Computação em Larga Escala

Message Passage Interface (MPI)

Eurico Pedrosa

António Rui Borges

Universidade de Aveiro - DETI

2025-03-30

Collective Communication

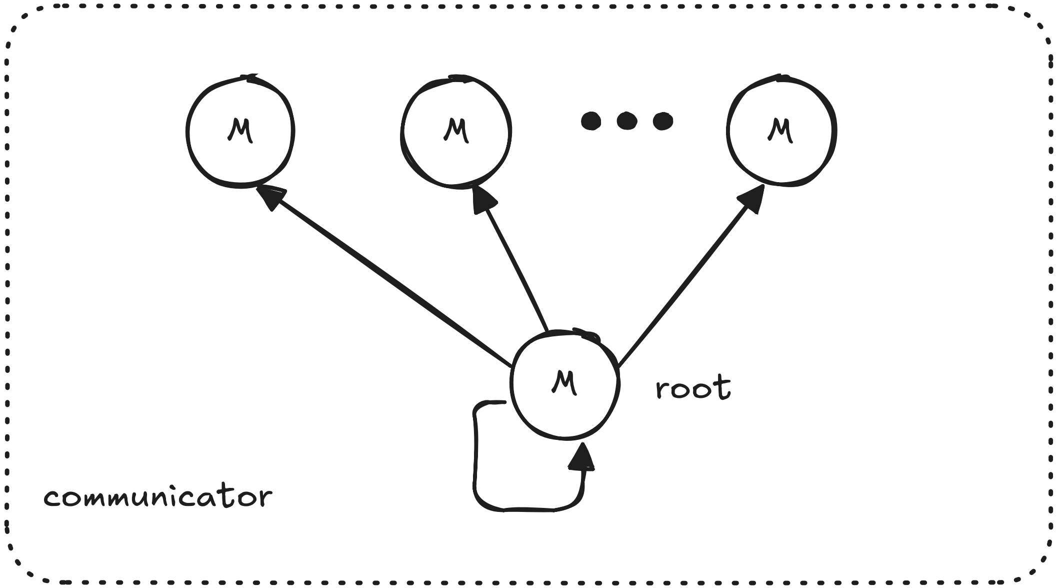
Collective communication refers to operations that involve **multiple processes** working together as part of a defined **communication group**. Unlike point-to-point communication, these operations enable more structured and efficient **data distribution and collection** among processes.

There are several types of collective communication in MPI:

- **Broadcast (`MPI_Bcast`)**: A message is sent from a **root process** to **all other processes** within the communication group, including the root itself. **Used when the same data must be distributed to all processes.**
- **Scatter (`MPI_Scatter`)**: A **root process** sends **distinct segments of data** to each process in the group, including itself. **Useful for parallelizing work where each process handles a portion of the data.**

- **Gather (`MPI_Gather`)**: Each process in the group sends its data to the **root process**, which collects all pieces into a single structure. **Ideal for collecting results computed in parallel.**

Broadcast in MPI



The **broadcast** operation (`MPI_Bcast`) enables a **single process**, identified as the **root**, to send the **same message (`M`)** to **all other processes** in a communication group, including itself.

Broadcast in MPI

Key characteristics:

- The message `M` is sent by the **root process** (identified by its **rank**) to **all participating processes**.
- In the **standard implementation**, `MPI_Bcast` is a **blocking operation**: All processes **wait** until the message is fully received before proceeding.
- Conceptually, the operation involves:
  - The **root process** performing a **send**
  - All other processes performing a **receive**

Broadcast in MPI

MPI_Bcast – Function Signature

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```

This function broadcasts a message from the **root process** to **all processes** in the given **communicator**.

🔧 Parameters:

- **`buffer`**: A pointer to the **memory region** where the message data is stored (in the root), or where it will be received (in other processes).
- **`count`**: The **number of elements** in the message buffer.
- **`datatype`**: The **MPI data type** of the message content (e.g., `MPI_INT`, `MPI_FLOAT`).
- **`root`**: The **rank** of the process that will act as the broadcaster.
- **`comm`**: The **communicator** that defines the **group of processes** participating in the broadcast (typically `MPI_COMM_WORLD`).

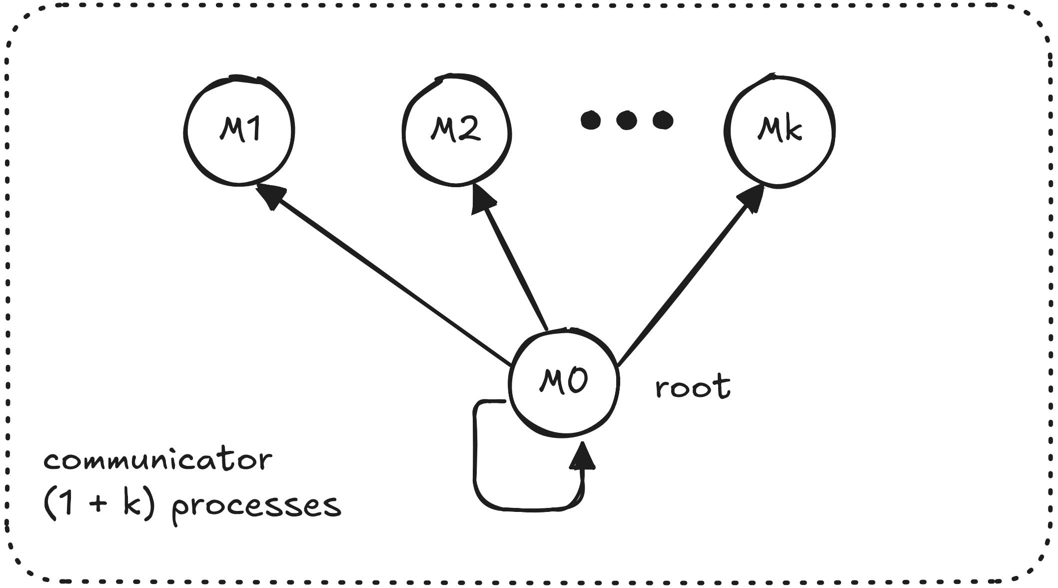
Returns

- Returns `MPI_SUCCESS` on success, or an error code otherwise.

Broadcast in MPI


```cpp
#include <mpi.h>
#include <iostream>

int main(int argc, char** argv) {
    int rank, size, data;

    MPI_Init(&argc, &argv);               // Initialize the MPI environment
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get the rank of the process
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the total number of processes

    if (rank == 0) {
        data = 42; // Root process sets the data
        std::cout << "Process " << rank << " broadcasting data = " << data << std::endl;
    }

    // Broadcast the value of 'data' from root (rank 0) to all processes
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // All processes (including root) now have the same value of 'data'
    std::cout << "Process " << rank << " received data = " << data << std::endl;

    MPI_Finalize(); // Finalize the MPI environment
    return 0;
}
```

Scatter in MPI



The **scatter** operation (`MPI_Scatter`) allows the **root process** to distribute **distinct parts of a message** to **each process** in a communication group, including itself.

Scatter in MPI

Key Characteristics:

- The root process holds a **collection of `k+1` messages**:

$$M_0, M_1, M_2, \ldots, M_k$$

  where `k + 1` is the **number of processes** in the communication group.
- Each process receives **one unique chunk** of the data:
  - Process 0 gets M_0, process 1 gets M_1, …, process `k` gets M_k.
- By default, `MPI_Scatter` is a **blocking operation**: All processes **wait** until their respective message part is received.
- Conceptually:
  - The **root process** performs multiple **sends** (one to each process).
  - All processes perform a **receive**.

Example Use Case: Distributing rows of a matrix or chunks of a large array to parallel workers for **independent computation**.

Scatter in MPI

MPI_Scatter – Function Signature

```
int MPI_Scatter(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm
);
```

Parameters:

- **`sendbuf`**: Pointer to the buffer holding all data to be sent. **(Significant only at the root process.)**
- **`sendcount`**: Number of elements to send **to each process**.
- **`sendtype`**: Data type of elements in the send buffer.
- **`recvbuf`**: Pointer to the buffer where each process will **store its received data**.
- **`recvcount`**: Maximum number of elements each process expects to receive.
- **`recvtype`**: Data type of elements in the receive buffer.
- **`root`**: Rank of the **root process**.
- **`comm`**: Communicator that defines the **group of processes**.

Scatter in MPI

MPI_Scatterv – Function Signature

```
int MPI_Scatterv(
    void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm
);
```

Extended Parameters:

- **`sendcounts`**: Array specifying the number of elements **to send to each process**.
- **`displs`**: Array specifying the **displacement** (offset) in `sendbuf` for each message. **(Each displacement is relative to the start of `sendbuf`.)**

This variant supports **non-uniform message sizes**, making it useful for **irregular data distributions**.

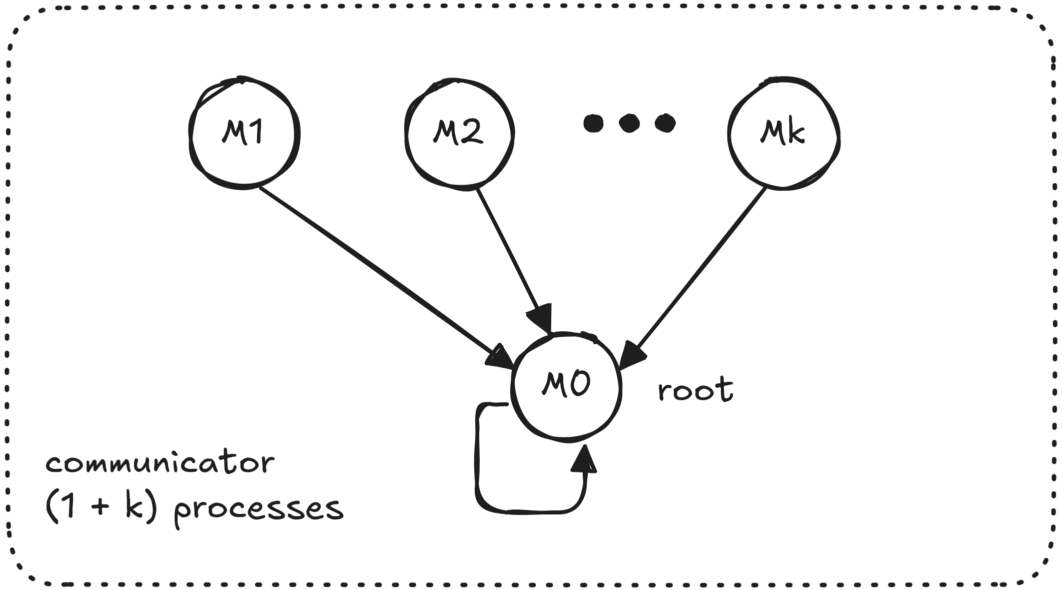
MPI_Scatter Example


```cpp
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int chunk_size = 2;
    std::vector<int> recvbuf(chunk_size);

    std::vector<int> sendbuf;
    if (rank == 0) {
        sendbuf.resize(size * chunk_size);
        for (int i = 0; i < size * chunk_size; ++i)
            sendbuf[i] = i + 1;
    }
    MPI_Scatter(sendbuf.data(), chunk_size, MPI_INT,
                recvbuf.data(), chunk_size, MPI_INT,
                0, MPI_COMM_WORLD);

    std::cout << "Process " << rank << " received:";
    for (int val : recvbuf) std::cout << " " << val;
    std::cout << std::endl;

    MPI_Finalize();
    return 0;
}
```

MPI_Scatterv Example


```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int recvcount;
    std::vector<int> sendbuf, sendcounts(size), displs(size);
    std::vector<int> recvbuf;

    if (rank == 0) {
        int total = 0;
        for (int i = 0; i < size; ++i) {
            sendcounts[i] = i + 1;  // different amount for each process
            displs[i] = total;
            total += sendcounts[i];
        }
        sendbuf.resize(total);
        for (int i = 0; i < total; ++i) sendbuf[i] = i + 1;
    }

    MPI_Scatter( sendcounts.data(), 1, MPI_INT, &recvcount, 1, MPI_INT, 0, MPI_COMM_WORLD);
    recvbuf.resize(recvcount);
    MPI_Scatterv(sendbuf.data(), sendcounts.data(), displs.data(), MPI_INT,
                 recvbuf.data(), recvcount, MPI_INT, 0, MPI_COMM_WORLD);

    std::cout << "Process " << rank << " received:";
    for (int val : recvbuf) std::cout << " " << val;
    std::cout << std::endl;

    MPI_Finalize();
    return 0;
}
```

Gather in MPI



The **gather** operation (`MPI_Gather`) collects individual messages from **all processes** in a communication group and assembles them into a single buffer on the **root process**.

Gather in MPI

Key Characteristics:

- Each process (including the root) sends a **message**:

$$M_0, M_1, M_2, \ldots, M_k$$

  where $k + 1$ is the **group size**.
- The **root process** collects all messages and stores them **in order of process ranks** in a receive buffer.
- By default, `MPI_Gather` is a **blocking operation**: The root process waits until it has **received all messages**.
- Conceptually:
  - All processes perform a **send**.
  - The **root process** performs a **receive** from each.

Example Use Case: Aggregating partial results (e.g., local sums, vectors) from all processes into a final array on the root for final processing or output.

Gather in MPI

MPI_Gather – Function Signature

```
int MPI_Gather(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm
);
```

Parameters:

- **`sendbuf`**: Pointer to the message to be sent by each process.
- **`sendcount`**: Number of elements each process sends.
- **`sendtype`**: Data type of elements in `sendbuf`.
- **`recvbuf`**: Pointer to the buffer where the **root** will store the collected data. **(Significant only at root.)**
- **`recvcount`**: Number of elements received **from each process**. **(Significant only at root.)**
- **`recvtype`**: Data type of the receive buffer elements. **(Significant only at root.)**
- **`root`**: Rank of the root process.
- **`comm`**: MPI communicator.

Gather in MPI

MPI_Gatherv – Function Signature

```
int MPI_Gatherv(
    void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,
    int root, MPI_Comm comm
);
```

Extended Parameters:

- **`recvcounts`**: Array specifying how many elements are received from each process.
- **`displs`**: Array specifying **displacements** (offsets) in `recvbuf` where each message should be stored.

`MPI_Gatherv` is used for **non-uniform data gathering**, where each process might send a **different number of elements**.

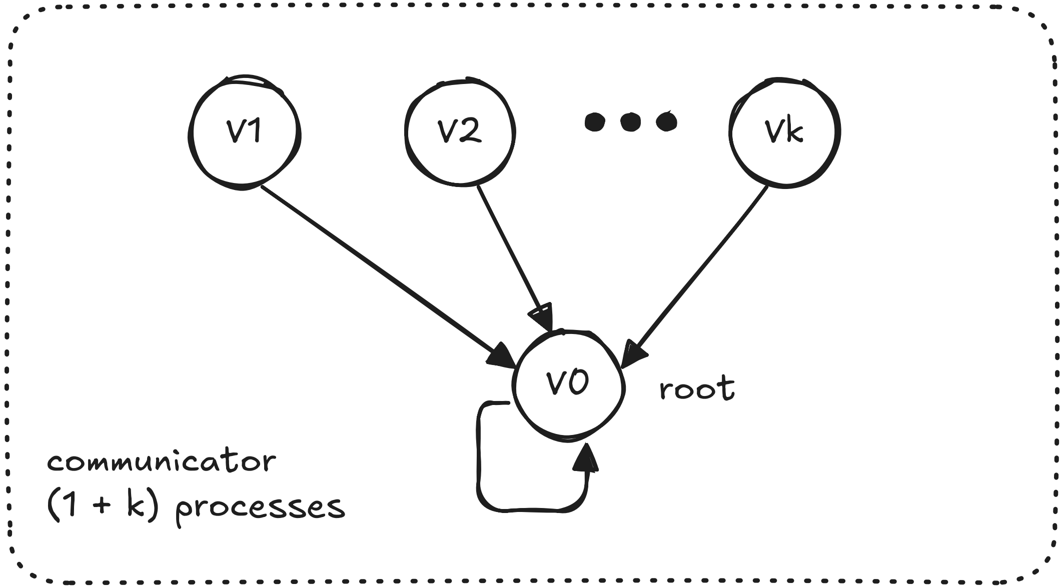
MPI_Gather Example


```cpp
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;

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

    int sendval = rank + 1;
    std::vector<int> recvbuf;

    if (rank == 0) recvbuf.resize(size); // collect one int from each process

    MPI_Gather(&sendval, 1, MPI_INT,            // each process sends one int
               recvbuf.data(), 1, MPI_INT,      // root receives one int from each process
               0, MPI_COMM_WORLD);

    if (rank == 0) {
        std::cout << "Gathered values:";
        for (int val : recvbuf) std::cout << " " << val;
        std::cout << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

MPI_Gatherv Example


```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Each process sends (rank + 1) values
    std::vector<int> sendbuf(rank + 1);
    for (int i = 0; i <= rank; ++i) sendbuf[i] = (rank + 1) * 10 + i;

    std::vector<int> recvcounts, displs, recvbuf;
    if (rank == 0) {
        recvcounts.resize(size);
        displs.resize(size);
        int offset = 0;
        for (int i = 0; i < size; ++i) {
            recvcounts[i] = i + 1;
            displs[i] = offset;
            offset += recvcounts[i];
        }
        recvbuf.resize(offset); // total size for root to collect everything
    }

    MPI_Gatherv(sendbuf.data(), sendbuf.size(), MPI_INT,
                recvbuf.data(), recvcounts.data(), displs.data(), MPI_INT,
                0, MPI_COMM_WORLD);

    if (rank == 0) {
        std::cout << "Gatherv result:";
        for (int val : recvbuf) std::cout << " " << val;
        std::cout << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

Reduce in MPI



The `MPI_Reduce` operation allows processes to perform a **global element-wise computation** on their data and deliver the final result to a designated **root process**.

Reduce in MPI

Key Characteristics:

- Each process provides a **value array**:

$$V_0, V_1, V_2, \ldots, V_k$$

  where $k + 1$ is the **number of processes** in the communication group.
- These arrays are **combined element-wise** using a **commutative binary operator**, such as:
  - `MPI_SUM`, `MPI_MAX`, `MPI_MIN`, `MPI_PROD`, etc.
- The **resulting array** is sent to the **root process**.

Reduce in MPI

Blocking Semantics:

- The operation is **blocking by default**: The **root process** waits until **all values have been received and reduced**.

Conceptual Model:

- All processes **send** their local arrays.
- The **root process** performs a **receive**, then reduces:

$$\mathrm{Result}[i] = V_0[i] \text{ op } V_1[i] \text{ op } \dots \text{ op } V_{k[i]}$$

Where op is a **commutative binary operation**, applied element-wise.

Reduce in MPI

MPI_Reduce – Function Signature

```c
int MPI_Reduce(
    void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm
);
```

Parameters:

- **`sendbuf`**: Pointer to the local data (input values) to be reduced. **(Each process sends its own buffer.)**
- **`recvbuf`**: Pointer to the buffer where the **root process** stores the result. **(Ignored by non-root processes.)**
- **`count`**: Number of elements in the array being reduced.
- **`datatype`**: MPI data type of the elements (e.g., `MPI_INT`, `MPI_FLOAT`).
- **`op`**: The **reduction operation** to apply. Must be a predefined **commutative** operator like:
  - `MPI_SUM`, `MPI_PROD`, `MPI_MAX`, `MPI_MIN`
  - Bitwise: `MPI_BAND`, `MPI_BOR`, `MPI_BXOR`
- **`root`**: Rank of the process that receives the **final reduced result**.
- **`comm`**: The MPI communicator (e.g., `MPI_COMM_WORLD`) that defines the group of processes.

MPI_Reduce Example

```cpp
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    const int N = 5;
    std::vector<int> local_array(N);
    std::vector<int> result_array(N); // only used at root
    // Seed the random number generator differently per process
    std::srand(static_cast<unsigned int>(std::time(0)) + rank);
    // Fill local array with random values between 1 and 10
    for (int i = 0; i < N; ++i)
        local_array[i] = std::rand() % 10 + 1;

    // Print each process's local data
    std::cout << "Process " << rank << " local array:";
    for (int val : local_array) std::cout << " " << val;
    std::cout << std::endl;

    // Perform reduction (sum) across all arrays
    MPI_Reduce(local_array.data(), result_array.data(),
               N, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        std::cout << "Root received reduced array (sum):";
        for (int val : result_array) std::cout << " " << val;
        std::cout << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

Suggested Reading

- The Art of HPC by Victor Eijkhout of TACC
  - Volume 2: Parallel Programming for Science and Engineering
- Rookie HPC (MPI Documentation)
  - https://rookiehpc.org/mpi/docs/index.html