



# Message-Passing Programming: MPI

National Tsing Hua University  
2019, Fall Semester

# Outline

## ■ MPI Introduction

- History & Evolution

## ■ Communication Methods

- Synchronous / Asynchronous
- Blocking / Non-Blocking

## ■ MPI API

- Point-to-Point Communication Routines
- Collective Communication Routines
- Group and Communicator Management Routines

## ■ MPI-IO

# What is MPI

- **MPI = Message Passing Interface**

- A **specification** for the developers and users of message passing libraries

  - By itself, it is an **interface NOT a library**

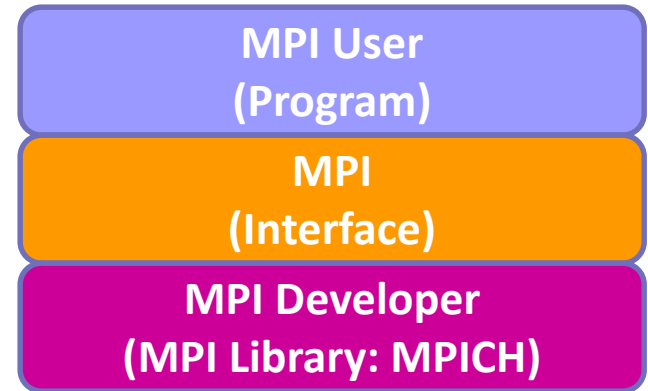
- Commonly used for **distributed memory system & high-performance computing**

- Goal:

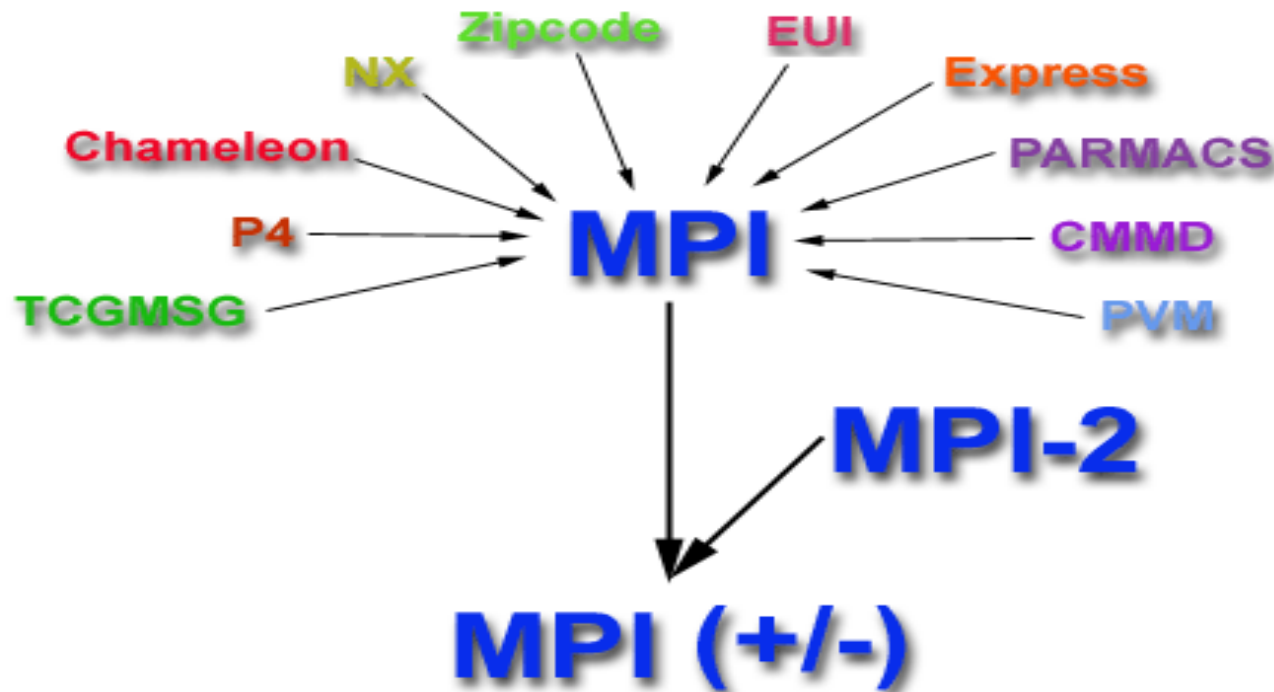
  - **Portable**: Run on different machines or platforms

  - Scalable: Run on million of compute nodes

  - Flexible: Isolate MPI developers from MPI programmers (users)

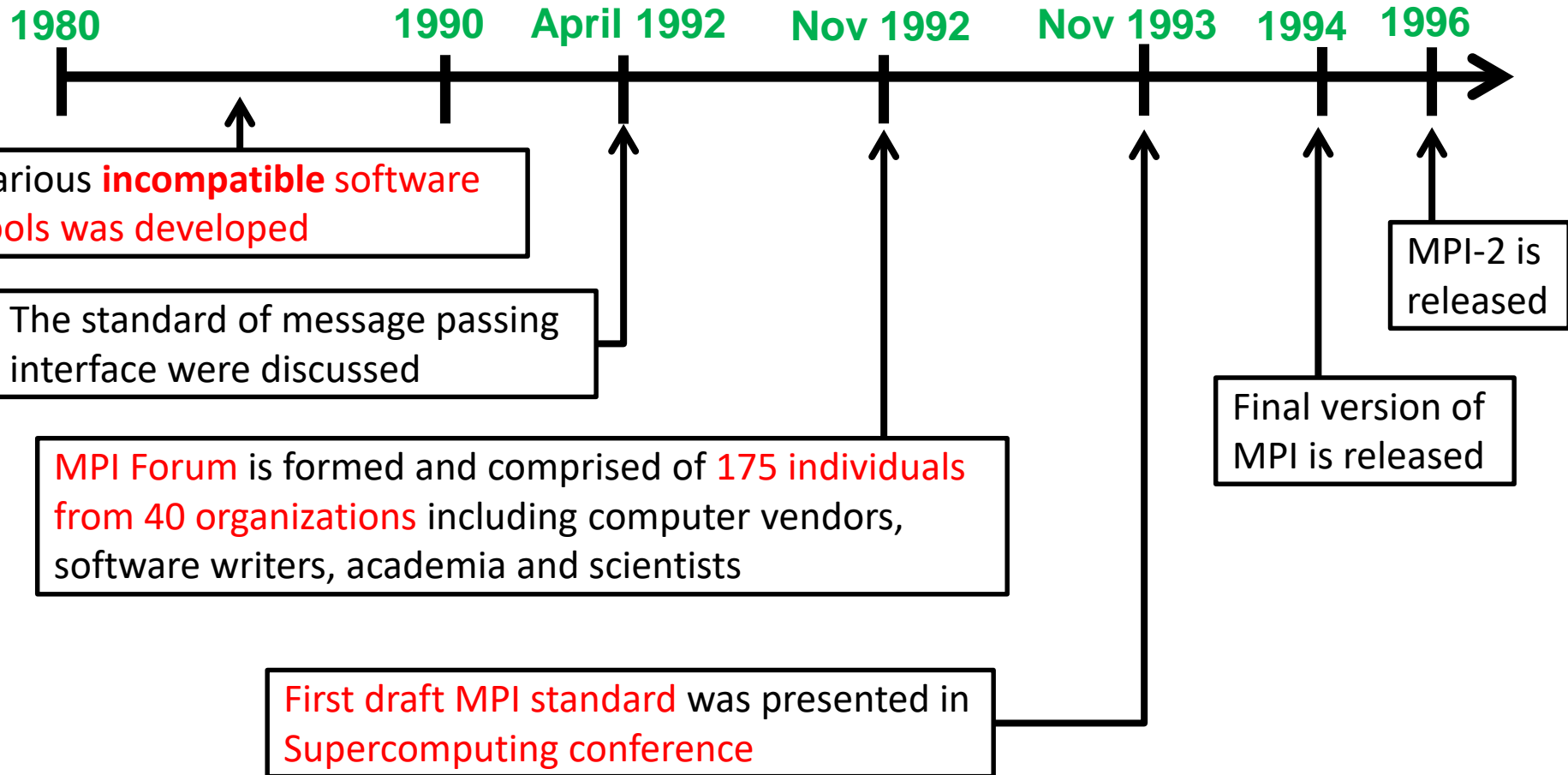


# History and Evolution



- MPI resulted from the efforts of numerous individuals and groups
- Today, MPI implementations are a combination of MPI-1 and MPI-2. A few implementations include the full functionality of both
- The MPI Forum is now drafting the MPI-3 standard

# History and Evolution



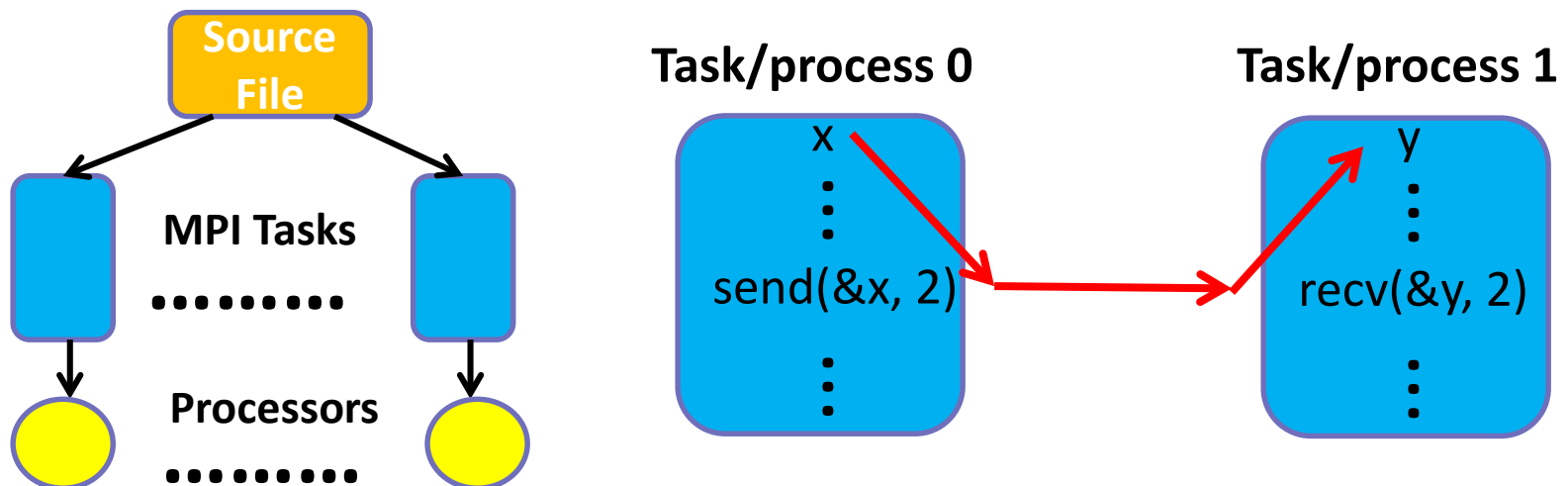
# Programming Model

## ■ SPMD: Single Program Multiple Data

- Allow **tasks** to **branch or conditionally execute only parts of the program** they are designed to execute
- MPI tasks of a parallel program **can not be dynamically spawned** during run time. (MPI-2 addresses this issue).

## ■ Distributed memory

- MPI provide method of **sending & receiving message**



# Outline

- MPI Introduction
- Communication Methods
  - Synchronous / Asynchronous
  - Blocking / Non-Blocking
- MPI API
- MPI-IO

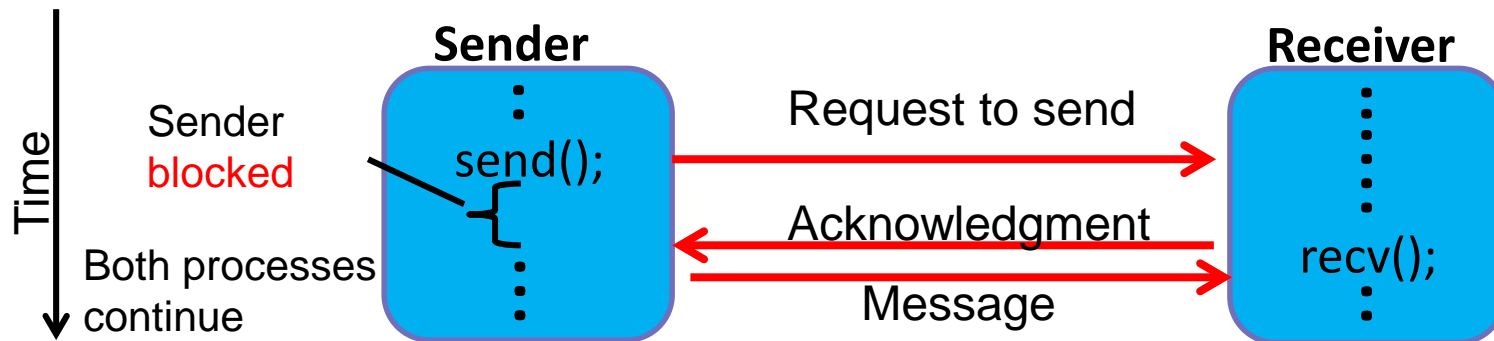
# Communication Methods

- From the view of **the pair of communicated processes**
  - **Synchronous communication** --- sending and receiving data occurs **simultaneously**
  - **Asynchronous communication** --- sending and receiving data occurs **non-simultaneously**
- From the view of **individual function call**
  - **Blocking** --- has been used to describe routines that do not **return until the transfer is completed**
  - **Non-blocking** --- has been used to describe routines that return whether or not the message had been received
- Synchronous vs. blocking:
  - Synchronous comm. commonly implemented by blocking call
  - Synchronous comm. intrinsically performs two action:  
**Transfer Data & Synchronize Processes**

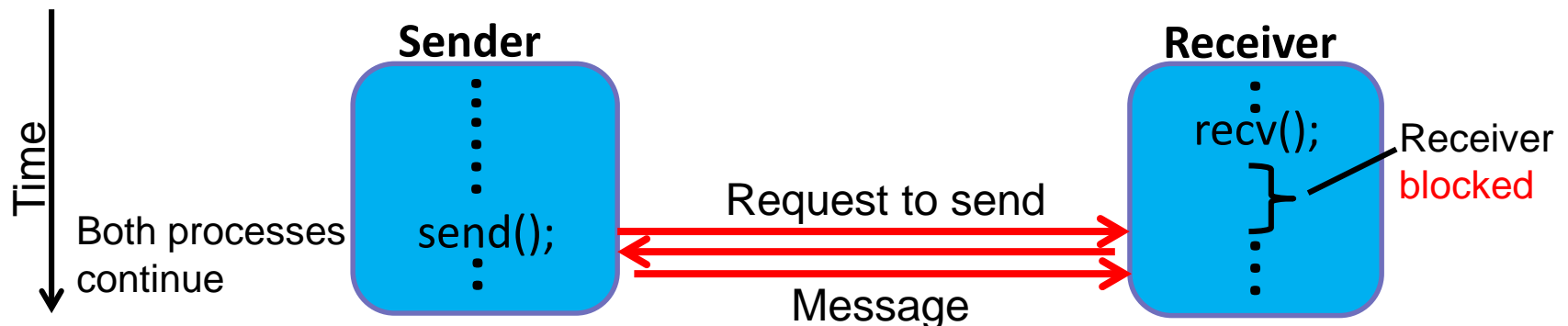


# Synchronous/Blocking Message Passing

- **Sender:** wait until the complete message can be accepted by the receiver before sending the message

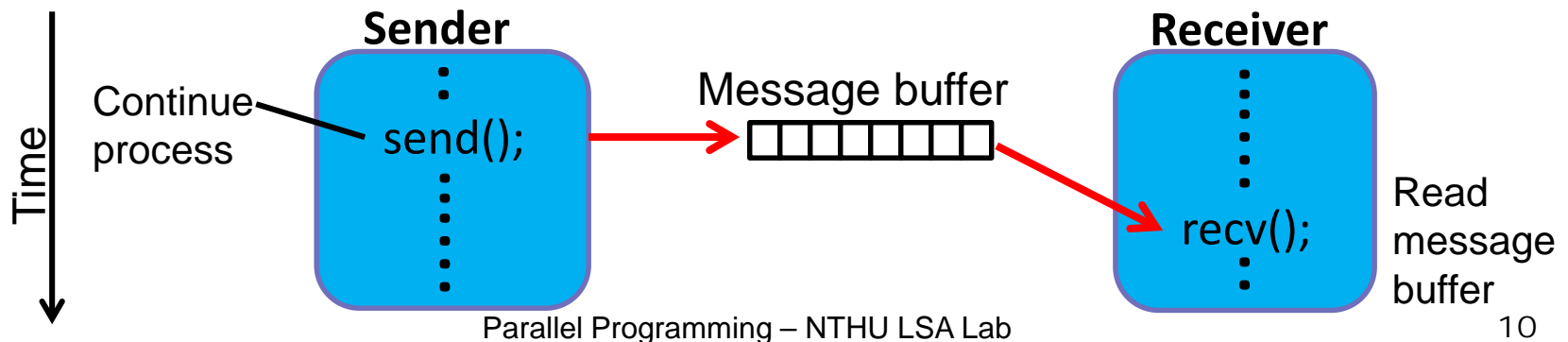


- **Receiver:** wait until the message it is expecting arrives



# Asynchronous/Non-Blocking Message Passing

- How message-passing routines can return before the message transfer has been completed?
  - Generally, a **message buffer** needed between source and destination to hold message
  - Message buffer is a **memory space** at the **sender and/or receiver side**
  - For **send routine**, once the **local actions** have been **completed** and the **message is safely on its way**, the process can continue with subsequent work



# Outline

- MPI Introduction
- Communication Methods
- MPI API
  - Getting Start
  - Environment Management Routine
  - Point-to-Point Communication Routines
  - Collective Communication Routines
  - Group and Communicator Management Routines

# Getting Start

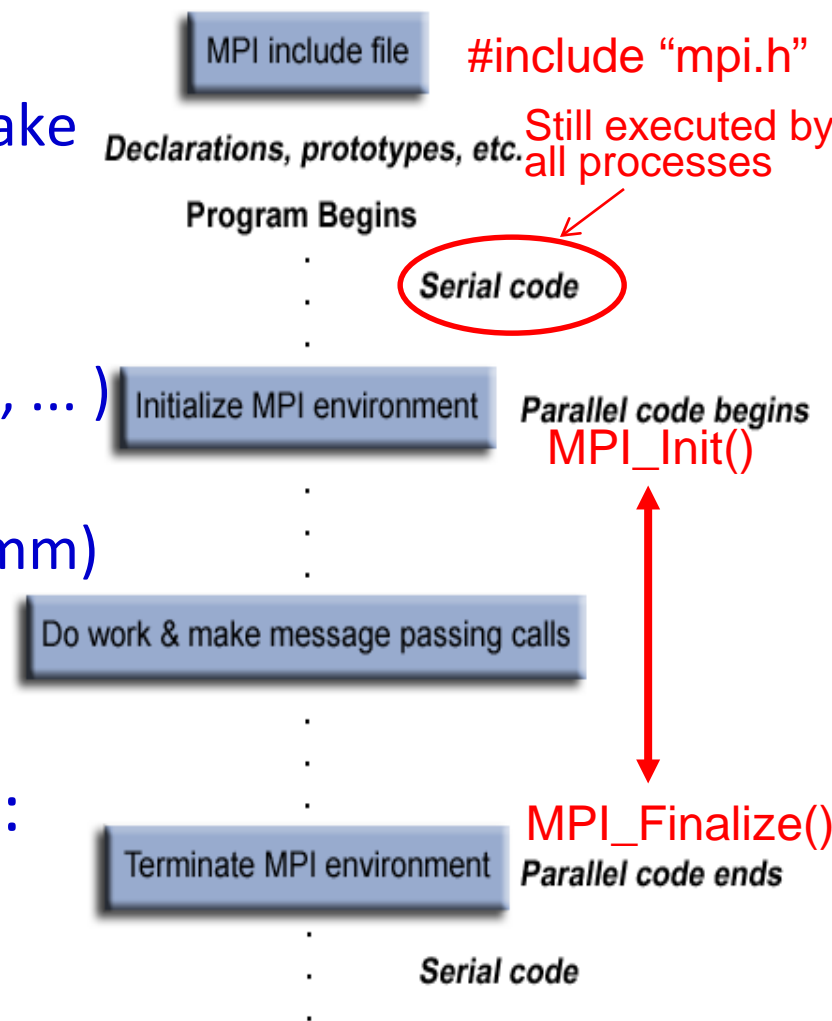
## ■ Header file: “mpi.h”

- Required for all programs that make MPI library call

## ■ MPI calls:

- **Format:** `rc = MPI_Xxx(parameter, ... )`
- **Example:** `rc = MPI_Bcast(&buffer, count, datatype, root, comm)`
- **Error code:** return as “rc”;  
`rc=MPI_SUCCESS` if successful

## ■ General MPI program structure:



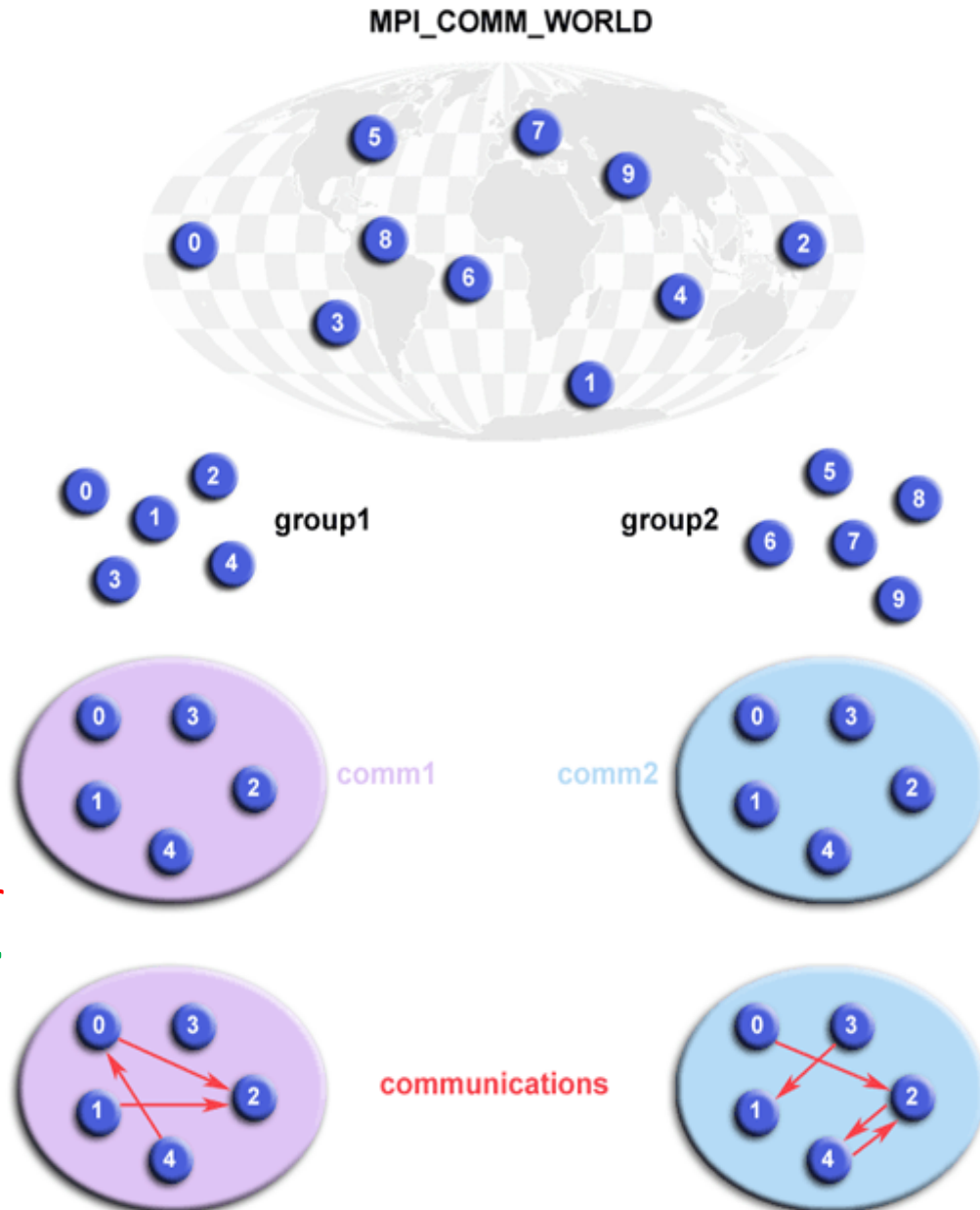
# Getting Start

## ■ Communicators and Groups:

- **Groups** define which **collection of processes** may communicate with each other
- Each group is associated with a **communicator** to perform its communication function calls
- **MPI\_COMM\_WORLD** is the **pre-defined** communicator for all processors

## ■ Rank

- An **unique identifier** (task ID) for each process in a **communicator**
- Assigned by the system when the process initializes
- Contiguous and **begin at zero**



# Environment Management Routines

## ■ **MPI\_Init ()**

- Initializes the MPI execution environment
- Must be called before any other MPI functions
- Must be called only once in an MPI program

## ■ **MPI\_Finalize ()**

- Terminates the MPI execution environment
- No other MPI routines may be called after it

## ■ **MPI\_Comm\_size (comm, &size)**

- Determines the number of processes in the group associated with a communicator

## ■ **MPI\_Comm\_rank (comm, &rank)**

- Determines the rank of the calling process within the communicator
- This rank is often referred to as a task ID

# Example

```
#include "mpi.h"
int main (int argc, char *argv[]) {
    int numtasks, rank, rc;
    rc = MPI_Init (&argc,&argv);
    if (rc != MPI_SUCCESS) {
        printf ("Error starting MPI program. Terminating.\n");
        MPI_Abort (MPI_COMM_WORLD, rc);
    }
    MPI_Comm_size (MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    printf ("Number of tasks= %d My rank= %d\n", numtasks, rank);
    MPI_Finalize ();
}
```

# Outline

- MPI Introduction
- Communication Methods
- MPI API
  - Getting Start
  - Environment Management Routine
  - Point-to-Point Communication Routines
  - Collective Communication Routines
  - Group and Communicator Management Routines



# Point-to-Point Communication Routines

Blocking send	<code>MPI_Send(buffer,count,type,dest,tag,comm)</code>
Non-blocking send	<code>MPI_Isend(buffer,count,type,dest,tag,comm,request)</code>
Blocking receive	<code>MPI_Recv(buffer,count,type,source,tag,comm,status)</code>
Non-blocking receive	<code>MPI_Irecv(buffer,count,type,source,tag,comm,request)</code>

- **buffer**: Address space that references the data to be sent or received
- **type**: `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_DOUBLE`, ...
- **count**: Indicates the number of data elements of a particular type to be sent or received
- **comm**: indicates the communication context
- **source/dest**: the rank (task ID) of the sender/receiver
- **tag**: arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations must match message tags. `MPI_ANY_TAG` is the wild card.
- **status**: status after operation
- **request**: used by non-blocking send and receive operations

# Blocking Example

Blocking send	<code>MPI_Send(buffer,count,type,dest,tag,comm)</code>
Blocking receive	<code>MPI_Recv(buffer,count,type,source,tag,comm,status)</code>

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank); /* find process rank */  
if (myRank == 0) {  
    int x=10;  
    MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);  
} else if (myRank == 1) {  
    int x;  
    MPI_Recv(&x, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, status);  
}
```

# Non-Blocking Example

Non-Blocking send	<code>MPI_Isend(buffer,count,type,dest,tag,comm,request)</code>
Non-Blocking receive	<code>MPI_Irecv(buffer,count,type,source,tag,comm,request)</code>

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find process rank */
if (myrank == 0) {
    int x=10;
    MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, req1);
    compute();
} else if (myrank == 1) {
    int x;
    MPI_Irecv(&x, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, req1);
}
MPI_Wait(req1, status);
```

- `MPI_Wait()` **blocks** until the operation has actually **completed**
- `MPI_Test()` returns with a flag set indicating whether operation completed at that time.

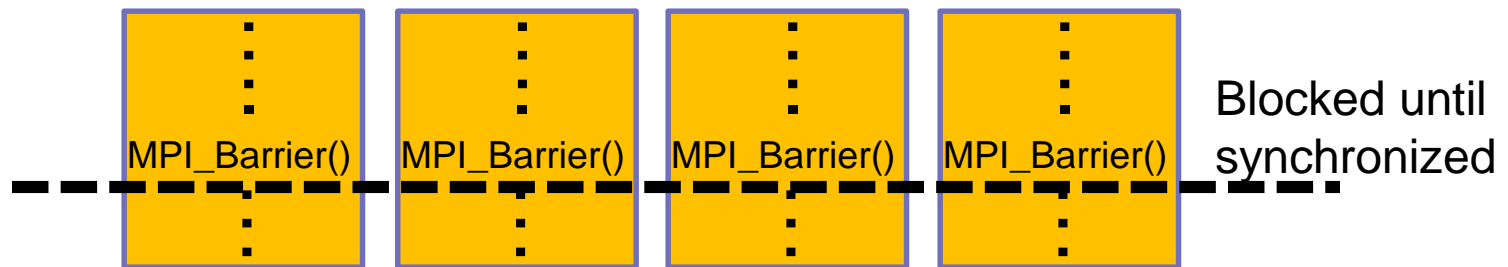
# Outline

- MPI Introduction
- Communication Methods
- MPI API
  - Getting Start
  - Environment Management Routine
  - Point-to-Point Communication Routines
  - **Collective Communication Routines**
  - Group and Communicator Management Routines
- MPI-IO

# Collective Communication Routines

## ■ MPI\_Barrier(comm)

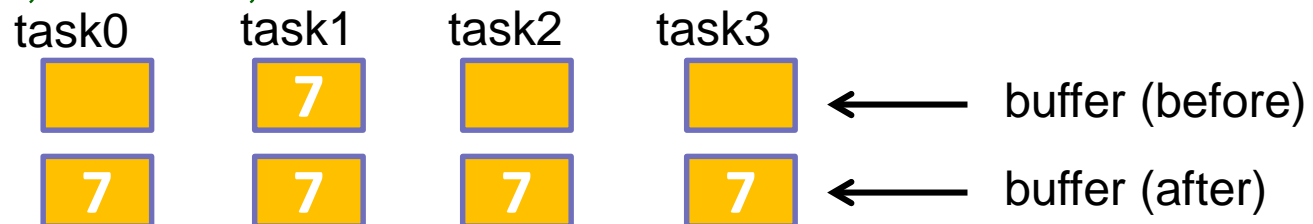
- Creates a barrier **synchronization** in a group
- **Blocks** until **all tasks** in the **group** reach the same MPI\_Barrier call



## ■ MPI\_Bcast(&buffer, count, datatype, root, comm)

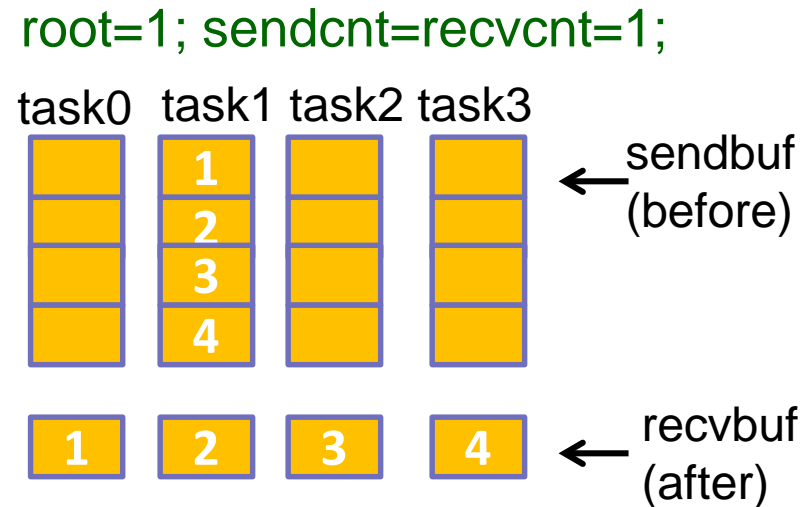
- Broadcasts (sends) a message from the process with rank "root" to all other processes **in the group**

root=1; count=1;

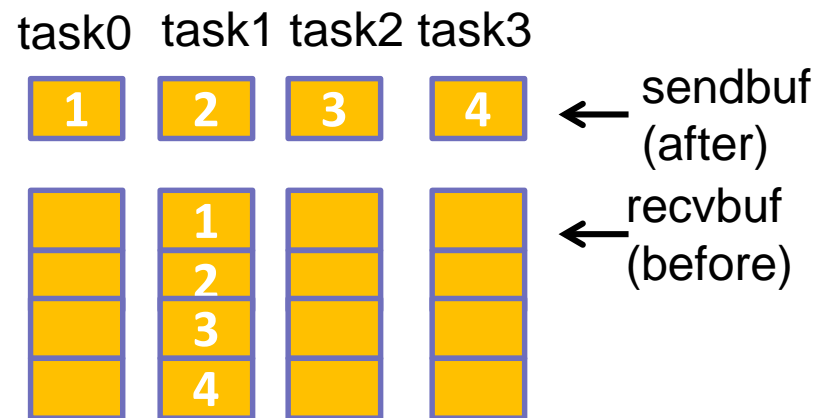


# Collective Communication Routines

- `MPI_Scatter` (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, **root**, comm)
  - Distributes **distinct** messages from a source task to all tasks



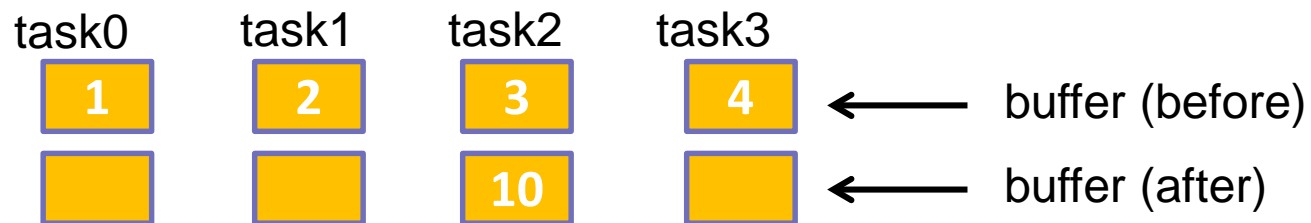
- `MPI_Gather` (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, **root**, comm)
  - Gathers **distinct** messages from each task in the group to a single destination task
  - This routine is the **reverse operation of MPI\_Scatter**



# Collective Communication Routines

- `MPI_Reduce (&sendbuf, &recvbuf, count, datatype, op, dest, comm)`
  - Applies a **reduction operation** on **all tasks** in the group and places the **result in one task**

`dest=2, count=1; op=MPI_SUM`



## ■ **Pre-defined** Reduction Operations

<code>MPI_MAX</code>	Maximum	<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum	<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical AND	<code>MPI_BAND</code>	Bit-wise AND
<code>MPI_LOR</code>	Logical OR	<code>MPI_BOR</code>	Bit-wise OR
<code>MPI_LXOR</code>	Logical XOR	<code>MPI_BXOR</code>	Bit-wise XOR

# Collective Communication Routines

- `MPI_Allgather` (&sendbuf, sendcount, sendtype, &recvbuf, rcvcount, recvtype, comm)
  - Concatenation of data to all tasks
  - This is equivalent to an `MPI_Gather` followed by an `MPI_Bcast`
- `MPI_Allreduce` (&sendbuf, &recvbuf, count, datatype, op, comm)
  - Applies a reduction operation and places the result in all tasks
  - This is equivalent to an `MPI_Reduce` followed by an `MPI_Bcast`

`sendcnt = rcvcnt = 1;`

task0 task1 task2 task3

1	2	3	4
---	---	---	---

← sendbuf  
(before)

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

← recvbuf  
(after)

`count=1; op=MPI_SUM`

task0 task1 task2 task3

1	2	3	4
---	---	---	---

← buffer  
(before)

10	10	10	10
----	----	----	----

← buffer  
(after)



# Outline

- MPI Introduction
- Communication Methods
- MPI API
  - Getting Start
  - Environment Management Routine
  - Point-to-Point Communication Routines
  - Collective Communication Routines
  - Group and Communicator Management Routines

# Group and Communicator Routines

\* All these calls are **collective call** that needs to be called by all the processes in the communicator

- Group & Communicator Data Type

- MPI\_Group

- MPI\_Comm

- MPI\_Comm\_group(Comm, &Group)

- Access the group associated with a given communicator

- MPI\_Group\_incl(Group, size, ranks[], &NewGroup)

- Produce a group by including a subset of members from an existing group

- all members must be distinct

- MPI\_Comm\_create(Comm, NewGroup, &NewComm)

- Create a new communicator

- The new communicator must be a **subset of the original group**

# Examples: Divide MPI tasks into two groups

```
int rank, numtasks;
MPI_Group orig_group, new_group;
MPI_Comm new_comm


MPI_Init();
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
int rank1[4] = {0,1,2,3}; int rank2[4] = {5,6,7,8};
if (rank < numtasks/2) MPI_Group_incl(orig_group, 4, rank1, &new_group);
else
    MPI_Group_incl(orig_group, 4, rank2, &new_group);

/* Create new communicator & Broadcast within the new group */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Barrier(new_comm);
MPI_Finalize();
```

All MPI tasks must call MPI\_Group\_incl, but they don't necessary to be included in the new group



# Outline

## ■ MPI Introduction

- History & Evolution

## ■ Communication Methods

- Synchronous / Asynchronous
- Blocking / Non-Blocking

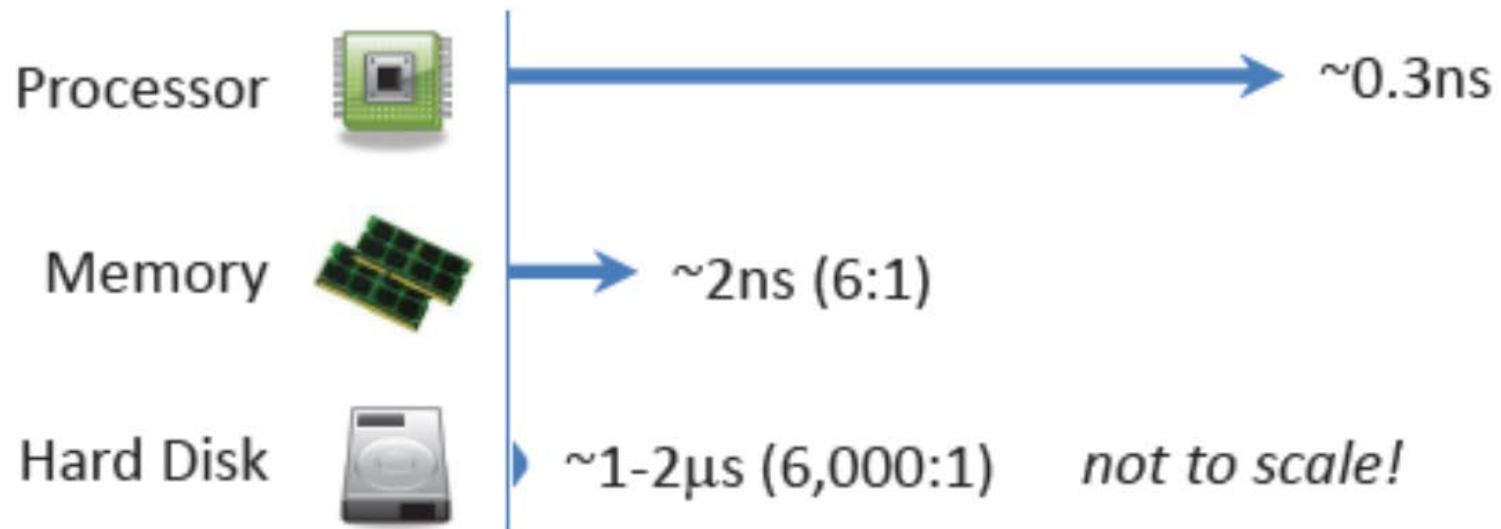
## ■ MPI API

- Point-to-Point Communication Routines
- Collective Communication Routines
- Group and Communicator Management Routines

## ■ MPI-IO

# Relative Speed of Components in HPC Platform

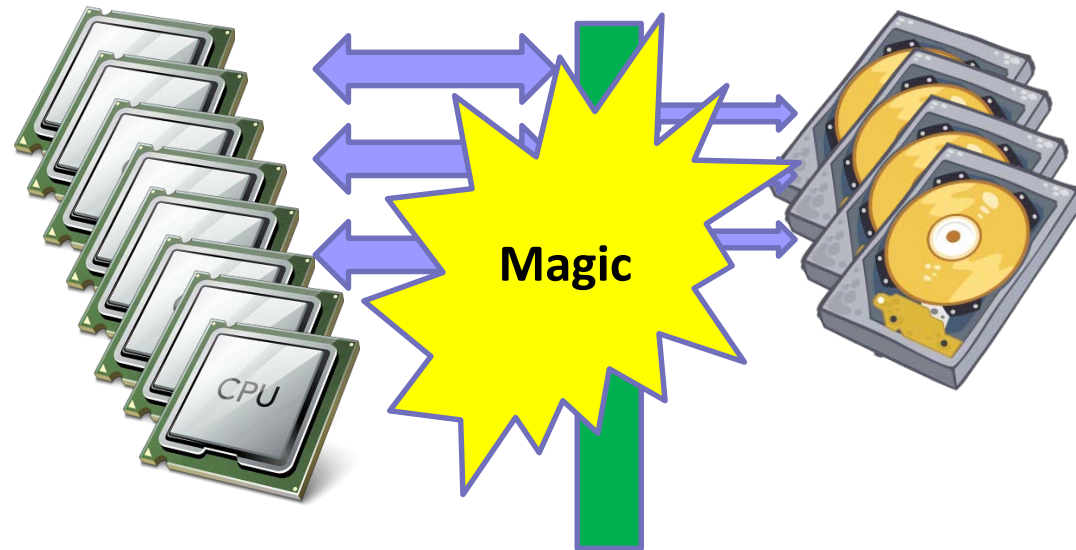
- An HPC platform's I/O subsystems are typically slow as compared to its other parts
- The I/O gap between memory speed and average disk access stands at roughly  $10^{-3}$



# Concurrent Data Access in a Cluster

We need some magic to make the collection of spinning disks act like a single disk ...

a few hundreds spinning disks

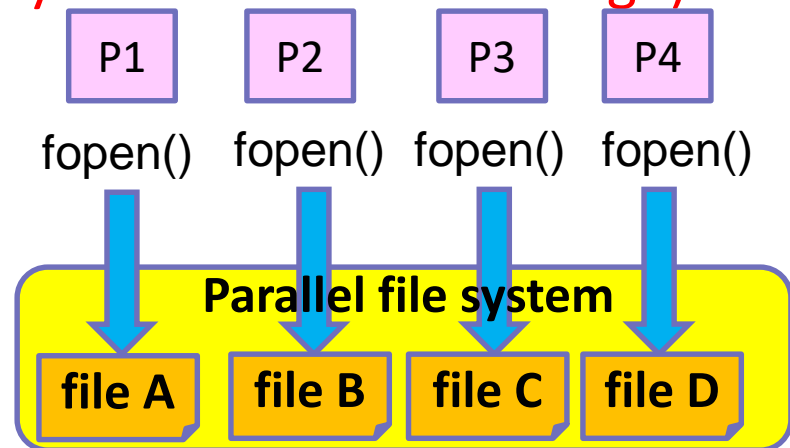
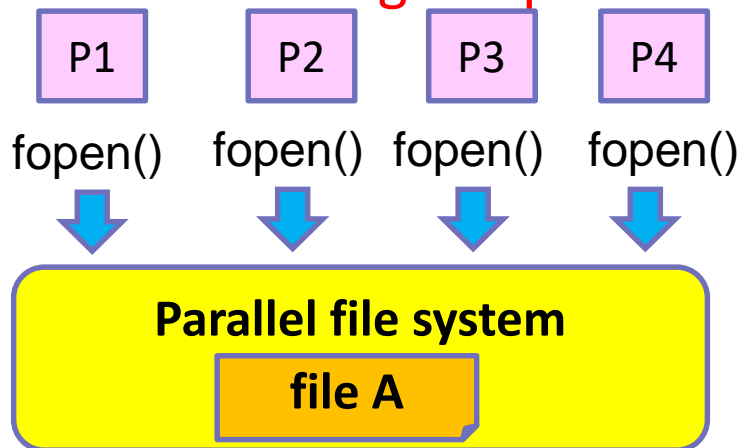


hundreds of thousands of processors

# POSIX File Access Operations

## ■ POSIX file system call “fopen()”:

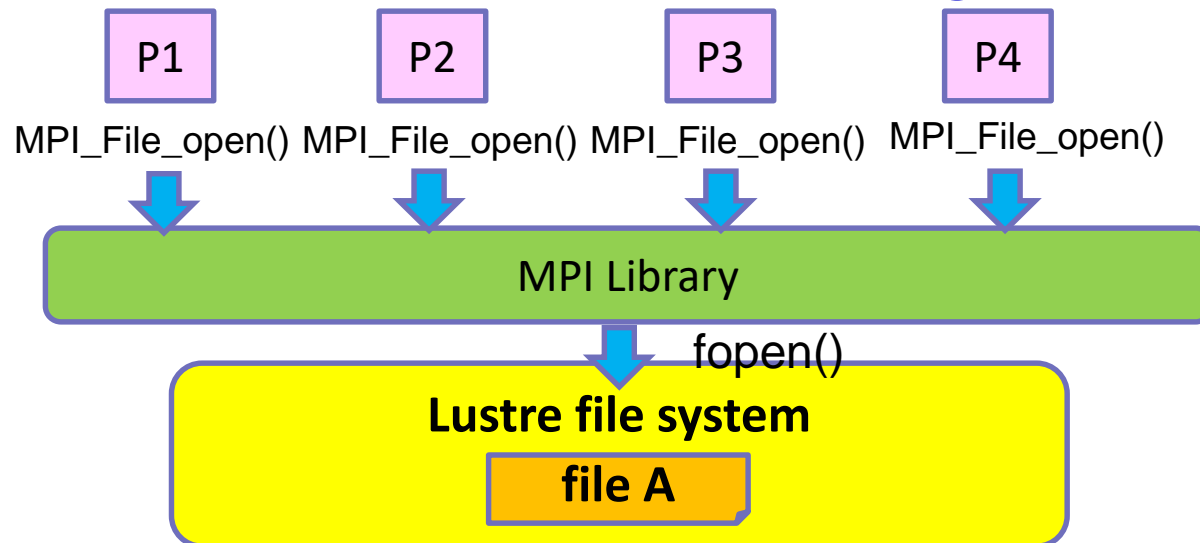
- The same is opened by each processes → multiple file handlers across your MPI processes
- Open the same file with read permission is OK
- But can't open with **write permission** together due file system **locking** mechanism → data inconsistency
- To write simultaneously must **create multiple files** (can't take advantage of parallel file system & hard to manage)



# MPI-IO File Access Operations

## ■ MPI-IO call “MPI\_File\_open()”

- File is **opened only once** in a **collective** manner
- MPI library will share and synchronize with each other to use the same file handler
- Can handle both read and write together

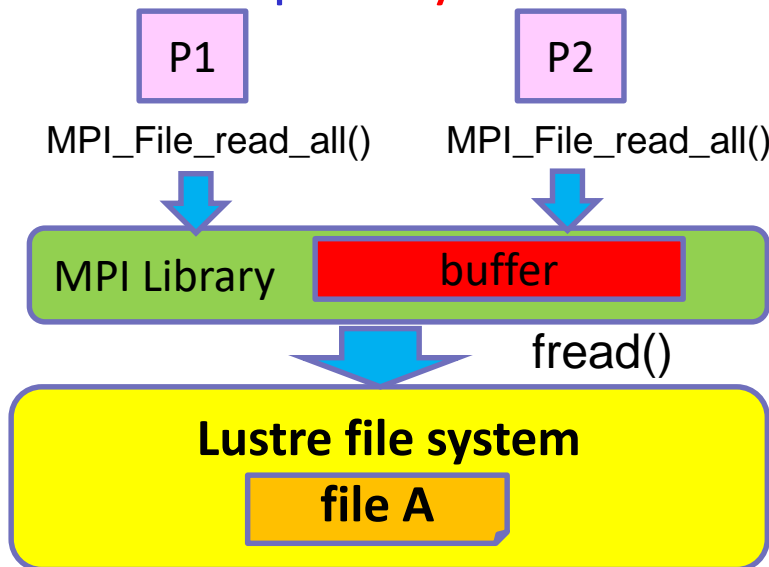




# MPI-IO Independent/Collective I/O

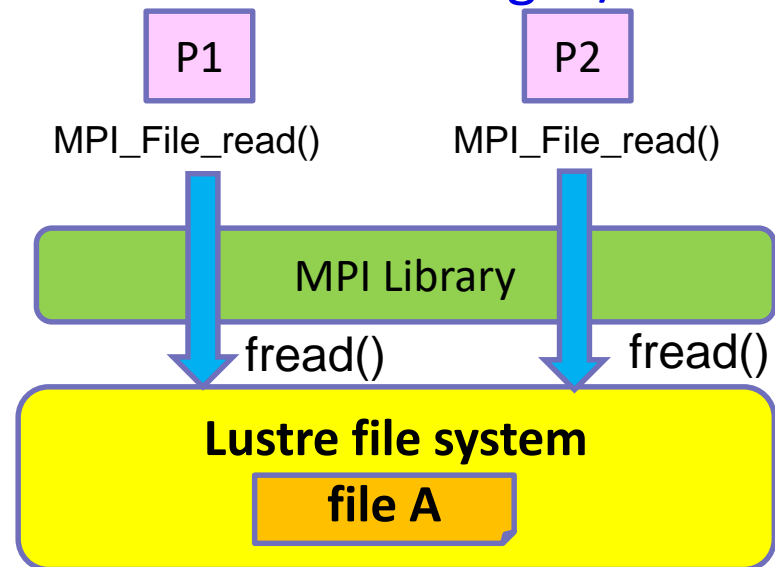
## ■ Collective I/O

- Read/write to a shared memory buffer, then issue **ONE file request**
- Reduce #I/O request  
➔ Good for small I/O
- Require **synchronization**



## ■ Independent I/O

- Read/write individually
- Prevent synchronization
- One request per process
- Request is **serialized if access the same OST**
- Good for large I/O



# MPI-IO API

- **MPI\_File\_open(MPI\_Comm comm, char \*filename, int amode, MPI\_Info info, MPI\_File \*fh)**
  - Open a file
- **MPI\_File\_close(MPI\_File \*fh)**
  - Close a file
- **MPI\_File\_read/write(MPI\_File fh, void \*buf, int count, MPI\_Datatype datatype, MPI\_Status \*status)**
  - **Independent** read/write using individual file pointer
- **MPI\_File\_read/write\_all(MPI\_File fh, void \*buf, int count, MPI\_Datatype datatype, MPI\_Status \*status)**
  - **Collectively** read/write using individual file pointer
- **MPI\_File\_sync(MPI\_File fh)**
  - **Flush** all previous writes to the storage device

# Reference

- Textbook:

- Parallel Computing Chap2

- MPI Tutorial:

- <https://computing.llnl.gov/tutorials/mpi/>

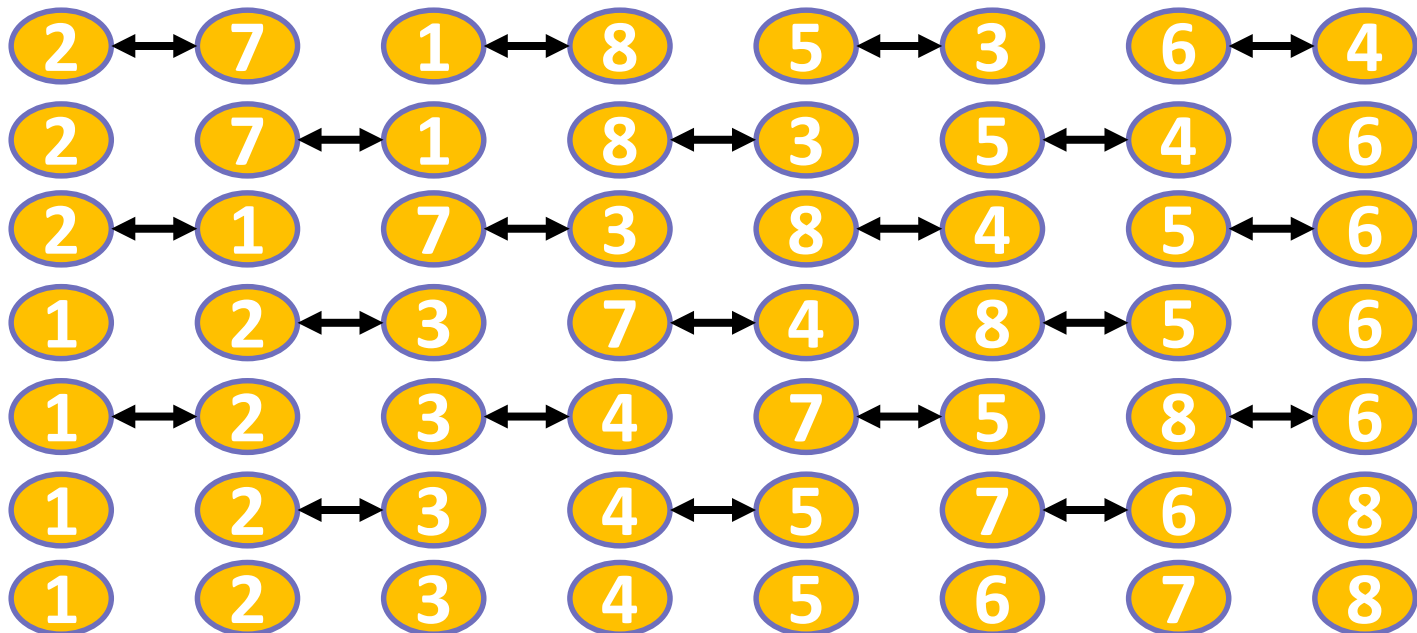
- MPI API:

- <http://www.mcs.anl.gov/research/projects/mpi/www/www3/>

# HW1: Odd-Even Sort

## ■ Algo:

- comparing & switch in order between all (odd, even)-indexed pairs of adjacent elements in the list
- comparing & switch in order between all (even, odd)-indexed pairs of adjacent elements in the list
- Repeat until the list is sorted



# HW1: Odd-Even Sort

## ■ Sequential code:

```
/* Assumes a is an array of values to be sorted. */  
var sorted = false;  
while(!sorted) {  
    sorted=true;  
    for(var i = 1; i < list.length-1; i += 2) {  
        if(a[i] > a[i+1]) { swap(a, i, i+1); sorted = false; }  
    }  
    for(var i = 0; i < list.length-1; i += 2) {  
        if(a[i] > a[i+1]) { swap(a, i, i+1); sorted = false; }  
    }  
}
```

# HW1: Odd-Even Sort

## ■ Parallel Code:

1. For each process with odd rank  $P$ , send its number to the process with rank  $P-1$ .
2. For each process with rank  $P-1$ , compare its number with the number sent by the process with rank  $P$  and send the larger one back to the process with rank  $P$ .
3. For each process with even rank  $Q$ , send its number to the process with rank  $Q-1$ .
4. For each process with rank  $Q-1$ , compare its number with the number sent by the process with rank  $Q$  and send the larger one back to the process with rank  $Q$ .
5. Repeat 1-4 until the numbers are sorted.