



Search Creators

**Visvesvaraya Technological
University(VTU)**

Prepared By

Hanumanthu

Dedicated To.

All Engineering Students

For More Updates

Website: <https://searchcreators.org/>
YouTube: <https://www.youtube.com/@searchcreators7348>
Instagram : <https://www.instagram.com/searchcreators/>
Telegram: <https://t.me/SearchCreators>
WhatsApp: +917348878215
Contact Number: +91 73488 78215

Scan Us | One Stop for All



COPYRIGHT & OWNERSHIP DECLARATION

Full Ownership:

All the content in this study material is the **exclusive intellectual property of Search Creators, Tumakuru.**

It is fully **created, compiled, and owned** by Search Creators.

Source:

The material is **prepared strictly based on VTU-prescribed textbooks** under the **2022 Scheme**, ensuring syllabus accuracy.

No AI / Website Use:

-  No ChatGPT, AI tools, or online generators were used.
-  No third-party websites or digital platforms contributed content.
-  100% manually prepared by the **Search Creators academic team.**

Usage Terms:

- ❖  Minimum cost for **personal student use only** (learning, revision, internal exam prep).
- ❖  **Not allowed** for resale, upload, sharing on public platforms, or reproduction in any form without written permission.
- ❖  Strictly prohibited from being published or claimed by any other party.

Prepared & Owned By:

Search Creators, Tumakuru

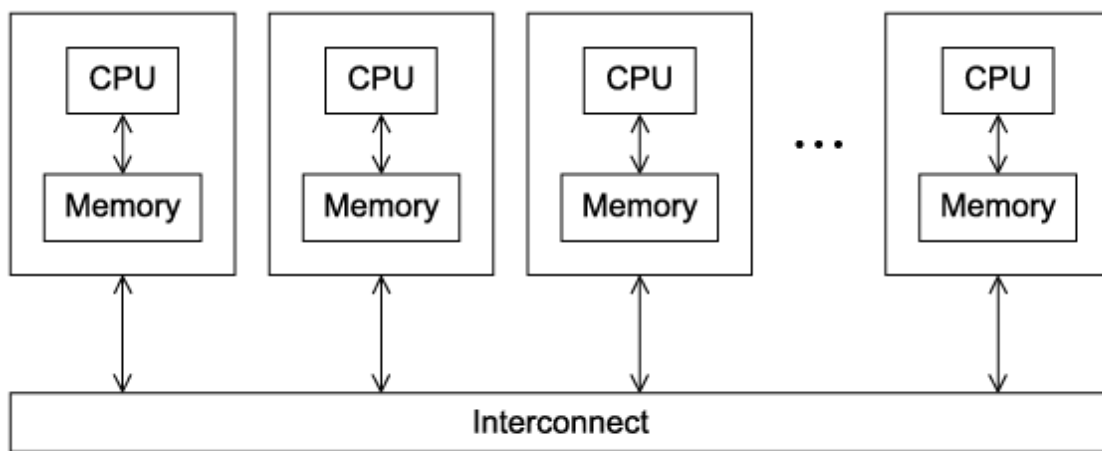
Final Year Project & Academic Support Center

 **Contact: 73488 78215**

 **Email: aryahanumanthu@gmail.com**

MODULE-3**Distributed memory programming with MPI****Distributed memory programming with MPI – MPI functions****Distributed Memory Programming with MPI: An Introduction to MPI Functions**

The world of parallel computing, particularly Multiple Instruction, Multiple Data (MIMD) systems, is broadly categorized into two architectural types from a programmer's perspective:

1. Distributed-Memory Systems (Figure 3.1):**FIGURE 3.1**

A distributed memory system.

- Consist of a collection of **core-memory pairs** connected by a **network**.
- The key characteristic is that the memory associated with a core is **directly accessible only to that specific core**. To access data residing in another core's memory, communication must happen explicitly, typically by sending messages.
- Programs running on these core-memory pairs are usually referred to as **processes**.

2. Shared-Memory Systems (Figure 3.2):

- Consist of a collection of cores that share access to a **globally accessible memory**.
- Each core can directly access any memory location, simplifying data sharing.

This chapter focuses on **distributed-memory systems** and how to program them using **message-passing**.

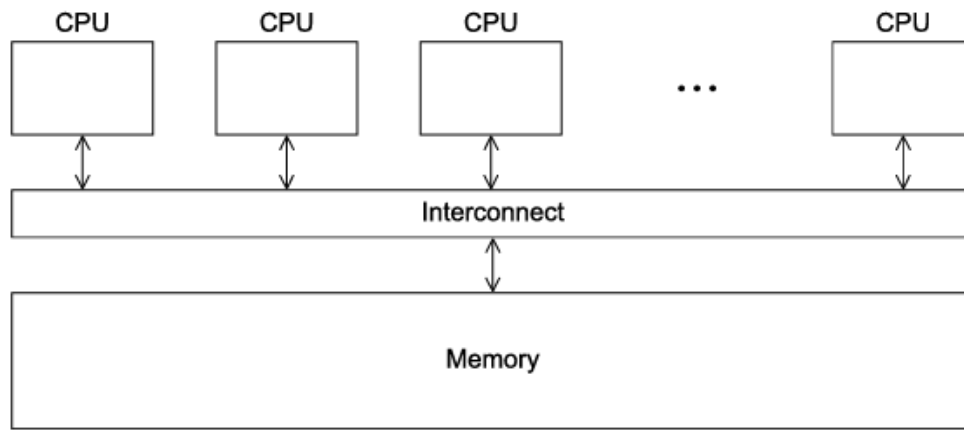


FIGURE 3.2

A shared memory system.

Message-Passing Interface (MPI)

MPI (Message-Passing Interface) is the standard implementation for message-passing programming that will be used. It's crucial to understand that MPI is **not a new programming language**. Instead, it defines a **library of functions** that can be invoked from existing programming languages like C and Fortran.

In message-passing programs, communication between processes happens through explicit function calls:

- One process calls a **send function** to transmit data.
- Another process calls a **receive function** to get that data.

Key MPI Functions and Concepts to be Explored:

The study of MPI will involve learning about various types of functions and fundamental issues in message-passing programming:

1. **Send and Receive Functions:** MPI offers a variety of send and receive functions, each with different blocking behaviors, buffering strategies, and performance characteristics. Understanding these is central to point-to-point communication.
2. **Collective Communications:** Beyond simple two-process communication, MPI provides "global" communication functions known as **collective communications**. These functions involve more than two processes participating in a coordinated communication operation (e.g., broadcasting data from one process to all others, or gathering data from all processes to one).
3. **Fundamental Issues in Message-Passing Programs:** As we delve into MPI functions, we will also address critical aspects of distributed-memory programming:
 - **Data Partitioning:** How to divide and distribute data across different processes' memories to minimize communication and maximize parallel efficiency.

- **I/O in Distributed-Memory Systems:** Specific considerations for handling input and output operations in an environment where each process has its own local memory and I/O streams are not inherently shared.
- **Parallel Program Performance:** Re-visiting and applying concepts of speedup, efficiency, and scalability in the context of MPI programs.

The trapezoidal rule in MPI

The trapezoidal rule approximates the area under the graph of a function $y=f(x)$ between two vertical lines $x=a$ and $x=b$.

Basic Idea:

1. Divide the interval $[a,b]$ into n equal subintervals.
2. Approximate the area above each subinterval using a trapezoid.
 - The base of the trapezoid is the subinterval.
 - The vertical sides are lines through the subinterval's endpoints, reaching up to the function's graph.
 - The fourth side is a secant line connecting the points on the graph.

Formulas:

- If the endpoints of a subinterval are x_i and x_{i+1} , the length of the subinterval (also called the step size) is $h = x_{i+1} - x_i$.
- The lengths of the vertical sides are $f(x_i)$ and $f(x_{i+1})$.
- The area of one trapezoid is:

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

- For n equal subintervals over $[a, b]$:

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

- The endpoints are $x_0 = a, x_1 = a + h, \dots, x_{n-1} = a + (n-1)h, x_n = b$.
- The sum of the areas of all trapezoids (the approximation of the total area) is:

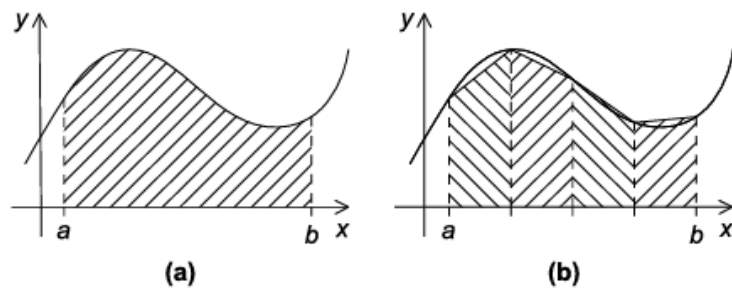
$$\text{Sum of trapezoid areas} = h \left[\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \dots + f(x_{n-1}) + \frac{f(x_n)}{2} \right]$$

Pseudocode for Serial Program:

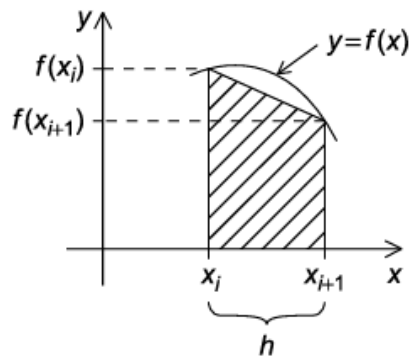
```

/* Input: a, b, n */
h = (b - a) / n;
approx = (f(a) + f(b)) / 2.0; // Half of f(a) and half of f(b)
for (i = 1; i <= n - 1; i++) {
    x_i = a + i * h;
    approx += f(x_i); // Add f(x_i) for intermediate points (counted once)
}
approx = h * approx; // Multiply by h at the end

```

**FIGURE 3.3**

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

**FIGURE 3.4**

One trapezoid.

Parallelizing the Trapezoidal Rule

"Parallelizing" refers to the process of converting a serial program into a parallel one. We follow a four-step design process:

1. Partition the problem solution into tasks:

- One type of task: calculating the area of a single trapezoid.
- Another type of task: summing these individual areas.

2. Identify communication channels between tasks:

- Each "calculate trapezoid area" task needs to send its result to the "sum areas" task.

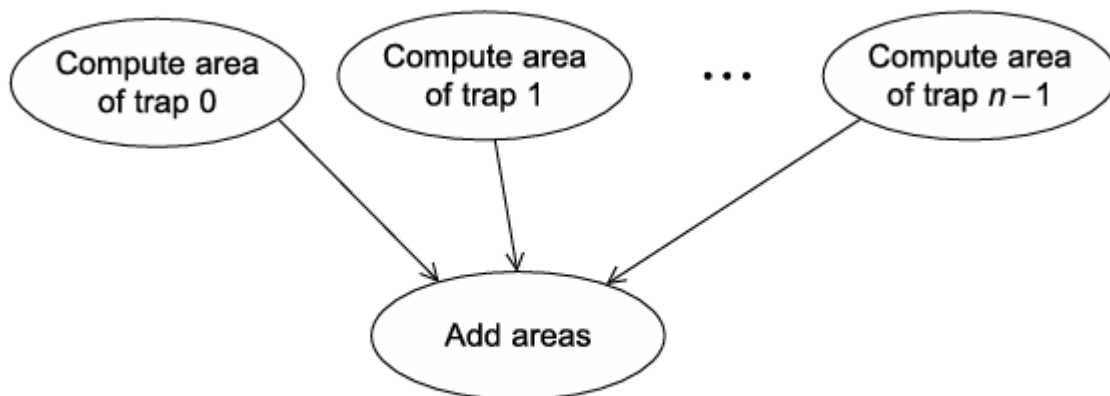
3. Aggregate tasks into composite tasks:

- Since we typically use many more trapezoids than available cores, we need to group the trapezoid calculations.
- A natural way is to divide the total interval $[a,b]$ into comm_sz (number of processes) subintervals.
- Each process then applies the trapezoidal rule to its assigned subinterval, calculating a local_integral .

4. Map composite tasks to cores:

- Each of the comm_sz processes calculates a local_integral for its assigned subinterval.
- One process (e.g., process 0) is designated to collect all the local_integral values and sum them to get the total_integral .

Simplifying Assumption: comm_sz (number of processes) evenly divides n (total number of trapezoids). This allows each process to handle $n/\text{comm_sz}$ trapezoids.

**FIGURE 3.5**

Tasks and communications for the trapezoidal rule.

Pseudocode for Parallel Program:

Get a, b, n ;

$h = (b - a) / n$;

$\text{local_n} = n / \text{comm_sz}$; // Number of trapezoids for this process

$\text{local_a} = a + \text{my_rank} * \text{local_n} * h$; // Starting point for this process

$\text{local_b} = \text{local_a} + \text{local_n} * h$; // Ending point for this process

```
local_integral = Trap(local_a, local_b, local_n, h); // Calculate local integral

if (my_rank != 0)
    Send local_integral to process 0;
else /* my_rank == 0 */
    total_integral = local_integral; // Process 0 starts with its own integral
    for (proc = 1; proc < comm_sz; proc++) {
        Receive local_integral from proc;
        total_integral += local_integral; // Accumulate integrals from other processes
    }
}

if (my_rank == 0)
    print result;
```

Variable Naming Convention (Context-Specific):

- **Local Variables:** Variables whose contents are significant only on the process using them (e.g., local_a, local_b, local_n).
- **Global Variables:** Variables whose contents are conceptually significant to all processes, even if they have copies (e.g., a, b, n). This differs from typical serial programming usage (function-local vs. globally accessible throughout the program), but the context makes the meaning clear.

MPI Program (Program 3.2): First Version of Trapezoidal Rule

The provided C code demonstrates the parallel trapezoidal rule using MPI, with a, b, and n hardcoded for simplicity.

The Trap function (Program 3.3, not shown but described as a serial trapezoidal rule implementation) calculates the integral for a given subinterval.

```
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); // Initializes the MPI environment
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Gets the rank (ID) of the calling process
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // Gets the total number of processes

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

// Determine the local subinterval for this process
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); // Calculate local integral
if (my_rank != 0) {
    // All processes except rank 0 send their local integral to rank 0
    MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
} else {
    // Process rank 0 initializes total_int with its own local integral
    total_int = local_int;
    // Process rank 0 receives local integrals from all other processes
    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; // Adds received integral to the total    }
    }

if (my_rank == 0) {
    // Only process rank 0 prints the final result
    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(); // Terminates the MPI environment
return 0;
} /* main */
```

Dealing with I/O

Dealing with I/O in MPI Programs

The previous version of the parallel trapezoidal rule program had a significant limitation: its input parameters (a, b, and n) were hardcoded. This section addresses how to handle user input and output in MPI programs, especially considering the complexities introduced by parallelism.

Output

In both the "greetings" program and the trapezoidal rule, we've implicitly assumed that **process 0 can write to stdout** (using printf) as expected.

MPI Standard and Implementations:

While the MPI standard doesn't strictly specify which processes have access to I/O devices, virtually all MPI implementations grant **all processes within MPI_COMM_WORLD full access to stdout and stderr**.

This means printf() and fprintf(stderr, ...) calls can be made by any process.

Nondeterminism:

The crucial point is that **most MPI implementations do not provide automatic scheduling of access** to these devices. This leads to **nondeterminism**:

- If multiple processes attempt to write to stdout concurrently, the **order of their output lines will be unpredictable**.
- Even worse, the output of one process can be **interrupted (interleaved)** by the output of another process, making the output difficult to parse or read.

Example (Program 3.4):

A simple MPI program where each process prints a message demonstrates this nondeterminism.

```
#include <stdio.h>
#include <mpi.h> // Include MPI header

int main(void) {
    int my_rank, comm_sz;
```

```
MPI_Init(NULL, NULL); // Initialize MPI

MPI_Comm_size(MPI_COMM_WORLD, &comm_sz); // Get total number of processes
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Get rank of current process


// Output statement by each process
printf("Proc %d of %d > Does anyone have a toothpick?\n", my_rank, comm_sz);


MPI_Finalize(); // Finalize MPI
return 0;
} // main
```

When run with 5 processes, the output might be ordered (e.g., Proc 0, Proc 1, ...). However, with 6 processes, the order frequently becomes unpredictable (e.g., Proc 0, 1, 2, 5, 3, 4), illustrating the competition for shared stdout.

This competition for shared resources like stdout leads to nondeterminism, meaning the output varies from run to run.

Handling Nondeterministic Output:

If a predictable or ordered output is desired, it's the programmer's responsibility to manage it. A common approach is to:

- Have all processes (other than process 0) **send their output data to process 0**.
 - Have **process 0** then print all the collected output in a desired order (e.g., in process rank order).
- This was the strategy used in the "greetings" program.

Input

Unlike output, most MPI implementations are more restrictive regarding standard input (stdin):

Process 0 Only:

Most MPI implementations **only allow process 0 in MPI_COMM_WORLD to access stdin**.

Rationale:

This design choice makes sense, as it avoids ambiguity about which process should receive which parts of the input data (e.g., should lines be distributed round-robin, or characters?). Centralizing input to process 0 simplifies this.

Reading Input in MPI Programs:

To use functions like `scanf` in MPI programs, you must branch on the process rank:

1. **Process 0** reads the necessary input data.
2. **Process 0** then sends this data to all other processes.

Get_input Function Example (Program 3.5):

This function demonstrates how process 0 reads input and broadcasts it to others.

```
void Get_input(
    int my_rank /* in */,
    int comm_sz /* in */,
    double* a_p /* out */, // Pointers to variables to store input
    double* b_p /* out */,
    int* n_p /* out */) {

    int dest;

    if (my_rank == 0) { // Only process 0 interacts with the user
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p); // Read input for a, b, n
        for (dest = 1; dest < comm_sz; dest++) {
            // Send a, b, and n to all other processes
            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 */
        // Other processes receive a, b, and n from process 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 */
```

This function uses the same basic communication pattern as the "greetings" program, but with process 0 sending data and other processes receiving it.

Integrating Get_input:

To use this function, a call to Get_input needs to be inserted into the main function of the MPI program, specifically after my_rank and comm_sz have been initialized.

```
...  
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); // Call to get input  
h = (b - a) / n; // Calculate h after a, b, n are set  
...
```

Collective communication

Collective Communication in MPI

In distributed-memory programming with MPI, **collective communication** refers to operations that involve **all the processes within a communicator**.

This is in contrast to **point-to-point communication** (like MPI_Send and MPI_Recv), which involves only two processes.

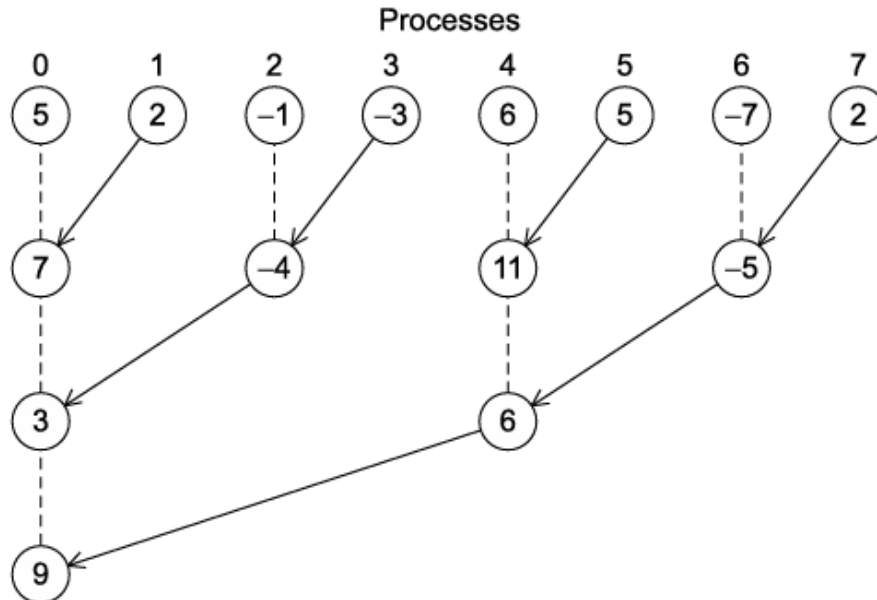
Collective communication functions are optimized by MPI implementations to leverage the underlying hardware and network topology, relieving the programmer of the burden of explicit, complex communication patterns.

Improving the Trapezoidal Rule: Beyond Point-to-Point

Our initial parallel trapezoidal rule program used a simple point-to-point communication pattern for the global sum: all processes with rank greater than 0 send their local_integral to process 0, and process 0 sums them up. This makes process 0 a bottleneck, doing comm_sz - 1 receives and additions, while others simply send and then become idle.

Tree-Structured Communication (Manual Approach)

To improve this, we can design more efficient communication patterns. A common approach is a **tree-structured global sum** (like a binary reduction tree, as shown in Figure 3.6).

**FIGURE 3.6**

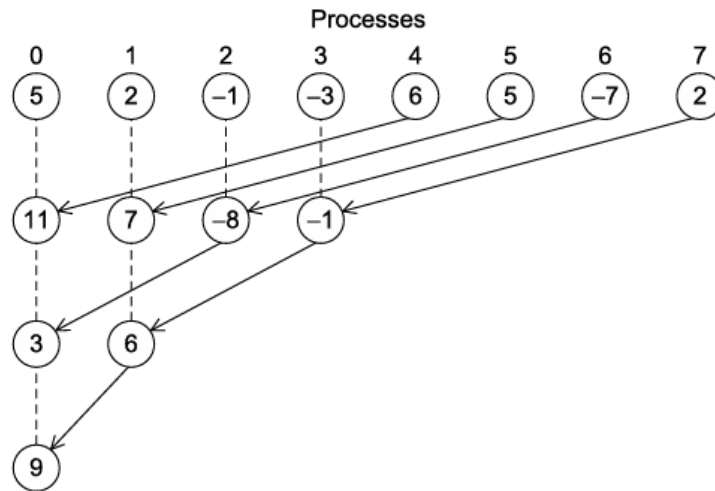
A tree-structured global sum.

How it works (example with 8 processes):

1. **Phase 1:** Processes 1, 3, 5, 7 send their values to processes 0, 2, 4, 6, respectively. The receivers add the values to their own. (4 sends, 4 adds, done concurrently).
 2. **Phase 2:** Processes 2 and 6 send their new sums to processes 0 and 4, respectively. The receivers add the values. (2 sends, 2 adds, done concurrently).
 3. **Phase 3:** Process 4 sends its sum to process 0. Process 0 adds the value. (1 send, 1 add).
- **Benefits:** This dramatically reduces the work on process 0. In an 8-process example, process 0 performs only 3 receives and 3 additions (compared to 7 in the original scheme). More importantly, many of these operations occur **concurrently** across different processes.
 - **Scalability:** For comm_sz processes, the original scheme requires $\text{comm_sz} - 1$ receives/adds on process 0. A tree-structured sum requires only $\log_2(\text{textcomm_sz})$ receives/adds on process 0. This is a significant improvement, especially for large comm_sz (e.g., reducing 1023 operations to 10 for 1024 processes).

The Challenge of Manual Implementation:

While efficient, implementing such tree-structured communication manually is complex. There are many possible tree structures (e.g., different pairings as in Figure 3.7), and the optimal one can depend on the number of processes and the underlying hardware. Coding and testing each alternative would be prohibitive for application developers.

**FIGURE 3.7**

An alternative tree-structured global sum.

MPI_Reduce: The General Global Reduction

Recognizing the complexity of implementing optimal global sum (and similar) operations, MPI provides **collective communication functions** that handle these patterns internally.

This shifts the burden of optimization to the MPI library implementers, who can leverage their knowledge of hardware and system software.

MPI_Reduce is a generalized global reduction function:

```
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 */
);
```

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

- **Purpose:** MPI_Reduce takes values from all processes in comm, applies a specified operator (like sum, max, min, product), and stores the single resulting value on dest_process.
- **operator (MPI_Op):** This argument is key to generalization. MPI defines several standard reduction operators (see Table 3.2), including MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD, MPI_LOR (logical OR), MPI_LAND (logical AND), etc. Users can also define custom operators.
- **Example (Trapezoidal Rule):** We can replace the MPI_Send/MPI_Recv loop in Program 3.2 with a single MPI_Reduce call:

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

Every process calls this function. Each supplies its local_int. MPI_Reduce internally performs the sum and deposits the final total_int only into output_data_p of process 0.

- **Arrays:** MPI_Reduce can also operate on arrays by setting count to the number of elements.

Collective vs. Point-to-Point Communications (Important Differences)

Participation:

All processes in the specified MPI_Comm **must call the same collective function**. Mismatched calls (e.g., MPI_Reduce on one process and MPI_Recv on another) are erroneous and will likely cause the program to hang or crash.

Compatible Arguments:

Arguments passed to a collective communication by all processes must be "compatible." For MPI_Reduce, if one process specifies dest_process as 0 and another as 1, it's an error.

Output Buffer Usage:

The `output_data_p` argument is only actually *written to* on the `dest_process`. However, *all* processes must still provide a valid argument for `output_data_p`, even if it's just `NULL` for those not receiving the result.

Matching:

- **Point-to-point:** Matched by message **tags** and **communicators**.
- **Collective:** Do **not use tags**. They are matched solely by the **communicator** and the **order in which they are called**. This means the *n*th collective call in one process must match the *n*th collective call in all other participating processes. (See Table 3.3 example: `b` receives the sum of first `MPI_Reduce` calls, `d` the sum of second `MPI_Reduce` calls, regardless of variable names).

Aliasing Restriction:

It is illegal in MPI to use the same buffer for both `input_data_p` and `output_data_p` in `MPI_Reduce` (e.g., `MPI_Reduce(&x, &x, ...)`).

This is because MPI prohibits aliasing (where two arguments refer to the same memory location) for input/output arguments, partly due to Fortran's strict aliasing rules.

This can lead to unpredictable results (incorrect output, crash, or even seemingly correct but non-portable behavior).

MPI_Allreduce: Result for All Processes

Often, all processes need the result of a global reduction, not just one. While one could follow `MPI_Reduce` with an `MPI_Bcast`, MPI provides a dedicated collective for this: `MPI_Allreduce`.

```
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 */  
);
```

- **Difference from MPI_Reduce:** The argument list is identical, but there is **no dest_process** argument, because the result is stored on the output_data_p of *all* processes in the comm.
- **Benefit:** MPI_Allreduce implementations are typically optimized to distribute the result efficiently (e.g., using a reverse tree or butterfly pattern, as shown in Figures 3.8 and 3.9), avoiding the need for two separate collective calls or complex manual distribution.

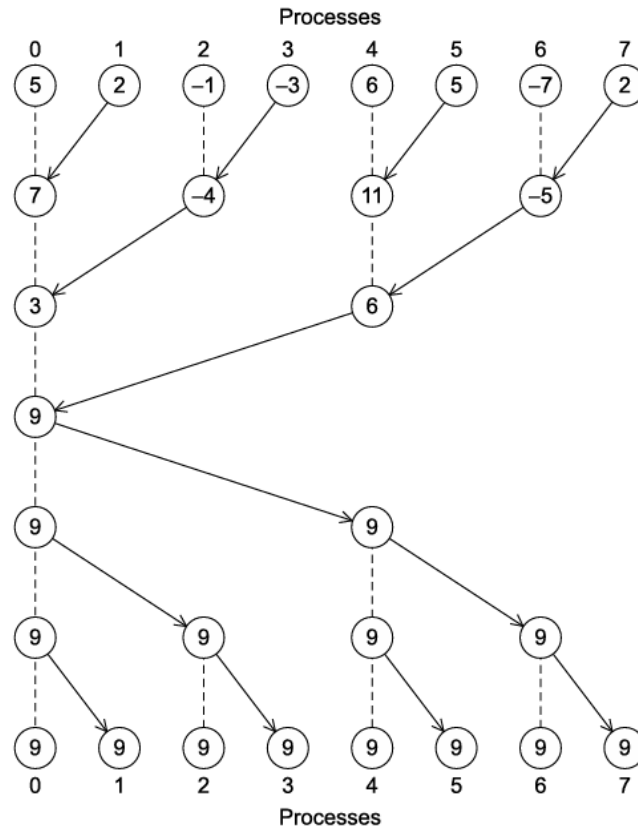


FIGURE 3.8

A global sum followed by distribution of the result.

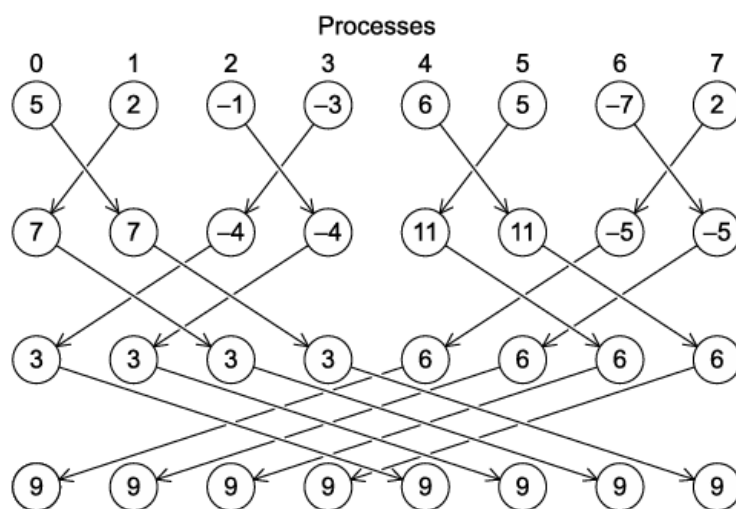


FIGURE 3.9

A butterfly-structured global sum.

Broadcast: Distributing Data from One to All

Just as MPI_Reduce aggregates data, MPI_Bcast (broadcast) distributes data from a single source process to all other processes in a communicator.

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

Purpose: The process with rank `source_proc` sends the contents of `data_p` to all other processes in `comm`.

data_p Argument: On `source_proc`, `data_p` is an **input** argument (it contains the data to be sent). On all *other* processes, `data_p` is an **output** argument (it will receive the data). This "in/out" label for collective arguments signifies this dual role.

Example (Get_input modification - Program 3.6): Instead of MPI_Send and MPI_Recv loops, MPI_Bcast can simplify the Get_input function for the trapezoidal rule: Process 0 reads `a`, `b`, and `n`, then calls MPI_Bcast three times (once for each variable) to distribute them to all other processes.

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

    if (my_rank == 0) {
        printf("Enter a, b, and n\n");
        scanf("%lf %lf %d", a_p, b_p, n_p);
    }
    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
} /* Get_input */
```

Program 3.6: A version of Get_input that uses MPI_Bcast.

Data Distributions for Vector Operations

When parallelizing operations on large data structures like vectors, how the data is distributed among processes is crucial.

Vector Addition Example ($z = x + y$):

Each process needs its portion of x and y to compute its portion of z . The individual additions of components are the tasks.

Types of Data Partitioning (for an n -component vector among comm_sz processes, assuming n is divisible by comm_sz , $\text{local_n} = n / \text{comm_sz}$):

Table 3.4 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

- Block Partition (Table 3.4, left):** Each process gets a contiguous block of local_n components.
 - Process 0 gets components 0 to $\text{local_n}-1$.
 - Process 1 gets components local_n to $2*\text{local_n}-1$, and so on.
- Cyclic Partition (Table 3.4, middle):** Components are assigned in a round-robin fashion.
 - Process 0 gets components 0, comm_sz , $2*\text{comm_sz}$, etc.
 - Process 1 gets components 1, $\text{comm_sz}+1$, $2*\text{comm_sz}+1$, etc.
- Block-Cyclic Partition (Table 3.4, right):** A hybrid where blocks of b components are assigned in a cyclic fashion. (e.g., block 0 to P0, block 1 to P1, block 2 to P2, block 3 to P0, etc.).

Parallel Vector Addition (Program 3.8): Once data is partitioned, each process simply performs the addition on its local local_n components. The code for the addition loop is essentially identical to the serial version, operating only on the process's assigned data.

Scatter: Distributing Blocks from One to All

If process 0 has an entire vector and needs to distribute *different contiguous blocks* of that vector to different processes (e.g., for a block distribution of x and y in vector addition), `MPI_Scatter` is the ideal collective communication.

```
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 */
);
```

```
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 */
```

Program 3.8: A parallel implementation of vector addition.

Purpose: MPI_Scatter divides the data referenced by send_buf_p on src_proc into comm_sz pieces. The first piece goes to process 0, the second to process 1, and so on. Each process (including src_proc) receives its designated piece into recv_buf_p.

send_count: This is the *number of elements sent to EACH process*, not the total number of elements in send_buf_p.

Example (Read_vector function - Program 3.9): Process 0 reads the full vector into a buffer. Then, MPI_Scatter distributes local_n (e.g., n/comm_sz) components to each process, and each process stores them in its local_vector (which only needs local_n space).

Restriction: This simple MPI_Scatter usage is specifically designed for **block distributions** where n is evenly divisible by comm_sz. More complex distributions (cyclic, block-cyclic) or non-divisible n require more advanced techniques.

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

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

Gather: Collecting Blocks from All to One

To collect data distributed across processes onto a single process (e.g., for printing a distributed vector), MPI_Gather is used.

```
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 */  
);
```

Purpose: The data from send_buf_p on each process is concatenated and stored in recv_buf_p on dest_proc. The data from process 0 goes first, then process 1, and so on.

recv_count: This is the *number of data items received from EACH process*, not the total number of items received by dest_proc. So if each process sends local_n items, recv_count will be local_n. The total buffer size on dest_proc for recv_buf_p must be comm_sz * recv_count.

Example (Print_vector function - Program 3.10): Each process provides its local_b (local components of the vector). Process 0 allocates a large enough buffer (b) to hold the entire vector. MPI_Gather collects all local_bs into b on process 0, which then prints the full vector.

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

Program 3.10: A function for printing a distributed vector.

Restriction: Similar to MPI_Scatter, this simple MPI_Gather usage also primarily works correctly with **block distributions** where all blocks have the same size.

Allgather: Collecting Blocks to All Processes

For situations where **all processes** need a complete copy of a concatenated dataset (e.g., a full vector assembled from distributed parts), MPI_Allgather is used.

This is particularly useful in iterative algorithms where a global data structure (like x in matrix-vector multiplication) needs to be available to all processes in each iteration.

```
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 */  
);
```

Purpose: This function concatenates the contents of each process's `send_buf_p` and then **stores this full, concatenated result in each process's `recv_buf_p`.**

Difference from MPI_Gather: Instead of collecting to a single `dest_proc`, `MPI_Allgather` makes the collected data available to *all* processes.

Matrix-Vector Multiplication Example ($y = Ax$):

- **Serial Logic:** $y[i] = \text{dot_product}(A[i_row], x)$. Each $y[i]$ requires the full vector x .
- **Parallel Strategy:** Distribute matrix A by rows (block distribution is easiest in MPI). Each process gets a `local_m` number of rows of A and calculates its corresponding `local_m` components of y .
- **The x problem:** For each process to compute its $y[i]$, it needs the *entire* vector x . If x itself is also distributed, how do all processes get the full x ?
- **Solution with MPI_Allgather:** If x is block-distributed, each process has a `local_x` segment. Before the inner loop that computes $y[i]$, an `MPI_Allgather` call can be made. Each process sends its `local_x` segment, and `MPI_Allgather` collects all these segments and distributes the full x vector to *every* process. This avoids potentially expensive `MPI_Gather` followed by `MPI_Bcast` pattern and leverages optimized internal algorithms.

Program 3.12: This program implements parallel matrix-vector multiplication using `MPI_Allgather` to ensure all processes have access to the full x vector.

These collective communication functions are fundamental tools in MPI, providing efficient and optimized ways to handle common data movement and aggregation patterns in parallel programs, significantly simplifying complex communication logic for application developers.

MPI-derived datatypes

MPI Derived Datatypes: Optimizing Communication

In distributed-memory systems, **communication is significantly more expensive than local computation**. Sending data between nodes takes much longer than performing calculations on data already in local memory.

A critical observation is that **sending a fixed amount of data in multiple small messages is far more costly than sending the same amount of data in a single, larger message**.

This is due to communication overheads (e.g., protocol negotiation, packet headers) associated with each message. For instance, sending 1000 doubles one by one can be 50 to 100 times slower than sending all 1000 doubles in one MPI_Send call.

Therefore, a key goal in MPI programming is to **reduce the total number of messages sent**.

MPI provides three primary ways to consolidate data for communication, thereby reducing message count:

1. **count argument:** We've already seen this. It allows grouping contiguous array elements into a single message (e.g., MPI_Send(x, 1000, MPI_DOUBLE, ...)).
2. **Derived Datatypes:** This allows grouping non-contiguous or mixed-type data into a single logical message.
3. **MPI_Pack/MPI_Unpack:** A more manual way to buffer disparate data into a contiguous buffer before sending.

This section focuses on **derived datatypes**.

What are MPI Derived Datatypes?

An MPI derived datatype allows you to describe a **collection of data items in memory by specifying their types and their relative locations (displacements)**.

The idea is that instead of sending individual pieces of data one by one, you define a single "blueprint" for a complex data structure.

When you use this derived datatype in a communication function (like MPI_Send or MPI_Bcast):

- **Sender:** The MPI implementation uses the blueprint to gather the specified data items from their scattered locations in memory into a contiguous buffer (packing) before transmitting them as a single message.
- **Receiver:** The MPI implementation uses the same blueprint to distribute the received data items into their correct, possibly scattered, destinations in the receiver's memory (unpacking).

Example: In the trapezoidal rule program, we had to call MPI_Bcast three times to send a (double), b (double), and n (int). With a derived datatype, we can define a single type representing "two doubles and one int" and use just one MPI_Bcast call.

Building a Derived Datatype with MPI_Type_create_struct

A derived datatype is formally a sequence of basic MPI datatypes, each with a corresponding **displacement** (offset from a base address).

Let's assume a, b, and n are stored at addresses 24, 40, and 48 bytes respectively on process 0.

- a (MPI_DOUBLE) at displacement 0 (relative to itself).
- b (MPI_DOUBLE) at displacement 16 bytes (40 - 24).
- n (MPI_INT) at displacement 24 bytes (48 - 24).

This could be represented as: {(MPI_DOUBLE, 0), (MPI_DOUBLE, 16), (MPI_INT, 24)}.

The function to build such a structured derived datatype is MPI_Type_create_struct:

```
int MPI_Type_create_struct(
    int          count,          /* in */ // Number of elements in the datatype
    int          array_of_blocklengths[], /* in */ // Array: block length for each element
    MPI_Aint     array_of_displacements[], /* in */ // Array: displacement (bytes) for each element
    MPI_Datatype array_of_types[], /* in */ // Array: MPI datatype for each element
    MPI_Datatype* new_type_p      /* out */ // Pointer to store the new derived datatype
);
```

Arguments Explained (for our a, b, n example):

1. **count:** The number of distinct elements in the datatype. For a, b, n, count would be 3.
2. **array_of_blocklengths:** An array of integers, where array_of_blocklengths[i] specifies the number of occurrences of the i-th type. If an element is a single scalar (not an array), its block length is 1.

- For a, b, n: `int array_of_blocklengths[3] = {1, 1, 1};`

3. **array_of_displacements:** An array of `MPI_Aint` (a special integer type large enough to hold memory addresses/offsets), specifying the offset in **bytes** of each element from the *start address of the first element* in the structure.

- To get these addresses, use `MPI_Get_address`:

```
int MPI_Get_address(
    void* location_p, /* in */ // Pointer to the variable
    MPI_Aint* address_p /* out */ // Pointer to store its address
);
```

- Example code to get displacements:

```
MPI_Aint a_addr, b_addr, n_addr;
MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0; // a is at the base
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr; // Displacement of b from a
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr; // Displacement of n from a
```

4. **array_of_types:** An array of `MPI_Datatypes`, specifying the MPI type of each element.

- For a, b, n: `MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE, MPI_INT};`

Putting it Together (Building the Type):

```
MPI_Datatype input_mpi_t;
// ... (initialize array_of_blocklengths, array_of_displacements, array_of_types)
MPI_Type_create_struct(3, array_of_blocklengths,
    array_of_displacements, array_of_types,
    &input_mpi_t);
```

Committing and Using the Derived Datatype

After creating a derived datatype, it **must be committed** before it can be used in any communication function:

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

- **Purpose of MPI_Type_commit:** This allows the MPI implementation to optimize its internal representation of the new datatype, preparing it for efficient use in communication.

Using the Derived Datatype in Communication: Once committed, the derived datatype can be used just like a basic MPI datatype in communication functions. For our example:

```
// Process 0 sends the values, others receive
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

Notice that the count argument here is 1, as we are sending "one instance" of our newly defined input_mpi_t datatype, which itself encapsulates three individual data items.

Freeing the Derived Datatype

Derived datatypes might involve internal memory allocation by the MPI library. To release these resources when the type is no longer needed:

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

This should be called when the derived datatype is no longer required, typically before MPI_Finalize().

The text mentions using these steps to define a Build_mpi_type function, which is then called by an updated Get_input function (Program 3.13, not shown in the snippet).

This illustrates how derived datatypes can significantly simplify and optimize I/O and communication patterns in MPI programs by reducing the number of individual messages.

Performance evaluation of MPI programs

Evaluating the performance of parallel programs, especially those written with MPI, is crucial to determine if they are faster than their serial counterparts and how efficiently they utilize parallel resources.

```

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.

Taking Timings

When measuring parallel program performance, we are typically interested in the **wall clock time** taken by the parallelized portion of the code, not the total program execution time (which includes I/O, initialization, etc.).

MPI_Wtime(): MPI provides double MPI_Wtime(void); which returns the number of seconds that have elapsed since an arbitrary point in the past. It's used to measure wall clock time.

```

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);
```

GET_TIME (for Serial Code): For serial code (where MPI libraries might not be linked), a POSIX function like `gettimeofday` or a macro like `GET_TIME` (from `timer.h`) can be used to get microsecond-resolution wall clock time.

```
#include "timer.h"

// ...

double start, finish;

// ...

GET_TIME(start);

/* Code to be timed */

GET_TIME(finish);

printf("Elapsed time = %e seconds\n", finish - start);
```

`GET_TIME` is a macro, so it operates directly on its double argument. Remember to tell the compiler where `timer.h` is if it's not in a standard include path.

Wall Clock Time vs. CPU Time:

- **Wall Clock Time (MPI_Wtime, GET_TIME):** Measures the total elapsed time, including time spent waiting (idle time), which is crucial for parallel programs. For instance, `MPI_Recv` might spend significant time waiting for a message.
- **CPU Time (e.g., C's clock function):** Measures only the time the CPU spends executing user code, library functions, and OS code, excluding idle time. This is less useful for parallel performance analysis as it doesn't capture communication overheads or synchronization waits.

Reporting a Single Time for Parallel Programs:

Since multiple processes run concurrently, each will report its own `local_elapsed` time. To get a meaningful single time for the parallel execution, we want the time it took for the *slowest* process to finish the timed block.

1. **Synchronization (MPI_Barrier):** Use `MPI_Barrier(comm)`; *before* starting the timer on all processes. This ensures all processes have reached this point before any of them proceeds, minimizing the impact of varying start times.
2. **Global Maximum Reduction (MPI_Reduce with MPI_MAX):** Each process calculates its `local_elapsed` time. Then, `MPI_Reduce` is used with the `MPI_MAX` operator to find the maximum `local_elapsed` among all processes and store it on process 0.

```

double local_start, local_finish, local_elapsed, elapsed;
// ...
MPI_Barrier(comm); // Ensure all processes are ready
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); // Find max
elapsed
if (my_rank == 0)
    printf("Elapsed time = %e seconds\n", elapsed); // Only process 0 prints

```

Variability and Reporting:

Timings of parallel programs can vary significantly between runs due to unpredictable interactions with the operating system and other processes.

It's common practice to report the **minimum run-time** over several runs, as this usually represents the performance on a "quiet" system, unaffected by external interference.

Hybrid Systems:

When running MPI on multicore nodes, typically only one MPI process is run per node to reduce contention for the interconnect, which can lead to better and more consistent run-times.

Results (Matrix-Vector Multiplication Example)

Table 3.5 shows example run-times for the matrix-vector multiplication.

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

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

Impact of Problem Size (n):

- As n (order of matrix) increases (for fixed comm_sz), run-times generally increase.
- For small comm_sz, doubling n roughly quadruples the run-time, consistent with an $O(n^2)$ serial complexity ($2n^2$ floating point operations for an n times n matrix).
- For large comm_sz, this relationship starts to break down.

Impact of Number of Processes (comm_sz):

- As comm_sz increases (for fixed n), run-times usually decrease.
- For large n, doubling comm_sz roughly halves the run-time (indicating good parallel scaling).
- For small n, increasing comm_sz offers little benefit, and can even slightly worsen performance (e.g., $n=1024$, comm_sz=8 vs. 16 had no change). This suggests that for small problems, the parallel overhead starts to dominate.

The Parallel Run-time Formula:

The observed behavior can be explained by the formula:

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

Where:

- $T_{\text{serial}}(n)$: Serial run-time (for problem size n). For matrix-vector multiplication, $T_{\text{serial}}(n) \approx c \cdot n^2$.

p : Number of processes (comm_sz).

$T_{\text{overhead}}(n, p)$: Overhead due to parallelization (communication, synchronization, load imbalance). In MPI, this often comes from communication (e.g., MPI_Allgather in this case).

Small p, Large n:

The $\frac{T_{\text{serial}}(n)}{p}$ term dominates. Doubling p roughly halves T_{parallel} (e.g., $T_{\text{parallel}}(8192, 2) \approx 2.0 \times T_{\text{parallel}}(8192, 4)$). This indicates good parallel efficiency.

Large p , Small n :

The T_{overhead} term dominates. Increasing p further yields diminishing returns or even performance degradation (e.g., $T_{\text{parallel}}(1024,8)$ vs.

$T_{\text{parallel}}(1024,16)$). The communication overhead outweighs the benefit of distributing computation.

Speedup and Efficiency

These are standard metrics to quantify parallel performance relative to a serial baseline.

Speedup ($S(n, p)$): The ratio of serial run-time to parallel run-time.

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

- Ideal speedup is p (linear speedup).
- Table 3.6 shows the speedups. For small p and large n , speedups are nearly linear. For large p and small n , speedups are much less than p (e.g., only 2.4 for $n=1024$, $p=16$).

Parallel Efficiency ($E(n, p)$): "Per-process" speedup, indicating how well each process is being utilized.

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

- Ideal efficiency is 1.0 (corresponds to linear speedup).
- Table 3.7 shows efficiencies. Similar to speedup, efficiencies are near linear for small p and large n , but much lower for large p and small n .

Table 3.6 Speedups of parallel matrix-vector multiplication.

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

Table 3.7 Efficiencies of parallel matrix-vector multiplication.

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

Scalability

Scalability describes how a parallel program's performance changes as the problem size and/or the number of processes increase.

General Definition (Rough): A program is scalable if the problem size can be increased at a rate such that the efficiency doesn't decrease as the number of processes increases.

Types of Scalability:

- **Strongly Scalable:** A program is strongly scalable if it can **maintain constant efficiency without increasing the problem size** as the number of processes increases.
- (Program A example: efficiency is 0.75 regardless of n or p). This implies that the overhead T_{overhead} is negligible or does not grow significantly with p.
- **Weakly Scalable:** A program is weakly scalable if it can **maintain constant efficiency if the problem size increases at the same rate as the number of processes**. (Program B example: if n and p both double, efficiency remains constant).
- This means the work per process remains constant, and T_{overhead} per process also remains relatively constant.

Matrix-Vector Multiplication Scalability:

- The matrix-vector multiplication program is **not strongly scalable** (efficiency generally decreases as p increases for a fixed n).
- However, it appears to be **weakly scalable**. Looking at Table 3.7, when p and n are both doubled (e.g., from (n=2048, p=4) to (n=4096, p=8) or (n=4096, p=8) to (n=8192, p=16)), the efficiency actually tends to increase or stay relatively constant, especially for larger p values.
- This suggests that as more resources become available, increasing the problem size proportionally allows the program to maintain its efficiency.

A parallel sorting algorithm

A Parallel Sorting Algorithm in a Distributed Memory Environment

In a distributed memory environment, a **parallel sorting algorithm** aims to sort a collection of keys that are initially distributed across multiple processes, and eventually, the sorted keys are also distributed among the processes according to a specific order.

- **Input:** We start with a total of n keys, with each of $p = \text{textcomm_sz}$ processes initially holding n/p keys (assuming n is evenly divisible by p). There are no restrictions on which keys are on which process at the start.
- **Output:** When the algorithm terminates:
 1. The n/p keys assigned to each individual process must be sorted (e.g., in increasing order).
 2. If $0 \leq q < r < p$, then every key assigned to process q must be less than or equal to every key assigned to process r . This means that if you conceptually concatenate the sorted lists from process 0, then process 1, and so on, the entire global list of keys would be sorted.

For simplicity, the keys are assumed to be ordinary integers.

Some Simple Serial Sorting Algorithms

Before diving into parallel sorting, it's useful to review some serial algorithms.

Bubble Sort (Program 3.14): This algorithm sorts an array a of n elements. It iteratively compares adjacent elements and swaps them if they are out of order. In each pass, the largest unsorted element "bubbles" to its correct position at the end of the unsorted portion of the list.

```
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
```

Parallelism Deficiency: Bubble sort is inherently sequential due to dependencies between "compare-swaps." The order of comparisons is critical for correctness. For example, if $a[i-1]=9$, $a[i]=5$, $a[i+1]=7$:

- Correct: (9,5) swap $\rightarrow$ (5,9,7). Then (9,7) swap $\rightarrow$ (5,7,9).
- Incorrect if out of order: (5,7) no swap $\rightarrow$ (9,5,7). Then (9,5) swap $\rightarrow$ (5,9,7). The 7 is now incorrectly placed. This strict ordering makes it difficult to parallelize effectively.

Odd-Even Transposition Sort (Program 3.15): This is a variant of bubble sort that allows for more parallelism by "decoupling" compare-swaps. It proceeds in a sequence of phases:

- **Even Phases:** Compare-swaps are performed on pairs $(a[0], a[1])$, $(a[2], a[3])$, $(a[4], a[5])$, ...
- **Odd Phases:** Compare-swaps are performed on pairs $(a[1], a[2])$, $(a[3], a[4])$, $(a[5], a[6])$, ...

Example: Start: 5,9,4,3

- Even phase: (5,9) and (4,3) are compare-swapped implies 5,9,3,4 (no change to 5,9; 4,3 becomes 3,4).
- Odd phase: (9,3) is compare-swapped implies 5,3,9,4.
- Even phase: (5,3) and (9,4) are compare-swapped implies 3,5,4,9.
- Odd phase: (5,4) is compare-swapped implies 3,4,5,9. (Sorted)

Theorem: For a list of n elements, odd-even transposition sort guarantees a sorted list in at most n phases.

```
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) { // Compare-swap (a[i-1], a[i]) for i=1,3,5,...
                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) { // Compare-swap (a[i], a[i+1]) for i=1,3,5,...
                if (a[i] > a[i+1]) {
                    temp = a[i]; a[i] = a[i+1]; a[i+1] = temp;
                }
            }
        }
    }
}
```

```

    }
  }
}
} // Odd_even_sort

```

Parallel Odd-Even Transposition Sort

The advantage of odd-even transposition sort for parallelism is that **all compare-swaps within a single phase can occur simultaneously**.

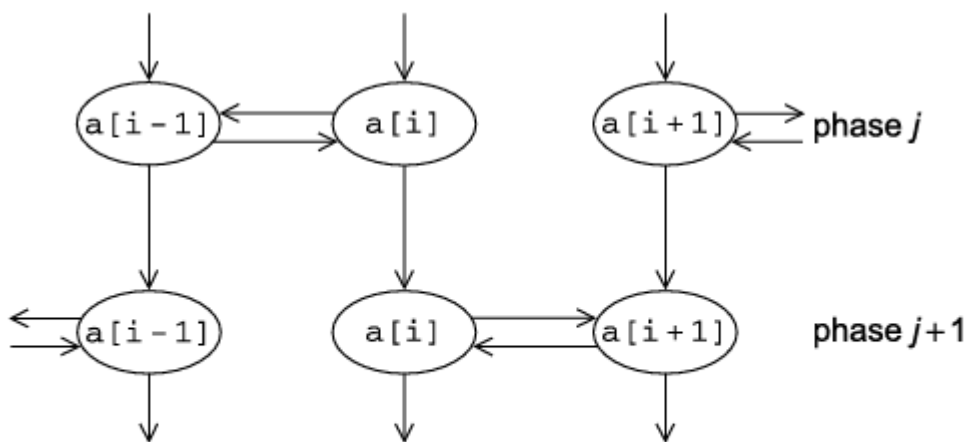


FIGURE 3.12

Communications among tasks in odd-even sort. Tasks determining $a[k]$ are labeled with $a[k]$.

Foster's Methodology Application:

- **Tasks:** Determining the value of $a[i]$ at the end of phase j .
- **Communications:** A task determining $a[i]$ needs to communicate with tasks determining $a[i-1]$ or $a[i+1]$ (its neighbors). Values from one phase are needed for the next.

Case 1:

$n = p$ (One key per process) If each process holds one key, the algorithm is straightforward: in even phases, process i exchanges its key with process $i-1$ (if i is odd) or $i+1$ (if i is even). In odd phases, the pairings are reversed.

After exchanging, each process decides which of the two keys to keep to maintain local sorted order. **Problem:** This is highly inefficient.

The cost of sending/receiving a message for each compare-exchange would quickly dominate the (trivial) computation for small n . Parallelism is only beneficial when local computation significantly outweighs communication.

Case 2:

$n/p > 1$ (Multiple keys per process) This is the more practical scenario. Each process holds a block of n/p keys.

1. **Local Sort:** Each process first sorts its n/p local keys using a fast serial sorting algorithm (e.g., qsort).
2. **Phased Compare-Exchange (Global Sort):** The algorithm proceeds in phases, similar to the serial odd-even sort, but now processes exchange *blocks of keys* with their partners.
 - In each phase, processes pair up based on their ranks (even-odd or odd-even).
 - Each pair of processes (e.g., Process q and Process r) exchange their entire local lists of n/p keys.
 - After exchanging, each process now has $2 \cdot n/p$ keys (its own plus its partner's). They then decide which n/p keys to keep:
 - If $\text{my_rank} < \text{partner}$, keep the *smaller* half of the combined keys.
 - If $\text{my_rank} > \text{partner}$, keep the *larger* half of the combined keys.
 - This ensures that keys "move" in the correct direction across process boundaries to achieve global sortedness.

Example (Table 3.8): $p=4$, $n=16$ (4 keys per process).

Table 3.8 Parallel odd-even transposition sort.

Time	Process			
	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

- **Start:** Unsorted keys distributed.
- **After Local Sort:** Each process's 4 keys are sorted locally.
- **Phase 0 (Even):** P0 pairs with P1, P2 pairs with P3.
 - P0 and P1 exchange their 4 keys. P0 keeps the 4 smallest, P1 keeps the 4 largest.
 - P2 and P3 exchange their 4 keys. P2 keeps the 4 smallest, P3 keeps the 4 largest.

- **Phase 1 (Odd):** P1 pairs with P2. (P0 and P3 are "idle" in this phase for simplicity in this example, though a typical odd-even sort would involve these pairs too if applicable).
 - P1 and P2 exchange their 4 keys. P1 keeps the 4 smallest, P2 keeps the 4 largest.
- This process continues.

Theorem: If parallel odd-even transposition sort is run with p processes, then after p phases, the input list will be sorted (globally distributed as per the output definition).

General Parallel Algorithm Structure:

Sort local keys; // Using a fast serial sort (e.g., qsort)

```
for (phase = 0; phase < comm_sz; phase++) {
    partner = Compute_partner(phase, my_rank); // Determine communication partner

    if (I'm not idle) { // If partner is valid
        // Communication: Exchange keys with partner
        // (Details to be refined for safety)

        if (my_rank < partner)
            Keep smaller keys; // Merge and keep smaller half
        else
            Keep larger keys; // Merge and keep larger half
    }
}
```

Computing the Partner Rank: The Compute_partner logic depends on the phase and rank:

C

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

```
    partner = my_rank + 1;
else // Even rank
    partner = my_rank - 1;
}

// Handle boundary cases (processes at ends of communicator)
if (partner == -1 || partner == comm_sz)
    partner = MPI_PROC_NULL; // MPI_PROC_NULL means no communication
```

MPI_PROC_NULL is a special MPI constant. When used as a source or destination in point-to-point communication, the communication call returns immediately without actually performing any transfer.

Safety in MPI Programs

A critical issue in MPI programming is **safety**, particularly when multiple processes try to send and receive simultaneously.

The Problem with MPI_Send and MPI_Recv (Unsafe Code):

```
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);
```

This sequence is **unsafe**. MPI_Send can behave in two ways:

1. **Buffered:** Copies message to an MPI-managed buffer and returns immediately.
2. **Blocking (Synchronous):** Blocks until the matching MPI_Recv on the destination starts. MPI implementations often have a threshold: small messages are buffered, large messages block. If both processes in a pair execute MPI_Send and block, they will both wait indefinitely for a receive that never starts, leading to **deadlock (hang)**. Such a program is "unsafe" because its behavior (running or hanging) can depend on message size or implementation details.

Checking for Safety (MPI_Ssend):

To explicitly test if a program is safe, replace MPI_Send with MPI_Ssend (synchronous send). MPI_Ssend is **guaranteed to block until the matching receive starts**.

If the program still runs correctly with MPI_Ssend, then the original MPI_Send usage was safe. If it hangs, it was unsafe.

Making Programs Safe (Restructuring Communication):

The most common cause of unsafe behavior is "simultaneous send-receive" patterns (like the partner exchange or a ring pass).

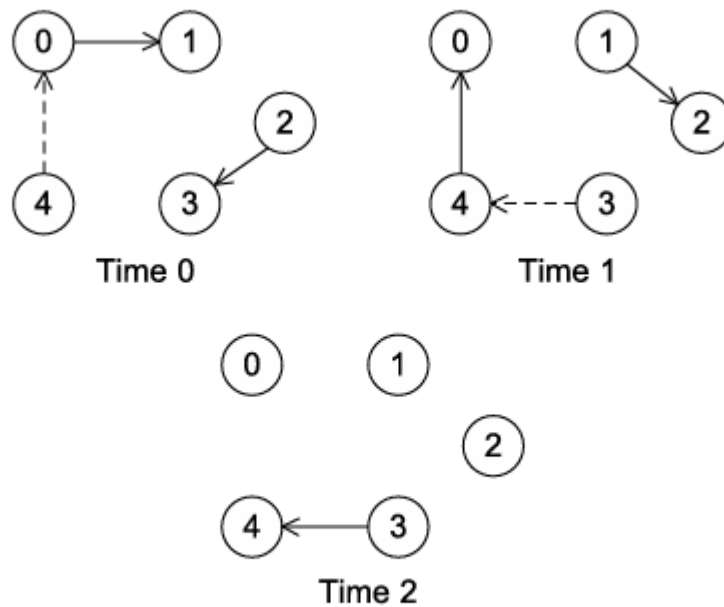
To make them safe, restructure the communication so that some processes receive before sending.

Example (Ring Pass):

```
// Unsafe (both try to send first)
// P0 send to P1, P1 send to P2, etc.
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);

// Safe (staggered send/receive based on rank parity)
if (my_rank % 2 == 0) { // Even ranks send then receive
    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 { // Odd ranks receive then send
    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 at least one process in each pair (or cycle) is ready to receive before the other tries to send, breaking the deadlock. This works for both even and odd comm_sz (as illustrated in Figure 3.13 for comm_sz=5).

**FIGURE 3.13**

Safe communication with five processes.

MPI_Sendrecv: MPI provides a dedicated function for safe simultaneous send and receive operations:

```

int MPI_Sendrecv(
    void* send_buf_p, /* in */
    int    send_buf_size, /* in */
    MPI_Datatype send_buf_type, /* in */
    int    dest, /* in */
    int    send_tag, /* in */
    void* recv_buf_p, /* out */
    int    recv_buf_size, /* in */
    MPI_Datatype recv_buf_type, /* in */
    int    source, /* in */
    int    recv_tag, /* in */
    MPI_Comm communicator, /* in */
    MPI_Status* status_p /* in */
);

```

MPI_Sendrecv performs a blocking send and a receive in a single call. The MPI implementation handles the internal scheduling to **guarantee safety**, preventing deadlocks. This function effectively replaces the complex `if (my_rank % 2 == 0)` logic.

MPI_Sendrecv_replace: If the send buffer and receive buffer are the same (i.e., you want to replace your local data with what you receive, while simultaneously sending out your original data), use MPI_Sendrecv_replace.

Final Details of Parallel Odd-Even Sort

Using MPI_Sendrecv is the preferred way to implement the communication step in the parallel odd-even sort:

```
MPI_Sendrecv(my_keys, local_n, MPI_INT, partner, 0,  
             recv_keys, local_n, MPI_INT, partner, 0, comm,  
             MPI_STATUS_IGNORE);
```

Where $local_n = n/comm_sz$. my_keys holds the keys to be sent, and recv_keys will hold the keys received from the partner.

Merging and Keeping Keys:

After the MPI_Sendrecv call, each process has local_n keys in my_keys (its original keys) and local_n keys in recv_keys (from its partner).

Both my_keys and recv_keys are already locally sorted. The task is to combine these 2cdottextlocal_n keys and extract the appropriate textlocal_n keys.

Merge_low (Program 3.16):

This function merges two sorted arrays (my_keys and recv_keys) into a temporary array (temp_keys) and then copies the first local_n elements (the smallest half) back into my_keys.

```
void Merge_low(int my_keys[], int recv_keys[], int temp_keys[], int local_n) {  
    int m_i, r_i, t_i;  
    m_i = r_i = t_i = 0; // Pointers for my_keys, recv_keys, temp_keys  
  
    // Merge the two sorted lists, taking the smallest local_n elements  
    while (t_i < local_n) {  
        if (my_keys[m_i] <= recv_keys[r_i]) {  
            temp_keys[t_i] = my_keys[m_i];  
            m_i++;  
        } else {
```

```

temp_keys[t_i] = recv_keys[r_i];
r_i++;
}
t_i++;
}
// Copy the smallest local_n elements back to my_keys
for (m_i = 0; m_i < local_n; m_i++)
    my_keys[m_i] = temp_keys[m_i];
} // Merge_low

```

This function is called by the process that needs to keep the smaller keys (i.e., $\text{my_rank} < \text{partner}$).

Merge_high (Implicit):

To keep the larger n/p keys (for $\text{my_rank} > \text{partner}$), the merge logic is reversed (start merging from the end of the arrays and take the largest local_n elements).

Optimization (Exercise 3.28):

Further optimization can avoid copying arrays by simply swapping pointers if the data is stored strategically.

Table 3.9 would show the run-times for the parallel odd-even sort. It's important to note that for $\text{comm_sz} = 1$, the time would be for a fast serial sort like quicksort (the initial local sort), not the much slower serial odd-even sort. This highlights the importance of choosing an efficient local sorting algorithm within the parallel framework.

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

Processes	Number of Keys (in thousands)				
	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