

# Programação Paralela PPGCA

## Programação com MPI

Prof. Guilherme Koslovski  
Prof. Maurício Pillon



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Referências

- Cursos da ERAD
  - <https://www2.sbc.org.br/erad/doku.php?id=start>
- MPI Fórum
  - <http://www.mpi-forum.org>
- Open MPI Documentation
  - <https://www.open-mpi.org/doc/current/>



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Agenda

- **Programação Paralela com Troca de Mensagens**
- MPI
- Exemplos
- Considerações Finais



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Programação Paralela com Troca de Mensagens

- Opções de Programação
  - Linguagem de programação paralela (específica)
- Occam (Transputer)
  - Extensão de linguagens de programação existentes
- CC++ (Extensão de C++)
- Fortran M



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Programação Paralela com Troca de Mensagens

- Geração automática usando anotações em Código e compilação (FORTRAN)
  - Linguagem padrão com biblioteca para troca de mensagens
- MPI (Message Passing Interface)
- PVM (Parallel Virtual Machine)



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Linguagem padrão com troca de mensagens

- Descrição explícita do paralelismo e troca de mensagens entre os processos
- Métodos Principais
- Criação de processos para execução em diferentes computadores
- Troca de mensagens (send/recv) entre processos
- Sincronização entre os processos



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Modelos

- SPMD – Single Program Multiple Data
  - Existe somente um programa
  - Um único programa executa em diversos hosts sobre um conjunto de dados distinto
- MPMD – Multiple Program Multiple Data
  - Existem diversos programas
  - Programas diferentes executam em hosts distintos
  - Cada máquina possui um programa e um conjunto de dados distintos



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Criação de processos

- **Criação Estática**

- Os processos são especificados antes da execução
- Número fixo de processos
- Modelo SPMD é o mais comum

- **Criação Dinâmica**

- Os processos são criados durante a execução (spawn)
- Encerramento dos processos é dinâmico
- Número variável de processos
- Modelo MPMD é o mais comum



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



# Troca de mensagens

- Realizada através de **primitivas** send e **receive**
- **Comunicação síncrona/bloqueante**
  - Send bloqueia o emissor até o receptor executar receive
- **Comunicação assíncrona/não bloqueante**
  - Send não bloqueia o emissor
  - Receive pode ser utilizado durante a execução

# Troca de mensagens

- Seleção de mensagens
- Filtro para receber uma mensagem de um determinado tipo (**message tag**) ou de um emissor específico
- Comunicação em Grupo
- **Broadcast**
- **Gather/Scatter**
  - Envio de partes de uma mensagem de um processo para diferentes processos de um grupo (distribuir), e recebimento de partes de mensagens de diversos processos de um grupo por um processo (coletar)



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Sincronização

- **Barreiras**
  - Permitem especificar pontos de sincronia entre os processos
  - Ao chegar na barreira, o processo fica esperando todos os demais processos do seu grupo chegarem na barreira.
  - O último processo libera todos os demais processos que estão bloqueados.

# Agenda

- Programação Paralela com Troca de Mensagens
- **MPI**
- Exemplos
- Considerações Finais

# ■ Message Passing Interface (MPI)

- Padrão definido **em 1994** pelo **MPI Fórum**
- Utiliza troca de mensagens entre os processos
- Versão Atual 3.1.2
- Implementações mais utilizadas
  - MPICH
  - LAMMPI
  - JAVA-MPI



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# ■ Message Passing Interface (MPI)

- Ambiente
  - O processo principal inicia a execução
  - A execução ocorre em um conjunto de computadores pré-definidos
- **Possui mais de 125 funções**
- Compiladores
  - Mpicc (linguagem C)
  - Mpicc++ (linguagem C++)
  - Mpi77 (linguagem Fortran)



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Diretivas Básicas

- **MPI\_INIT** inicia um processo MPI, estabelecendo o ambiente e sincronizando os processos para iniciar a aplicação paralela.

```
int MPI_Init(int *argc, char* argv[])
```

└─ Ponteiro com o número de argumentos  
└─ Array de argumentos

- **O MPI deve ser inicializado uma única vez (não realize chamadas subsequentes de MPI\_Init ou MPI\_Init\_Thread)**
- **MPI\_FINALIZE** encerra o um processo MPI. Utilizado para sincronizar os processos para o término da aplicação paralela.

```
Int MPI_Finalize()
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Exemplo

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

int main(int argc, char **argv){
    int rank, size;

    MPI_Init(&argc, &argv);

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

    printf("Hello World! I'm %d of %d\n",rank,size);

    MPI_Finalize();
    return 0;
}
```



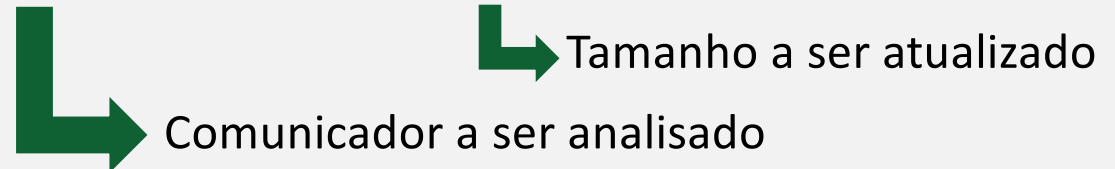
**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



# MPI – Diretivas Básicas

- **MPI\_COMM\_SIZE** retorna o número de processos dentro de determinado grupo.

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```



- Caso o comunicador seja o **MPI\_COMM\_WORLD** o **MPI\_COMM\_SIZE** retorna quantidade total de processos.

# MPI – Diretivas Básicas

- **MPI\_COMM\_RANK** retorna o rank do processo em determinado comunicador.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```



Comunicador a ser analisado



Rank a ser atualizado

- Utilizado amplamente em programas estilo **mestre – escravo**
  - O processo com rank 0 é o mestre e os demais processos escravos



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Exemplo

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

int main(int argc, char **argv){
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Processo %d de %d\n",rank,size);
    if(rank == 0) {
        printf("(%d) -> Primeiro a escrever!\n",rank);
        MPI_Barrier(MPI_COMM_WORLD);
    }else{
        MPI_Barrier(MPI_COMM_WORLD);
        printf("(%d) -> Agora posso escrever!\n",rank);
    }
    MPI_Finalize();
    return 0;
}
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Exemplo

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

int main(int argc, char **argv){
    int rank, size;
    int tag=0;
    MPI_Status status;
    char msg[20];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank == 0) {
        strcpy(msg, "Hello World!\n");
        for(i=1; i<size; i++) {
            printf("0 enviando 20 para %d\n", i);
            MPI_Send(msg, 20, MPI_CHAR, i, tag,
                MPI_COMM_WORLD);
        }
    }
```

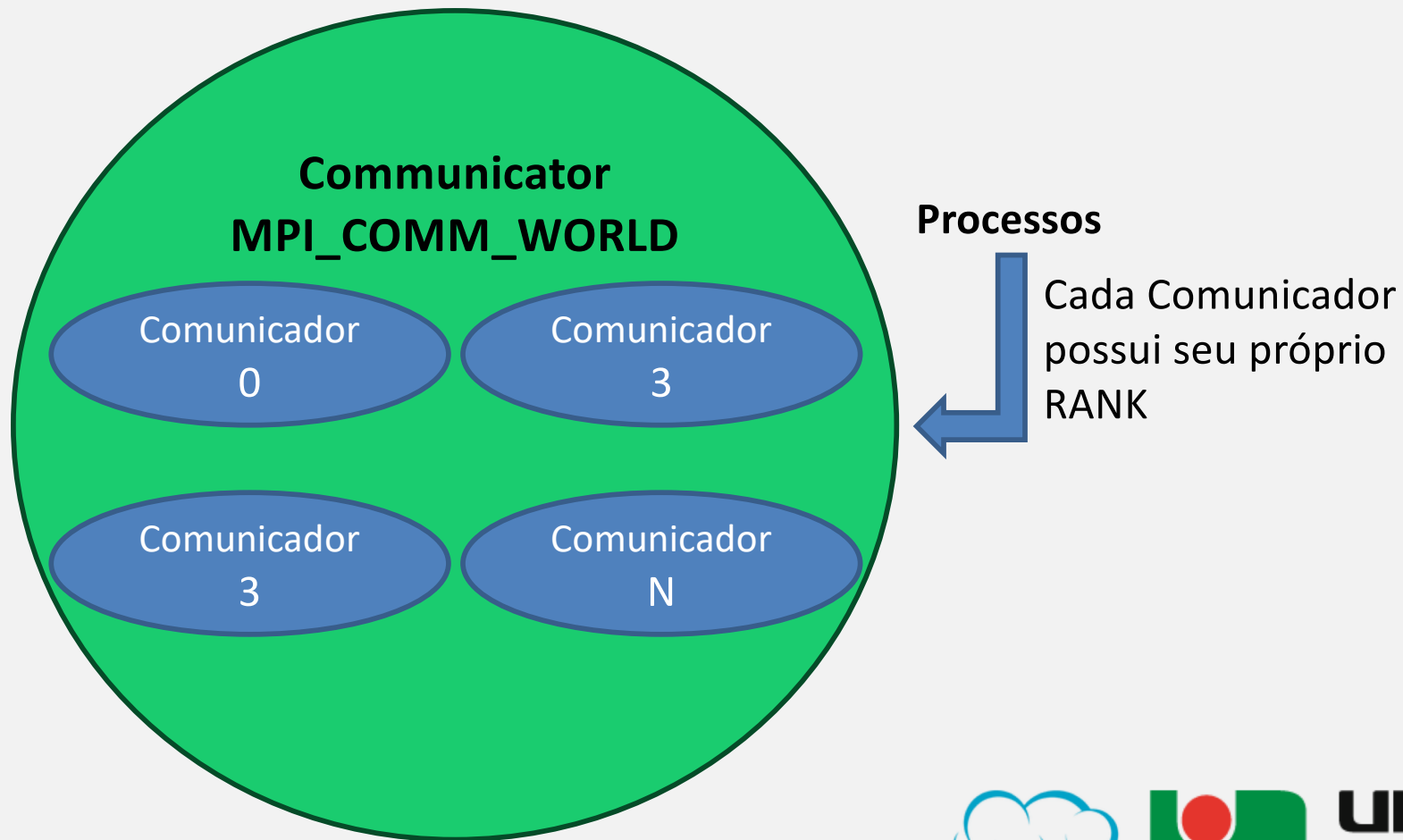
```
    else{
        printf("%d esta
            esperando\n", rank);

        MPI_Recv(msg, 20, MPI_CHAR, 0,
            tag,
            MPI_COMM_WORLD,
            &status);
        printf("Message received:
            %s\n", msg);
    }
    MPI_Finalize();
    return 0;
}
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Componentes



# MPI – Tipos de Dados

- Dados do tipo **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**

# MPI – Comunicação

- Basicamente as mensagens possuem a seguinte configuração

Origem	Destino	TAG	Dados
--------	---------	-----	-------

# ■ MPI – Funções Bloqueantes

- MPI\_Send(void\* buf, int count, MPI\_Datatype datatype, int dest, int tag, MPI\_Comm comm)
- 
- Endereço inicial do buffer
- Quantidade de elementos enviados
- Tipo de dados dos elementos no buffer
- Rank do processo de destino
- Identificador
- Comunicador



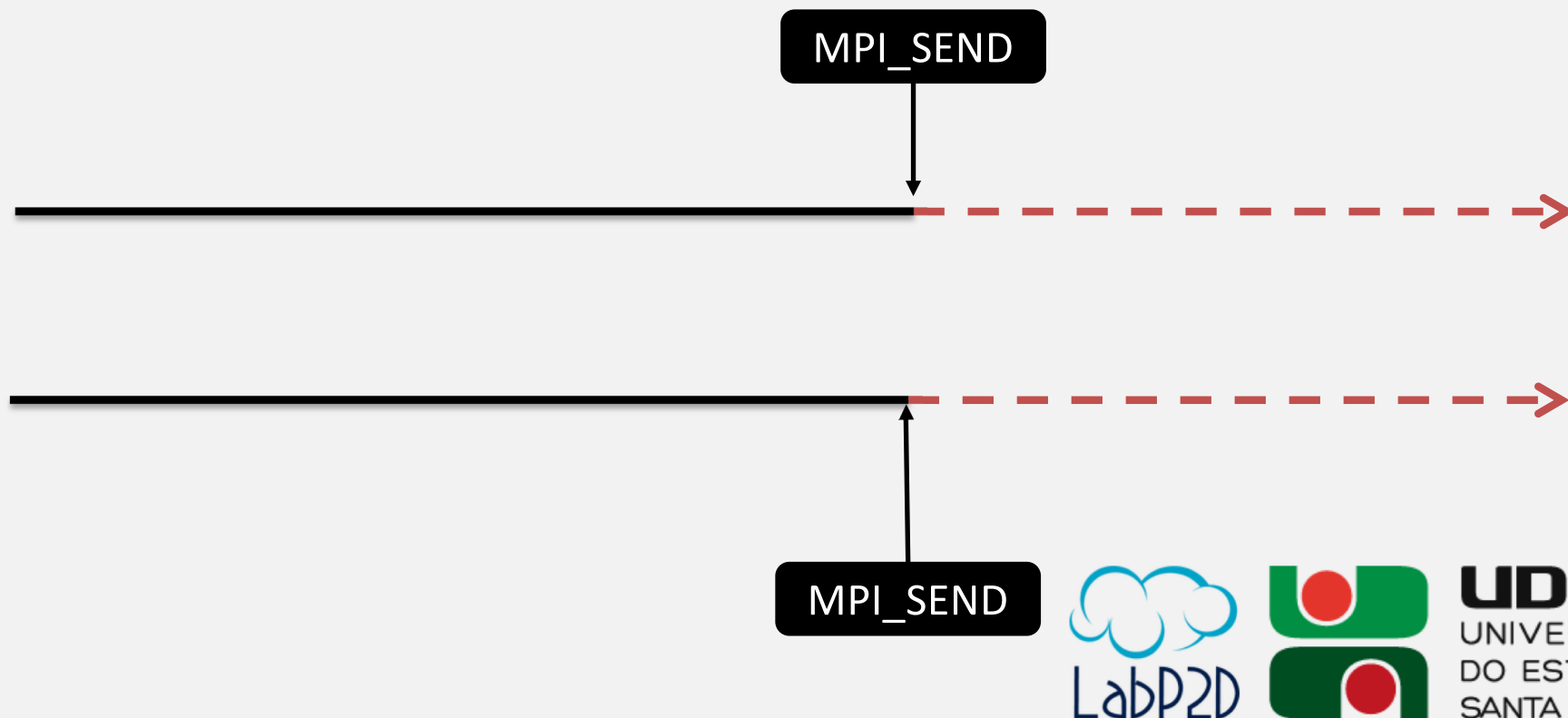
# ■ MPI – Funções Bloqueantes

- `MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm)`
  - Endereço inicial do buffer
  - Quantidade de elementos enviados
  - Tipo de dados dos elementos no buffer
  - Identificador
  - Comunicador
  - Rank do processo de origem da mensagem

# MPI – Funções Bloqueantes

## DEADLOCK

- Este fenômeno ocorre quando todos os processos estão aguardando eventos que não foram iniciados



# ■ MPI – Funções Não Bloqueantes

- `MPI_Isend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - Endereço inicial do buffer
  - Quantidade de elementos enviados
  - Tipo de dados dos elementos no buffer
  - Identificador
  - Comunicador
  - Rank do processo de destino

# ■ MPI – Funções Não Bloqueantes

- `MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm)`
  - Endereço inicial do buffer
  - Quantidade de elementos enviados
  - Tipo de dados dos elementos no buffer
  - Rank do processo de origem da mensagem
  - Identificador
  - Comunicador

# MPI – Exemplo

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

int main(int argc, char **argv){
    int rank,size,i;
    int tag=0;
    MPI_Status status;
    char msg[20];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank == 0) {
        strcpy(msg,"Hello World!\n");
        for(i=1;i<size;i++)
```


```
        MPI_Send(msg,13,
MPI_CHAR,i,tag,
MPI_COMM_WORLD);
    } else {
        MPI_Recv(msg,20,
MPI_CHAR,0,tag,
MPI_COMM_WORLD, &status);
        printf("Message received:
%s\n", msg);
    }
    MPI_Finalize();
    return 0;
}
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# ■ MPI – Funções Não Bloqueantes

- Espera a requisição de envio ou recebimento de mensagem seja completada

Requisição de  
send ou recv 

 Status da requisição

- `MPI_Wait(MPI_Request *request, MPI_Status *status)`



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# ■ MPI – Funções Não Bloqueantes

- Testa se o envio ou recebimento de mensagem foi completada

Requisição de  
send ou recv ←

True se a requisição  
foi completada, falso  
caso contrario ←

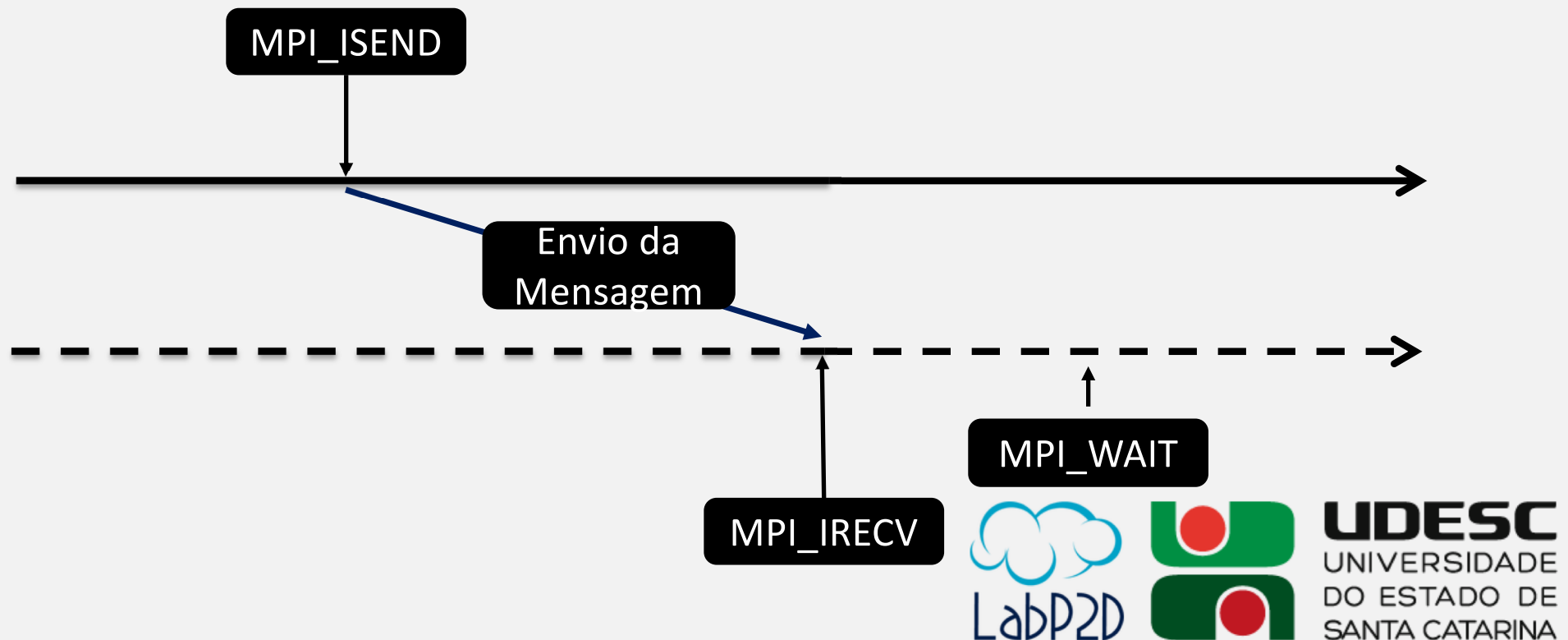
Status da requisição ←

- `MPI_Test(MPI_Request *request, int flag, MPI_Status *status)`

# MPI – Funções Não Bloqueantes

## DEADLOCK (?)

- O processo não espera que o recebimento da mensagem esteja concluído.

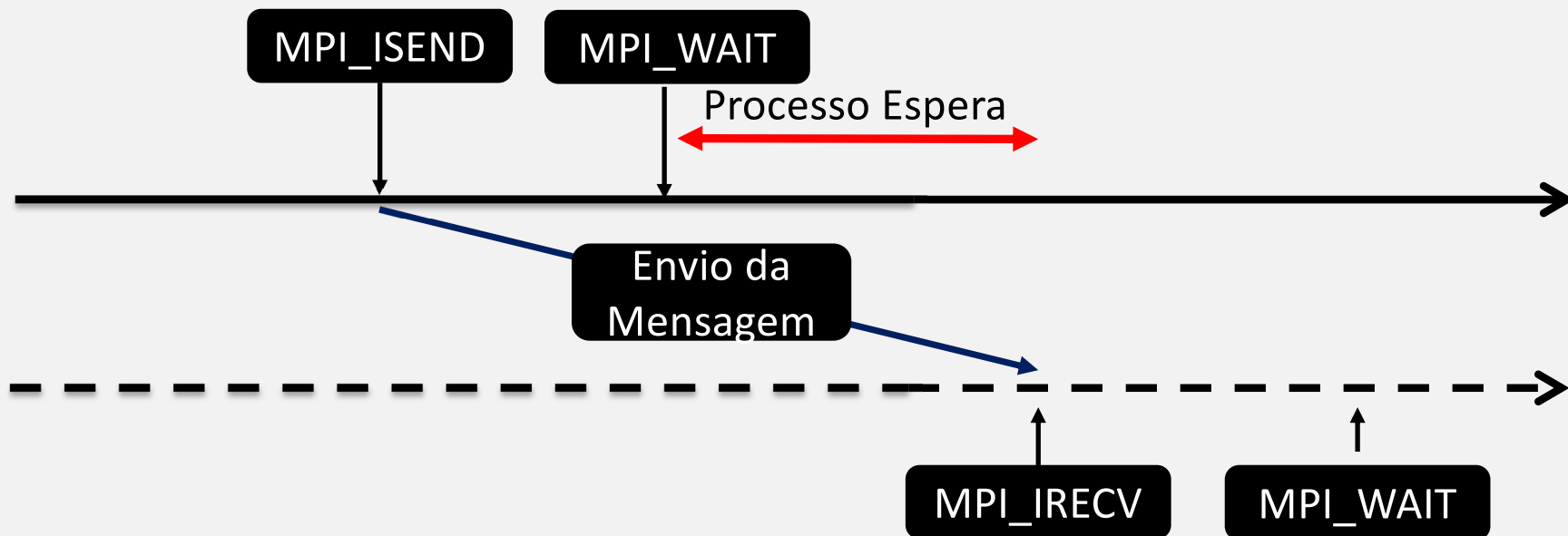




# MPI – Funções Não Bloqueantes

## DEADLOCK (?)

- O processo espera que o recebimento da mensagem esteja concluído.



# MPI – Barreiras

- A função **MPI\_BARRIER** realiza a sincronização explícita de todos os processos de um determinado comunicador/grupo
- O processos que utilize **MPI\_BARRIER** para de executar até que todos os membros de seu grupo também executem o **MPI\_BARRIER**

Identificador do comunicador

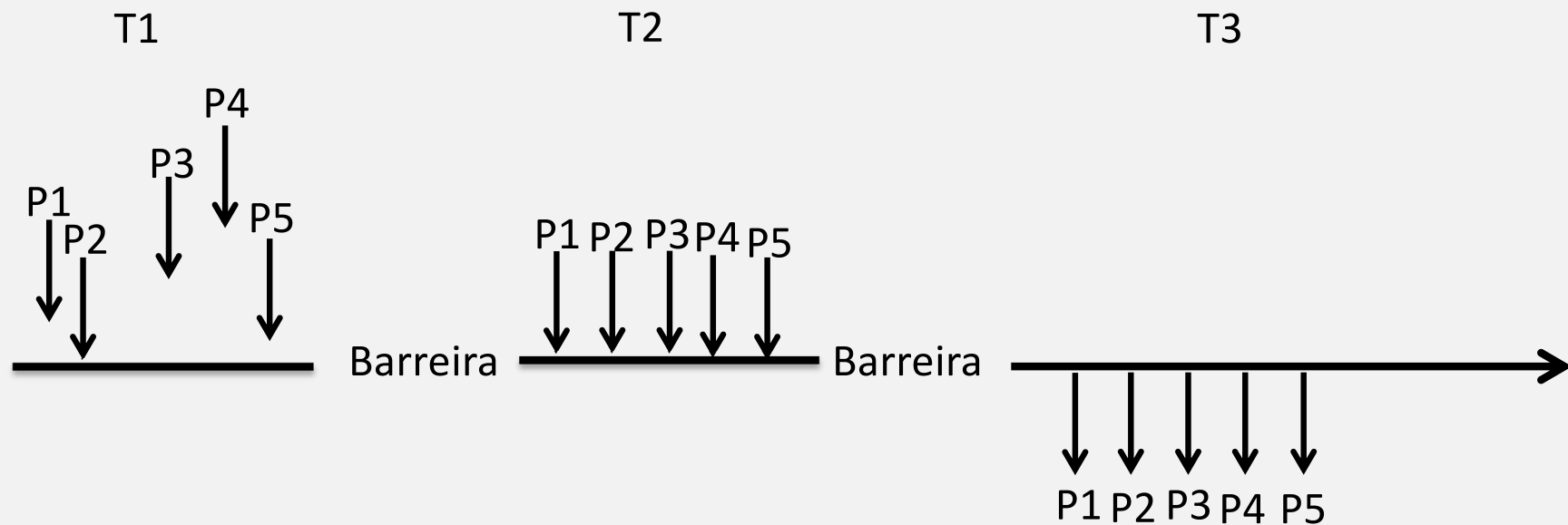
Int MPI\_Barrier ( MPI\_Comm comm)



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Barreiras

- O processo espera que o recebimento da mensagem esteja concluído.



# MPI – Exemplo

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

int main(int argc, char **argv){
    int rank,size,i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I'm %d of %d\n",rank,size);
    if(rank == 0) {
        printf("(%d) -> Primeiro a escrever!\n",rank);
        MPI_Barrier(MPI_COMM_WORLD);
    }else{
        MPI_Barrier(MPI_COMM_WORLD);
        printf("(%d) -> Agora posso escrever!\n",rank);
    }
    MPI_Finalize();
    return 0;
}
```

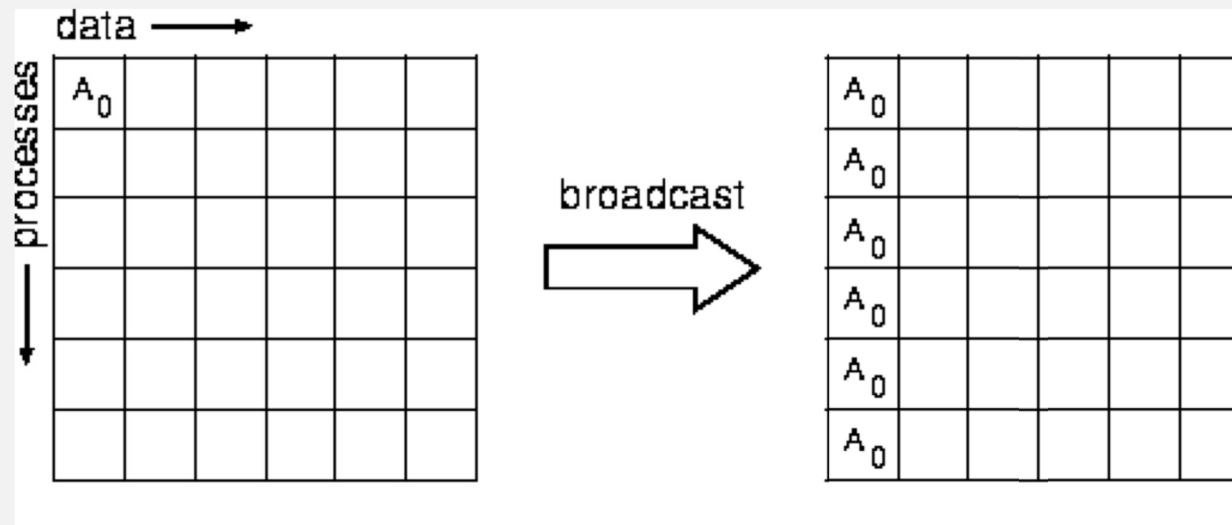


**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Comunicação em grupo

## Broadcast

- A função **MPI\_Bcast** permite o processo enviar dados para todos os membros do grupo
- Todos os processos do grupo devem utilizar o mesmo **comm** e **root**



# MPI – Comunicação em grupo

## Broadcast

Endereço inicial  
do buffer ←

Quantidade de  
elementos  
enviados ↗

Tipo de dados dos  
elementos no buffer ↗

```
Int MPI_Bcast(void *buffer, int count, MPI_Datatype, int root,  
MPI_Comm comm)
```

↘ Comunicador/grupo

Rank do  
emissor do  
broadcast ↘

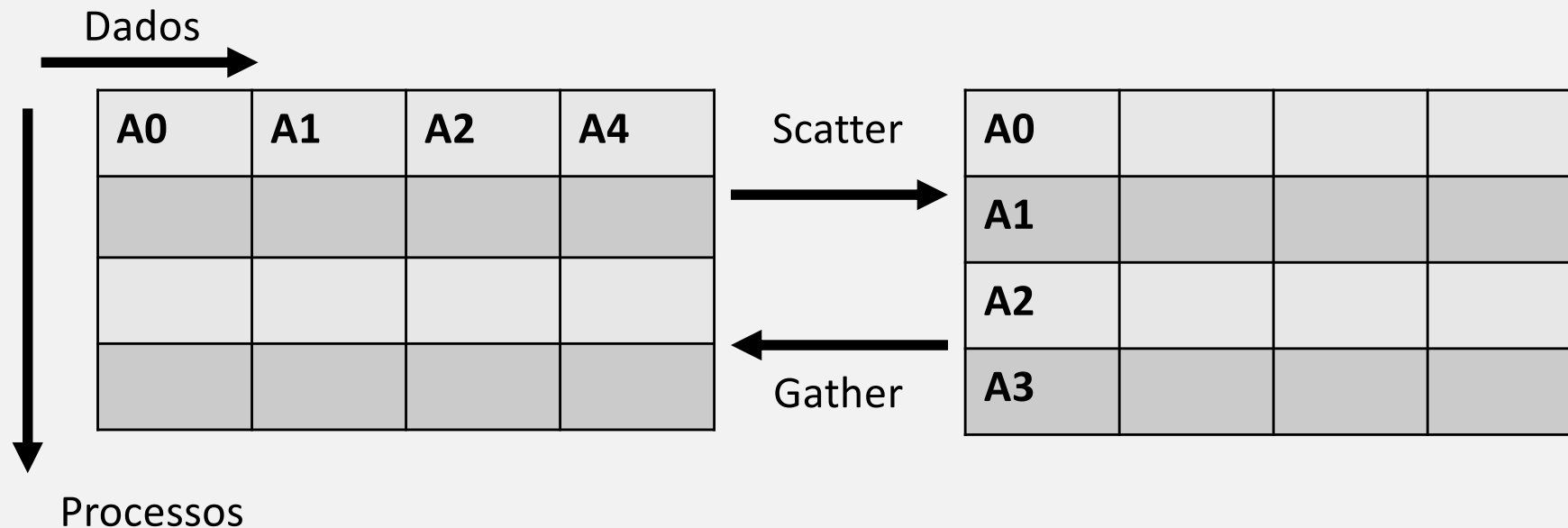
# MPI – Comunicação em grupo

## Scatter e Gather

- **Scatter**
  - Um processo necessita distribuir dados em **N** segmentos iguais.
  - O segmento **N** é enviado para o processo **N**
- **Gather**
  - Um processo necessita coletar dados de **n** processos do grupo

# MPI – Comunicação em grupo

## Scatter e Gather





# MPI – Comunicação em grupo

## Gather

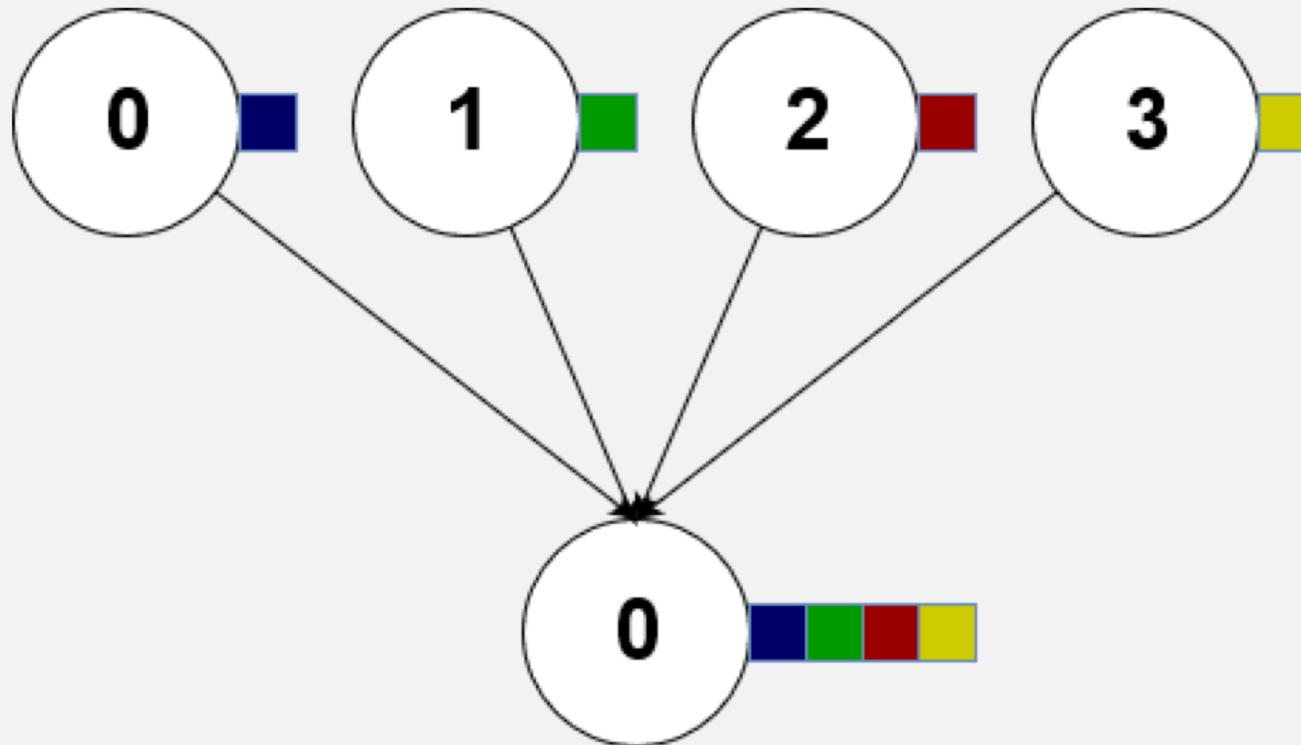
```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype  
sendtype, void recvbuf, int recvcount, MPI_Datatype recvtype, int  
root, MPI_Comm comm)
```

Diagram illustrating the parameters of the `MPI_Gather` function:

- `sendbuf`: Endereço inicial do buffer
- `sendcount`: Quantidade de elementos enviados
- `MPI_Datatype sendtype`: Tipo de dados dos elementos no buffer
- `recvbuf`: Endereço do buffer do receptor
- `recvcount`: Quantidade de elementos recebidos
- `MPI_Datatype recvtype`: Tipo de dados dos elementos no buffer
- `root`: Rank do processo receptor
- `MPI_Comm comm`: Comunicador/grupo

# MPI – Comunicação em grupo

## Gather



# MPI – Exemplo

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

int main(int argc, char **argv){
    int sndbuffer, *recvbuffer, rank, size, i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    recvbuffer = (int *)malloc(size*sizeof(int));
    sndbuffer = rand*rank;

    MPI_Gather(&sndbuffer, 1, MPI_INT, recvbuffer, 1, MPI_INT, 0,
              MPI_COMM_WORLD);
    if(rank == 0)
        printf("(%d) – Recebi vetor: ", rank);
        for(i=0; i<size; i++)
            printf("%d", recvbuffer[i]);

    MPI_Finalize();
    return 0;
}
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Comunicação em grupo

## Scatter

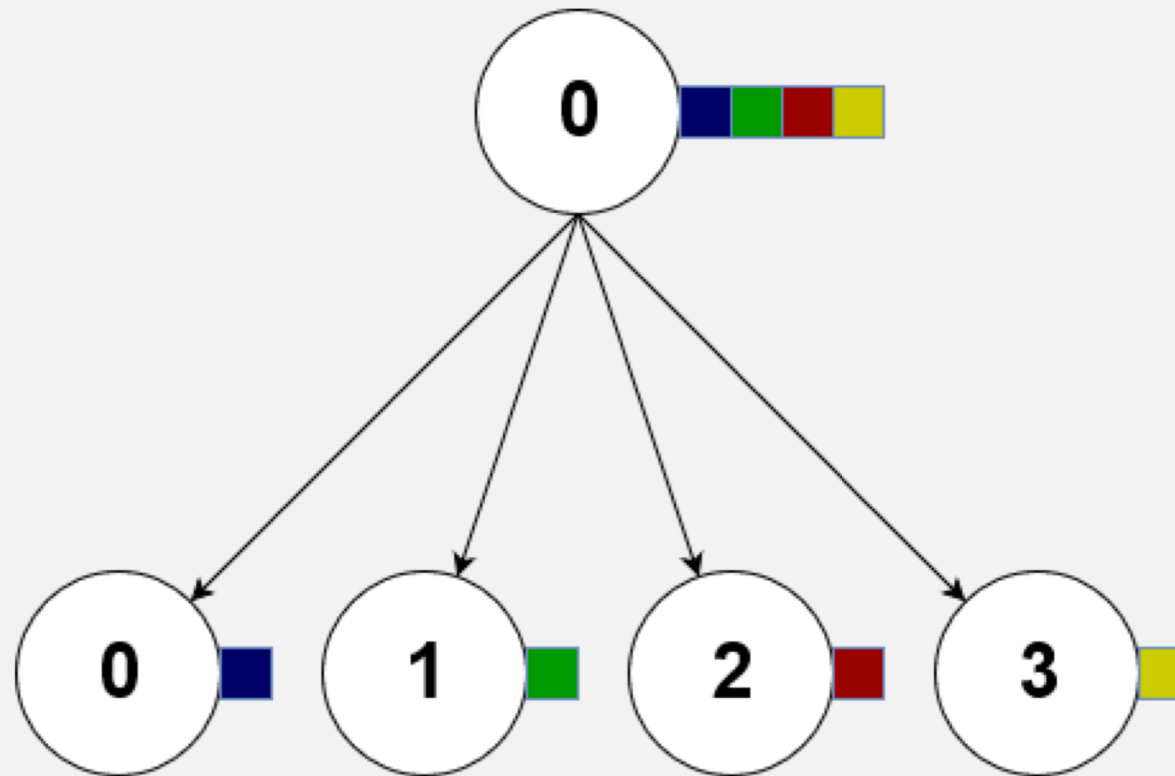
int MPI\_Scatter(const void sendbuf, int sendcount, MPI\_Datatype sendtype, void \*recvbuf, int recvcount, MPI\_Datatype recvtype, int root, MPI\_Comm comm)

Diagram illustrating the parameters of the MPI\_Scatter function:

- Rank do processo receptor (points to `root`)
- Comunicador/grupo (points to `comm`)
- Endereço inicial do buffer (points to `sendbuf`)
- Quantidade de elementos enviados (points to `sendcount`)
- Tipo de dados dos elementos no buffer (points to `sendtype`)
- Endereço do buffer do receptor (points to `recvbuf`)
- Quantidade de elementos recebidos (points to `recvcount`)
- Tipo de dados dos elementos no buffer (points to `recvtype`)

# MPI – Comunicação em grupo

## Scatter



# MPI – Exemplo

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

int main(int argc, char **argv){
    int *sndbuffer, recvbuffer, rank, size, i;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    sndbuffer = (int *)malloc(size*sizeof(int));
    if(rank == 0) {
        for(i=0; i<size; i++) sndbuffer[i] = i*i;
    }
    MPI_Scatter(sndbuffer, 1, MPI_INT, &recvbuffer, 1, MPI_INT, 0,
               MPI_COMM_WORLD);
    if(rank != 0)
        printf("(%d) – Received %d\n", rank, recvbuffer);
    MPI_Finalize();
    return 0;
}
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Considerações Finais

- pThreads, OpenMP e MPI são amplamente utilizados
- MPI define uma interface padrão para troca de mensagens entre processos distribuídos
- Diversas bibliotecas implementam esta API