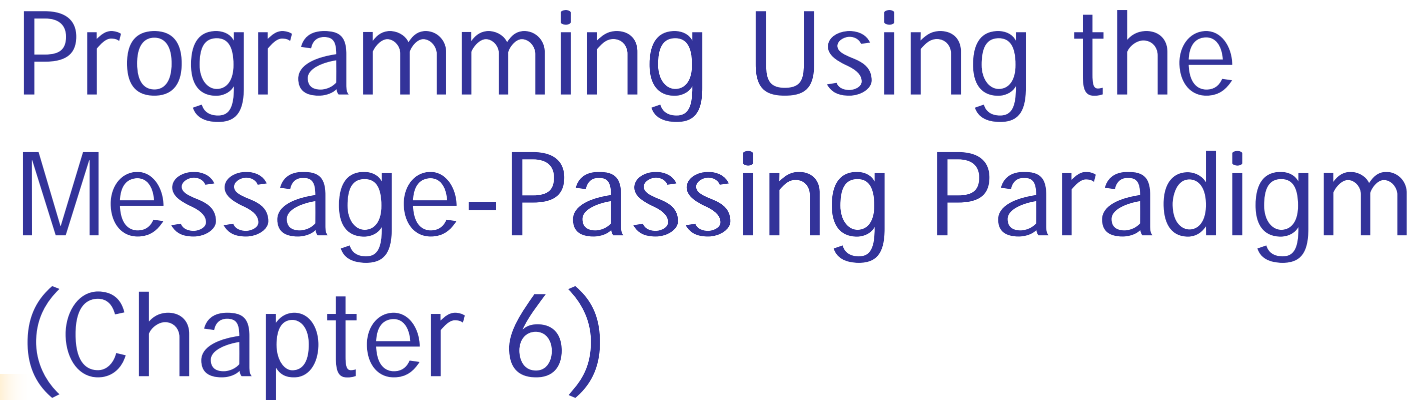


Programming Using the Message-Passing Paradigm (Chapter 6)

Alexandre David

B2-206

Topic Overview

- Principles of Message-Passing Programming
- MPI: the Message Passing Interface
- Topologies and Embedding
- Overlapping Communication with Computation
- Collective Communication and Computation Operations
- Groups and Communicators

Why MPI?

- One of the oldest libraries (supercomputing 1992).
- Wide-spread adoption, portable.
- Minimal requirements on hardware.
- Explicit parallelization.
  - Intellectually demanding.
  - High performance.
  - Scales to large number of processors.

MPI: The Message Passing Interface

- Standard library to develop portable message-passing programs using either C or Fortran.
- The API defines the syntax and the semantics of a core set of library routines.
  - Vendor implementations of MPI are available on almost all commercial parallel computers.
- It is possible to write fully-functional message-passing programs by using only the six routines.

MPI Features

- Communicator information.
- Point to point communication.
- Collective communication.
- Topology support.
- Error handling.

```
send(const void *sendbuf, int nelem, int dest)
receive(void recvbuf, int nelem, int src)
```

Six Golden MPI Functions

- Total 125 functions.
- 6 most used function.

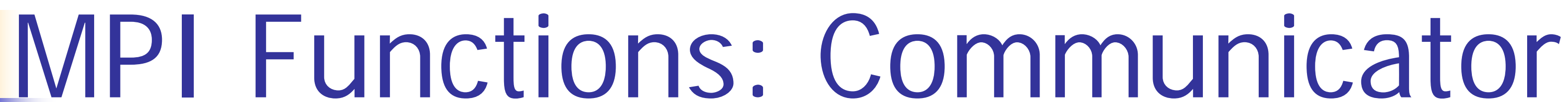
MPI_Init	Initializes MPI.
MPI_Finalize	Terminates MPI.
MPI_Comm_size	Determines the number of processes.
MPI_Comm_rank	Determines the label of the calling process.
MPI_Send	Sends a message.
MPI_Recv	Receives a message.

MPI Functions: Initialization

- Must be called once by all processes.
- MPI_SUCCESS (if successful).
- #include <mpi.h>

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

MPI Functions: Communicator



- Concept of communication domain.
- MPI_COMM_WORLD default for all processes involved.
- If there is a single process per processor, *MPI_Comm_size(MPI_COMM_WORLD, &size)* returns the number of processors.

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

Hello World!

```
#include <mpi.h>
int main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("From process %d out of %d, Hello world!\n",
                myrank, npes);
        MPI_Finalize();
        return 0;
}
```

MPI Functions: Send, Recv

- Wildcard for source: MPI_ANY_SOURCE.

```
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)
```

```
typedef struct MPI_Status {
        int MPI_SOURCE;
        int MPI_TAG;
        int MPI_ERROR;
};
```

Length of Received Message

- Not directly accessible.
- Reminder: The returned **int** says if the call was successful or not.

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

MPI Functions: Data Types

- MPI_Datatype.
- Correspondence MPI ↔ C data types.
- MPI_BYTE and MPI_PACKED MPI specifics.
- See table 6.2.

Principles of Message-Passing Programming

- 2 key attributes:
  - partitioned address space &
  - only explicit parallelization.

Expensive but costs are explicit.

- Logical view: *p* processes, each with its own exclusive address space.
  - Each piece of data must belong to a partition, i.e., explicit partitioned & placed.
  - All interactions require cooperation of two processes. Point to point communication.

MPI Programming Structure

- Asynchronous.
  - Hard to reason about.
  - Non-deterministic.
- Loosely synchronous.
  - Synchronize to perform interactions.
  - Asynchronous in-between.
  - Easier to reason about.
- **S**ingle **P**rogram **M**ultiple **D**ata.

Send/Recv Example

P0

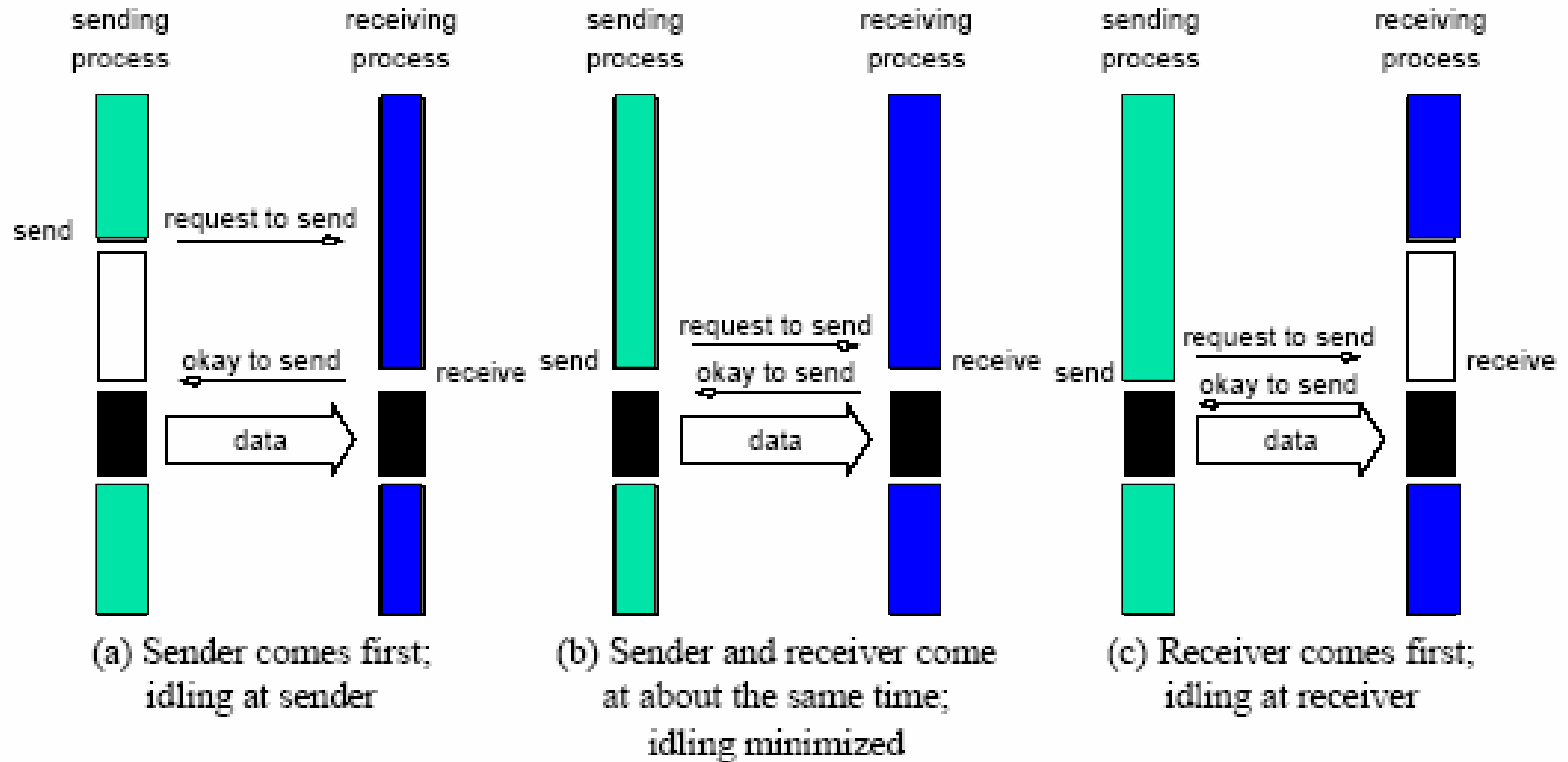
```
a=100;
send(&a, 1, 1);
a=0;
```

P1

```
receive(&a, 1, 0);
printf("%d\n",a);
```

- Expected: what P1 receives is the value of ‘a’ when it was sent.
- But depending on the implementation…
- Design carefully the protocol.

Blocking Non-Buffered Communication



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

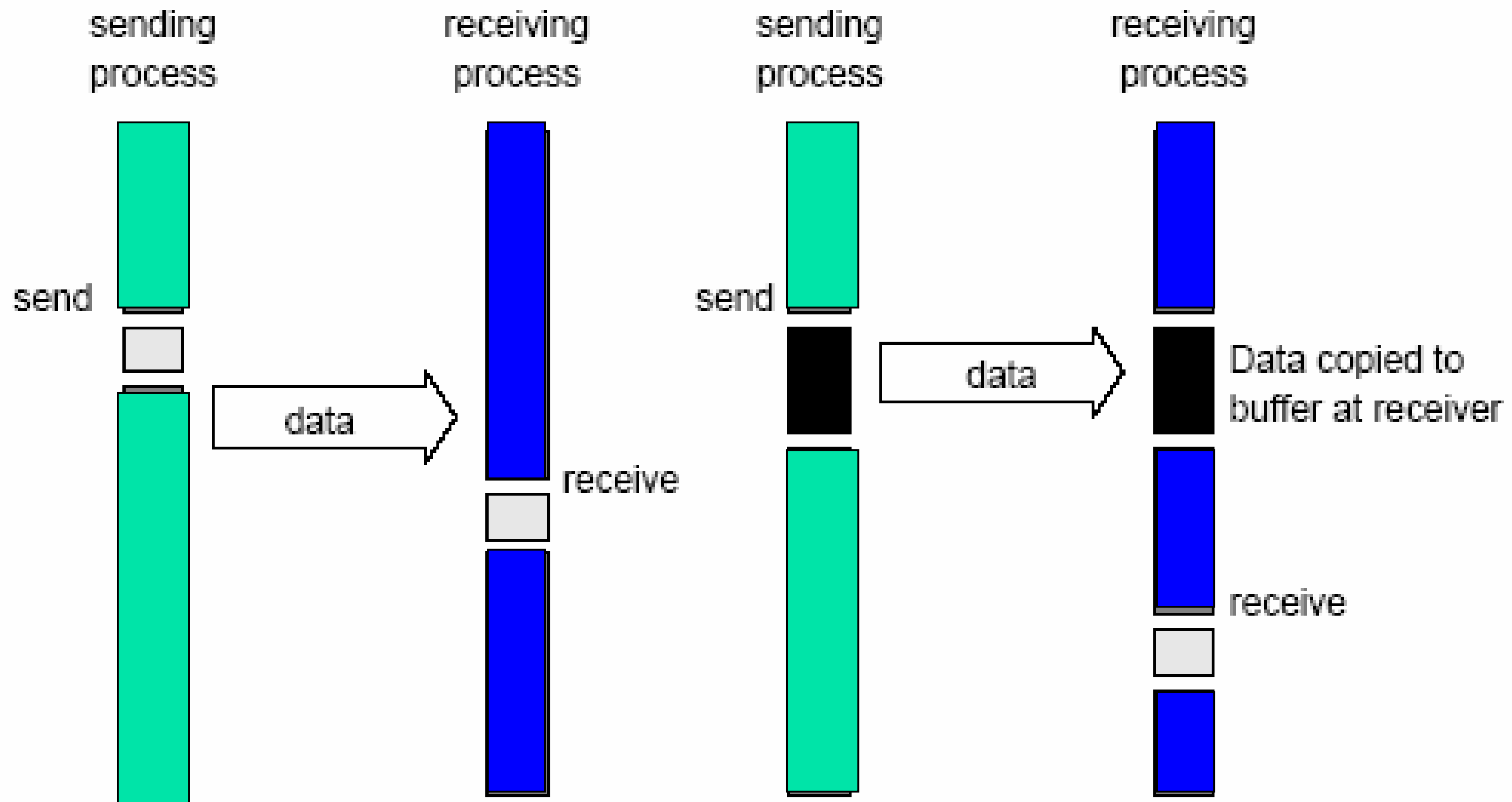
What Happens There?

Simple exchange of ‘a’?

```
P0
send(&a, 1, 1);
recv(&b, 1, 1);
```

```
P1
send(&a, 1, 0);
recv(&b, 1, 0);
```

Blocking Buffered Communication



Examples

OK?

P0
```
for(i = 0;i < 1000; i++) {
        produce_data(&a);
        send(&a, 1, 1);
}
```

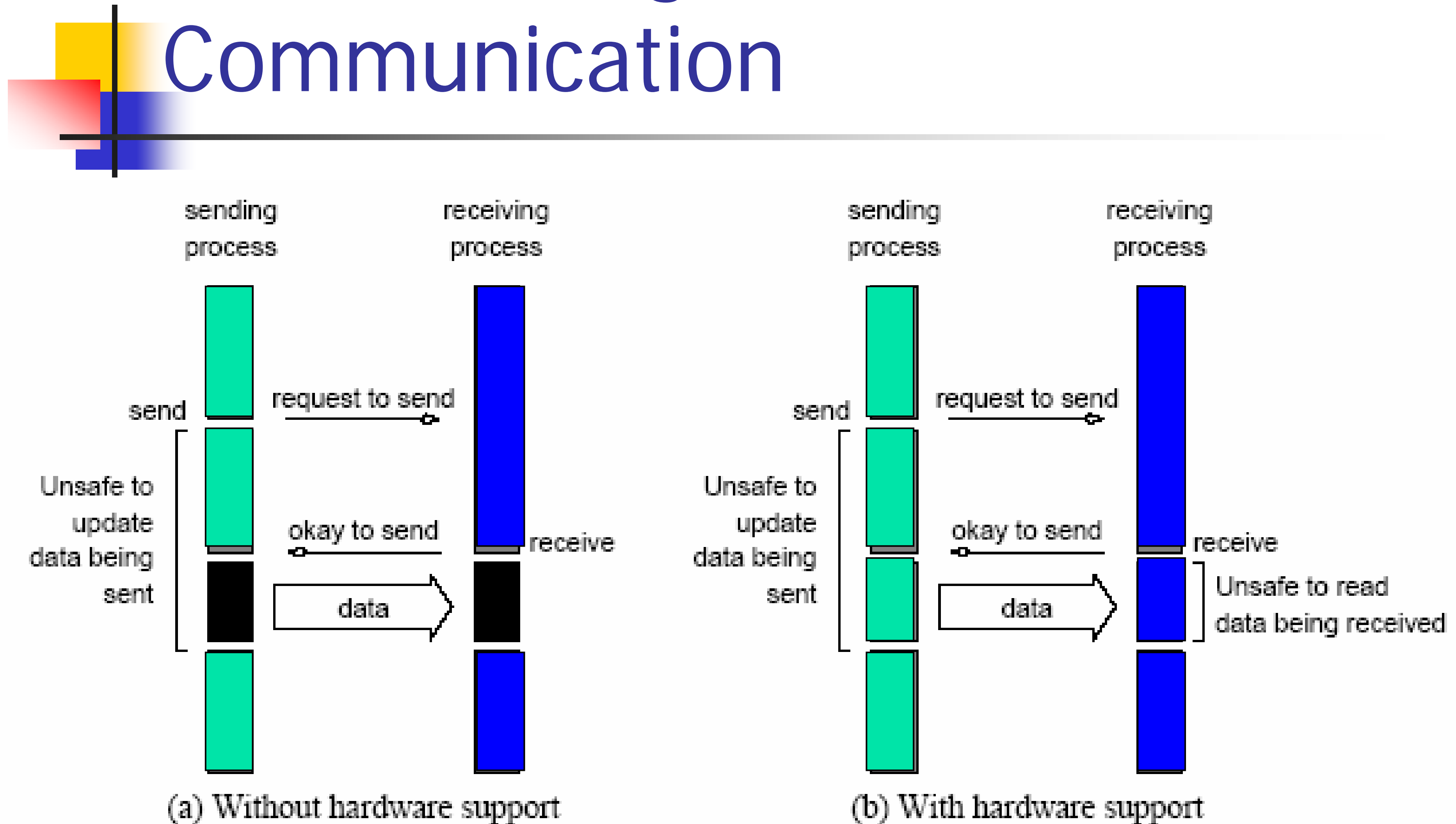
P1
```
for(i = 0; i < 1000; i++) {
        receive(&a, 1, 0);
        consume_data(&a);
}
```

Deadlock

P0
```
receive(&a, 1, 1);
send(&b, 1, 1);
```

P1
```
receive(&a, 1, 0);
send(&b, 1, 0);
```

Non-Blocking Non-Buffered Communication



(a) Without hardware support

(b) With hardware support

Unsafe Program

```
int a[10], b[10], myrank;
MPI_Statu
...
MPI_Comm                                                      );
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myra        ) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
```

Match the order in which the send and the receive operations are issued.

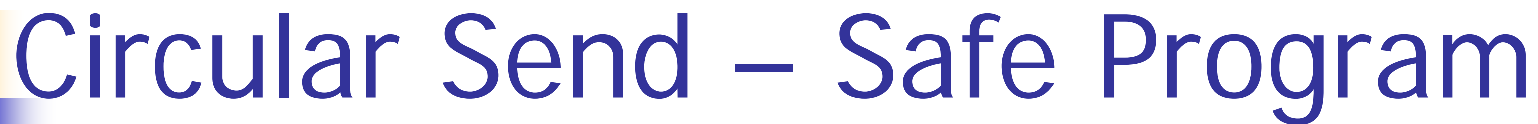
Programmer's responsibility.

Circular Dependency – Unsafe Program

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                    MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                    MPI_COMM_WORLD);
```

Circular Send – Safe Program



```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD);
} else {
        MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
                MPI_COMM_WORLD);
        MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
                MPI_COMM_WORLD);
}
```

Sending and Receiving Messages Simultaneously

- No circular deadlock problem.

```
int MPI_Sendrecv(void *sendbuf,
int sendcount, MPI_Datatype senddatatype,int dest, int sendtag,
void *recvbuf,
int recvcount,MPI_Datatype recvdatatype, int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

Or with replace:

```
int MPI_Sendrecv_replace(void *buf,
int count, MPI_Datatype datatype, int dest, int sendtag,
int source, int recvtag,
MPI_Comm comm, MPI_Status *status)
```

Topologies and Embedding

- MPI allows a programmer to organize processors into logical k-d meshes.
- The processor IDs in `MPI_COMM_WORLD` can be mapped to other communicators (corresponding to higher-dimensional meshes) in many ways.
- The goodness of any such mapping is determined by the interaction pattern of the underlying program and the topology of the machine.
- MPI does not provide the programmer any control over these mappings… but it finds good mapping automatically.

Topologies and Embeddings

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Row-major mapping.

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Column-major mapping.

0	3	4	5
1	2	7	6
14	13	8	9
15	12	11	10

Space-filling curve mapping.

0	1	3	2
4	5	7	6
12	13	15	14
8	9	11	10

Hypercube mapping.

Creating and Using Cartesian Topologies

- Create a new communicator.
- All processes in **comm_old** must call this.
- Embed a virtual topology onto the parallel architecture.

```
int MPI_Cart_create(MPI_Comm comm_old,
int ndims, int *dims, int *periods, int reorder,
MPI_Comm *comm_cart)
```

Rank-Coordinates Conversion

- Dimensions must match.
- Shift processes on the topology.

```
int MPI_Cart_coord(MPI_Comm comm_cart,
int rank, int maxdims, int *coords)
```

```
int MPI_Cart_rank(MPI_Comm comm_cart,
int *coords, int *rank)
```

```
int MPI_Cart_shift(MPI_Comm comm_cart,
int dir, int s_step, int *rank_source, int *rank_dest)
```

Overlapping Communication with Computation

- Transmit messages without interrupting the CPU.
- Recall how blocking send/receive operations work.
- Sometimes desirable to have non-blocking.

Overlapping Communication with Computation

- Functions return before the operations are completed.

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

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

Testing Completion

- Sender: before overriding the data.
- Receiver: before reading the data.
- Test or wait completion.
- De-allocate request handler.

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

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

Previous Example: Safe Program

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Isend(a, 10, MPI_INT, 1, 1, ...);
    MPI_Isend(b, 10, MPI_INT, 1, 2, ...);
}
else if (myrank == 1) {
    MPI_Irecv(b, 10, MPI_INT, 0, 2, ...);
    MPI_Irecv(a, 10, MPI_INT, 0, 1, ...);
}
```

One of these 2 to unblocking calls is enough.