

# Programação Paralela OPRP001

## 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)



# 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)



# 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 ultimo processo libera todos os demais processos que estão bloqueados.

# Agenda

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



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# ■ 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
  - IAMMPI
  - 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 realizar chamdas 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)
```



Comunicador a ser analisado



Tamanho a ser atualizado

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



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# 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 – Diretivas Básicas

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

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



Comunicador a ser analisado



Tamanho a ser atualizado

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



**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);
        }
    }
```

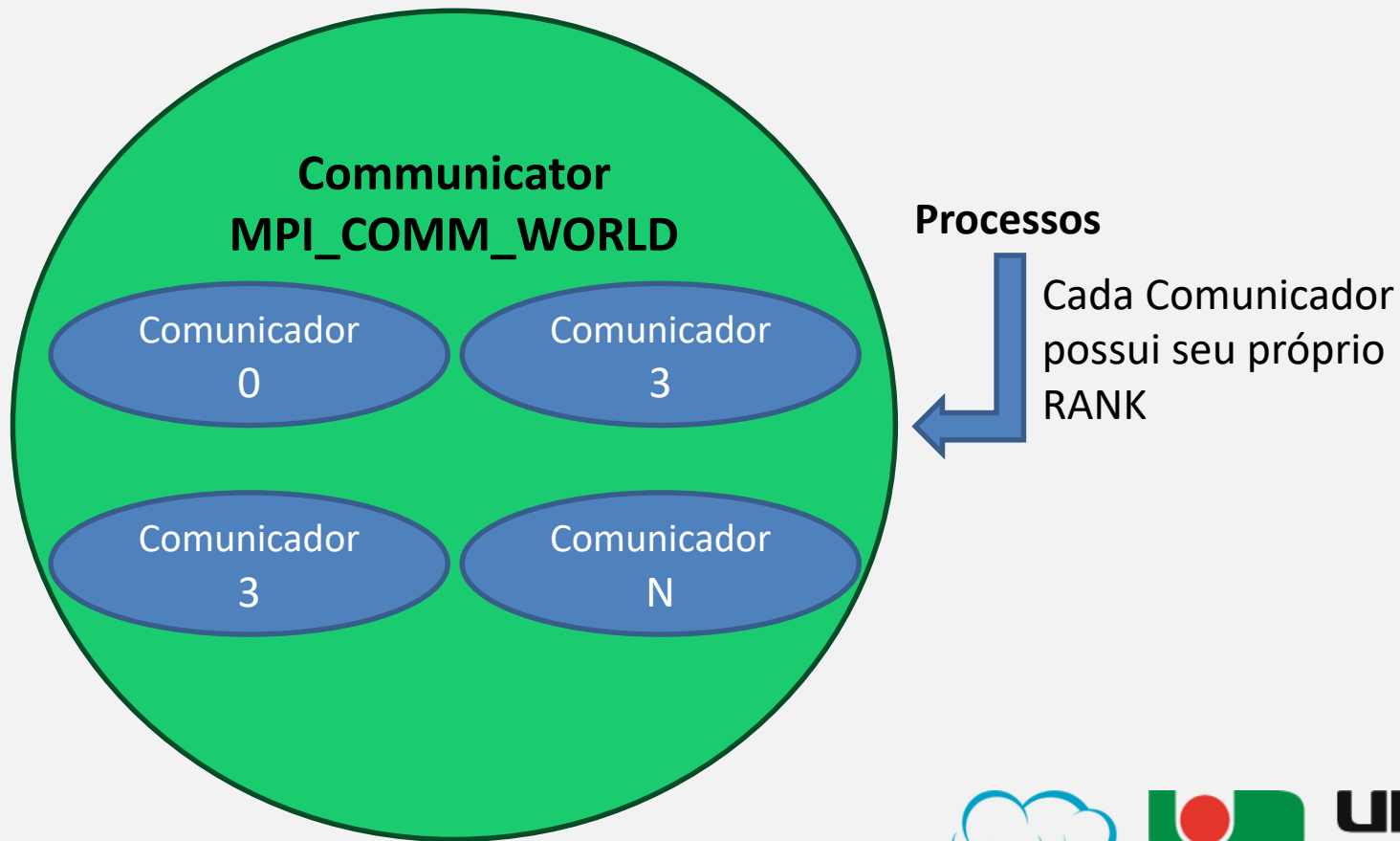
```
    else{
        printf("%d esta
            esperando\n", rank);

        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**

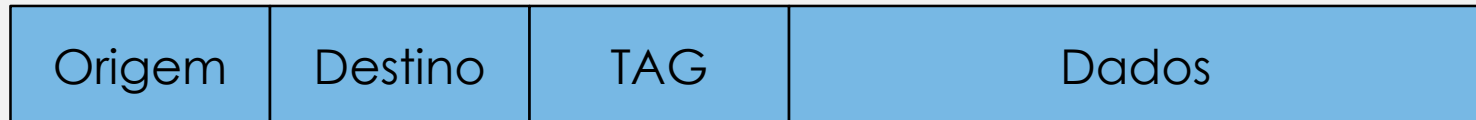


**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



# MPI – Comunicação

- Basicamente as mensagens possuem a seguinte configuração



# 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

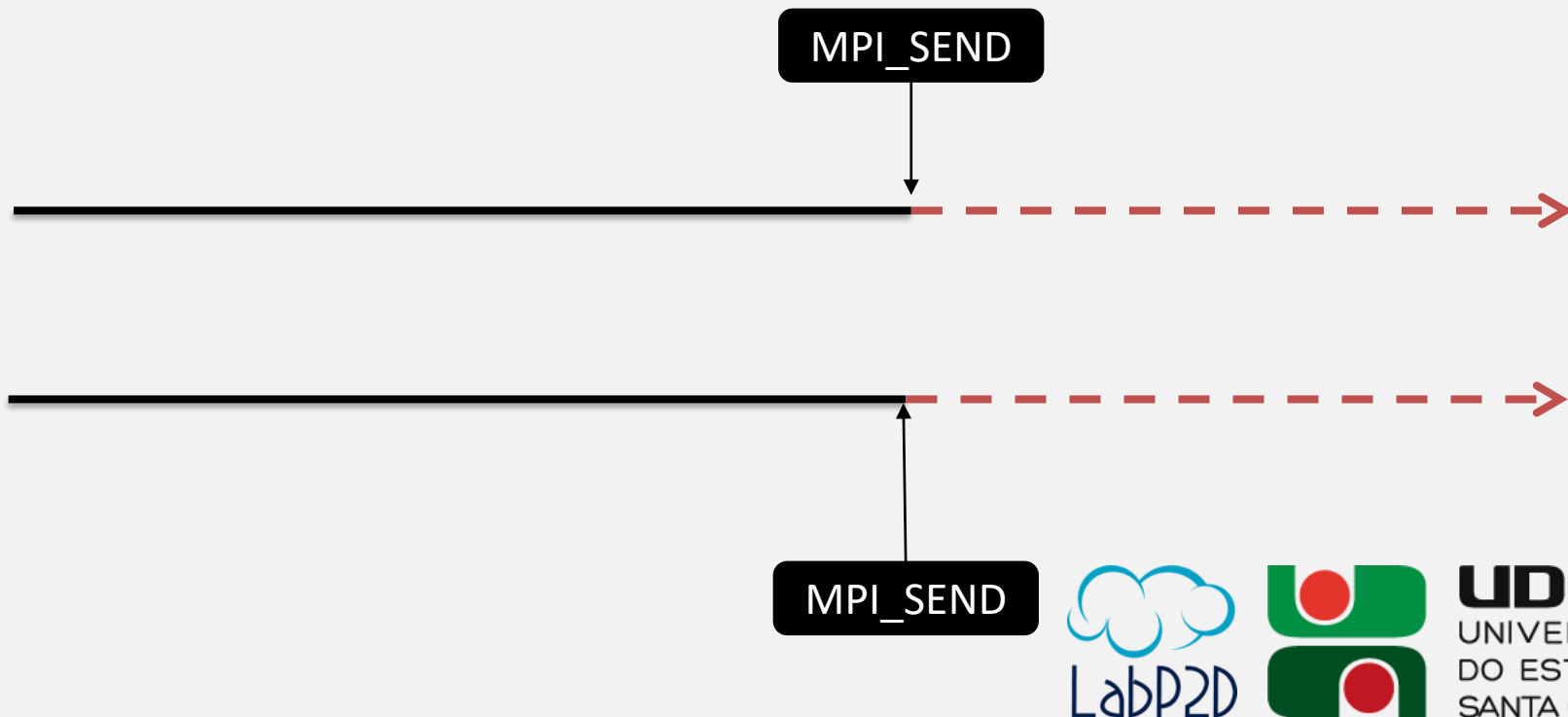
# MPI – Funções Bloqueantes

- `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
  - Rank do processo de destino
  - Identificador
  - Comunicador

# ■ 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 – Exercício 01

- Construa um programa com comunicação em anel utilizando MPI
  - Cada processo deve receber uma mensagem do processo anterior (o rank)
  - Some seu rank com o valor da mensagem recebida
  - Envie o valor atualizado para o processo seguinte do anel
- O processo com rank 0 inicia com token 0
- Utilize as função bloqueante MPI\_Send() e MPI\_Recv()
- Realize testes com 4,8,12,16 hosts



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA



# MPI – Exercício 01

- Construa um programa com comunicação em anel utilizando MPI
  - Cada processo deve receber uma mensagem do processo anterior (o rank)
  - Some seu rank com o valor da mensagem recebida
  - Envie o valor atualizado para o processo seguinte do anel
- O processo com rank 0 inicia com token 0
- Utilize as função bloqueante MPI\_Send() e MPI\_Recv()
- Realize testes com 4,8,12,16 hosts



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Exercício 02

- Crie um programa em MPI que realize um somatório do produto das linhas de uma matriz  $M[10][30]$ 
  - O master envia uma linha da matriz para os processos filhos
  - Os slaves realizam a multiplicação da linha da matriz
  - Ao terminar, o slave envia seu resultado ao master
  - O master realiza a soma dos resultados obtidos pelos slaves
  - OBS: Tomar cuidado para não iniciar um slave que não irá receber uma linha da matriz (  $\text{rank\_slave} < \text{qnt\_linhas}$  )



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Exercício 02

- Utilize o MPI\_ANY\_SOURCE
- Para enviar uma matriz via MPI existem 2 opções
  - Enviar um vetor de tamanho NxM
  - Garantir que a matriz alocada tenha memória continua

```
int** alloc_2d_int(int linhas, int colunas) {  
    int *data = (int *)malloc(linhas*colunas*sizeof(int));  
    int **array= (int **)malloc(linhas*sizeof(int*));  
    for (int i=0; i<linhas; i++)  
        array[i] = &(data[colunas*i]);  
    return array;  
}
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Exercício 02b

- Modifique o exercício anterior para que cada Slaves receba um vetor referente a linha da matriz que deve ser computada
- Envie mensagens para os Slaves informando o tamanho do vetor que estes vão receber
- Utilize alocação dinâmica
- Esta abordagem reduz o fluxo de dados enviados para a rede.



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Exercício 03

- Implementar em MPMD o exercício 02
- A partir do Código do exercício 02, separe em dois arquivos as funções do master e slave.
  - O primeiro código executa o envio dos dados, coordenação entre os processos e o somatório final.
  - O Segundo código recebe os dados, processa a tarefa de multiplicação de matriz e devolve os resultados ao processo pai.
    - Será necessário incluir as funções de finalização do MPI.
- Para executar
  - `Mpirun -n 1 ./master_code : -n qnt_filhos ./slave_code`



# ■ 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)`

# ■ 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 ←

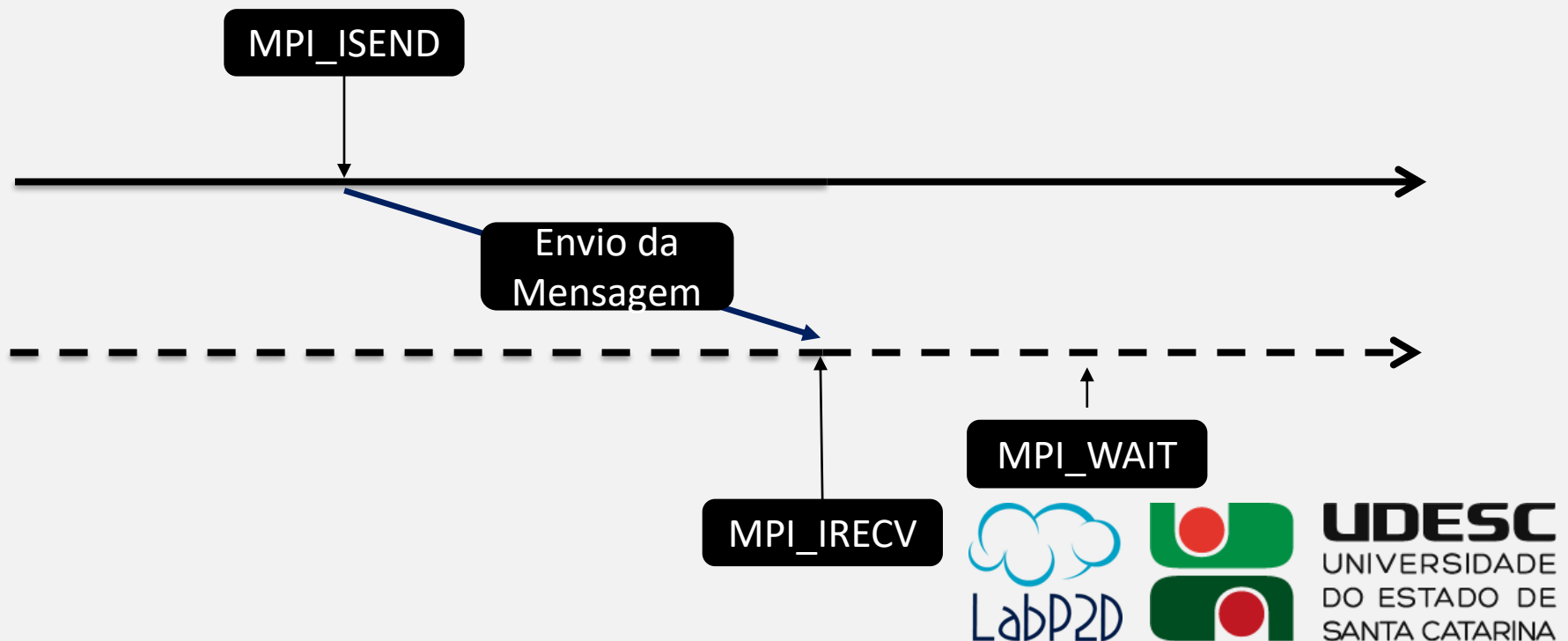
Status da requisição ←

- `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.

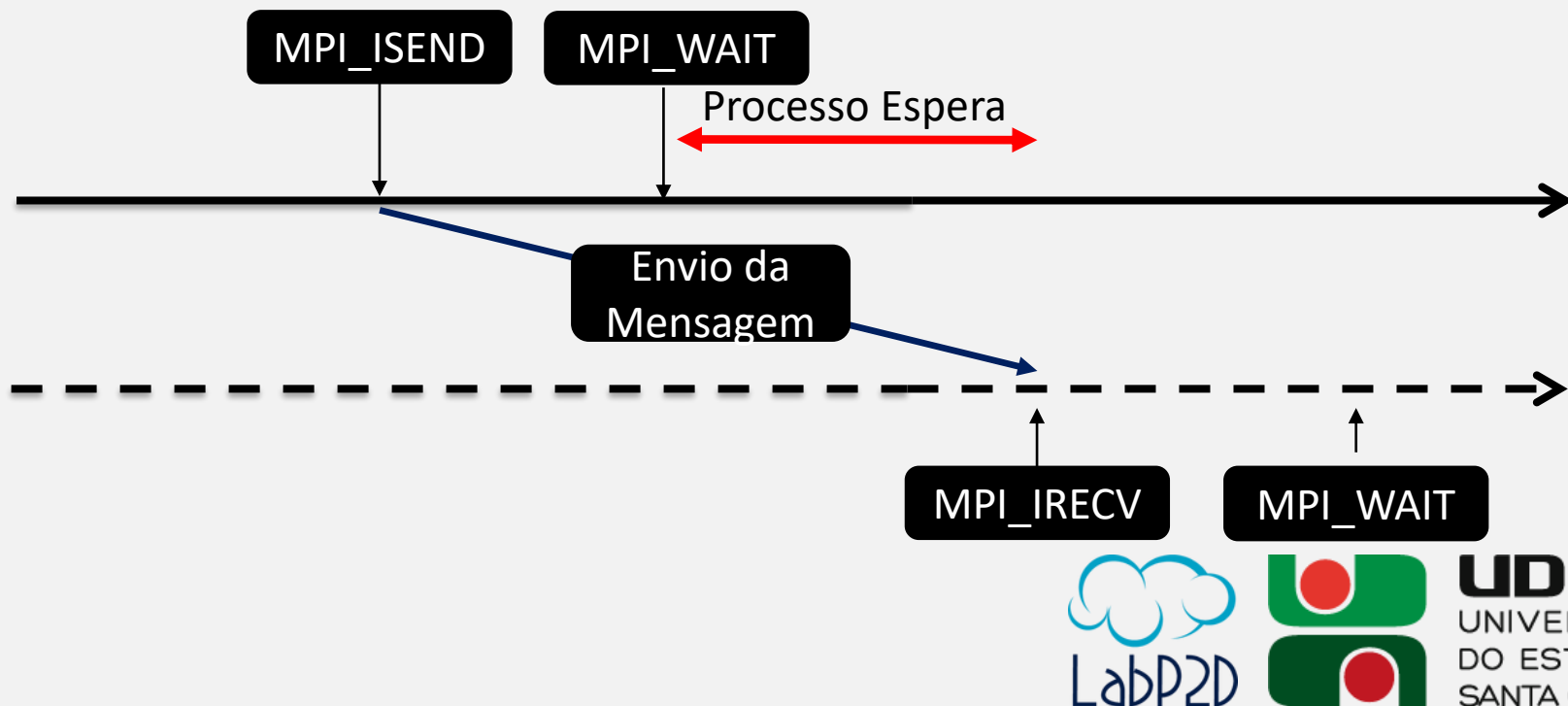




# MPI – Funções Não Bloqueantes

## 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

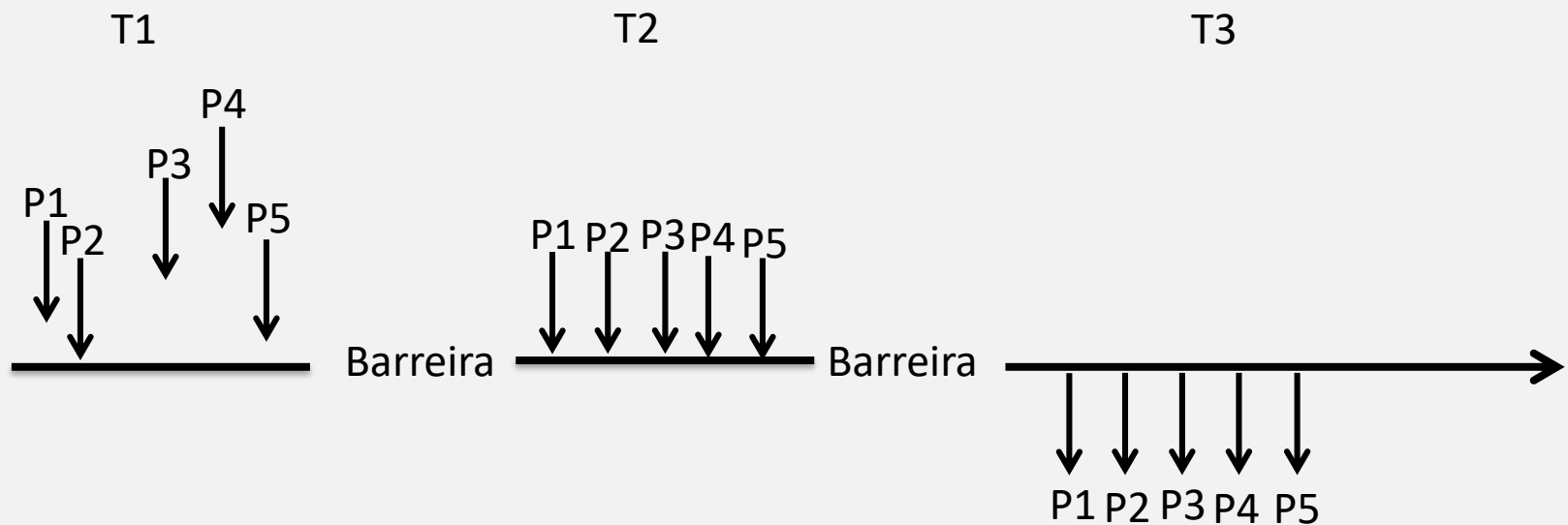
Int MPI\_Barrier ( MPI\_Comm comm)



**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;
}
```

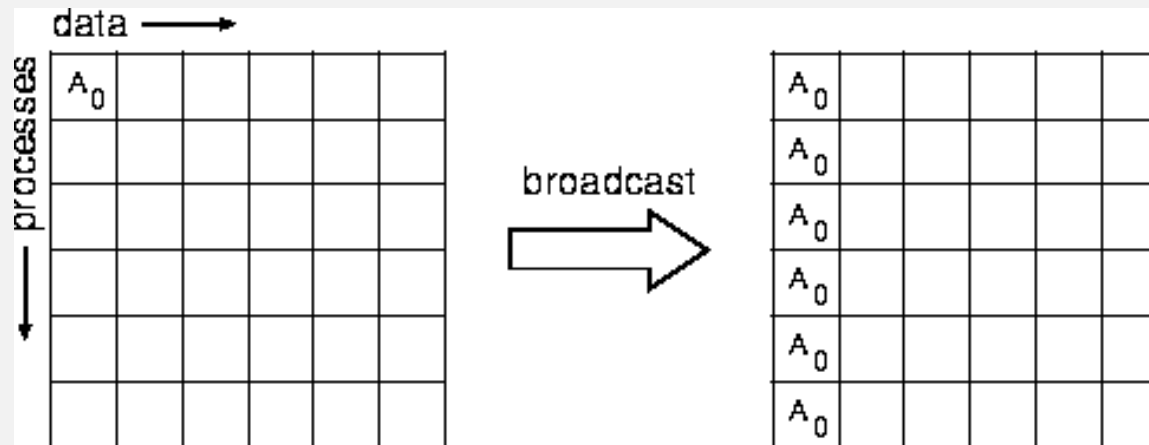


**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# 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

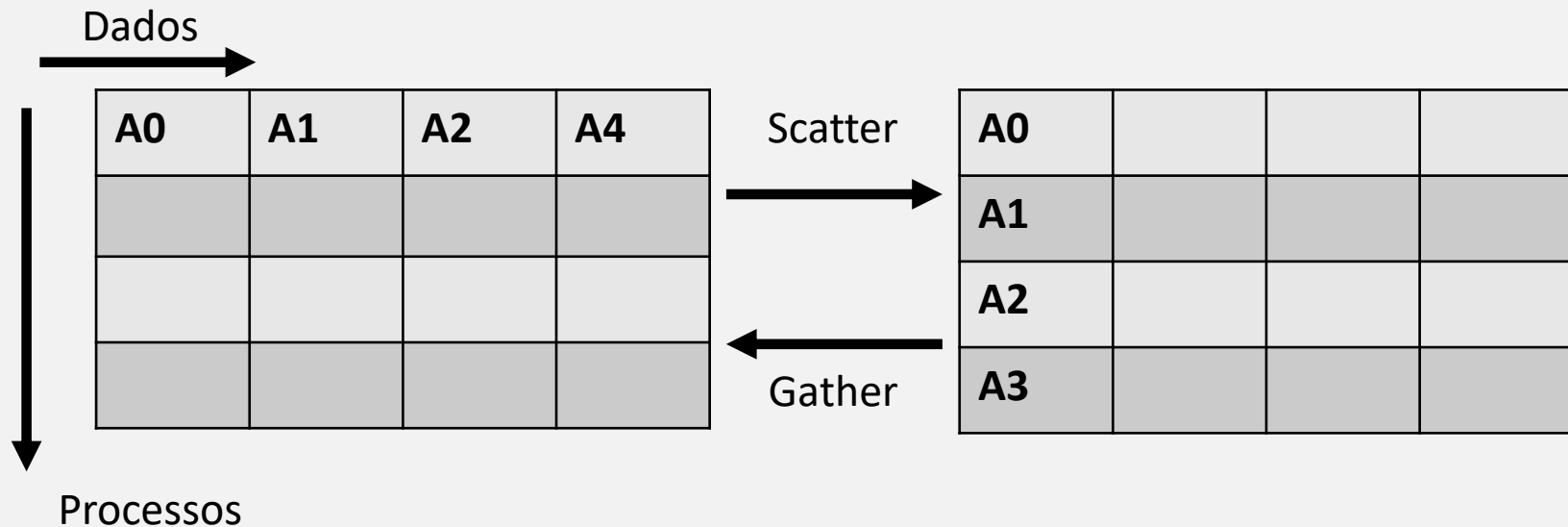
# MPI – Comunicação em grupo

## 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

Diagram illustrating the MPI\_Gather function signature and its parameters:

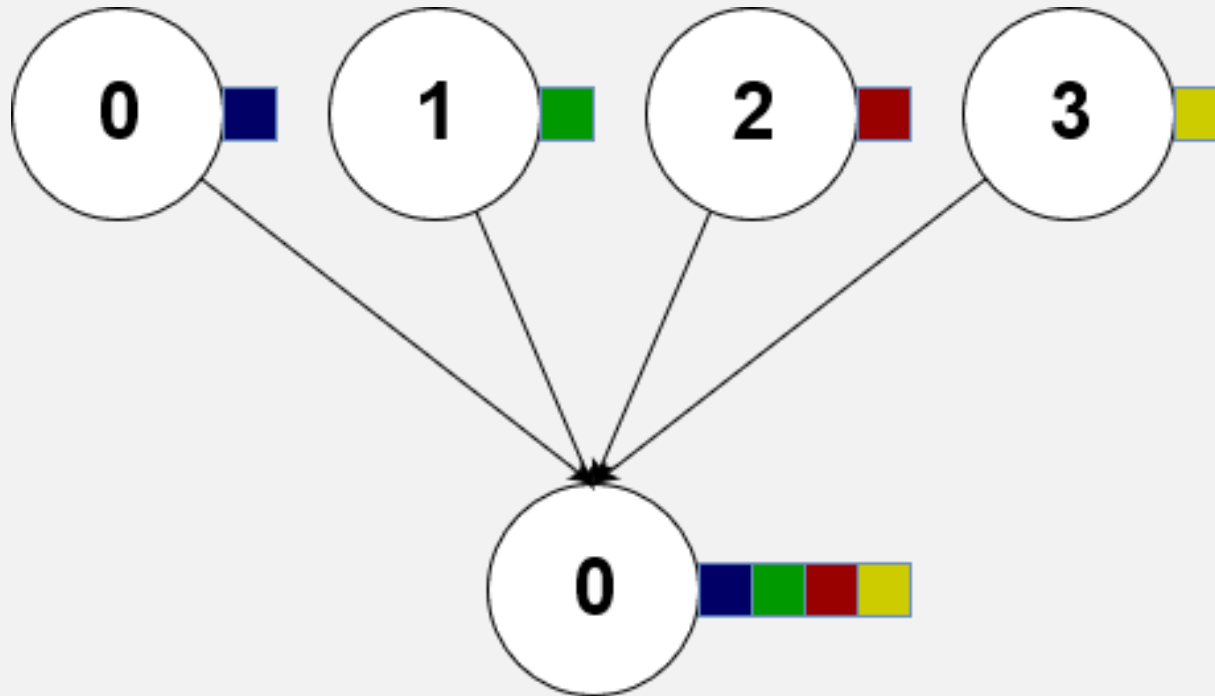
```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype  
sendtype, void recvbuf, int recvcount, MPI_Datatype recvtype, int  
root, MPI_Comm comm)
```

Annotations for MPI\_Gather parameters:

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

# 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

Diagram illustrating the MPI\_Scatter function signature and its parameters:

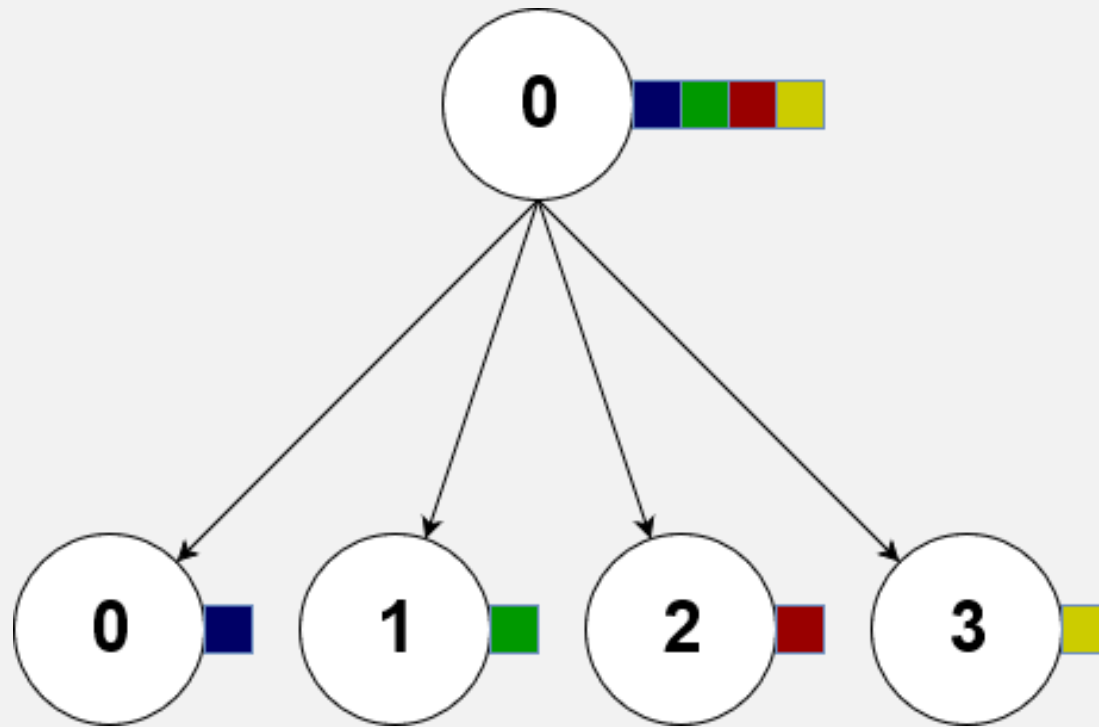
```
int MPI_Scatter(const void sendbuf, int sendcount, MPI_Datatype  
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int  
root, MPI_Comm comm)
```

Annotations for MPI\_Scatter:

- 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;
}
```

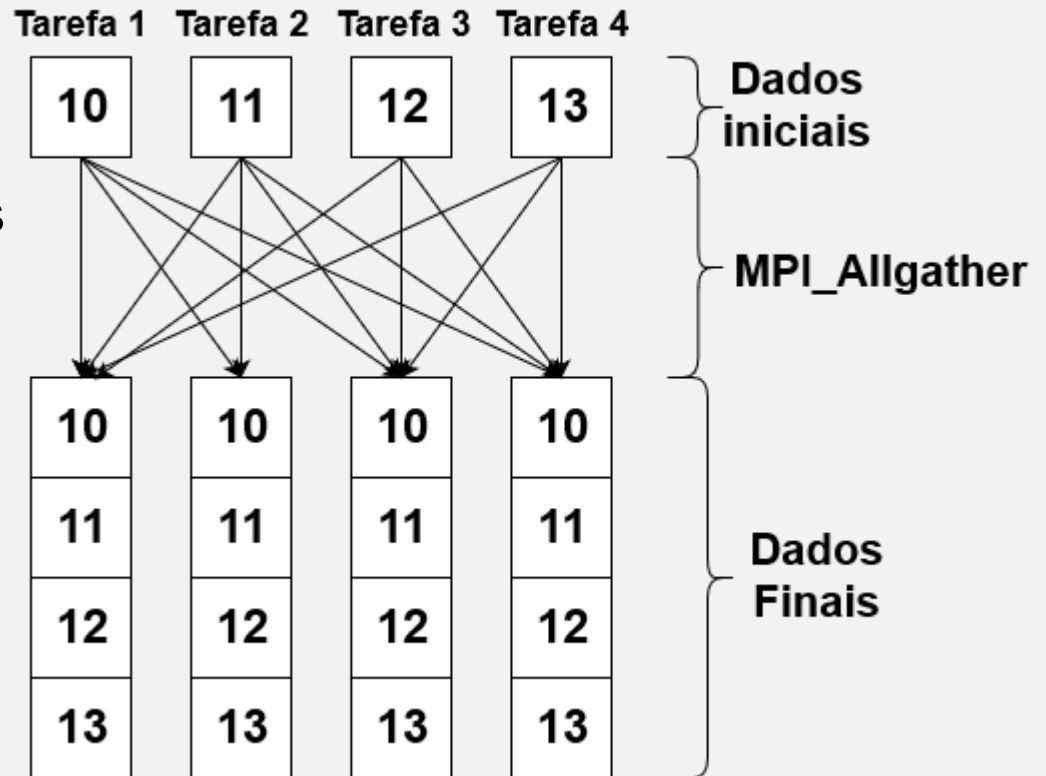


**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Comunicação em grupo

## Allgather

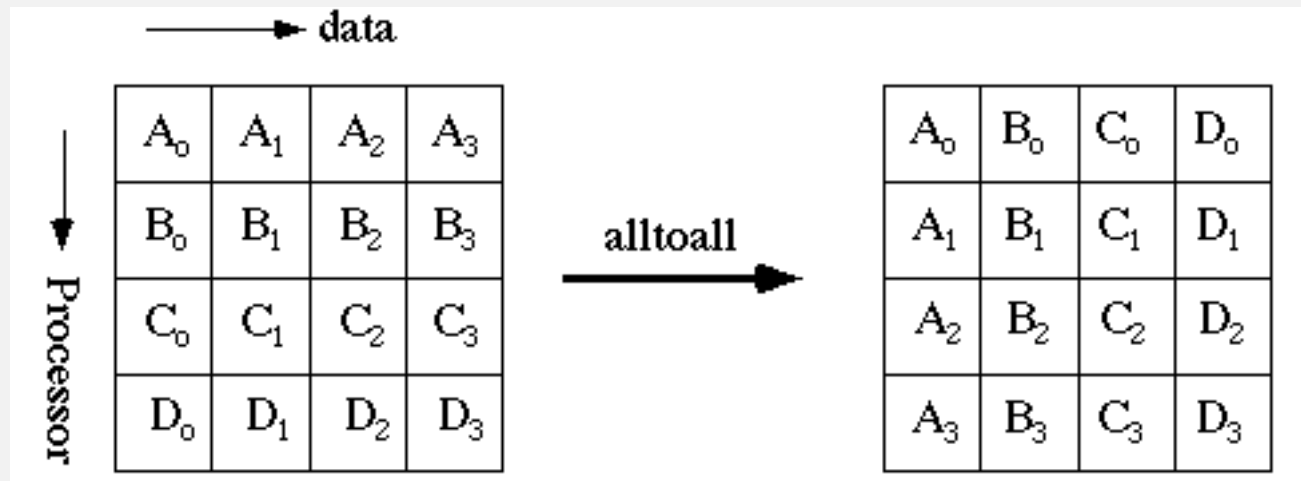
- Reune todos os elementos em todas as tarefas
- Os elementos são ordenados pelo rank do processo emissor



# MPI – Comunicação em grupo

## MPI\_AllToAll

- Esta função faz com que todos os processos enviem e coletem dados de cada processo da aplicação.
- Cada processo efetua um Broadcast.





# MPI – Comunicação em grupo

## MPI\_AlltoAll

Diagram illustrating the parameters of the `MPI_Scatter` function:

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

Annotations:

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

# 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));
    recvbuffer = (int *)malloc(size*sizeof(int));
    if(rank == 0) {
        for(i=0; i<size; i++) sndbuffer[i] = i*i;
    }
    for(i=0; i<size; i++) sndbuffer[i] = i*i+rank;
    printvector(rank, sndbuffer, size);
    MPI_Alltoall(sndbuffer, 1, MPI_INT, recvbuffer, 1, MPI_INT,
                 MPI_COMM_WORLD);
    printvector(rank, sndbuffer, size);
    MPI_Finalize();
    return 0;
}
```

```
void printvector(int rank,
                 int *buffer, int size){
    int i;
    printf("Vetor:\n");
    for (i=0; i<size - 1; i++)
        printf("%d\n", buffer[i]);
}
```



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Comunicação em grupo

## MPI\_Reduce

- O resultado parcial de um processo em determinado grupo é combinado e retornado para um processo específico
- Utiliza um **tipo de função de operação**

Função	Resultado
MPI_MAX	Valor máximo
MPI_MIN	Valor mínimo
MPI_SUM	Somatório
MPI_PROD	produto



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – Comunicação em grupo

## MPI\_Reduce

int MPI\_Alltoall(const void \*sendbuf, void \*recvbuf , int count, int  
sendcount, MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm  
comm)

Endereço inicial  
do buffer ↩

Endereço do  
buffer do  
receptor ↩

Quantidade  
de elementos ↩

↩ Tipo de dados dos  
elementos no buffer

↩ Operação de reduce

↩ Comunicador/grupo

# MPI – Dados

- O MPI possui suporte para dados compostos
- MPI\_Type\_vector
- MPI\_Type\_struct

**Após a criação do novo tipo de dado é necessário dar um commit!**

**MPI\_Type\_commit(&new\_type);**



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# MPI – MPI\_Type\_vector

- Cria um novo tipo de dados com blocos igualmente espaçados

Quantidade de elementos em cada bloco

Número de blocos

Quantidade de elementos entre o início de cada bloco

MPI\_Type\_vector(int Count, int Blocklen, int Stride, MPI\_Datatype Oldtype, MPI\_Datatype \*Newtype)

Dado primitivo

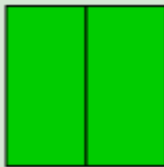
Novo tipo de dados

- **Atenção ao valor inserido no STRIDE**

# MPI – MPI\_Type\_vector

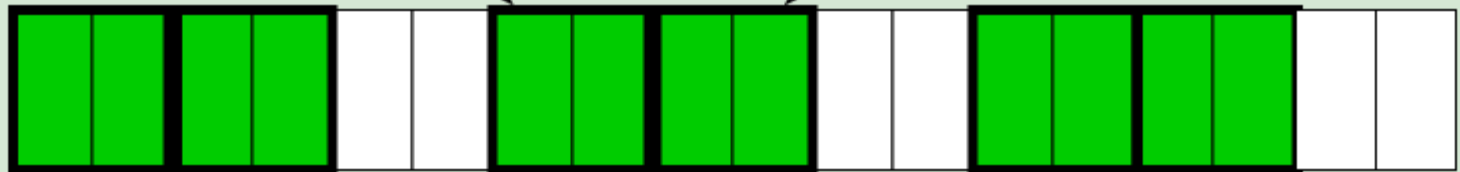
`MPI_Type_vector(3,2,3,Tipo  
Primitivo, Novo Tipo, Erro)`

Tipo  
Primitivo



**BLOCKLEN=2**

Novo  
Tipo

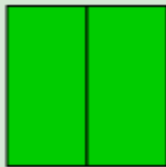


**STRIDE=3**

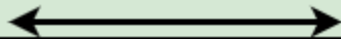
# MPI – MPI\_Type\_vector

`MPI_Type_vector(3,2,2,Tipo  
Primitivo, Novo Tipo, Erro)`

Tipo  
Primitivo



**BLOCKLEN=2**



Novo  
Tipo



**STRIDE=2**





# MPI – MPI\_Type\_struct

- Cria um novo tipo de dados heterogêneos

Quantidade de elementos em cada bloco

Número de blocos

Deslocamentos em cada bloco

```
MPI_Type_vector(int Count, int *Blocklen, MPI_Aint* Stride,  
MPI_Datatype *Types, MPI_Datatype *Newtype)
```

Array com tipos dos dados primitivos

Tipo da struct criada

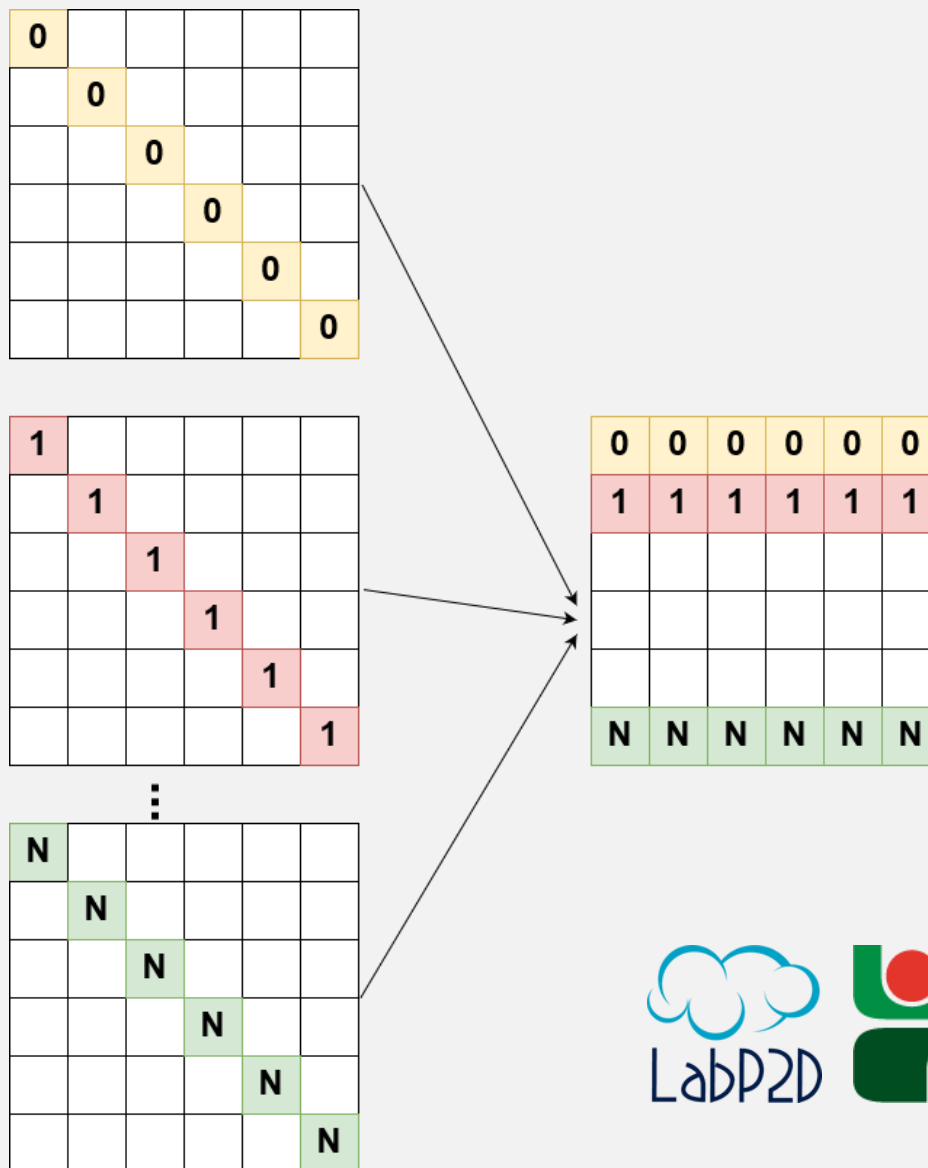
# Exercício 4

- Cada tarefa inicializa uma matriz de tamanho **NxN**, onde **N** é a quantidade de tarefas
- Todos os elementos da matriz são compostos pela formula:
$$a_{i,j} = (j * 10 + i * 3) \% (rank * 2 + 1)$$
- Cada tarefa deve enviar para o Master (rank 0) um vetor contendo a diagonal da matriz
- A tarefa 0 recebe os vetores das demais tarefas e monta uma nova matriz

# Exercício 4

- Utilize o `MPI_Type_vector` para criar um tipo vetor
- Envie um único vetor para a tarefa com rank 0
- Os elementos armazenados pela rank 0 deve ser de forma contínua. O vetor da tarefa **1** é copiado para a linha **1**, tarefa **N** para linha **N**

# Exercício 4



# Exercício 5

- Faça um algoritmo que realize a multiplicação de matrizes na forma

$$C = A \times B \Rightarrow C_{ij} = \sum_{k=0}^N a_{ik} b_{kj}$$

- As matrizes A, B e C possuem tamanhos **NxN**
- A e B são iniciadas pela tarefa de rank 0, através de:

$$a_{ij} = i * j \qquad b_{ij} = \frac{1}{i * j}$$

- Busque minimizar o número de chamadas do MPI
- Como saída mostre as três matrizes separadas por uma linha em branco
- Utilize 2 casas decimais



# Exercício 5

- Cada tarefa é responsável pela linha da matriz correspondente ao seu rank

<b>P0</b>	<b>C11</b>	=	<b>A11</b>	<b>B11</b>	+	<b>A12</b>	<b>B21</b>	+	<b>A12</b>	<b>B31</b>	...
<b>P1</b>	<b>C21</b>	=	<b>A21</b>	<b>B11</b>	+	<b>A22</b>	<b>B21</b>	+	<b>A22</b>	<b>B31</b>	...
<b>P2</b>	<b>C31</b>	=	<b>A31</b>	<b>B11</b>	+	<b>A32</b>	<b>B21</b>	+	<b>A32</b>	<b>B31</b>	...
<b>P3</b>	<b>C41</b>	=	<b>A41</b>	<b>B11</b>	+	<b>A42</b>	<b>B21</b>	+	<b>A42</b>	<b>B31</b>	...

Mesmo elemento para  
todas as tarefas



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA

# Exercícios Testes

- Os arquivos de testes para comparação estão em:

<https://github.com/leorrodriques/mpi>



**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



# Exemplos

- Os exemplos estão disponíveis no moodle



**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



**UDESC**  
UNIVERSIDADE  
DO ESTADO DE  
SANTA CATARINA