WASHINGTON STATE UNIVERSITY
VANCOUVER

# CS 563



# Concurrent Programming

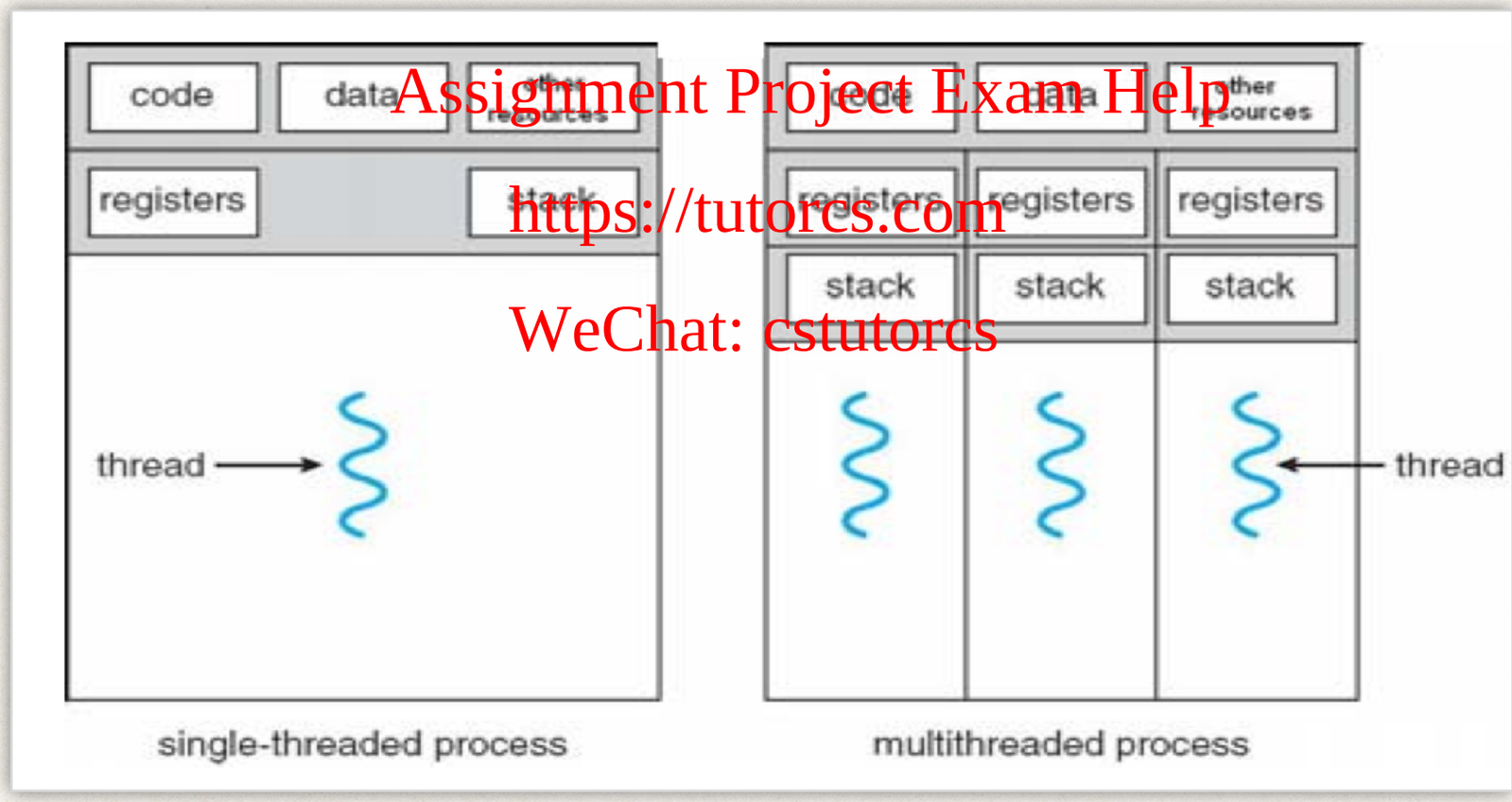




## Lecture 8: Message Passing Interface (MPI)

# Processes vs. Threads





# The Message Passing Model



- MPI is for communication among processes (Inter-Process Communication), which have separate address spaces



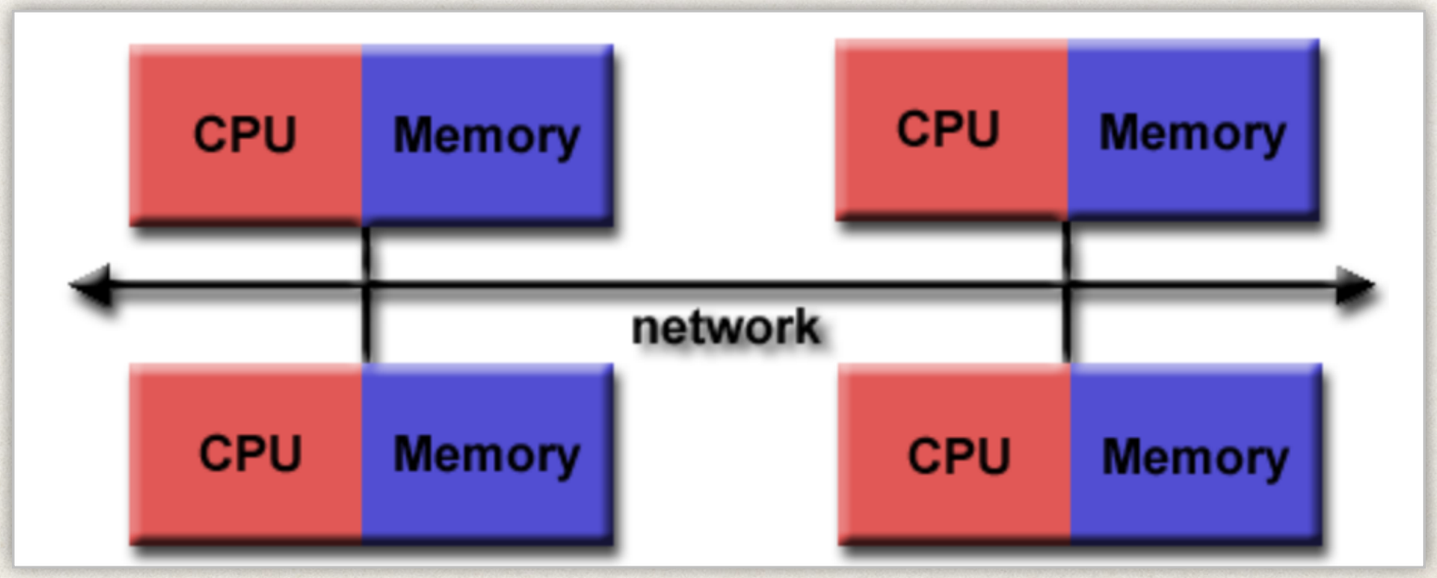


- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process's address space to another's

# MPI Library



- One program, copy loaded onto every node
- SPMD programming style (single program, multiple data)



# MPI Library

- Today, MPI runs on virtually any hardware platform



  - Distributed memory



  - Shared memory



  - Hybrid

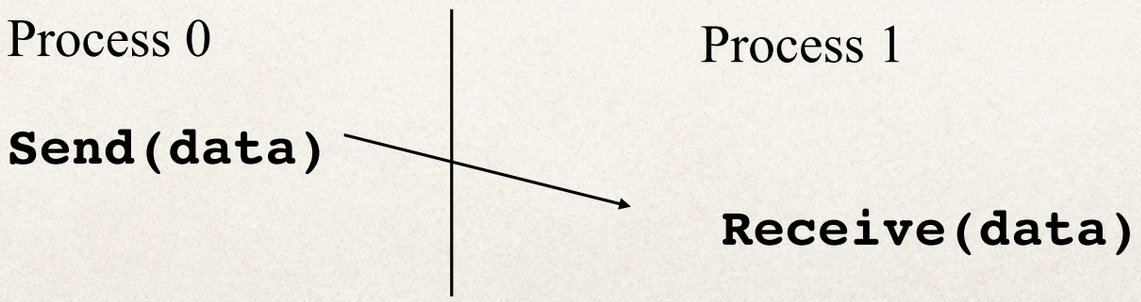
- The programming model clearly remains a distributed memory model, regardless the underlying physical architecture

- All parallelism is explicit

# Cooperative Operations for Communication

- The message-passing approach makes the exchange of data *cooperative*.
- Data is explicitly *sent* by one process and *received* by another.
- An advantage is that any change in the receiver process's memory is made with the receiver's explicit participation.
- Communication and synchronization are combined.





# What is MPI

- A message-passing library specification
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Designed to provide access to advanced parallel hardware

# MPI Resources on Web



- The Standard itself:
  - at http://www.mpi-forum.org

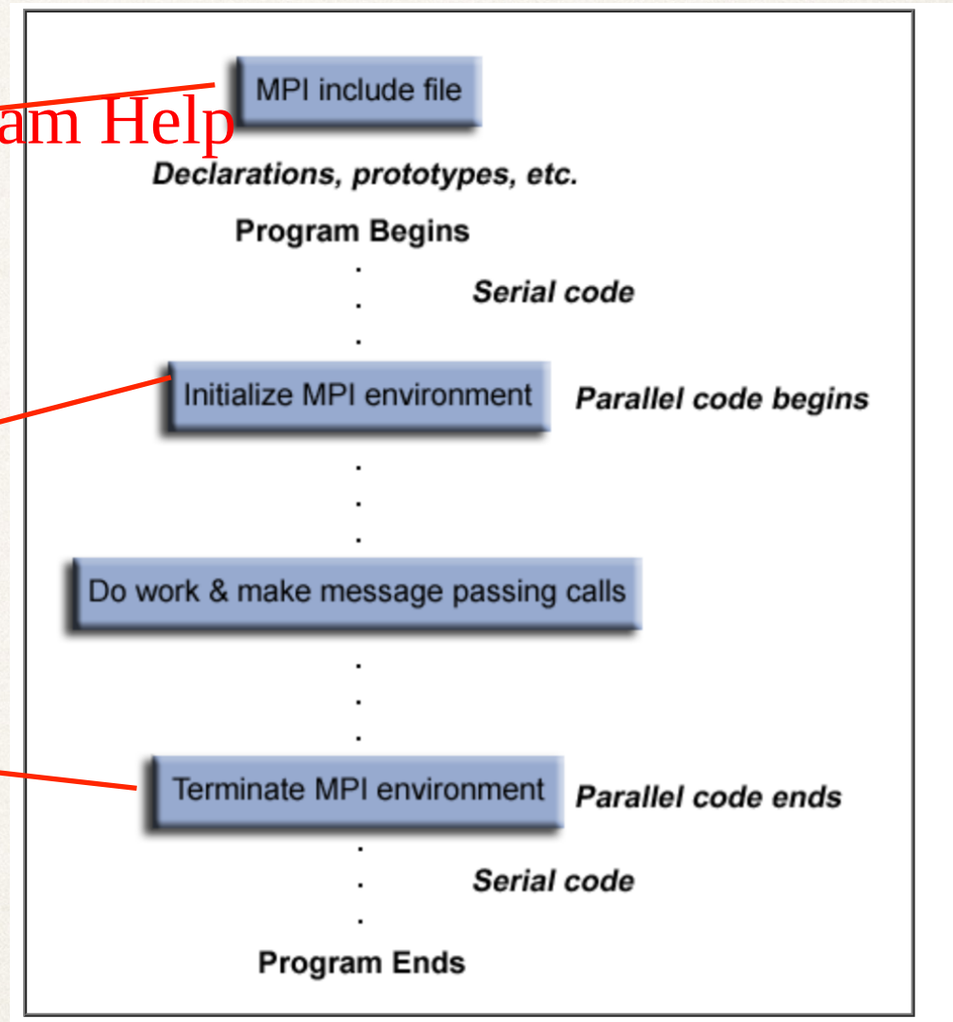
# MPI Books

- Books:
  - Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - *MPI: The Complete Reference*, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
  - Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
  - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
  - MPI: The Complete Reference Vol 1 and 2,MIT Press, 1998(Fall).

# A Minimal MPI Program



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

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```



# C and MPI



- mpi.h must be #included
- MPI functions return error codes or MPI_SUCCESS
- By default, an error causes all processes to abort.

# Running MPI Programs



- In general, starting an MPI program is dependent on the implementation of MPI you are using, and might require various scripts, program arguments, and/or environment variables.
  - mpiexec
  - mpirun

# Finding Out About the Environment



- Two important questions that arise early in a parallel program are:



  - How many processes are participating in this computation?
  - Which one am I?

# Finding Out About the Environment



- MPI provides functions to answer these questions:
  - MPI_Comm_size — how many
  - MPI_Comm_rank — who am I

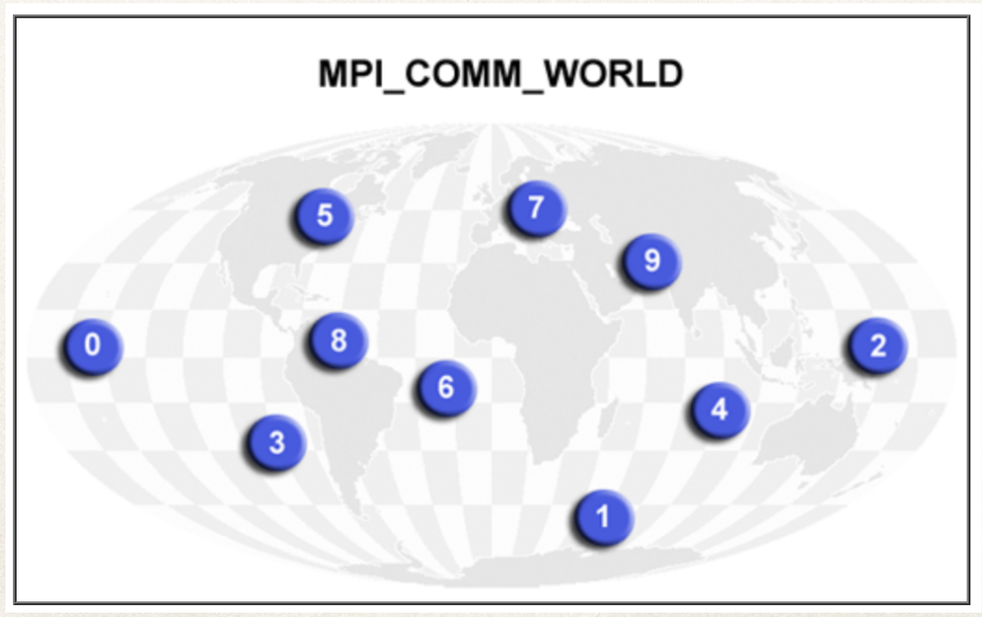
# Better Hello

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

# Communicators

- MPI uses objects called communicators and groups to define which collection of processes may communicate with each other
- Most MPI routines require you to specify a communicator as an argument
- For now, simply use MPI_COMM_WORLD wherever a communicator is required





# MPI Basic Send/Receive

- We need to fill in the details in

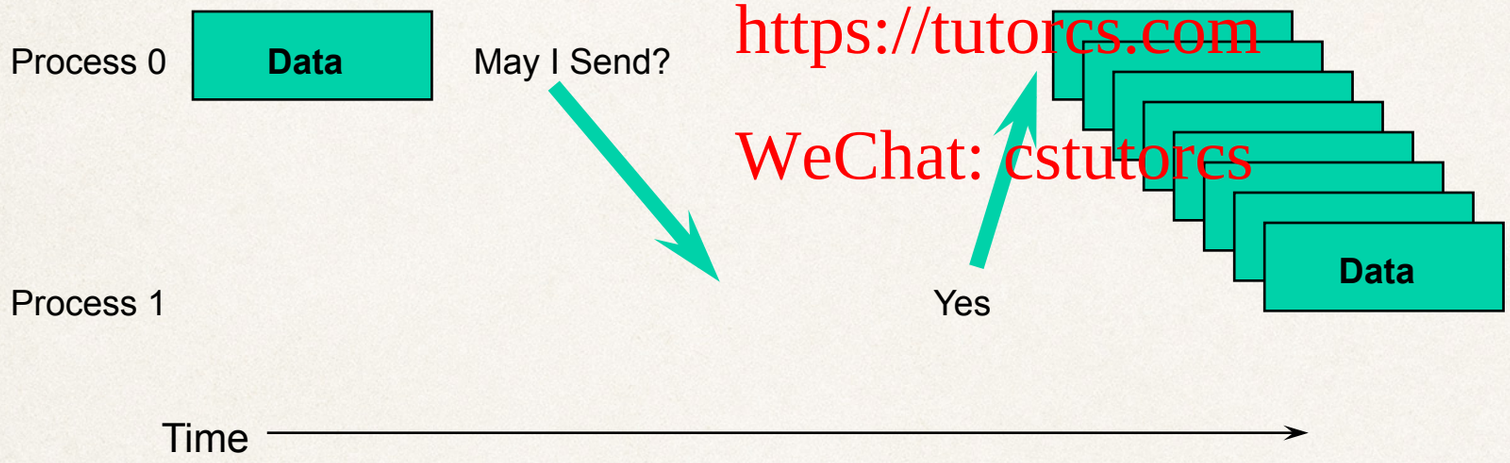



- Things that need specifying
  - how will data be described?
  - how are processes identified?
  - how does the receiver recognize/screen messages?
  - what does it mean for these operations to complete?

# What is Message Passing

- Data transfer plus synchronization








- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

# MPI Datatypes



- The data in a message to sent or received is described by a triple (address, count, datatype)
- Predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
- Datatype can also be an array of the above primitive types

# MPI Tags



- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.

- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.

- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes.

# MPI Basic (Blocking) Send



- MPI_SEND (start, count, datatype, dest, tag, comm)
- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused.

# MPI Basic (Blocking) Receive

- MPI_RECV(start, count, datatype, source, tag, comm, status)
  - Waits until a matching (on **`source`** and **`tag`**) message is received from the system, and the buffer can be used.
  - **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**.
  - **`status`** contains further information
  - Receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error.

# Even Better Hello

```
int main(int argc, char *argv[]){
   char idstr[32];
   char buff[BUFSIZE];
   int numprocs;
   int myid;
   int i;
   MPI_Status stat;
/* all MPI programs start with MPI_Init; all N' processes exist thereafter */
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /* find out how big the SPMD world is */
MPI_Comm_rank(MPI_COMM_WORLD,&myid); /* and this processes' rank is */
/* At this point, all the programs are running equivalently, the rank is used to
   distinguish the roles of the programs in the SPMD model, with rank 0 often used
   specially... */
if(myid == 0){
  printf("%d: We have %d processors\n", myid, numprocs);
  for(i=1;i<numprocs;i++){
    sprintf(buff, "Hello %d! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
  }
  for(i=1;i<numprocs;i++){
      MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
      printf("%d: %s\n", myid, buff);
  }
} else {
      /* receive from rank 0: */
       MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
       sprintf(idstr, "Processor %d ", myid);
       strcat(buff, idstr);
       strcat(buff, "reporting for duty\n");
       /* send to rank 0: */
       MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
}
   /* MPI Programs end with MPI Finalize; this is a weak synchronization point */
   MPI_Finalize();
return 0; }
```

# MPI Example

```
#include <mpi.h>

main(int argc, char *argv[]) {
  int myid, otherid, size;
  int length = 1, tag = 1;
  int myvalue, othervalue;
  MPI_Status status;

  /* initialize MPI and get own id (rank) */
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  if (myid == 0) {
    otherid = 1; myvalue = 14;
  } else {
    otherid = 0; myvalue = 25;
  }
  MPI_Send(&myvalue, length, MPI_INT, otherid,
           tag, MPI_COMM_WORLD);
  MPI_Recv(&othervalue, length, MPI_INT, MPI_ANY_SOURCE,
           tag, MPI_COMM_WORLD, &status);
  printf("process %d received a %d\n", myid, othervalue);

  MPI_Finalize();
}
```



Exchange Values

# Retrieving Further Information

- MPI_RECV(start, count, datatype, source, tag, comm, status)



- status is a data structure allocated in the user's program



- Example:



```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

# MPI is Simple



- Many parallel programs can be written using just these six functions, only two of which are non-trivial:

```
MPI_INIT
MPI_FINALIZE
MPI_COMM_SIZE
MPI_COMM_RANK
MPI_SEND
MPI_RECV
```

**Point-to-point (send/recv) isn't the only way...**

# Collective Operations in MPI

- Collective operations are called by all processes in a communicator.



- Unexpected behavior, including program failure, can occur if even one process in the communicator doesn't participate





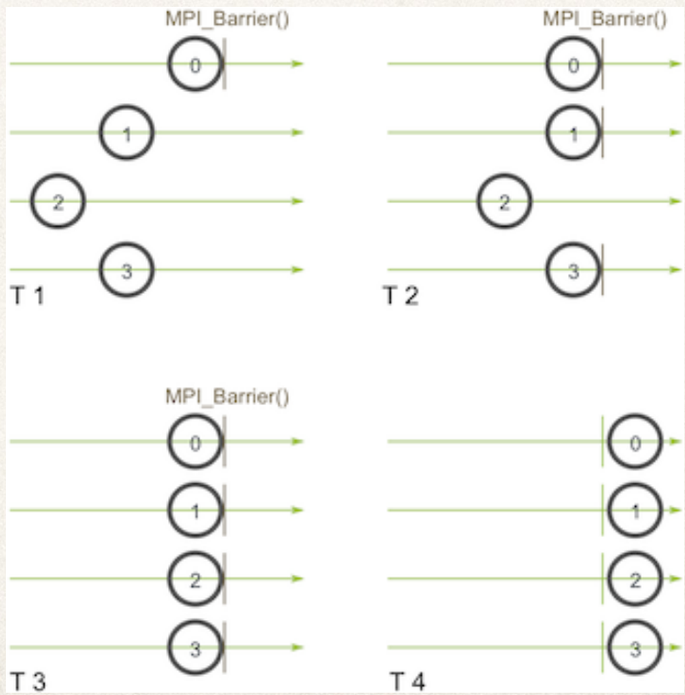
- Types of collective operations:
  - Synchronization
  - Data movement
  - Collective computation

# Barrier

- **`MPI_Barrier`** creates a barrier synchronization in a group.
- Each task, when reaching the MPI_Barrier call, blocks until all tasks in the group reach the same MPI_Barrier call. Then all tasks are free to proceed.



- MPI_Barrier(comm)



```c
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "math.h"
#include "mpi.h"
```

---

```c
int main(int argc,char** argv)
{
   int          taskid, ntasks;
   int          ierr,i,j,itask;
   int           buffsize;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Comm_size(MPI_COMM_WORLD,&ntasks);

   if ( taskid == 0 ){
     printf("\n\n\n\n\n");
   }

   ierr=MPI_Barrier(MPI_COMM_WORLD);

   if ( taskid == 0 )printf("Hel");
   if ( taskid == 1 )printf("lo ");
   if ( taskid == 2 )printf("Wor");
   if ( taskid == 3 )printf("ld!");

   ierr=MPI_Barrier(MPI_COMM_WORLD);

   if ( taskid == 0 ){
     printf(" (Ordered)\n\n\n\n\n");
   }


   MPI_Finalize();

}
```



Ordered?

```c
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include "math.h"
#include "mpi.h"
```

```c
int main(int argc,char** argv)
{
   int          taskid, ntasks;
   int          ierr,i,j,itask;
   int           buffsize;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Comm_size(MPI_COMM_WORLD,&ntasks);

   if ( taskid == 0 ){
     printf("\n\n\n\n");
   }

   ierr=MPI_Barrier(MPI_COMM_WORLD);

   if ( taskid == 0 )printf("Hel");
   ierr=MPI_Barrier(MPI_COMM_WORLD);
   if ( taskid == 1 )printf("lo ");
   ierr=MPI_Barrier(MPI_COMM_WORLD);
   if ( taskid == 2 )printf("Wor");
   ierr=MPI_Barrier(MPI_COMM_WORLD);
   if ( taskid == 3 )printf("ld!");
   ierr=MPI_Barrier(MPI_COMM_WORLD);

   if ( taskid == 0 ){
     printf(" (Ordered)\n\n\n\n\n");
   }


   MPI_Finalize();

}
```

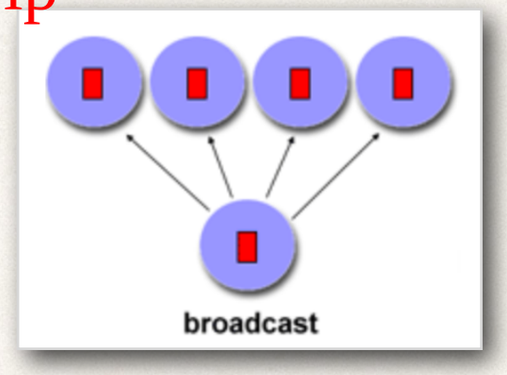
# Broadcast

- **`MPI_Bcast`** distributes data from one process (the root) to all others in a communicator.

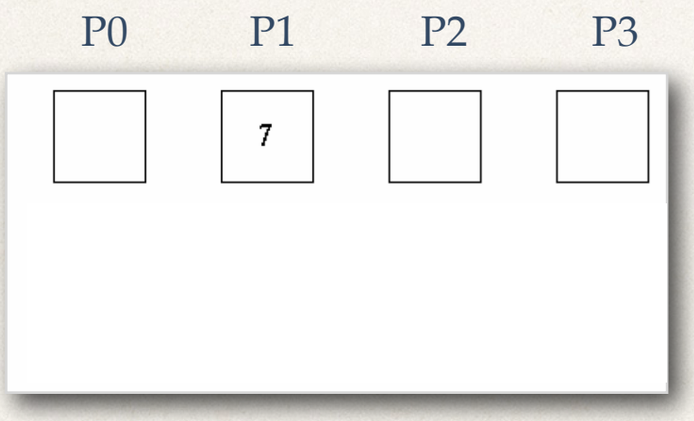


- MPI_Bcast(start, count, datatype, root, comm)







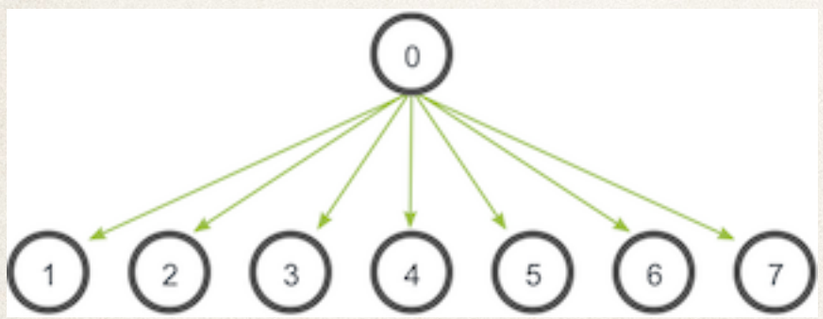



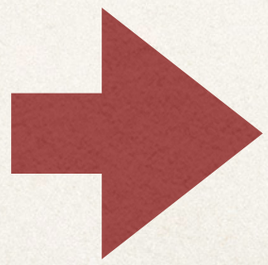

```
count = 1;
source = 1;
MPI_Bcast(buffer, count, MPI_INT, source, MPI_COMM_WORLD);
```

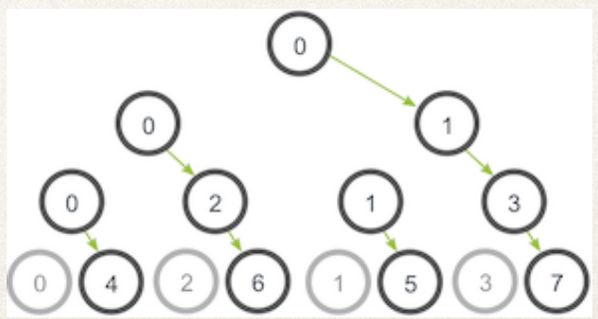
```
void function(void* data, int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
             MPI_STATUS_IGNORE);
  }
}
```

| Processors | my_bcast | MPI_Bcast |
|---|---|---|
| 2 | 0.0344 | 0.0344 |
| 4 | 0.1025 | 0.0817 |
| 8 | 0.2685 | 0.1084 |
| 16 | 0.5109 | 0.1296 |







# Scatter

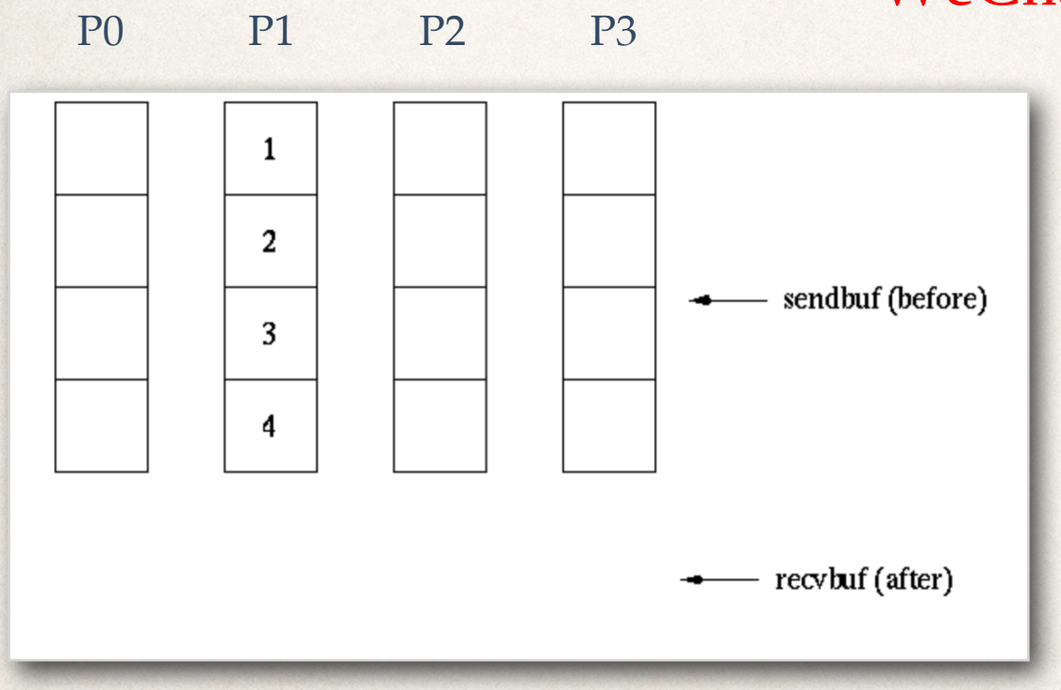
- **`MPI_Scatter`** distributes distinct data from one process (the root) to each process in a communicator.

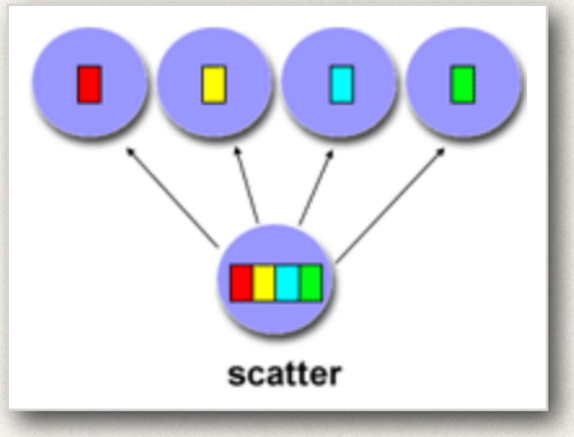


- MPI_Scatter(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, source, comm)








```
sendcnt = 1;
recvcnt = 1;
source = 1;
MPI_Scatter(sendbuf, sendcnt, MPI_INT, recvbuf, recvcnt,
   MPI_INT, source, MPI_COMM_WORLD);
```

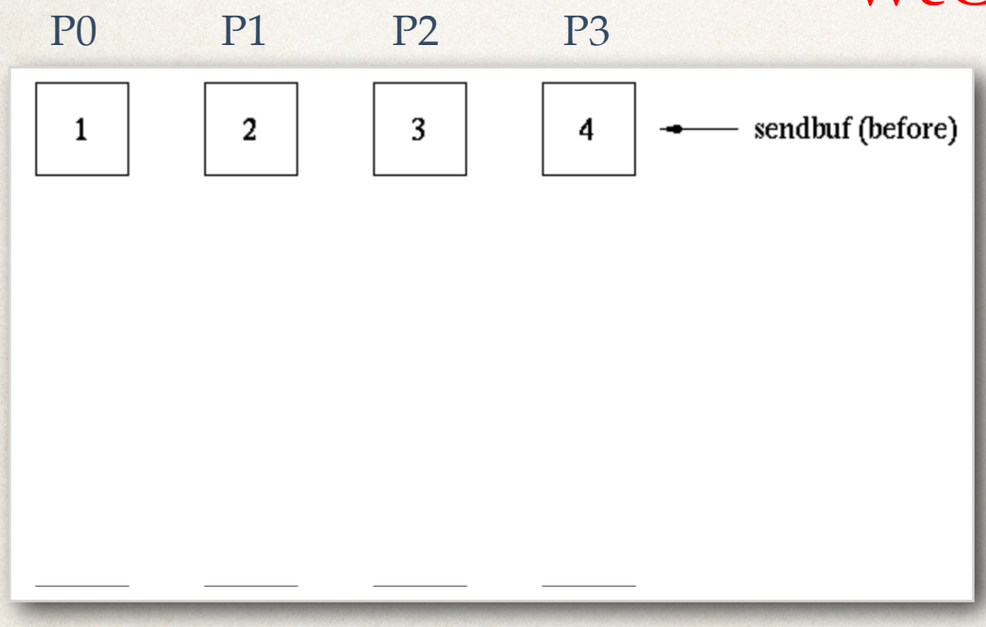
# Gather

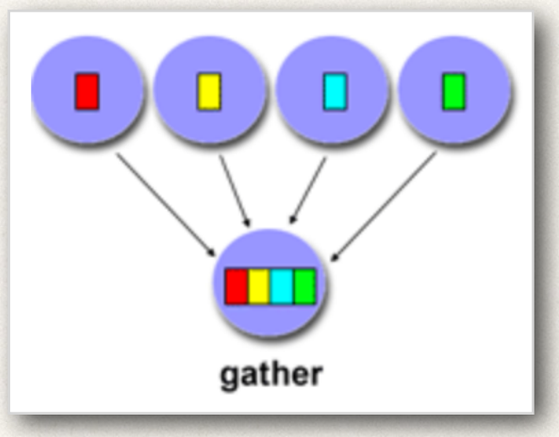
- **`MPI_Gather`** gathers distinct data from each process in a communicator to a single destination task.
- Reverse operation of MPI_Scatter



- MPI_Gather(sendbuf, sendcnt, sendtype, recvbuf, recvcnt, recvtype, dest, comm)






```
sendcnt = 1;
recvcnt = 1;
dest = 1;
MPI_Gather(sendbuf, sendcnt, MPI_INT, recvbuf, recvcnt,
   MPI_INT, dest, MPI_COMM_WORLD);
```

# Reduce

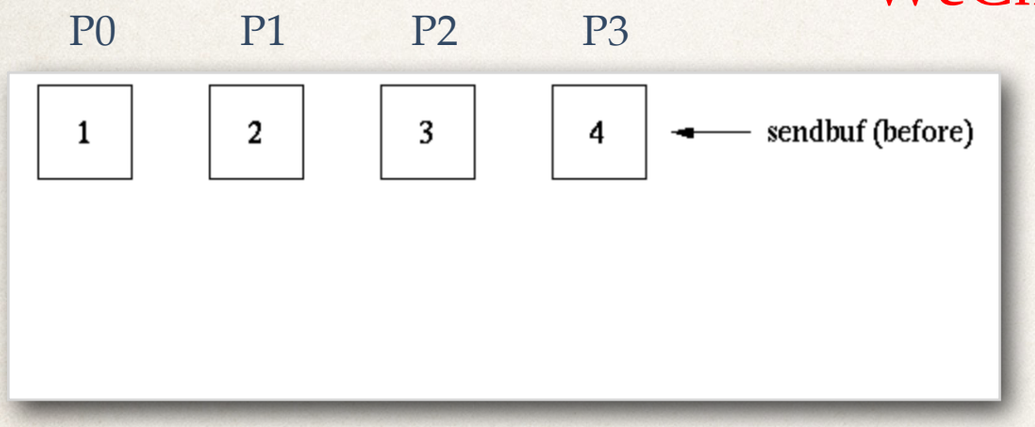
- **`MPI_Reduce`** applies a reduction operation on all processes in the communicator and places the result in one process.

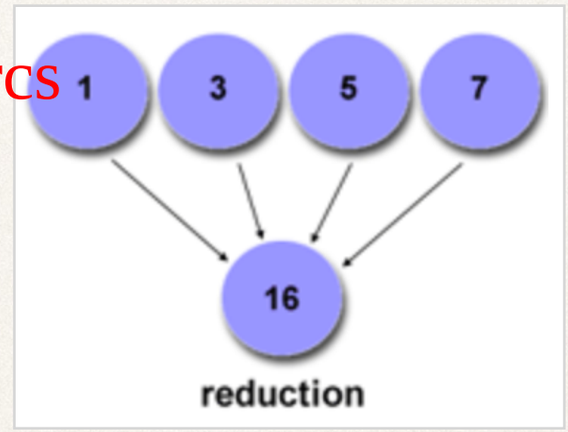


- MPI_Reduce(sendbuf, recvbuf, count, datatype, op, dest, comm)







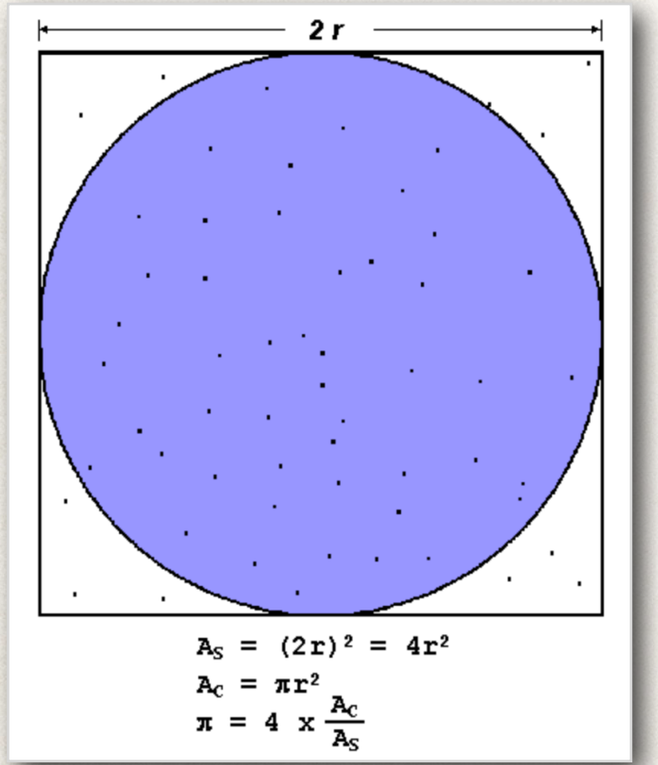




```
count = 1;
dest = 1;
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT, MPI_SUM, dest,
   MPI_COMM_WORLD);
```

# MPI Reduce Operations

- MPI_MAX
- MPI_MIN
- MPI_SUM
- MPI_PROD
- MPI_LAND
- MPI_LOR
- MPI_BAND
- MPI_BOR
- MPI_MAXLOC
- MPI_MINLOC

# Example: Pi




$A_S = (2r)^2 = 4r^2$

$A_C = \pi r^2$

$\pi = 4 \times \frac{A_C}{A_S}$



```
npoints = 10000
circle_count = 0

do j = 1,npoints
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```

Sequential code