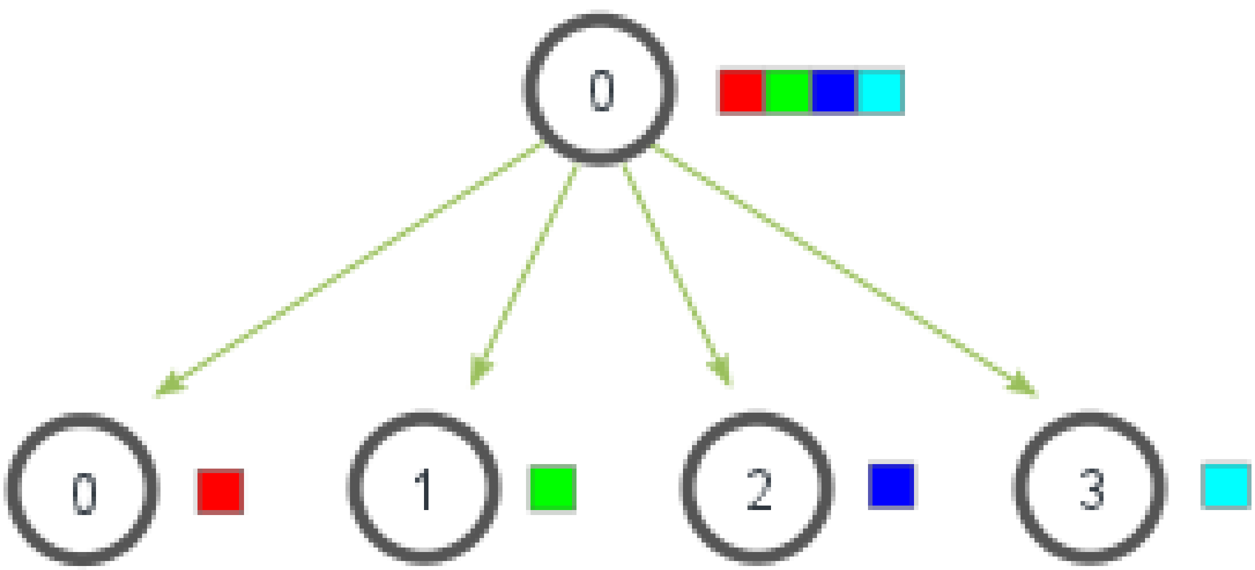
Week11

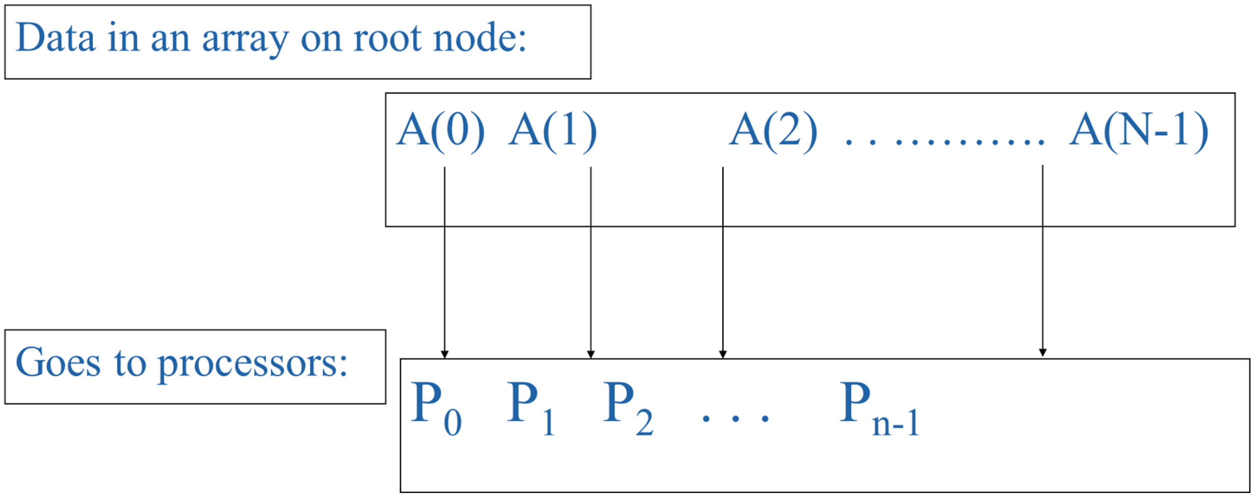
MPI_Scatter and MPI_Gather

These are two fundamental **collective communication** operations in MPI used for **distributing** and **collecting** data among processes.

---

1. `MPI_Scatter` - Distribute Data from Root to All Processes





Prototype

```
int MPI_Scatter(
    const void* sendbuf,    // Buffer at root containing all data (input, significant only at root)
    int sendcount,          // Number of elements sent to each process
    MPI_Datatype sendtype,  // Data type of sent elements (e.g., MPI_INT)
    void* recvbuf,          // Buffer where received data will be stored (output)
    int recvcount,          // Number of elements to receive (should match sendcount)
    MPI_Datatype recvtype,  // Data type of received elements
    int root,              // Rank of the root process (the one that scatters)
    MPI_Comm comm          // Communicator (usually MPI_COMM_WORLD)
);
```

Why Use `MPI_Scatter`?

- **Distributes a large dataset** from the **root process** to **all other processes**.
- **Balanced workload**: Each process gets an equal chunk of data.
- **Efficient alternative** to manually sending data with `MPI_Send`/`MPI_Recv`.

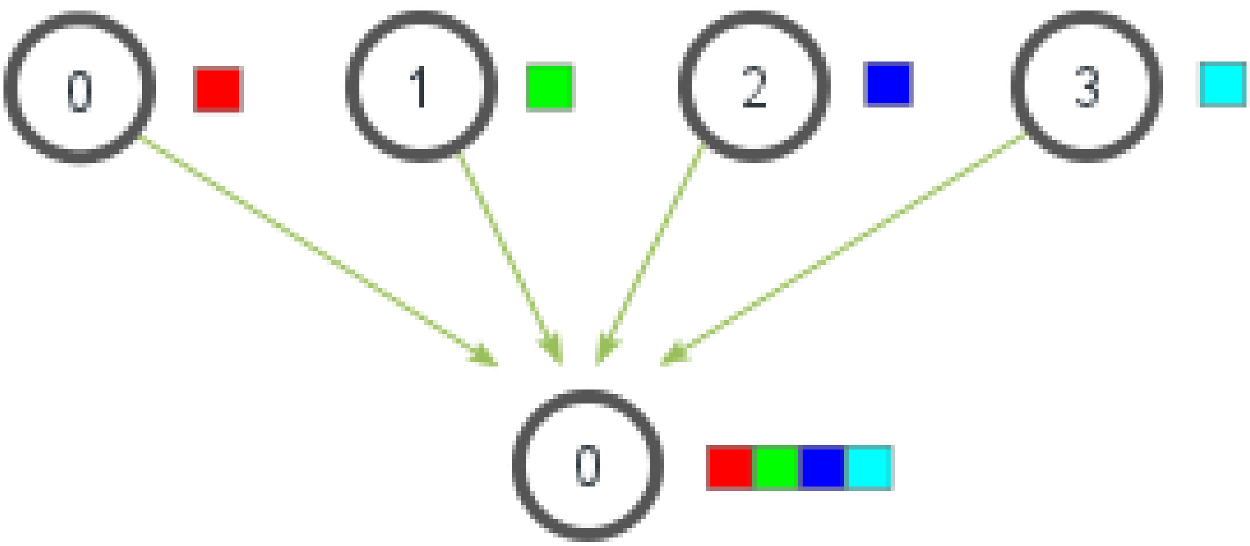
Example

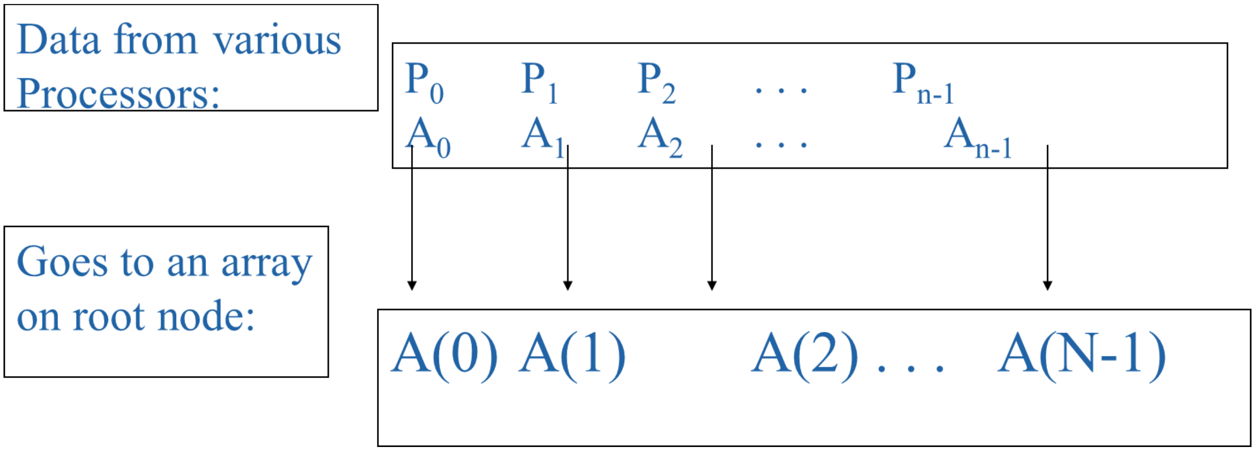
```
int data[12] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}; // Only root has this
int local_data[3]; // Each process gets 3 elements

MPI_Scatter(
```

```
    data,        // Root sends this entire array
    3,           // Each process gets 3 elements
    MPI_INT,     // Data type
    local_data, // Each process stores its part here
    3,           // Each process expects 3 elements
    MPI_INT,
    0,           // Root (rank 0) does the scattering
    MPI_COMM_WORLD
);

// Now:
// - Process 0 gets {1, 2, 3}
// - Process 1 gets {4, 5, 6}
// - Process 2 gets {7, 8, 9}
// - Process 3 gets {10, 11, 12}
```

---

2. `MPI_Gather` - Collect Data from All Processes to Root





Prototype

```
int MPI_Gather(
    const void* sendbuf,   // Data to be sent by each process (input)
    int sendcount,         // Number of elements sent by each process
    MPI_Datatype sendtype, // Data type of sent elements
    void* recvbuf,         // Buffer at root to store gathered data (output, si
gnificant only at root)
    int recvcount,         // Number of elements received per process (should m
atch sendcount)
    MPI_Datatype recvtype, // Data type of received elements
    int root,             // Rank of the root process (collects data)
    MPI_Comm comm         // Communicator (usually MPI_COMM_WORLD)
);
```

Why Use `MPI_Gather`?

- **Combines results** from all processes into **a single array at the root**.
- **Useful after parallel computation** (e.g., each process computes a part, then gathers results).
- **More efficient** than manually collecting data with `MPI_Send`/`MPI_Recv`.

Example

```
int local_data[3] = {rank+1, rank+2, rank+3}; // Each process has its own data
int gathered_data[12]; // Only root will store the full result
```

```c
MPI_Gather(
    local_data,     // Each process sends its 3 elements
    3,              // Each process sends 3 elements
    MPI_INT,
    gathered_data,  // Root collects all data here
    3,              // Root expects 3 elements from each process
    MPI_INT,
    0,              // Root (rank 0) collects data
    MPI_COMM_WORLD
);

// If rank == 0, gathered_data will be:
// {1, 2, 3, 2, 3, 4, 3, 4, 5, 4, 5, 6} (assuming 4 processes)
```

Key Differences Between `MPI_Scatter` and `MPI_Gather`

Feature	`MPI_Scatter`	`MPI_Gather`
Direction	Root → All	All → Root
Use Case	Distribute data for parallel work	Collect results after computation
Buffer Usage	`sendbuf` (root only), `recvbuf` (all)	`sendbuf` (all), `recvbuf` (root only)

When to Use Them?

- `MPI_Scatter`
  - Splitting a large dataset for parallel processing.
  - Example: Distributing rows of a matrix to different processes.
- `MPI_Gather`
  - Combining partial results after computation.

    - Example: Collecting local sums into a global sum at the root.

Summary

- `MPI_Scatter` = Distribute data (one-to-many).
- `MPI_Gather` = Collect data (many-to-one).
- Both are **collective operations** (all processes must call them).
- **More efficient** than point-to-point communication (`MPI_Send`/`MPI_Recv`) for structured data distribution.

These operations are **essential for parallel algorithms** where work is divided among processes! 🚀

1. MPI_Scatter Example (Distribute Data)

Objective: Root process splits an array and sends chunks to all processes.

```c
#include <stdio.h>
#include <mpi.h>

int main() {
    MPI_Init(NULL, NULL);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int data[12] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120}; // Only root has this
    int local_data[3]; // Each process will receive 3 elements

    // Scatter the data from root (rank 0)
```

```
    MPI_Scatter(
        data, 3, MPI_INT,     // Send 3 ints from root's 'data'
        local_data, 3, MPI_INT, // Each process receives 3 ints into 'local_dat
a'
        0, MPI_COMM_WORLD      // Root is rank 0
    );

    // Each process prints its received data
    printf("Process %d received: %d, %d, %d\n",
           rank, local_data[0], local_data[1], local_data[2]);

    MPI_Finalize();
    return 0;
}
```

Output (for 4 processes):

```
Process 0 received: 10, 20, 30
Process 1 received: 40, 50, 60
Process 2 received: 70, 80, 90
Process 3 received: 100, 110, 120
```

---

2. MPI_Gather Example (Collect Data)

Objective: All processes send their data to the root for aggregation.

```
#include <stdio.h>
#include <mpi.h>

int main() {
    MPI_Init(NULL, NULL);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int local_data[2] = {rank * 10, rank * 10 + 5}; // Each process has 2 valu
es
```

```
    int gathered_data[8]; // Root will collect all data (4 processes × 2 elements)

    // Gather data at root (rank 0)
    MPI_Gather(
        local_data, 2, MPI_INT,     // Each process sends 2 ints
        gathered_data, 2, MPI_INT, // Root receives 2 ints per process
        0, MPI_COMM_WORLD          // Root is rank 0
    );

    // Root prints the gathered data
    if (rank == 0) {
        printf("Gathered data at root: ");
        for (int i = 0; i < 8; i++) {
            printf("%d ", gathered_data[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output (for 4 processes):

```
Gathered data at root: 0 5 10 15 20 25 30 35
```

Key Notes

1. `MPI_Scatter`:
    - Root divides an array and sends chunks to all processes.
    - Non-root processes only need a receive buffer.
2. `MPI_Gather`:
    - All processes send data to the root.
    - Only root needs a receive buffer.

MPI_Reduce

is a collective communication operation in MPI (Message Passing Interface) that combines values from all processes in a communicator using a specified operation (like sum, max, min, etc.) and stores the result in a single target process (called the *root* process).

Prototype of `MPI_Reduce`

```
int MPI_Reduce(
    const void* sendbuf,     // Address of the local data to be reduced (input)
    void* recvbuf,           // Address where the reduced result will be stored
(output, significant only at root)
    int count,               // Number of elements in the send buffer
    MPI_Datatype datatype,   // Data type of the elements (e.g., MPI_INT, MPI_FL
OAT)
    MPI_Op op,               // Reduction operation (e.g., MPI_SUM, MPI_MAX)
    int root,                // Rank of the process that receives the result
    MPI_Comm comm            // Communicator (usually MPI_COMM_WORLD)
);
```

Why is `MPI_Reduce` Used?

1. **Aggregates Data Efficiently**
    - Instead of manually gathering data to one process and then computing the result, `MPI_Reduce` performs the reduction in a single optimized step.
    - Example: Summing values from all processes (`MPI_SUM`), finding the maximum (`MPI_MAX`), or computing logical AND (`MPI_LAND`).

2. **Supports Various Reduction Operations**
    - Common operations:
        - `MPI_SUM` (Summation)
        - `MPI_PROD` (Product)
        - `MPI_MAX` (Maximum value)
        - `MPI_MIN` (Minimum value)
        - `MPI_LAND` (Logical AND)
        - `MPI_BAND` (Bitwise AND)
        - And more...
3. **Better Performance**
    - MPI implementations optimize `MPI_Reduce` for the underlying hardware (e.g., using tree-based algorithms for scalability).
4. **Simplifies Parallel Computations**
    - Used in many numerical algorithms (e.g., dot products, matrix-vector multiplication, global statistics).

Example Breakdown

1. Each Process Computes a Local Sum

```
int a = rand() % 100;       // Random number 1
int b = rand() % 100;       // Random number 2
int local_sum = a + b;      // Local sum (different for each process)
```

2. Reduction to a Single Process (Root)

```
MPI_Reduce(
    &local_sum,      // Input: Each process sends its local_sum
    &global_sum,     // Output: Only root gets the final sum
    1,               // Only 1 value is being reduced
    MPI_INT,         // Data type is integer
```

```
    MPI_SUM,          // Operation: Summation
    0,                // Root process (rank 0 gets the result)
    MPI_COMM_WORLD   // Use all processes
);
```

3. Root Process Prints the Final Result

```
if (rank == 0) {
    printf("Total sum from all processes: %d\n", global_sum);
}
```

Key Notes

- `MPI_Reduce` **vs** `MPI_Allreduce`
  - `MPI_Reduce` sends the result to **only one process (root)**.
  - `MPI_Allreduce` sends the result to **all processes** (useful when every process needs the reduced value).
- **Blocking Operation**
  - All processes must call `MPI_Reduce` (it synchronizes).
- **Efficiency**
  - Prefer `MPI_Reduce` over manual gather + compute for better performance.

Common Use Cases

1. **Parallel Summation** (e.g., computing total energy in simulations)
2. **Finding Global Min/Max** (e.g., optimization problems)
3. **Dot Products** (used in linear algebra)
4. **Statistical Aggregation** (e.g., computing mean/variance)

This makes `MPI_Reduce` a fundamental operation in parallel computing! 🚀

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>

int main() {
    MPI_Init(NULL, NULL);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // Each process gets different random numbers
    srand(time(NULL) + rank);
    int a = rand() % 100;
    int b = rand() % 100;
    int local_sum = a + b;

    printf("Process %d: %d + %d = %d\n", rank, a, b, local_sum);

    // Sum all local sums to process 0
    int global_sum;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("\nTotal sum from all processes: %d\n", global_sum);
    }

    MPI_Finalize();
    return 0;
}
```

How to Use:

1. Save as `simple_mpi_sum.c`
2. Compile: `mpicc simple_mpi_sum.c -o simple_mpi_sum`
3. Run: `mpirun -n 4 ./simple_mpi_sum`

Sample Output:

```
Process 1: 28 + 65 = 93
Process 2: 14 + 70 = 84
Process 3: 53 + 41 = 94
Process 0: 67 + 32 = 99

Total sum from all processes: 370
```

Key Features:

- Each process generates just 2 random numbers (0-99)
- Calculates local sum
- Uses MPI_Reduce to sum all local sums
- Process 0 displays final result
- Clean and minimal - focuses just on the reduction operation