

PARALLEL COMPUTING		Semester	VII
Course Code	BCS702	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory/Practical		

Course outcomes (Course Skill Set):

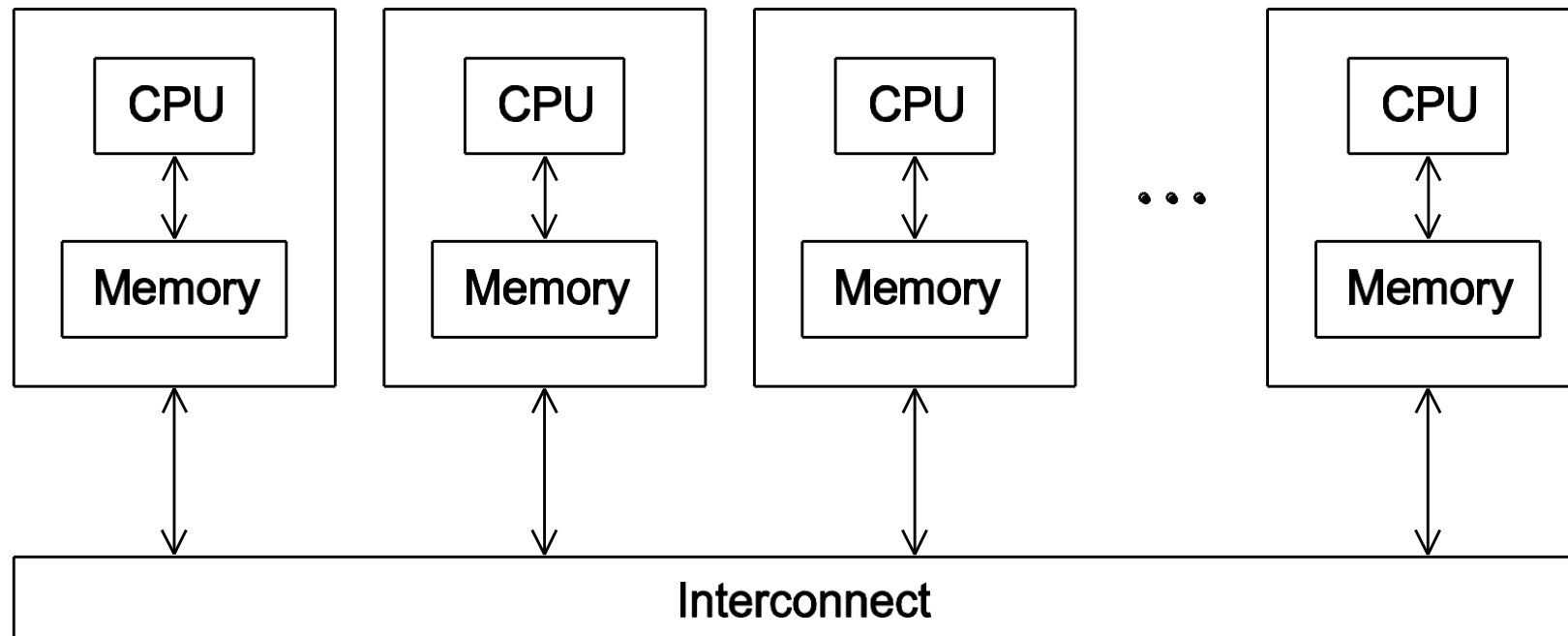
At the end of the course, the student will be able to:

- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem.

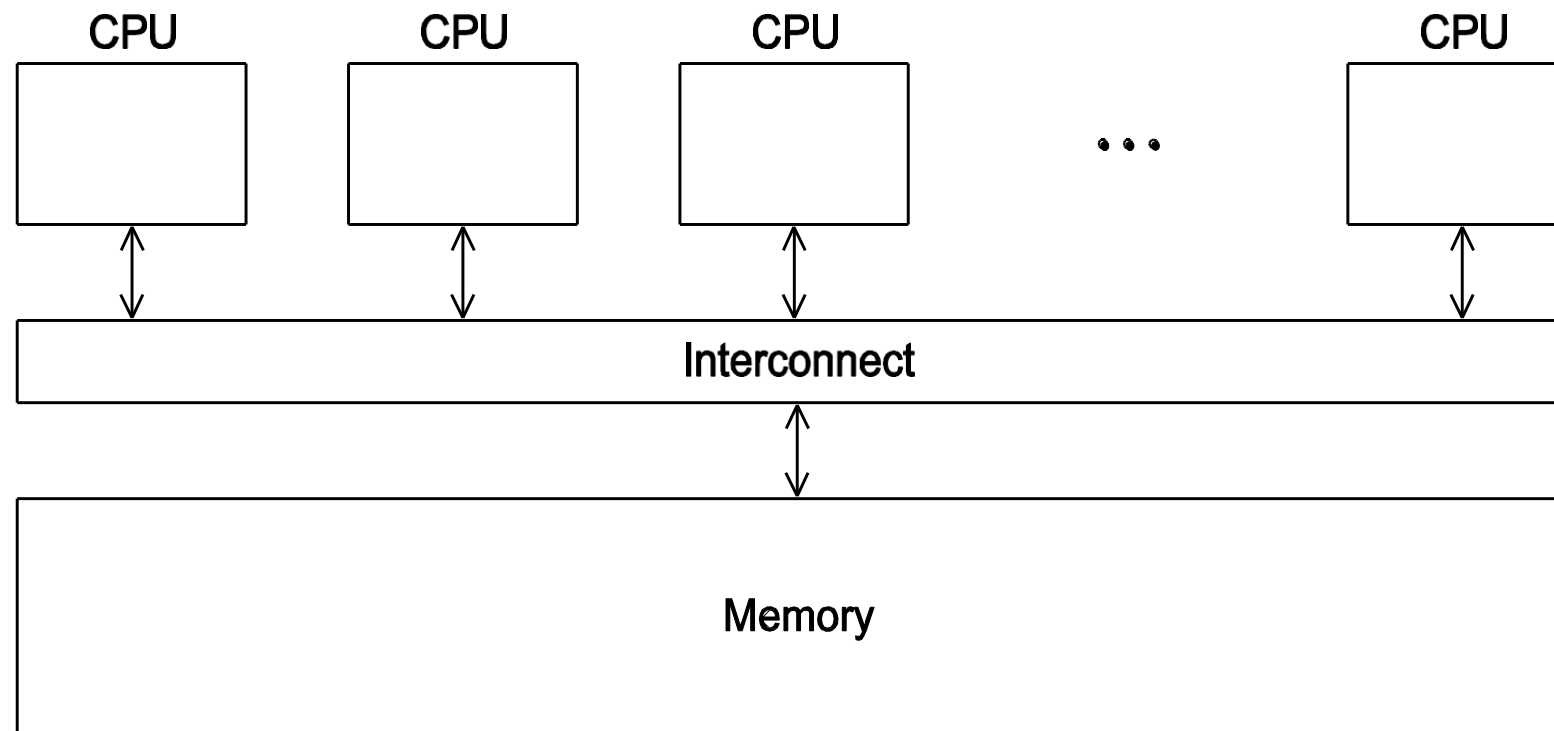
MODULE-3

Distributed memory programming with MPI – MPI functions, The trapezoidal rule in MPI, Dealing with I/O, Collective communication, MPI-derived datatypes, Performance evaluation of MPI programs, A parallel sorting algorithm.

A distributed memory system



A shared memory system



Hello World!

```
#include <stdio.h>

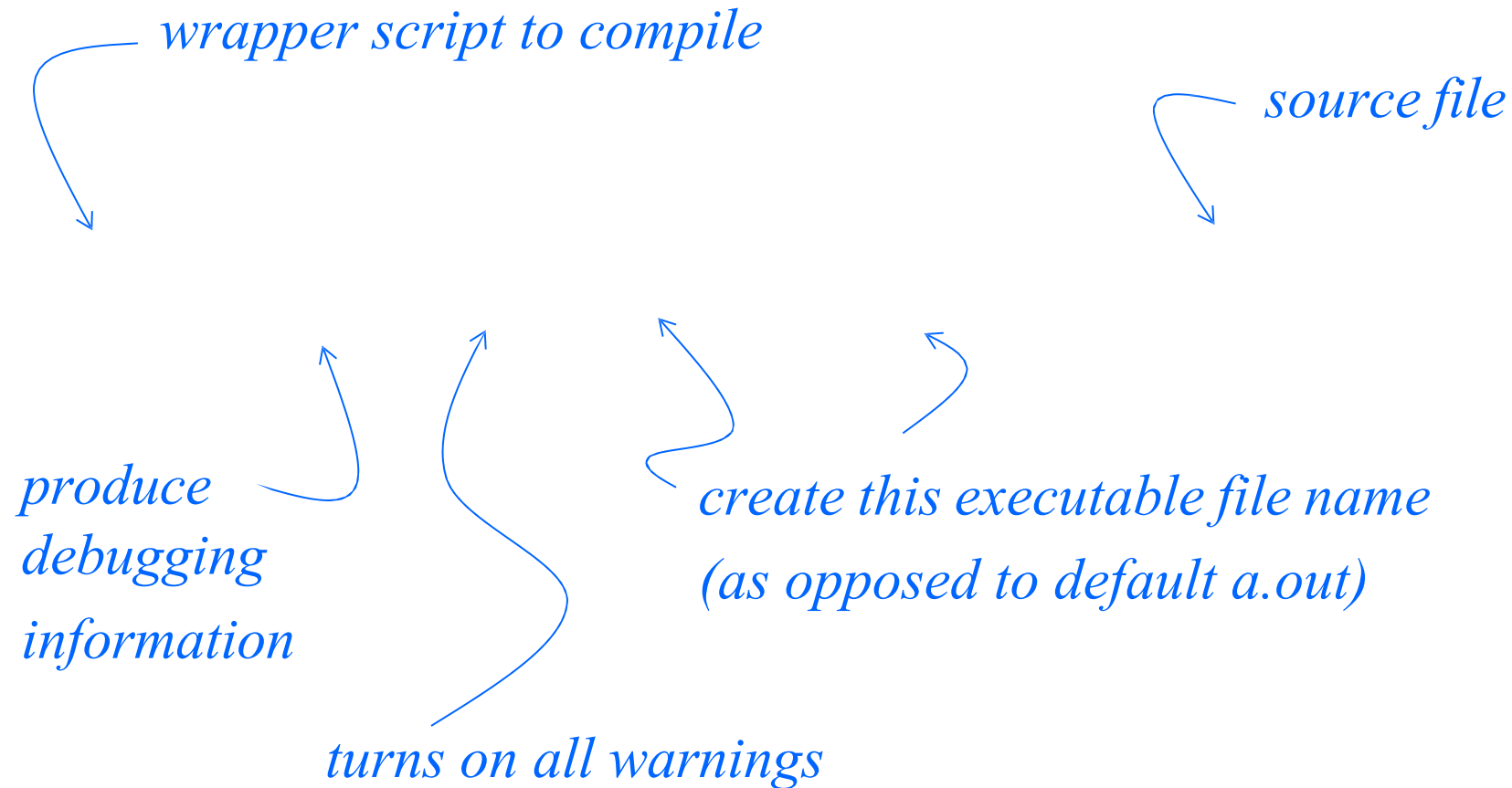
int main(void) {
    printf("hello, world\n");

    return 0;
}
```

Identifying MPI processes

Common practice to identify processes by nonnegative integer ranks.

p processes are numbered $0, 1, 2, \dots, p-1$



Compilation

`mpiexec -n <number of processes> <executable>`

`mpiexec -n 1 ./mpi_hello`

run with 1 process

`mpiexec -n 4 ./mpi_hello`

run with 4 processes

Execution

```
mpiexec -n 1 ./mpi_hello
```

Greetings from process 0 of 1 !

```
mpiexec -n 4 ./mpi_hello
```

Greetings from process 0 of 4 !

Greetings from process 1 of 4 !

Greetings from process 2 of 4 !

Greetings from process 3 of 4 !

MPI Programs

Written in C.

Has main.

Uses `stdio.h`, `string.h`, etc.

Need to add `mpi.h` header file.

Identifiers defined by MPI start with “MPI_”.

First letter following underscore is uppercase.

For function names and MPI-defined types.

Helps to avoid confusion.

MPI Components

MPI_Init

Tells MPI to do all the necessary setup.

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

MPI_Finalize

Tells MPI we're done, so clean up anything allocated for this program.

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

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

. . .
int main(int argc, char* argv[]) {
    . . .
    /* No MPI calls before this */
    MPI_Init(&argc, &argv);
    . . .
    MPI_Finalize();
    /* No MPI calls after this */
    . . .
    return 0;
}
```

A collection of processes that can send messages to each other.

MPI_Init defines a communicator that consists of all the processes created when the program is started. Called **MPI_COMM_WORLD**.

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

number of processes in the communicator

Communicators

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

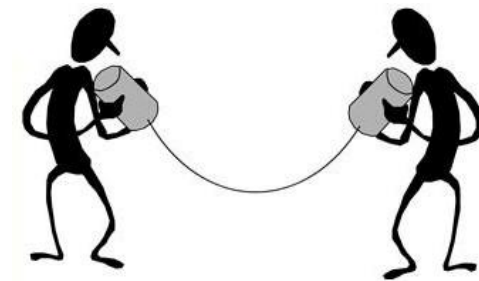
my rank

(the process making this call)

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	

Data types

```
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 */);
```



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

MPI_Send
src = q



MPI_Recv
dest = r

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

Message matching

Receiving messages

A receiver can get a message without knowing:

the amount of data in the message,

the sender of the message,

or the tag of the message.

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

MPI_Status*



MPI_SOURCE

MPI_TAG

MPI_ERROR

MPI_Status* status;

status.MPI_SOURCE

status.MPI_TAG

```
int MPI_Get_count(  
    MPI_Status* status_p /* in */,  
    MPI_Datatype type /* in */,  
    int* count_p /* out */);
```



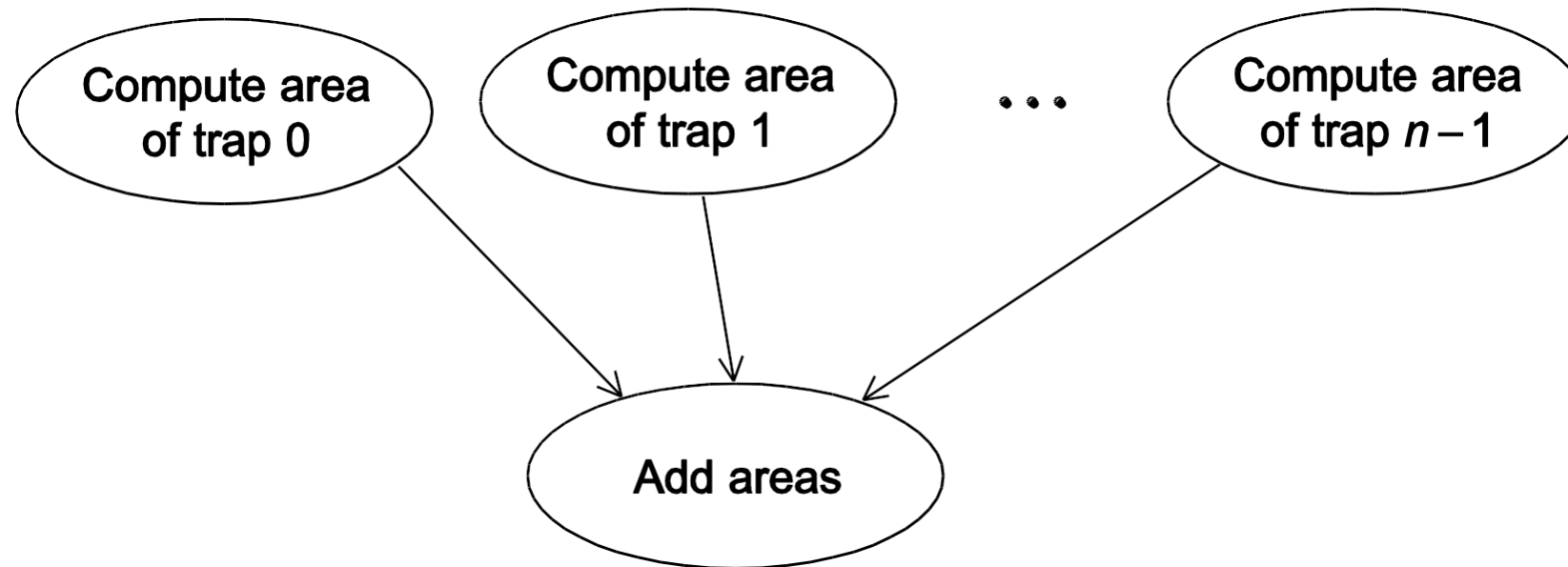
Exact behavior is determined by the MPI implementation.

MPI_Send may behave differently with regard to buffer size, cutoffs and blocking



```
/* Input:  a, b, n */  
h = (b-a)/n;  
approx = (f(a) + f(b))/2.0;  
for (i = 1; i <= n-1; i++) {  
    x_i = a + i*h;  
    approx += f(x_i);  
}  
approx = h*approx;
```

Tasks and communications for Trapezoidal Rule



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

int main(void) {
    int my_rank, comm_sz;

    MPI_Init(NULL, NULL);
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

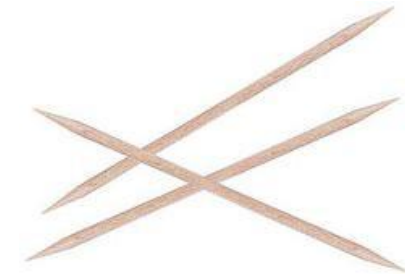
    printf("Proc %d of %d > Does anyone have a toothpick?\n",
           my_rank, comm_sz);

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

Dealing with I/O

```
Proc 0 of 6 > Does anyone have a toothpick?  
Proc 1 of 6 > Does anyone have a toothpick?  
Proc 2 of 6 > Does anyone have a toothpick?  
Proc 4 of 6 > Does anyone have a toothpick?  
Proc 3 of 6 > Does anyone have a toothpick?  
Proc 5 of 6 > Does anyone have a toothpick?
```

unpredictable output

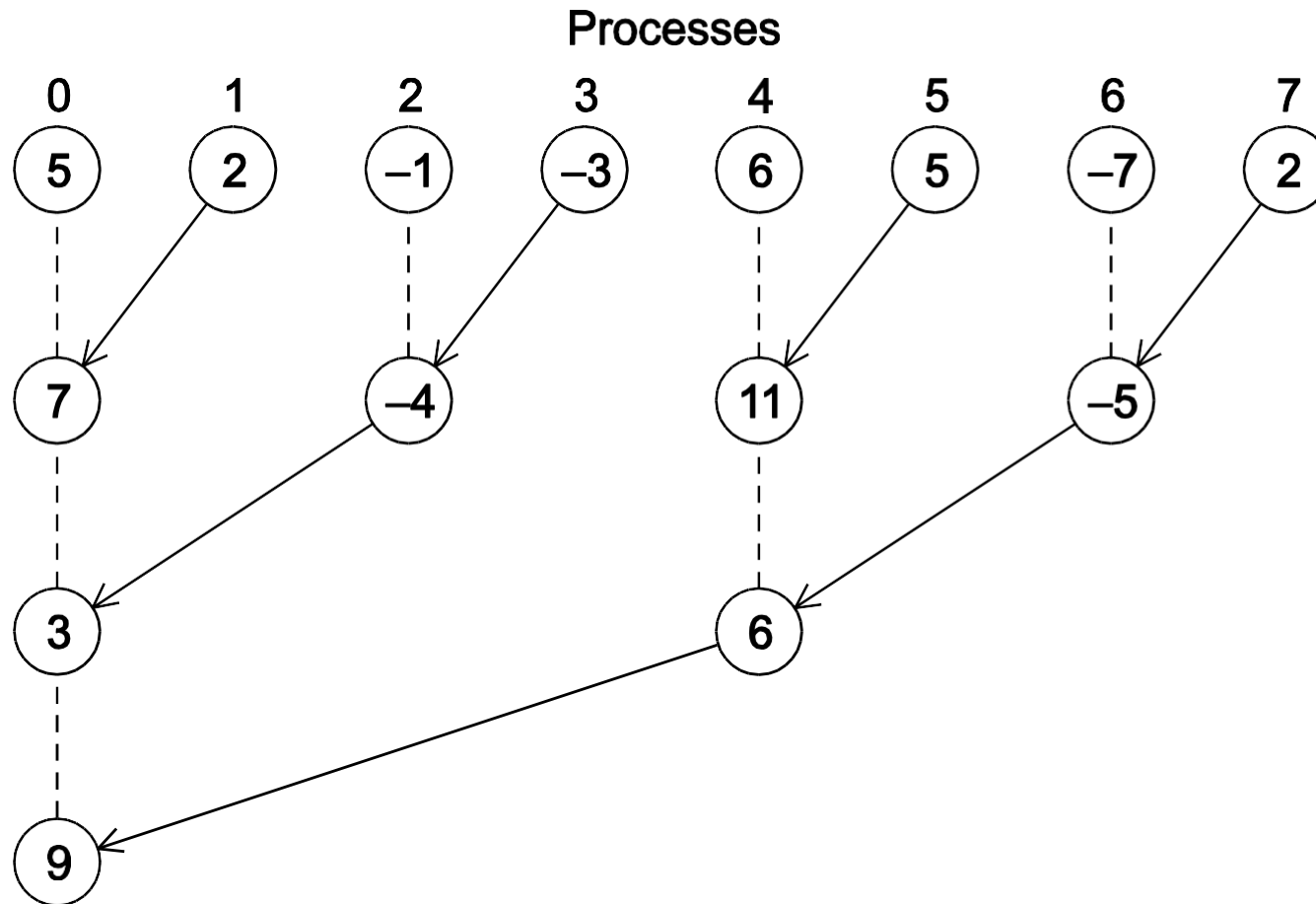


```
. . .  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);  
  
Get_data(my_rank, comm_sz, &a, &b, &n);  
  
h = (b-a)/n;  
. . .
```

Input



Collective communication



A tree-structured global sum

MPI_Reduce

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];  
.  
.  
.  
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,  
MPI_COMM_WORLD);
```

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

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 is erroneous, and, in all likelihood, the program will hang or crash.

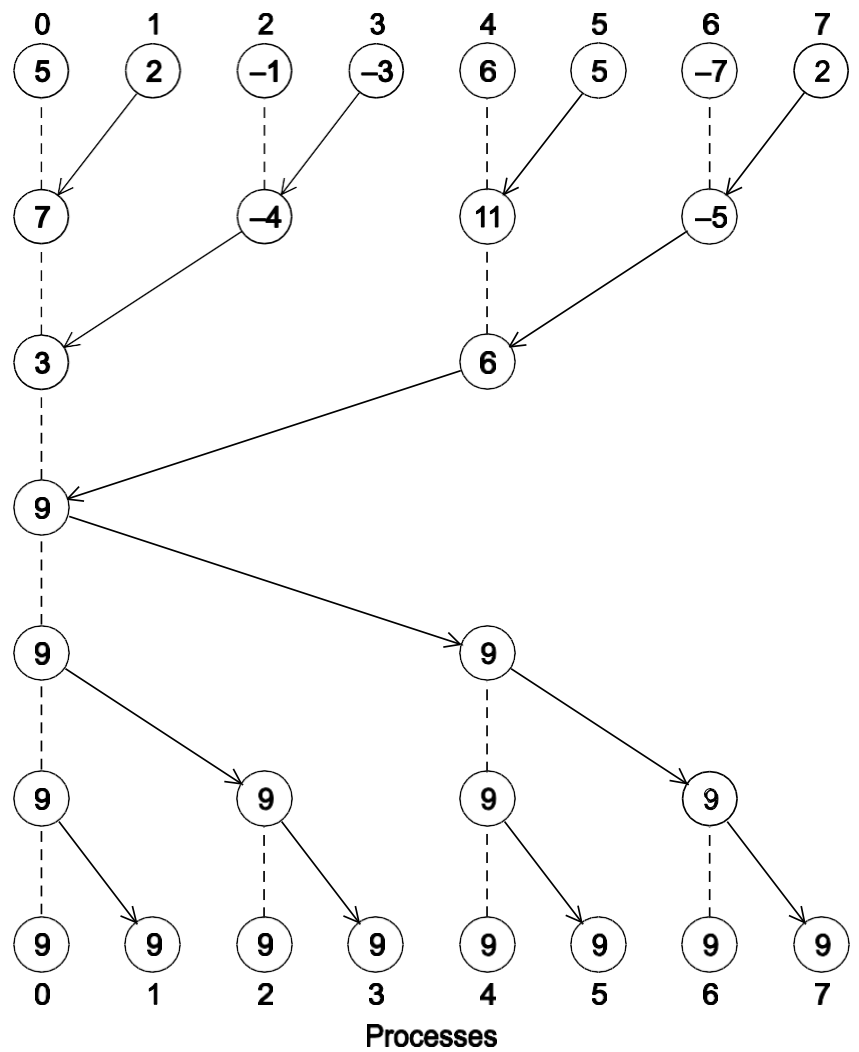
Example (1)

Time	Process 0	Process 1	Process 2
0	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>	<code>a = 1; c = 2</code>
1	<code>MPI_Reduce(&a, &b, ...)</code>	<code>MPI_Reduce(&c, &d, ...)</code>	<code>MPI_Reduce(&a, &b, ...)</code>
2	<code>MPI_Reduce(&c, &d, ...)</code>	<code>MPI_Reduce(&a, &b, ...)</code>	<code>MPI_Reduce(&c, &d, ...)</code>

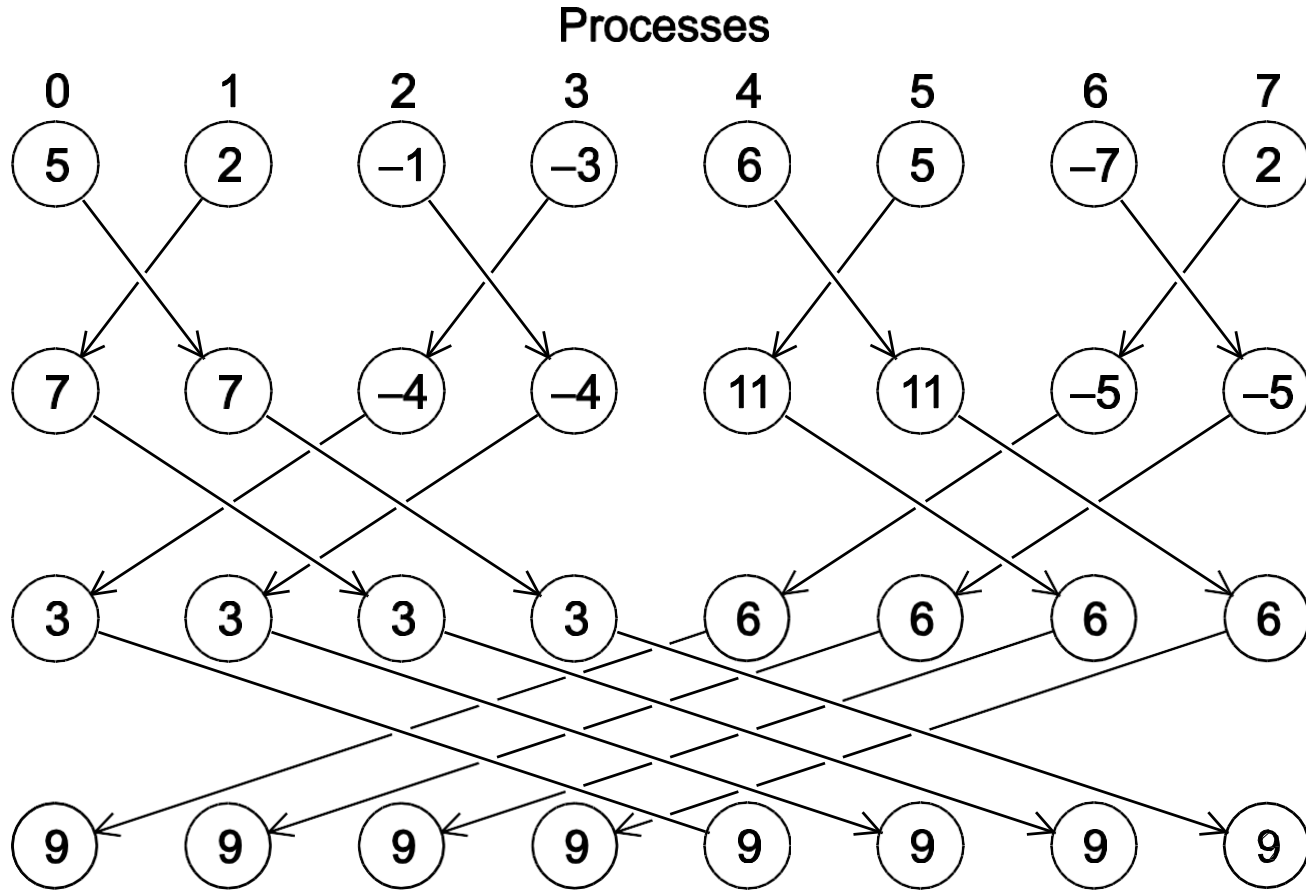
Multiple calls to MPI_Reduce

```
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 */);
```

MPI_Allreduce



*A global sum followed
by distribution of the
result.*

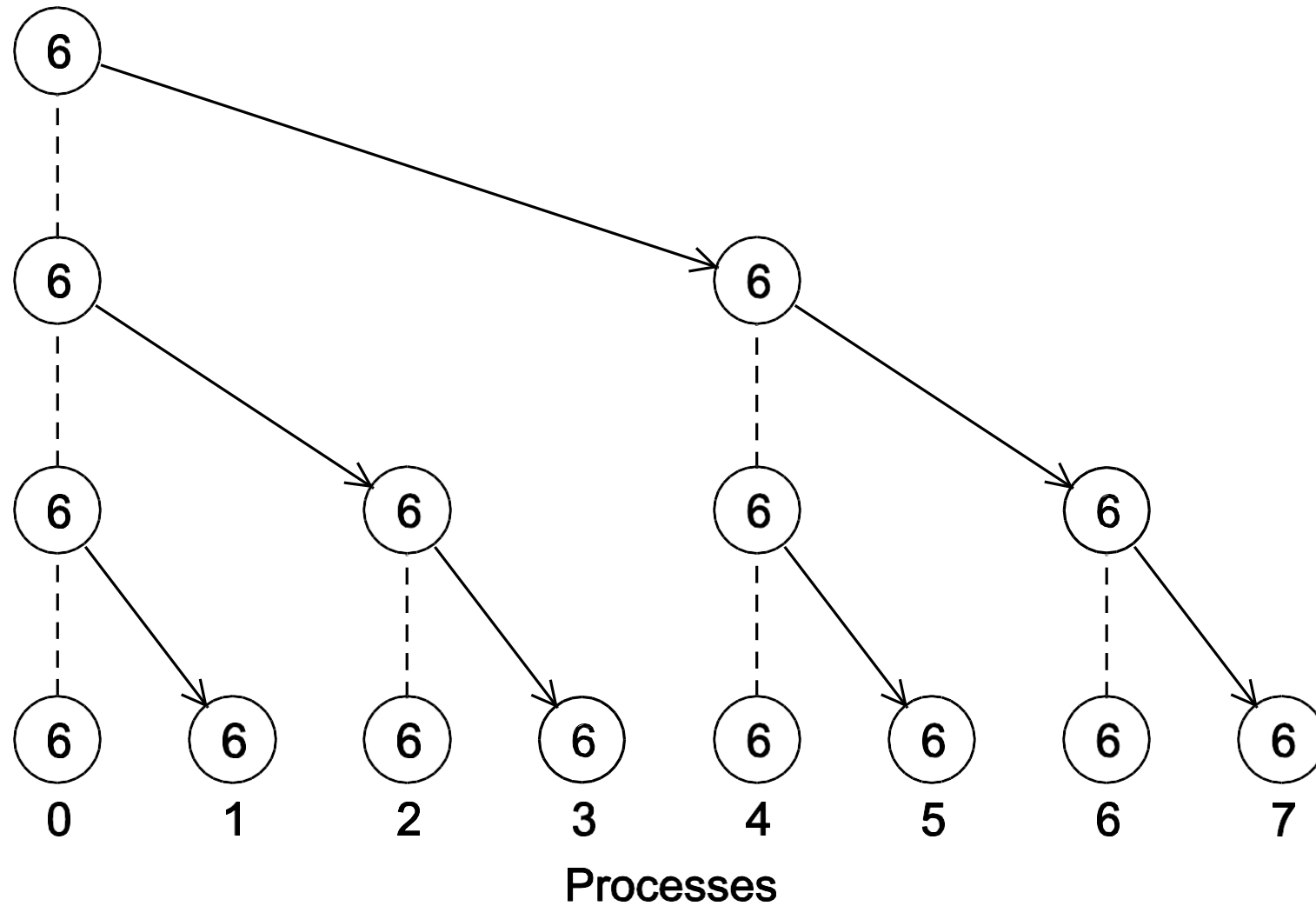


A butterfly-structured global sum.

Broadcast

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

A tree-structured broadcast.



$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \dots, x_{n-1}) + (y_0, y_1, \dots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \dots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

Compute a vector sum.

Different partitions of a 12-component vector among 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

Partitioning options

Block partitioning

Assign blocks of consecutive components to each process.

Cyclic partitioning

Assign components in a round robin fashion.

Block-cyclic partitioning

Use a cyclic distribution of blocks of components.

Scatter

```
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 */);
```

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_Gather(  
    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        dest_proc     /* in */,  
    MPI_Comm    comm         /* in */);
```

Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

Matrix-vector multiplication

a_{00}	a_{01}	$\cdots$	$a_{0,n-1}$
a_{10}	a_{11}	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
a_{i0}	a_{i1}	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

x_0
x_1
$\vdots$
x_{n-1}

 $=$

y_0
y_1
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
y_{m-1}

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11



Mpi derived datatypes

```
int MPI_Type_create_struct(  
    int          count          /* in */,  
    int          array_of_blocklengths[] /* in */,  
    MPI_Aint     array_of_displacements[] /* in */,  
    MPI_Datatype array_of_types[] /* in */,  
    MPI_Datatype* new_type_p     /* out */);
```



Performance evaluation

Elapsed parallel time

Elapsed serial time

MPI_Barrier

```
int MPI_Barrier(MPI_Comm comm /* in */);
```



Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

Speedup

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)}$$

Efficiency

A parallel sorting algorithm

Serial odd-even transposition sort

Parallel odd-even transposition sort

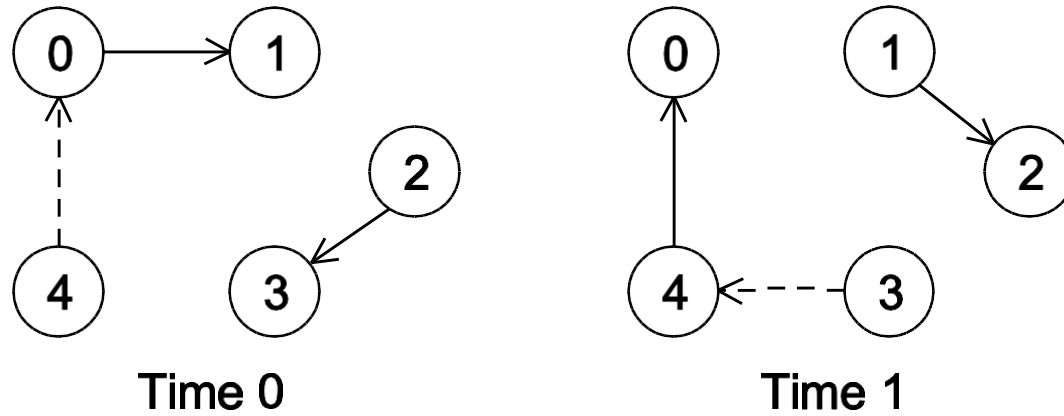
Time	Process			
	0	1	2	3
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

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

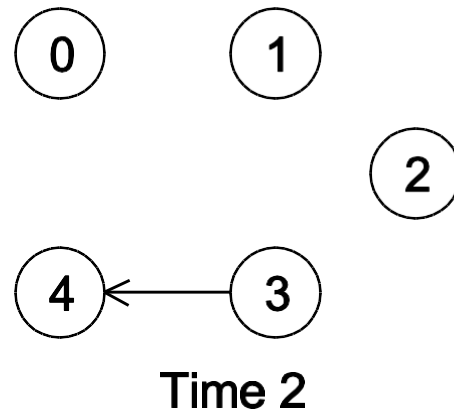
MPI_Ssend

```
int MPI_Sendrecv(  
    void*      send_buf_p      /* in */,  
    int        send_buf_size   /* in */,  
    MPI_Datatype send_buf_type /* in */,  
    int        dest            /* in */,  
    int        send_tag        /* in */,  
    void*      recv_buf_p      /* out */,  
    int        recv_buf_size   /* in */,  
    MPI_Datatype recv_buf_type /* in */,  
    int        source          /* in */,  
    int        recv_tag        /* in */,  
    MPI_Comm   communicator    /* in */,  
    MPI_Status* status_p       /* in */);
```

MPI_Sendrecv



Safe communication with five processes





A T M E
College of Engineering



Thank you