Many slides of this lecture are adopted and slightly modified from:
• Gerassimos Barlas
• Peter S. Pacheco
In all MPI implementations, all processes in MPI_COMM_WORLD have access to stdout and stderr.

BUT .. In most of them there is no scheduling of access to output devices!
Running 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?

unpredictable output!!

- Processes are competing for stdout
- Result: nondeterminism!
How About Input?

• Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.

• Process 0 must read the data and send to the other processes.
Function for reading user input

```c
void Get_input(
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p /* out */,
    double* b_p /* out */,
    int* n_p /* out */) {
  int dest;

  if (my_rank == 0) {
    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 { /* my_rank != 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);
  }
} /* Get_input */
```
Let’s apply what we’ve learned so far, to solve an example more sophisticated than printing strings!
To find this area, we approximate it with trapezoids.
One trapezoid

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

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

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

$x_0 = a, \ x_1 = a + h, \ x_2 = a + 2h, \ldots, \ x_{n-1} = a + (n-1)h, \ x_n = b$

Sum of trapezoid areas $= h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$
Pseudo-code for a serial program

```c
/* Input: a, b, n */
float h = (b-a)/n;
float approx = (f(a) + f(b))/2.0;
for (int i = 1; i <= n-1; i++) {
    float x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```
Parallelizing the Trapezoidal Rule

1. Partition problem solution into tasks ... As many tasks as possible.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.
Tasks and communications for Trapezoidal Rule

1. Compute area of trap 0
2. Compute area of trap 1
3. ... (indicated by ellipsis)
4. Compute area of trap $n-1$

Add areas
Parallel pseudo-code

```java
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);
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;
```
```c
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 */
double Trap(
    double left_endpt /* in */,
    double right_endpt /* in */,
    int trap_count /* in */,
    double base_len /* in */) {

double estimate, x;
int i;

estimate = (f(left_endpt) + f(right_endpt))/2.0;
for (i = 1; i <= trap_count - 1; i++) {
    x = left_endpt + i*base_len;
    estimate += f(x);
}
estimate = estimate*base_len;

    return estimate;
} /* Trap */
The Final Sum ... Again!!

1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   (b) Processes 0, 2, 4, and 6 add in the received values.
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (d) Processes 0 and 4 add the received values into their new values.

2. (a) Process 4 sends its newest value to process 0.
   (b) Process 0 adds the received value to its newest value.
A tree-structured global sum
An alternative tree-structured global sum

Is this better? or the previous one?  
A: Depends on the underlying system!
Reduction

- Reducing a set of numbers into a smaller set of numbers via a function
  - Example: reducing the group [1, 2, 3, 4, 5] with the sum function $\rightarrow 15$
- MPI provides a handy function that handles almost all of the common reductions that a programmer needs to do in a parallel application
Every process has an element

Every process has an array of elements
MPI_Reduce

has size:
sizeof(datatype) * count

only relevant to dest_process

MPI_Reduce is called by all processes involved. This is why it is called *collective call*.
Predefined reduction operators in MPI

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>

Location = rank of the process that owns it
Collective vs. Point-to-Point Communications

• All the processes in the communicator must call the same collective function.
  – For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous.

• The arguments passed by each process to an MPI collective communication must be “compatible.”
  – For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous.
Collective vs. Point-to-Point Communications

• The `output_data_p` argument is only used on `dest_process`.

• However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it's just `NULL`.

• All collective communication calls are blocking.
Collective vs. Point-to-Point Communications

• Point-to-point communications are matched on the basis of tags and communicators.

• Collective communications don’t use tags.

• They’re matched solely on the basis of the communicator and the order in which they’re called.
Example

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
</tr>
<tr>
<td>1</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
</tr>
<tr>
<td>2</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
<td>MPI_Reduce(&amp;a, &amp;b, ...)</td>
<td>MPI_Reduce(&amp;c, &amp;d, ...)</td>
</tr>
</tbody>
</table>

Assume:
- all processes use the operator MPI_SUM
- destination is process 0

What will be the final values of b and d??
Yet Another Example

MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);

This is illegal in MPI and the result is non-predictable!
MPI_Allreduce

- Useful in a situation in which all of the processes need the result of a global sum in order to complete some larger computation.

```c
int MPI_Allreduce(
    void* input_data_p, /* in */,
    void* output_data_p, /* out */,
    int count, /* in */,
    MPI_Datatype datatype, /* in */,
    MPI_Op operator, /* in */,
    MPI_Comm comm /* in */);
```

No destination argument!
Broadcast

- Data belonging to a single process is sent to all of the processes in the communicator.

```c
int MPI_Bcast(
    void* data_p, /* in/out */, 
    int count,    /* in */
    MPI_Datatype datatype, /* in */
    int source_proc, /* in */
    MPI_Comm comm, /* in */
);
```

**ALL** processes in the communicator must call MPI_Bcast()
A tree-structured broadcast.
A version of Get_input that uses MPI_Bcast

```c
void Get_input(
    int my_rank    /* in */,
    int comm_sz   /* in */,
    double* a_p   /* out */,
    double* b_p   /* out */,
    int* n_p      /* out */) {

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }

    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* Get_input */
```
Conclusions

• A **communicator** is a collection of processes that can send messages to each other.

• **Collective communications** involve all the processes in a communicator.

• When studying MPI be careful of the caveats (i.e. usage that leads to crash, nondeterministic behavior, ... ).