



# Collective Communication in MPI and Advanced Features

---

Pacheco's book. Chapter 3

T. Yang, CS240A 2016

Part of slides from the text book, CS267 K. Yelick from UC Berkeley and B. Gropp, ANL

# Outline

---

- **Collective group communication**
- **Application examples**
  - Pi computation
  - Summation of long vectors
- **More applications**
  - Matrix-vector multiplication
    - performance evaluation
  - Parallel sorting
- **Safety and other MPI issues.**

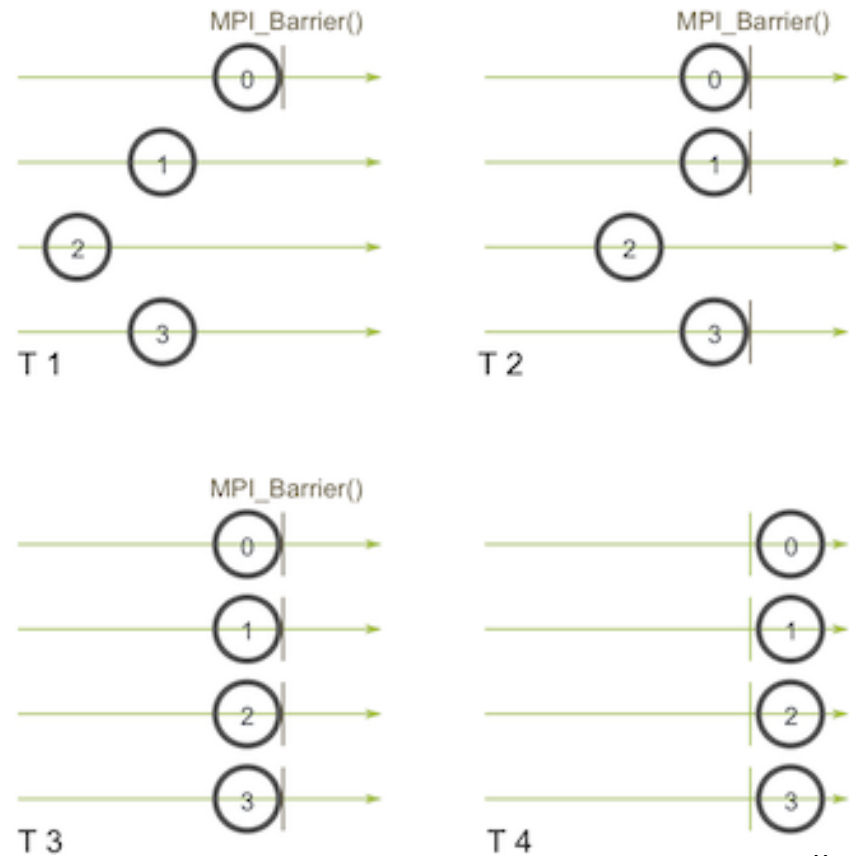
# MPI Collective Communication

---

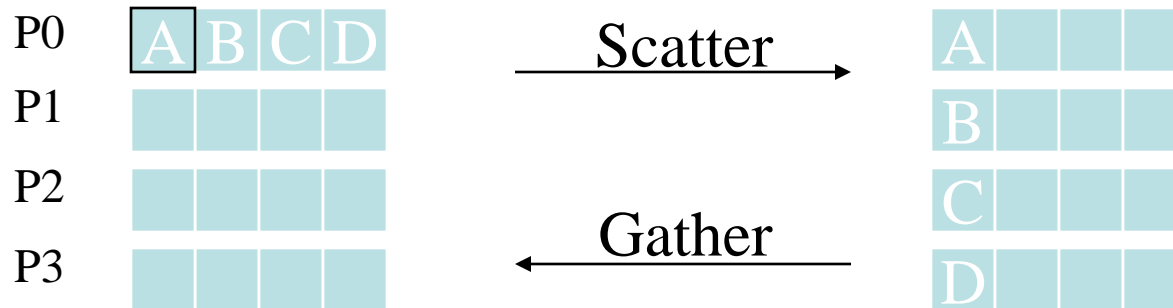
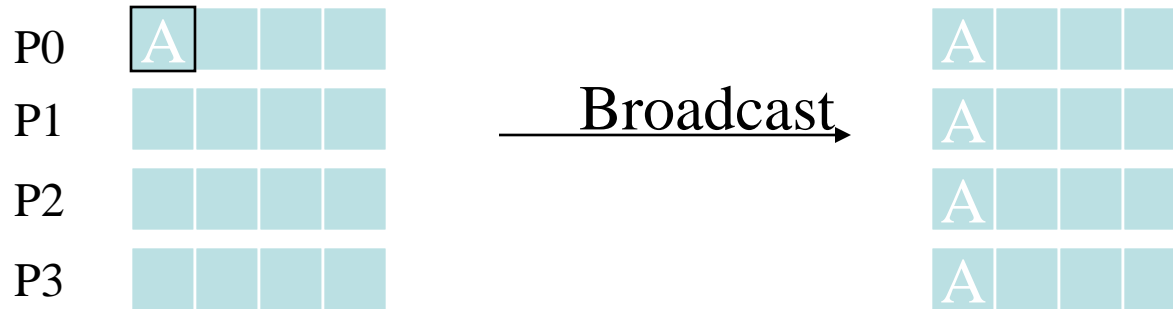
- **Collective routines provide a higher-level way to organize a parallel program**
  - Each process executes the same communication operations
  - Communication and computation is coordinated among a group of processes in a communicator
  - Tags are not used
  - No non-blocking collective operations.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

# Synchronization

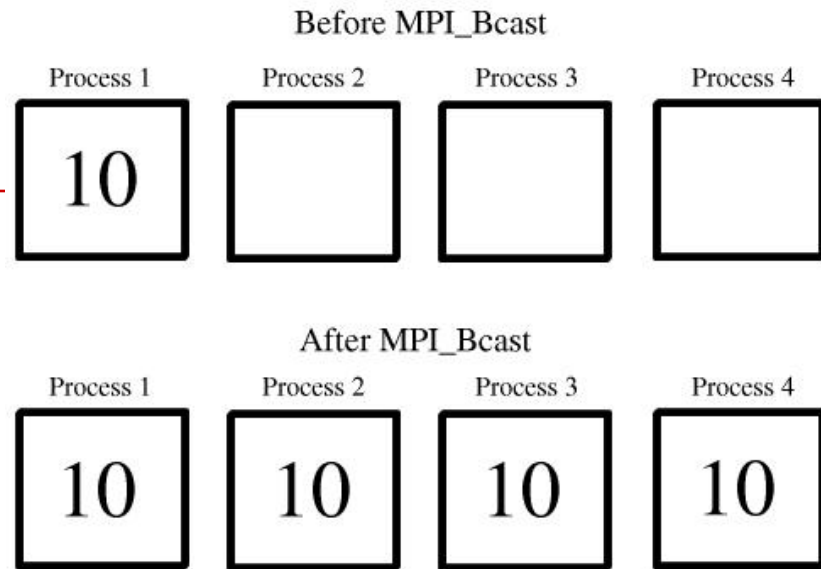
- `MPI_Barrier( comm )`
- Blocks until all processes in the group of the communicator `comm` call it.
- Not used often. Sometime used in measuring performance and load balancing



# Collective Data Movement: Broadcast, Scatter, and Gather



# Broadcast

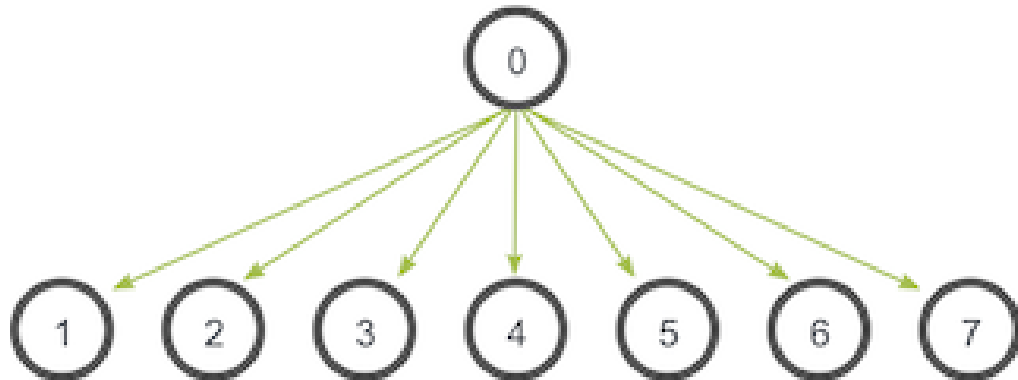


- **Data belonging to a single process is sent to all of the processes in the communicator.**

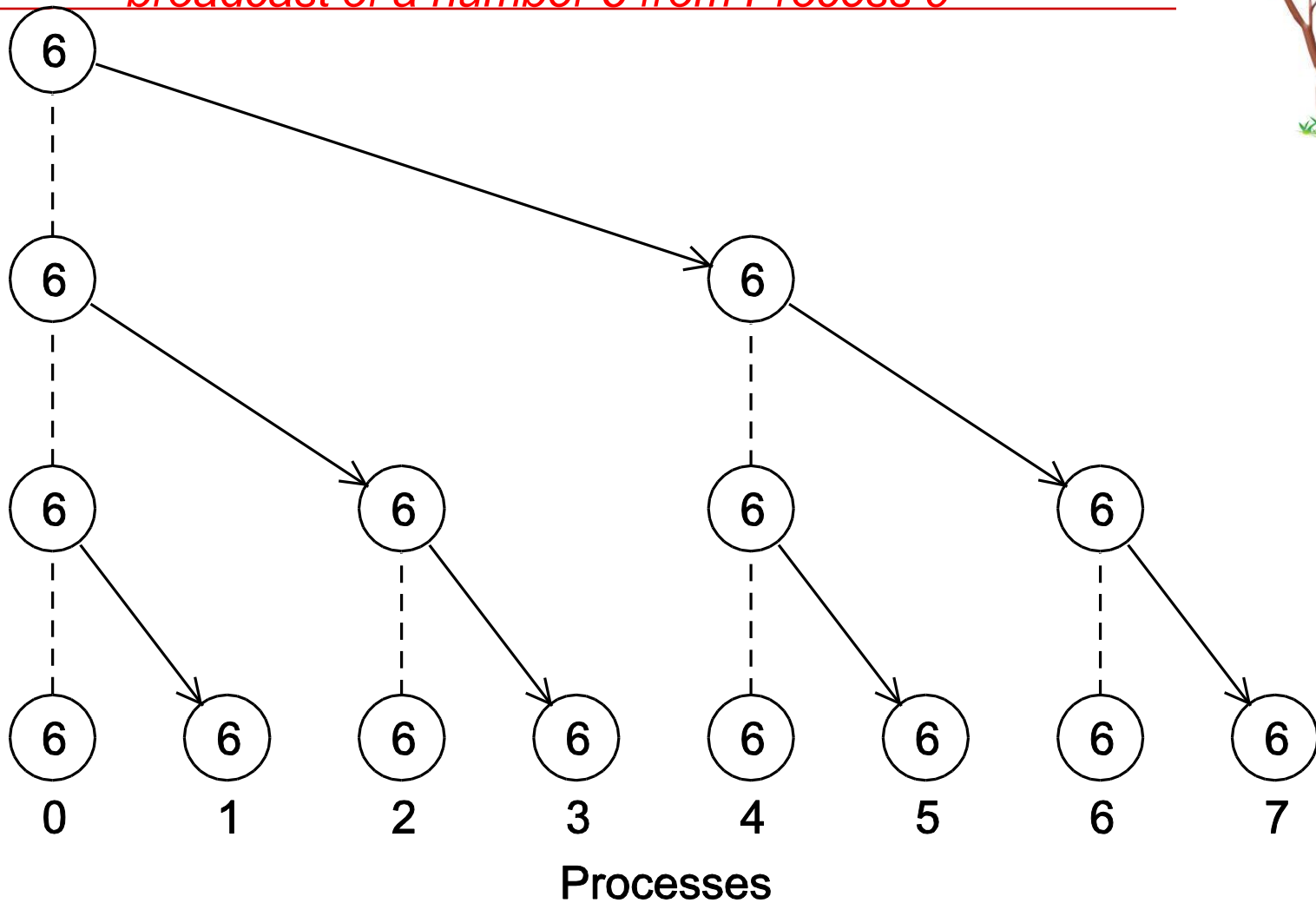
```
int MPI_Bcast(  
    void* data_p          /* in/out */,  
    int count            /* in     */,  
    MPI_Datatype datatype /* in     */,  
    int source_proc      /* in     */,  
    MPI_Comm comm        /* in     */);
```

# Comments on Broadcast

- All collective operations must be called by *all* processes in the communicator
- MPI\_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI\_Bcast is not a “multi-send”
  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive



*Implementation View: A tree-structured broadcast of a number 6 from Process 0*



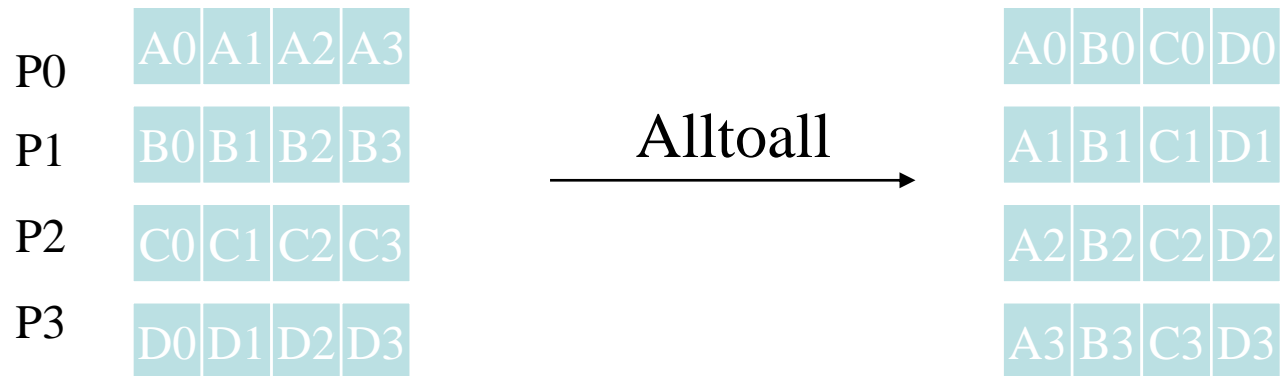
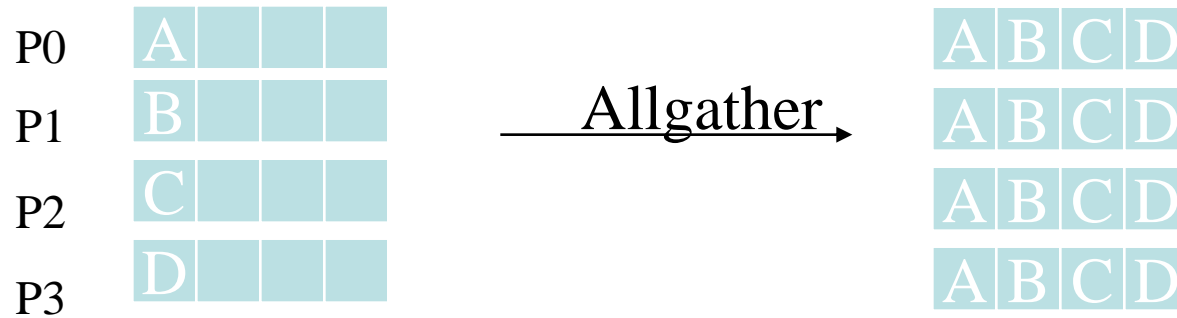


# A version of `Get_input` that uses `MPI_Bcast` in the trapezoidal program

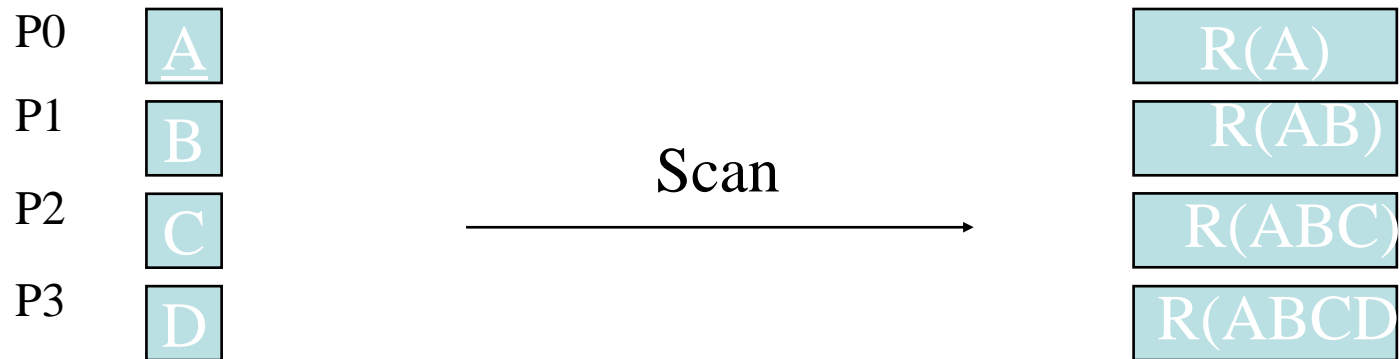
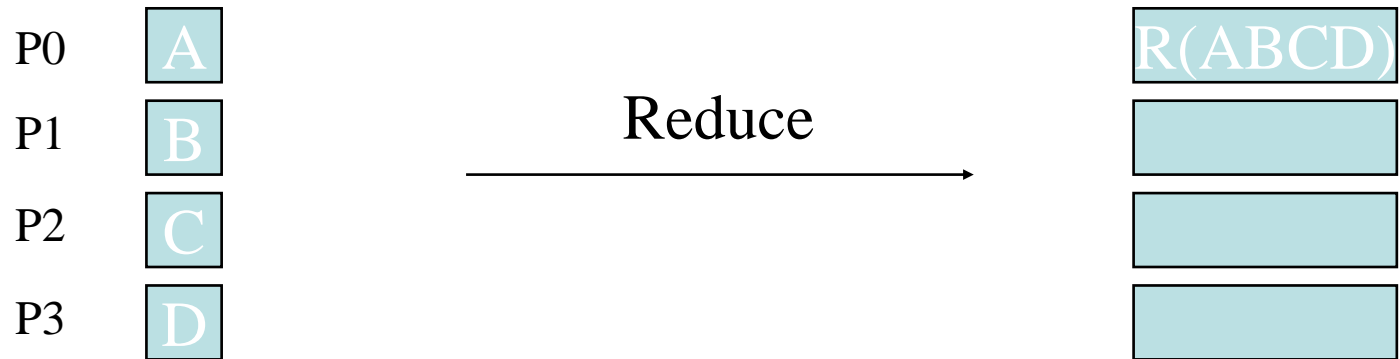
```
void Get_input(
    int      my_rank    /* in */,
    int      comm_sz    /* in */,
    double*  a_p        /* out */,
    double*  b_p        /* out */,
    int*     n_p        /* out */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_input */
```

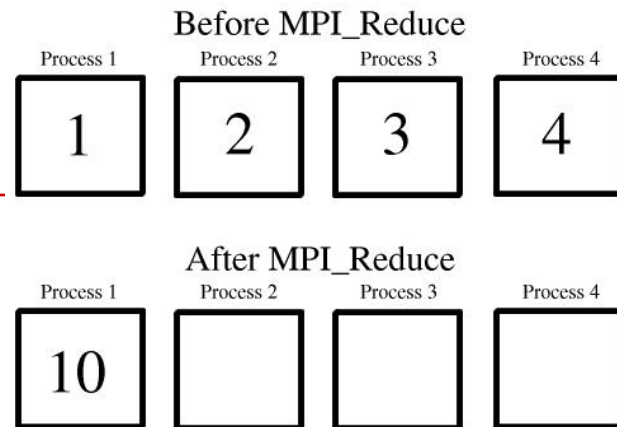
# Collective Data Movement: Allgather and AlltoAll



# Collective Computation: Reduce vs. Scan



# 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 */);
```

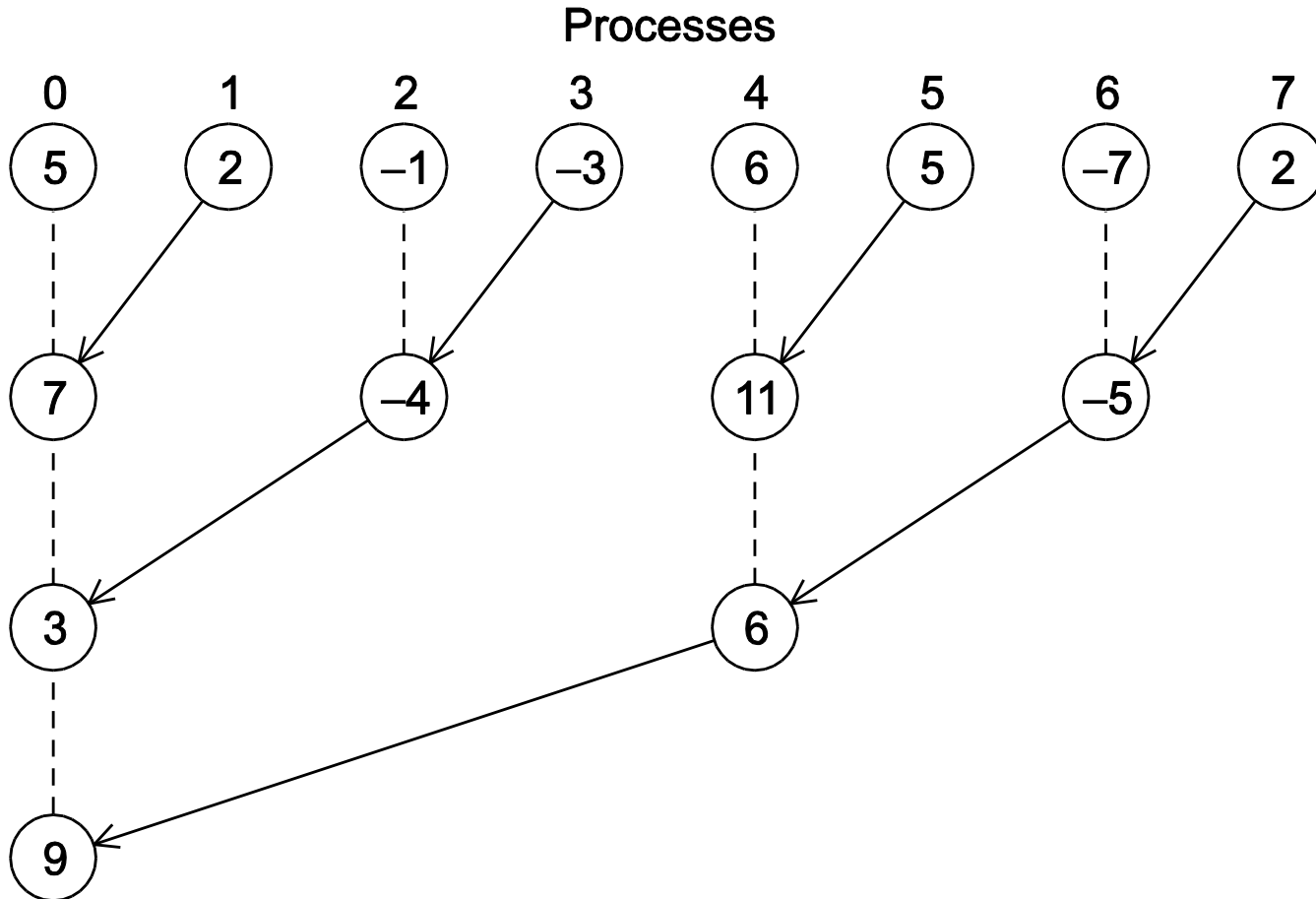
```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
          MPI_COMM_WORLD);
```

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

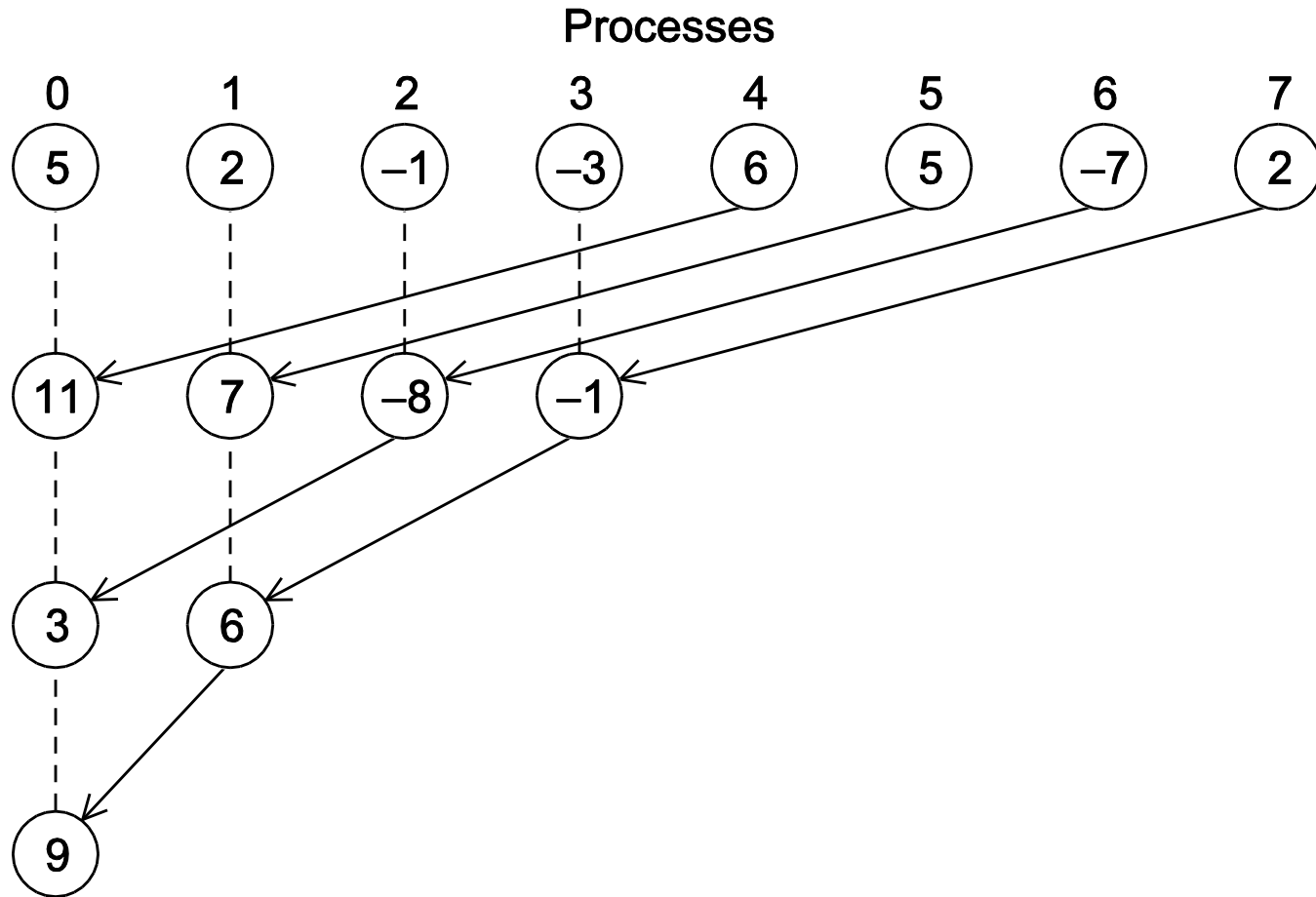
# Predefined reduction operators in MPI

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

# Implementation View of Global Reduction using a tree-structured sum

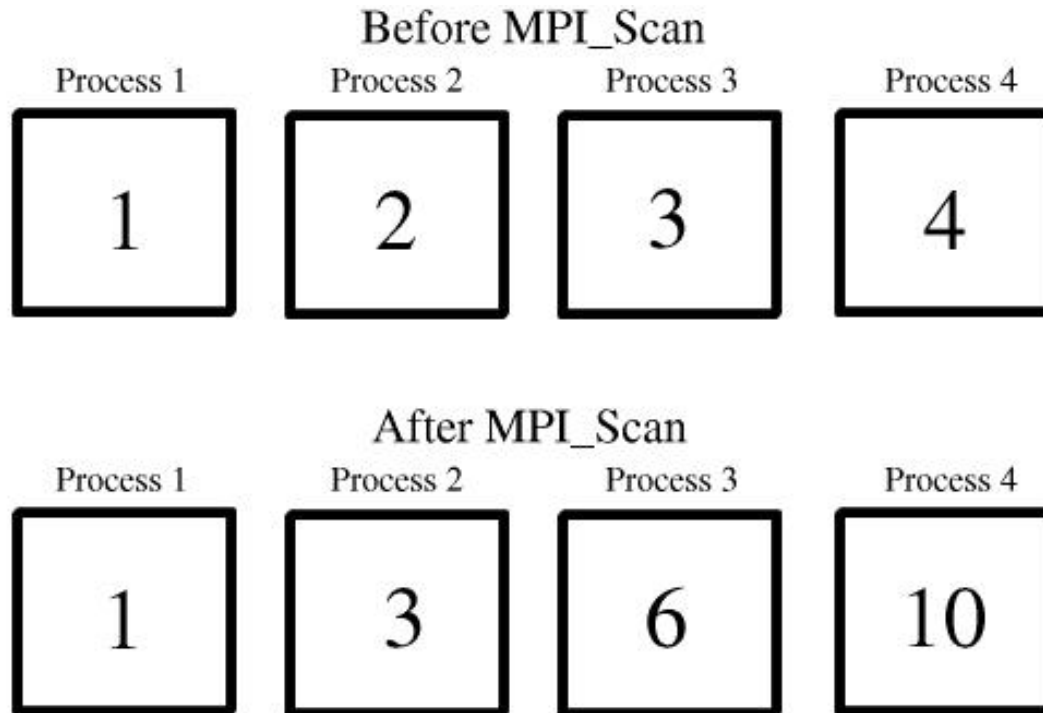


# An alternative tree-structured global sum



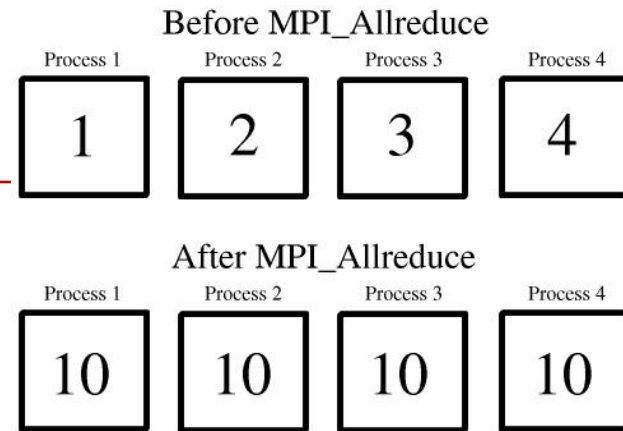
# MPI Scan

```
MPI_Scan( void *sendbuf, void *recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm );
```



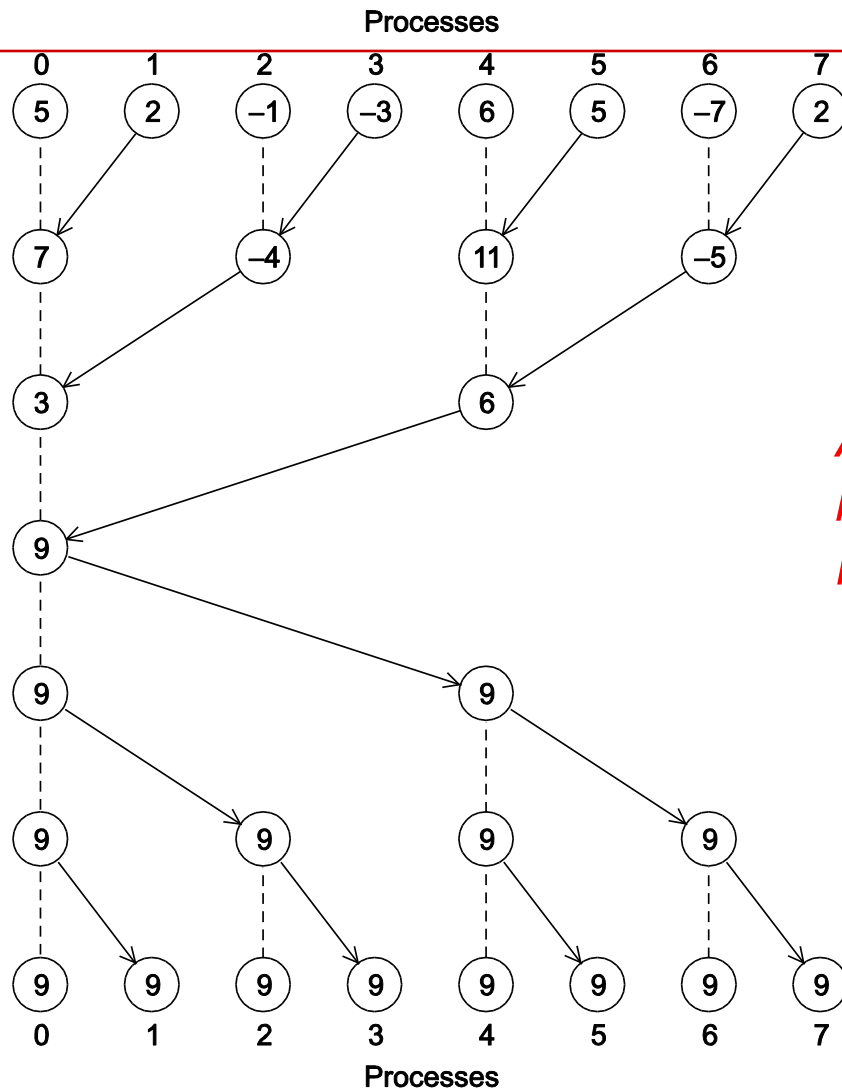


# MPI\_Allreduce



- **Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.**

```
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 */);
```



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

# MPI Collective Routines: Summary

---

- **Many Routines:** `Allgather`, `Allgatherv`, `Allreduce`, `Alltoall`, `Alltoallv`, `Bcast`, `Gather`, `Gatherv`, `Reduce`, `Reduce_scatter`, `Scan`, `Scatter`, `Scatterv`
- **All versions deliver results to all participating processes.**
- **V versions allow the hunks to have variable sizes.**
- **Allreduce, Reduce, Reduce\_scatter, and Scan take both built-in and user-defined combiner functions.**
- **MPI-2 adds Alltoallw, Exscan, intercommunicator versions of most routines**

# Example of MPI PI program using 6 Functions

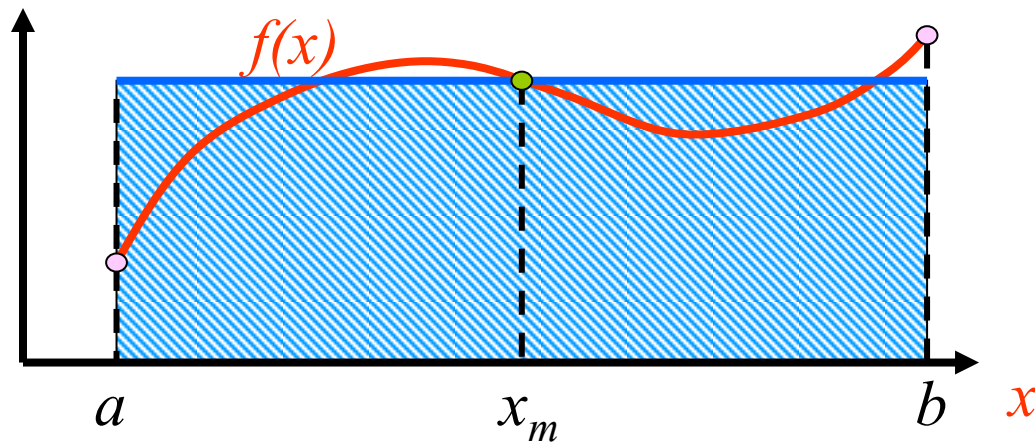
$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

- Using basic MPI functions:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
- Using MPI collectives:
  - `MPI_BCAST`
  - `MPI_REDUCE`

# Midpoint Rule for

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

$$\int_a^b f(x) dx \approx (b-a) f(x_m)$$



$$\int_{x=0}^1 \frac{1}{1+x^2} \approx \sum_{i=1}^n \frac{1}{1 + \left(\frac{i-0.5}{n}\right)^2}$$

# Example: PI in C - 1

```
#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 = 3.141592653589793238462643;
    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);
```

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

```
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

## Example: PI in C - 2

$$\int_{x=0}^1 \frac{1}{1+x^2} \approx \sum_{i=1}^n \frac{1}{1 + \left(\frac{i-0.5}{n}\right)^2}$$

```
h    = 1.0 / (double) n;      Compute local pi values
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;
```

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

# Collective vs. Point-to-Point Communications

---

- **All the processes in the communicator must call the same collective function.**
  - Will this program work?

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



# 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.

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

# Collective vs. Point-to-Point Communications

- The arguments passed by each process to an MPI collective communication must be “compatible.”
  - Will this program work?

```
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);
```

# 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 `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

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, ...)

Multiple calls to MPI\_Reduce with MPI\_SUM and Proc 0 as destination (root)

Is b=3 on Proc 0 after two MPI\_Reduce() calls?

Is d=6 on Proc 0?

## Example: Output results

---

- However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`.
- The order of the calls will determine the matching so the value stored in b will be  $1+2+1 = 4$ , and the value stored in d will be  $2+1+2 = 5$ .

# Parallel Matrix Vector Multiplication

---

Collective Communication Application

Textbook p. 113-116

## Matrix-vector multiplication: $y = A * x$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 * 1 + 2 * 2 + 3 * 3 \\ 4 * 1 + 5 * 2 + 6 * 3 \\ 7 * 1 + 8 * 2 + 9 * 3 \end{pmatrix} = \begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix}$$

**Problem:**  $y = A * x$  where  $A$  is a  $n \times n$  matrix and  $x$  is a column vector of dimension  $n$ .

**Sequential code:**

```
for  $i = 1$  to  $n$  do
   $y_i = 0$ ;
  for  $j = 1$  to  $n$  do
     $y_i = y_i + a_{i,j} * x_j$ ;
  endfor
endfor
```

# Partitioning and Task graph for matrix-vector multiplication

Partitioned code:

```
for  $i = 1$  to  $n$  do
   $S_i$  :  $y_i = 0$ ;
    for  $j = 1$  to  $n$  do
       $y_i = y_i + a_{i,j} * x_j$ ;
    endfor
endfor
```

$S_i$  : Read row  $A_i$  and vector  $x$ .

Write element  $y_i$

$$y_i = \text{Row } A_i * x$$

Task graph:

(S1)

(S2)

(S3)

(Sn)



# Execution Schedule and Task Mapping

$S_i$  : Read row  $A_i$  and vector  $x$ .

Write element  $y_i$

$$y_i = \text{Row } A_i * x$$

Task graph:



Schedule:

0	1		p-1
S1	Sr+1		
S2	Sr+2		
Sr	S2r		Sn

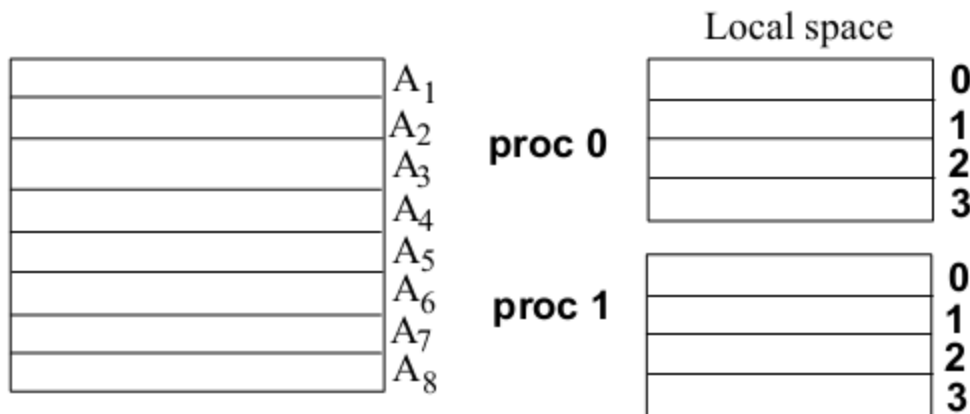
Mapping function of tasks  $S_i$ :

$$proc\_map(i) = \lfloor \frac{i-1}{r} \rfloor \text{ where } r = \lceil \frac{n}{p} \rceil.$$

# Data Partitioning and Mapping for $y = A \cdot x$

**Data partitioning:** for the above schedule:

Matrix  $A$  is divided into  $n$  rows  $A_1, A_2, \dots, A_n$ .



**Data mapping:**

Row  $A_i$  is mapped to processor  $proc\_map(i)$ , the same as task  $i$ . The indexing function is:

$local(i) = (i - 1) \bmod r$ . Vectors  $x$  and  $y$  are replicated to all processors.

# SPMD Code for $y = A * x$

Schedule:

0	1		p-1
S1	Sr+1		
S2	Sr+2		
Sr	S2r		Sn

```
int x[n], y[n], a[r][n];
```

```
me=mynode();
```

```
for  $i = 1$  to  $n$  do
```

```
  if  $proc\_map(i) == me$ , then do  $S_i$ :
```

```
     $S_i$  :  $y[i] = 0$ ;
```

```
      for  $j = 1$  to  $n$  do
```

```
         $y[i] = y[i] + a[local(i)][j] * x[j]$ ;
```

```
      endfor
```

```
endfor
```

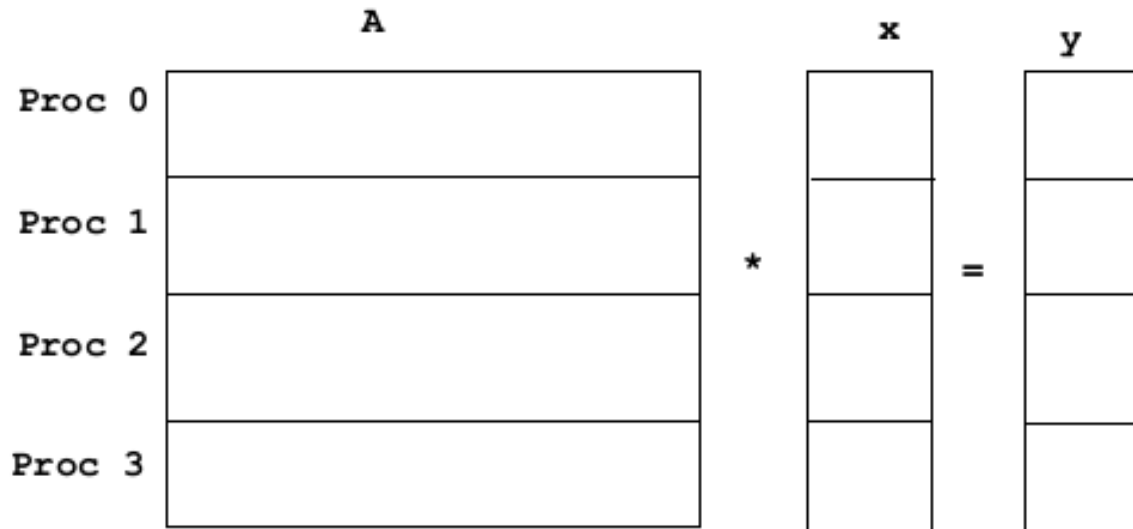
## Evaluation: Parallel Time

---

- Ignore the cost of local address calculation.
- Each task performs  $n$  additions and  $n$  multiplications.
- Each addition/multiplication costs  $\omega$
- The parallel time is approximately  $\frac{n}{p} \times 2n\omega$

# How is initial data distributed?

Assume initially matrix  $A$  and vector  $x$  are distributed evenly among processes

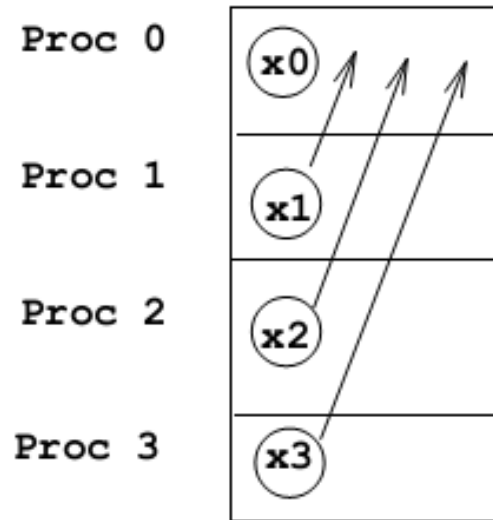


Need to redistribute vector  $x$  to everybody in order to perform parallel computation!

What MPI collective communication is needed?

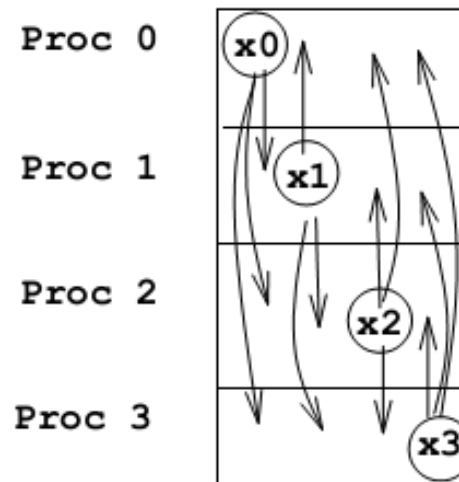
# Communication Pattern for Data Redistribution

Data requirement for  
Process 0



MPI\_Gather

Data requirement for  
all processes



MPI\_Allgather

# MPI Code for Gathering Data

---

Data gather for  
Process 0

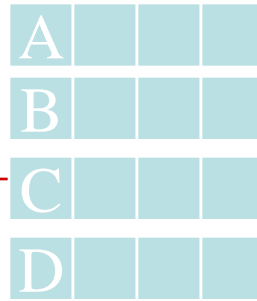
```
float local_x[]; /*local storage for x*/  
float global_x[]; /*storage for all of x*/  
  
MPI_Gather(local_x, n/p, MPI_FLOAT,  
          global_x, n/p, MPI_FLOAT,  
          0, MPI_COMM_WORLD);
```

Repeat for all processes

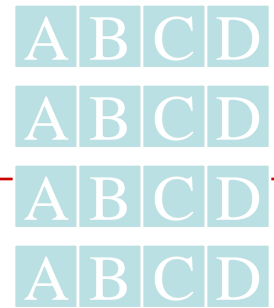
It is the same as:

```
MPI_All_gather(local_x, n/p, MPI_FLOAT,  
              global_x, n/p, MPI_FLOAT,  
              MPI_COMM_WORLD);
```

# Allgather



→ Allgather →



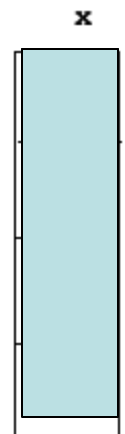
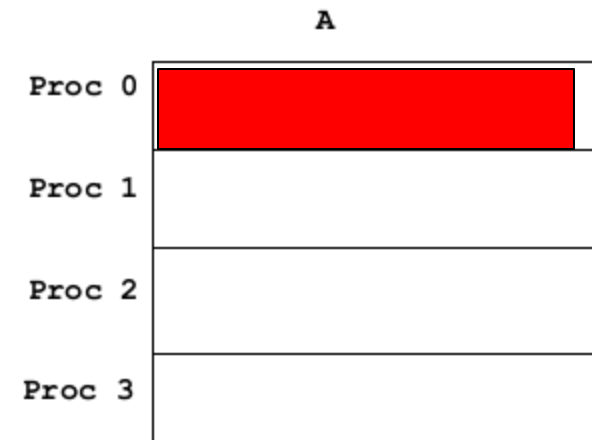
- Concatenates the contents of each process' `send_buf_p` and stores this in each process' `recv_buf_p`.
- As usual, `recv_count` is the amount of data being received from each process.

```
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 */);
```



# MPI SPMD Code for $y=A*x$

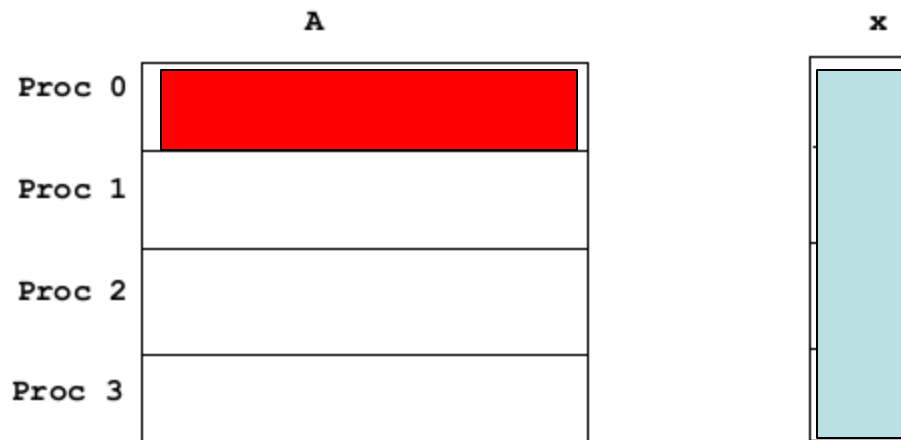
```
void Parallel_matrix_vector_prod(
    LOCAL_MATRIX_T local_A
    int m
    int n
    float local_x[]
    float global_x[]
    float local_y[]
    int local_m
    int local_n) {
    /* local_m = n/p, local_n = n/p */
    MPI_Allgather(local_x, local_n, MPI_FLOAT
                  global_x, local_n, MPI_FLOAT,
                  MPI_COMM_WORLD);
```



# MPI SPMD Code for $y=A*x$

```
for (i = 0; i < local_m; i++) {  
    local_y[i] = 0.0;  
    for (j = 0; j < n; j++)  
        local_y[i] = local_y[i] +  
            local_A[i][j]*global_x[j];  
}
```

```
}
```



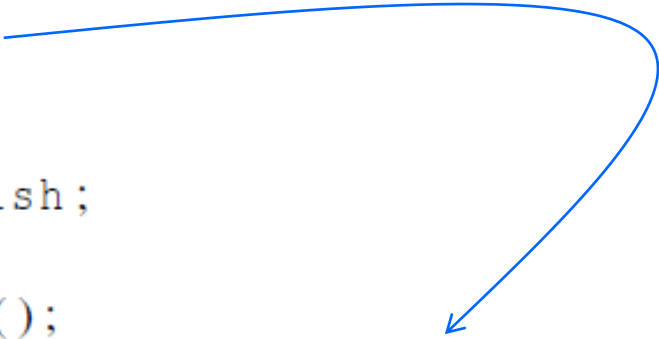


# Performance Evaluation of Matrix Vector Multiplication

# How to measure elapsed parallel time

- Use `MPI_Wtime()` that returns the number of seconds that have elapsed since some time in the past.

```
double MPI_Wtime(void);  
  
double start, finish;  
...  
start = MPI_Wtime();  
/* Code to be timed */  
...  
finish = MPI_Wtime();  
printf("Proc %d > Elapsed time = %e seconds\n"  
       my_rank, finish-start);
```



# Measure elapsed sequential time in Linux



- This code works for Linux without using MPI functions
- Use `GET_TIME()` which returns time in microseconds elapsed from some point in the past.

- Sample code for `GET_TIME()`

```
#include <sys/time.h>
```

```
/* The argument now should be a double (not a pointer to a  
double) */
```

```
#define GET_TIME(now) {  
    struct timeval t;  
    gettimeofday(&t, NULL);  
    now = t.tv_sec + t.tv_usec/1000000.0;  
}
```

# Measure elapsed sequential time

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

# Use MPI\_Barrier() before time measurement

Start timing until every process in the communicator has reached the same time stamp

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .

local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
          MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
```

# 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

*(Seconds)*



# Speedup and Efficiency

---

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

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

# Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

# Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

# Scalability



- A program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.
- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**.
- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**.
-

# **Safety Issues in MPI programs**

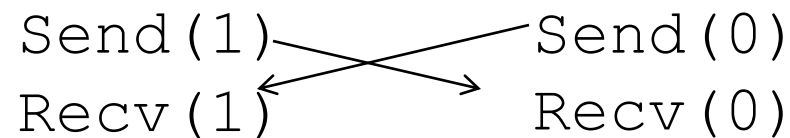
---

# Safety in MPI programs

- Is it a safe program? (Assume tag/process ID is assigned properly)

Process 0

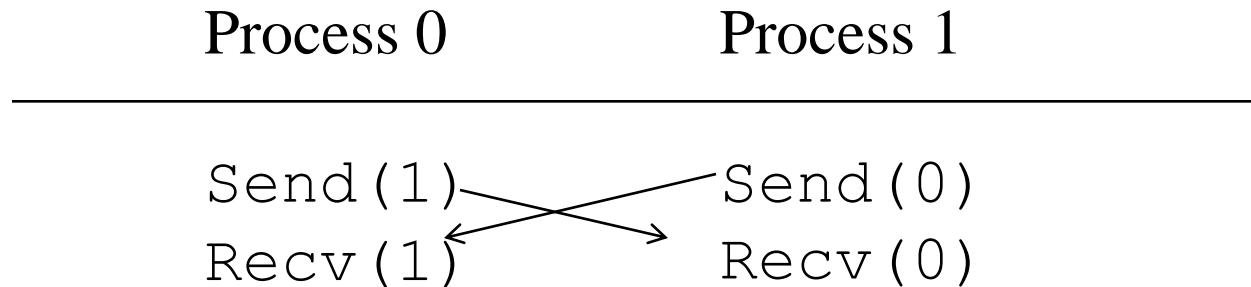
Process 1



```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
0, comm, MPI_STATUS_IGNORE);
```

# Safety in MPI programs

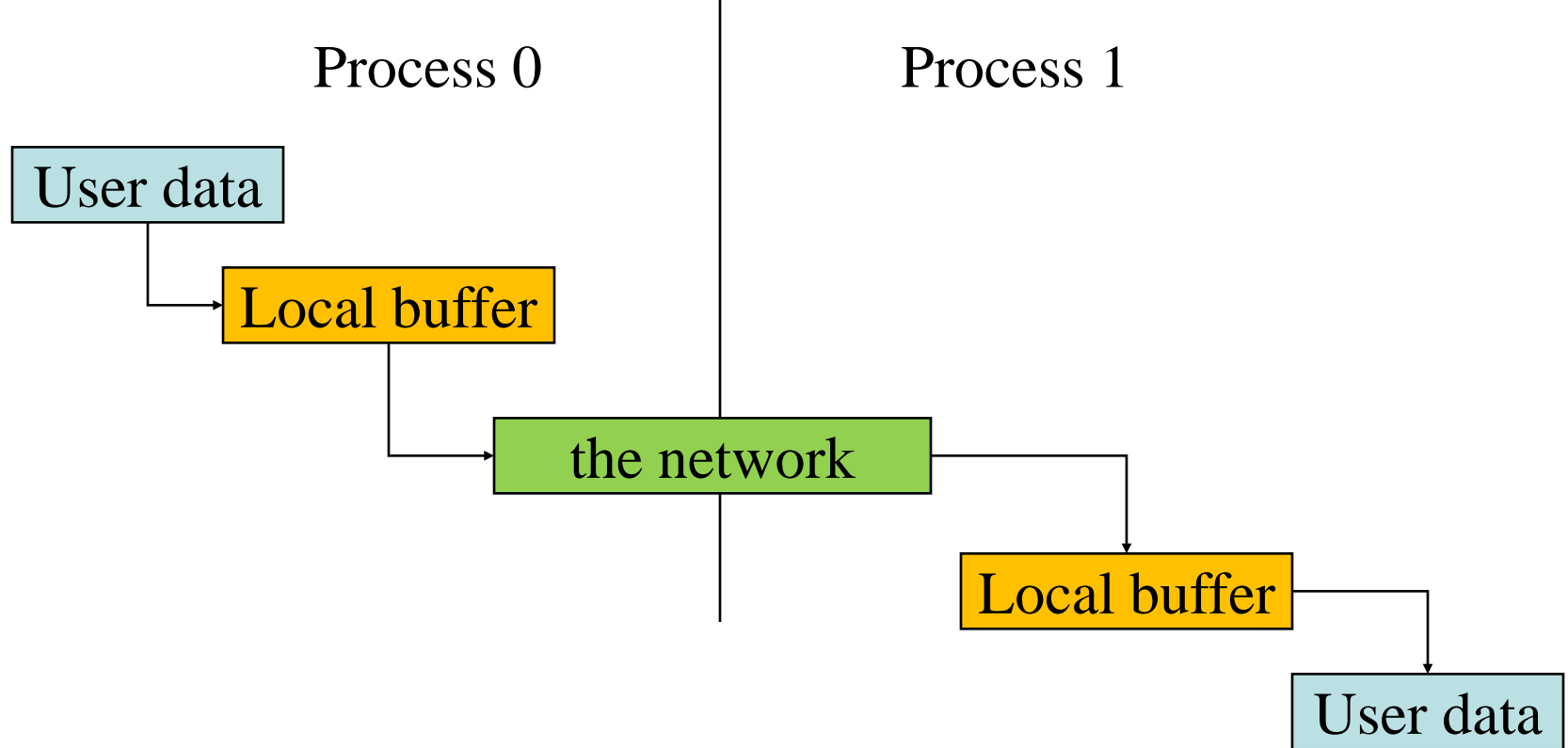
- **Is it a safe program? (Assume tag/process ID is assigned properly)**



- **May be unsafe because MPI standard allows MPI\_Send to behave in two different ways:**
  - it can simply copy the message into an MPI managed buffer and return,
  - or it can block until the matching call to MPI\_Recv starts.

# Buffer a message implicitly during MPI\_Send()

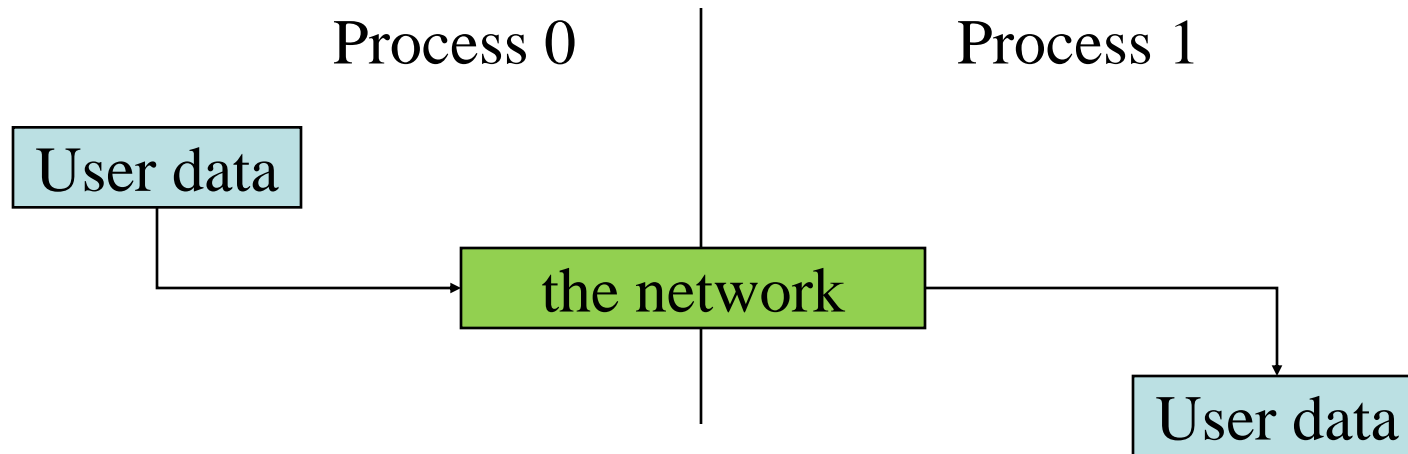
- When you send data, where does it go? One possibility is:





# Avoiding Buffering

- **Avoiding copies uses less memory**
- **May use more time**



`MPI_Send()` waits until a matching receive is executed.

# Safety in MPI programs

---

- Many implementations of MPI set a threshold at which the system switches from buffering to blocking.
  - Relatively small messages will be buffered by MPI\_Send.
  - Larger messages, will cause it to block.



**If the MPI\_Send() executed by each process blocks, no process will be able to start executing a call to MPI\_Recv, and the program will hang or **deadlock**.**

- Each process is blocked waiting for an event that will never happen.

## Example of unsafe MPI code with possible deadlocks

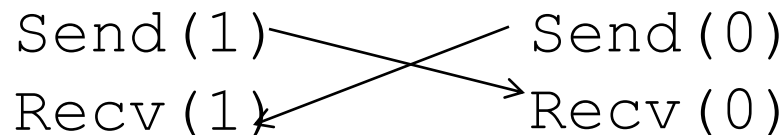
- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)

Process 0

Process 1

---

Send (1)      Send (0)  
Recv (1)      Recv (0)



- This may be “unsafe” because it depends on the availability of system buffers in which to store the data sent until it can be received

# Safety in MPI programs

---

- A program that relies on MPI provided buffering is said to be **unsafe**.
- Such a program may run without problems for various sets of input, but it may hang or crash with other sets.

# How can we tell if a program is unsafe

- Replace MPI\_Send() with MPI\_Ssend()
- The extra “s” stands for synchronous and MPI\_Ssend is guaranteed to block until the matching receive starts.
- **If the new program does not hang/crash, the original program is safe.**
- MPI\_Send() and MPI\_Ssend() have the same arguments

```
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 */);
```

## Some Solutions to the “unsafe” Problem

---

- Order the operations more carefully:

Process 0

Process 1

---

Send (1)

Recv (0)

Recv (1)

Send (0)

- Simultaneous send and receive in one call

Process 0

Process 1

---

Sendrecv (1)

Sendrecv (0)

# Restructuring communication in odd-even sort

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
         0, comm, MPI_STATUS_IGNORE.
```



```
if (my_rank % 2 == 0) {  
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
            0, comm, MPI_STATUS_IGNORE.  
} else {  
    MPI_Recv(new_msg, size, MPI_INT, (my_rank+comm_sz-1) % comm_sz,  
            0, comm, MPI_STATUS_IGNORE.  
    MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0, comm);  
}
```

# Use MPI\_Sendrecv()

to conduct a blocking send and a receive in a single call.

```
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 */);
```



## More Solutions to the “unsafe” Problem

- Supply own space as buffer for send

Process 0

Process 1

---

Bsend(1)

Bsend(0)

Recv(1)

Recv(0)

- Use non-blocking operations:

Process 0

Process 1

---

Isend(1)

Isend(0)

Irecv(1)

Irecv(0)

Waitall

Waitall

# Concluding Remarks (1)

---

- **MPI works in C, C++, or Fortran.**
- **A communicator is a collection of processes that can send messages to each other.**
- **Many parallel programs use the SPMD approach.**
- **Most serial programs are deterministic:** if we run the same program with the same input we'll get the same output.
  - Parallel programs often don't possess this property.
- **Collective communications involve all the processes in a communicator.**

## Concluding Remarks (2)

---

- **Performance evaluation**
  - Use elapsed time or “wall clock time”.
  - $\text{Speedup} = \text{sequential}/\text{parallel time}$
  - $\text{Efficiency} = \text{Speedup}/p$
  - If it's possible to increase the problem size ( $n$ ) so that the efficiency doesn't decrease as  $p$  is increased, a parallel program is said to be scalable.
- **An MPI program is unsafe if its correct behavior depends on the fact that MPI\_Send is buffering its input.**