High Performance Linear Algebra

Lecture 8: Basic functions of MPI

Ph.D. program in High Performance Scientific Computing

Fabio Durastante **Pasqua D'Ambra** **Salvatore Filippone**

January 15, 2026 — 16.00:18.00

Dipartimento
di Matematica
Università di Pisa

Last time on High Performance Linear Algebra

- Durign the last lecture we have introduced the basics of parallel programming using the **Message Passing Interface (MPI)** paradigm.
- We have discussed the main features of MPI and we have seen how to setup a simple MPI environment.
- We have also discussed the usage of SLURM to handle queues on HPC systems.

Today we will continue our discussion on MPI by introducing some of the most used MPI functions.

Table of Contents

2 Message Passing Interface (MPI)

Point-to-Point Communication

- **Send** and **Receive** operations between pairs of processes
- Essential building block for distributed algorithms
- Two main types: Blocking and Non-blocking

Blocking Send and Receive


```
program point_to_point
    use mpi
    implicit none
    integer :: rank, size, ierr, status(MPI_STATUS_SIZE)
    integer :: send_data, recv_data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    if (rank == 0) then
        send_data = 42
```

Blocking Send and Receive


```
        call MPI_Send(send_data, 1, MPI_INTEGER, 1, 0, &
                      MPI_COMM_WORLD, ierr)
    else if (rank == 1) then
        call MPI_Recv(recv_data, 1, MPI_INTEGER, 0, 0, &
                      MPI_COMM_WORLD, status, ierr)
        print *, 'Received:', recv_data
    end if

    call MPI_Finalize(ierr)
end program point_to_point
```

Blocking Send and Receive

Let us analyze the code line by line:

- `MPI_Init`: Initialize the MPI environment
- `MPI_Comm_rank`: Get the rank of the calling process in the communicator
- `MPI_Comm_size`: Get the total number of processes in the communicator
- `MPI_Send`: Locally Blocking send operation
- `MPI_Recv`: Blocking receive operation
- `MPI_Finalize`: Clean up the MPI environment

Blocking Send and Receive

Let us analyze the code line by line:

- `MPI_Init`: Initialize the MPI environment
- `MPI_Comm_rank`: Get the rank of the calling process in the communicator
- `MPI_Comm_size`: Get the total number of processes in the communicator
- `MPI_Send`: Locally Blocking send operation
- `MPI_Recv`: Blocking receive operation
- `MPI_Finalize`: Clean up the MPI environment

A call is said to be **blocking** if the function does not return control to the calling process until the operation is complete.

- `MPI_Send` returns only after the data has been copied out of the *send buffer*
- `MPI_Recv` returns only after the data has been received and placed in the receive buffer
- This can lead to inefficiencies if processes are waiting for each other! However, blocking calls are often simpler to implement and reason about.

Arguments of the MPI Send/Receive

The argument of the Send call are:

- `send_data`: starting address of the send buffer
- `1`: number of elements to send
- `MPI_INTEGER`: datatype of each element
- `1`: rank of the destination process
- `0`: message tag (used to identify messages)
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Send` function is:

```
call MPI_Send(send_data, counter, datatype, dest, tag, comm, ierr)
```

Arguments of the MPI Send/Receive

The argument of the Receive call are:

- `recv_data`: starting address of the receive buffer
- `1`: number of elements to receive
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the source process
- `0`: message tag (used to identify messages)
- `MPI_COMM_WORLD`: communicator
- `status`: status object (contains information about the received message)
- `ierr`: error code

The prototype of the `MPI_Recv` function is:

```
call MPI_Recv(recv_data, counter, datatype, source, tag, comm, status, ierr)
```

Arguments of the MPI Send/Receive

The argument of the Receive call are:

- `recv_data`: starting address of the receive buffer
- `1`: number of elements to receive
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the source process
- `0`: message tag (used to identify messages)
- `MPI_COMM_WORLD`: communicator
- `status`: status object (contains information about the received message)
- `ierr`: error code

The `status` object can be used to retrieve additional information about the received message, such as the actual number of elements received or the source of the message.

MPI Datatypes

MPI provides a variety of predefined datatypes to represent different kinds of data. The following table lists them:

Fortran Type	MPI Datatype
`INTEGER`	`MPI_INTEGER`
`REAL`	`MPI_REAL`
`DOUBLE PRECISION`	`MPI_DOUBLE_PRECISION`
`COMPLEX`	`MPI_COMPLEX`
`DOUBLE COMPLEX`	`MPI_DOUBLE_COMPLEX`
`LOGICAL`	`MPI_LOGICAL`
`CHARACTER`	`MPI_CHAR`

Questions

- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

Questions

- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

The program will hang if there is only 1 process, as the receiver will wait indefinitely for a message that will never arrive. If run with more than 2 processes, only ranks 0 and 1 will participate in the communication; other ranks will do nothing.

Questions

- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

- The program will hang if there is only 1 process, as the receiver will wait indefinitely for a message that will never arrive. If run with more than 2 processes, only ranks 0 and 1 will participate in the communication; other ranks will do nothing.
- Mismatched tags or datatypes will lead to errors or unexpected behavior. The receiver may not receive the intended message, leading to data corruption or program crashes.

Questions

- What happens if we run the code with 1 process only? What if we run it with more than 2 processes?
- What happens if the sender and receiver have mismatched tags or datatypes?
- How can we handle errors in MPI calls?

- The program will hang if there is only 1 process, as the receiver will wait indefinitely for a message that will never arrive. If run with more than 2 processes, only ranks 0 and 1 will participate in the communication; other ranks will do nothing.
- Mismatched tags or datatypes will lead to errors or unexpected behavior. The receiver may not receive the intended message, leading to data corruption or program crashes.
- MPI functions return an error code that can be checked after each call. Additionally, MPI provides error handling routines to manage errors more gracefully.

MPI Error Handling

- Every MPI function returns an error code in the `ierr` variable
- `ierr == MPI_SUCCESS` indicates successful execution
- Error codes can be converted to human-readable messages using `MPI_Error_string`

```
integer :: ierr
character(len=MPI_MAX_ERROR_STRING) :: error_string
integer :: error_len

call MPI_Send(data, 1, MPI_INTEGER, dest, tag, comm, ierr)
if (ierr /= MPI_SUCCESS) then
    call MPI_Error_string(ierr, error_string, error_len, ierr)
    print *, 'Error: ', error_string(1:error_len)
    call MPI_Finalize(ierr)
    stop
end if
```

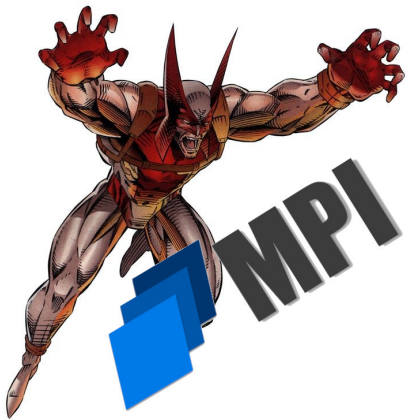
MPI Error Handling

- Every MPI function returns an error code in the `ierr` variable
- `ierr == MPI_SUCCESS` indicates successful execution
- Error codes can be converted to human-readable messages using `MPI_Error_string`

- Always check error codes after critical MPI calls
- Use `MPI_Error_string` to get descriptive error messages
- Call `MPI_Finalize` before terminating on error

Deadlock in MPI

- A **deadlock** occurs when processes are blocked indefinitely, waiting for events that will never occur
- Common cause: circular waiting patterns in blocking send/receive operations
- **Example:** Two processes each waiting to receive before sending



Deadlock Example: Circular Wait


```
program deadlock_example
    use mpi
    implicit none
    integer :: rank, size, ierr, status(MPI_STATUS_SIZE)
    integer :: data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    if (rank == 0) then
        ! Rank 0: First receive, then send
```

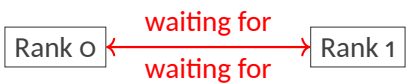
Deadlock Example: Circular Wait


```
        call MPI_Recv(data, 1, MPI_INTEGER, 1, 0, &
                      MPI_COMM_WORLD, status, ierr)
        call MPI_Send(data, 1, MPI_INTEGER, 1, 0, &
                      MPI_COMM_WORLD, ierr)
    else if (rank == 1) then
        ! Rank 1: First receive, then send (same pattern!)
        call MPI_Recv(data, 1, MPI_INTEGER, 0, 0, &
                      MPI_COMM_WORLD, status, ierr)
        call MPI_Send(data, 1, MPI_INTEGER, 0, 0, &
                      MPI_COMM_WORLD, ierr)
    end if
```

Deadlock Example: Circular Wait


```
    call MPI_Finalize(ierr)
end program deadlock_example
```

- Rank 0 calls `MPI_Recv` first and waits for data from rank 1
- Rank 1 calls `MPI_Recv` first and waits for data from rank 0
- Both processes are now blocked, each waiting for the other to send
- Neither process can proceed $\Rightarrow$ **deadlock**!



Solution 1: Order Communication


```
if (rank == 0) then
    ! Rank 0: Send first, then receive
    call MPI_Send(data, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, ierr)
    call MPI_Recv(data, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, status, ierr)
else if (rank == 1) then
    ! Rank 1: Receive first, then send (opposite order)
    call MPI_Recv(data, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, status, ierr)
    call MPI_Send(data, 1, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, ierr)
end if
```

- Break the circular dependency by using different orderings
- Rank 0 sends first, so rank 1's receive completes
- Rank 1 can then send, so rank 0's receive completes

Non-Blocking Communication


```
integer :: request
! Start non-blocking send
call MPI_Isend(send_data, 1, MPI_INTEGER, 1, 0, &
               MPI_COMM_WORLD, request, ierr)
! Do computation while message is in transit
! ...
! Wait for completion
call MPI_Wait(request, status, ierr)
```

- `MPI_Isend`: initiate non-blocking send operation
- Returns immediately without waiting for the send to complete
- Enables overlap of communication and computation

Non-Blocking Communication


```
integer :: request
! Start non-blocking receive
call MPI_Irecv(recv_data, 1, MPI_INTEGER, 0, 0, &
               MPI_COMM_WORLD, request, ierr)
! Do computation while waiting for message
! ...
! Wait for completion
call MPI_Wait(request, status, ierr)
```

- `MPI_Irecv`: initiate non-blocking receive operation
- Returns immediately without waiting for the message to arrive
- `MPI_Wait`: blocks until the operation completes and data is available

The request argument and the wait commands

- The `request` argument is an integer that uniquely identifies the operation
- It is used to track the status of the operation and is required for completion routines like `MPI_Wait`
- `MPI_Wait` blocks the calling process until the specified non-blocking operation completes
- It takes the request handle and a status object as arguments
- The status object can provide information about the completed operation, such as the source, tag, and error code

If we have multiple non-blocking operations, we can use `MPI_Waitall` to wait for all of them to complete:

```
call MPI_Waitall(count, request_array, status_array, ierr)
```

Solution 2: Use Non-Blocking Operations

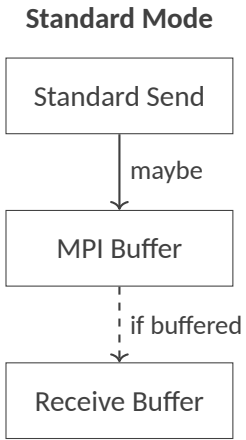

```
integer :: req_send, req_recv
integer :: status_array(MPI_STATUS_SIZE, 2)

! Both ranks initiate sends and receives simultaneously
call MPI_Isend(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, &
    req_send, ierr)
call MPI_Irecv(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, &
    req_recv, ierr)
! Wait for both operations to complete
call MPI_Waitall(2, (/req_send, req_recv/), &
                 status_array, ierr)
```

- Non-blocking operations return immediately
- Both sends and receives can progress independently
- MPI runtime handles message buffering and ordering

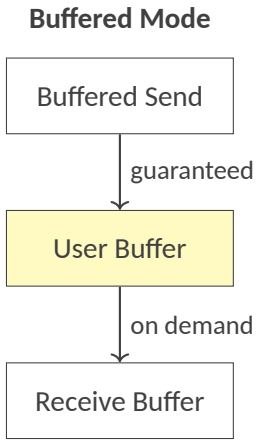
Buffered Communication

- **Standard mode** (default): MPI decides whether to buffer or not
- **Buffered mode**: User provides explicit buffer for send operations
- Useful when you want guaranteed buffering without relying on MPI's internal buffers
- Allows more predictable behavior in certain scenarios



Buffered Communication

- **Standard mode** (default): MPI decides whether to buffer or not
- **Buffered mode**: User provides explicit buffer for send operations
- Useful when you want guaranteed buffering without relying on MPI's internal buffers
- Allows more predictable behavior in certain scenarios



MPI Buffered Send


```
integer :: buffer_size, provided_size
integer, allocatable :: buffer(:)

! Determine required buffer size
call MPI_Pack_size(1, MPI_INTEGER, MPI_COMM_WORLD, buffer_size, ierr)
buffer_size = buffer_size + MPI_BSEND_OVERHEAD
allocate(buffer(buffer_size))

! Attach the buffer to MPI
call MPI_Buffer_attach(buffer, buffer_size, ierr)

! Now use MPI_Bsend instead of MPI_Send
call MPI_Bsend(data, 1, MPI_INTEGER, dest, tag, MPI_COMM_WORLD, ierr)
```

MPI Buffered Send


```
! Detach buffer when done
call MPI_Buffer_detach(buffer, buffer_size, ierr)
deallocate(buffer)
```

- `MPI_Bsend`: buffered send operation
- `MPI_Buffer_attach`: attach user-provided buffer
- `MPI_Buffer_detach`: detach buffer (must wait for all sends to complete)

Solution 3: Use Buffered Send


```
integer :: buffer_size
integer, allocatable :: buffer(:)

call MPI_Pack_size(1, MPI_INTEGER, MPI_COMM_WORLD, buffer_size, ierr)
buffer_size = buffer_size + MPI_BSEND_OVERHEAD
allocate(buffer(buffer_size))

call MPI_Buffer_attach(buffer, buffer_size, ierr)

! Both ranks can now safely use Bsend
call MPI_Bsend(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, ierr)
call MPI_Recv(data, 1, MPI_INTEGER, 1-rank, 0, MPI_COMM_WORLD, &
              status, ierr)
```

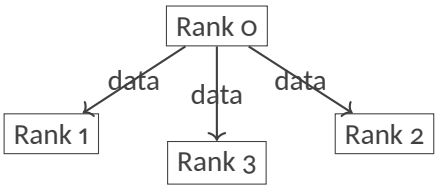
Solution 3: Use Buffered Send


```
call MPI_Buffer_detach(buffer, buffer_size, ierr)
deallocate(buffer)
```

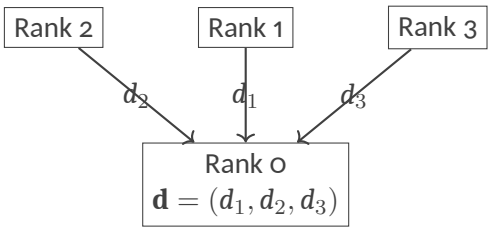
- `MPI_Bsend` copies data to user buffer immediately and returns
- Eliminates deadlock by guaranteeing buffering before receive is called
- Requires explicit buffer management overhead

Collective Communications

- Collective communication are operations that **involve all processes in a communicator**.
- Examples include broadcasting, gathering, scattering, and reducing data.
- These operations are essential for synchronizing data among processes.



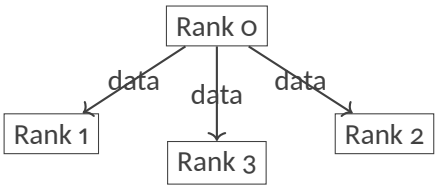
Broadcast



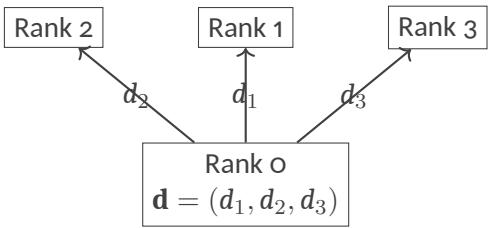
Gather

Collective Communications

- Collective communication are operations that **involve all processes in a communicator**.
- Examples include broadcasting, gathering, scattering, and reducing data.
- These operations are essential for synchronizing data among processes.



Broadcast



Scatter

Broadcast Operation

- The **broadcast** operation sends data from one process (the root) to all other processes in the communicator.
- Useful for distributing initial data or parameters.

```
program mpi_broadcast
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
```

Broadcast Operation


```
    if (rank == 0) then
        data = 100  ! Root process initializes data
    end if

    ! Broadcast data from root process (rank 0) to all processes
    call MPI_Bcast(data, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

    print *, 'Process', rank, 'received data:', data

    call MPI_Finalize(ierr)
end program mpi_broadcast
```

Broadcast Operation

The arguments of the `MPI_Bcast` function are:

- `buffer`: starting address of the buffer to be broadcasted
- `1`: number of elements to broadcast
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Bcast` function is:

```
call MPI_Bcast(buffer, counter, datatype, root, comm, ierr)
```

The value of `buffer` is significant only at the root process during the call; all other processes will receive the broadcasted value into their own `buffer` variable.

Gather Operation

- The **gather** operation collects data from all processes and sends it to a root process.
- Useful for aggregating results from multiple processes.

```
program mpi_gather
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data, recv_data(4)

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    send_data = rank + 1  ! Each process sends its rank + 1
```

Gather Operation


```
    ! Gather data at root process (rank 0)
    call MPI_Gather(send_data, 1, MPI_INTEGER, recv_data, 1, MPI_INTEGER,
    ↪  0, MPI_COMM_WORLD, ierr)

    if (rank == 0) then
        print *, 'Root process received data:', recv_data
    end if

    call MPI_Finalize(ierr)
end program mpi_gather
```

Gather Operation

The arguments of the `MPI_Gather` function are:

- `send_buffer`: starting address of the send buffer
- `1`: number of elements sent by each process
- `MPI_INTEGER`: datatype of each element
- `recv_buffer`: starting address of the receive buffer (only significant at root)
- `1`: number of elements received from each process
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Gather` function is:

```
call MPI_Gather(send_buffer, send_count, send_datatype, &
                recv_buffer, recv_count, recv_datatype, &
                root, comm, ierr)
```

Scatter Operation

- The **scatter** operation distributes data from a root process to all other processes.
- Useful for distributing chunks of data for parallel processing.

```
program mpi_scatter
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data(4), recv_data
    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    if (rank == 0) then
        send_data = (/10, 20, 30, 40/)  ! Root process initializes data
    end if
```

Scatter Operation


```
    ! Scatter data from root process (rank 0) to all processes
    call MPI_Scatter(send_data, 1, MPI_INTEGER, recv_data, 1, MPI_INTEGER,
    ↪  0, MPI_COMM_WORLD, ierr)
    print *, 'Process', rank, 'received data:', recv_data
    call MPI_Finalize(ierr)
end program mpi_scatter
```

Scatter Operation

The arguments of the `MPI_Scatter` function are:

- `send_buffer`: starting address of the send buffer (only significant at root)
- `1`: number of elements sent to each process
- `MPI_INTEGER`: datatype of each element
- `recv_buffer`: starting address of the receive buffer
- `1`: number of elements received by each process
- `MPI_INTEGER`: datatype of each element
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Scatter` function is:

```
call MPI_Scatter(send_buffer, send_count, send_datatype, &
                 recv_buffer, recv_count, recv_datatype, &
                 root, comm, ierr)
```

MPI_Gatherv Operation

- The **gatherv** operation collects variable amounts of data from each process to a root process.
- Useful when processes have different amounts of data to contribute.

```
program mpi_gatherv
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data(10), recv_data(25)
    integer :: send_count, recv_counts(4), displs(4)

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    ! Each process sends different amount of data
```

MPI_Gatherv Operation


```
    send_count = rank + 1
    send_data(1:send_count) = rank
    ! Root process specifies how much to receive from each process
    recv_counts = (/1, 2, 3, 4/)
    displs = (/0, 1, 3, 6/)  ! Displacements in receive buffer
    call MPI_Gatherv(send_data, send_count, MPI_INTEGER, &
                     recv_data, recv_counts, displs, MPI_INTEGER, &
                     0, MPI_COMM_WORLD, ierr)
    if (rank == 0) then
        print *, 'Root received data:', recv_data(1:10)
    end if

    call MPI_Finalize(ierr)
end program mpi_gatherv
```

MPI_Scatterv Operation

- The **scatterv** operation distributes variable amounts of data from a root process to all processes.
- Useful for load balancing when processes need different data sizes.

```
program mpi_scatterv
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data(10), recv_data(10)
    integer :: send_counts(4), displs(4), recv_count

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)
    if (rank == 0) then
```

MPI_Scatterv Operation


```
        send_data = (/1, 2, 3, 4, 5, 6, 7, 8, 9, 10/)
    end if
    ! Root specifies how much to send to each process
    send_counts = (/1, 2, 3, 4/)
    displs = (/0, 1, 3, 6/)  ! Displacements in send buffer
    recv_count = rank + 1
    call MPI_Scatterv(send_data, send_counts, displs, MPI_INTEGER, &
                      recv_data, recv_count, MPI_INTEGER, &
                      0, MPI_COMM_WORLD, ierr)
    print *, 'Process', rank, 'received:', recv_data(1:recv_count)
    call MPI_Finalize(ierr)
end program mpi_scatterv
```

MPI_Gatherv and MPI_Scatterv Parameters

MPI_Gatherv

- `send_buffer`: data to send
- `send_count`: amount sent by this process
- `recv_counts`: array of receive counts per process
- `displs`: array of displacements in receive buffer

MPI_Scatterv

- `send_counts`: array of send counts per process
- `displs`: array of displacements in send buffer
- `recv_buffer`: buffer to receive data
- `recv_count`: amount received by this process

Reduce Operation

- The **reduce** operation combines data from all processes and sends the result to a root process.
- Commonly used for summing values or finding maximum/minimum.

```
program mpi_reduce
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data, recv_data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    send_data = rank + 1  ! Each process sends its rank + 1
```

Reduce Operation Arguments

The arguments of the `MPI_Reduce` function are:

- `send_buffer`: starting address of the send buffer
- `1`: number of elements to reduce
- `MPI_INTEGER`: datatype of each element
- `MPI_SUM`: reduction operation (MPI_SUM, MPI_MAX, MPI_MIN, etc.)
- `0`: rank of the root process
- `MPI_COMM_WORLD`: communicator
- `ierr`: error code

The prototype of the `MPI_Reduce` function is:

```
call MPI_Reduce(send_buffer, recv_buffer, counter, datatype, op, root,
↪  comm, ierr)
```

The reduce buffer is only significant at the root process.

MPI Reduction Operations

MPI provides several built-in reduction operations:

Operation	Description	Operation	Description
`MPI_SUM`	Sum of values	`MPI_MAX`	Maximum value
`MPI_PROD`	Product of values	`MPI_MIN`	Minimum value
`MPI_MAXLOC`	Maximum value and its location (rank)	`MPI_MINLOC`	Minimum value and its location (rank)
`MPI_LAND`	Logical AND	`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR	`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical XOR	`MPI_BXOR`	Bitwise XOR

MPI Collective — Reduce

You can also define your own custom reduction operations. This is done using the

```
call MPI_Op_create(user_function, commute, op, ierr)
```

where the user-defined function has the interface:

```
subroutine user_function(invec, inoutvec, len, datatype)
    implicit none
    integer :: len
    integer :: datatype
    ! invec and inoutvec are assumed-size arrays
    ! Operation: inoutvec = invec op inoutvec
end subroutine user_function
```

The operation `op` is *assumed* to be associative; if `commute == .false.` the order of the operands must be forced in ascending process rank order, see the naive implementation example in the MPI standard document for details.

MPI Collective — Reduce

What is the output of a collective communication?

Collective features

- If the underlying operation is *not* associative, the results *cannot* be the same with different number of processes;
- If the collective is implemented *without* enforcing ordering, even *two successive runs on the same machine* will give different outputs.

Warnings

- Never test a floating point result for exact match;
- Never expect a specific value from different machine configurations;
- Always use the result of a collective to govern global application behaviour;
- Always test results for appropriate bounds.

Allreduce Operation

Combines data from all processes and distributes the result to **all** processes:

```
program mpi_allreduce
    use mpi
    implicit none
    integer :: rank, size, ierr
    integer :: send_data, recv_data

    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, size, ierr)

    send_data = rank + 1
```

Allreduce Operation


```
    ! All processes receive the result
    call MPI_Allreduce(send_data, recv_data, 1, MPI_INTEGER, &
                       MPI_SUM, MPI_COMM_WORLD, ierr)

    print *, 'Process', rank, 'received:', recv_data

    call MPI_Finalize(ierr)
end program mpi_allreduce
```

- Useful when all processes need the reduced result
- No root process specification needed
- Commonly used in iterative algorithms (e.g., convergence checks, orthogonalization coefficients)
- May be a bottleneck at scale due to synchronization requirements

Conclusions and next steps

3 Conclusions

We have covered:

- ✔ Point-to-point communication (blocking, non-blocking, buffered)
- ✔ Collective communication (broadcast, gather, scatter, reduce)

Next steps:

- Investigate the cost of communication in parallel applications
- Measuring time, and putting barriers
- Explore few advanced MPI features (derived datatypes, communicators)
- Reuse and adapt code examples for our linear algebra tasks