PARALLEL PROGRAMMING WITH MPI

Dr Noor Mahammad Sk

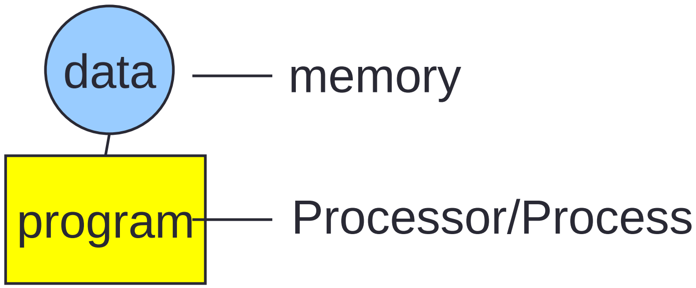
MPI (Message Passing Interface)?

- Standard Message Passing Library Specification (IEEE)
  - For parallel computers, clusters, and heterogeneous networks
  - Not a specific product, compiler specification etc.,
  - Many implementations, MPICH, LAM, OpenMPI
- Portable, with Fortran and C/C++ interfaces
- Many Functions
- Real Parallel Programming
- Notoriously difficult to debug

Hprcse

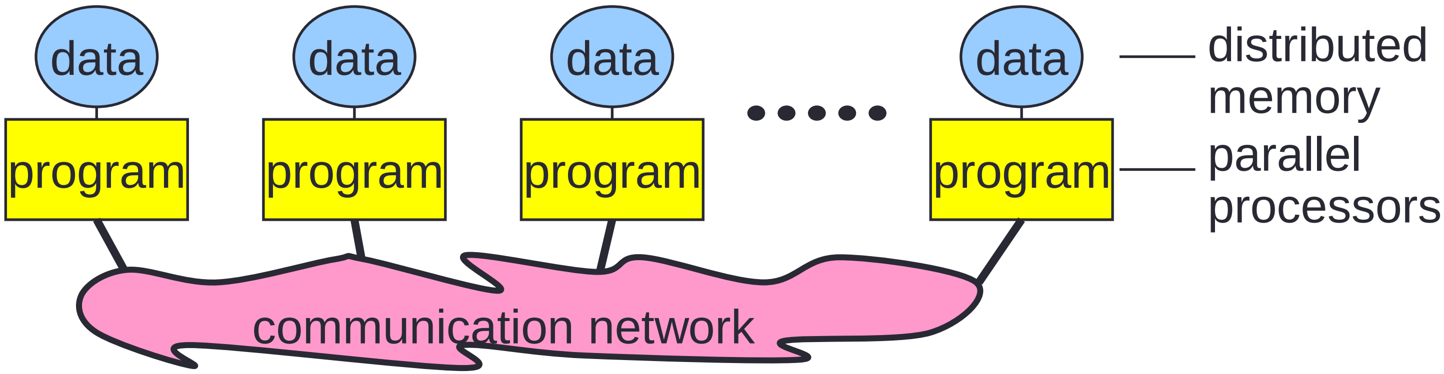
The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**



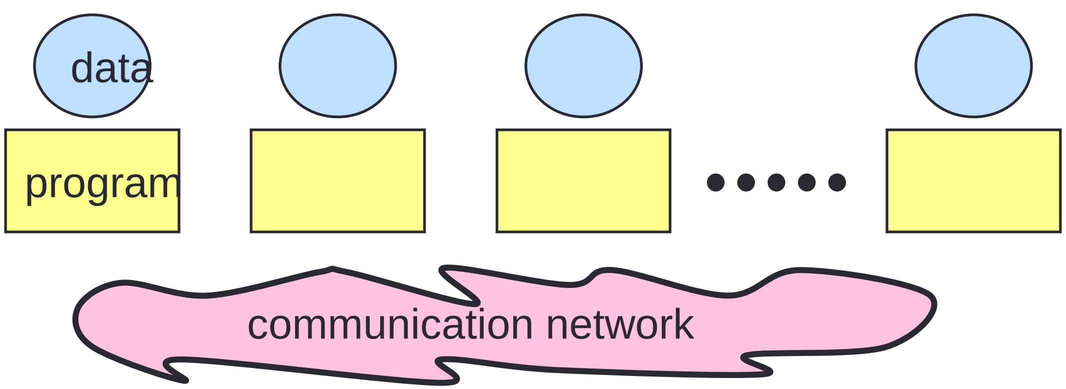
A processor may
run many processes

- Message-Passing Programming Paradigm



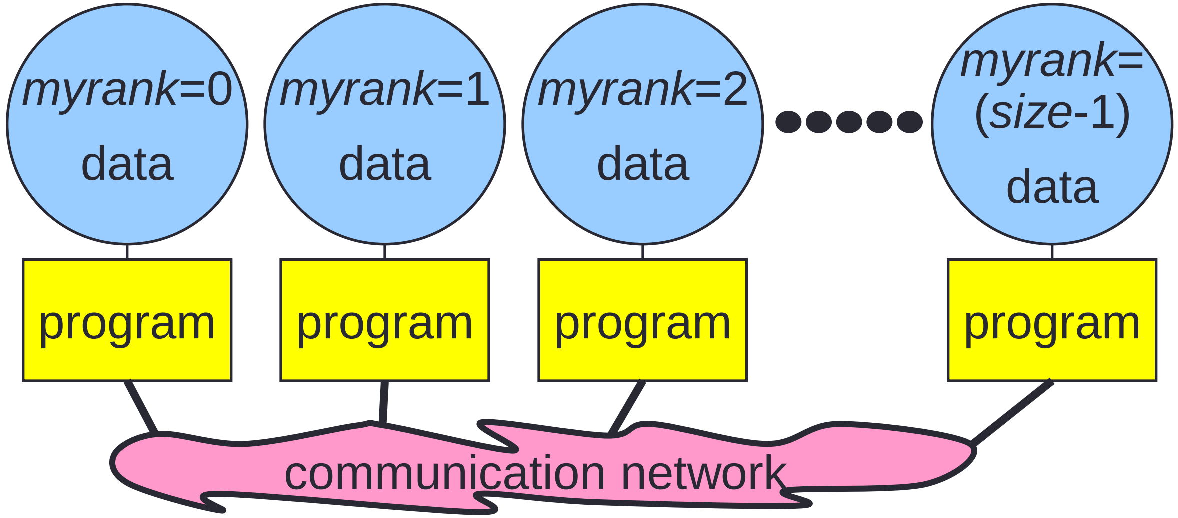
MPI Operation

- A **process** is a program performing a task on a **processor**
- Each processor/process in a message passing program runs a instance/copy of a ***program:***
- Written in a conventional sequential language, e.g., C or Fortran
- Typically a single program operating of multiple dataset
- The variables of each sub-program have
  - The same name
  - But different locations (distributed memory) and different data
  - i.e., all variables are local to a process
- Communicate via special send and receive routines (message passing)



Data and Work Distribution

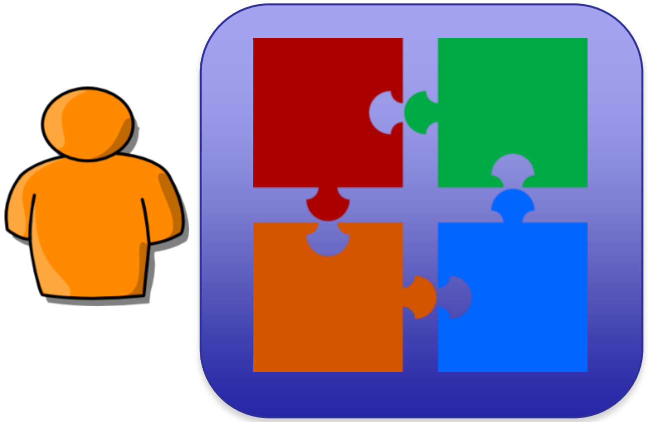
- To communicate together MPI processes need identifiers: **rank = identifier number**
- All distribution decision are based on the **rank**
- i.e., which process works on which data



What is SPMD

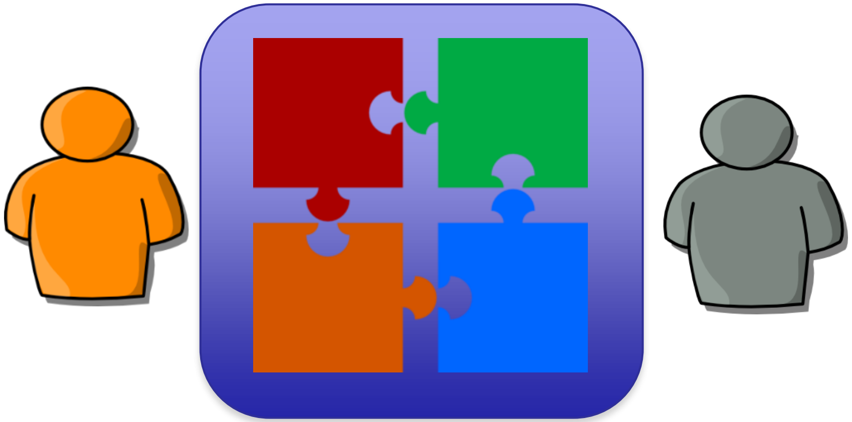
- Single Program Multiple Data
- Same (sub) program runs on each processor
- MPI allows also MPMD, i.e., **Multiple** Program Multiple Data, ...
- But some vendors may be restricted to SPMD
- MPMD can be emulated with SPMD
  - MPMD can be emulated with SPMD

Serial Computing



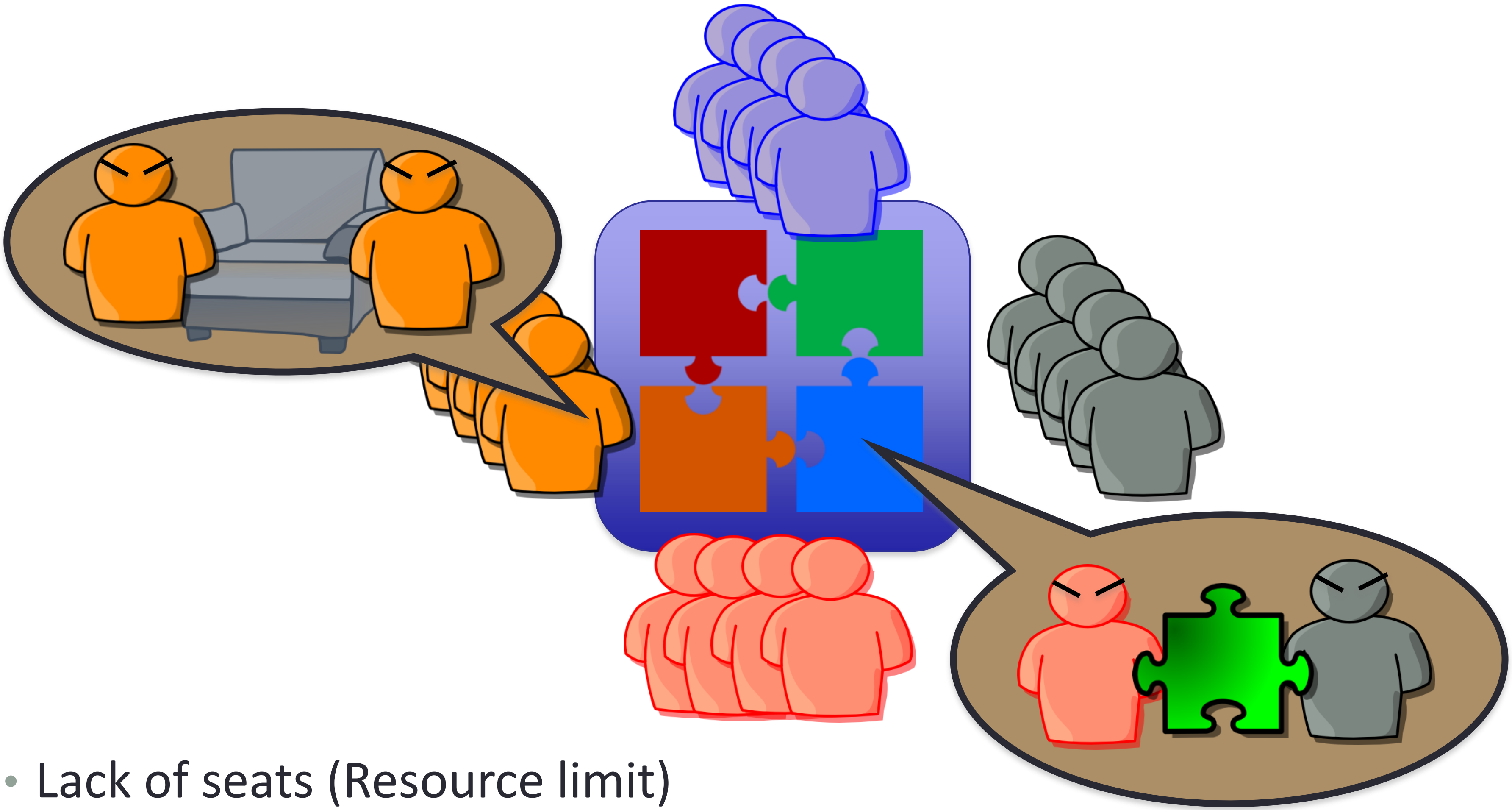
- 1k pieces puzzle
- Takes 10 hours

Parallelism on Shared Memory



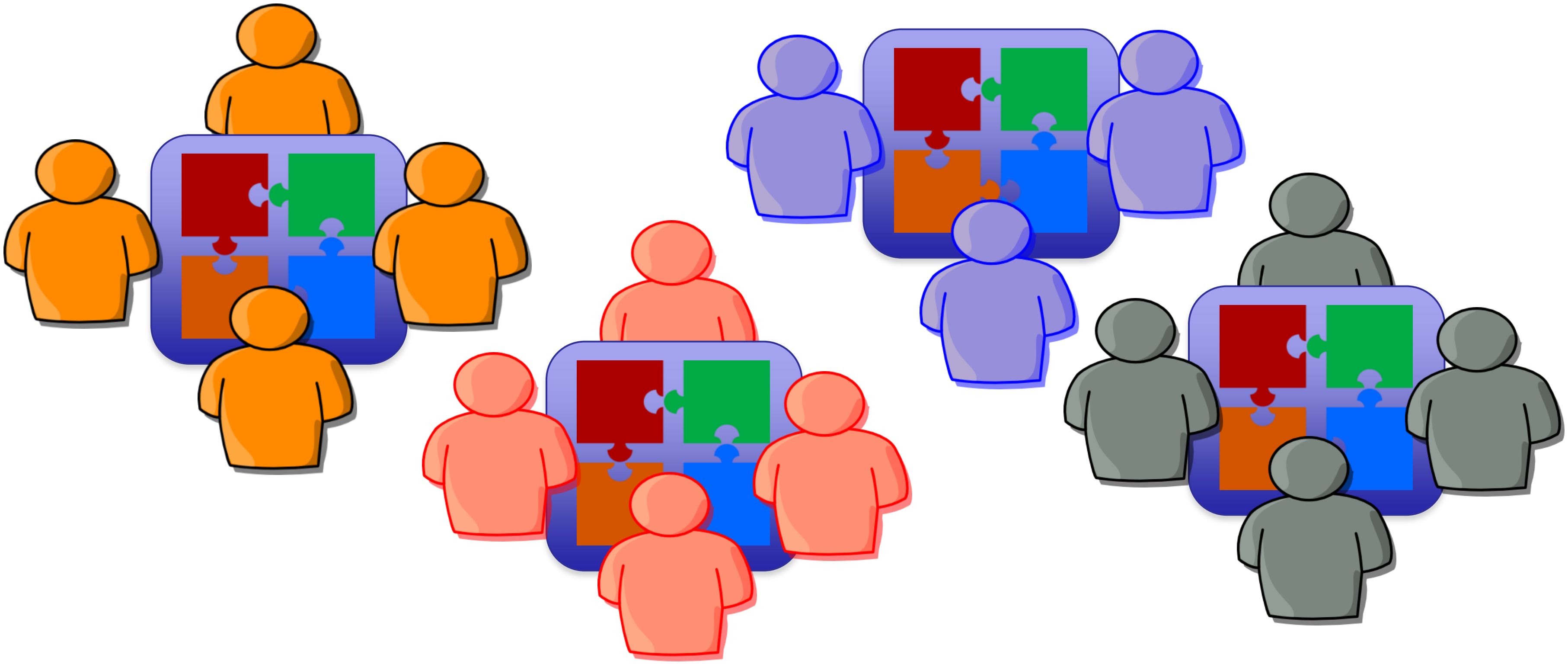
- Orange and green share the puzzle on the same table
- Takes 6 hours
  (not 5 due to communication & contention)

The more, the better??



- Lack of seats (Resource limit)
- More contention among people

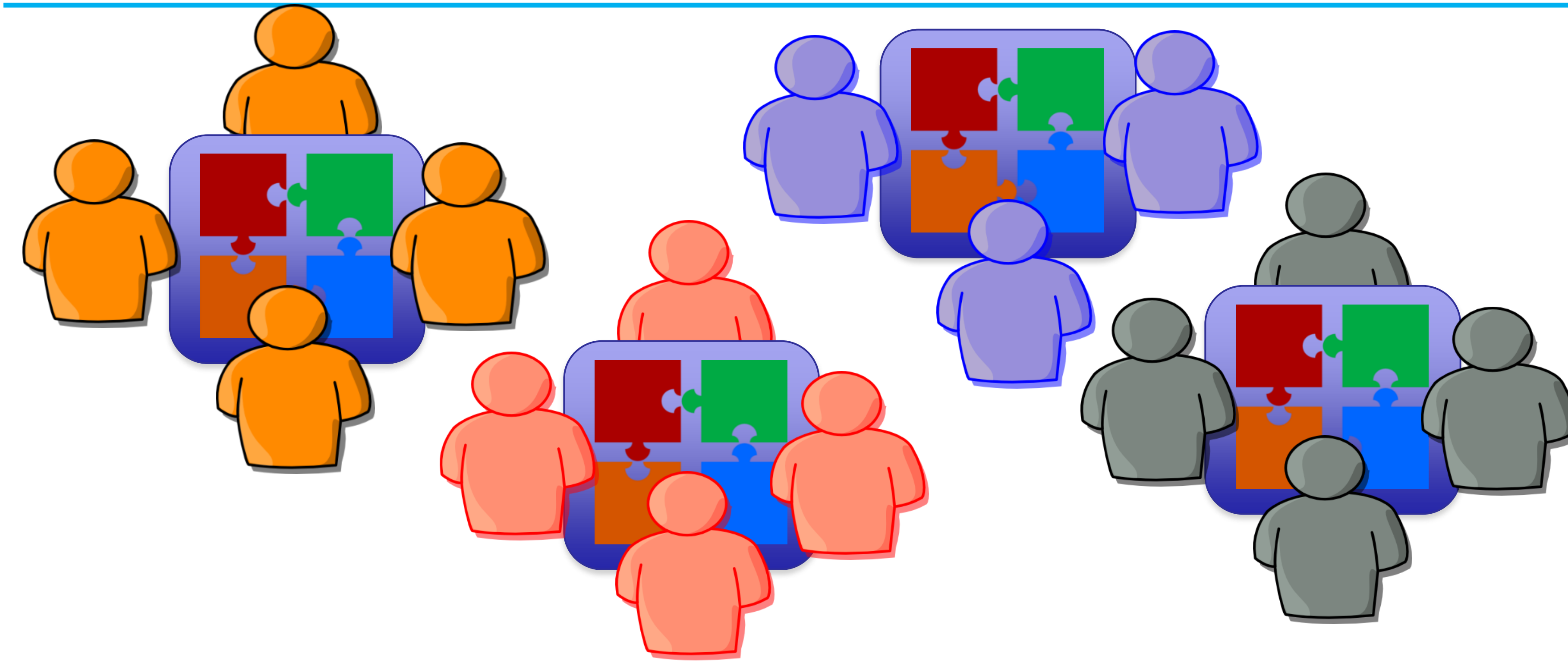
Parallelism on Distributed Systems



- Scalable seats (Scalable Resource)
- Less contention from private memory spaces

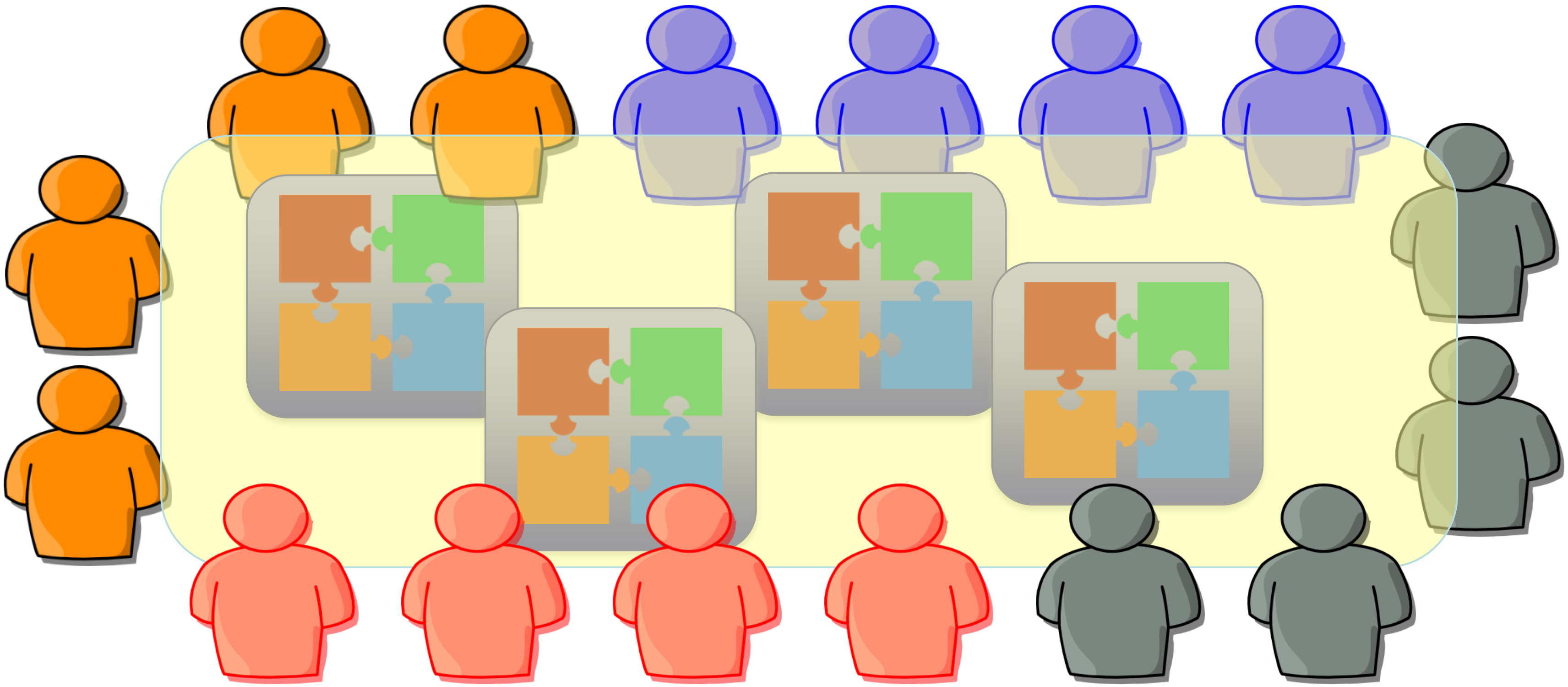
Hprcse

How to share the puzzle?



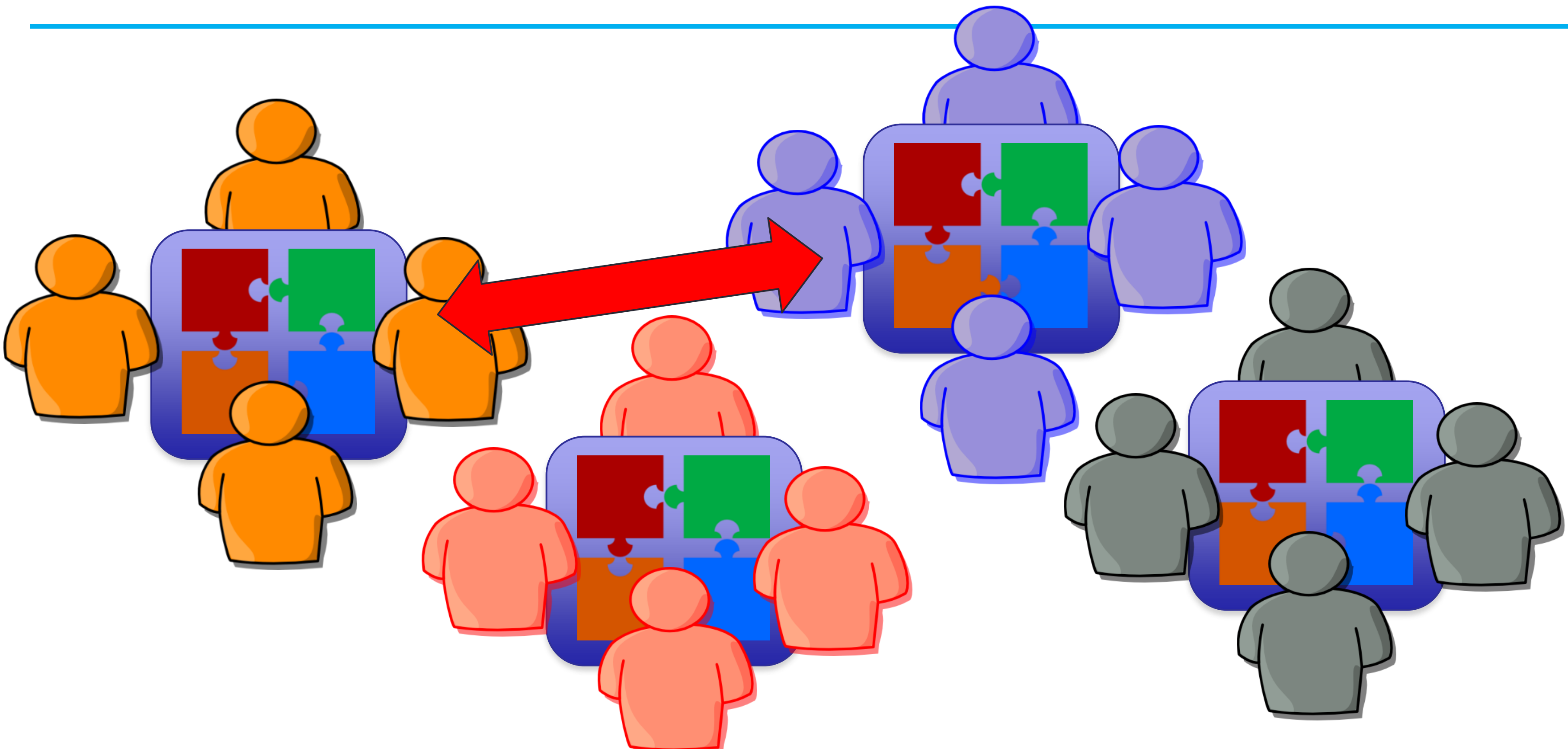
- DSM (Distributed Shared Memory)
- Message Passing

DSM (Distributed Shared Memory)



- Provides shared memory physically or virtually
- Pros - Easy to use
- Cons - Limited Scalability, High coherence overhead

Message Passing



- Pros – Scalable, Flexible
- Cons – Someone says it's more difficult than DSM

MPI (Message Passing Interface)

- A standard message passing specification for the vendors to implement
- **Context**: distributed memory parallel computers
  - Each processor has its own memory and cannot access the memory of other processors
  - Any data to be shared must be explicitly transmitted from one to another
- Most message passing programs use the *single program multiple data (SPMD)* model
  - Each processor executes the same set of instructions
  - Parallelization is achieved by letting each processor operation a different piece of data
  - MIMD (Multiple Instructions Multiple Data*)*

SPMD example

```
main(int argc, char **argv){
 if(process is assigned Master role){
      /* Assign work and coordinate workers
 and collect results */
      MasterRoutine(/*arguments*/);
    } else {  /* it is worker process */
      /* interact with master and other
 workers. Do the work and send results to
 the master*/
      WorkerRoutine(/*arguments*/);
    }
}
```

Why MPI?

- Small
  - Many programs can be written with only 6 basic functions
- Large
  - MPI’s extensive functionality from many functions
- Scalable
  - Point-to-point communication
- Flexible
  - Don’t need to rewrite parallel programs across platforms

What we need to know

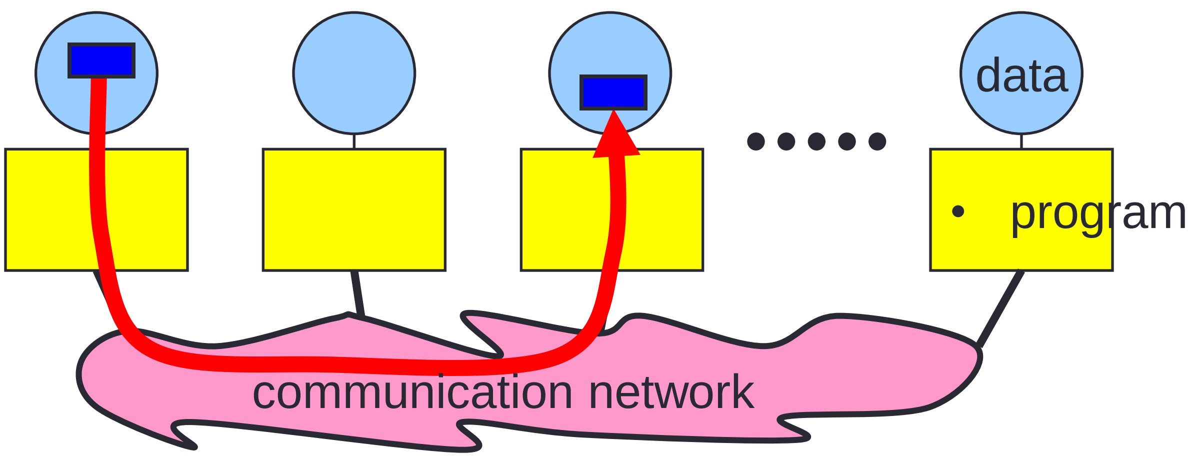
Hprcse

Emulation of MPMD

```
main(int argc, char **argv)
{
    if (myrank < .... /* process should run the ocean model */)
     {
          ocean( /* arguments */ );
     }else{
          weather( /* arguments */ );
     }
  }
```

Message Passing

- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
- Sending Process, Receiving process → i.e., the ranks
- Source location, Destination Location
- Source Data type, Destination Data type
- Source Data Size, Destination buffer size



Access

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - phone line
  - mail box
  - fax machine
  - etc.
- MPI:
  - program must be linked with an MPI library
  - program must be started with the MPI startup tool

Basic functions

FUNCTION	DESCRIPTION
`int MPI_Init(int *argc, char **argv)`	Initialize MPI
`int MPI_Finalize()`	Exit MPI
`int MPI_Comm_size(MPI_Comm comm, int *size)`	Determine number of processes within a comm
`int MPI_Comm_rank(MPI_Comm comm, int *rank)`	Determine process rank within a comm
`int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`	Send a message
`int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)`	Receive a message

Communicator

- An identifier associated with a group of processes
  - Each process has a unique rank within a specific communicator from 0 to (nprocesses-1)
  - Always required when initiating a communication by calling an MPI function
- Default: **`MPI_COMM_WORLD`**
  - Contains all processes
- Several communicators can co-exist
  - A process can belong to different communicators at the same time

Hello World

```
#include "mpi.h"
int main( int argc, char *argv[] ) {
 int nproc, rank;
 MPI_Init (&argc,&argv); /* Initialize MPI */

 MPI_Comm_size(MPI_COMM_WORLD,&nproc); /* Get Comm Size*/
 MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* Get rank */

 printf("Hello World from process %d\n", rank);

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

Hprcse

How to compile

- Need to tell the compiler where to find the MPI include files and how to link to the MPI libraries.
- Fortunately, most MPI implementations come with scripts that take care of these issues:
  - mpicc mpi_code.c –o a.out
- Two widely used (and free) MPI implementations
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich)
  - OPENMPI (http://www.openmpi.org)

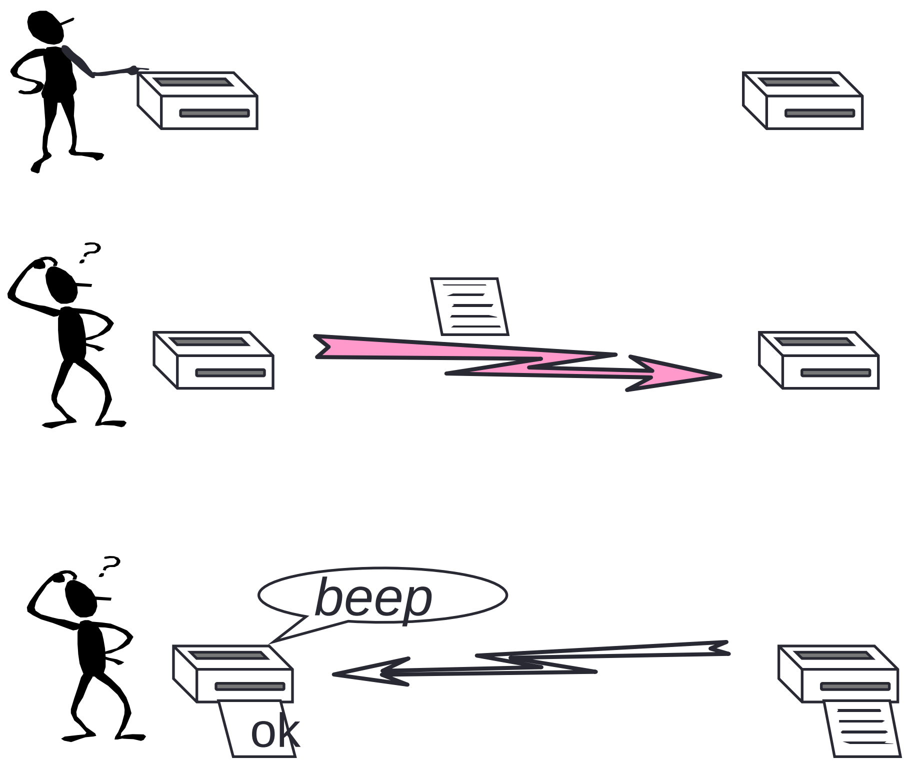
Hprcse

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send

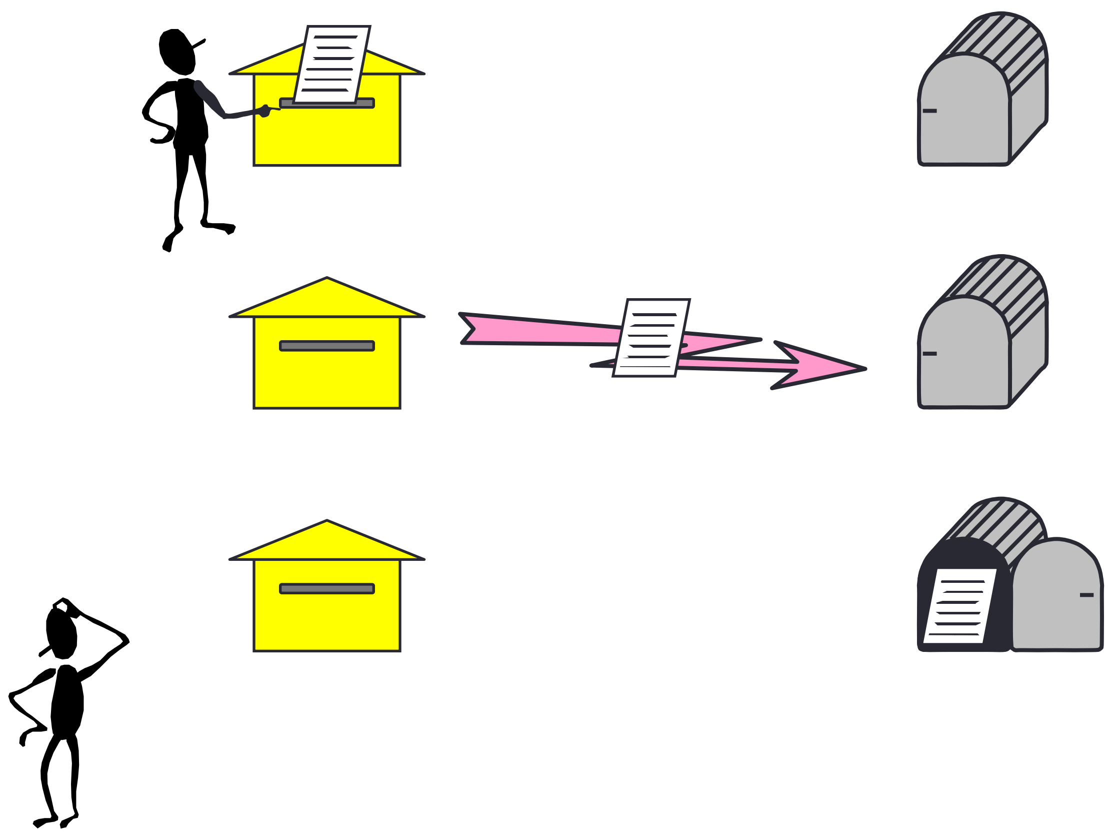
Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.



Buffered = Asynchronous Sends

- Only know when the message has left.



Blocking Operations

- **Some sends/receives may block until another process acts**:
- Synchronous send operation **blocks until** receive is issued;
- Receive operation **blocks until** message is sent.
- Blocking subroutine returns only when the operation has completed.

Blocking Message Passing

- The call waits until the data transfer is done:
- The sending process waits until all data are transferred to the system buffer
- The receiving process waits until all data are transferred from the system buffer to the receive buffer
- Buffers can be freely reused

Blocking Message Send

- `MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm);`

• **buf**	Specifies the starting address of the buffer.
• **count**	Indicates the number of buffer elements
• **dtype**	Denotes the datatype of the buffer elements
• **dest**	Specifies the rank of the destination process in the group associated with the communicator comm
• **tag**	Denotes the message label
• **comm**	Designates the communication context that identifies a group of processes

Blocking Message Send

`Standard(MPI_Send)`	The sending process returns when the system can buffer the message or when the message is received and the buffer is ready for reuse.
`Buffered (MPI_Bsend)`	The sending process returns when the message is buffered in an application-supplied buffer.
`Synchronous(MPI_Ssend)`	The sending process returns only if a matching receive is posted and the receiving process has started to receive the message.
`Ready (MPI_Rsend)`	The message is sent as soon as possible.

Hprcse

Blocking Message Receive

- `MPI_Recv (void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status);`

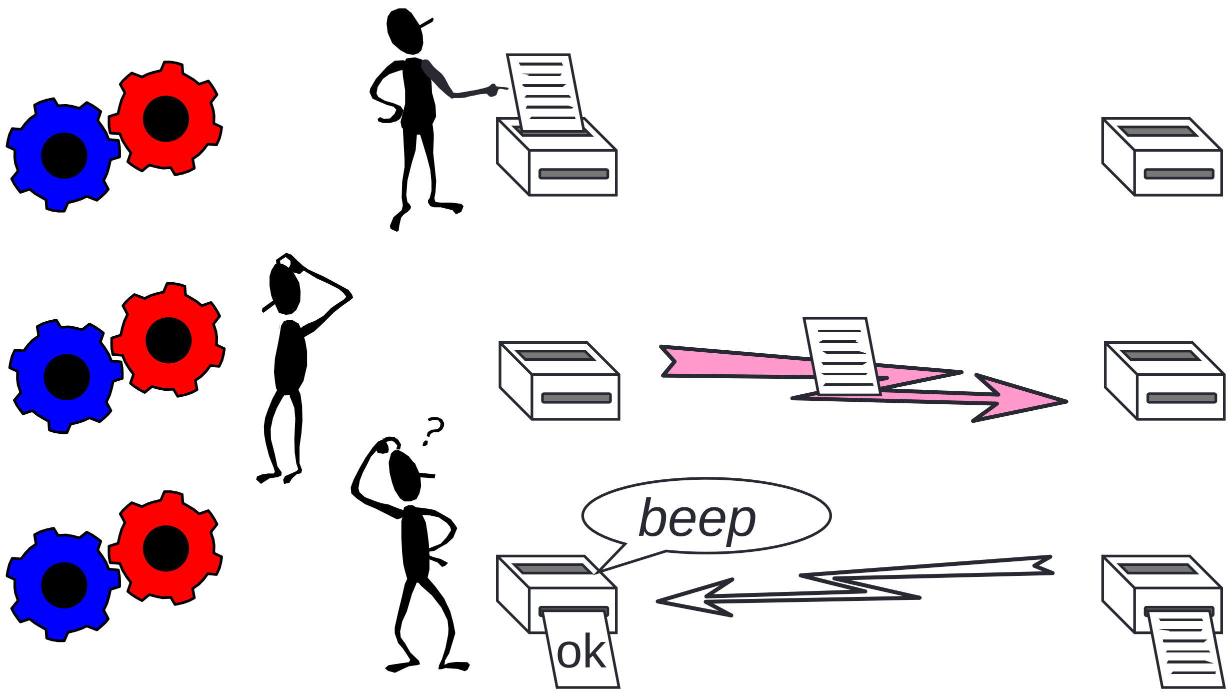
• **buf**	Specifies the starting address of the buffer.
• **count**	Indicates the number of buffer elements
• **dtype**	Denotes the datatype of the buffer elements
• **source**	Specifies the rank of the source process in the group associated with the communicator comm
• **tag**	Denotes the message label
• **comm**	Designates the communication context that identifies a group of processes
• **status**	Returns information about the received message

Example

```
…
if (rank == 0) {
    for (i=0; i<10; i++)  buffer[i] = i;
    MPI_Send(buffer, 10, MPI_INT, 1, 123,
 MPI_COMM_WORLD);
} else if (rank == 1) {
    for (i=0; i<10; i++)  buffer[i] = -1;
    MPI_Recv(buffer, 10, MPI_INT, 0, 123,
 MPI_COMM_WORLD, &status);
    for (i=0; i<10; i++)
   if (buffer[i] != i)
         printf("Error: buffer[%d] = %d but is
 expected to be %d\n", i, buffer[i], i);
 }
…
```

Non-Blocking Operations

- Non-blocking operations return immediately and allow the sub-program to perform other work.



Non-blocking Message Passing

- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
- Need to be careful though
  - When send and receive buffers are updated before the transfer is over, the result will be wrong

Non-blocking Message Passing

```
MPI_Isend (void *buf, int count, MPI_Datatype dtype,
  int dest, int tag, MPI_Comm comm, MPI_Request *req);

MPI_Recv (void *buf, int count, MPI_Datatype dtype, int
  source, int tag, MPI_Comm comm, MPI_Request *req);

MPI_Wait(MPI_Request *req, MPI_Status *status);
```

• **req**	Specifies the request used by a completion routine when called by the application to complete the send operation.

Blocking	MPI_Send	MPI_Bsend	MPI_Ssend	MPI_Rsend	MPI_Recv
Non-blocking	MPI_Isend	MPI_Ibsend	MPI_Issend	MPI_Irsend	MPI_Irecv

Non-blocking Message Passing

```
…
right = (rank + 1) % nproc;
left = rank - 1;
if (left < 0)          left = nproc - 1;
MPI_Irecv(buffer, 10, MPI_INT, left, 123,
 MPI_COMM_WORLD, &request);
MPI_Isend(buffer2, 10, MPI_INT, right, 123,
 MPI_COMM_WORLD, &request2);
MPI_Wait(&request, &status);
MPI_Wait(&request2, &status);
…
```

How to execute MPI codes?

- The implementation supplies scripts to launch the MPI parallel calculation
  - **`mpirun -np #proc a.out`**
  - **`mpiexec -n #proc a.out`**
- A copy of the same program runs on each processor core within its own process (private address space)
- Communication
  - through the network interconnect
  - through the shared memory on SMP machines

PBS: Portable Batch System

- A cluster is shared with others
  - Need to use a job submission system
- PBS will allocate the job to some other computer, log in as the user, and execute it
  - The script must contain cd's or absolute references to access files
- Useful Commands
  - **`qsub`** : submits a job
  - **`qstat`** : monitors status
  - **`qdel`** : deletes a job from a queue

PBS script

PBS	Description
`#PBS -N jobname`	Assign a name to job
`#PBS -M email_address`	Specify email address
`#PBS -m b`	Send email at job start
`#PBS -m e`	Send email at job end
`#PBS -m a`	Send email at job abort
`#PBS -o out_file`	Redirect stdout to specified file
`#PBS -e errfile`	Redirect stderr to specified file
`#PBS -q queue_name`	Specify queue to be used
`#PBS -l select=chunk specification`	Specify MPI resource requirements
`#PBS -l walltime=runtime`	Set wallclock time limit

PBS script example

```
#!/bin/bash
# request 4 nodes,  each node runs 2 processes for 2 hours
#PBS -l nodes=4:ppn=2,walltime=02:00:00
#  specify job queue
#PBS -q dque
# declare a name for this job
#PBS -N job_name
# specify your email address
#PBS -M usename@domain
# mail is sent to you when the job starts and when it terminates or aborts
#PBS -m bea

cd $WORK_DIR
mpirun -np 8 a.out
```

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow **optimized internal** implementations, e.g., tree based algorithms

Collective communications

- A single call handles the communication between all the processes in a communicator

- There are 3 types of collective communications
  - Data movement (e.g. MPI_Bcast)
  - Reduction (e.g. MPI_Reduce)
  - Synchronization (e.g. MPI_Barrier)

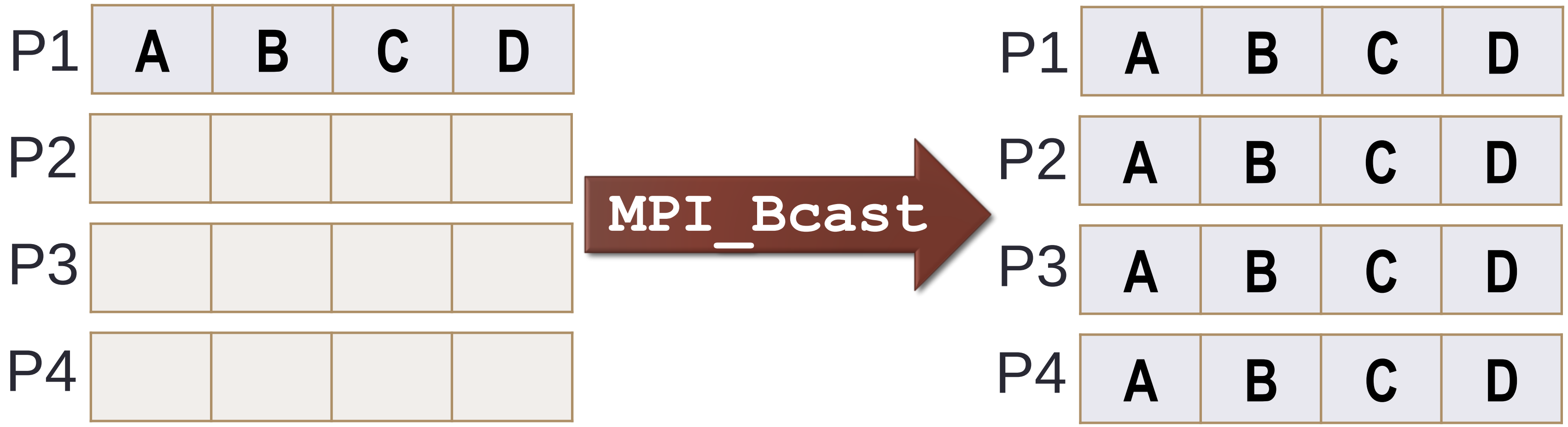
Broadcast

- A one-to-many communication.

Hprcse

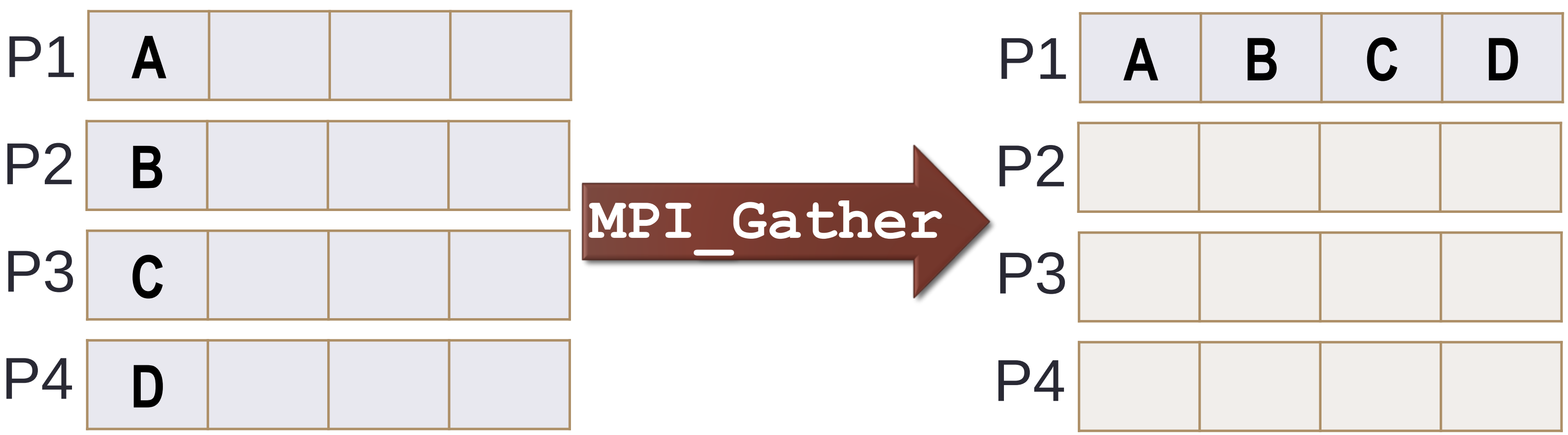
Broadcast

- `int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);`
- One process (root) sends data to all the other processes in the same communicator
- Must be called by all the processes with the same arguments



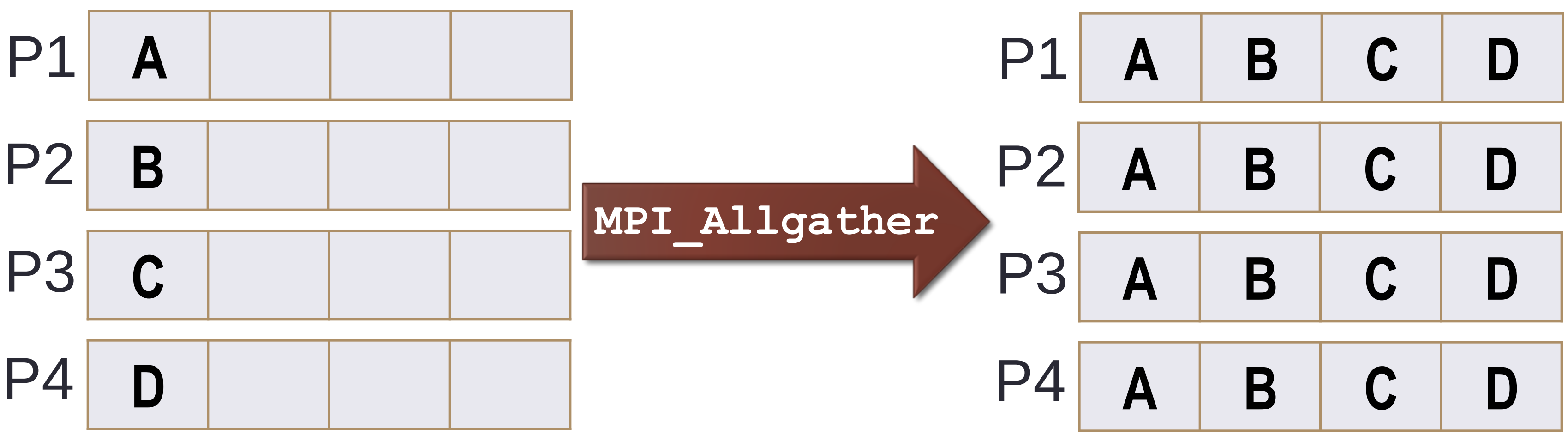
Gather

- `int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype,  int root, MPI_Comm comm)`
- One process (root) collects data to all the other processes in the same communicator
- Must be called by all the processes with the same arguments



Gather to All

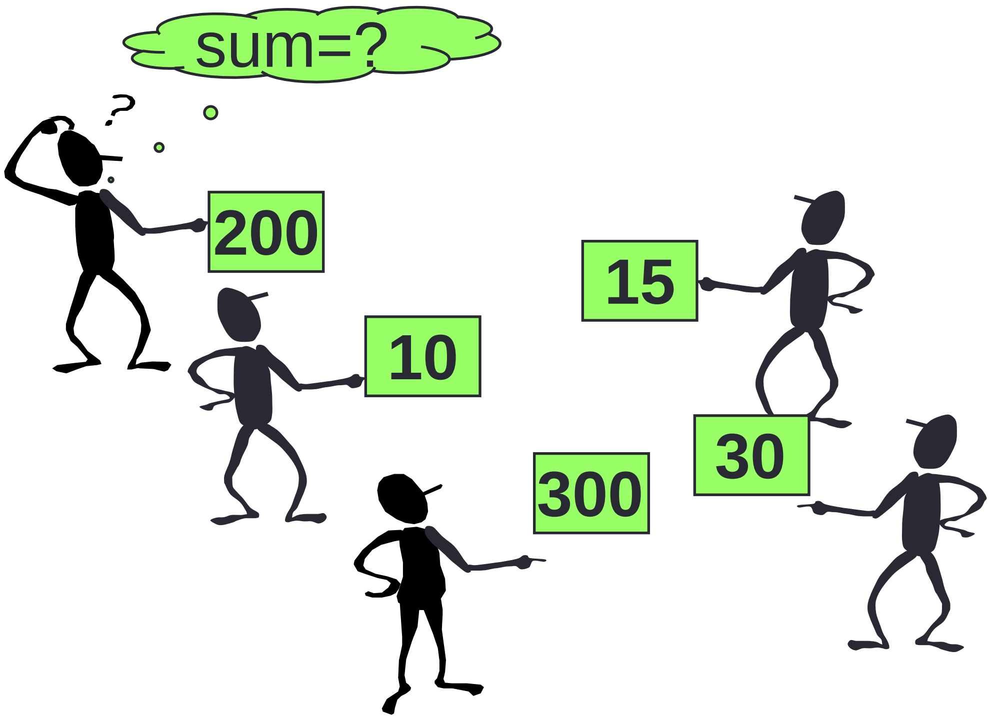
- `int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)`
- All the processes collects data to all the other processes in the same communicator
- Must be called by all the processes with the same arguments



Hprcse

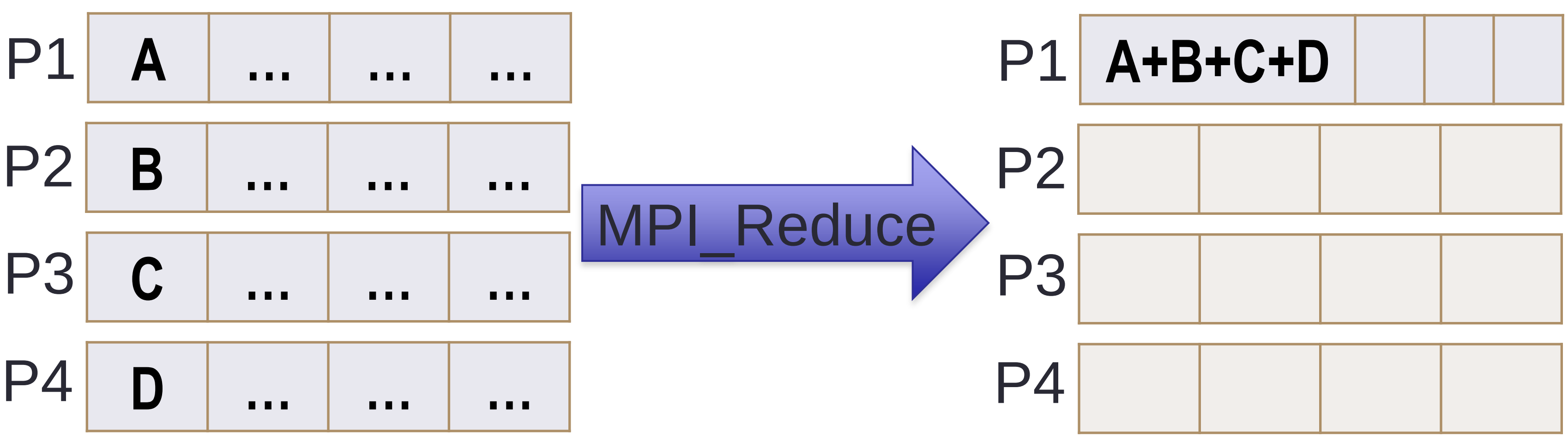
Reduction Operations

- Combine data from several processes to produce a single result.



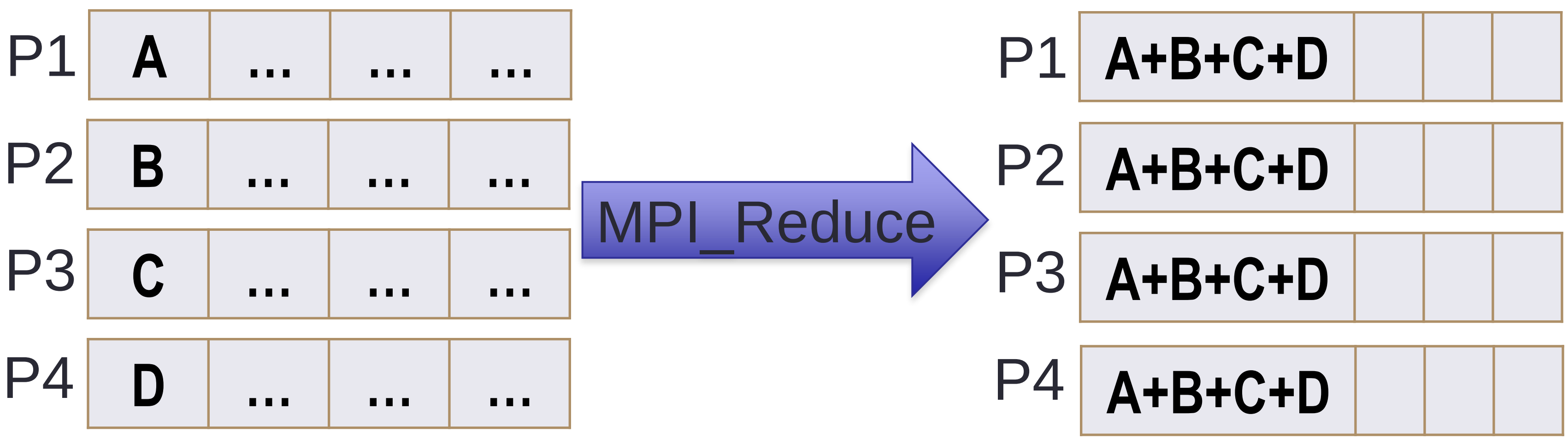
Reduction

- `int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
- One process (root) collects data to all the other processes in the same communicator, and performs an operation on the data
- MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- **`MPI_Op_create()`**: User defined operator



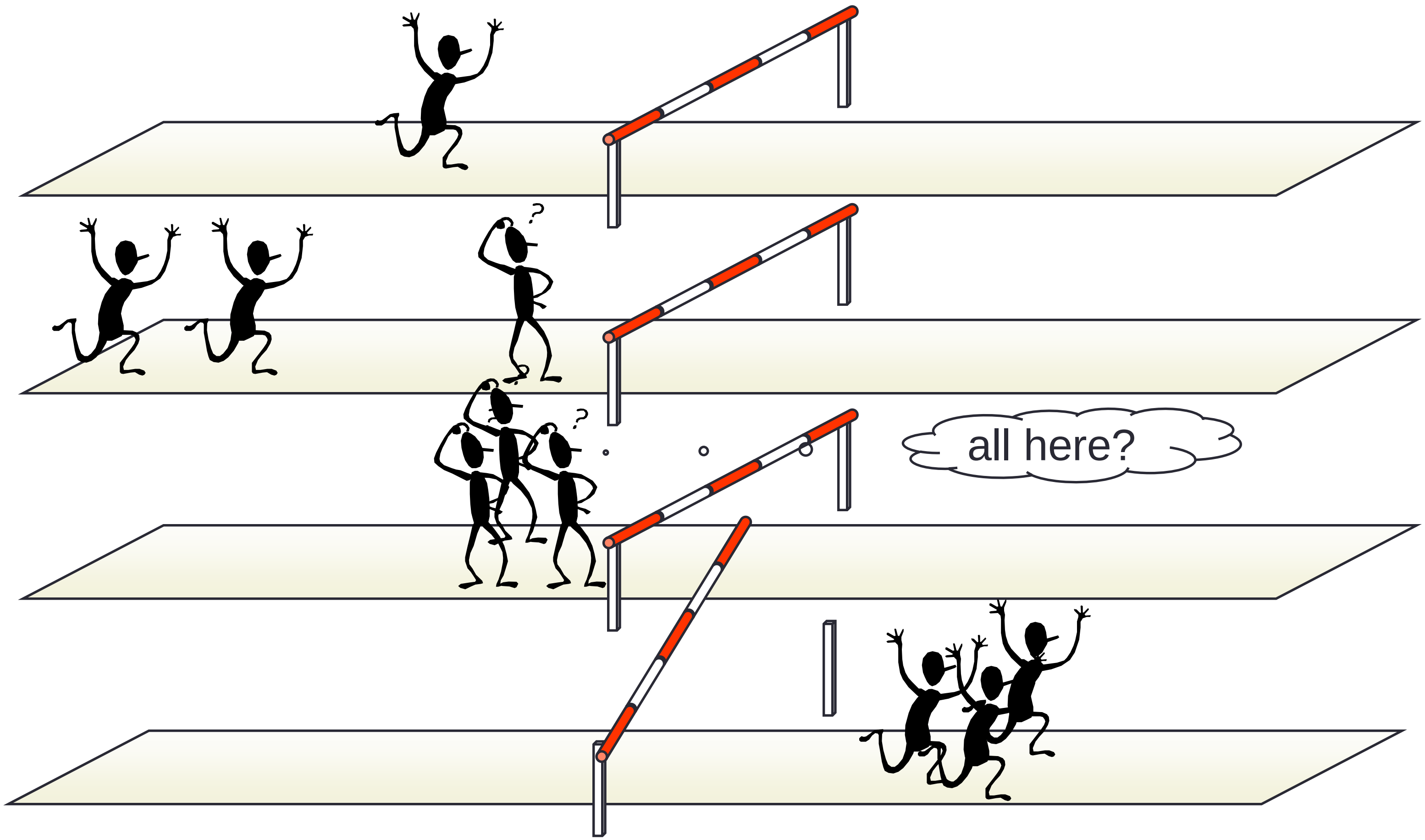
Reduction to All

- `int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
- All the processes collect data to all the other processes in the same communicator, and perform an operation on the data
- MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- **`MPI_Op_create()`**: User defined operator



Barriers

- Synchronize processes.



Synchronization

- `int MPI_Barrier(MPI_Comm comm)`

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Hello, world.  I am %d of %d\n", rank,
 nprocs);
    MPI_Finalize();
    return 0;
}
```