MODULE-3

Distributed memory programming with MPI –
MPI functions, The trapezoidal rule in MPI, Dealing with I/O, Collective communication, MPI-derived datatypes, Performance evaluation of MPI programs, A parallel sorting algorithm.

Distributed memory programming with MPI

In the world of **parallel MIMD (Multiple Instruction, Multiple Data)** systems, we typically distinguish between **distributed-memory** and **shared-memory** architectures. In a **distributed-memory system** (see **Fig. 3.1**), each **core** has access to its **own private memory**, and cores are connected via a **network**. A core can **only access its own memory** directly. Communication with other cores requires **explicit message-passing**.

In contrast, a **shared-memory system** (see **Fig. 3.2**) allows all cores to access a **common global memory**. This shared access simplifies some aspects of programming but introduces new challenges like **synchronization** and **cache coherence**.

To program distributed-memory systems, we use **message-passing programming models**. A program running on each core-memory pair is referred to as a **process**, and communication occurs using **send** and **receive** functions. The widely-used model is **MPI (Message-Passing Interface)**—a **standardized library** of functions callable from **C** and **Fortran**.

MPI supports not only basic point-to-point communication but also **collective communication**, where **multiple processes** participate in operations like **broadcast**, **gather**, **scatter**, and **reduce**. These **collective functions** are crucial for implementing scalable distributed algorithms.

While learning MPI, it's also necessary to understand the fundamental concerns in message-passing systems, such as **data partitioning**, which involves splitting the problem and data among different processes, and **distributed I/O**, which deals with efficiently managing input and output across processes.

These programming aspects directly impact **parallel program performance**, making it essential to study **communication cost**, **load balancing**, and **synchronization overheads** in distributed-memory environments.

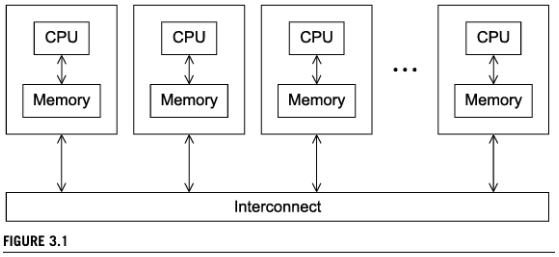

FIGURE 3.1
A distributed memory system.

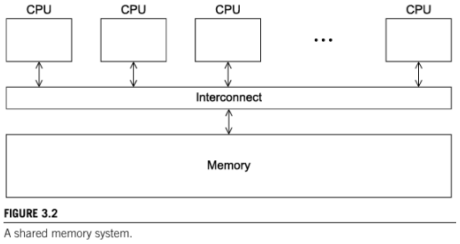

FIGURE 3.2
A shared memory system.

3.1 Getting Started

The traditional "hello, world" program from **Kernighan and Ritchie's text** introduces basic program output. In parallel programming using **MPI (Message-Passing Interface)**, we adapt

this by letting **only one process (rank 0)** handle the output, while the **other processes send messages** to it.

MPI identifies processes by **non-negative integer ranks**, ranging from 0 to p−1, where **p is the total number of processes**.

In this **MPI-based hello world** example (see **Program 3.1**), the workflow is:

- **Process 0** acts as the **receiver and printer**.
- All **other processes** use MPI_Send to send greeting messages to process 0.
- Process 0 uses MPI_Recv to **receive messages** and prints them.

3.1.1 Compilation and Execution

The method for **compiling and executing MPI programs** depends on the system in use. On most systems, developers write the code using a **text editor** and then use the **command line** to compile and run it.

```
1  #include <stdio.h>
2  #include <string.h>  /* For strlen             */
3  #include <mpi.h>     /* For MPI functions, etc */
4
5  const int MAX_STRING = 100;
6
7  int main(void) {
8     char        greeting[MAX_STRING];
9     int         comm_sz;  /* Number of processes */
10    int         my_rank;  /* My process rank     */
11
12    MPI_Init(NULL, NULL);
13    MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
14    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
15
16    if (my_rank != 0) {
17       sprintf(greeting, "Greetings from process %d of %d!",
18             my_rank, comm_sz);
19       MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
20             MPI_COMM_WORLD);
21    } else {
22       printf("Greetings from process %d of %d!\n",
23             my_rank, comm_sz);
24       for (int q = 1; q < comm_sz; q++) {
25          MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
26                0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
27          printf("%s\n", greeting);
28       }
29    }
30
31    MPI_Finalize();
32    return 0;
33 }  /* main */
```

Program 3.1: MPI program that prints greetings from the processes.

Compilation

Most systems provide an *mpicc* command to compile MPI programs. mpicc is a **wrapper script** around the C compiler. It automatically includes:

- Necessary **header files**
- Required **MPI libraries**

Example Command:

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

- -g enables debugging information
- -Wall enables all warnings
- -o mpi_hello specifies the output executable

Execution

Many systems also support program startup with `mpiexec`:

```
$ mpiexec -n <number of processes> ./mpi_hello
```

So to run the program with one process, we'd type

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

and to run the program with four processes, we'd type

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

With one process, the program's output would be

```
Greetings from process 0 of 1!
```

and with four processes, the program's output would be

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

- The command mpiexec **starts the specified number of instances** of the compiled MPI program.
- It may also control **which cores** the instances are assigned to.
- The **MPI implementation** ensures that these processes can **communicate with each other** as needed for the program to function.

3.1.2 MPI Programs

MPI programs are written in **standard C**, which means they start just like any other C program. For instance, the program includes the usual **C header files** such as <stdio.h> and <string.h>, and it defines a standard **main function**.

What distinguishes an MPI program is the inclusion of the **<mpi.h>** header file (as shown in **Line 3** of the code). This header is essential because it provides **function prototypes**, **macro definitions**, **type definitions**, and all other declarations required to compile an **MPI-based program**.

An important convention in MPI programming is the **naming scheme** used for its identifiers. All identifiers provided by MPI begin with **MPI_**. For function names and **MPI-defined types**, the first letter after the underscore is capitalized. For example:

- Function: MPI_Init, MPI_Send, MPI_Comm_rank
- Type: MPI_Comm, MPI_Status

In contrast, **MPI-defined macros and constants** are written in **all capital letters**, such as MPI_CHAR, MPI_COMM_WORLD, and MPI_STATUS_IGNORE. This consistent naming makes it easy to distinguish between **MPI-provided elements** and **user-defined identifiers**.

3.1.3 MPI_Init and MPI_Finalize

In **Line 12** of the program, the call to **MPI_Init** is crucial as it tells the **MPI system** to perform all necessary **initialization steps**. These steps may include allocating **message buffers** and assigning **ranks** to processes. Importantly, **no other MPI function** should be called before MPI_Init. Its syntax is:

```
int MPI_Init(int* argc_p, char*** argv_p);
```

The parameters argc_p and argv_p are pointers to the program's command-line arguments (argc and argv). However, if the program doesn't use command-line arguments, we can simply pass **NULL** for both. Like most MPI functions, MPI_Init returns an **integer error code**, which we usually **ignore** to avoid cluttering the code.

At the end of the program (**Line 31**), the call to **MPI_Finalize** signals that the program is done using MPI and any resources allocated can be released. Its syntax is very simple:

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

As a **rule**, no MPI calls should be made **after** MPI_Finalize.

A typical structure of an **MPI program** includes these functions as:

```
#include <mpi.h>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    // MPI code
    MPI_Finalize();
    return 0;
```

3.1.4 Communicators, MPI_Comm_size, and MPI_Comm_rank

In **MPI**, a **communicator** is a group of processes that can communicate with each other. One of the key tasks of MPI_Init is to create the default communicator **MPI_COMM_WORLD**, which includes **all processes** initiated when the program starts. To retrieve details about the processes in this communicator, we use:

```
int MPI_Comm_size(MPI_Comm comm, int* comm_sz_p);
int MPI_Comm_rank(MPI_Comm comm, int* my_rank_p);
```

The parameter comm is of type MPI_Comm and refers to the communicator (typically MPI_COMM_WORLD). The function MPI_Comm_size stores the **total number of processes** in comm_sz_p, while MPI_Comm_rank stores the **rank** (or ID) of the **calling process** in my_rank_p. Conventionally, we use the variables comm_sz for total number of processes and my_rank for the process's rank within MPI_COMM_WORLD.

3.1.5 SPMD Programs

Despite the fact that different processes might perform **different tasks**, we usually compile and execute a **single program** in MPI. This programming style is known as **Single Program, Multiple Data (SPMD)**. In our example, although **process 0** prints messages and the other processes **send messages**, all processes still run the **same compiled code**. The **if-else construct** in lines 16–29 helps differentiate the behavior based on each process's **rank**.

This approach makes the program **flexible and scalable**—it can run with **any number of processes**, such as 1, 4, 1000, or even more, depending on the system's resources. Even though MPI doesn't require this, it is **good practice** to write MPI programs that can adapt to **varying numbers of processes**, since the **available resources** can change from system to system or over time.

3.1.6 Communication

In the MPI greeting program, **each process except process 0** constructs a message using sprintf, which formats the message into a **string** instead of printing it to the console. Then, the message is sent to **process 0** using MPI_Send.

Process 0, on the other hand, prints its own greeting directly using printf, and then uses a **loop** to **receive messages** from all other processes (1 to comm_sz − 1) using MPI_Recv. This pattern illustrates a basic example of **point-to-point communication** in MPI, where **one process sends** and **another receives** a message.

3.1.7 MPI_Send

The function MPI_Send is used to **transmit messages** between processes. Its prototype is:

```
int MPI_Send(
  void* msg_buf_p,      // pointer to message buffer
  int msg_size,         // number of elements
  MPI_Datatype msg_type,// datatype of elements (e.g., MPI_CHAR)
  int dest,             // destination rank
  int tag,              // message tag
  MPI_Comm communicator // communicator group
);
```

- msg_buf_p points to the **data buffer** (e.g., greeting).
- msg_size and msg_type define **how much data** is sent (e.g., strlen(greeting)+1 with type MPI_CHAR).
- dest is the **rank of the destination** process.
- tag is a **message identifier**, used to distinguish between different types of messages.
- communicator (e.g., MPI_COMM_WORLD) defines the **scope** of communication.

Table 3.1 Some predefined MPI datatypes.

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	

Important: MPI types such as MPI_CHAR, MPI_INT, etc., are of type MPI_Datatype, not regular C types.

Example Use-Case for Tags: If process 1 sends some data to process 0 and it includes two kinds of data (e.g., print vs. compute), **tags** like 0 and 1 can help **differentiate** them.

Communicators help isolate communications. For example, if you're using **two independent MPI-based libraries** (say, atmosphere and ocean simulations), using **separate communicators** ensures their messages don't interfere with each other. This avoids complicated **tag-based coordination** and improves **modularity**.

MPI_Recv

The corresponding **receive function** is MPI_Recv:

```
int MPI_Recv(
  void* msg_buf_p,       // receive buffer
  int buf_size,          // max elements to receive
  MPI_Datatype buf_type, // type of elements
  int source,            // source rank
  int tag,               // message tag
  MPI_Comm communicator, // communicator group
  MPI_Status* status_p   // status info (can use MPI_STATUS_IGNORE)
);
```

- msg_buf_p is where the received message will be **stored**.
- buf_size and buf_type define **how much can be stored**.
- source specifies the **rank of the sender**.
- tag must **match** the sender's tag.
- communicator must match that of the sender.
- status_p is used to **inspect** additional info (e.g., actual source or tag); it can be ignored using MPI_STATUS_IGNORE.

MPI_Recv needs to **match** the send exactly in **data type, size, communicator, source, and tag** to complete successfully.

3.1.9 Message Matching

In MPI, for a message sent by one process to be **successfully received** by another, several parameters must **match** between the MPI_Send and MPI_Recv calls.

Matching Conditions

Suppose process **q** sends a message:

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

And process **r** receives a message:

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

The **message is received** successfully only if the following conditions hold:

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

These are necessary, but not sufficient. The **buffers** must also be **compatible**:

If recv_type == send_type **and** recv_buf_sz ≥ send_buf_sz, the message can be received

Unpredictable Order Handling

In many cases, the receiver doesn't know the **order** in which messages will arrive. For example:

- Process 0 distributes work to others (1, 2, ..., comm_sz−1)
- Those processes return results to process 0
- Some tasks may finish earlier, but process 0 **must wait in order** if using fixed ranks

This creates delays.

Solution: MPI_ANY_SOURCE

To allow receiving from **any process**, MPI provides the **wildcard constant** MPI_ANY_SOURCE.

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE, result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

This allows process 0 to **receive results as they come in**, **not** in process-rank order.

Wildcard for Tags

Similarly, if a process is expecting **multiple message types** (e.g., with different tags), it can use:

MPI_ANY_TAG

This enables the process to **receive any incoming message**, regardless of its tag.

Wildcard constants (MPI_ANY_SOURCE, MPI_ANY_TAG) can only be used in **receive** operations.

- Senders **must specify** both **destination rank** and **tag** explicitly.
- Hence, MPI uses a **push** model (data is sent to receivers), **not pull**.

No wildcard for communicators:

- Both sender and receiver **must agree** on the **communicator** (MPI_COMM_WORLD, etc.).

3.1.10 The status_p Argument

In **MPI**, a process can receive a message without prior knowledge of the **message size**, **sender**, or **tag**. This is made possible through the final argument of MPI_Recv, which is a pointer to a structure of type MPI_Status. This structure includes fields such as **MPI_SOURCE**, **MPI_TAG**, and **MPI_ERROR**. After a receive call like

MPI_Status status;

MPI_Recv(..., &status);

the **sender's rank** and the **message tag** can be accessed using status.MPI_SOURCE and status.MPI_TAG.

However, to determine the **number of elements** received, we use the function MPI_Get_count, since this information isn't directly stored in MPI_Status. Its syntax is:

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

This function computes the count based on the **data type** and stores it in count_p. The reason for using a function instead of a struct field is that the **count depends on data type** and might involve computations like dividing the number of received bytes by the size of the data type. To avoid **unnecessary overhead**, this calculation is only done when explicitly requested by the programmer.

3.1.11 Semantics of MPI_Send and MPI_Recv

When a process executes MPI_Send, the message is assembled with both **data** and **envelope information** such as **destination rank**, **sender rank**, **tag**, **communicator**, and **message size**. After this, two behaviors are possible: the MPI system may either **buffer** the message or **block** the sending process. If **buffered**, the message is stored internally and MPI_Send returns immediately. If **blocked**, the function waits until it can transmit the message. Importantly, once MPI_Send returns, we cannot be certain that the message has been transmitted—only that the **send buffer is safe for reuse**. For more control, MPI offers **alternative send functions** with different behaviors.

The behavior of MPI_Send also depends on the **message size**. Many implementations use a **cutoff size**: messages smaller than the cutoff are typically buffered, while larger ones cause blocking.

Unlike MPI_Send, MPI_Recv **always blocks** until a matching message has been received. Thus, after it returns, we are guaranteed that a message resides in the **receive buffer** (barring any errors). There also exists a **non-blocking alternative** for receiving, which checks availability and returns regardless.

MPI ensures that messages are **non-overtaking**: if **process q sends two messages** to **process r**, the **first message must be received before** the second. However, messages from **different processes (e.g., q and t)** may arrive in **any order**, regardless of their sending sequence. This flexibility is necessary since MPI cannot enforce network performance—e.g., **a message from Mars can't be guaranteed to arrive before one from San Francisco**, even if sent earlier.

3.1.12 Some Potential Pitfalls

A key pitfall in MPI programming lies in the semantics of **MPI_Recv**, which **blocks** until a **matching send** is issued. If no such send exists, the process will **hang indefinitely**. This highlights the importance of ensuring that **every receive has a corresponding send**. Coding

errors, such as mismatched **tags** or incorrect **source/destination ranks**, can prevent a match, causing unexpected hangs or, worse, messages being received by the **wrong process**.

On the sending side, a call to **MPI_Send** may either **block** or **buffer**. If it blocks and no matching receive is available, the **sending process will hang**. If it buffers but still lacks a matching receive, the **message will be lost**. Hence, meticulous attention must be paid during both **design and coding** to maintain correct communication patterns and avoid these silent failures.

3.2 The Trapezoidal Rule in MPI

While simple programs like printing messages help us understand MPI basics, the real benefit of **parallel programming** comes when we apply it to more useful computations. A good example is the **trapezoidal rule for numerical integration**, which is a method used to estimate the definite integral of a function. In a **serial implementation**, this involves dividing the integration interval into smaller subintervals, computing the function at each point, and applying the trapezoidal formula. In **MPI**, this process can be **parallelized** by dividing the integration interval among multiple processes. Each process computes the area under the curve over its assigned subinterval and sends its **partial result** to a **designated process** (usually rank 0), which then combines them to produce the **final result**. This example demonstrates how **computation** and **communication** are coordinated in MPI programs, moving from basic I/O tasks to solving actual mathematical problems.

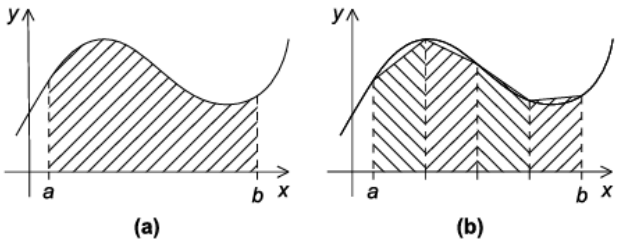

FIGURE 3.3

The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids.

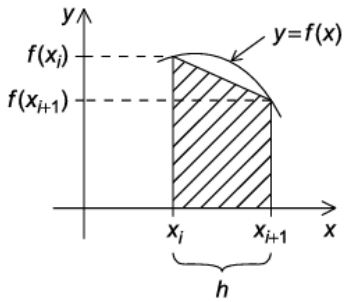

FIGURE 3.4

One trapezoid.

3.2.1 The Trapezoidal Rule

The **trapezoidal rule** provides a method to **numerically approximate definite integrals** by dividing the interval [a,b][a, b][a,b] into **n equal subintervals** and summing up the areas of **trapezoids** under the curve. As shown in **Figure 3.3**, we consider the area between the graph of y=f(x)y = f(x)y=f(x), two vertical lines, and the x-axis. Each subinterval is treated as a **trapezoid**, with its area approximated using the function values at the subinterval's endpoints. According to **Figure 3.4**, the height of each trapezoid is the **interval width** $h = \frac{b-a}{n}$, and its area is calculated as:

```
Area = (h / 2) * [f(xᵢ) + f(xᵢ₊₁)]
```

Summing the areas of all trapezoids gives the total approximation:

```
Total Area ≈ h * [f(x₀)/2 + f(x₁) + f(x₂) + ... + f(xₙ₋₁) + f(xₙ)/2]
```

Here, the x-values are uniformly spaced:

$x_0 = a, x_1 = a + h, ..., x_n = b$

A **serial pseudocode** for implementing this is:

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

This method provides a foundation for **parallelization using MPI**, where the interval and corresponding computations can be distributed across multiple processes.

3.2.2 Parallelizing the Trapezoidal Rule

To **parallelize** the trapezoidal rule, we follow the standard steps in parallel program design: **(1)** *partition* the computation into tasks, **(2)** *identify communication*, **(3)** *aggregate tasks*, and **(4)** *map tasks to cores*. In this case, we identify two main tasks: **computing the area** of each trapezoid and **summing the results**. As shown in **Figure 3.5**, each area-computing task sends its result to a central summing task.

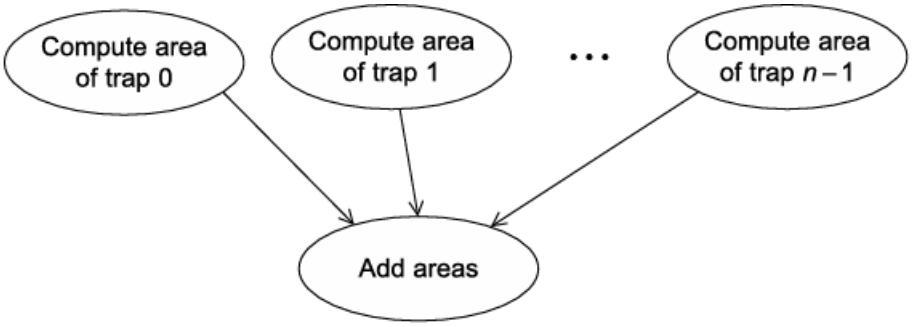

FIGURE 3.5

Tasks and communications for the trapezoidal rule.

Assuming the number of processes comm_sz evenly divides the number of trapezoids n, we can assign **n/comm_sz** trapezoids to each process. Each process computes a **local integral** over its subinterval [local_a, local_b]. After computing, processes **send their local results** to **process 0**, which **receives** and **sums** them to get the final approximation.

The **pseudocode** is as follows:

```
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 {
   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;
```

Here, Trap() is a function implementing the serial trapezoidal rule. Note the distinction between **local variables** (like local_a, local_b, and local_n, which are private to each process) and **global variables** (a, b, and n, which are meaningful across all processes). Although these definitions differ from traditional programming terms, the context in parallel programming makes them intuitive.

MPI program shown in Program 3.2.TheTrap function is just an implementation of the serial trapezoidal rule. See Program 3.3.

```c
int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;

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

   h = (b-a)/n;          /* h is the same for all processes */
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);

   if (my_rank != 0) {
      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
   } else {
      total_int = local_int;
      for (source = 1; source < comm_sz; source++) {
         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         total_int += local_int;
      }
   }

   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }
   MPI_Finalize();
   return 0;
} /*  main  */
```

Program 3.2: First version of MPI trapezoidal rule.

```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  */
```

Program 3.3: `Trap` function in MPI trapezoidal rule.

3.3 Dealing with I/O

The current version of the **parallel trapezoidal rule** has a major **limitation**: it only works for the interval **[0, 3]** using **1024 trapezoids**. While we *could* manually edit the code and recompile for new values, this is **inconvenient** compared to simply typing in **three new input values** during execution. Therefore, it's necessary to **address the issue of user input** in

parallel programs. And while discussing **input handling**, it's also a good opportunity to look at **output mechanisms** in parallel MPI programs.

3.3.1 Output

In both the **"greetings" program** and the **trapezoidal rule program**, we assumed that **process 0** can write to **stdout** via printf, and this behaves as expected. While the **MPI standard** doesn't strictly define which processes can access I/O devices, most **MPI implementations** allow all processes in **MPI_COMM_WORLD** to access both **stdout** and **stderr**. However, there is **no automatic scheduling** of output access. If **multiple processes** try to write simultaneously to **stdout**, the output order becomes **unpredictable**, and one process's output might even be **interrupted** by another's.

For instance, when running a program (as in *Program 3.4*) where each process prints a line, the output order may appear sequential with **five processes**, but becomes **nondeterministic** with **six or more processes**. This behavior occurs because the processes **compete** for access to **shared output**, leading to **race conditions** in printing. The result is **nondeterminism**—output may differ between runs.

To maintain a **predictable output order**, it's the programmer's responsibility to **manage output**. A common solution is to have each **non-zero process** send its output as a **message** to **process 0**, which then **prints** in **rank order**—as demonstrated in the **"greetings" program**.

```
#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 */
```

Program 3.4: Each process just prints a message.

3.3.2 Input

Unlike output, **most MPI implementations** allow **only process 0** in **MPI_COMM_WORLD** to access **stdin**. This restriction is **logical**, since if multiple processes accessed **stdin**, it would be ambiguous as to **which process** should receive **which part** of the input. For example, should **process 0** receive the first line and **process 1** the second, or should input be distributed **character-by-character**?

To handle input using functions like **scanf**, MPI programs typically **branch on process rank**. That is, **process 0** reads the input and then **broadcasts** or **sends** it to the other processes. For instance, in the **parallel trapezoidal rule program**, the Get_input function (see *Program 3.5*) allows **process 0** to read the values for

```
1  void Get_input(
2        int       my_rank   /* in  */,
3        int       comm_sz   /* in  */,
4        double*   a_p       /* out */,
5        double*   b_p       /* out */,
6        int*      n_p       /* out */) {
7     int dest;
8
9     if (my_rank == 0) {
10       printf("Enter a, b, and n\n");
11       scanf("%lf %lf %d", a_p, b_p, n_p);
12       for (dest = 1; dest < comm_sz; dest++) {
13          MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14          MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15          MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16       }
17    } else { /* my_rank != 0 */
18       MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
19             MPI_STATUS_IGNORE);
20       MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
21             MPI_STATUS_IGNORE);
22       MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
23             MPI_STATUS_IGNORE);
24    }
25 }  /* Get_input */
```

Program 3.5: A function for reading user input.

a, **b**, and **n**, and then **sends** these values to each of the other processes. This communication follows the same pattern as in the **"greetings" program**, except that now, **process 0** sends the data, while the other processes **receive** it.

To use this input function, a call to Get_input should be inserted inside **main**, after initializing **my_rank** and **comm_sz**.

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

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

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

3.4 Collective Communication

In our trapezoidal rule MPI program, one inefficiency is evident in how the **global sum** is computed. After calculating local integrals, every process **sends** its value to **process 0**, which

alone performs the summation. This results in **poor workload distribution**, as process 0 does most of the work, while the other processes become idle after sending. Such imbalance is inefficient, especially when resources are underutilized. A better approach would be to **distribute the summation work**, so all processes contribute equally, similar to how students can collaboratively add numbers rather than relying on one to sum them all.

3.4.1 Tree-Structured Communication

An efficient solution is the use of **tree-structured communication**, such as a **binary tree** (see **Fig. 3.6**). Initially, pairs like processes 1 & 0, 3 & 2, 5 & 4, and 7 & 6 exchange and add values. In successive steps, the **intermediate results** are sent and summed by fewer processes, until **process 0** receives the final total. This method reduces the number of receives and additions **by process 0** from comm_sz - 1 to just $\log_2(\text{comm\_sz})$ operations. For example, with comm_sz = 1024, the number of operations drops from **1023** to just **10**, significantly improving performance.

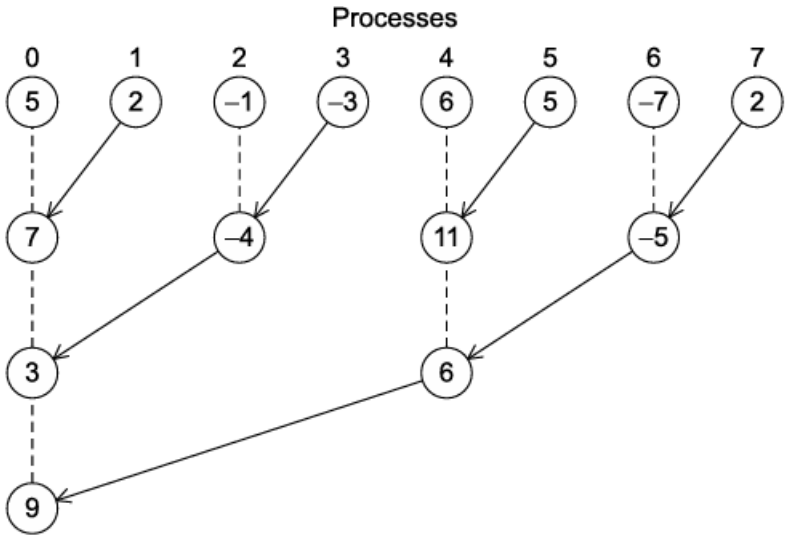

FIGURE 3.6

A tree-structured global sum.

The advantage lies in **concurrent execution**—several adds and receives happen **simultaneously** in early phases, which reduces **total execution time**. However, **coding this manually** can be complex, especially when choosing optimal **pairings** (e.g., pairing 0 & 4, 1 & 5, etc., as in **Fig. 3.7**). Each strategy may perform differently based on the **system architecture** and **number of processes**, making it difficult to predict the best approach without **experimentation** or **system-specific optimization**.

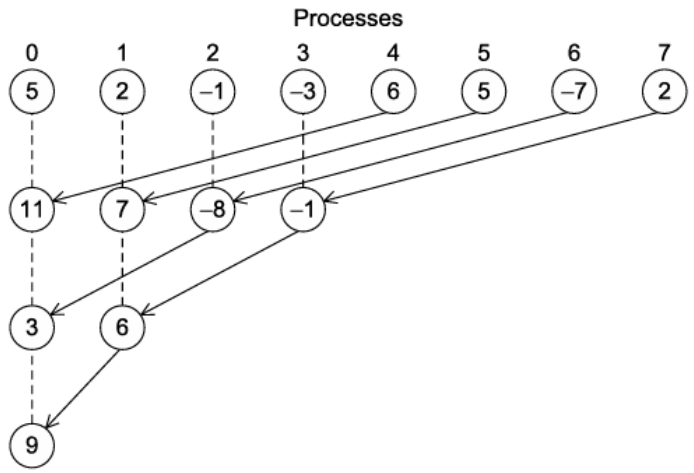

FIGURE 3.7

An alternative tree-structured global sum.

3.4.2 MPI_Reduce

To avoid redundant effort in writing optimized global sum functions, MPI offers a built-in collective communication routine called **MPI_Reduce**, which shifts the responsibility of optimization to the **MPI implementation**. This function enables multiple processes in a communicator (like MPI_COMM_WORLD) to perform a **reduction operation**—such as **sum, maximum, minimum, or product**—on data distributed across processes. Such **collective communications** contrast with **point-to-point operations** (MPI_Send, MPI_Recv) by involving **all processes** in the communicator.

The prototype for MPI_Reduce is:

```
int MPI_Reduce(
    void* input_data_p,     // input buffer
    void* output_data_p,    // result buffer (valid only at root)
    int count,              // number of elements
    MPI_Datatype datatype,  // type of data (e.g., MPI_DOUBLE)
    MPI_Op operator,        // operation to perform (e.g., MPI_SUM)
    int dest_process,       // rank of root process
    MPI_Comm comm           // communicator (e.g., MPI_COMM_WORLD)
);
```

The **fifth parameter**—the operator of type MPI_Op—is central to its generality. MPI defines several built-in operations for this parameter, such as **MPI_SUM**, **MPI_PROD**, **MPI_MAX**, and **MPI_MIN**, which are listed in **Table 3.2**. Programmers can also define **custom reduction operations** using MPI_Op_create.

Table 3.2 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

In the **parallel trapezoidal rule program**, this call replaces the manual summation logic in the root process:

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

This line aggregates local_int from each process into total_int at **process 0**. For arrays, MPI_Reduce works similarly. For instance, with double local_x[N];, one can compute a component-wise sum as:

```
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Thus, **MPI_Reduce provides a simple, efficient, and portable** way to perform global reductions across processes.

3.4.3 Collective vs. Point-to-Point Communications

Collective communications in MPI differ significantly from point-to-point communications in the following ways:

1. **All processes in the communicator must call the same collective function**. For instance, calling MPI_Reduce on one process and MPI_Recv on another is incorrect and will likely **cause the program to hang or crash**.
2. **Arguments passed by each process must be compatible**. If one process uses dest_process = 0 and another uses dest_process = 1 in a call to MPI_Reduce, the result is erroneous.
3. **The output_data_p argument is used only by the destination process**, but all processes must still pass a valid argument (e.g., NULL for non-destination processes).
4. **Collective communications do not use tags**, unlike point-to-point operations. They are matched **only by communicator and the order of calls**. As shown in **Table 3.3**, if each process calls MPI_Reduce twice, the sequence determines matching—not variable names—so values accumulate in the order of function invocation.

Table 3.3 Multiple Calls to `MPI_Reduce`.

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, ...)`

A final **cautionary point**: using the **same buffer for input and output** (e.g., MPI_Reduce(&x, &x, ...)) is **illegal in MPI**, due to **aliasing** restrictions. This restriction, rooted in Fortran compatibility, ensures that output arguments don't overlap with input memory, preventing undefined or erroneous behavior.

3.4.4 MPI_Allreduce

In some parallel programs, **all processes may need access to the result of a global computation**, such as a sum. While MPI_Reduce provides the result only to a designated destination process, **MPI_Allreduce ensures that the result is distributed to all processes in the communicator**. This eliminates the need to implement custom distribution schemes, such as reversing the branches of a tree structure (as in **Fig. 3.8**) or designing a **butterfly communication pattern** (**Fig. 3.9**), which could be complex to code and optimize.

MPI handles this efficiently through the following function:

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

The parameters are identical to those of MPI_Reduce, except that **there is no dest_process**—because **every process receives the result**. This function simplifies parallel programming by removing the burden of manually distributing aggregated data across processes and allows all processes to **proceed with further computation using the shared result**.

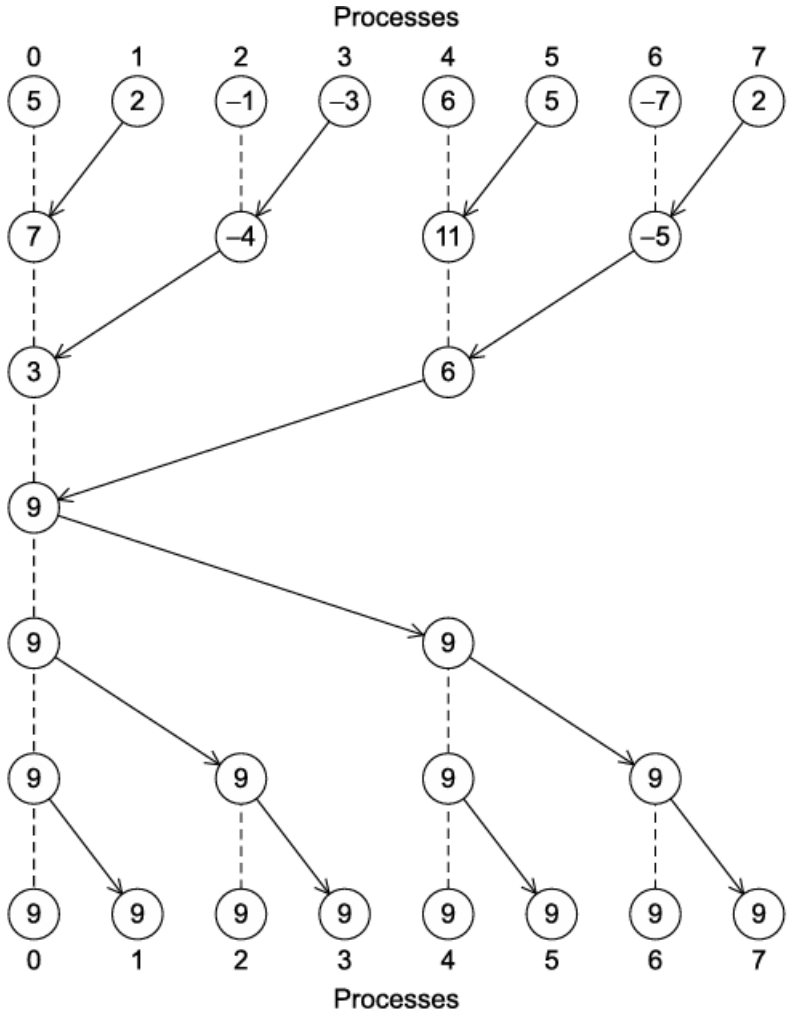

FIGURE 3.8

A global sum followed by distribution of the result.

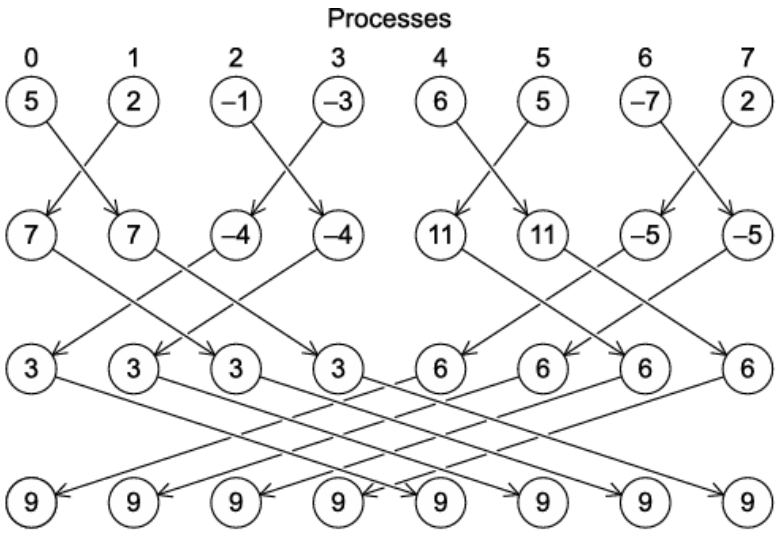

FIGURE 3.9

A butterfly-structured global sum.

To optimize **input distribution** in a parallel program—similar to how we optimized the global sum using tree-structured communication—we can use **broadcasting**. A **broadcast** is a collective communication operation where **data from a single process (the source)** is **sent to all other processes** in the communicator. For example, if process 0 reads the input data, it can **broadcast** the values of a, b, and n to all other processes using a tree structure like the one in **Fig. 3.10**.

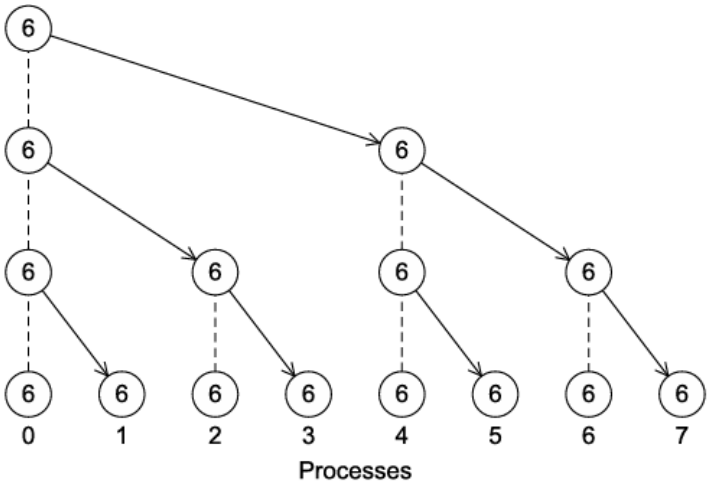

FIGURE 3.10

A tree-structured broadcast.

MPI provides the MPI_Bcast function for this purpose:

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

In this function, the process identified by source_proc sends data to all processes in the communicator comm. The data_p parameter is an **input** on the source process and an **output** on all others. Although labeled **in/out**, its role depends on the process's rank.

Using MPI_Bcast, we **avoid writing separate send operations to each process** and benefit from **optimized implementations** provided by MPI—just like with MPI_Reduce.

```
1  void Get_input(
2        int       my_rank  /* in  */,
3        int       comm_sz  /* in  */,
4        double*   a_p      /* out */,
5        double*   b_p      /* out */,
6        int*      n_p      /* out */) {
7
8     if (my_rank == 0) {
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```

Program 3.6: A version of `Get_input` that uses `MPI_Bcast`.

When testing a **vector addition function** in MPI, it's helpful to input the dimension and elements of vectors x and y. While **MPI_Bcast** works well for broadcasting the vector size, using it to broadcast entire vectors would be inefficient. For example, with 10 processes and vectors of 10,000 elements, broadcasting the full vectors would force every process to allocate memory for 10,000 elements—despite only working on a subset.

To solve this, MPI offers **MPI_Scatter**, which allows **process 0 to distribute only the required segment** of a vector to each process. This enables each process to **store and operate only on its assigned subvector**, greatly optimizing memory usage. The function signature is:

```
int MPI_Scatter(
  void* send_buf_p,     // in
  int send_count,       // in (per process)
  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
);
```

If using a **block distribution** and if n is divisible by comm_sz, then send_count and recv_count should both be local_n = n / comm_sz, and send_type / recv_type should be MPI_DOUBLE. Each process receives its local_n chunk from send_buf_p, while process 0 reads and holds the entire input vector. The MPI_Scatter call sends the **first block to process 0, second to process 1, and so on**, making it ideal for **block distributions** only. This approach is used in the Read_vector function in **Program 3.9**.

```
1  void Read_vector(
2        double     local_a[]   /* out */,
3        int        local_n     /* in  */,
4        int        n           /* in  */,
5        char       vec_name[]  /* in  */,
6        int        my_rank     /* in  */,
7        MPI_Comm   comm        /* in  */) {
8
9     double* a = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       a = malloc(n*sizeof(double));
14       printf("Enter the vector %s\n", vec_name);
15       for (i = 0; i < n; i++)
16          scanf("%lf", &a[i]);
17       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
18             MPI_DOUBLE, 0, comm);
19       free(a);
20    } else {
21       MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
22             MPI_DOUBLE, 0, comm);
23    }
24 } /* Read_vector */
```

Program 3.9: A function for reading and distributing a vector.

A key caveat: this strategy **only works efficiently when n is evenly divisible by comm_sz** and block distribution is used.

3.4.8 Gather

To make a vector addition program useful, we need a way to **view the results**. Since the resulting vector is distributed across processes, we need to **collect all subvector results onto process 0**, where they can be printed in full. MPI provides the collective function **MPI_Gather** for this purpose. The function allows each process to send its portion of the vector to a single destination process (typically rank 0), which stores the data in the correct order.

The function prototype is:

```
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
);
```

Each process sends the contents of send_buf_p to dest_proc. On the destination process (e.g., process 0), these chunks are stored in recv_buf_p in **rank order**: the data from **process 0 goes into the first block**, from **process 1 into the second**, and so on. So, if using **block distribution** with each process holding local_n elements, then recv_count should also be local_n. Importantly, recv_count refers to the number of elements **from each process**, **not the total number** collected.

This is effectively the **reverse operation of MPI_Scatter**, and similar constraints apply: it **assumes all blocks are of equal size** and works best with **block-distributed vectors**. The gathering logic for printing a distributed vector is implemented in **Program 3.10**.

```
1  void Print_vector(
2        double     local_b[]  /* in */,
3        int        local_n    /* in */,
4        int        n          /* in */,
5        char       title[]    /* in */,
6        int        my_rank    /* in */,
7        MPI_Comm   comm       /* in */) {
8
9     double* b = NULL;
10    int i;
11
12    if (my_rank == 0) {
13       b = malloc(n*sizeof(double));
14       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
15             MPI_DOUBLE, 0, comm);
16       printf("%s\n", title);
17       for (i = 0; i < n; i++)
18          printf("%f ", b[i]);
19       printf("\n");
20       free(b);
21    } else {
22       MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
23             MPI_DOUBLE, 0, comm);
24    }
25 } /* Print_vector */
```

Program 3.10: A function for printing a distributed vector.

3.4.9 Allgather

To perform **parallel matrix-vector multiplication**, we begin with the mathematical formulation: if **A** is an m×n matrix and **x** is a vector of n components, then **y = Ax** results in a vector **y** of mmm components, where $y_i = a_{i0}x_0 + a_{i1}x_1 + \ldots + a_{i,n-1}x_{n-1}$To parallelize this, we use **block row distribution** for matrix A so that each process gets a set of rows, and the corresponding **block distribution of y** so that process q computes the values of y for its rows of A.

The challenge arises with vector **x**, since each component of y requires **all components of x**. In a serial code, this is straightforward, but in parallel code—especially in **iterative applications** where y becomes the next x—we typically use the **same block distribution** for both x and y. To compute the result, each process needs the **entire vector x**, not just its block.

One naïve approach is to use **MPI_Gather** followed by **MPI_Bcast**, but this introduces redundant communication. Instead, **MPI provides MPI_Allgather**, a collective operation that gathers data from all processes and distributes the complete result to all processes in a single call.

```
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
);
```

This function collects send_buf_p from each process and **concatenates the segments** into recv_buf_p on **every process**. Each process ends up with a **full copy of the distributed vector**, enabling them to compute their dot-products locally. Typically, send_count = recv_count, and the function is efficient for **repeated calls**, such as in iterative solvers, where x is reused. For performance, it's advisable to allocate x once in the **calling function** and pass it to the matrix-vector multiplication routine, as done in **Program 3.12**.

```
1  void Mat_vect_mult(
2        double    local_A[]  /* in  */,
3        double    local_x[]  /* in  */,
4        double    local_y[]  /* out */,
5        int       local_m    /* in  */,
6        int       n          /* in  */,
7        int       local_n    /* in  */,
8        MPI_Comm  comm       /* in  */) {
9     double* x;
10    int local_i, j;
11    int local_ok = 1;
12
13    x = malloc(n*sizeof(double));
14    MPI_Allgather(local_x, local_n, MPI_DOUBLE,
15          x, local_n, MPI_DOUBLE, comm);
16
17    for (local_i = 0; local_i < local_m; local_i++) {
18       local_y[local_i] = 0.0;
19       for (j = 0; j < n; j++)
20          local_y[local_i] += local_A[local_i*n+j]*x[j];
21    }
22    free(x);
23 }  /* Mat_vect_mult */
```

Program 3.12: An MPI matrix-vector multiplication function.

3.5 MPI-derived Datatypes

In **distributed-memory systems**, communication is significantly more **costly** than local computation. For instance, sending a double between nodes is far slower than a local addition. Moreover, sending many small messages is much less efficient than sending one large message. As demonstrated in the example with double x[1000], a loop sending individual elements takes **50–100× longer** than sending the entire array in a single call. To **reduce message overhead**, MPI supports three strategies: using the **count** parameter in communication functions, **derived datatypes**, and **MPI_Pack/Unpack**.

Derived datatypes allow grouping multiple memory elements—including those of different types—into a single transferable unit. For instance, in the trapezoidal rule program, three separate MPI_Bcast calls (for a, b, and n) can be replaced with a single call using a derived

datatype composed of **two MPI_DOUBLEs** and **one MPI_INT**. This requires knowledge of the **displacements** of variables in memory. Suppose the addresses of a, b, and n are 24, 40, and 48 respectively. Then the derived type is defined by:

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

To construct this derived datatype, MPI provides:

```
c

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

For our example:

```
c

int array_of_blocklengths[3] = {1, 1, 1};
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};
```

Displacements can be computed using:

```
c

MPI_Get_address(&a, &a_addr);
MPI_Get_address(&b, &b_addr);
MPI_Get_address(&n, &n_addr);
```

Then:

```
c

array_of_displacements[0] = 0;
array_of_displacements[1] = b_addr - a_addr;
array_of_displacements[2] = n_addr - a_addr;
```

After building the datatype using `MPI_Type_create_struct`, we must **commit it** with:

```
c

MPI_Type_commit(&input_mpi_t);
```

And then we can broadcast all three values at once:

```
c

MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

When done, we **free** the derived type using:

```
c

MPI_Type_free(&input_mpi_t);
```

This method simplifies communication and enhances performance. These steps are implemented in a reusable Build_mpi_type function, which is called from an updated Get_input routine (see **Program 3.13**). This technique is highly useful for compact communication in **heterogeneous data structures**.

```
void Build_mpi_type(
      double*         a_p              /* in  */,
      double*         b_p              /* in  */,
      int*            n_p              /* in  */,
      MPI_Datatype*   input_mpi_t_p    /* out */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE,
      MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};

   MPI_Get_address(a_p, &a_addr);
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */

void Get_input(int my_rank, int comm_sz, double* a_p,
      double* b_p, int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   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, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

Program 3.13: The `Get_input` function with a derived datatype.

3.6 Performance Evaluation of MPI Programs

One of the main reasons we write **parallel MPI programs** is to achieve better performance compared to serial versions. But how do we measure this improvement? The key is to evaluate **only the computational part** (like matrix-vector multiplication), ignoring unrelated tasks like input and output.

MPI provides a built-in timing function called **MPI_Wtime()**, which returns the **elapsed wall clock time** (real-world time, including wait and idle periods). This differs from CPU time (returned by functions like clock()), which doesn't account for waiting during communication.

3.6.1 Taking Timings

When timing MPI code, we typically **exclude non-computational steps** (e.g., input, output). We use MPI_Wtime() as follows:

```
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);
```

For **serial code**, MPI is not needed. We can use the **GET_TIME macro** (defined in timer.h, downloadable from the book's website) which wraps gettimeofday():

```
#include "timer.h"
double start, finish;
GET_TIME(start);
/* Code to be timed */
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish - start);
```

Be aware that GET_TIME is a **macro**, not a function, and takes a **double** (not a pointer). Also, if timer.h is not in your working directory, compile with:

```
gcc -g -Wall -I/home/user/my_include -o prog prog.c
```

Both **MPI_Wtime()** and **GET_TIME** report **wall clock time**, which includes communication delays (e.g., time spent waiting in MPI_Recv). This is more relevant in parallel environments where idle time impacts real performance.

To obtain a **single elapsed time** for the parallel run (instead of separate times for each process), we use **MPI_Barrier** to synchronize processes and **MPI_Reduce** with the **MPI_MAX** operator to gather the **maximum elapsed time**:

```
double local_start, local_finish, local_elapsed, elapsed;
MPI_Barrier(comm);  // Synchronize all processes
local_start = MPI_Wtime();
/* Code to be timed */
local_finish = MPI_Wtime();
local_elapsed = local_finish - local_start;

MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE, MPI_MAX, 0, comm);

if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed);
```

This gives a more realistic **total parallel execution time**.

Also, we must account for **variability** in timings due to **OS activity** and other system-level factors. Even with identical input and system load, execution time may vary across runs. Hence, we usually report the **minimum observed runtime**, assuming the fastest run occurred with the least interference.

When running MPI on **hybrid multicore systems**, it's often more efficient to run **one MPI process per node**, reducing communication overhead and contention, leading to more consistent and improved timings.

3.6.2 Results

The **timing results** for the **matrix-vector multiplication** program are summarized in **Table 3.5**. The input matrices are **square**, and times are shown in **milliseconds**, rounded to **two significant digits**.

- For **comm_sz = 1**, the timings represent a **serial run** on a single core.
- As expected, for a fixed number of processes, increasing the matrix size **n** leads to longer runtimes.
- For **small numbers of processes**, doubling **n** roughly **quadruples** the runtime. But for **larger process counts**, this pattern does not always hold.

Table 3.5 Run-times of serial and parallel matrix-vector multiplication (times are in milliseconds).

	Order of Matrix				
comm_sz	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

If we **fix n** and **increase comm_sz** (number of processes):

- **Runtime usually decreases**, especially for **large values of n**.
- Doubling the number of processes roughly **halves the runtime**.
- But for **small n**, adding more processes offers **diminishing or no benefit**.

  For instance:

  ```
  T_parallel(1024, 8) ≈ T_parallel(1024, 16)
  ```

  This behavior is **typical** of parallel programs. As **problem size increases**, runtimes increase **regardless of the number of processes**, though the **rate of increase** depends on the number of processes:

- **One-process times** increase steadily.
- **Multi-process times** can fluctuate (especially at 16 processes).

A useful performance model expresses **parallel time** as:

```
T_parallel(n, p) = T_serial(n) / p + T_overhead
```

Where:

- **T_serial(n)** is the time taken by the **serial program** on input size **n**.
- **T_parallel(n, p)** is the time taken by the **parallel program** on input size **n** using **p processes**.
- **T_overhead** represents the **parallelization cost**, often due to **communication**, especially in **MPI programs**.

For the matrix-vector multiplication, the serial code has two nested loops:

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

This performs:

- **2n** floating point operations per row (n multiplications + n additions),
- Executed **m times**, giving **2mn** total floating point operations.

So, if **m = n**, the serial runtime can be approximated as:

```
T_serial(n) ≈ a * n²
```

(for some constant **a**).

In the **parallel version**:

- Each process handles **(n/p) × n** elements.
- So the local computation time is approximately:

```
T_local_compute = a * n² / p
```

But there's an additional overhead due to **MPI_Allgather**, so the parallel time becomes:

```
T_parallel(n, p) = T_serial(n) / p + T_allgather
```

Empirical Observations

For **small p (2 or 4)** and **large n**, the runtime behaves close to **T_serial(n) / p**:

Doubling **p** approximately halves the runtime:

```
T_serial(4096) ≈ 1.9 × T_parallel(4096, 2)
T_parallel(8192, 2) ≈ 2.0 × T_parallel(8192, 4)
```

Doubling **n** roughly quadruples the time:

```
scss

T_serial(4096) ≈ 4.0 × T_serial(2048)
T_parallel(4096, 2) ≈ 3.9 × T_parallel(2048, 2)
```

Thus, the **MPI_Allgather overhead** appears to have **little impact** when:

- **n is large**, and **p is small**.

Breakdown for Small n and Large p

However, the approximation **fails** when:

- **n is small**, and **p is large**.

For example:

```
T_parallel(1024, 8) = T_parallel(1024, 16)
T_parallel(2048, 16) = 1.5 × T_parallel(1024, 16)
```

This indicates that for **small problem sizes**, the **communication cost (T_allgather)** dominates the performance, limiting scalability.

3.6.3 Speedup and Efficiency

An essential metric in evaluating parallel program performance is **speedup**, denoted as:

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

This represents how many times **faster** the parallel program runs with **p processes** compared to the serial version on input of size **n**. The **ideal case** is when:

$$S(n,p) = p$$

This is called **linear speedup**, indicating perfect scaling—i.e., a program with 16 processes is 16 times faster than the serial one. However, **linear speedup is rare in practice** due to communication overhead and load imbalance.

In the case of the **matrix-vector multiplication program**, the **speedups** obtained are listed in **Table 3.6**. Observations include:

Table 3.6 Speedups of parallel matrix-vector multiplication.

	Order of Matrix				
comm_sz	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

- For **small p** and **large n**, the program achieved **near-linear speedup**.
- For **large p** and **small n**, the speedup was **significantly less** than p.
- The **worst performance** occurred at **n = 1024** with **p = 16**, where the speedup dropped to **only 2.4**, far from the ideal value of 16.

Another related metric is **parallel efficiency**, defined as the **speedup per process**:

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

A value of **1.0** indicates **perfect efficiency**, where each process contributes fully to the speedup. In general, efficiency is **less than 1**, and decreases as **communication and synchronization overheads** increase.

The **efficiencies** for matrix-vector multiplication are shown in **Table 3.7**:

- When **n is large** and **p is small**, efficiency is **close to 1.0**.
- As **p increases** and **n remains small**, efficiency drops **significantly**.

This reinforces the principle that **parallelism pays off** when applied to **large problems** and that adding more processes to small problems often results in **poor scalability**.

Table 3.7 Efficiencies of parallel matrix-vector multiplication.

	Order of Matrix				
comm_sz	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

3.6.4 Scalability

When analyzing the performance of parallel programs, it's not enough to look only at **speedup** and **efficiency**. Another crucial factor is **scalability**.

A parallel program is said to be **scalable** if the **problem size can be increased at a rate that maintains the same efficiency** as the number of processes (**p**) increases. In other words, a **scalable program keeps working well** even when we **use more processors**, provided we **increase the workload proportionally**.

However, this definition has some ambiguity—specifically in the phrase: *"problem size can be increased at a rate..."* Let's examine two hypothetical programs:

- **Program A** maintains a **constant efficiency of 0.75** for all p≥2, **regardless of problem size**.
- **Program B** has efficiency defined as:

  $$E(n,p) = \frac{n}{625p}, \quad \text{for } p \geq 2 \text{ and } 1000 \leq n \leq 625p$$

  In Program B, if we **increase both n and p** proportionally, we can **maintain constant efficiency**. For example:

  - When $n = 1000, p = 2$, then $E = \frac{1000}{1250} = 0.8$
  - If we double both: $n = 2000, p = 4$, then $E = \frac{2000}{2500} = 0.8$

  So **both programs are technically scalable**, but **Program A is more scalable** because it does not even require an increase in problem size to maintain efficiency. This leads to two common categories of scalability:

- **Strong scalability**: Constant efficiency **without** increasing problem size (Program A).
- **Weak scalability**: Constant efficiency **if** the problem size increases proportionally with the number of processes (Program B).

Matrix-Vector Multiplication and Scalability

Looking at the **parallel efficiency values in Table 3.7**, we can conclude:

- The matrix-vector multiplication program is **not strongly scalable**. For nearly every case, **increasing the number of processes results in reduced efficiency**.
- However, the program shows traits of **weak scalability**. When both **p** and **n** are **doubled**, **parallel efficiency often improves**.

  For instance:

- Increasing p from 4 to 8 or 8 to 16 **with a corresponding increase in n** consistently improves or maintains efficiency.
- The **only slight deviation** happens when p = 2 and p=4, but this is usually not a concern, since scalability discussions often focus on **larger processor counts**.

  Thus, the **matrix-vector multiplication program is weakly scalable**. While not ideal for small problems or low processor counts, it **handles larger data and processor configurations reasonably well**—a **desirable property** in large-scale parallel computing.

3.7 A Parallel Sorting Algorithm

In a **distributed memory environment**, a *parallel sorting algorithm* deals with data (keys) spread across multiple processes. The **input and output** depend on the data distribution—keys can either be **distributed among processes** or **assigned to a single process**. In this case, we focus on an algorithm where both **input and output are distributed**, meaning each process starts and ends with keys, assuming n is divisible by the number of processes p=comm_sz. Initially, keys are **randomly distributed** across processes. The goal is to ensure that, after execution:

- Each process's local keys are **sorted in increasing order**, and
- For any two processes q<r, **every key in process q** is **less than or equal to** every key in process r.

Thus, if we concatenate the keys from all processes in order of increasing rank, we get a **fully sorted list**. For simplicity, we assume the keys are of type int. This sets the stage for designing efficient and scalable **parallel sorting algorithms** that operate correctly and coherently in a **distributed setting**.

3.7.1 Some Simple Serial Sorting Algorithms

To better understand parallel sorting, let's briefly review a couple of **serial sorting techniques**. One well-known method is **Bubble Sort**, which repeatedly compares and swaps adjacent values to move larger elements toward the end of the list. On each outer loop pass, the largest unsorted value "bubbles" into its correct position, and the unsorted portion shrinks by one.

However, **Bubble Sort is inherently sequential**—a swap at one position may affect adjacent comparisons. For instance, a sequence like 9, 5, 7 needs ordered swaps to correctly become 5, 7, 9; swapping 5 and 7 prematurely would yield the wrong result.

```
void Bubble_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int list_length, i, temp;

   for (list_length = n; list_length >= 2; list_length--)
      for (i = 0; i < list_length-1; i++)
         if (a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
         }

}  /* Bubble_sort */
```

Program 3.14: Serial bubble sort.

To introduce **parallelism**, we turn to **Odd-Even Transposition Sort**, which breaks dependencies using phases:

- **Even phases** compare and swap: (a[0], a[1]), (a[2], a[3]), ...
- **Odd phases** compare and swap: (a[1], a[2]), (a[3], a[4]), ...

Example:

```
Start:       5, 9, 4, 3
Even phase:  5, 9, 3, 4
Odd phase:   5, 3, 9, 4
Even phase:  3, 5, 4, 9
Odd phase:   3, 4, 5, 9 (Sorted)
```

A **theorem guarantees** that *n elements will be sorted in at most n phases*.

3.7.2 Parallel Odd-Even Transposition Sort

In a **distributed memory system**, the same concept is extended across **p processes**, each holding n/p elements (assuming n is evenly divisible by p).

When n=p (each process has one key), the algorithm is intuitive: At each phase, process i exchanges keys with either i-1 or i+1 based on phase type (even or odd), and keeps either the **smaller** or **larger** key based on its rank.

But this is impractical for real applications (we usually have n>p), and communication overhead outweighs benefits for small inputs.

```
void Odd_even_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int phase, i, temp;

   for (phase = 0; phase < n; phase++)
      if (phase % 2 == 0) { /* Even phase */
         for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               temp = a[i];
               a[i] = a[i-1];
               a[i-1] = temp;
            }
      } else { /* Odd phase */
         for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) {
               temp = a[i];
               a[i] = a[i+1];
               a[i+1] = temp;
            }
      }
}  /* Odd_even_sort */
```

Program 3.15: Serial odd-even transposition sort.

When each process holds multiple keys, the process is:

1. **Locally sort keys** using fast serial sort like qsort.

2. In each phase:
    - Determine a **partner** (left or right neighbor depending on phase).
    - **Exchange full key arrays** with the partner.
    - **Merge the two arrays**:
        - Lower-ranked process keeps the **smaller half**.
        - Higher-ranked process keeps the **larger half**.

    This ensures that:

- Each process maintains sorted keys.
- Global ordering progresses toward a fully sorted distributed list.

Table 3.8 illustrates this procedure across 4 processes and 16 keys.

Table 3.8 Parallel odd-even transposition sort.

	Process			
Time	**0**	**1**	**2**	**3**
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

Theorem

If parallel odd-even transposition sort is run with **p processes**, then after **p phases**, the **entire input list will be sorted.**

Pseudocode

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank);
    if (partner != MPI_PROC_NULL) {
        Send my keys to partner;
        Receive partner's keys;
        if (my_rank < partner)
            Keep smaller keys;
        else
            Keep larger keys;
    }
}
```

Computing the Partner:

```c
if (phase % 2 == 0) {  // Even phase
    if (my_rank % 2 != 0)
        partner = my_rank - 1;
    else
        partner = my_rank + 1;
} else {               // Odd phase
    if (my_rank % 2 != 0)
        partner = my_rank + 1;
    else
        partner = my_rank - 1;
}

if (partner < 0 || partner >= comm_sz)
    partner = MPI_PROC_NULL;
```

MPI_PROC_NULL is a **special constant** indicating no communication should occur for that process in that phase.

3.7.3 Safety in MPI Programs

In a parallel MPI program, communication between processes is typically implemented using MPI_Send and MPI_Recv calls. For example:

```
MPI_Send(my_keys, n / comm_sz, MPI_INT, partner, 0, comm);
MPI_Recv(temp_keys, n / comm_sz, MPI_INT, partner, 0, comm, MPI_STATUS_IGNORE);
```

However, this approach can lead to problems such as **program hanging or crashing**, especially when processes engage in mutual communication simultaneously. The behavior of MPI_Send is defined by the MPI standard to be either buffered (non-blocking) or blocking. Buffered mode allows the program to proceed without waiting for a corresponding receive, while blocking mode waits until a matching MPI_Recv is initiated.

Most MPI implementations use a **threshold** mechanism: small messages are typically buffered, whereas larger messages invoke blocking behavior. When all processes perform MPI_Send simultaneously with large messages, none may reach the MPI_Recv stage, leading to a **deadlock**—a situation where every process is waiting indefinitely for another to proceed.

A program that implicitly assumes buffered communication is termed an **unsafe MPI program**. While such a program might execute correctly for small problem sizes, it is prone to failure when the message sizes increase, or system buffering behavior changes.

Detecting Unsafe Communication

To determine whether a program is safe, the MPI standard provides a synchronous version of MPI_Send called MPI_Ssend. The call to MPI_Ssend ensures that the send operation blocks until the matching receive is initiated by the receiving process.

```
MPI_Ssend(my_keys, n / comm_sz, MPI_INT, partner, 0, comm);
```

By substituting MPI_Send with MPI_Ssend, one can check the safety of the program. If the program executes correctly with MPI_Ssend, it is considered safe. Otherwise, if it hangs or crashes, the original version using MPI_Send is unsafe.

Causes of Unsafe Programs and Restructuring

Unsafe communication often arises when all processes attempt to **send first and receive later**. For instance, in a ring communication pattern, where process q sends to (q + 1) % comm_sz, simultaneous send operations can result in a deadlock:

```
MPI_Send(msg, size, MPI_INT, (my_rank + 1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank + comm_sz - 1) % comm_sz, 0, comm, MPI_STATUS_IGNORE);
```

To make such communication patterns safe, the communication must be **restructured**, so that some processes receive before sending. An example of a safe structure is:

if (my_rank % 2 == 0) {
MPI_Send(msg, size, MPI_INT, (my_rank + 1) % comm_sz, 0, comm);
MPI_Recv(new_msg, size, MPI_INT, (my_rank + comm_sz - 1) % comm_sz, 0, comm, MPI_STATUS_IGNORE);
} else {
MPI_Recv(new_msg, size, MPI_INT, (my_rank + comm_sz - 1) % comm_sz, 0, comm, MPI_STATUS_IGNORE);
MPI_Send(msg, size, MPI_INT, (my_rank + 1) % comm_sz, 0, comm);
}

*This pattern ensures that **half of the processes perform Recv before Send**, breaking the circular dependency that leads to deadlock. This method is effective for both **even and odd values of comm_sz**, as shown in **Figure 3.13**, which illustrates how a safe communication order can be maintained even in an odd-sized process ring.*

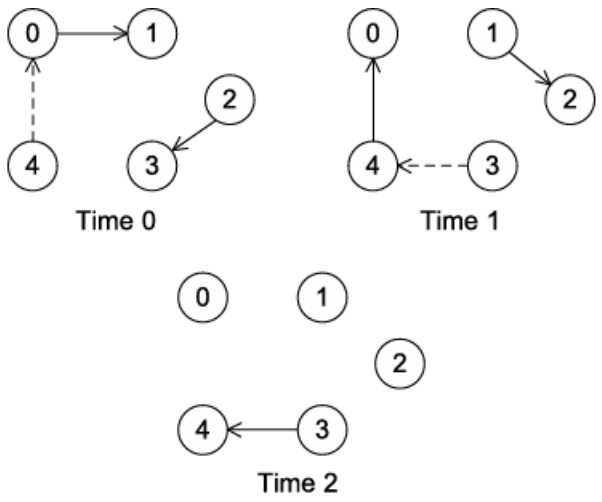

FIGURE 3.13

Safe communication with five processes.

Using MPI_Sendrecv for Safe Communication

An alternative and cleaner approach to avoid explicit coordination is to use the MPI_Sendrecv function, which performs a ***send and receive operation in a single call****, ensuring safe execution:*

```
MPI_Sendrecv(send_buf, send_size, send_type, dest, send_tag,
             recv_buf, recv_size, recv_type, source, recv_tag,
             comm, &status);
```

This method guarantees that no deadlock will occur because ***MPI internally schedules the communication*** *to ensure that one side sends while the other receives.*
Additionally, if the ***send and receive buffers are the same****, MPI provides:*

```
MPI_Sendrecv_replace(buf, buf_size, buf_type, dest, send_tag,
                     source, recv_tag, comm, &status);
```

This function sends the data in buf, replaces it with the received data, and is useful for ***in-place communication patterns****, such as in odd-even transposition sort.*

3.7.4 Final Details of Parallel Odd-Even Sort

The final implementation of the parallel odd-even transposition sort uses a safe communication strategy by combining the send and receive operations into a single call to MPI_Sendrecv. This helps avoid **deadlock** issues that arise from unsafe usage of MPI_Send and MPI_Recv, particularly when all processes attempt to send first. The improved communication structure ensures reliable exchange of keys between paired processes during each phase. The general algorithm remains the same: each process sorts its local keys first, and during each phase, communicates with a partner process, compares the combined keys, and retains either the smaller or larger half depending on its rank.

To make the process efficient, we avoid sorting the full list of combined keys. Instead, we **merge** the two sorted sublists—each of size n/p—to select the required subset. If a process is supposed to keep the smaller keys, it performs a forward merge and stops once n/p keys are obtained. If it must keep the larger keys, it merges backward from the end of the arrays. This **partial merge** is faster than full sorting. An additional **optimization** skips the need to copy arrays entirely by simply **swapping pointers**, which reduces memory operations (this pointer-swapping trick is elaborated in Exercise 3.28).

The **performance** of this final version is reflected in **Table 3.9**, which shows the run-times. It's important to note that for a single process, the sorting is done using **serial quicksort**, which is significantly faster than a serial odd-even sort. As expected, run-times improve with increasing number of processes, especially when n is large enough to offset communication costs. However, for smaller n, gains are modest.

This final version represents a practical and scalable parallel sorting strategy for **distributed memory systems**, leveraging **local sorting, partner communication, safe message passing, and efficient merging**

```
void Merge_low(
        int   my_keys[],      /* in/out     */
        int   recv_keys[],    /* in         */
        int   temp_keys[],    /* scratch    */
        int   local_n         /* = n/p, in  */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

Program 3.16: The `Merge_low` function in parallel odd-even transposition sort.

Table 3.9 Run-times of parallel odd-even sort (times are in milliseconds).

	Number of Keys (in thousands)				
Processes	**200**	**400**	**800**	**1600**	**3200**
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130