

Introduction to Code Parallelization Using MPI

Marinus Pennings

April 3, 2020

Original slides created by **Ping Luo**

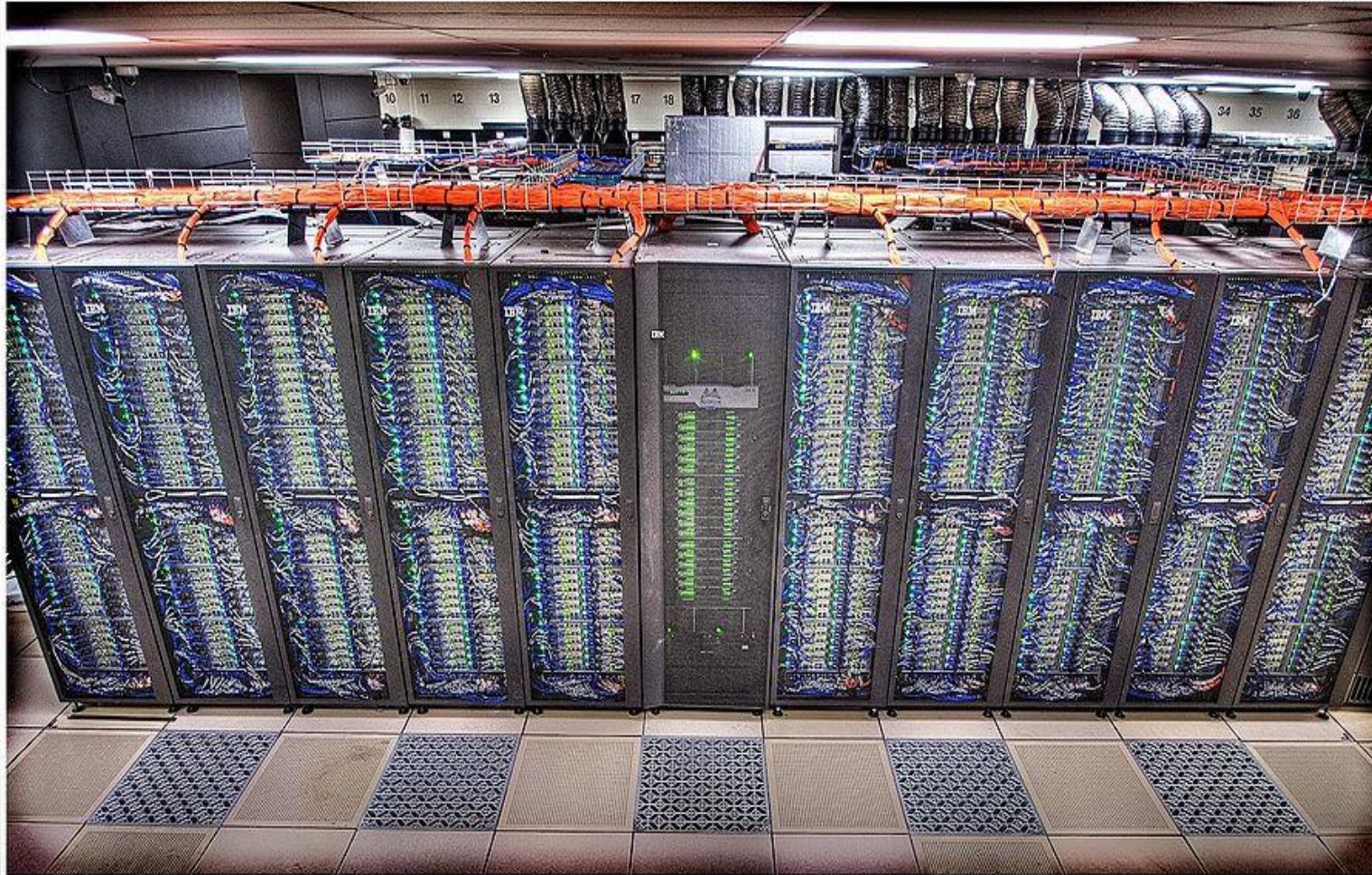
Outline

- Parallel programming models
- Layout of an MPI program
- Compiling and running an MPI program
- Basic MPI concepts
- Point-to-point communication
- Collective communication
- bonus topics/considerations (time permitting)

To Setup shortcourse examples, type: [/scratch/training/mpi/setup.sh](#)

(either terra or ada)

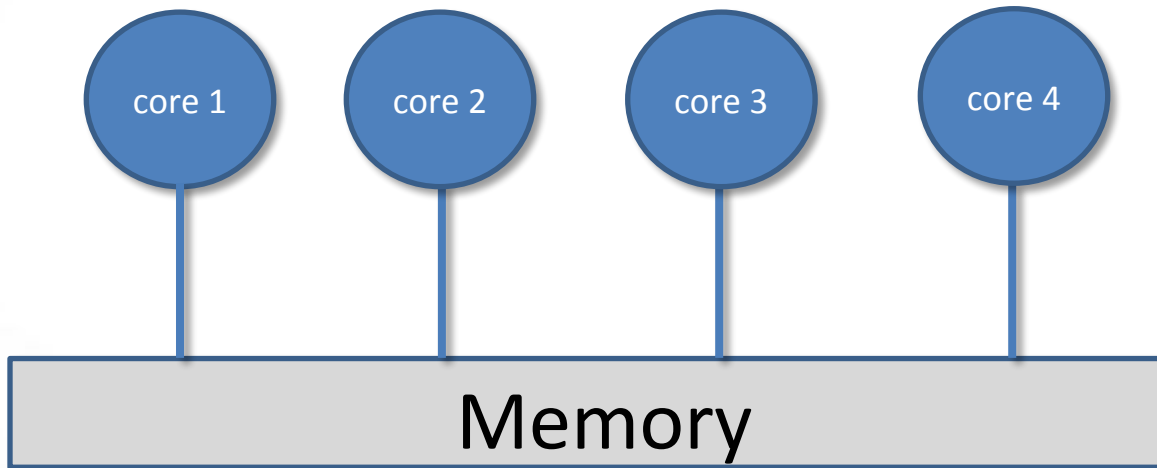
HPRC Systems Summary



- terra has 304 compute nodes and each node has 28 cores.
- ada has 852 nodes and each node has 20 cores.

Parallel Programming Models

- **Shared Memory System** – an abstraction of a parallel system where all processors share the same memory subsystem



Examples:

- single terra/ada node
- your desktop/laptop

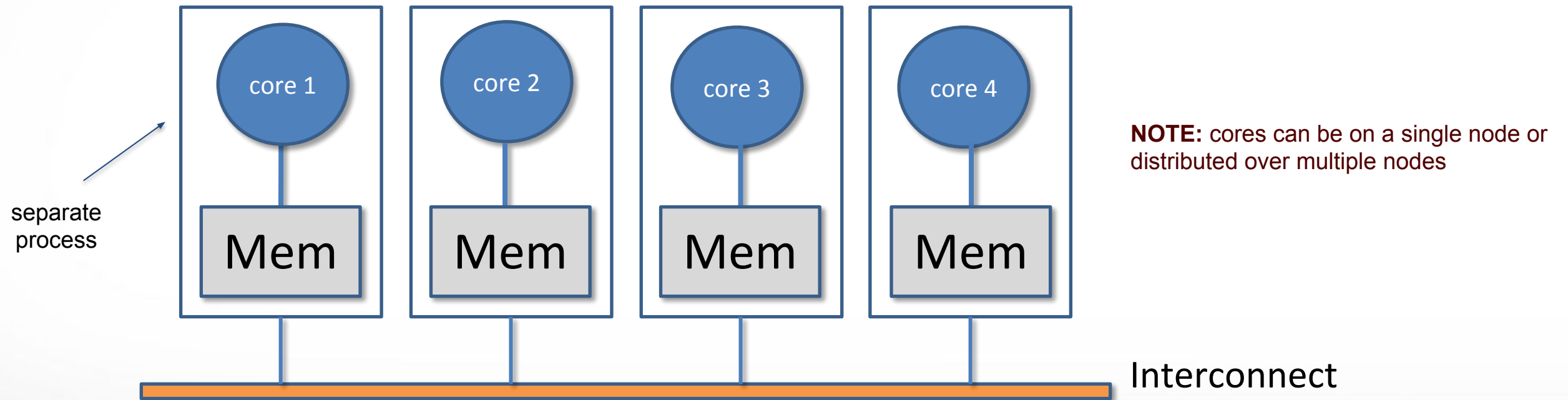
NOTE: every node on terra has 28 cores (20 on ada).
Compared to average desktop with 4 - 8 cores

Programming model for shared memory Systems: OpenMP

(OpenMP is most popular programming model for shared memory parallelism, another example is pthreads)

Parallel Programming Models

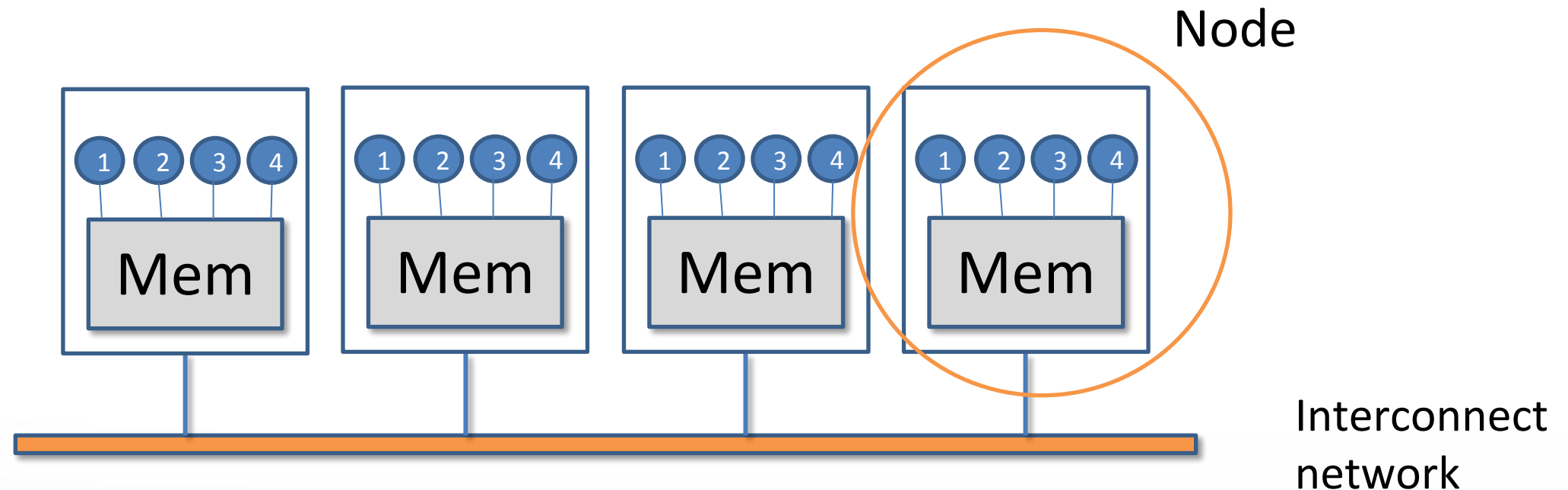
- **Distributed Memory System** – an abstraction of a parallel system where each processor has its own local private memory



Programming model for distributed memory Systems: MPI

Parallel Programming models

- Hybrid model (will discuss later)



Hybrid Programming model: MPI + OpenMP

(for example: one MPI task per node with 4 OpenMP threads per MPI task)

What is MPI?

- **M**essage **P**assing **I**nterface: a specification for the library interface that implements message passing in parallel programming.
- Is standardized by the MPI forum for implementing portable, flexible, and reliable codes for **distributed memory systems**, regardless of the architecture underneath.
 - First edition: MPI-1(1994)
 - Evolving over time: MPI-2(1998), MPI-3(2012), MPI-3.1(2015)
 - MPI-4.0 is under discussion
- Has C/C++/Fortran bindings.
 - C++ binding deprecated since MPI-2.2
- Different implementations (libraries) available: **Intel MPI**, MPICH, OpenMPI, etc.
- It is the most widely used parallel programming paradigm for large scale scientific computing.

Example 1: Hello World (C/C++)

Serial C Code

```
#include <stdlib.h>  
  
int main(int argc, char **argv){  
  
    printf("Hello, world\n");  
  
}
```



Parallel C Code Using MPI

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


Example 1: Hello World (Fortran)

Serial Fortran Code

```
program hello  
implicit none  
  
print *, "Hello, world"  
  
end program hello
```



Parallel Fortran Code using MPI

```
program hello  
use mpi  
implicit none  
  
call MPI_INIT(ierr)  
print *, "Hello, world"  
call MPI_Finalize(ierr)  
end program hello
```

demo example1-hello_world.
How to compile?

Compiling and Linking MPI Programs

We will use Intel MPI implementation in the examples

(Using Intel compiler underneath)

```
mpiicc   prog.c   [flags] -o prog.exe   (C)
mpicpc   prog.cpp [flags] -o prog.exe   (C++)
mpiifort prog.f   [flags] -o prog.exe   (Fortran)
```

(Using GNU compilers underneath)

```
mpicc    prog.c   [flags] -o prog.exe   (C)
mpicxx   prog.cpp [flags] -o prog.exe   (C++)
mpif90   prog.f   [flags] -o prog.exe   (Fortran)
```

don't forget to load an mpi toolchain. Will use intel/2019b here

demo how to compile, also try to run

Running an MPI Program

- Load the module

```
module load intel/2019b
```

- Use mpirun to run your program

```
mpirun -np n [options] prog.exe [prog_args]
```

(**n** is number of "tasks" that will be started)

- Useful MPI options

```
-ppn/-perhost, -hosts, -hostfile
```

```
example:mpirun -np 4 -hosts login1,login2 -ppn 1 mympi.exe
```

(**NOTE:** some flags are specific for each MPI implementation)

how are tasks distributed?



NOTE: When testing always try on single node and multiple nodes

https://hprc.tamu.edu/wiki/Ada:Compile:All#MPI_Programs

run examples, single/multiple node, also use hostname

Running MPI Program in Batch

ada

```
#BSUB -J MPIBatchExample
#BSUB -L /bin/bash
#BSUB -W 24:00
#BSUB -n 40
#BSUB -R "span[ptile=20]"
#BSUB -R "rusage[mem=2560]"
#BSUB -M 2560
#BSUB -o MPIBatchExample.%J

module load intel/2019b
mpirun prog.exe
```

terra

```
#!/bin/bash
#SBATCH --export=NONE
#SBATCH --get-user-env=L
#SBATCH --job-name=MPIBatchExample
#SBATCH --time=24:00:00
#SBATCH --ntasks=56
#SBATCH --ntasks-per-node=28
#SBATCH --mem=56000M
#SBATCH --output=MPIBatchExample.%j

module load intel/2019b
mpirun prog.exe
```

No need to specify number of tasks when running MPI jobs using the batch system?

run batch job

Layout of an MPI Program

```
C
#include <mpi.h>
int main(
    int argc, char **argv)
{
    ... no mpi calls
    MPI_Init(&argc, &argv);

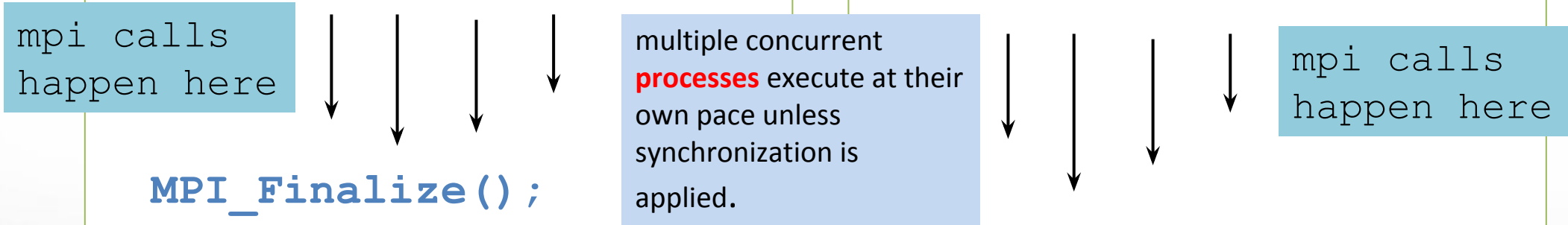
    mpi calls
    happen here

    MPI_Finalize();
    ... no mpi calls
}
```

```
Fortran
PROGRAM SAMPLE1
USE MPI !F90
!f77: include "mpif.h"
integer ierr
... no mpi calls
CALL MPI_INIT(ierr)

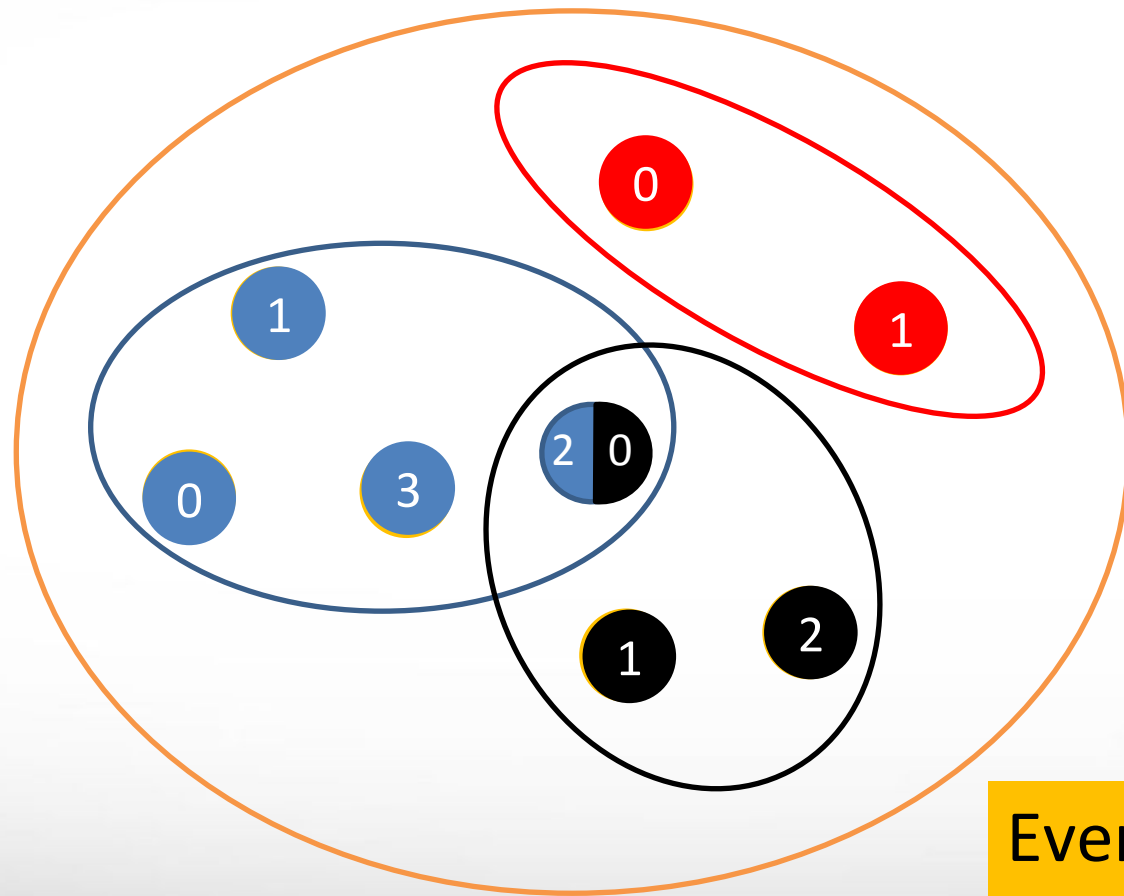
mpi calls
happen here

CALL MPI_FINALIZE(ierr)
... no mpi calls
END PROGRAM SAMPLE1
```



Basic MPI Concept: Communicator

communicator, size, rank



Communicator: MPI_COMM_WORLD

Size: 8

Rank: 0, 1, ..., 7

Communicator: comm1

Size: 2

Rank: 0, 1

Communicator: comm2

Size: 4

Rank: 0, 1, 2, 3

Communicator: comm3

Size: 3

Rank: 0, 1, 2

Every MPI communication must specify a communicator.

Communicator

- In MPI, a communicator is a software structure through which we specify a group of processes.
- Each process inside a communicator is assigned a unique **rank** (an integer) ranging from 0 to (group_size - 1). group_size is the **size** of the communicator.
- The constant **MPI_COMM_WORLD** (obtained from MPI include file) is an initial communicator that includes all the MPI processes activated in the running of a program. MPI_COMM_WORLD is the most common communicator.
- Communicators are especially useful in characterizing the tasks that different groups of processes carry out.

Size and Rank

- How many processes in a communicator?

C	<code>int MPI_Comm_size(MPI_Comm comm, int *size)</code>
Fortran	<code>SUBROUTINE MPI_COMM_SIZE(comm, size, ierr)</code> <code>integer comm, size, ierr</code>

- What's the rank (id) of each process in a communicator?

C	<code>int MPI_Comm_rank(MPI_Comm comm, int *rank)</code>
Fortran	<code>SUBROUTINE MPI_COMM_RANK(comm, rank, ierr)</code> <code>integer comm, rank, ierr</code>

Example: Hello world (part 2)

C

```
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv){
    int np, rank, number;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    fprintf("Hello from task %d out of %d tasks",rank,np);
    MPI_Finalize();
}
```

Fortran

```
program simple
use mpi
implicit none
integer ierr, np, rank, number, status ;

call MPI_INIT(ierr)

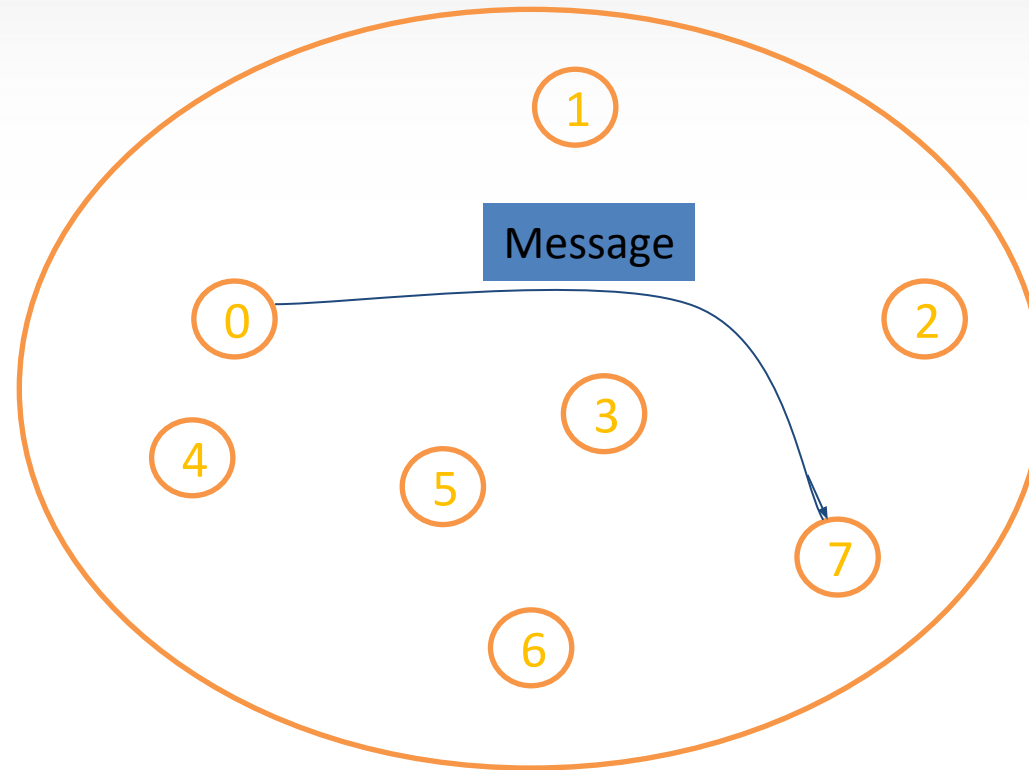
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

print *, "hello from task ", rank, "out of ", np, " tasks"

call MPI_Finalize(ierr)
end program simple
```

exercise ex1

Point-to-Point Communication



- Blocking `MPI_Send, MPI_Recv`
- Non-blocking `MPI_Isend, MPI_Irecv`
- Send-Receive `MPI_Sendrecv`

Blocking Send

```
C int MPI_Send(void *buf, int count, MPI_Datatype
           datatype, int dest, int tag, MPI_Comm comm)
```

```
Fortran MPI_SEND(buf, count, datatype, dest, tag, comm, ierr)
         <type> buf(*)
         integer count, datatype, dest, tag, comm, ierr
```

buf	starting address of send buffer
count	number of elements in send buffer
datatype	datatype of each send buffer element
dest	rank of destination
tag	message tag
comm	communicator

Blocking Receive

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

Fortran `MPI_RECV(buf, count, datatype, source, tag, comm, status, ierr)`
`<type> buf(*)`
`integer count, datatype, source, tag, comm, ierr, status[MPI_STATUS_SIZE]`

<code>buf</code>	initial address of receive buffer
<code>count</code>	number of elements in receive buffer
<code>datatype</code>	datatype of each receive buffer element
<code>source</code>	rank of source or <code>MPI_ANY_SOURCE</code>
<code>tag</code>	message tag or <code>MPI_ANY_TAG</code>
<code>comm</code>	communicator
<code>status</code>	status object

`MPI_ANY_SOURCE` and `MPI_ANY_TAG` are MPI defined wildcards.

Example 2 – One Sender and One Receiver

C

```
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv){
    int np, rank, number;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0){
        number = 1234;
        MPI_Send(&number, 1, MPI_INT, 1, 0,
                MPI_COMM_WORLD);
        printf("Process %d sends %d to process 1\n", rank,
number);
    }else if(rank == 1){
        MPI_Recv(&number, 1, MPI_INT, 0, 0,
                MPI_COMM_WORLD, &status);
        printf("Process %d receives %d from process 0\n",
rank,number);
    }
    MPI_Finalize();
}
```

Fortran

```
program simple
use mpi
implicit none
integer ierr, np, rank, number, status ;
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, np, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
if (rank == 0) then
    number = 1234
    call MPI_SEND(number, 1, MPI_INTEGER, 1, 0,
                  MPI_COMM_WORLD, ierr)
    print *, "process ", rank, " sends ", number
else if (rank == 1) then
    call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD,
                  status, ierr)
    print *, "process ",rank," receives ", number
endif
call MPI_Finalize(ierr)
end program simple
```

example2-send_receive



Send and Receive a Message

```
program simple
use mpi
implicit none
integer ierr, np, rank, number, status ;
```

Data

Envelope

Destination

```
MPI_SEND(number, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)
MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)
```

```
if (rank == 0) then
number = 1234
call MPI_SEND(number, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)
print *, "process ", rank, " sends ", number
else if (rank == 1) then
call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)
print *, "process ", rank, " receives ", number
endif

call MPI_Finalize(ierr)
end program simpleC
```

Buffer
starting
address

Data

MPI

source

tag

communicator

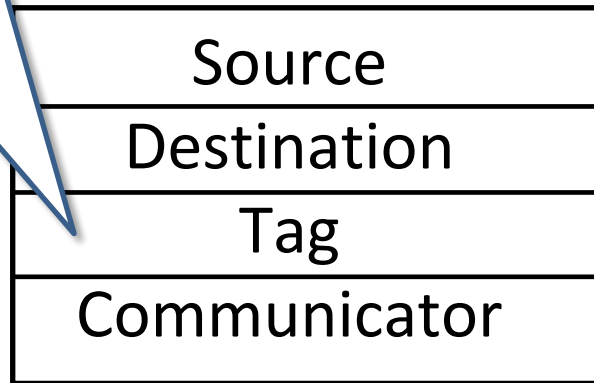
count

datatype

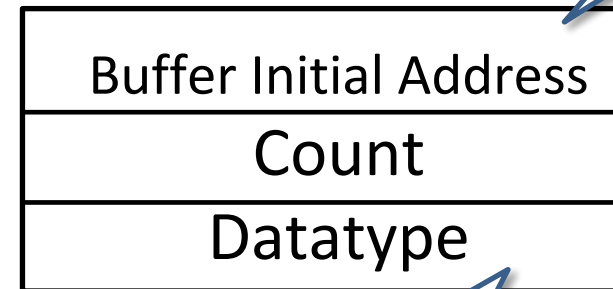
Message

Tag is an integer used in a message to differentiate one message from other messages.

Envelope



Data



Where to fetch/store the data

Type of the data to be sent or received. Datatype can be predefined or user defined.
Commonly used predefined datatypes, also called MPI basic datatypes in Fortran:
MPI_INTEGER, MPI_REAL, MPI_REAL8,
MPI_CHARACTER, MPI_LOGICAL

Comments on Blocking Send

`MPI_SEND(buf, count, datatype, dest, tag, comm)`

- The calling process causes `count` many contiguous elements of type `datatype` to be sent, starting from `buf`.
- The message sent by `MPI_SEND` can be received by either `MPI_RECV` or `MPI_Irecv`.
- `MPI_SEND` doesn't return (i.e., `blocked`) until it is safe to write to the send buffer.
 - Safe means the message has been copied either into a system buffer, or into the receiver's buffer, depending on which mode the send call is currently working under.

Comments on Blocking Receive

`MPI_RECV (buf, count, datatype, source, tag, comm, status)`

- The calling process attempts to receive a message with specified envelope (source, tag, communicator).
 - `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are valid values.
- When the matching message arrives, elements of the specified `datatype` are placed in the buffer in contiguous locations, starting at the address of `buf`.
- The buffer starting at `buf` is assumed pre-allocated and has capacity for at least `count` many `datatype` elements.
 - An error returns if `buf` is smaller than data received.

Comments on Blocking Receive

`MPI_RECV (buf, count, datatype, source, tag, comm, status)`

- MPI_RECV can receive a message send by MPI_SEND or MPI_ISEND.
- Agreement in `datatype` between the send and receive is required.
- MPI_RECV is `blocked` until the message has been copied into `buf`.
- The actual size of the message received can be extracted with MPI_GET_COUNT.

Return Status

- The argument `status` in `MPI_Recv` provides a way of retrieving `message source`, `message tag`, and `message error` from the message.
- `status` is useful when MPI wildcards (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`) are used in `MPI_Recv`.
- `status` can be ignored with `MPI_STATUS_IGNORE`

C

```
MPI_Status status
...
MPI_Recv(..., &status)
Source_id = status.MPI_SOURCE
tag       = status.MPI_TAG
```

Fortran

```
integer status(MPI_STATUS_SIZE)
...
CALL MPI_RECV(..., status, ierr)
source_id = status(MPI_SOURCE)
tag       = status(MPI_TAG)
```

Timing

MPI_WTIME()

- returns a floating-point number in seconds, representing elapsed wall clock time since some time in the past.

C

```
double t1, t2, elapsed;
t1 = MPI_Wtime();
...
// code segment to be timed
...
t2 = MPI_Wtime();
elapsed = t2 - t1;
```

Fortran

```
real*8 t1, t2
real*8 elapsed
t1 = MPI_WTIME()
...
! Code segment to be timed
...
t2 = MPI_WTIME()
elapsed = t2 - t1
```

Case Study: Computing Pi (1)

Monte Carlo method to compute Pi:

- assume a circle with diameter 1, enclosing box will be 1 by 1
- surface of circle = $\pi * r^2 = \pi * (1/2)^2 = \pi/4$, surface enclosing box = 1
- Generate a large number of random points (x,y)
- for every point, determine if it's inside the box using $\sqrt{x^2+y^2} \leq 1$, and count
- fraction $\#inside / \#total = (\pi/4)/1 \rightarrow \pi = (4*\#inside)/\#total$

First Attempt

1. Root distributes large array of random points among all tasks
2. every task computes $\#points$ are inside the circle
3. every task sends results back to root
4. root combines all the results and computes pi

Non-blocking Send

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer
IN	count	number of elements in send buffer (non-negative integer)
IN	datatype	datatype of each send buffer element
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator
OUT	request	communication request (a handle that can be used later to refer the outstanding receive)

Non-blocking Receive

MPI_IRecv(buf, count, datatype, source, tag, comm, request)

IN	buf	initial address of send buffer
IN	count	number of elements in receive buffer (non-negative integer)
IN	datatype	datatype of each receive buffer element
IN	source	rank of source or MPI_ANY_SOURCE
IN	tag	message tag or MPI_ANY_TAG
IN	comm	communicator
OUT	request	communication request (a handle that can be used later to refer the outstanding receive)

Auxiliary Routines for Non-blocking Send/Receive

- Auxiliary routines are used to complete a non-blocking communication or communications.
- Commonly used auxiliary routines:

MPI_Wait(<i>request</i> , status)	The calling process waits for the completion of a non-blocking send/receive identified by <i>request</i> .
MPI_Waitall(count, <i>requests</i> , statuses)	The calling process waits for all pending operations in a list of <i>requests</i> .
MPI_Test(<i>request</i> , flag, status)	The calling process tests a non-blocking send/receive specified by <i>request</i> has completed delivery/receipt of a message.

Non-blocking Send/Receive

- A non-blocking send/receive call initiates the send/receive operation, and **returns immediately** with a request handle, before the message is copied out/into the send/receive buffer.
- A **separate send/receive complete call** is needed to complete the communication before the buffer can be accessed again.
- A non-blocking send can be matched by a blocking receive; a non-blocking receive can be matched by a blocking send.
- Used correctly, non-blocking send/receive can improve program performance.
- They also make the point-to-point transfers “safer” by not depending on the size of the system buffers.
 - No deadlock caused by unavailable buffer
 - No buffer overflow

MPI_WAIT

MPI_WAIT(request, status)

request	request (handle)
status	status object (Status)

C

```
MPI_Request request;  
MPI_Status status;  
...  
MPI_Irecv(recv_buf, count, ...,  
          comm, &request);  
  
//do some computations ...  
  
MPI_Wait(&request, &status);
```

Fortran

```
integer request  
integer status(MPI_STATUS_SIZE)  
...  
call MPI_Irecv(recv_buf, count, ...&  
              comm, request, ierr)  
  
!do some computations ...  
  
call MPI_WAIT(request, status, ierr)
```

status can be ignored with MPI_STATUS_IGNORE

MPI_WAITALL

MPI_WAITALL(count, requests, statuses)

count lists length (non-negative integer)
requests array of requests (array of handles)
statuses array of status objects (array of Status)

Fortran

```
integer reqs(4)
integer statuses(MPI_STATUS_SIZE,4)
...
call MPI_ISEND(..., reqs(1), ierr)
call MPI_IRECV(..., reqs(2), ierr)
call MPI_ISEND(..., reqs(3), ierr)
call MPI_IRECV(..., reqs(4), ierr)
...
... do some computations ...
...
call
MPI_WAITALL(4, reqs, statuses, ierr)
```

C

```
MPI_Request reqs[4];
MPI_Status  status[4];
...
MPI_Isend(..., &reqs[0]);
MPI_Irecv(..., &reqs[1]);
MPI_Isend(..., &reqs[2]);
MPI_Irecv(..., &reqs[3]);
...
... do some com computations ...
...
MPI_Waitall(4, reqs, statuses);
```

`status` can be ignored with `MPI_STATUSES_IGNORE`

Example 4 (non blocking send)

C

```
MPI_Request *requests;
....
if (rank == 0){
    printf("Type any number from the input: ");
    scanf("%d", &number);
    requests = (MPI_Request
*)(malloc(npof(MPI_Request)*(np-1)));

    for (i=1; i<np; i++)
        MPI_Isend(&number, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                  &requests[i-1]);
    MPI_Waitall(np-1, requests, MPI_STATUSES_IGNORE);
    free(requests);
}else{
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("My id is %d. I received %d\n", rank, number);
}
```

Fortran

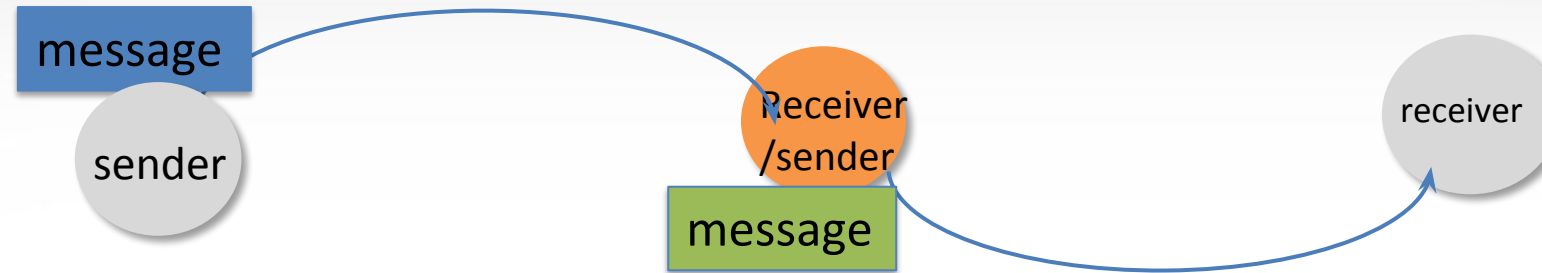
```
integer, allocatable::requests(:)
....
if (rank == 0) then
    print *, "Type an integer from the input"
    read *, number
    allocate(requests(np-1))
    do i=1, np-1
        call MPI_ISEND(number, 1, MPI_INTEGER, i, 0, &
                       MPI_COMM_WORLD, ierr)
    enddo
    call MPI_WAITALL(np-1, requests, MPI_STATUSES_IGNORE, ierr)
    deallocate(requests)
else
    call MPI_RECV(number, 1, MPI_INTEGER, 0, 0, &
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE, ierr)
    print "(2(A,I6))", "Process ", rank, " received ", number
endif
```

example4-non_blocking_send_receive

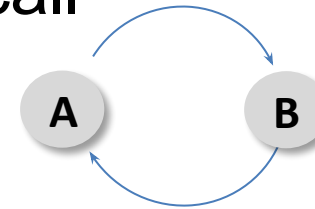


Send-Receive

`MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)`



- Combines send and receive operations in one call
- The source and destination can be the same.
- The message sent out by send-receive can be received by blocking/non-blocking receive or another send-receive
- It can receive a message sent by blocking/non-blocking send or another send-receive.
- Useful for executing a **shift operation** across a chain of processes.

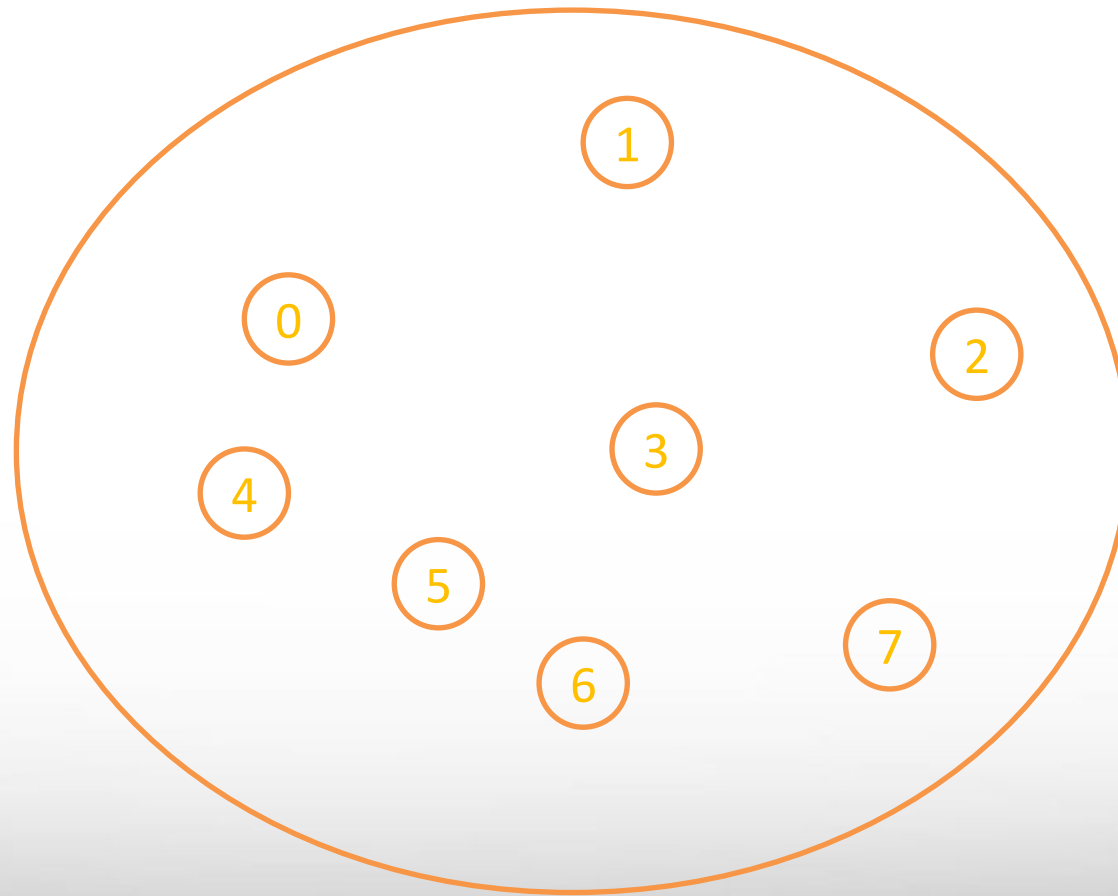


- Dependencies will be taken care of by the communication subsystem to eliminate the possibility of deadlock.

exercise4

Collective Communication

- A collective communication refers to a communication that involves **all processes** in a communicator.



Routines for Collective Communication

MPI_BARRIER	All processes within a communicator will be blocked until all processes within the communicator have entered the call.
MPI_BCAST	Broadcasts a message from one process to members in a communicator.
MPI_REDUCE	Performs a reduction operation to the vector of elements in the sendbuf of the group members and places the result in recvbuf on root.
MPI_GATHER MPI_GATHERV	Collects data from the sendbuf of all processes in comm and place them consecutively to the recvbuf on root based on their process rank.
MPI_SCATTER MPI_SCATTERV	Distribute data in sendbuf on root to recvbuf on all processes in comm.
MPI_ALLREDUCE	Same as MPI_REDUCE, except the result is placed in recvbuf on all members in a communicator.
MPI_ALLGATHER MPI_ALLGATHERV	Same as GATHER/GATHERV, except now data are placed in recvbuf on all processes in comm.

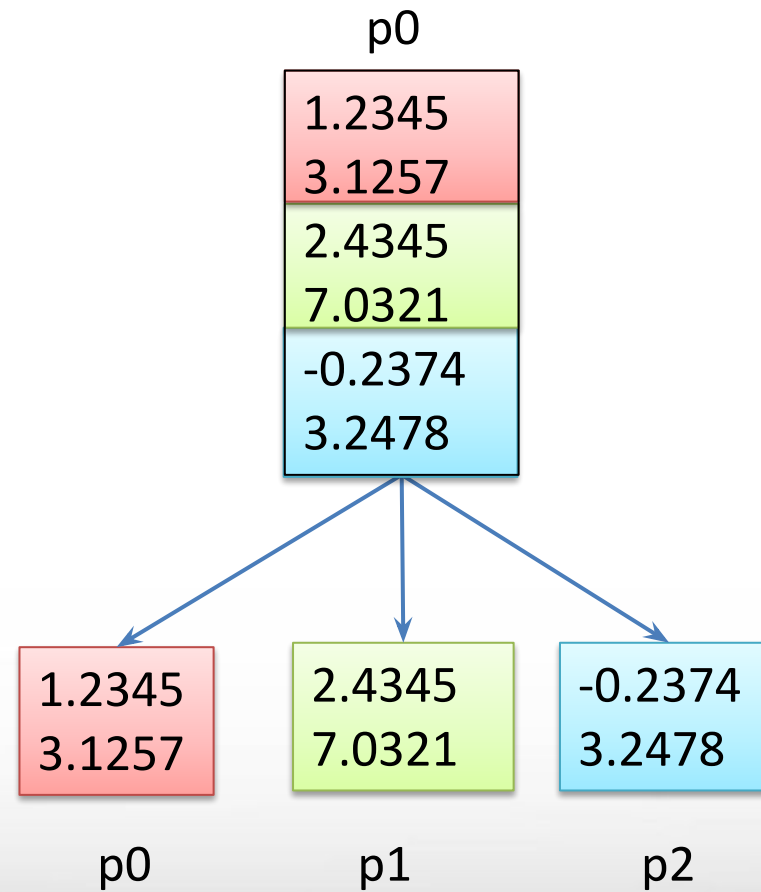
MPI_BARRIER

MPI_BARRIER(comm)

- Blocks all processes in **comm** until all processes have called it.
- Is used to synchronize the progress of all processes in **comm**.

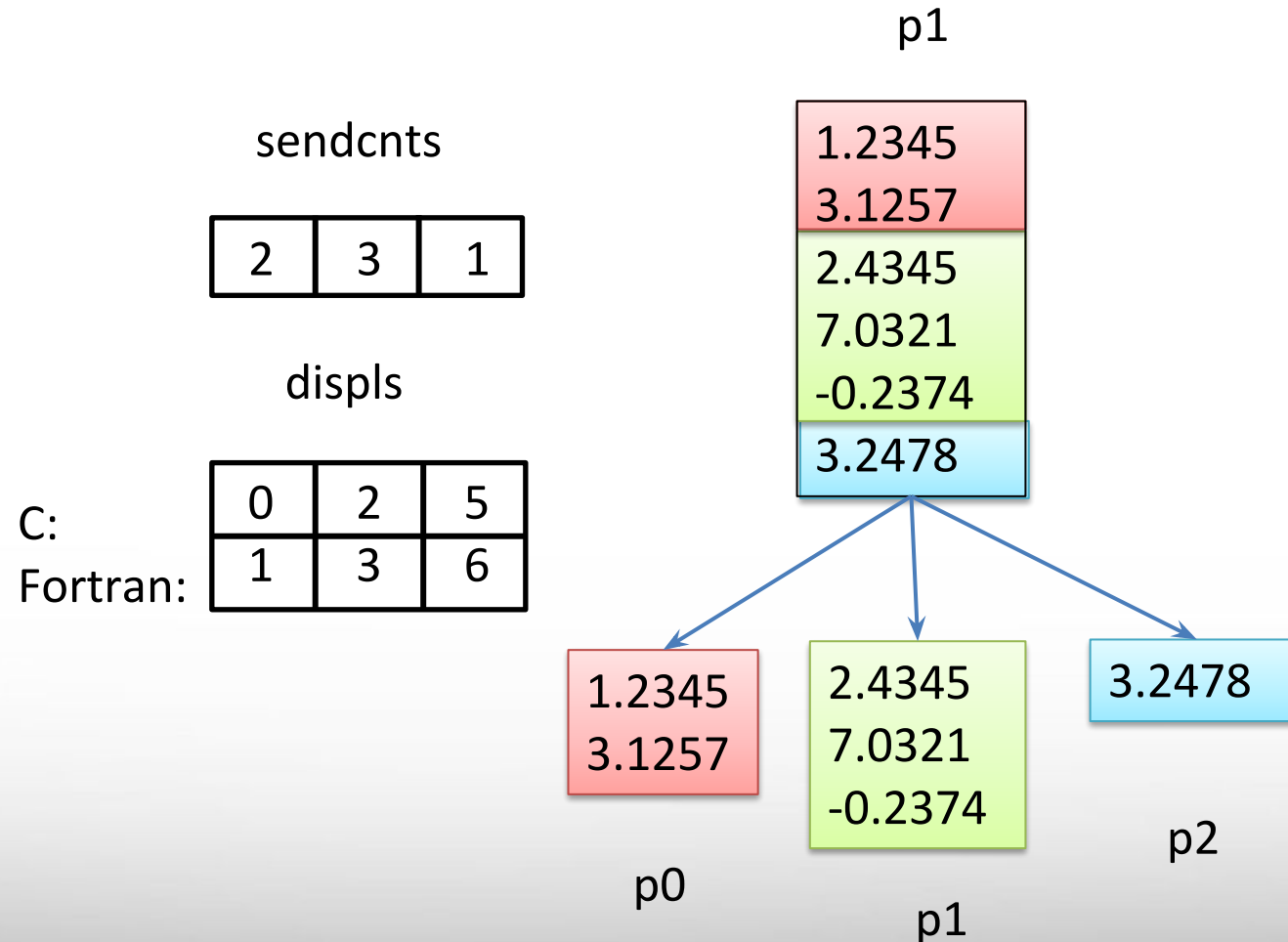
MPI_SCATTER

`MPI_SCATTER(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)`



MPI_SCATTERV

MPI_SCATTERV(sendbuf, **sendcnts**, **displs**, sendtype, recvbuf, recvcnt, recvtype, root, comm)

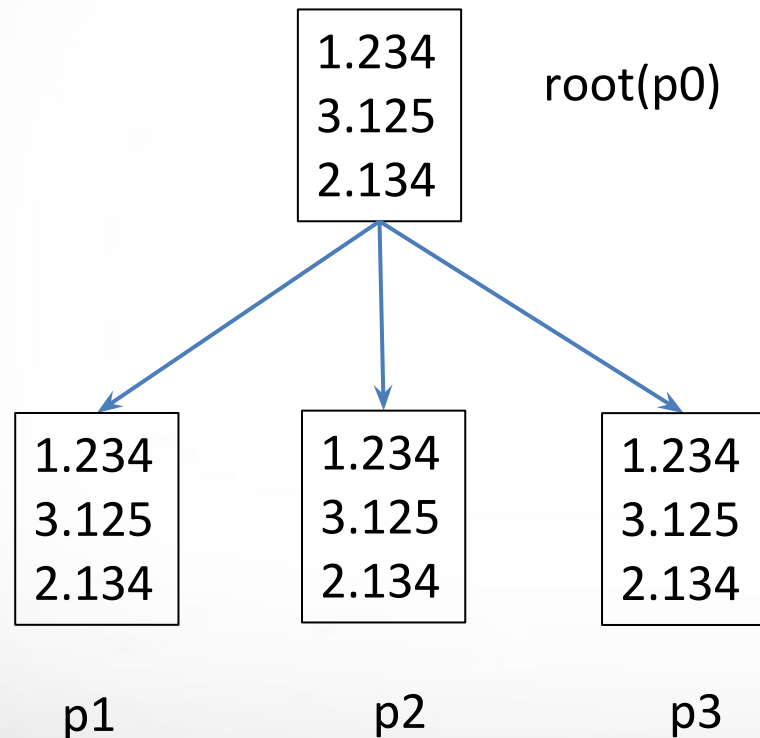


Case Study: Computing Pi (2)

In the first version, the root distributed all the random points one by one among all the tasks. In this exercise, instead of sending all the random points, we will use the `MPI_SCATTER` function to distribute the input

MPI_BCAST

MPI_BCAST(buffer, count, datatype, root, comm)



- Root process: sends a message to all processes (including root) in the communicator **comm**.
- Non-root processes: receives a message from the specified **root**.
- Each receiving process blocks until the message has arrived its **buffer**.
- All processes in **comm** must call this routine.

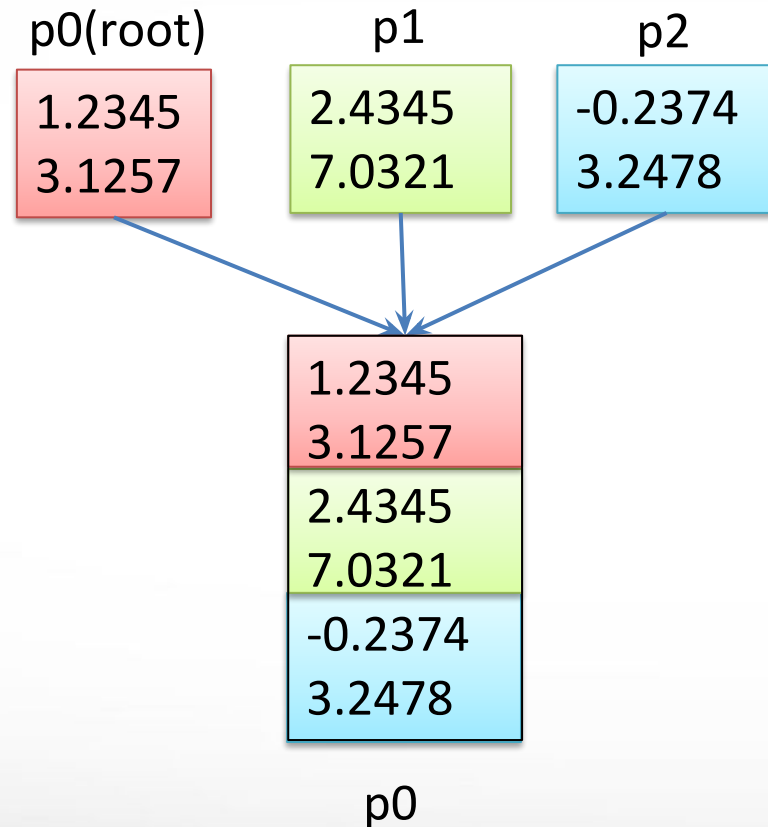
Case Study: Computing Pi (3)

Lets adjust the `compute_pi` program. Instead of sending the random numbers (or total number of points) we will broadcast the value. Every task will generate the random points.

This also shows impact of communication

MPI_GATHER

`MPI_GATHER(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, root, comm)`



- Gathers together data from all process in `comm` and stores in `root` process.
- Data received by root are stored in rank order.
- `recvcnt` is number of elements received per process
- `Recvbuf`, `recvcnt`, `recvtype` are significant only at root.

MPI_GATHERV

MPI_GATHERV(sendbuf,sendcnt,sendtype,recvbuf,**recvcnts**,**displs**,
recvtype,root,comm)

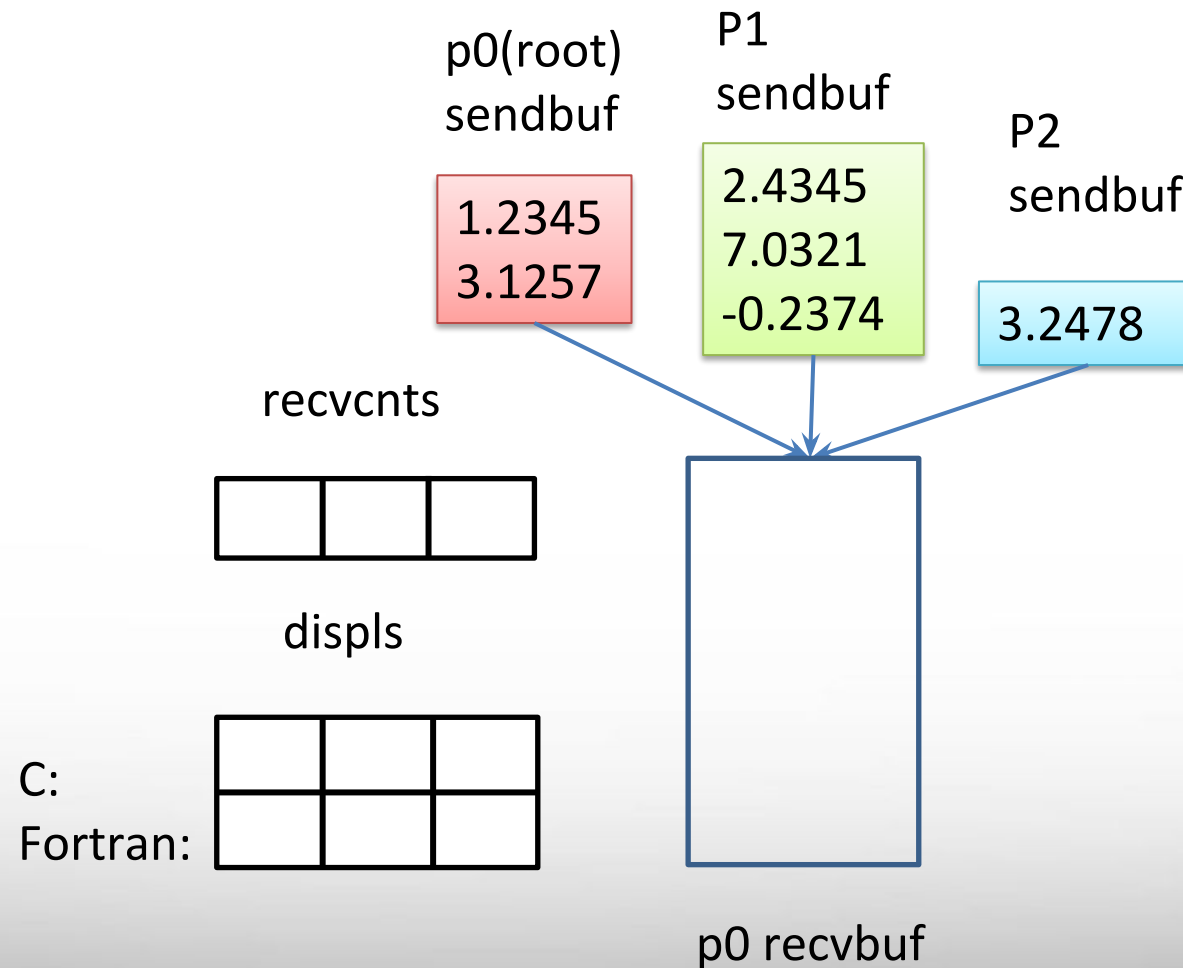
- IN recvcnts an integer array of size of *comm*. recvcnts[i] = number of elements received from process i.
- IN displs an integer array of size of *comm*. displs[i] = displacement from *recvbuf* for process i.

Fortran
integer recvcnts(*), displs(*)

C
int recvcnts[], displs[]

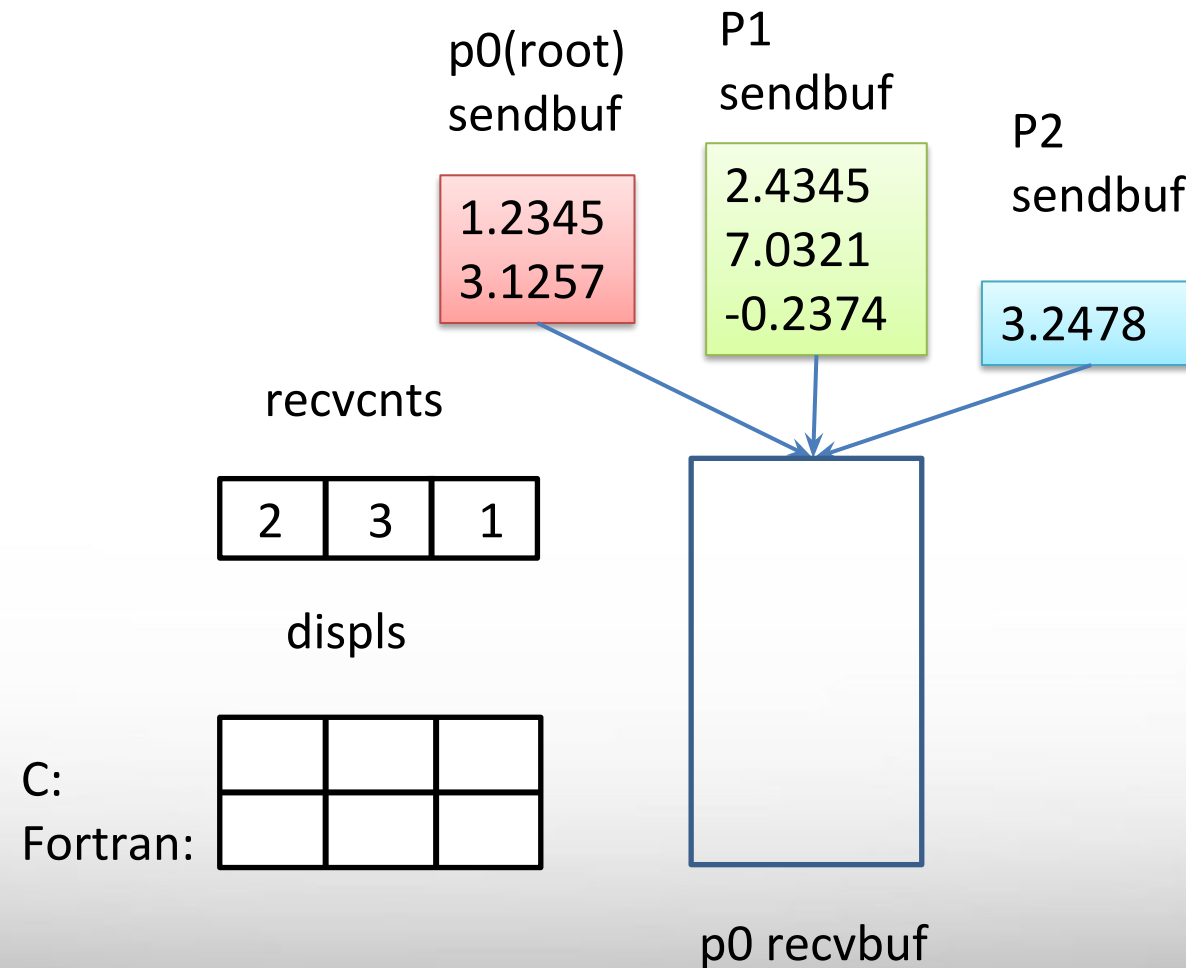
MPI_GATHERV (cont.)

MPI_GATHERV(sendbuf,sendcnt,sendtype,recvbuf,**recvcnts**,**displs**,
recvtype,root,comm)



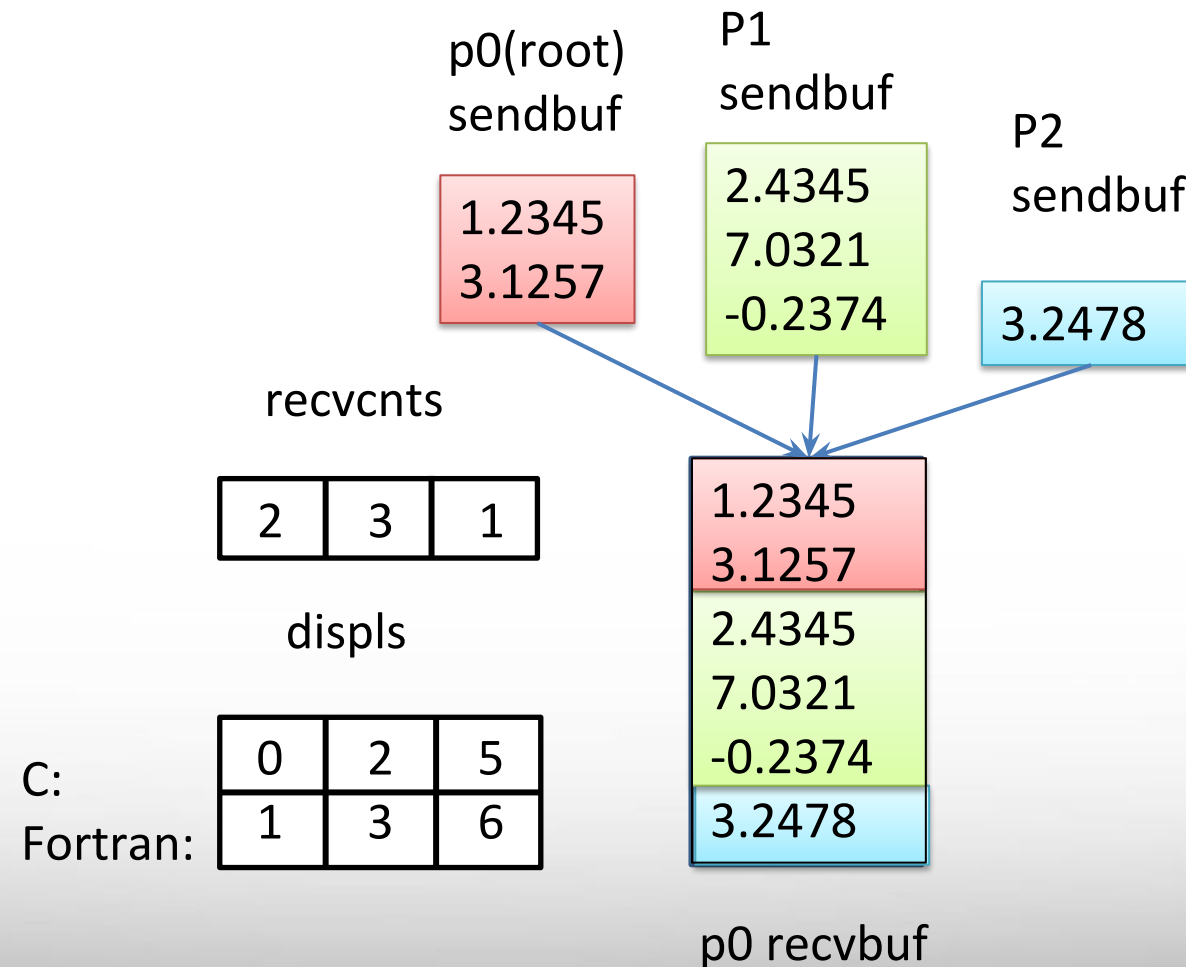
MPI_GATHERV (cont.)

MPI_GATHERV(sendbuf,sendcnt,sendtype,recvbuf,**recvcnts**,**displs**,
recvtype,root,comm)



MPI_GATHERV (cont.)

MPI_GATHERV(sendbuf,sendcnt,sendtype,recvbuf,**recvcnts**,**displs**,
recvtype,root,comm)



MPI_ALLGATHER/MPI_ALLGATHERV

`MPI_ALLGATHER(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,recvtype,comm)`

`MPI_ALLGATHERV(sendbuf,sendcnt,sendtype,recvbuf,recvcnts,displs,recvtype,comm)`

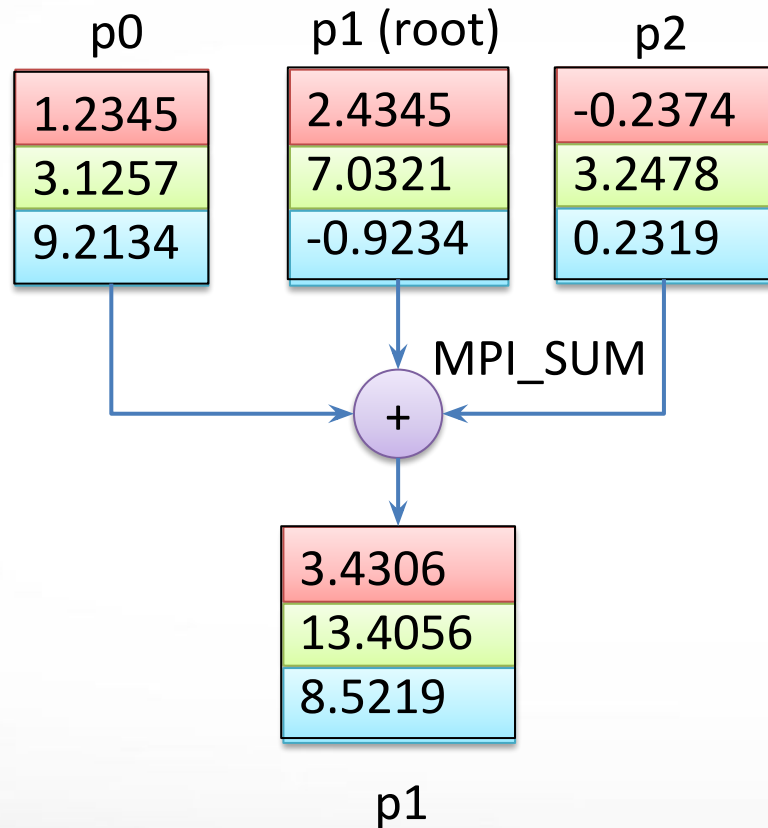
- Same as MPI_GATHER/MPI_GATHERV, except no root.
- Root is not needed since every process in the communicator stores the data gathered in its recvbuff.

Case Study: Computing Pi (4)

In the previous version, all the tasks sent their results back to the root and the root received them one by one and combined (i.e. reduced) the results. In this exercise we will use `MPI_GATHER` to collect all the results

MPI_REDUCE

`MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)`



C: MPI_Op op
Fortran: integer op

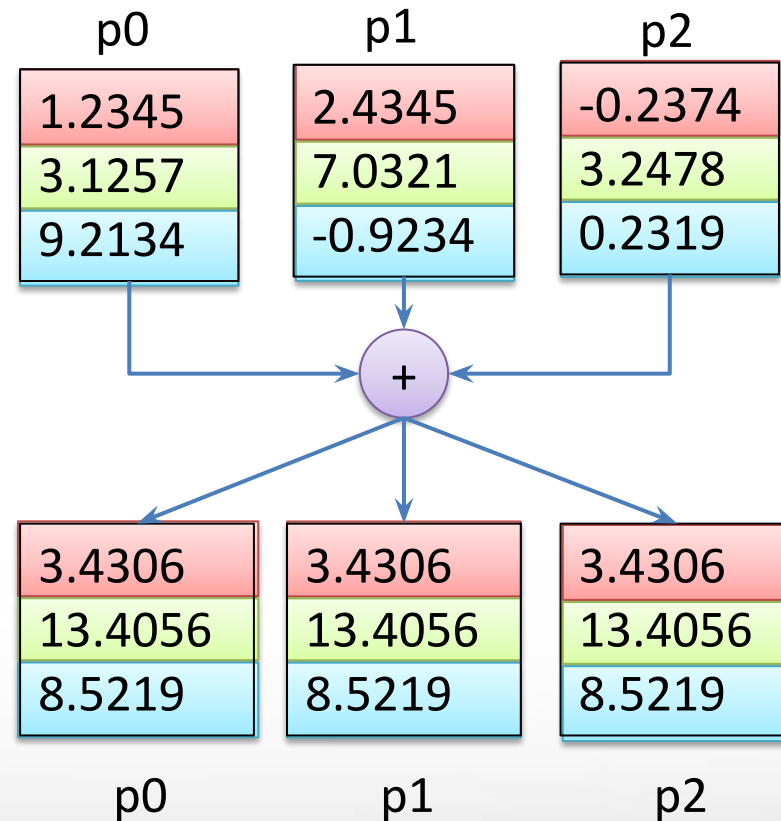
- Performs a reduction operation on all elements with same index in *sendbuf* on all processes and stores results in *recvbuf* of the root process.
- *recvbuf* is significant only at root.
- *sendbuf* and *recvbuf* cannot be the same.
- The size of *sendbuf* and *recvbuf* is equal to **count**.

Predefined Reduction Operations

PREDEFINED OPERATIONS	MPI DATATYPES
MPI_SUM, MPI_PROD	MPI_REAL8, MPI_INTEGER, MPI_COMPLEX, MPI_DOUBLE, MPI_INT, MPI_SHORT, MPI_LONG
MPI_MIN, MPI_MAX	MPI_INTEGER, MPI_REAL8, MPI_INT, MPI_SHORT, MPI_LONG, MPI_DOUBLE
MPI_LAND, MPI_LOR, MPI_LXOR	MPI_LOGICAL, MPI_INT, MPI_SHORT, MPI_LONG
MPI_BAND, MPI_BOR, MPI_BXOR	MPI_INTEGER, MPI_INT, MPI_SHORT, MPI_LONG

MPI_ALLREDUCE

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, **op**, comm)



Case Study: Computing Pi (5)

In the previous version, the root gathered the partial results and the root combined (i.e. reduced) the results manually. Now we will use `MPI_REDUCE` for the last step.

DEMO: matvec

$$A\vec{b} = (\vec{a}_1 \quad \dots \quad \vec{a}_n)\vec{b} = b_1\vec{a}_1 + b_2\vec{a}_2 + \dots + b_n\vec{a}_n = \vec{c}$$

\vec{a}_i is a column vector.

In this program, strips of consecutive columns of A are distributed to all processes. Each process carries out a part of the linear vector sum. The root will scatter the column vector, every task will compute partial sum and all partial sums will be reduced

$$b_i\vec{a}_i + \dots + b_j\vec{a}_j$$

Bonus Topics and Considerations

- Exploring parallelism
 - Task-parallelism
 - Data-parallelism
- Data Distribution
- MPI/OMP hybrid programming

Important Note On Using MPI

- All parallelism is explicit: the **programmer is responsible** for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Exploring Parallelism

- Task-parallelism: The programmer identifies different tasks of a program and distribute the tasks among different processors
- Data-parallelism: The programmer partitions the data used in a program and distribute them among different processors, each performing similar operations on the subset of data assigned.

3 chefs need to prepare a three-course menu for 12 guests

salad steak desert



Preparing
12 salads

Task 1



Preparing
12 steaks

Task 2



Preparing
12 deserts

Task 3



4 meals
salad
steak
desert



4 meals
salad
steak
desert



4 meals
salad
steak
desert

Task parallelism

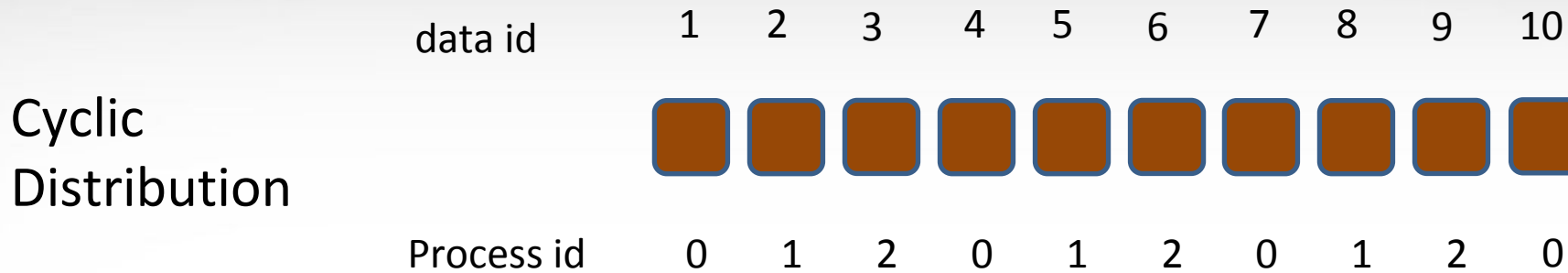
Data parallelism

<https://101.clipart.com/wp-content/uploads/02/Clipart%20Chef%20Cooking%2012.jpg>

<https://classroomclipart.com/images/gallery/Clipart/Culinary/chef-thumbs-up-holding-plate-hot-food-clipart-622.jpg>

<http://www.marbellafamilyfun.com/images/chef-and-waiter-wanted-for-summer-job-from-14th-august-for-2-weeks-21516737.jpg>

Data Distribution: cyclic



Data is distributed in a round robin manner among the processes.

C

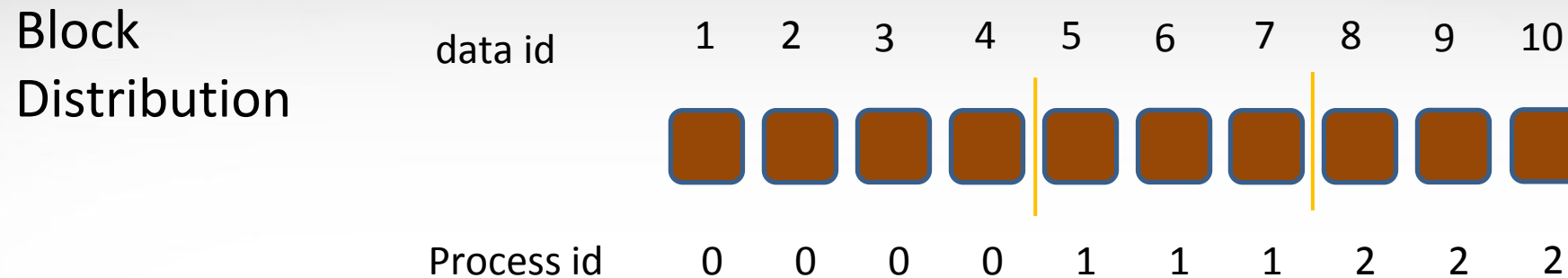
```
for (i=myid+1; i<=N; i+=nprocs){  
    x = h*(i-0.5);  
    sum += 4.0/(1.0+x*x);  
}  
sum = sum*h;
```

Fortran

```
do i=myid+1, N, nprocs  
    x = h*(i-0.5d0)  
    sum = sum+4.0d0/(1.0d0+x*x)  
enddo  
sum = sum*h;
```

example6-data_distribution/calc_PI_cyclic

Data Distribution: block



Data is partitioned into n contiguous parts, where n is equal to the number of processes. Each process will take one part of the data.

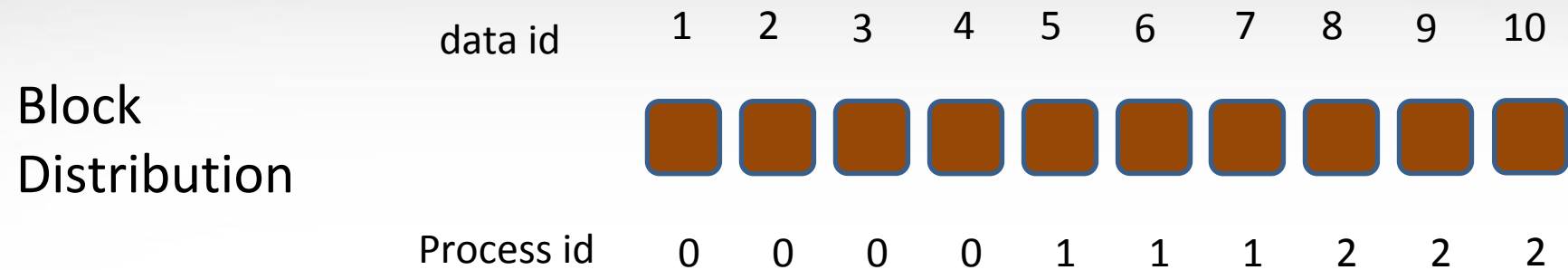
C

```
block_map(1,N,nprocs,myid,&l1,&l2);  
for (i=l1; i<=l2; i++){  
    x = h*(i-0.5);  
    sum += 4.0/(1.0+x*x);  
}  
sum = sum*h;
```

Fortran

```
call block_map(1,N,nprocs,myid,l1,l2)  
do i=l1, l2  
    x = h*(i-0.5d0)  
    sum = sum + 4.0d0/(1.0d0+x*x)  
enddo  
sum = sum*h
```

Data Distribution: block



C

```
void block_map(int n1, int n2, int nprocs,
               int myid, int *l1, int *l2)
{
    int block, rem;
    block = (n2-n1+1)/nprocs;
    rem = (n2-n1+1)%nprocs;
    if (myid < rem){
        block++;
        *l1 = n1+myid*block;
    }else
        *l1 = n1+rem+block*myid;

    *l2 = *l1+block-1;
}
```

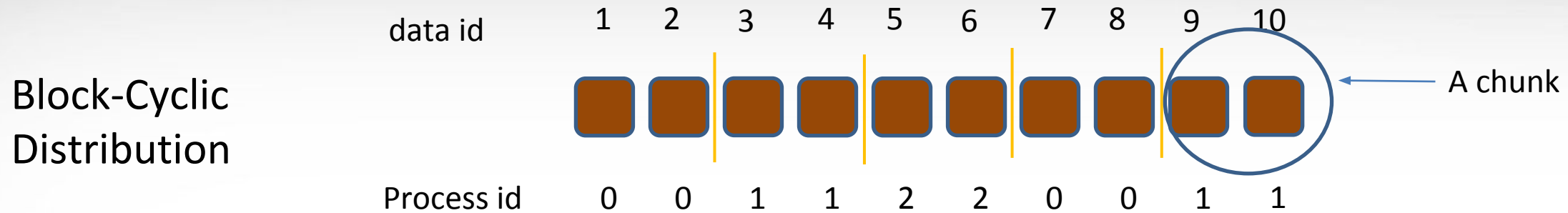
Fortran

```
subroutine block_map(n1,n2, nprocs, myid, l1, l2)
    Integer n1, n2, nprocs, myid, l1,l2

    integer block, rem
    block = (n2-n1+1)/nprocs
    rem = mod(n2-n1+1, nprocs)
    if (myid < rem) then
        block = block+1
        l1 = n1+myid*block
    else
        l1 = n1+rem+block*myid
    end if
    l2 = l1+block-1
end subroutine block_map
```

example6-data_distribution/calc_PI_block

Data Distribution: block cyclic



Data is divided into chunks of contiguous blocks and the chunks are distributed in a round-robin manner

Loop through chunks

C

```
for (i=myid*BLK+1; i<=N; i+=nprocs*BLK) {
  for (j=i; j<=MIN(N,i+BLK-1); j++) {
    x = h*(j-0.5);
    sum += 4.0/(1.0+x*x);
  }
}
sum = sum*h;
```

Fortran

```
do i=myid*BLK+1, N, nprocs*BLK
  do j=i, MIN(N,i+BLK-1)
    x = h*(j-0.5d0)
    sum = sum+4.0d0/(1.0d0+x*x)
  enddo
enddo
sum = sum*h
```

Loop through blocks inside a chunk

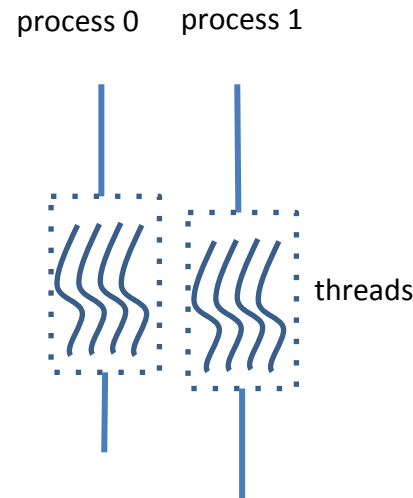
example6-data_distribution/calc_PI_bc

MPI/OpenMP Hybrid Programming

- Simplest and intuitive form:
master-only: only master thread can execute MPI calls

```
Call MPI_INIT(ierr)
...
Call MPI_SEND(...)
...
!$OMP DO
DO i=1, N
...
ENDDO
!$OMP END DO
...
CALL MPI_FINALIZE(ierr)
```

(no MPI calls in
the openMP
parallel region)



- Starting MPI-2, the standard provides guidelines on how to interact MPI with threads
- Four levels of thread support
 - MPI_THREAD_SINGLE: Only one thread will execute
 - MPI_THREAD_FUNNELED: Only master thread will make MPI-calls
 - MPI_THREAD_SERIALIZED: Multiple threads may make MPI-calls, but only one at a time
 - MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions

`MPI_INIT_THREAD(required, provided, ierr)`

`mpiifort -qopenmp [options] prog.f90 -o prog.exe`

Example 9: Hybrid Programming

```
int MPI_Init_thread(int *argc, char **argv, int required, int *provided)
```

```
MPI_Init_thread(required, provided, ierr)
```

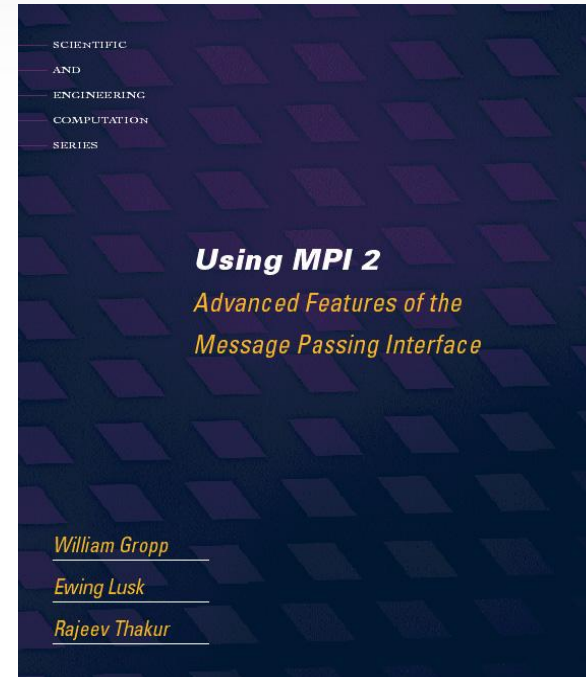
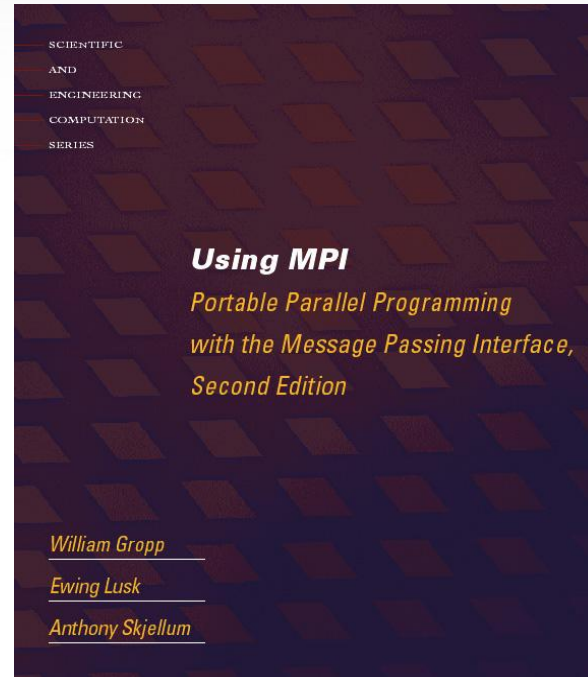
- Four possible values for the parameter **required**:
 - MPI_THREAD_SINGLE `ex2_single.c`
 - MPI_THREAD_FUNNELED `ex2_funnel.c`
 - MPI_THREAD_SERIALIZED `ex2_serialized.c`
 - MPI_THREAD_MULTIPLE `ex2_multiple.c`

Acknowledgement

- Part of the content is adapted from former Supercomputing Facility staff Mr. Spiros Vellas' introductory MPI short course

Resources

- Two books: [Using MPI](#) and [Using MPI2](#)



- MPI standard: <http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>
- List of all MPI routines: <https://www.mpich.org/static/docs/v3.2/>
- Examples for the course are on Ada: </general/public/training/mpi/Spring2019>