



Collective Communication in MPI and Advanced Features

Pacheco. Chapter 3

T. Yang, CS140 2014

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

Outline

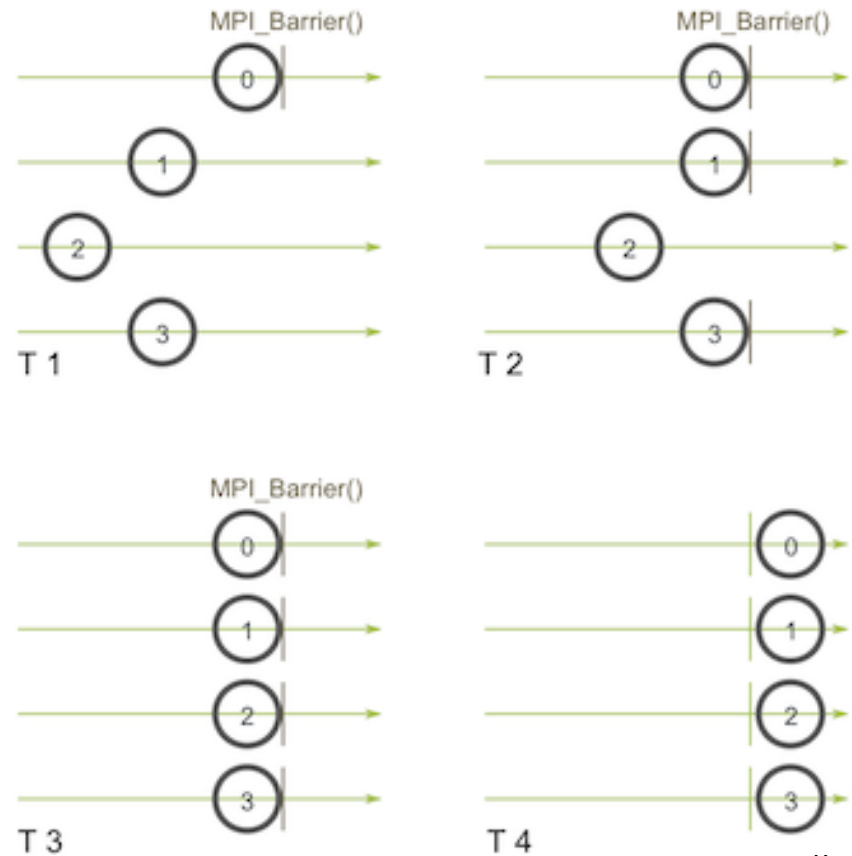
- **Collective group communication**
- **Examples**
- **Advanced features**

MPI Collective Communication

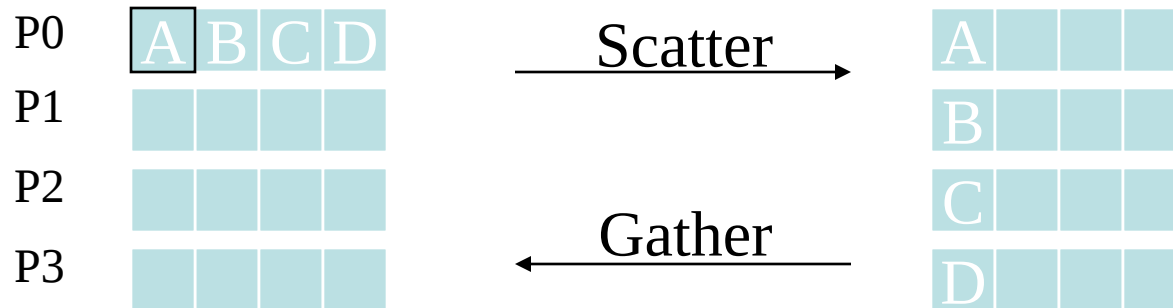
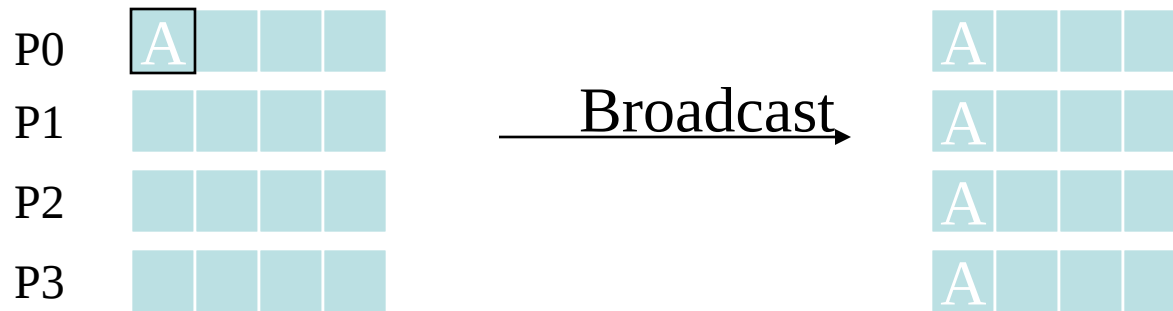
- ***Collective* routines provide a higher-level way to organize a parallel program**
 - Each process executes the same communication operations
 - Communication and computation is coordinated among a group of processes in a communicator
 - Tags are not used
 - No non-blocking collective operations.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

Synchronization

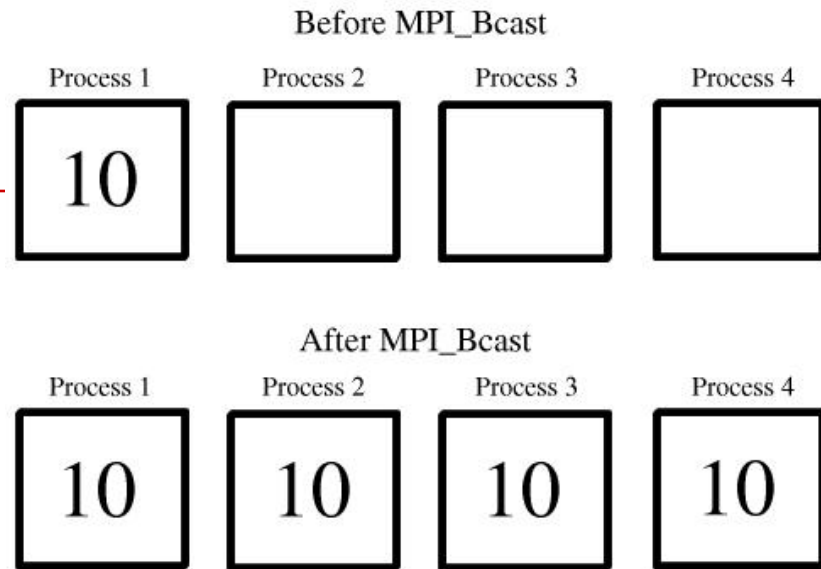
- `MPI_Barrier( comm )`
- Blocks until all processes in the group of the communicator `comm` call it.
- Not used often. Sometime used in measuring performance and load balancing



Collective Data Movement: Broadcast, Scatter, and Gather



Broadcast

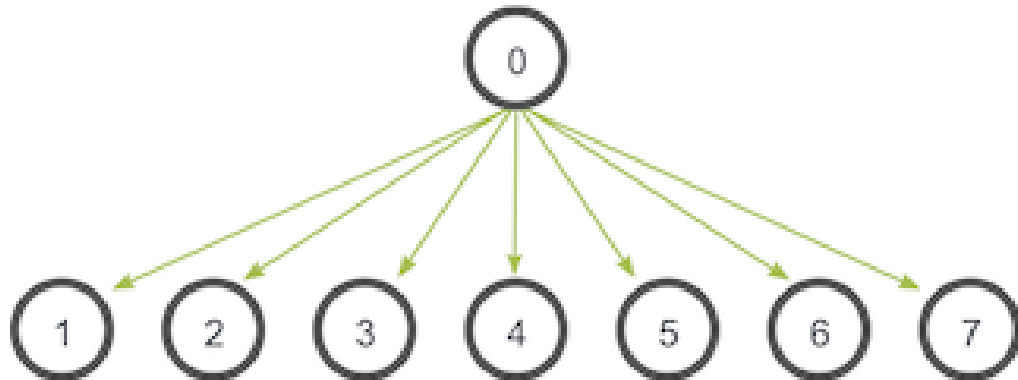


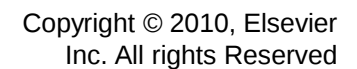
- Data belonging to a single process is sent to all of the processes in the communicator.

```
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

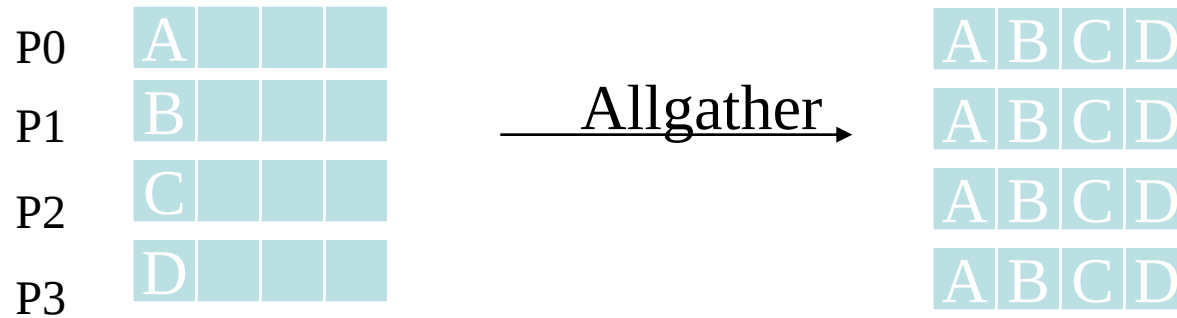




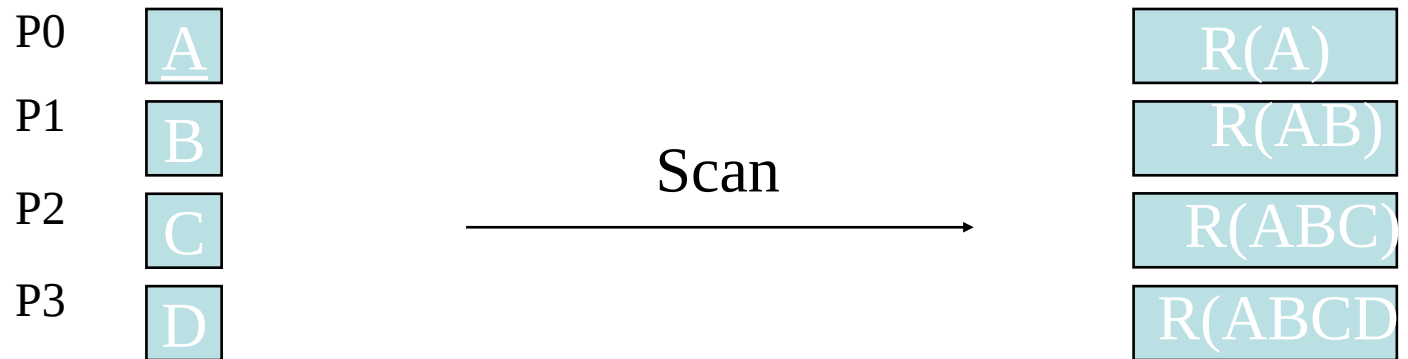
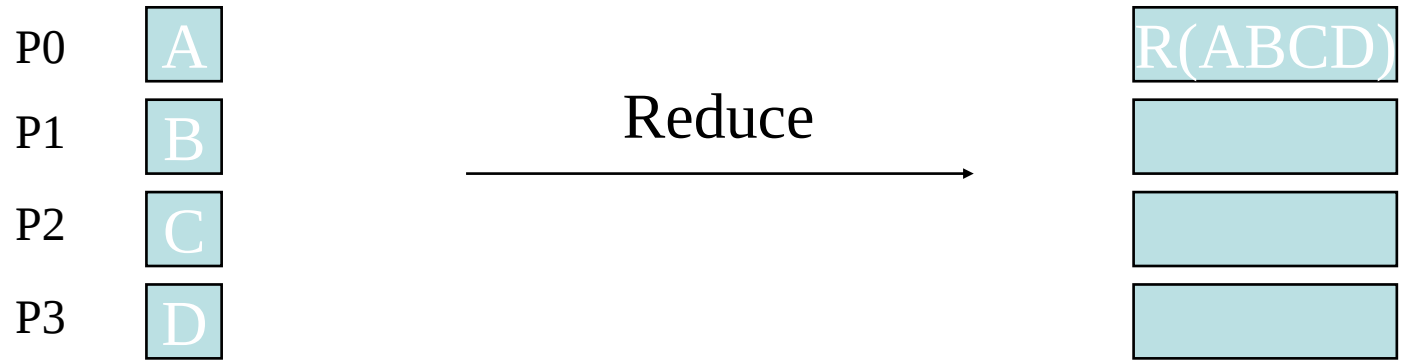
A version of Get_input that uses MPI_Bcast in the trapezoidal program

```
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 */
```

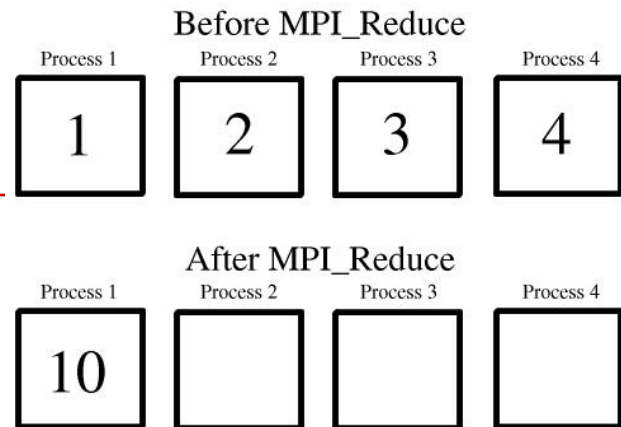
Collective Data Movement: Allgather and AlltoAll



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 */);
```

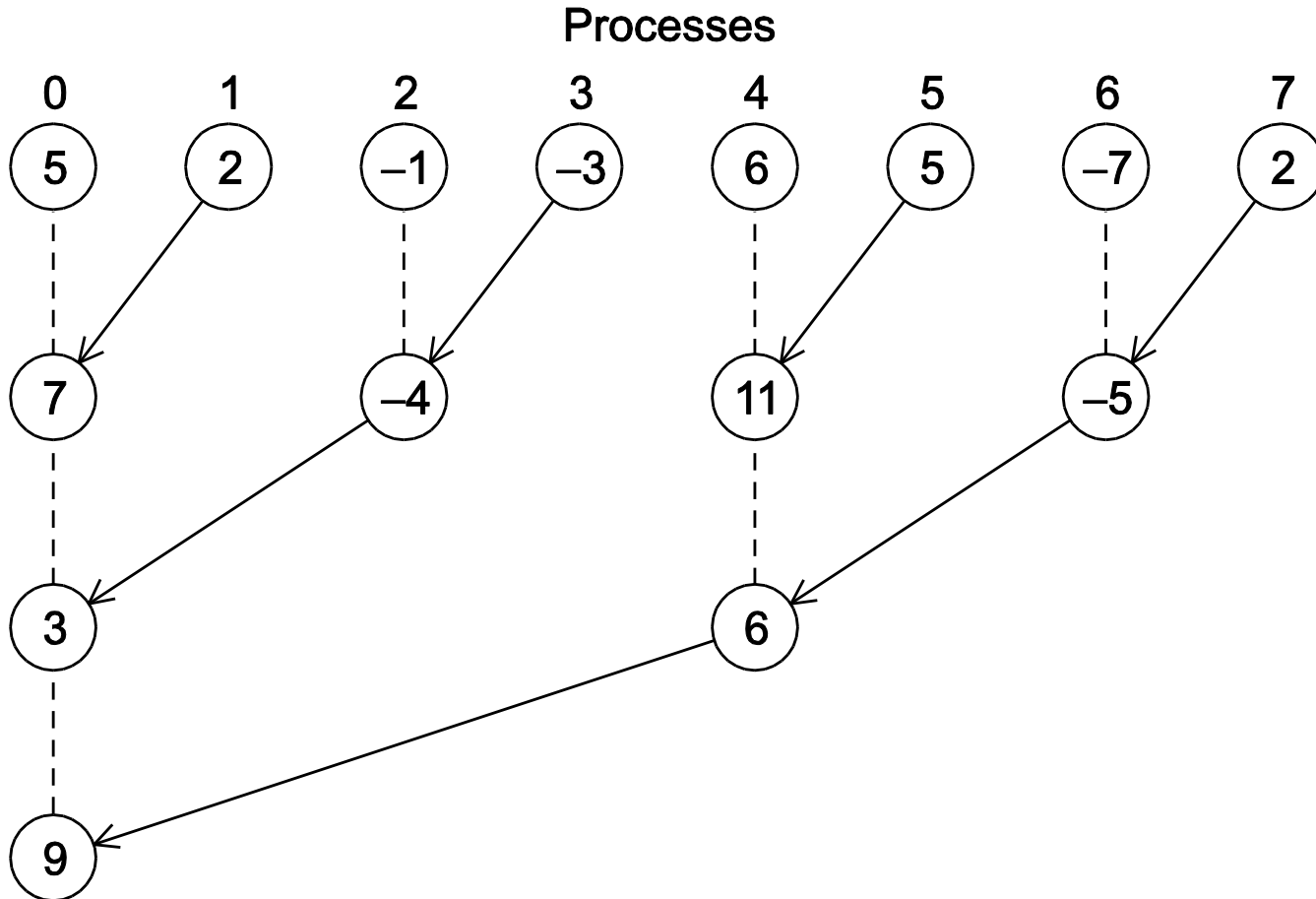
```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,  
MPI_COMM_WORLD);
```

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

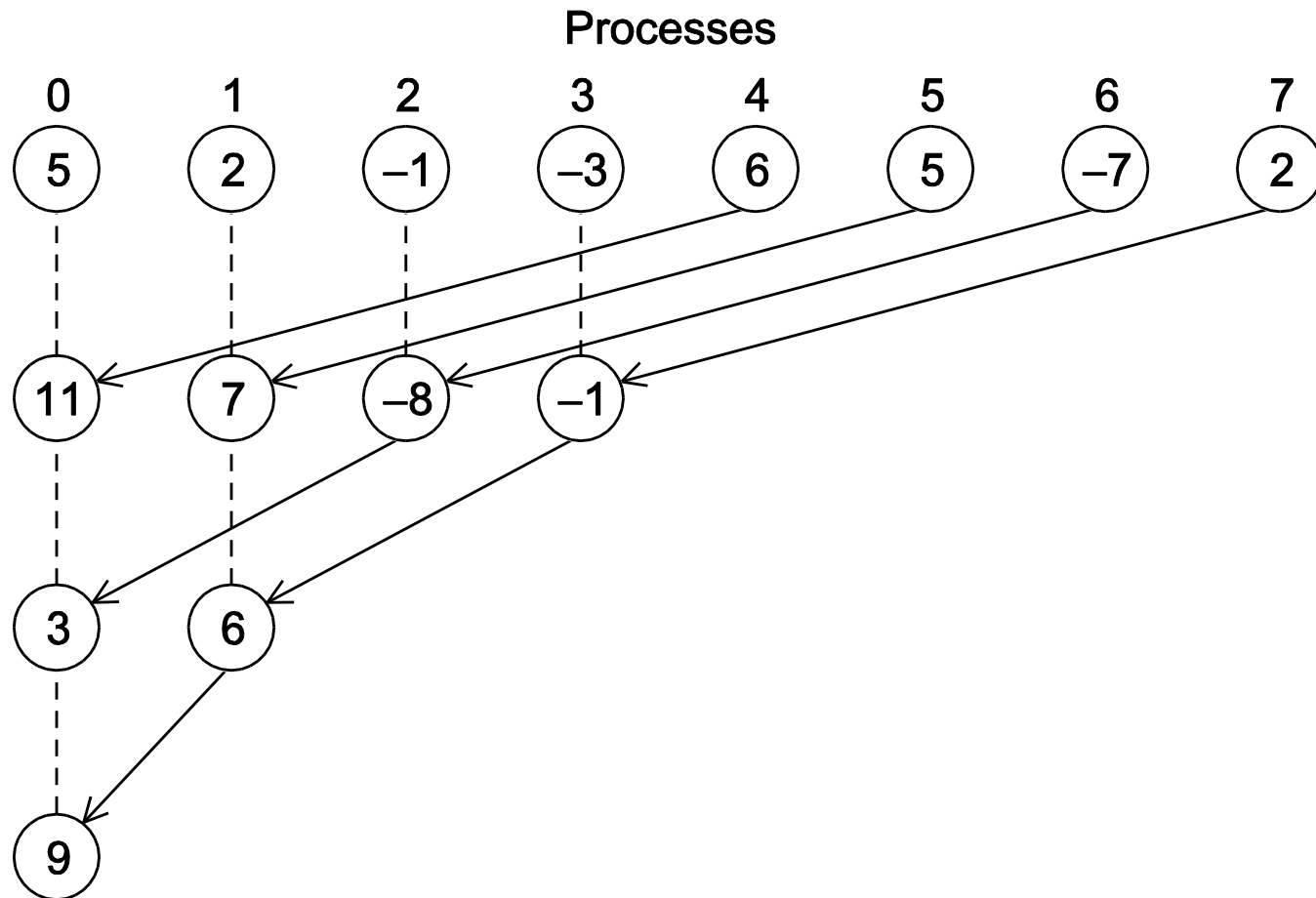
Predefined reduction operators in MPI

Operation Value	Meaning
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical and
<code>MPI_BAND</code>	Bitwise and
<code>MPI_LOR</code>	Logical or
<code>MPI BOR</code>	Bitwise or
<code>MPI_LXOR</code>	Logical exclusive or
<code>MPI_BXOR</code>	Bitwise exclusive or
<code>MPI_MAXLOC</code>	Maximum and location of maximum
<code>MPI_MINLOC</code>	Minimum and location of minimum

Implementation View of Global Reduction using a tree-structured sum

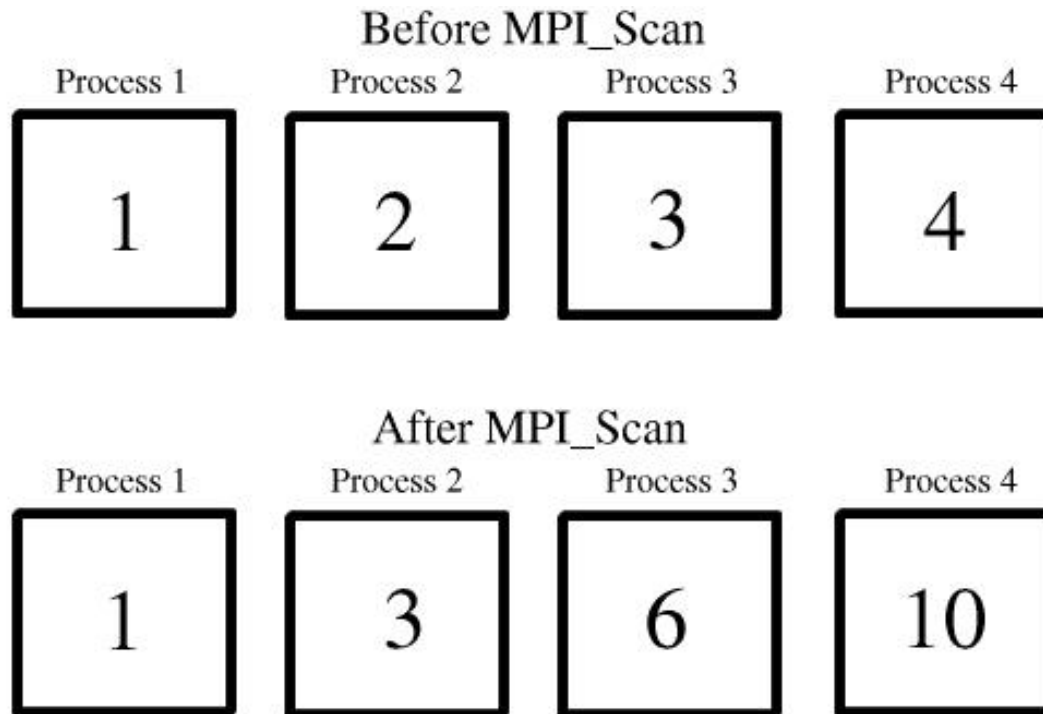


An alternative tree-structured global sum

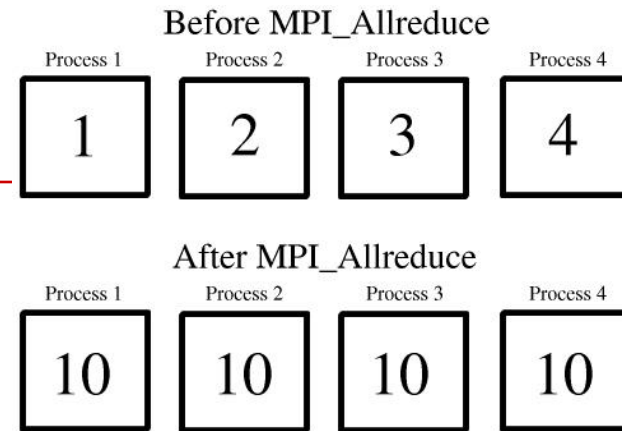


MPI Scan

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

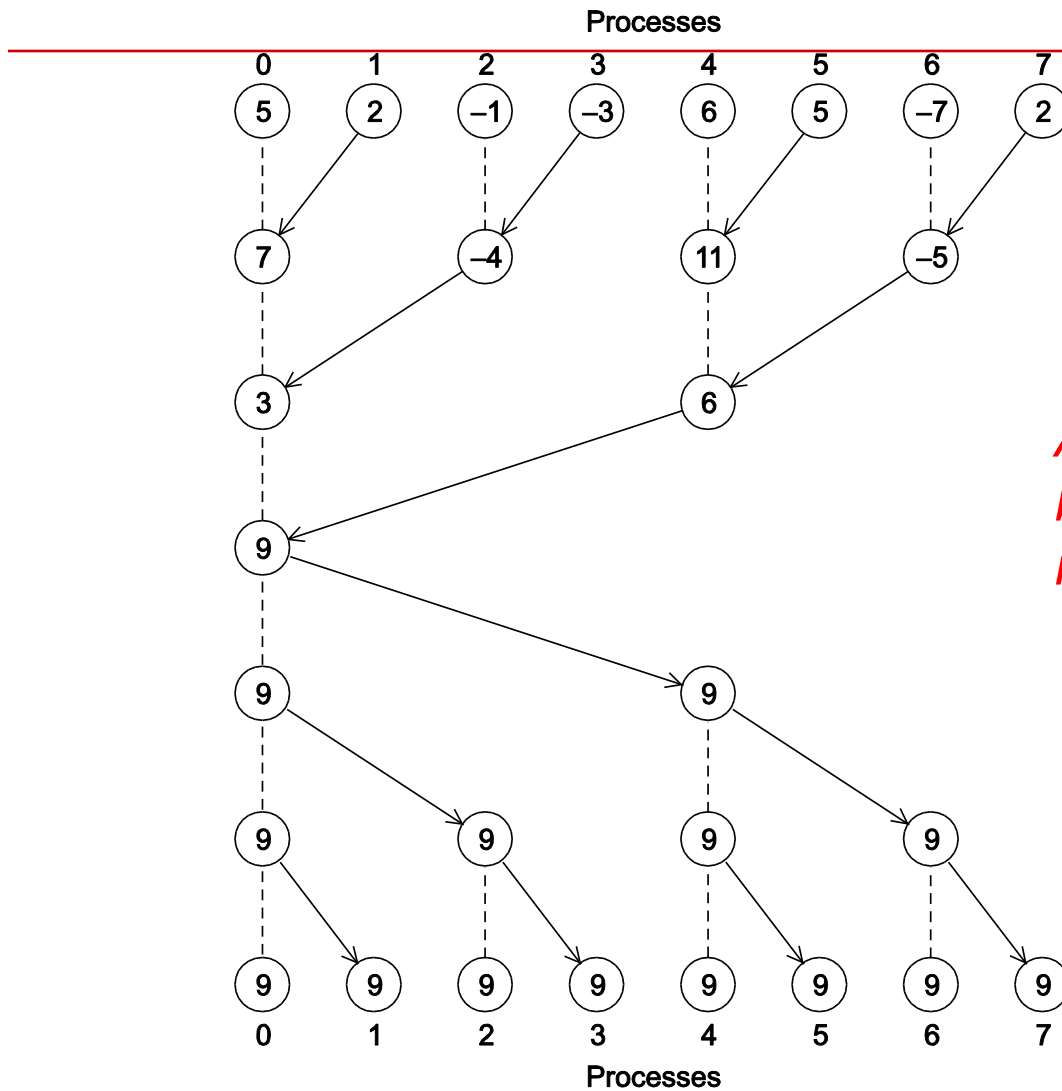


MPI_Allreduce



- 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

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

```
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.**
 - 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.”
 - 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 $2+1+2 = 5$.

Summation of Two Long Vectors

Collective Communication Application
Textbook p.109-111

Application Example: Distributed Summation of Two Long Vectors

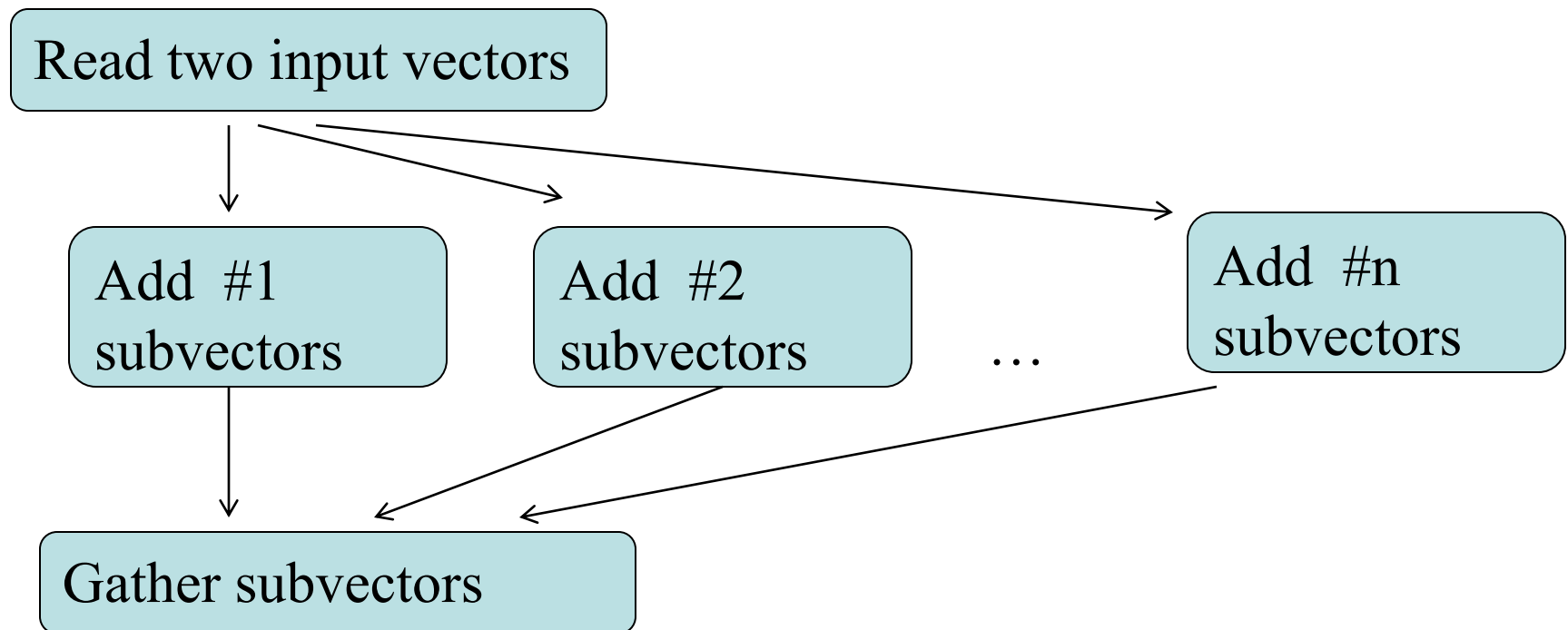
$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \dots, x_{n-1}) + (y_0, y_1, \dots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \dots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

Sequential code for computing a vector sum.

```
void Vector_sum(double x[], double y[], double z[], int n) {  
    int i;  
  
    for (i = 0; i < n; i++)  
        z[i] = x[i] + y[i];  
} /* Vector_sum */
```

Parallel implementation of vector addition

1. Divide each vector into n subvectors and distribute data
2. Add two subvectors at each process in parallel.
3. Gather the sum subvector from each process.



Partitioning options for distributing vectors

- **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.

Partitioning examples for data distribution

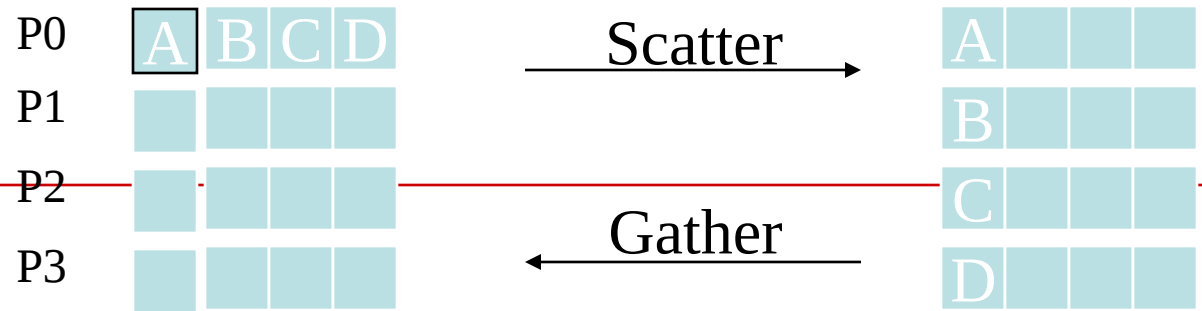
Each vector is divided into 12 subvectors
and then distributed to 3 processes

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

Local code for subvector addition

```
void Parallel_vector_sum(  
    double  local_x[]  /* in  */,  
    double  local_y[]  /* in  */,  
    double  local_z[]  /* out */,  
    int      local_n    /* in  */) {  
    int local_i;  
  
    for (local_i = 0; local_i < local_n; local_i++)  
        local_z[local_i] = local_x[local_i] + local_y[local_i];  
} /* Parallel_vector_sum */
```

Scatter for data distribution



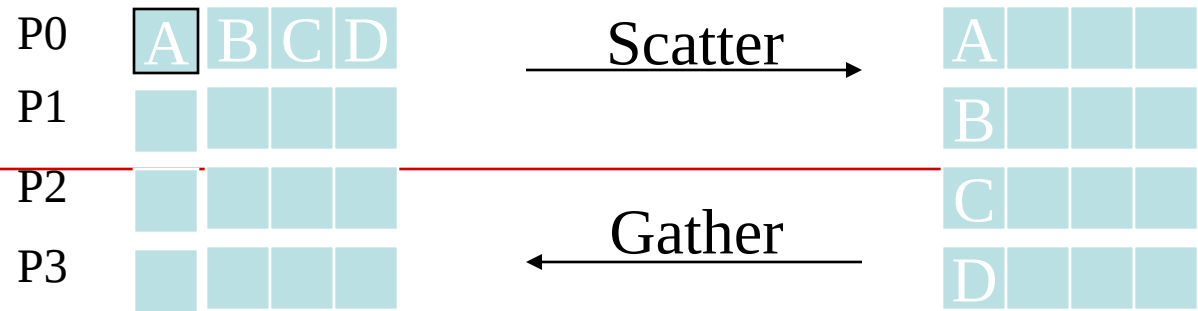
- **MPI_Scatter** can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```
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 */);
```

Reading and distributing a vector

```
void Read_vector(  
    double    local_a[]    /* out */,  
    int       local_n      /* in  */,  
    int       n            /* in  */,  
    char      vec_name[]   /* in  */,  
    int       my_rank      /* in  */,  
    MPI_Comm  comm        /* in  */) {  
  
    double* a = NULL;  
    int i;  
  
    if (my_rank == 0) {  
        a = malloc(n*sizeof(double));  
        printf("Enter the vector %s\n", vec_name);  
        for (i = 0; i < n; i++)  
            scanf("%lf", &a[i]);  
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,  
                    0, comm);  
        free(a);  
    } else {  
        MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,  
                    0, comm);  
    }  
} /* Read_vector */
```

Gather data from everybody



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

```
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  */);
```

MPI code to gather and print a distributed vector

```
void Print_vector(  
    double    local_b[]    /* in */,  
    int       local_n      /* in */,  
    int       n            /* in */,  
    char      title[]      /* in */,  
    int       my_rank      /* in */,  
    MPI_Comm  comm         /* in */) {  
  
    double* b = NULL;  
    int i;
```

Gather and print a distributed vector (2)

```
if (my_rank == 0) {
    b = malloc(n*sizeof(double));
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
               0, comm);
    printf("%s\n", title);
    for (i = 0; i < n; i++)
        printf("%f ", b[i]);
    printf("\n");
    free(b);
} else {
    MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
               0, comm);
}
} /* Print_vector */
```

Allgather

A			
B			
C			
D			

→ Allgather →

A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

- 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 */);
```

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

Task graph:

(S1)

(S2)

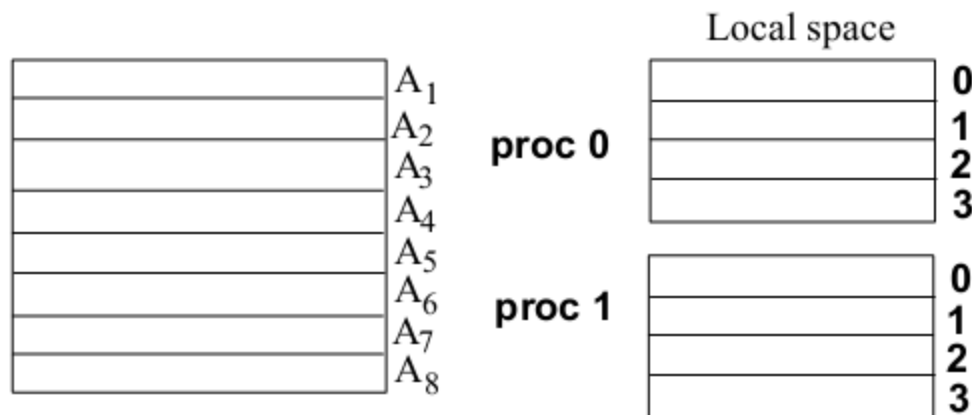
(S3)

(Sn)

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 $y = A * x$

```
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
```

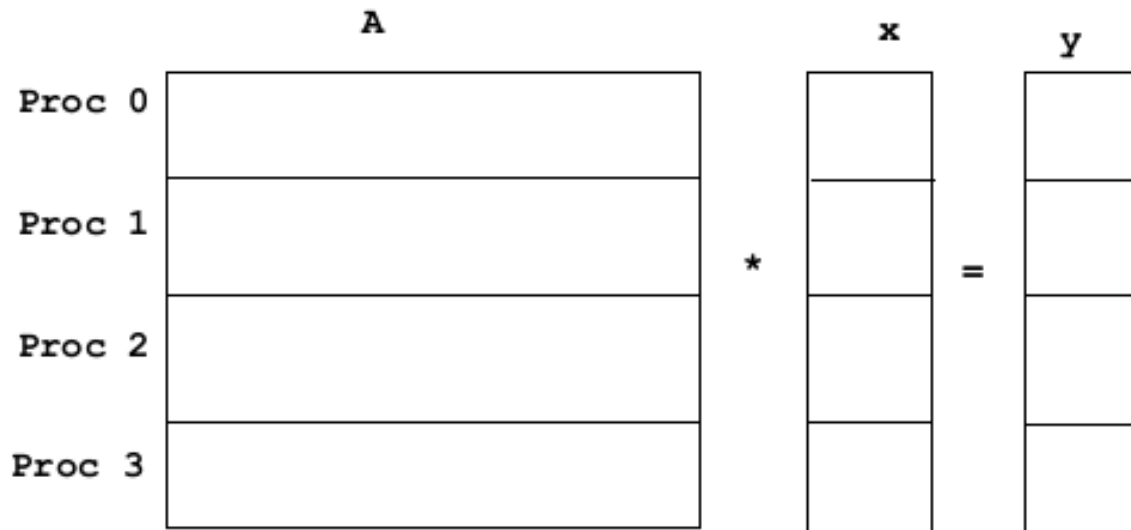
```
endfor
```

Evaluation: Parallel Time

- Ignore the cost of local address calculation.
- Each task performs n additions and n multiplications.
- Each addition/multiplication costs ω
- 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

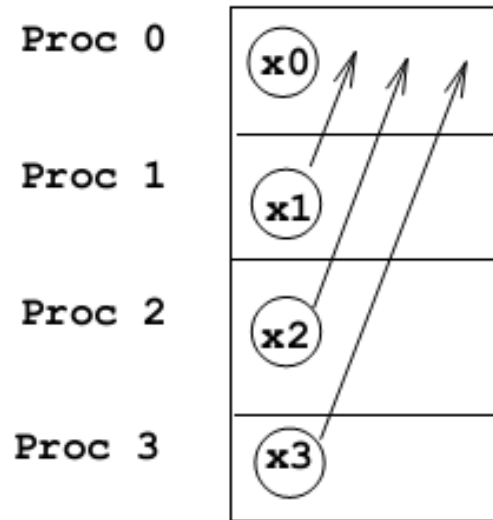


Need to redistribute vector x to everybody in order to perform parallel computation!

What MPI collective communication is needed?

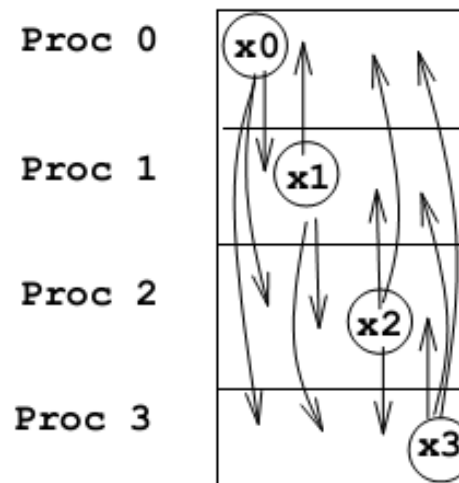
Communication Pattern for Data Redistribution

Data requirement for
Process 0



MPI_Gather

Data requirement for
other 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);
```

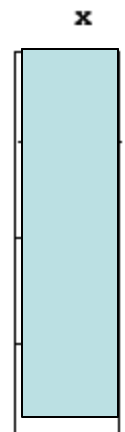
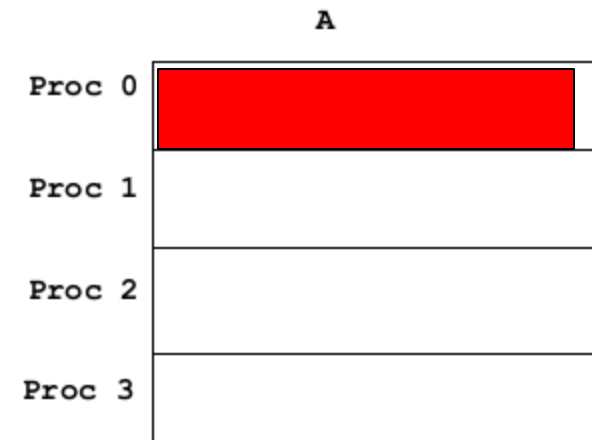
Repeat for all processes

It is the same as:

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

MPI Code for $y=A*x$

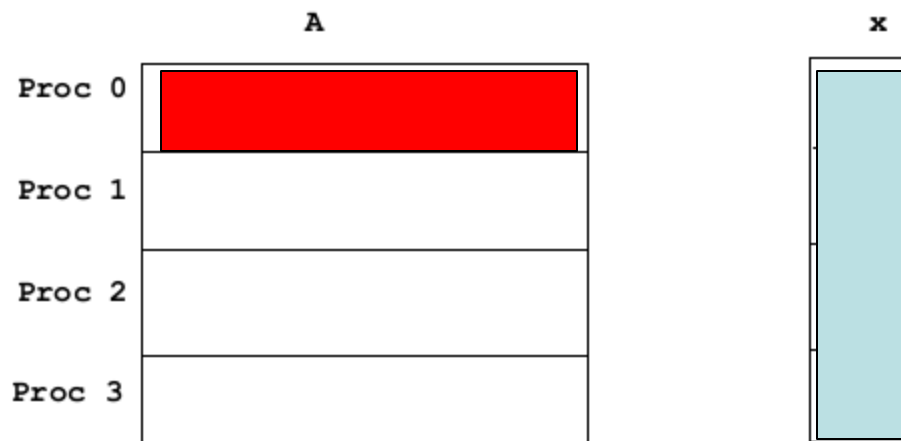
```
void Parallel_matrix_vector_prod(
    LOCAL_MATRIX_T  local_A
    int              m
    int              n
    float            local_x[]
    float            global_x[]
    float            local_y[]
    int              local_m
    int              local_n) {
    /* local_m = n/p, local_n = n/p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



MPI 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];  
}
```

```
}
```



$A = (a_{ij})$ is an $m \times n$ matrix

x is a vector with n components

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}

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

$\swarrow$
i-th component of y

$\nearrow$
Dot product of the i th row of A with x .

Use one dimensional C array to represent 2D matrix

$A = (a_{ij})$ is an $m \times n$ matrix

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

stored as

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

Sequential code for $y=A*x$

```

void Mat_vect_mult(
    double  A[] /* in */,
    double  x[] /* in */,
    double  y[] /* out */,
    int     m  /* in */,
    int     n  /* in */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
} /* Mat_vect_mult */

```

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}

Textbook MPI code for matrix-vector multiplication

```
void Mat_vect_mult(  
    double    local_A[]    /* in  */,  
    double    local_x[]    /* in  */,  
    double    local_y[]    /* out */,  
    int        local_m      /* in  */,  
    int        n            /* in  */,  
    int        local_n      /* in  */,  
    MPI_Comm   comm        /* in  */) {  
    double* x;  
    int local_i, j;  
    int local_ok = 1;
```

Textbook MPI code for $y=A*x$

```
x = malloc(n*sizeof(double));
MPI_Allgather(local_x, local_n, MPI_DOUBLE,
              x, local_n, MPI_DOUBLE, comm);

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



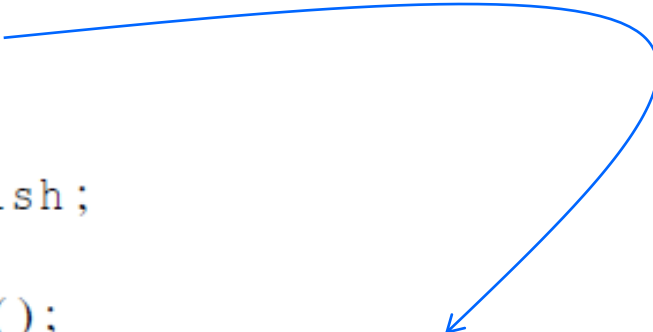


PERFORMANCE EVALUATION

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 this case, you don't need to link in the MPI libraries.
- 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 of MPI_Barrier before time measurement

- Ensures that no process will return from calling it until every process in the communicator has started calling it.

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



Example of using MPI_Barrier

```
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**.

-

A PARALLEL SORTING ALGORITHM

Textbook p. 127-136

Sorting

- **n keys and $p = \# \text{ of processes}$.**
- **n/p keys assigned to each process.**
- **When the algorithm terminates:**
 - The keys assigned to each process should be sorted in (say) increasing order.
 - If $0 \leq q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r .

Simple bubble sort with pairwise swaps



```
void Bubble_sort(  
    int a[] /* in/out */,  
    int n /* in */) {  
    int list_length, i, temp;  
  
    for (list_length = n; list_length >= 2; list_length--)  
        for (i = 0; i < list_length-1; i++)  
            if (a[i] > a[i+1]) {  
                temp = a[i];  
                a[i] = a[i+1];  
                a[i+1] = temp;  
            }  
    } /* Bubble_sort */
```

	6	1	2	3	4	5	unsorted
	6	1	2	3	4	5	6 > 1, swap
	1	6	2	3	4	5	6 > 2, swap
	1	2	6	3	4	5	6 > 3, swap
	1	2	3	6	4	5	6 > 4, swap
	1	2	3	4	6	5	6 > 5, swap

Odd-even sort

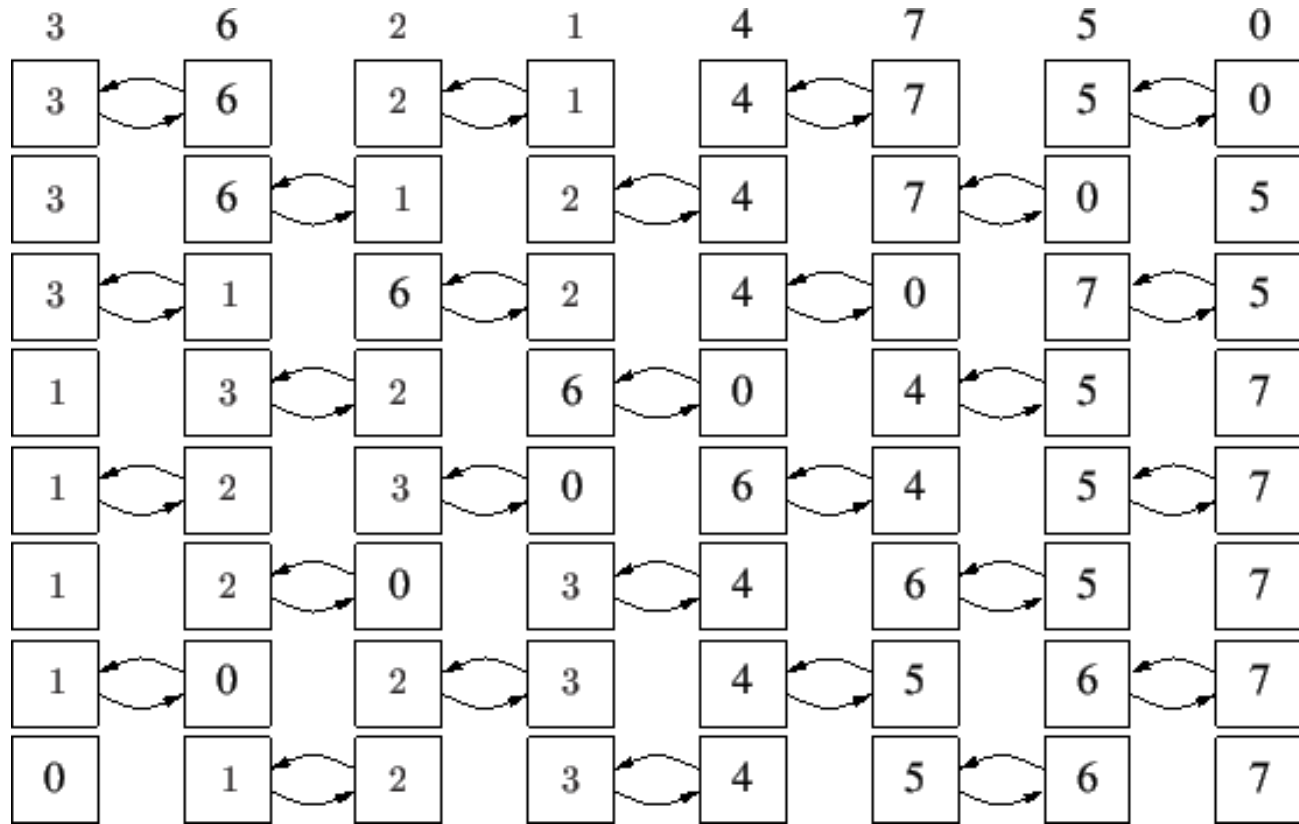
- **Expose more parallelism with pairwise swaps**
 - Also called Odd-even transposition sort or brick sort.
- **Algorithm: Repeat at most n phases**
 - Even phases, compare swaps:
$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \dots$$
 - Odd phases, compare swaps:
$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \dots$$
 - Complexity: best case $O(n)$. Worst case $O(n^2)$

Example of odd-even sort

Start: 3, 6, 2, 1, 4, 7, 5, 0

Even phase: compare-swap (3,6), (2,1), (4,7), (5,0)
getting the list 3, 6, 1, 2, 4, 7, 0, 5

Odd phase: compare-swap (6,1), (2,4), (7,0)



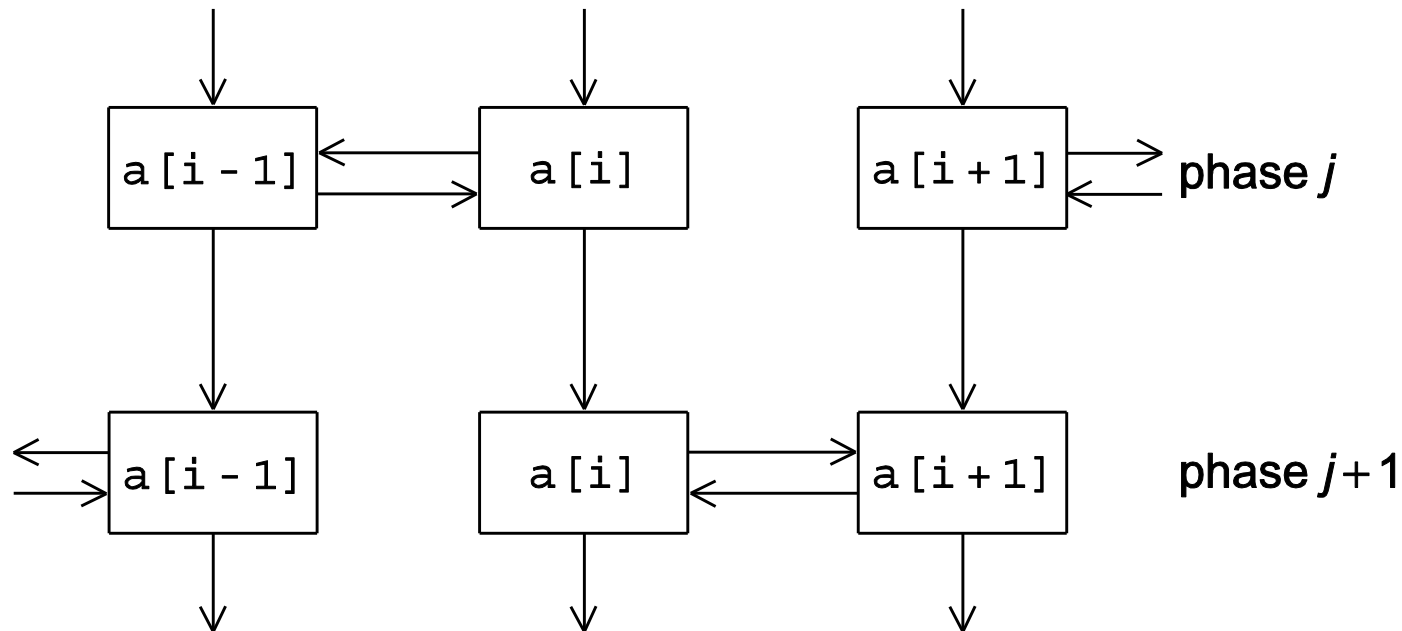
Sequential odd-even sort

```
void Odd_even_sort(  
    int a[] /* in/out */,  
    int n /* in */) {  
    int phase, i, temp;  
  
    for (phase = 0; phase < n; phase++)  
        if (phase % 2 == 0) { /* Even phase */  
            for (i = 1; i < n; i += 2)  
                if (a[i-1] > a[i]) {  
                    temp = a[i];  
                    a[i] = a[i-1];  
                    a[i-1] = temp;  
                }  
        } else { /* Odd phase */  
            for (i = 1; i < n-1; i += 2)  
                if (a[i] > a[i+1]) {  
                    temp = a[i];  
                    a[i] = a[i+1];  
                    a[i+1] = temp;  
                }  
        }  
    } /* Odd_even_sort */  
}
```

Communications among tasks in odd-even sort

Computation contains a set of tasks

Each task handling $a[i]$ is labeled with " $a[i]$ ".



Parallel odd-even sort for n keys and p processes ($n=p$)

		P ₀	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇				
Time	Step												
	0	4	↔	2	7	↔	8	5	↔	1	3	↔	6
	1	2		4	↔	7	8	↔	1	5	↔	3	6
	2	2	↔	4	7	↔	1	8	↔	3	5	↔	6
	3	2		4	↔	1	7	↔	3	8	↔	5	6
	4	2	↔	1	4	↔	3	7	↔	5	8	↔	6
	5	1		2	↔	3	4	↔	5	7	↔	6	8
	6	1	↔	2	3	↔	4	5	↔	6	7	↔	8
7	1	2	↔	3	4	↔	5	6	↔	7	8		

- Weakness**

- When $n \gg p$, modify to let each process handle n/p keys
- Too much communication overhead with key-level fine-grain data exchange/swap

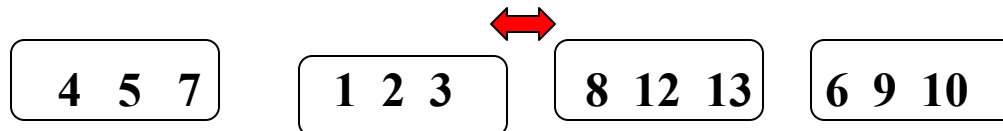
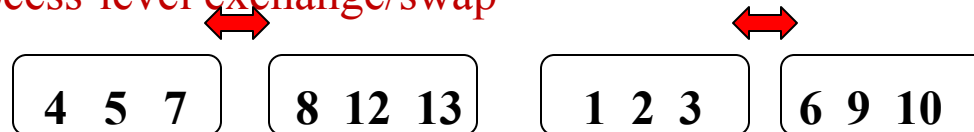
Parallel odd-even sort for n keys and p processes ($n \gg p$)



Local sort



Process-level exchange/swap



SORTED: 1 2 3 4 5 6 7 8 9 10 12 13

Parallel odd-even sort of n keys with p processes

- Each process owns n/p keys.
- First each process sorts its keys locally in parallel.
 - E.g. call C library `qsort` for quick sorting
- Repeat at most p phases
 - Even phases, process with even ID exchanges data with odd ID and swaps keys
 - $(P_0, P_1), (P_2, P_3), (P_4, P_5) \dots$
 - Odd phases, compare swaps:
 - $(P_1, P_2), (P_3, P_4), (P_5, P_6) \dots$

Textbook example of parallel odd-even 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

Parallel time of odd-even sort

- **Total cost**
 - Local sorting using the best algorithm.
 - At most p phases
 - Neighbor process data exchanges of n/p keys
 - Merge and split two n/p key lists
- **$T_{\text{par}} = (\text{local sort}) + (p \text{ data exchanges}) + (p \text{ merges/splits})$**
 - $= O((n/p)\log(n/p)) + p*O(n/p) + p*O(n/p)$**
 - $= O((n/p)\log(n/p)) + O(2n)$**

Pseudo-code

Comm_sz= # of processes

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (I'm not idle) {
        Send my keys to partner;
        Receive keys from partner;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

Compute_partner(phase,my_rank)

```
if (phase % 2 == 0)          /* Even phase */
    if (my_rank % 2 != 0)     /* Odd rank */
        partner = my_rank - 1;
    else                      /* Even rank */
        partner = my_rank + 1;
else                          /* Odd phase */
    if (my_rank % 2 != 0)     /* Odd rank */
        partner = my_rank + 1;
    else                      /* Even rank */
        partner = my_rank - 1;
if (partner == -1 || partner == comm_sz)
    partner = MPI_PROC_NULL;
```

Merge/split in parallel odd-even sort

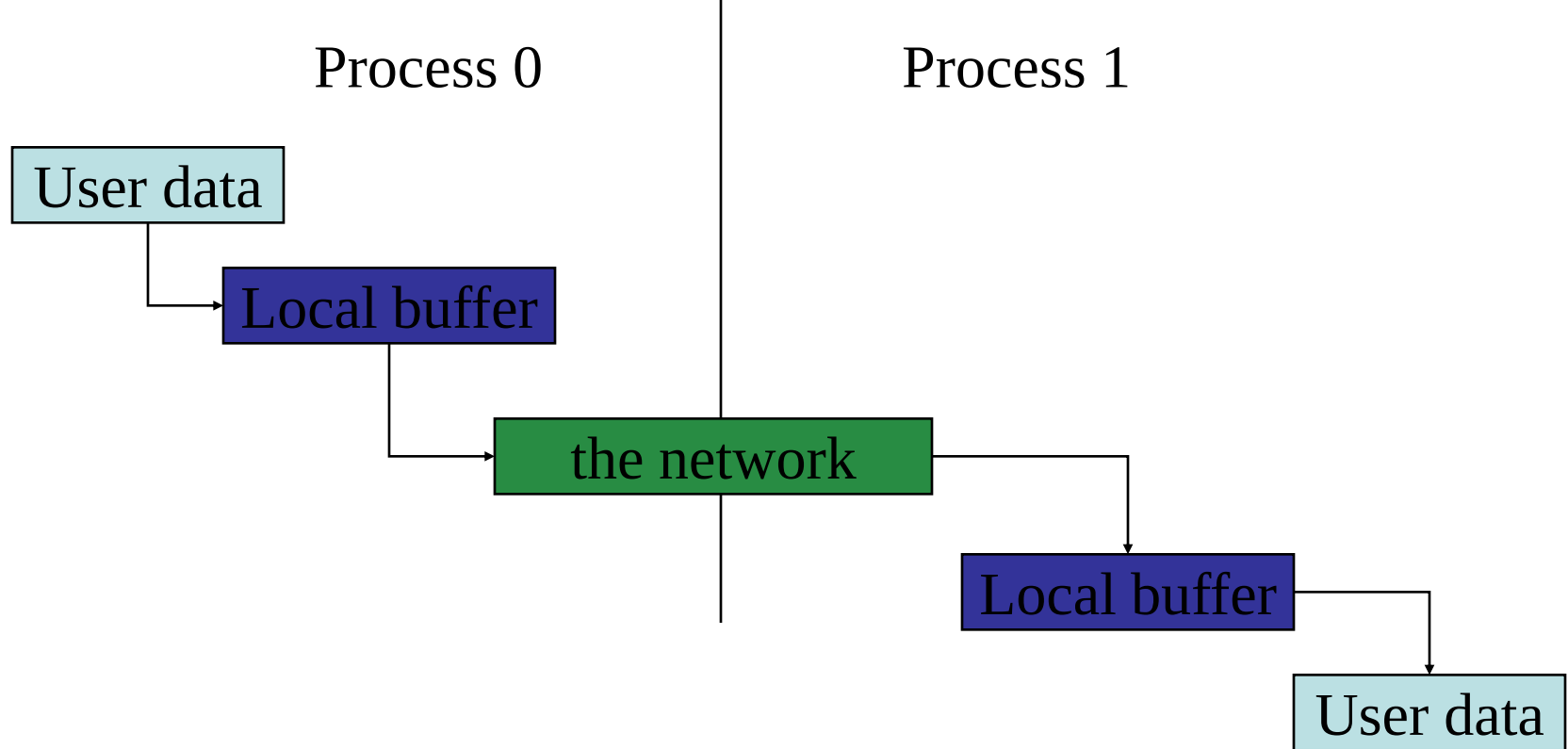
```
void Merge_low(  
    int  my_keys[],      /* in/out    */  
    int  recv_keys[],   /* in      */  
    int  temp_keys[],   /* scratch */  
    int  local_n        /* = n/p, in */) {  
    int m_i, r_i, t_i;  
  
    m_i = r_i = t_i = 0;  
    while (t_i < local_n) {  
        if (my_keys[m_i] <= recv_keys[r_i]) {  
            temp_keys[t_i] = my_keys[m_i];  
            t_i++; m_i++;  
        } else {  
            temp_keys[t_i] = recv_keys[r_i];  
            t_i++; r_i++;  
        }  
    }  
  
    for (m_i = 0; m_i < local_n; m_i++)  
        my_keys[m_i] = temp_keys[m_i];  
} /* Merge_low */
```

Safety in MPI programs

- **The 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.

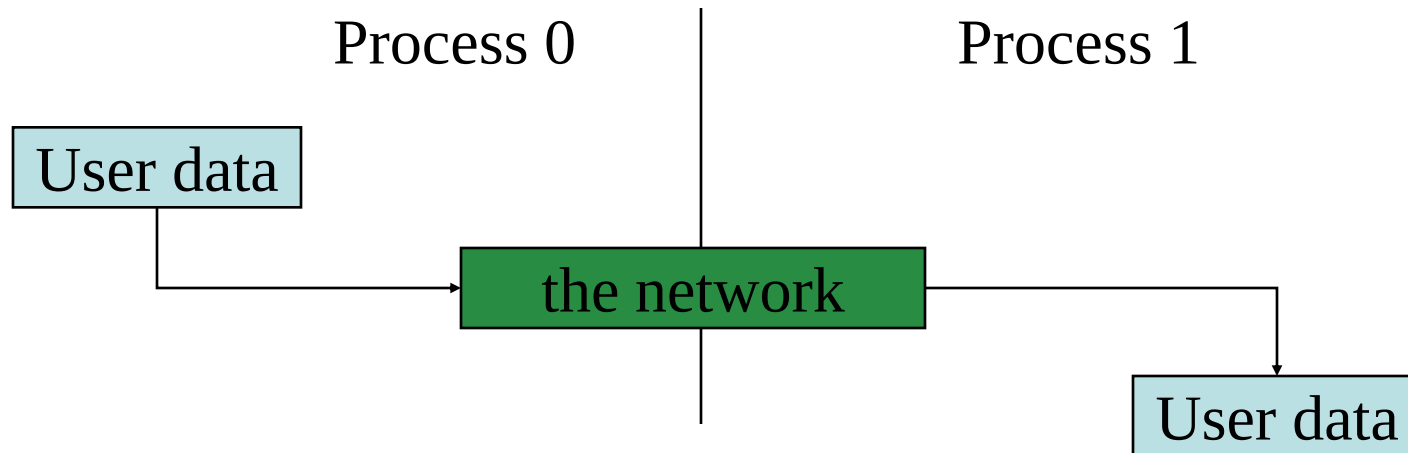
Buffers

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



Avoiding Buffering

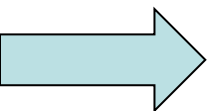
- Avoiding copies uses less memory
- May use more or less time



This requires that `MPI_Send` wait on delivery, or that `MPI_Send` return before transfer is complete, and we wait later.

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.

Will there be a deadlock?

- Assume tag/process ID is assigned properly.

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

Example of Unsafe MPI code with possible eadlocks

- **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)
- **What happens with this code?**

Process 0

Process 1

Send (1)

Send (0)

Recv (1)

Recv (0)

- This is called “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)

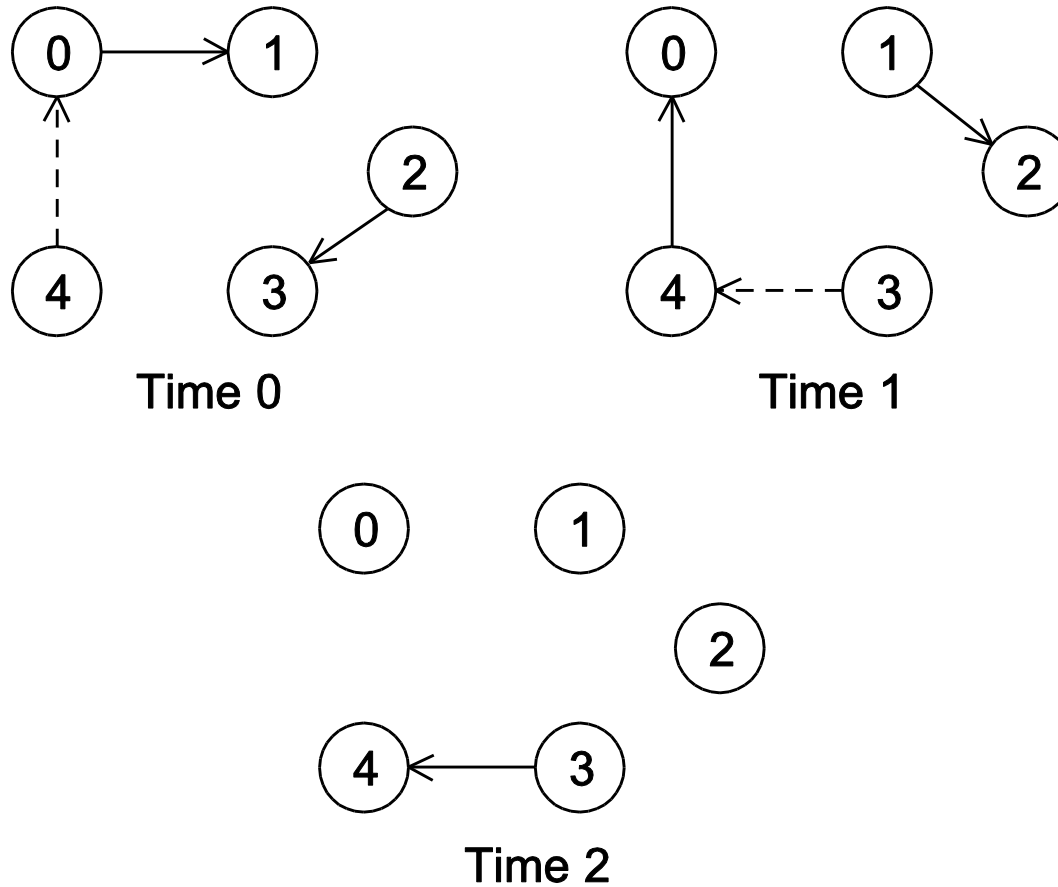
Restructuring communication in odd-even sort

```
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);  
}
```

Uncertainty with five processes



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 */);
```

Use MPI_Sendrecv() in odd-even sort

- An alternative to scheduling deterministic communications
 - The dest and the source can be the same or different.
 - Send and receive datatypes may be different
 - Can use Sendrecv with plain Send or Recv (or Irecv or Ssend_init, ...)
- Ensure safer communication behavior so that the program won't hang or crash.

MPI_Sendrecv(
mykeys, n/comm_sz, MPI_INT, partner, 0,
recvkeys, n/comm_sz, MPI_INT, partner, 0,
comm, MPI_Status_ignore)

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

Run-times of parallel odd-even sort

Processes	Number of Keys (in thousands)				
	200	400	800	1600	3200
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

(times are in milliseconds)

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”.
 - $\text{Speedup} = \text{sequential time} / \text{parallel time}$
 - $\text{Efficiency} = \text{Speedup} / p$
 - 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.**

Backup Slides

-
- CS267 from J. Demmel/K. Yelick at Berkeley

MPI's Non-blocking Operations

- Non-blocking operations return (immediately) “request handles” that can be tested and waited on:

```
MPI_Request request;
```

```
MPI_Status status;
```

```
MPI_Isend(start, count, datatype,  
          dest, tag, comm, &request);
```

```
MPI_Irecv(start, count, datatype,  
          dest, tag, comm, &request);
```

```
MPI_Wait(&request, &status);  
(each request must be Waited on)
```

- One can also test without waiting:

```
MPI_Test(&request, &flag, &status);
```

Communication

Modes

- MPI provides multiple *modes* for sending messages:
 - Synchronous mode (**MPI_Ssend**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
 - Buffered mode (**MPI_Bsend**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.)
 - Ready mode (**MPI_Rsend**): user guarantees that a matching receive has been posted.
 - Allows access to fast protocols
 - undefined behavior if matching receive not posted
- **Non-blocking versions (MPI_Issend, etc.)**
- **MPI_Recv receives messages sent in any mode.**
- See www.mpi-forum.org for summary of all flavors

Not

Covered

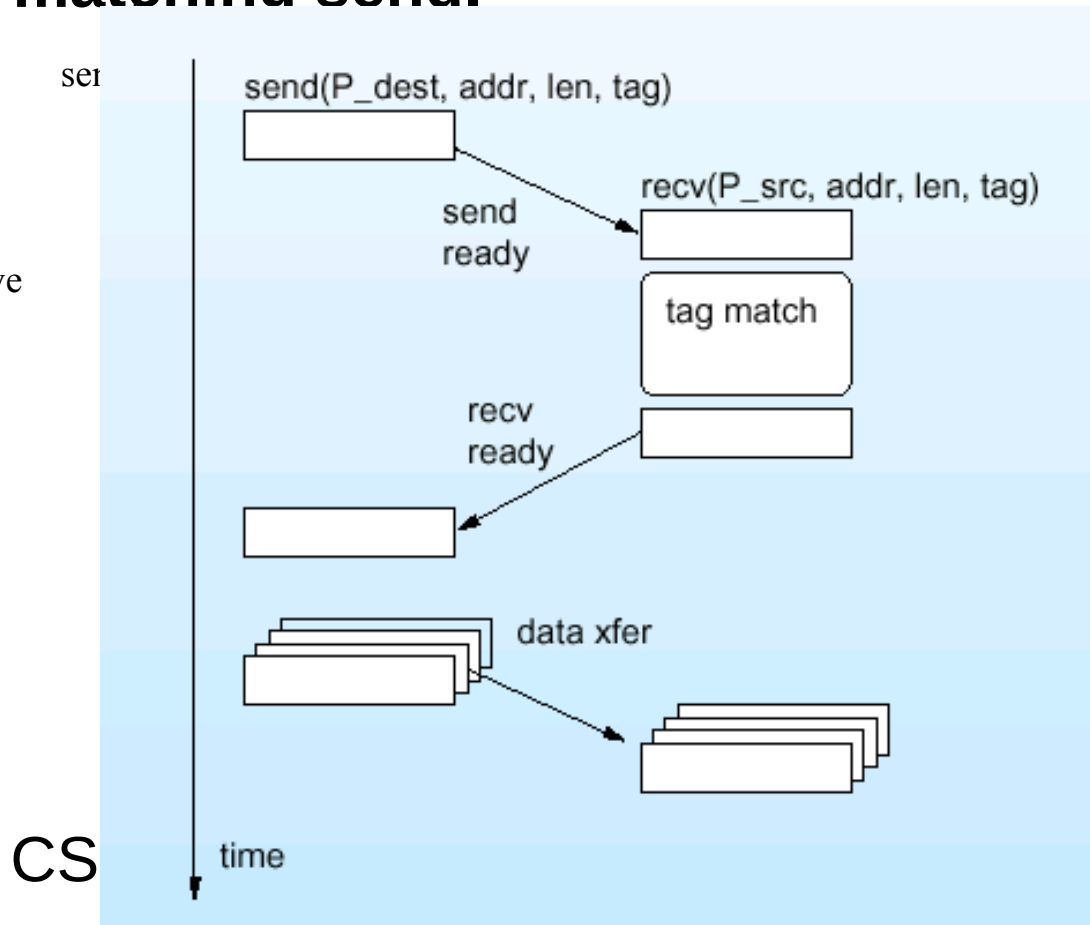
- **Topologies: map a communicator onto, say, a 3D Cartesian processor grid**
 - Implementation can provide ideal logical to physical mapping
- **Rich set of I/O functions: individual, collective, blocking and non-blocking**
 - Collective I/O can lead to many small requests being merged for more efficient I/O
- **One-sided communication: puts and gets with various synchronization schemes**
 - Implementations not well-optimized and rarely used
 - Redesign of interface is underway
- **Task creation and destruction: change number of tasks during a run**
 - Few implementations available

Implementing Synchronous Message

Passing

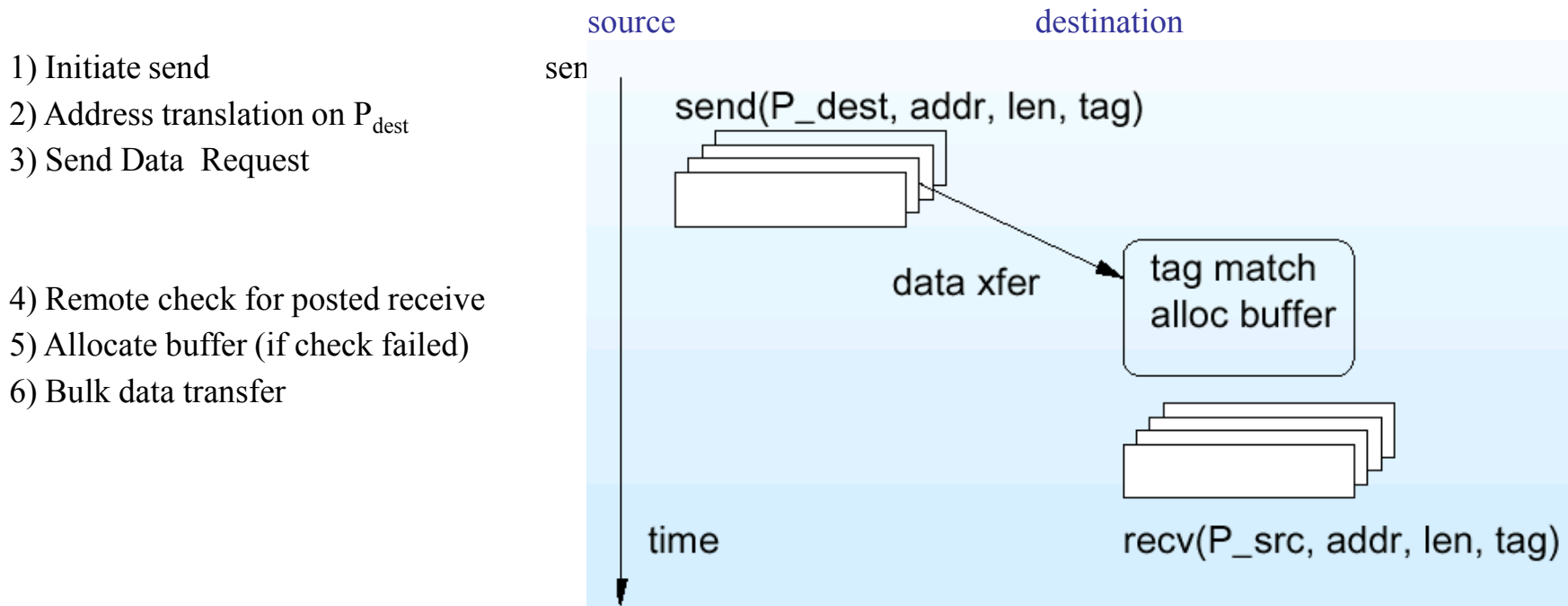
- Send operations complete after matching receive and source data has been sent.
- Receive operations complete after data transfer is complete from matching send.

- 1) Initiate send
- 2) Address translation on P_{dest}
- 3) Send-Ready Request
- 4) Remote check for posted receive
- 5) Reply transaction
- 6) Bulk data transfer



Implementing Asynchronous Message Passing

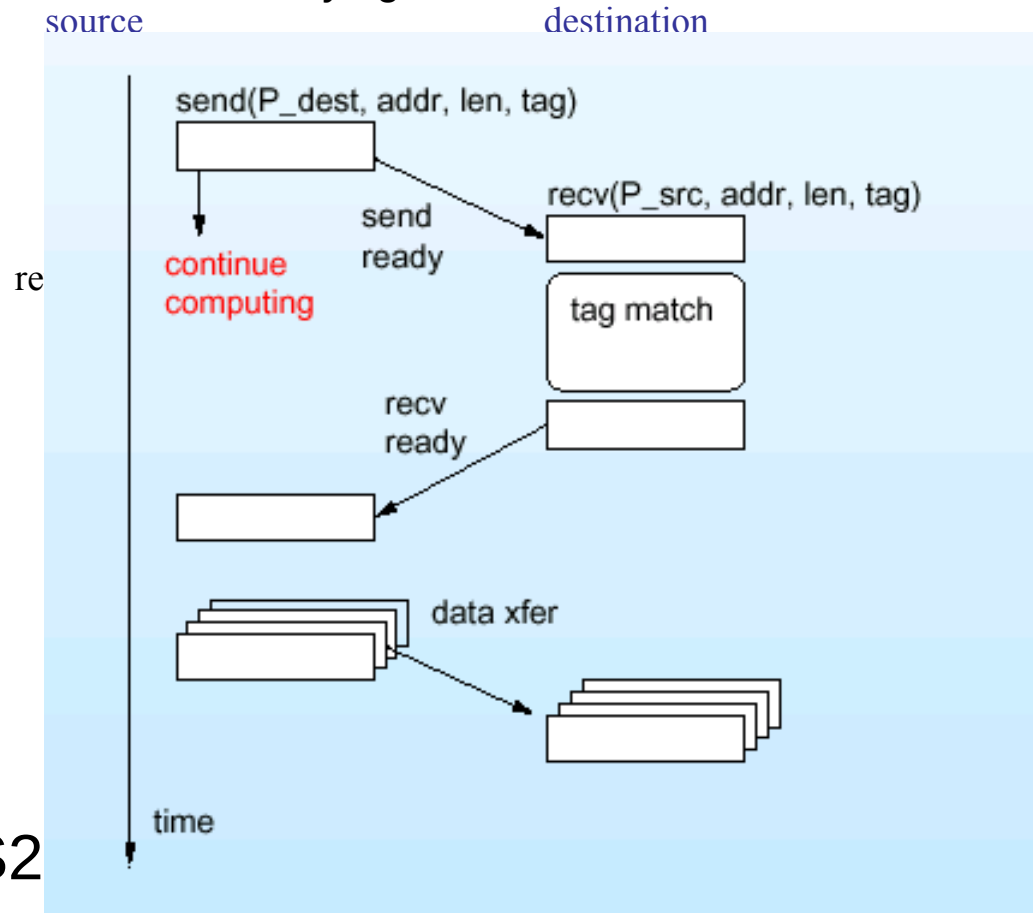
- **Optimistic single-phase protocol assumes the destination can buffer data on demand.**



Safe Asynchronous Message Passing

- Use 3-phase protocol
- Buffer on sending side
- Variations on send completion
 - wait until data copied from user to system buffer
 - don't wait -- let the user beware of modifying data

- 1) Initiate send
- 2) Address translation on P_{dest}
- 3) Send-Ready Request
- 4) Remote check for posted receive
record send-rdy
- 5) Reply transaction
- 6) Bulk data transfer



Books on MPI

- ***Using MPI: Portable Parallel Programming with the Message-Passing Interface (2nd edition)*** by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- ***Using MPI-2: Portable Parallel Programming with the Message-Passing Interface***, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- ***MPI: The Complete Reference - Vol 1 The MPI Core***, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- ***MPI: The Complete Reference - Vol 2 The MPI Extensions***, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- ***Designing and Building Parallel Programs***, by Ian Foster, Addison-Wesley, 1995.
- ***Parallel Programming with MPI***, by Peter Pacheco, Morgan-Kaufmann, 1997.

