Parallel Programming
Lecture 10

Prof. Dr. Jonas Posner

✉ jonas.posner@cs.hs-fulda.de

Hochschule Fulda
University of Applied Sciences

Winterterm 2025/2026
⬇ E-Learning: AI5085

16th January 2026

Organization—Important Dates

- *Updated from Slides 07*
- Submission Deadline OpenMP Exam 1:
  23rd January 2026, 09:00 AM
- MPI assignment release: 23rd January 2026
  - Submission deadline: 20th February 2026
- Oral Exams: 23rd–24th February 2026 (possibly 25th as well)
  - *More details soon*

Plan for Today

- Recap MPI
  - Standard Mode Communication
  - MPI Hello World
- New:
  - MPI communication modes
    - buffered, synchronous, ready
  - Nonblocking communication
  - Completion operations and best practices

MPI Send and Receive Variants

- MPI_Send
  - MPI_BSend
  - MPI_SSend
  - MPI_RSend
  - MPI_ISend
  - MPI_ISSend
  - MPI_IBSend
  - MPI_IRSend

- MPI_Recv
  - MPI_IRecv

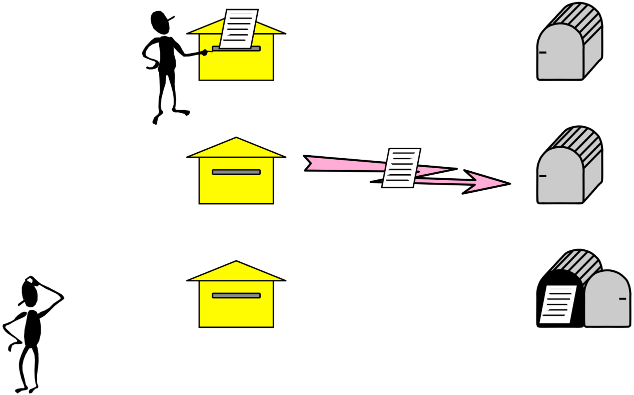
- **B** → **Buffered Mode:** Return of Send does not depend on Recv
- **S** → **Synchronous Mode:** Send returns only after Recv has started
- **R** → **Ready Mode:** Send may start only if Recv has already started
- **I** → **Immediate (non-blocking):** call returns immediately

Buffered and Synchronous Mode (1/4)

- **Advantages and disadvantages of buffering:**
  - **Pro:**
    - `MPI_Send` can complete even if `MPI_Recv` has not yet been called
  - **Con:**
    - Additional copy overhead and memory usage
    - No feedback about the receiver's progress
- **Buffered mode:** forces buffering if no matching receive has started
  - Use `MPI_Bsend(...)` (same parameters as `MPI_Send`)
  - The program must provide the buffer explicitly via `MPI_Buffer_attach()`

Buffered and Synchronous Mode (2/4)

- **Buffered = Asynchronous Send**
- Sender only knows when the message has left the local buffer
- No guarantee about receiver's state

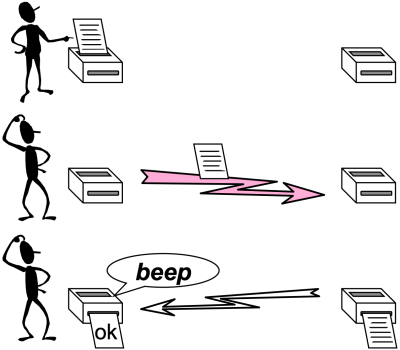


Buffered and Synchronous Mode (3/4)

- **Synchronous mode:**
  - Disables buffering
    - `MPI_Ssend(...)` returns only when the matching receive has started
  - Useful for testing whether a program is **safe** (no dependence on internal buffers)
- The **sender** always decides which mode is used

Buffered and Synchronous Mode (4/4)

- **Synchronous Send**
- Sender receives acknowledgment that the matching receive has started
- Analogous to a fax confirmation receipt



Buffered Mode—Implementation Details

- `int MPI_Buffer_attach(void *buffer, int size)`
  - Attaches a user-provided buffer for buffered sends
  - One buffer per **process**, not per send operation
- The required buffer size can be determined with:
  - ⇒ Total size = `size + MPI_BSEND_OVERHEAD`
- `int MPI_Buffer_detach(void *buffer, int *size)`
  - Detaches the buffer; returns pointer and size of the detached buffer
  - Note: for `attach`, the first argument is a pointer to the buffer; for `detach`, it is a pointer **to a pointer**
- Since MPI 4.1: `MPI_BUFFER_AUTOMATIC` enables automatic buffering
- **Live-Demo**

Communication Mode Guidelines

- **Standard Send (`MPI_Send`)**
  - Minimal transfer time
  - May block due to synchronous behavior ⇒ risks of sync. send
- **Synchronous Send (`MPI_Ssend`)**
  - Risk of deadlock
  - Risk of serialization
  - Risk of waiting ⇒ idle time
  - High latency, but best bandwidth
- **Buffered Send (`MPI_Bsend`)**
  - Low latency, but poor bandwidth
- **Ready Send (`MPI_Rsend`)**
  - **Never use** unless **absolutely sure** that `Recv` is already posted
  - Can be the fastest in ideal conditions

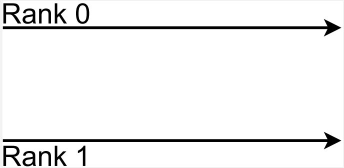
Example:
Ping Pong

Ping Pong (Blocking)

```c
...
double start_time = MPI_Wtime();
while (ping_pong_count < PING_PONG_LIMIT) {
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Send(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD);
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 }
}
if (rank == 0) {
  printf("pingpong_blocking time: %.4f seconds\n",
        MPI_Wtime() - start_time); }
...
```

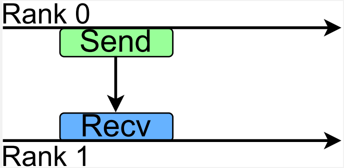
Ping Pong (Blocking)

```c
...
double start_time = MPI_Wtime();
while (ping_pong_count < PING_PONG_LIMIT) {
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Send(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD);
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 }
}
if (rank == 0) {
  printf("pingpong_blocking time: %.4f seconds\n",
        MPI_Wtime() - start_time); }
...
```



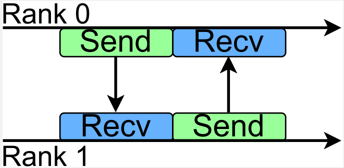
Ping Pong (Blocking)

```
...
double start_time = MPI_Wtime();
while (ping_pong_count < PING_PONG_LIMIT) {
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Send(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD);
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 }
}
if (rank == 0) {
  printf("pingpong_blocking time: %.4f seconds\n",
        MPI_Wtime() - start_time); }
...
```



Ping Pong (Blocking)

```c
...
double start_time = MPI_Wtime();
while (ping_pong_count < PING_PONG_LIMIT) {
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Send(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD);
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 }
}
if (rank == 0) {
  printf("pingpong_blocking time: %.4f seconds\n",
        MPI_Wtime() - start_time); }
...
```

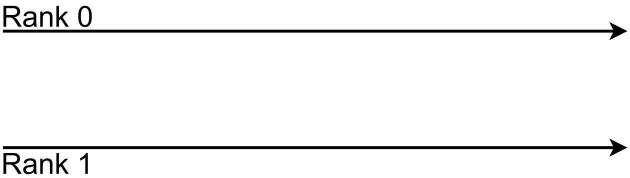


Blocking Communication

```
...
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// What happens if we do some computation here?
 }
...
```

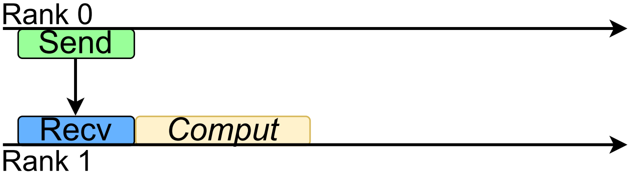
Blocking Communication

```
...
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// What happens if we do some computation here?
 }
...
```



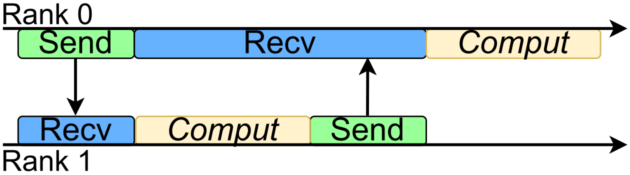
Blocking Communication

```
...
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// What happens if we do some computation here?
 }
...
```



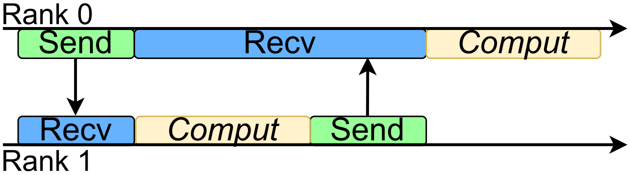
Blocking Communication

```
...
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// What happens if we do some computation here?
 }
...
```



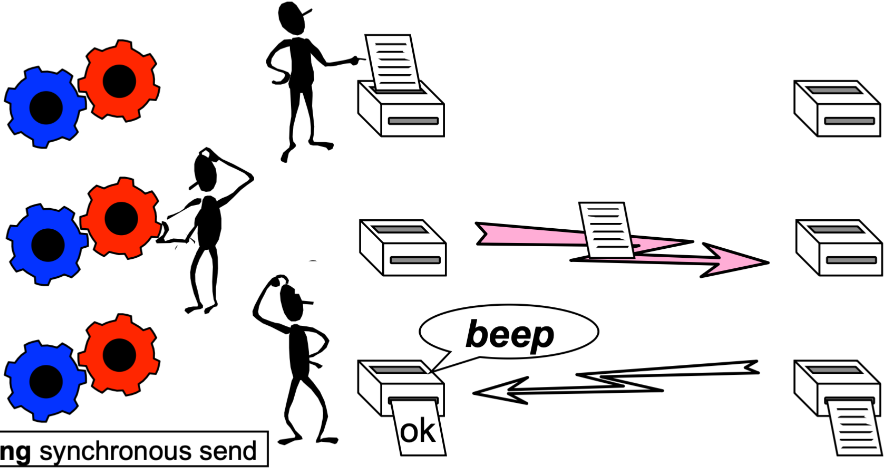
Blocking Communication

```
...
 } else {
  MPI_Recv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
          MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// What happens if we do some computation here?
 }
...
```



- Blocking operations can cause waiting times
  - The blocked process cannot perform useful work

Nonblocking Communication: Overview (1/3)



nonblocking synchronous send

Nonblocking Communication: Key Concepts (2/3)

- Enables overlap of **communication and computation**
  - (without explicitly using threads)
- To avoid idle time, deadlocks and serializations
- Sending and receiving are divided into two phases:
  - **Initiation:** the operation is started
    - Returns immediately
    - Routine name starting with `MPI_I`
    - It is local, i.e., it returns independently of any other process' activity
  - **Completion:** the operation is finished
    - I.e. the send buffer is read out, or the receive buffer is filled in

Nonblocking Communication: Buffer Semantics (3/3)

- After initiation, the sender or receiver can continue working while communication proceeds in the background
- The send buffer must not be modified until completion
  - The receive buffer is valid only after completion is confirmed
- **Initiation:** `MPI_Isend`, `MPI_Irecv`, etc.
  - Same parameters as `MPI_Send`/`MPI_Recv`, plus `MPI_Request`
- **Completion:** `MPI_Wait`, `MPI_Test`, etc.
- Sending and receiving can be independently blocking or nonblocking
- A nonblocking call immediately followed by a matching wait is equivalent to a blocking operation

Ping Pong (Non-Blocking)

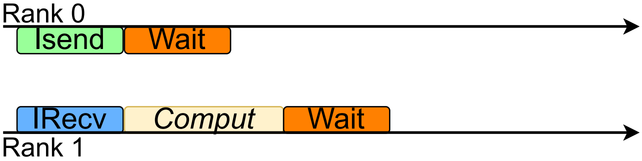
```c
...
MPI_Request request;
while (ping_pong_count < PING_PONG_LIMIT) {
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Isend(&ping_pong_count, 1, MPI_INT,
           1-rank, 0, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 } else {
  MPI_Irecv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
           MPI_COMM_WORLD, &request);

  MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
}
...
```

Ping Pong (Non-Blocking)

```
...
MPI_Request request;
while (ping_pong_count < PING_PONG_LIMIT) {
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Isend(&ping_pong_count, 1, MPI_INT,
           1-rank, 0, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 } else {
  MPI_Irecv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
           MPI_COMM_WORLD, &request);
// What happens if we do some computation here?
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
}
...
```

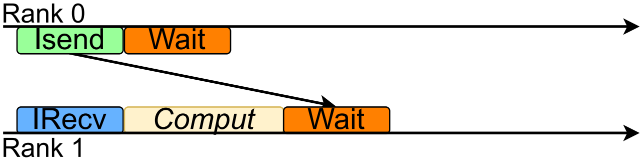
Ping Pong (Non-Blocking)

```
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Isend(&ping_pong_count, 1, MPI_INT,
            1-rank, 0, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 } else {
  MPI_Irecv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
            MPI_COMM_WORLD, &request);
(*// What happens if we do some computation here?*)
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
```



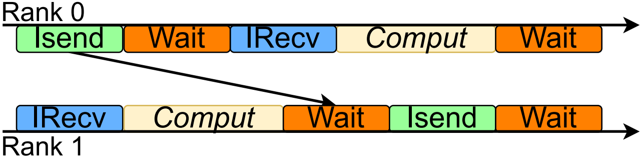
Ping Pong (Non-Blocking)

```
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Isend(&ping_pong_count, 1, MPI_INT,
           1-rank, 0, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 } else {
  MPI_Irecv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
           MPI_COMM_WORLD, &request);
(*// What happens if we do some computation here?*)
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
```



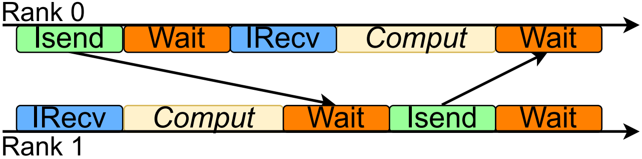
Ping Pong (Non-Blocking)

```
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Isend(&ping_pong_count, 1, MPI_INT,
           1-rank, 0, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 } else {
  MPI_Irecv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
           MPI_COMM_WORLD, &request);
(*// What happens if we do some computation here?*)
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
```



Ping Pong (Non-Blocking)

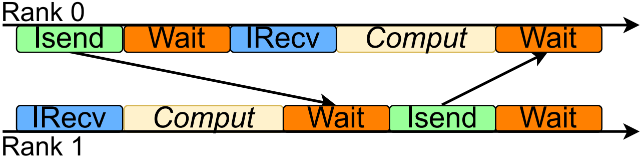
```
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Isend(&ping_pong_count, 1, MPI_INT,
            1-rank, 0, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 } else {
  MPI_Irecv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
            MPI_COMM_WORLD, &request);
(*// What happens if we do some computation here?*)
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
```



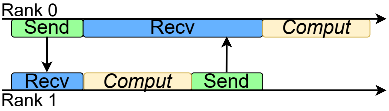
Ping Pong (Non-Blocking)

```
 if (rank == ping_pong_count % 2) {
  ping_pong_count++;
  MPI_Isend(&ping_pong_count, 1, MPI_INT,
           1-rank, 0, MPI_COMM_WORLD, &request);
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 } else {
  MPI_Irecv(&ping_pong_count, 1, MPI_INT, 1-rank, 0,
           MPI_COMM_WORLD, &request);
(*// What happens if we do some computation here?*)
  MPI_Wait(&request, MPI_STATUS_IGNORE);
 }
```

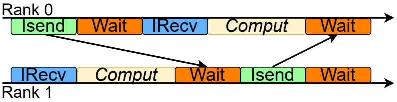
→ **Overlapping communication & computation!**



Experiment: Ping Pong—Blocking vs. Non-Blocking

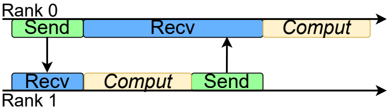


vs.

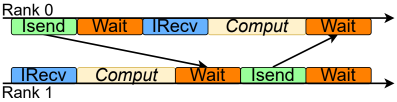


- `PING_PONG_LIMIT` = 100
- Computation = 10 ms
- Two processes

Experiment: Ping Pong—Blocking vs. Non-Blocking



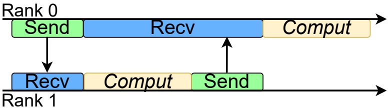
vs.



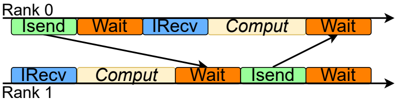
- `PING_PONG_LIMIT` = 100
- Computation = 10 ms
- Two processes

Variant	**Overlap?**	**Measured Time**	**Speedup**
Blocking	✗ No	≈ 1.00 sec	✗ 1
Non-Blocking	✓ Yes	≈ 0.50 sec	✓ 2

Experiment: Ping Pong—Blocking vs. Non-Blocking



VS.



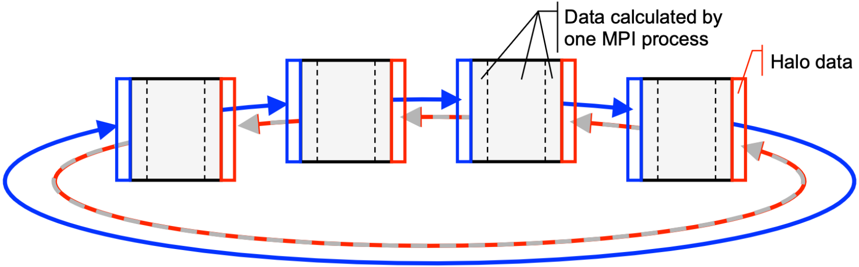
- PING_PONG_LIMIT = 100
- Computation = 10 ms
- Two processes

Variant	**Overlap?**	**Measured Time**	**Speedup**
Blocking	✗ No	≈ 1.00 sec	✗ 1
Non-Blocking	✓ Yes	≈ 0.50 sec	✓ 2

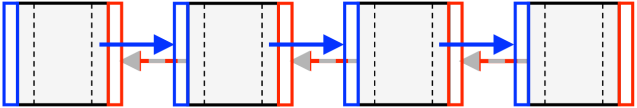
- **How to “see” this in real programs?**
  → *Performance Analysis Tools* → *Lecture 12!*

Halo Exchange: Another Motivating Example

- With cyclic boundary conditions:



- Non-cyclic:



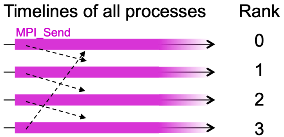
- Focus on the blue direction (left to right, clockwise)

Blocking Routines

- Blocking operations carry inherent risks:
  - Deadlocks when processes wait circularly
  - Serialization reducing parallel efficiency

For cyclic boundary:

```
MPI_Send(..., right_rank, ...)
MPI_Recv(  ..., left_rank,  ...)
```

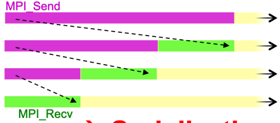


→ **Deadlock**

(If the MPI library chooses the synchronous protocol, i.e. MPI_Send waits until MPI_Recv is called)

For non-cyclic boundary:

```
if (myrank < size-1)
  MPI_Send(..., left, ...);
if (myrank > 0)
  MPI_Recv(  ..., right,  ...);
```

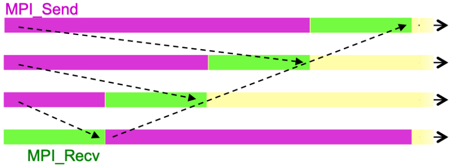


→ **Serialization**

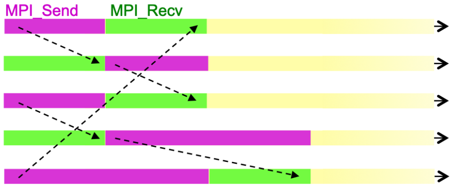
(If the MPI library chooses the synchronous protocol)

Cyclic Communication—Other bad Ideas

```
if (myrank < size-1) {
   MPI_Send(…, left, …);
   MPI_Recv(  …, right,  …);
} else {
   MPI_Recv(  …, right,  …);
   MPI_Send(…, left, …);
}
```



```
if (myrank%2 == 0) {
   MPI_Send(…, left, …);
   MPI_Recv(  …, right,  …);
} else {
   MPI_Recv(  …, right,  …);
   MPI_Send(…, left, …);
}
```

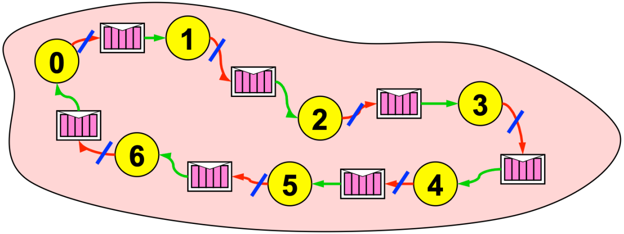


→ **Serialization** ↯

(If the MPI library chooses
the synchronous protocol)

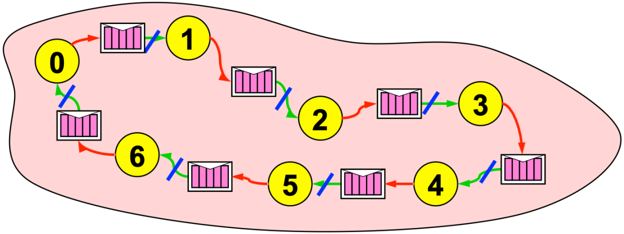
Nonblocking Send—Ring Pattern

- Initiate nonblocking send
  - → Initiate nonblocking send to right neighbor
- Perform useful work:
  - → Receive message from left neighbor
- Message transfer can complete in background
- Wait for nonblocking send to complete

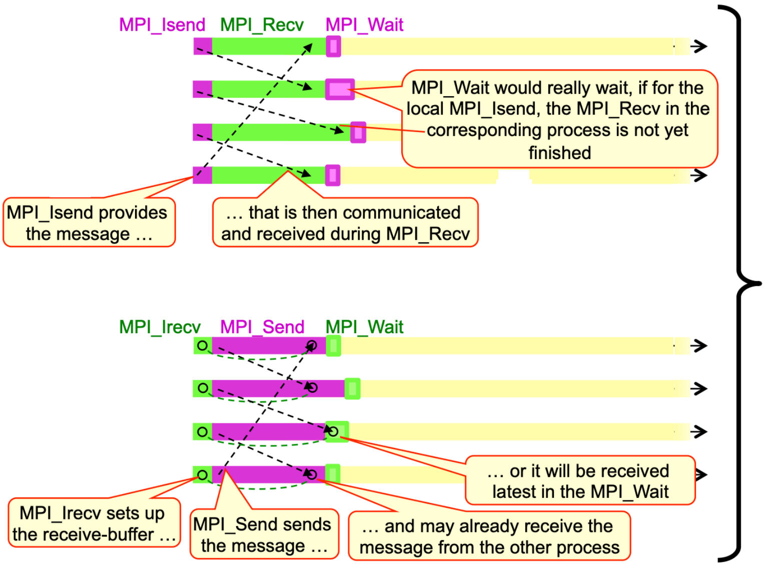


Nonblocking Receive—Ring Pattern

- Initiate nonblocking receive
  - → Initiate nonblocking receive from left neighbor
- Perform useful work:
  - → Send message to right neighbor
- Message transfer can complete in background
- Wait for nonblocking receive to complete



Timeline Comparison: Both Approaches



MPI_Wait would really wait, if for the local MPI_Isend, the MPI_Recv in the corresponding process is not yet finished

MPI_Isend provides the message …

… that is then communicated and received during MPI_Recv

MPI_Irecv sets up the receive-buffer …

MPI_Send sends the message …

… and may already receive the message from the other process

… or it will be received latest in the MPI_Wait

No Serialization, no deadlock

Nonblocking Communication: Practical Notes

- The `request` is a handle to an internal MPI data structure storing communication state
  - Declared as: `MPI_Request request`
- `MPI_Isend`/`MPI_Irecv` execute independently of the completion call
  - Calling `MPI_Wait` or `MPI_Test` does **not** speed up the communication
- On the sender side, there are also: `MPI_Ibsend`, `MPI_Issend`
  - Same parameters as the blocking variants
- `MPI_Irecv` is frequently used:
  - Often allows faster completion of the matching send, since no buffering is required

Nonblocking Synchronous Send

```
MPI_Issend(buf, count, datatype, dest, tag, comm,
&request_handle);
```

```
MPI_Wait(&request_handle, &status);
```

- The buf must **not be modified** between the calls to MPI_Issend and MPI_Wait
- Using Issend immediately followed by Wait is equivalent to a blocking MPI_Ssend
- No information is returned in status (since send operations do not produce a status)

Nonblocking Receive

```
MPI_Irecv(buf, count, datatype, source, tag, comm,
&request_handle);
```

```
MPI_Wait(&request_handle, &status);
```

- The `buf` must **not be accessed or modified** between `MPI_Irecv` and `MPI_Wait`

- The message **status** is returned by `MPI_Wait`

Completion Operations

- Nonblocking communication must always be completed to release associated resources
- `MPI_Wait(&request_handle, &status)`
  - Blocks until the operation has completed
- `MPI_Test(&request_handle, &flag, &status)`
  - Returns immediately
  - `*flag = 1` if the operation is complete, 0 otherwise
  - `status` contains information only on the receiver for `flag = 1`
- `MPI_Request_free`
  - Frees the resources; **does not** cancel the operation
- `MPI_Waitall`, `MPI_Testany`, `MPI_Waitsome`, etc.
  - Operate on arrays of requests
  - May use `MPI_STATUSES_IGNORE` if status output is not needed

Additional Point-to-Point Operations

- MPI_Sendrecv and MPI_Sendrecv_replace
  - System combines send and receive as efficiently as possible
  - With replace, send and receive share the same buffer
- Probing for incoming messages without receiving them:
  - Functions: MPI_Probe, MPI_Iprobe, MPI_Request_get_status
  - Use case: determine message size before allocating receive buffer
- Persistent communication for frequent partners:
  - Functions: MPI_Send_init, MPI_Recv_init, MPI_Start, MPI_Startall
  - Reduces overhead for repeated communication patterns
- Cancelling requests: MPI_Cancel(...)
  - Rarely needed; use with caution

Performance Considerations

- Which is the **fastest neighbor communication?**
  - MPI_Irecv + MPI_Send
  - MPI_Irecv + MPI_Isend
  - MPI_Isend + MPI_Recv
  - MPI_Isend + MPI_Irecv
  - MPI_Sendrecv

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming,
but the efficiency of MPI applications is not portable!

Use Cases for Nonblocking Operations

- **Avoid serialization and deadlocks**
  - Enables overlapping of multiple communication operations
- **Achieve true overlap between:**
  - Multiple concurrent communication operations
  - Communication and computation
- **Key insight:** nonblocking operations decouple initiation from completion, providing greater flexibility in program design