Message Passing Interface (MPI)

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

Programming with MPI

What is MPI?

- M P I = Message Passing Interface
- MPI is a specification for the developers and users of message passing libraries
  - By itself, it is NOT a library - but rather the specification of what such a library should be.
- MPI addresses the message-passing parallel programming model:
  - data is moved from the address space of one process to that of another process through cooperative operations on each process.

Programming Model

- 1980s - early 1990s
  - MPI was designed for distributed memory architectures, which were popular at that time
- Later Shared memory SMPs were combined over networks
- Today, MPI runs on virtually any hardware platform:
  - Distributed Memory
  - Shared Memory
  - Hybrid
    - However the programming model clearly remains a distributed memory model regardless of the underlying physical architecture
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Reasons for Using MPI

- Standardization
  - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms
- Portability
  - There is little or no need to modify your source code when you port your application to a different platform that supports the MPI standard
- Performance Opportunities
  - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms
- Functionality
  - There are over 430 routines defined in MPI-3. Most MPI programs can be written using a dozen or less routines
- Availability
  - A variety of implementations are available

Installation

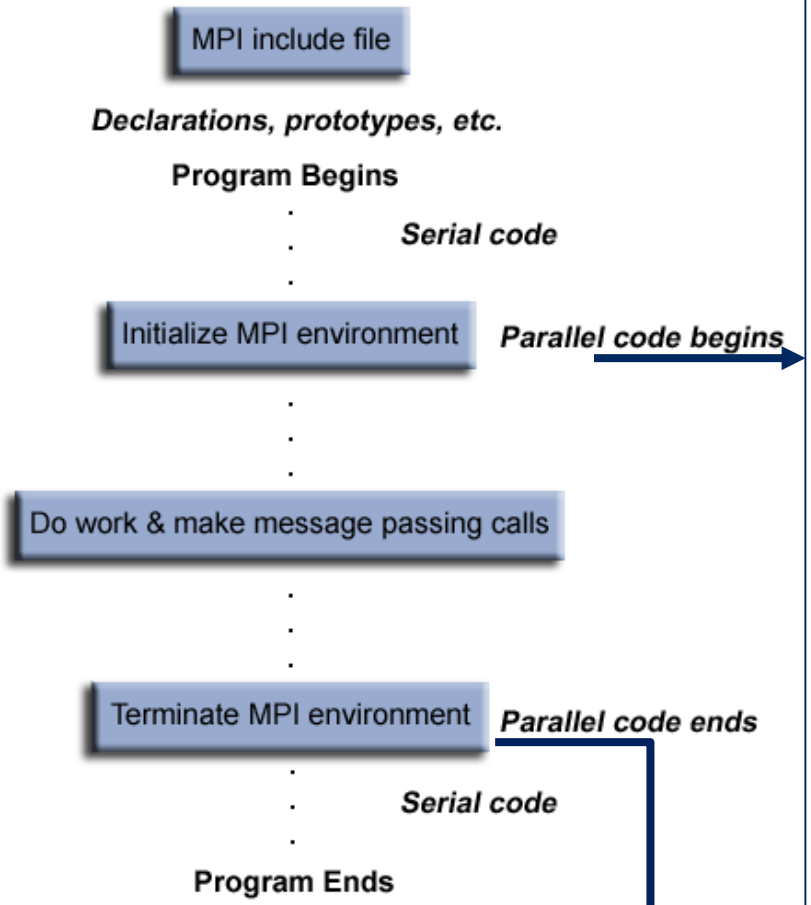
- There are several implementations of MPI
  - MPICH, OpenMPI, LAMMPI etc

Compilation and Execution

- Compile
  - mpicc −g −Wall −o mpi_hello mpi_hello.c
  - mpicc is a script that's a wrapper for the C compiler
- Execute
  - To run the program
    - mpiexec −n <number of processes> ./mpi hello OR
    - mpirun –np <number of processes> ./mpi hello

General MPI Program Structure

BITS Pilani




```
#include <mpi.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COM
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_CO
  }
else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_CO
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COM
  }
MPI_Finalize();
}
```

General MPI Program Structure

BITS Pilani

- MPI_Init
  - MPI_Init tells the MPI system to do all of the necessary setup
  - no other MPI functions should be called before the program calls MPI_Init

```
int MPI Init(
int* argc /* in/out */,
char** argv /* in/out */);
```

```
#include <mpi.h>

int main (int argc, char *argv[])
{
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COM
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COM
  }
else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COM
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COM
  }
MPI_Finalize();
}
```

Communicators and Groups

- MPI uses objects called communicators and groups to define which collection of processes may communicate with each other
  - Most MPI routines require you to specify a communicator as an argument
  - MPI_COMM_WORLD is the predefined communicator that includes all of MPI processes
- Rank
  - Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero
  - Used by the programmer to specify the source and destination of messages. Often used conditionally by the application to control program execution (if rank=0 do this / if rank=1 do that).

Knowing Environment

BITS Pilani

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - MPI_Comm_size reports the number of processes.
  - MPI_Comm_rank reports the rank, a number between 0 and size-1, identifying the calling process

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

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

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

SPMD Programs

- A single program is written so that different processes carry out different actions, and this is achieved by simply having the processes branch on the basis of their process rank
  - this approach to parallel programming is called single program, multiple data, or SPMD
- No separate program for each process
- We try to write programs that will run with any number of processes, because we usually don't know in advance the exact resources available to us
  - We might have a 20-core system available today, but tomorrow we might have access to a 500-core system

innovate achieve lead

BITS Pilani
Pilani Campus

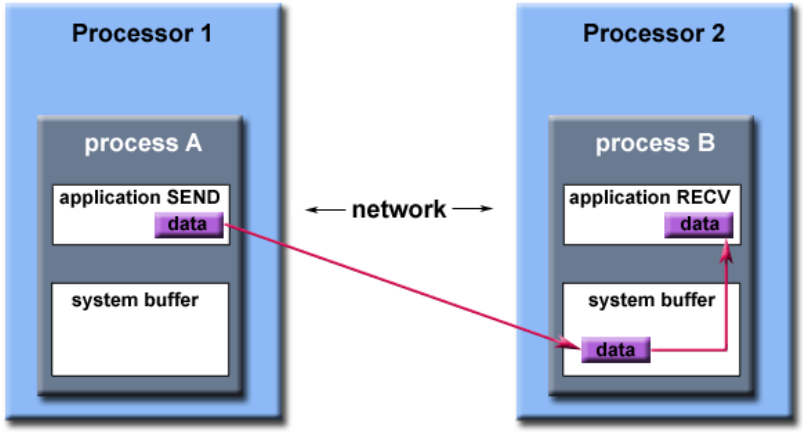
Point-to-Point Communication

Point to Point Communication

- MPI point-to-point operations involve message passing between two, and only two, different MPI tasks
  - One task is performing a send operation and the other task is performing a matching receive operation
- There are different types of send and receive routines used for different purposes
  - For example:
    - Synchronous send, Blocking send / blocking receive, Non-blocking send / non-blocking receive, Buffered send Combined send/receive
  - Any type of send routine can be paired with any type of receive routine.
- MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived

Buffering

- In a perfect world, every send operation would be perfectly synchronized with its matching receive
  - This is rarely the case. Somehow or other, the MPI implementation must be able to deal with storing data when the two tasks are out of sync
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases
  - Typically, a system buffer area is reserved to hold data in transit



Path of a message buffered at the receiving process

System buffer space is opaque to the programmer and managed entirely by the MPI library

Blocking vs. Non-blocking

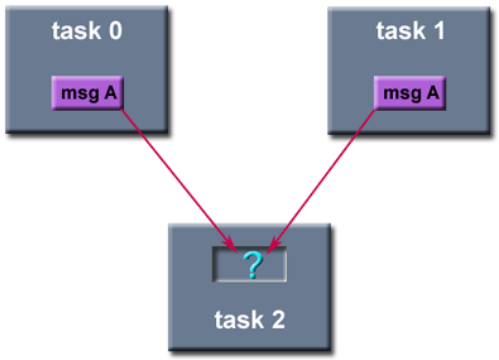
- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode
- Blocking
  - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse
    - Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Blocking vs. Non-blocking

- Non-blocking
  - Non-blocking send and receive will return almost immediately
    - do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message
  - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
  - It is unsafe to modify the application buffer (your variable space) until it is known that operation was performed by the library
    - There are "wait" routines used to do this.
  - Non-blocking communications are used to overlap computation with communication and enhance possible performance gains

Order and Fairness

- MPI guarantees that messages will not overtake each other
  - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2
    - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
- Order rules do not apply if there are multiple threads participating in the communication operations
- MPI does not guarantee **fairness** - it's up to the programmer to prevent "operation starvation“
  - Example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.



MPI Message Passing Routines

BITS Pilani

- Blocking send
  - MPI_Send(buffer,count,type,dest,tag,comm)
- Non-blocking sends
  - MPI_Isend(buffer,count,type,dest,tag,comm,request)
- Blocking receive
  - MPI_Recv(buffer,count,type,source,tag,comm,status)
- Non-blocking receive
  - MPI_Irecv(buffer,count,type,source,tag,comm,request)

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

```
int MPI_Recv(
void* msg_buf_p /* out */,
int buf_size /* in */,
MPI_Datatype buf_type /* in */,
int source /* in */,
int tag /* in */,
MPI_Comm communicator /* in */,
MPI_Status* status p /* out */);
```

MPI Message Passing Routines

BITS Pilani

- MPI_Send
  - The first three arguments, msg_buf_p, msg_size, and msg_type, determine the contents of the message
  - The remaining arguments, dest, tag, and communicator, determine the destination of the message
    - msg_buf_p is a pointer to the block of memory containing the contents of the message. msg_type and msg_size tells the system how much data being sent.
    - To send “Hello\0”, msg_type is MPI_CHAR and msg_size is number of such characters (=6)

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

```
int MPI_Recv(
void* msg_buf_p /* out */,
int buf_size /* in */,
MPI_Datatype buf_type /* in */,
int source /* in */,
int tag /* in */,
MPI_Comm communicator /* in */,
MPI_Status* status p /* out */);
```

MPI Message Passing Routines

- MPI_Send
  - dest specifies the rank of the process that should receive the message
  - tag, is a nonnegative integer
    - It can be used to distinguish between messages
    - Suppose process 1 is sending floats to process 0. Some of the floats should be printed, while others should be used in a computation. Tag 0 can be used for printing, tag 1 can be used for computation

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

```
int MPI_Recv(
void* msg_buf_p /* out */,
int buf_size /* in */,
MPI_Datatype buf_type /* in */,
int source /* in */,
int tag /* in */,
MPI_Comm communicator /* in */,
MPI_Status* status p /* out */);
```

MPI Message Passing Routines

BITS Pilani

- MPI_Recv
  - The first six arguments to MPI_Recv correspond to the first six arguments of MPI_Send
  - Status indicates the source of the message and the tag of the message, the actual number of bytes received

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

```
int MPI_Recv(
void* msg_buf_p /* out */,
int buf_size /* in */,
MPI_Datatype buf_type /* in */,
int source /* in */,
int tag /* in */,
MPI_Comm communicator /* in */,
MPI_Status* status p /* out */);
```

MPI Message Passing Routines

BITS Pilani

- Message Matching
  - Message sent by q is received by r if
    - recv_comm=send_comm
    - send_tag=recv_tag
    - dest=r
    - source=q
  - In addition
    - send_type=recv_type
    - msg_size=buf_size

```
int MPI_Send(
void* msg_buf_p /* in */,
int msg_size /* in */,
MPI_Datatype send_type /* in */,
int dest /* in */,
int send_tag /* in */,
MPI_Comm send_comm /* in */);
```

```
int MPI_Recv(
void* msg_buf_p /* out */,
int buf_size /* in */,
MPI_Datatype recv_type /* in */,
int source /* in */,
int recv_tag /* in */,
MPI_Comm recv_comm /* in */,
MPI_Status* status p /* out */);
```

MPI Message Passing Routines

BITS Pilani

- It can happen that one process is receiving messages from multiple processes, and the receiving process doesn't know the order in which the other processes will send the messages
- If process 0 has given work to p1 … pn, and waiting for replies using MPI_Recv, what should be the source value?
  - Which process will complete first is not known
- MPI provides MPI_ANY_SOURCE, MPI_ANY_TAG

```
int MPI_Send(
void* msg_buf_p /* in */,
int msg_size /* in */,
MPI_Datatype send_type /* in */,
int dest /* in */,
int send_tag /* in */,
MPI_Comm send_comm /* in */);
```

```
int MPI_Recv(
void* msg_buf_p /* out */,
int buf_size /* in */,
MPI_Datatype recv_type /* in */,
int source /* in */,
int recv_tag /* in */,
MPI_Comm recv_comm /* in */,
MPI_Status* status p /* out */);
```

MPI Message Passing Routines

BITS Pilani

- The status_p argument
  - a receiver can receive a message without knowing
    - the amount of data in the message,
    - the sender of the message, or
    - the tag of the message
  - how can the receiver find out these values?
  - The MPI_type MPI_Status is a struct with at least the three members MPI_SOURCE, MPI_TAG, and MPI_ERROR
  - The amount of data NOT directly accessible to the application program
    - can be retrieved with a call to MPI_Get_count

```
int MPI_Send(
void* msg_buf_p /* in */,
int msg_size /* in */,
MPI_Datatype send_type /* in */,
int dest /* in */,
int send_tag /* in */,
MPI_Comm send_comm /* in */);
```

```
int MPI_Recv(
void* msg_buf_p /* out */,
int buf_size /* in */,
MPI_Datatype recv_type /* in */,
int source /* in */,
int recv_tag /* in */,
MPI_Comm recv_comm /* in */,
MPI_Status* status_p /* out */);
```

Blocking Message Passing Routines

- MPI_Send
  - Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse
- MPI_Recv
  - Receive a message and block until the requested data is available in the application buffer in the receiving task.
- MPI_Ssend
  - Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message
- MPI_Sendrecv
  - Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message
- MPI_Wait
  - MPI_Wait blocks until a specified non-blocking send or receive operation has completed.

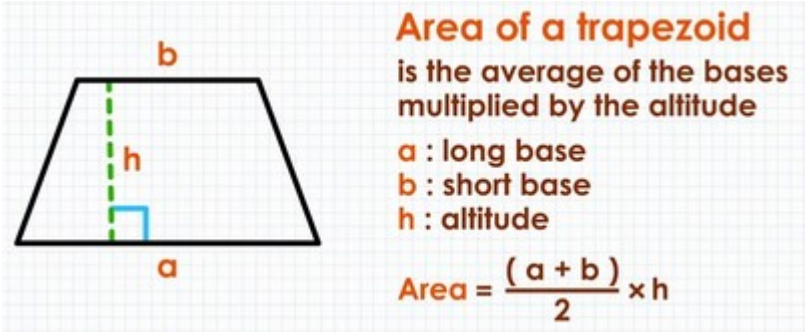
Non-blocking Message Passing Routines

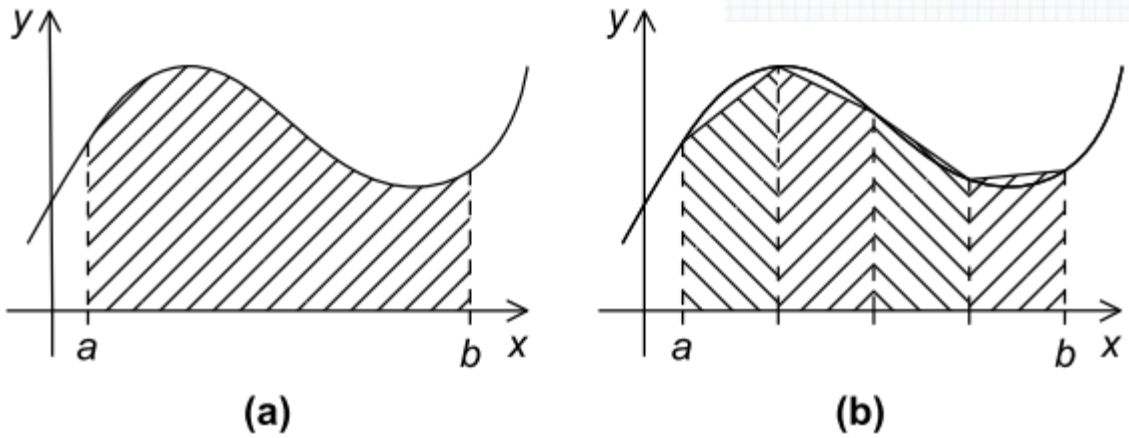
- MPI_Isend
  - Identifies an area in memory to serve as a send buffer. Processing continues immediately without waiting for the message to be copied out from the application buffer
- MPI_Irecv
  - Identifies an area in memory to serve as a receive buffer. Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer.
- MPI_Issend
  - Non-blocking synchronous send. Similar to MPI_Isend(), except MPI_Wait() or MPI_Test() indicates when the destination process has received the message

Example: The Trapezoidal Rule

BITS Pilani

- Trapezoidal rule is used to approximate the area between the graph of a function, y = f(x), two vertical lines, and the x-axis
- Divide the interval on the x-axis into n equal subintervals.
  - Then approximate the area lying between the graph and each subinterval by a trapezoid



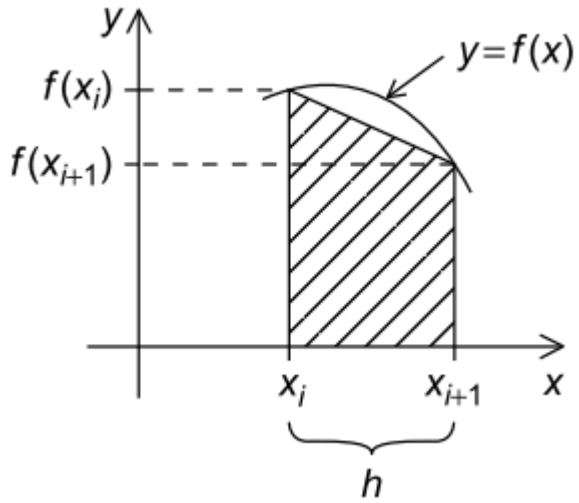


The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids

Example: The Trapezoidal Rule

BITS Pilani

- Area of one trapezoid $=(h/2)[f(x_i)+f(x_{i+1})]$ where h is $x_{i+1} - x_i$
- If the vertical lines bounding the region are x = a and x = b, then $h =(b-a)/n$
  - divide into n equal subintervals
  - $x_0= a$, $x_1= a+h$, $x_2= a + 2h$, ... , $x_{n-1}= a + (n-1)h$, $x_n= b$
  - Area= $f(x_0)+f(x_1)(h/2) + f(x_1)+f(x_2)(h/2) + \ldots. + f(x_{n-1})+f(x_n)\ (h/2)$
  - $h(f(x_0)/2+f(x_1)+f(x_2)+\ldots\ldots+f(x_{n-1})+f(x_n)/2)$



One trapezoid

Pseudo-code for a Serial program

- Area
  - $h(f(x_0)/2+f(x_1)+f(x_2)+\ldots\ldots+f(x_{n-1})+f(x_n)/2)$

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

Parallelizing the Trapezoidal Rule

- We can design a parallel program using four basic steps
  1. Partition the problem solution into tasks.
  2. Identify the communication channels between the tasks.
  3. Aggregate the tasks into composite tasks.
  4. Map the composite tasks to cores.
- In the partitioning phase, we try to identify as many tasks as possible
  - For the trapezoidal rule, we might identify two types of tasks: one type is finding the area of a single trapezoid, and the other is computing the sum of these areas
- The communication channels will join each of the tasks of the first type to the single task of the second type

Parallelizing the Trapezoidal Rule

- So how can we aggregate the tasks and map them to the cores?
  - More the trapezoids we use, the more accurate our estimate will be
    - That is, we should use many trapezoids, and we will use many more trapezoids than cores.
  - Thus, we need to aggregate the computation of the areas of the trapezoids into groups.
    - split the interval [a,b] up into comm sz subintervals
    - If comm sz evenly divides n, the number of trapezoids, we can simply apply the trapezoidal rule with n / comm sz trapezoids to each of the comm sz subintervals
  - To finish, we can have one of the processes, say process 0, add the estimates.

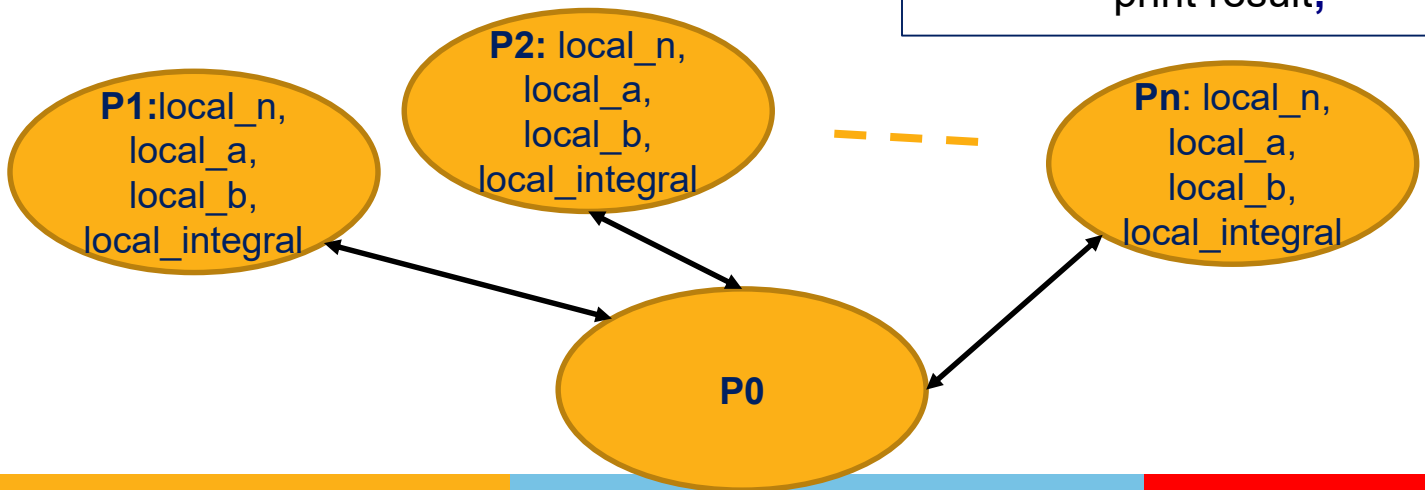
Program Outline

BITS Pilani

- P0 adds all local_integrals received from all other processes

```
Get a, b, n;
h = (b−a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b,
local_n, h);
```

```
if (my_rank != 0)
        Send local_integral to process 0;
else{ /* my rank == 0 */
  total_integral = local_integral;
 for (proc = 1; proc < comm_sz; proc++){
  Receive local_integral from proc;
  total_integral += local_integral;
 }
 }
if(my_rank == 0)
        print result;
```



```
int main(void) {
 int my_rank, comm_sz, n = 1024, local_n;
 double a = 0.0, b = 3.0, h, local_a, local_b;
 double local_int, total_int;
 int source;

 MPI_Init(NULL, NULL);
 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
 MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
 h = (b-a)/n; /* h is the same for all processes */
 local_n = n/comm_sz; /* So is the number of trapezoids */
 local_a = a + my_rank*local_n*h;
 local_b = local_a + local_n*h;
 local_int = Trap(local_a, local_b, local_n, h);
 if (my_rank != 0) {
        MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
 } else {
         total_int = local_int;
         for (source = 1; source < comm_sz; source++) {
         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
         total_int += local_int;
 } }
 if (my_rank == 0) {
        printf("With n = %d trapezoids, our estimate\n", n);
        printf("of the integral from %f to %f = %.15e\n",
        a, b, total_int);
 }
 MPI_Finalize();
 return 0;
 } /*main*/
```

BITS Pilani

Dealing With I/O

- Unlike output, most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin
- In order to write MPI programs that can use scanf, we need to branch on process rank, with process 0 reading in the data and then sending it to the other processes

```
09 if (my_rank == 0) {
10          printf("Enter a, b, and n\n");
11          scanf("%lf %lf %d", a,b,n);
12 for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16 }
17 } else { /* my rank != 0 */
18          MPI_Recv(a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
20          MPI_Recv(b, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS IGNORE);
22          MPI_Recv(n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS IGNORE);
24 }
```

Dealing with I/O

- MPI implementations allow all the processes in MPI_COMM_WORLD full access to stdout and stderr
- if multiple processes are attempting to write to stdout the order in which the processes' output appears will be unpredictable

References

- https://computing.llnl.gov/tutorials/mpi/
- “An introduction to Parallel Programming”, Peter S. Pacheco (Chapter 3)

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You