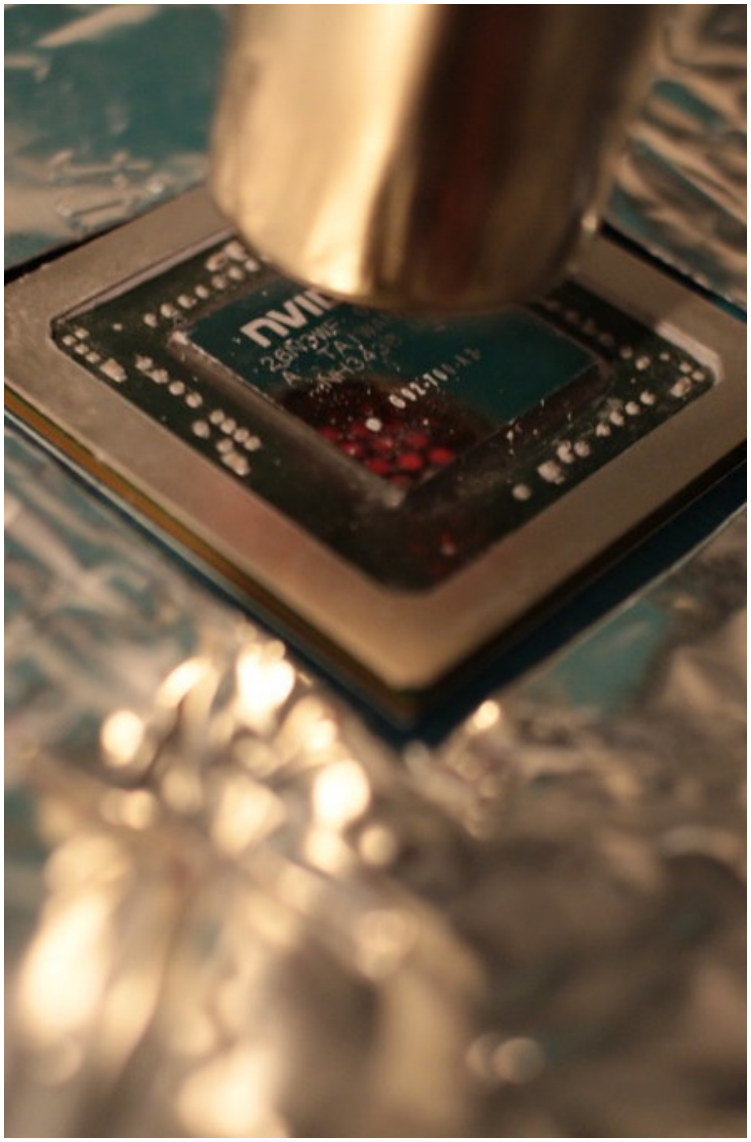
Insper

Supercomputação



MPI

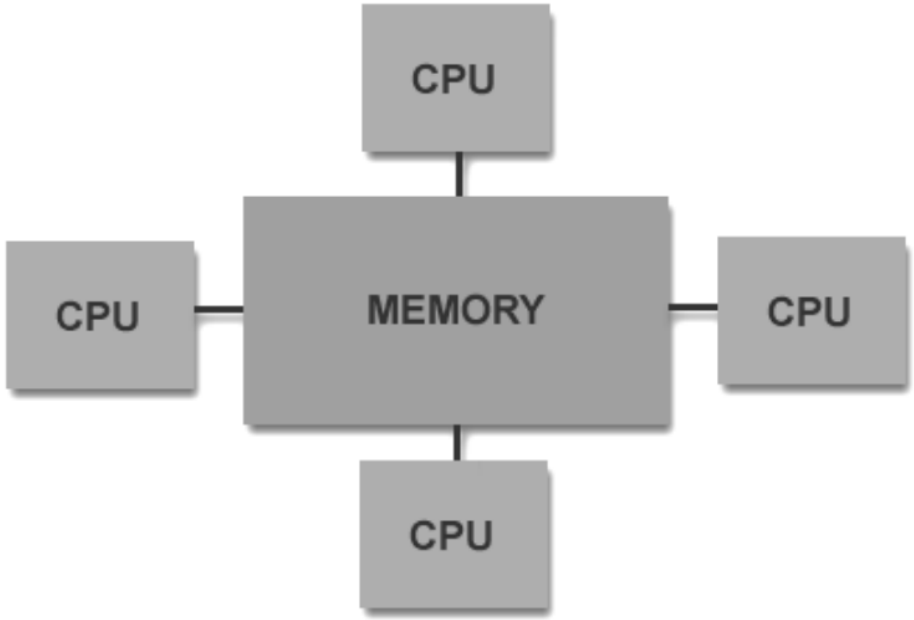
- Introdução ao MPI (*Message Passing Interface*)

O que vimos até hoje?

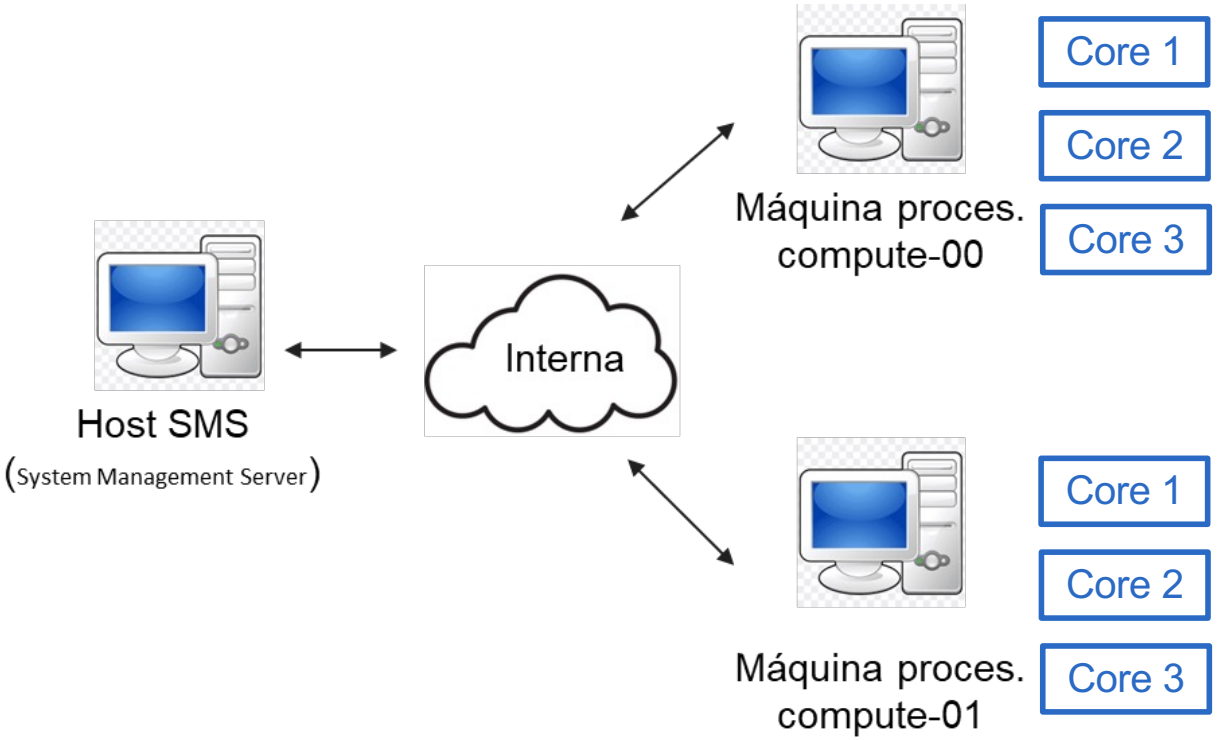
- Temos um “problemão” para resolver (muitos dados e/ou muitos cálculos)
- Como podemos tratar o problema?

Até onde vai OpenMP?

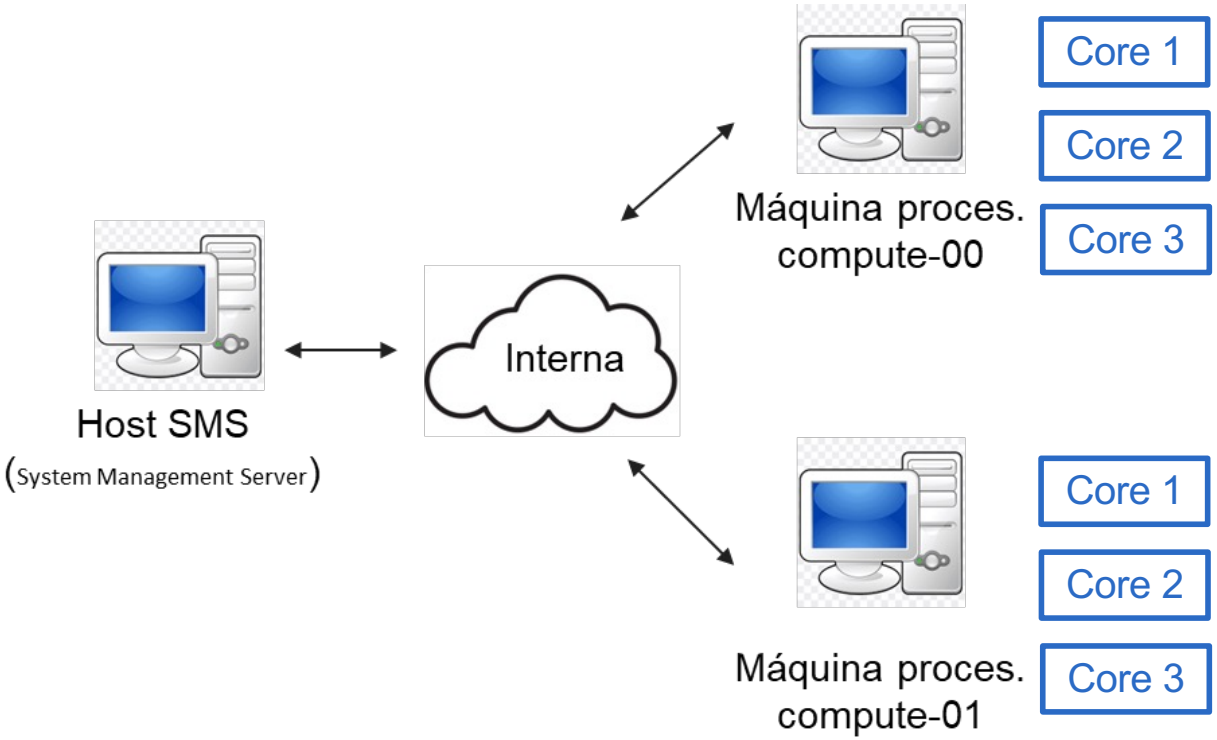
- Focado em ambientes de memória compartilhada
- "Paralelismo local"
- Estou "usando ao máximo" **a máquina** que tenho



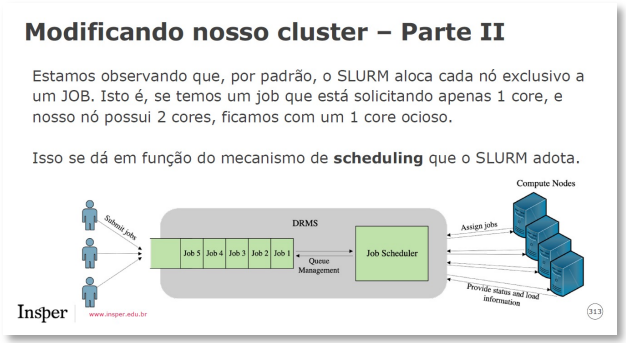
E o nosso cluster?



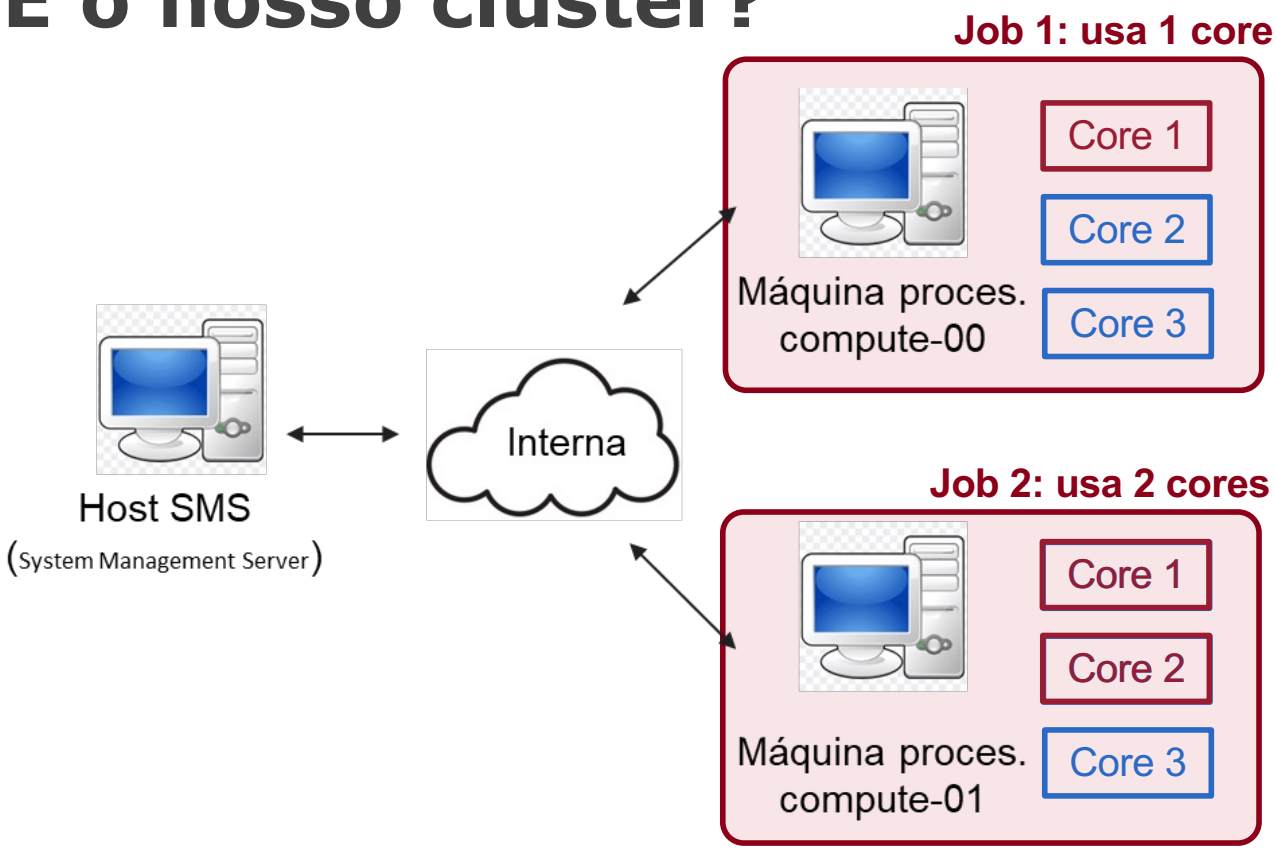
E o nosso cluster?



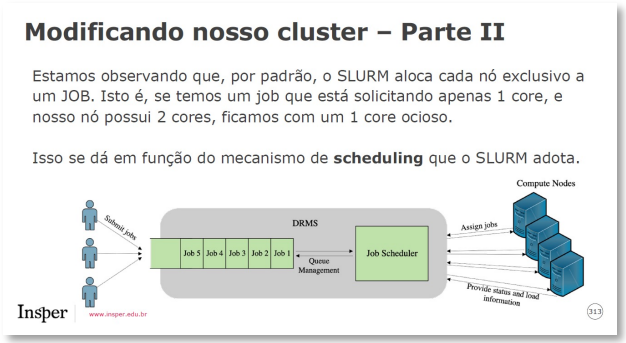
Funcionamento padrão



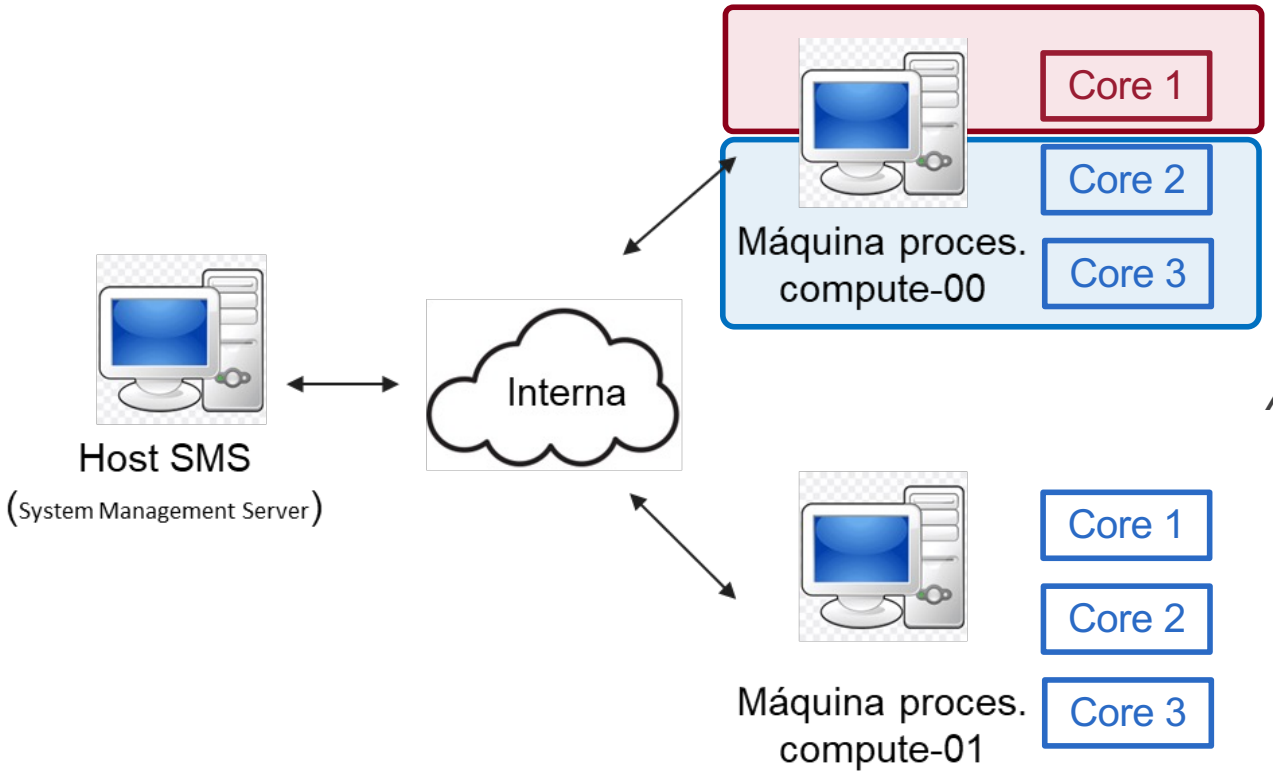
E o nosso cluster?



Funcionamento padrão



E o nosso cluster?

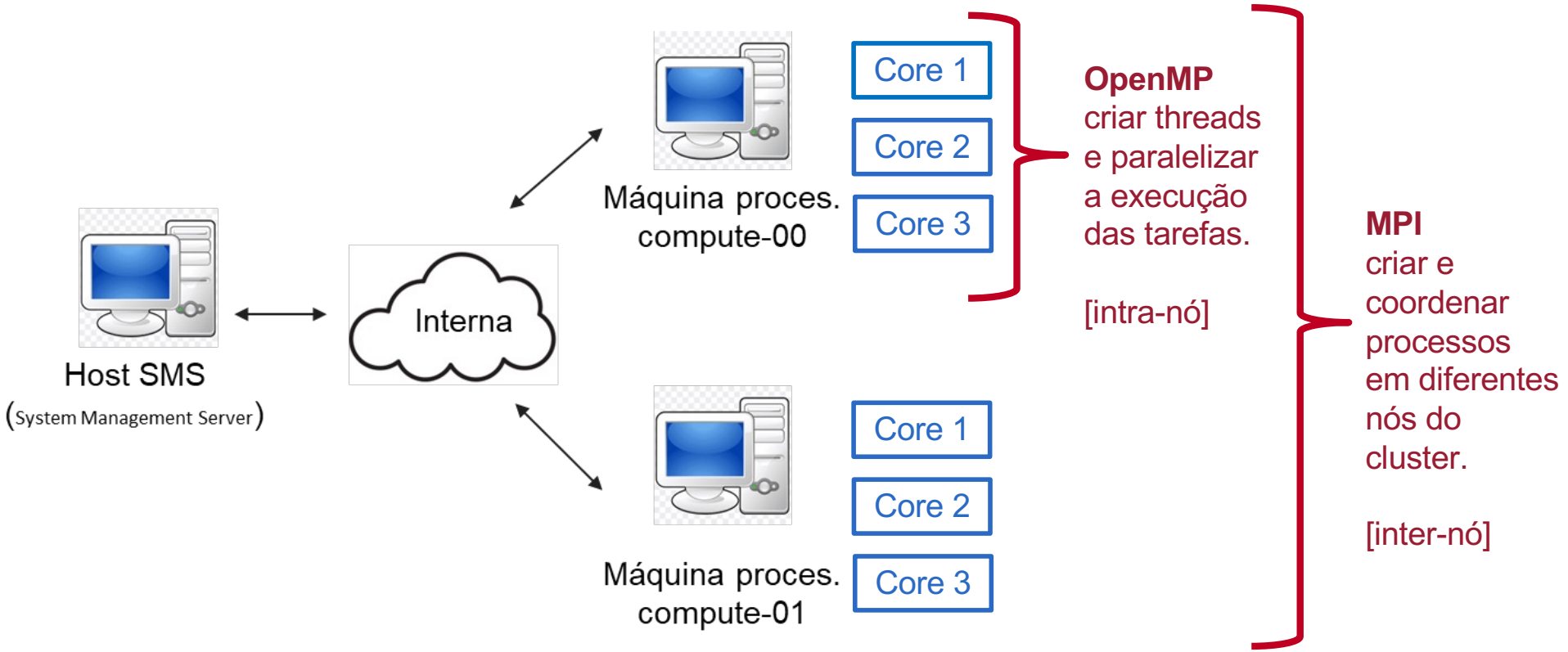


*Alterando o funcionamento padrão para "**recursos consumíveis**"*

Então...

- **OpenMP**: paralelismo local, com memória compartilhada
- **Cluster**: paralelismo de tarefas, em memória distribuída
- Podemos combinar ambos? SIM!
    - MPI

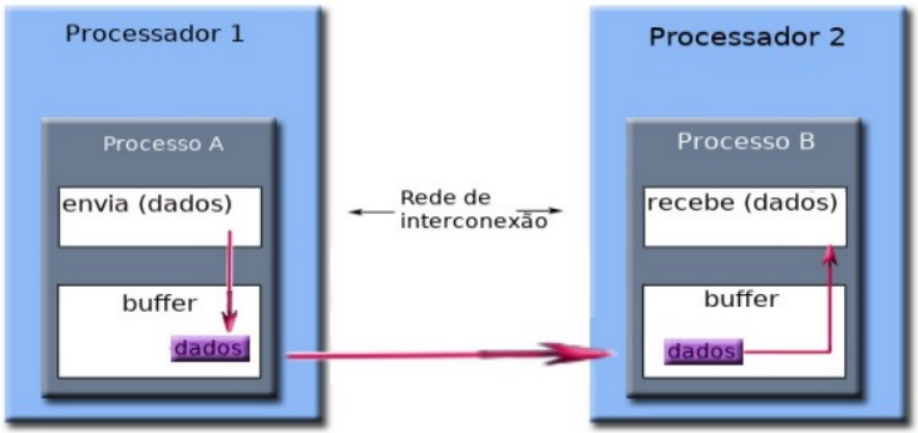
Combinando OpenMP com MPI



MPI : Message Passing Interface

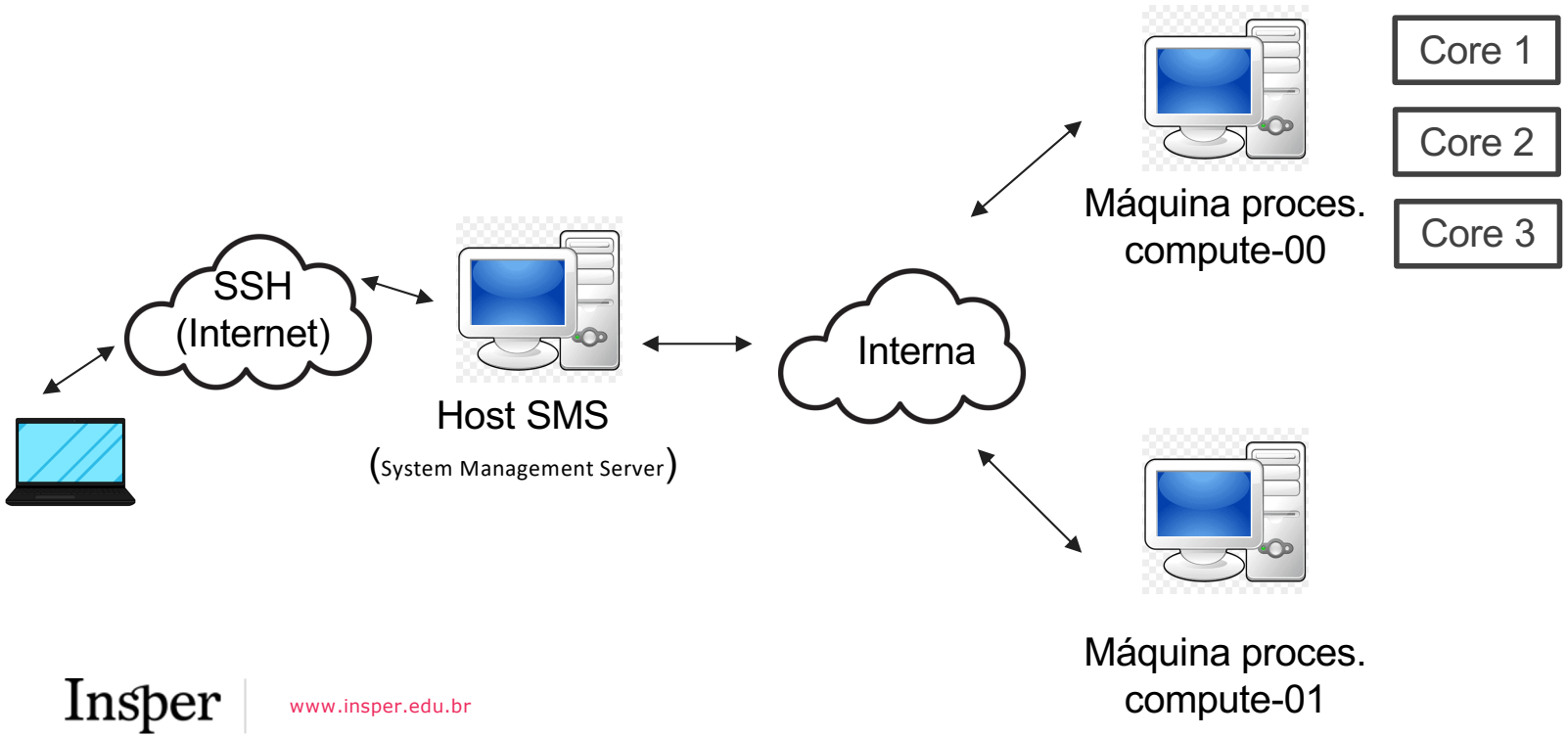
- É um padrão da indústria, não uma linguagem/implementação
- Assume que não há compartilhamento de memória
- O programador deve:
  - Dividir os dados
  - Trabalhar com interações são bilaterais (*send* / *receive*)
  - Reduzir comunicação (quando possível), pra otimizar desempenho

Posso usar MPI localmente?
(em ambiente de memória compartilhada)



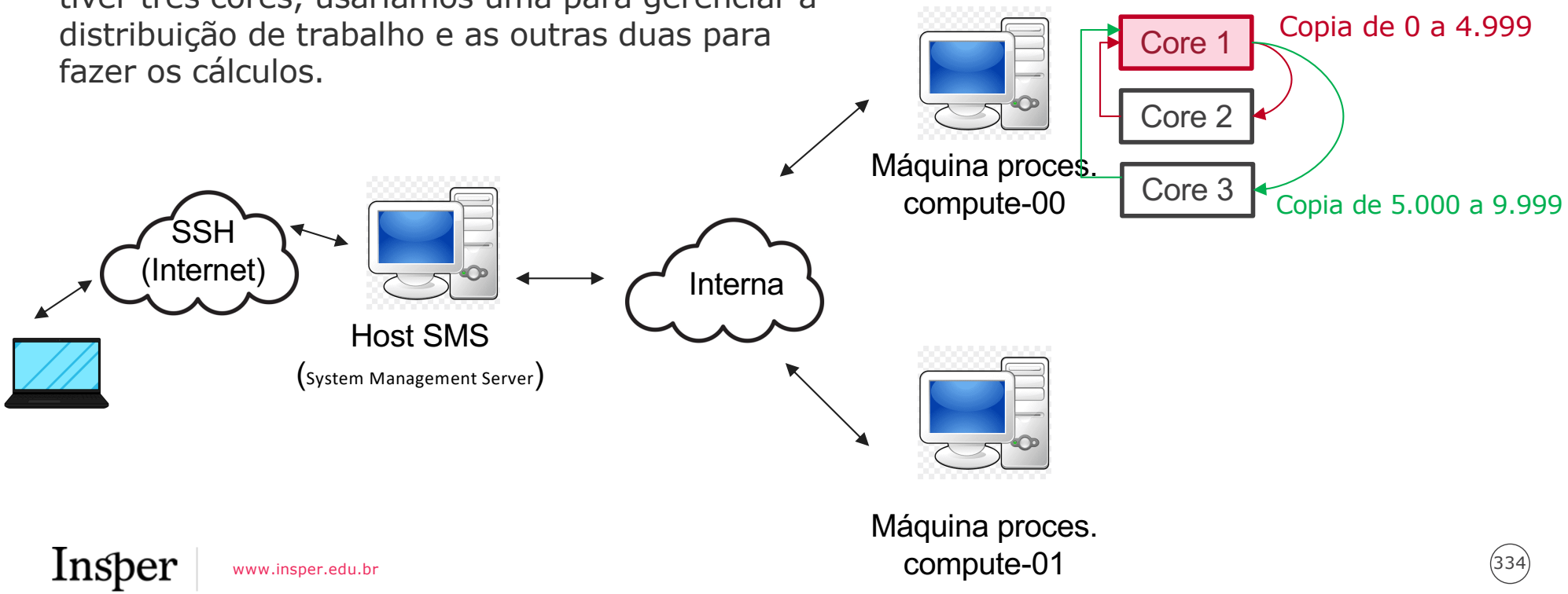
MPI : Exemplo (local)

```
1  int s = 0, v[ ] = {1,2,3,4};
2
3  for (int i=0; i<4; i++)
4      s += v[i];
```



MPI : Exemplo (local)

Se o vetor tivesse 10.000 entradas e a máquina tiver três cores, usaríamos uma para gerenciar a distribuição de trabalho e as outras duas para fazer os cálculos.

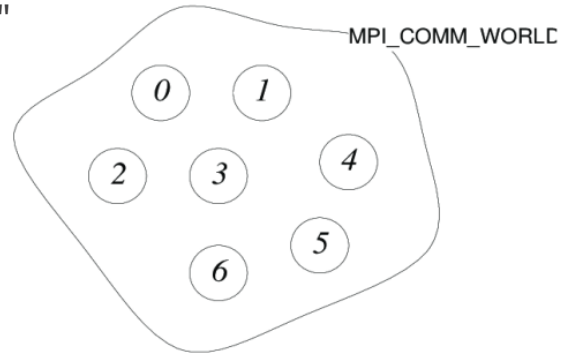


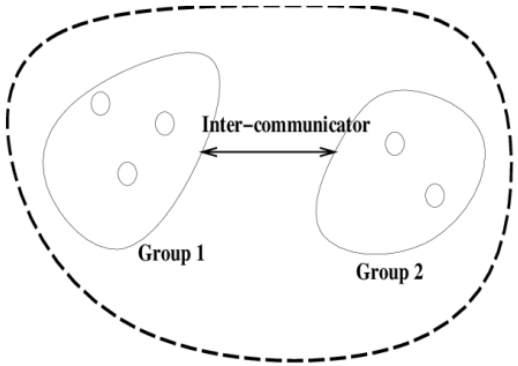
MPI : Conceitos

- **Rank**: Todo processo tem uma única identificação, atribuída pelo sistema quando o processo é iniciado. Essa identificação é continua e começa no 0 até n-1 processos.

- **Group**: Grupo é um conjunto ordenado de N processos. Todo e qualquer grupo é associado a um "communicator" e, inicialmente, todos os processos são menbros de um grupo com um "communicator" já pré-estabelecido (**MPI_COMM_WORLD**).

- **Communicator**: O "communicator" define uma coleção de processos (grupo), que poderão se comunicar entre si (contexto). O MPI utiliza essa combinação de grupo e contexto para garantir uma comunicação segura e evitar problemas no envio de mensagens entre os processos.





MacDonald et al. (2020) Fagg et al. (1997)

MPI : Código | Execução

Include

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

Compilação

```
mpic++ programa.cpp -o programa
```

Execução (básica)

```
mpirun -np <num processos> ./programa <argumentos do programa>
```

Na execução:

- ***-np*** especifica quantidade de processos a serem criados; cada um executa uma cópia do executável (SPMD)
- Há limites para ***–np***: quantidade de slots disponíveis na arquitetura onde o executável será executado. ***Slots*** representam quantidade de processadores físicos disponíveis.
- ***--use-hwthread-cpus*** permite usar também os processadores lógicos
- ***--oversubscribe*** permite mais de um processo por ***slot***
- ***--hostfile <hostfilename>***,especifica nós que podem ser usados para atribuir os processos gerados (mapeamento de processos em processadores).
  - O parâmetro ***hostfilename*** é um arquivo texto com endereços ou nomes dos nós (hosts/máquinas). Um hostname por linha.
  - Os nós podem ser especificados sem o ***–hostfile*** (usar ***–host/-host/-H***)

MPI : Tipos de dados

Tipo do MPI	Tipo do C
MPI_CHAR	char
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG_INT	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_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

MPI : Funções básicas

- `int MPI_Init(int *argc, char ***argv)`
- `int MPI_Finalize()`
- `int MPI_Comm_size(MPI_Comm comm, int *size)`
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
- `int MPI_Send(void bufferE, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `int MPI_Recv(void bufferR, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status status).`

MPI : Exemplo

Mensagem=dado(3 parâmetros de informações) + envelope(3 parâmetros de informações)

Ex.: CALL MPI_SEND(**sndbuf, count, datatype,** **dest, tag, comm,** mpierr)

DADO **ENVELOPE**

CALL MPI_RECV(**recvbuf, count, datatype,** **source, tag, comm,** status, mpierr)

DADO **ENVELOPE**

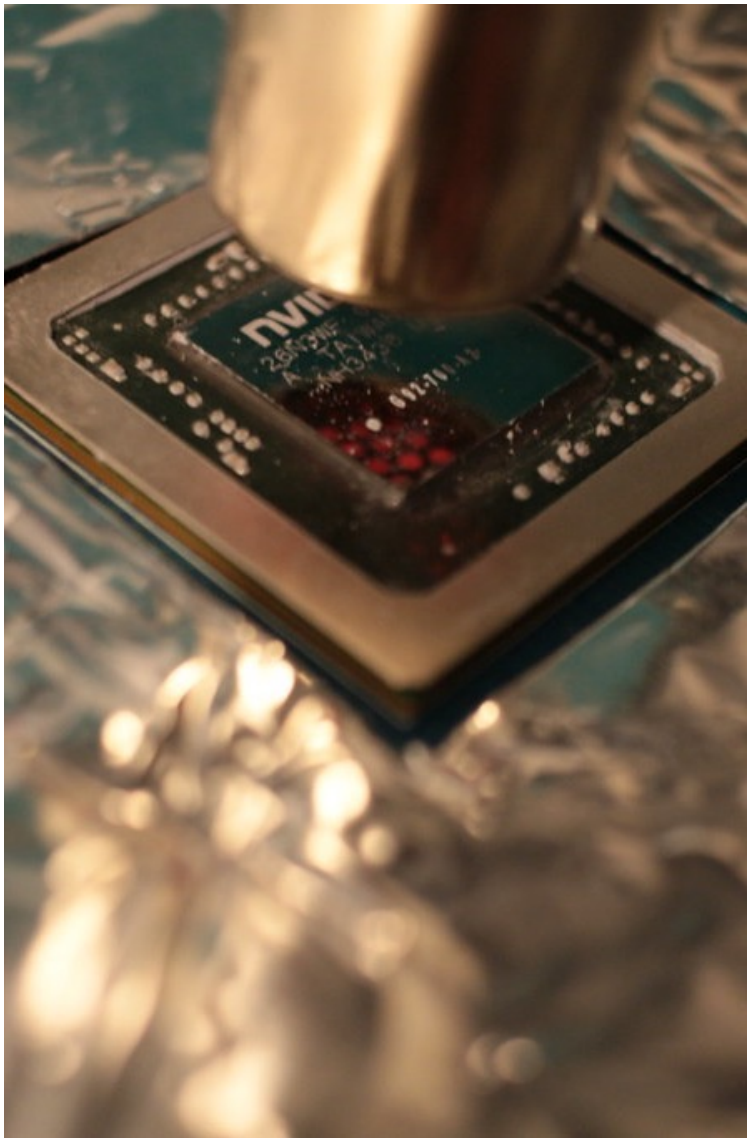
- mpcc hello.c -o hello
- mpirun -np 2 ./hello

C

```
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv )
{
  char message[20];
  int i,rank, size, type=99;
  MPI_Status status;

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

  if(rank == 0) {
   strcpy(message, "Hello, world");
   for (i=1; i<size; i++)
   MPI Send(message, 13, MPI CHAR, i,
            type, MPI_COMM_WORLD);
         }
  else
   MPI_Recv(message, 13, MPI_CHAR, 0,
            type, MPI_COMM_WORLD, &status);

   printf( "Message from node =%d : %.13s\n",
          rank,message);
   MPI_Finalize();
}
```

Insper

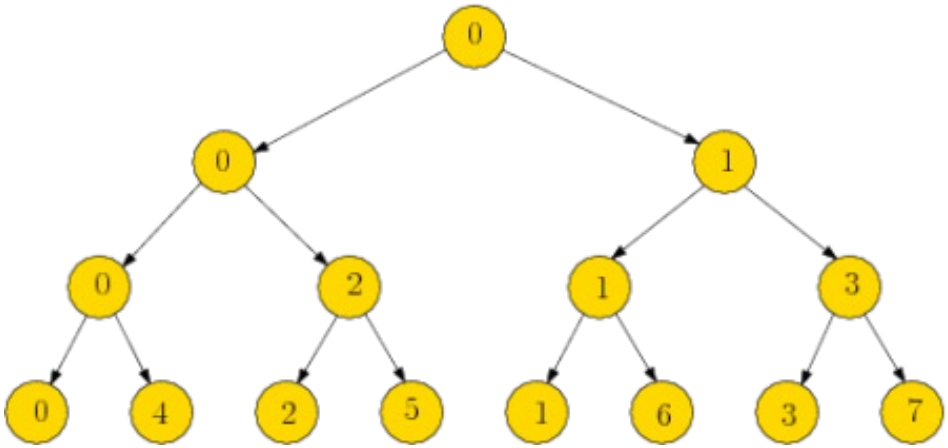
Supercomputação



MPI – Comunicação

- Modos de Comunicação Ponto-a-Ponto

Broadcast e Reduce



- Mandar uma mensagem para todos os processos: *MPI_Bcast*
- Receber mensagens dos nós filhos: *MPI_Reduce*

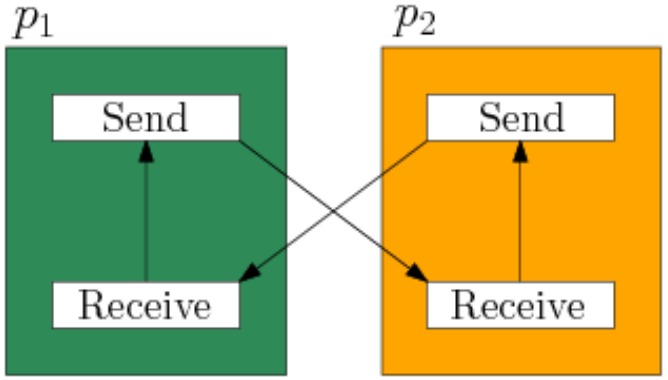
Formas de comunicação

- Bloqueante vs não bloqueante
- Aspectos de hardware influenciam fortemente:
  - Há buffers associados ao send ou ao receive?
  - Há um hardware específico para realizar a comunicação em paralelo à CPU?

Formas de comunicação

- A forma mais comum de comunicação entre dois processos no MPI é através de Sends e Receives, implementados pelas funções MPI_Send e MPI_Recv, respectivamente.

- Em teoria, são bloqueantes



Formas de comunicação

- A maior parte das implementações do padrão MPI, provê um **buffer** para que o send não seja bloqueante.

- Ao chamar *MPI_Send*, a mensagem é copiada em um buffer e a execução do programa continua. Aí quando o destinatário chama *MPI_Recv*, ele lê a mensagem do buffer.

- Em implementações onde não há o esse buffer, deve-se usar *MPI_Bsend*, onde o buffer deve ser explicitamente alocado.

Modos de comunicação ponto a ponto

- O MPI possui 08 *sends* e 02 *receives* (semânticas distintas)

- Há 04 modos para o *send*:
  - ***Standard***
    - Pode ou não ter buffer interno do MPI. Transferência síncrona ou assíncrona.
    - Implementação decide
  - ***Buffered***
    - O usuário cria previamente um buffer (espaço usuário) e o utiliza para o *send*
  - ***Synchronous***
    - Exige a sincronização entre o *send* e o início da execução do *receive no mínimo*
    - Depende se usa buffer e se é bloqueante ou não bloqueante
  - ***Ready***
    - *Receive* deve ser executado antes do *send*. Programador garante isso

  - Cada um destes modos pode ser bloqueante ou não bloqueante (temos 08 *sends*)

- Há 01 modo para o *receive*:
  - ***Standard***
    - Análogo ao *send standard*, porém, para o recebimento da mensagem

  - O *receive standard* pode ser bloqueante ou não bloqueante (temos 02 *receives*)

Modos de comunicação ponto a ponto

- *Sends* bloqueantes no MPI

- ***Standard***

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

- ***Buffered***

*int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

- ***Synchronous***

*int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

- ***Ready***

*int MPI_Rsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

Modos de comunicação ponto a ponto

- ***Buffered*** ***(um pouco mais de detalhe)***

*int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)*

- O buffer precisa ser criado pelo programador

- ***int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm,***
- ***int *size)***
  - Retorna em ***size** o limite superior necessário para empacotar msg no buffer
  - Buffer pode ser usado por mais de um ***MPI_Bsend***. Neste caso considera o total

- ***int MPI_Buffer_attach(void *buf, int size)***
  - Associa um buffer ao MPI no espaço do usuário para enviar msgs
  - Deve considerar: ***MPI_BSEND_OVERHEAD*** por mensagem que usar o buffer

- ***MPI_Bsend()***
  - Envia o conteúdo do buffer indicado em ***MPI_Bsend()***

- ***int MPI_Buffer_detach(void *buf, int *size)***
  - Desassocia o buffer do MPI e espera o término de msgs que estejam usando o buffer
  - Não desaloca da memória, apenas desassocia do MPI

Modos de comunicação ponto a ponto

- *Sends* não bloqueantes no MPI

- ***Standard***

*int MPI_**I**send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm,* ***MPI_Request *request****);*

- ***Buffered***

*int MPI_**I**bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,* ***MPI_Request *request****)*

- ***Synchronous***

*int MPI_**I**ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,* ***MPI_Request *request****)*

- ***Ready***

*int MPI_**I**rsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,* ***MPI_Request *request****)*

- O parâmetro de saída ****request* é um *handle*** que identifica a msg não bloq em andamento
  - Permite verificar se a comunicação já foi finalizada com o ***MPI_Test*** ou ***MPI_Wait***

Modos de comunicação ponto a ponto

- *Receives* bloqueantes e não bloqueantes no MPI (apenas ***Standard***)

- ***Receive* Bloqueante**

*int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)*

- ***Receive Não Bloqueante***

*int MPI_**I**recv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, **MPI_Request *request**)*

- ***MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)***
  - Pode ser usada com ***status** para obter informações da msg recebida

Modos de comunicação ponto a ponto

- Há primitivas testam e/ou esperam o fim de primitivas não bloqueantes no MPI

- ***int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)***
  - *F*az um teste não bloqueante da primitiva não bloqueante indicada por ****request***
  - O parâmetro ****flag*** tem esse retorno

- ***int MPI_Wait(MPI_Request *request, MPI_Status *status)***
  - *F*az um teste bloqueante da primitiva não bloqueante indicada por ****request***
  - Se a mensagem ainda não estiver segura, o processo chamador de ***MPI_Wait*** bloqueia até que a mensagem esteja segura.

- ***MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)***
  - Pode ser usada com ****status*** para obter informações da msg recebida

Prática

https://encurtador.com.br/vyDO6

1. Submeter o “Hello, World!” de MPI no cluster com slurm

2. Submeter um job de MPI que sobrecarrega a comunicação no cluster, observando o uso da rede no Ganglia

3. Faça você mesmo:
    1. Crie um programa via C++/MPI que calcula a soma de um vetor de 10k números.
    2. Submeta este programa no cluster via slurm:
        1. Faça uma execução local (usando apenas um node)
        2. Faça uma execução global (usando todos os nodes e recursos disponíveis).
        3. Observe e compare os tempos de execução