Introduction to MPI

Preeti Malakar

pmalakar@cse.iitk.ac.in

An Introductory Course on High-Performance Computing in Science and Engineering

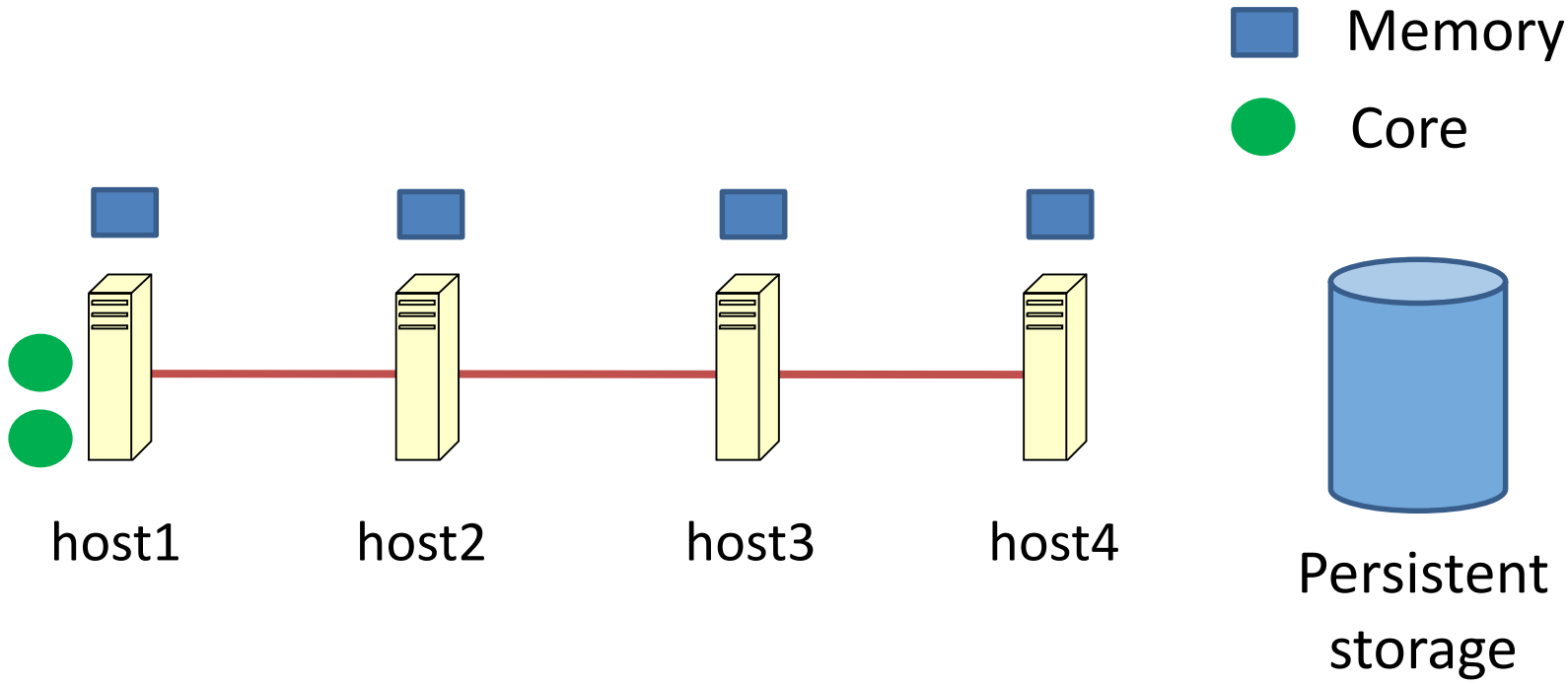
25th February 2019

Parallel Programming Models

Libraries	MPI, TBB, Pthread, OpenMP, …
New languages	Haskell, X10, Chapel, …
Extensions	Coarray Fortran, UPC, Cilk, OpenCL, …

- Shared memory
  - OpenMP, Pthreads, …
- Distributed memory
  - MPI, UPC, …
- Hybrid
  - MPI + OpenMP

Hardware and Network Model



- Interconnected systems
- Distributed memory
- NO centralized server/master

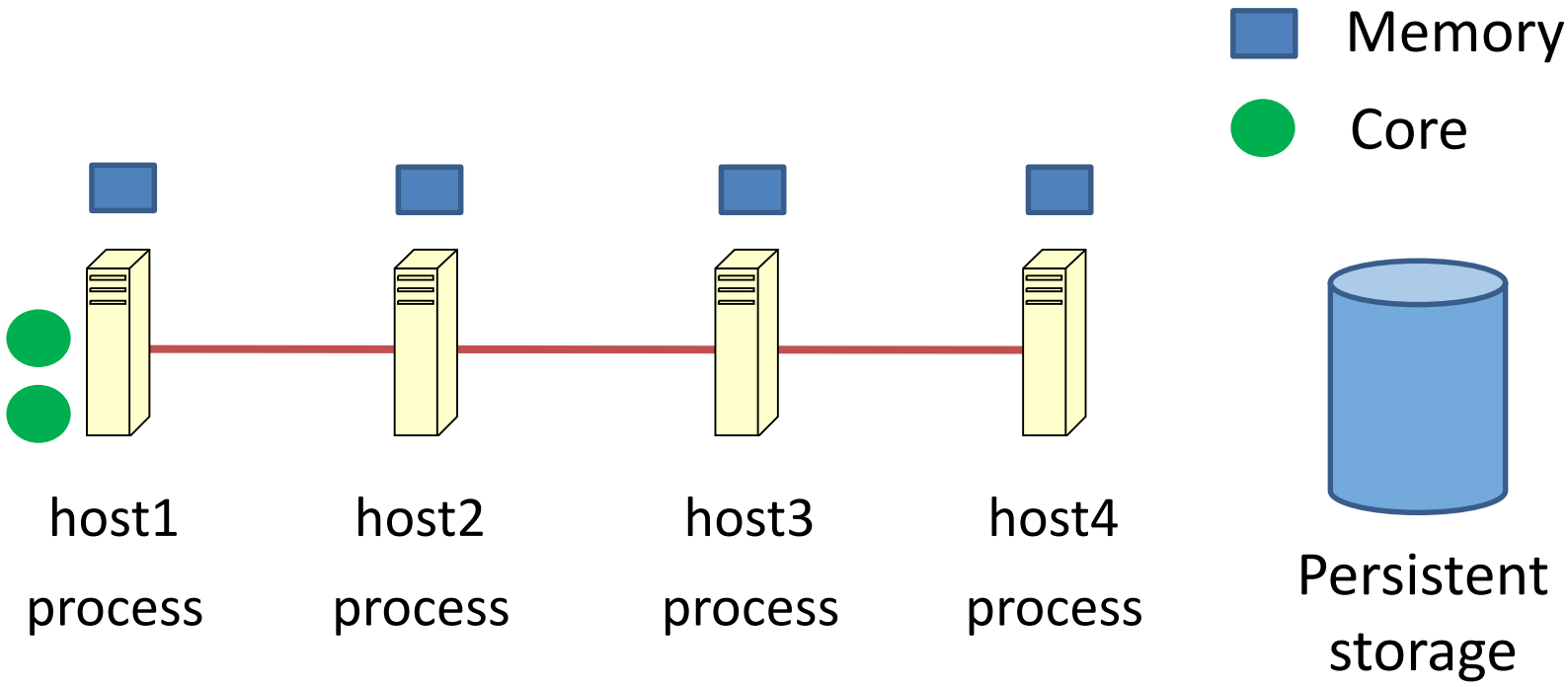
Message Passing Interface (MPI)

- Standard for message passing in a distributed memory environment
- Efforts began in 1991 by Jack Dongarra, Tony Hey, and David W. Walker
- MPI Forum
  - Version 1.0: 1994
  - Version 2.0: 1997
  - Version 3.0: 2012

MPI Implementations

- MPICH (ANL)
- MVAPICH (OSU)
- Intel MPI
- OpenMPI

MPI Processes



MPI Internals

Process Manager

- Start and stop processes in a scalable way
- Setup communication channels for parallel processes
- Provide system-specific information to processes

Job scheduler

- Schedule MPI jobs on a cluster/supercomputer
- Allocate required number of nodes
  - Two jobs generally do not run on the same core
- Enforce other policies (queuing etc.)

Communication Channels

- Sockets for network I/O
- MPI handles communications, progress etc.

Reference: Design and Evaluation of Nemesis, a Scalable, Low-Latency, Message-Passing Communication Subsystem by Buntinas et al.

Message Passing Paradigm

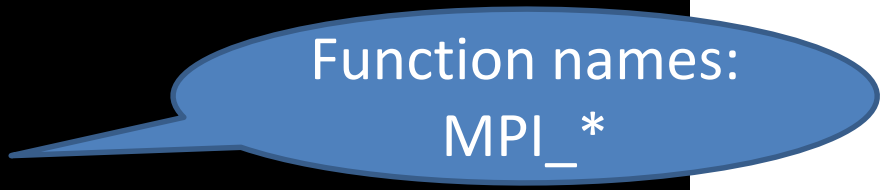
- Message sends and receives
- Explicit communication

Communication types

- Blocking
- Non-blocking

Getting Started

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

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

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello I am rank %d out of %d processes\n", rank, size);

    // Finalize the MPI environment.
    MPI_Finalize();
}

~
~
```



Initialization and Finalization

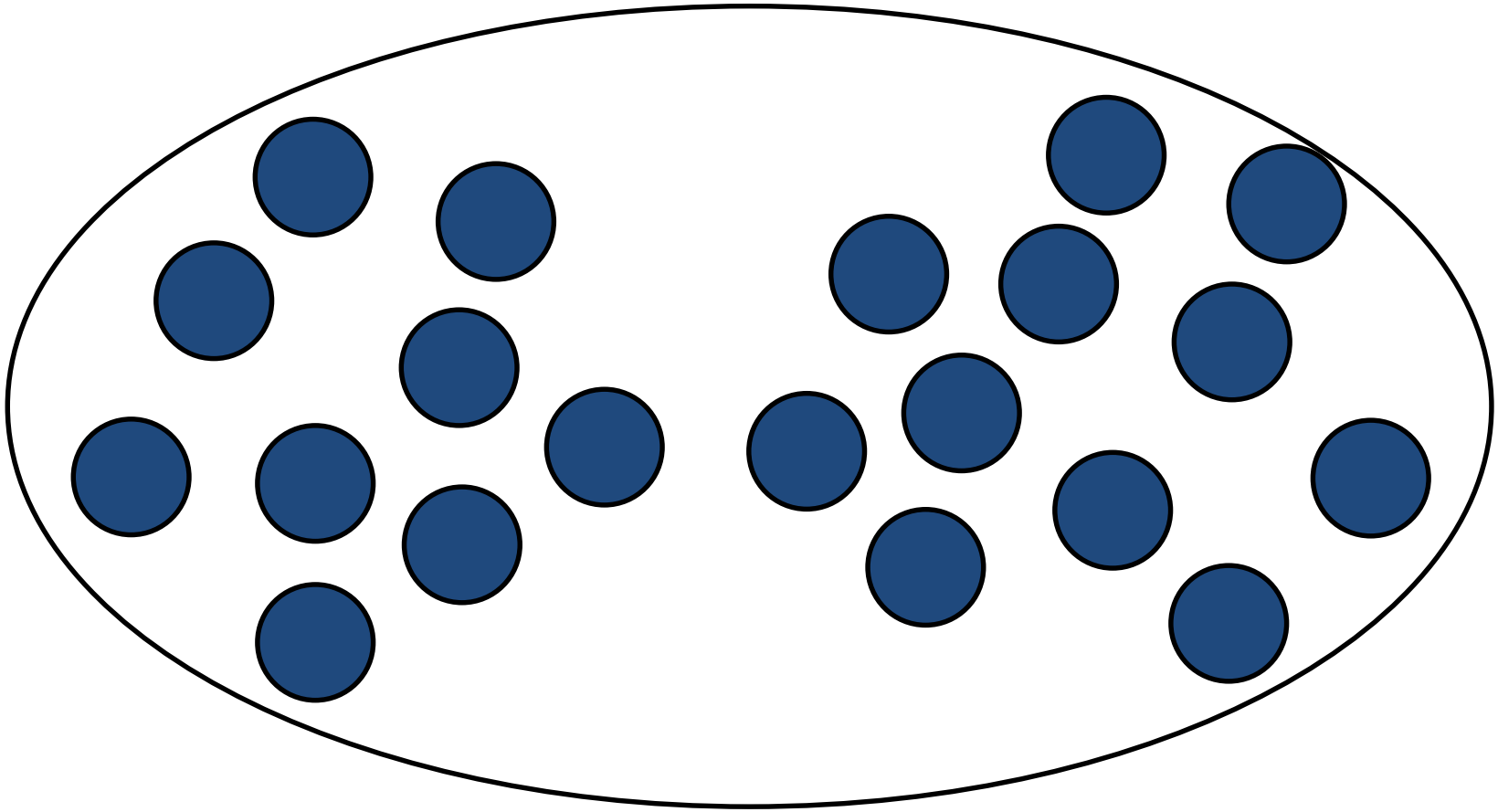
MPI_Init

- gather information about the parallel job
- set up internal library state
- prepare for communication

MPI_Finalize

- cleanup

MPI_COMM_WORLD



Communication Scope

Communicator (communication handle)

- Defines the scope
- Specifies communication context

Process

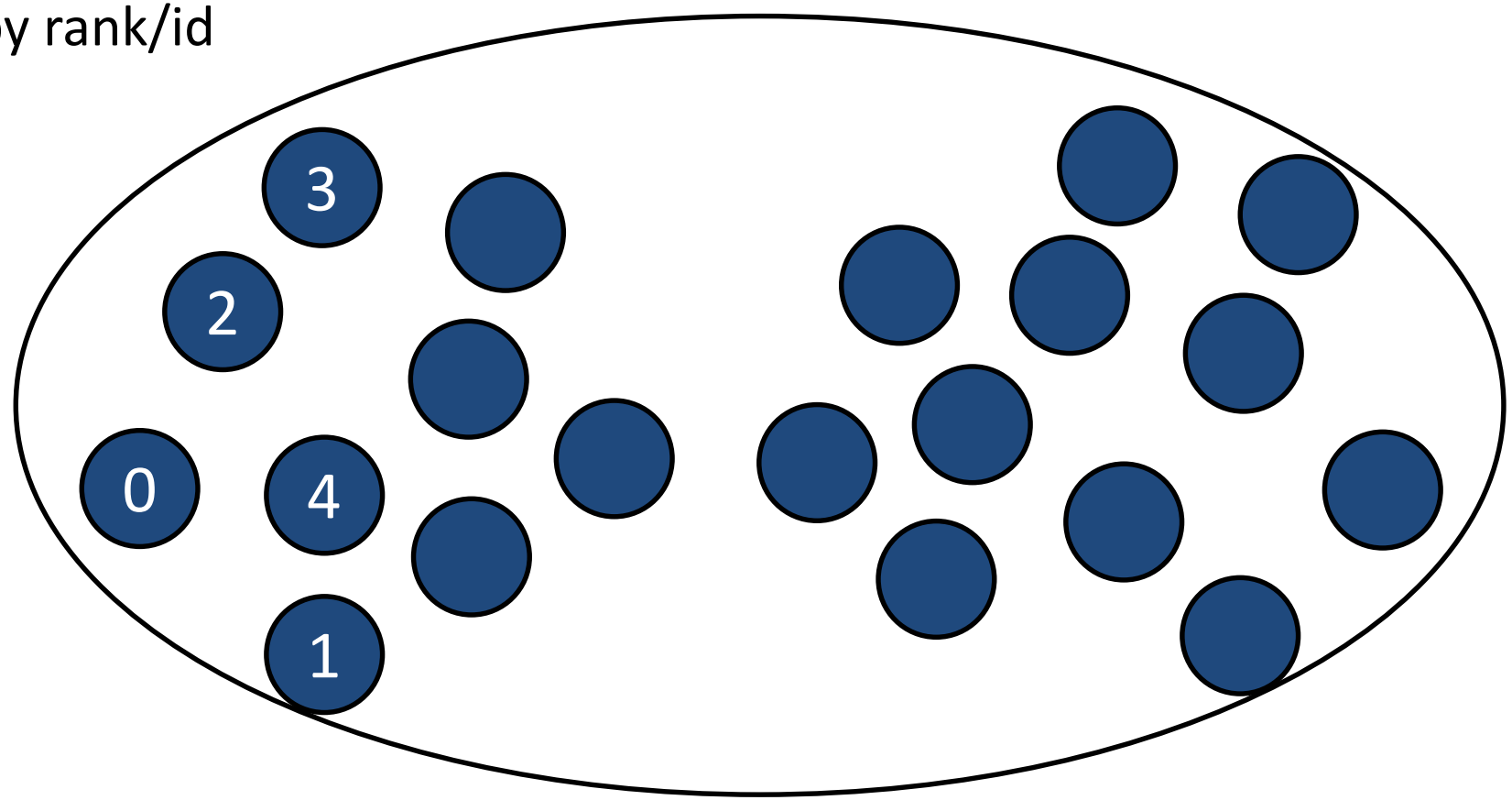
- Belongs to a group
- Identified by a rank within a group

Identification

- MPI_Comm_size – total number of processes in communicator
- MPI_Comm_rank – rank in the communicator

MPI_COMM_WORLD

Process identified
by rank/id



Getting Started

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

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

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Get the rank of the process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello I am rank %d out of %d processes\n", rank, size);

    // Finalize the MPI environment.
    MPI_Finalize();
```

Total number of processes

Rank of a process

MPI Message

- Data and header/envelope
- Typically, MPI communications send/receive messages

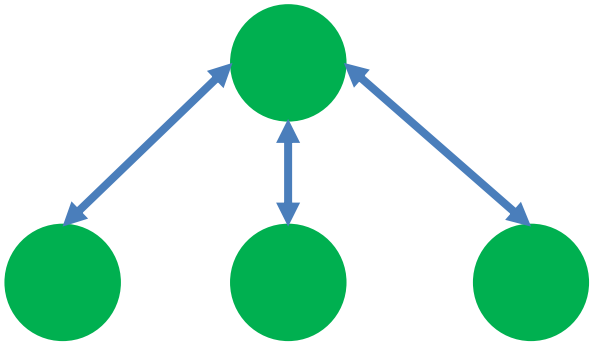
Message Envelope

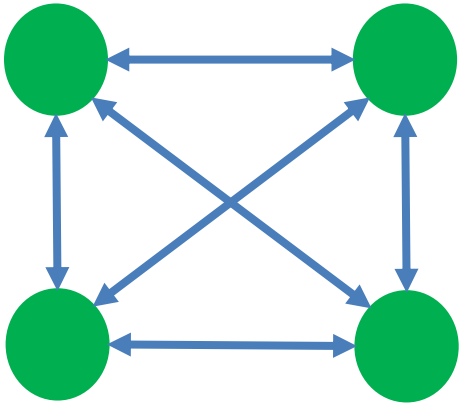
Source: Origin of message
Destination: Receiver of message
Communicator
Tag (0:MPI_TAG_UB)

MPI Communication Types

Point-to-point

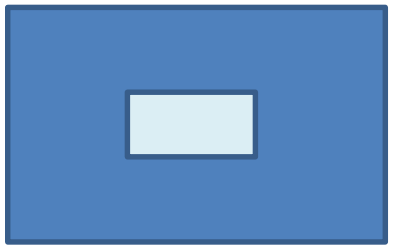
Collective





Point-to-point Communication

- MPI_Send



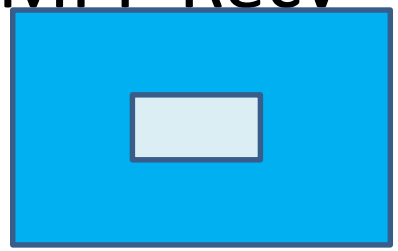
SENDER

Blocking send and receive

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

Tags should match

- MPI_Recv



RECEIVER

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

MPI_Datatype

- MPI_BYTE
- MPI_CHAR
- MPI_INT
- MPI_FLOAT
- MPI_DOUBLE

Example 1

```
MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

// Sender process
if (myrank == 0)      /* code for process 0 */
{
  strcpy (message,"Hello, there");
  MPI_Send (message, strlen(message)+1, MPI_CHAR, 1, 99,
MPI_COMM_WORLD);
}

// Receiver process
else if (myrank == 1)  /* code for process 1 */
{
  MPI_Recv (message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
  printf ("received :%s\n", message);
}
```

MPI_Status

- Source rank
- Message tag
- Message length
  - MPI_Get_count

MPI_ANY_*

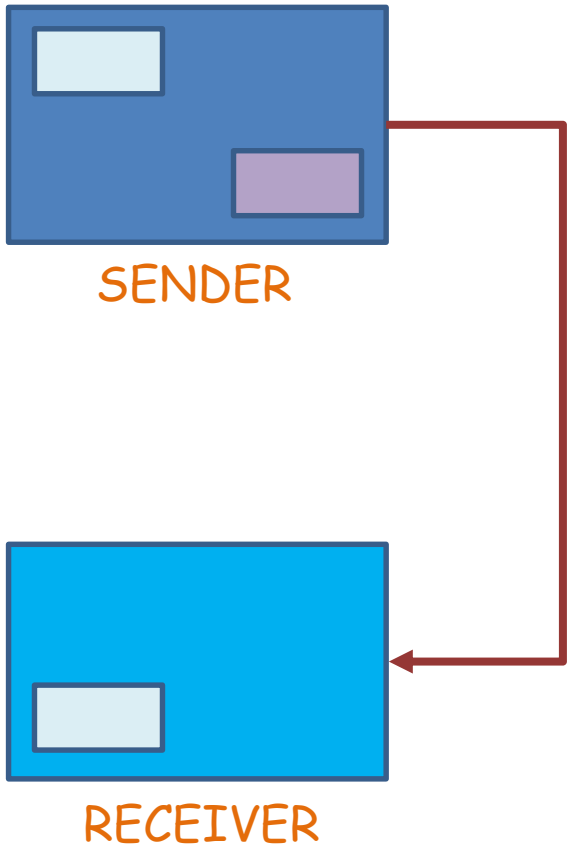
- MPI_ANY_SOURCE
  - Receiver may specify wildcard value for source
- MPI_ANY_TAG
  - Receiver may specify wildcard value for tag

Example 2

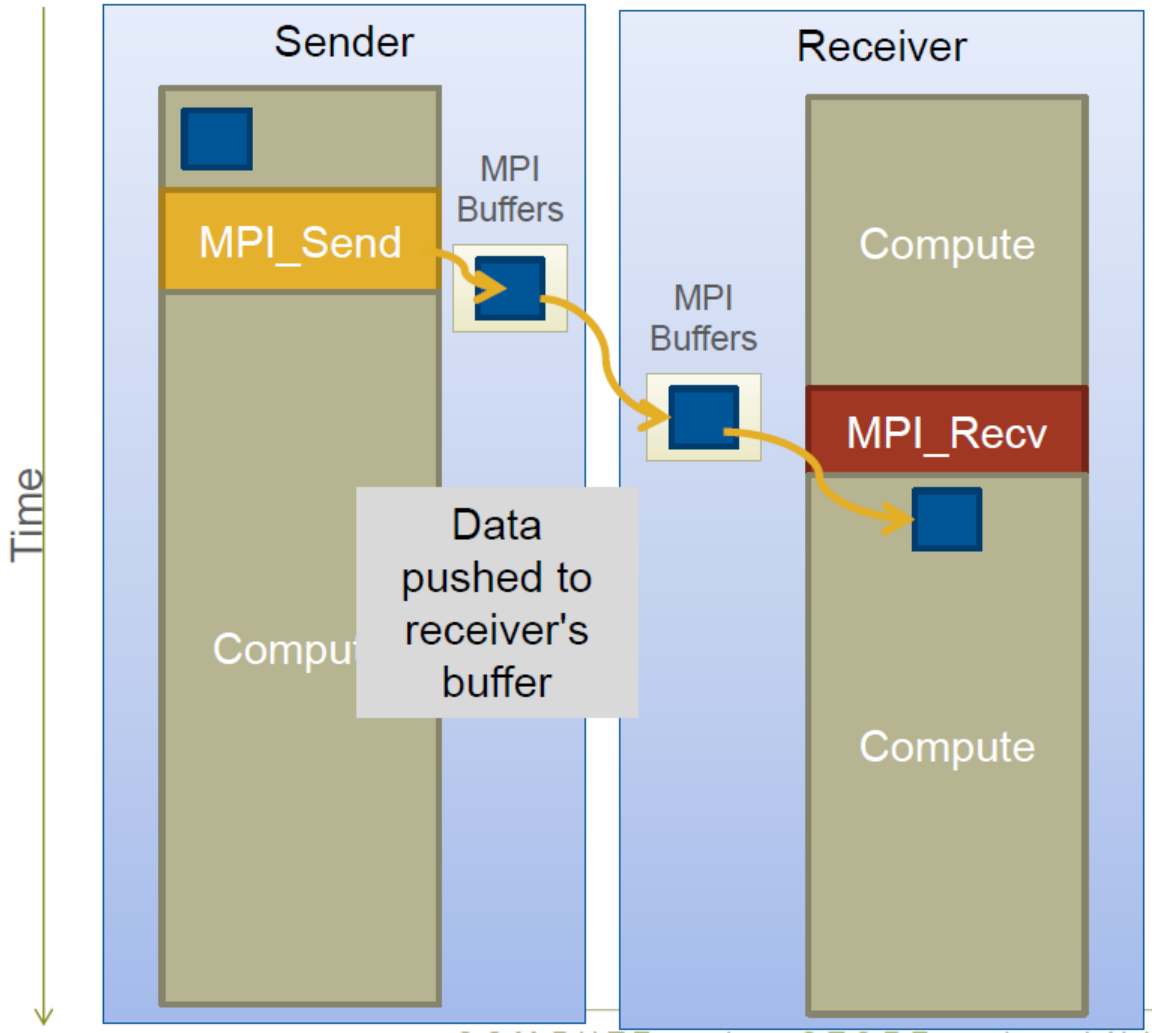
```
MPI_Comm_rank (MPI_COMM_WORLD, &myrank);

// Sender process
if (myrank == 0)      /* code for process 0 */
{
  strcpy (message,"Hello, there");
  MPI_Send (message, strlen(message)+1, MPI_CHAR, 1, 99,
MPI_COMM_WORLD);
}

// Receiver process
else if (myrank == 1)  /* code for process 1 */
{
  MPI_Recv (message, 20, MPI_CHAR, MPI_ANY_SOURCE, 99,
MPI_COMM_WORLD, &status);
  printf ("received :%s\n", message);
}
```

MPI_Send (Blocking)

- Does not return until buffer can be reused
- Message buffering
- Implementation-dependent
- Standard communication mode



Buffering



[Source: Cray presentation]

Safety

MPI_Send MPI_Send	MPI_Recv MPI_Recv	Safe
MPI_Send MPI_Recv	MPI_Send MPI_Recv	Unsafe
MPI_Send MPI_Recv	MPI_Recv MPI_Send	Safe
MPI_Recv MPI_Send	MPI_Recv MPI_Send	Unsafe

Message Protocols

- Short
  - Message sent with envelope/header
- Eager
  - Send completes without acknowledgement from destination
  - Small messages – typically 128 KB (at least in MPICH)
  - MPIR_CVAR_CH3_EAGER_MAX_MSG_SIZE (check mpivars)
- Rendezvous
  - Requires an acknowledgement from a matching receive
  - Large messages

Other Send Modes

- MPI_Bsend Buffered
  - May complete before matching receive is posted
- MPI_Ssend Synchronous
  - Completes only if a matching receive is posted
- MPI_Rsend Ready
  - Started only if a matching receive is posted

Non-blocking Point-to-Point

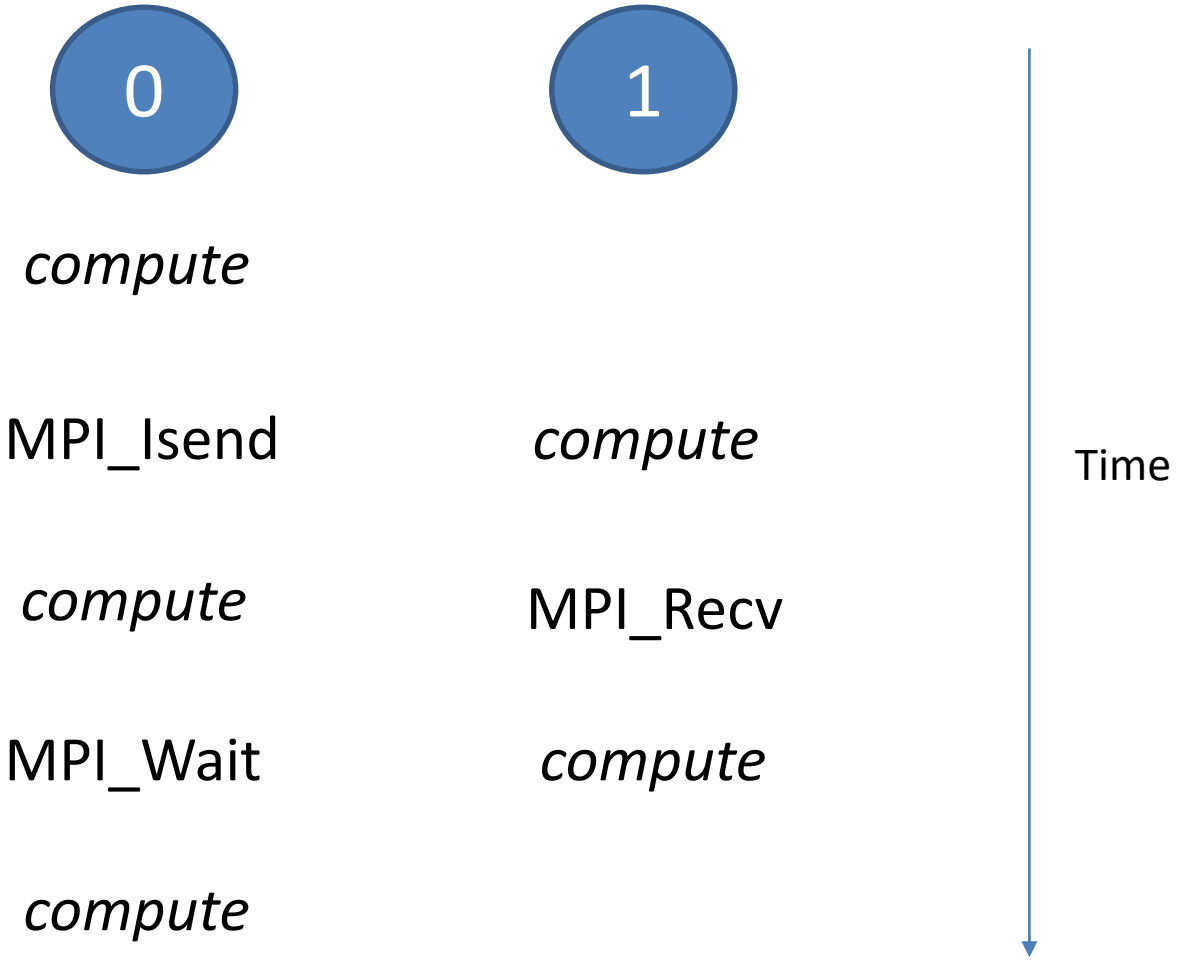
- MPI_Isend (buf, count, datatype, dest, tag, comm, request)
- MPI_Irecv (buf, count, datatype, source, tag, comm, request)
- MPI_Wait (request, status)

MPI_Isend
MPI_Recv

MPI_Isend
MPI_Recv

Safe

Computation Communication Overlap



Collective Communications

- Must be called by all processes that are part of the communicator

Types

- Synchronization (MPI_Barrier)
- Global communication (MPI_Bcast, MPI_Gather, …)
- Global reduction (MPI_Reduce, …)

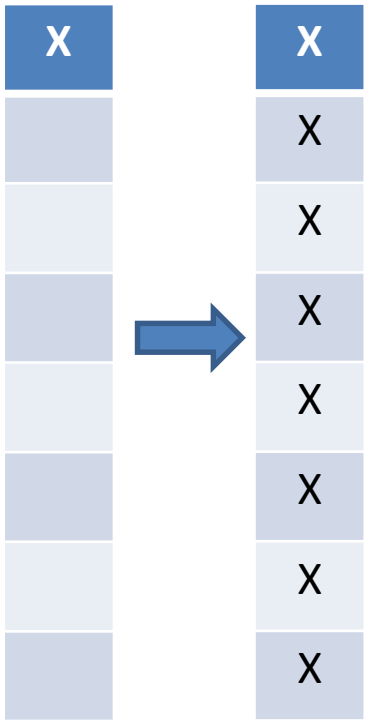
Barrier

- Synchronization across all group members
- Collective call
- Blocks until all processes have entered the call
- MPI_Barrier (comm)

Broadcast

- Root process sends message to all processes
- Any process can be root process but has to be the same in all processes
- int MPI_Bcast (buffer, count, datatype, root, comm)
- Number of elements in buffer – count
- buffer – Input or output?

Q: Can you use point-to-point communication for the same?

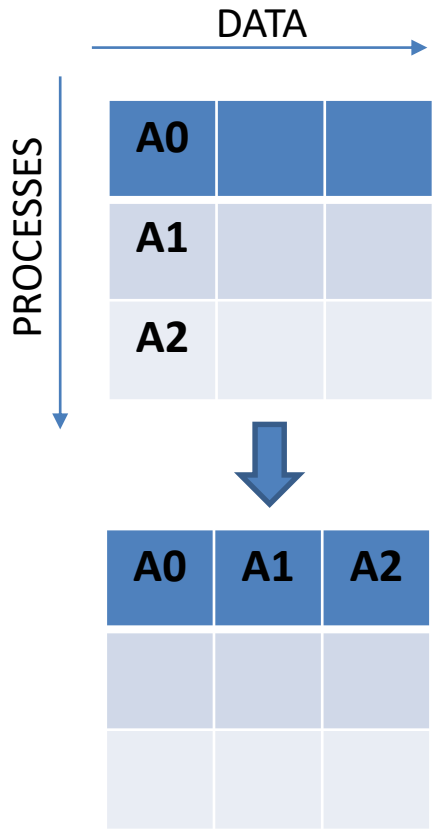


Example 3

```
int rank, size, color;
MPI_Status status;

MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);

color = rank + 2;
int oldcolor = color;
MPI_Bcast (&color, 1, MPI_INT, 0, MPI_COMM_WORLD);

printf ("%d: %d color changed to %d\n", rank, oldcolor, color);
```

Gather

- Gathers values from all processes to a root process
- int MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- Arguments recv* not relevant on non-root processes

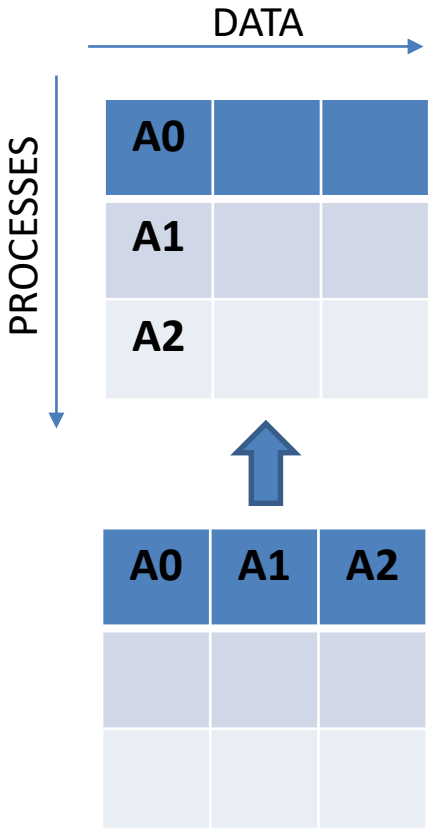


Example 4

```
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  color = rank + 2;

  int colors[size];
  MPI_Gather (&color, 1, MPI_INT, colors, 1, MPI_INT, 0,
MPI_COMM_WORLD);

  if (rank == 0)
   for (i=0; i<size; i++)
    printf ("color from %d = %d\n", i, colors[i]);
```

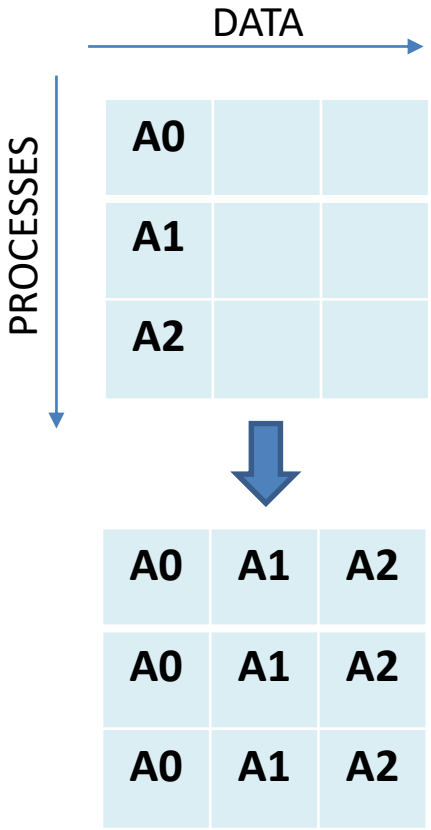
Scatter

- Scatters values to all processes from a root process
- int MPI_Scatter (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- Arguments send* not relevant on non-root processes
- Output parameter – recvbuf



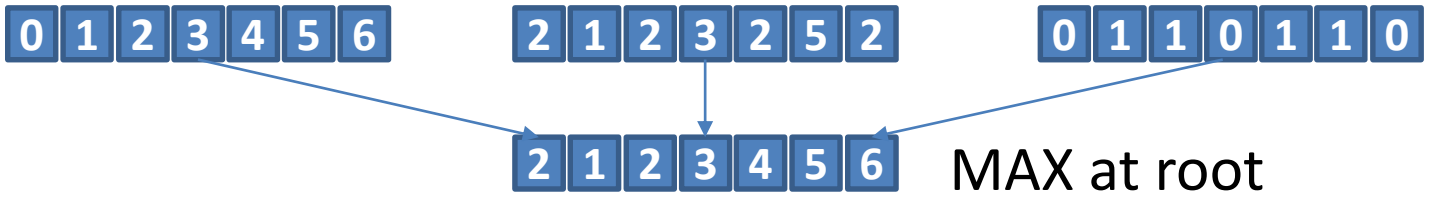
Allgather

- All processes gather values from all processes
- int MPI_Allgather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)



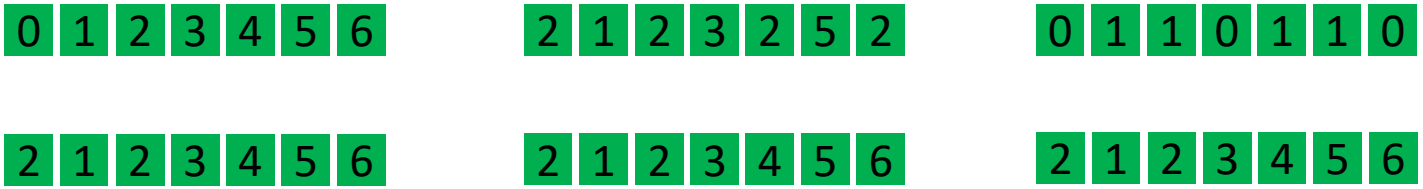
Reduce

- MPI_Reduce (inbuf, outbuf, count, datatype, op, root, comm)
- Combines element in inbuf of each process
- Combined value in outbuf of root
- op: MIN, MAX, SUM, PROD, …



Allreduce

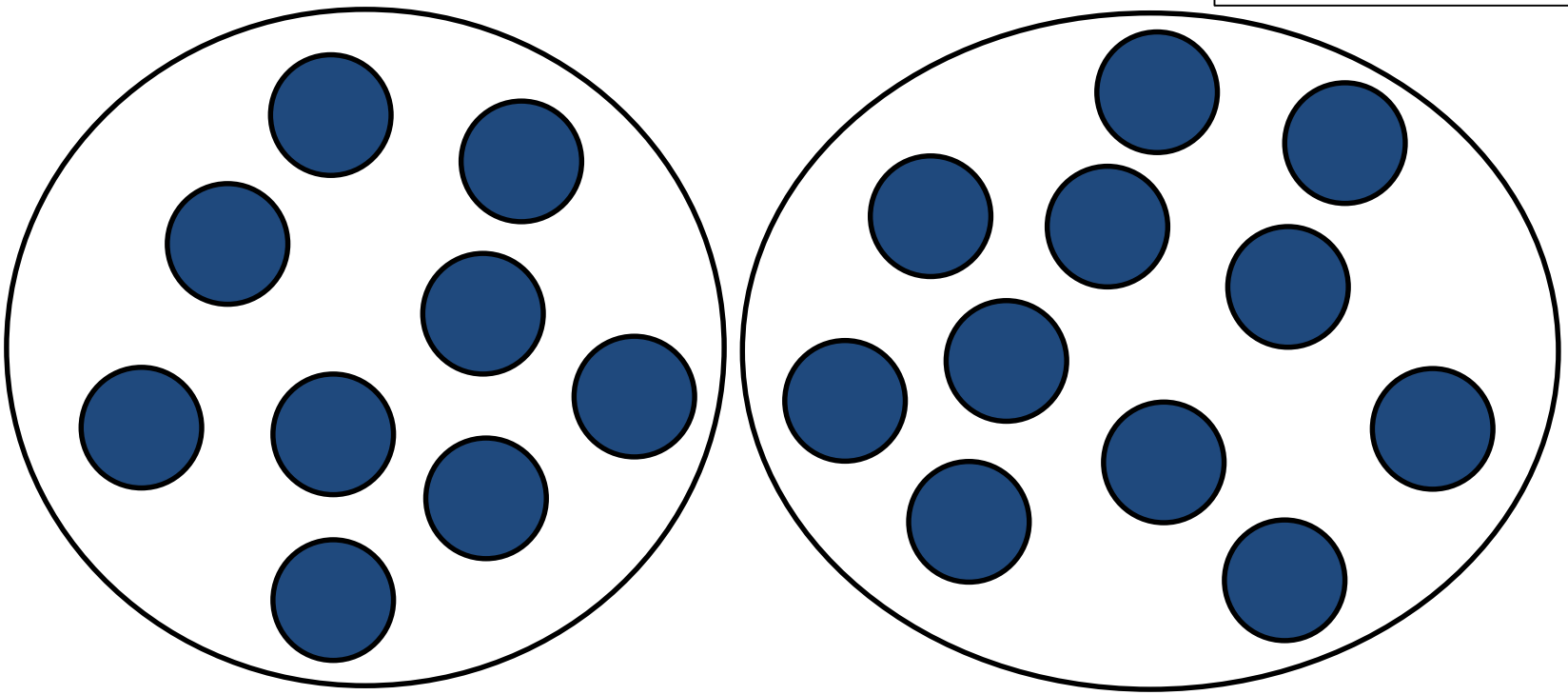
- MPI_Allreduce (inbuf, outbuf, count, datatype, op, comm)
- op: MIN, MAX, SUM, PROD, …
- Combines element in inbuf of each process
- Combined value in outbuf of each process



MAX

Sub-communicator

- Logical subset
- Different contexts

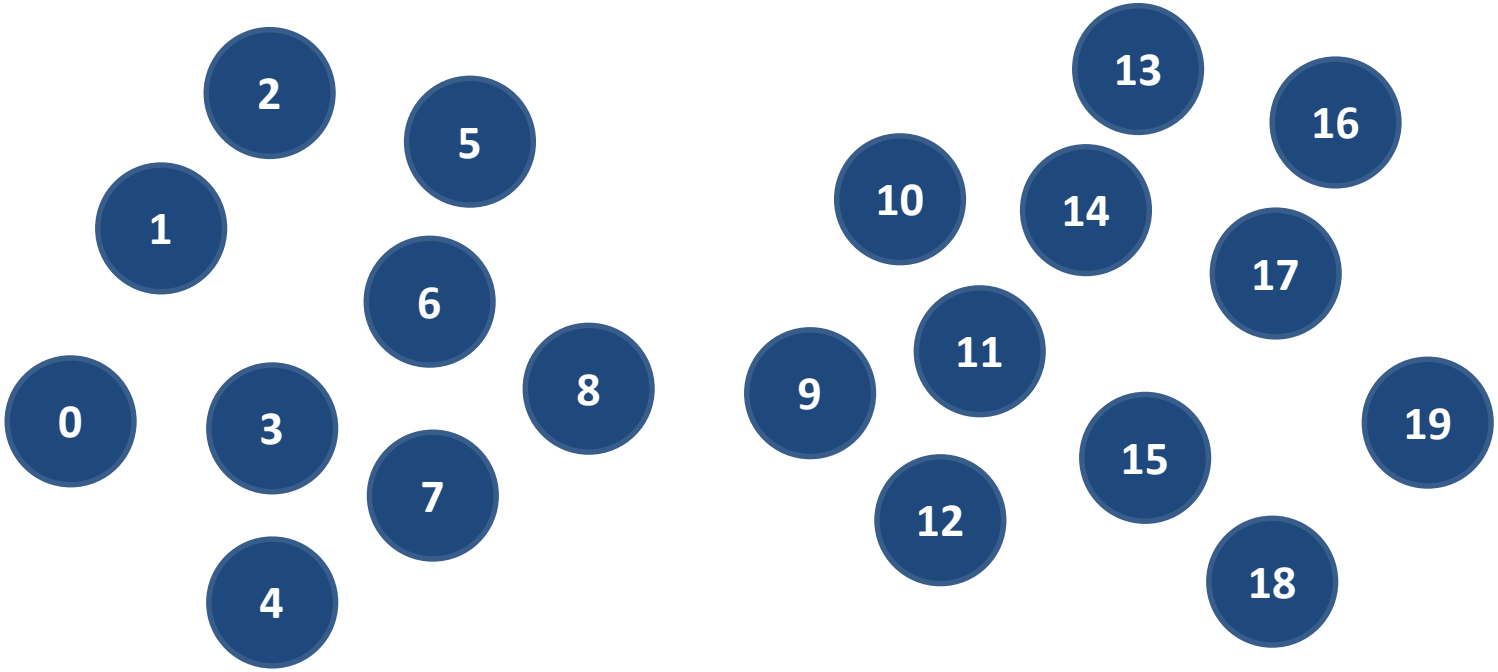


MPI_COMM_SPLIT

MPI_Comm_split (MPI_Comm oldcomm, int color, int key, MPI_Comm *newcomm)

- Collective call
- Logically divides based on *color*
  - Same color processes form a group
  - Some processes may not be part of newcomm (MPI_UNDEFINED)
- Rank assignment based on *key*

Logical subsets of processes



0 → 0
2 → 1
4 → 2
…

1 → 0
3 → 1
5 → 2
…

How do you assign one color to odd processes and another color to even processes ?
color = rank % 2

MPI Programming
Hands-on

How to run an MPI program on a cluster?

MPI process MPI process | MPI process MPI process | MPI process MPI process | MPI process MPI process

4 nodes, ppn=2

mpiexec –n <number of processes> -f <hostfile> ./exe

<hostfile>

host1:2
host2:2
host3:2
…

How to run an MPI program on a managed cluster/supercomputer?

MPI process MPI process | MPI process MPI process | MPI process MPI process | MPI process MPI process

4 nodes, ppn=2

Execution on HPC2010: qsub sub.sh

Practice Examples

- egN directory (N=1,2,3,4)
  - egN.c
- Compile
  - source /opt/software/intel/initpaths intel64
  - make
- Execute
  - qsub sub.sh

Your Code

- Each sub-directory (a1, a2, a3) has
  - sub.sh [Required number of cores mentioned]
  - Makefile
  - .c
  - Edit the .c [Look for “WRITE YOUR CODE HERE”]

Assignments

1. Even processes send their data to odd processes

2. Element-wise sum of distributed arrays

3. Sum of array elements of 2 large arrays
   - Make two groups of processes {0,2,4,6} and {1,3,5,7}
   - The 0^{th} process of each group should distribute its array to the other group members (equal division)
   - All processes sum up their individual array chunks

Reference Material

- Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker and Jack Dongarra, MPI - The Complete Reference, Second Edition, Volume 1, The MPI Core.
- William Gropp, Ewing Lusk, Anthony Skjellum, Using MPI : portable parallel programming with the message-passing interface, 3rd Ed., Cambridge MIT Press, 2014.