Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 11:

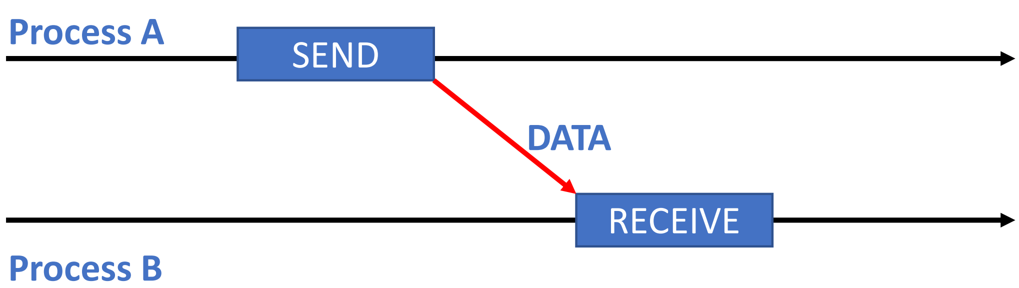
- Distributed-Memory Programming with MPI
  - Blocking Point-to-Point Communication
  - Non-Blocking Point-to-Point Communication

MPI: Point-to-Point Communication

The goal is to enable (explicit) communication between processes that do not share a memory address space.

Explicit message passing requires:

- Send and Receive operations
- Addresses of both sender and receiver
- Specification of what to be sent and received



MPI: Send Operation

Sending a message:

- ***data*** - the location of the send buffer
- ***count*** - the number of elements to be sent
- ***type*** - the handle of the **MPI_Datatype** of the buffer content
- ***dest*** - the rank of the receiver
- ***tag*** - an additional identifier of the message ranging from **0** to **MPI_TAG_UB**
- ***comm*** - the communication context

To whom?

C

```
MPI_Send( void * data , int count , MPI_Datatype type , int dest, int tag , MPI_COMM comm )
```

What to send?

MPI: Receive Operation

Receiving a message:

- ***data*** – the location of the receive buffer
- ***count*** – the number of elements to be received
- ***type*** – the handle of **MPI_Datatype** of the buffer content
- ***src*** – the rank of the sender or **MPI_ANY_SOURCE** (wildcard)
- **tag** – an additional identifier of the message or **MPI_ANY_TAG** (wildcard)
- **comm** – communication context
- ***status*** – the status of the receive operation or **MPI_STATUS_IGNORE**

From whom?

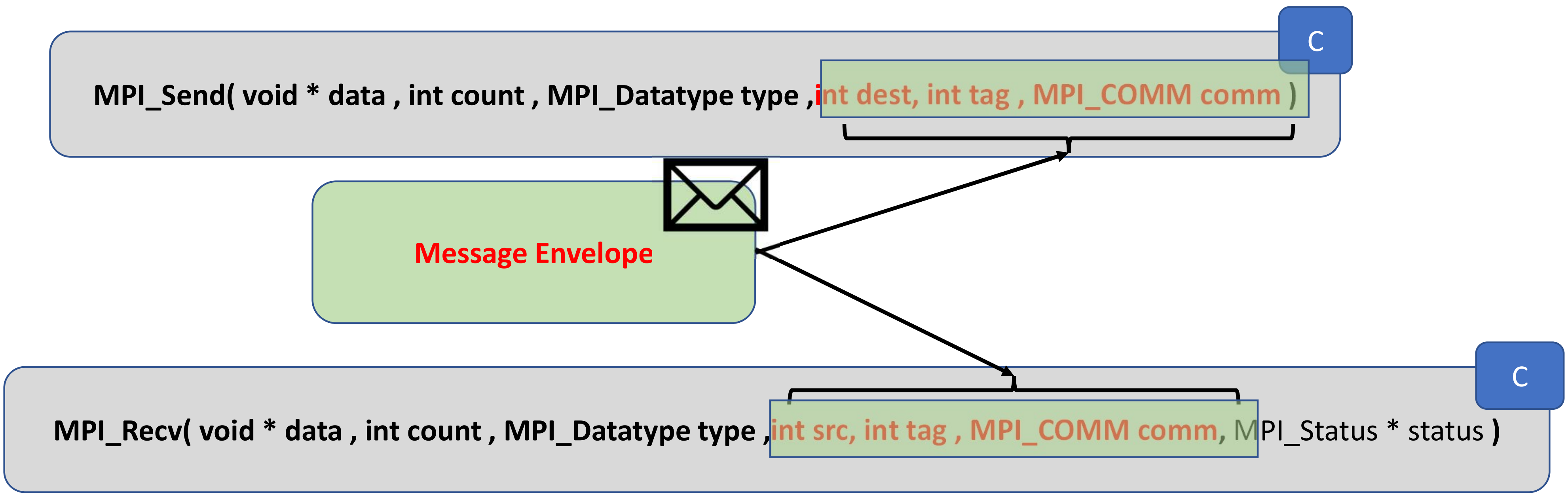
C

MPI_Recv(void * data , int count , MPI_Datatype type , int src, int tag , MPI_COMM comm, MPI_Status * status)

What to receive?

MPI: Message Envelope and Matching

The message matching mechanism is done using **the message envelope** .



MPI: Message Envelope and Matching

	Sender	Receiver
Source	Implicit	Explicit / Wild Card (MPI_ANY_SOURCE)
Destination	Explicit	Implicit
Tag	Explicit	Explicit / Wild Card (MPI_ANY_TAG)
Communicator	Explicit	Explicit

MPI: Message Envelope and Matching

The correctness of receiving a message also depends on the **data type**.

- The **type signature** must match – The MPI runtime may not check this and this can lead to undefined behavior.

One send operation is matched only with one received operation:

- Messages do not aggregate – **no single receive for multiple sends**
- A message is not separate into multiple messages - **no multiple receives for a single send**

C

MPI_Send(void * data , int count , MPI_Datatype type , int dest, int tag , MPI_COMM comm)

C

MPI_Recv(void * data , int count , MPI_Datatype type , int src, int tag , MPI_COMM comm, MPI_Status * status)

MPI: Message Size

The receive buffer must be sufficiently large to accommodate the entire message.

- Send count $\leq$ Receive count $\rightarrow$ **OK**
  - The receiver can check the actual message size using the status object
- Send count $>$ Receive count $\rightarrow$ **Error**
  - Message Truncation Error

A **message size query** can be done using

C

```
MPI_Get_count( MPI_Status * status, MPI_Datatype type, int * count )
```

- If the amount of data specified in the status object is not an exact multiple of the size of the given data type, the count variable is set to **MPI_UNDEFINED**.

MPI: Status Object

The status object contains information about the received message.

- There may be additional implementation-specific attributes.

C

```
typedef struct _MPI_Status
{
        int MPI_SOURCE ,   //message source rank
        int MPI_TAG,        //message tag
        int MPI_ERROR      //received status code
} MPI_Status ;
```

MPI: Message Availability

The availability of messages can be checked using

C

```
MPI_ Probe( int src, int tag, MPI_COMM comm, MPI_Status * status )
```

- The call blocks until there is a matching message.
- A message is not received.
  - A separate call to **MPI_Recv** is required to receive the message.
- The message envelope and size are stored in the status object.

Caveat:

- Never use with multithreading.

MPI: Message Availability

Any message in a given communicator can be checked using

C

```
MPI_ Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status )
```

The receiver must use specific values from the status object to receive the enquired message.

C

```
MPI_Status status;
…
MPI_Probe( MPI_ANY_SOURCE, 0 , MPI_COMM_WORLD, &status );
…//the receiver can now allocate a receive buffer based on the message size
…
MPI_Recv( buffer, size, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORD, &status );
```

Use the envelope data from the status object.

MPI: Operation’s Completion

An MPI operation completes locally as soon as the associated buffer is no longer in use by the MPI runtime environment and is thus free for reuse.

A send operation completes:

- as soon as the message is constructed

AND

- placed completely onto the network **OR** buffered completely by the MPI runtime, the OS or the network layer

A receive operation completes:

- The entire message has arrived and has been placed into the buffer.

Blocking MPI calls only return once the corresponding operations has completed.

- **MPI_Send** and **MPI_Recv** are blocking.

MPI: Point-to-Point Send Modes

	Call	Semantics
Buffered	**MPI_Bsend**	The MPI runtime uses **an extra buffer provided by the user**, whereby the runtime copies the data from the message buffer to this buffer and allows the call to return. The actual message transfer may happen at a later point in time after the call returns.
Synchronous	**MPI_Ssend**	The call explicitly waits until the receiver starts the receiving process.
Standard	**MPI_Send**	The call may follow the buffered semantics (using an **internal system buffer** provided by the MPI runtime)
Ready	**MPI_Rsend**	The sender assumes that the receiver must have posted a matching receive operation at the other end.

MPI: Point-to-Point Send Modes

	Call	Semantics
Buffered	**MPI_Bsend**	The MPI runtime uses **an extra buffer provided by the user**, whereby the runtime copies the data from the message buffer to this buffer and allows the call to return. The actual message transfer may happen at a later point in time after the call returns.
Synchronous	**MPI_Ssend**	The call explicitly waits until the receiver starts the receiving process.
Standard	**MPI_Send**	The call may follow the buffered semantics (using an **internal system buffer** provided by the MPI runtime)
Ready	**MPI_Rsend**	The sender assumes that the receiver must have posted a matching receive operation at the other end.

Avoid using this mode.
It is hard to make sure that the receiving partner has already posted a matching receive call.

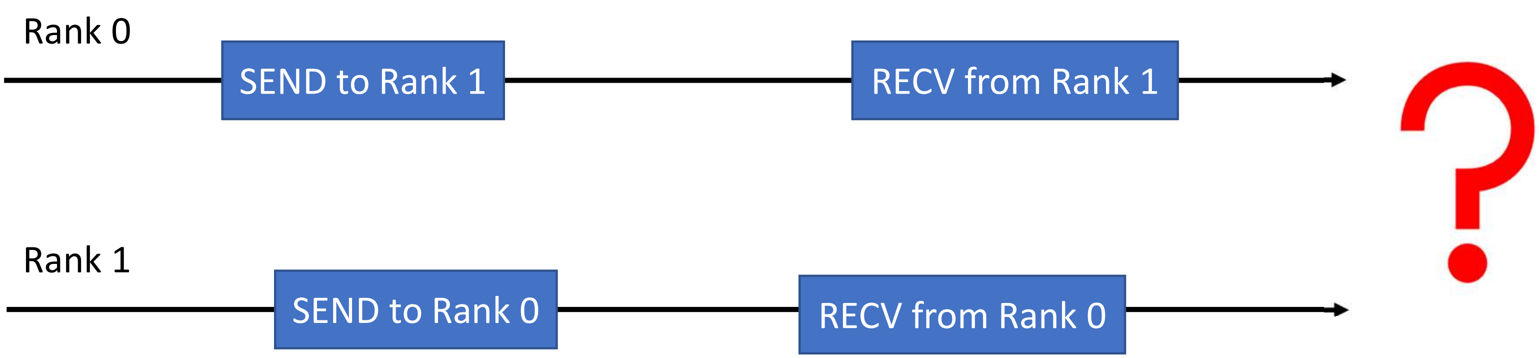
MPI: Deadlock

Both **MPI_Send** and **MPI_Recv** are blocking operations.

We already know that the standard-mode send can have **two implementation-dependent modes** of operation:

- The **buffered mode** where the call can immediately return
- The **synchronous mode** where the call waits for the receiver to start receiving the message

Never rely on such implementation-dependent behavior !!!



MPI: Deadlock

Both **MPI_Send** and **MPI_Recv** are blocking operations.

We already know that the standard-mode send can have **two implementation-dependent modes** of operation:

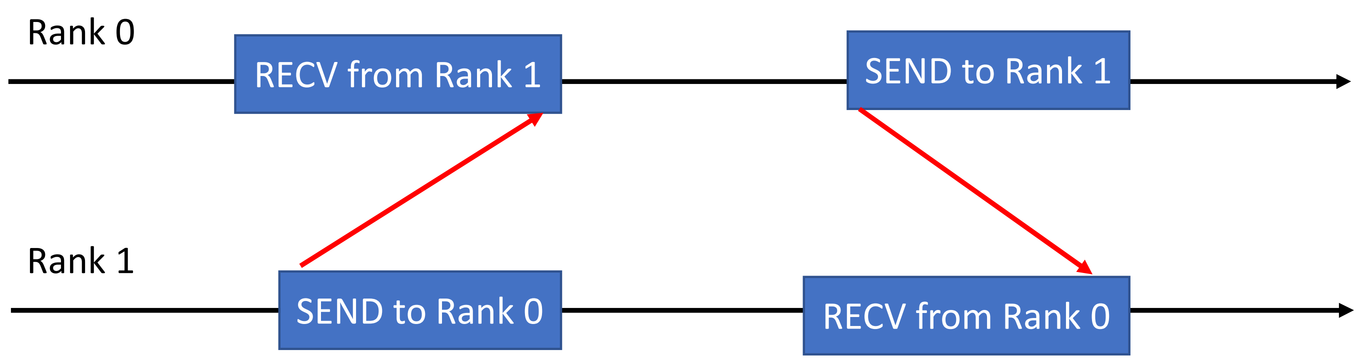
- The **buffered mode** where the call can immediately return
- The **synchronous mode** where the call waits for the receiver to start receiving the message

Never rely on such implementation-dependent behavior !!!

MPI: Deadlock

The example on the previous slide illustrates a scenario where a deadlock may ensue when both **MPI_Send** on both ranks follow the **synchronous mode** of operation, instead of the **buffered mode** of operation.

A solution for such a scenario can be done by swapping the send and the receive on either one of the two ranks.



MPI: Combined Send and Receive

C

```
MPI_ Sendrecv( void * sendbuff, int sendcount, MPI_Datatype sendtype, int dest , int sendtag,
               void * recvbuff, int recvcount, MPI_Datatype recvtype, int src, int recvtag,
               MPI_COMM comm, MPI_Status * status )
```

- This combined send-receive call sends one message and receive one message (in any order) without deadlocking.
- The send buffer and the receive buffer must not overlap.

MPI: Combined Send and Receive

C

MPI_ Sendrecv_replace(void * buff, int count, MPI_Datatype datatype, int dest , int sendtag, , int src, int recvtag, MPI_COMM comm, MPI_Status * status)

- This combined send-receive call sends one message and receive one message (in any order) without deadlocking.
- The send buffer and the receive buffer **use the same memory location**, element count and data type.
- It may be slower than **MPI_Sendrecv**.

MPI: Message Ordering

Order is preserved for point-to-point operations

- **in a given communicator**
- **between any pair of processes**

A receive (or a probe) returns the earliest matching message.

Order is **NOT** guaranteed for

- **Messages send from different communicators**
- **Messages arriving from different senders**
- **Messages sent from different threads**

MPI: Non-blocking Calls

- A **non-blocking** call returns before the associated operation has completed.
- A separate call is needed to complete the operation.
- A **request handle** **MPI_Request** is used to track the completion status of the operation.
- A non-blocking call can be used to **overlap computation with communication** as well as to **prevent deadlock**.

MPI: Non-blocking Calls

In MPI, most non-blocking calls have **an I-prefix**, such as **MPI_Isend**, **MPI_Irecv** etc.

- Some non-blocking calls don't, however.

Think of a non-blocking call as **the initiation of the operation** associated with the call.

- The actual operation may occur at a later point in time.
- A **request handle** is initialized on success, which can be used to track the progress of the operation.

C

MPI_Isend(void * buff, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request * request)

C

MPI_Irecv(void * buff, int count, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Request * request)

MPI: Waiting for Request Completion

C

MPI_Wait(MPI_Request * request, MPI_Status * status)

- The call **blocks**, waiting for **a single request** to complete
- When the operation has completed, the status object holds the information of the message.
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

MPI: Waiting for Request Completion

C

```
MPI_Waitany( int count, int * index, MPI_Request  array_of_requests [], MPI_Status * status )
```

- The call **blocks**, waiting for **any one of the requests** to complete
- When the operation has completed, ***index*** holds the array index pointing to the request handle corresponding to the completed operation, and the status object holds the information of the message.
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

MPI: Waiting for Request Completion

C

MPI_Waitsome(int incount, int * index, MPI_Request * [] request,
int * outcount, int array_of_indices [], MPI_Status array_of_status [])

- The call **blocks**, waiting for **some of the requests (not necessarily all)** to complete
- If some of the awaited operations have completed,
  - ***outcounts*** holds the number of completed operations.
  - ***array_of_indices*** holds the array indices of the request handles corresponding to the completed operations.
  - ***array_of_status*** holds the status objects of the completed operations.
- If none of the request handles corresponds to an active non-blocking operation, **outcount** is set to **MPI_UNDEFINED**.
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

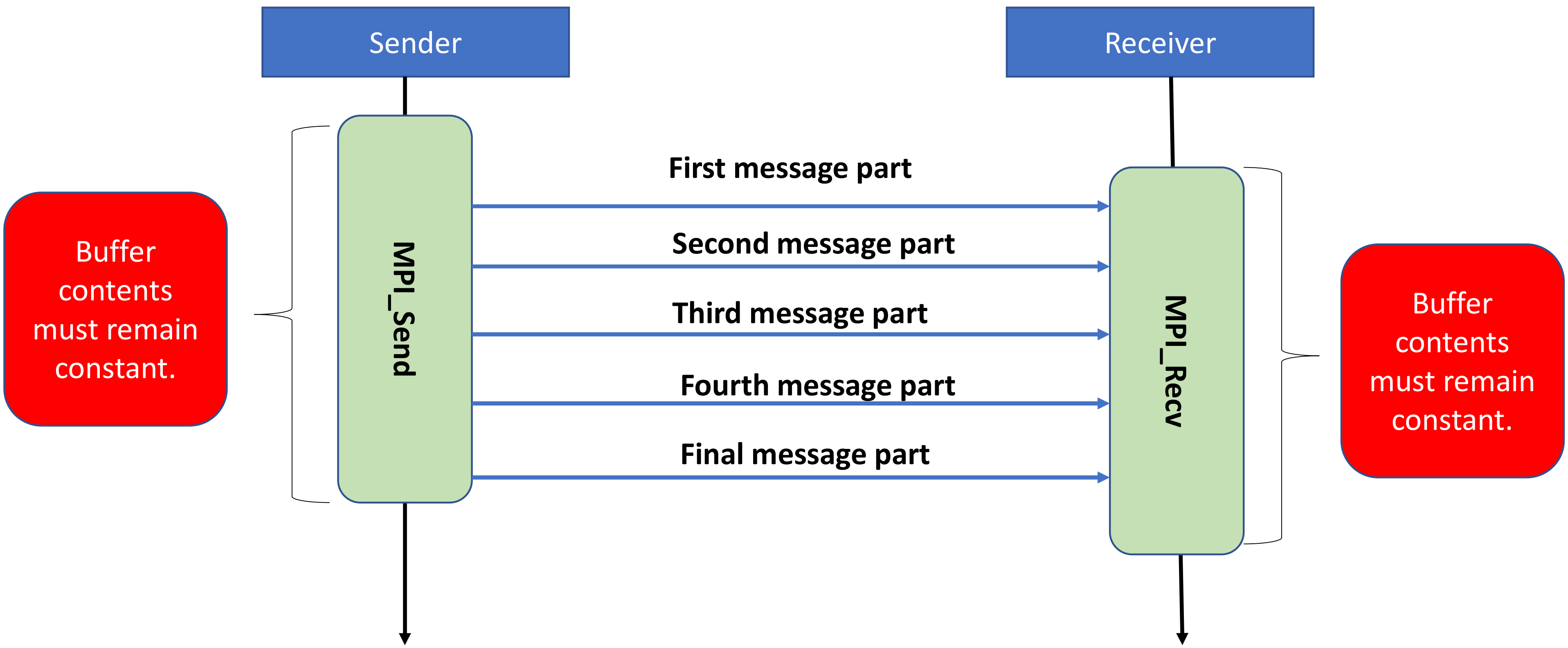
MPI: Waiting for Request Completion

C

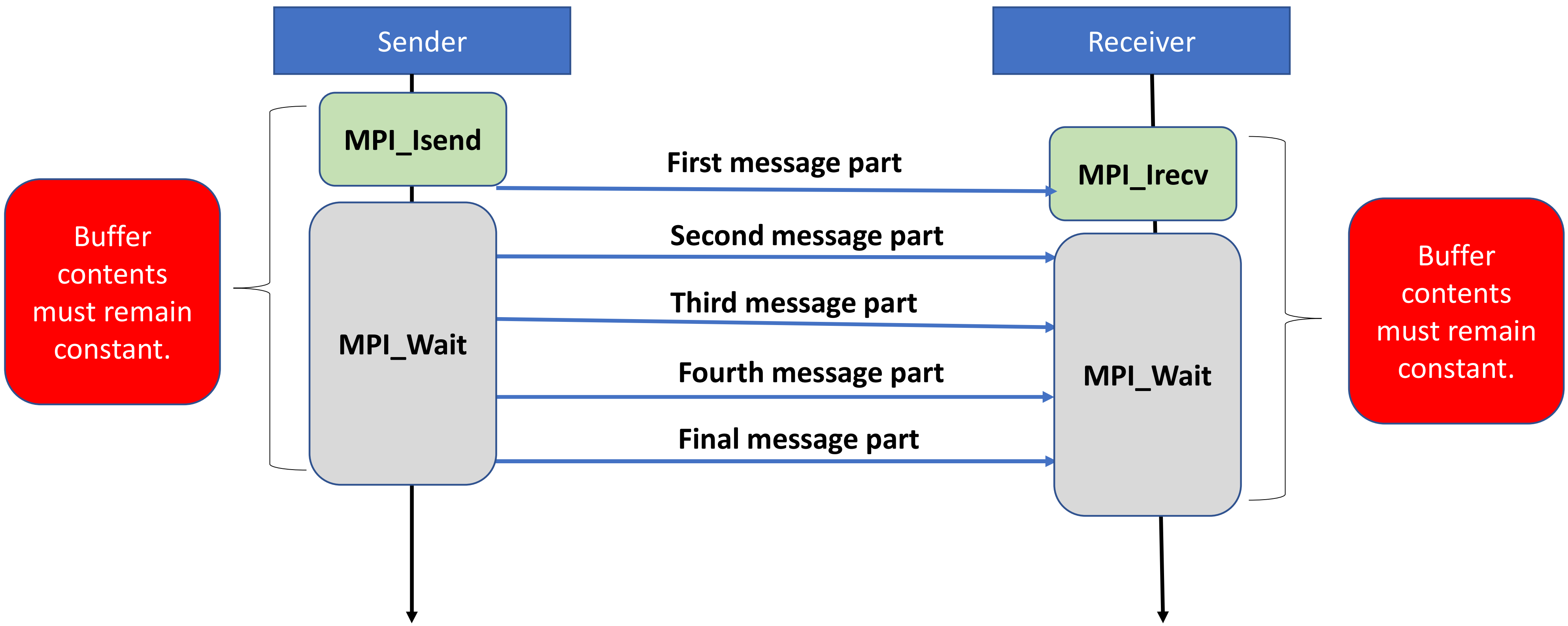
MPI_Waitall(int count, MPI_Request array_of_requests [], MPI_Status array_of_status [])

- The call **blocks**, waiting for **all of the requests** to complete
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

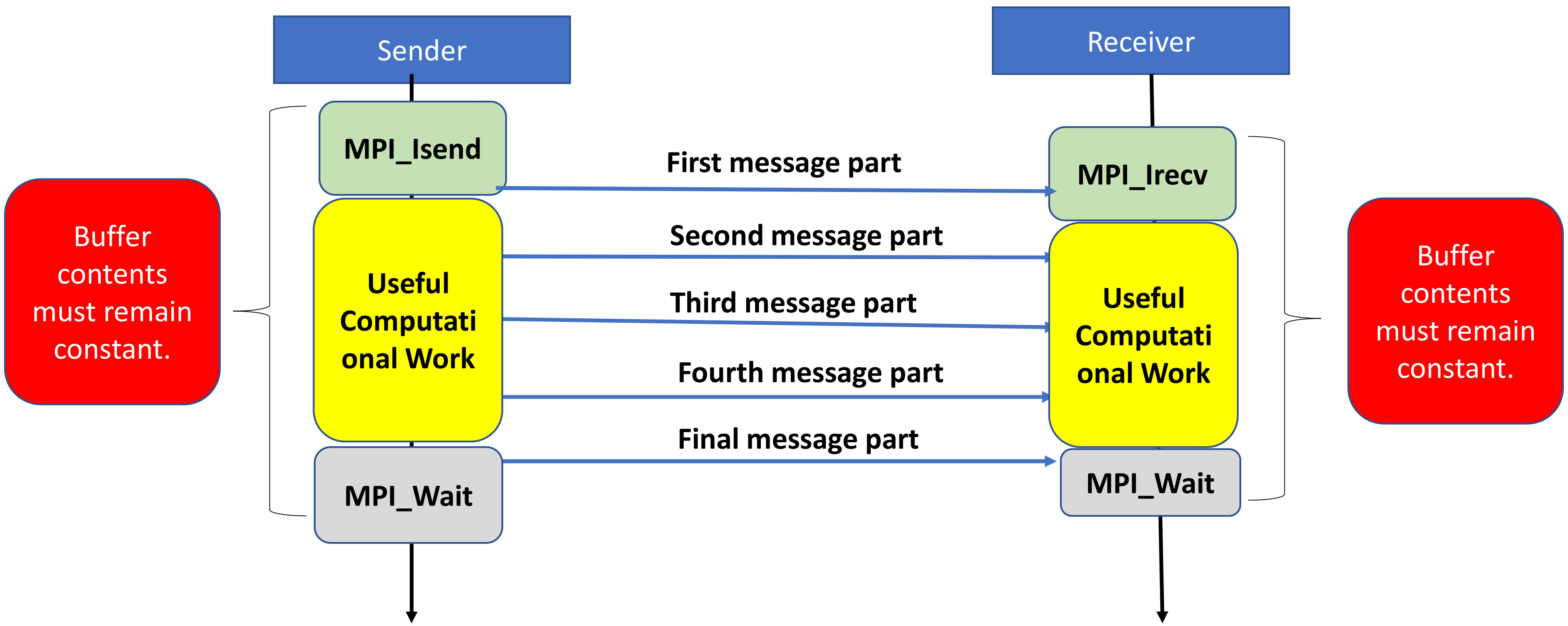
MPI: Blocking Send and Receive



MPI: Computation-Communication Overlap?



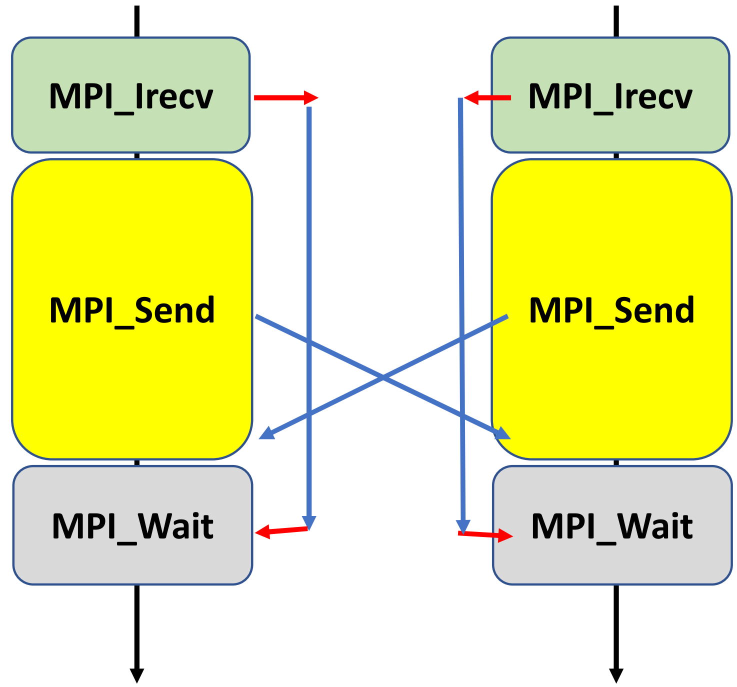
MPI: Computation-Communication Overlap ✓



MPI: Deadlock Prevention

Non-blocking calls can be used to prevent deadlocks **in symmetric code**.

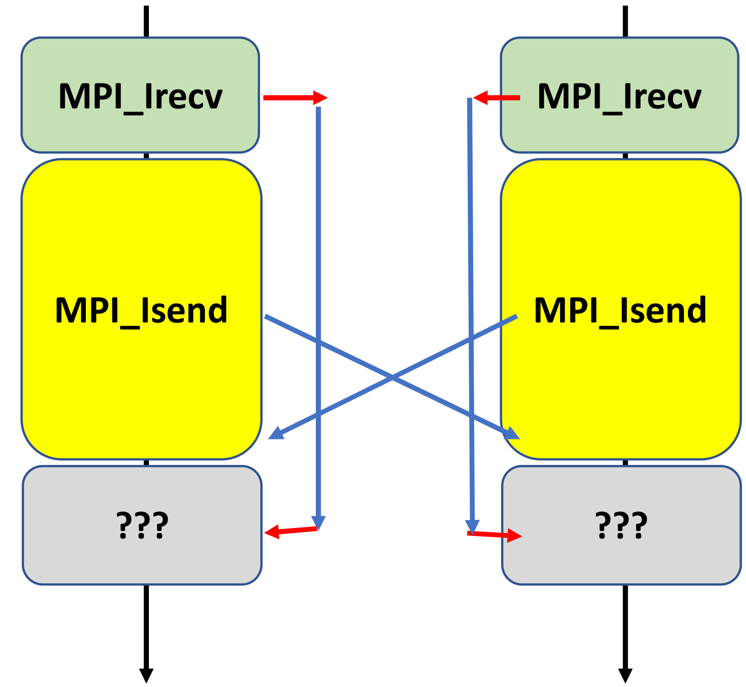
This is how **MPI_Sendrecv** is implemented.



MPI: Deadlock Prevention

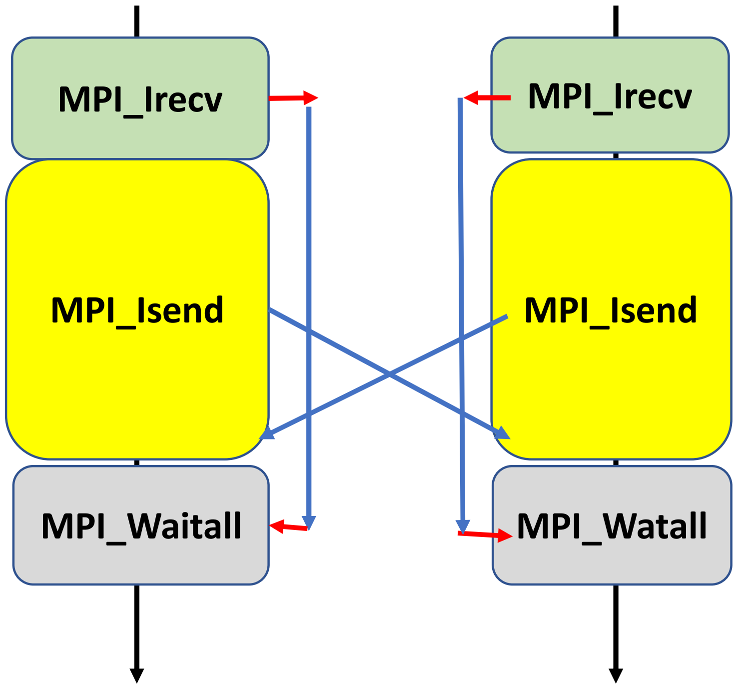
Non-blocking calls can be used to prevent deadlocks **in** **symmetric code**.

What operations are needed here?



MPI: Deadlock Prevention

Non-blocking calls can be used to prevent deadlocks **in** **symmetric code**.



MPI: Test for Request Completion

C

MPI_Test(MPI_Request * request, int * flag, MPI_Status * status)

- The call is **non-blocking** and returns, setting ***flag*** to indicate whether the request has completed.
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

MPI: Test for Request Completion

C

MPI_Testany(int count, MPI_Request array_of_requests [], int * index, int * flag,   MPI_Status * status)

- The call is **non-blocking**
  - testing for the completion of **any one of the requests**
  - setting ***flag*** to indicate whether the request has completed
  - setting ***index*** to point to the completed request in the array
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

MPI: Test for Request Completion

C

```
MPI_Testsome( int incount, MPI_Request array_of_requests [], int * outcount,
int array_of_indices [],   MPI_Status array_of_status [] )
```

- The call is **non-blocking**
  - testing for the completion of **some of the requests** (not necessarily all)
  - setting ***outcome*** to hold the number of completed requests
  - setting ***array_of_indices*** to hold the indices of the completed requests
  - setting ***array_of_status*** to hold the status objects of the completed requests
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

MPI: Test for Request Completion

C

MPI_Testall(int incount, MPI_Request array_of_requests [], int * flag, ,MPI_Status array_of_status [])

- The call is **non-blocking**
  - testing for the completion of **all of the requests**
  - setting ***flag*** to indicate whether all the requests have completed
  - setting ***array_of_status*** to hold the status objects of the completed requests
- Use **MPI_STATUS_IGNORE** to ignore the status of the message.

MPI: Request Completion

When testing or waiting for the completion of a request/any/some/all,

- On success, the request handle(s) are set to **MPI_REQUEST_NULL**.

When called with **null requests** (**MPI_REQUEST_NULL**):

- MPI_Wait returns immediately with an empty status.
- MPI_Test sets ***flag*** to **true** and returns an empty status.

References

[1] *William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press.*

[2] *Marc-Andre Hermanns. 2021. MPI in Small Bites. PPCES 2021.*