Message Passing Interface (MPI): Modos de Comunicação Ponto-a-Ponto

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente

CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP

Semânticas das primitivas *send* e *receive*

- Primitivas *send* e *receive* podem ter **diferentes semânticas**.

- Por exemplo, considere que estes processos concorrentes estão executando:

Processo 0	Processo 1
0 ...	*0 ...*
1 dest = 1; qtde = 1;	*1 src = 0; qtdemax = 1;*
2 a0 = 100;	*2 receive($a1, qtdemax, src);*
3 send(&a0, qtde, dest);	*3 printf("%d \n", a1);*
4 a0 = 0;	*4 ...*
5 ...	*5 ...*

- Qual será o valor recebido em *a1* pelo Processo 1?
  - Resposta: Depende da semântica considerada.

  - Um caso usual: bloqueante vs não bloqueante

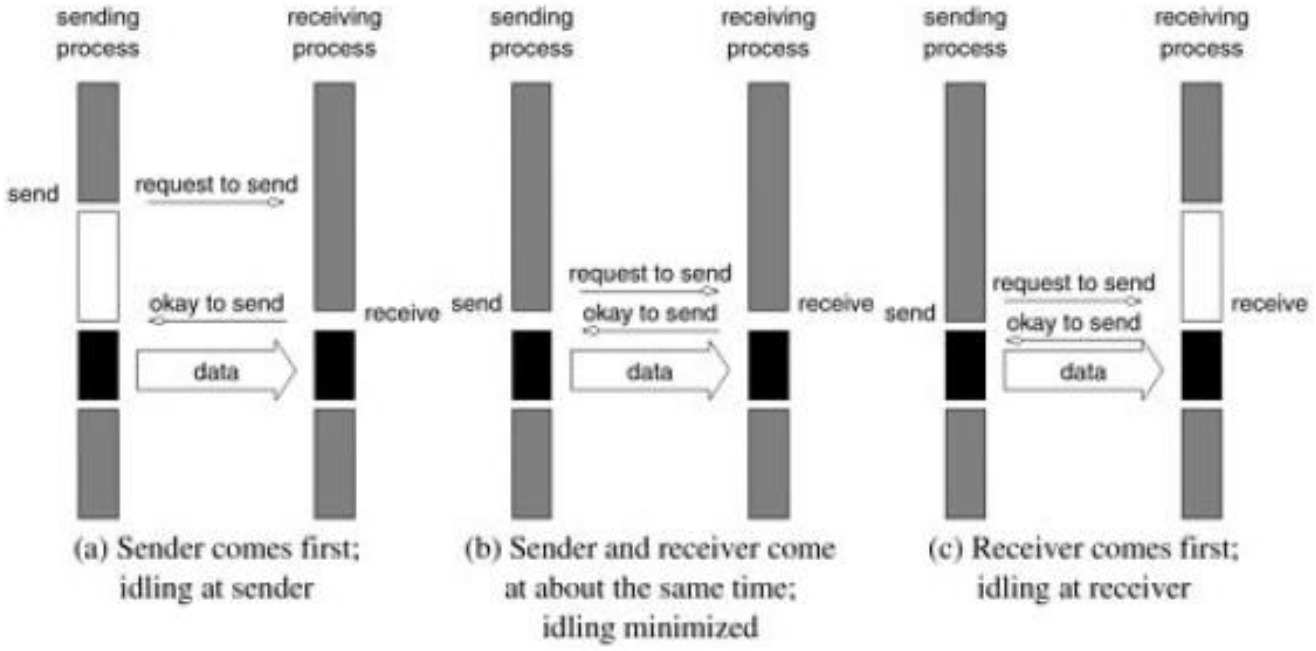
- Aspectos de hardware influenciam fortemente:
  - Há buffers associados ao *send* ou ao *receive*?
  - Há um hardware específico para realizar a comunicação em paralelo à CPU?
    - Como p.ex. DMA ou múltiplos cores

Passagem de Mensagens Bloqueantes

- *Send* **retorna quando a mensagem está segura**
  - Conteúdo da msg não poderá mais ser alterado depois do *send* retornar
  - Msg pode ter sido enviada ao destino **ou só copiada** para um buffer intermediário
    - Caso esse buffer intermediário exista!

- Passagem de mensagens bloqueantes **sem buffers**
  - Neste caso a mensagem precisa ser enviada e recebida pelo processo destino antes do *send* retornar para que a semântica bloqueante do *send* seja satisfeita

  - Há *handshake*
  - Tempo ocioso pode ser grande
  - *Deadlock* caso haja inversão do receive com um outro send

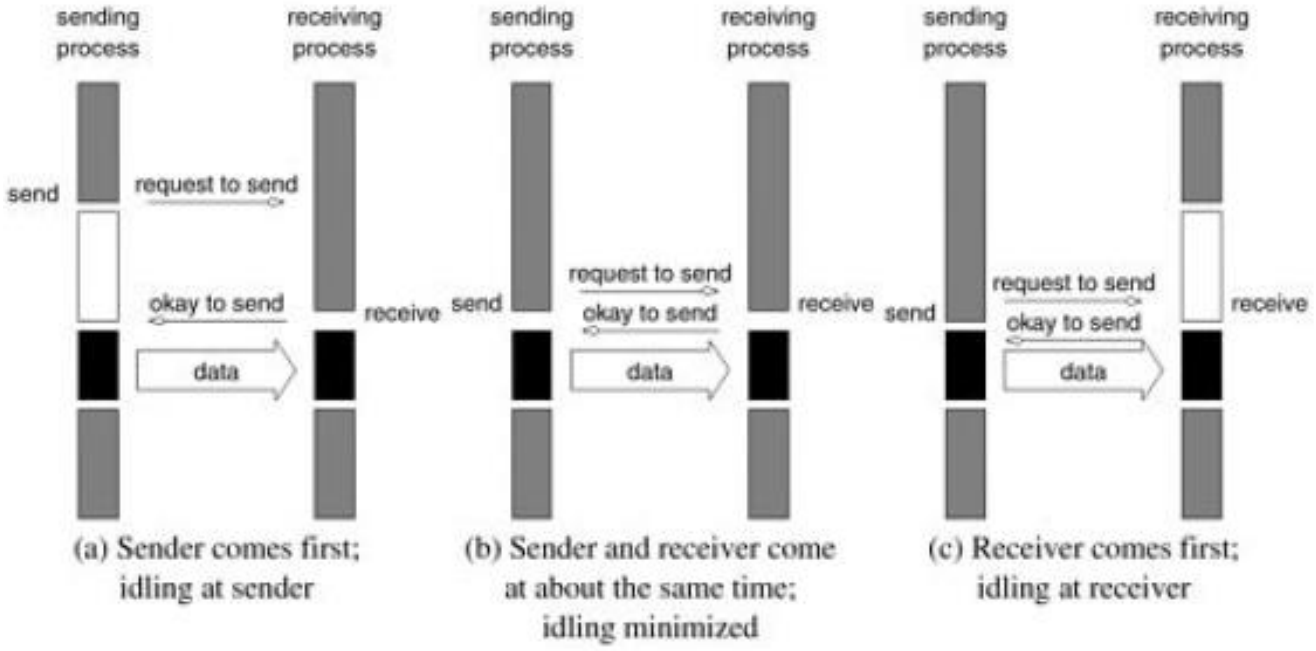


Grama et al. (2003)

Passagem de Mensagens Bloqueantes

- *Exemplificando um possível deadlock quando não há buffers em Pass Msgs Bloqueantes*

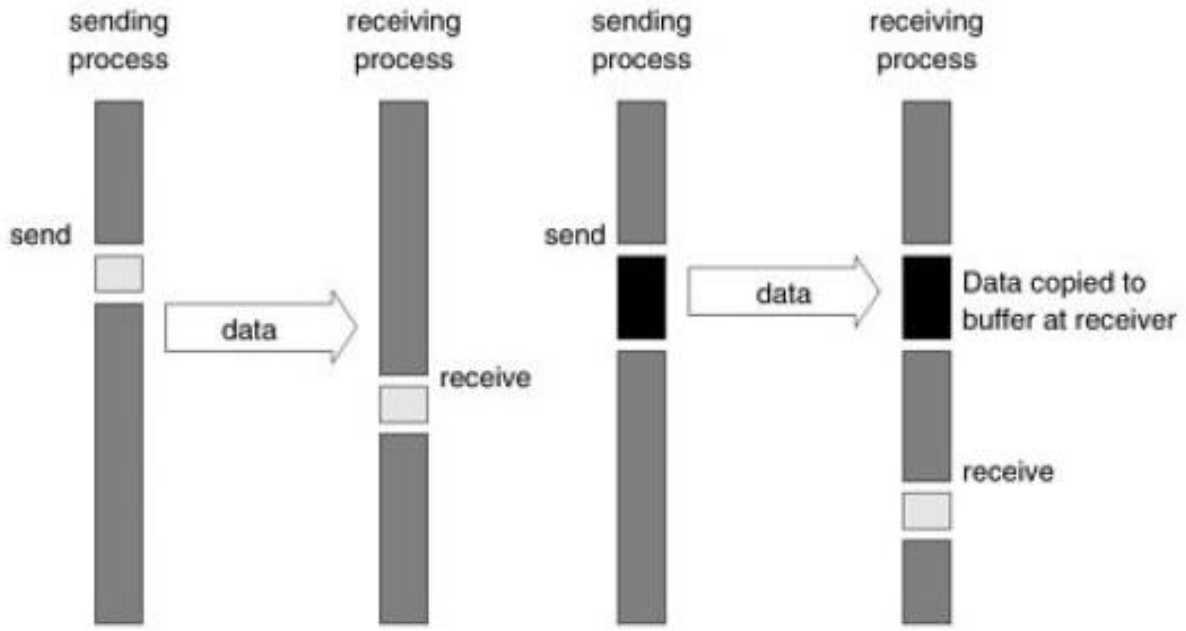
	Processo 0	Processo 1
1	*send(&a, 1, 1);*	*send(&a, 1, 0);*
2	*receive(&b, 1, 1);*	*receive(&b, 1, 0);*



Grama et al. (2003)

Passagem de Mensagens Bloqueantes

- Passagem de mensagens bloqueantes **com buffers (é o caso comum!)**
  - Neste caso a msg estará segura assim que for copiada no buffer
  - O ***send* pode retornar assim que a msg for copiada para o buffer**
    - Não precisa aguardar o *receive* executar
    - Diminui tempo ocioso do processo
    - Aumenta *overhead* para transmissão da msg devido cópia para/do buffer.
      - Overhead varia se há hardware para tratar a comm em paralelo ou não



Grama et al. (2003)

Passagem de Mensagens Bloqueantes

- Passagem de mensagens bloqueantes **com buffers**
  - Deve considerar o problema de buffers com capacidade finita de bytes

Processo 0

```
1 ...
2 for (i = 0; i < 1000; i++)
3 {
4     produce_data(&a);
5     send(&a, 1, 1);
6 }
7 ...
```

Processo 1

```
1 ...
2 for (i = 0; i < 1000; i++)
3 {
4     receive(&a, 1, 0);
5     consume_data(&a);
6 }
7 ...
```

- Há *deadlock* se dois processos executam *receives*, um aguardando uma msg do outro

Processo 0

```
1 receive(&a, 1, 1);
2 send(&b, 1, 1);
```

Processo 1

```
1 receive(&a, 1, 0);
2 send(&b, 1, 0);
```

- ***Receives* bloqueantes** *r*etornam quando recebem a mensagem:
  - diretamente do *send* (quando não há buffers) ou
  - do seu buffer de recebimento (se este existe)

Grama et al. (2003)

Passagem de Mensagens Bloqueantes

- Conceito importante e que deve ficar registrado em todos:
  - **Bloqueante não é sinônimo de síncrono em aplicações paralelas**
  - Um *send* é bloqueante se ele espera o conteúdo da mensagem estar seguro para ser enviado antes de retornar da sua chamada
    Ele não espera, necessariamente, que a mensagem seja enviada
    A mensagem pode ter sido enviada, mas não necessariamente foi (se houver buffer)
  - Um *receive* é bloqueante se ele tem o conteúdo da mensagem seguro com ele
    - Este *receive* não necessariamente sincronizou com o *send* (caso haja buffer)

Grama et al. (2003)

Passagem de Mensagens Não Bloqueantes

- Reduzem *overheads* da passagem de mensagens bloqueantes
  - Possível tempo ocioso quando não há buffer
  - Cópia/gerência do buffer quando eles existem (este é o caso comum)

- Primitivas não bloqueantes retornam antes da mensagem estar segura
  - Para transmissão no caso do *send*
  - Para uso no caso do *receive*

- Não há garantia que a msg tenha sido copiada para um buffer ou transmitida (*send*)
- Não há garantia que a msg tenha sido de fato recebida (*receive*)

- Programador deve verificar o status da transmissão após o retorno da primitiva para poder alterar ou usar seu conteúdo
  - Primitivas adicionais permitem essa verificação (*test, wait, probe, ...*)

- Vantagens das primitivas não bloqueantes:
  - Permitem comunicação em paralelo à uma computação ou outra comunicação
    - Ganha-se tempo com o paralelismo durante as trocas de mensagens
  - Deve haver hardware para tratar a comunicação em paralelo (DMA, *multicore*, ...)
    - Caso não haja, a CPU é multiplexada e o ganho de desempenho é menor

Grama et al. (2003)

Passagem de Mensagens Não Bloqueantes

- Primitivas não bloqueantes podem ou não ter buffers
  - Não há tempo ocioso pois retornam sem esperar a conclusão
    - ou da transmissão ou da cópia para o buffer

- **Um aspecto importante para os *receives* não bloqueantes**
  - **Retorno da chamada do *receive* vs tempo de vida da primitiva**

- A depender da API e do protocolo utilizados, um *receive* não bloqueante funciona diferente

  - Primeiro caso possível (sockets, por exemplo)
    - *receive* não bloqueante finaliza quando retorna com ou sem a msg
    - Um novo *receive* precisa ser executado para verificar se a msg chegou, caso ainda não tenha sido recebida

  - Segunda possibilidade (MPI, por exemplo)
    - *Receive* não bloqueante não finaliza quando retorna sem a msg
      - Fica executando em *background* esperando pela msg chegar
    - Uma outra primitiva deve testar se a msg já chegou
      - Alguns exemplos de tais primitivas são *test* e *wait*
    - Quando a msg chegar, o *receive* não bloqueante encerra

Passagem de Mensagens Síncronas

- Comunicações síncronas garantem que *send* e *receive* sincronizarão
  - Sincronizam em tempo de execução e no nível da aplicação
  - Pode ser quando a transmissão é iniciada ou quando é finalizada
  - Aguardam *handshake* para continuar

- Comunicações **bloqueantes sem buffers** (ambos os lados) são síncronas

- Comunicações assíncronas (este é o caso comum)
  - Não garantem que *send* e *receive* sincronizaram no nível da aplicação
  - Quando o *send* foi executado, o *receive* não necessariamente foi executado ao mesmo tempo
  - Quando o *receive* foi executado o *send* pode já ter sido finalizado.
    - Não houve a sincronização no nível da aplicação

- **Mais uma vez: bloqueante não é sinônimo de síncrono em aplicações paralelas!**

Modos de comunicação ponto-a-ponto no MPI

- 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*)

Opções de comunicação ponto-a-ponto no MPI

- *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)*

Opções de comunicação ponto-a-ponto no MPI

- ***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

Opções de comunicação ponto-a-ponto no MPI

- *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***

Opções de comunicação ponto-a-ponto no MPI

- *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

Verificação de comunicação não bloqueante no MPI

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

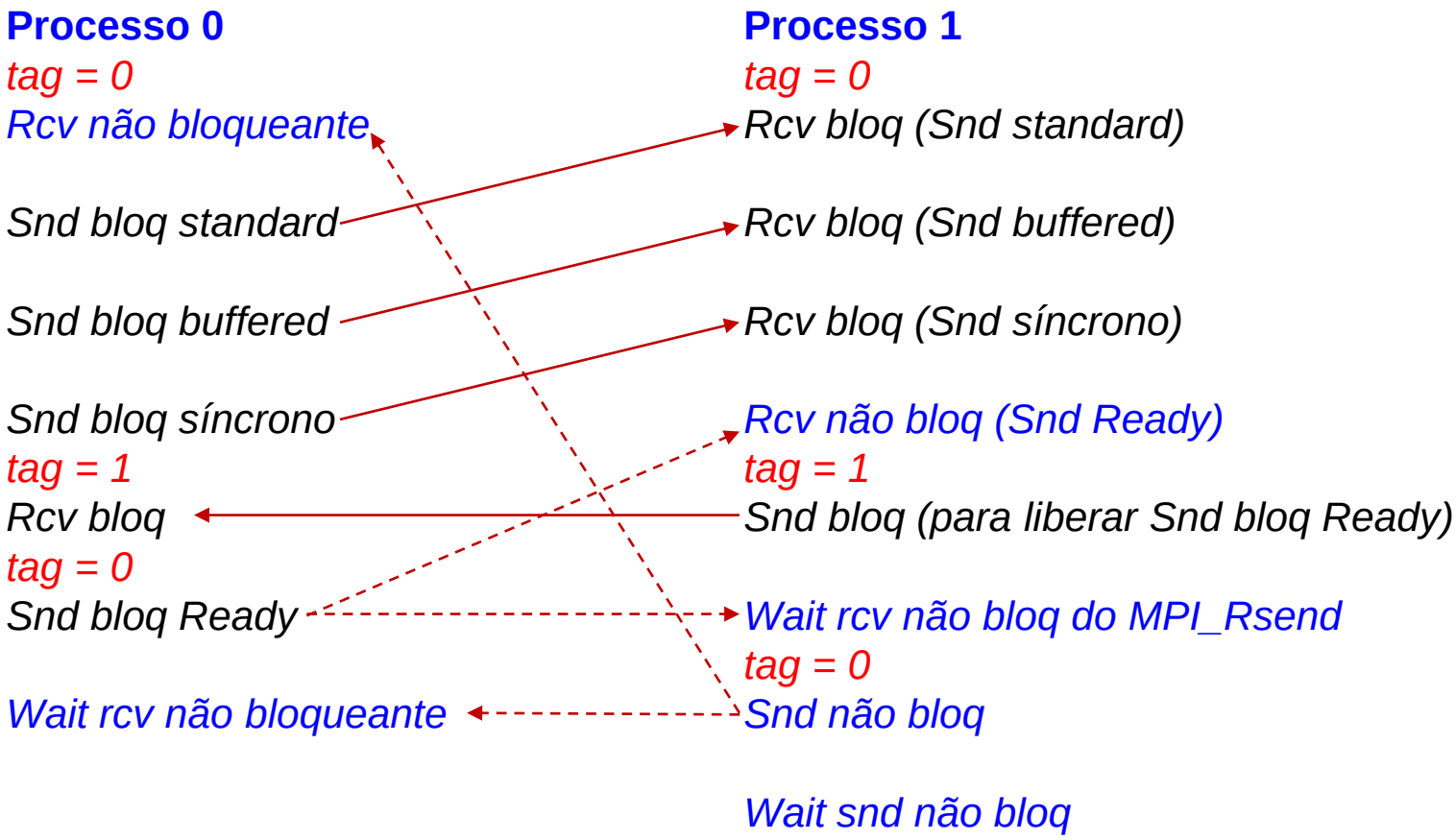
- ***int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)***
  - *F*az um teste não bloqueante da primitiva não bloqueante indicada por ****request***
  - O parâmetro ****flag*** tem esse retorno

- ***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

Exemplo de Código MPI

- Código C/MPI usa dois processos para exemplificar diferentes modos de comunicação



Referências

Rauber, T., & Rünger, G. (2013). *Parallel Programming*. Springer. Second edition. Capítulo 5.

Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier. Capítulo 3.

Barlas, G. (2014). *Multicore and GPU Programming: An integrated approach*. Elsevier. Capítulo 5.

Grama, A., Kumar, V., Gupta, A., & Karypis, G. (2003). Introduction to parallel computing. Pearson Education. Capítulo 6.

Apostila de Treinamento: Introdução ao MPI (Unicamp). https://www.cenapad.unicamp.br/servicos/treinamentos/apostilas/apostila_MPI.pdf

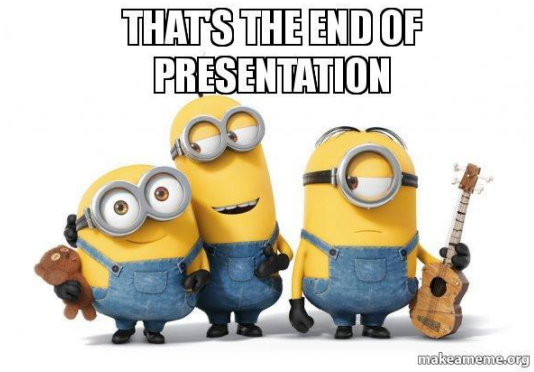
MacDonald, N; Minty, E.; Malard, J.; Harding, T.; Brown, S.; Antonioletti, M. Writing Message Passing Parallel Programs with MPI. 2020. Disponível em: https://www.researchgate.net/publication/239179288_Writing_Message_Passing_Parallel_Programs_with_MPI (último acesso em 27/10/2020)

Fagg, Graham; Dongarra, Jack; Geist, Al. Heterogeneous MPI Application Interoperation and Process Management under PVMPI. 91-98. 1997. Disponível em: https://www.researchgate.net/figure/Inter-communicator-formed-inside-a-single-MPI-COMM-WORLD_fig1_221597084 (último acesso em 27/10/2020)

Message Passing Interface (MPI): Modos de Comunicação Ponto-a-Ponto

Paulo Sérgio Lopes de Souza
pssouza@icmc.usp.br

Universidade de São Paulo / ICMC / SSC – São Carlos
Laboratório de Sistemas Distribuídos e Programação Concorrente



CNPq Conselho Nacional de Desenvolvimento Científico e Tecnológico
CAPES
FAPESP