NYU
New York University

# CSCI-UA.0480-003
# Parallel Computing

## Lecture 8: MPI - II

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Many slides of this lecture are adopted and slightly modified from:
- Gerassimos Barlas
- Peter S. Pacheco

# Dealing with I/O

```
#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 */
```

In all MPI implementations, all processes in MPI_COMM_WORLD have access to stdout and sterr.

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?

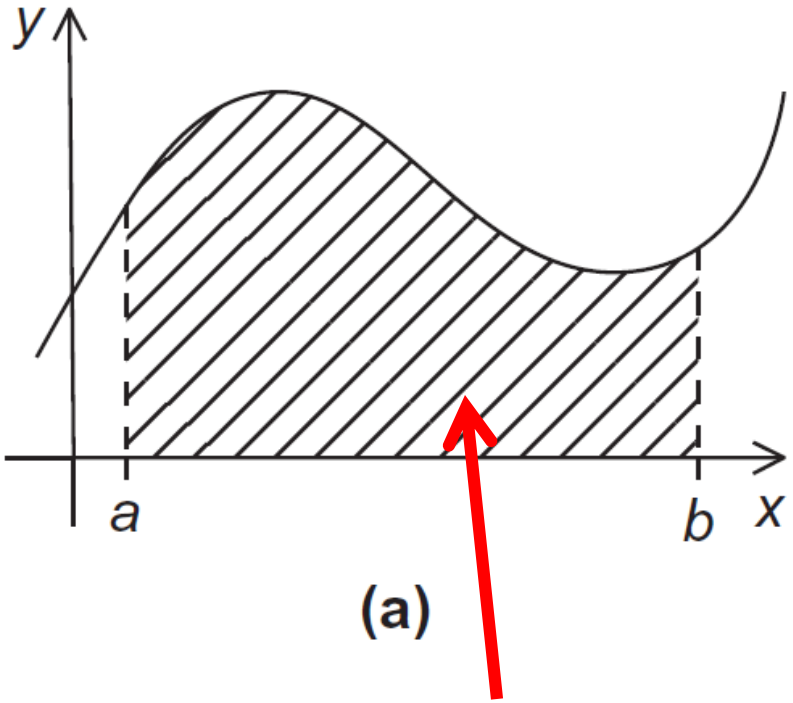
- Most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin.
- Process 0 must read the data and send to the other processes.

# Function for reading user input

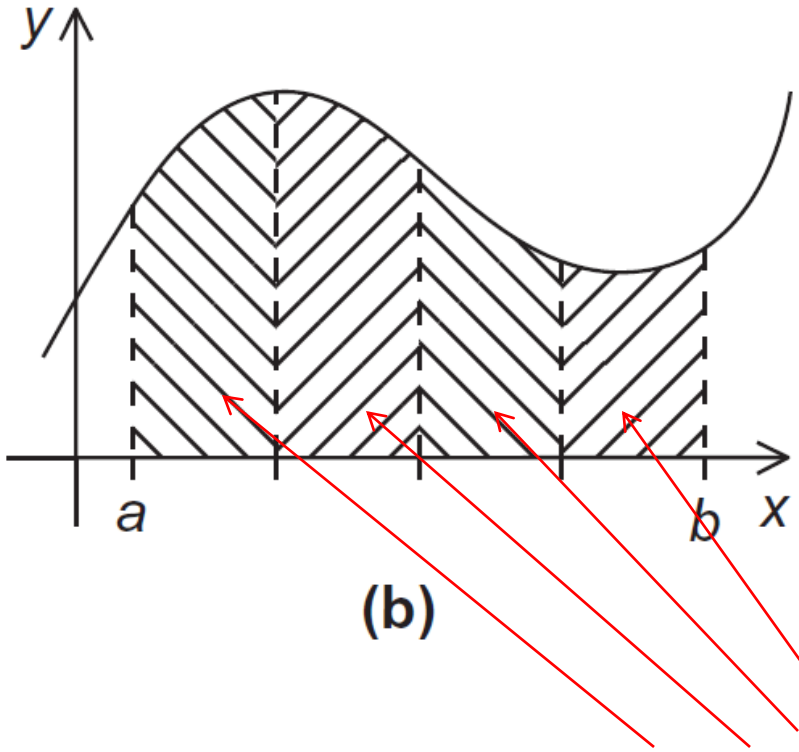
```
void Get_input(
      int       my_rank   /* in  */,
      int       comm_sz   /* in  */,
      double*   a_p       /* out */,
      double*   b_p       /* out */,
      int*      n_p       /* out */) {
   int dest;

   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!

# The Trapezoidal Rule

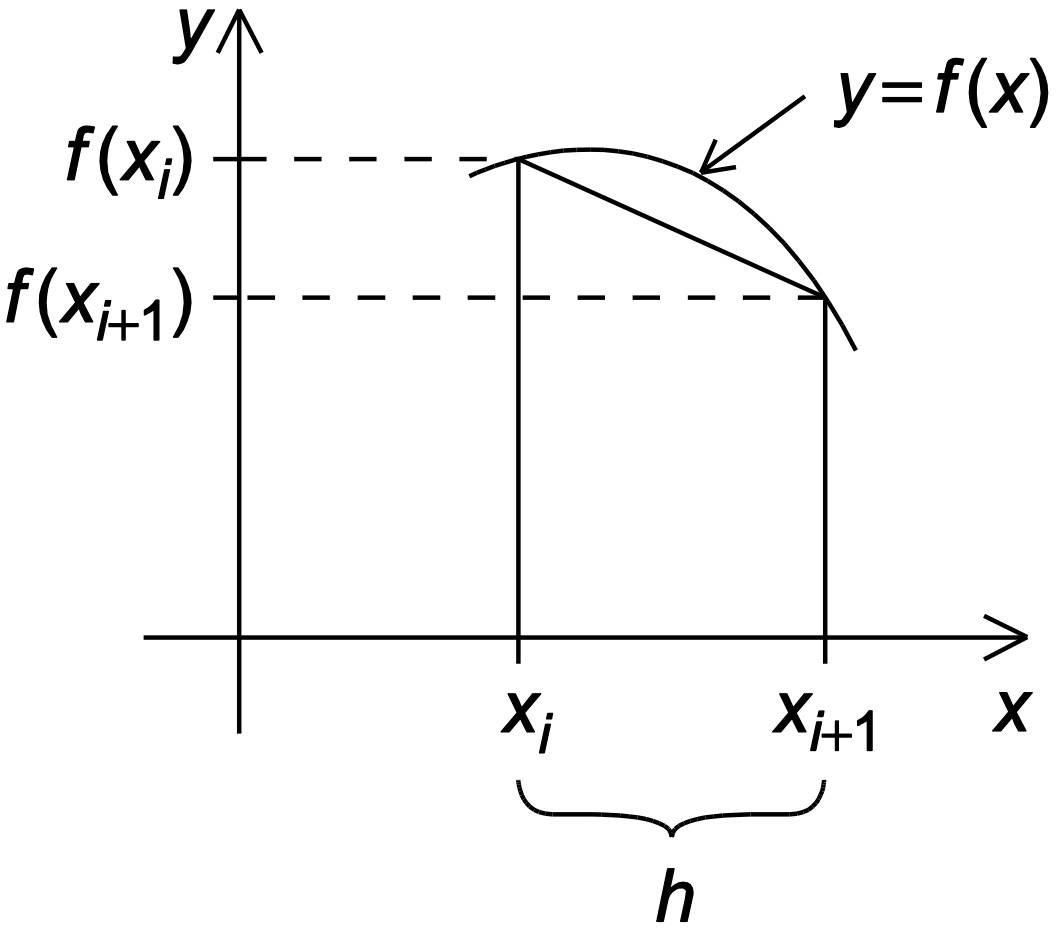



To find this area




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

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

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

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

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

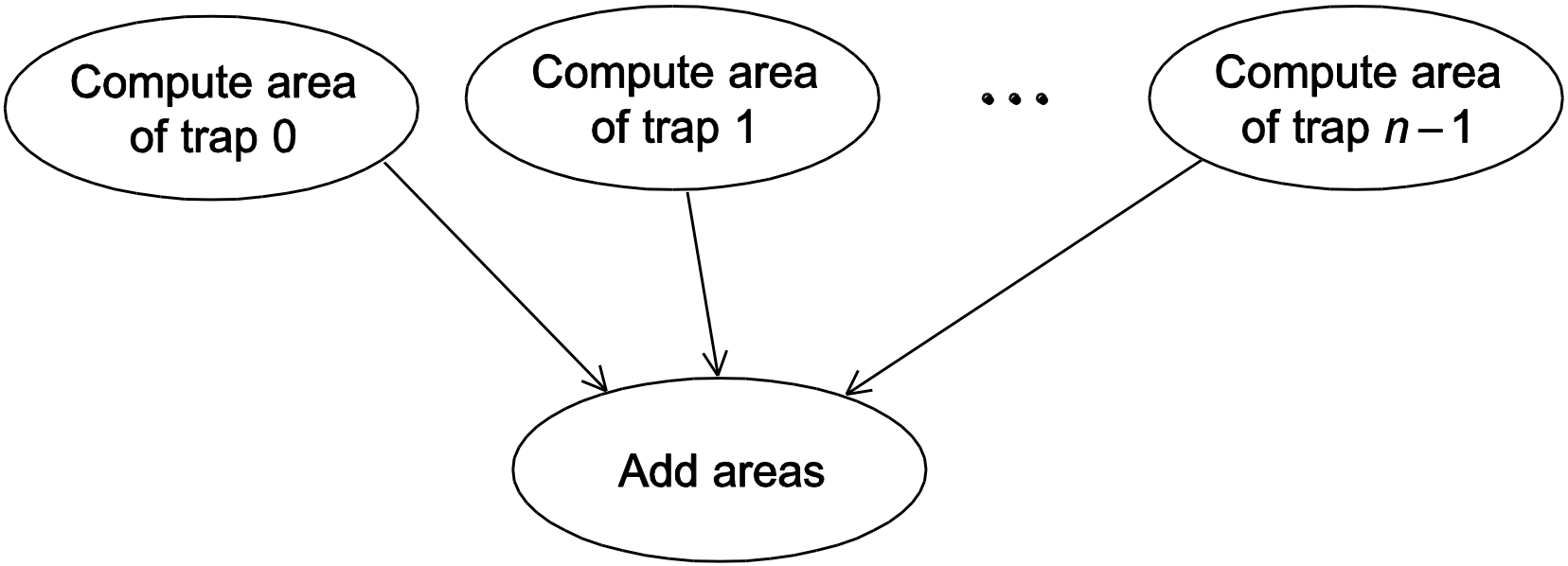
# Pseudo-code for a serial program

```
/* 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

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



# Parallel pseudo-code

```
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;
```

# First version (1)

```
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);
```

# First version (2)

```
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  */
```

# First version (3)

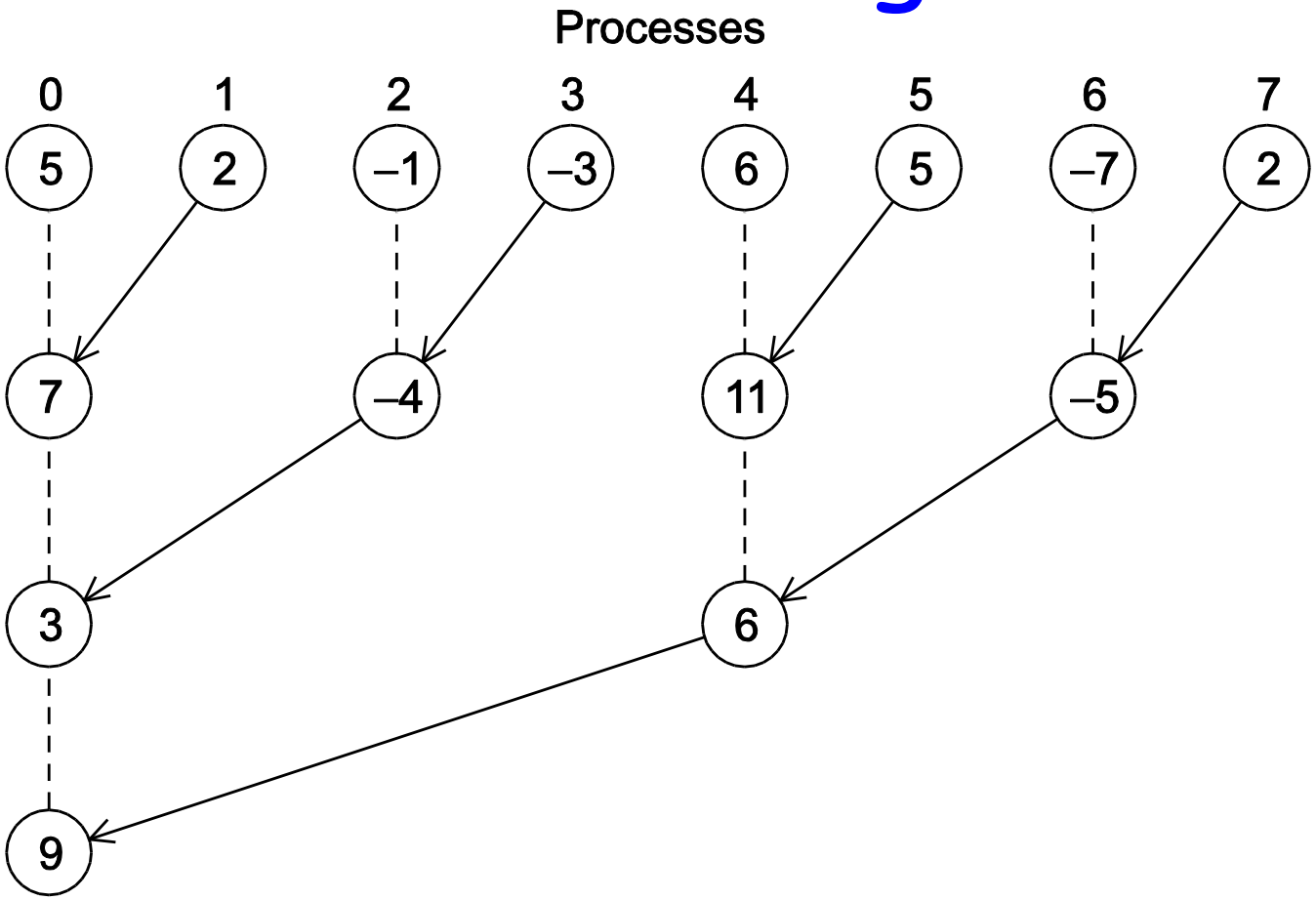
```
1  double Trap(
2        double left_endpt  /* in */,
3        double right_endpt /* in */,
4        int    trap_count  /* in */,
5        double base_len    /* in */) {
6     double estimate, x;
7     int i;
8
9     estimate = (f(left_endpt) + f(right_endpt))/2.0;
10    for (i = 1; i <= trap_count-1;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

# The Final Sum … Again!!

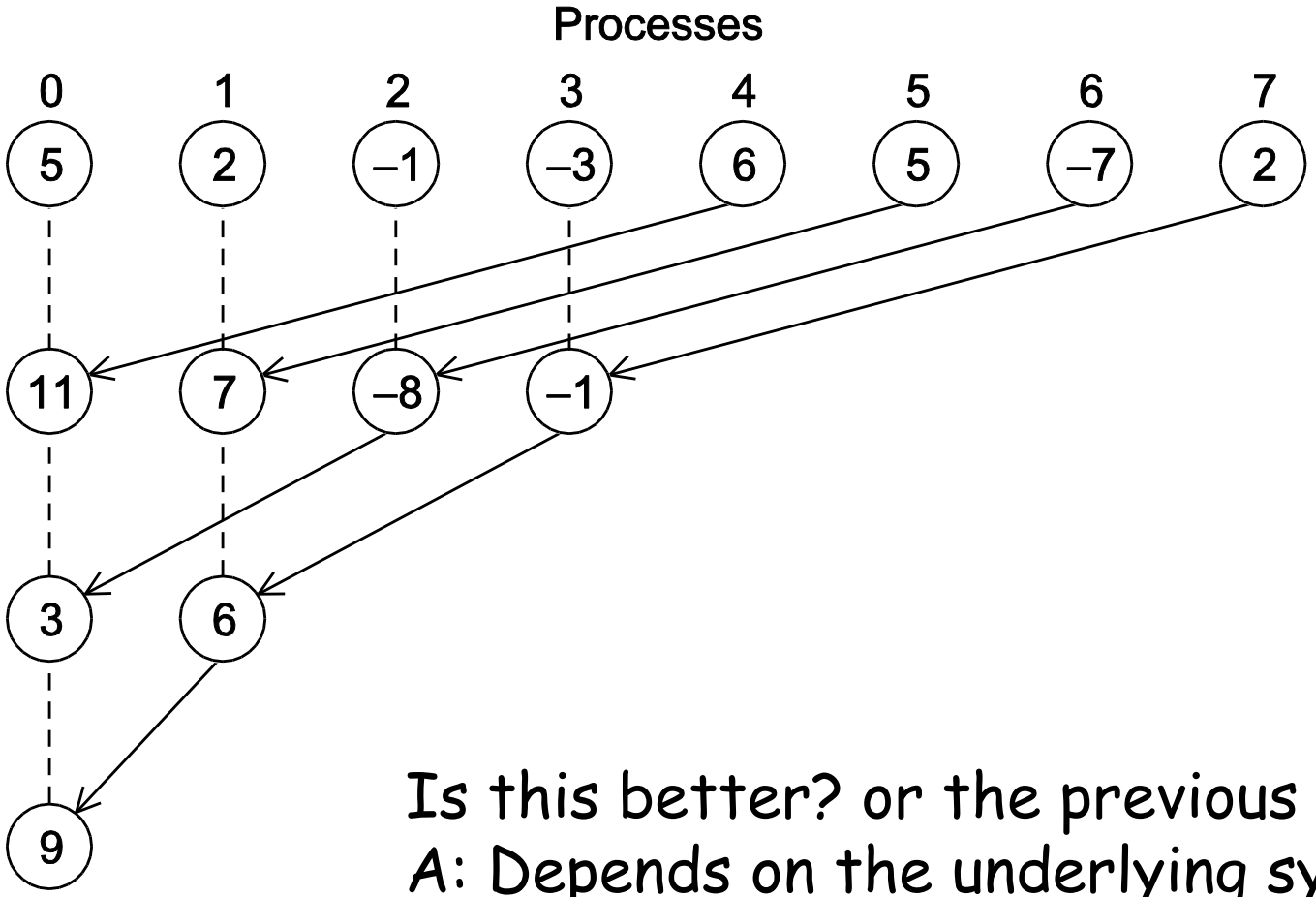
1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   (b) Processes 0, 2, 4, and 6 add in the received values.
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (d) Processes 0 and 4 add the received values into their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



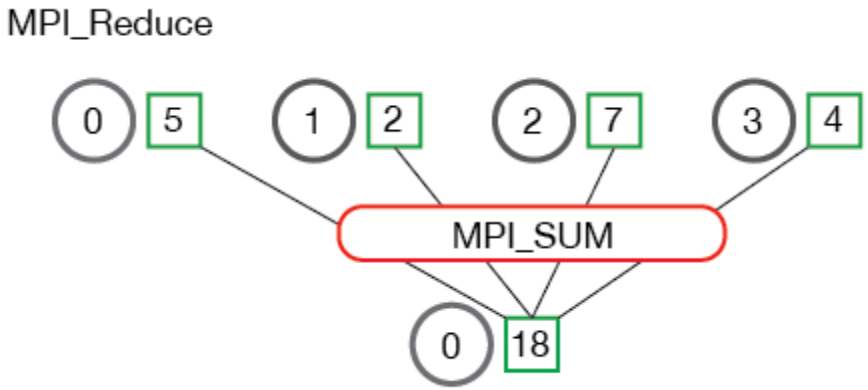
# An alternative tree-structured global sum



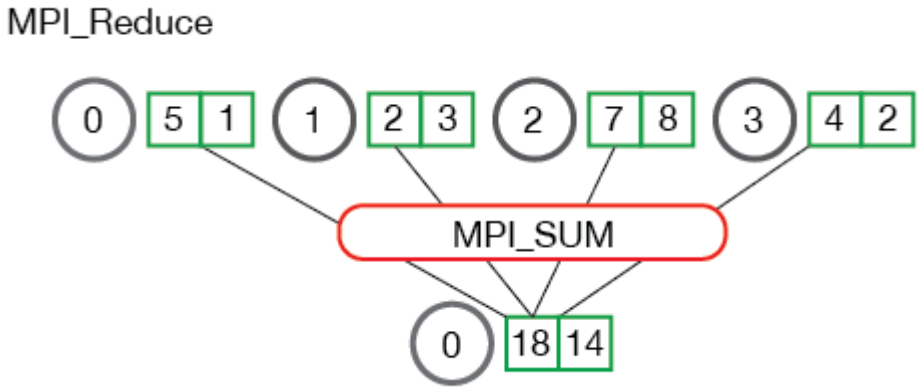

Is this better? or the previous one?
A: Depends on the underlying system!

# 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 → 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

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

only relevant
to dest_process

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

MPI_Reduce is called by all processes involved.
This is why it is called **collective call**.

# Predefined reduction operators in MPI

| Operation Value | Meaning |
|---|---|
| MPI_MAX | Maximum |
| MPI_MIN | Minimum |
| MPI_SUM | Sum |
| MPI_PROD | Product |
| MPI_LAND | Logical and |
| MPI_BAND | Bitwise and |
| MPI_LOR | Logical or |
| MPI_BOR | Bitwise or |
| MPI_LXOR | Logical exclusive or |
| MPI_BXOR | Bitwise exclusive or |
| MPI_MAXLOC | Maximum and location of maximum |
| MPI_MINLOC | Minimum and location of minimum |

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

| Time | Process 0 | Process 1 | Process 2 |
|---|---|---|---|
| 0 | `a = 1; c = 2` | `a = 1; c = 2` | `a = 1; c = 2` |
| 1 | `MPI_Reduce(&a, &b, ...)` | `MPI_Reduce(&c, &d, ...)` | `MPI_Reduce(&a, &b, ...)` |
| 2 | `MPI_Reduce(&c, &d, ...)` | `MPI_Reduce(&a, &b, ...)` | `MPI_Reduce(&c, &d, ...)` |

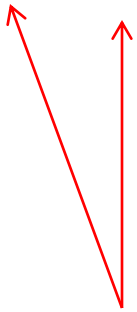
Assume:

- 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.

```
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!**

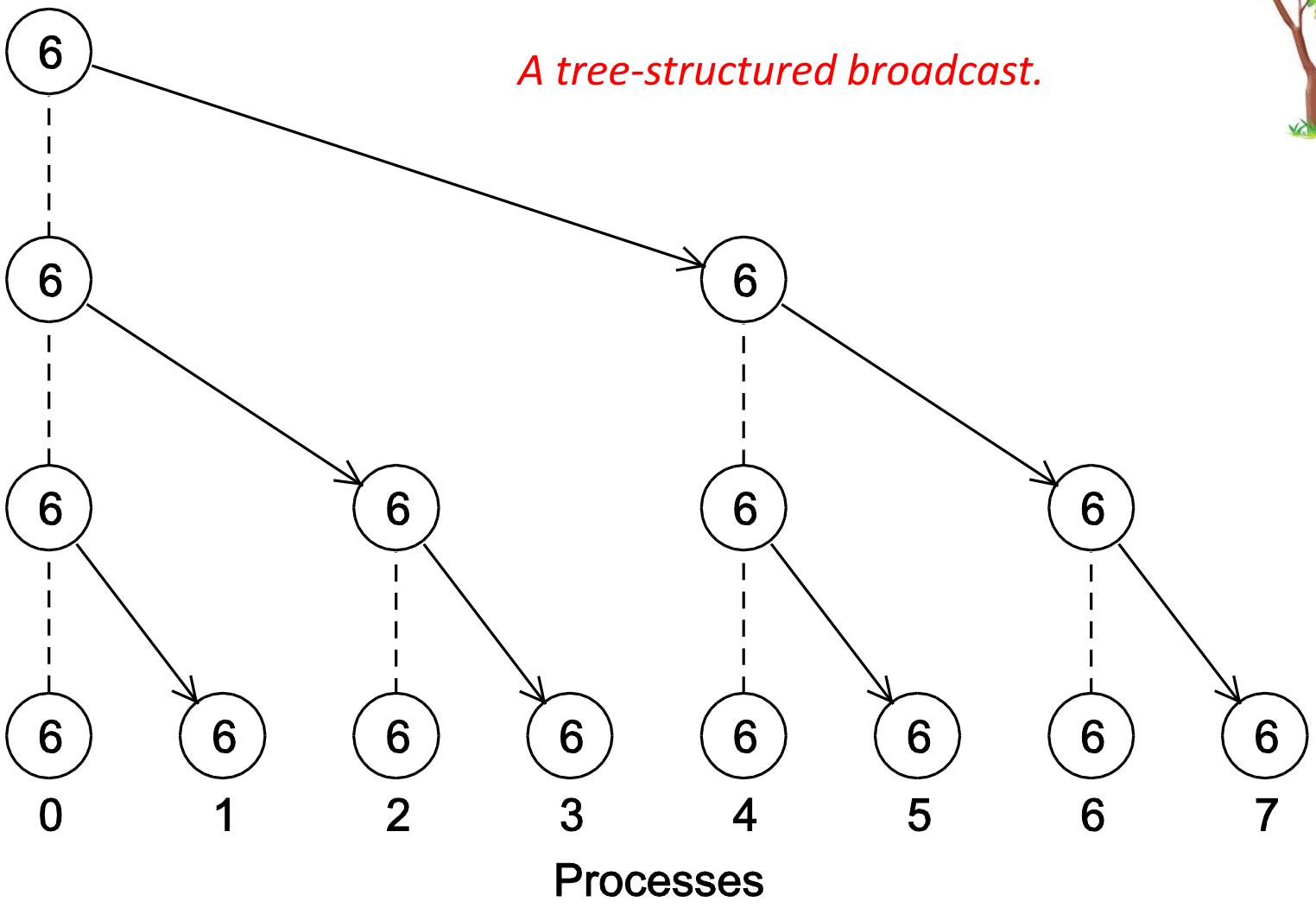
# Broadcast

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

```
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

```
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, … ).