

Introduction to Parallel Programming Using MPI (2)

Jun Ni, Ph.D.

Associate Professor

Department of Radiology, College of Medicine

Information Technology Services

The University of Iowa

Collective Communication in MPI

- In previous example (trap.c), after carrying out the basic setup tasks (MPI_Init, MPI_Comm_size, and MPI_Comm_rank), processes are idle while process 0 collect the input data.
- The inefficient case happens at the end of program.
- Question is how to efficiently collect data from other processes for output.

- Parallel programming with collective communication using broadcasting
 - not point to point communication
 - one process sends data to every processes
 - solve static implementation using assignments of input parameters
 - broadcast to all of the processes
 - It is group communication

- Parallel programming with collective communication using broadcasting
 - one process share data with other processes
 - reduction to perform summation by master process using MPI_Reduce with MPI_SUM (others are MPI_MAX, MPI_MIN, MPI_MAXLOC, MPI_MINLOC)

- Parallel programming with collective communication using broadcasting
 - Using
 - MPI_Init and MPI_finalize
 - MPI_Comm_rank and MPI_Comm_size
 - MPI_Bcast, MPI_Reduce, MPI_SUM

Collective Communication in MPI

- Algorithm using tree-structured communication
- It is also called binary or hierarchy structure. It provides the better efficiency of message transferring.

Step 1

Process 0



Process 1

Passing message

Step 2

Process 0



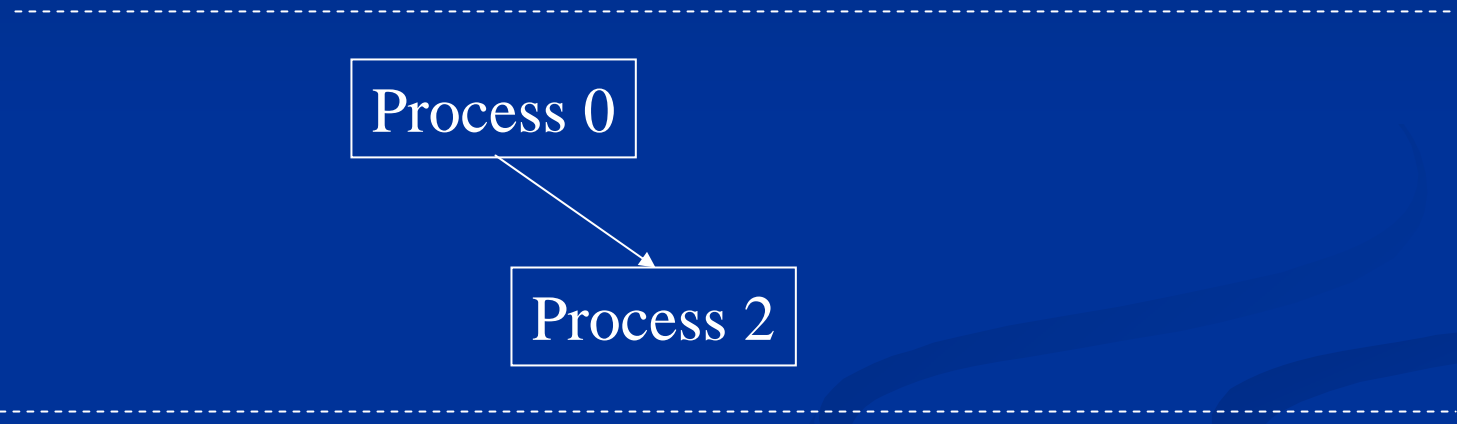
Process 2

Step 3

Process 0



Process 3



Step 4

Process 0

Passing message

Process 4

Step 5

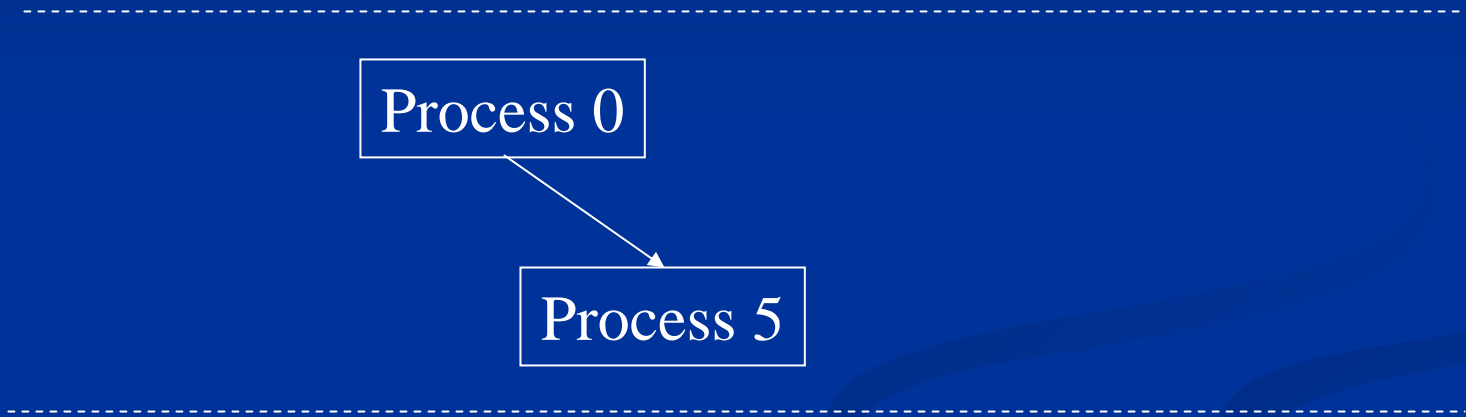
Process 0

Process 5

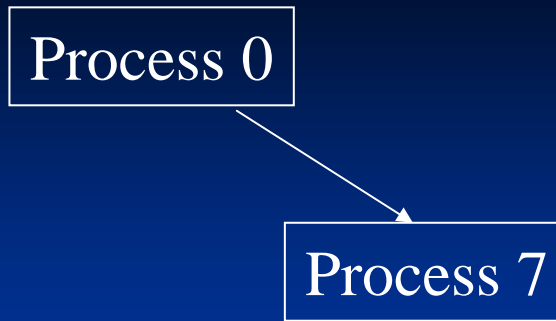
Step 6

Process 0

Process 6



Step 7



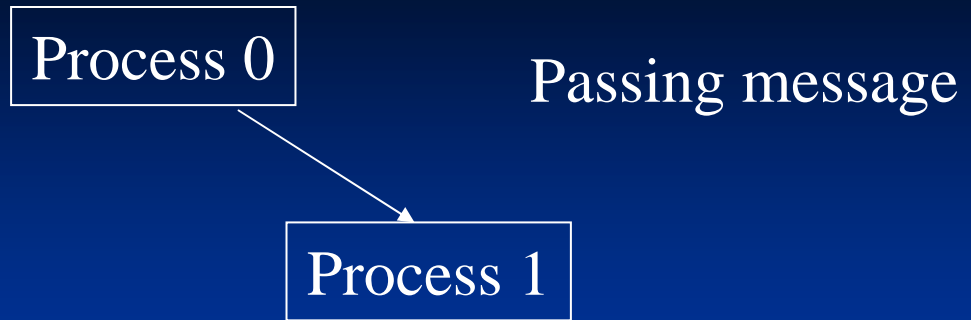
Passing message

Total 7 steps to accomplish the message passing procedure.

Not efficient!

Let's propose an alternative communication approach.

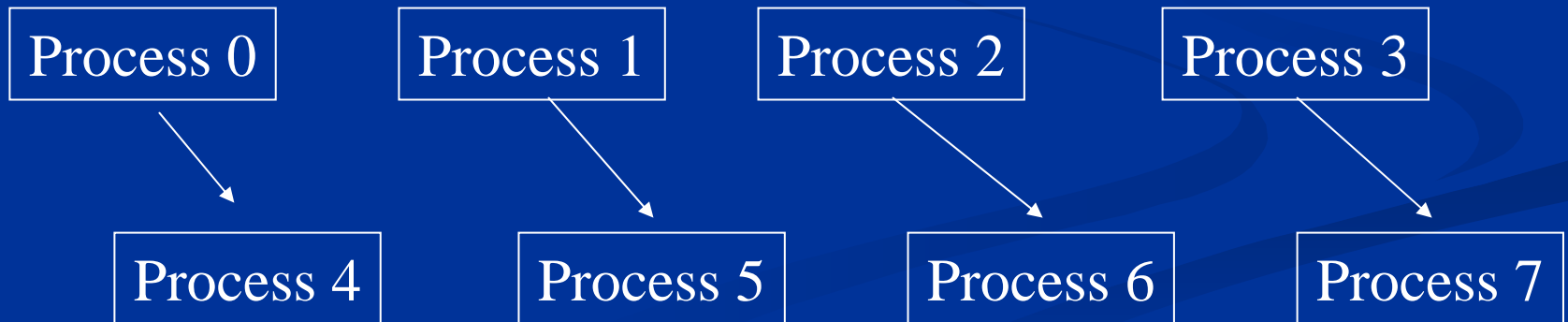
Step 1



Step 2



Step 3



Collective Communication in MPI

- In general, if we have p processes, the communication procedure allows us to distribute the input data in $\log_2(p)$ stages, rather than $p-1$ stages. (previous example, $p=8$, we reduced steps from 7 to 3).
- Modify our input data function from `get_data()`, we may have the following code.

```
/* get_data1.c -- Parallel Trapezoidal Rule; uses a hand-coded
 *   tree-structured broadcast.
 *
 * Input:
 *   a, b: limits of integration.
 *   n: number of trapezoids.
 * Output: Estimate of the integral from a to b of f(x)
 *   using the trapezoidal rule and n trapezoids.
 *
 * Notes:
 *   1. f(x) is hardwired.
 *   2. the number of processes (p) should evenly divide
 *      the number of trapezoids (n).
 *
 * See Chap. 5, pp. 65 & ff. in PPMPI.
 */
```

```
#include <stdio.h>

/* We'll be using MPI routines, definitions, etc. */
#include "mpi.h"

main(int argc, char** argv)
{
    int    my_rank; /* My process rank */
    int    p;      /* The number of processes */
    float  a;      /* Left endpoint */
    float  b;      /* Right endpoint */
    int    n;      /* Number of trapezoids */
    float  h;      /* Trapezoid base length */
    float  local_a; /* Left endpoint my process */
    float  local_b; /* Right endpoint my process */
    int    local_n; /* Number of trapezoids for */
```

```
/* my calculation          */
float    integral; /* Integral over my interval */
float    total;    /* Total integral          */
int      source;   /* Process sending integral */
int      dest = 0; /* All messages go to 0    */
int      tag = 0;
MPI_Status status;

void Get_data1(float* a_ptr, float* b_ptr, int* n_ptr, int my_rank,
              int p);
float Trap(float local_a, float local_b, int local_n,
          float h); /* Calculate local integral */

/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);
```

```
/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

Get_data1(&a, &b, &n, my_rank, p);

h = (b-a)/n; /* h is the same for all processes */
local_n = n/p; /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h. So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);
```

```
/* Add up the integrals calculated by each process */
if (my_rank == 0)
{
    total = integral;
    for (source = 1; source < p; source++)
    {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                MPI_COMM_WORLD, &status);
        total = total + integral;
    }
}
else
{
    MPI_Send(&integral, 1, MPI_FLOAT, dest,
            tag, MPI_COMM_WORLD);
}
```

```
/* Print the result */
if (my_rank == 0)
{
    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %f\n",
        a, b, total);
}

/* Shut down MPI */
MPI_Finalize();
} /* main */

/*****/
/* Ceiling of log2(x) is just the number of times
```

```
* times x-1 can be divided by 2 until the quotient
* is 0. Dividing by 2 is the same as right shift.
*/
int Ceiling_log2(int x /* in */)
{
    /* Use unsigned so that right shift will fill
    * leftmost bit with 0
    */
    unsigned temp = (unsigned) x - 1;
    int result = 0;

    while (temp != 0) {
        temp = temp >> 1;
        result = result + 1 ;
    }
    return result;
} /* Ceiling_log2 */
```

```
/**/
int I_receive(
    int stage /* in */,
    int my_rank /* in */,
    int* source_ptr /* out */)
{
    int power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if ((power_2_stage <= my_rank) &&
        (my_rank < 2*power_2_stage)){
        *source_ptr = my_rank - power_2_stage;
        return 1;
    } else return 0;
} /* I_receive */
```

```

/*****/
int I_send(
    int stage /* in */,
    int my_rank /* in */,
    int p /* in */,
    int* dest_ptr /* out */)
{
    int power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if (my_rank < power_2_stage){
        *dest_ptr = my_rank + power_2_stage;
        if (*dest_ptr >= p) return 0;
        else return 1;
    } else return 0;
} /* I_send

```

```
*/  
  
/*****  
void Send(  
    float a    /* in */,  
    float b    /* in */,  
    int  n     /* in */,  
    int  dest  /* in */)   
{  
  
    MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);  
    MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);  
    MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);  
} /* Send */
```

```
/**
void Receive(
    float* a_ptr /* out */,
    float* b_ptr /* out */,
    int* n_ptr /* out */,
    int source /* in */)
{
    MPI_Status status;

    MPI_Recv(a_ptr, 1, MPI_FLOAT, source, 0,
             MPI_COMM_WORLD, &status);
    MPI_Recv(b_ptr, 1, MPI_FLOAT, source, 1,
             MPI_COMM_WORLD, &status);
    MPI_Recv(n_ptr, 1, MPI_INT, source, 2,
             MPI_COMM_WORLD, &status);
} /* Receive */
```

```
/**
 *
 */
/* Function Get_data1
 * Reads in the user input a, b, and n.
 * Input parameters:
 * 1. int my_rank: rank of current process.
 * 2. int p: number of processes.
 * Output parameters:
 * 1. float* a_ptr: pointer to left endpoint a.
 * 2. float* b_ptr: pointer to right endpoint b.
 * 3. int* n_ptr: pointer to number of trapezoids.
 * Algorithm:
 * 1. Process 0 prompts user for input and
 *    reads in the values.
 * 2. Process 0 sends input values to other
```

```
*      processes using hand-coded tree-structured
*      broadcast.
*/
```

```
void Get_data1(
    float*  a_ptr  /* out */,
    float*  b_ptr  /* out */,
    int*    n_ptr  /* out */,
    int     my_rank /* in  */,
    int     p      /* in  */)
```

```
{
```

```
    int source;
    int dest;
    int stage;
```

```
    int Ceiling_log2(int x);
```

```
int I_receive( int stage, int my_rank, int* source_ptr);
int I_send(int stage, int my_rank, int p, int* dest_ptr);
void Send(float a, float b, int n, int dest);
void Receive(float* a_ptr, float* b_ptr, int* n_ptr, int source);

if (my_rank == 0)
{
    printf("Enter a, b, and n\n");
    scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
}
for (stage = 0; stage < Ceiling_log2(p); stage++)
    if (I_receive(stage, my_rank, &source))
        Receive(a_ptr, b_ptr, n_ptr, source);
    else if (I_send(stage, my_rank, p, &dest))
        Send(*a_ptr, *b_ptr, *n_ptr, dest);
} /* Get_data1 */
```

```

/*****
float Trap(
    float local_a /* in */,
    float local_b /* in */,
    int local_n /* in */,
    float h /* in */)
{

float integral; /* Store result in integral */
float x;
int i;

float f(float x); /* function we're integrating */

integral = (f(local_a) + f(local_b))/2.0;

```

```
x = local_a;  
for (i = 1; i <= local_n-1; i++)  
{  
    x = x + h;  
    integral = integral + f(x);  
}  
integral = integral*h;  
return integral;  
} /* Trap */
```

```
/**  
float f(float x)  
{  
    float return_val;  
    /* Calculate f(x). */  
    /* Store calculation in return_val. */  
    return_val = x*x;  
    return return_val;  
} /* f */
```

Collective Communication in MPI

■ Broadcast

- Communication pattern that involves all the processes in a communicator is a collective communication.
- Broadcast is a collective communication in which a single process sends the same data to every process in the communicator.
- Tree-structured communication is more efficient but complicated.
- Without knowing the details of the topology of the system, it is not quite to be sure how to implement the algorithms.
- In addition, it is not portable.

Collective Communication in MPI

■ Broadcast

- MPI has broadcast function, called MPI_Bcast, syntax as

```
Int MPI_Bcast(  
    void *                message    /*in/out**/,  
    int                   count      /*in */,  
    MPI_Datatype          datatype   /*in */,  
    int                   root       /* in */,  
    MPI_Comm              comm      /* in */)
```

Collective Communication in MPI

■ Broadcast

- It simply sends a copy of the data in message on the process with rank root to each process in the communicator, comm
- Broadcast message cannot be received with `MPI_Recv`.
- The count and datatype have the same function that they have in `MPI_Send` and `MPI_Recv`
- It is not point-to-point communication

```
/* get_data2.c -- Parallel Trapezoidal Rule.  
 * Uses 3 calls to MPI_Bcast to  
 *   distribute input data.  
 *  
 * Input:  
 *   a, b: limits of integration.  
 *   n: number of trapezoids.  
 * Output: Estimate of the integral from a to b of f(x)  
 *   using the trapezoidal rule and n trapezoids.  
 *  
 * Notes:  
 *   1. f(x) is hardwired.  
 *   2. the number of processes (p) should evenly divide  
 *       the number of trapezoids (n).  
 *  
 * See Chap. 5, pp. 69 & ff in PPMPI.  
 */
```

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

```
/* We'll be using MPI routines, definitions, etc. */
```

```
#include "mpi.h"
```

```
main(int argc, char** argv) {
```

```
    int    my_rank; /* My process rank      */
```

```
    int    p;      /* The number of processes */
```

```
    float  a;      /* Left endpoint          */
```

```
    float  b;      /* Right endpoint         */
```

```
    int    n;      /* Number of trapezoids  */
```

```
    float  h;      /* Trapezoid base length */
```

```
    float  local_a; /* Left endpoint my process */
```

```
    float  local_b; /* Right endpoint my process */
```

```
    int    local_n; /* Number of trapezoids for */
```

```
                /* my calculation          */
```

```
float    integral; /* Integral over my interval */
float    total;    /* Total integral          */
int      source;   /* Process sending integral */
int      dest = 0; /* All messages go to 0    */
int      tag = 0;
MPI_Status status;

void Get_data2(float* a_ptr, float* b_ptr, int* n_ptr, int my_rank);
float Trap(float local_a, float local_b, int local_n,
           float h); /* Calculate local integral */

/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

```
/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

Get_data2(&a, &b, &n, my_rank);

h = (b-a)/n; /* h is the same for all processes */
local_n = n/p; /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h. So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);
```

```
/* Add up the integrals calculated by each process */
if (my_rank == 0) {
    total = integral;
    for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
                MPI_COMM_WORLD, &status);
        total = total + integral;
    }
} else {
    MPI_Send(&integral, 1, MPI_FLOAT, dest,
            tag, MPI_COMM_WORLD);
}

/* Print the result */
if (my_rank == 0) {
    printf("With n = %d trapezoids, our estimate\n",
        n);
}
```

```
printf("of the integral from %f to %f = %f\n",  
      a, b, total);  
}
```

```
/* Shut down MPI */
```

```
MPI_Finalize();
```

```
} /* main */
```

```
/**/
```

```
/* Function Get_data2
```

```
* Reads in the user input a, b, and n.
```

```
* Input parameters:
```

```
* 1. int my_rank: rank of current process.
```

```
* 2. int p: number of processes.
```

```
* Output parameters:
```

- * 1. float* a_ptr: pointer to left endpoint a.
- * 2. float* b_ptr: pointer to right endpoint b.
- * 3. int* n_ptr: pointer to number of trapezoids.
- * Algorithm:
- * 1. Process 0 prompts user for input and
- * reads in the values.
- * 2. Process 0 sends input values to other
- * processes using three calls to MPI_Bcast.
- */

```
void Get_data2(  
    float* a_ptr /* out */,  
    float* b_ptr /* out */,  
    int* n_ptr /* out */,  
    int my_rank /* in */) {
```

```
if (my_rank == 0) {
    printf("Enter a, b, and n\n");
    scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
}
MPI_Bcast(a_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(b_ptr, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(n_ptr, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_data2 */
```

```
/***/
```

```
float Trap(
    float local_a /* in */,
    float local_b /* in */,
    int local_n /* in */,
    float h /* in */) {
```

```
float integral; /* Store result in integral */
float x;
int i;

float f(float x); /* function we're integrating */

integral = (f(local_a) + f(local_b))/2.0;
x = local_a;
for (i = 1; i <= local_n-1; i++) {
    x = x + h;
    integral = integral + f(x);
}
integral = integral*h;
return integral;
} /* Trap */
```

```
/**  
float f(float x) {  
    float return_val;  
    /* Calculate f(x). */  
    /* Store calculation in return_val. */  
    return_val = x*x;  
    return return_val;  
} /* f */
```

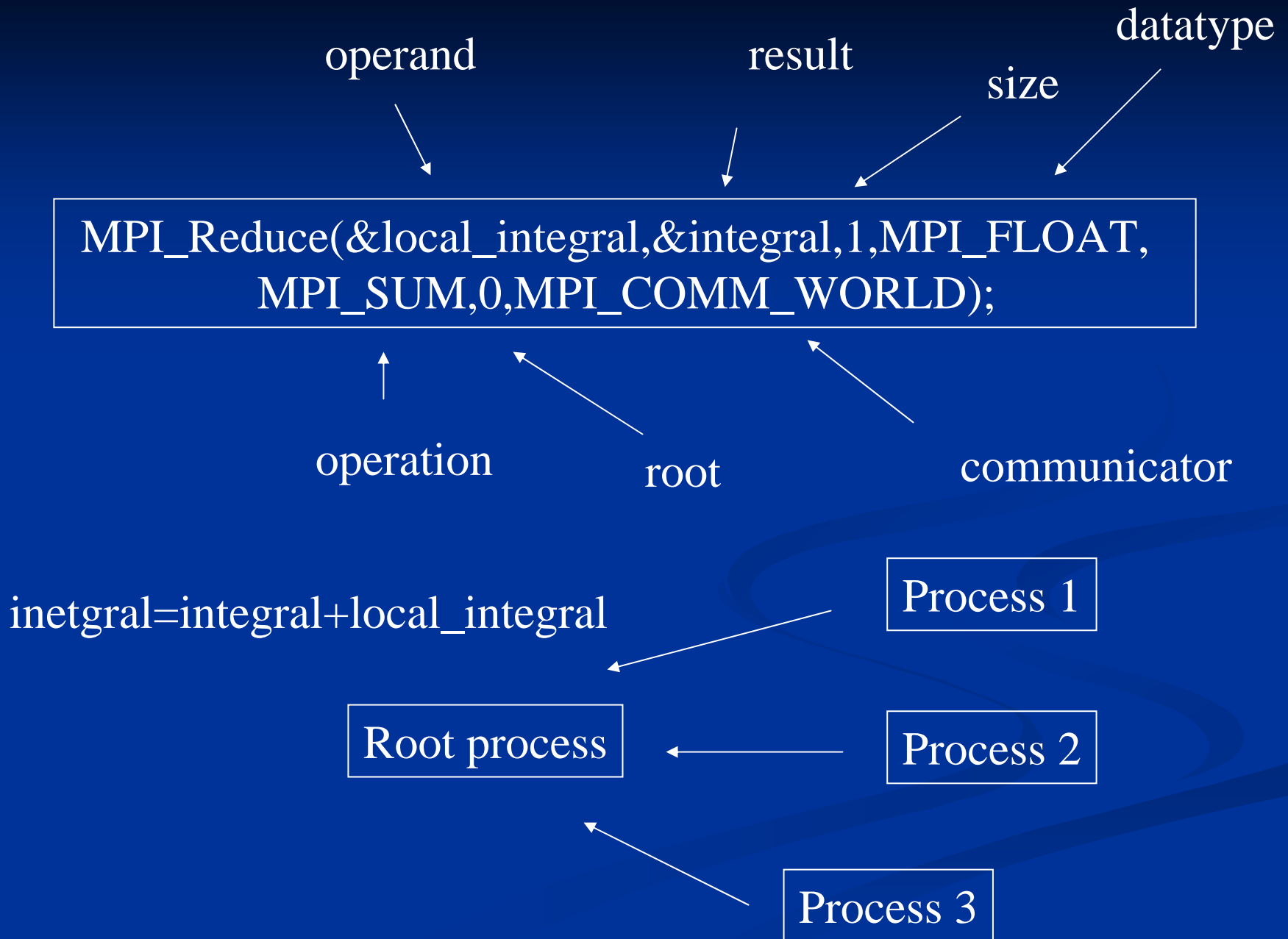
Collective Communication in MPI

■ Reduce

- We can use the reversing procedure to pass the result back to the process 0 explicitly.
- In collective communication, we introduce a reduction operation, in which all the processes in a communicator contribute data that is combining using a binary operation. The binary operations are addition, max, min, logical, and etc.
- The MPI function is called `MPI_Reduce()`

Int MPI_Reduce(
void * operand /*in */,
void * result /*out */,
int count /* in */,
MOI_Datatype datatype /* in */,
MPI_Op operator /* in */,
int root /* in */,
MPI_Comm comm /* in */)

Operation Name	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_MINLOG	Minimum and location of minimum



```
/* hpcsoft_intNC -- Parallel version of numerical integration  
with Newton-Cotes methods, which includes  
rectangle rule (one-point rule),  
trapezoidal rule (two-point rule),  
Simpson rule(three-point rule)
```

Designed and programmed by Dr. Jun Ni

```
*/
```

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

```
main(int argc, char** argv)  
{  
    int    my_rank;
```

```
int      p;
float    a = 0.0, b=1.0, h;
int      n = 2048;
int      mode=3; /* mode=1,2,3  rectangle,
                 trapezoidal, and Simpson */

float    local_a, local_b, local_h;
int      local_n;

float    local_integral, integral;
MPI_Status status;

/* function prototypes */
void Get_data02(float* a_ptr, float* b_ptr,
               int* n_ptr, int my_rank, int p, int *mode_ptr);
float rect(float local_a, float local_b, int local_n, float h);
float trap(float local_a, float local_b, int local_n, float h);
float simp(float local_a, float local_b, int local_n, float h);
```

```
/* MPI starts */
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

Get_data02(&a, &b, &n, my_rank, p, &mode);

h = (b-a)/n;
local_n = n/p;
local_a = a + my_rank*(b-a)/p;
local_b = a + (my_rank+1)*(b-a)/p;
local_h = h;

switch(mode)
{
case(1):
```

```
local_integral = rect(local_a, local_b, local_n, local_h);
    break;
case(2):
    local_integral = trap(local_a, local_b, local_n, local_h);
    break;
case(3):
    local_integral = simp(local_a, local_b, local_n, local_h);
}
```

```
MPI_Reduce(&local_integral,&integral,1,MPI_FLOAT,
           MPI_SUM,0,MPI_COMM_WORLD);
```

```
if(my_rank==0)
{
    if (mode==1)
        printf("Rectangle rule (0-point rule) is selected\n");
```

```
else if (mode==2)
    printf("Trapezoidal rule (2-point rule) is selected\n");
else /* defaulted */
    printf("Simpson rule (3-point rule) is selected\n");
printf("With n = %d, the total integral from %f to %f
    = %f\n",n, a,b,integral);
}
```

```
/* MPI finished */
    MPI_Finalize();
}
```

```
/******************************************************************/
```

```
/* Function Get_data02
```

- * Reads in the user input a, b, and n.
- * Input parameters:

- * 1. int my_rank: rank of current process.
- * 2. int p: number of processes.
- * Output parameters:
 - * 1. float* a_ptr: pointer to left endpoint a.
 - * 2. float* b_ptr: pointer to right endpoint b.
 - * 3. int* n_ptr: pointer to number of trapezoids.
- 3. int* mode_ptr: pointer to mode of rule of
- * Newton-Cotes methods
- * Algorithm:
 - * 1. Process 0 prompts user for input and
 - * reads in the values.
 - * 2. Process 0 sends input values to other
 - * processes using four calls to MPI_Bcast.
- */

```
void Get_data02(
    float* a_ptr /* out */,
    float* b_ptr /* out */,
    int* n_ptr /* out */,
```

```
int  my_rank /* in */,
int  p      /* in */,
int*  mode_ptr /* out */)
{

    MPI_Status status;

    if (my_rank == 0)
    {
        do
        {
            printf("Enter a, b, n(1024), and mode(1--rect, 2-- trap, 3-- simp):\n");
            scanf("%f %f %d %d", a_ptr, b_ptr, n_ptr, mode_ptr);
        } while (*mode_ptr<1 || *mode_ptr>3);

    }
}
```

```
MPI_Bcast(a_ptr,1,MPI_FLOAT,0,MPI_COMM_WORLD);
MPI_Bcast(b_ptr,1,MPI_FLOAT,0,MPI_COMM_WORLD);
MPI_Bcast(n_ptr,1,MPI_INT,0,MPI_COMM_WORLD);
MPI_Bcast(mode_ptr,1,MPI_INT,0,MPI_COMM_WORLD);

} /* Get_data02*/
```

```
float rect( float local_a, float local_b, int local_n, float local_h )
{
    float local_integral;
    float x;
    int i;

    float f(float x);

    local_integral = f(local_a);
```

```
x = local_a;
for (i = 1; i <= local_n-1; i++)
{
    x = x + local_h;
    local_integral += f(x);
}
local_integral *=local_h;
return local_integral;
}
```

```
float trap( float local_a, float local_b, int local_n, float local_h )
{
    float local_integral;
    float x;
    int i;

    float f(float x);
```

```
local_integral = f(local_a) + f(local_b);
  x = local_a;
  for (i = 1; i <= local_n-1; i++)
  {
    x = x + local_h;
    local_integral += 2.0*f(x);
  }
  local_integral *=local_h/2.0;
  return local_integral;
}
```

```
float simp( float local_a, float local_b, int local_n, float local_h )
{
  float local_integral;
  float x;
  int i;
```

```
float f(float x);

local_integral = f(local_a) + f(local_b);
x = local_a;
for (i = 1; i < local_n; i++)
{
    x = x + local_h;
    if (i % 2 == 0)          /* if i is even */
        local_integral = local_integral + 2 * f(x);
    else                    /* if i is odd */
        local_integral = local_integral + 4 * f(x);
}

local_integral *= local_h/3.0;
return local_integral;
}
```

```
float f(float x)
{
    return x*x;
/*
    return sin(x);
*/
}
```

- Example of parallel programming using collective communication in Fortran

Program Example1_3

```
c#####  
c This is an MPI example on parallel integration  
c It demonstrates the use of :  
c * MPI_Init  
c * MPI_Comm_rank  
c * MPI_Comm_size  
c * MPI_Bcast  
c * MPI_Reduce
```

```
c * MPI_SUM
```

```
c * MPI_Finalize
```

```
c
```

```
c#####
```

```
implicit none
```

```
integer n, p, i, j, ierr, master
```

```
real h, result, a, b, integral, pi
```

```
include "mpif.h" !! This brings in pre-defined MPI constants, ...
```

```
integer Iam, source, dest, tag, status(MPI_STATUS_SIZE)
```

```
real my_result
```

```
data master/0/
```

```
c**Starts MPI processes ...
```

```
call MPI_Init(ierr)
```

```
!! starts MPI
```

```
call MPI_Comm_rank(MPI_COMM_WORLD, Iam, ierr)
```

```
!! get current process id
```

```

call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)
    !! get number of processes
pi = acos(-1.0)  !! = 3.14159...
a = 0.0        !! lower limit of integration
b = pi*1./2.   !! upper limit of integration
dest = 0       !! define the process that computes the final result
tag = 123      !! set the tag to identify this particular job
if(Iam .eq. master) then
    print *, 'The requested number of processors =', p
    print *, 'enter number of increments within each process'
    read(*,*)n
endif

c**Broadcast "n" to all processes
call MPI_Bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
h = (b-a)/n/p  !! length of increment
my_result = integral(a,Iam,h,n)

```

```
write(*, "('Process ',i2,' has the partial result of',f10.6)")  
&    Iam,my_result  
    call MPI_Reduce(my_result, result, 1, MPI_REAL, MPI_SUM,  
&    dest, MPI_COMM_WORLD, ierr)
```

```
if(Iam .eq. master) then  
    print *, 'The result =',result  
endif
```

```
call MPI_Finalize(ierr)  
stop  
end
```

!! let MPI finish up ...

```
real function integral(a,i,h,n)  
implicit none  
integer n, i, j
```

```
real h, h2, aij, a
real fct, x
fct(x) = cos(x)           !! kernel of the integral
integral = 0.0           !! initialize integral
h2 = h/2.
do j=0,n-1                !! sum over all "j" integrals
  aij = a + (i*n + j)*h   !! lower limit of "j" integral
  integral = integral + fct(aij+h2)*h
enddo
return
end
```

Result:

```
% /bin/time mpirun -np 8 example1_3
```

```
The requested number of processors =      8  
enter number of increments within each process
```

```
20
```

```
Process 0 has the partial result of 0.195091
```

```
Process 7 has the partial result of 0.019215
```

```
Process 1 has the partial result of 0.187594
```

```
Process 4 has the partial result of 0.124363
```

```
Process 5 has the partial result of 0.092410
```

```
Process 6 has the partial result of 0.056906
```

```
Process 2 has the partial result of 0.172887
```

```
Process 3 has the partial result of 0.151537
```

```
The result = 1.000004
```

```
real 24.721
```

```
user 0.005
```

```
sys 0.053
```

```
% /bin/time mpirun -np 8 example1_3
The requested number of processors =      8
enter number of increments within each process
40
Process 0 has the partial result of 0.195091
Process 1 has the partial result of 0.187593
Process 4 has the partial result of 0.124363
Process 5 has the partial result of 0.092410
Process 6 has the partial result of 0.056906
Process 7 has the partial result of 0.019215
Process 3 has the partial result of 0.151537
Process 2 has the partial result of 0.172887
The result = 1.000001
real 4.381
user 0.005
sys 0.047
```

Result:

Enter the order of the vectors

3

Enter the first vector

1 2 3

Enter the second vector

3 4 6

The dot product is 29.000000

Another example: Vector's dot product

$$\mathbf{x} = (x_0, x_1, x_2, \dots, x_{n-1})^T$$

$$\mathbf{y} = (y_0, y_1, y_2, \dots, y_{n-1})^T$$

$$z = \mathbf{x} * \mathbf{y} = x_0 * y_0 + x_1 * y_1 + x_2 * y_2 + \dots + x_{n-1} * y_{n-1}$$

Another example: Vector's dot product

```
/* serial_dot.c -- compute a dot product on a single processor.
 *
 * Input:
 *   n: order of vectors
 *   x, y: the vectors
 *
 * Output:
 *   the dot product of x and y.
 *
 * Note: Arrays containing vectors are statically allocated.
 *
 * See Chap 5, p. 75 in PPMPI.
 */
```

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

```
#define MAX_ORDER 100
```

```
main() {
```

```
    float x[MAX_ORDER];
```

```
    float y[MAX_ORDER];
```

```
    int n;
```

```
    float dot;
```

```
    void Read_vector(char* prompt, float v[], int n);
```

```
    float Serial_dot(float x[], float y[], int n);
```

```
    printf("Enter the order of the vectors\n");
```

```
    scanf("%d", &n);
```

```
    Read_vector("the first vector", x, n);
```

```
Read_vector("the second vector", y, n);
    dot = Serial_dot(x, y, n);
    printf("The dot product is %f\n", dot);
} /* main */

/*****
void Read_vector(
    char* prompt /* in */,
    float v[] /* out */,
    int n /* in */) {
    int i;

    printf("Enter %s\n", prompt);
    for (i = 0; i < n; i++)
        scanf("%f", &v[i]);
} /* Read_vector */
```

```
/**  
float Serial_dot(  
    float x[] /* in */,  
    float y[] /* in */,  
    int n /* in */) {  
  
    int i;  
    float sum = 0.0;  
  
    for (i = 0; i < n; i++)  
        sum = sum + x[i]*y[i];  
    return sum;  
} /* Serial_dot */
```

```
/* parallel_dot.c -- compute a dot product of a vector distributed among
 * the processes. Uses a block distribution of the vectors.
 *
 * Input:
 *   n: global order of vectors
 *   x, y: the vectors
 *
 * Output:
 *   the dot product of x and y.
 *
 * Note: Arrays containing vectors are statically allocated. Assumes
 *   n, the global order of the vectors, is divisible by p, the number
 *   of processes.
 *
 * See Chap 5, pp. 75 & ff in PPMPI.
 */
```

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

#define MAX_LOCAL_ORDER 100

main(int argc, char* argv[])
{
    float local_x[MAX_LOCAL_ORDER];
    float local_y[MAX_LOCAL_ORDER];
    int n;
    int n_bar; /* = n/p */
    float dot;
    int p;
    int my_rank;
```

```
void Read_vector(char* prompt, float local_v[], int n_bar, int p,  
    int my_rank);  
float Parallel_dot(float local_x[], float local_y[], int n_bar);  
  
MPI_Init(&argc, &argv);  
MPI_Comm_size(MPI_COMM_WORLD, &p);  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
  
if (my_rank == 0)  
{  
    printf("Enter the order of the vectors (n>= %d):\n", p);  
    scanf("%d", &n);  
}  
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);  
n_bar = n/p;
```

```
Read_vector("the first vector", local_x, n_bar, p, my_rank);  
Read_vector("the second vector", local_y, n_bar, p, my_rank);
```

```
dot = Parallel_dot(local_x, local_y, n_bar);
```

```
if (my_rank == 0)  
    printf("The dot product is %f\n", dot);
```

```
MPI_Finalize();
```

```
}
```

```
/**  
*****  
*/
```

```
void Read_vector
```

```
(  
    char* prompt /* in */,  
    float local_v[] /* out */,
```

```
int  n_bar    /* in */,
int  p        /* in */,
int  my_rank  /* in */)
{
int i, q;
float temp[MAX_LOCAL_ORDER];
MPI_Status status;

if (my_rank == 0)
{
printf("Enter %s\n", prompt);
for (i = 0; i < n_bar; i++)
scanf("%f", &local_v[i]);
for (q = 1; q < p; q++)
{
for (i = 0; i < n_bar; i++)
```

```

scanf("%f", &temp[i]);
MPI_Send(temp, n_bar, MPI_FLOAT, q,
          0, MPI_COMM_WORLD);
}
}
else
{
MPI_Recv(local_v, n_bar, MPI_FLOAT,
          0, 0, MPI_COMM_WORLD,
          &status);
}
} /* Read_vector */
/*****
float Serial_dot
(
    float x[] /* in */,

```

```
float y[] /* in */,
int n /* in */)
{
int i;
float sum = 0.0;
for (i = 0; i < n; i++)
    sum = sum + x[i]*y[i];
return sum;
} /* Serial_dot */
```

```
/**/
```

```
float Parallel_dot (
float local_x[] /* in */,
float local_y[] /* in */,
int n_bar /* in */)
{
```

```
float local_dot;  
float dot = 0.0;  
float Serial_dot(float x[], float y[], int m);  
  
local_dot = Serial_dot(local_x, local_y, n_bar);  
MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,  
          MPI_SUM, 0, MPI_COMM_WORLD);  
return dot;  
} /* Parallel_dot */
```

```
mpirun -np 8 a.out
```

```
Enter the order of the vectors (n >= 8):
```

```
9
```

```
Enter the first vector
```

```
1 2 3 4 5 3 4 5 6
```

```
Enter the second vector
```

```
2 4 6 7 8 9 10 1 1
```

```
The dot product is 191.000000
```

Collective Communication in MPI

- Allreduce
 - result of reduction is returned to all the processes
 - there is no root
 - `MPI_Allreduce()`

Int MPI_Reduce(
void * operand /*in */,
void * result /*out */,
int count /* in */,
MOI_Datatype datatype /* in */,
MPI_Op operator /* in */,
int root /* in */,
MPI_Comm comm /* in */)

Modified parallel version for vector dot product

```
*  
* See Chap 5, pp. 76 & ff in PPMPI.  
*/  
#include <stdio.h>  
#include "mpi.h"  
  
#define MAX_LOCAL_ORDER 100  
  
main(int argc, char* argv[]) {  
    float local_x[MAX_LOCAL_ORDER];  
    float local_y[MAX_LOCAL_ORDER];  
    int n;  
    int n_bar; /* = n/p */  
    float dot;  
    int p;  
    int my_rank;
```

```
void Read_vector(char* prompt, float local_v[], int n_bar, int p,  
    int my_rank);  
float Parallel_dot(float local_x[], float local_y[], int n_bar);  
void Print_results(float dot, int my_rank, int p);  
  
MPI_Init(&argc, &argv);  
MPI_Comm_size(MPI_COMM_WORLD, &p);  
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);  
  
if (my_rank == 0) {  
    printf("Enter the order of the vectors (n >= %d):\n", p);  
    scanf("%d", &n);  
}  
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);  
n_bar = n/p;  
  
Read_vector("the first vector", local_x, n_bar, p, my_rank);  
Read_vector("the second vector", local_y, n_bar, p, my_rank);
```

```
dot = Parallel_dot(local_x, local_y, n_bar);

Print_results(dot, my_rank, p);

MPI_Finalize();
} /* main */

/*****/
void Read_vector(
    char* prompt /* in */,
    float local_v[] /* out */,
    int n_bar /* in */,
    int p /* in */,
    int my_rank /* in */) {
    int i, q;
    float temp[MAX_LOCAL_ORDER];
    MPI_Status status;
```

```
if (my_rank == 0) {
    printf("Enter %s\n", prompt);
    for (i = 0; i < n_bar; i++)
        scanf("%f", &local_v[i]);
    for (q = 1; q < p; q++) {
        for (i = 0; i < n_bar; i++)
            scanf("%f", &temp[i]);
        MPI_Send(temp, n_bar, MPI_FLOAT, q, 0, MPI_COMM_WORLD);
    }
} else {
    MPI_Recv(local_v, n_bar, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
            &status);
}
} /* Read_vector */
```

```
/**  
float Serial_dot(  
    float x[] /* in */,  
    float y[] /* in */,  
    int n /* in */) {  
  
    int i;  
    float sum = 0.0;  
  
    for (i = 0; i < n; i++)  
        sum = sum + x[i]*y[i];  
    return sum;  
} /* Serial_dot */
```

```
/**
float Parallel_dot(
    float local_x[] /* in */,
    float local_y[] /* in */,
    int n_bar /* in */) {

    float local_dot;
    float dot = 0.0;
    float Serial_dot(float x[], float y[], int m);

    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Allreduce(&local_dot, &dot, 1, MPI_FLOAT,
        MPI_SUM, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */
```

```
/**
 *
 */
void Print_results(
    float dot /* in */,
    int my_rank /* in */,
    int p /* in */) {
    int q;
    float temp;
    MPI_Status status;

    if (my_rank == 0) {
        printf("dot = \n");
        printf("Process 0 > %f\n", dot);
        for (q = 1; q < p; q++) {
            MPI_Recv(&temp, 1, MPI_FLOAT, q, 0, MPI_COMM_WORLD,
                &status);
        }
    }
}
```

```
printf("Process %d > %f\n", q, temp);
    }
} else {
    MPI_Send(&dot, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
}

} /* Print_results */
```

```
mpirun -np 8 a.out
```

```
Enter the order of the vectors (n >= 8):
```

```
9
```

```
Enter the first vector
```

```
1 2 3 4 5 6 7 8 9
```

```
Enter the second vector
```

```
1 3 5 7 9 11 13 15
```

```
dot =
```

```
Process 0 > 310.000000
```

```
Process 1 > 310.000000
```

```
Process 2 > 310.000000
```

```
Process 3 > 310.000000
```

```
Process 4 > 310.000000
```

```
Process 5 > 310.000000
```

```
Process 6 > 310.000000
```

```
Process 7 > 310.000000
```

Collective Communication in MPI

- Gather and Scatter
 - Example: Matrix products a vector

$$\mathbf{y}=\mathbf{A}*\mathbf{x}, \text{ where } \mathbf{A}=[a_{ij}]_{m \times n} \quad \text{and } \mathbf{x}=[x_i]_{n \times 1}$$

- Serial code

```
/* serial_mat_vect.c -- computes a matrix-vector
* product on a single processor.
*
* Input:
*   m, n: order of matrix
*   A, x: the matrix and the vector to be multiplied
*
* Output:
*   y: the product vector
*
* Note: A, x, and y are statically allocated.
*
* See Chap 5, p. 78 & ff in PPMPI.
*/
#include <stdio.h>

#define MAX_ORDER 100
```

```
typedef float MATRIX_T[MAX_ORDER][MAX_ORDER];
```

```
main() {
```

```
    MATRIX_T A;
```

```
    float x[MAX_ORDER];
```

```
    float y[MAX_ORDER];
```

```
    int m, n;
```

```
    void Read_matrix(char* prompt, MATRIX_T A, int m, int n);
```

```
    void Read_vector(char* prompt, float v[], int n);
```

```
    void Serial_matrix_vector_prod(MATRIX_T A, int m, int n,  
        float x[], float y[]);
```

```
    void Print_vector(float y[], int n);
```

```
    printf("Enter the order of the matrix (m x n)\n");
```

```
scanf("%d %d", &m, &n);
  Read_matrix("the matrix", A, m, n);
  Read_vector("the vector", x, m);
  Serial_matrix_vector_prod(A, m, n, x, y);
  Print_vector(y, n);
} /* main */
```

```
/******  
void Read_matrix(  
    char*    prompt /* in */,  
    MATRIX_T A    /* out */,  
    int     m     /* in */,  
    int     n     /* in */) {  
    int i, j;
```

```
printf("Enter %s\n", prompt);
for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
        scanf("%f", &A[i][j]);
} /* Read_matrix */
```

```
/******  
void Read_vector(  
    char* prompt /* in */,  
    float v[] /* out */,  
    int n /* in */) {  
    int i;
```

```
printf("Enter %s\n", prompt);
    for (i = 0; i < n; i++)
        scanf("%f", &v[i]);
} /* Read_vector */

/*****/
void Serial_matrix_vector_prod(
    MATRIX_T A /* in */,
    int m /* in */,
    int n /* in */,
    float x[] /* in */,
    float y[] /* out */) {

    int k, j;
```

```

for (k = 0; k < m; k++) {
    y[k] = 0.0;
    for (j = 0; j < n; j++)
        y[k] = y[k] + A[k][j]*x[j];
    }
} /* Serial_matrix_vector_prod */
/*****/
void Print_vector(
    float y[] /* in */,
    int n /* in */) {
    int i;

    printf("Result is \n");
    for (i = 0; i < n; i++)
        printf("%4.1f ", y[i]);
    printf("\n");
} /* Print_vector */

```

Enter the order of the matrix (m x n)

5

3

Enter the matrix

2 3 5 6 7

3 1 4 5 7

2 3 4 5 6

Enter the vector

1 3 2

1 2

Result is

21.0 33.0 23.0

Collective Communication in MPI

- Block-row or panel distribution
 - Simplest way
 - partition the matrix into blocks of consecutive rows or panels and assign a panel to each process

Process

Elements of A

0	a_{00}	a_{01}	a_{02}	a_{03}	\dots
	a_{10}	a_{11}	a_{12}	a_{13}	\dots
1	a_{20}	a_{21}	a_{22}	a_{23}	\dots
	a_{30}	a_{31}	a_{32}	a_{33}	\dots
2	a_{40}	a_{41}	a_{42}	a_{43}	\dots
	a_{50}	a_{51}	a_{52}	a_{53}	\dots
3	a_{60}	a_{61}	a_{62}	a_{63}	\dots
	a_{70}	a_{71}	a_{72}	a_{73}	\dots

Matrix A

vector x

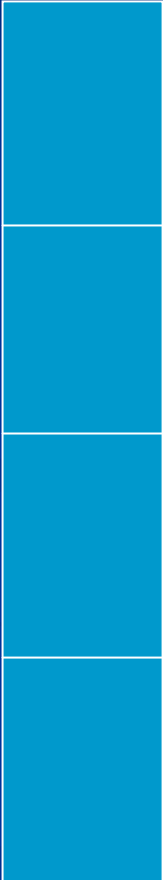
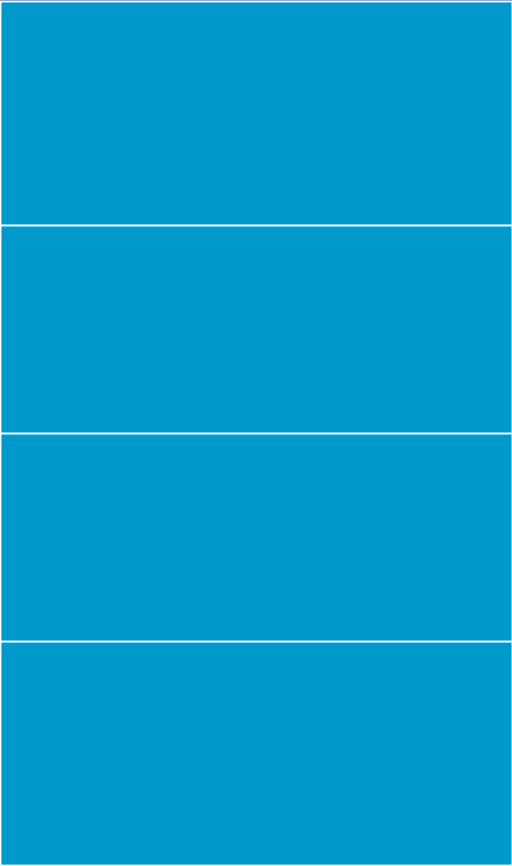
vector y

process 0

process 1

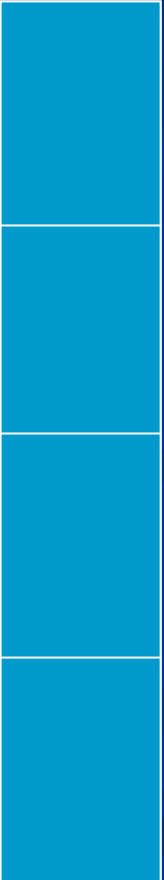
process 2

process 3



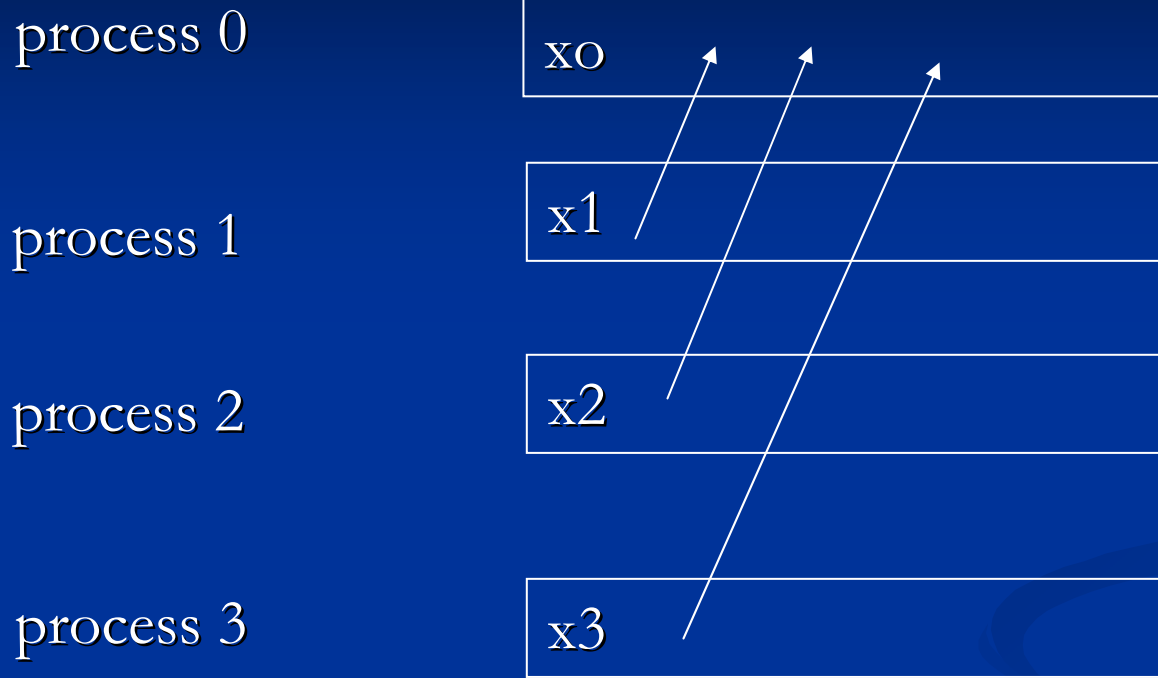
*

=

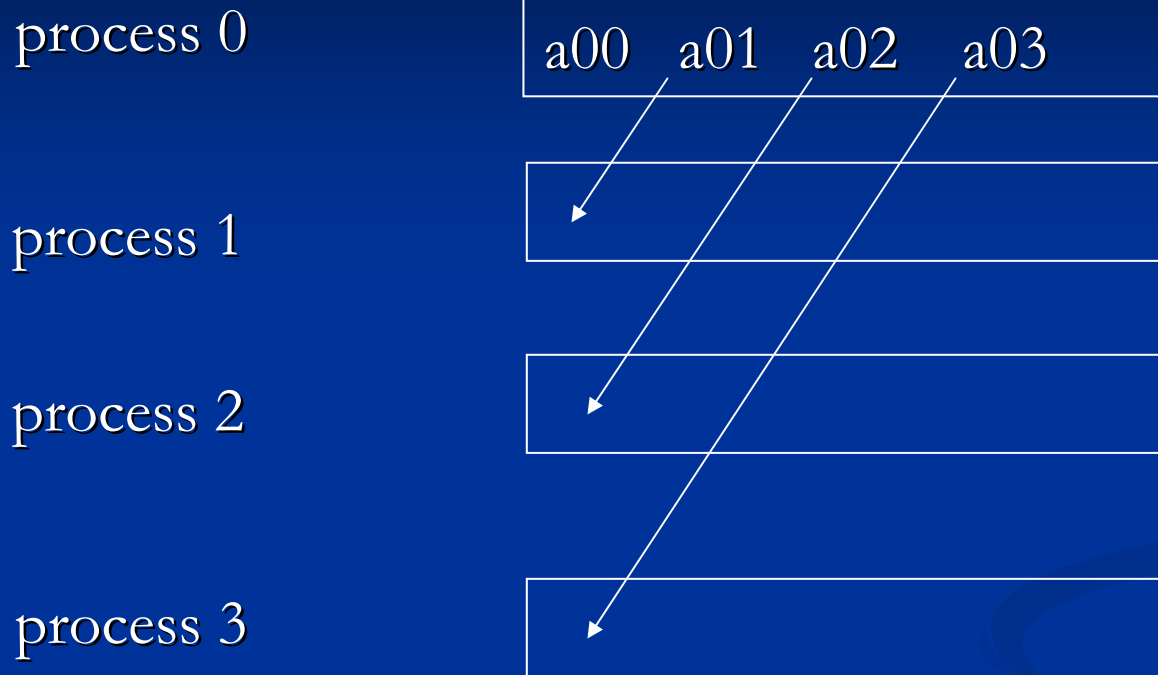


Collective Communication in MPI

- Use butterfly communication algorithm
 - gather all of x onto each process or scatter each row of A across the process
 - assign values of x (x_1, x_2, x_3, \dots) to process 0 (that collection is called **gather**)
 - assign values of a_{01} to process 1, a_{02} to process 2, ... (that collection is called **scatter**)
 - MPI provide both functions



Gather communication structure



Scatter communication structure

Collective Communication in MPI

■ MPI_Gather()

```
int MPI_Gather(  
    void *          send_data          /* in */,  
    int            send_count         /* in */,  
    MPI_Datatype   send_type          /* in */,  
    void *          recv_data         /* out */,  
    int            recv_count         /* in */,  
    MPI_Datatype   recv_type          /* in */,  
    int            root                /* in */,  
    MPI_Comm       comm               /* in */)
```

Collective Communication in MPI

- `MPI_Gather` collects the data referenced by `send_data` from each process in the communicator, and stores the data in process rank order on the process root in the memory referenced by `recv_data`.