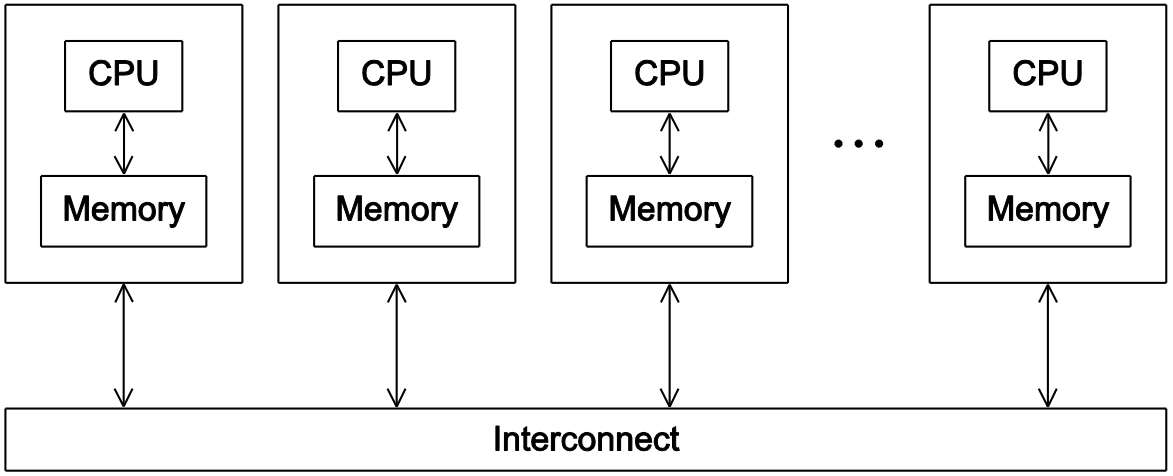
Distributed Memory Programming with MPI

Part 1

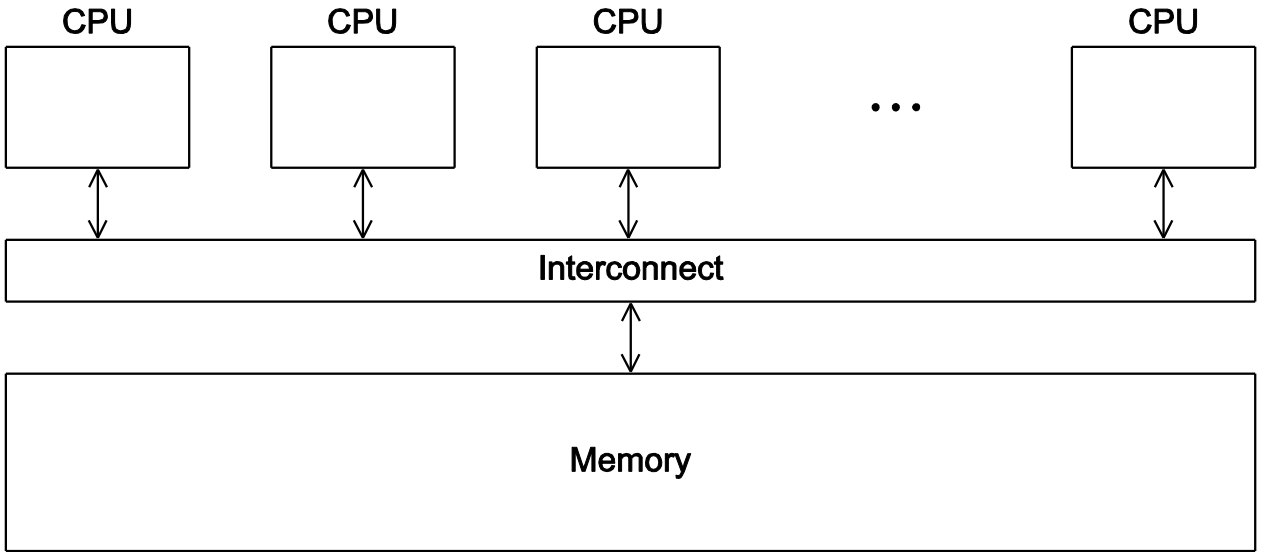
Bryan Mills, PhD

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

wrapper script to compile

source file

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

produce debugging information

create this executable file name (as opposed to default a.out)

turns on all warnings

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

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)

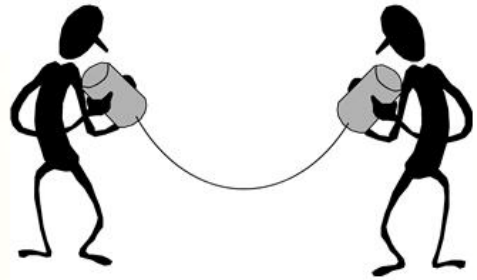
SPMD

- Single-Program Multiple-Data
- We compile <u>one</u> 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 */)
```

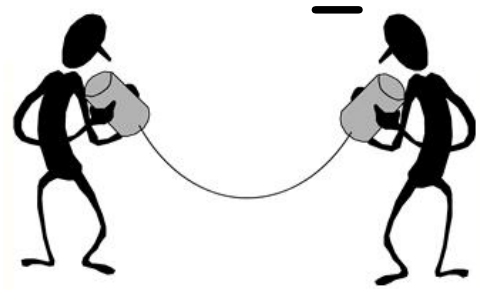


Data types

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

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

r

MPI_Send

src = q

MPI_Recv

dest = r

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

q

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

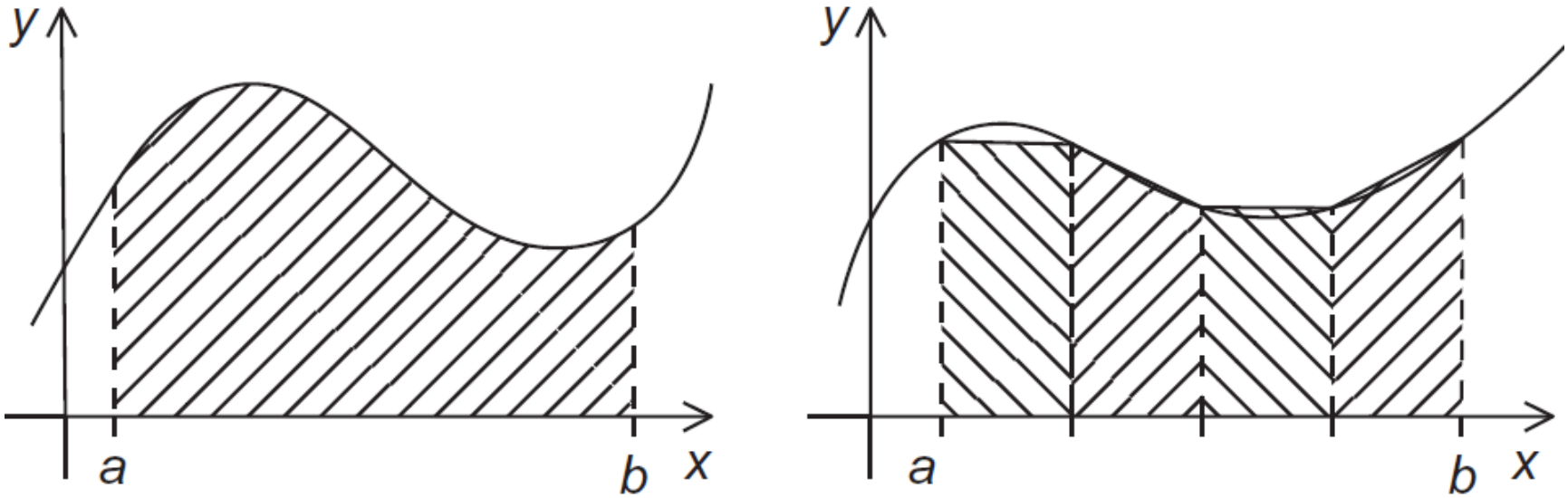
How much data am I receiving?

```
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

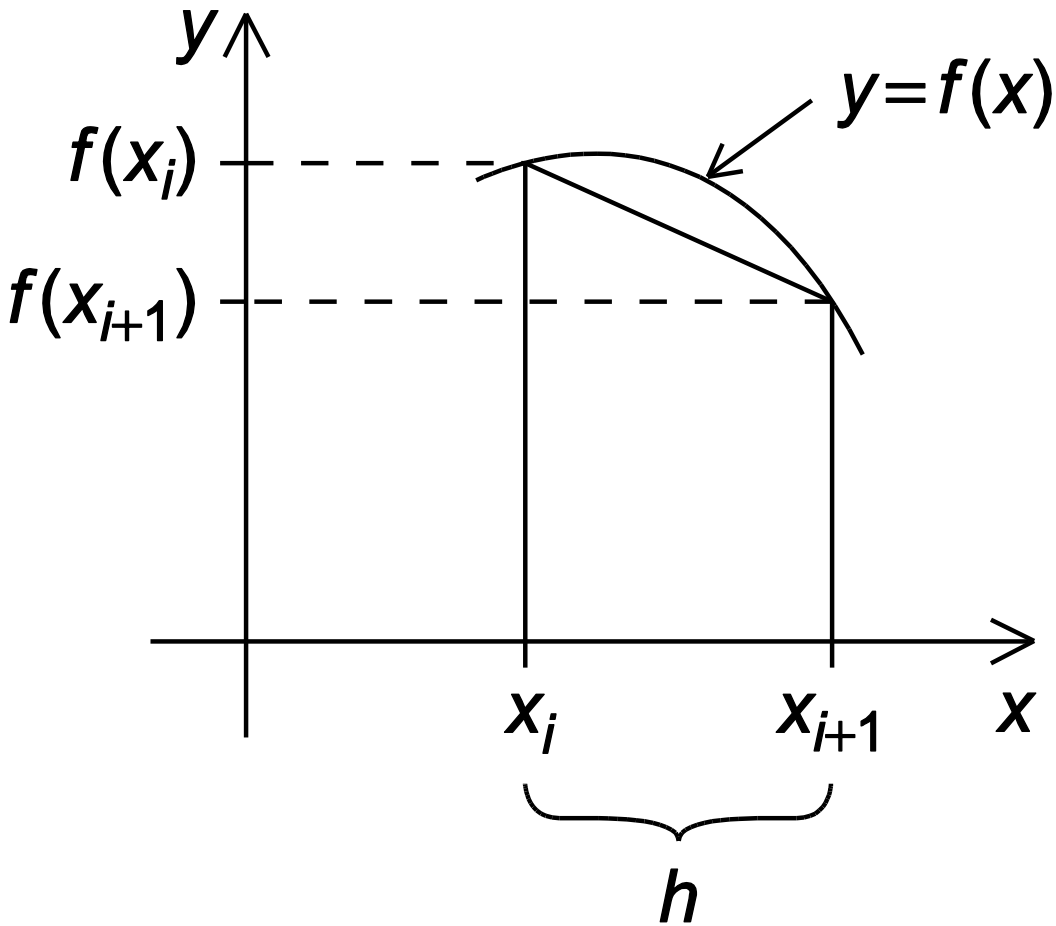
$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

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

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

One trapezoid



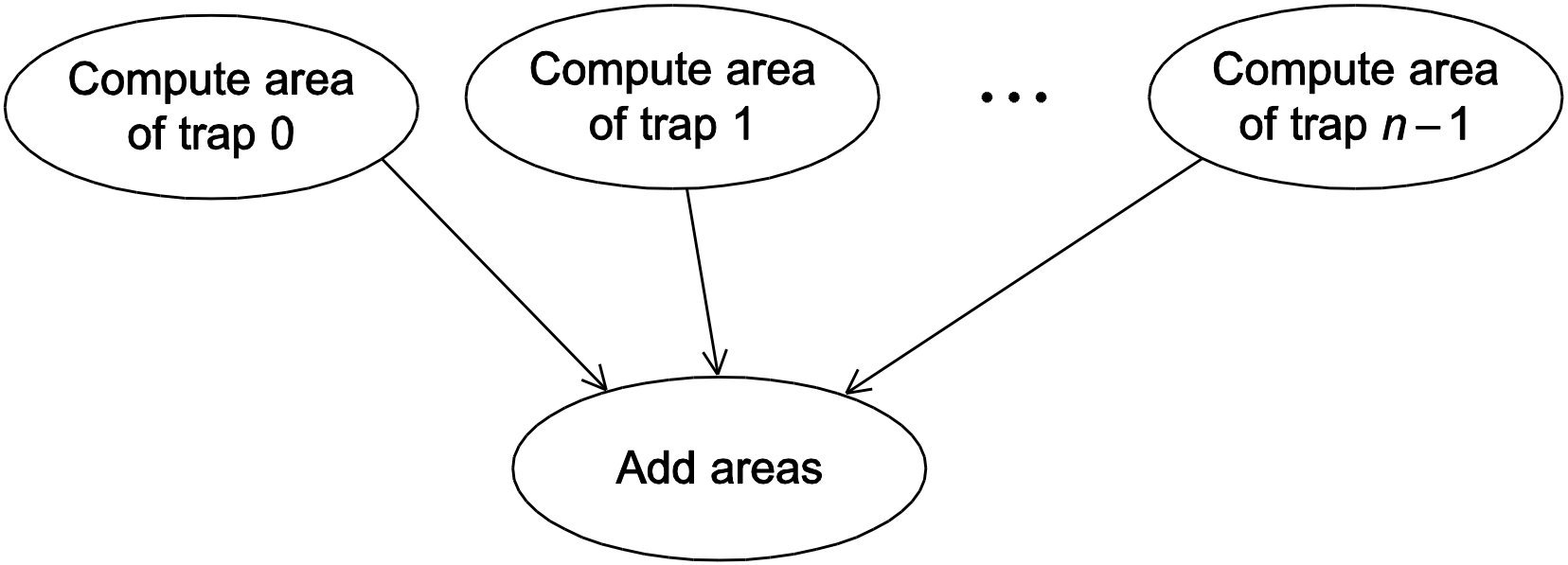
Parallelizing the Trapezoidal Rule

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

```
Get a, b, n;
h = (b−a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
   Send local_integral to process 0;
else /* my_rank == 0 */
   total_integral = local_integral;
   for (proc = 1; proc < comm_sz; proc++) {
      Receive local_integral from proc;
      total_integral += local_integral;
   }
}
if (my_rank == 0)
   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     */);
```

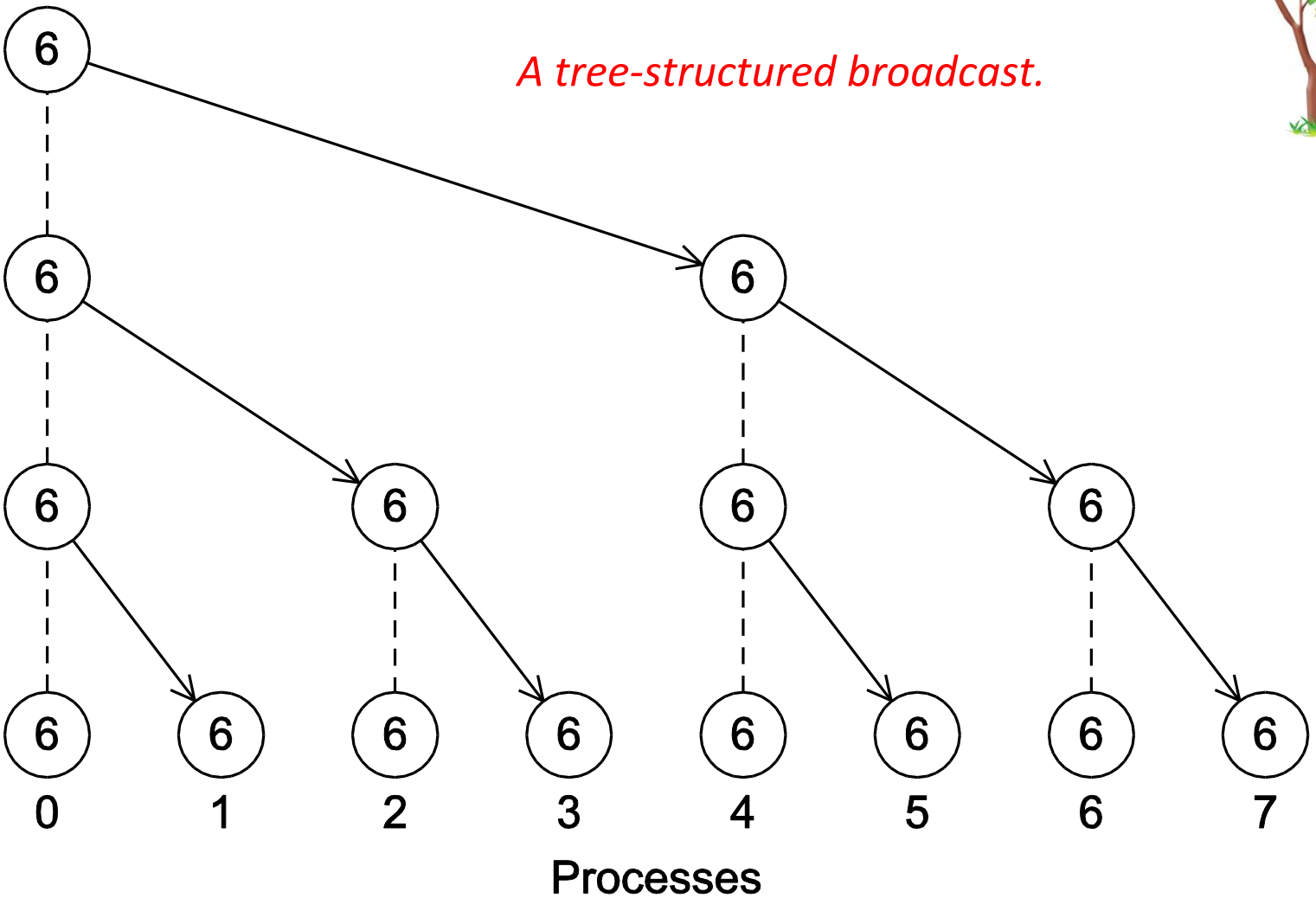
Comments on Broadcast

- 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.



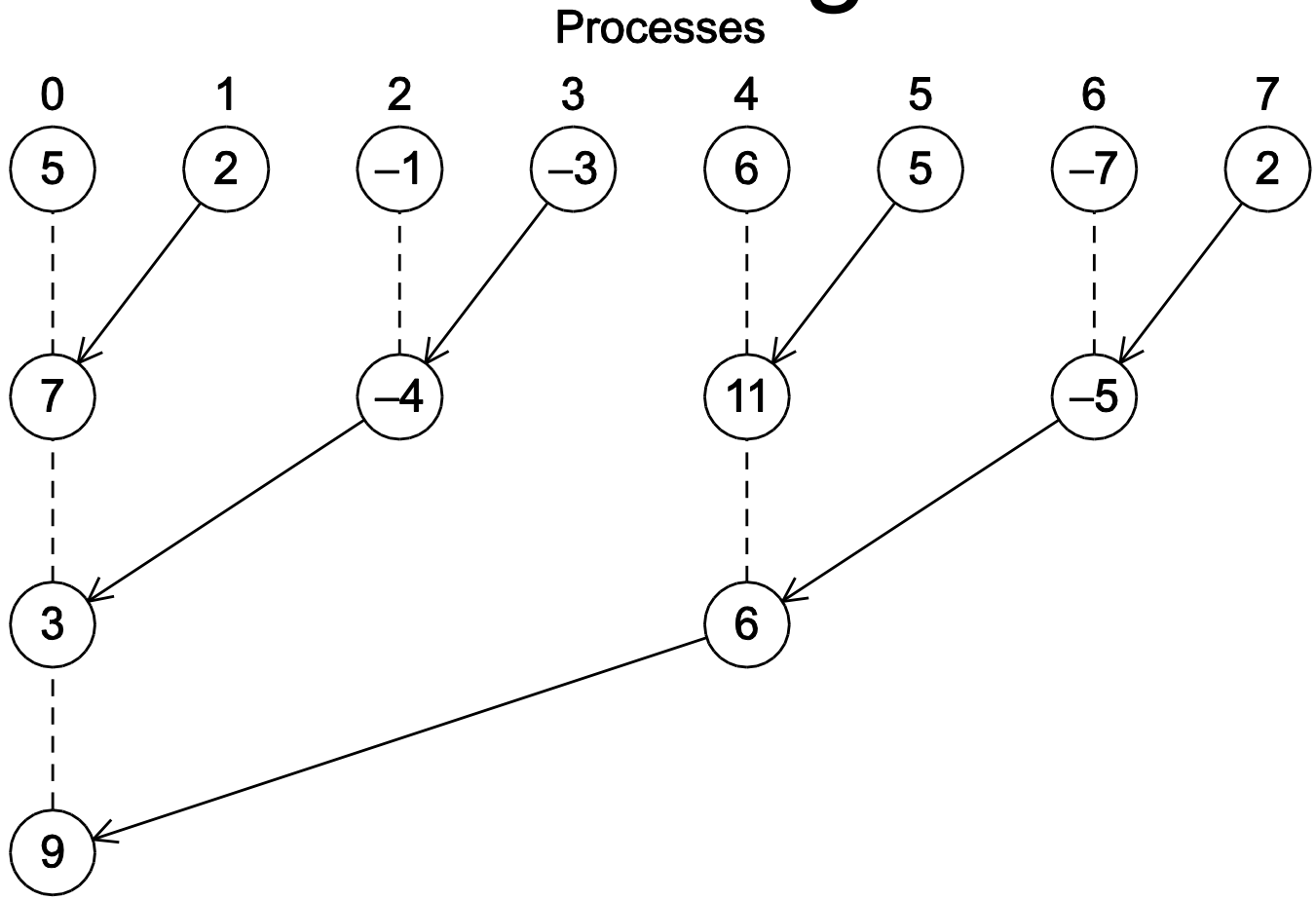
A version of Get_input that uses MPI_Bcast

```
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 */
```

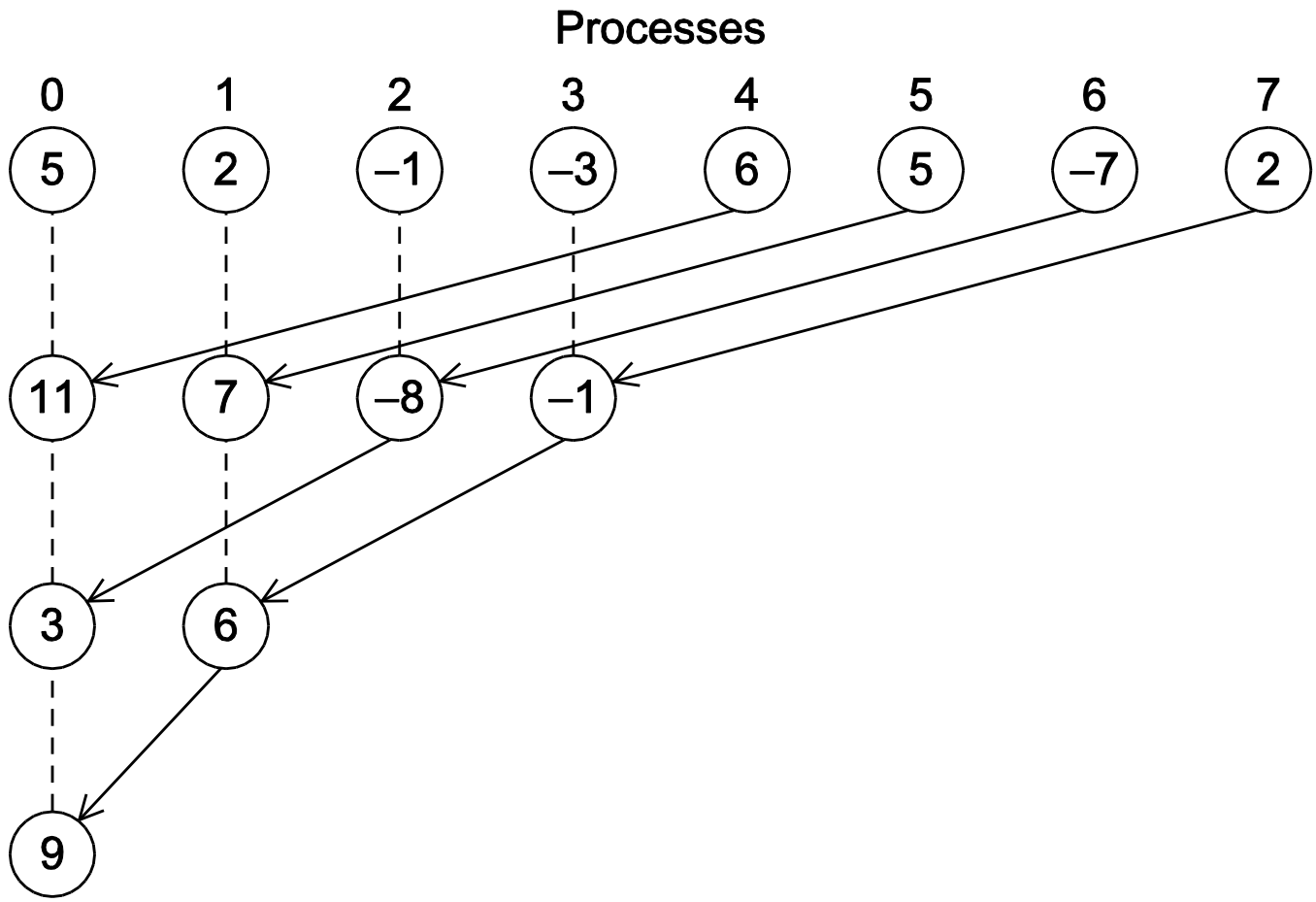
MPI_Bcast for Input

```
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 */
```

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

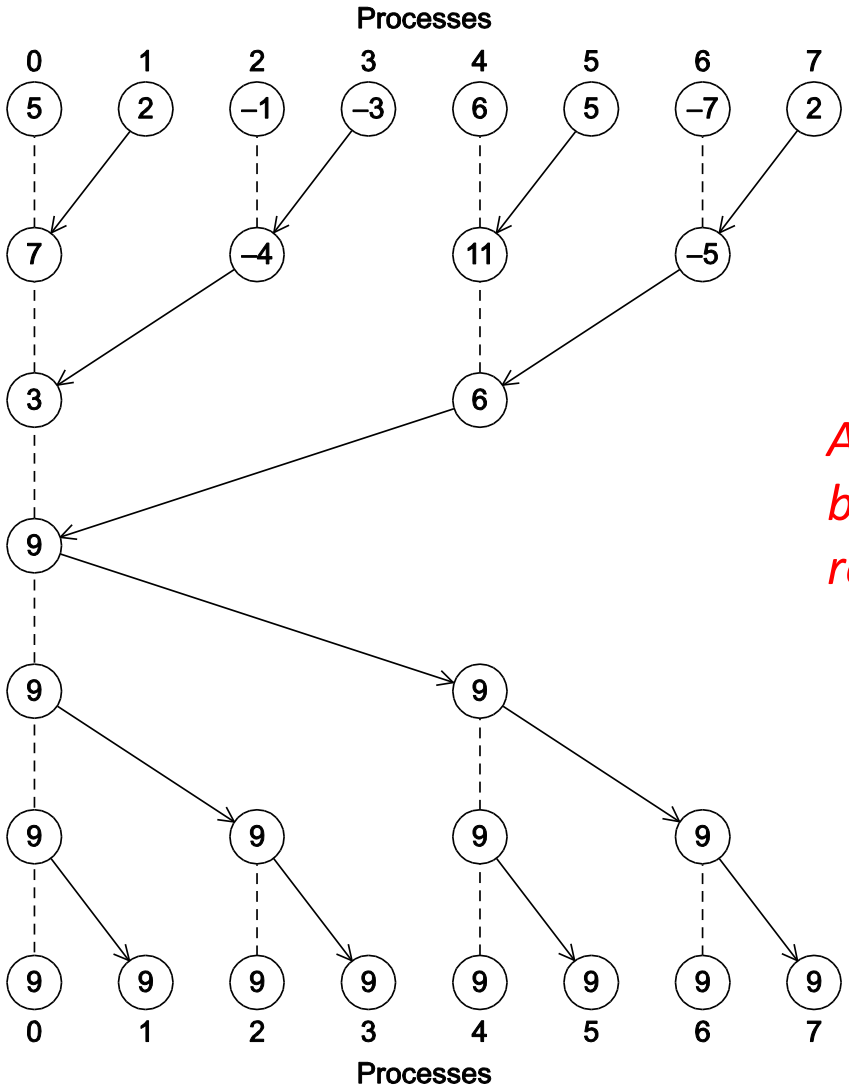
Example

- 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.

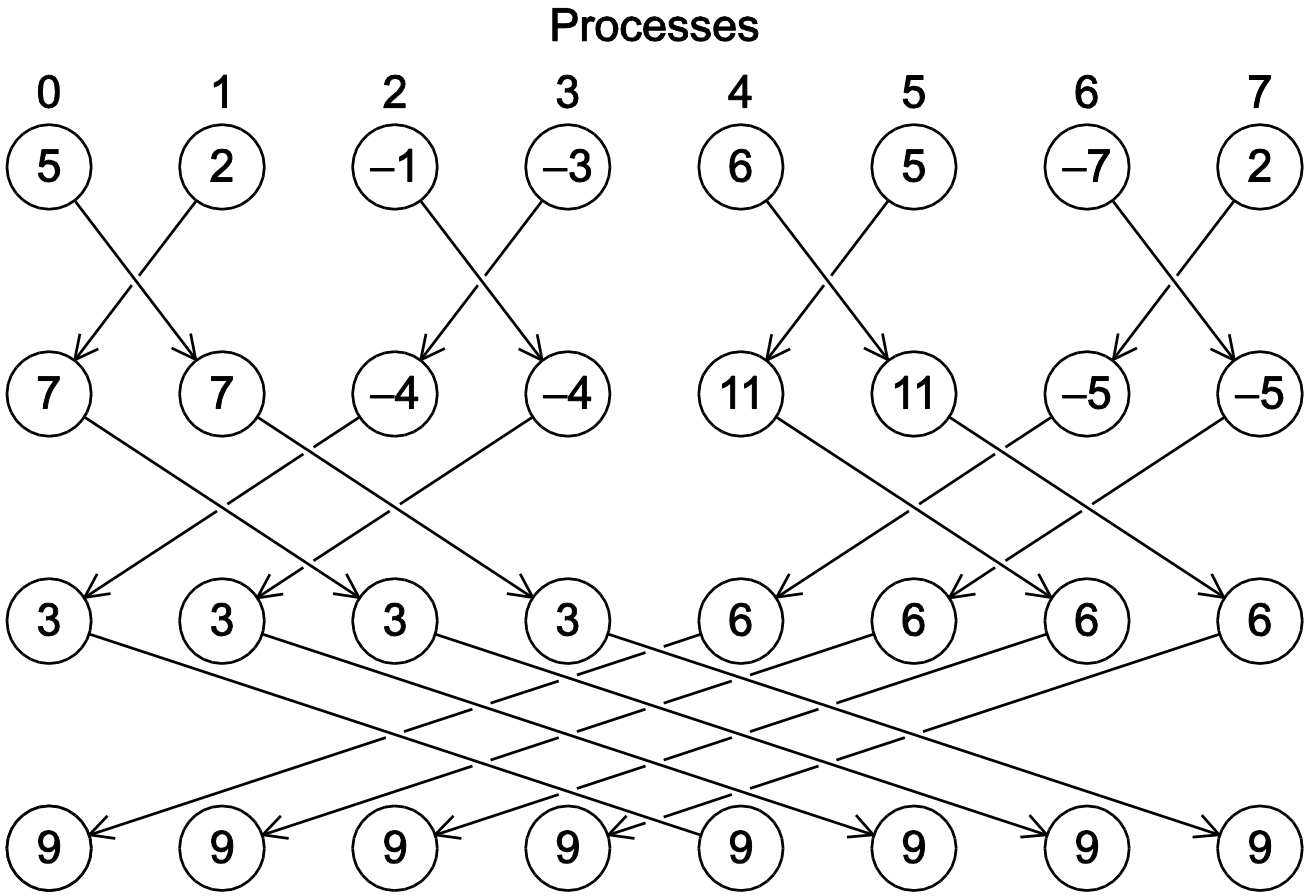
MPI_Allreduce

- 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.

More MPI

Bryan Mills, PhD

Spring 2017

MPI So Far

- Communicators
- Blocking Point-to-Point
  - MPI_Send
  - MPI_Recv
- Collective Communications
  - MPI_Bcast
  - MPI_Barrier
  - MPI_Reduce
  - MPI_Allreduce

Non-blocking Send

```
int MPI_Isend(
     void*          msg_buf  /* input */,
     int            msg_size /* input */,
     MPI_Datatype msg_type /* input */,
     int            dest     /* input */,
     int            tag      /* input */,
     MPI_Comm       comm     /* input */,
     MPI_Request   &request /* output */)
```

Identical to MPI_Send but added argument for getting handle to MPI_Request struct.

Non-blocking Receive

```
int MPI_Irecv(
    void*          msg_buf  /* output */,
    int            msg_size /* input */,
    MPI_Datatype   msg_type /* input */,
    int            source   /* input */,
    int            tag      /* input */,
    MPI_Comm       comm     /* input */,
    MPI_Request    &request /* output */)
```

Identical to MPI_Read but replaces MPI_Status argument with MPI_Request struct. Status is now retrieved in the MPI_Wait.

Blocking Wait

```
int MPI_Wait(
   MPI_Request  request /* input */,
   MPI_Status   &status /* output */)
```

Same status that would have been returned from MPI_Recv, not as useful in MPI_Send.

```
int MPI_Waitall(
   int              count  /* input */,
   MPI_Request**    requests /* input */,
   MPI_Status**     &statuses /* output */)
```

Non-blocking Test

```
int MPI_Test(
   MPI_Request   request  /* input */,
   int           &flag    /* output */
   MPI_Status    &status  /* output */)
```

Flag returns 1 if completed, otherwise 0.

```
MPI_Testall(incount, requests,
   flag, statuses)
MPI_Testsome(incount, requests,
   outcount, indices, statuses)
MPI_Testany(incount, requests,
   index, flag, status)
```

Serial implementation of vector addition

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

Parallel implementation of vector addition

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

Scatter

- MPI_Scatter can be used in a function that reads in an entire vector on process 0 but only sends the needed components to each of the other processes.

```
int MPI_Scatter(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int            src_proc     /* in  */,
      MPI_Comm       comm         /* in  */);
```

Reading and distributing a vector

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Read_vector */
```

Gather

- Collect all of the components of the vector onto process 0, and then process 0 can process all of the components.

```
int MPI_Gather(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
      int            dest_proc    /* in   */,
      MPI_Comm       comm         /* in   */);
```

Print a distributed vector (1)

```
void Print_vector(
      double    local_b[]  /* in */,
      int       local_n    /* in */,
      int       n          /* in */,
      char      title[]    /* in */,
      int       my_rank    /* in */,
      MPI_Comm  comm       /* in */) {

   double* b = NULL;
   int i;
```

Print a distributed vector (2)

```
   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n, MPI_DOUBLE,
            0, comm);
   }
}  /* Print_vector */
```

Allgather

- Concatenates the contents of each process' send_buf_p and stores this in each process' recv_buf_p.
- As usual, recv_count is the amount of data being received from each process.

```
int MPI_Allgather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

Matrix-vector multiplication

$A = (a_{ij})$ is an $m \times n$ matrix

$\mathbf{x}$ is a vector with n components

$\mathbf{y} = A\mathbf{x}$ is a vector with m components

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}$$

i-th component of y

Dot product of the ith row of A with x.

Matrix-vector multiplication

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
\quad = \quad
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

Multiply a matrix by a vector

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

Serial pseudo-code

C style arrays

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

stored as

0 1 2 3 4 5 6 7 8 9 10 11

Serial matrix-vector multiplication

```
void Mat_vect_mult(
        double  A[]  /* in  */,
        double  x[]  /* in  */,
        double  y[]  /* out */,
        int     m    /* in  */,
        int     n    /* in  */) {
    int i, j;

    for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i*n+j]*x[j];
    }
}  /* Mat_vect_mult */
```

An MPI matrix-vector multiplication function (1)

```
void Mat_vect_mult(
        double    local_A[]  /* in  */,
        double    local_x[]  /* in  */,
        double    local_y[]  /* out */,
        int       local_m    /* in  */,
        int       n          /* in  */,
        int       local_n    /* in  */,
        MPI_Comm  comm       /* in  */) {
    double* x;
    int local_i, j;
    int local_ok = 1;
```

An MPI matrix-vector multiplication function (2)

```
    x = malloc(n*sizeof(double));
    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
          x, local_n, MPI_DOUBLE, comm);

    for (local_i = 0; local_i < local_m; local_i++) {
       local_y[local_i] = 0.0;
       for (j = 0; j < n; j++)
          local_y[local_i] += local_A[local_i*n+j]*x[j];
    }
    free(x);
}  /* Mat_vect_mult */
```

PERFORMANCE EVALUATION

Elapsed parallel time

- Returns the number of seconds that have elapsed since some time in the past.

```
double MPI_Wtime(void);
```

```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
printf("Proc %d > Elapsed time = %e seconds\n"
      my_rank, finish-start);
```

Run-times of serial and parallel matrix-vector multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

(Seconds)

Speedups of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

Efficiencies of Parallel Matrix-Vector Multiplication

comm_sz	Order of Matrix				
	1024	2048	4096	8192	16,384
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

Scalability

- A program is scalable if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be strongly scalable.

- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be weakly scalable.