Introduction to MPI: Part II

Summary of Part I:

To write working MPI (Message Passing Interface) parallel programs you only need:

- MPI_Init
- MPI_Comm_rank
- MPI_Comm_size
- MPI_Send
- MPI_Recv
- MPI_Finalize

Outline

In this talk will introduce two more advanced MPI concepts:

- collective communications
- non-blocking communications

Example: Numerical integration

Trapezoid rule

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(x_0) + f(x_n)) + h\sum_{i=1}^{n-1} f(x_i)$$

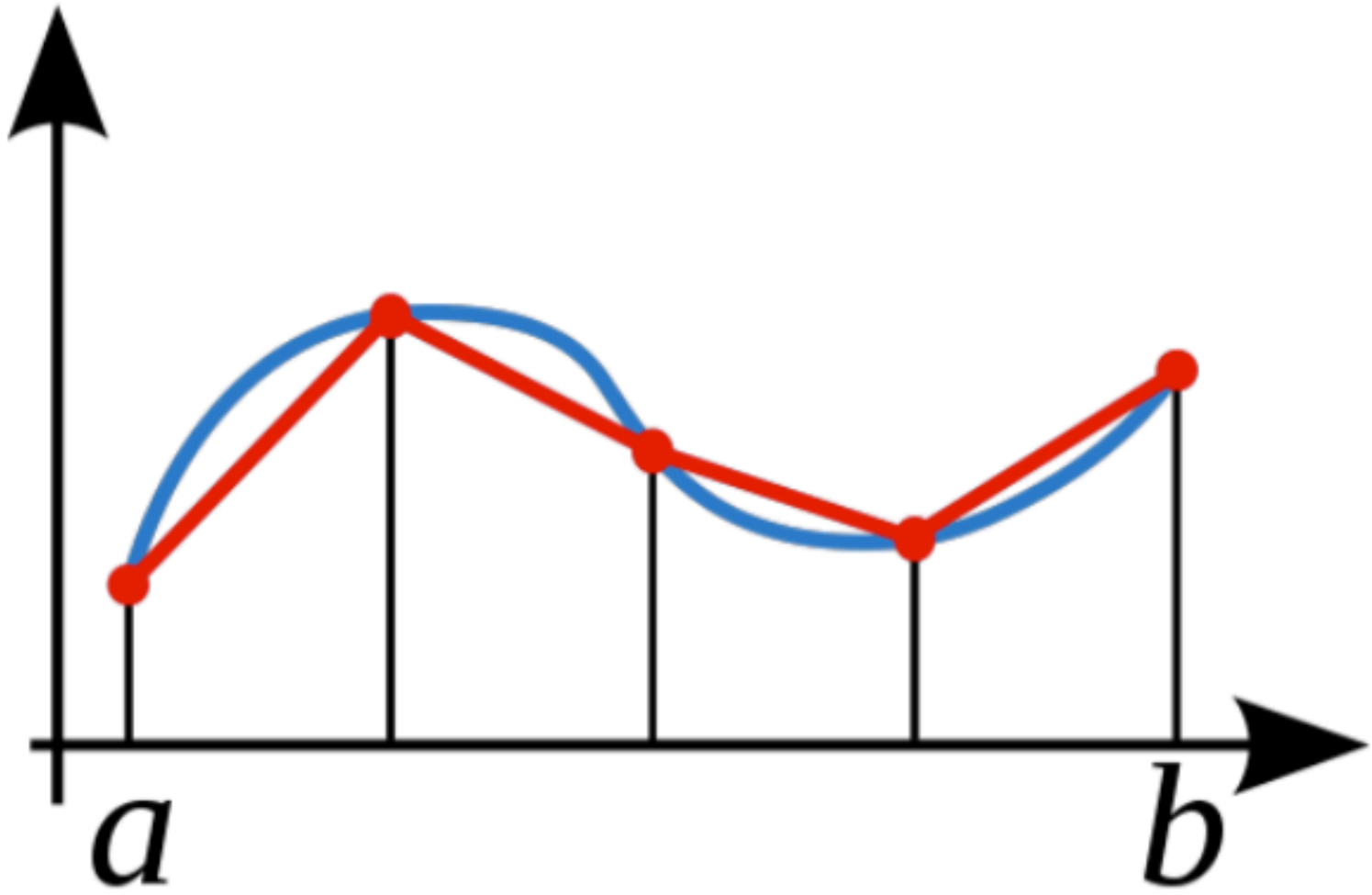
where $h = (b-a)/n$, $x_i = a + ih$

Given p processes, each process can work on n/p intervals

Note: for simplicity will assume n/p is an integer

process	interval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
...	...
p-1	$[a + (p-1)\frac{n}{p}h, b]$

Example: Numerical integration with trapezoid rule



Parallel trapezoid

Assume $f(x) = x^2$

Of course could have chosen any desired (integrable) function here.

Write function f(x) in

```
/* func.c */

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

Serial trapezoid rule

```c
/* traprule.c */

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

float Trap(float a, float b, int n, float h) {
  float integral;   /* Store result in integral  */
  float x;
  int i;

  integral = (f(a) + f(b))/2.0;
  x = a;
  for ( i = 1; i <= n-1; i++ )
    {
      x = x + h;
      integral = integral + f(x);
    }
  return integral*h;
}
```

Parallel trapezoid rule

```
/* trap.c -- Parallel Trapezoidal Rule
 *
 * Input: None.
 * Output:  Estimate of the integral from a to b of f(x)
 *    using the trapezoidal rule and n trapezoids.
 *
 * Algorithm:
 *    1.  Each process calculates "its" interval of
 *        integration.
 *    2.  Each process estimates the integral of f(x)
 *        over its interval using the trapezoidal rule.
 *    3a. Each process != 0 sends its integral to 0.
 *    3b. Process 0 sums the calculations received from
 *        the individual processes and prints the result.
 *
 *        The number of processes (p) should evenly divide
 *        the number of trapezoids (n = 1024)
 */
```

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

Main program

```
int main(int argc, char** argv) {
  int          my_rank;    /* My process rank           */
  int          p;          /* The number of processes   */
  float        a = 0.0;    /* Left endpoint             */
  float        b = 1.0;    /* Right endpoint            */
  int          n = 1024;   /* 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      */
  float        integral;   /* Integral over my interval */
  float        total=-1;   /* Total integral            */
  int          source;     /* Process sending integral  */
  int          dest = 0;   /* All messages go to 0      */
  int          tag = 0;    MPI_Status  status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &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. */
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);
      printf("PE %d <- %d, %f\n", my_rank,source,
                integral);
      total = total + integral;
      }
  }
```

```
else
    {
    printf("PE %d -> %d, %f\n", my_rank, dest, integral);
    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);
    }

  MPI_Finalize();
  return 0;
}
```

Collective communications

Introduction

Collective communication involves all the processes in a communicator

We will consider:

- Broadcast
- Reduce
- Gather
- Scatter

Reason for use: convenience and speed

Broadcast

Broadcast: a single process sends data to all processes in a communicator

```
int MPI_Bcast(void *buffer , int count,
        MPI_Datatype datatype, int root, MPI_Comm comm)
```

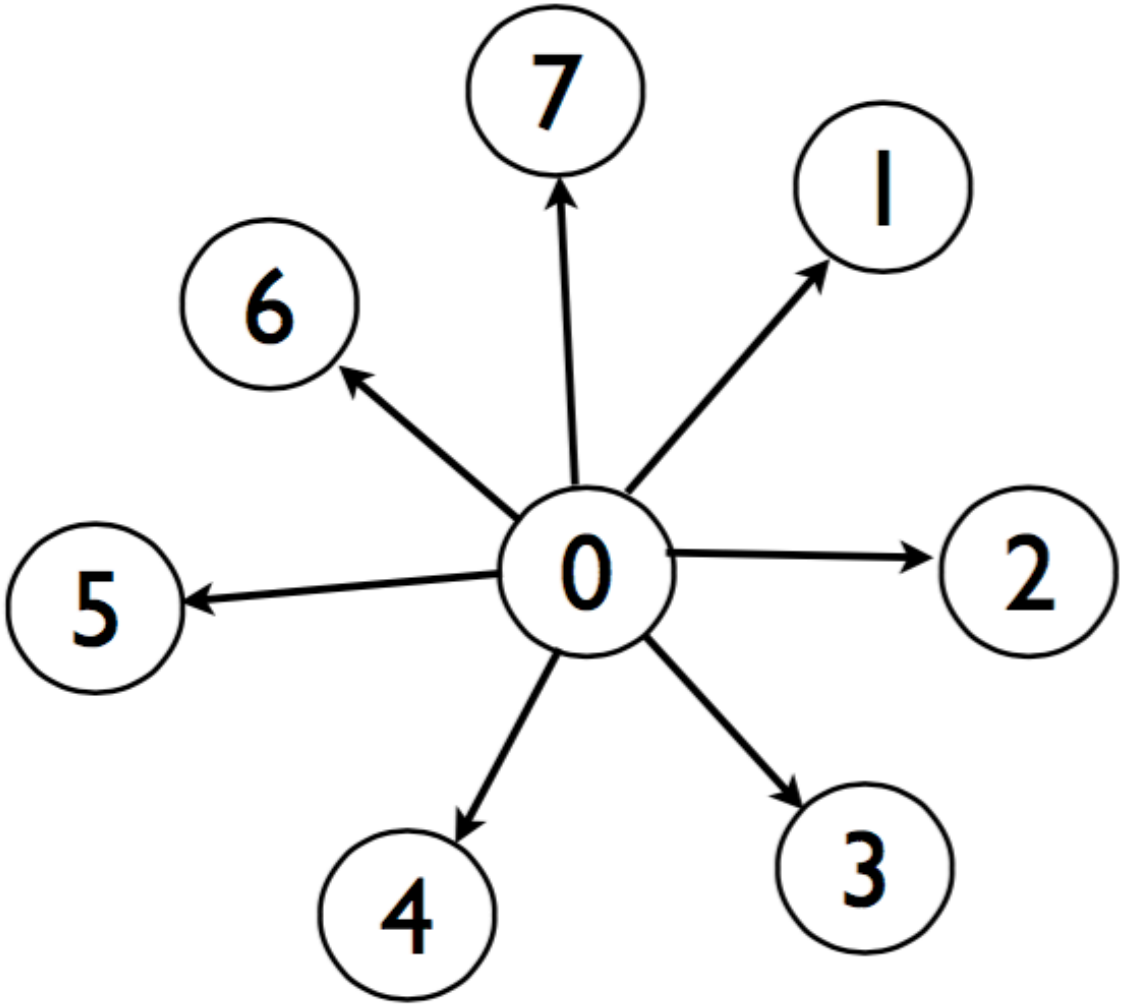
- buffer starting address of buffer (in/out)
- count number of entries in buffer
- datatype data type of buffer root
- rank - rank of broadcast root
- comm - communicator

MPI_Bcast

- MPI_Bcast sends a copy of the message on process with rank root to each process in comm
- must be called in each process
- data is sent in root and received by all other processes
- buffer is 'in' parameter in root and 'out' parameter in the rest of processes
- cannot receive broadcasted data with MPI_Recv

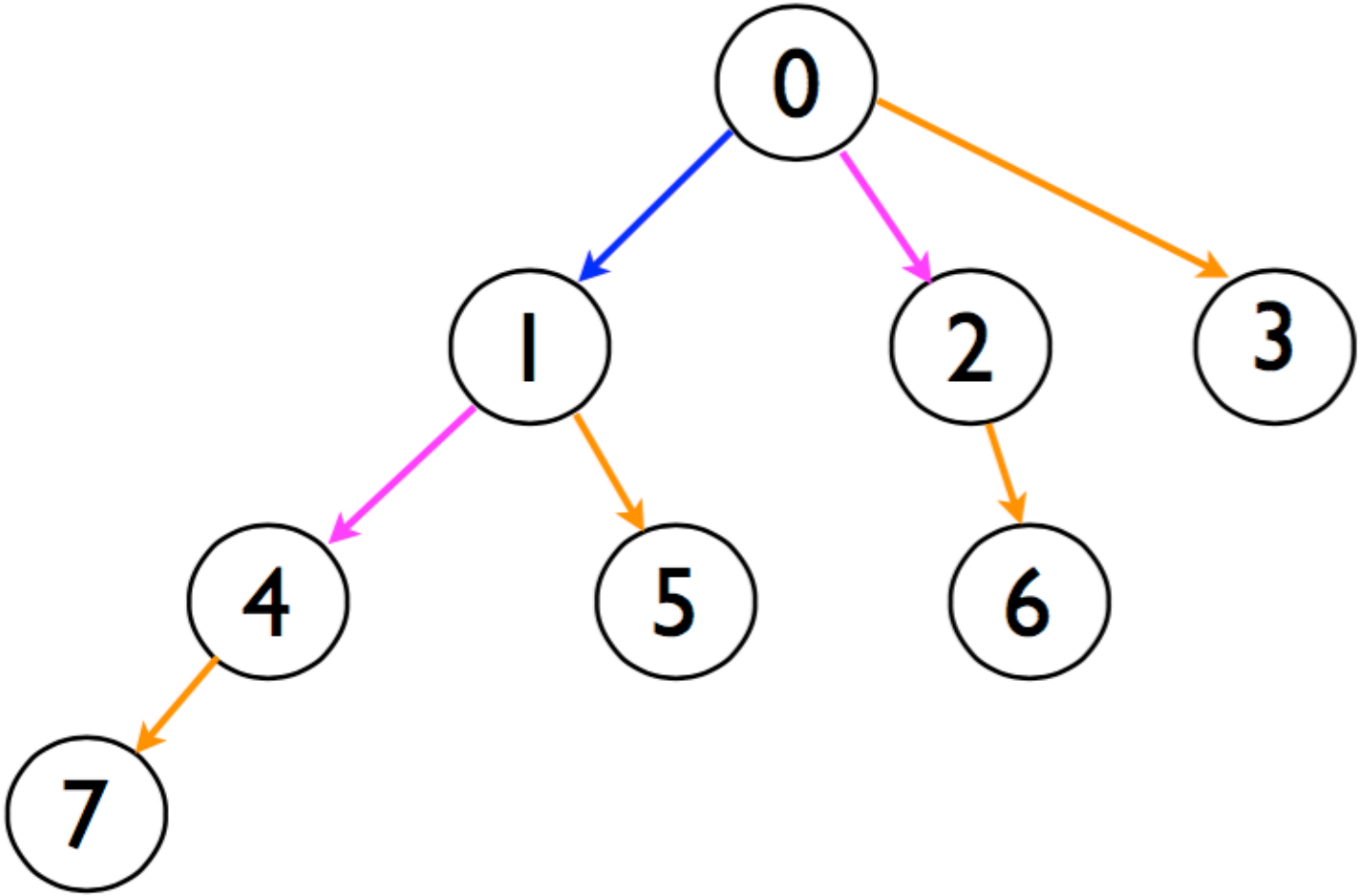
Broadcast - poor implementation

- Serial, 7 time steps needed



Broadcast - actual, parallel implementation

- Parallel, 3 time steps needed



Example: reading and broadcasting data

Code adapted from P. Pacheco, PP with MPI

```
/* getdata2.c */

/* Function Get_data
 * 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.
 */
```

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

void Get_data(float* a_ptr, float* b_ptr, int* n_ptr,
              int my_rank)
{
  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);
}
```

Reduce

Data from all processes are combined using a binary operation

```
int MPI_Reduce( void *sendbuf, void *recvbuf, int count,
        MPI Datatype datatype, MPI_Op op,
        int root, MPI_Comm comm)
```

- sendbuf - address of send buffer
- recvbuf - address of receive buffer, significant only at root
- count - number of entries in send buffer
- datatype - data type of elements in send buffer
- op - reduce operation; predefined, e.g. MPI_MIN, MPI_SUM, or user defined
- root - rank of root process
- comm - communicator
- Must be called in all processes in a communicator, BUT result only available in root process

Example - trapezoid with reduce

Code adapted from P. Pacheco, PP with MPI

```
/* redtrap.c */
#include <stdio.h>
#include "mpi.h"
extern void Get_data2(float* a_ptr, float* b_ptr,
                      int* n_ptr, int my_rank);
extern float Trap(float local_a, float local_b,
                  int local_n, float h);

int main(int argc, char** argv)
{
  int          my_rank, p;
  float        a, b, h;
  int          n;
  float        local_a, local_b, local_n;
  float        integral;  /* Integral over my interval */
  float        total;     /* Total integral            */

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
```

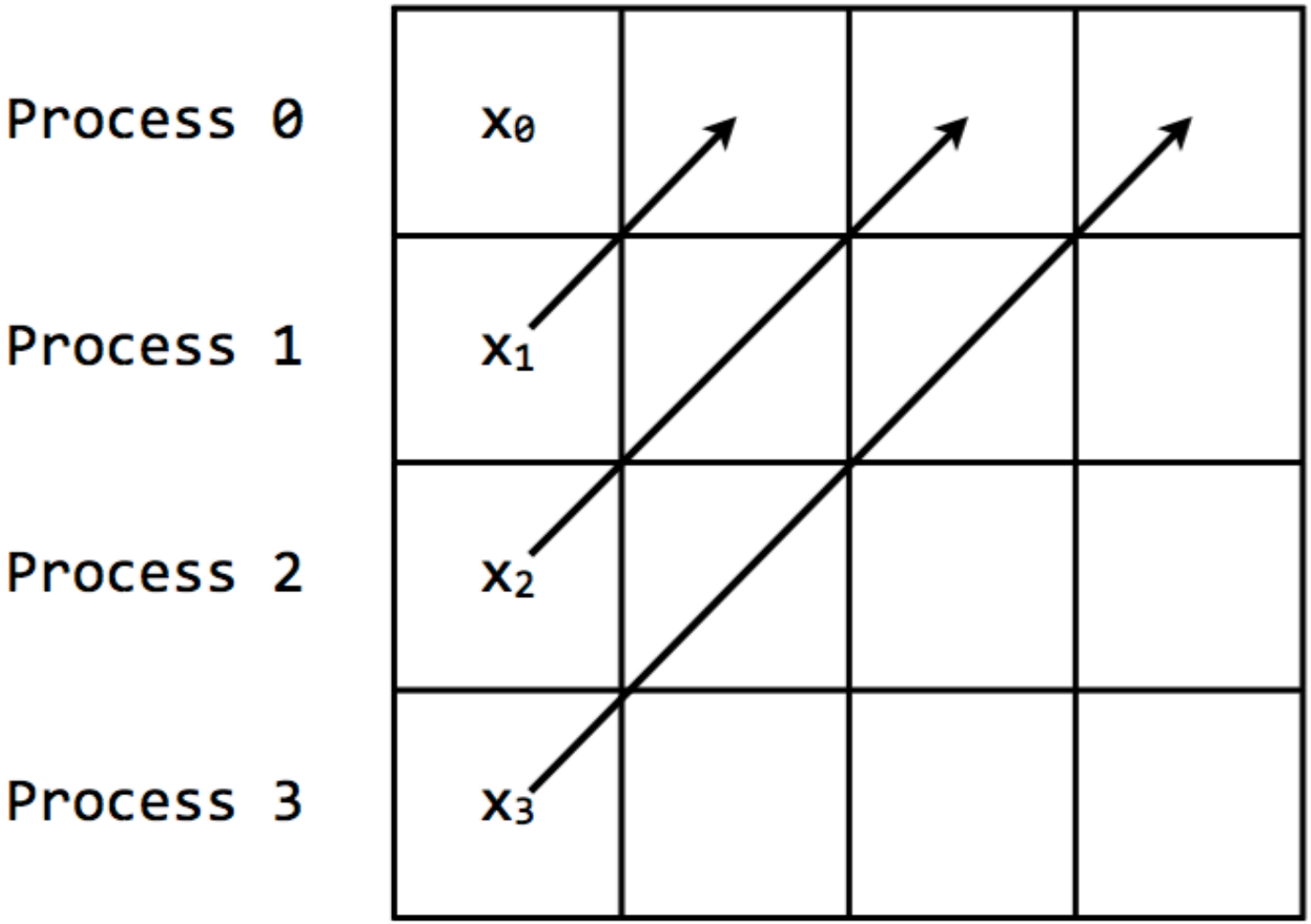
```
Get_data2(&a, &b, &n, my_rank);

  h = (b-a)/n;
  local_n = n/p;

  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 */
  MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
             MPI_SUM, 0, MPI_COMM_WORLD);

  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);
    }

  MPI_Finalize();
  return 0;
}
```

Allreduce

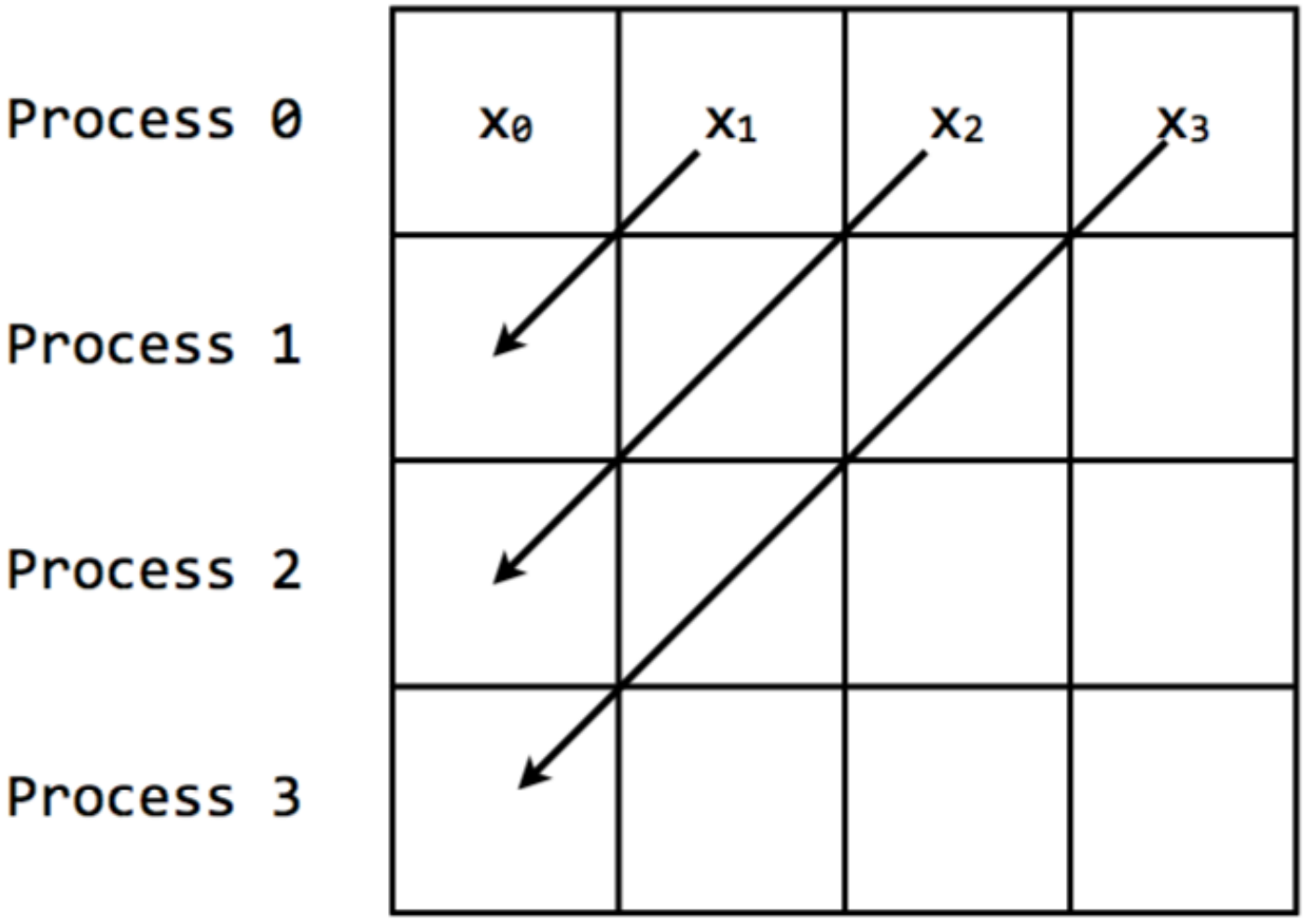
```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
          int count, MPI_Datatype datatype,
          MPI_Op op, MPI_Comm comm)
```

Similar to MPI Reduce except the result is returned to the receive buffer of each process in comm

Gather



Scatter



Summary for Collective Communications

- Amount of data sent must match amount of data received
- Blocking versions only
- No tags: calls are matched according to order of execution
- A collective function can return as soon as its participation is complete

Understanding Communications

Concepts

Buffering

Safe programs

Blocking communications

Non-blocking communications

Blocking communication

- Assume that process 0 sends data to process 1
- In a blocking communication, the sending routine returns only after the buffer it uses is ready to be reused
- Similarly, in process 1, the receiving routine returns after the data is completely stored in its buffer
- Blocking send and receive: MPI_Send and MPI_Recv
- MPI_Send: sends data; does not return until the data have been safely stored away so that the sender is free to access and overwrite the send buffer
- The message might be copied directly into the matching receive buffer, or it might be copied into a temporary system buffer.
- MPI Recv: receives data; it returns only after the receive buffer contains the newly received message

Message structure

Each message consists of two parts:

1. Data transmitted
2. Envelope, which contains:

- rank of the receiver
- rank of the sender
- a tag
- a communicator

Receive does not need to know the exact size of data arriving but it must have enough space in its buffer to store it.

Buffering

Suppose we have

```
if (rank==0)
   MPI_Send(sendbuf,...,1,...)
if (rank==1)
   MPI_Recv(recvbuf,...,0,...)
```

These are blocking communications, which means they will not return until the arguments to the functions can be safely modified by subsequent statements in the program.

Assume that process 1 is not ready to receive

There are 3 possibilities for process 0:

1. stops and waits until process 1 is ready to receive
2. copies the message at sendbuf into a system buffer (can be on process 0, process 1 or somewhere else) and returns from MPI_Send
3. fails

Buffering

As long as buffer space is available, (2) is a reasonable alternative

An MPI implementation is permitted to copy the message to be sent into internal storage, but it is not required to do so

What if not enough space is available?

- In applications communicating large amounts of data, there may not be enough memory (left) in buffers
- Until receive starts, no place to store the send message
- Practically, (1) results in a serial execution

A programmer should not assume that the system provides adequate buffering

Example

Consider a program executing

Process 0	Process 1
MPI_Send to process 1	MPI_Send to process 0
MPI_Recv from process 1	MPI_Recv from process 0

Such a program may work in many cases, but it is certain to fail for message of some size that is large enough

Possible solutions

- Ordered send and receive - make sure each receive is matched with send in execution order across processes
- This matched pairing can be difficult in complex applications. An alternative is to use **MPI_Sendrecv**. It performs both send and receive such that if no buffering is available, no deadlock will occur
- Buffered sends. MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least left in buffer) via **MPI_Bsend**
- Nonblocking communication. Initiated, then program proceeds while the communication is ongoing, until a check that communication is completed later in the program. **Important**: must make certain data not modified until communication has completed.

MPI_Sendrecv

```
int MPI_Sendrecv( void *sendbuf, int sendcount,
        MPI_Datatype sendtype, int dest, int sendtag,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype, int source, int recvtag,
        MPI_Comm comm, MPI_Status *status )
```

Combines:

- MPI_Send - send data to process with rank=dest
- MPI_Recv - receive data from process with rank=source

Source and dest may be the same

MPI_Sendrecv may be matched by ordinary MPI_Send or MPI_Recv

Performs Send and Recv, and organizes them in such a way that even in systems with no buffering program won't deadlock

MPI_Sendrecv_replace

```
int MPI_Sendrecv_replace( void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag,
        MPI_Comm comm, MPI_Status *status )
```

- Sends and receives using a single buffer
- Process sends contents of buf to dest, then replaces them with data from source
- If source=dest, then function swaps data between process which calls it and process source

Safe programs

- A program is safe if it will produce correct results **even if the system provides no buffering**.
- Need safe programs for portability.
- Most programmers expect the system to provide some buffering, hence many unsafe MPI programs are around.
- Write safe programs using matching send with receive, MPI_Sendrecv, allocating own buffers, nonblocking operations

Nonblocking communications

- nonblocking communications are useful for **overlapping** communication with computation, and ensuring safe programs
- a nonblocking operation requests the MPI library to perform an operation (when it can)
- nonblocking operations do not wait for any communication events to complete
- nonblocking send and receive: return almost immediately
- can safely modify a send (receive) buffer only after send (receive) is completed
- "wait" routines will let program know when a nonblocking operation is done

MPI_Isend

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,
        int dest, int tag , MPI_Comm comm,
        MPI_Request *request)
```

- buf - starting address of buffer
- count - number of entries in buffer
- datatype - data type of buffer
- dest - rank of destination
- tag - message tag
- comm - communicator
- request - communication request (out)

MPI_Irecv

```
int MPI_Irecv(void* buf, int count, MPI_Datatype datatype,
        int source , int tag ,MPI_Comm comm,
        MPI_Request *request)
```

- buf - starting address of buffer (out)
- count - number of entries in buffer
- datatype - data type of buffer
- source - rank of source
- tag - message tag
- comm - communicator
- request - communication request (out)

Wait routines

```
int MPI_Wait (MPI_Request *request , MPI_Status *status)
```

Waits for MPI_Isend or MPI_Irecv to complete

- request - request (in), which is out parameter in MPI_Isend and MPI_Irecv
- status - status output, replace with MPI_STATUS_IGNORE if not used

Wait routines

Other routines include:

- MPI_Waitall waits for all given communications to complete
- MPI_Waitany waits for any of given communications to complete
- MPI_Test tests for completion of send or receive, i.e returns true if completed, false otherwise
- MPI_Testany tests for completion of any previously initiated communication in the input list

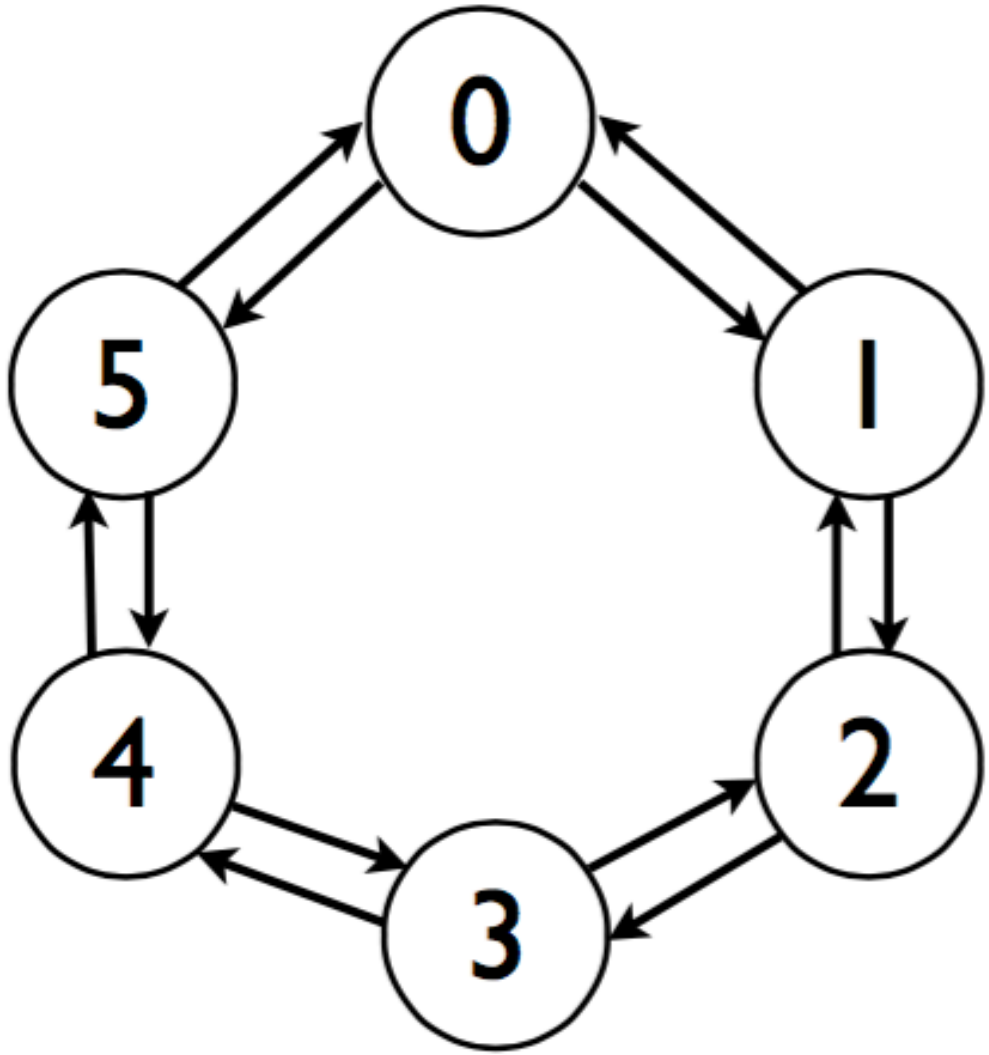
MPI_Waitall

```
int MPI_Waitall (int count,
                 MPI_Request array_of_requests[],
                 MPI_Status array_of_statuses[])
```

Waits for all given communications to complete

- count - list length
- array_of_requests - each request is an output parameter in MPI_Isend and MPI_Irecv
- array_of_statuses - array of status objects, replace with MPI_STATUSES_IGNORE if never used

Example: Communication between processes in ring topology



Example - Communication between processes in ring topology

- With blocking communications it is not possible to write a simple code to accomplish this data exchange.
- For example, if we have MPI_Send first in all processes, program will get stuck as there will be no matching MPI_Recv to send data to
- Nonblocking communication avoids this problem

Ring topology example

```c
/* nonb.c */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char *argv[]) {
  int numtasks, rank, next, prev,
    buf[2], tag1=1, tag2=2;

  tag1=tag2=0;
  MPI_Request reqs[4];
  MPI_Status stats[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Ring topology example

```
prev = rank-1;
next = rank+1;
if (rank == 0)              prev = numtasks - 1;
if (rank == numtasks - 1) next = 0;
MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1,
          MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2,
          MPI_COMM_WORLD, &reqs[1]);
MPI_Isend(&rank, 1, MPI_INT, prev, tag2,
          MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1,
          MPI_COMM_WORLD, &reqs[3]);
MPI_Waitall(4, reqs, stats);
printf("Task %d communicated with tasks %d & %d\n",
       rank,prev,next);
MPI_Finalize();
return 0; }
```

MPI_Test

```
int MPI_Test ( MPI_Request *request,int *flag,
        MPI_Status *status)
```

- request - (input) communication handle, which is output parameter in MPI_Isend and MPI_Irecv
- flag - true if operation completed (logical)
- status - status output, replace with MPI_STATUS_IGNORE if not used

MPI_Test - can be used to test if communication completed, can be called multiple times, in combintation with nonblocking send/receive, to control execution flow between processes

Non-deterministic workflow

```
if (my_rank == 0){
(... do computation ...)
/* send signal to other processes */
 for (proc = 1; proc < nproc; proc++){
  MPI_Send(&buf, 1, MPI_INT, proc, tag, MPI_COMM_WORLD)
} }
else{
/* initiate nonblocking receive */
 MPI_Irecv(&buf,1, MPI_INT, 0, tag, MPI_COMM_WORLD,&reqs);
 for(i = 0; i <= Nlarge; i++){
/* test if Irecv completed */
  MPI_Test(&reqs, &flag, &status);
      if(flag){
         break;     /* terminate loop */
      }
      else{
         (... do computation ...)
      } } }
```

Summary for Nonblocking Communications

- nonblocking send can be posted whether a matching receive has been posted or not
- send is completed when data has been copied out of send buffer
- nonblocking send can be matched with blocking receive and vice versa
- communications are initiated by sender
- a communication will generally have lower overhead if a receive buffer is already posted when a sender initiates a communication

Further MPI features to explore

- Communicators
- Topologies
- User defined datatypes
- Parallel input/output operations
- Parallel algorithms
- Parallel libraries (eg. Scalapack)