirun -np 3 ./intro0 ola 3 4**, o que mudou?
6. Observe e avalie os resultados

Comunicadores/Grupos

- **Comunicador**: um grupo de processos, os quais podem trocar mensagens entre si
- *MPI_Init* define o **comunicador** que inclui todos os processos criados quando o programa é iniciado
- A variável **MPI_COMM_WORLD** é o comunicador que identifica esse grupo mais geral

Identificador de um processo

MPI_Comm_rank: obtém o **rank** ou identificador de um processo dentro de um grupo (o primeiro identificador tem valor 0)

```
int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

Número de processos criados

MPI_Comm_size: obtém a quantidade de processos pertencentes a um grupo

```
int MPI_Comm_size(
      MPI_Comm  comm        /* in  */,
      int*      comm_sz_p   /* out */);
```

Identificar os processos de um grupo

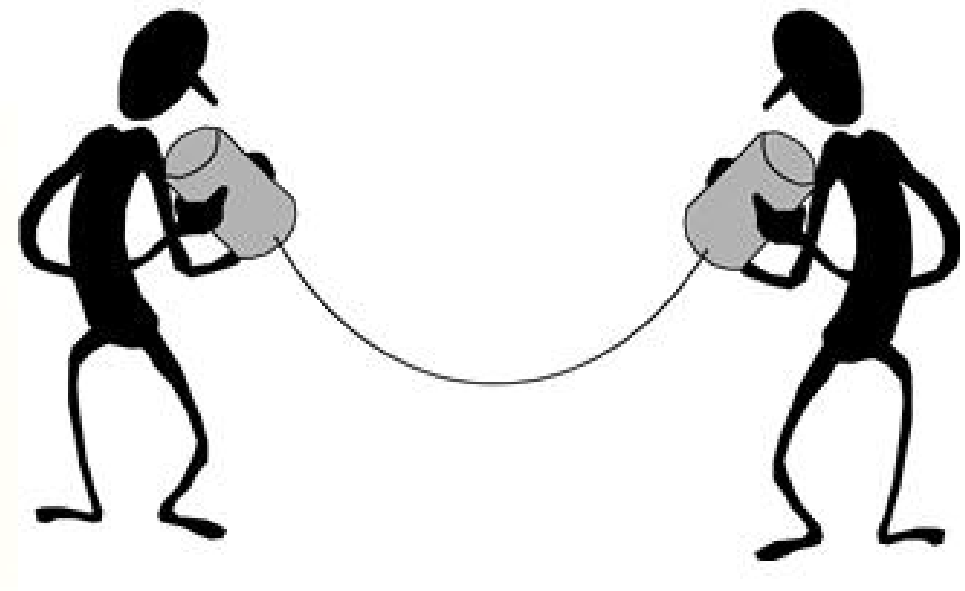
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
  int numtasks, rank, ret;
  ret = MPI_Init(&argc, &argv);
  if (ret == MPI_SUCCESS){
    //obtem o numero de processos dentro do grupo
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    //obtem o rank do processo dentro do grupo
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf ("Processo %d de %d\n", rank, numtasks);
  }
  MPI_Finalize();
  return 0;
}
```

Identificar os processos de um grupo

Exercício

1. Abra o arquivo **intro1.c**
2. Compile o programa
3. Execute o programa com um único processo
4. Execute o programa **várias vezes** com 10 processos
5. Observe e avalie os resultados

Comunicação um-para-um



Funções para enviar e receber mensagens

- **MPI_Send (buffer,count,type,dest,tag,comm)**: envia uma mensagem a um destinatário (retorna após o buffer de envio ficar livre)
- **MPI_Recv (buffer,count,type,source,tag,comm,status)**: recebe uma mensagem de um remetente (bloqueante)

O **tag** é um identificador de tipo de mensagem

As mensagens são constituídas de:

- **dados** (valores + tipo)
- **cabeçalho** (origem, destino, tag e comunicador)

Pares de chamadas de MPI_Send e MPI_Recv devem possuir o mesmo **tag** e o mesmo **comunicador**

Função para enviar mensagens

```
int MPI_Send(
   void*          msg_buf_p      /* in */,
   int            msg_size       /* in */,
   MPI_Datatype   msg_type       /* in */,
   int            dest           /* in */,
   int            tag            /* in */,
   MPI_Comm       communicator   /* in */);
```

Tipos de dados definidos pelo MPI

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

Função para receber mensagens

```
int MPI_Recv(
      void*          msg_buf_p      /* out */,
      int            buf_size       /* in  */,
      MPI_Datatype   buf_type       /* in  */,
      int            source         /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* out */);
```

Casamento de mensagens (envio-recebimento)

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

r

MPI_Send

src = q

MPI_Recv

dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

q

Funções para enviar e receber mensagens

```
int main(int argc,char **argv) {
  int rank, rc, tag=1;  char inmsg, outmsg;
  MPI_Status estado;
  rc = MPI_Init(&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    outmsg='x';
    rc=MPI_Send(&outmsg, 1, MPI_CHAR, 1, tag,
                              MPI_COMM_WORLD);
  } else if (rank == 1) {
    rc=MPI_Recv(&inmsg, 1, MPI_CHAR, 0, tag,
                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
  MPI_Finalize();
  return 0;
}
```

Funções para enviar e receber mensagens

Exercício

1. Abra o arquivo `intro2.c` (compreenda como ele funciona)
2. Execute o programa com um único processo
3. Execute o programa várias vezes com 2 processos
4. Execute o programa várias vezes com 4 processos
5. Observe e avalie os resultados

Recebendo mensagens

Um processo pode receber uma mensagem sem saber:

- o emissor da mensagem
- a tag da mensagem


```
MPI_Recv(&val, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &estado);
...
if (estado.MPI_TAG==99) ...
if (estado.MPI_SOURCE==0) ...
```

Função para obter a quantidade de dados recebidos

```
int  MPI_Get_count(
          MPI_Status*   status_p   /* in  */,
          MPI_Datatype  type       /* in  */,
          int*          count_p    /* out */);
```

Outro exemplo de comunicação um-para-um

```
void main(int argc, char **argv) {
 char msg1[20], msg2[20]; MPI_Status status;
 int  i, rank, size, tag = 99, tam;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 if (rank == 0) { strcpy(msg1, "Hello, world");
   tam = strlen(msg1) + 1;
   for (i = 1; i < size; i++)
    MPI_Send(msg1,tam,MPI_CHAR,i,tag,MPI_COMM_WORLD);
 } else
    MPI_Recv(msg2, 20, MPI_CHAR, 0, tag,
                           MPI_COMM_WORLD, &status);
 MPI_Finalize();
}
```

Funções para enviar e receber mensagens

Exercício

1. Abra o arquivo **mpi_hello.c** e compreenda como ele funciona
2. Execute o programa várias vezes com 2 processos
3. Execute o programa várias vezes com 4 processos
4. Observe e avalie os resultados

Referências bibliográficas

1. *An Introduction to Parallel Programming*, Peter Pacheco, Morgan Kaufmann, 2011.