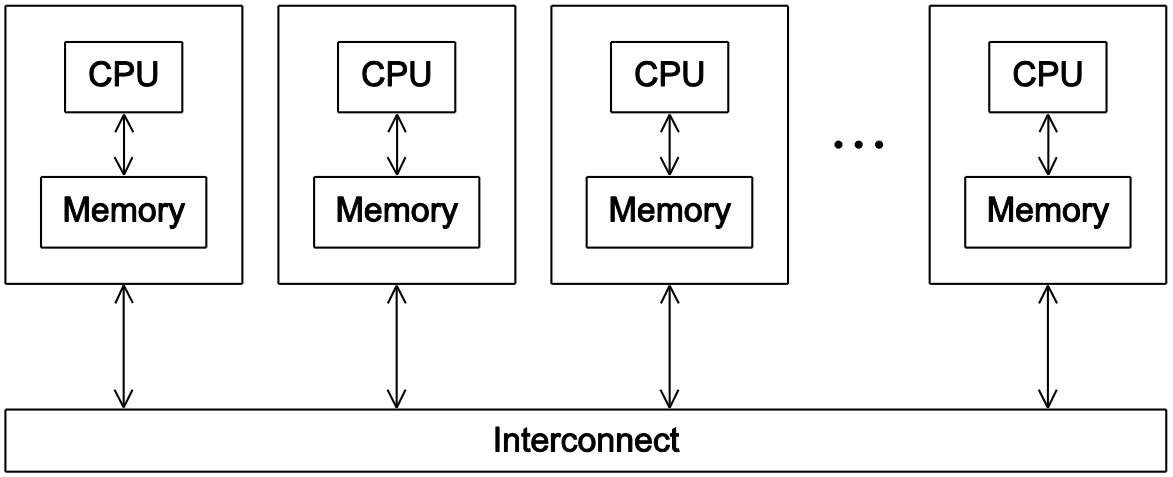
Distributed Memory Programming with MPI

Part 1

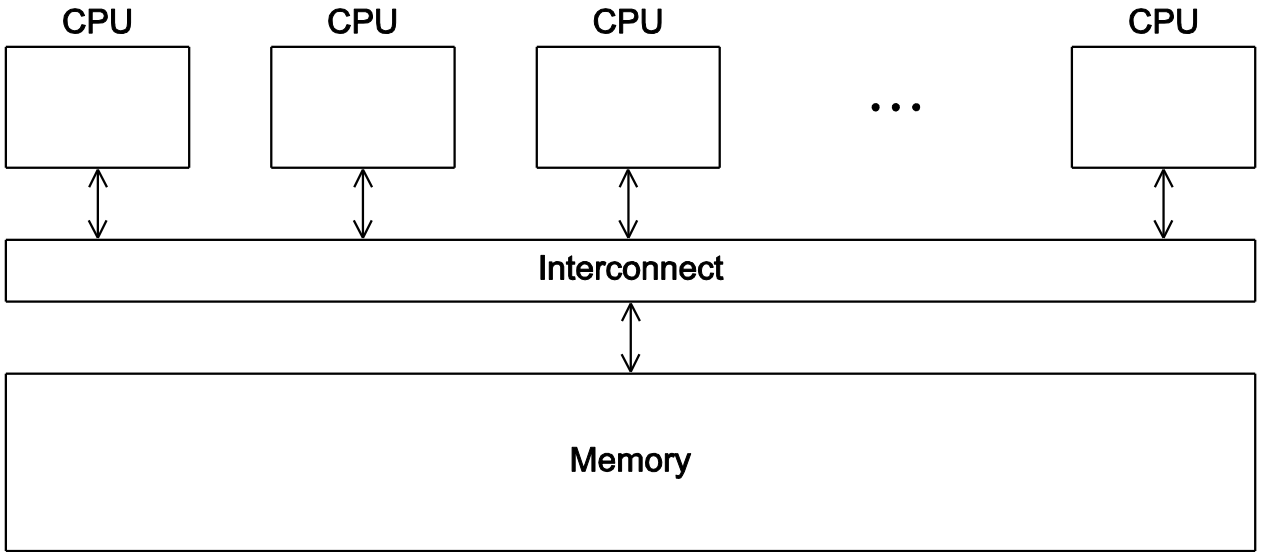
Bryan Mills, PhD

Spring 2017

A distributed memory system



A shared memory system

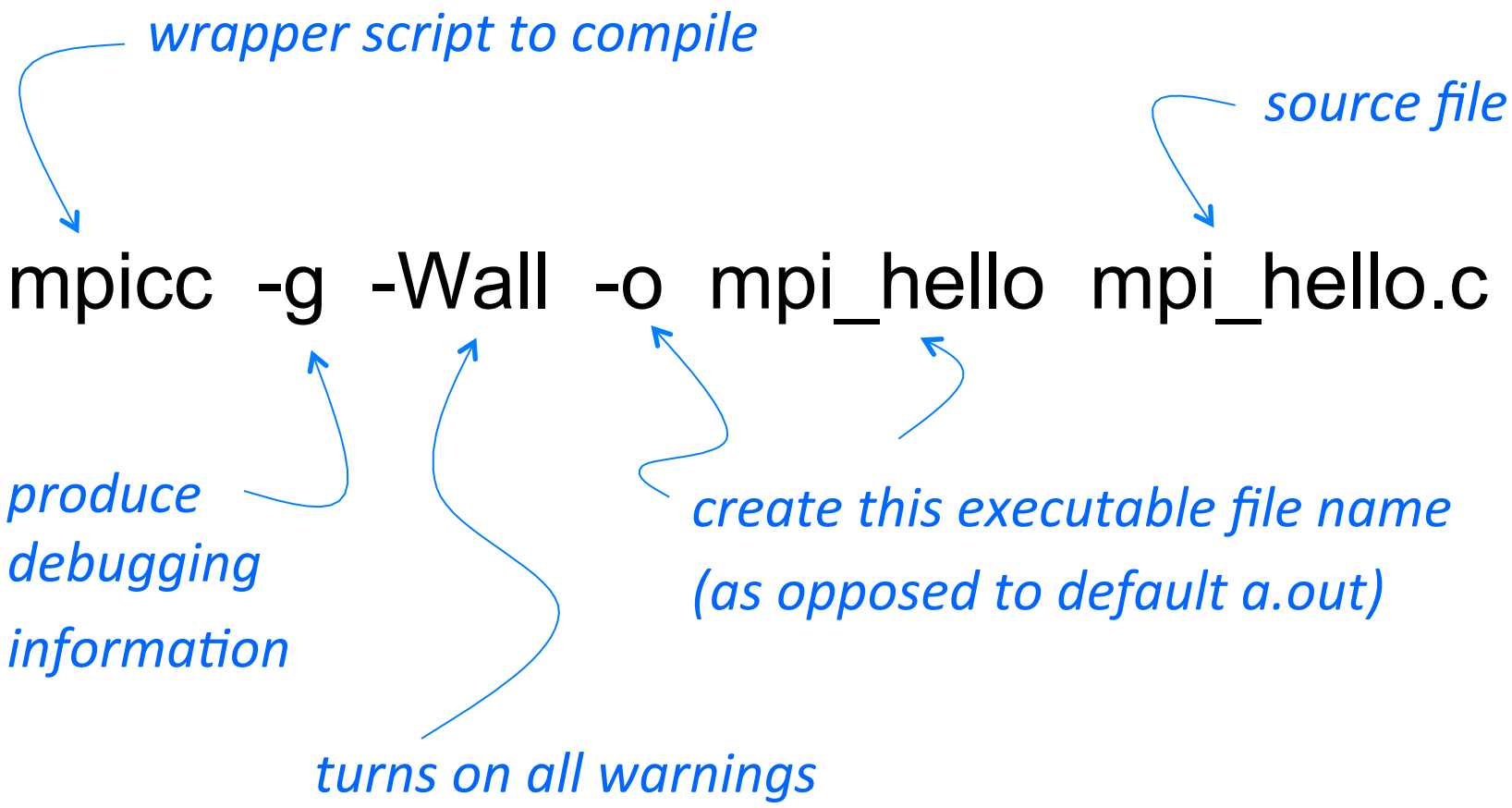


Identifying MPI processes

- Common practice to identify processes by nonnegative integer ranks.
- *p* processes are numbered *0, 1, 2, .. p-1*

```
MPI_Init(NULL, NULL);
/* Get the number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
/* Get my rank among all the processes */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank != 0) {
  /* Create message */
  sprintf(greeting, "Greetings from process %d of %d!",
          my_rank, comm_sz);
  /* Send message to process 0 */
  MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
           MPI_COMM_WORLD);
} else {
  /* Print my message */
  printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
  for (int q = 1; q < comm_sz; q++) {
    /* Receive message from process q */
    MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
             0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /* Print message from process q */
    printf("%s\n", greeting);
  }
}
/* Shut down MPI */
MPI_Finalize();
```

Compilation



Execution

Older versions of MPI used mpirun, some still do, but mpiexec is preferred

```
mpiexec  -n  <number of processes>  <executable>
```

```
mpiexec  -n  1  ./mpi_hello
```

run with 1 process

```
mpiexec  -n  4  ./mpi_hello
```

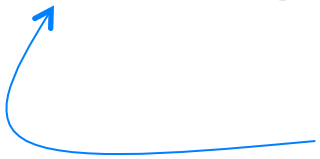
run with 4 processes

Execution on Comet

ibrun -n <number of processes> <executable>

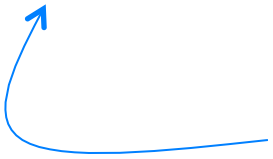
---

ibrun -n 16 ./mpi_hello



run with 16 process on 1 node

ibrun -n 64 ./mpi_hello



run with 64 processes on 4 nodes

Execution on Comet

- Using sbatch
- Using Interactive Debug:

```
srun  --partition=debug
      --pty
      --nodes=1
      --ntasks-per-node=24
      -t 00:30:00
      --wait=0
      --export=ALL
      /bin/bash
```

Execution on Comet

What if you want 8 tasks on your debug node?

---

ibrun –n 8 ./mpi_hello

Execution on Comet with Batch

- Using sbatch
- Using Interactive Debug:

```
srun  --partition=debug
      --pty
      --nodes=1
      --ntasks-per-node=24
      -t 00:30:00
      --wait=0
      --export=ALL
      /bin/bash
```

http://www.sdsc.edu/support/user_guides/comet.html

Execution

```
ibrun -n  1  ./mpi_hello

    Greetings from process 0 of 1 !
```

```
ibrun -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

- Written in C.
  - Has main.
  - Uses stdio.h, string.h, etc.
- Need to add mpi.h header file.
- Identifiers defined by MPI start with "MPI_".
- First letter following underscore is uppercase.
  - For function names and MPI-defined types.
  - Helps to avoid confusion.

MPI Components

- MPI_Init
  - Tells MPI to do all the necessary setup.

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

- MPI_Finalize
  - Tells MPI we're done, so clean up anything allocated for this program.

```
int MPI_Finalize(void);
```

Basic Outline

```
. . .
#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;
}
```

Communicators

- A collection of processes that can send messages to each other.
- MPI_Init defines a communicator that consists of all the processes created when the program is started.

- Called MPI_COMM_WORLD.

Communicators

```
int MPI_Comm_size(
    MPI_Comm comm  /* input */,
    int* comm_size /* output */)
```

number of processes in the communicator

```
int MPI_Comm_rank(
    MPI_Comm comm  /* input */,
    int* my_rank /* output */)
```

my rank
(the process making this call)

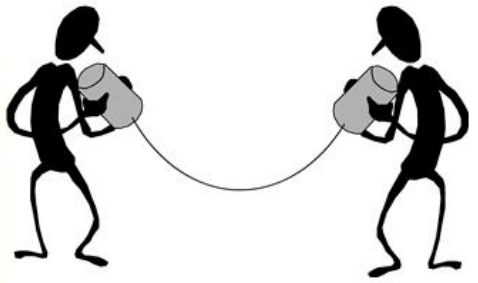
SPMD

- Single-Program Multiple-Data
- We compile one program.
- Process 0 does something different.
  - Receives messages and prints them while the other processes do the work.

- The if-else construct makes our program SPMD.

Communication

```
int MPI_Send(
      void*         msg_buf  /* input */,
      int           msg_size /* input */,
      MPI_Datatype  msg_type /* input */,
      int           dest     /* input */,
      int           tag      /* input */,
      MPI_Comm      comm     /* input */)
```

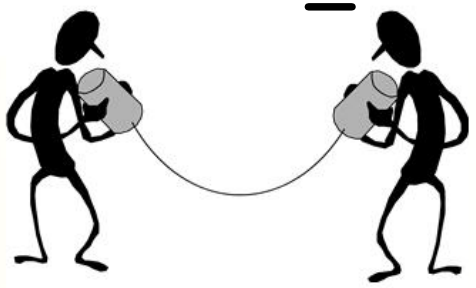


Data types

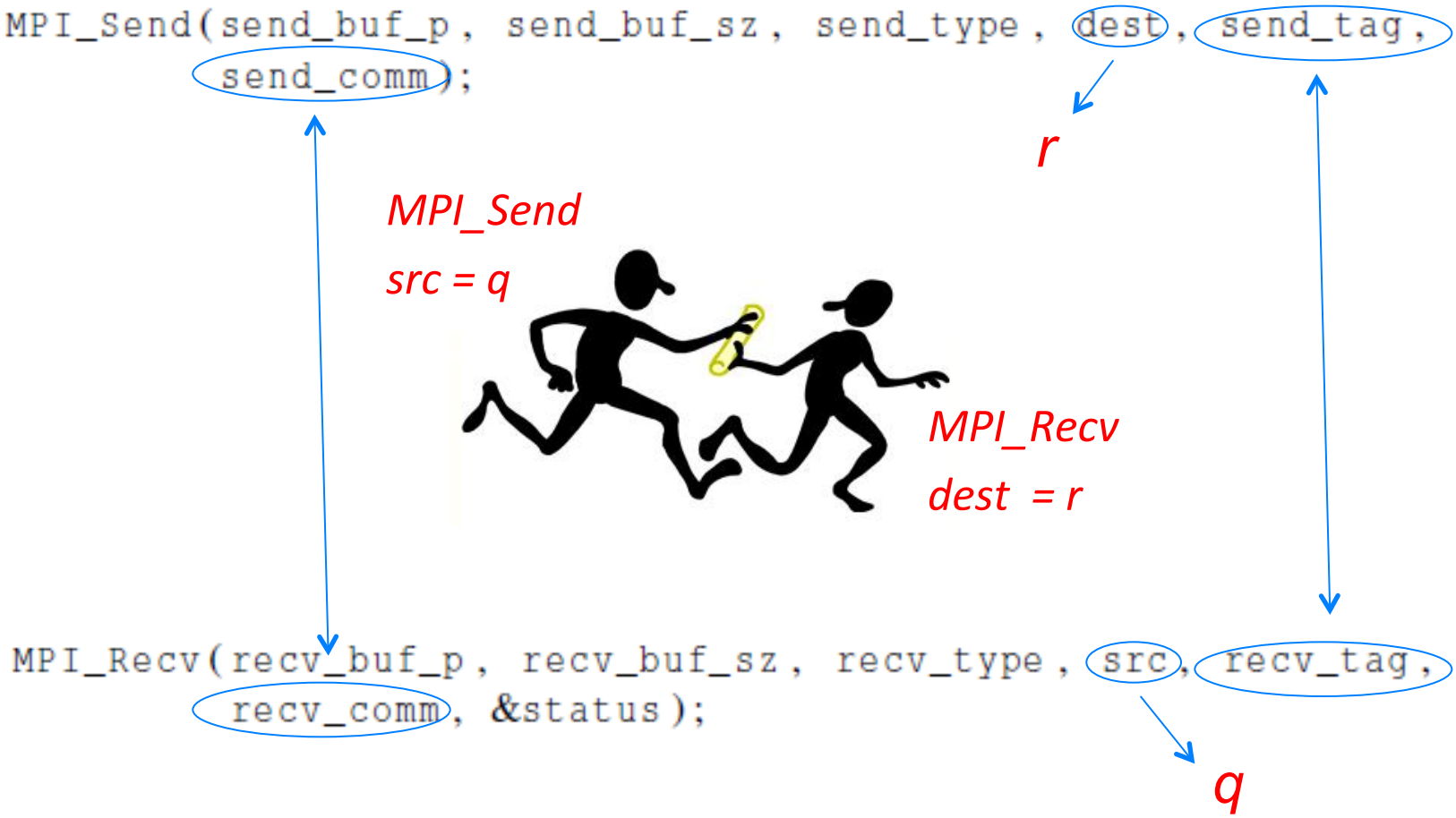
MPI datatype	C datatype
MPI_CHAR	signed **char**
MPI_SHORT	signed **short int**
MPI_INT	signed **int**
MPI_LONG	signed **long int**
MPI_LONG_LONG	signed **long long int**
MPI_UNSIGNED_CHAR	**unsigned char**
MPI_UNSIGNED_SHORT	**unsigned short int**
MPI_UNSIGNED	**unsigned int**
MPI_UNSIGNED_LONG	**unsigned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

Communication

```
int MPI_Recv(
      void*         msg_buf  /* output */,
      int           msg_size /* input */,
      MPI_Datatype  msg_type /* input */,
      int           source   /* input */,
      int           tag      /* input */,
      MPI_Comm      comm     /* input */,
      MPI_Status*   status_p /* output */)
```



Message matching

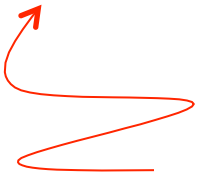


Receiving messages

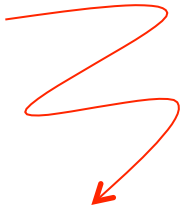
- A receiver can get a message without knowing:
  - the amount of data in the message,
  - the sender of the message,
  - or the tag of the message.

status_p argument

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



MPI_Status*



MPI_SOURCE

MPI_TAG

MPI_ERROR

MPI_Status* status;

status.MPI_SOURCE

status.MPI_TAG

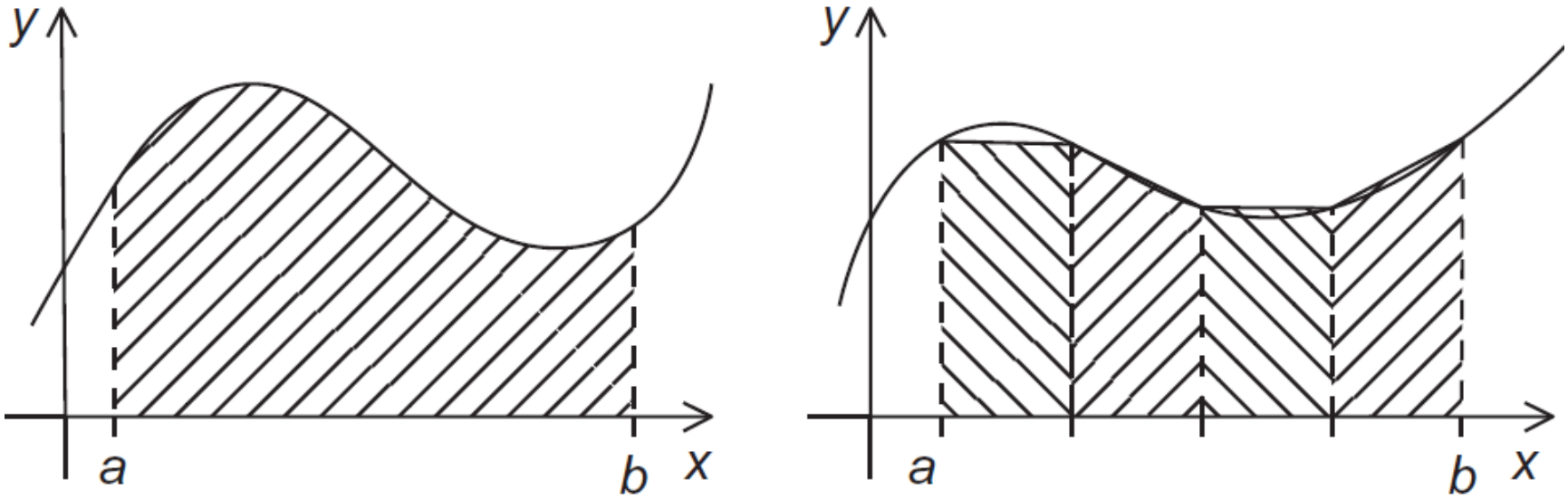
How much data am I receiving?

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

Issues with send and receive

- Exact behavior is determined by the MPI implementation.
- MPI_Send may behave differently with regard to buffer size, cutoffs and blocking.
- MPI_Recv always blocks until a matching message is received.
- Know your implementation; don't make assumptions!

The trapezoidal rule




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

The Trapezoidal Rule

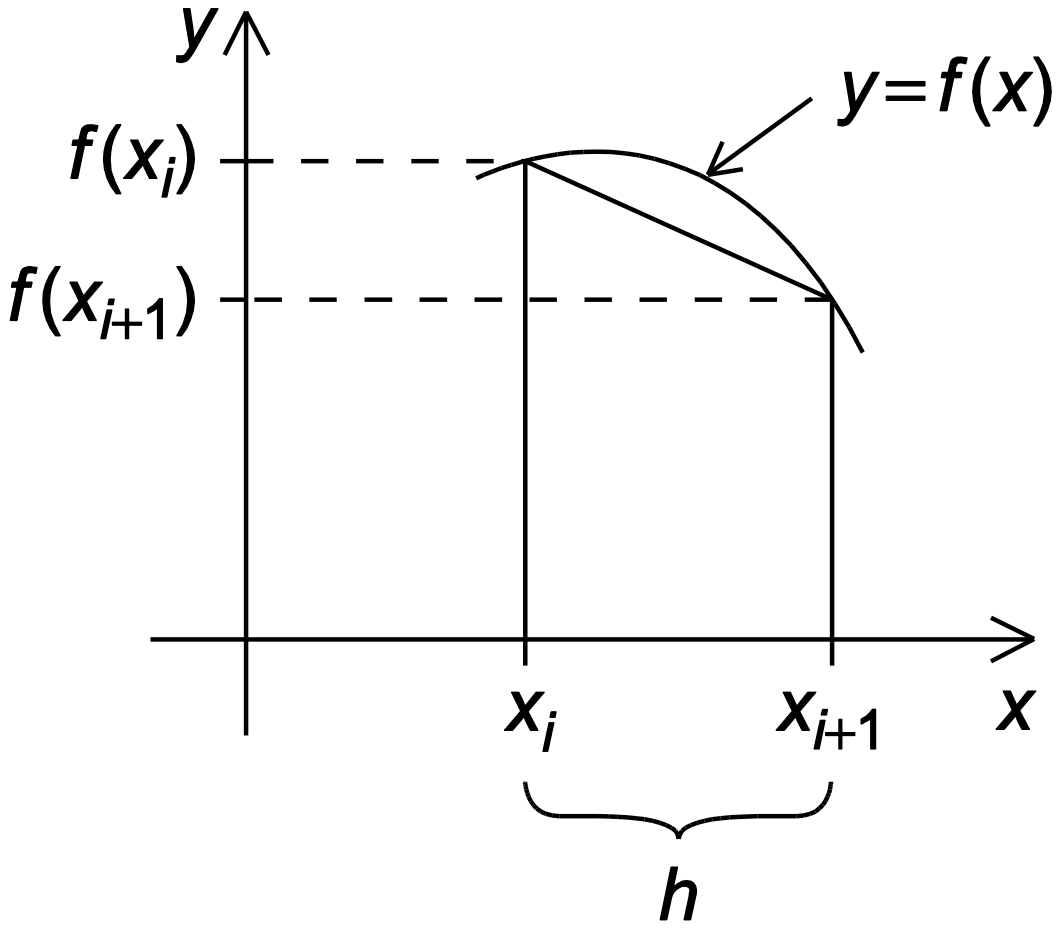
$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

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

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

One trapezoid



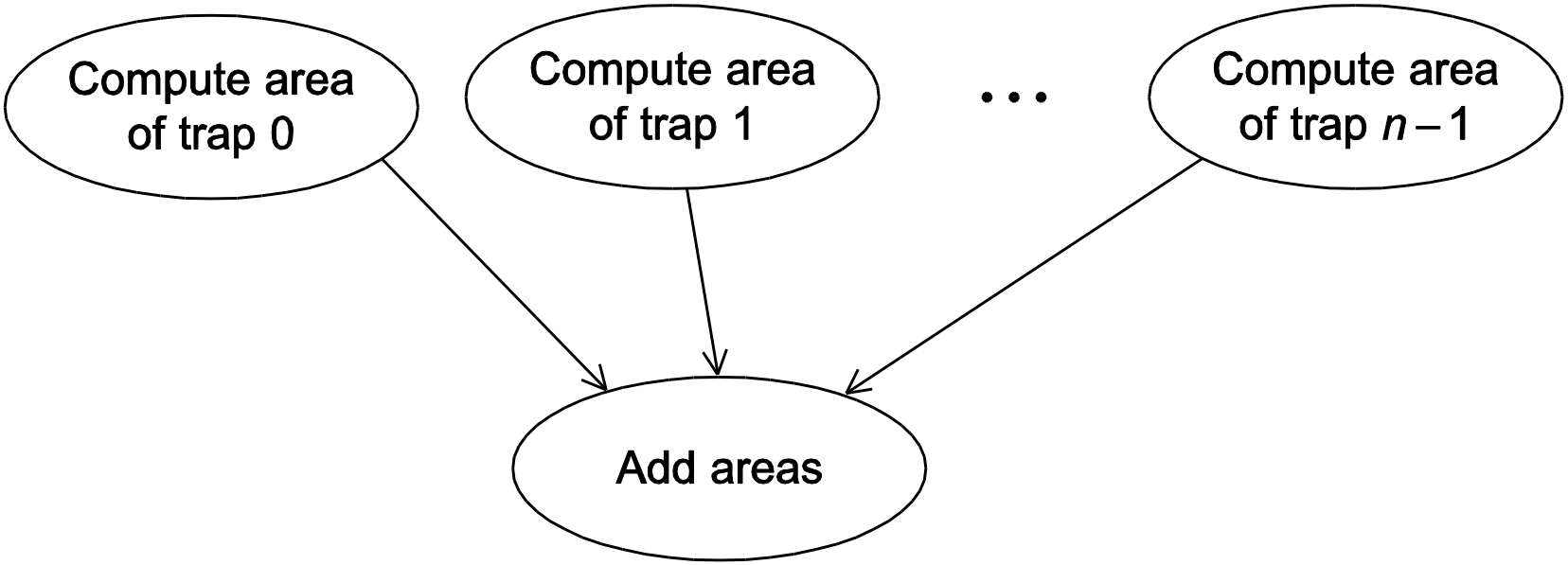
Parallelizing the Trapezoidal Rule

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

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;
```

Tasks and communications for Trapezoidal Rule



Dealing with output

```
int main(void) {
  char      greeting[MAX_STRING];  /* String storing message */
  int       comm_sz;               /* Number of processes    */
  int       my_rank;               /* My process rank        */

  /* Start up MPI */
  MPI_Init(NULL, NULL);

  /* Get the number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

  /* Get my rank among all the processes */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  printf("Hello from %d out of %d\n", my_rank, comm_sz);

  /* Shut down MPI */
  MPI_Finalize();

  return 0;
}  /* main */
```

Each process just prints a message.

Input

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

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

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

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

Dealing with input

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

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     */);
```

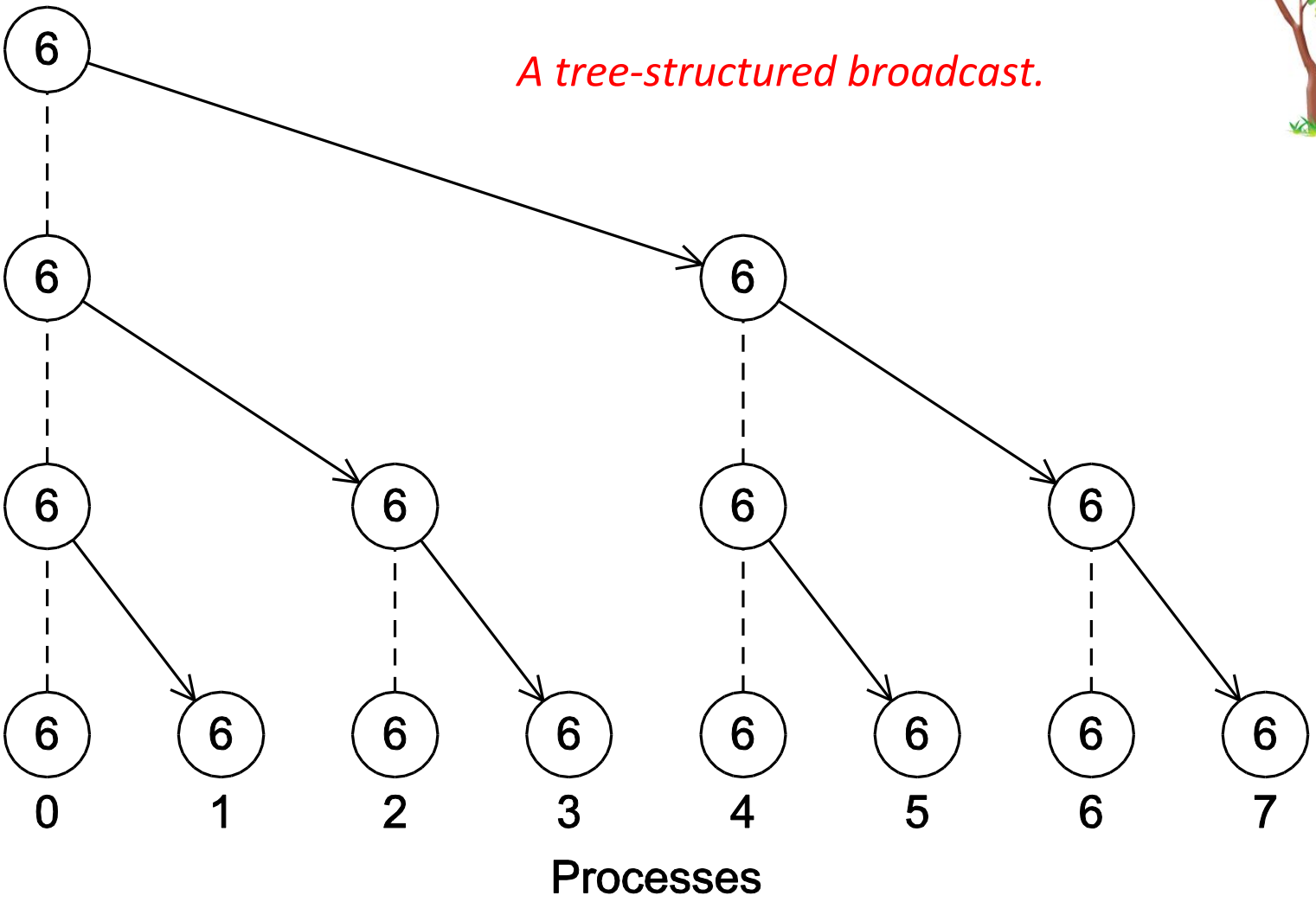
Comments on Broadcast

- All collective operations must be called by *all* processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI_Bcast is not a “multi-send”
  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

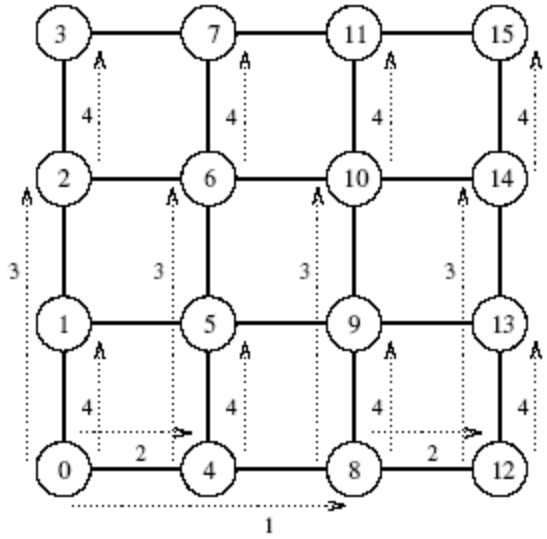
Recall our Simple Broadcast

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

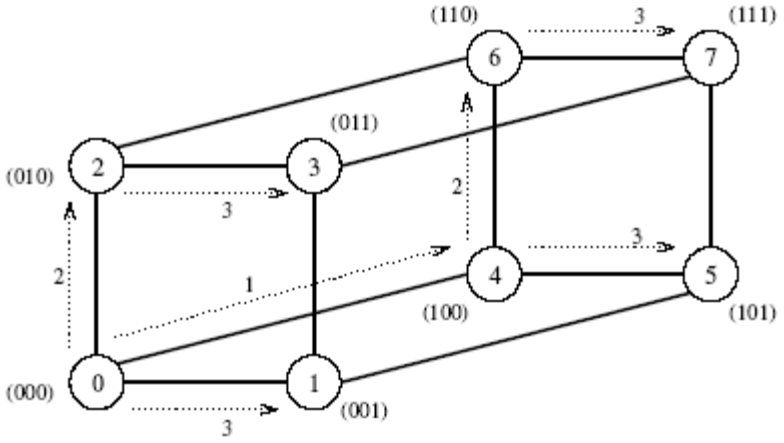
A tree-structured broadcast.



Mesh structured broadcast.



Hypercube 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 */
```

MPI_Bcast for Input

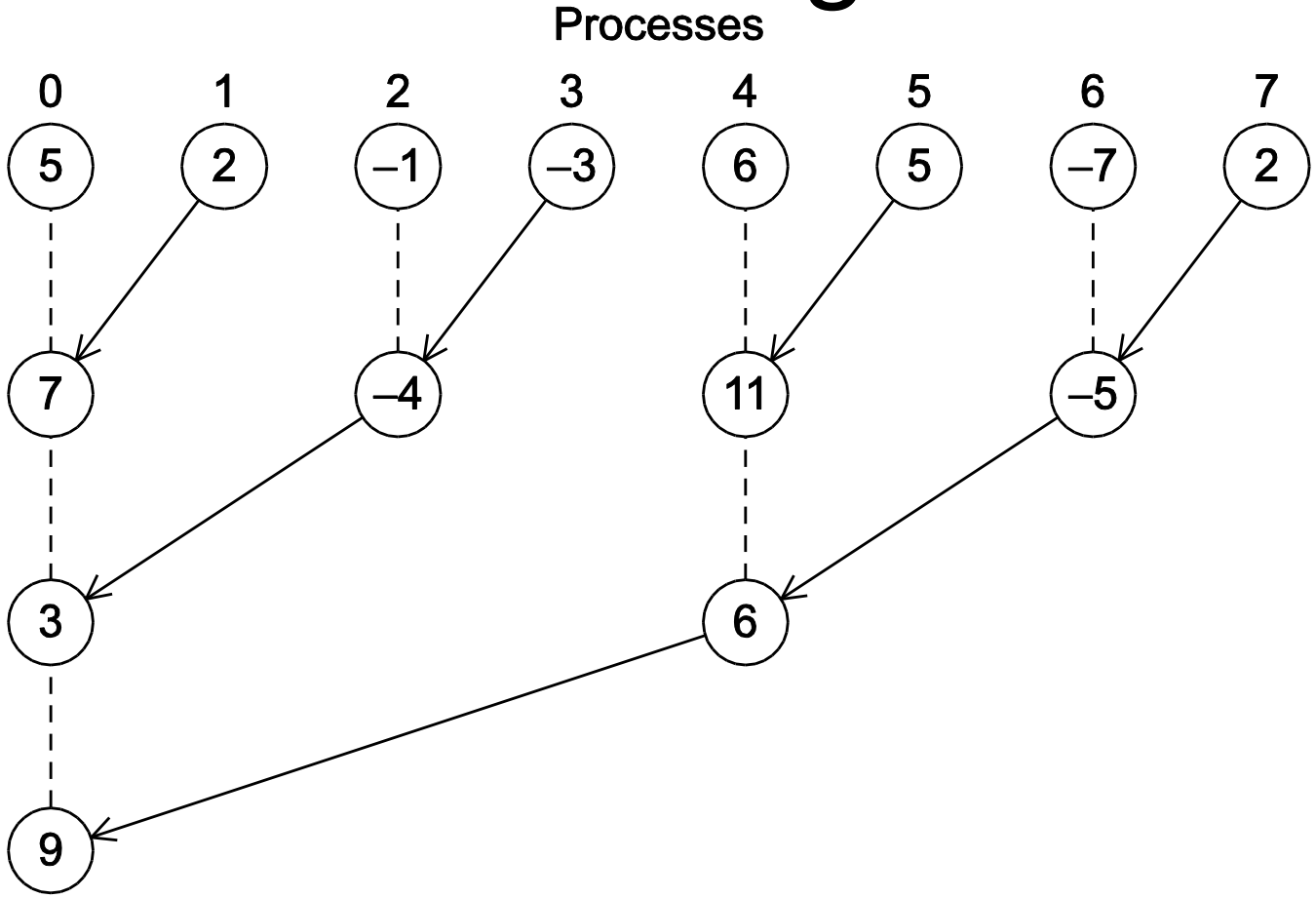
```
void Get_input(int my_rank, int comm_sz, double* a_p, double* b_p,
      int* n_p) {

   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 */
```

Other Places for Tree-Structure?

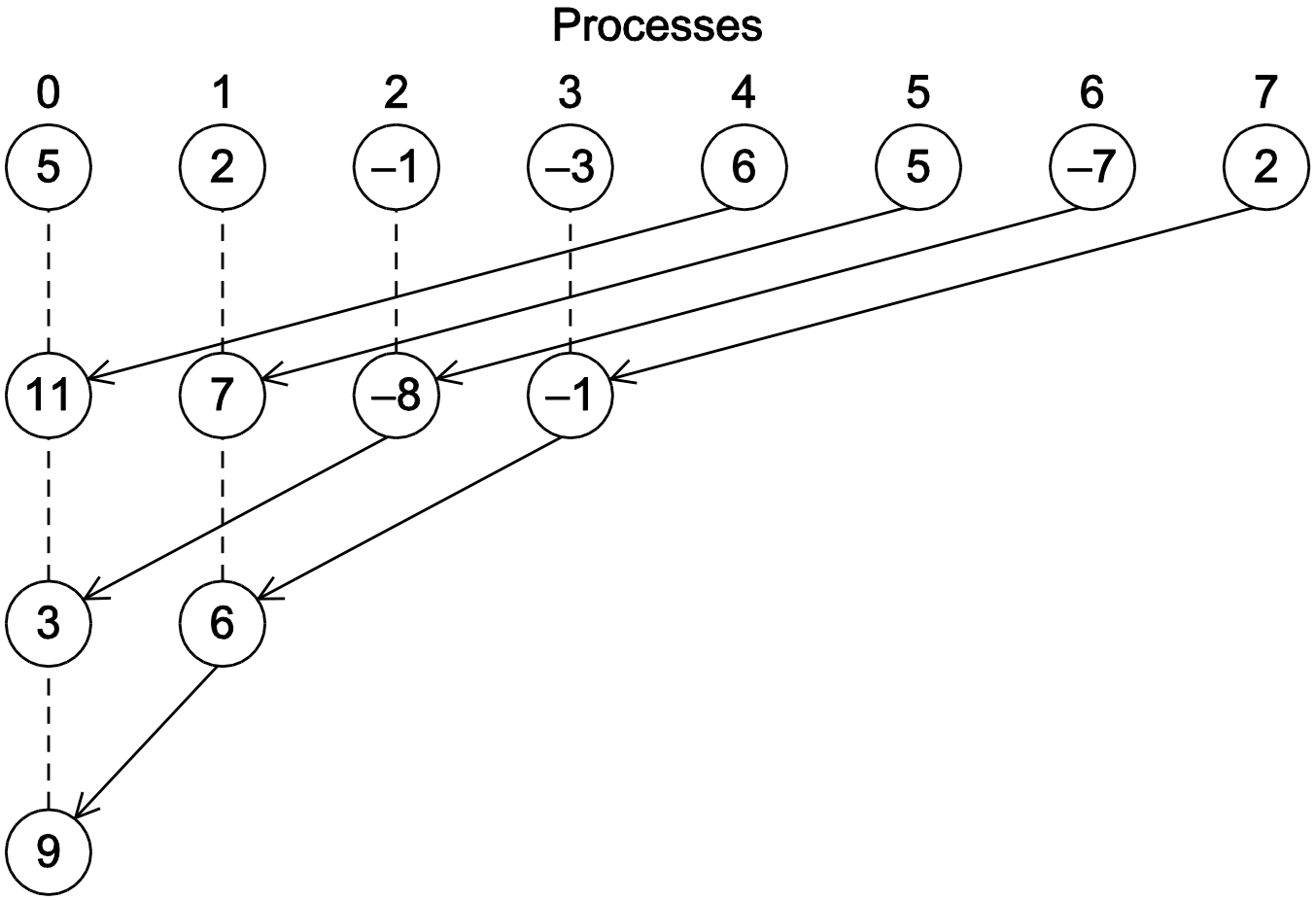
1. In the first phase:
   (a) Process 1 sends to 0, 3 sends to 2, 5 sends to 4, and 7 sends to 6.
   (b) Processes 0, 2, 4, and 6 add in the received values.
   (c) Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   (d) Processes 0 and 4 add the received values into their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



MPI_Reduce

```
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  */);
```

```
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);
```

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

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, and, in all likelihood, the program will hang or crash.

Collective vs. Point-to-Point Communications

- 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, and, once again, the program is likely to hang or crash.

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.

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, ...)`

Multiple calls to MPI_Reduce

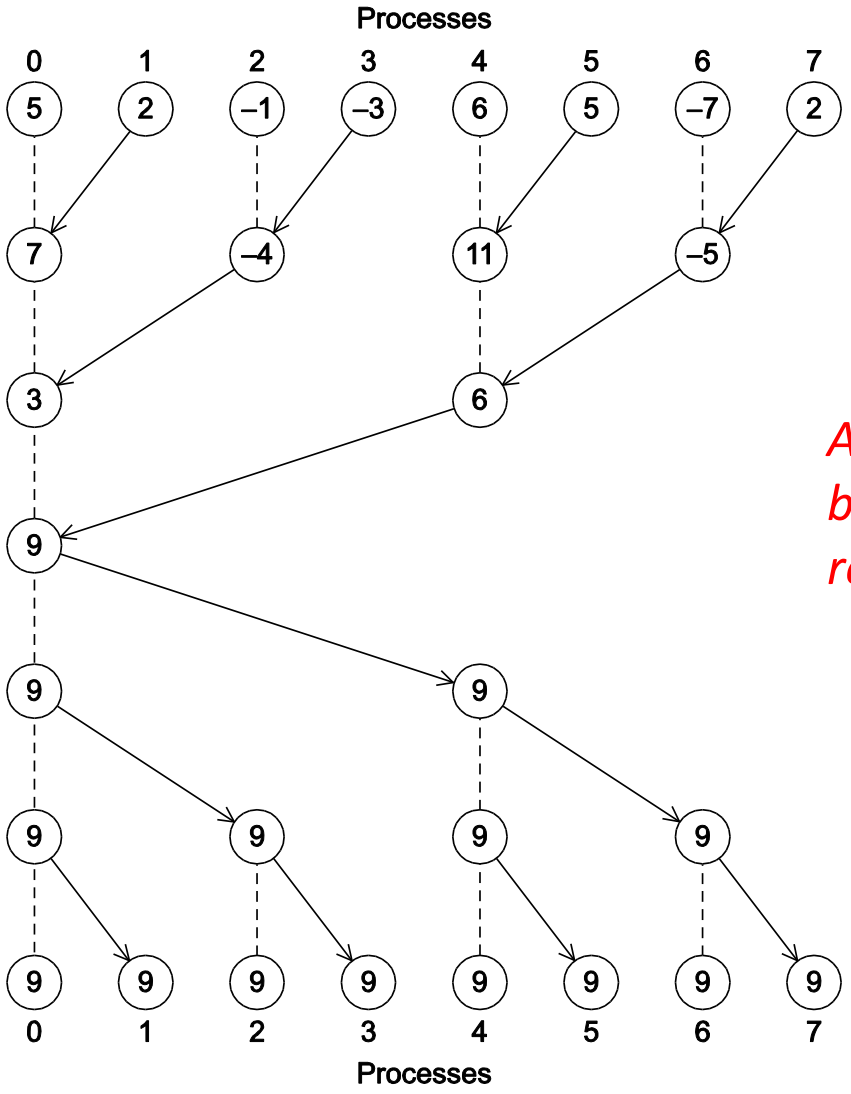
Example

- Suppose that each process calls MPI_Reduce with operator MPI_SUM, and destination process 0.
- At first glance, it might seem that after the two calls to MPI_Reduce, the value of b will be 3, and the value of d will be 6.
- However, the names of the memory locations are irrelevant to the matching of the calls to MPI_Reduce.
- The order of the calls will determine the matching so the value stored in b will be 1+2+1 = 4, and the value stored in d will be 2+1+2 = 5.

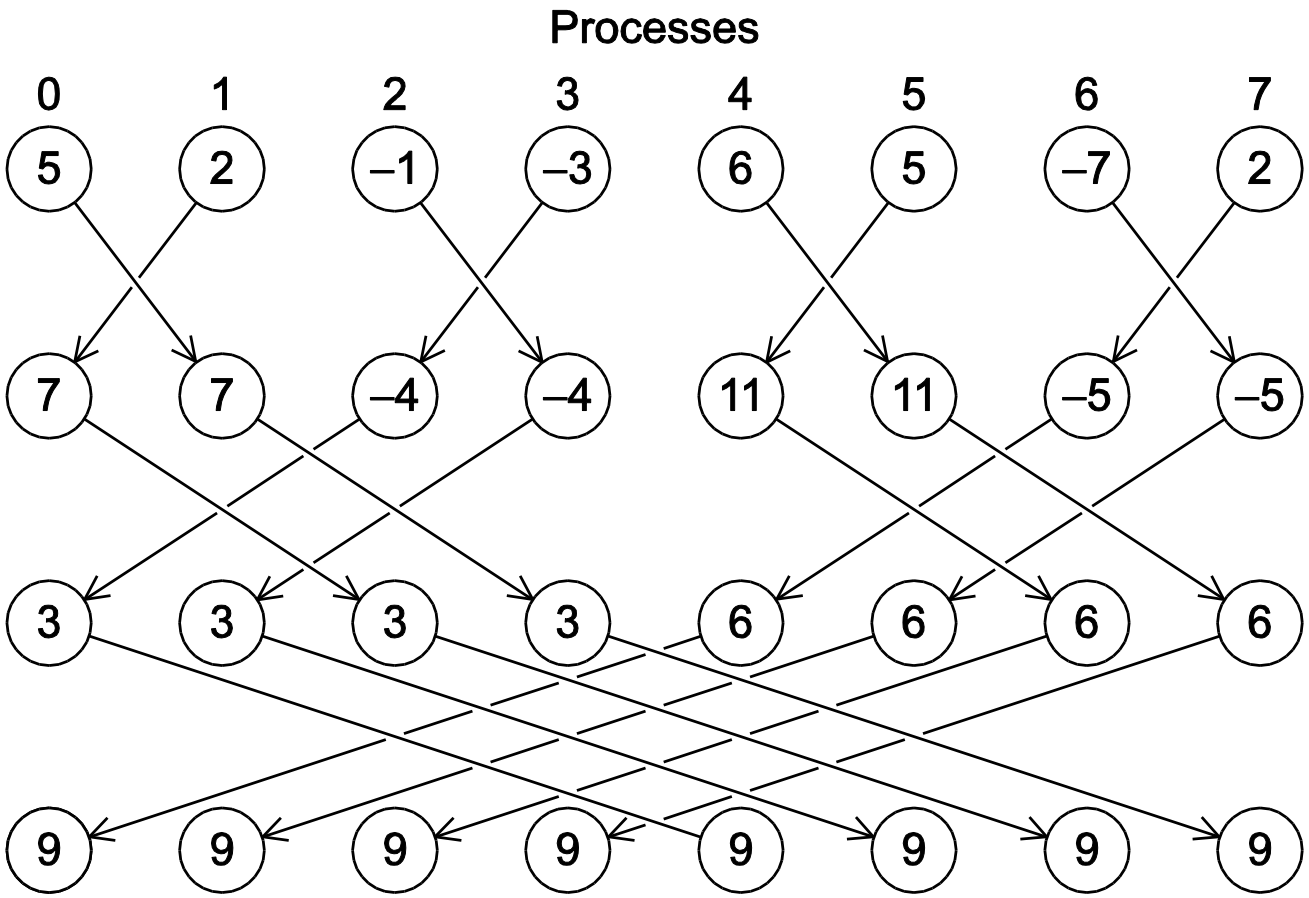
MPI_Allreduce

- 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  */);
```



A global sum followed by distribution of the result.



A butterfly-structured global sum.