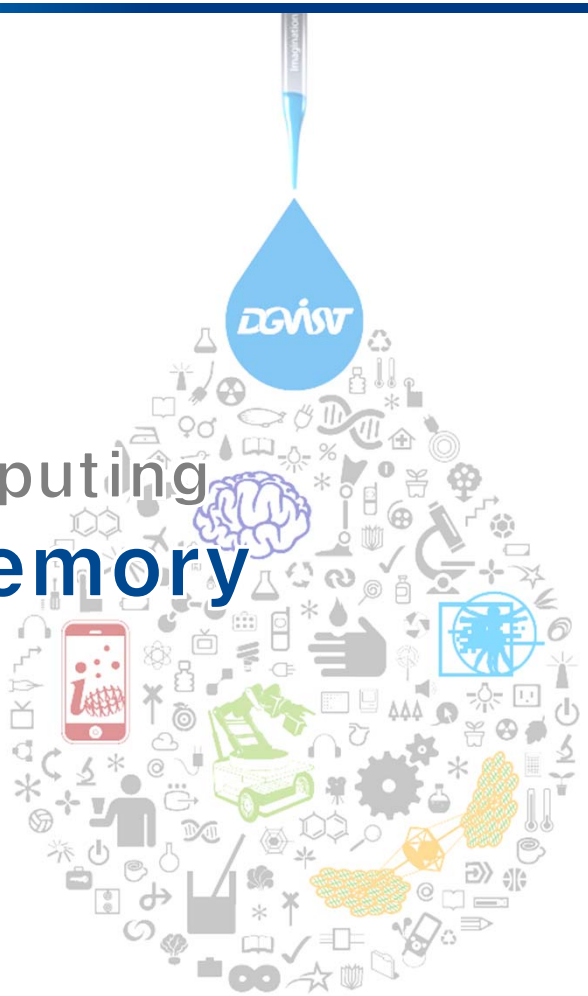
InfoLab



IC621: Distributed and Parallel Computing

Lecture 03 : Distributed-Memory Programming with MPI (I)

Min-Soo Kim

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

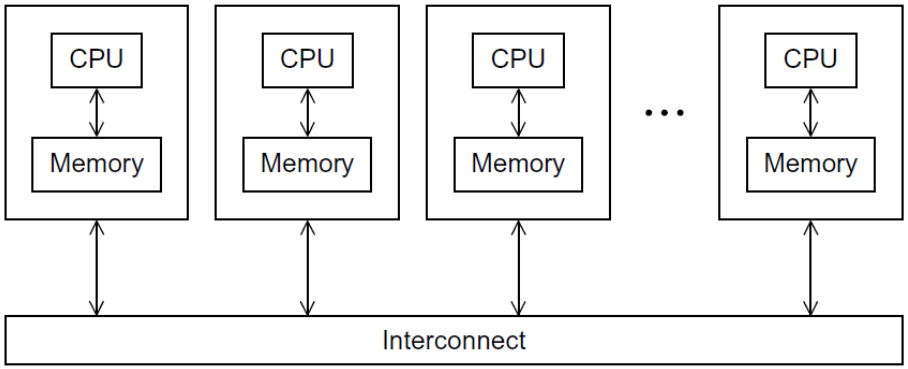
InfoLab

GETTING STARTED

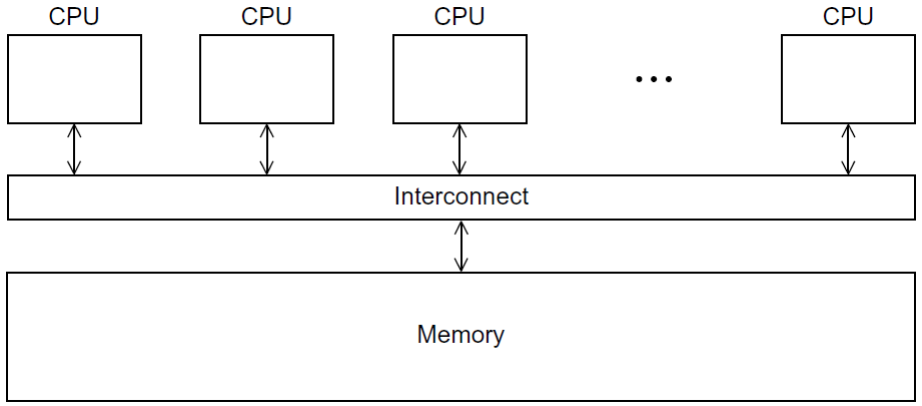
DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

From a programmer's point of view

- **Distributed-memory system**
  - a collection of core-memory pairs connected by a network
  - a memory is directly accessible only to an associated core
- **Shared-memory system**
  - a collection of cores connected to a globally accessible memory
  - each core can have access to any memory location



A distributed-memory system



A shared-memory system

Message-Passing Interface (MPI)

- **A program running on one core-memory pair is usually called a process**
- **Two processes can communicate by calling functions**
  - one process calls a send function
  - the other calls a receive function
- **MPI defines a library of functions** **(called from C, C++, and Fortran programs)**
  - some of MPI's different send and receive functions
  - some "global" communication functions that can involve more than two processes (collective communication)
  - some fundamental issues: data partitioning, I/O in distributed-memory systems, …

Hello world example

- **Structure**
  - one process gets messages and output them (process 0)
  - the other processes send messages to it

- **Rank**
  - nonnegative integer for identifying each process
  - e.g., {0, 1, 2, … , p-1} for p processes

- **Compilation**
  - `mpicc` is a wrapper script for the C compiler

```
$ mpicc -g -Wall -o mpi_hello mpi_hello.c
```

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char       greeting[MAX_STRING];
9     int        comm_sz;  /* Number of processes */
10    int        my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n", my_rank,
             comm_sz);
23       for (int q = 1; q < comm_sz; q++) {
24          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
25             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          printf("%s\n", greeting);
27       }
28    }
29
30    MPI_Finalize();
31    return 0;
32 }  /* main */
```

Execution

```
$ mpiexec –n <number of processes> ./mpi_hello

$ 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!
```

MPI programs

- **MPI_Init**
  - `argc_p` and `argv_p` are pointers to the arguments to `argc` and `argv` of `main`

```
int MPI_Init(
      int*      argc_p   /* in/out */,
      char***   argv_p   /* in/out */);
```

- **MPI_Finalize**
  - any resources allocated for MPI are freed

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

Communicator

- a collection of processes that can send messages to each other
- `MPI_COMM_WORLD` : a communicator that consists of all of the processes started by the user when she started the program

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

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

Single Program Multiple Data(SPMD)

- **We compiled a single program (in the example)**
  - ➢ we didn’t compile a different program for each process
    - in spite of the fact that process 0 is doing something fundamentally different from the other processes
    - process 0 is receiving a series of messages and printing them, while each of the other processes is creating and sending a message

- **SPMD is quite common in parallel programming**
  - ➢ SPMD is achieved by simply having the processes branch on the basis of their process rank
  - ➢ most MPI programs are written in this way

MPI_Send

- **tag**
  - nonnegative integer that can be used to distinguish messages
- **communicator**
  - message sent by a process using one communicator cannot be received by a process that's using a different communicator

```
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 */);
```

```
sprintf(greeting, "Greetings from process %d of %d!",
      my_rank, comm_sz);
MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
      MPI_COMM_WORLD);
```

MPI_Recv

- **status_p**
  - it is not used in many cases
  - MPI_STATUS_IGNORE: special MPI constant

```
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 */);
```

Table 3.1 Some Predefined MPI 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	

```
MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
    0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("%s\n", greeting);
```

Message matching

- **Suppose process `q` calls MPI_Send**
- **Suppose process `r` calls MPI_Recv**

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

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

- **Message type & size**
  - If `recv_type` = `send_type` and `recv_buf_sz` ≥ `send_buf_sz`, then the message sent by q can be successfully received by `r`

Problem about source & tag

- one process is receiving messages from multiple processes
- receiving process doesn't know the order in which the other processes will send the messages

Solution : wildcard argument

- using a special constant MPI_ANY_SOURCE in MPI_Recv
- using a special constant MPI_ANY_TAG in MPI_Recv

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

The `status_p` argument

- **A receiver can receive a message without knowing**
  - the amount of data in the message
  - the sender of the message
  - the tag of the message

- **MPI_Status `status` (struct type)**
  - status.MPI_SOURCE
  - status.MPI_TAG

```
int MPI_Get_count(
          MPI_Status*  status_p   /* in  */,
          MPI_Datatype type       /* in  */,
          int*         count_p    /* out */);
```

Semantics of `MPI_Send` and `MPI_Recv`

- **The sending process will assemble the message**
- **Once the message has been assembled,**
  - the sending process can buffer the message
    - the MPI system will place the message (data and envelope) into its own internal storage
    - the call to MPI_Send will return
  - the sending process can block it
    - it will wait until it can begin transmitting the message
    - the call to MPI_Send may not return immediately
- **Note**
  - if we use MPI_Send, when the function returns, we don't actually know whether the message has been transmitted
  - MPI provides alternative functions for sending so as to know that the message has been transmitted

- **Behavior of MPI_Send is determined by MPI implementation**
  - typical implementations have a default "cutoff" message size
  - if the size of a message is less than the cutoff, it will be buffered
  - if the size of the message is greater than the cutoff, MPI_Send will block

- **MPI_Recv always blocks until a matching message has been received**
  - when a call to MPI_Recv returns, there is a message stored in the receive buffer

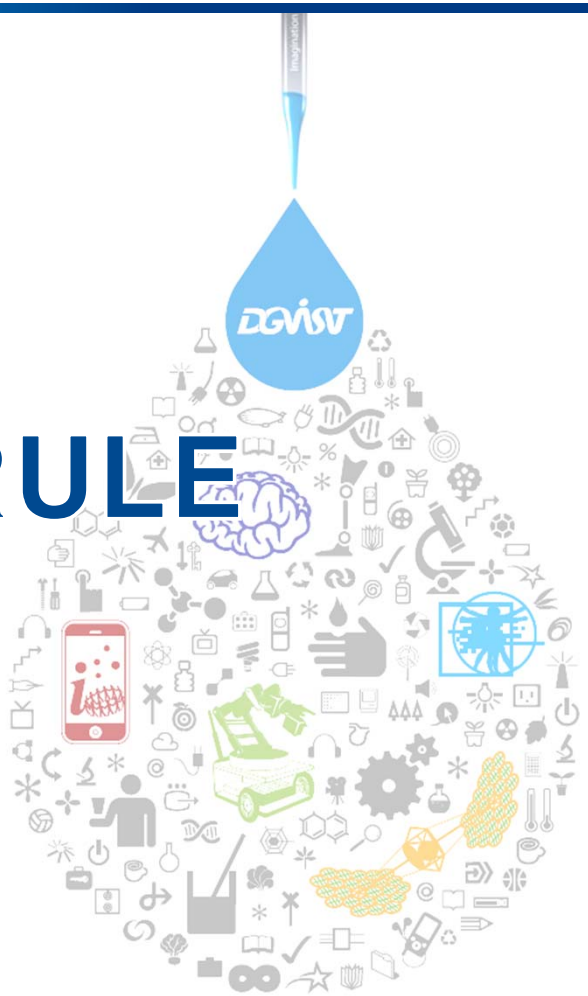
- **MPI requires that messages be nonovertaking**
  - if process q sends two messages to process r, then the first message sent by q must be available to r before the second message
  - but, there is no restriction on the arrival of messages sent from different processes
    - MPI cannot impose performance on a network

Some potential pitfalls

- **If a process tries to receive a message and there's no matching send**
  - the process will block forever (i.e., the process will hang)
- **If a call to MPI_Send blocks and there's no matching receive**
  - the sending process can hang
- **If a call to MPI_Send is buffered and there's no matching receive**
  - the message will be lost

InfoLab

THE TRAPEZOIDAL RULE IN MPI

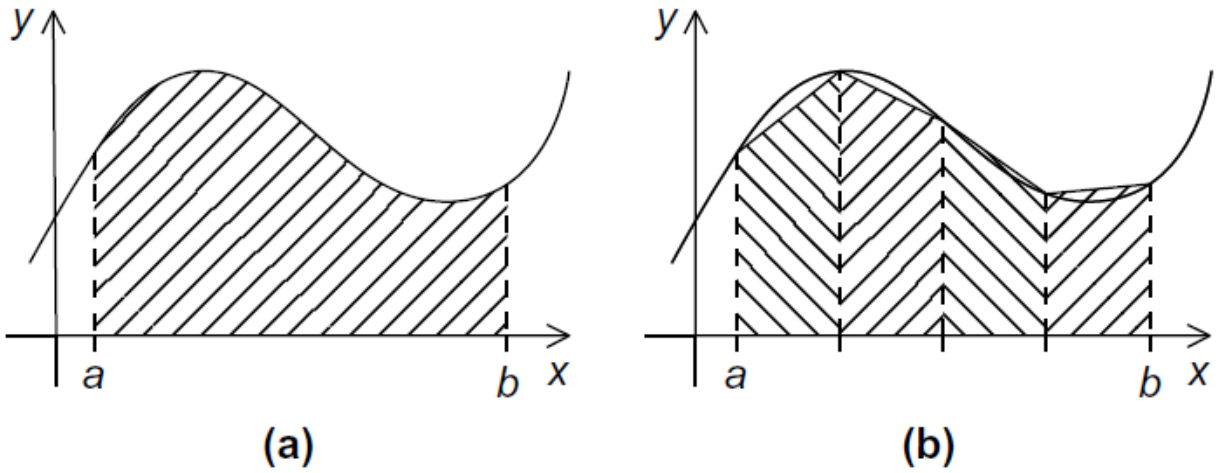


DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

The trapezoidal rule

- **Approximation of the area between the graph of a function, y = f(x), two vertical lines, and the x-axis**
- **Basic idea**
  - divide the interval on the *x*-axis into *n* equal subintervals
  - approximate the area lying between the graph and each subinterval

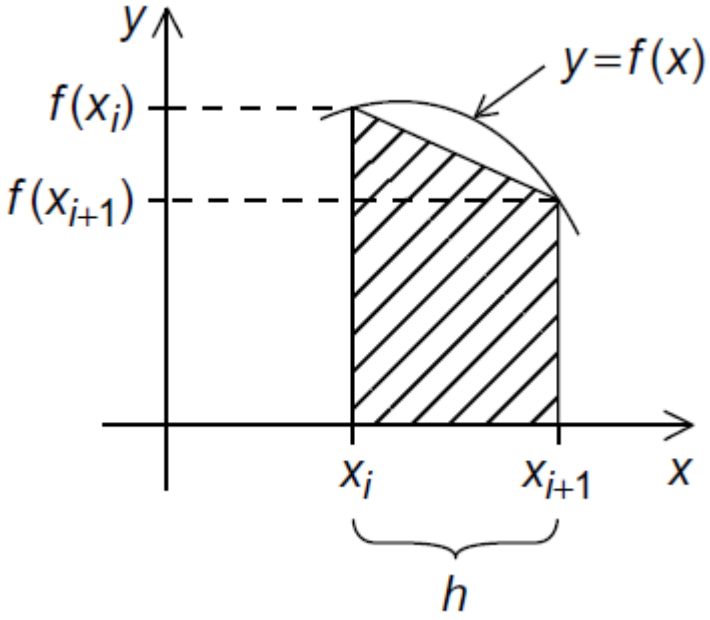
$$\text{Area of one trapezoid} = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$



Pseudo-code for a serial program

$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

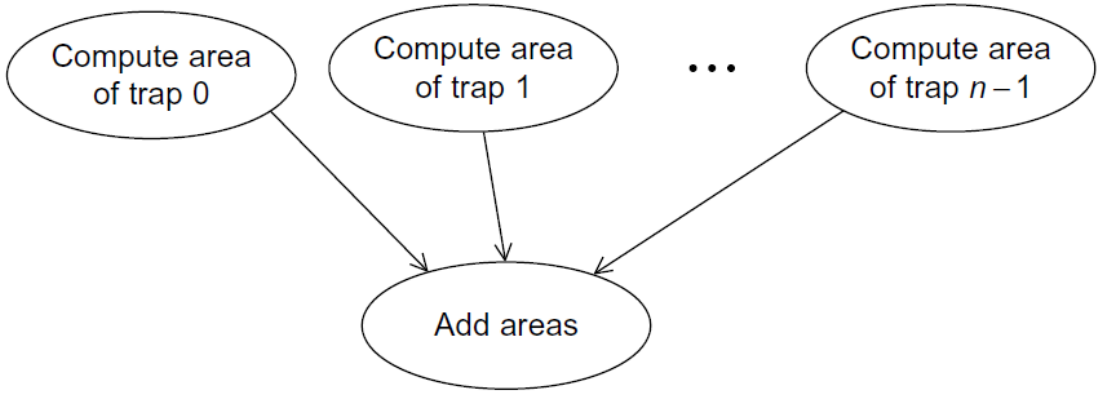
```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```



Parallelizing the trapezoidal rule

- **Our intuition**
  - the more trapezoids we use, the more accurate our estimate will be
  - we will use many more trapezoids than cores

```
1   Get a, b, n;
2   h = (b-a)/n;
3   local_n = n/comm_sz;
4   local_a = a + my_rank*local_n*h;
5   local_b = local_a + local_n*h;
6   local_integral = Trap(local_a, local_b, local_n, h);
```



- **Local variables**
  - local_a, local_b, local_n
- **Global variables**
  - a, b, n

```
 7   if (my_rank != 0)
 8      Send local_integral to process 0;
 9   else /* my_rank == 0 */
10      total_integral = local_integral;
11      for (proc = 1; proc < comm_sz; proc++) {
12         Receive local_integral from proc;
13         total_integral += local_integral;
14      }
15   }
16   if (my_rank == 0)
17      print result;
```

```
1  int main(void) {
2     int my_rank, comm_sz, n = 1024, local_n;
3     double a = 0.0, b = 3.0, h, local_a, local_b;
4     double local_int, total_int;
5     int source;
6
7     MPI_Init(NULL, NULL);
8     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
10
11    h = (b-a)/n;           /* h is the same for all processes */
12    local_n = n/comm_sz;   /* So is the number of trapezoids  */
13
14    local_a = a + my_rank*local_n*h;
15    local_b = local_a + local_n*h;
16    local_int = Trap(local_a, local_b, local_n, h);
17
18    if (my_rank != 0) {
19       MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       total_int = local_int;
23       for (source = 1; source < comm_sz; source++) {
24          MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
25                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
26          total_int += local_int;
27       }
28    }
29
30    if (my_rank == 0) {
31       printf("With n = %d trapezoids, our estimate\n", n);
32       printf("of the integral from %f to %f = %.15e\n",
33           a, b, total_int);
34    }
35    MPI_Finalize();
36    return 0;
37 } /*  main  */
```

```
1  double Trap(
2        double left_endpt  /* in */,
3        double right_endpt /* in */,
4        int    trap_count  /* in */,
5        double base_len    /* in */) {
6     double estimate, x;
7     int i;
8
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10    for (i = 1; i <= trap_count-1; i++) {
11       x = left_endpt + i*base_len;
12       estimate += f(x);
13    }
14    estimate = estimate*base_len;
15
16    return estimate;
17 } /*  Trap  */
```

InfoLab

DEALING WITH I/O

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology

Output

- **Most MPI implementations**
  - allow all the processes in MPI_COMM_WORLD full access to `stdout` and `stderr`
  - so, allow all processes to execute `printf` and `fprintf(stderr, ...)`
- **But, most MPI implementations**
  - don't provide any automatic scheduling of access to these devices
  - if multiple processes are attempting to write to `stdout`, the order in which the processes' output appears will be unpredictable
- **Reason of unpredictable output**
  - MPI processes are "competing" for access to the shared output device
  - it's impossible to predict the order in which the processes' output will be queued up (nondeterminism)
  - c.f., "greetings" program : each process send its output to process 0, and process 0 can print the output in process rank order

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

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;
}  /* main */
```

expected output with 5 processes

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

unpredictable output with 6 processes

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

Input

- **Unlike output, most MPI implementations only allow process 0 in MPI_COMM_WORLD access to `stdin`**
- **To use `scanf`, we need to branch on process rank**
  - process 0 reads in the data and then sending it to the other processes
- **Example**
  - `Get_input` function for our parallel trapezoidal rule program
  - being careful to put it after initialization of `my_rank` and `comm_sz`

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_input(my_rank, comm_sz, &a, &b, &n);

h = (b-a)/n;
. . .
```

```
1  void Get_input(
2        int      my_rank   /* in  */,
3        int      comm_sz   /* in  */,
4        double*  a_p       /* out */,
5        double*  b_p       /* out */,
6        int*     n_p       /* out */) {
7     int dest;
8
9     if (my_rank == 0) {
10       printf("Enter a, b, and n\n");
11       scanf("%lf %lf %d", a_p, b_p, n_p);
12       for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16       }
17    } else { /* my_rank != 0 */
18       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19             MPI_STATUS_IGNORE);
20       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21             MPI_STATUS_IGNORE);
22       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23             MPI_STATUS_IGNORE);
24    }
25 }  /* Get_input */
```

InfoLab

Thank you!

DGIST 대구경북과학기술원
Daegu Gyeongbuk Institute of Science & Technology