Multicore Computing
Lecture13 - MPI

SUNG KYUN KWAN UNIVERSITY

남 범 석

bnam@skku.edu

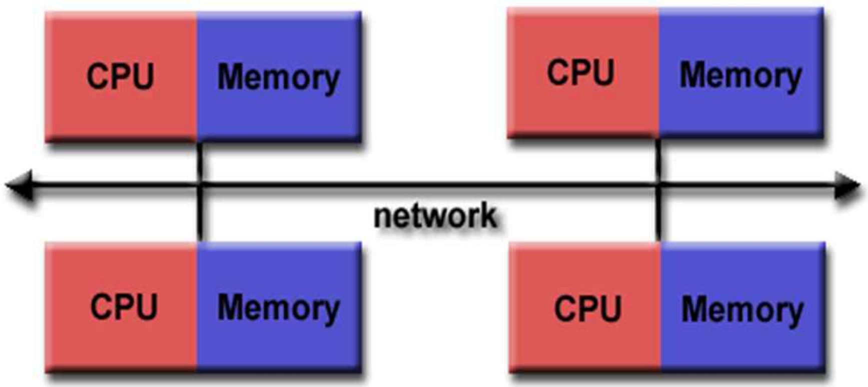
Topics

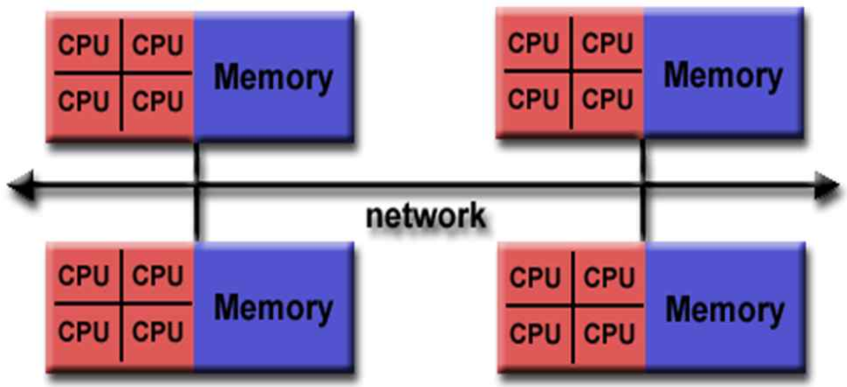
- Principles of Message-Passing Programming
- Building Blocks
  - Send and Receive Operations
- MPI: the Message Passing Interface
- Topologies and embedding
- Overlapping communication with Computation
- Collective communication and computation Operations
- Groups and communicators
- MPI-derived data types

A Distributed Address Space System

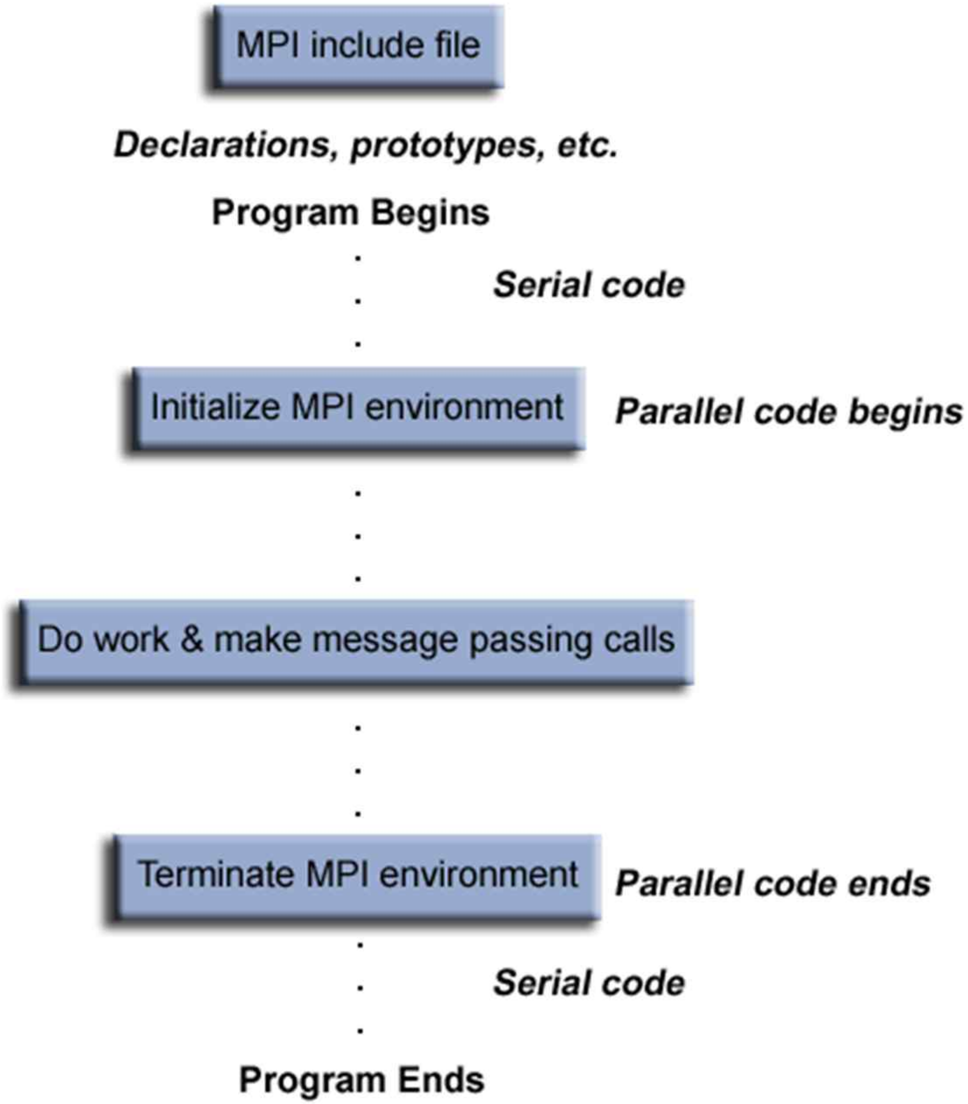
- MPI was originally designed for distributed memory architecture

- Today, MPI runs on virtually any HW platform
  - Distributed memory
  - Shared memory
  - Hybrid





General MPI Program Structure



Principles of Message-Passing

- The logical view of a message-passing paradigm
  - p processes
  - Each with its own exclusive address space
- Data must be explicitly partitioned and placed.
- All interactions (read-only or read/write) are two-sided
  - Process that has the data
  - Process that wants to access the data.
  - Underlying costs are explicit
- Using the single program multiple data (SPMD) model

Send and Receive Operations

- Prototypes

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

- Consider the following code segments:

```
P0                      P1
a = 100;                receive(&a, 1, 0)
send(&a, 1, 1);         printf("%d\n", a);
a = 0;
```

- The semantics of the send
  - Value received by process P1 must be 100, not 0
  - Motivates the design of the send and receive protocols
    - Non-buffered blocking message passing
    - Buffered blocking message passing
    - Non-blocking message passing

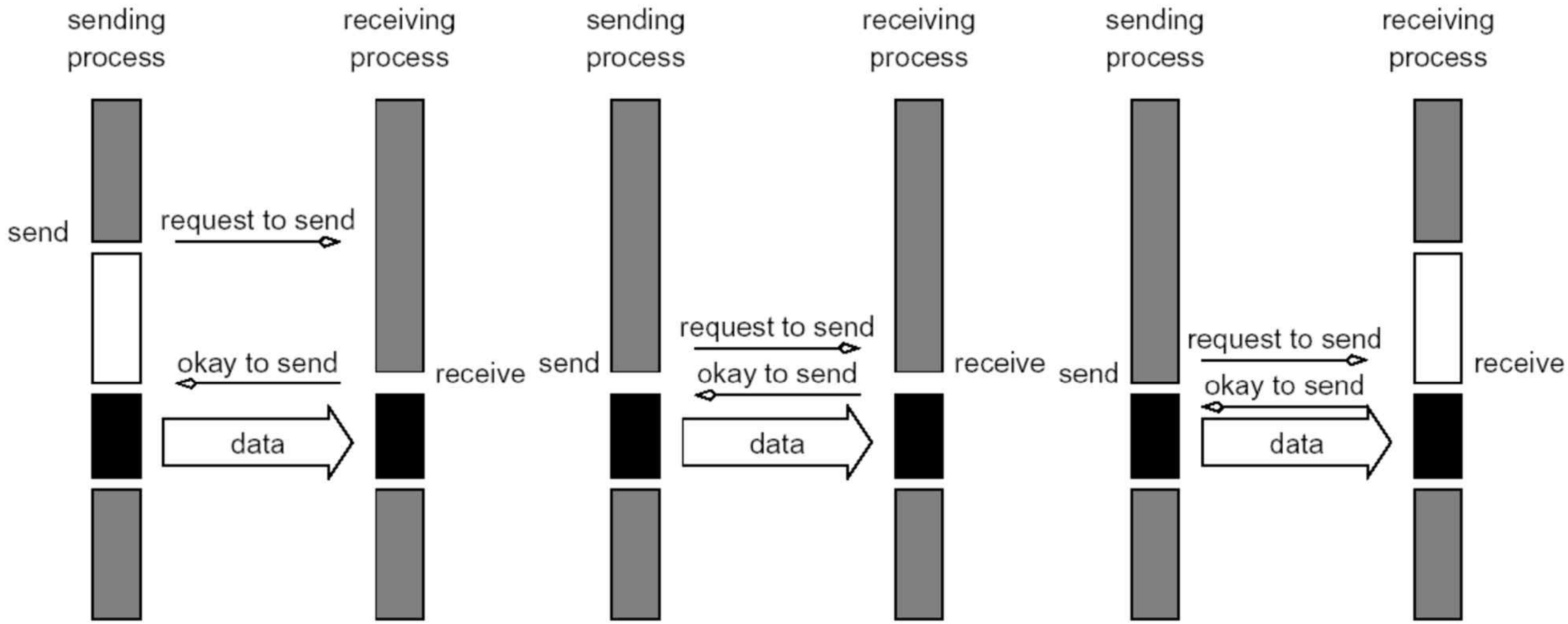
Non-Buffered Blocking Message Passing

- A simple method
  - Send operation to return only when it is safe to do so
  - Send does not return until the matching receive has been encountered
- Issues
  - Idling and deadlocks
    - Deadlock example

P0:	P1:
send(&a, 1, 1);	send(&a, 1, 0);
receive(&b, 1, 1);	receive(&b, 1, 0);

Non-Buffered Blocking Message Passing

Handshake for a blocking non-buffered send/receive operation



(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

Idling occurs when sender and receiver do not reach communication point at similar times

Buffered Blocking Message Passing

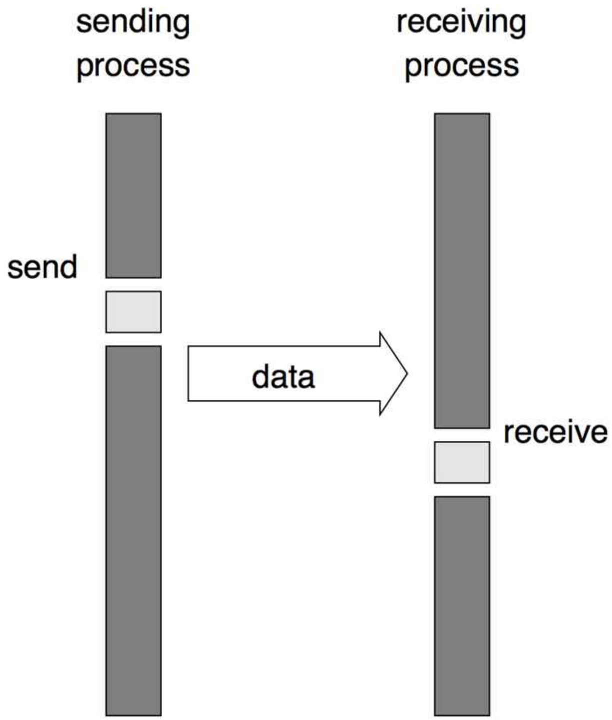
- Process
  - Sender copies data into buffer
  - Sender returns after the copy completes
  - Data may be buffered at the receiver
- A simple solution to idling and deadlock
- Trade-off
  - Buffering trades idling overhead for buffer copying overhead

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

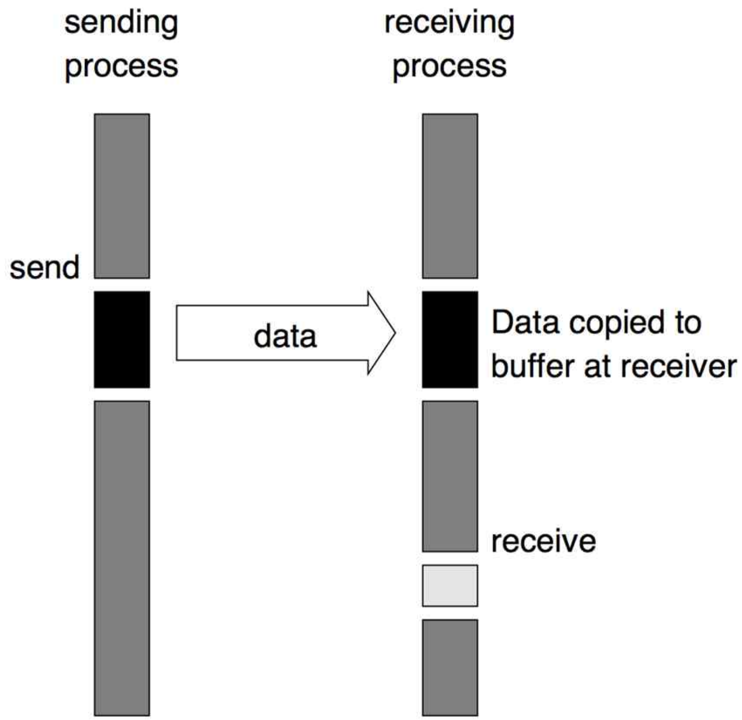
```
P1:
send(&a, 1, 0);
receive(&b, 1, 0);
```

Buffered Blocking Message Passing

Blocking buffered transfer protocols



(a) With communication hardware



(b) w/o communication hardware: sender interrupts receiver and deposits data in buffer at receiver end.

Buffered Blocking Message Passing

- Bounded buffer sizes can have significant impact on performance

```
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);
}
```

- Buffer overflow leads to blocking sender. Programmers need to be aware of bounded buffer requirements

Buffered Blocking Message Passing

- Deadlocks are still possible with buffering since receive operations block.

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

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

Non-Blocking Message Passing

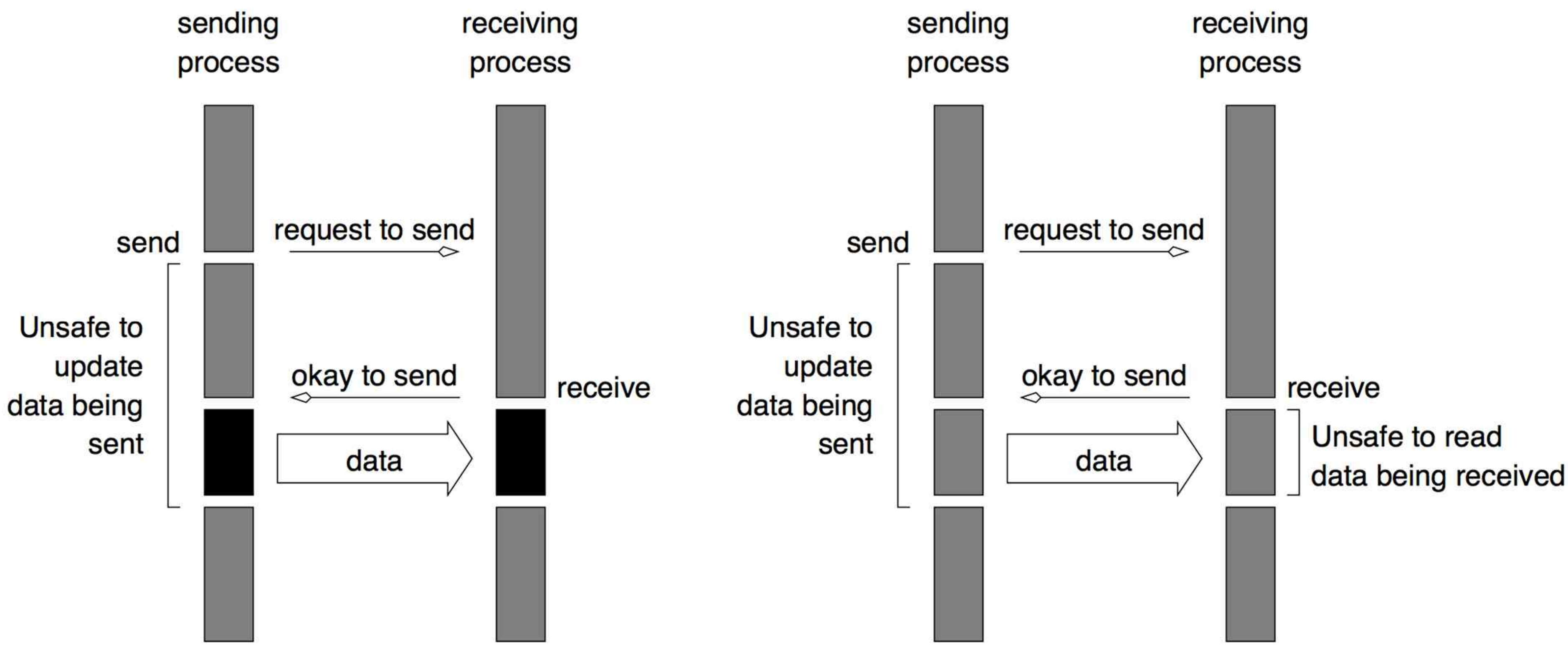
- Send and receive returns before it is semantically safe
  - Sender: data can be overwritten before it is sent
  - Receiver: data can be read before it is received

- Programmer must ensure semantics of the send and receive.
  - A check-status operation is accompanied

- Benefit
  - Capable of overlapping communication overheads with useful computations

- Message passing libraries provide both blocking and non-blocking primitives

Non-Blocking Message Passing



(a) Without communication hardware (b) With communication hardware

MPI: Message Passing Interface

- Standard library for message-passing
  - Portable
  - Ubiquitously available
  - High performance
  - C and Fortran APIs
- MPI standard defines
  - Syntax as well as the semantics of library routines
- Details
  - MPI routines, data-types, and constants
  - Prefixed by “MPI_”
- 6 Golden MPI functions
  - 125 functions but 6 most used functions

The minimal set of MPI routines.

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

Starting and Terminating MPI Programs

- `int MPI_Init(int *argc, char ***argv)`
  - Initialize the MPI environment
    - Strips off any MPI related command-line arguments.
  - Must be called prior to other MPI routines

- `int MPI_Finalize()`
  - Must be called at the end of the computation
  - Performs various clean-up tasks to terminate the MPI environment.

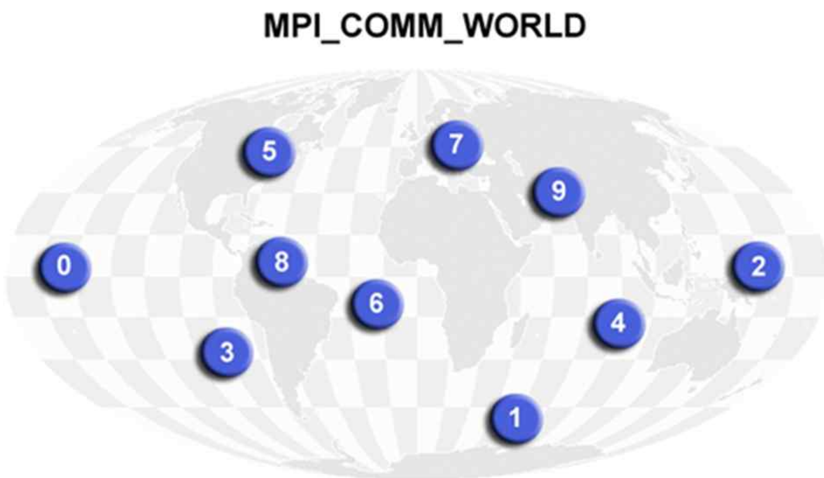
- Return code
  - MPI_SUCCESS
  - MPI_ERROR

A minimal MPI program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello, world!\n");
  MPI_Finalize();
  Return 0;
}
```

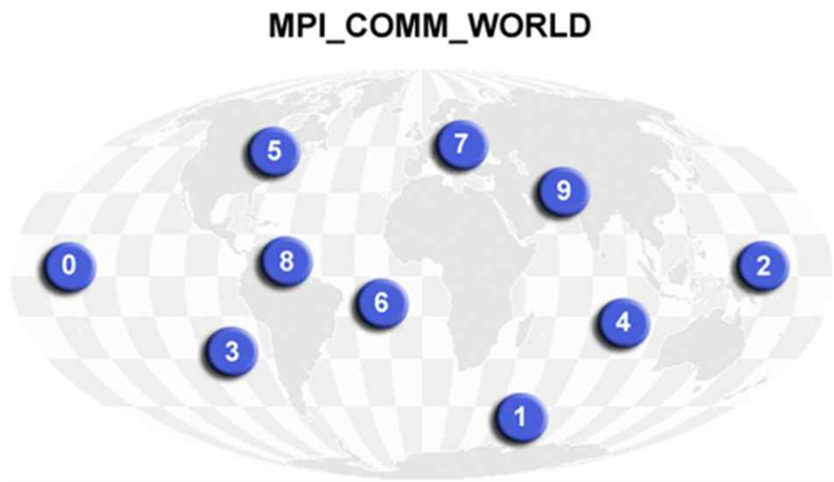
Communicators

- A communicator defines a communication domain
  - A set of processes allowed to communicate with each other
- Type `MPI_Comm`
  - Specifies the communication domain
  - Used as arguments to all message transfer MPI routines
- A process can belong to many different communication domains
- `MPI_COMM_WORLD`
  - Default communicator
  - Includes all the processes



Querying Information in Communicator

- `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Get the number of processes
- `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Index of the calling process
  - 0 <= rank < communicator size



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)
  ```

Hello World using MPI

```
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Compilation

- Compile

wrapper script to compile *source file*

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
$ mpic++ -g -Wall -o mpi_hello mpi_hello.cpp
```

Execution

```
$ mpiexec -n <number of processes> <executable>
$ mpiexec -n 4 --machinefile hosts.txt --map-by node a.out
```

- --machinefile tells MPI to run the program on the machines listed in hosts.txt file.

$ mpiexec -n 4 --hostfile hosts.txt --map-by node hostname

swin

swye

swji

swin

$ mpiexec -n 4 --hostfile hosts.txt hostname

swin

swin

swin

swin

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)
  ```
- buf
  - Pointer to a sending/receiving buffer
- count
  - # of items to transfer
  - Of type specified by datatype

MPI Datatypes

MPI Datatype	C Datatype
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_LONG_DOUBLE	long double
MPI_BYTE	8 bits
MPI_PACKED	Packed sequence of bytes

Sending and Receiving Messages

- Message tag
  - Tags allow programmers to deal with the arrival of messages in an orderly manner
  - Range of tag
    - 0 .. 32767 (2^{15} -1) are guaranteed
    - The upper bound is provided by MPI_TAG_UB
    - `MPI_ANY_TAG` can be used as a wildcard value

Message matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
        send_comm);
```

r

MPI_Send

src = q

MPI_Recv

dest = r

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
        recv_comm, &status);
```

q

Receiving Messages

- Two wildcards of MPI_recv
  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

MPI_ANY_SOURCE

MPI_ANY_TAG

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

- A receiver can get a message without knowing
  - The amount of data in the message
  - Sender of the message
  - Tag of the message

Receiving Messages

- MPI_Status
  - Stores information about the MPI_Recv operation.
  - Data structure contains:

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

- `int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)`
  - Returns the precise count of data items received