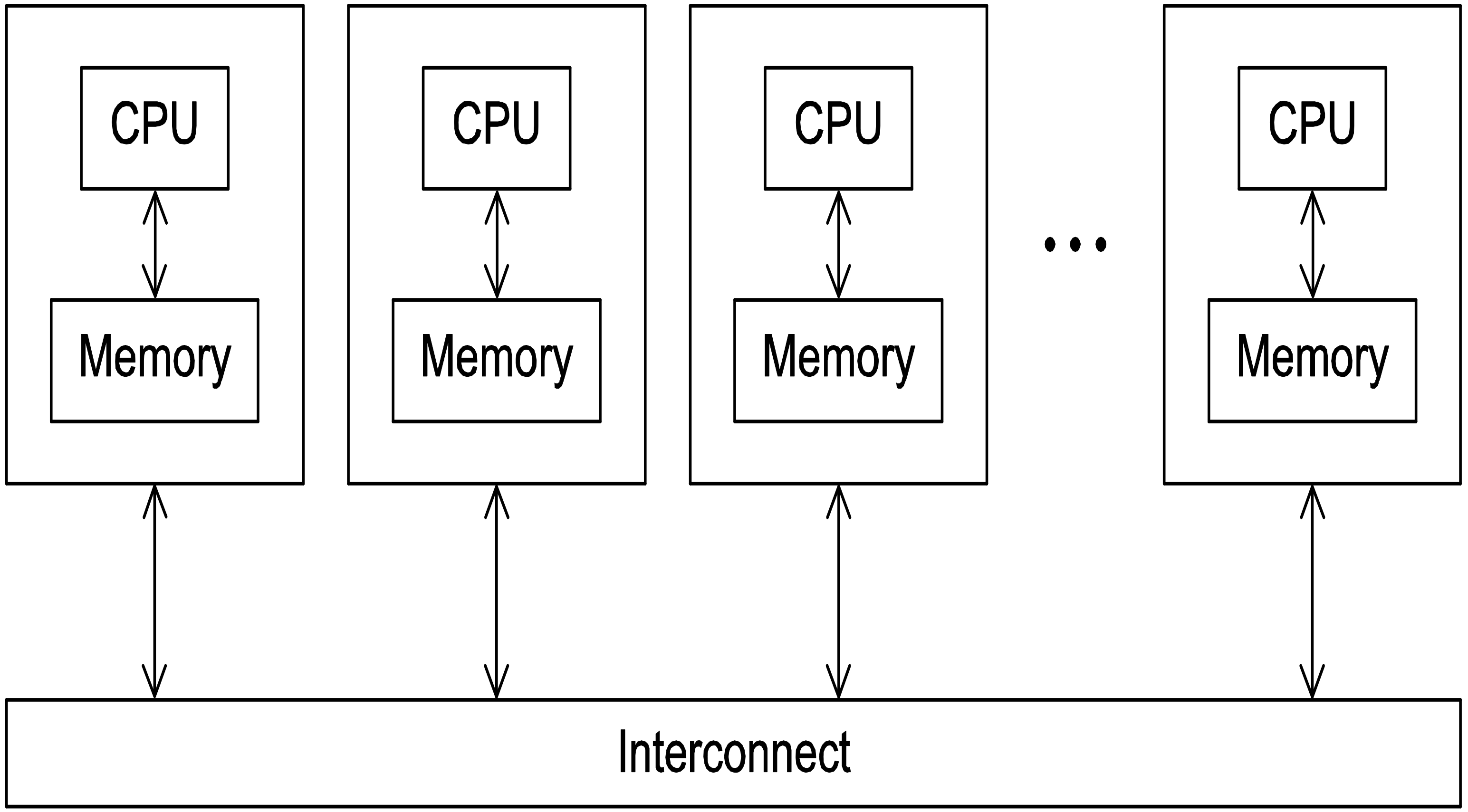
Unit-IV

D.Venkata Vara Prasad

A distributed memory system



A distributed memory system

- MIMD, computers are divided into **distributed-memory** and **shared-memory** systems.
- A distributed-memory system consists of a collection of core-memory pairs.
- These are connected by a network, and the memory associated with a core is directly accessible only to that core.

A distributed memory system

- In message-passing programs, 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
  - and the other calls a **receive** function.

A distributed memory system

- The implementation of message-passing can be done by **MPI**, which is an abbreviation of **Message-Passing Interface.**
- MPI is not a new programming language.
- It defines a library of functions that can be called from C, C++, and Fortran programs

MPI programs :Hello World!

```
#include <stdio.h>

int main(void) {
   printf("hello, world\n");

   return 0;
}
```

MPI programs

- Instead of having each process simply print a message, we’ll designate one process to do the output, and the other processes will send it messages, which it will print.

- Common practice to identify processes by nonnegative integer ranks.

- *p* processes are numbered *0, 1, 2, .. p-1*

MPI programs

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

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

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      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);
   } else {
      printf("Greetings from process %d of %d!\n", my_rank, comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

MPI programs :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

MPI programs :Compilation

- **mpicc** is a script that's a wrapper for the **C compiler.**
- A wrapper script is a script whose main purpose is to run some program.
- In this case, the program is the C compiler.
- The wrapper simplifies the running of the compiler by
- telling it where to find the necessary header files and which libraries to link with the object file.

MPI programs :Execution

```
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 :Execution

```
mpiexec  -n  1  ./mpi_hello

      Greetings from process 0 of 1 !
```

```
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

- 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_Init and MPI_Finalize

- In Line 12 the call to MPI Init tells the MPI system to do all of the necessary setup.
- For example :
  - o it might allocate storage for message buffers,
  - o it might decide which process gets which rank.

MPI_Init and MPI_Finalize

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

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

- The arguments, argc p and argv p, are pointers to the arguments to main, argc, and argv.
- when our program doesn’t use these arguments, we can pass NULL for both

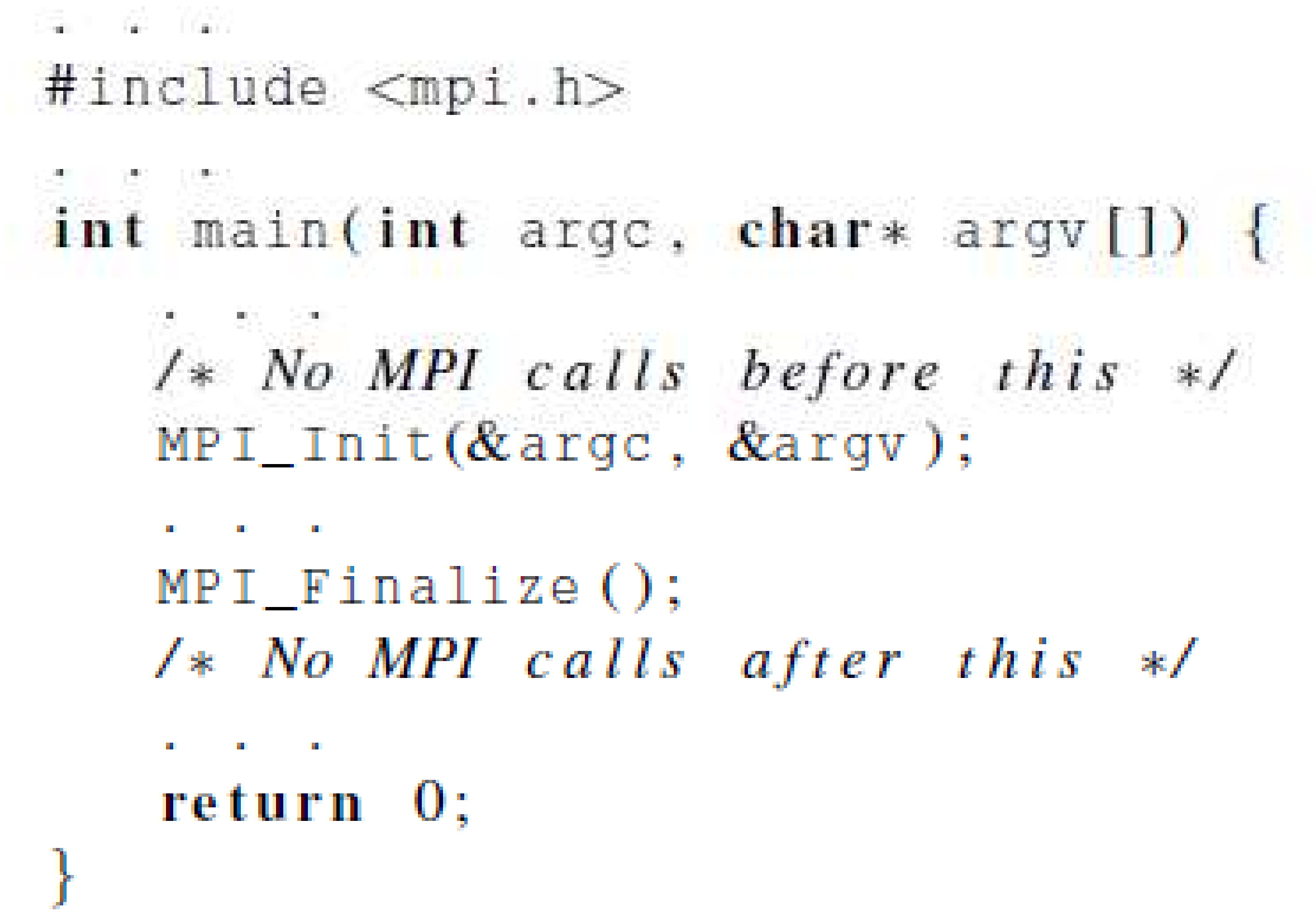
MPI_Init and MPI_Finalize

- **MPI_Finalize**
- In Line 30 the call to MPI_Finalize tells the MPI system that :
  - we're done using MPI
  - so clean up anything allocated for this program
  - any resources allocated for MPI can be freed.
- The syntax is :

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

- In general, no MPI functions should be called after the call to MPI Finalize.

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         /* in  */,
      int*       comm_sz_p    /* out */);
```

number of processes in the communicator

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

my rank
(the process making this call)

Communicators

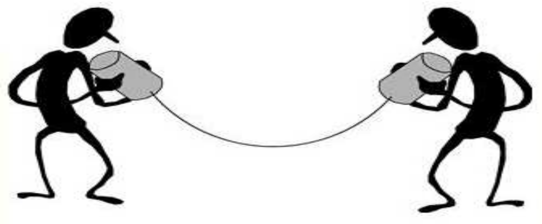
- For both functions, the first argument is a special type defined by MPI for communicators, MPI_Comm.
- MPI_Comm _size returns in its second argument the number of processes in the communicator
- MPI_Comm_ rank returns in its second argument the calling process' rank in the communicator

SPMD

- Single-Program Multiple-Data
- We compile <u>one</u> program.
- we didn’t compile a different program for each process
- Process 0 does something different.
  - Receives messages and prints them
  - while the other processes is creating and sending a message.

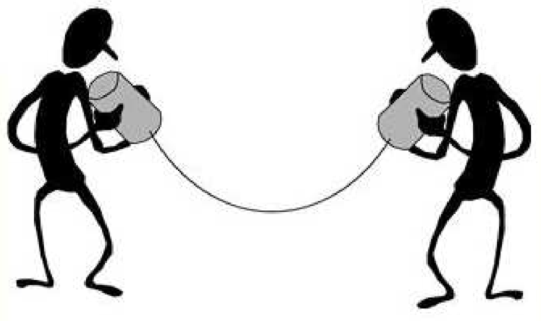
Communication

- Lines 17 & 18, each process, other than process 0, creates a message & will send to process 0.
- The function sprintf is very similar to printf, except that instead of writing to stdout, it writes to a string.
- Lines 19–20 actually send the message to process 0.
- Process 0, on the other hand, simply prints its message using printf, and then uses a for loop to receive and print the messages sent by processes 1, 2,… ,comm_sz-1.
- Lines 24–25 receive the message sent by process q, for q = 1, 2, : : : ,comm_sz -1



Communication

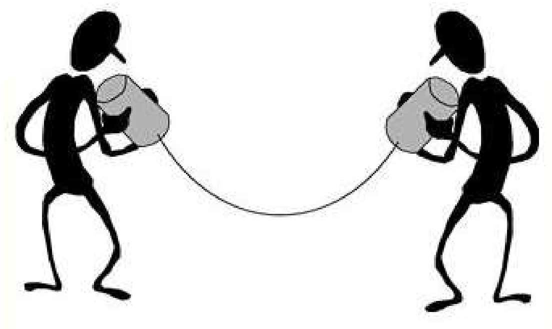
```
int MPI_Send(
    void*          msg_buf_p     /* in */,
    int            msg_size      /* in */,
    MPI_Datatype   msg_type      /* in */,
    int            dest          /* in */,
    int            tag           /* in */,
    MPI_Comm       communicator  /* in */);
```



Communication

- The first three arguments, **msg_buf_p**, **msg_size**, and **msg_type**, determine the contents of the message.

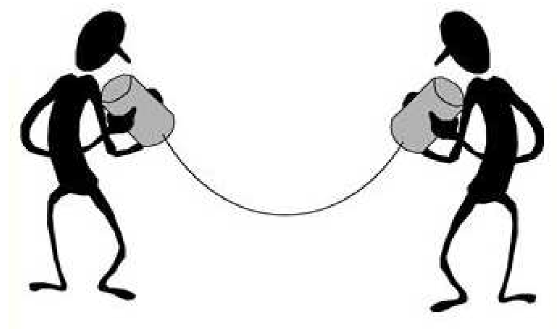
- The remaining arguments, dest, tag, and communicator, determine the destination of the message.



Communication

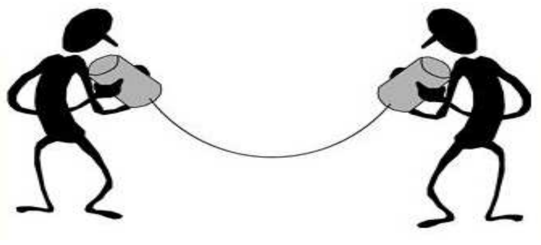
- msg_buf_p, is a pointer to the block of memory containing the contents of the message.

- In our program, this is just the string containing the message greeting.



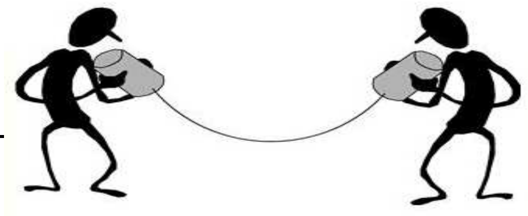
Communication

- msg_size determine the amount of data to be sent.
- In our program, the msg_size argument is the number of characters in the message plus one character for the ‘\0’ character that terminates C strings.
- msg type argument is MPI CHAR.
- These two arguments together tell the system that the message contains strlen(greeting)+1 chars.



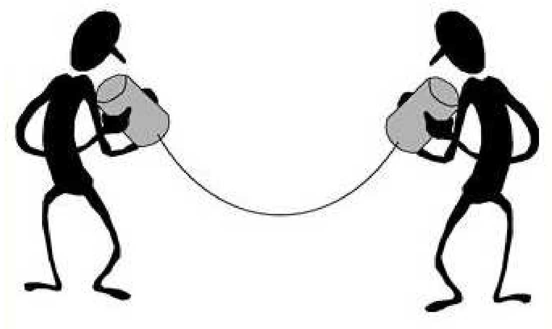
Data types

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	



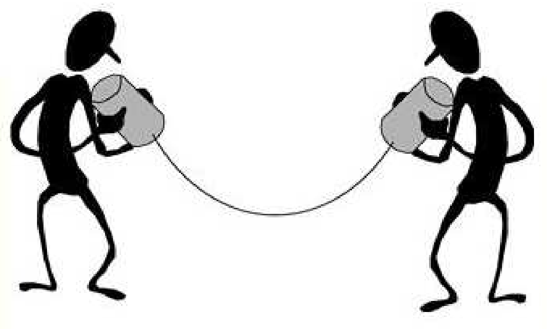
Communication

- Dest argument specifies the rank of the process that should receive the message
- Tag argument, is a nonnegative int. It can be used to distinguish messages that are otherwise identical.
- Ex: suppose process 1 is sending floats to process 0 Some of the floats should be printed, while others should be used in a computation.



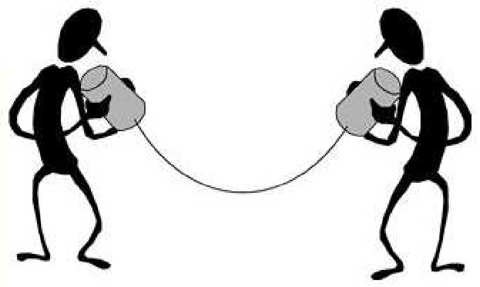
Communication

- communicator. All MPI functions that involve communication have a communicator argument



Communication

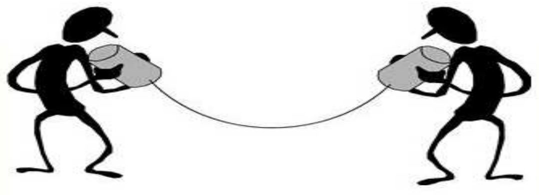
```
int MPI_Recv(
        void*          msg_buf_p      /* out */,
        int            buf_size       /* in  */,
        MPI_Datatype   buf_type       /* in  */,
        int            source         /* in  */,
        int            tag            /* in  */,
        MPI_Comm       communicator   /* in  */,
        MPI_Status*    status_p       /* out */);
```



Communication

- The first six arguments to MPI Recv correspond to the first six arguments of MPI Send.
- The first three arguments specify the memory available for receiving the message:

- **msg_buf_p** points to the block of memory
- **buf_size** determines the number of objects that can be stored in the block
- **buf type** indicates the type of the objects.



Communication

- The next three arguments identify the message.
- **source** argument specifies the process from which the message should be received.
- **tag** argument should match the tag argument of the message being sent.
- **communicator** argument must match the communicator used by the sending process.

Message matching

- Suppose process q calls MPI_Send with

  MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send _tag, send_comm);

- Also suppose that process r calls MPI_Recv with
- **MPI_Recv (recv_buf_p, recv_buf_sz, recv_type, src, recv_ tag,recv_ comm, &status);**
-

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

Message matching

- Then the message sent by q with the above call to MPI_Send can be received by r with the call to MPI_Recv if

  - o recv_comm = send_comm
  - o recv_tag = send_tag
  - o dest = r
  - o src = q.

- If recv_type = send_type & recv_buf_sz>= send_ buf_ sz
- then the message sent by q can be successfully received by

Status_p argument

- MPI_Status is a struct with at least the three members:
  - MPI_SOURCE,
  - MPI_TAG,
  - MPI_ERROR.

- Suppose our program contains the definition

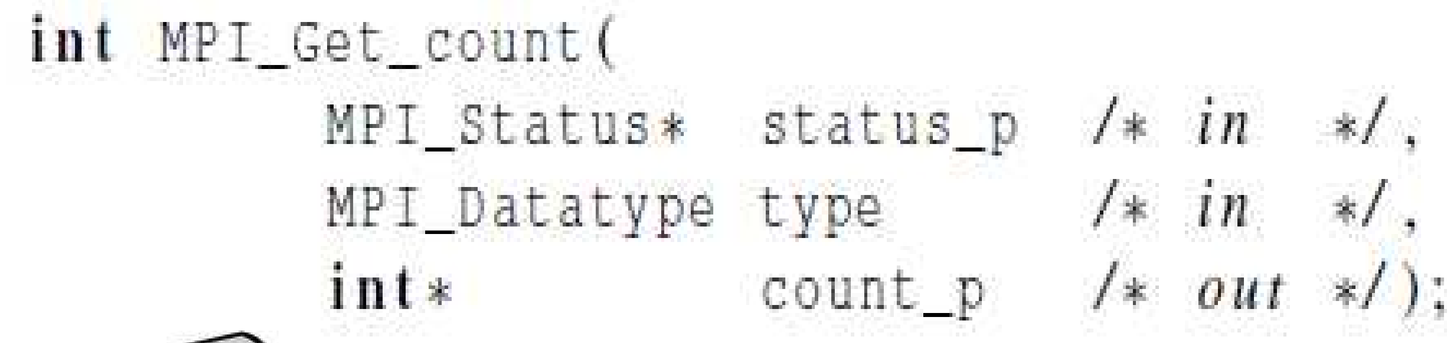
```
MPI_Status status;
```

Status_p argument

- A call to MPI_Recv in which & status is passed as the last argument.
- we can determine the sender and tag by examining the two members:
  - status.MPI_SOURCE
  - status.MPI_TAG

How much data am I receiving?

- The amount of received isn’t stored in a field that’s directly accessible to the application program.
- It can be retrieved with a call to MPI_Get_ count.



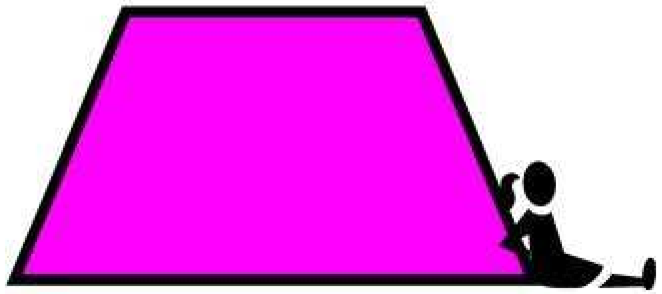
```
int MPI_Get_count(
        MPI_Status*   status_p  /* in   */,
        MPI_Datatype  type      /* in   */,
        int*          count_p   /* out  */);
```

How much data am I receiving?

- The count isn’t directly accessible as a member of the MPI_ Status variable.
- Because it depends on the type of the received data, consequently, determining it would probably require a calculation

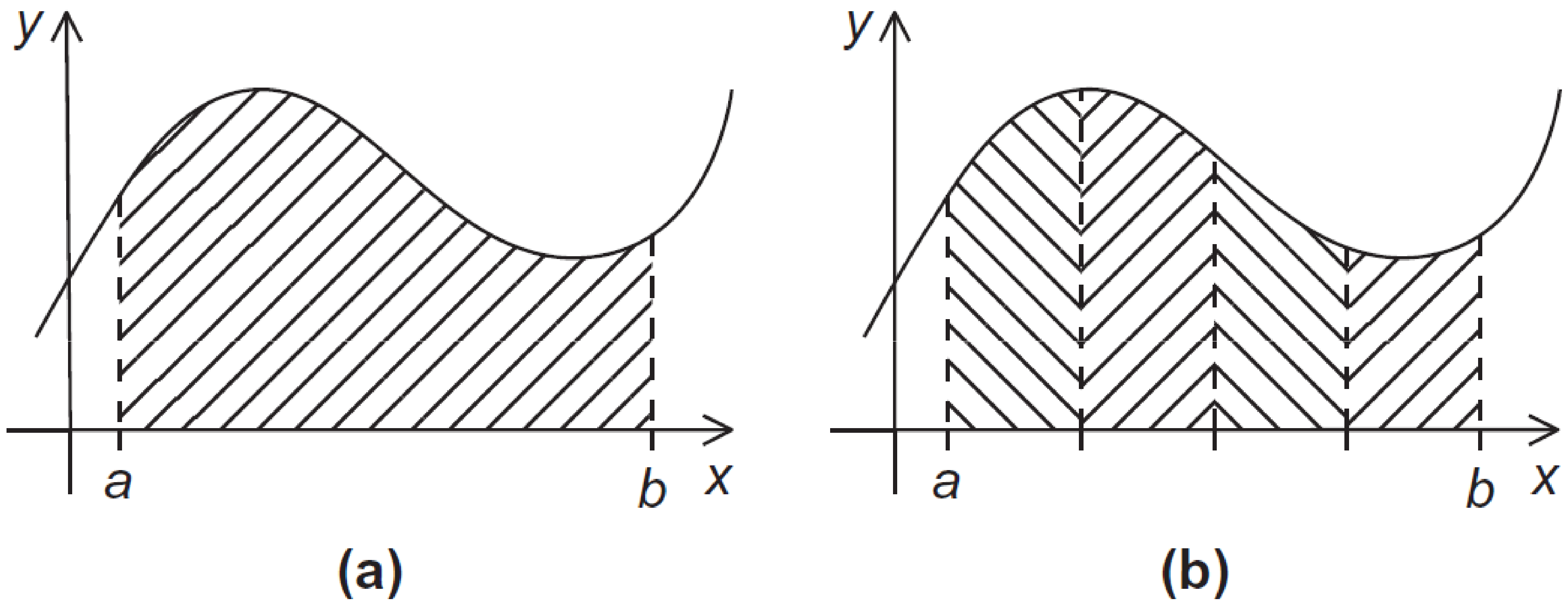
Issues with send and receive

- 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!



TRAPEZOIDAL RULE IN MPI

The Trapezoidal Rule



The Trapezoidal Rule

- Use the trapezoidal rule to approximate the area between the graph of a function, y =f (x), two vertical lines, and the x-axis.

- The basic idea is to divide the interval on the x-axis into **n** equal subintervals.

The Trapezoidal Rule

- Then we approximate the area lying between the graph and each subinterval by a trapezoid whose
  - base is the subinterval,
  - vertical sides are the vertical lines through the endpoints of the subinterval,
  - fourth side is the secant line joining the points where the vertical lines cross the graph.

The Trapezoidal Rule

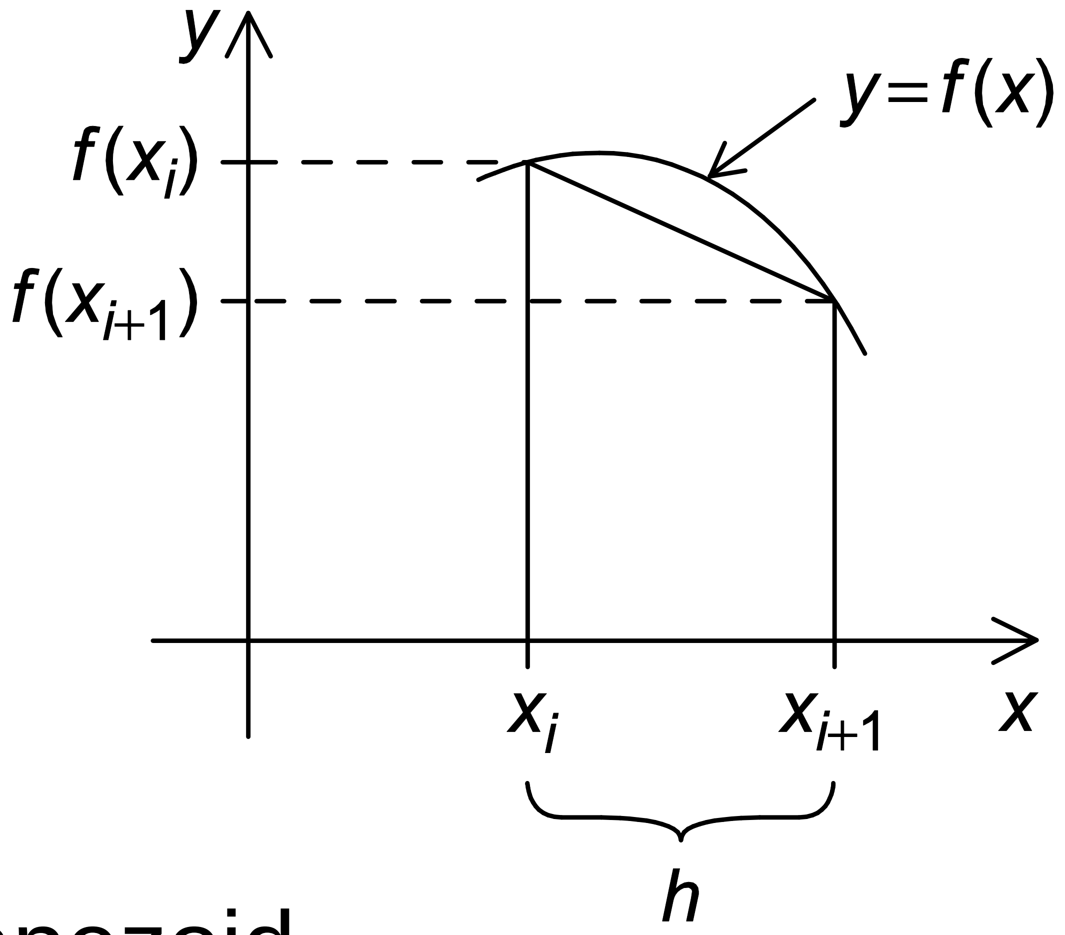
$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})]$$

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

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

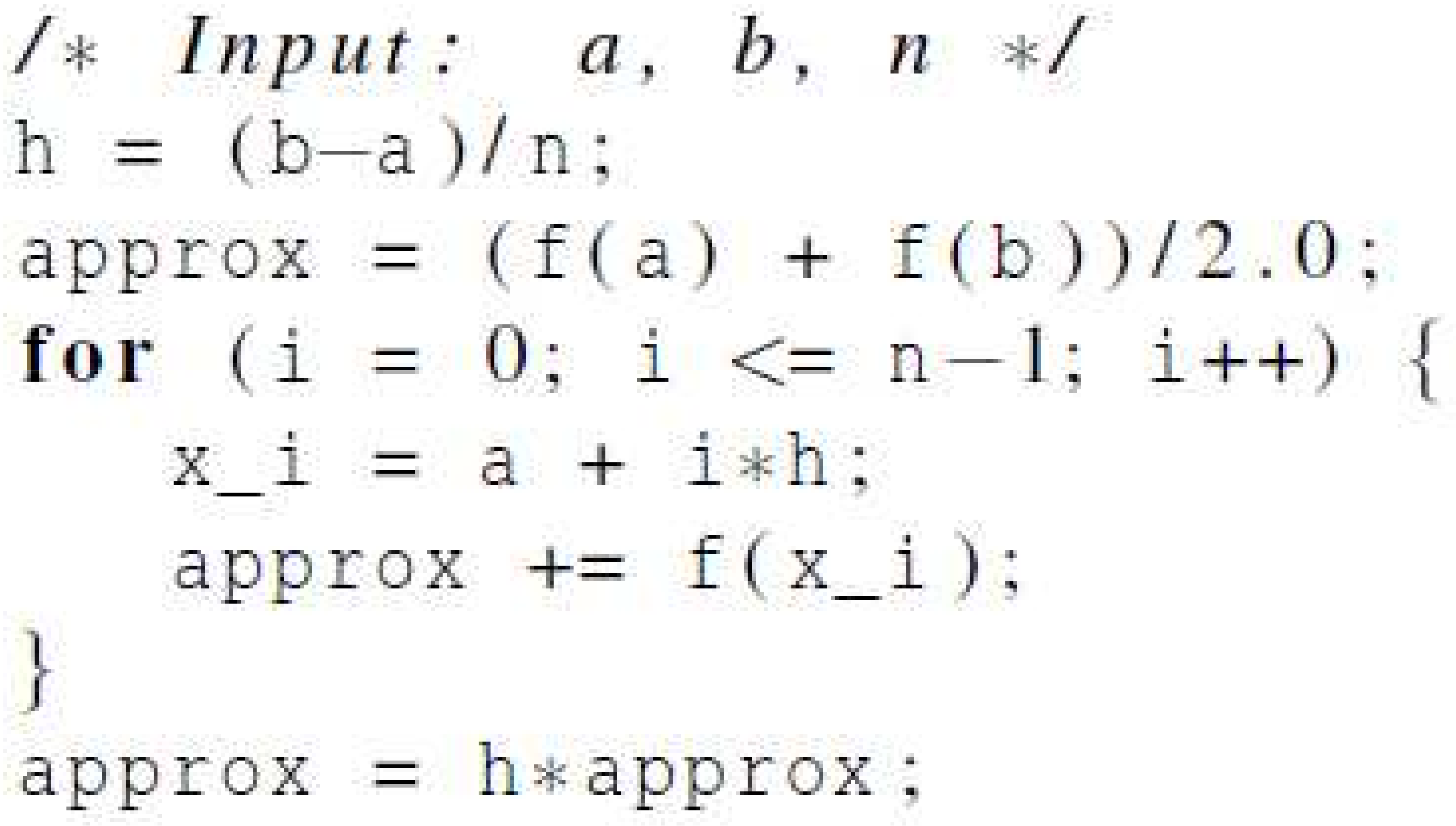
$$\text{Sum of trapezoid areas } = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$$

The Trapezoidal Rule



One trapezoid

Pseudo-code for a serial program



```
/* Input:  a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 0; i <= n-1; i++) {
    x_i = a + i*h;
    approx += f(x_i);
}
approx = h*approx;
```

Parallelizing the Trapezoidal Rule

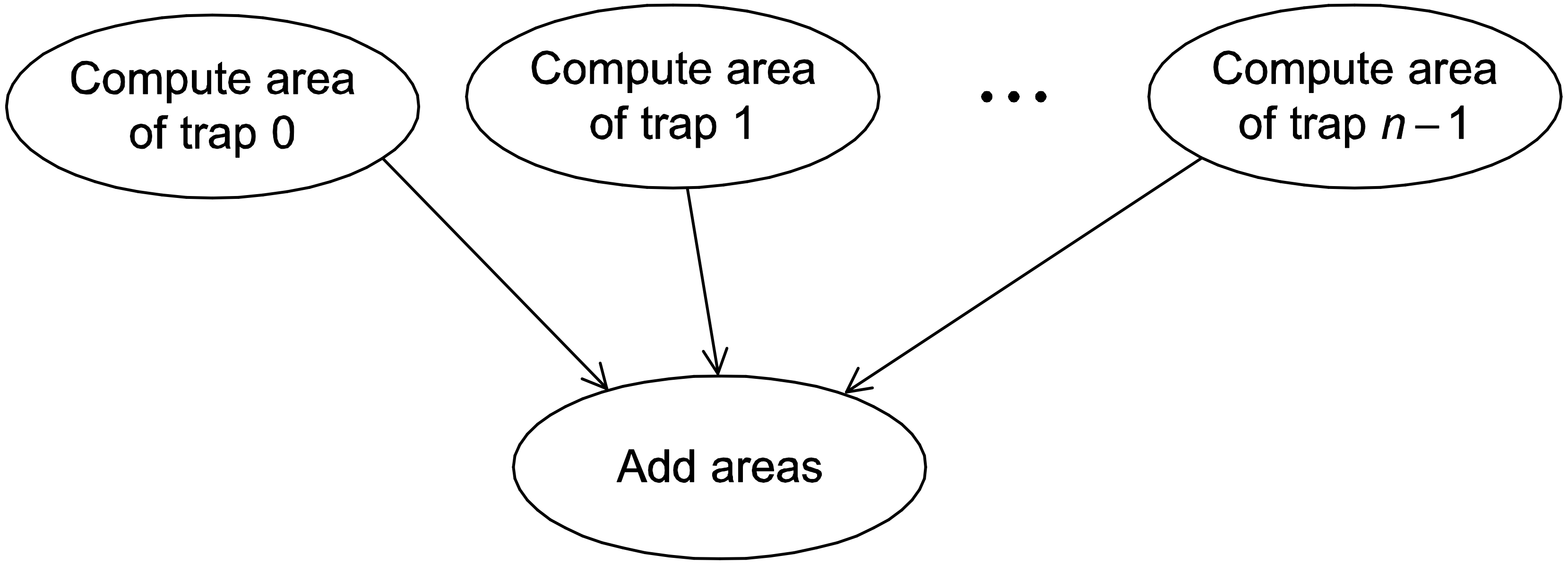
1. Partition problem solution into tasks.
2. Identify communication channels between tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.

Parallelizing the Trapezoidal Rule

- partitioning phase :
- try to identify as many tasks as possible.
- For the trapezoidal rule, we might identify two types of tasks:
  - one type is finding the area of a single trapezoid,
  - the other is computing the sum of these areas.

Parallelizing the Trapezoidal Rule

- communication channels will join each of the tasks of the first type to the single task of the second type



Parallelizing the Trapezoidal Rule

- aggregate the tasks and map them to the cores.
- we should use many trapezoids, and map them to cores.
- Aggregate the computation of the areas of the trapezoids into groups.

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;
```

First version (1)

```
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);
```

First version (2)

```c
   } 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  */
```

First version (3)

```c
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  */
```

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 */
```

Each process just prints a message.

Dealing with I/O

- MPI implementations allow all the processes in MPI_COMM_WORLD full access to stdout and stderr.
- Most MPI implementations allow all processes to execute printf and fprintf(stderr, ...).
- 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.

Dealing with I/O

- when we run it with six processes, the order of the output lines is unpredictable:

```
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 5 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
```

or

```
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?
```

Dealing with I/O

- The reason is that the MPI processes are “competing” for access to the shared output device, stdout.
- It’s impossible to predict the order in which the processes’ output will be queued up.
- Such a competition results in nondeterminism.

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;
. . .
```

Input

- Unlike output, most MPI implementations only allow process 0 in **MPI_COMM _WORLD** access to stdin.
- If multiple processes have access to stdin, which process should get which parts of the input data.
  - Should process 0 get the first line?
  - Process 1 the second? Or
  - should process 0 get the first character?

Input

- In order to write MPI programs that can use scanf, we need to branch on process rank, with process 0 reading in the data and then sending it to the other processes.

Function for reading user input

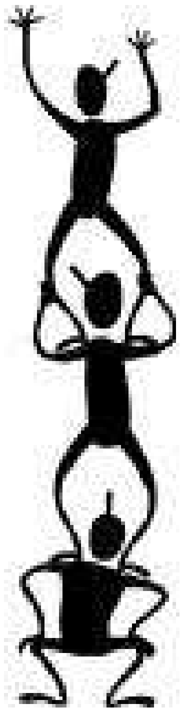
```
void Get_input(
      int      my_rank   /* in  */,
      int      comm_sz   /* in  */,
      double*  a_p       /* out */,
      double*  b_p       /* out */,
      int*     n_p       /* out */) {
   int dest;

   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 */
```

Function for reading user input

- In this function process 0 is sending to each process, while the other processes are receiving.
- To use this function, we can simply insert a call to it inside our main function to put it after initializing my_rank and comm_sz:

COLLECTIVE COMMUNICATION

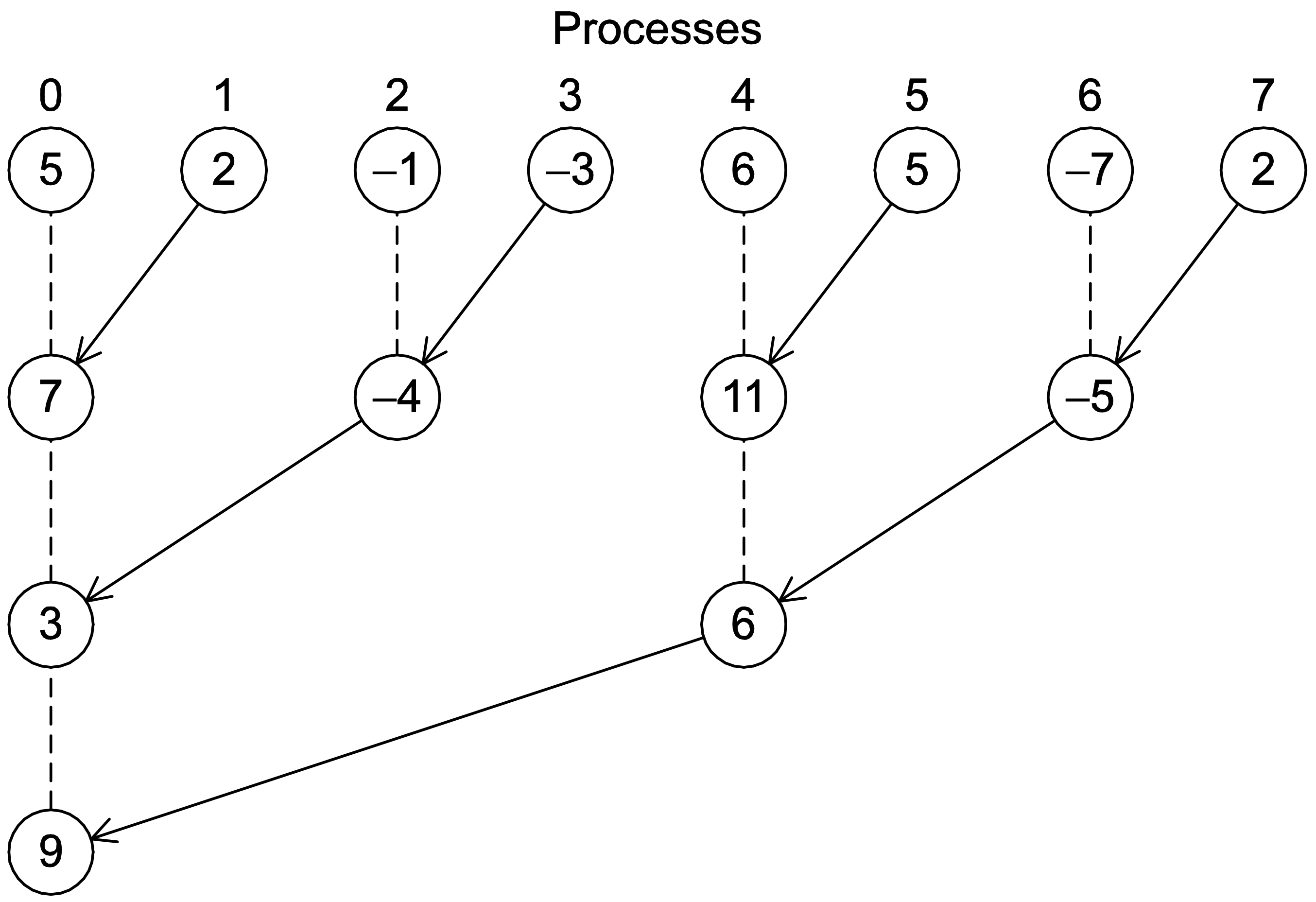


TREE-STRUCTURED COMMUNICATION

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



A TREE-STRUCTURED GLOBAL SUM

- The original scheme required ***comm_sz-1*** = seven receives and seven adds by process 0,
- The new scheme only requires three, and all the other processes do no more than two receives and adds.

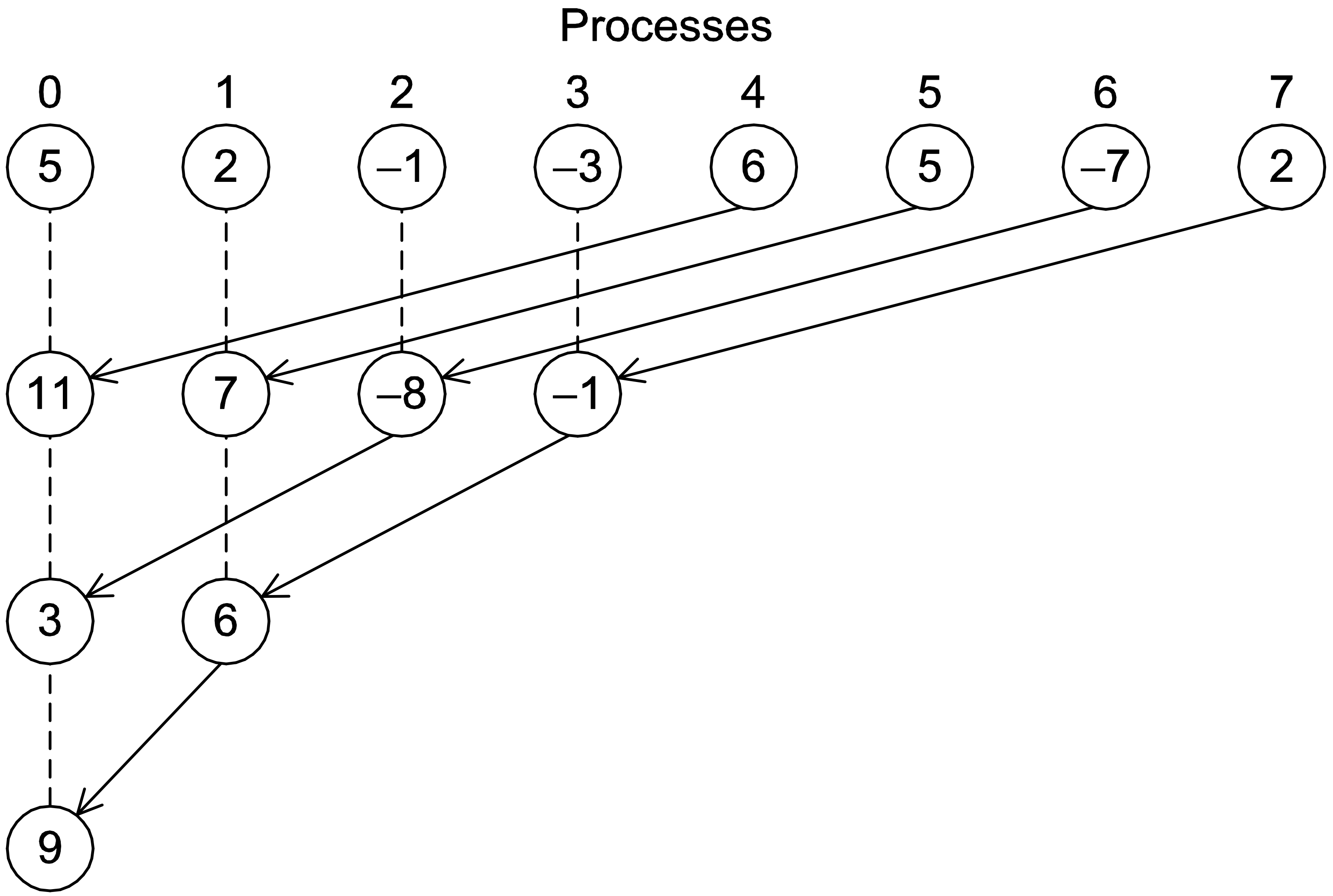
A TREE-STRUCTURED GLOBAL SUM

- In the first phase, the receives and adds by processes 0, 2, 4, & 6 can all take place simultaneously.

- If the processes start at the same time, the total time required to compute the global sum will be the time required by process 0, that is, three receives and three additions.

- We’ve reduced the overall time by more than 50%.

AN ALTERNATIVE TREE-STRUCTURED GLOBAL SUM



AN ALTERNATIVE TREE-STRUCTURED GLOBAL SUM

- we might pair 0 and 4, 1 and 5, 2 and 6, and 3 and 7 in the first phase.
- Then we could pair 0 and 2, and 1 and 3 in the second phase, and 0 and 1 in the final.

MPI_REDUCE

- A “global-sum function” will require communication.
- Unlike the MPI_Send – MPI_Recv pair, the global-sum function may involve more than two processes.
- MPI_Send and MPI_Recv are often called point-to-point communications
- Global sum is a special case of class of collective communications

MPI_REDUCE

- MPI generalized the global-sum function so that any one of these possibilities can be implemented with a single function:

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);
```

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.

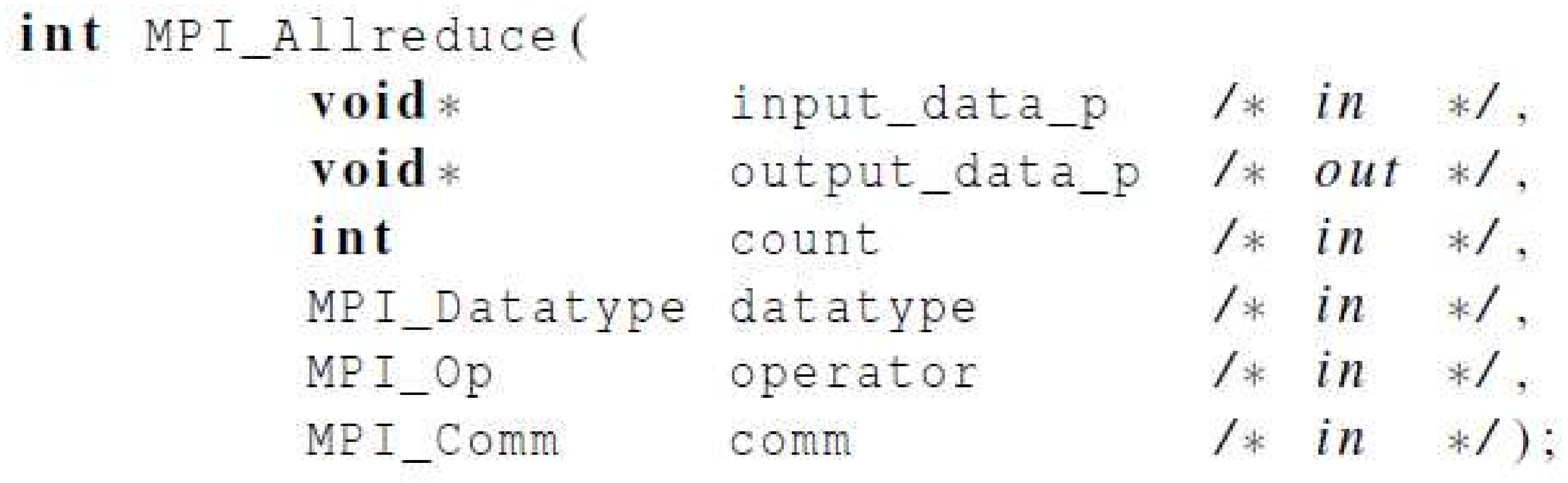
COLLECTIVE VS. POINT-TO-POINT COMMUNICATIONS

- 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.

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.

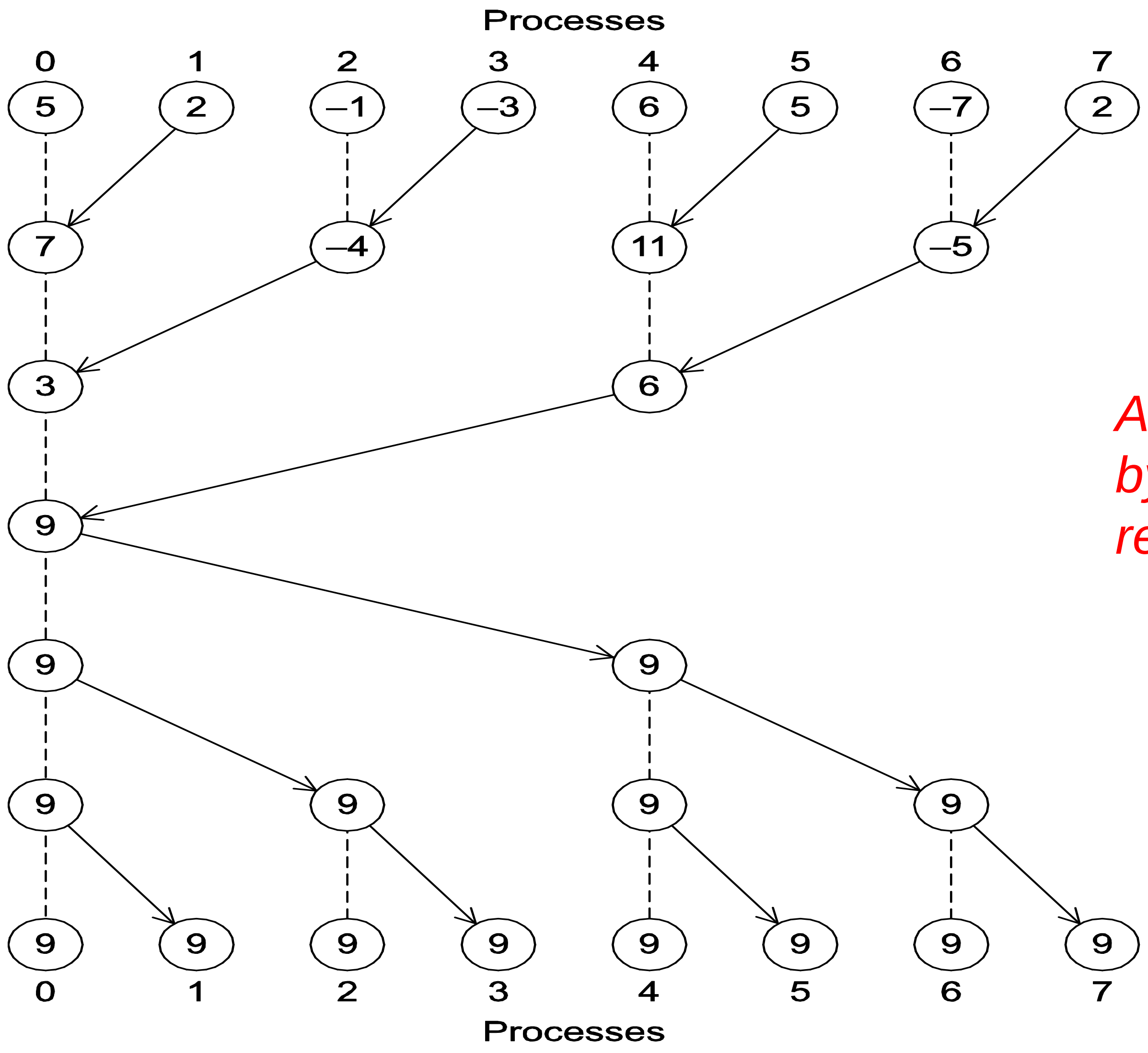


```
int MPI_Allreduce(
        void*         input_data_p   /* in  */,
        void*         output_data_p  /* out */,
        int           count          /* in  */,
        MPI_Datatype  datatype       /* in  */,
        MPI_Op        operator       /* in  */,
        MPI_Comm      comm           /*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

MPI_Allreduce

- If we use a tree to compute a global sum, we might “reverse” the branches to distribute the global sum

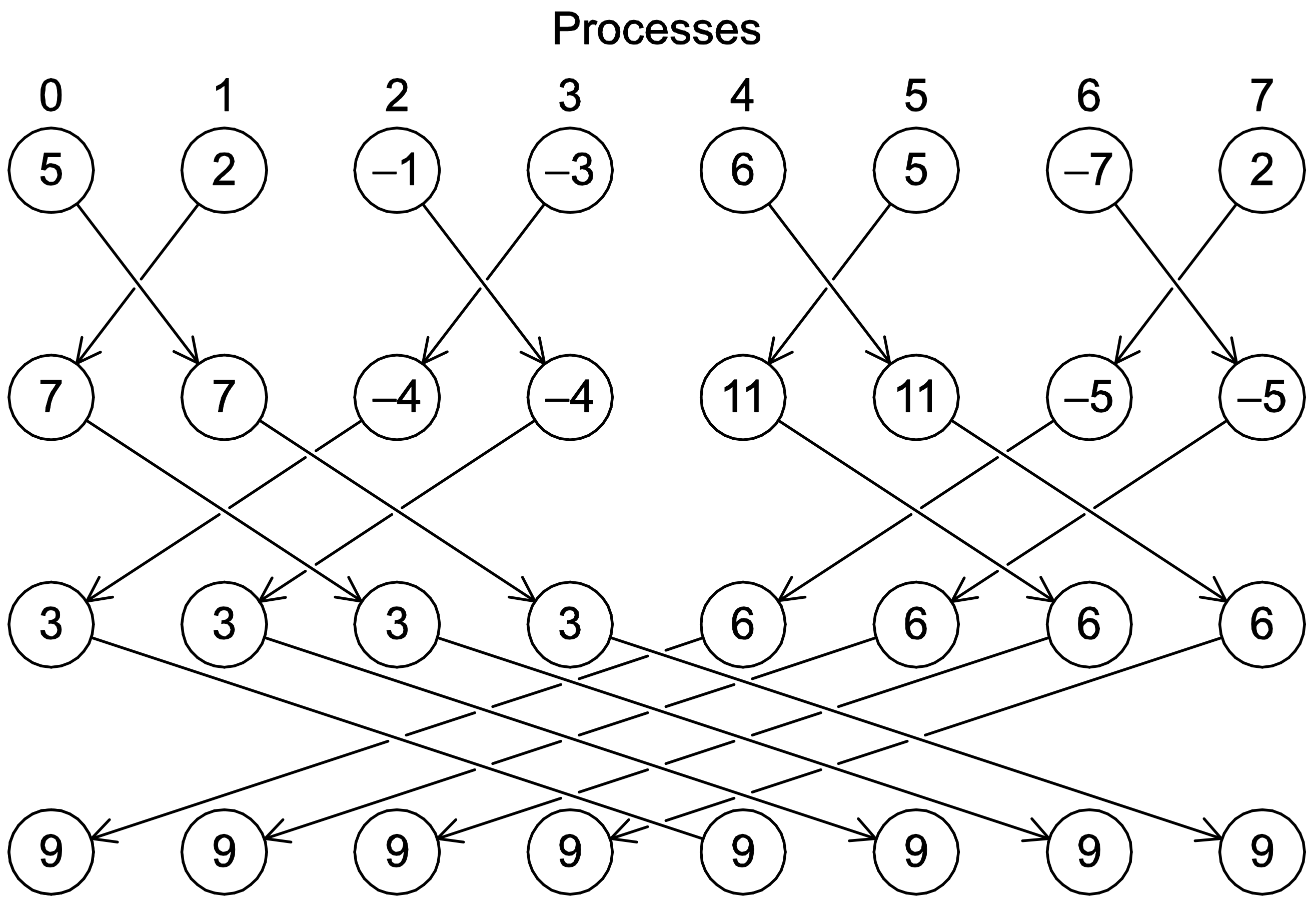
MPI_Allreduce



A global sum followed by distribution of the result.

MPI_Allreduce

- Alternatively, we might have the processes exchange partial results instead of using one-way communications.
- Such a communication pattern is called a butterfly .
- we don’t want to have to decide on which structure to use, or how to code it for optimal performance.



A butterfly-structured global sum.

Broadcast

- A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is ca.lled a broadcast.

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

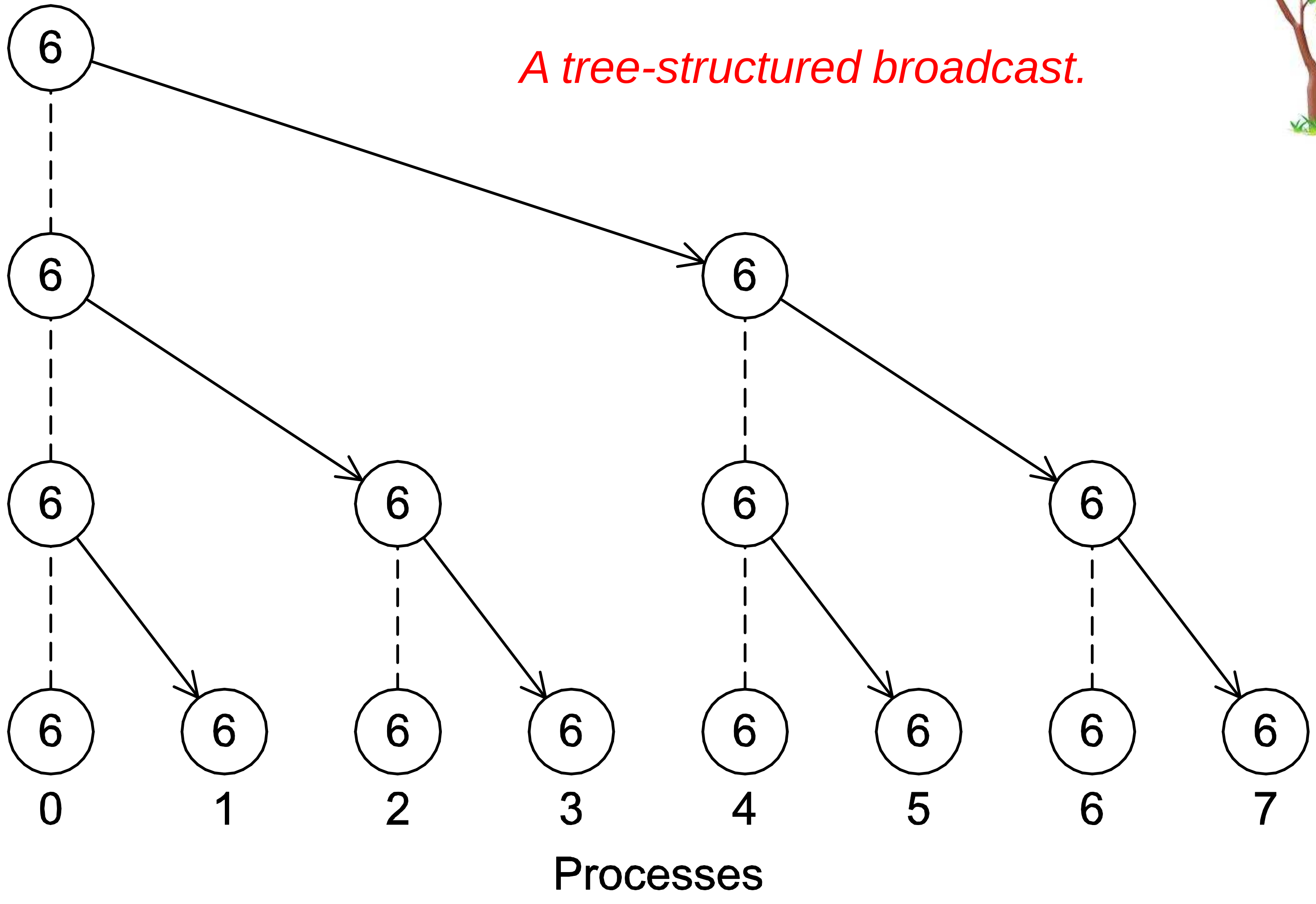
Broadcast

- The process with rank **source_proc** sends the contents of the memory referenced by **data_p** to all the processes in the communicator **comm.**

- For **MPI_Bcast**, however, the **data_p** argument is an input argument on the process with rank **source_proc** and an output argument on the other processes.

Broadcast

A tree-structured broadcast.



Data distributions

$$\begin{aligned}
\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
&= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
&= (z_0, z_1, \ldots, z_{n-1}) \\
&= \mathbf{z}
\end{aligned}$$

Compute a vector sum.

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 */
```

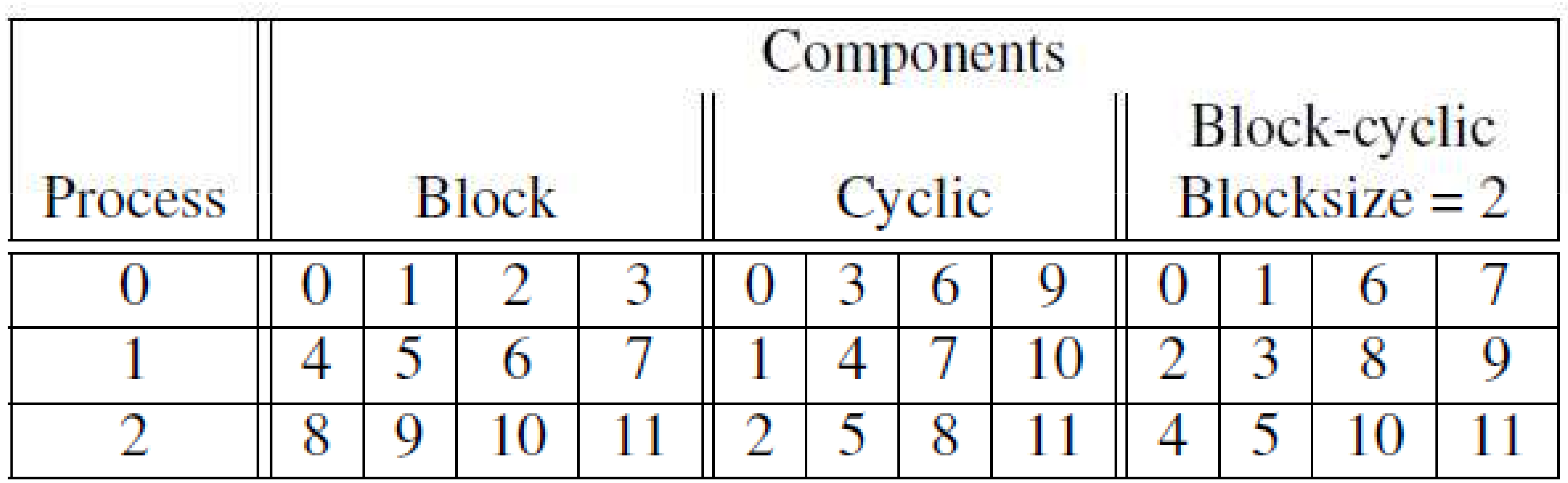
Data distributions

- It involves adding the individual components of the vectors.
- The tasks are just the additions of corresponding components.
- There is no communication between the tasks .
- parallelizing vector addition boils down to aggregating the tasks and assigning them to the cores.

Data distributions

- If the number of components is ***n.***
- we have comm_sz cores or processes
- assume that n evenly divides comm_sz
- we define local_ n= n/ comm_sz.
- Then we can simply assign blocks of local_n consecutive components to each process.

Different partitions of a 12-component vector among 3 processes



Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Partitioning options

- Block partitioning
  - Assign blocks of consecutive components to each process.
- Cyclic partitioning
  - Assign components in a round robin fashion.
- Block-cyclic partitioning
  - Use a cyclic distribution of blocks of components.

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 */
```

Parallel implementation of vector addition

- Each process simply adds its assigned components
- Each process will have ***local_n*** components of the vector
- We can store these on each process as an array of ***local_n*** elements

Scatter

- To read in the dimension of the vectors:
  - process 0 can prompt the user,
  - read in the value,
  - broadcast the value to the other processes.
  - Process 0 could read them in and broadcast them to the other processes

Scatter

- If there are 10 processes
- The vectors have 10,000 components
- Each process will need to allocate storage for vectors with 10,000 components
- It is operating on subvectors with 1000 components.

Scatter

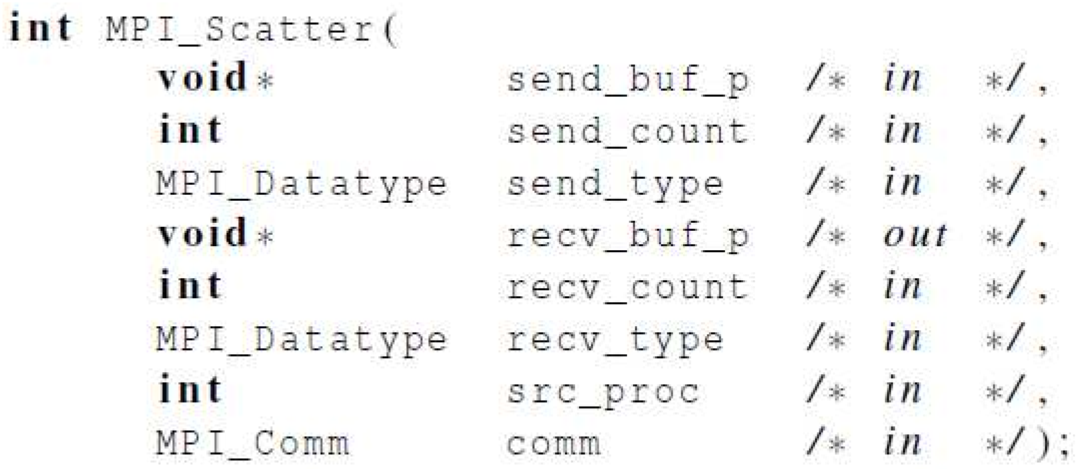
- If process 0 sent
  - components1000 to 1999 to process 1,
  - components 2000 to 2999 to process 2, & so on.
  - Using this approach, processes 1 to 9 would only need to allocate storage for the components they’re actually using.

Scatter

- If the communicator ***comm*** contains ***comm_sz*** processes,
- MPI Scatter divides the data referenced by send_ buf_ p into ***comm_sz*** pieces
- The first piece goes to process 0, the second to process 1, the third to process 2, and so on.

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.



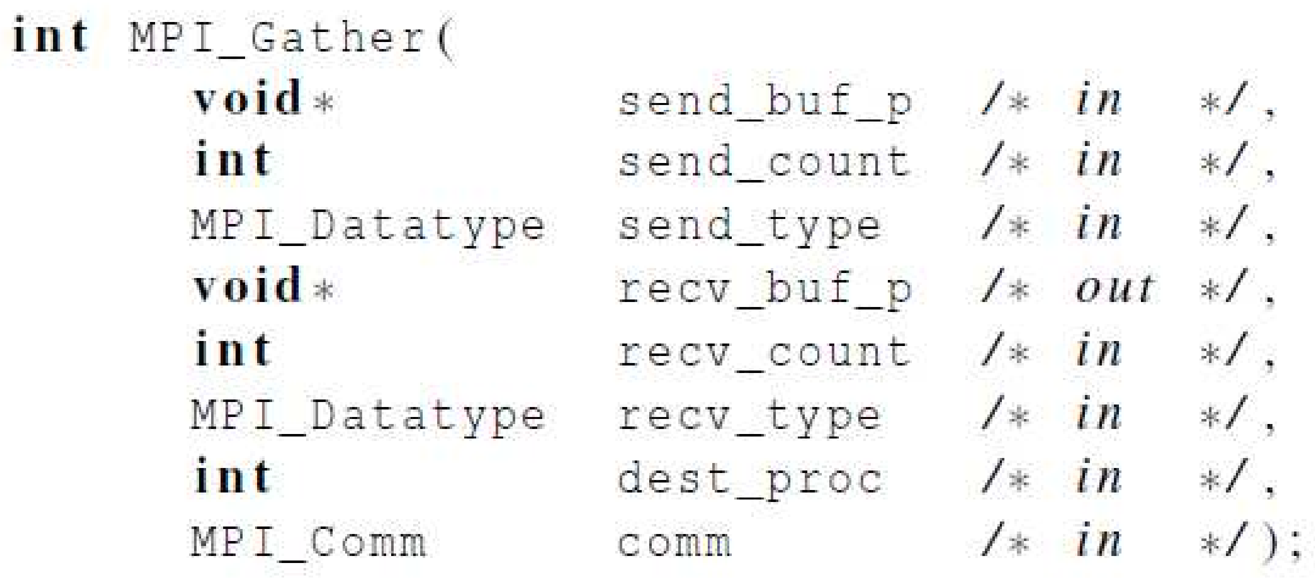
```
int MPI_Scatter(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      int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

Scatter

- suppose we're using a block distribution and process 0 has read in all of an n-component vector into ***send_buf_p***.
- process 0 will get the first ***local_n*** = n/ ***comm_sz*** components,
- process 1 will get the next ***local_ n*** components, and so on.
- Each process should pass its local vector as the ***recv_buf_p*** argument & the ***recv_count*** argument should be ***local_n***.
- Both send type and ***recv_type*** should be ***MPI_DOUBLE*** and ***src_ proc*** should be 0.

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   */);
```

Gather

- The data stored in the memory referred to by ***send_buf_p*** on process 0 is stored in the first block in recv_buf_p.
- the data stored in the memory referred to by ***send_buf_p*** on process 1 is stored in the second block referred to by ***recv_buf_ p***, and so on.
- ***recv_count*** is the number of data items received from each process, not the total number of data items received.
- The restrictions on the use of MPI_Gather are similar to those on the use of MPI_Scatter

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

x is a vector with n components

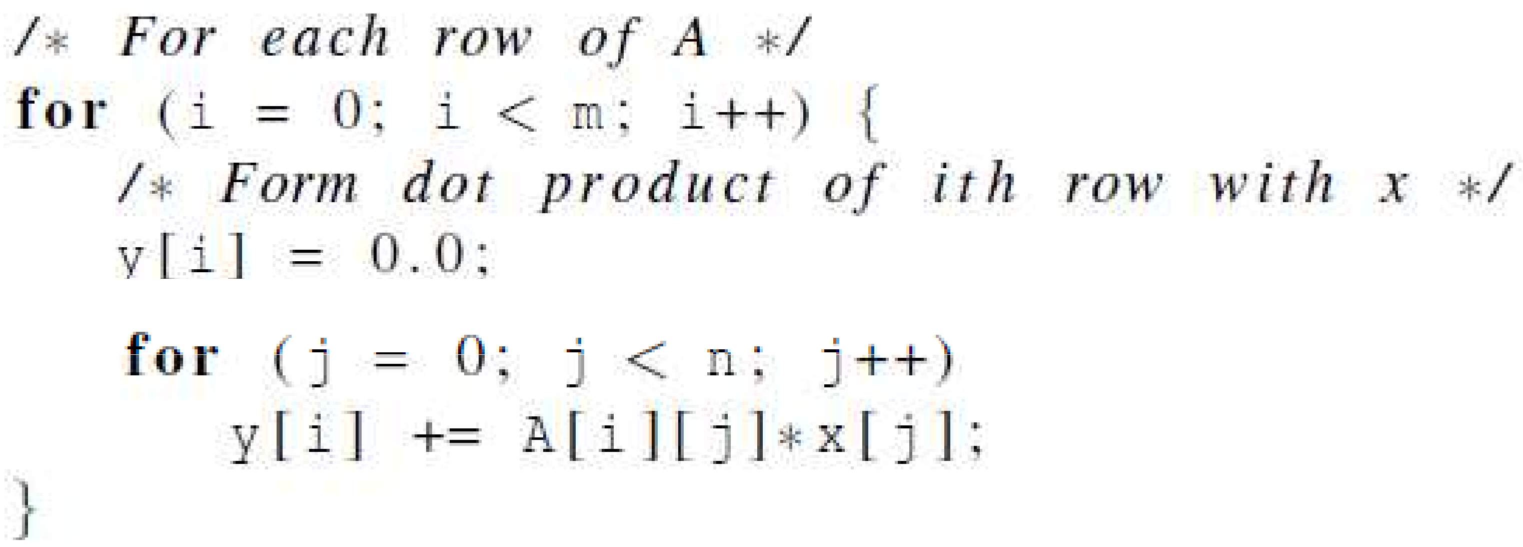
$\mathbf{y} = A\mathbf{x}$ is a vector with m components

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

i-th component of y

Dot product of the ith row of A with x.

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

Matrix-vector multiplication

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

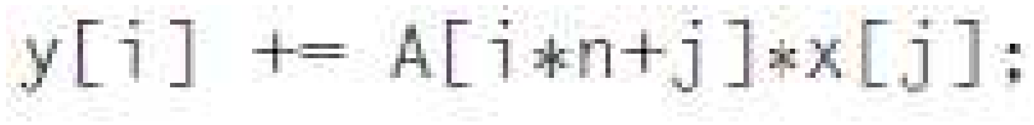
C style arrays

the two-dimensional array

$$\begin{tikzcd}
\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix} \arrow[d, bend left, "\textit{stored as}"] \\
0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11
\end{tikzcd}$$

Serial matrix-vector multiplication

- If an array has ***n*** columns, the element stored in row ***i*** and column ***j*** is located in position ***ixn+j*** in the one-dimensional array.
- An individual task can be the multiplication of an element of A by a component of x and the addition of this product into a component of y



```
y[i] += A[i*n+j]*x[j];
```

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

- The computation of ***y[i]*** involves
  - all the elements in the ***ith*** row of ***A*** and
  - all the components of x,
  - simply assigning all of x to each process we could minimize the amount of communication .
-

```
for (j = 0; j < n; j++)
   y[i] += A[i*n+j]*x[j];
```

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 */
```

MPI DERIVED DATATYPES

MPI DERIVED DATATYPES

- In distributed-memory systems, communication is more expensive than local computation.
- Example, sending a ***double*** from one node to another will take far longer than adding two doubles stored in the local memory of a node.
-

MPI DERIVED DATATYPES

- The cost of sending a fixed amount of data in multiple messages is usually much greater than the cost of sending a single message with the same amount of data.

MPI DERIVED DATATYPES

- Used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory.
- The idea is that if a function that sends data knows this information about a collection of data items, it can collect the items from memory before they are sent.
- Similarly, a function that receives data can distribute the items into their correct destinations in memory when they’re received.

MPI DERIVED DATATYPES

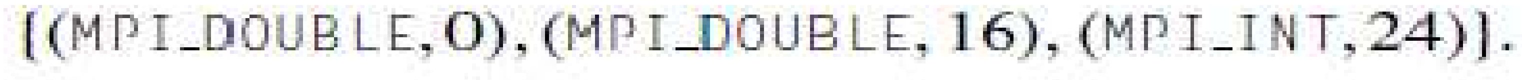
- Formally, consists of a sequence of basic MPI data types together with a displacement for each of the data types.
- Trapezoidal Rule example:

Variable	Address
a	24
b	40
n	48

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}$$

MPI DERIVED DATATYPES

- Then the following derived datatype could represent these data items:



{(MPI_DOUBLE,0),(MPI_DOUBLE,16),(MPI_INT,24)}.

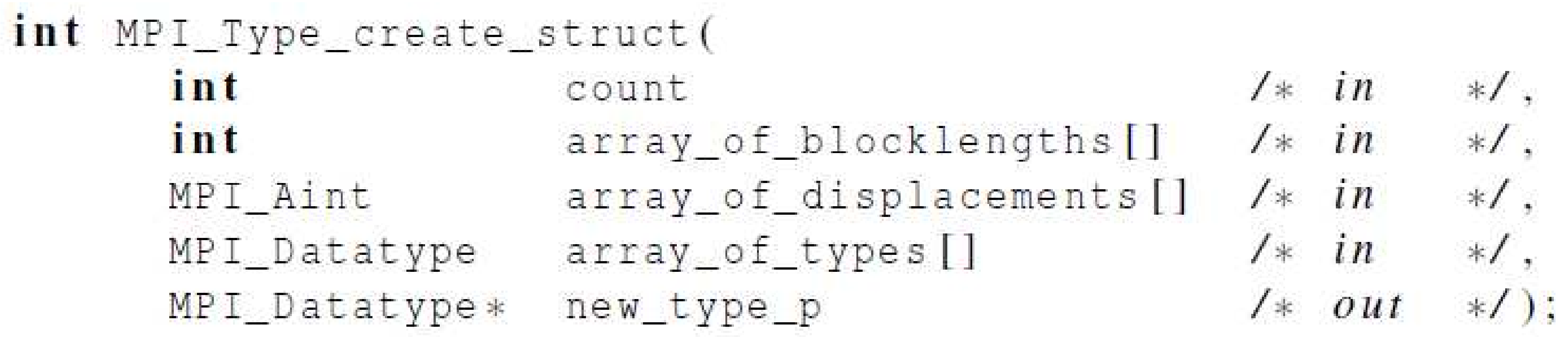
- The first element of each pair corresponds to the type of the data,
- The second element of each pair is the displacement of the data element from the beginning of the type

MPI DERIVED DATATYPES

- The type begins with ***a***, so it has displacement 0,
- The other elements have displacements measured, in bytes, from a: b is 40-24= 16 bytes beyond the start of a,
- n is 48-24= 24 bytes beyond the start of ***a***.

MPI_TYPE CREATE_STRUCT

- Builds a derived datatype that consists of individual elements that have different basic types.



```
int MPI_Type_create_struct(
      int           count                      /* in   */,
      int           array_of_blocklengths[]    /* in   */,
      MPI_Aint      array_of_displacements[]   /* in   */,
      MPI_Datatype  array_of_types[]           /* in   */,
      MPI_Datatype* new_type_p                 /* out  */);
```

MPI_TYPE CREATE_STRUCT

- The argument ***count*** is the number of elements in the datatype, so for our example, it should be three.
- Each of the array arguments should have ***count*** elements.
- The first array, ***array_ of_ block_lengths***, allows for the possibility that the individual data items might be arrays or subarrays.
- If, for example, the first element were an array containing five elements, we would have

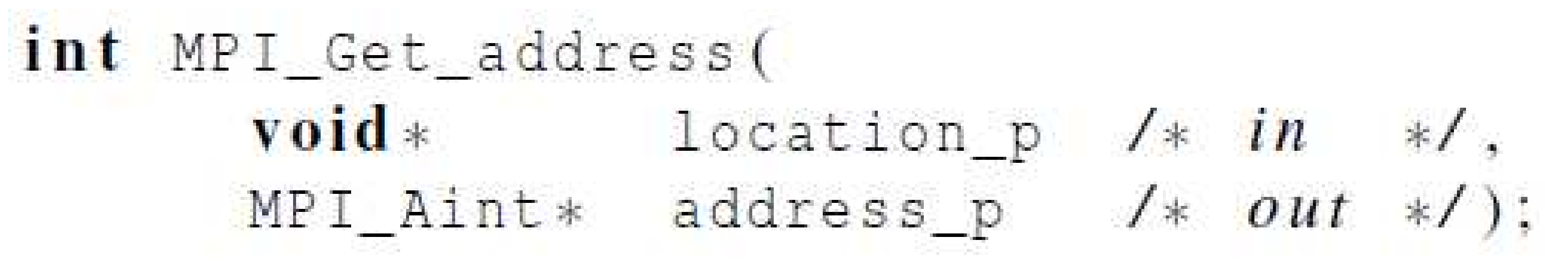
  array_ of_ block_lengths[0] =5

MPI_TYPE CREATE_STRUCT

- The third argument to ***MPI_Type_create_ struct***, ***array_of_displacements***, specifies the displacements, in bytes, from the start of the message.
-

MPI_GET_ADDRESS

- Returns the address of the memory location referenced by location_p.
- The special type MPI_Aint is an integer type that is big enough to store an address on the system.



```
int MPI_Get_address(
        void*        location_p  /* in   */,
        MPI_Aint*    address_p   /* out  */);
```

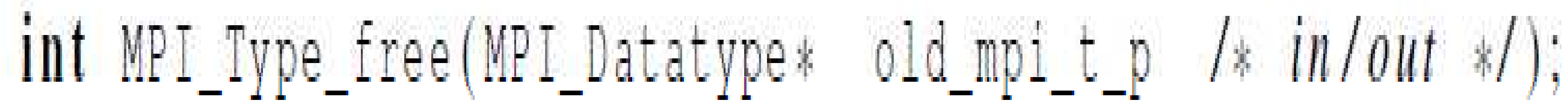
MPI_TYPE_COMMIT

- Allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

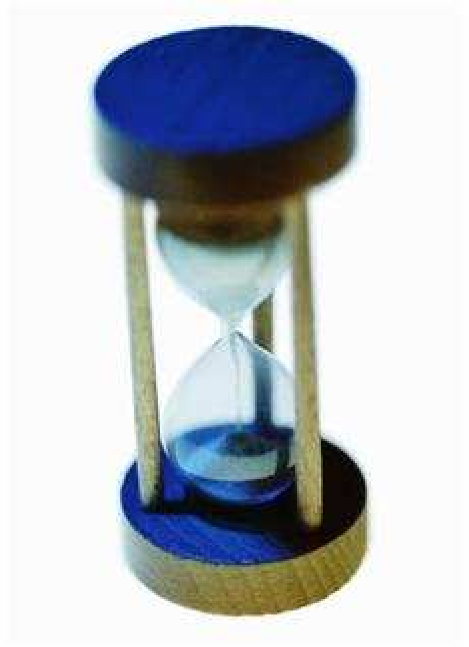
```
int MPI_Type_commit(MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

MPI_TYPE_FREE

- When we’re finished with our new type, this frees any additional storage used.



```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```



PERFORMANCE EVALUATION

Performance evaluation

- We denote the serial run-time by Tserial.
- it depends on the size of the input, n, we'll frequently denote it as Tserial(n).
- We denote the parallel run-time by Tparallel.
- it depends on both the input size, n, and the number of processes, comm_sz = p,
- we'll frequently denote it as Tparallel.(n,p).
-

Performance evaluation

- The parallel program will divide the work of the serial program among the processes, and add in some overhead time, which we denoted Toverhead:

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}.$$

Speedup

•The relation between the serial and the parallel run-times is the speedup.

•It’s the ratio of the serial run-time to the parallel run-time

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}$$

Speedup

- The ideal value for S(n,p) is **p**.
- If S(n,p) = **p**, then the parallel program with comm_ sz = p processes is running **p** times faster than the serial program.
- This speedup, sometimes called linear speedup, is rarely achieved.

Efficiency

- This is “per process” speedup

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}$$

- Linear speedup corresponds to a parallel efficiency of p/p =1.0.
- In general, we expect that our efficiencies will be less than 1.

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.

Scalability

- 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.