MPI
Message Passing Interface

Prof. Dr. Márcio Castro
marcio.castro@ufsc.br

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

1 Introdução

Introdução

- ***Message Passing Interface* (MPI)**
  - É uma **interface de programação** baseada no modelo de trocas de mensagens
  - Um padrão define sua **interface** e suas **funcionalidades**

- **MPI-Forum**
  - Define a **padronização** da interface
  - Discute **inovações** e **novas direções**
  - Verifica a **corretude** e **qualidade** do padrão

https://www.mpi-forum.org

Introdução

- **Escalabilidade**
  - Pode ser utilizado em *clusters* com um grande número de nós
  - Muito utilizado em supercomputadores
- **Portabilidade**
  - Disponíveis em diferentes SOs e linguagens de programação
  - Maior nível de abstração que *sockets*
- **Desempenho**
  - Possui algoritmos de comunicação otimizados

Introdução

- **Implementações de MPI**
  - **OpenMPI:** http://www.open-mpi.org
  - **MPICH:** http://www.mpich.org
  - **LAM/MPI:** http://www.lam-mpi.org

- **OpenMPI no Linux (C/C++)**
  - **Pacote:** `mpi-default-dev`
  - Ferramentas de compilação (`mpicc` e `mpic++`) e execução (`mpiexec` e `mpirun`)

- **MPI para Python**
  - **Pacote:** `python-mpi4py`

Introdução

- **MPI: Modelo Single Program Multiple Data (SPMD)**
  - Todos os processos executam **o mesmo código-fonte**
  - Processos são identificados por **um número único denominado *rank***
  - Os *ranks* dos processos são normalmente utilizados para definir o que cada processo deve executar

Introdução

Exemplo: *Hello world*

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

int main(int argc, char **argv) {
   MPI_Init(&argc, &argv);

   printf("Hello World!\n");

   MPI_Finalize();
   return 0;
}
```

Biblioteca do MPI

Inicializa o MPI

Finaliza o MPI

Compilação

```
$ mpicc hello.c -o hello
```

Execução

Número de processos MPI

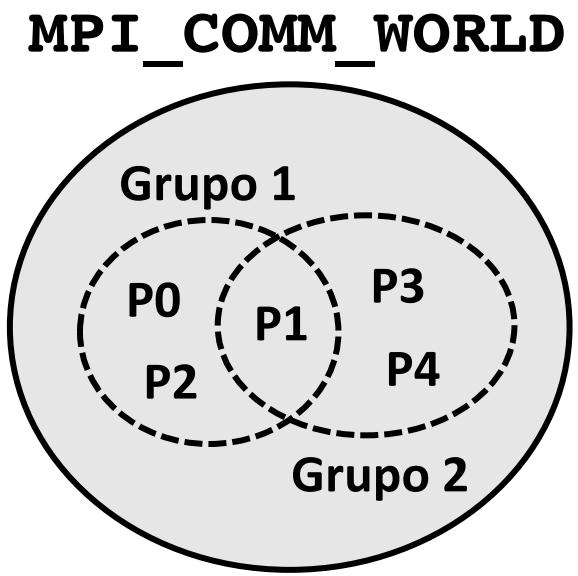
```
$ mpirun -np 3 ./hello
```

Resultado da execução

```
Hello World!
Hello World!
Hello World!
```

Introdução

- **Comunicadores MPI**
  - Permitem agrupar processos
  - Todos os processos criados fazem parte do grupo **`MPI_COMM_WORLD`**



Introdução

Quantos processos MPI existem?

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

- **comm:** comunicador MPI
- **psize:** endereço da variável que irá armazenar o número de processos

Exemplo:

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Introdução

Qual é o *rank* do processo?

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

- **comm:** comunicador MPI
- **rank:** endereço da variável que irá armazenar o resultado (número do *rank*)

Exemplo:

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Introdução

Exemplo: *Hello world 2*

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

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

   if (rank == 0)
      printf("Nb. of processes: %d\n", size);

   printf("Hello from rank %d!\n", rank);

   MPI_Finalize();
   return 0;
}
```

Compilação

```
$ mpicc hello2.c -o hello2
```

Execução

```
$ mpirun -np 3 ./hello2
```

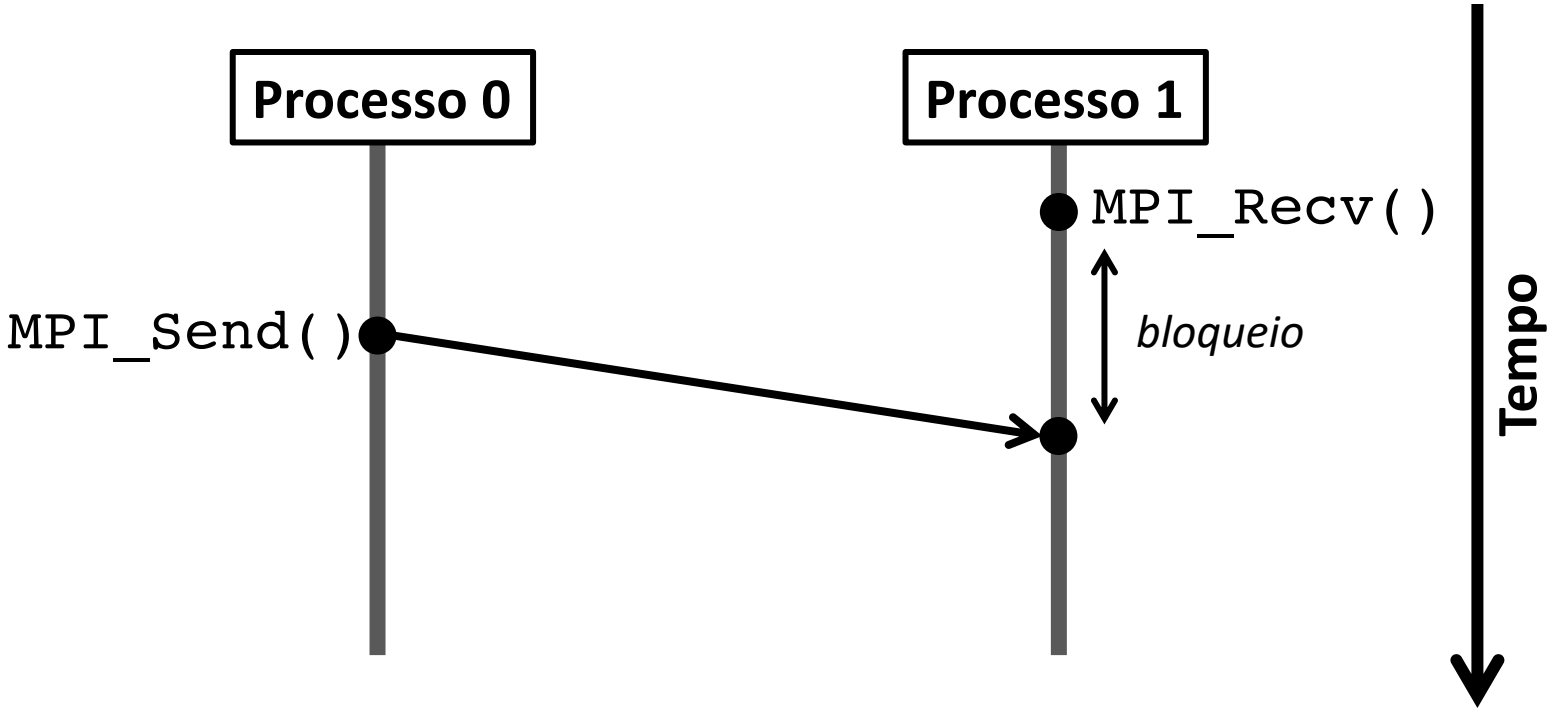
Resultado da execução

```
Hello from rank 1!
Nb. of processes: 3
Hello from rank 0!
Hello from rank 2!
```

2 Comunicação síncrona

Comunicação síncrona

- Destinatário aguarda recebimento da mensagem (**bloqueio**)
- **Remetente desbloqueia o destinatário** quando a mensagem chega no destino



Comunicação síncrona

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
int dest, int tag, MPI_Comm comm)
```

- **buf**: endereço do dado a ser enviado
- **count**: número de elementos a serem enviados
- **dtype**: tipo de dados dos elementos
- **dest**: *rank* do destinatário
- **tag**: um número para "classificar" mensagens
- **comm**: comunicador MPI

Exemplo:

```
int buffer[2] = {1, 2};
MPI_Send(&buffer, 2, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

Comunicação síncrona

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
int src, int tag, MPI_Comm comm, MPI_Status *status)
```

- **buf**: endereço que será utilizado para receber o dado
- **count**: número de elementos a serem recebidos
- **dtype**: tipo de dados dos elementos
- **src**: *rank* do remetente (`MPI_ANY_SOURCE` para receber de qualquer *rank*)
- **tag**: um número para "classificar" mensagens (`MPI_ANY_TAG` para receber mensagens com qualquer `tag`)
- **comm**: comunicador MPI
- **status**: informações sobre a mensagem (`MPI_STATUS_IGNORE` para ignorar)

Exemplo:

```
int buffer[2];
MPI_Status st;
MPI_Recv(&buffer, 2, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &st);
```

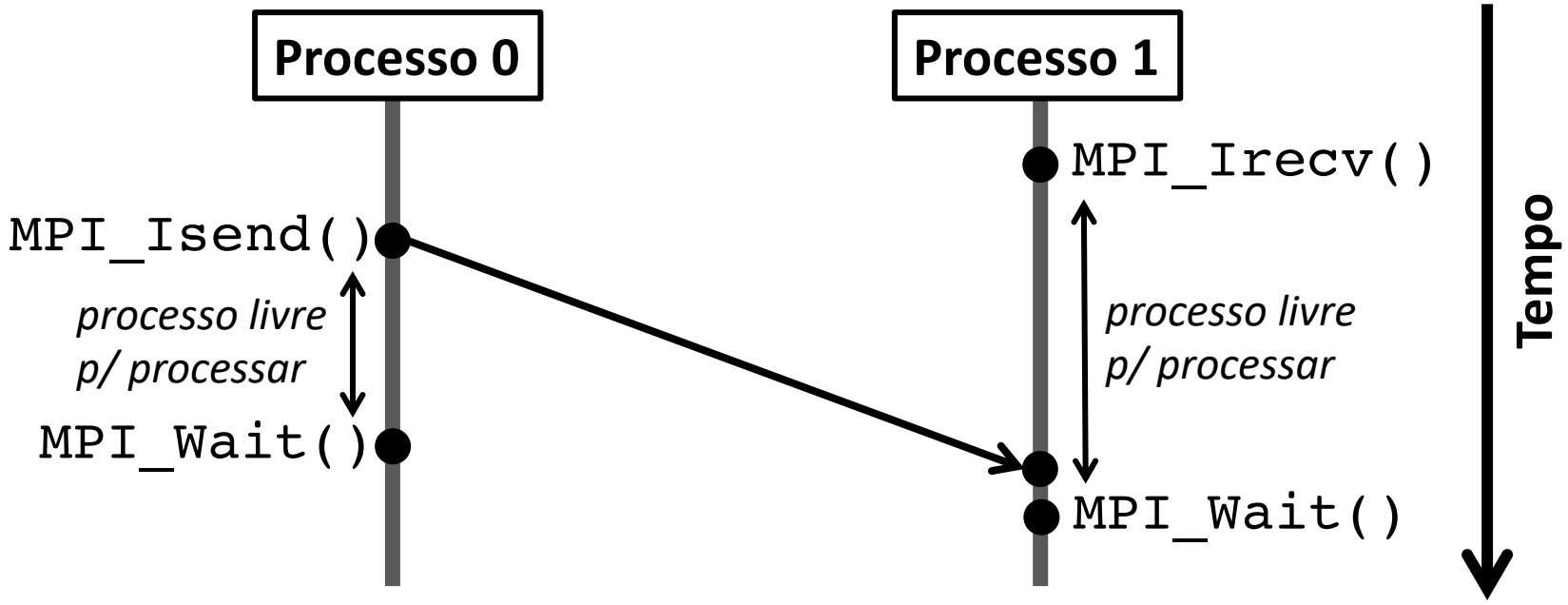
Comunicação síncrona: exemplo

```
int main(int argc, char **argv) {
  int size, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    for (int i = 1; i < size; i++) {
      MPI_Send(&i, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
      printf("Rank %d: sent %d to proc. %d\n", rank, i, i);
    }
  }
  else {
    MPI_Status st; int data;
    MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &st);
    printf("Rank %d: received %d from proc. %d\n", rank, data, st.MPI_SOURCE);
  }
  MPI_Finalize();
  return 0;
}
```

3 Comunicação assíncrona

Comunicação assíncrona

- Destinatário se prepara para receber uma mensagem **sem bloquear**
- Remetente **despacha o envio** e **continua sua execução**
- Uma sincronização com **`MPI_Wait()`** garante que o remetente copiou os dados de envio para um *buffer* do MPI ou que destinatário tenha recebido a mensagem



Comunicação assíncrona

```
int MPI_Isend(void *buf, int count, MPI_Datatype dtype,
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

- **buf**: endereço do dado a ser enviado
- **count**: número de elementos a serem enviados
- **dtype**: tipo dos elementos do *buffer*
- **dest**: *rank* do destinatário
- **tag**: um número para "classificar" as mensagens
- **comm**: comunicador MPI
- **request**: usado para verificar se a mensagem foi enviada ou para aguardar o envio

Exemplo:

```
int buffer;
MPI_Request rq;
MPI_Isend(&buffer, 1, MPI_INT, 2, 0, MPI_COMM_WORLD, &rq);
```

Comunicação assíncrona

```
int MPI_Irecv(void *buf, int count, MPI_Datatype dtype,
int src, int tag, MPI_Comm comm, MPI_Request *request)
```

- **`buf`**: endereço que será utilizado para receber o dado
- **`count`**: número de elementos a serem recebidos
- **`dtype`**: tipo dos elementos do *buffer*
- **`src`**: *rank* do remetente (`MPI_ANY_SOURCE` para receber de qualquer *rank*)
- **`tag`**: um número para "classificar" mensagens (`MPI_ANY_TAG` para receber mensagens com qualquer `tag`)
- **`comm`**: comunicador MPI
- **`request`**: usado para verificar se a mensagem foi recebida ou para aguardar o recebimento

Exemplo:

```
int buffer;
MPI_Request rq;
MPI_Irecv(&buffer, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &rq);
```

Comunicação assíncrona

int **MPI_Wait**(MPI_Request ***request**, MPI_Status ***status**)

- **request**: é o `request` usado no **`MPI_Isend()`** ou **`MPI_Irecv()`**
- **status**: armazena informações sobre a mensagem recebida (`MPI_STATUS_IGNORE` para ignorar esse parâmetro)

Exemplo:

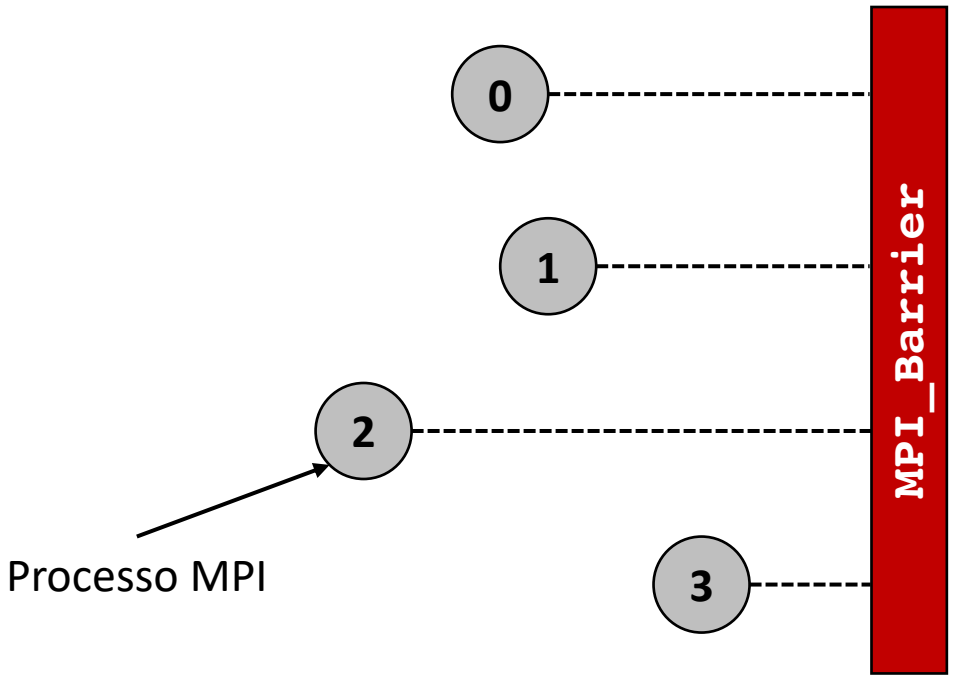
```
int buffer;
MPI_Status st;
MPI_Request rq;
MPI_Irecv(&buffer, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &rq);
MPI_Wait(&rq, &st);
```

4 Comunicações coletivas

Comunicações coletivas: barreira

`int` **`MPI_Barrier(MPI_Comm comm)`**

- Barreira de sincronização entre todos os processos
- Todos os processos do comunicador precisam chamar a função



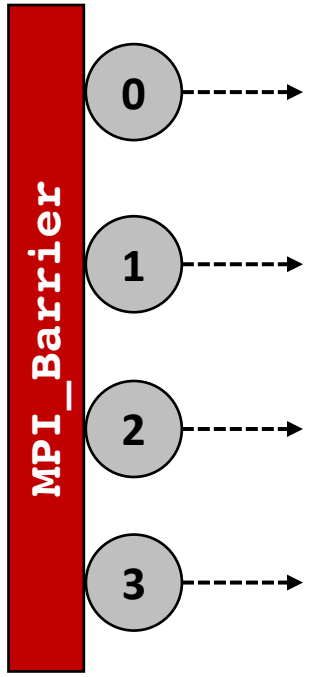
Exemplo:

```
MPI_Barrier(MPI_COMM_WORLD);
```

Comunicações coletivas: barreira

`int` **`MPI_Barrier(MPI_Comm comm)`**

- Barreira de sincronização entre todos os processos
- Todos os processos do comunicador precisam chamar a função



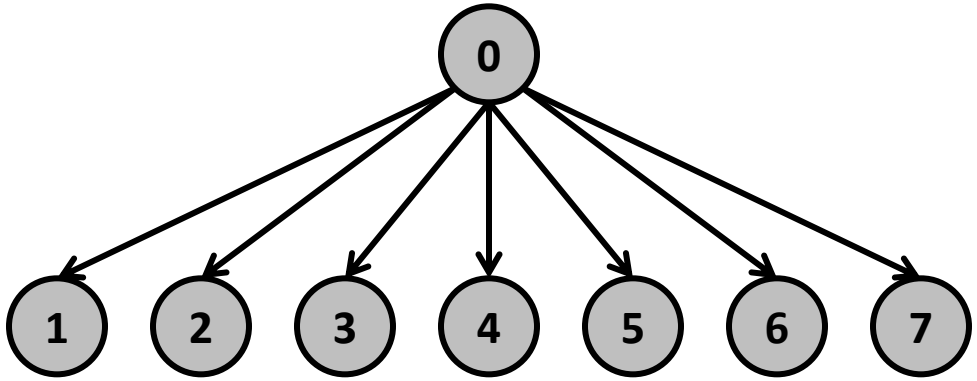
Exemplo:

```
MPI_Barrier(MPI_COMM_WORLD);
```

Comunicações coletivas: *broadcast*

- **Broadcast**
  - É possível implementar um *broadcast* com `MPI_Send()` e `MPI_Recv()`

Exemplo com 8 processos



Comunicações coletivas: *broadcast*

```
int MPI_Bcast(void *buf, int count, MPI_Datatype
dtype, int root, MPI_Comm comm)
```

- **buf**: endereço o dado a ser enviado/recebido
- **count**: número de elementos a serem enviados/recebidos
- **dtype**: tipo dos elementos do *buffer*
- **root**: *rank* do processo que envia
- **comm**: comunicador MPI

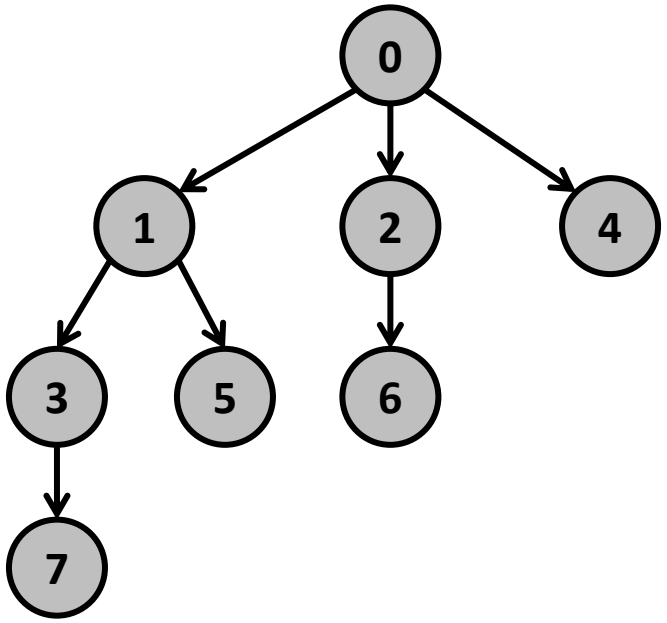
Exemplo:

```
int buffer;
MPI_Bcast(&buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Comunicações coletivas: *broadcast*

```
int MPI_Bcast(void *buf, int count,
MPI_Datatype dtype, int root, MPI_Comm comm)
```

Exemplo com 8 processos



5 Visão geral de outras *features* do MPI

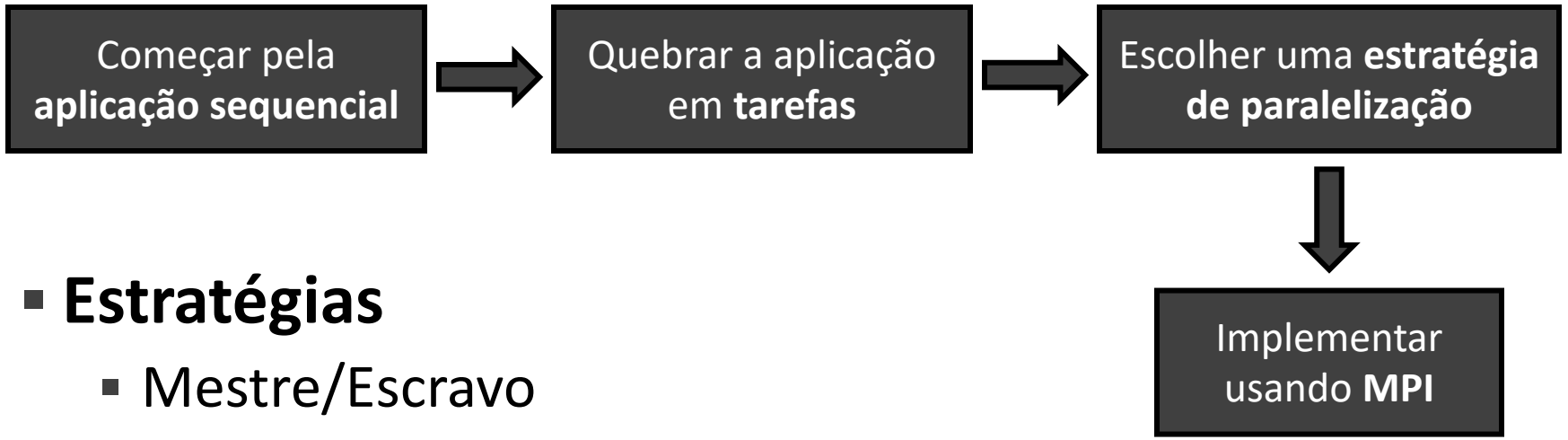
Outras *features* do MPI

- **Grande variedade de comunicações coletivas:**
  - Scatter, gather, gather-to-all, all-to-all scatter/gather, scan, ...
  - *Global reduction operations*
- **Topologia de processos**
  - Permite definir o mapeamento de processos MPI para os recursos de *hardware*
- ***One-sided communications***
  - Put, get, *accumulate functions, ...*
- **Operações de I/O**
- ***Profiling interface***

6 Estratégias de paralelização

Estratégias de paralelização

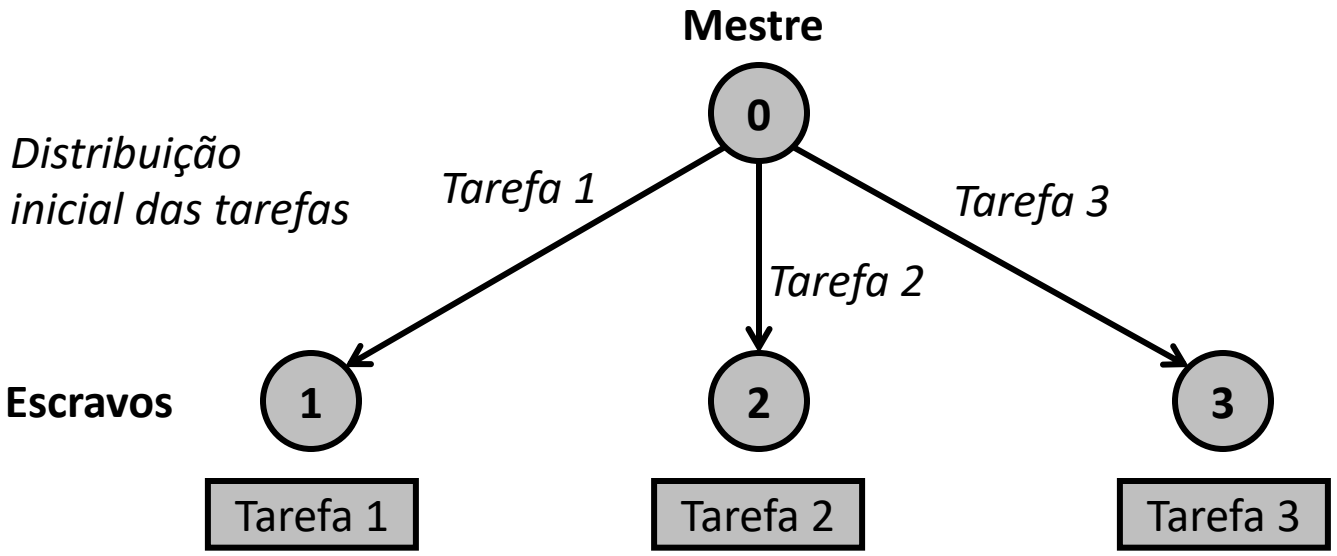
Paralelização de uma aplicação sequencial



- **Estratégias**
  - Mestre/Escravo
  - Pipeline
  - Divisão e Conquista

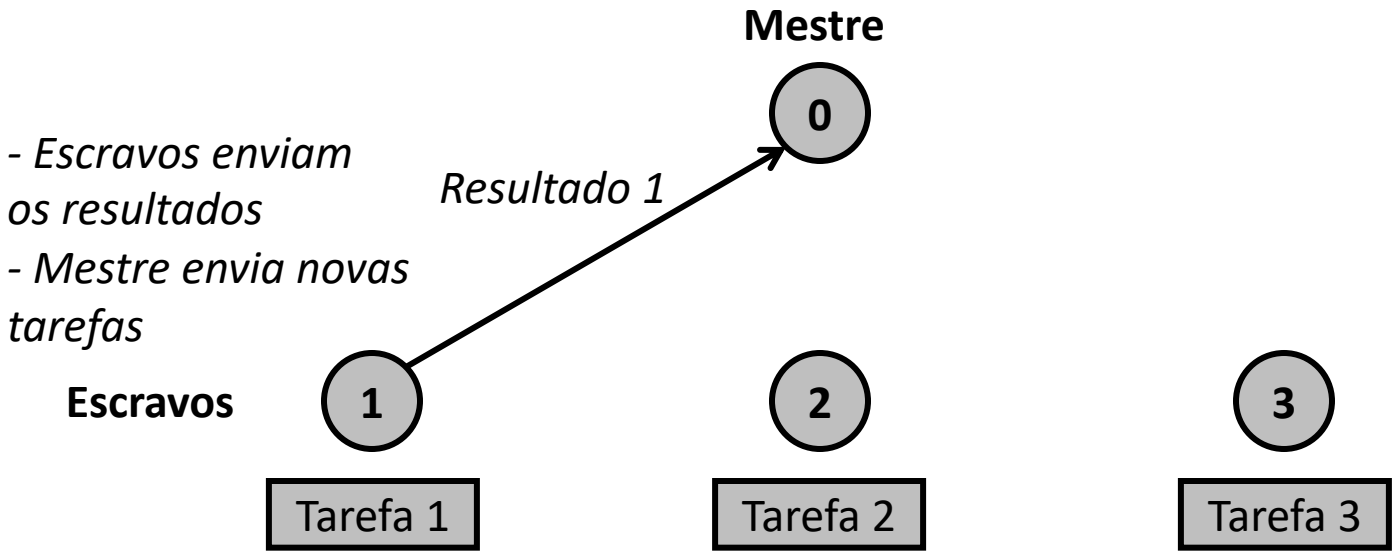
Estratégias de paralelização: mestre/escravo

- **Processo mestre centraliza** todas as tarefas
- Processo mestre **envia tarefas** de acordo com a **demanda**
- **Processo escravos:**
  - **Processam** as tarefas
  - **Enviam os resultados** ao processo mestre
  - **Solicitam** mais tarefas ao processo mestre



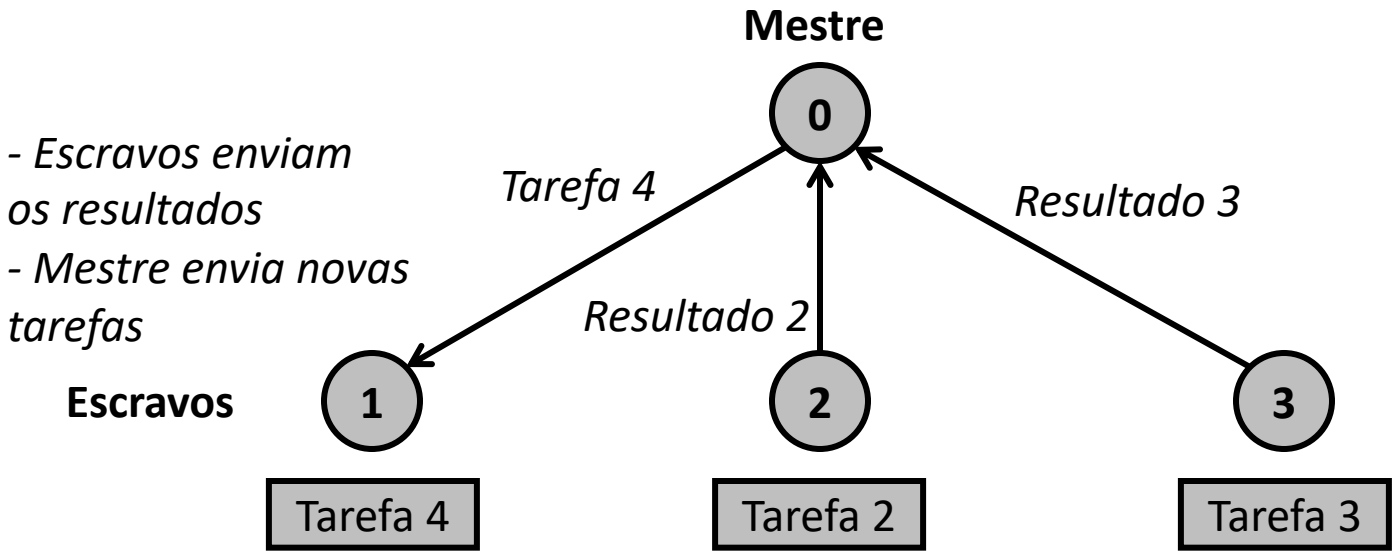
Estratégias de paralelização: mestre/escravo

- **Processo mestre centraliza** todas as tarefas
- Processo mestre **envia tarefas** de acordo com a **demanda**
- **Processo escravos:**
  - **Processam** as tarefas
  - **Enviam os resultados** ao processo mestre
  - **Solicitam** mais tarefas ao processo mestre



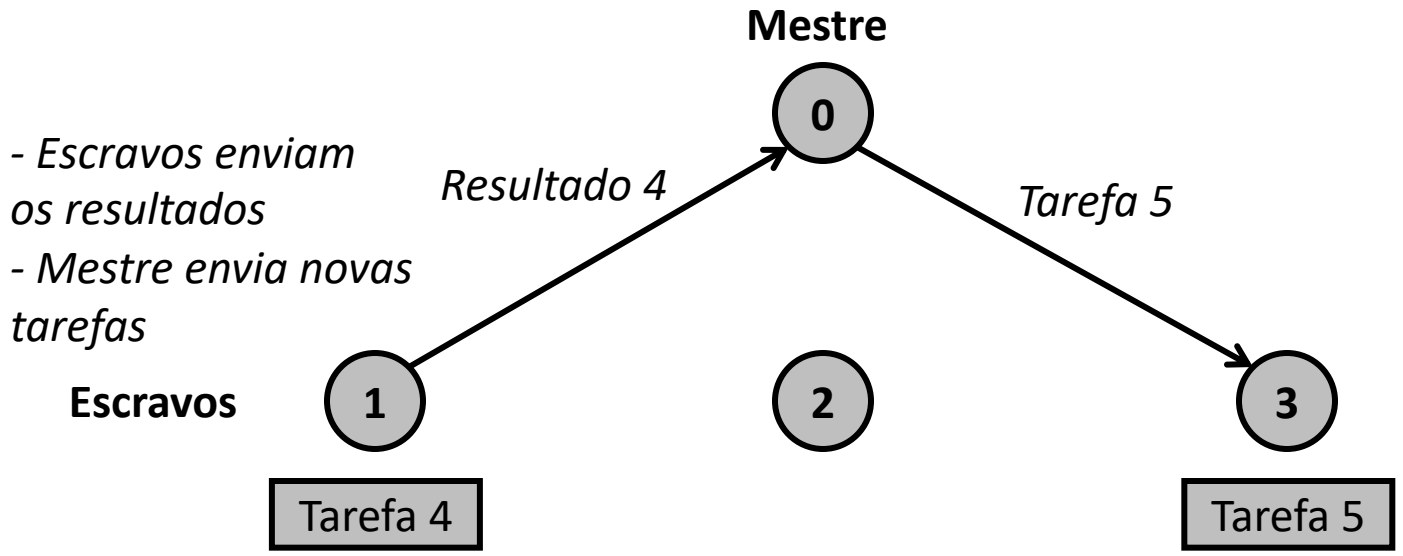
Estratégias de paralelização: mestre/escravo

- **Processo mestre centraliza** todas as tarefas
- Processo mestre **envia tarefas** de acordo com a **demanda**
- **Processo escravos:**
  - **Processam** as tarefas
  - **Enviam os resultados** ao processo mestre
  - **Solicitam** mais tarefas ao processo mestre



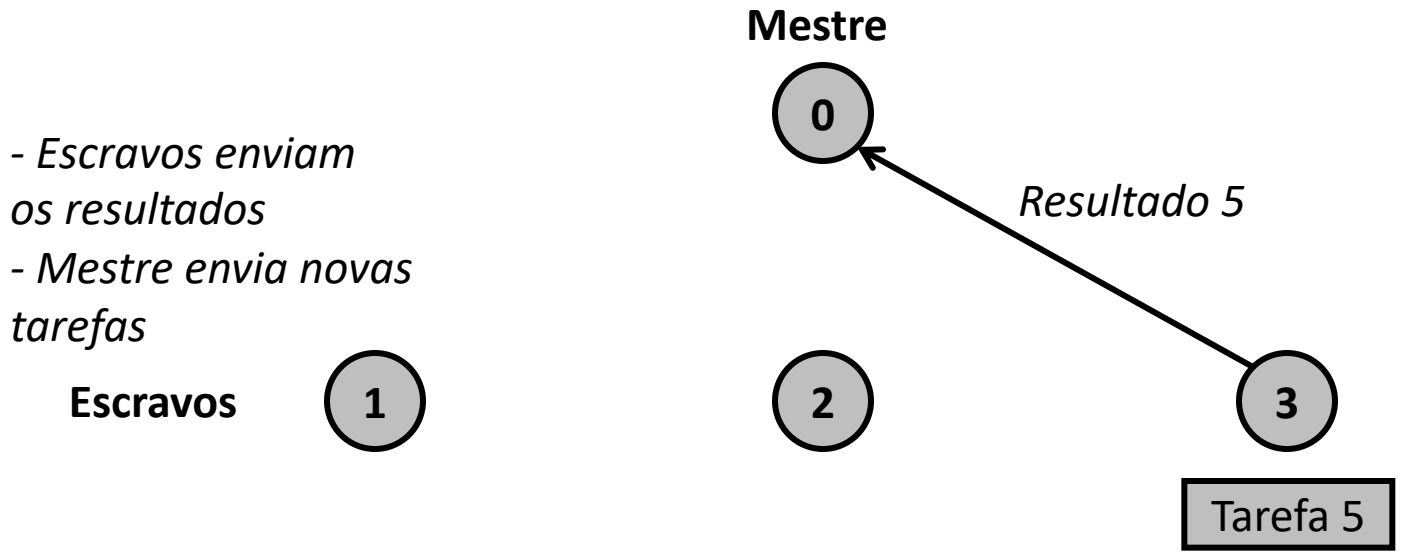
Estratégias de paralelização: mestre/escravo

- **Processo mestre centraliza** todas as tarefas
- Processo mestre **envia tarefas** de acordo com a **demanda**
- **Processo escravos:**
  - **Processam** as tarefas
  - **Enviam os resultados** ao processo mestre
  - **Solicitam** mais tarefas ao processo mestre



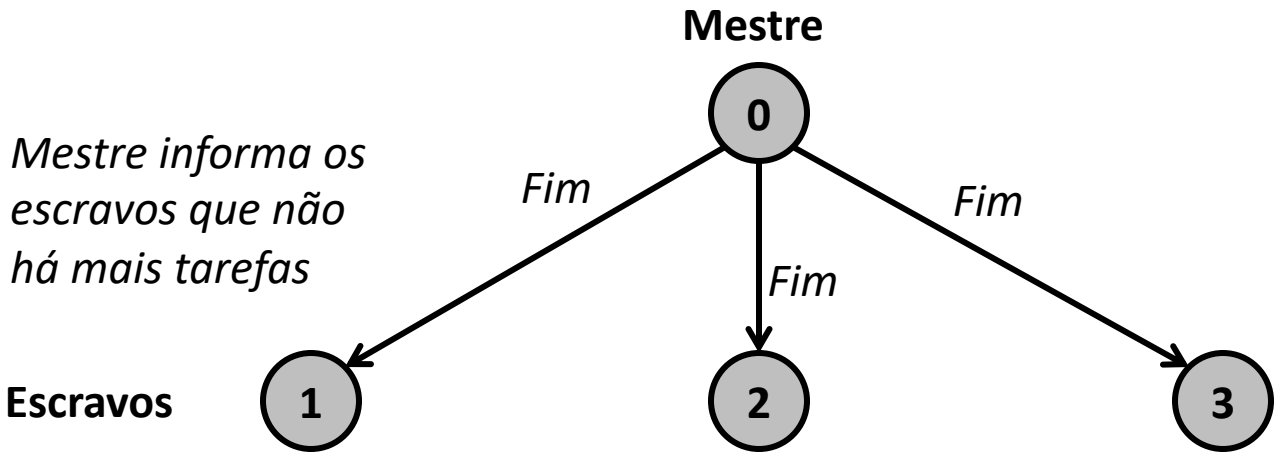
Estratégias de paralelização: mestre/escravo

- **Processo mestre centraliza** todas as tarefas
- Processo mestre **envia tarefas** de acordo com a **demanda**
- **Processo escravos:**
  - **Processam** as tarefas
  - **Enviam os resultados** ao processo mestre
  - **Solicitam** mais tarefas ao processo mestre



Estratégias de paralelização: mestre/escravo

- **Processo mestre centraliza** todas as tarefas
- Processo mestre **envia tarefas** de acordo com a **demanda**
- **Processo escravos:**
  - **Processam** as tarefas
  - **Enviam os resultados** ao processo mestre
  - **Solicitam** mais tarefas ao processo mestre

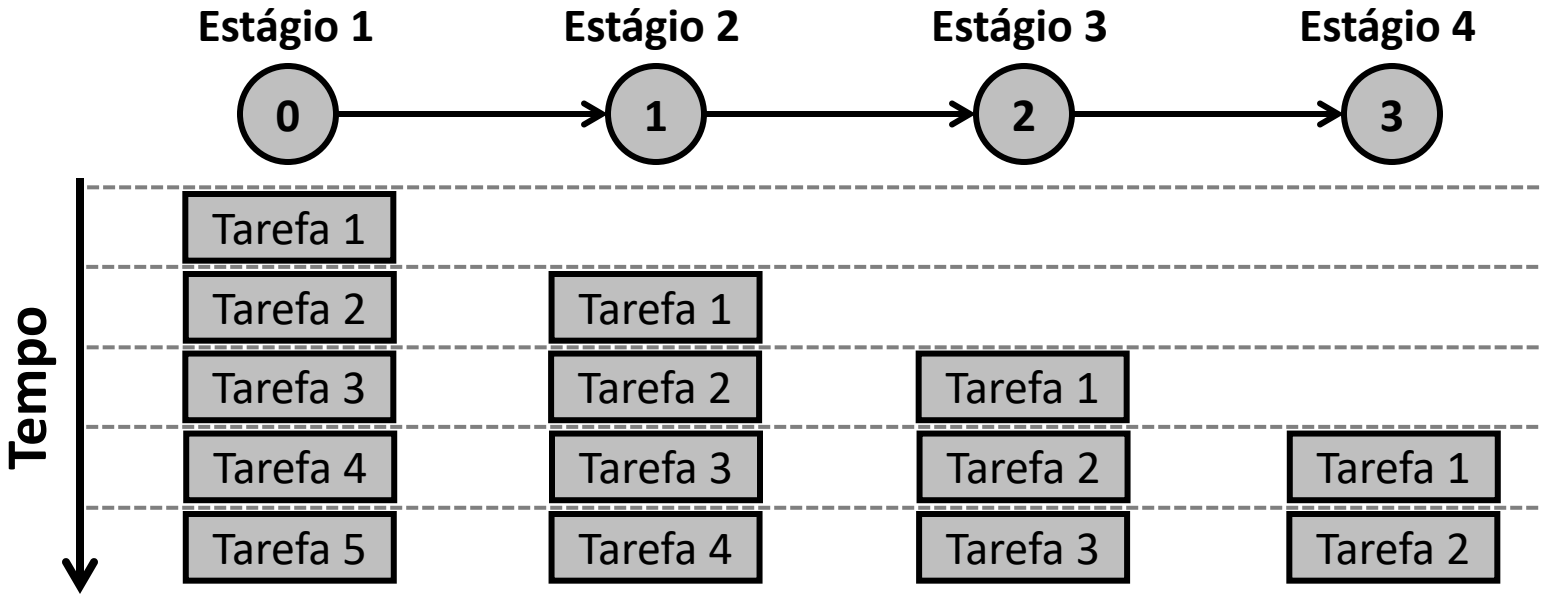


Estratégias de paralelização: mestre/escravo

- **Prós**
  - **Balanceamento de carga**
  - Bom para **plataformas heterogêneas**
- **Contras**
  - Mestre pode se tornar um **gargalo**
  - **Escalabilidade** pode ficar **limitada** devido a centralização das tarefas no mestre

Estratégias de paralelização: pipeline

- Processamento é dividido em **estágios**
- Cada processo faz uma **operações específicas em cada estágio**
- Paralelismo máximo quando todos os estágios estão processando

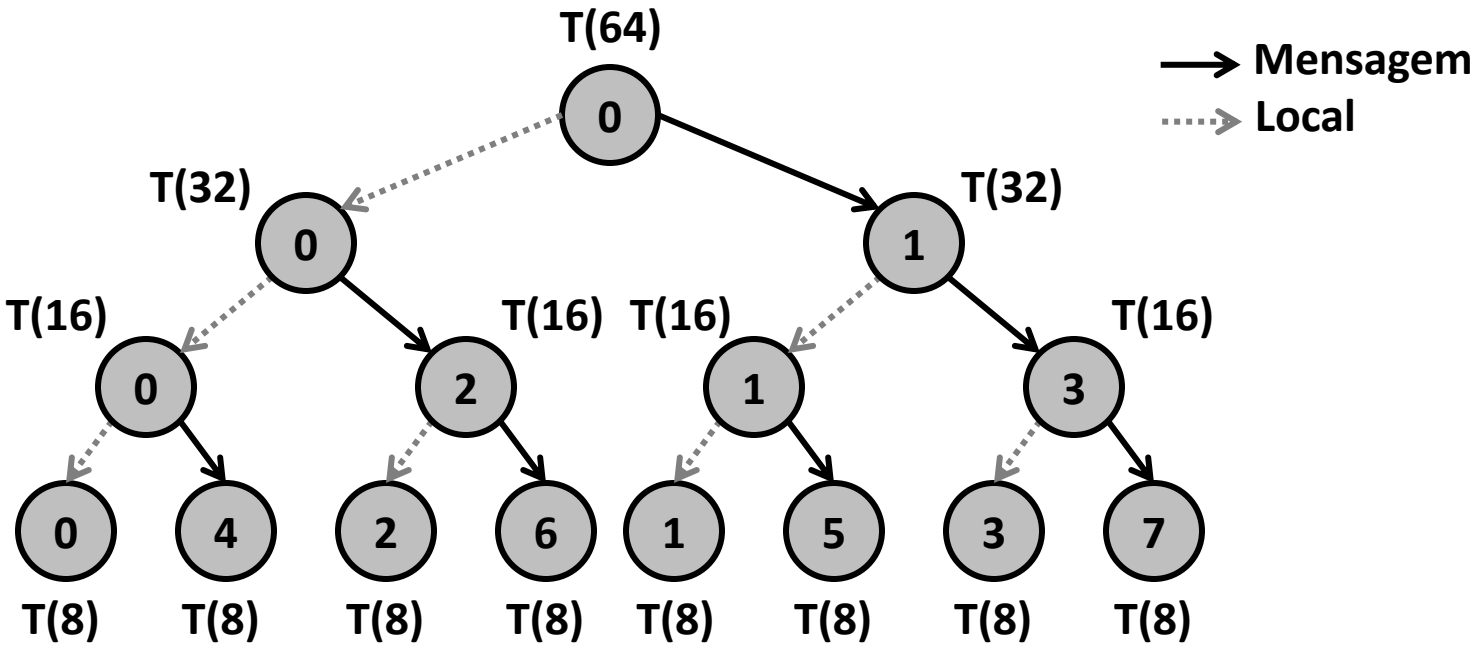


Estratégias de paralelização: pipeline

- **Prós**
  - Bom para problemas que são parcialmente sequenciais por natureza
- **Contras**
  - **Escalabilidade limitada** pelo número de estágios do *pipeline*
  - Sincronização pode resultar em **bolhas**
    - **Exemplo**: estágio 2 muito mais lento que o estágio 3
  - **Ruim para plataformas heterogêneas**

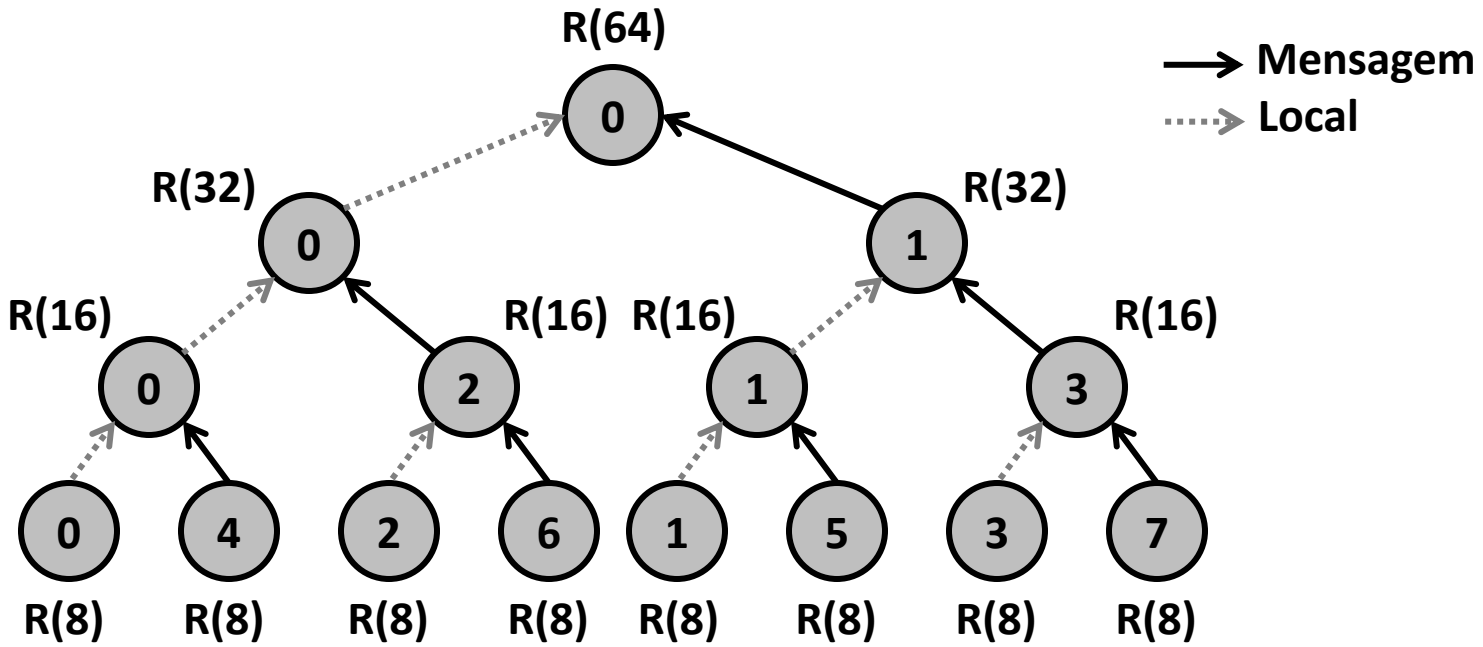
Estratégias de paralelização: divisão e conquista

- Parte de uma única tarefa a ser processada
- Divide **recursivamente** as tarefas em tarefas de menor tamanho
- Processa, sequencialmente, tarefas quando possuírem um tamanho **suficientemente pequeno**



Estratégias de paralelização: divisão e conquista

- Parte de uma única tarefa a ser processada
- Divide **recursivamente** as tarefas em tarefas de menor tamanho
- **Processa**, sequencialmente, tarefas quando possuírem um tamanho **suficientemente pequeno**



Estratégias de paralelização: divisão e conquista

- **Prós**
  - **Escalável**
  - Bom para problemas **recursivos**
  - Bom para problemas de **otimização combinatória**
- **Contras**
  - Na prática, pode ser difícil de quebrar as tarefas de forma arbitrária
  - Pode gerar desbalanceamento de carga se as tarefas forem irregulares

Obrigado pela atenção!

Dúvidas? Entre em contato:

- marcio.castro@ufsc.br
- www.marciocastro.com

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

LaPeSD

Distributed Systems Research Lab

www.lapesd.inf.ufsc.br