COSC 407
Intro to Parallel Computing

MPI: Intro, Point-to-Point

Outline

Previous Live Lecture:

- CUDA – Performance Revisited

Today:

- Distributed Memory Programming
- Intro to MPI
- MPI: Point to Point communication
- Example: Area under the curve in MPI
- Dealing with I/O

Next Lecture:

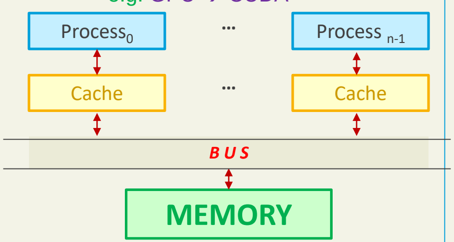
- MPI: Collective communication

Remember: Parallel Programing

- Computer architectures classified by memory organizations:

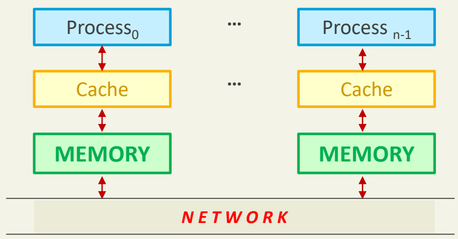
Multiprocessor with Shared Memory

Cores share same memory.
- Multiple cores on a single chip
  - e.g. CPU → OpenMP
- Multiple cores on separate chips
  - e.g. GPU → CUDA



Multi-node with Distributed Memory

Cores don't share memory
- But they are connected on a network



Distributed Memory Programming

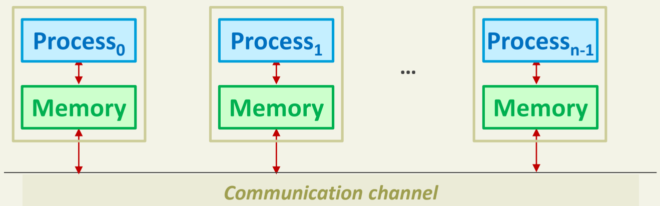
- Several models, including:
  - Hadoop
    - Uses MapReduce that implements *split-apply-combine* strategy
      - 1) Data is distributed using Hadoop Distributed File System (HDFS)
      - 2) A certain operation is applied to each data element (this is mapping).
      - 3) Reduction can then be applied over the results.
    - Has good fault tolerance (system doesn't crash when a fault is present)
    - Suffers from low performance
  - Spark
    - Was developed to overcome the efficiency problems of Hadoop.
    - May be used in the same scenarios as Hadoop.

MPI
(Message Passing Interface)
- May be used to develop ANY parallel code that runs on multiple machines
  - For example, for implementing dataflow type parallel programs such as Spark, but MPI might not be the best choice then.
    - More than one study shows that MPI/OpenMP outperforms Spark in terms of processing speed, but Spark shows better data management.

- source:1 J. L.Reyes-Ortiz (2015), Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf
  source:2 D Kumar (2017), Performance Evaluation of Apache Spark Vs MPI: A Practical Case Study on Twitter Sentiment Analysis

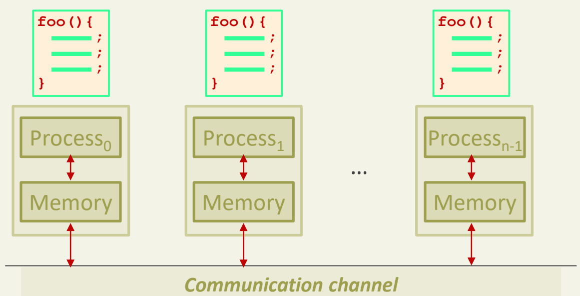
Basics of Message Passing: Overview

- We have ***n processes*** (or ***nodes***), each has a ***unique ID*** (***rank***)
- The nodes are interconnected by ***channels*** through which nodes can send messages to each other.
  - The nodes do not share memory (usually!)
  - *A* ***communicator*** defines the collection of processes that can communicate with each other.
- Each node can `send` and `receive`.
  - This can be simple ***point-to-point*** send/receive operation, or ***collective*** communication



SPMD

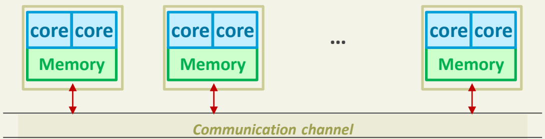
- Most message passing programs use SPMD (**Single-Program Multiple-Data**) model
  - That is, the **same** program runs on all nodes



- You then ***decide*** what each process does based on its *rank* using an if-else statement.

Concepts and Ideas

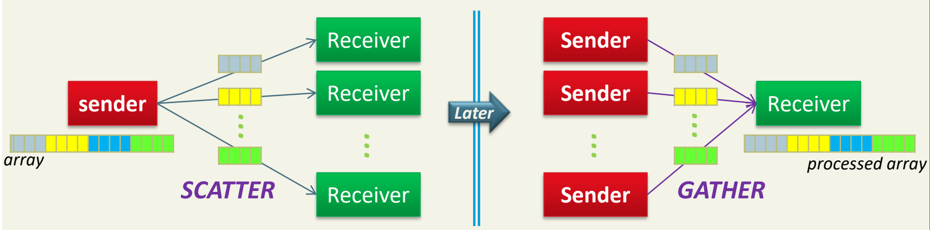
- **What is a message?**
  - A **message** is **data**. It can be as simple as a ***single value*** (e.g. float) or as a ***complicated structure*** (e.g. array).
- **Can *Message Passing* run on single machine?**
  - Yes! While the aim is run processes on separate compute nodes (*this is why we use distributed model, right?*),
    processes, however, could be configured to run on a single machine, time-sharing its resources.
- **Can we use both *Message Passing* and *OpenMP*?**
  - Yes! This is called ***hybrid programming*** in which each node can run several threads to process the data.
    - but for simplicity we will not discuss this in this course.



Scope of Communication

- ***Point-to-point***
  - involves one sender and one receiver.

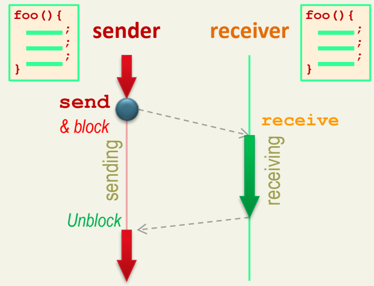
- ***Collective***
  - Several processes are involved
  - ***Example***: a process scatters data to other processes, then later we aggregate scattered data after they have been processed.

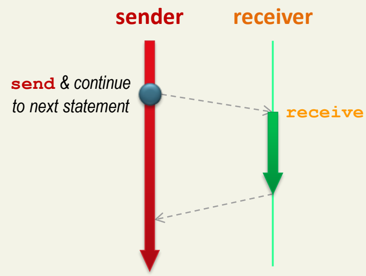


Types of Communication

Blocking (***synchronous***)

- A function called on the sender will block until the communication is finished.
  - When a process runs the **`send`** **routine**, it will block until the receiver runs the ***matching*** **`receive`** **routine**.
    - If a process runs the **`receive`** routine first, it will block until the sender runs the **`send`** routine.
  - Remember that we use SPMD, so the **same program** must include matching functions to run on **both sender & receiver**





Non-blocking (***asynchronous***)

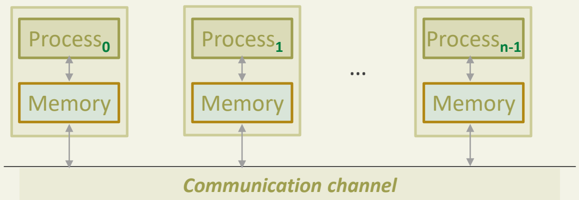
- A function call returns immediately even if the communication is not finished, and the sending process continues to next statements.

Intro to MPI: Overview

- Efforts started in 1991. Version 1.0 was released in in 1994.
  We are now at version 4.1.2 (as of Nov. 2021)
- Developed by MPI forum
  - first effort involved 80 people from 40 organizations in *industry* (major vendors of concurrent computers), *academia* and *government*.
- MPI provides a set of clearly defined *routines* that can be used to build high-performance parallel programs
- MPI 1.0 had no standard for shared memory concept, but later versions supported shared memory communication
  - i.e. nodes in MPI 2.0+ can use shared memory to communicate
    - *this is outside the scope of this course.*

MPI: A Distributed Memory System

- ***nodes*** are connected by a channel (e.g., a network)
- One ***root machine*** spawns programs on other machines in its ***MPI COMMUNICATION WORLD***.
  - *Again, remember that all nodes run the **<u>same</u>** program **(SPMD).***
- Processes are independent and don't share memory. They can only communicate through messages over the network.
  - Important to have good network bandwidth and throughput
- *n* processes are numbered *0, 1, 2, .. n-1*



Why MPI?

- Scalable
  - Can be scaled up to many, many nodes.
- It has a ***massive library of routines*** that provide a powerful way to express parallel programs
  - Usually we use less than a dozen of them for a given program
- Relatively easy to use.
  - As long as you know the basics of parallelism
- Has several implementations that are ***freely*** available to programmers.
  - MPICH (most common, great online support)
  - OpenMPI (also well-known with good support)
  - etc.

MPI Basic Syntax

- **MPI identifiers:** start with by **`MPI_`** followed by letters for:
  - ***Functions***: First Letter upper case then lower case letters
    - e.g. **`MPI_Send`**
  - ***Constants***: ALL upper case
    - e.g., Derived data types (see below): **`MPI_FLOAT`**

- **MPI Derived Data Types**
  - Many MPI functions require the **type of data** to be sent between the different node.
    - Why? Because **different nodes** might have a different CPU architectures and hence might have **different data representation**.
  - The data type needs to be **sent as a parameter**. However, C doesn't allow passing a type as a parameter. Therefore, MPI defines a few constants `MPI_INT`, `MPI_FLOAT`, etc to represent `int`, `float`, etc types.

MPI Derived Datatypes

MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned long int**
MPI_FLOAT	**float**
MPI_DOUBLE	**double**
MPI_LONG_DOUBLE	**long double**
MPI_BYTE	
MPI_PACKED	

Program Structure

Remember: this same program is executed by all nodes

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    int my_rank, comm_sz;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //get my rank
    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); //get # of nodes
    doSomething();        //e.g. send/receive/process data/…etc
    MPI_Finalize();
    return 0;
}
```

Has a main method

Initialise MPI computation

Terminate MPI computation

MPI Components

- MPI has of 100+ routines defined in **mpi.h**, but we will focus only on the most common ones.
  - **MPI_Init** Initialise MPI computation
  - **MPI_Finalize** Terminate MPI computation
  - **MPI_Comm_size** Get number of processes
  - **MPI_Comm_rank** Get current process id
  - **MPI_Send**, **MPI_Recv** Blocking send / receive
  - **MPI_Isend**, **MPI_Irecv** Non-blocking send / receive
- Handling errors:
  - Almost all MPI functions return an **int** error code. Use the constant `MPI_SUCCESS`, i.e. no error, to check for errors.

MPI Components, cont'd

MPI_Init(&argc, &argv)

- Tells MPI to do all the necessary setup.
  - e.g. allocate storage for **message buffers**, decide which process gets which **rank**, etc.
  - Part of this is to define a **communicator** that consists of **all the processes** created when the program is started.
    - Called MPI_COMM_WORLD
- The arguments
  - `&argc` is a pointer to the number of arguments `argc` in `main()`
  - `&argv` are pointers argument vector `argv` in `main()`.
  - When our program doesn't use these arguments, just pass NULL for both.

MPI_Finalize

we're done! MPI cleans up anything allocated for this program.

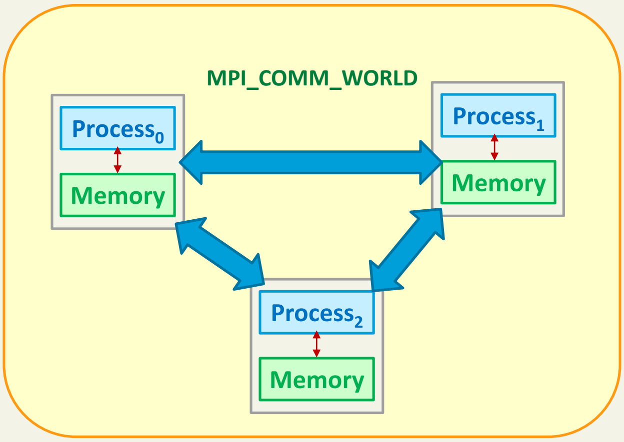
MPI Components, cont'd

```
MIP_Comm_size(
        MPI_COMM_WORLD,        /* in: the communicator */
        &comm_sz               /* out: # of processes in the
                                           communicator*/
)

MPI_Comm_rank(
        MPI_COMM_WORLD,        /* in: the communicator */
        &my_rank               /* out: get rank or process
                                     making this call */
)
```

MPI_COMM_WORLD

- The default communicator, which groups all the processes when the program started.



- We can create other communicators (i.e. groups of nodes), but this is outside the scope of this course.

Compilation and Execution

Compile

source file

mpicc -o mpi_program mpi_program.c

create this executable file

wrapper script to compile

Execute

mpiexec -n 1 ./mpi_program

run with 1 process

mpiexec -n 4 ./mpi_program

run with 4 processes

Hello World: MPI Version 1

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
    int my_rank, comm_sz;          // Number of processes, my process rank
    MPI_Init(&argc, &argv);        // initialize communicator
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);        // get my rank
    MPI_Comm_size(MPI_COMM_WORLD,&comm_sz);        // get # of processes
    printf("Greetings from process %d/%d\n",my_rank, comm_sz);
    MPI_Finalize();                                    // we're done
    return 0;
}
```


Compilation and Execution

mpiexec -n 1 ./mpi_hello

run with 1 process

Output
```
Greetings from process 0 of 1
```

mpiexec -n 4 ./mpi_hello

run with 4 processes

Possible outputs
```
Greetings from process 0 of 4
Greetings from process 2 of 4
Greetings from process 1 of 4
Greetings from process 3 of 4
```

Arbitrary output

Communication

- Point-to-point involves two processes: sender and receiver.
  - Example MPI functions: **`MPI_Send`** and **`MPI_Recv`**
- In order to send a message, we need to specify a few parameters such as:
  - Which process is the **sender**?
  - Data sent:
    - *Location*: Where is the data on the sending process?
    - *Type*: What is the type of that data?
    - *Size*: How much data to send?
  - Which process is the **receiver**?
  - Data received:
    - *Location*: Where should the data be left on the receiving process?
    - *Type*: What is the type of received data?
    - *Size*: How much data to receive?

See code on the next next slide.

Ex.2: Simple Point-to-Point

- Let's start with a simple program where
  - all processes send a greeting message to `Process0`
  - `Process0` displays the received messages along with its own message.

Ex.2: Simple Point-to-Point

```c
const int LEN = 100;
int main() {
   char msg[LEN];                              // String storing message
   int  comm_sz, my_rank;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // get communicator size
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // get my rank
   if (my_rank != 0) { //Everyone EXCEPT Process 0
      //Create and send message to process 0
      sprintf(msg, "Greetings from process %d of %d!", my_rank, comm_sz);
      MPI_Send(msg,LEN,MPI_CHAR,0,0,MPI_COMM_WORLD);
   } else {                    //ONLY Process 0
      printf("Hi from process %dof%d!\n",my_rank,comm_sz); // Print my message
      for (int q = 1; q < comm_sz; q++) {    // Print others' messages
        MPI_Recv(msg,LEN,MPI_CHAR,q,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        printf("%s\n", msg);
      }
   }
   MPI_Finalize();
   return 0;
}
```


Ex.2: Simple Point-to-Point

```c
const int LEN = 100;
int main() {
   char msg[LEN];                              // String storing message
   int  comm_sz, my_rank;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // get communicator size
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // get my rank
   if (my_rank != 0) { //Everyone EXCEPT Pr
      //Create and send message to process 0
      sprintf(msg, "Greetings from process %d of %d!", my_rank, comm_sz);
      MPI_Send(msg,LEN,MPI_CHAR,0,0,MPI_COMM_WORLD);
   } else {                    //ONLY Process 0
      printf("Hi from process %dof%d!\n",my_rank,comm_sz
      for (int q = 1; q < comm_sz; q++) {    // Print others' messages
        MPI_Recv(msg,LEN,MPI_CHAR,q,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        printf("%s\n", msg);
      }
   }
   MPI_Finalize();
   return 0;
}
```

Destination Process ID

Same MPI WORLD

Source Process ID

Ex.2: Simple Point-to-Point

```
MPI_Send(     msg, LEN, MPI_CHAR,  /* in: message info */
              0,                          /* in: destination process ID*/
              0,                          /* in: tag */
              MPI_COMM_WORLD);            /* in: in which communicator?*/
```

```
MPI_Recv(  msg, LEN, MPI_CHAR,     /* in: message info */
              q,                          /* in: from which source ID? */
              0,                          /* in: with which tag? */
              MPI_COMM_WORLD              /* in: in which communicator?*/
              MPI_STATUS_IGNORE);  /* out:don't need sender's info */
```

Use MPI_ANY_SOURCE to receive from all sources

Use MPI_ANY_TAG to receive from any tag

MPI_Send and MPI_Recv Syntax

```
int MPI_Send(
  void*          msg_p,           /* in: address of input data */
  int            msg_size,        /* in: size of msg in terms of datatype. */
  MPI_Datatype   datatype,        /* in: MPI_INT, MPI_FLOAT, etc */
  int            dest_process,    /* in: where to send msg? */
  int            tag,          /* in: tag */
  MPI_Comm       comm          /* in: which communicator? */ );

int MPI_Recv(
  void*          msg_p,           /* out: address of output data */
  int            msg_size,        /* in: size of msg received. */
  MPI_Datatype   datatype,        /* in: MPI_INT, MPI_FLOAT, etc */
  int            source_process,/* in: from whom I am receiving msg? */
  int            tag,          /* in: tag */
  MPI_Comm       comm          /* in: which communicator? */
  MPI_Status*    status_p      /* in: status*/ );
```

Ex.2: Simple Point-to-Point

```
mpiexec -n 1 ./mpi_hello

    Greetings from process 0 of 1 !
```

```
mpiexec -n 4 ./mpi_hello

    Greetings from process 0 of 4 !
    Greetings from process 1 of 4 !
    Greetings from process 2 of 4 !
    Greetings from process 3 of 4 !
```

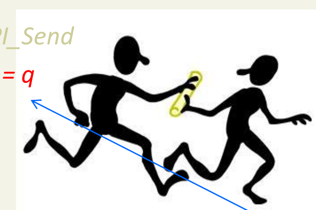

Message Matching

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

r

MPI_Send

src = q



MPI_Recv

dest = r

q

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Wildcard Arguments

- A receiver can get a message without specifying the following:
  - the sender of the message
    - Use MPI_ANY_SOURCE *wildcard* constant for the source.
  - the message tag
    - Use MPI_ANY_TAG *wildcard* constant
  - the amount of data in the message
    - Can be obtained later (see next slide)

```
MPI_Recv( data, data_size, data_type, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

- This is useful when one process is receiving from many processes without knowing the order of the messages.

- **Senders cannot use wildcard arguments!**
  - Senders must specify a target process rank and non-negative tag.

Aside: Sender's STATUS

- You can use the status argument to know info about the sender

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

- MIP_Status members know:
  - The source
  - The tag
  - An error status
    - Use for error handling

```
MPI_Status*  status;

status.MPI_SOURCE
status.MPI_TAG
status.MPI_ERROR
```

- Use MPI_Get_count function to know about the size of the received message. Example for character message

```
int* count;
MPI_Get_count(&status, MPI_CHAR, &count);
```

Issues with Send and Receive

- MPI_Send() *EITHER*
  - puts the message into a **buffer** and returns,
  - or it **blocks** until it starts transmission and then it returns.

  After it returns, we don't know if the message is sent successfully.

- MPI_Receive always **blocks** until it receives a matching message.
  - The process may wait forever **if no matching message** is received.
  - After it returns, we know message is received and placed into a buffer.

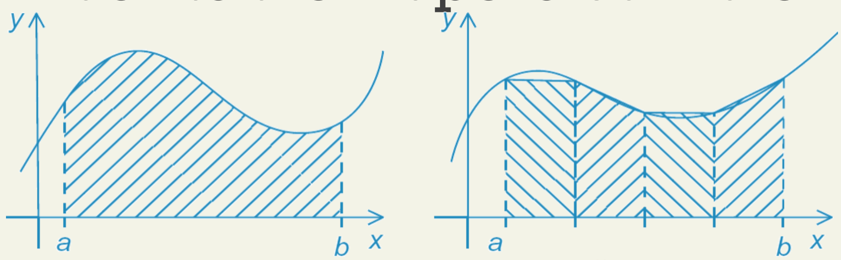
- Exact behavior is **determined by the MPI implementation**.
  - Know your implementation; don't make assumptions!

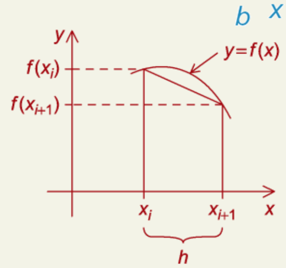
Scope of Communication (again)

- **Blocking communication:**
  - The functions do not return (block) until the communication is finished.
  - Examples: MPI_Send() and MPI_Receive()
    - MPI_Receive() blocks until it receives a matching message
    - MPI_Send() blocks until its message is either sent or at least placed in a system buffer.

- **Non-blocking communication:**
  - The function return immediately even if the communication is not finished (if the OS allows for them to work in the background), and then the program continues to the next statements
  - Example, MPI_Isend() and MPI_Irecv(): there is no guarantee that send_buffer is sent, or receive_buffer is filled with received data.
    - *When to use?* to improve efficiency when you want your process to do some computation while the data is being sent/received.
    - There are ways to know if the send/receive has finished.
  - Not going to focus on this type in this course.

Back to the Trapezoidal Rule





$$h = \frac{b-a}{n}$$

$$Area\ of\ one\ trapezoid = h\left(\frac{f(x_i) + f(x_{i+1})}{2}\right)$$

$$Sum\ of\ trapezoid\ areas = h\left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right)$$

$x_0 = a$, $x_1 = a + h$, …. , $x_i = a + i * h$, … , $x_2 = a + 2h$, $x_n = b$

Area Under a Curve Serial algorithm

$$h = \frac{b-a}{n}$$

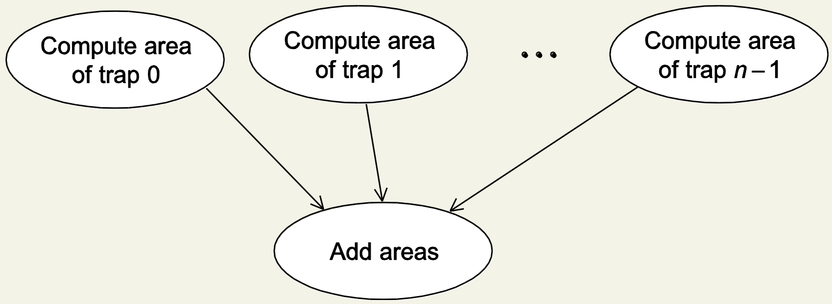
$$Sum\ of\ trapezoid\ areas = h\left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right)$$

$$x_i = a + i * h$$

```
/* Serial code: input: a, b, n */
h = (b - a) / n;              //calculate h only once
approx = (f(a) + f(b)) / 2.0; //first and last terms
for (i = 1; i <= n-1; i++) {  //remaining terms
      xi = a + i * h;
      approx += f(xi);
}
approx = h * approx;          //approximate area
```

Parallelizing the Trapezoidal Rule

1. Partition problem into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.



Trap Rule (Parallel pseudo-code)

```
//1) each process determines its interval of integration
Get a, b, n;
h = (b-a)/n;
my_n = n / comm_sz;
my_a = a + my_rank * my_n * h;
my_b = my_a + my_n * h;
//2) each process computes its own results (partial result) over its integral
my_sum = Trap(my_a, my_b, my_n, h);
//3) everyone sends partial results to process 0 which aggregates them
if(my_rank != 0) //Everyone except process 0
    Send my_sum to process 0;
else {              //Only process 0
    total_sum = my_sum;
    for(source = 1; source < comm_sz; source++){
        other_sum = receive partial results from other processors
        total_sum += other_sum;
    }
}
if(m_rank == 0)
    print total_sum;
```

Version 1: Trap Rule

```c
int main() {
   double a = 0.0, b = 3.0, n = 1024;
   int my_rank, comm_sz, my_n, source;
   double h, my_a, my_b, my_sum, total_sum;
   //initialize, get rank and comm_sz
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   //break down the problem to subproblems
   h = (b-a) / n;              // the same for all processes
   my_n = n / comm_sz;                  // the same for all processes
   my_a = a + my_rank * my_n * h;       // unique to each process
   my_b = my_a + my_n * h;              // unique to each process
   my_sum= Trap(my_a,my_b,my_n,h); // find partial result from one process
   if (my_rank != 0) {                  // send my partial result to process 0
      MPI_Send(&my_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   } else {                             // process 0 combines the partial results
      total_sum = my_sum;
      for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&my_sum,1,MPI_DOUBLE, source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        total_sum += my_sum;
      }
   }
   if (my_rank == 0)  printf("%.15e\n", total_sum);
   MPI_Finalize();
   return 0;
}
```

Problem: values of a, b, and n are hardwired (we will see later how to read them from the user)

Version 1: Trap Rule, *cont'd*

```c
//Serial function for estimating a definite integral
double Trap(double a  , double b , int n, double h ) {
   double estimate, x;
   int i;
   estimate = (f(a) + f(b))/2.0;
   for (i = 1; i <= n-1; i++) {
      x = a + i*h;
      estimate += f(x);
   }
   estimate = estimate*h;

   return estimate;
}

//assume function below is what we want to integrate – it could be any other function
double f(double x) {return x*x;}
```

Printing Output

- Most MPI implementations allow ALL processes in MPI_COMM_WORLD full access to stdout and stderr

```
int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
         my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}
```

Each process just prints a message on stdout

Q. Can you think of a way to control the order of the output?
(assume all outputs are directed to one machine)

unpredictable order

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```


Reading Input

- Most MPI implementations **ONLY** allow **process 0** in MPI_COMM_WORLD access to stdin.
  - **How to read input?** Process 0 must read the data (scanf) and send to the other processes.

Trap Rule: V.2 (Reading Input)

```
int main(){
   ...
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   read_input(my_rank, comm_sz, &a, &b, &n);
   //partition problem
   h = (b-a) / n;
   ... etc. (the same as before)
}
```

```
void read_input(int my_rank,int comm_sz,double* a_p,double* b_p,int* n_p){
   int dest;
   if (my_rank==0){         //only Process 0 reads input
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
   } else {                    //all other processes receive data from process 0
        MPI_Recv(a_p,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(b_p,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(n_p,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   }
}
```


Trap Rule: V.2 (Main Method)

```
int main() {
   int my_rank, comm_sz, n, my_n, source;
   double a, b, h, my_a, my_b, my_sum, total_sum;
   //initialize, get rank and comm_sz
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   read_input(my_rank, comm_sz, &a, &b, &n);
   //break down the problem to subproblems
   h = (b-a) / n;          // the same for all processes
   my_n = n / comm_sz;              // the same for all processes
   my_a = a + my_rank * my_n * h;     //unique to each process
   my_b = my_a + my_n * h;          //unique to each process
   my_sum = Trap(my_a, my_b, my_n, h); //find partial result from one process
   if (my_rank != 0) {              //send my partial result to process 0
      MPI_Send(&my_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   } else {                         //process 0 combines the partial results
      total_sum = my_sum;
      for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&my_sum,1,MPI_DOUBLE, source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        total_sum += my_sum;
      }
   }
   if (my_rank == 0)  printf("%.15e\n", total_sum);
   MPI_Finalize();
   return 0;
}
```

Before Developing an MPI Program

- You need to come up with a high-level algorithm that involves the process of sending/receiving data. For example:
    - P0 reads data,
    - P0 sends to everyone,
    - Everyone performs an operation and sends back to P0
    - P0 collects results and display them.
- You also need to think about a few parameters:
    - Which process is sending?
    - Sending the data (message):
        - Which data (and what type) is being sent?
        - How much data is there? and how much is being sent?
            - e.g. are we sending a whole array, or part of it?
    - Which process(es) is(are) receiving?
    - Receiving the data:
        - Where should the data be saved on the receiving process?
        - What amount of data (and what type) is being received?

Conclusion

Today:

Distributed Memory Programming

Intro to MPI

MPI: Point to Point communication

Example: Area under the curve in MPI

Dealing with I/O

Next Lecture:

MPI: Collective communication