CSC367 Parallel computing

Lecture 14: Distributed Memory Architectures and their Parallel Programming Model

Thanks to Scinet, NSF XSEDE, James Demmel, and Kathy Yelick

Message passing – logical & physical view

- Logical view: Each process is a separate entity with its own memory
  - Each process runs a separate instance of the same (parallel) program
  - A process cannot access another process's memory except via explicit messages
  - Message passing implies a distributed address space
- Physical view: the underlying architecture could be either distributed memory or shared memory
  - Message passing on a physical shared memory architecture: messages can be simulated by copying (or mapping) data between different processes' memory space
- Take away:
  - You can use message passing on a multicore/shared memory machine, however, the message passing interface simulates a “distributed environment” on the shared memory machine.
  - Consider a core is a processor and just see your shared memory machine as a distributed memory machine.

Message passing compared to OpenMP/Pthreads

- Idea: let the processes compute independently and coordinate only rarely to exchange information
  - Communication adds overheads, so it should be done rarely
- Need to structure the program differently, to incorporate the message passing operations for exchanging data
- Need to partition data such that we maximize locality and minimize transfers
  - Recall the concepts discussed a few lectures ago

Building Blocks

- Interactions are carried out via passing messages between processes
- Building blocks: send and receive primitives – general form:
  - `send(void *sendbuf, int nelems, int dest)`
  - `receive(void *recvbuf, int nelems, int source)`
- Complexity lies in how the operations are carried out internally
- Example (pseudocode):

```
P0                          P1
---                         ---
msg = 5;                    recv(&msg, 1, 0);
send(&msg, 1, 1);           printf("%d", msg);
msg = 200;
```

- Key question: What will P1 receive?
  - Send operation may be implemented to return before the receipt is confirmed
  - Supporting this kind of send is not a bad idea

Blocking operations

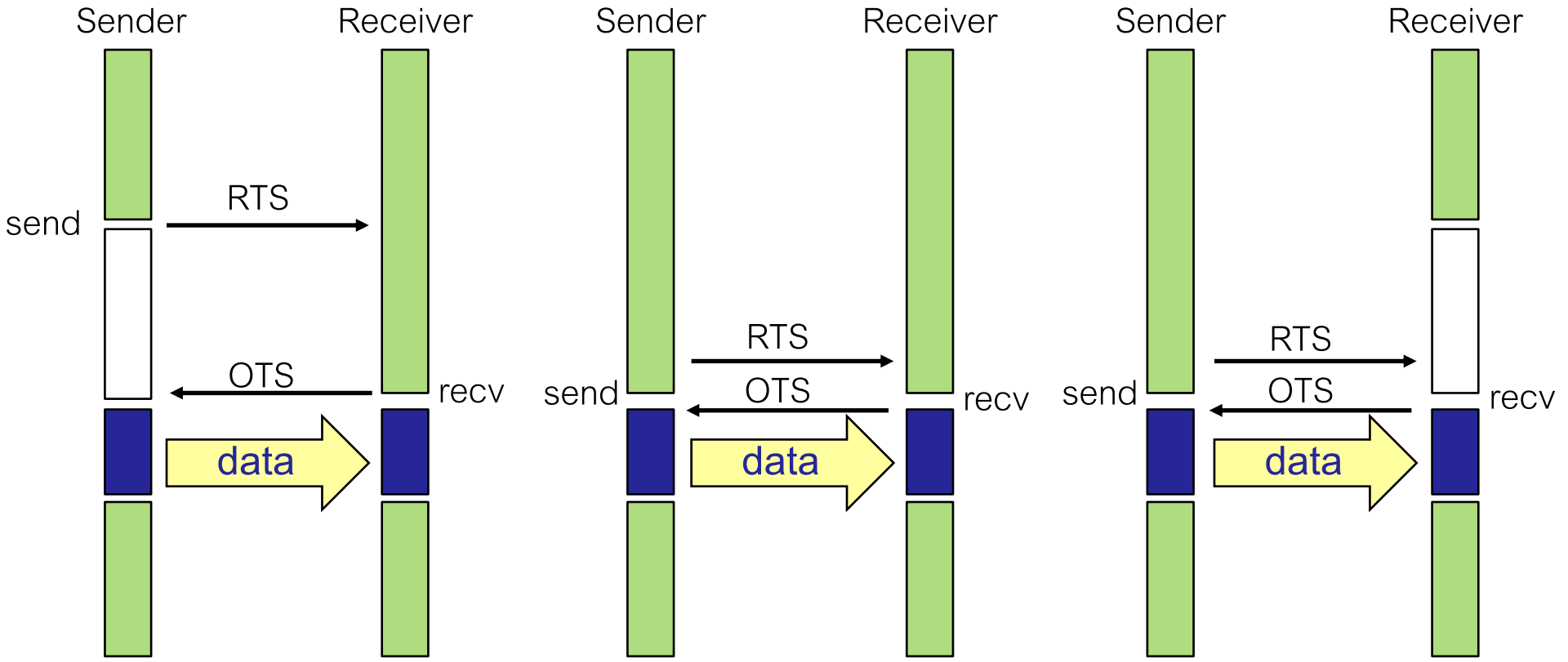
- Only return from an operation once it's safe to do so
  - Not necessarily when the msg has been received, just guarantee semantics
- Two possibilities:
  - Blocking non-buffered send/receive
  - Blocking buffered send/receive

Blocking non-buffered send/recv

- Send operation does not return until matching receive is encountered at the receiver and communication operation is completed
- Non-buffered handshake protocol – idling overheads:

RTS = request to send

OTS = ok to send



1. Sender is first; idling at sender	2. Same time; idling minimized	3.Receiver is first; idling at recv

Deadlocks in blocking non-buffered comm

- Deadlocks can occur with certain orderings of operations, due to blocking
- Example – this deadlocks:

```
P0                          P1
---                         ---
send(&m1, 1, 1);            send(&m1, 1, 0);
recv(&m2, 1, 1);            recv(&m2, 1, 0);
```

- Solution: switch order in one of the processes
  - But, more difficult to write code this way, and could create bugs

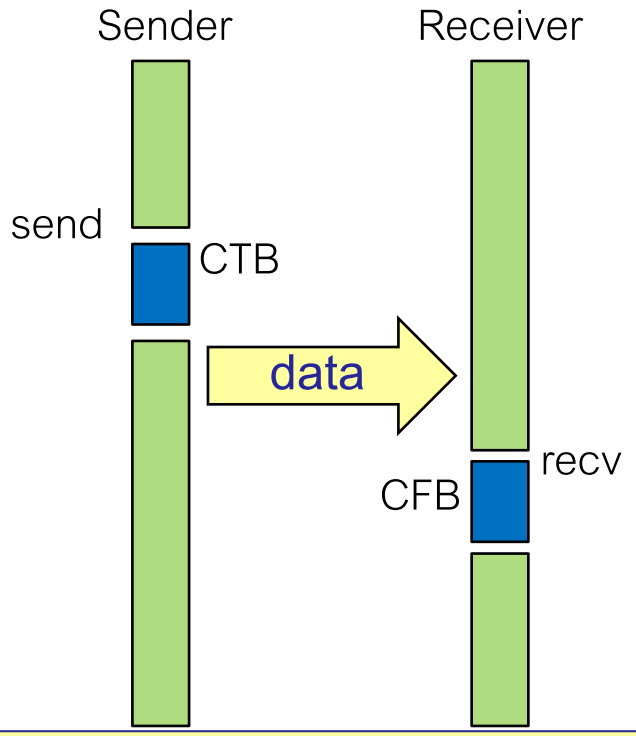
Hardware support for send/receives

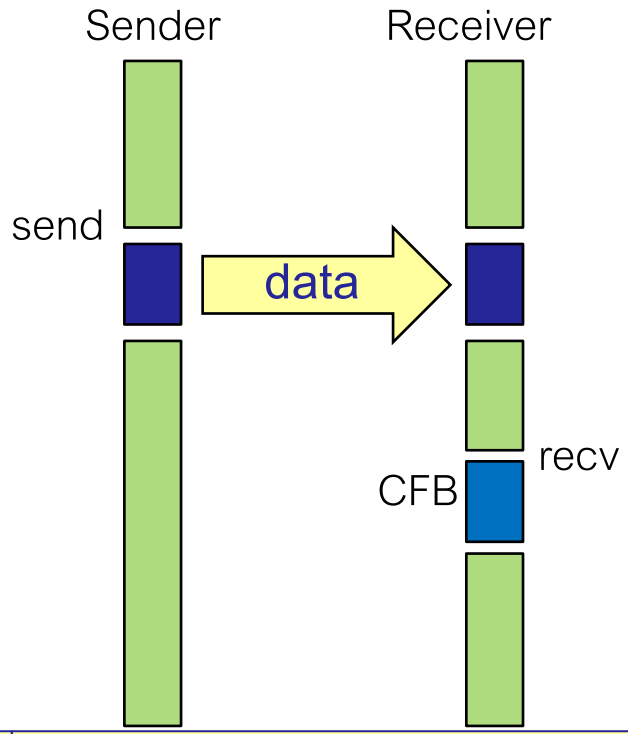
- Most message passing platforms have additional hardware support for sending and receiving messages.

- They may support DMA (direct memory access) and asynchronous message transfer using network interface hardware.
  - **Network interfaces** allow the transfer of messages from buffer memory to desired location without CPU intervention.
  - Similarly, **DMA** allows copying of data from one memory location to another (e.g., communication buffers) without CPU support

Blocking buffered send/recv

- Sender copies data into buffer and returns once the copy to buffer is completed
- Receiver must also store the data into a buffer until it reaches the matching recv
- Buffered transfer protocol – with or without hardware support:





CTB = Copy to buffer

CFB = Copy from buf

In both cases, no idling overheads! But, now buffer management overheads!

1. Use buffer at both sender and receiver
Communication handled by H/W (network interface)

2. Buffer only on one side. E.g., sender interrupts receiver and deposits the data in a buffer (or vice-versa)

Problems with blocking buffered comm

- 1. Potential problems with finite buffers
  - Example:

```
P0 (producer)                          P1 (consumer)
---                                    ---
for(i = 0; i < 1000000; i++){          for(i = 0; i < 1000000; i++){
   create_message(&m);                    recv(&m, 1, 0);
   send(&m, 1, 1);                        digest_message(&m);
}                                      }
```

- 2. Deadlocks still possible
  - Example:

```
P0                                     P1
---                                    ---
recv(&m1, 1, 1);                       recv(&m1, 1, 0);
send(&m2, 1, 1);                       send(&m2, 1, 0);
```

- Solution is similar: break circular waits
- Unlike previously, in this protocol, deadlocks can only be caused by waits on recv

Non-blocking operations

- Why non-blocking? Performance!
- User is responsible to ensure that data is not changed until it's safe
  - Typically a check-status operation indicates if correctness could be violated by a previous transfer which is still in flight
- Can also be buffered or non-buffered
  - Can be implemented with or without hardware support

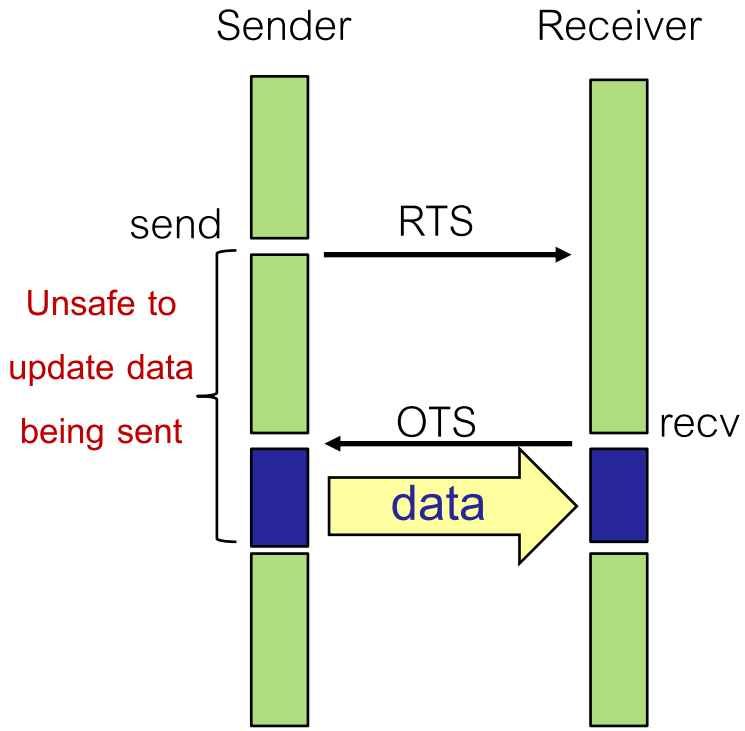
Example: non-blocking non-buffered

- Sender issues a request to send and returns immediately
- When the receive is encountered, communication is initiated

RTS = request to send

OTS = ok to send

Without hardware support



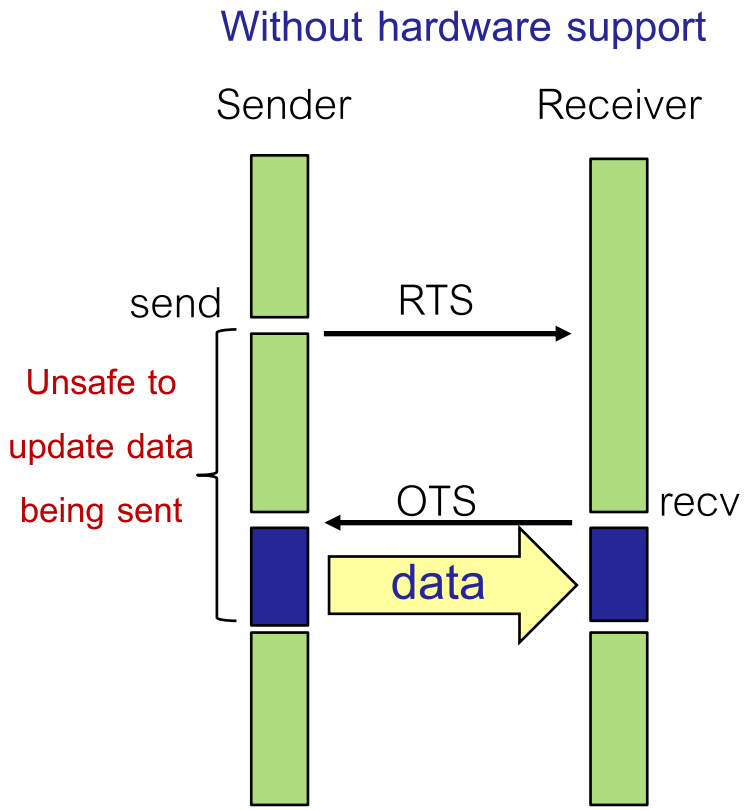
1. When recv is encountered, transfer is handled by interrupting the sender

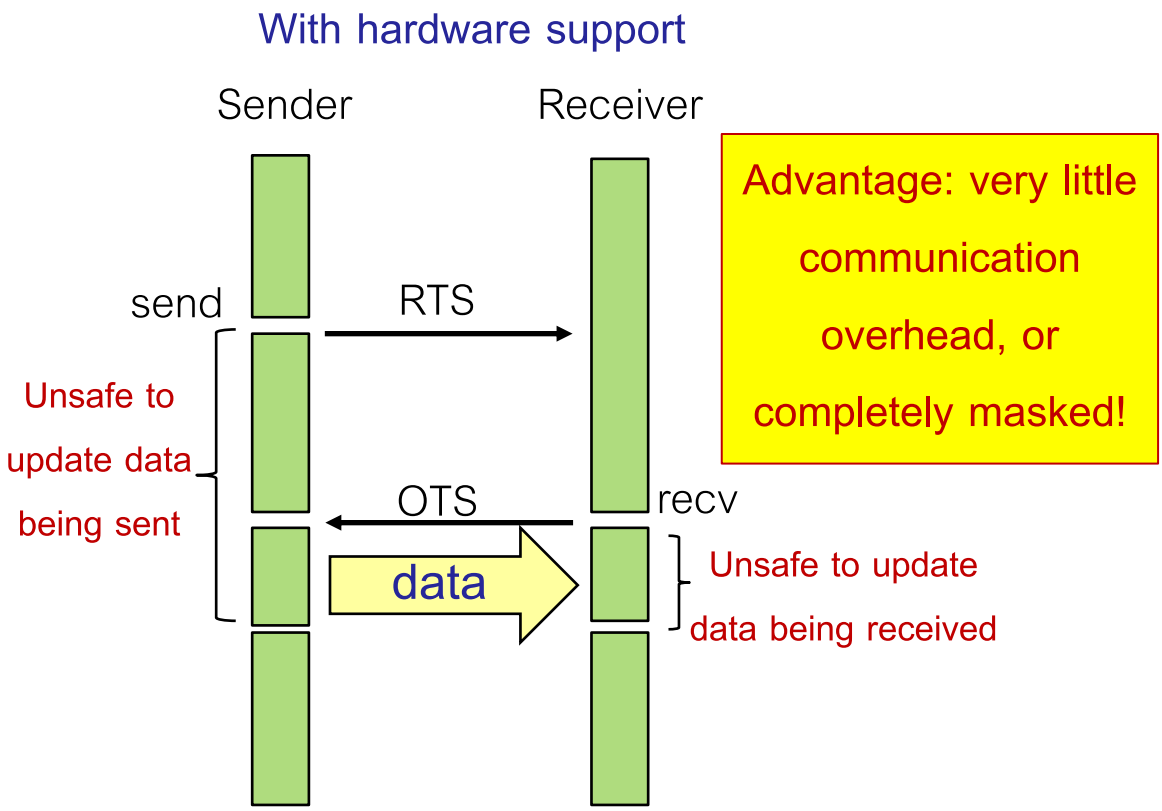
Example: non-blocking non-buffered

- Sender issues a request to send and returns immediately
- When the receive is encountered, communication is initiated

RTS = request to send

OTS = ok to send





Advantage: very little communication overhead, or completely masked!

1. When recv is encountered, transfer is handled by interrupting the sender

2. When recv is found, comm. hardware handles the transfer and receiver can continue doing other work

Summery

	Buffered	Non-Buffered	
Blocking Operations	Sending process returns after data has been copied into communication buffer.	Sending process blocks until matching receive operation has been encountered.	send and recv semantics ensured by corresponding operation
Non-blocking Operations	Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return.		Programmer must explicitly ensure semantics by polling to verify completion

Take-aways

- Carefully consider the implementation guarantees
  - Communication protocol and hardware support
  - Blocking vs. non-blocking, buffered vs. non-buffered
- Tradeoffs in terms of correctness and performance
  - Need automatic correctness guarantees => might not hide communication overhead that well
  - Need performance => user is responsible for correctness via polling

The Message Passing Interface (MPI)

MPI standard

- **Standard** library for message passing
  - Write portable message passing algorithms, mostly using C or Fortran
  - Rich API (over 100 routines, but only a handful are fundamental)
  - Must install OpenMPI or MPICH2, etc.
  - Include mpi.h header
- Example run command: `mpirun -np 8 ./myapp arg1 arg2`
- Basic routines:

MPI_Init: initialize MPI environment

MPI_Finalize: terminate the MPI environment

MPI_Comm_size: get number of processes

MPI_Comm_rank: get the process ID of the caller

MPI_Send: send message

MPI_Recv: receive message

MPI basics

- MPI_Init: only called once at start by one thread, to initialize the MPI environment
  - **`int MPI_init(int *argc, char ***argv);`**
  - Extracts and removes the MPI parts of the command line (e.g., mpirun –np 8) from argv
  - Process your application's command line arguments only after the MPI_Init
  - On success => MPI_SUCCESS, otherwise error code

- MPI_Finalize: called at the end, to do cleanup and terminate the MPI environment
  - **`int MPI_Finalize();`**
  - On success => MPI_SUCCESS, otherwise error code
  - No MPI calls allowed after this, not even a new MPI_init!

- These calls are made by all participating processes, otherwise results in undefined behaviour

MPI Communication domains

- MPI communication domain = set of processes which are allowed to communicate with each other
- Communicators (`MPI_Comm` variables) store info about communication domains
- Common case: all processes need to communicate to all other processes
  - Default communicator: `MPI_COMM_WORLD` includes all processes
- In special cases, we may want to perform tasks in separate (or overlapping) groups of processes => define custom communicators
  - No messages for a given group will be received by processes in other groups
- Communicator size and id of current process can be retrieved with:
  - `int MPI_Comm_size(MPI_Comm comm, int *size);`
  - `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - The process calling these routines must be in the communicator `comm`

Hello (C)

```c
#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( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Timing measurements

- Can use MPI_Wtime()
- Example:

```
double t1, t2;
t1 = MPI_Wtime();
...
t2 = MPI_Wtime();
printf("Elapsed time: %f\n", t2 - t1);
```

MPI data types

- Equivalent to built-in C types, except for MPI_BYTE and MPI_PACKED

MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	N/A
MPI_PACKED	N/A

Flavors of communication in MPI

- Collective operations: All processes in the communicator or group have to participate!
  - Barrier, Broadcast, Reduction, Prefix sum, Scatter / Gather, All-to-all, etc.

- Point-to-point operations: A processor explicitly communicants with another processor with send and receive messages

Point-to-point communication

Blocking sends and receives

Non-blocking sends and receives

Sending and receiving messages

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

- Each message has a tag associated to distinguish it from other messages
- The source and tag can be **MPI_ANY_SOURCE/MPI_ANY_TAG**
- The status can be used to get info about the **MPI_recv** operation:

```
typedef struct MPI_Status {
   int MPI_SOURCE; // source of the received message
   int MPI_TAG;    // tag of the received message
   int MPI_ERROR;  // a potential error code
};
```

- The length of the received message can be retrieved using:

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

Implementations of MPI_Recv and MPI_Send

- **MPI_Recv** is blocking
  - It returns only after message is received and copied into the buffer!
  - Buffer can be safely reused right after MPI_Recv
- **MPI_Send** can be implemented with two options:
  - Option 1: returns only after the matching MPI_Recv is executed and the message was sent
  - Option 2: copy msg into buf and returns, without waiting for MPI_Recv
  - In both, the buffer can be safely reused right after MPI_Send

Deadlock avoidance

- Restrictions on MPI_Send/MPI_Recv, in order to avoid deadlocks
- Example: behaviour is MPI_Send implementation-dependent

```
int a[20], b[20], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
  MPI_Send(a, 20, MPI_INT, 1, 1, MPI_COMM_WORLD);
  MPI_Send(b, 20, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
  MPI_Recv(b, 20, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
  MPI_Recv(a, 20, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
}
...
```

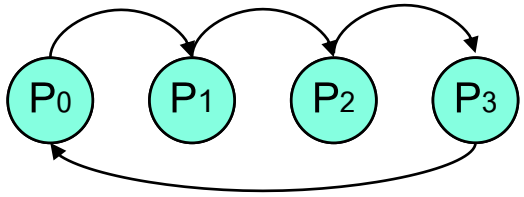
- If MPI-send is option 1 then deadlock, option 2 leads to no deadlock!
- Fix by matching the order of the send and recv operations
- We want to write "safe" programs which are not implementation dependent!

Deadlock avoidance

- Another example: Circular chain of send/recv operation:

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
MPI_Recv(b,20, MPI_INT, (myrank-1+np)%np, 1, MPI_COMM_WORLD,&status);
```

- Works fine if send proceeds after copying to data

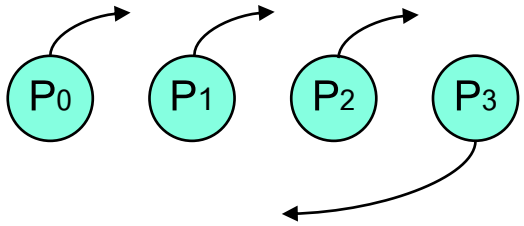


Deadlock avoidance

- Another example: Circular chain of send/recv operation:

```
int a[20], b[20], myrank, np;
MPI_Status status;
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a,20, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);
MPI_Recv(b,20, MPI_INT, (myrank-1+np)%np, 1, MPI_COMM_WORLD,&status);
```

- Works fine if send proceeds after copying to data, but deadlocks if send has to wait for the receive.



Sends are waiting for receive: deadlock!

- Must rewrite the code to make it safe: