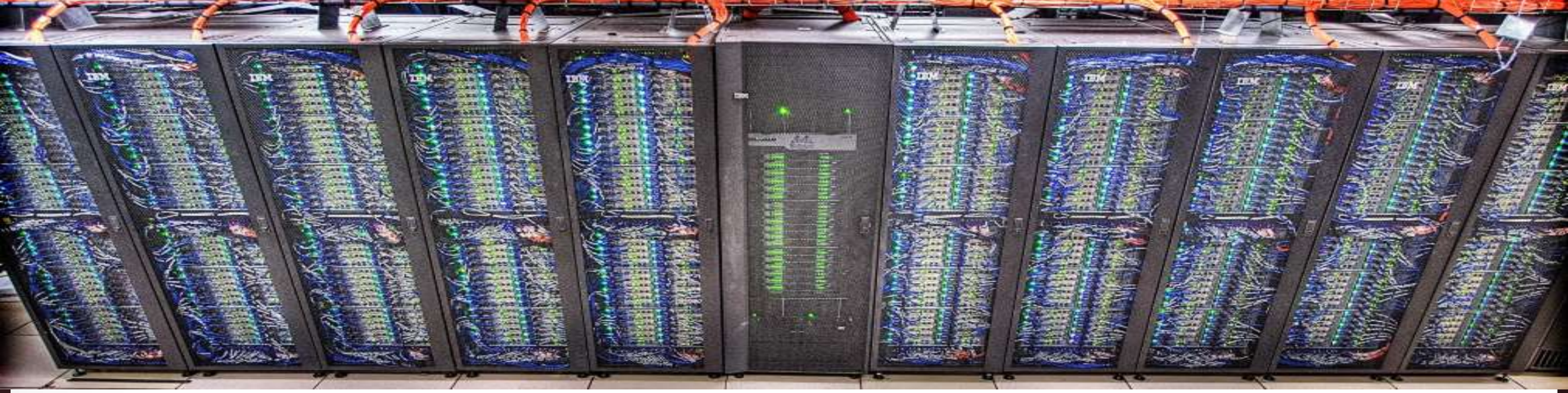


Introduction to Code Parallelization Using MPI

Ping Luo

TAMU HPRC

March 29, 2019

HPRC Short Course – Spring 2019

HIGH PERFORMANCE RESEARCH COMPUTING
TEXAS A&M UNIVERSITY

Outline

- Parallel programming models
  - OpenMP for Shared Memory System
  - MPI for Distributed Memory System
  - MPI+OpenMP for hybrid systems
- Layout of an MPI program
- Compiling and running an MPI program
- Basic MPI concepts
  - size, rank, communicator, message, MPI datatype, tag
- Point-to-point communication
- Collective communication

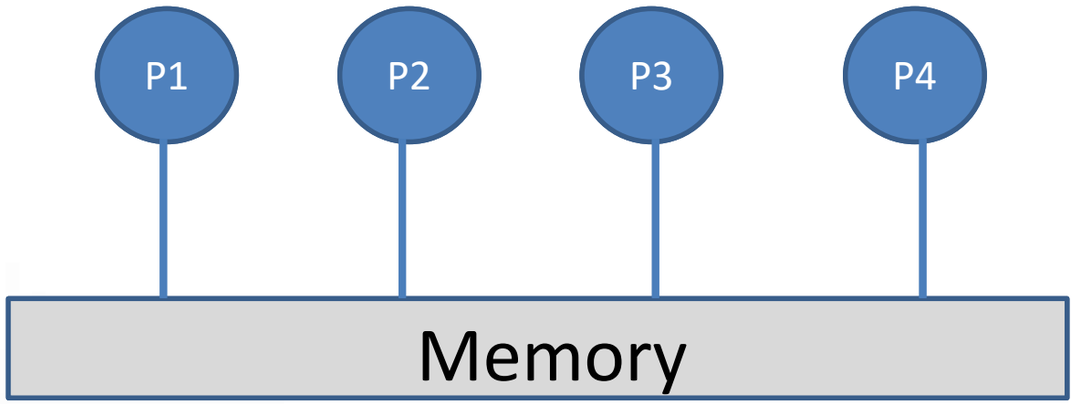
Parallel Computing Systems

We use a processor or a CPU core to refer to the smallest physical processing unit where a program is executed

- Shared Memory Systems
- Distributed Memory Systems
- Hybrid Systems

Parallel Computing Systems

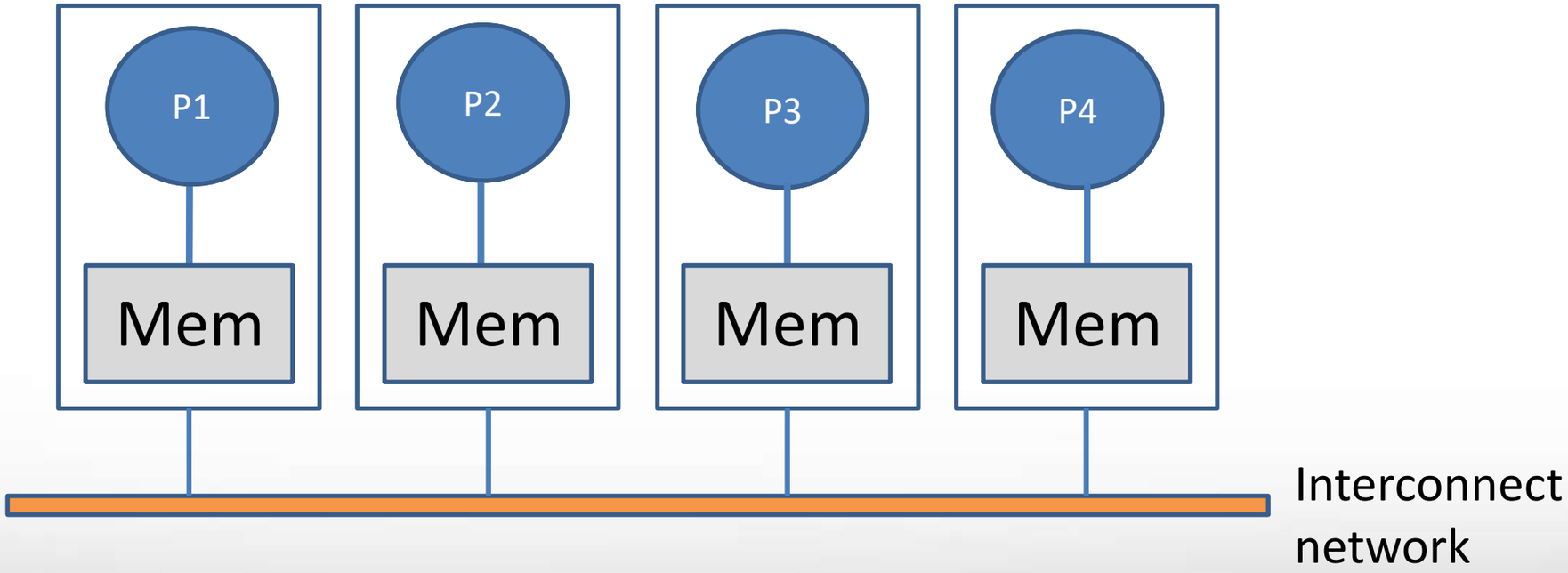
- Shared Memory System – an abstraction to a parallel system where all processors share the same memory subsystem



Example: a node on Ada – an IBM NextScale nx360 M4 dual socket server

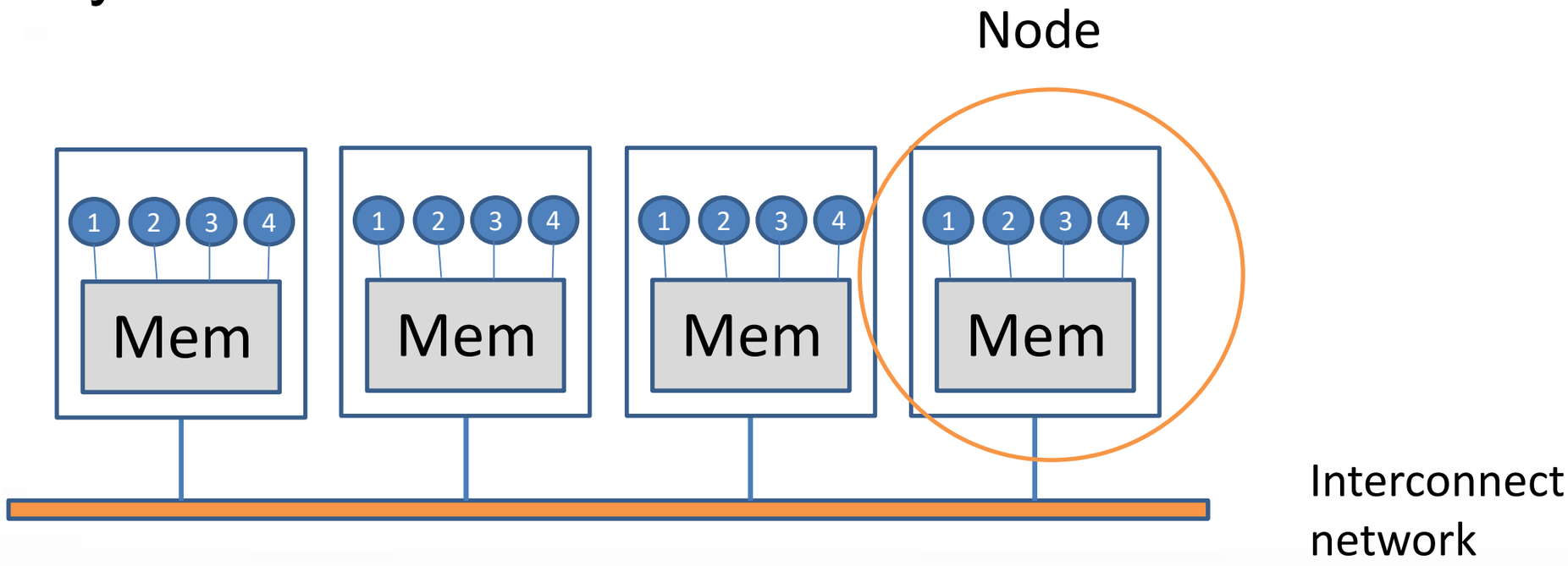
Parallel Computing Systems

- Distributed Memory System – an abstraction to a parallel system where each processor has its own local memory and the processors don't share a global memory subsystem.



Parallel Computing Systems

- Hybrid System

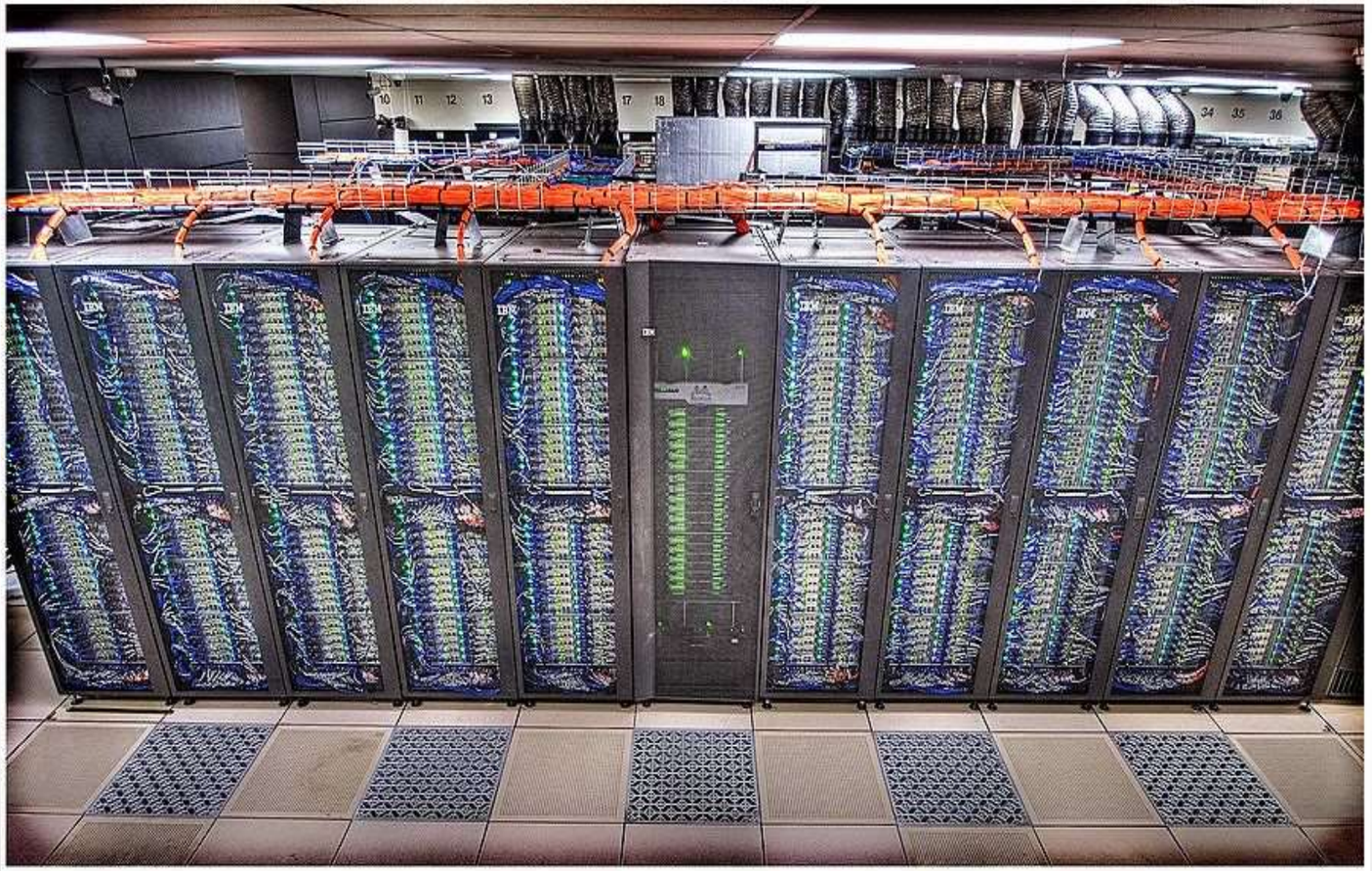


Parallel Programming Models

- Mapping from the parallel programming models to the parallel computing systems

OpenMP	Shared Memory System
MPI	Distributed Memory System
MPI+OpenMP	Hybrid System

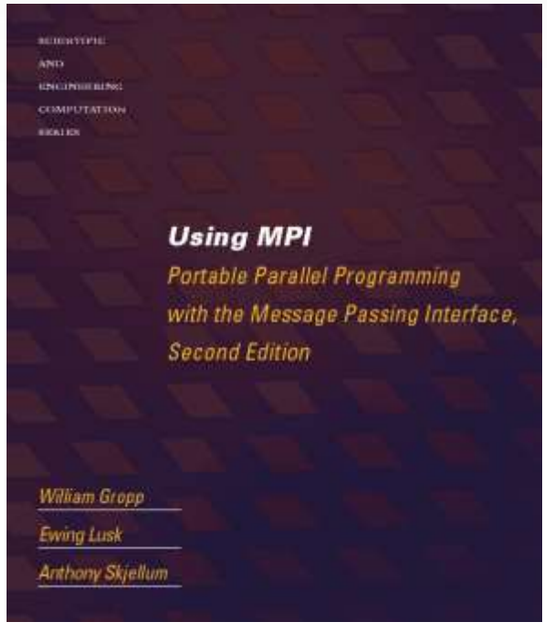
Ada and Terra

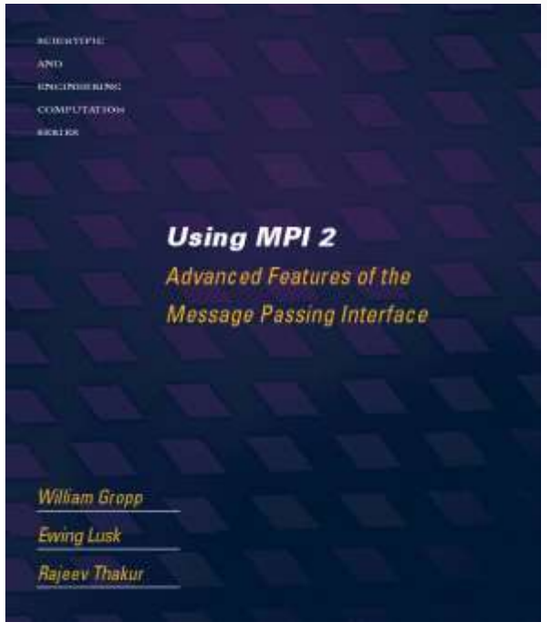


- Ada has 852 nodes and each node has 20 cores.
- Terra has 304 compute nodes and each node has 28 cores.
- OpenMP can only run within a cluster node.
- MPI and Hybrid can run on multiple nodes.

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

Example 1: Hello World

C

```
#include <stdlib.h>

int main(int argc, char **argv){

    printf("Hello, world\n");

}
```

Fortran

```
program hello
implicit none


print *, “Hello, world”

end program hello
```

Example 1: Hello World

C

```
#include <stdlib.h>

int main(int argc, char **argv){

    printf("Hello, world\n");

}
```

Fortran

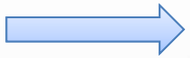
```
program hello
implicit none


print *, “Hello, world”

end program hello
```

C

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

int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    printf("Hello, world\n");
    MPI_Finalize();
}
```

Fortran

```
program hello
use mpi
implicit none

call MPI_INIT(ierr)
print *, “Hello, world”
call MPI_Finalize(ierr)
end program hello
```



Layout of an MPI Program

C

```
#include <mpi.h>
int main(
   int argc, char **argv)
{
   …
   MPI_Init(&argc, &argv);

   MPI_Finalize();
   …
}
```

no mpi calls

mpi calls happen here

multiple concurrent processes execute at their own pace unless synchronization is applied.

Fortran

```
PROGRAM SAMPLE1
USE MPI !F90
!f77: include "mpif.h"
integer ierr
…
CALL MPI_INIT(ierr)

CALL MPI_FINALIZE(ierr)
…
END PROGRAM SAMPLE1
```

no mpi calls

mpi calls happen here

no mpi calls

Compiling and Linking MPI Programs

```
module load intel/2017b
mpiicc    prog.c   [flags] -o prog.exe   (C)
mpicpc    prog.cpp [flags] -o prog.exe   (C++)
mpiifort  prog.f   [flags] -o prog.exe   (Fortran)
(Intel compilers)
mpicc     prog.c   [flags] -o prog.exe   (C)
mpicxx    prog.cpp [flags] -o prog.exe   (C++)
mpif90    prog.f   [flags] -o prog.exe   (Fortran)
(GNU compilers)
```

We will use the Intel MPI and the Intel compilers in ensuing examples.

See HPRC user guide for more information:

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

Running an MPI Program on Login Nodes

- Load the modules first

```
module load intel/2017b
```

- Run the mpi program on the login nodes for testing and debugging. No more than 8 MPI tasks can be launched at once per our policy.

```
mpirun -np n [options] \
prog.exe [prog_args]
(n is number of MPI tasks
and n<=8)
```

- Useful MPI options

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

- When testing the code on the login nodes, make sure the code works on multiple login nodes.

```
mpirun -np 4 -hosts  \
login1,login2,login3,\
login4 -ppn 1 mympi.exe
```

Running an MPI Program in Batch

- Batch Examples

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/2017b
mpirun prog.exe
```

bsub < mpibatch.job

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/2017b
mpirun prog.exe
```

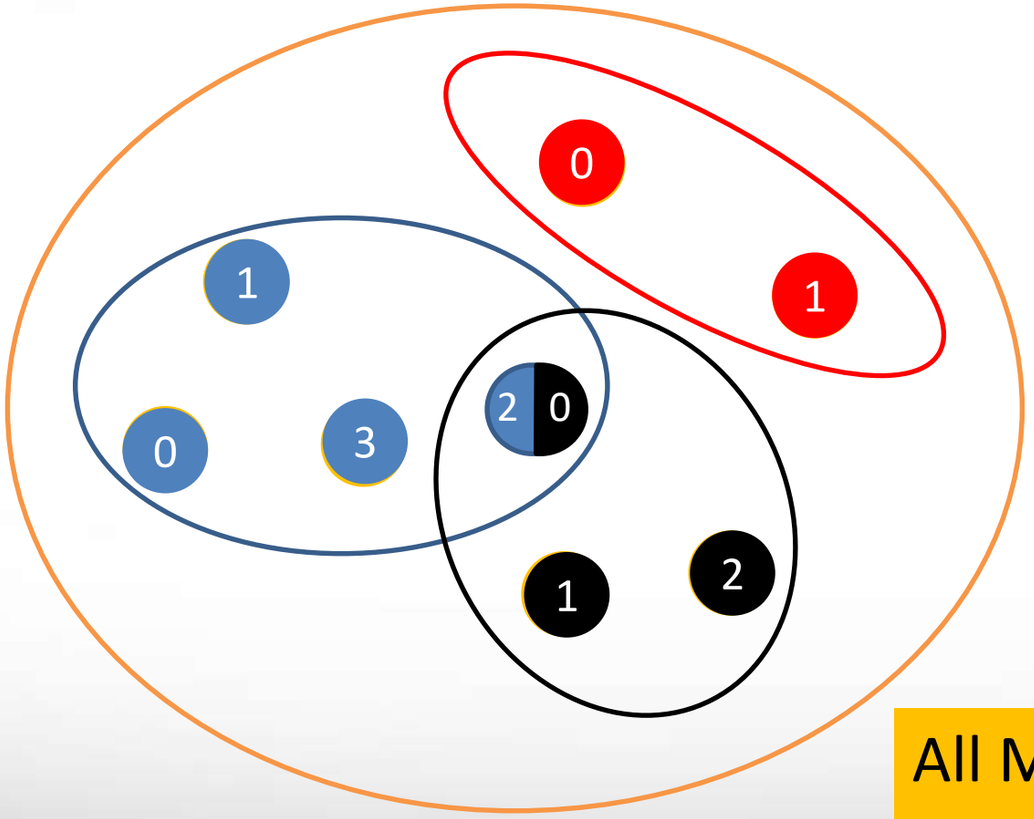
sbatch mpibatch.job

What is MPI

- Message Passing Interface: a specification for the library interface that implements message passing in parallel programing.
- Is standardized by the MPI forum for implementing portable, flexible, and reliable codes for distributed memory systems, regardless of the underneath architecture.
  - 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 programing paradigm for large scale scientific computing.

Basic MPI Concepts

communicator, size, rank

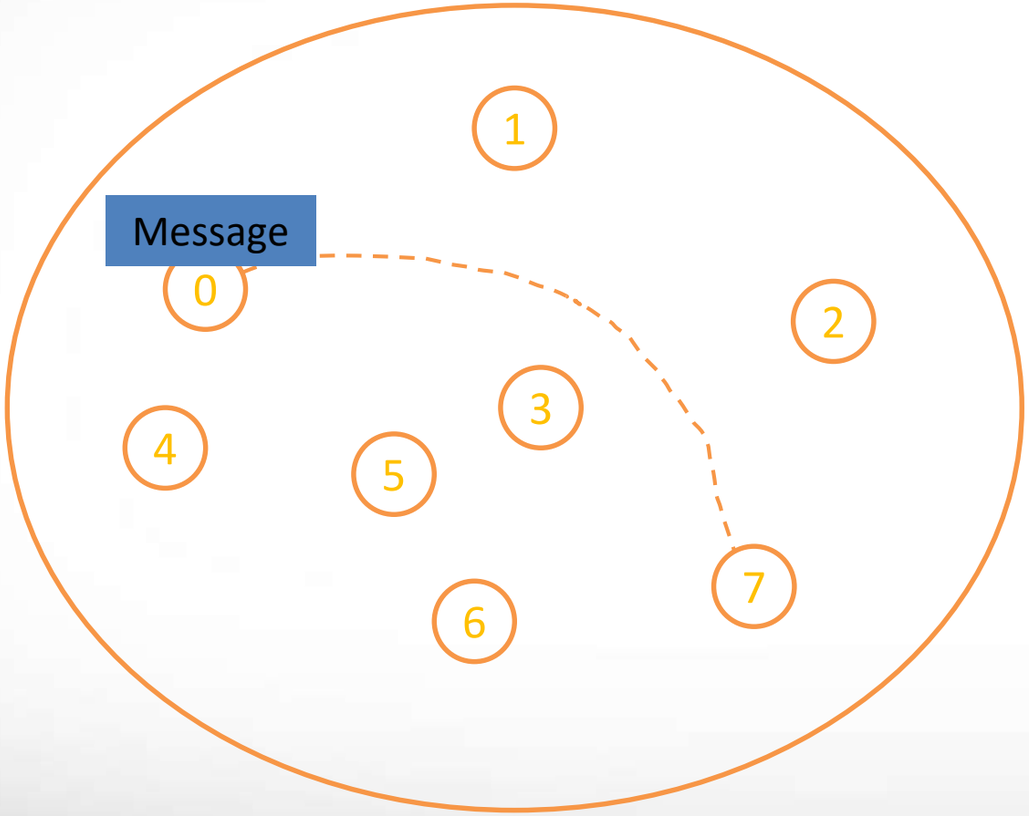


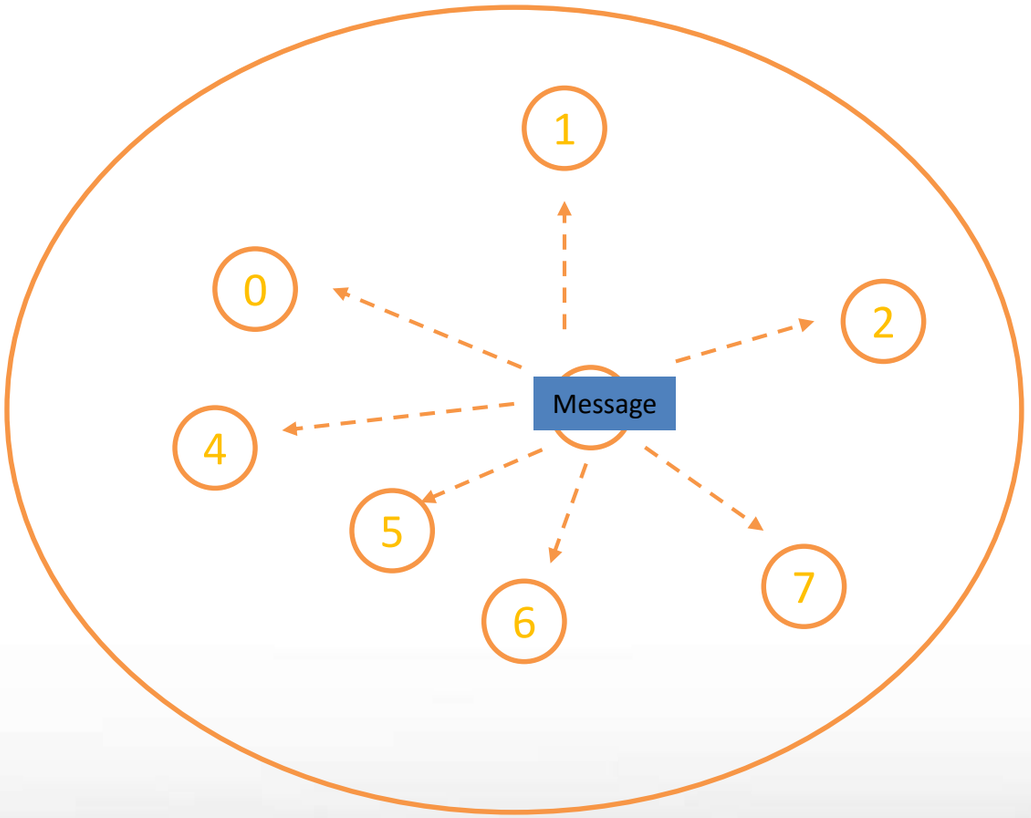
Communicator: MPI_COMM_WORLD
Size: 8
Rank: 0, 1, …, 7
Communicator: comm1
Size: 2
Rank: 0, 1
Communicator: comm2
Size: 4
Rank: 0, 1, 2, 3
Communicator: comm3
Size: 3
Rank: 0, 1, 2

All MPI communication must specify a communicator.

Basic MPI Concepts

message, point-to-point communication, collective communication





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 out %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
```

Communicator

- In MPI, a communicator is a software structure through which we specify a group of processes.
- Each process in 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 typically the most used communicator.
- Communicators are especially useful in characterizing the tasks that different groups of processes carry out.

Size and Rank

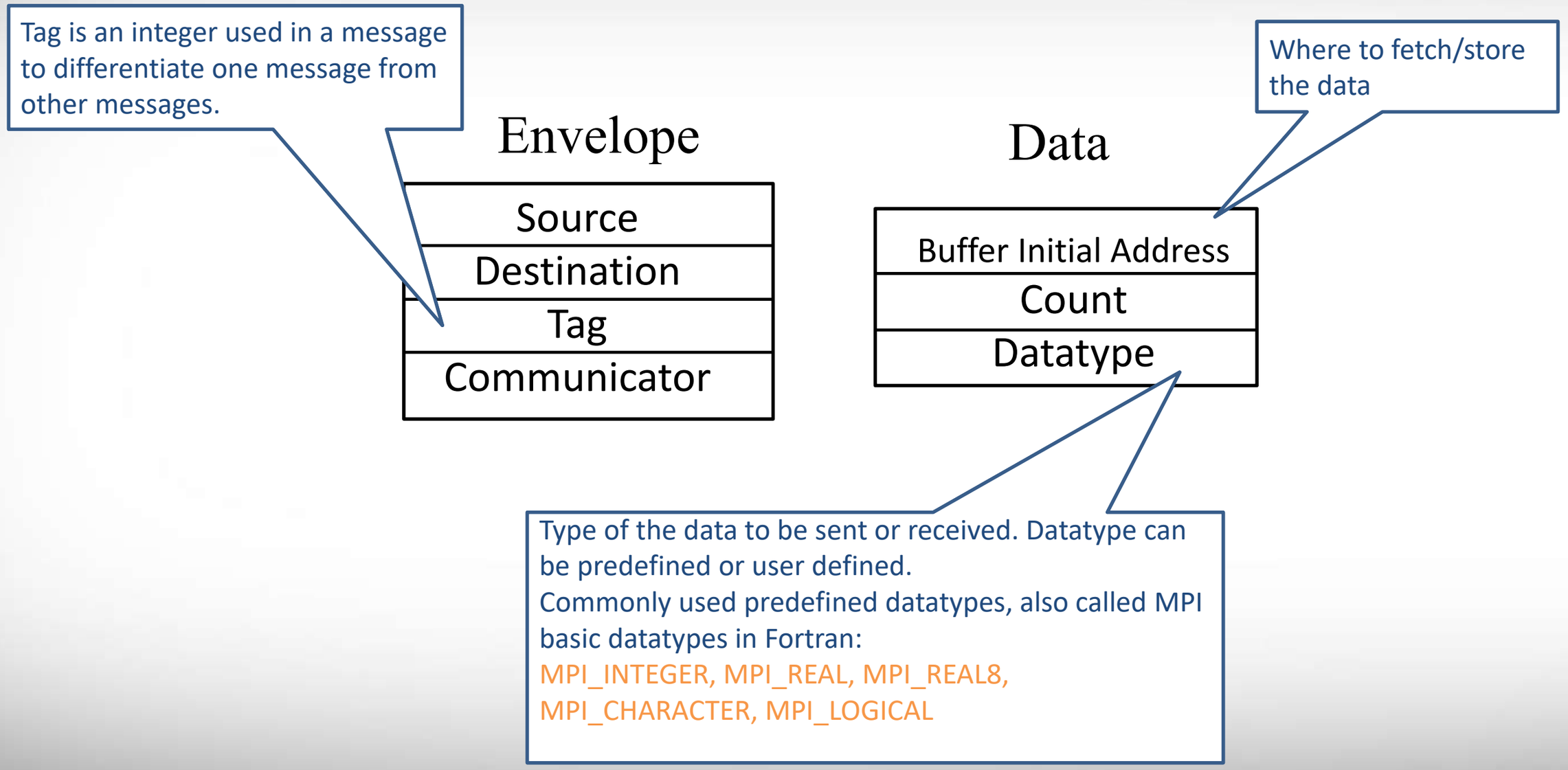
- How many processes in a communicator?

C	`int MPI_Comm_size(MPI_Comm comm, int *size)`
Fortran	`SUBROUTINE MPI_COMM_SIZE(comm, size, ierr)` `integer comm, size, ierr`

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

C	`int MPI_Comm_rank(MPI_Comm comm, int *rank)`
Fortran	`SUBROUTINE MPI_COMM_RANK(comm, rank, ierr)` `integer comm, rank, ierr`

Message


Tag is an integer used in a message to differentiate one message from other messages.
Envelope
Source
Destination
Tag
Communicator
Where to fetch/store the data
Data
Buffer Initial Address
Count
Datatype
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

Send and Receive a Message

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

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 ==
   call MPI_REC , 1, MI , 0, 0, MPI_COMM_WORLD, status, ierr)
   print *, "process ",rank," receives ", number
endif

call MPI_Finalize(ierr)
end program simpleC
```

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)

Buffer starting address

Data count

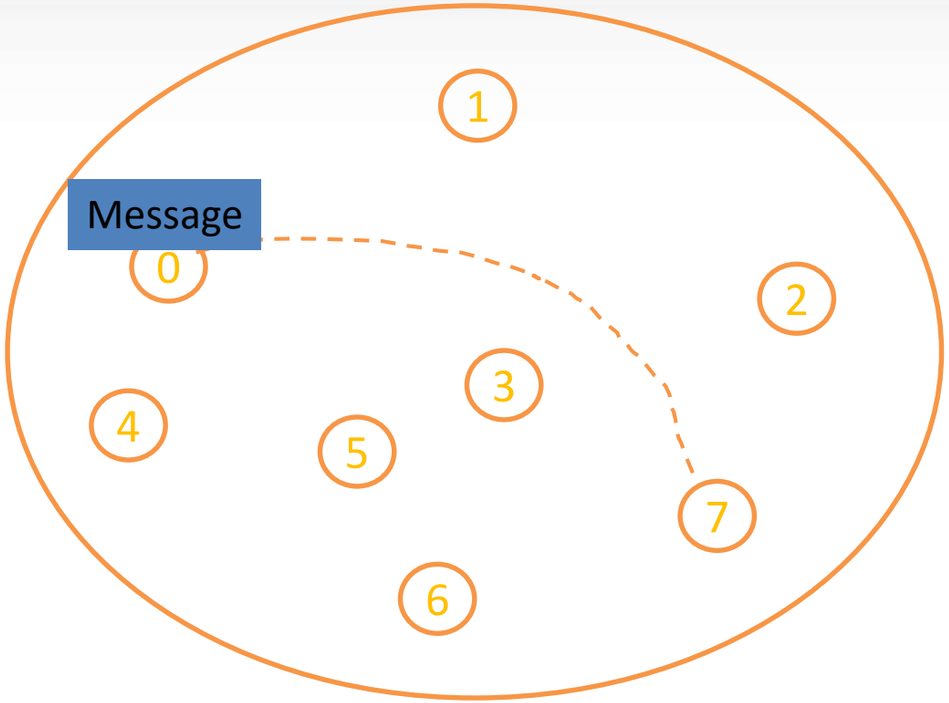
MPI datatype

source

tag

communicator

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	initial 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

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.

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]

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

MPI_ANY_SOURCE and MPI_ANY_TAG are MPI defined wildcards.

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)
```

Example 3

- One sender and multiple receivers
- The sender sends out a number to each and every receiver

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv){
   int number, size, rank;
   int i;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (size < 2){
      MPI_Abort(MPI_COMM_WORLD, 99);
   }
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank == 0){
      printf("Type any number from the input: ");
      scanf("%d", &number);
      for (i=1; i<size; i++)
         MPI_Send(&number, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
   }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);
   }
   MPI_Finalize();
}
```

```fortran
program ex3
use mpi
implicit none
integer rank, np, ierr, number, i

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

if (np < 2) then
   call MPI_ABORT(MPI_COMM_WORLD, 99, ierr)
endif
if (rank == 0) then
   print *, "Type an integer from the input"
   read *, number
   do i=1, np-1
      call MPI_SEND(number, 1, MPI_INTEGER, i, 0, MPI_COMM_WORLD, ierr)
   enddo
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
call MPI_FINALIZE(ierr)
end program ex3
```

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)

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

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

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)

C

```
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)
```

Fortran

```
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);
```

statuses can be ignored with MPI_STATUSES_IGNORE

Example 4

- One sender and multiple receivers
- The sender sends out a number to each and every receiver
- The sender uses non-blocking send

C

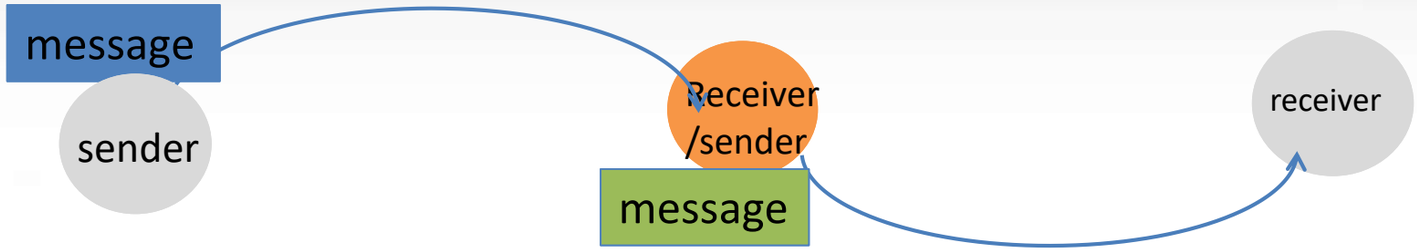
```
MPI_Request *requests;
   ....
if (rank == 0){
   printf("Type any number from the input: ");
   scanf("%d", &number);
   requests = (MPI_Request *)(malloc(npof(MPI_Request)*(np-1)));

   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

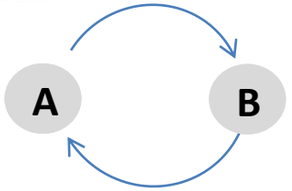
```
integer, allocatable::requests(:)
  ....
if (rank == 0) then
   print *, "Type an integer from the input"
   read *, number
   allocate(requests(np-1))
   do i=1, np-1
      call MPI_ISEND(number, 1, MPI_INTEGER, i, 0,   &
                              MPI_COMM_WORLD, ierr)
   enddo
   call MPI_WAITALL(np-1, 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
```

Send-Receive

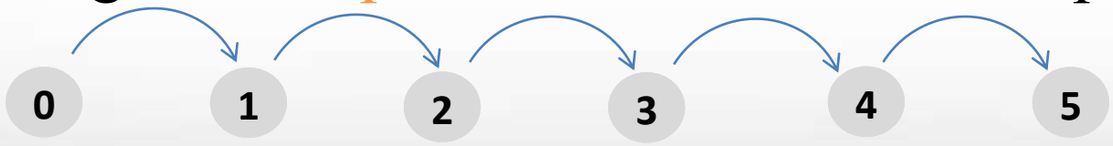
MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)



- Combines send and receive operations in one call
- The source and destination can be the same.



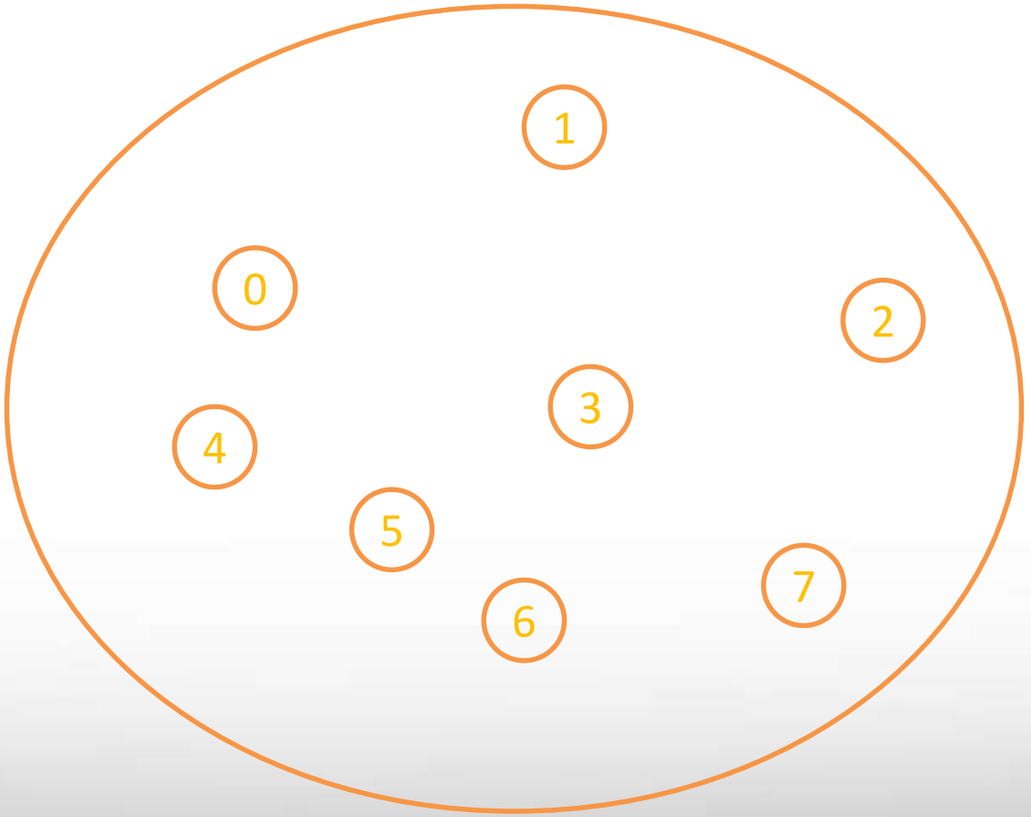
- The message sent out by send-receive can be received by blocking/non-blocking receive or another send-receive
- It can receive a message sent by blocking/non-blocking send or another send-receive.
- Useful for executing a shift operation across a chain of processes.



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

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_ALLTOALL	The j-th block of the sendbuf at process i is send to process j and placed in the i-th block of the recvbuf of process j.

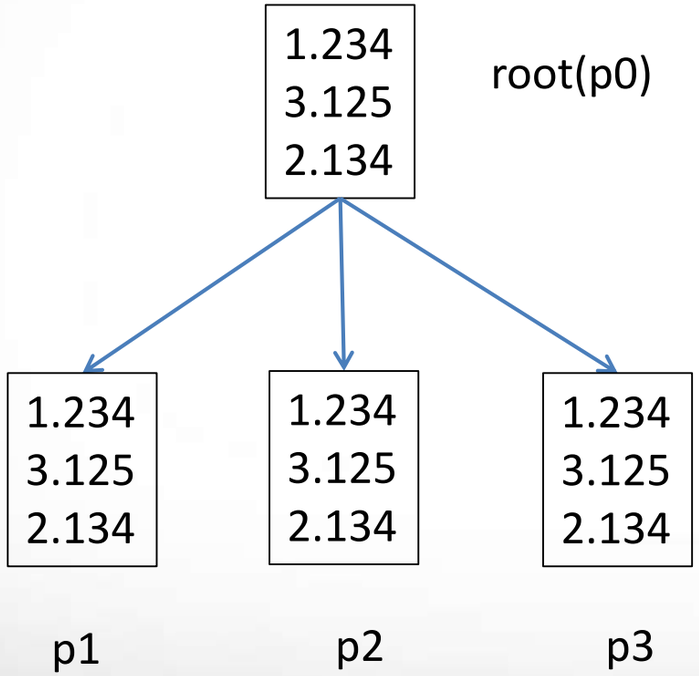
MPI_BARRIER

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_BCAST

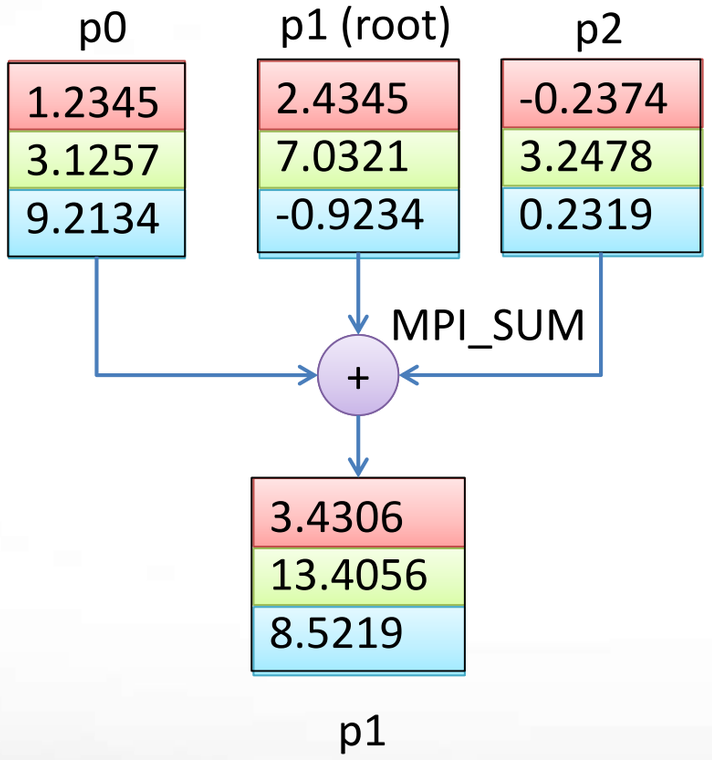
MPI_BCAST(buffer, count, datatype, root, comm)



- 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.

MPI_REDUCE

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



C: MPI_Op op

Fortran: integer op

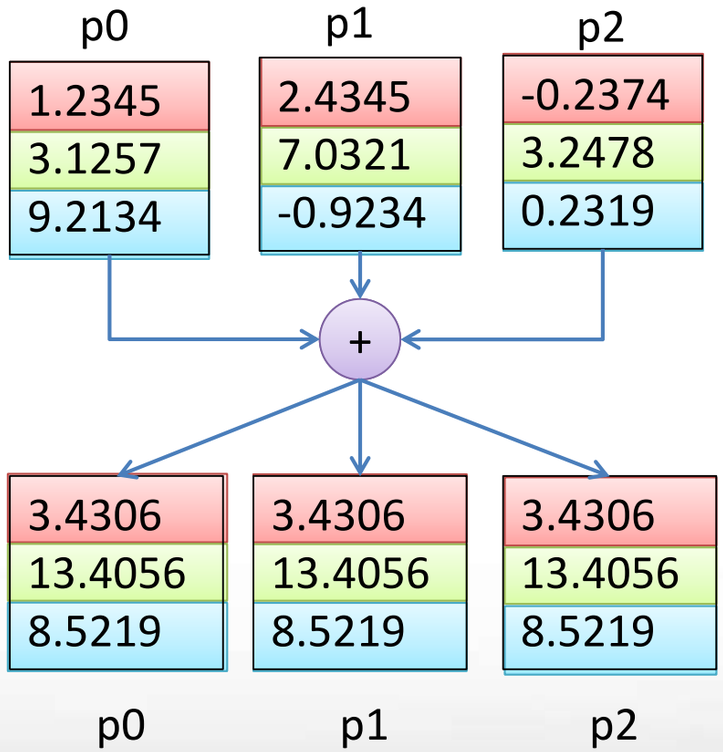
- Performs a reduction operation on all elements with same index in *sendbuf* on all processes and stores results in *recvbuf* of the root process.
- *recvbuf* is significant only at root.
- *sendbuf* and *recvbuf* cannot be the same.
- The size of *sendbuf* and *recvbuf* is equal to count.

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

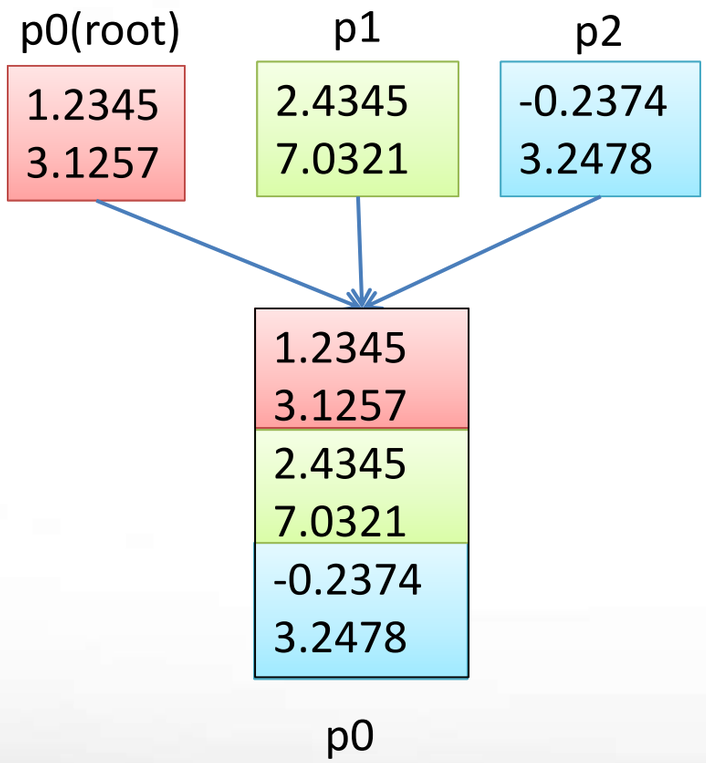
MPI_ALLREDUCE

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



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*. recvnts[i] = number of elements received from process i.

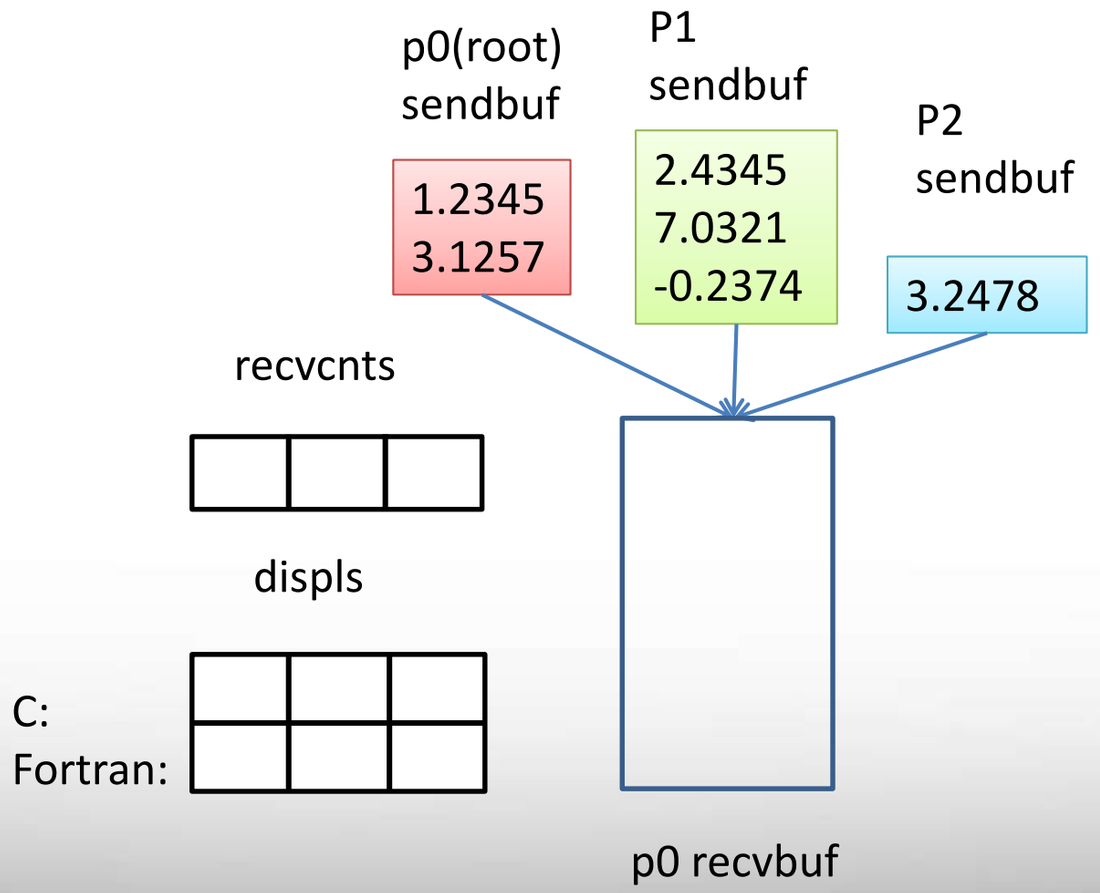
IN displs an integer array of size of *comm*. displs[i] = displacement from *recevbuf* for process i.

Fortran
integer recvcnts(*), displs(*)

C
int recvcnts[], displs[]

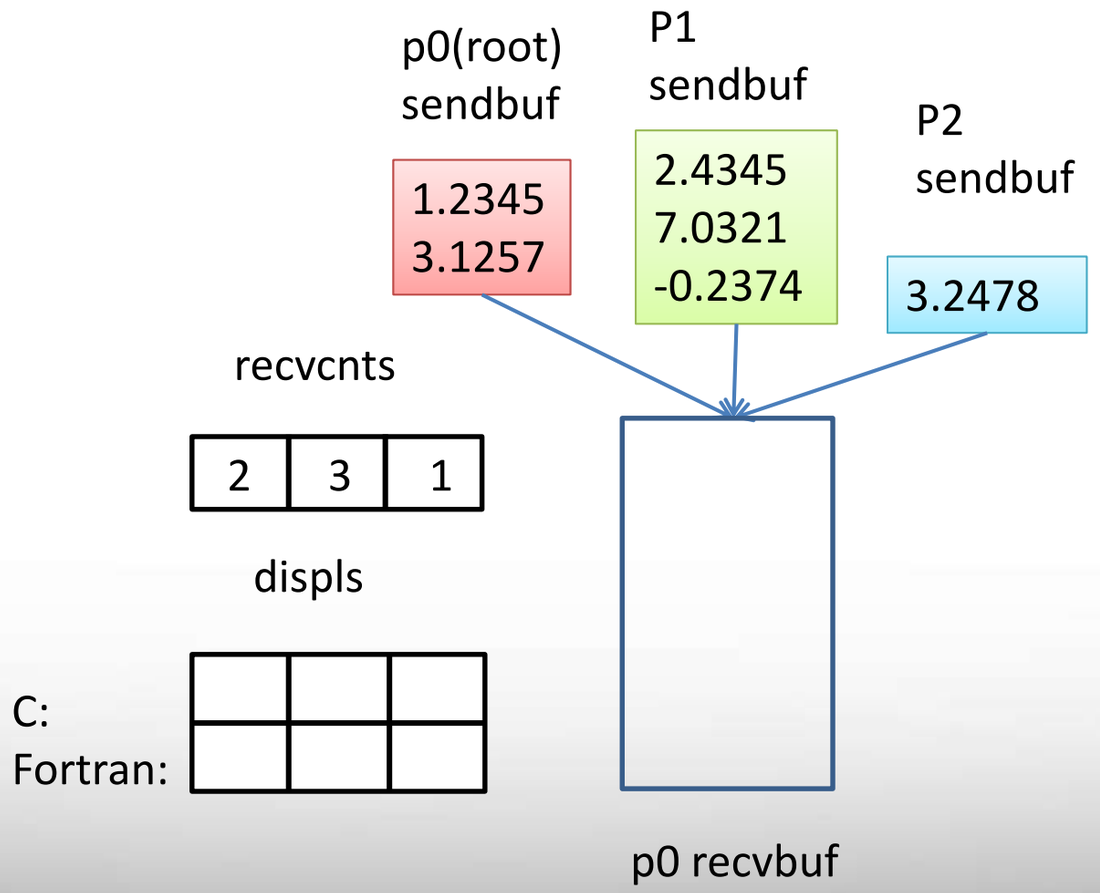
MPI_GATHERV (cont.)

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



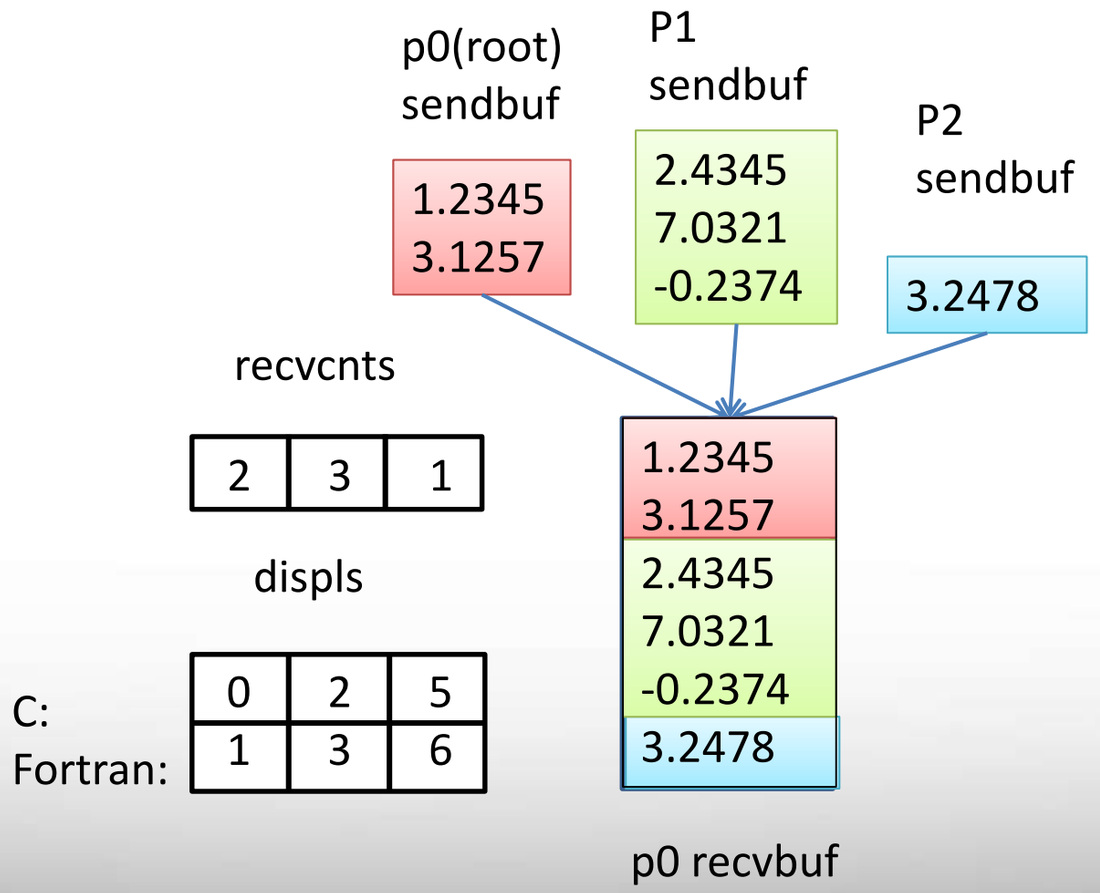
MPI_GATHERV (cont.)

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

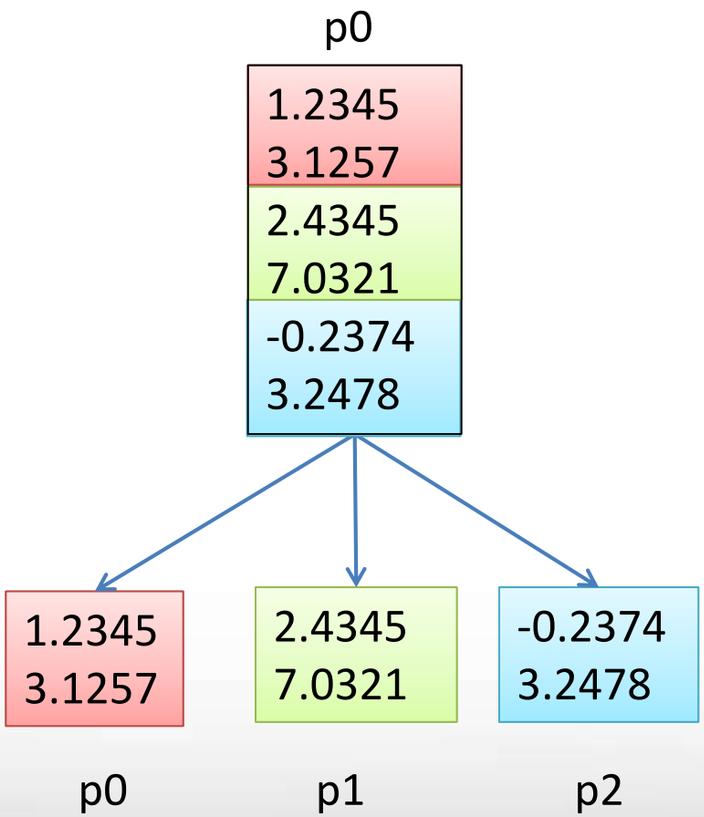
MPI_ALLGATHER(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,recvtype,comm)

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.

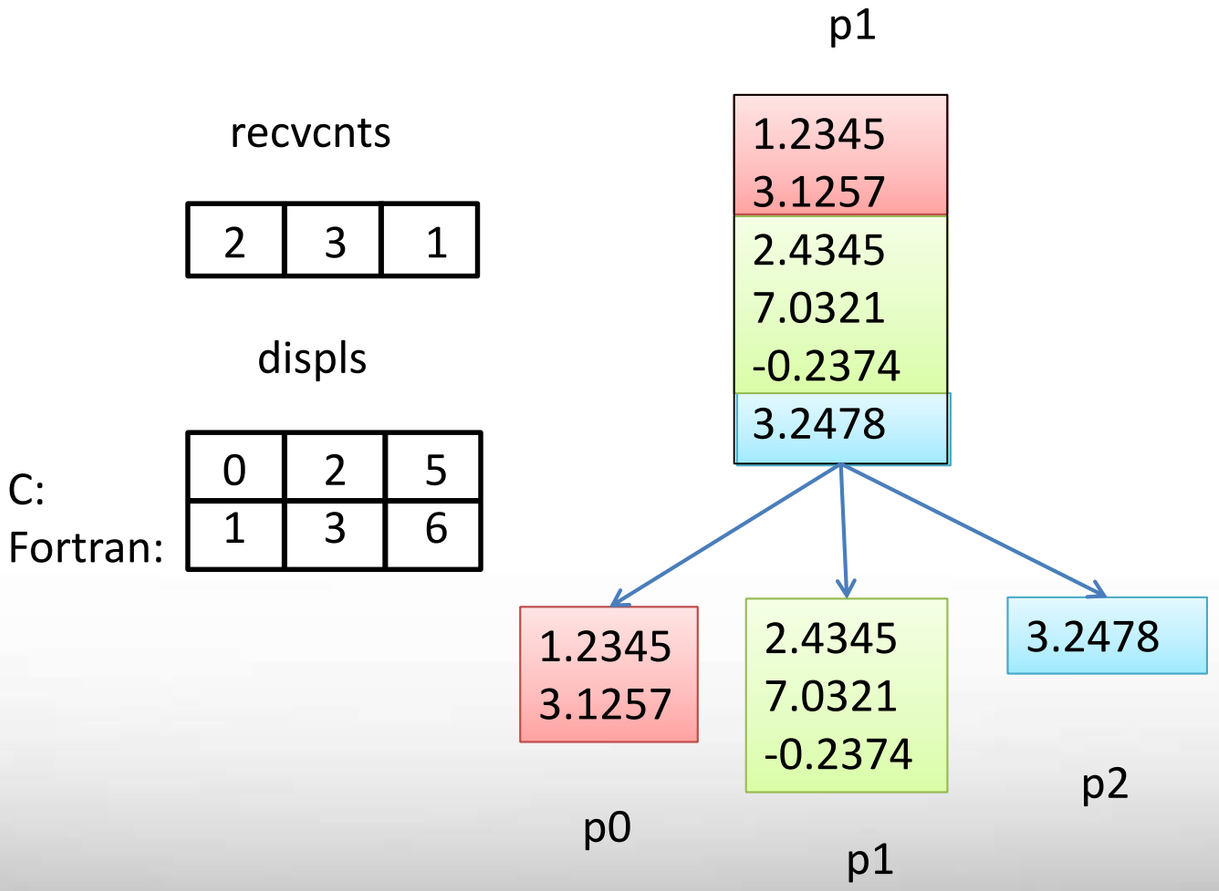
MPI_SCATTER

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



MPI_SCATTERV

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



Timing Routine

MPI_WTIME()

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

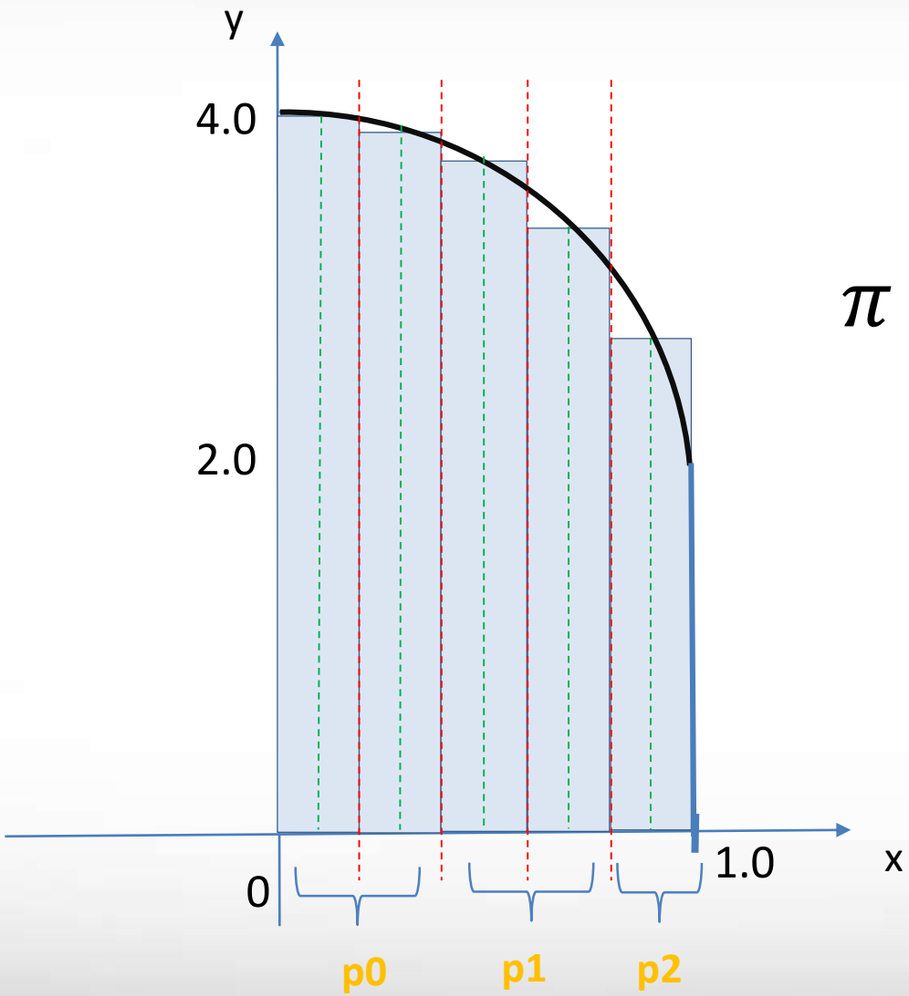
C

```
double t1, t2;
double 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
```

Example 5: Calculate PI



$$\pi = \int_0^1 \frac{4.0}{1+x^2}\,dx$$

Break

Outline for MPI Part II

- Exploring parallelism
  - Task-parallelism
  - Data-parallelism
- Matrix-vector multiplication
- Solving the 2D Poisson equation
- Domain decomposition
- MPI/OMP hybrid programming
- MPI/OMP: A Case Study

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

Task parallelism

4 meals

salad steak desert

4 meals

salad steak desert

4 meals

salad steak desert

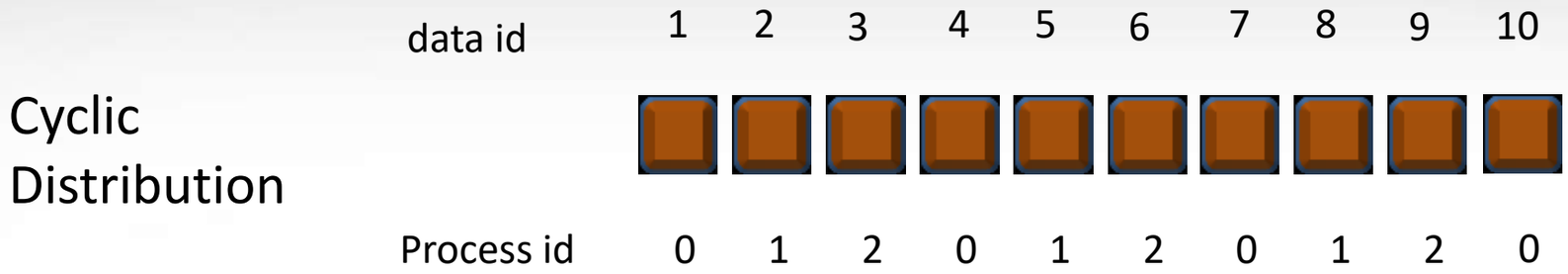
Data parallelism

https://101clipart.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

Example 6: Data Distribution



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

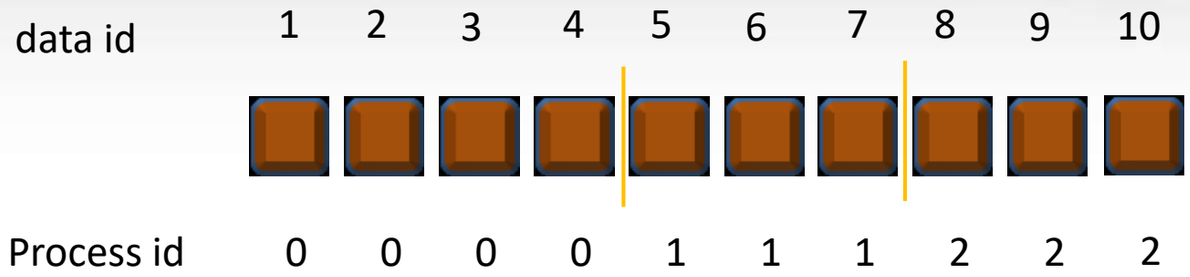
C	Fortran

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

calc_PI_cyclic.c

Example 6: Data Distribution

Block Distribution



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

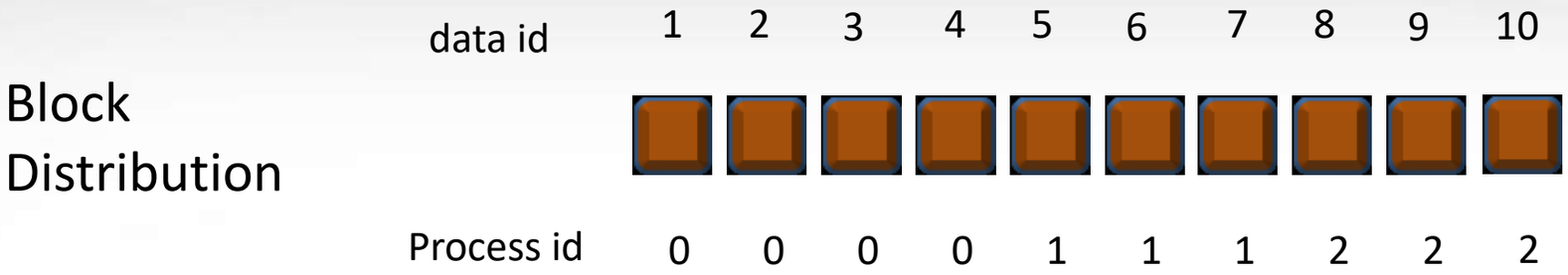
```
block_map(1,N,nprocs,myid,&l1,&l2);
for (i=l1; i<=l2; i++){
   x = h*(i-0.5);
   sum += 4.0/(1.0+x*x);
}
sum = sum*h;
```

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
```

calc_PI_block.c

Example 6: Data Distribution



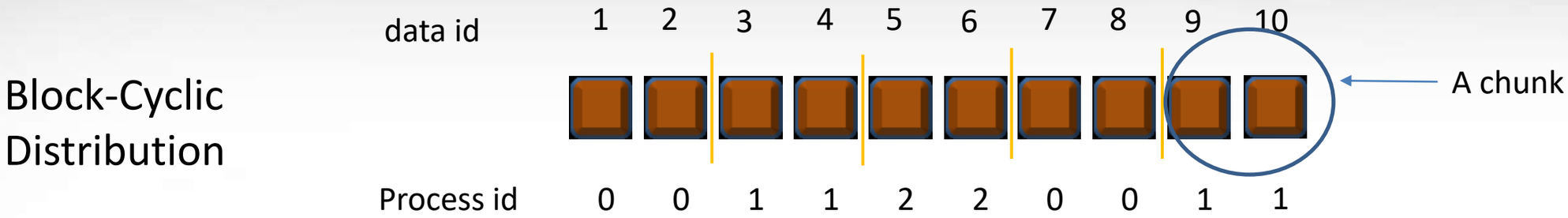
C

```
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
```

Example 6: Data Distribution



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

Loop through chunks

Loop through blocks inside a chunk

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
```

calc_PI_bc.c

Example 7: matvec-scatterv

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

$\vec{a}_i$ 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

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

Example 8: Solving the x-y Poisson Equation

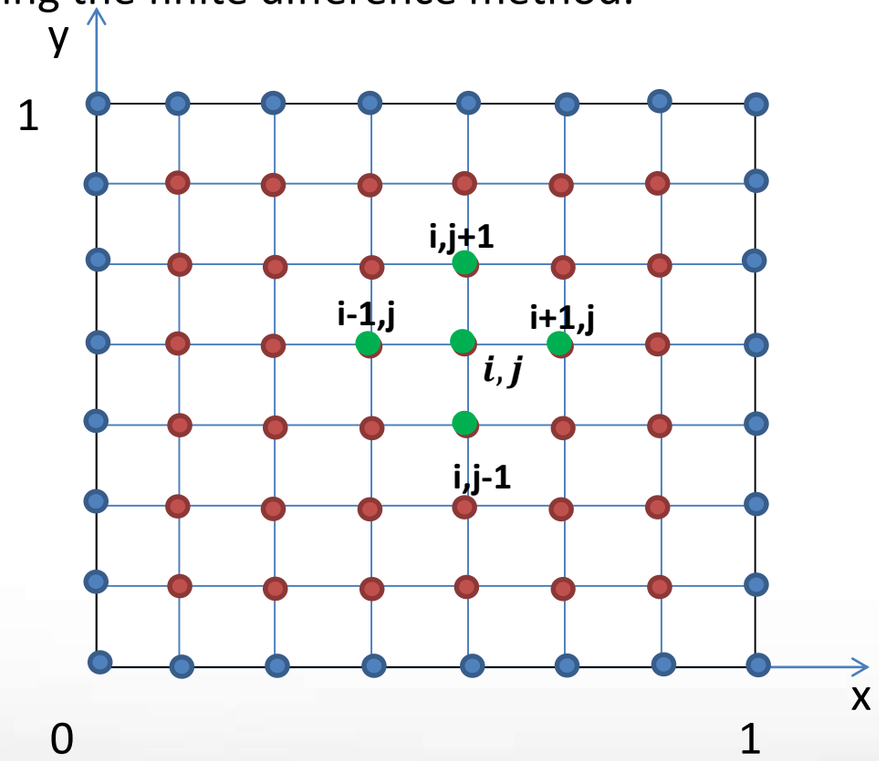
Solve the partial differential equation

$$\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y} = f(x,y)$$

where $x, y \in [0,1]$

and $u = g(x,y)$ on boundary

using the finite difference method.



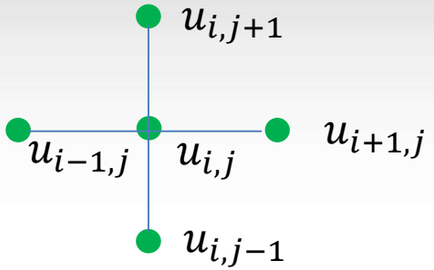
Discretize the domain along x and y using n internal points in each direction.

The increment is
$h = 1/(n+1)$

$x_i = ih, y_j = jh$
$(0 \le i, j \le n+1)$

$u_{ij} = u(x_i, y_j) = u(ih, jh)$
$(0 < i, j < n+1)$

Example 8: the x-y Poisson Equation



5-point finite difference stencil approximation:

$$u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j} = h^2 f_{i,j} \quad (0 < i,j < n+1)$$

$$u_{i,j} = 0.25 * (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - h^2 f_{i,j})$$

k+1 Jacobi iteration step at $x_i = ih;\ y_j = jh; i,j = 1{:}n$

$$u^{k+1}{}_{i,j} = 1/4(u^k{}_{i-1,j} + u^k{}_{i,j+1} + u^k{}_{i,j-1} + u^k{}_{i+1,j}) - h^2 f_{i,j}$$

Jacobi iteration across all points:

```
do j=1, n
   do i = 1, n
      unew(i, j) = 0.25*(u(i-1,j) + u(i, j+1) + u(i+1,j) + i(i, j-1)) – f(i,j)*h^2
   end do
end do
```

Example 8: Solving the x-y Poisson Equation

Boundary conditions (stay fixed):

$$u(x_i, 0) = \frac{\cos(\pi x_i) - \pi^2}{\pi^2} \qquad (0 \le x \le 1)$$

$$u(x_i, 1) = \frac{\cos(\pi x_i) - \pi^2 \cos(x_i)}{\pi^2} \qquad (0 \le x \le 1)$$

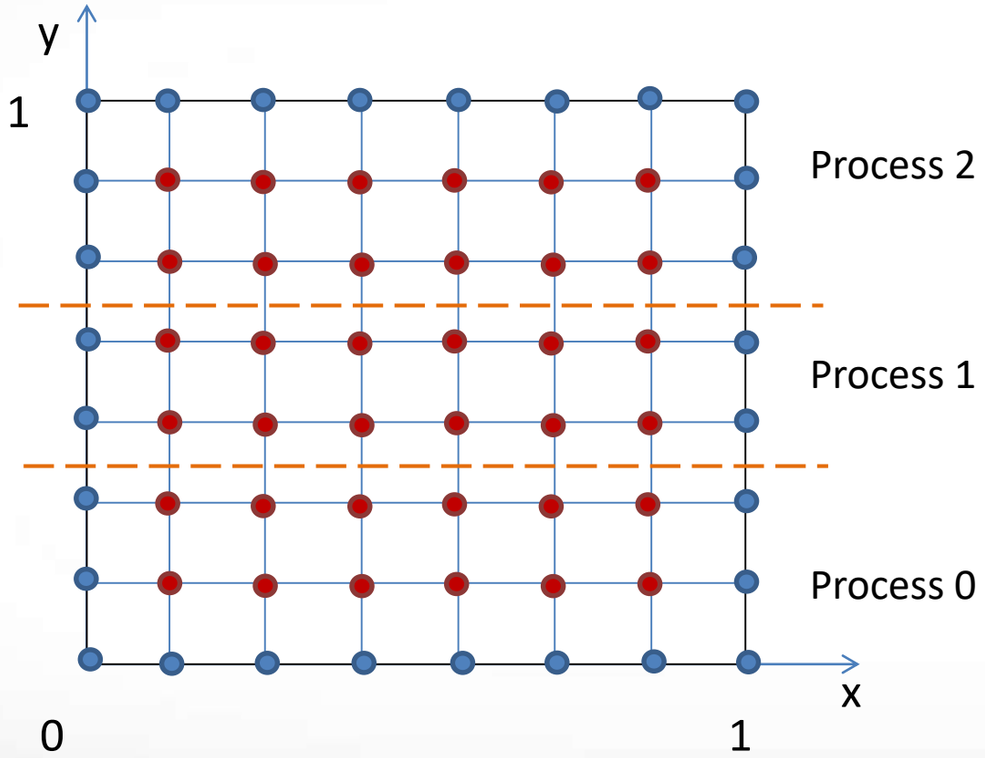
$$u(0, y_j) = \frac{1}{\pi^2} - 1 \qquad (0 \le y \le 1)$$

$$u(1, y_j) = -\left(\frac{1}{\pi^2} + \cos(y_j)\right) \qquad (0 \le y \le 1)$$

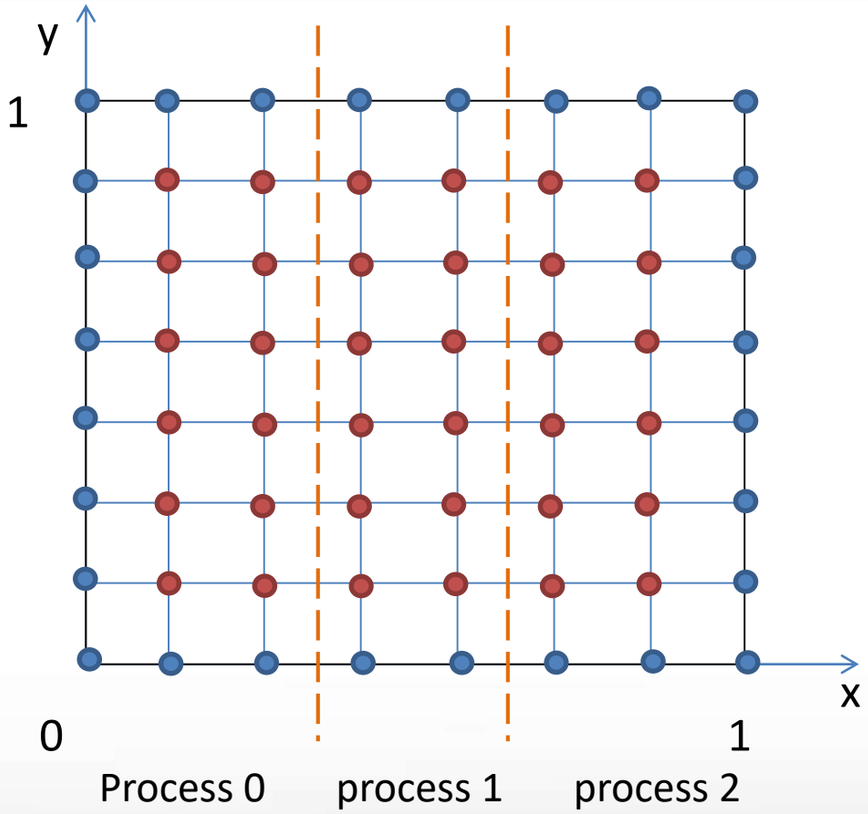
RHS: $f(x_i, y_j) = ({x_i}^2 + {y_j}^2)\cos(x_i y_j) - \cos(\pi x_i)$

1d-Domain Decomposition

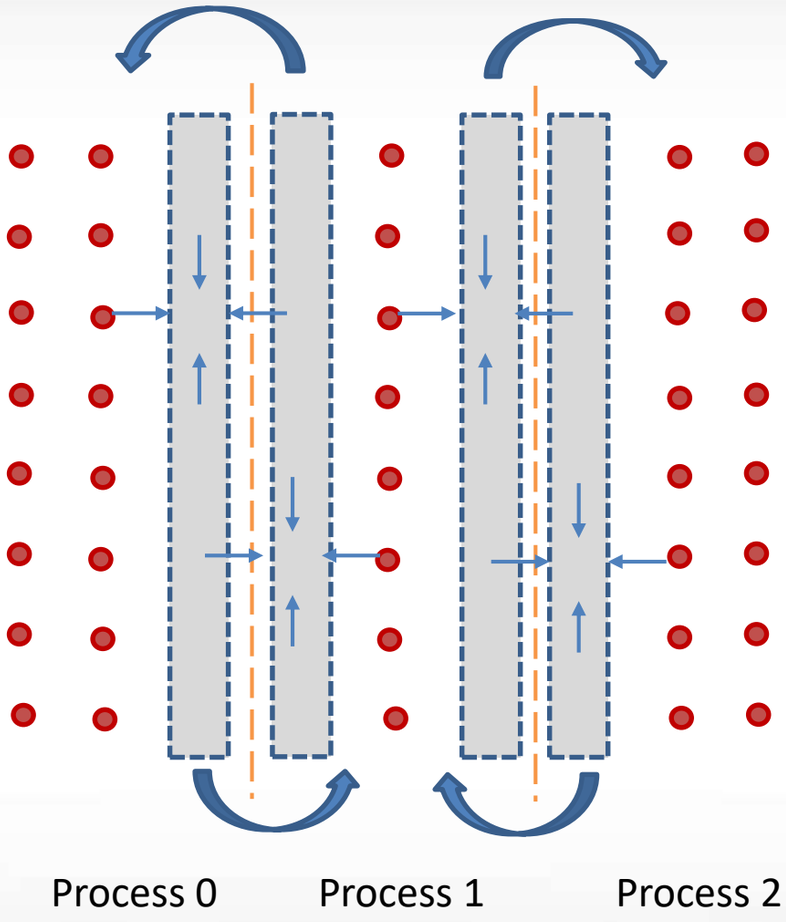
C: decompose along y axis



Fortran: decompose along x axis



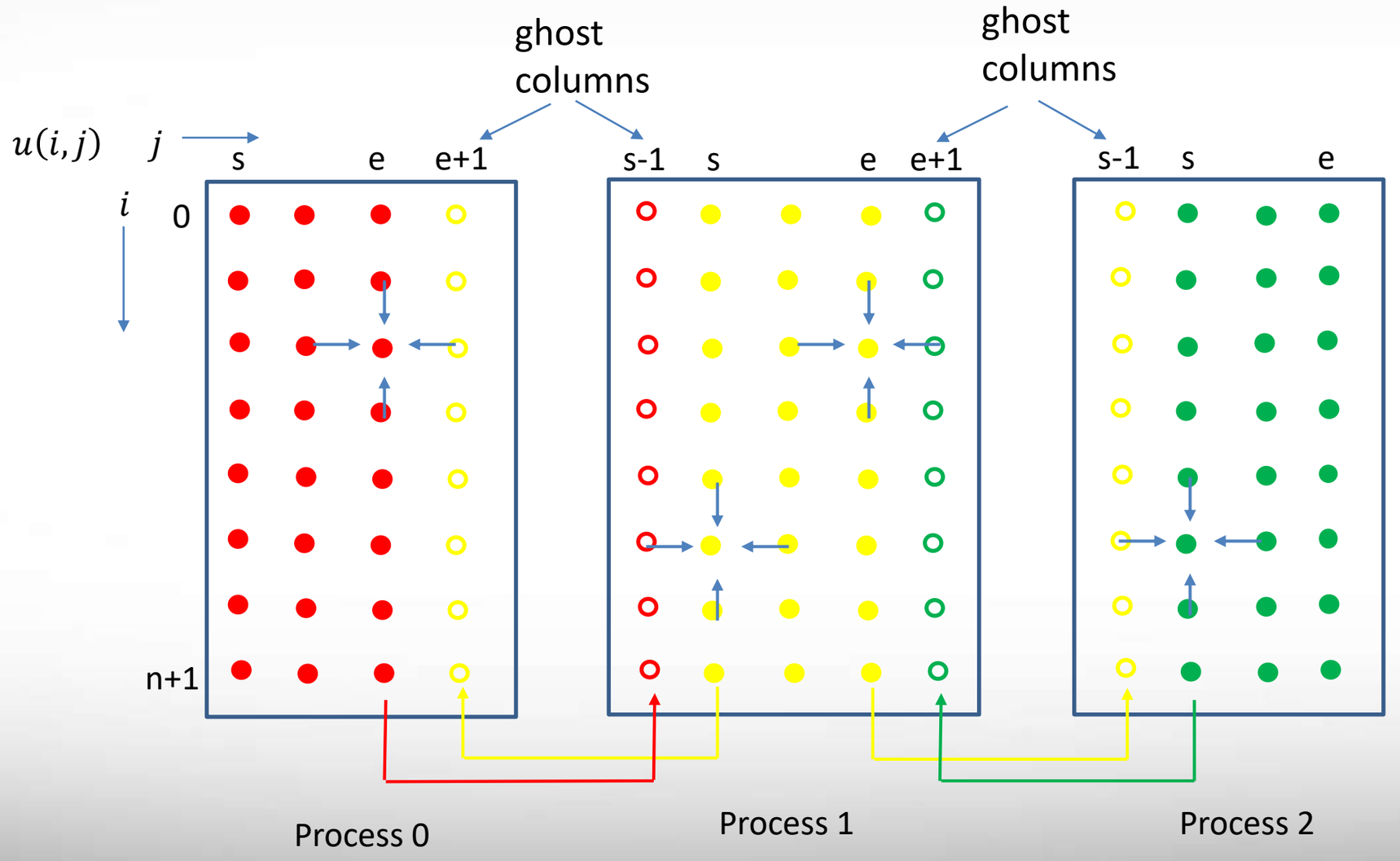
1d-Domain Decomposition



Border columns need to be copied into the memory space of the neighboring processes for the five-point stencil calculation. Same for row-wise decomposition.

1d-Domain Decomposition

Exchanged border columns are stored in ‘ghost’ columns in each process to be used in five-point stencil calculation

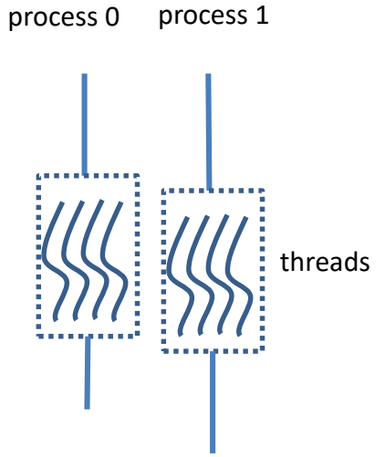


MPI/OpenMP Hybrid Programming

- 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_FUNNELD — ex2_funnel.c
  - MPI_THREAD_SERIALIZED — ex2_serialized.c
  - MPI_THREAD_MULTIPLE — ex2_multiple.c

Example 9: poisson_1d_hybrid

- Master-only implementation
- Performance comparison using a 800x800 mesh

Pure MPI Single-node		Pure MPI 2-node		Hybrid OMP_NUM_THREADS=2		Hybrid OMP_NUM_THREADS=4	
np=8	80.37s	np=8,ppn=4	84.14s	np=4	161.18s	np=2	312.33s
np=16	43.24s	np=16,ppn=8	46.04s	np=8	84.57s	np=4	162.67s
np=20	36.48s	np=20,ppn=10	39.19s	np=10	72.21s	np=5	130.51s
		np=40,ppn=20	25.88s	np=20,ppn=10	52.37s	np=10,ppn=5	74.02s

- Not all program can benefit from hybrid programming

Acknoledgement

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