Distributed Memory Programming with Message-Passing

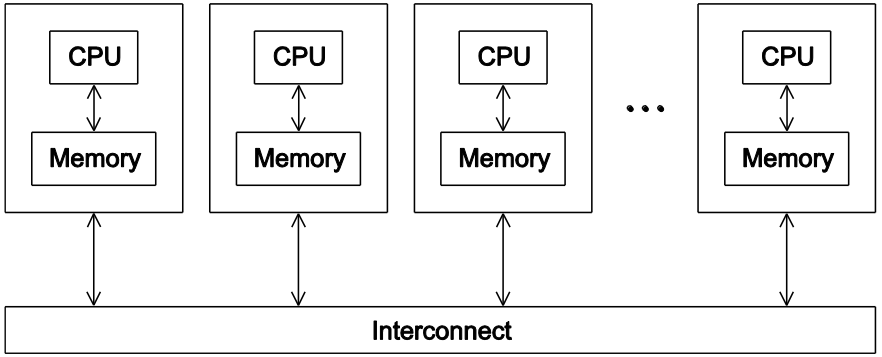
T. Yang, CS140

Part of slides from B. Gropp

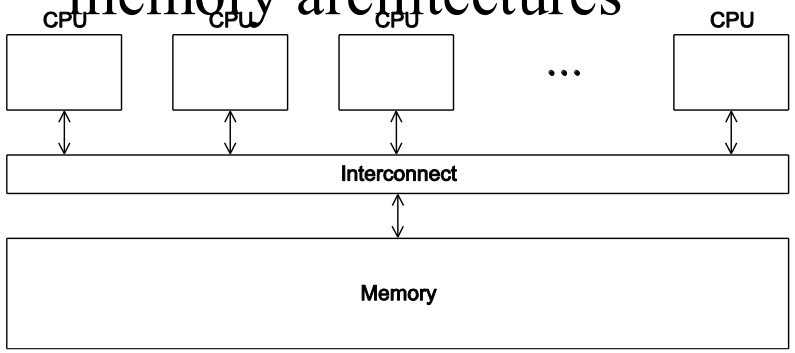
Outline

- An overview of MPI programming
  - Six MPI functions and hello sample
- Send/receive
- Collective communication

MPI: mainly for distributed memory architectures



MPI also works for shared memory architectures

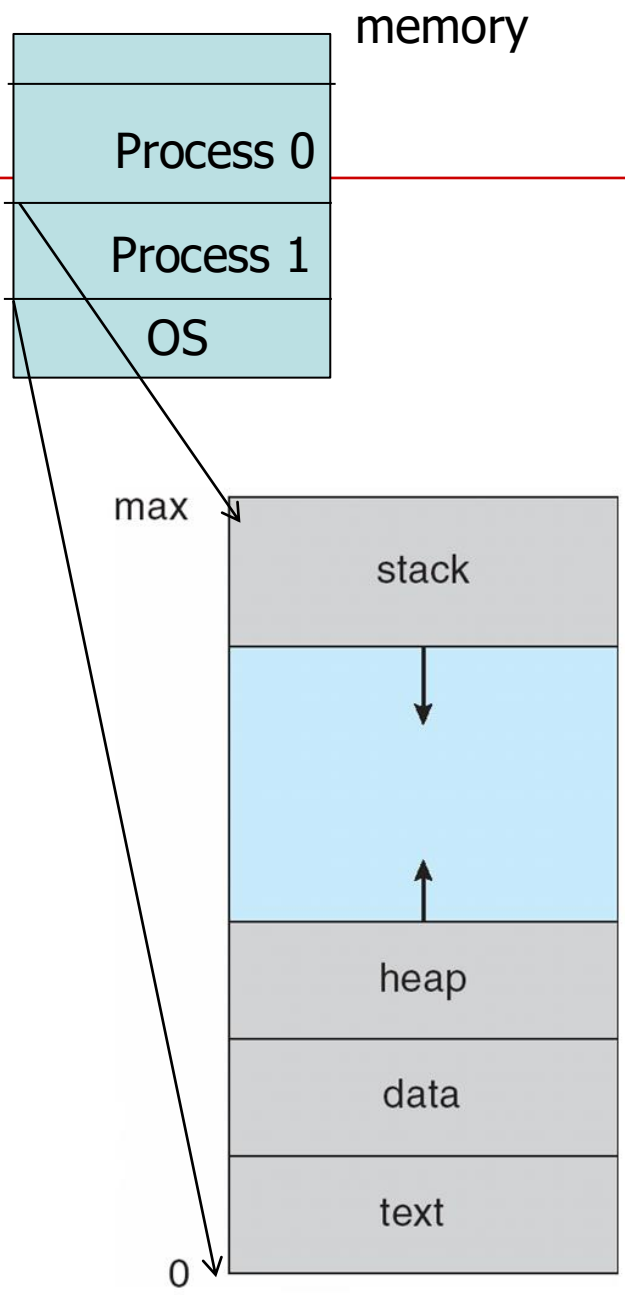


Message Passing Libraries

- MPI, Message Passing Interface, now the industry standard, for C/C++ and other languages
- **Running as a set of processes.**
  - **No shared variables among processes**
- **All communication, synchronization require subroutine calls**
  - Enquiries
    - How many processes? Which one am I? Any messages waiting?
  - Communication
    - point-to-point: Send and Receive
    - Collectives such as broadcast
  - Synchronization
    - Barrier

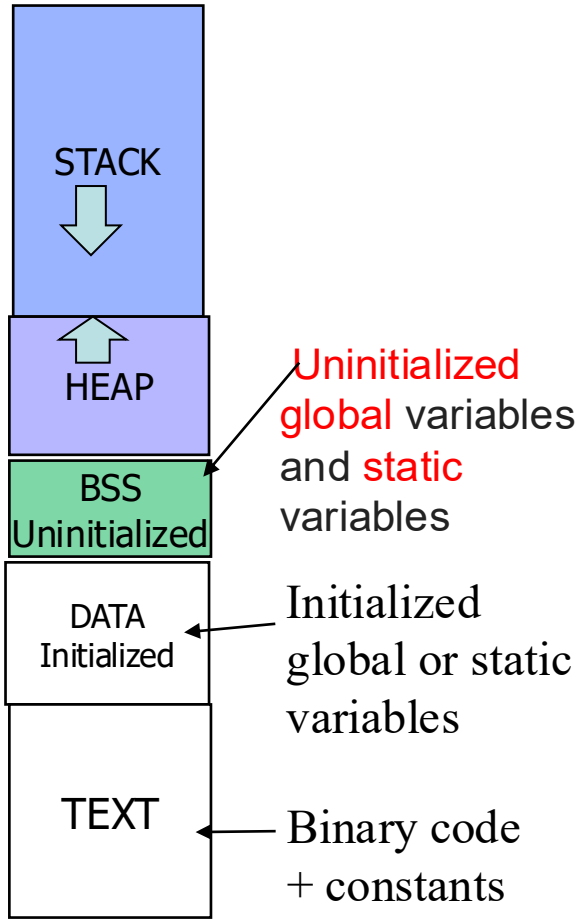
Process Concept from OS

- **Process – a program in execution;**
  - progress in sequential fashion
- **A process in memory includes:**
  - program counter
  - Stack/heap
  - Data/instruction (text) section
- **Processes do not share space**
  - Need memory protection and
  - address translation



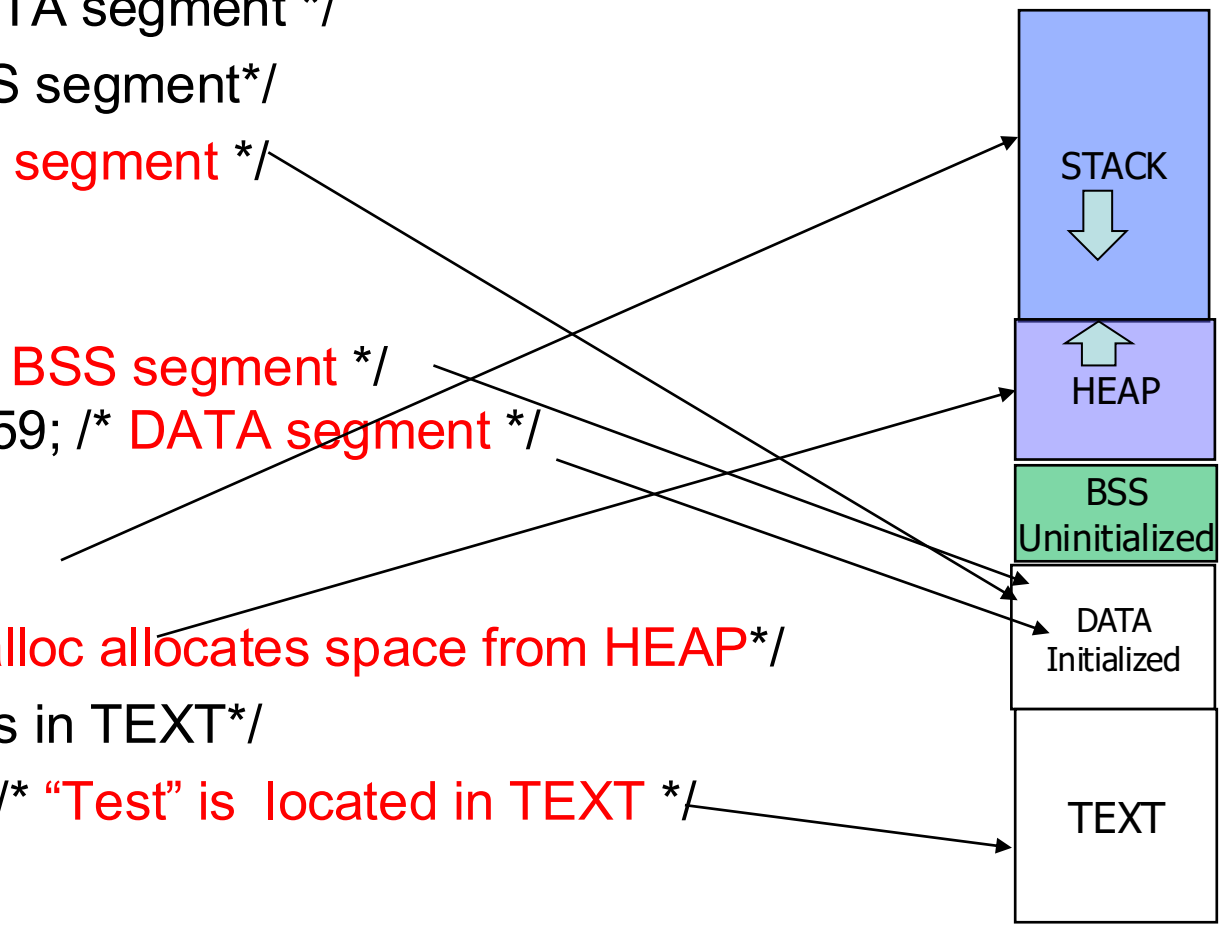
TEXT, DATA, BSS, HEAP and STACK in C

```
int f3=3; /*Initialized DATA segment */
int f1; /*Unitialized BSS segment*/
char def[] = "1";  Where is def?
int main(void) {
    static char abc[12]; /* BSS segment */
    static float pi = 3.14159;          Where is pi?
    int i = 3; /* Stack*/
    char *cp;      where is cp?
    cp= malloc(10);  /* HEAP for allocated chunk*/
    f1= add1(i); /* code is in TEXT. f1 on STACK*/
    strcpy(abc , "Test" );          Where is "Test"?
}
int add1( int f3){     where is f3?
    return f3+1;
}
```



TEXT, DATA, BSS, HEAP, and STACK in C

```
Int f3=3; /* Initialized DATA segment */
Int f1; /*Uninitialized BSS segment*/
char def[] = "1"; /* DATA segment */
int main(void)
    {
    static char abc[12], /* BSS segment */
    static float pi = 3.14159; /* DATA segment */
    int i = 3; /* Stack*/
    char *cp;    /*stack*/
    cp= malloc(10);  /*malloc allocates space from HEAP*/
    f1= add1(i);  /* code is in TEXT*/
    strcpy(abc , "Test" ); /* "Test" is  located in TEXT */
}
int add1( int f3}{/*stack*/
    return f3+1;
}
```



Process Communication

- Normally processes do not share memory
- Processes communicate messages
- If they share some memory through special arrangement, communicate through shared content

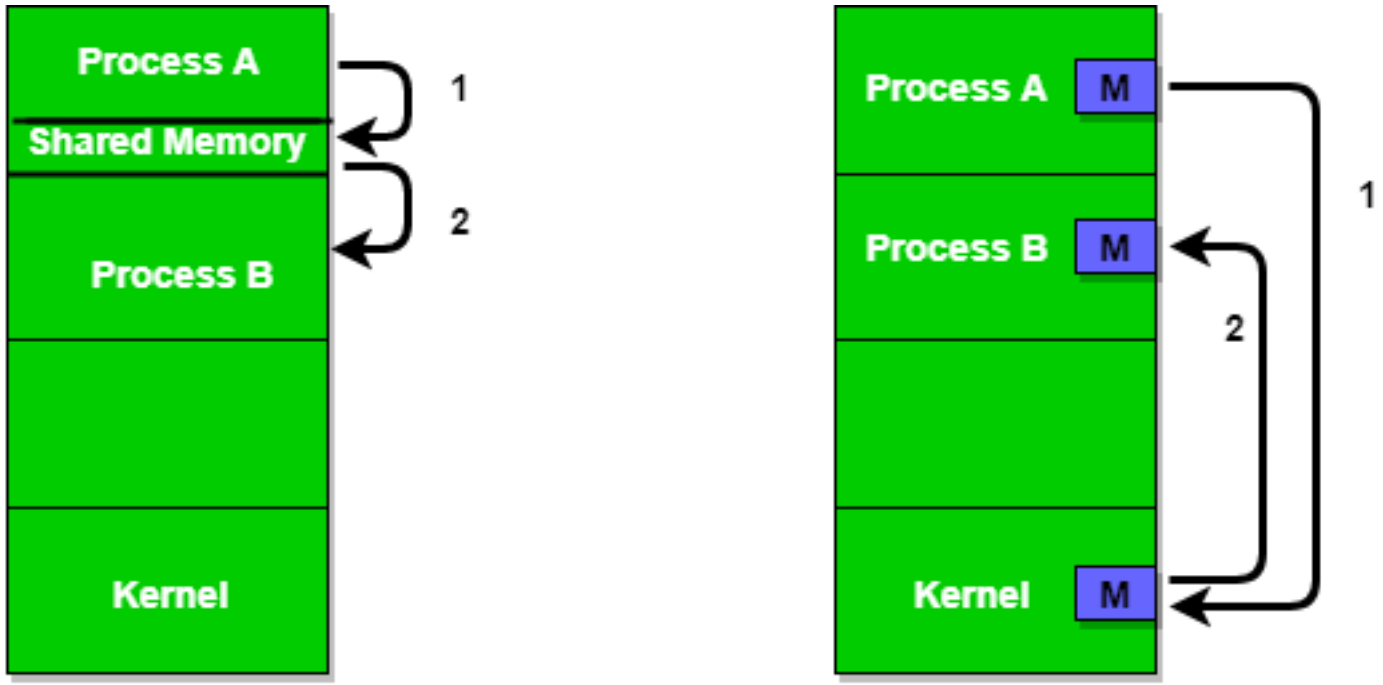


Figure 1 - Shared Memory and Message Passing

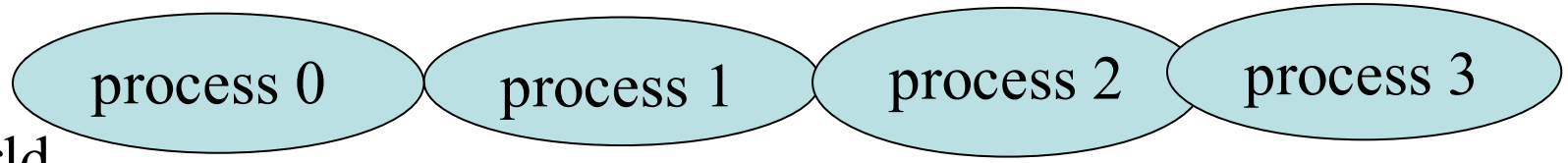
SPMD code in MPI for Hello World!

```
#include <stdio.h>

int main(void) {
    printf("hello, world\n");

    return 0;
}
```

Screen output when running on 4 processes



hello, world

hello, world

hello, world

hello, world

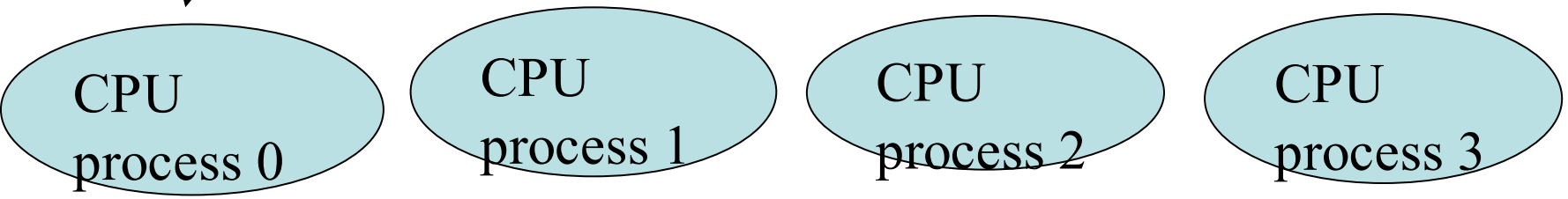
MPI_hello.c Running on 4 nodes

```
#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;
}
```

Screen output:

I am 1 of 4!
I am 3 of 4!
I am 2 of 4!
I am 0 of 4!



MPI Components

- **MPI_Init:** do all the necessary setup.

```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

- **MPI_Finalize:** clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

- Processes are grouped as a **communicator**
  - Each message is sent & received in the same communicator
  - There is a default communicator whose group contains all initial processes, called MPI_COMM_WORLD

Communicators

```
int MPI_Comm_size(
      MPI_Comm   comm          /* in  */,
      int*       comm_sz_p     /* out */);
```

Collect number of processes in the communicator

```
int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

Return my rank (the process making this call)

a number between 0 and size-1, identifying the calling process

p processes are numbered *0, 1, 2, .. p-1*

Basic Send

msg_size=4 elments

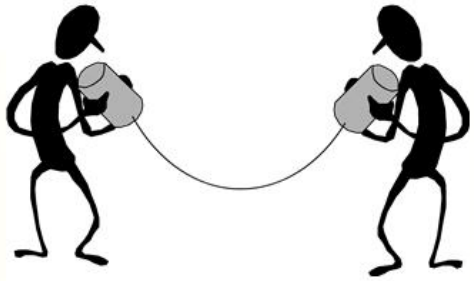
msg_type=MPI_INT

What is size of this buffer in bytes?

```
int MPI_Send(
    void*          msg_buf_p      /* in */,
    int            msg_size       /* in */,
    MPI_Datatype   msg_type       /* in */,
    int            dest           /* in */,
    int            tag            /* in */,
    MPI_Comm       communicator   /* in */);
```

Not in bytes

- **Send a message**
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?

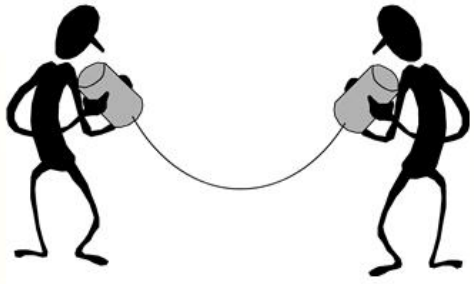


Data types

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

Basic Receive: Block until a matching message is received

```
int MPI_Recv(
      void*          msg_buf_p      /* out */,
      int            buf_size       /* in  */,
      MPI_Datatype   buf_type       /* in  */,
      int            source         /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* out */);
```



- **Things to specify:**
  - Where to receive data
  - How will the receiver recognize/screen messages?
  - What is the actual message received

Message matching

MPI_send() does not guarantee the other party receives the sent message. The system may just copy the message to an internal buffer of the MPI library and then this function returns.

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```


```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

MPI_Recv() waits until a message is received

Receiving messages without knowing source or tag

- **A receiver can get a message without knowing:**
  - the amount of data in the message,
  - the sender of the message,
    - `Specify the source as MPI_ANY_SOURCE`
  - or the tag of the message.
    - `Specify the tag as MPI_ANY_TAG`

How to check who sent me a message and what tag is?

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Find out *who sent me, what tag is, error code, actual message length in* **MPI_Status***

Retrieving Further Information from status argument in C

- `Status` is a data structure allocated in the user's program.
- In C:

```
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 );
```

Slide source: Bill Gropp, ANL

MPI Example: Simple send/receive when no of processes =3

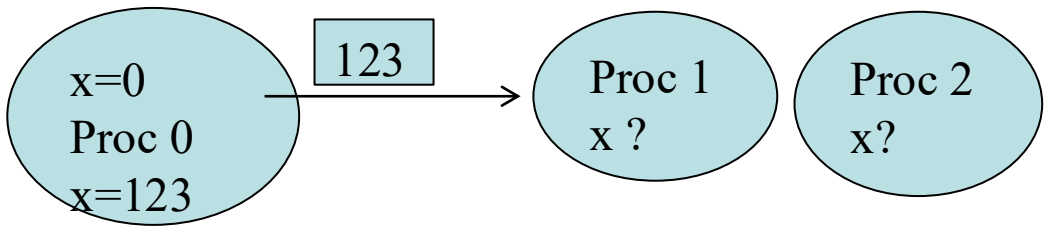
```
int x=0;
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
if (rank == 0) {
  x = 123;
  MPI_Send( &x, 1, MPI_INT, 1, 0,
        MPI_COMM_WORLD);
  x=456;
} else if (rank == 1) {
  MPI_Recv( &x, 1, MPI_INT, 0, 0,
        MPI_COMM_WORLD,  &status );
}
printf( "Proc %d: x=%d\n", rank, buf );
```

What to print?

Proc 0: x=? 456

Proc 1: x=? 123

Proc 2: x=? 0



What MPI Functions are commonly used

- **Startup**
  - MPI_Init, MPI_Finalize
- **Information on the processes**
  - MPI_Comm_rank, MPI_Comm_size
- **Point-to-point communication**
  - MPI_Send, MPI_Recv
- **Collective communication**
  - MPI_Bcast, MPI_Allreduce, MPI_Reduce, MPI_Allgather
- **`To measure time: MPI_Wtime()`**

double Start time= MPI_Wtime();
Code segment to be timed
double End time=MPI_Wtime();
Time spent is *end_time – start_time.*

Collective Communication in MPI and Advanced Features

- Collective communication primitives
- Collective vs. Point-to-Point Communications

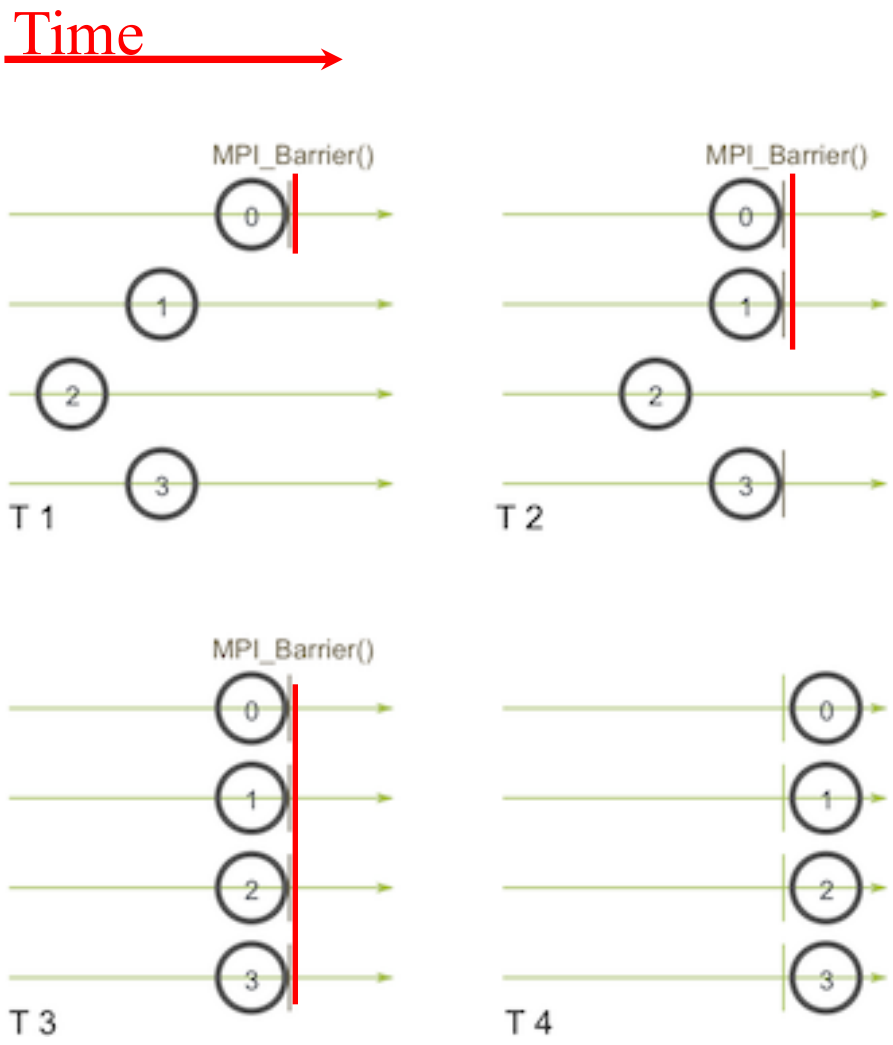
http://mpitutorial.com/mpi-broadcast-and-collective-communication/

MPI Collective Communication

- ***Collective* routines provide a higher-level way to organize a parallel program**
  - Each process executes the same communication operations
  - Communication and computation is coordinated among a group of processes in a communicator
  - Tags are not used
  - No non-blocking collective operations.
- **Operation types:** synchronization, data movement, collective computation.
  - MPI_Barrier
  - MPI Broadcast, Scatter, Gather, Reduce

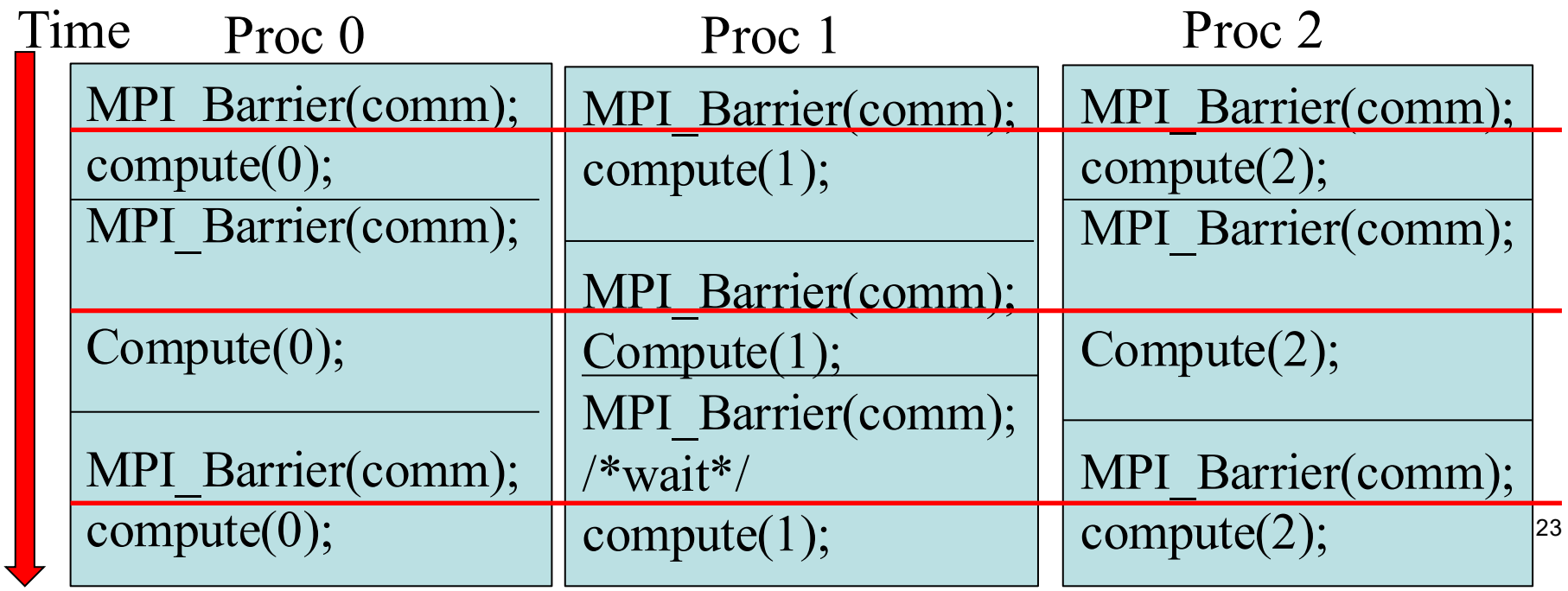
Synchronization with `MPI_Barrier(comm)`

- Blocks until all processes in the group of the communicator `comm` call it.
- Used in synchronized actions among processes
  - Measuring performance
- The right example illustrates MPI_Barrier is called by 4 processes and they wait for each other from time T1 to T4 (a synchronization point)

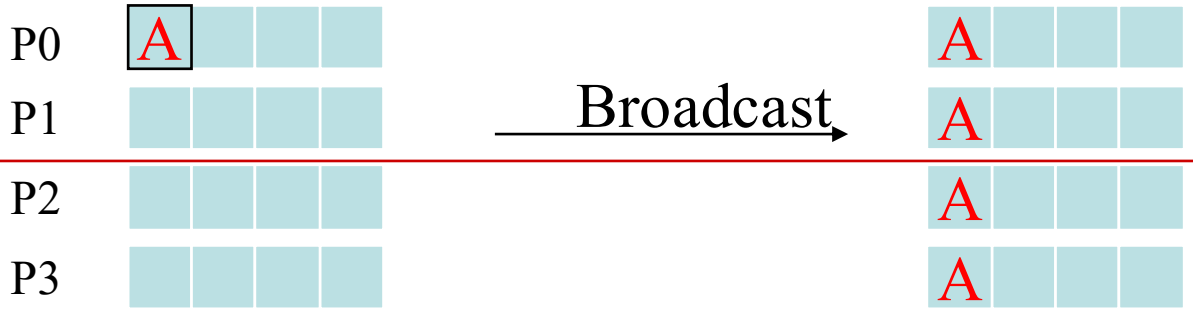


Exampe of SPMD MPI Loop with Barriers

```
MPI_Comm comm= MPI_COMM_WORLD;
MPI_Comm_rank(comm,&my_rank);
for (i=0; i<3; i++){
     MPI_Barrier(comm);
     compute(my_rank);}
```



Broadcast



- **Data belonging to a single process is sent to all of the processes in the communicator.**

in for source processes; out for others

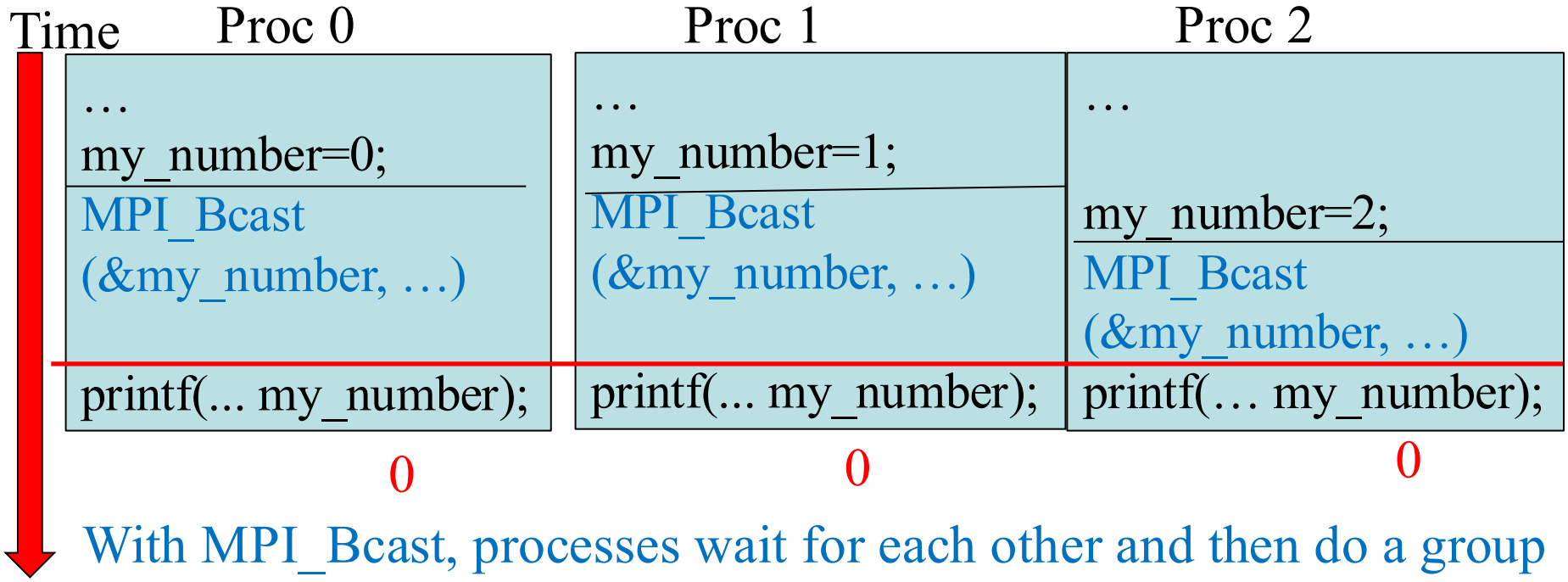
```
int MPI_Bcast(
        void*          data_p        /* in/out */,
        int            count         /* in     */,
        MPI_Datatype   datatype      /* in     */,
        int            source_proc   /* in     */,
        MPI_Comm       comm          /* in     */);
```

- MPI_Bcast is called by both the sender (called the root process) and the processes that receive the broadcast
  - MPI_Bcast is not a “multi-send”
  - “source_proc” is the rank of the sender; this tells MPI which process originates the broadcast and which receive

Broadcast example

Contribute 1 integer.
Broadcast from Proc 0

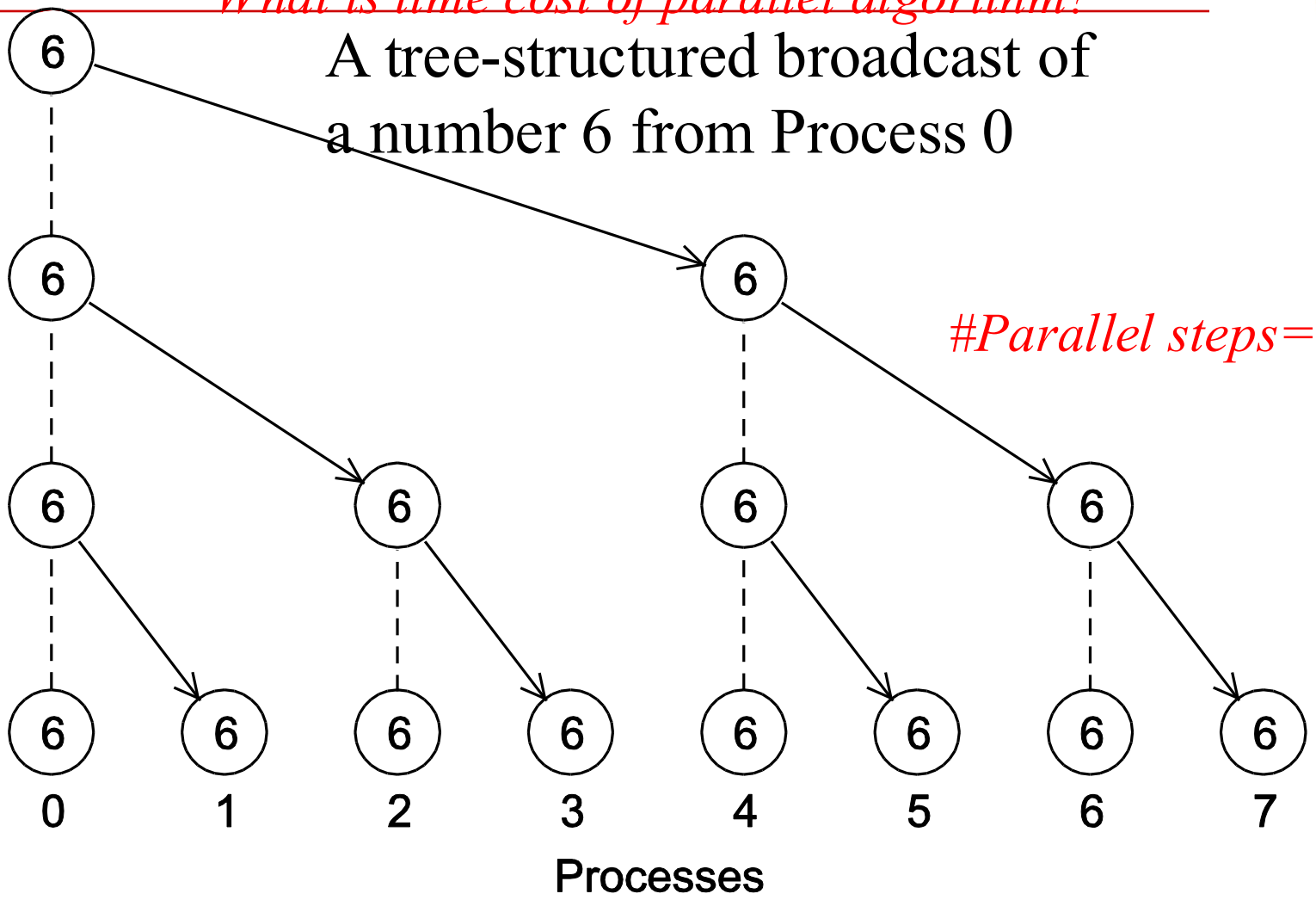
```
int my_rank, my_number;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
my_number=my_rank;
MPI_Bcast(&my_number,1, MPI_INT, 0,MPI_COMM_WORLD);
printf("%d\n",my_number);
```



With MPI_Bcast, processes wait for each other and then do a group communication with Proc 0 as the source of broadcasting

How is Broadcast implemented?

What is time cost of parallel algorithm?

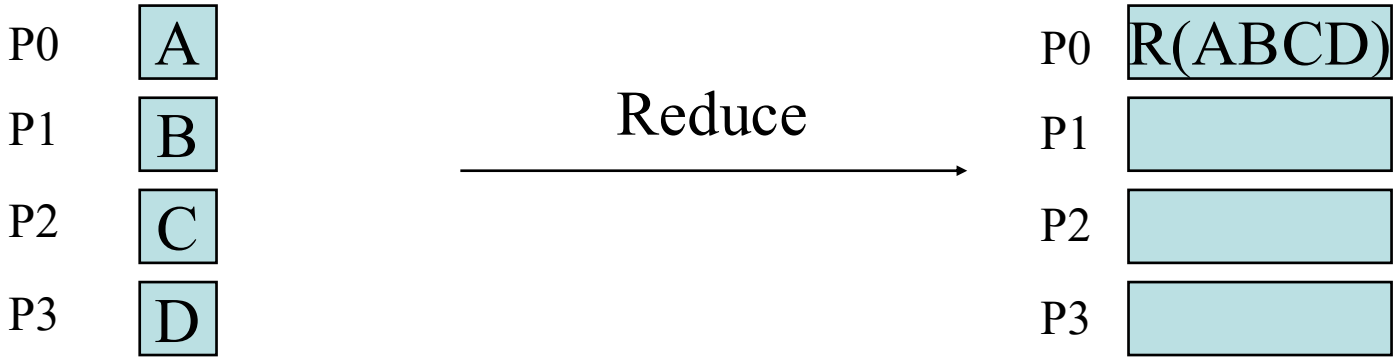


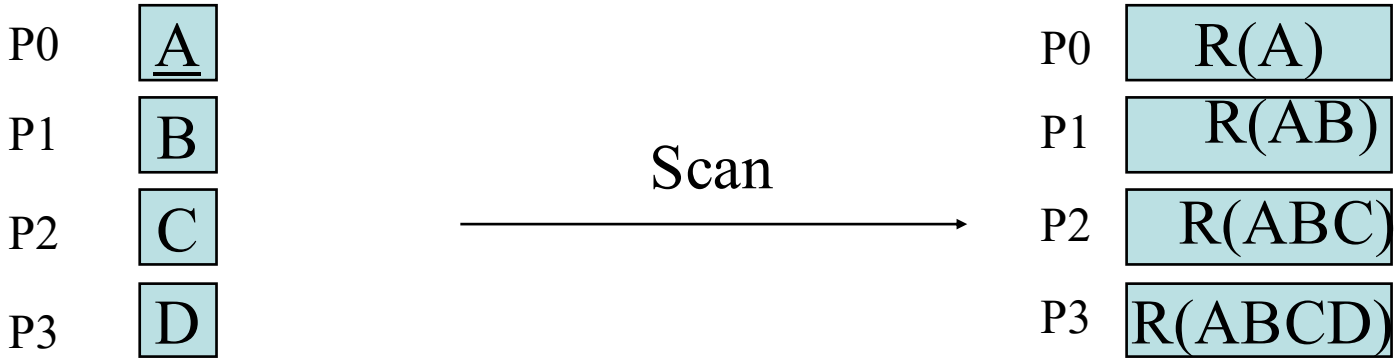
#Parallel steps=log p

- *All processes participate in group communication*
- *Tree structure is the reverse structure of tree summation*

Collective Computation: Reduce vs. Scan

R is the reduction function. For example, sum, max, min, product





MPI_Reduce

```
int MPI_Reduce(
      void*         input_data_p    /* in  */,
      void*         output_data_p   /* out */,
      int           count           /* in  */,
      MPI_Datatype  datatype        /* in  */,
      MPI_Op        operator        /* in  */,
      int           dest_process    /* in  */,
      MPI_Comm      comm            /* in  */);
```

Predefined reduction operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

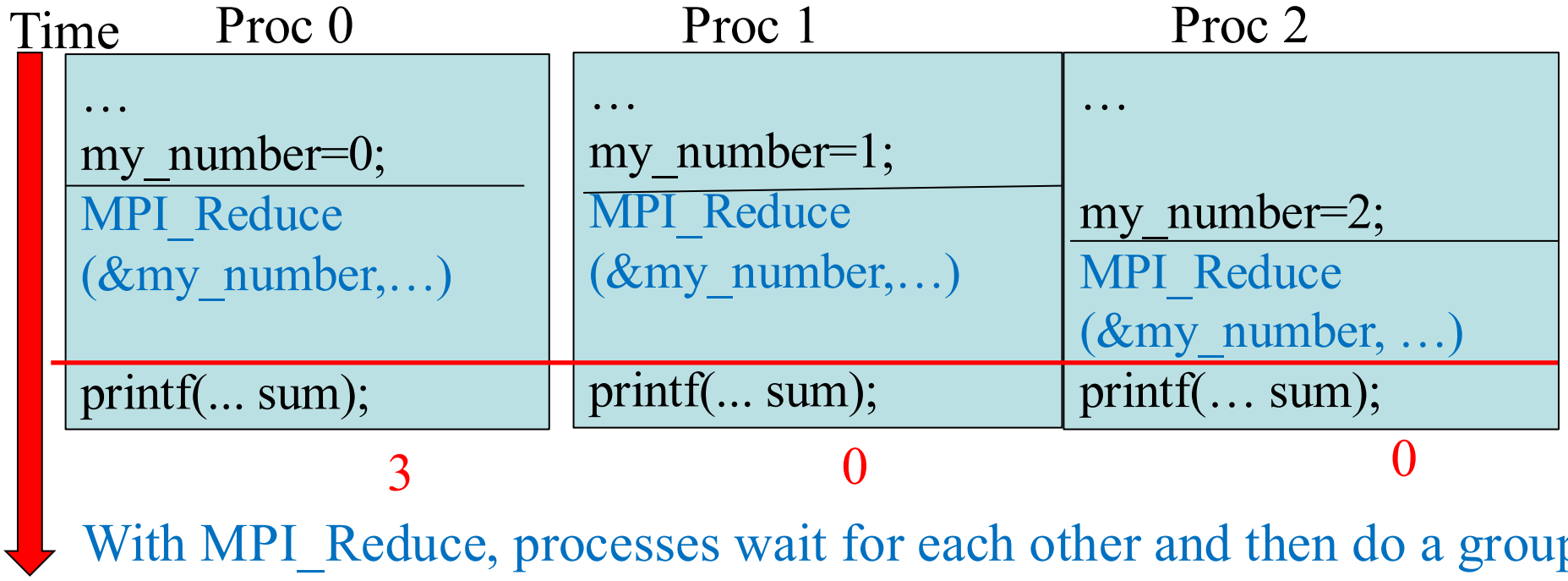
Predefined reduction operators in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Reduce example

Contribute 1 integer.
Reduced to Proc 0

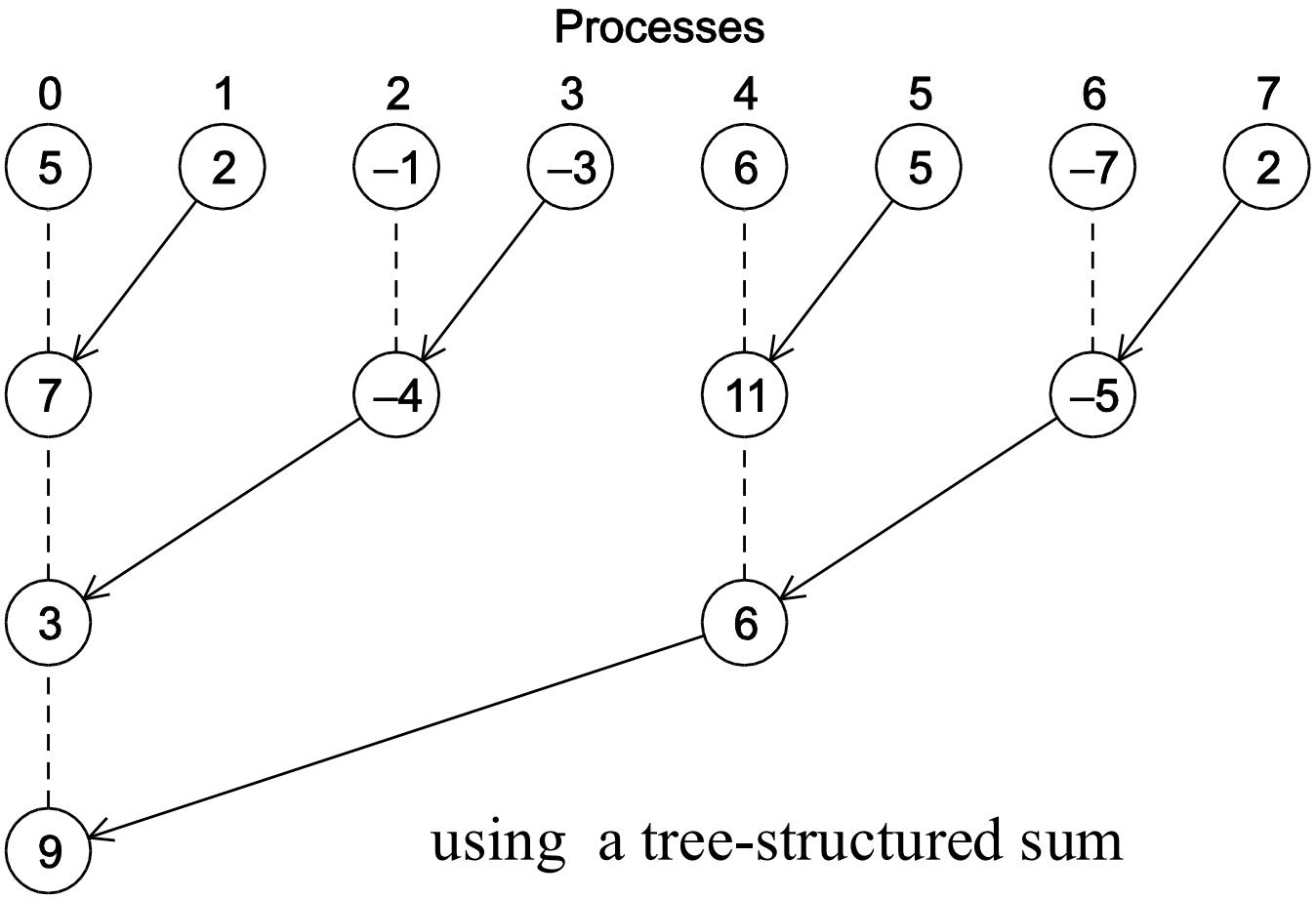
```
int my_rank, my_number, sum=0;
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
My_number=my_rank;
MPI_Reduce(&my_number,&sum, 1, MPI_INT, MPI_SUM,
   0,MPI_COMM_WORLD);
printf("%d\n",sum);
```



With MPI_Reduce, processes wait for each other and then do a group communication with Proc 0 holding the reduction result.

How to implement global reduction?

What is time cost of parallel algorithm?



#Parallel steps=log p

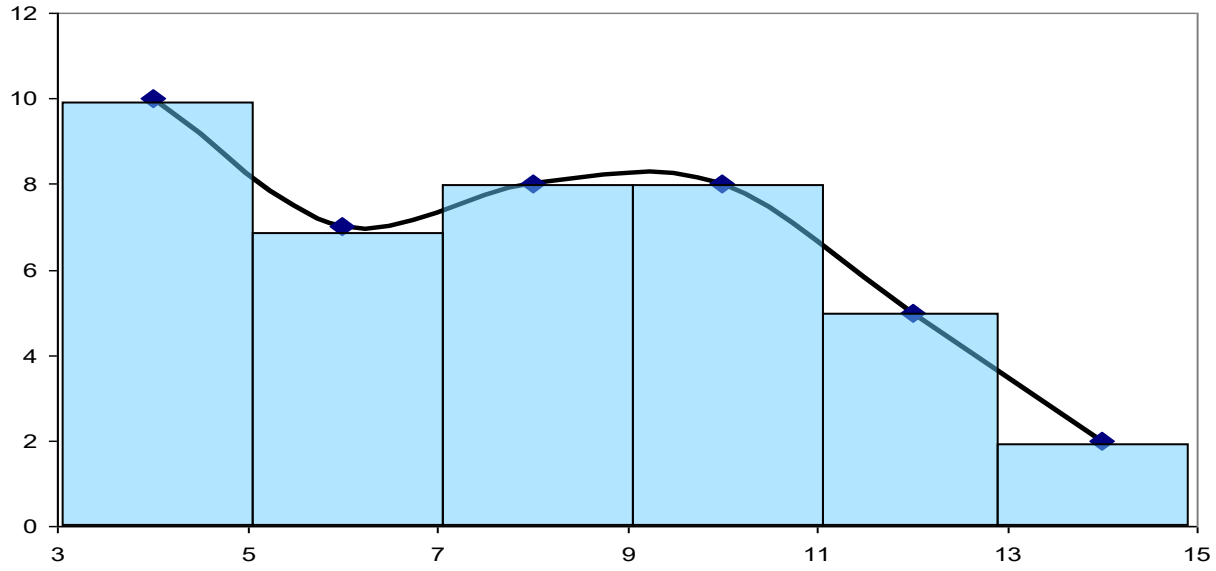
Example: Use of MPI_Reduce in Computing Pi with Numerical Integration

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

Divide an integral area into n segments. Approximate as

$$\pi = \sum 1/(1+x^2)$$

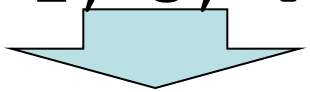
```
h   = 1.0
sum = 0.0;
for(i=1; i <= n; i++) {
  x = h *(i - 0.5)/n;
  sum += 4/ (1+ x*x);
}
mypi = h * sum;
```



Mapping of loop iterations with SPMD code

Mapping with 2 processes (myID=0, 1)

Iteration 1, 2, 3, 4, 5, 6, 7, 8



Processes 0, 0, 0, 0, 1, 1, 1, 1

What is the mapping of this SPMD code?

```
for (i = myID + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
```

Assume 2 processes

myID=0 → Iterations? 1, 3, 5, 7

myID=1 → Iterations? 2, 4, 6, 8

Cyclic pattern

Options for Iteration-to-Process Mapping

- **Block mapping**
  - Assign blocks of consecutive components to each process.
- **Cyclic mapping**
  - Assign components in a round robin fashion.
- **Block-cyclic mapping**
  - Use a cyclic distribution of blocks of components.

Example: map 12 iterations to 3 processes

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Use of Parallel Integral for Pi Computation

$$\pi = 4\int_0^1 \frac{1}{1+x^2}\,dx = \sum 1/(1+x^2)$$

```
#include "mpi.h"
#include <math.h>
   #include <stdio.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.14159265358979323846264 3;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

```
while (!done)  {
  if (myid == 0) {
    printf("Enter the number of intervals: (0 quits) ");
    scanf("%d",&n);
  }
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0) break;
```

Input and broadcast parameters

Slide source: Bill Gropp, ANL

Example: Pi computation

1. Compute local pi values

```
h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
  x = h * ((double)i - 0.5);
  sum += 4.0 / (1.0 + x*x);
}
mypi = h * sum;
```

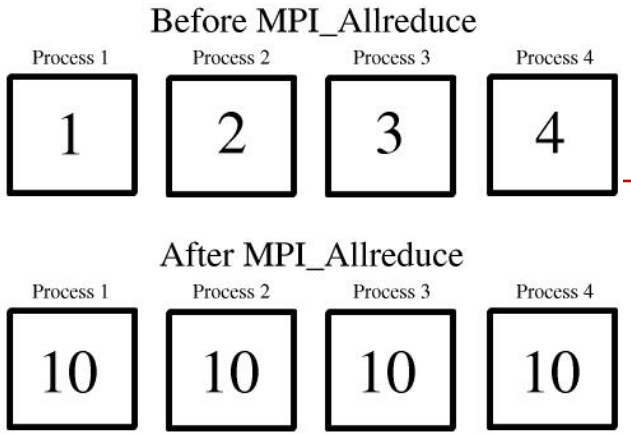
Iteration-to process cyclic mapping

2. Compute summation

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
if (myid == 0)
  printf("pi is approximately %.16f, Error is .16f\n",
         pi, fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
}
```

Slide source: Bill Gropp, ANL

MPI_Allreduce

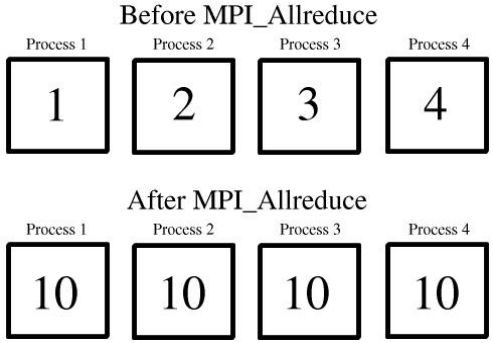


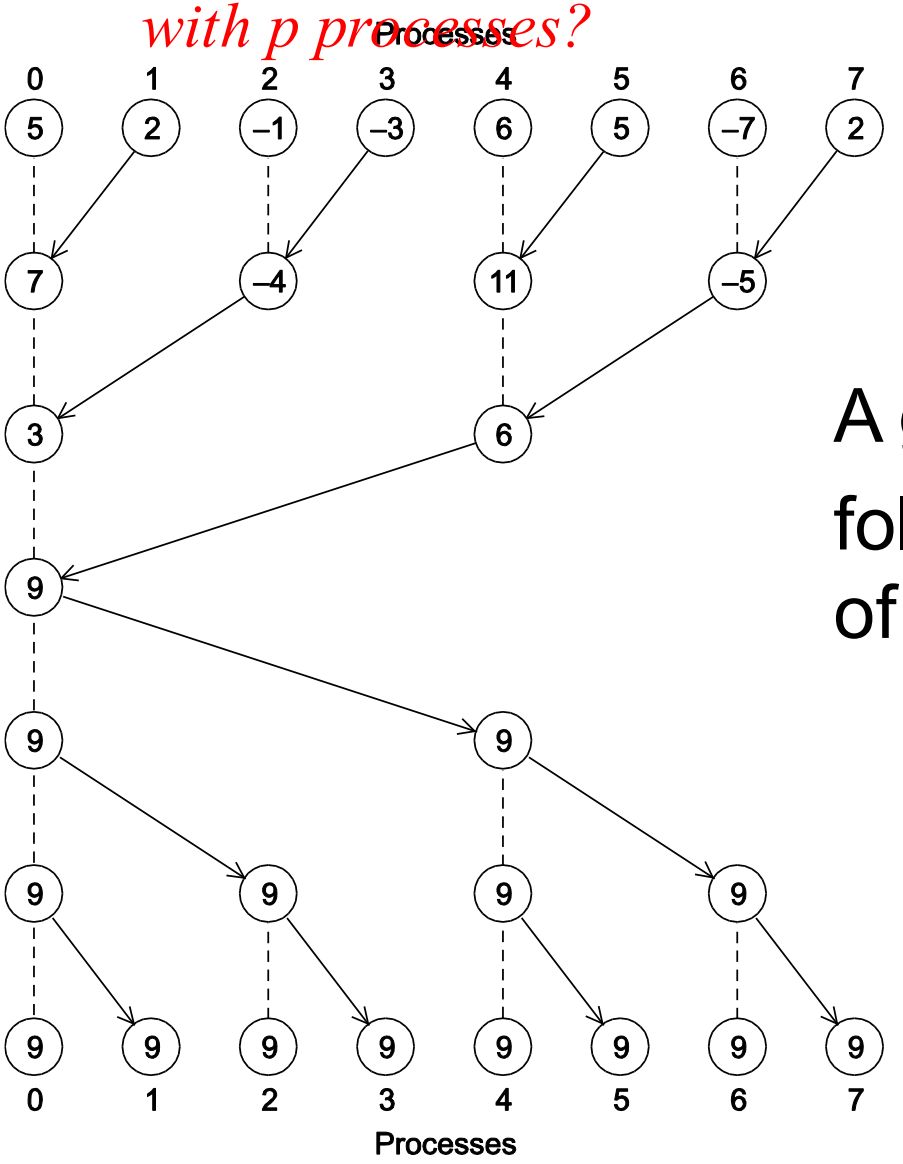
- **Do global reduction first and then make the result available to all of the processes.**

```
int MPI_Allreduce(
        void*        input_data_p   /* in  */,
        void*        output_data_p  /* out */,
        int          count          /* in  */,
        MPI_Datatype datatype       /* in  */,
        MPI_Op       operator       /* in  */,
        MPI_Comm     comm           /* in  */);
```

Implementation of MPI_Allreduce

What is time cost of your parallel algorithm with p processes?

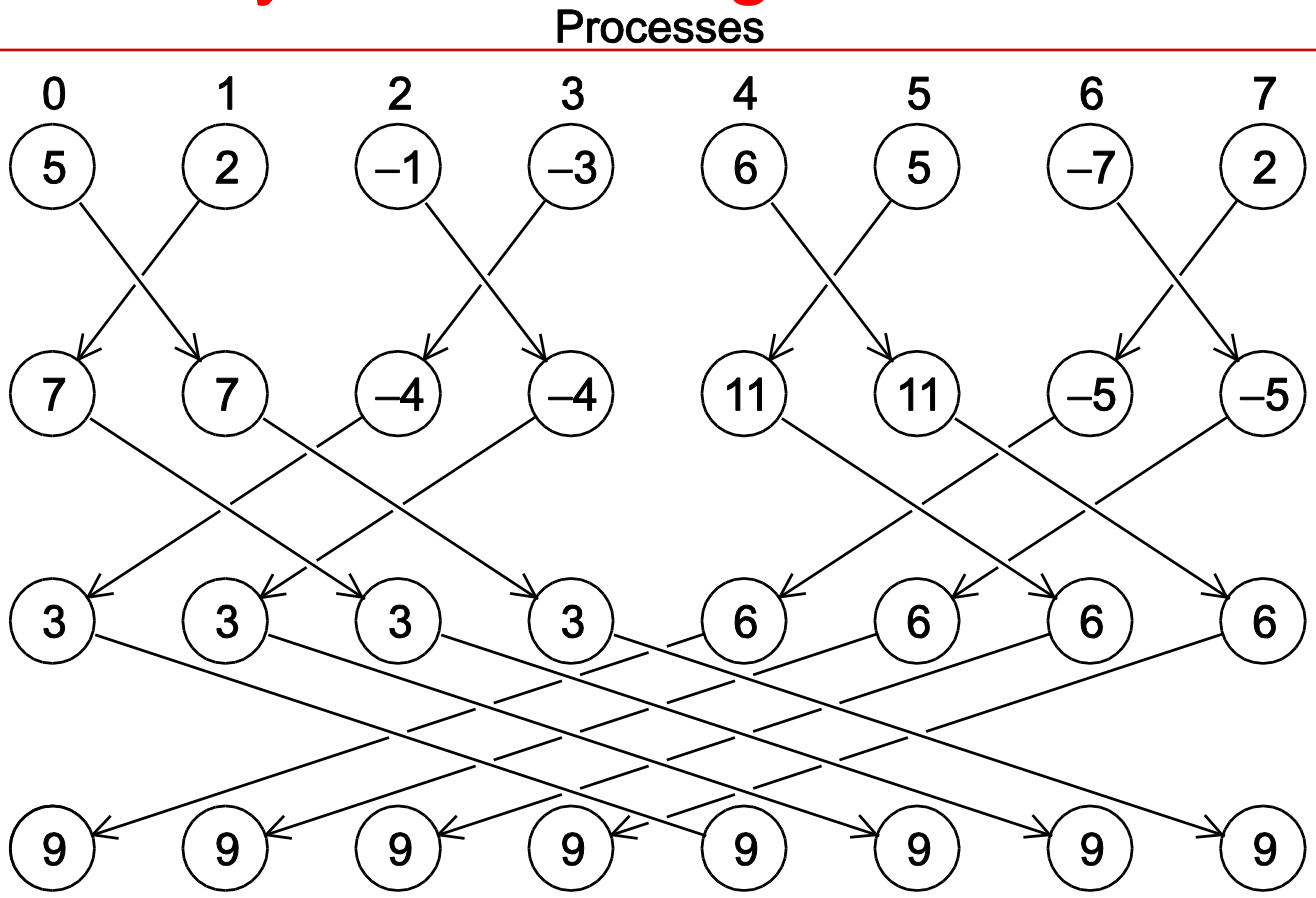




A global tree reduction followed by tree broadcast of result.

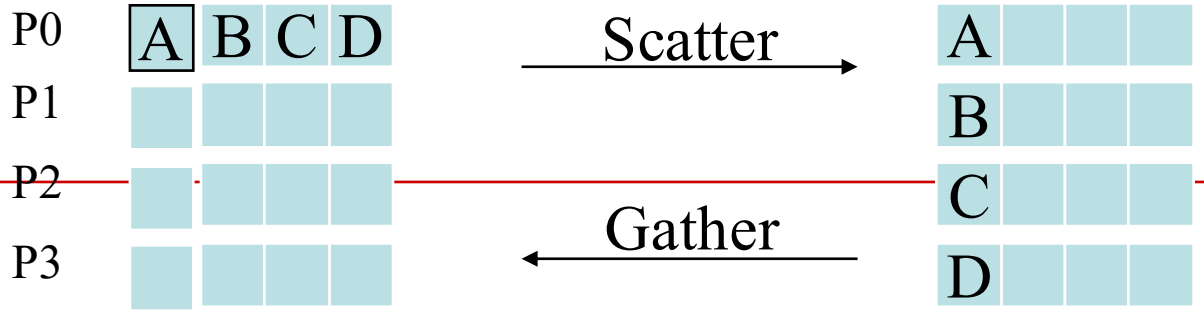
#Parallel steps=2log p

A butterfly-structured global sum



Fast group communication is a challenging problem and we just call the vendor-supplied library functions which have a good implementation

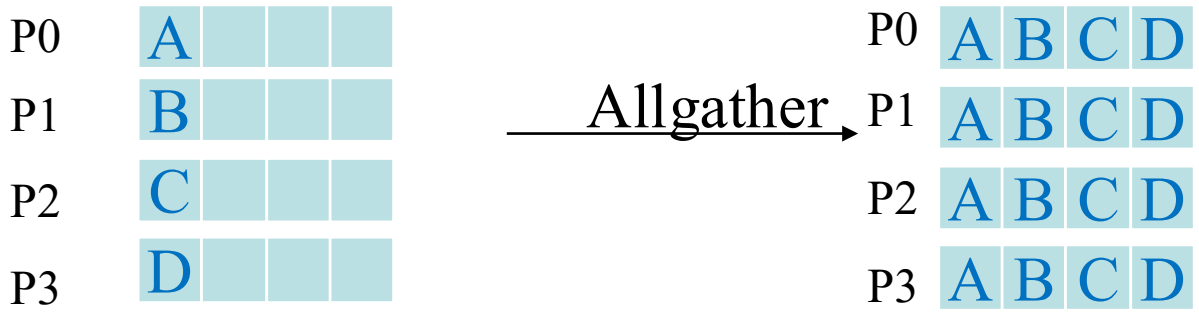
Scatter for data distribution

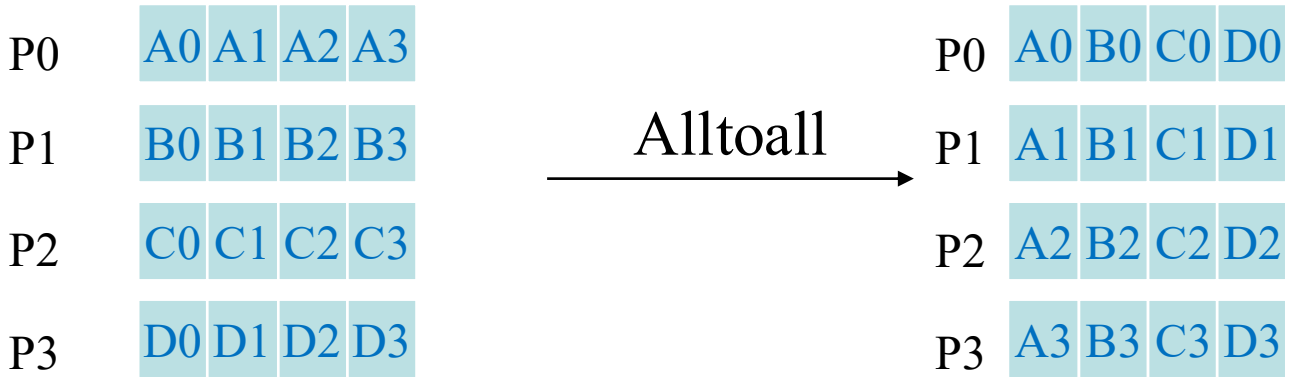


- **MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.**

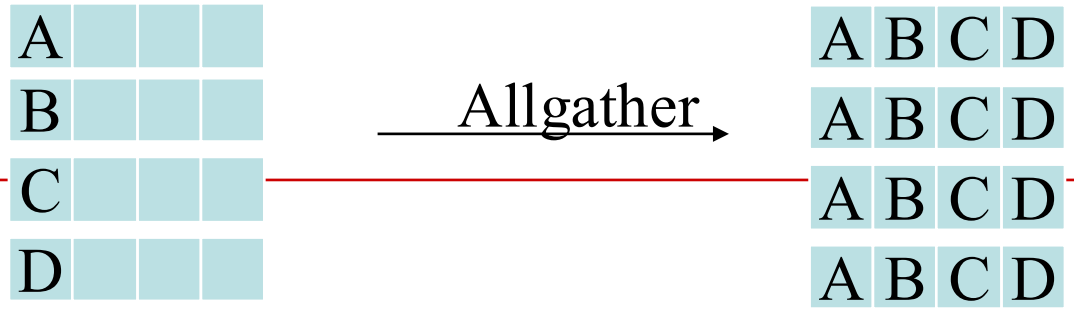
```
int MPI_Scatter(
      void*          send_buf_p  /* in  */,
      int            send_count  /* in  */,
      MPI_Datatype   send_type   /* in  */,
      void*          recv_buf_p  /* out */,
      int            recv_count  /* in  */,
      MPI_Datatype   recv_type   /* in  */,
      int            src_proc    /* in  */,
      MPI_Comm       comm        /* in  */);
```

Collective Data Movement: Allgather and AlltoAll





Allgather



- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.
- As usual, recv_count is the amount of data being received from each process.

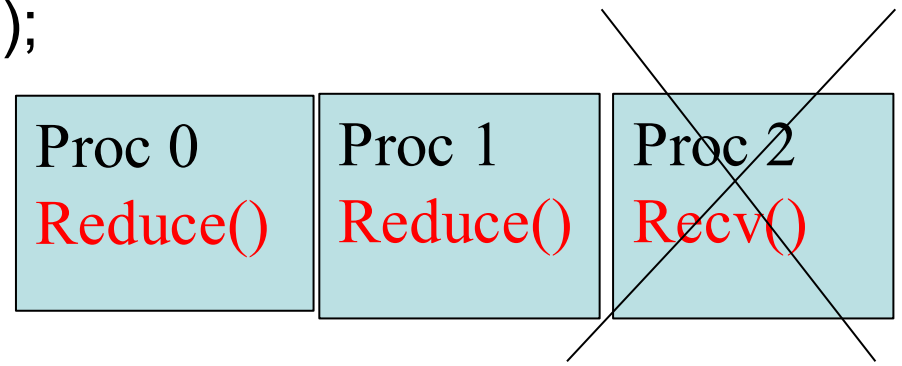
```
int MPI_Allgather(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      MPI_Comm      comm        /* in  */);
```

Collective vs. Point-to-Point Communications

- **All the processes in the communicator must call the same collective function.**
  - For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process. What will happen?

```
if(my_rank==2) MPI_Recv(&a, MPI_INT, MPI_SUM,0,0,
   MPI_COMM_WORLD);
Else MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 0,
   MPI_COMM_WORLD);
```

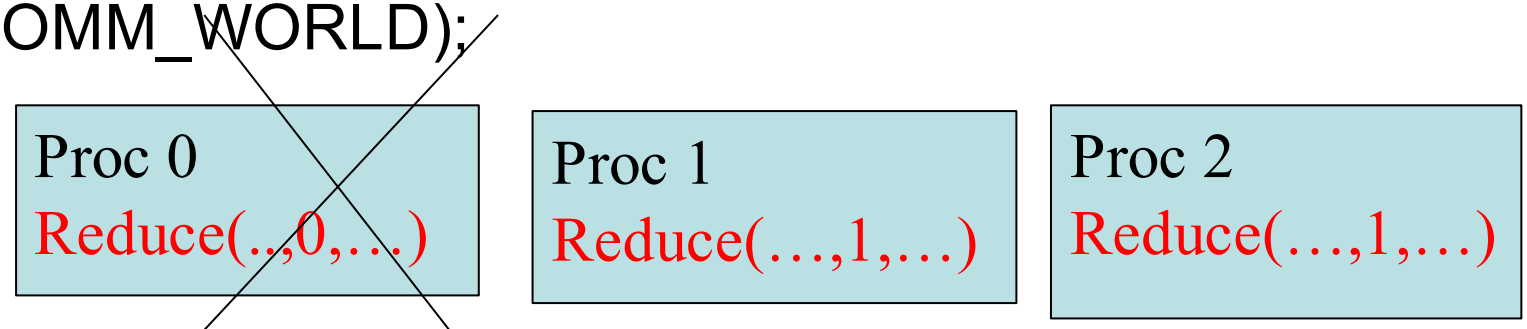
The program will hang or crash.



Collective vs. Point-to-Point Communications

- **The arguments passed by each process to an MPI collective communication must be "compatible."**
  - For example, if one process passes in 0 as the final dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

```
if(my_rank==0)  MPI_Reduce(&a,&b,1, MPI_INT,
   MPI_SUM, 0, MPI_COMM_WORLD);
else MPI_Reduce(&a,&b,1, MPI_INT, MPI_SUM, 1,
   MPI_COMM_WORLD);
```



Example of MPI_Reduce execution

Multiple calls to MPI_Reduce with MPI_SUM and Proc 0 as destination (root)

Will this code exectuon hang?

int a, b, c,d;

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

Is b=3 on Proc 0 after two MPI_Reduce() calls?

Is d=6 on Proc 0?

Example of MPI_Reduce execution

Multiple calls to MPI_Reduce with MPI_SUM and Proc 0 as destination (root)

Will this code exectuon hang?

int a, b, c,d;

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2			

After Step 1: Proc 0: b=4 which is the sum of 1 + 2 +1

The names of the memory locations are less relevant to the matching of the calls to MPI_Reduce.

Example of MPI_Reduce execution

Multiple calls to MPI_Reduce with MPI_SUM and Proc 0 as destination (root)

Will this code exectuon hang?

int a, b, c,d;

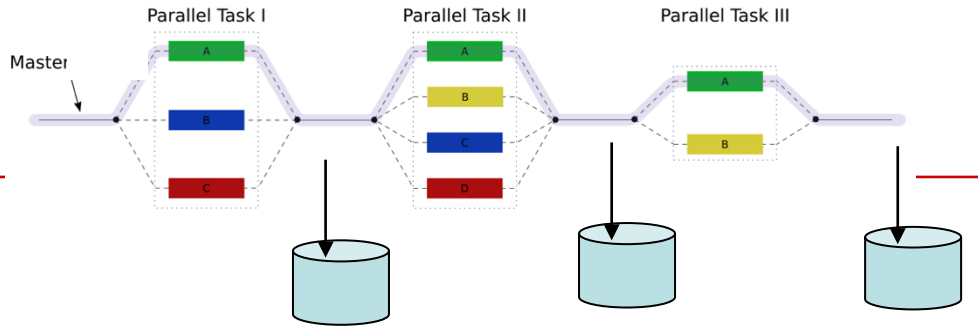
Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1			
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

After Step 1: Proc 0: b=4.

After Step 2: Proc 0: b=4.

Proc 0: d=2 + 1 +2

Checkpointing



- **Computers fail periodically**
  - Result loss of an expensive long-running job (e.g. that lasts days or weeks)
- **Checkpointing technique**
  - A program periodically stores information on its intermediate running state
  - A program can restart and resume its execution from a checkpoint file
- **Use of checkpointing in an MPI program**
  - A process periodically gathers running results from other parallel processes and saves to a file.
  - This program is designed to be restartable after reading from a checkpoint file.

MPI Functions: Summary

- **Startup/finishing:** MPI_Init, MPI_Finalize
- **Information on the processes**
  - MPI_Comm_rank, MPI_Comm_size
- **Point-to-point communication:** MPI_Send, MPI_Recv
  - matched on the basis of tags and communicators.
- **Collective communication.** Don't use tags.
  - They're matched solely on the basis of the communicator and function definition.
- **Checkpointing helps fault tolerance of long-running jobs**

- **To measure time:**
  `MPI_Wtime()`

double Start time= MPI_Wtime();
Code segment to be timed
double End time=MPI_Wtime();
Time spent is *end_time – start_time.*