Aula 02 - Comunicação Ponto-a-ponto e criação de novos grupos de processos

Conceitos de primitivas send e receive bloqueantes & não bloqueantes
presença ou não de buffers
Grama et al.

Primitivas send e receive podem ter diferentes semânticas. Por exemplo

P0	P1
A0=100;	receive($a1, 1, 0);
send(&a0, 1, 1);	printf("%d\n", a1);
a0=0;	

Qual será o valor recebido por P1? Isso depende da semântica considerada.

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

Passagem de Mensagem Ponto-a-Ponto Bloqueante

Send só **retorna quando a mensagem está segura** e não poderá mais ser alterada pelo usuário
A msg pode ter sido enviada ao destino ou copiada para um buffer intermediário

Sem buffers: Se não há a presença de buffers intermediários, então a mensagem precisa ser enviada e recebida do outro lado antes do send retornar. Há um handshake entre transmissor e receptor.

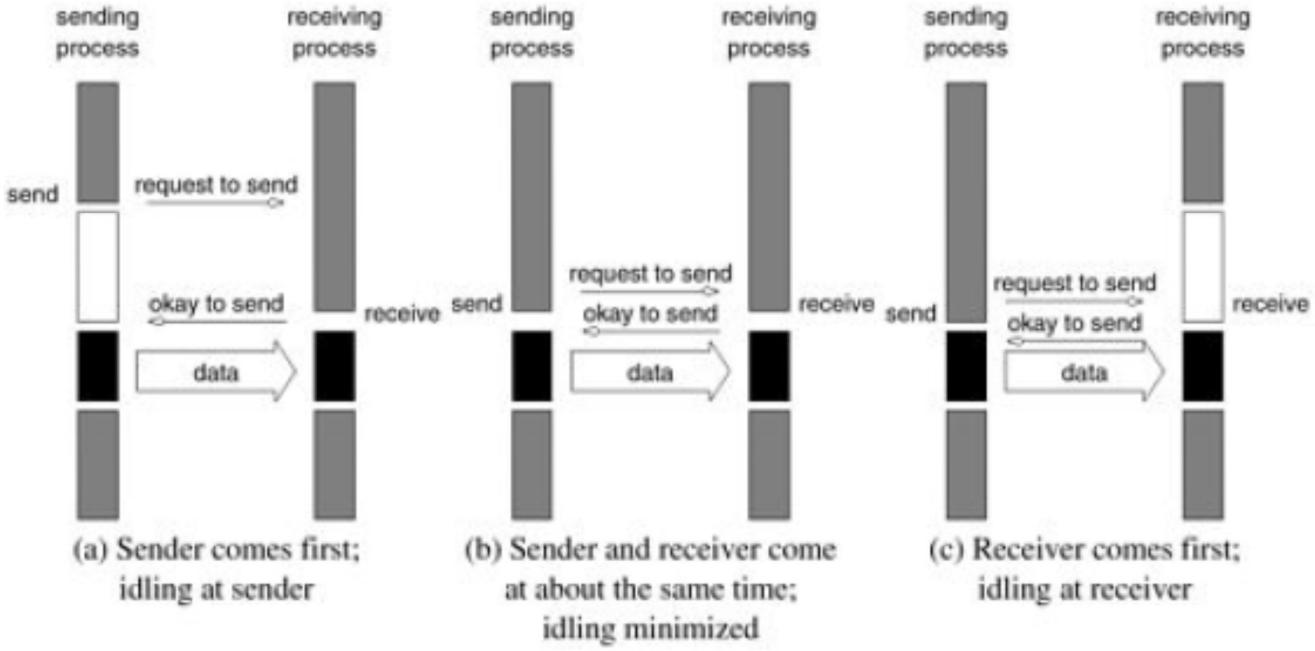

Figura 6.1 seção 6.2.1 Grama

Tempo ocioso pode ser grande, em relação à computação feita pelo processo.
Deadlock quando dois processos usam estes sends e aguardam o recebimento um do outro. Ficarão bloqueados para sempre.

```
1       P0                  P1
2
3       send(&a, 1, 1);             send(&a, 1, 0);
4       receive(&b, 1, 1);          receive(&b, 1, 0);
```

Com buffers: Se há presença de buffers então 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.

Comportamento diferente quando há hardware para tratar a comunicação em paralelo ou não.

É o caso comum.

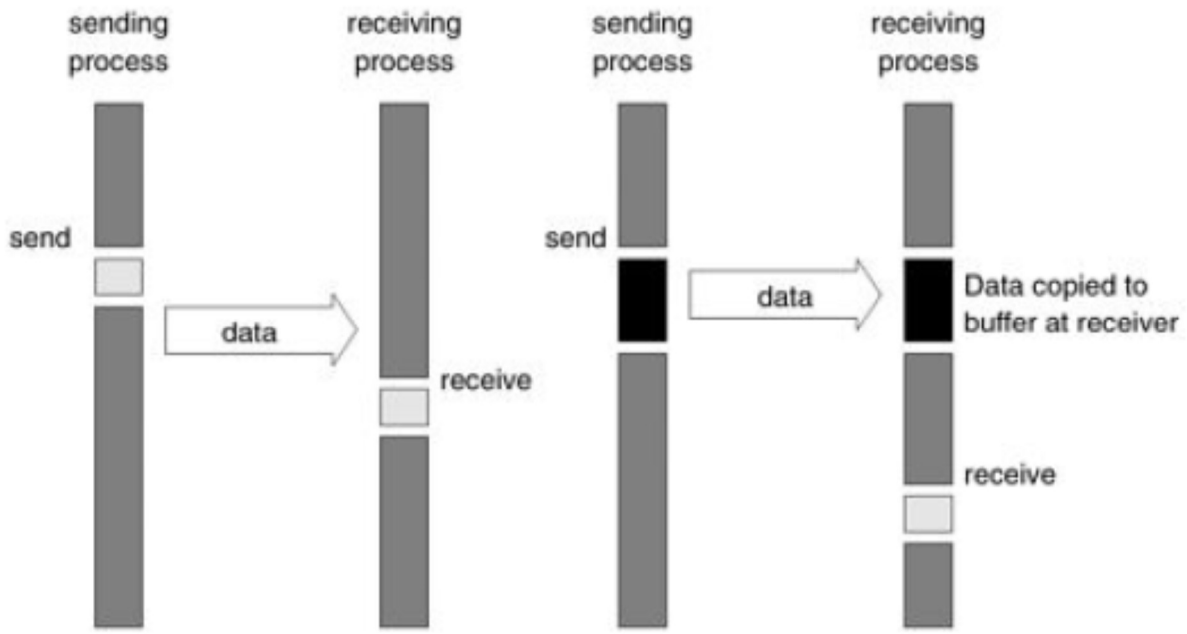

Figura 6.2 seção 6.2.1 Grama

Problema de buffers com capacidade finita de bytes.

```
1       P0                              P1
2
3       for (i = 0; i < 1000; i++) {    for (i = 0; i < 1000; i++) {
4               produce_data(&a);               receive(&a, 1, 0);
5               send(&a, 1, 1);                 consume_data(&a);
6       }                               }
```

Deadlock ocorre quando dois processos executam receives um aguardando uma mensagem do outro. Nunca sairão daí.

```
1           P0                              P1
2
3           receive(&a, 1, 1);              receive(&a, 1, 0);
4           send(&b, 1, 1);                 send(&b, 1, 0);
```

Receives bloqueantes só retornam quando recebem a mensagem, diretamente do send (quando não há buffers) ou do seu buffer de recebimento.

Passagem de Mensagens Ponto-a-Ponto Não Bloqueante

A semântica bloqueante tem overhead na forma de tempo ocioso (quando não há buffer) ou na cópia/gerência dos buffers.
Comunicações não bloqueantes reduzir estes overheads, retornando das primitivas send ou receive antes que a mensagem esteja segura para a transmissão ou tenha sido transmitida de fato. Em outras palavras, não há garantia que a mensagem tenha sido copiada para o buffer ou transmitida.
Desvantagem: o programador é o responsável por não alterar o conteúdo da mensagem antes que ela esteja segura.

Primitivas adicionais verificam o status da operação de comunicação, para indicar se o conteúdo pode já ser alterado ou não, no lado do transmissor. No lado do receptor tais primitivas de verificação indicam se o conteúdo a ser recebido já pode ser utilizado ou não.

Quando as primitivas não bloqueantes retornam, os processos podem fazer outra computação concorrentemente à comunicação potencialmente em andamento. Se há hardware específico para tratar a E/S referente à comunicação então toda a transmissão é feita sem interromper novamente a CPU que executa outra computação. São não houver tal hardware, a CPU tratará da computação em background.

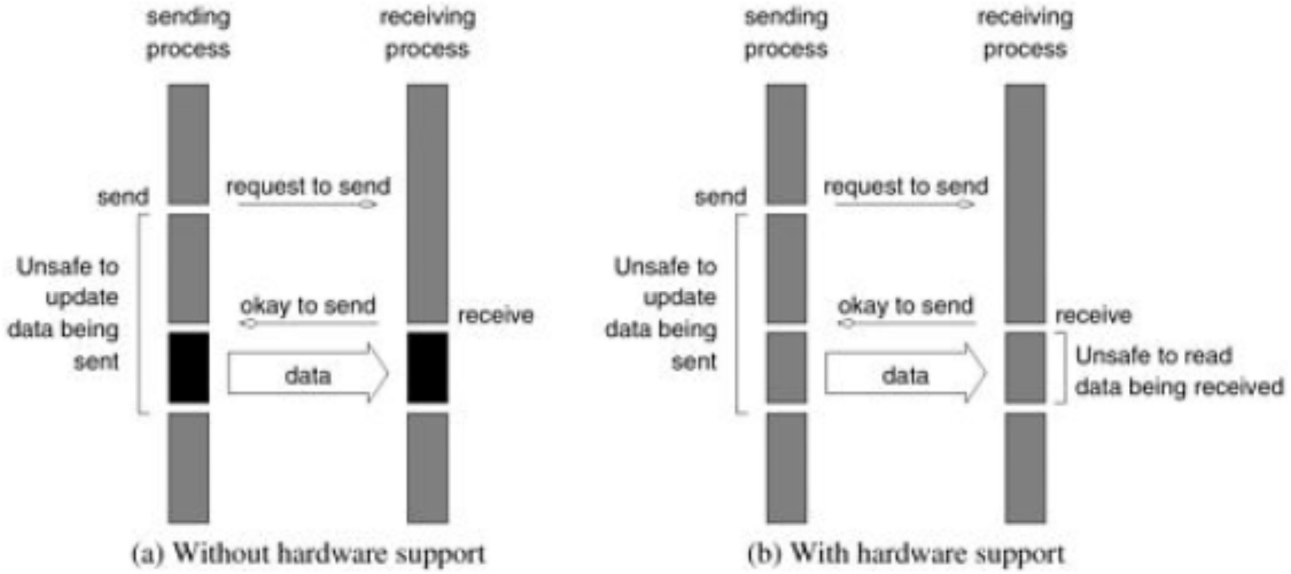

Figura 6.4 seção 6.2.2 Grama

Primitivas não bloqueantes podem ou não ter buffers, assim como ocorre com as primitivas bloqueantes. No caso de não haver buffers, o tempo ocioso não ocorre, pois neste momento as primitivas não-bloqueantes já retornaram e permitem que o processo continue a ser executado normalmente. Cabe ao programador verificar o fim da comunicação e estabelecer caminhos alternativos de computação enquanto a comunicação não termina.

As diferentes semânticas aplicadas para operações bloqueantes e não bloqueantes, com e sem buffers podem ser resumidas na Figura 6.3 de Grama seção 6.2.2.

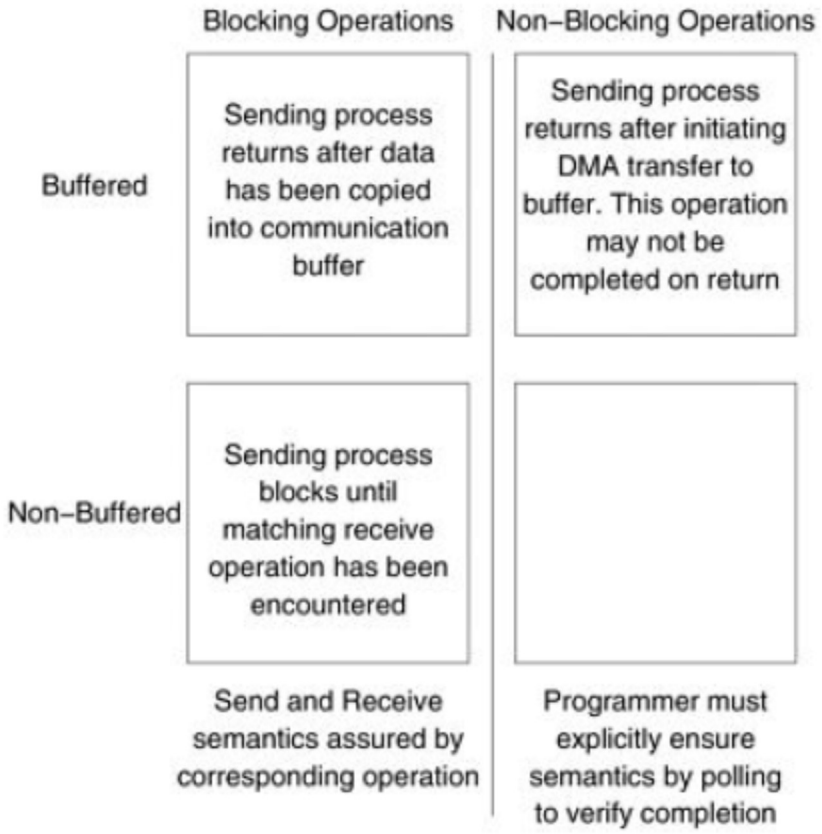

Figura Grama 6.3

Passagem de Mensagens Síncronas e Assíncronas

As comunicações síncronas são aquelas que garantem que as primitivas send e receive irão sincronizar em tempo de execução, no nível da aplicação. Em um determinado momento, quer seja quando a transmissão é iniciada ou quando a transmissão é finalizada as primitivas send e receive aguardam que haja um handshake para poderem continuar.

Mensagens bloqueantes sem buffers, por exemplo, são síncronas.

As comunicações assíncronas são aquelas que não garantem que as primitivas send e receive sincronizaram em tempo de execução, no nível da aplicação. Quando a operação send foi realizada (iniciada e finalizada) a operação receive não necessariamente foi executada. Por outro lado, quando a operação receive foi realizada (iniciada e finalizada) a operação send pode já ter sido finalizada e, dessa forma, não houve a sincronização.

É importante observar que os termos bloqueantes e síncronas não são sinônimos neste contexto.

Modos de comunicação ponto-a-ponto no MPI

send: standard, buffered, synchronous, ready

receive: standard

Comunicação ponto-a-ponto bloqueante vs não bloqueante

MPI_Test & MPI_Wait

Há 4 modos para o Send (seção 2.8 apostila MPI)

Standard: Ou a mensagem é copiada para um buffer interno do MPI e transferida assincronamente para o processo destino, ou os processos transmissor e receptor sincronizam para a troca da mensagem.

A implementação é livre para escolher o que fazer, considerando caso-a-caso (a implementação não necessita ter a mesma semântica sempre).

Buffered: exige o uso de buffer na transmissão. Programador é o responsável por gerenciar o buffer que deve ter o tamanho suficiente. Se for insuficiente o resultado é indefinido pelo padrão. Funções MPI_BUFFER_ATTACH e MPI_BUFFER_DETACH permitem a gerência dos buffers pelo programador.

Synchronous: Send síncrono exige que sincronização entre o send e o início da execução do receive para ser finalizado. Não precisa esperar, necessariamente, a conclusão da transmissão. Isso vai depender se ela é bloqueante ou não e seu usa buffers ou não.

Ready:exige que o receive correspondente ao send tenha sido executado antes e esteja apto a receber a mensagem. Se isso não ocorrer, o resultado é indefinido pelo padrão. É responsabilidade do programador garantir tal ordem de execução. Pode reduzir a sobrecarga de comunicação usando-o. Não bloqueante não espera pelo fim da transmissão e retorna assim que a msg atinge o receptor.

Há 1 modo para o Receive
Standard

Para cada uma destas há uma versão bloqueante e outra não bloqueante
Há 8 sends e 2 receives.

Passagem de mensagem ponto-a-ponto

Podem ser ponto-a-ponto bloqueante ou não não Bloqueante

Considerando o Send: primitivas send bloqueantes retornam assim que a msg esteja segura (copiada para buffer ou transmitida).
Send síncrono bloqueante só retorna após a transmissão ser finalizada.

Primitivas send bloqueantes:
Standard bloqueante
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

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

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

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

Send não bloqueantes retornam antes que a mensagem esteja segura e permite a sobreposição da comunicação com outra comunicação ou com computação.
Send síncrono não bloqueante pode retornar assim que a transmissão começa.

Standard não bloqueante
int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request);

Buffered não bloqueante
int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

Synchronous não bloqueante
int MPI_Issend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

Ready não bloqueante
int MPI_Irsend(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 usado para identificar a mensagem não bloqueante em andamento. Com ele é possível verificar se a comunicação já foi finalizada.

Considerando os receives: bloqueantes retornam apenas quando recebem a mensagem e ela está segura para ser usada pelo programa.

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

Receives não bloqueantes retornam mesmo que a mensagem não esteja disponível. Cabe ao programador gerenciar esta situação. Após o retorno, no caso do não recebimento da mensagem, o receive não finaliza; ele fica executando em background permitindo que a comunicação ocorra concorrentemente à outra comunicação ou à computação.

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

Associadas às comunicações não bloqueantes, send ou receive, há funções adicionais para testar e/ou esperar por uma mensagem não bloqueante. Dois exemplos das principais instruções são:

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

A primeira delas faz um teste não bloqueante do send/receive não bloqueante em andamento. A segunda aguarda até que a comunicação seja finalizada (i.e., é bloqueante). A verificação ocorre sobre a comunicação apontada por *request. O resultado da verificação é encontrado no valor de retorno da função. O parâmetro *status contém mais informações sobre a comunicação realizada, as quais podem ser obtidas com a função MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count). Esta mesma função pode ser usada após um receive bloqueante.

Criação de novos grupos e novos comunicadores

Message Passing Fundamentals pp 78-102

MPI_Comm_group()
Determina o handle de grupo de um comunicador
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_Group_incl()
Cria um novo grupo a partir de um já existente e determina novos membros por inclusão.
int MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)

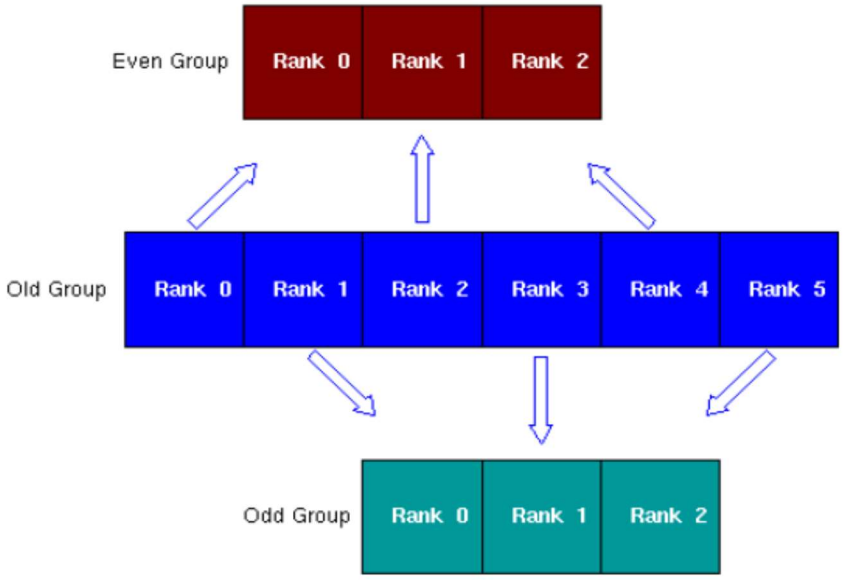

Figure 7.1. Graphical representation of MPI_GROUP_INCL example

MPI_Group_excl()
Cria um novo grupo a partir de um já existente e determina novos membros por exclusão
int MPI_Group_excl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup)

MPI_Group_rank()
Retorna o nr do rank processo no grupo
int MPI_Group_rank(MPI_Group group, int *rank)

MPI_Group_free()
Libera um grupo
int MPI_Group_free(MPI_Group *group)

MPI_Comm_Create()
Cria um novo comunicador para um grupo identificado por grupo
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

MPI_Comm_split ()
Forma novos comunicadores a partir de um já existente.
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

Códigos vistos em sala de aula e solicitados como exercícios:

Execução em sala:
02a-mode-comm/modo-com,
02b-group_comm/group_comm,
02b-group_comm/split_example

Exercícios solicitados:
Não foram solicitados exercícios extra classe

Bibliografia:
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