RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Introduction to High Performance Computing

Lecture 06 – Messaging

Holger Fröning
Institut für Technische Informatik
Universität Heidelberg

Review and Background

- Distributed exclusive address spaces
  - A pointer valid for one process is not valid for another process
  - Hardware running each process can be completely different
    - 32bit vs. 64bit
    - X86 vs. IA64/ARM/SPARC/etc
    - Big-endian vs. Little-endian
- MPI has to ensure that appropriate conversion takes place
  - Such heterogenous systems rather seldom today…
  - GPUs are different
- Finally, in MPI, it‘s all about copying data…

Review and Background

- Identifying copies = Tag matching
- Applicable for send/receive messaging scheme
  - Opposed to the Put/Get model

- Tag matching
  - If receive is pre-posted, search list of posted receives
  - If receive is not posted yet, copy message to unexpected queue and upon receive search this queue

- In any case, tag matching = O(N), N number of buffer entries
  - One of the biggest sources of overhead in message passing
  - Besides progress (see later)

Review and Background

- Up to now: Blocking/non-blocking send/receive
    - Returning from the function call guarantees that corresponding operation is complete
    - I.e., message has been sent or received (what about both?)
    - Real life examples
- Messaging in MPI is a little bit more complicated than you might expect
    - Example: P0 sends 16 data words to P2
        - P0 has to tell MPI: destination, count, type, tag (label)
        - P2 has to tell MPI: source, count, type, tag
        - P0 has to use a certain *communication mode* or *send mode*
- What is guaranteed after returning?
- Who is copying from where to where?

Terminology

- Latency
  - Overhead associated with sending a zero-size message
  - Between at least two MPI processes
  - Hard- and software components, ratio highly implementation dependent
  - HPC: usually measured in micro-seconds
- Bandwidth
  - Data rate which can be transmitted between two MPI processes
  - Hard- and software components, ratio highly implementation dependent
  - Usually measured in (mega-)bytes per second (MB/s)
- Synchronous/asynchronous communication
  - A completing synchronous call guarantees that data has been successfully received by receiving process
  - An asynchronous call can return anytime, without any guarantees about the state of the receiving process

Terminology

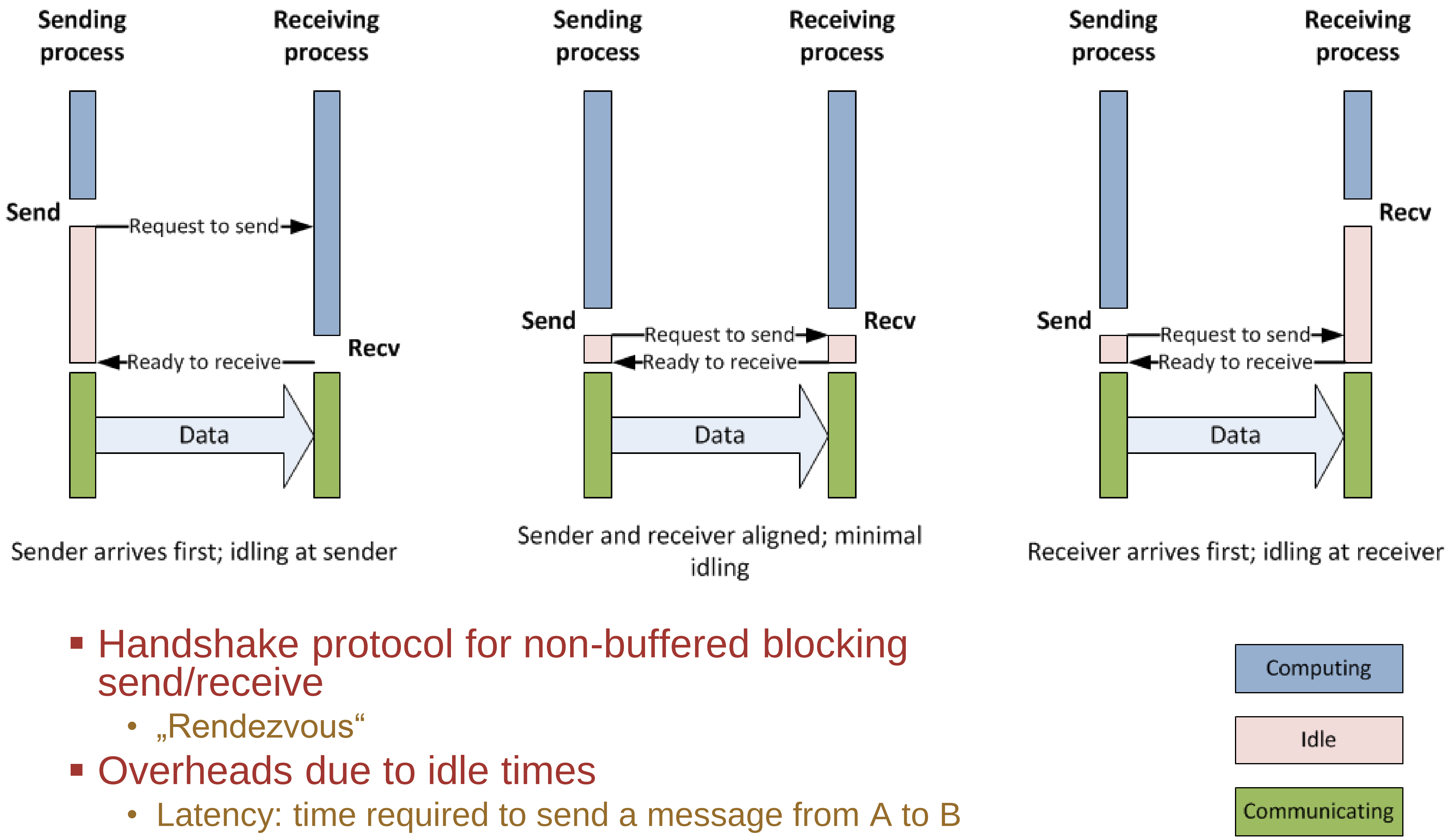
- User or application buffer
  - User-level address space that holds data to be sent or received
  - Passed as pointer to the MPI send/receive calls
  - Resides in virtual user-level address space
- System buffer
  - System address space for temporarily storing messages
  - Not visible to user/programmer
  - Key for asynchronous communication
  - **Can** also reside in system-level address space
- Message envelope
  - Meta data like source, destination, tag, size, type, communicator, … (implementation dependent)
  - Messages consists of an envelope and a data (or payload) part

MPI Communication Modes

MPI Communication Modes

- MPI standard defines four send modes:
  - **Standard**: MPI_Send
  - **Synchronous**: MPI_Ssend
  - **Buffered**: MPI_Bsend
  - **Ready**: MPI_Rsend
- All in combination with blocking/non-blocking
  - MPI_{",I}{",S,B,R}send
- Receives: only blocking/non-blocking, no other types!
  - Send call determines communication mode

Non-Buffered Blocking Send



- Handshake protocol for non-buffered blocking send/receive
  - „Rendezvous“
- Overheads due to idle times
  - Latency: time required to send a message from A to B

Non-Buffered Blocking Send

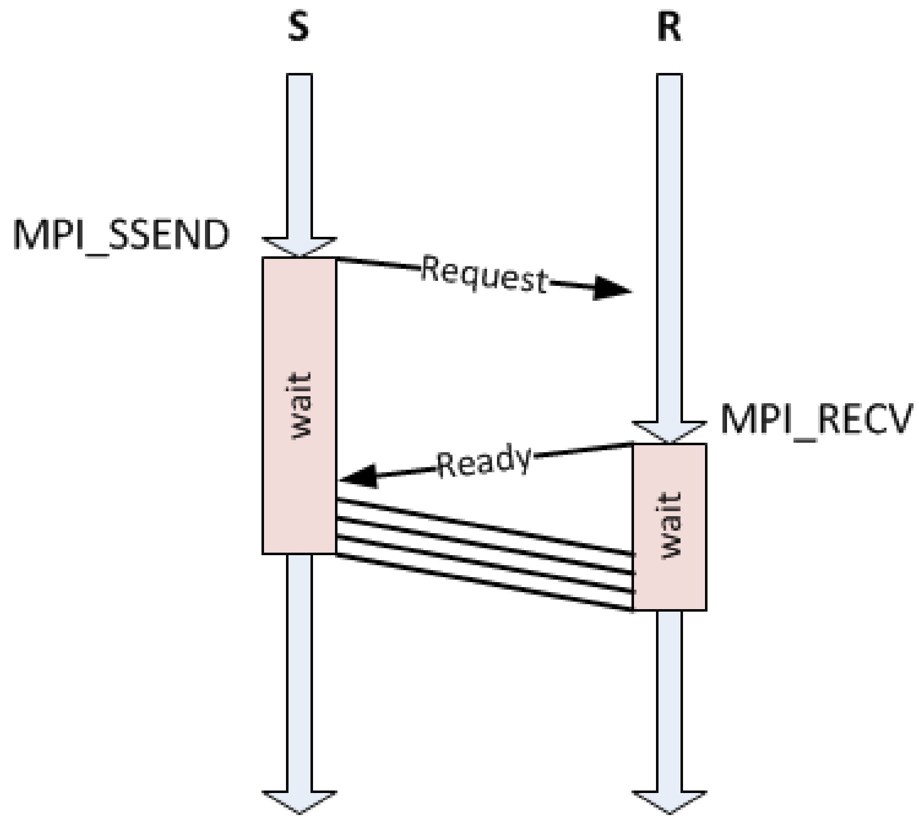
- In general: *return only when it‘s safe to do so*

1. Non-buffered blocking send
   - Returns if the matching receive has been encountered
   - Major issues: idle times and deadlocks
   - Provides synchronization between sender & receiver
   - **Non-local**: completion may depend on matching receive
2. Buffered blocking send
   - Returns if the buffer can be used again
     - I.e., buffer has been copied internally
   - Reduces idle times, additional costs for copying
   - Improved decoupling between sender & receiver
   - **Local**: completion does not depend on matching receive

- In any case, after returning the buffer can be reused

Blocking Synchronous Send

- Communication mode is selected while invoking the send routine
- Synchronization overhead at sender
  - Wait for receive to be executed and handshake to arrive
  - Then, transfer message
- Synchronization overhead at receiver
  - Wait for handshake to complete
- Overhead incurred while copying from buffer to the network
- **Non-local, no buffers required**

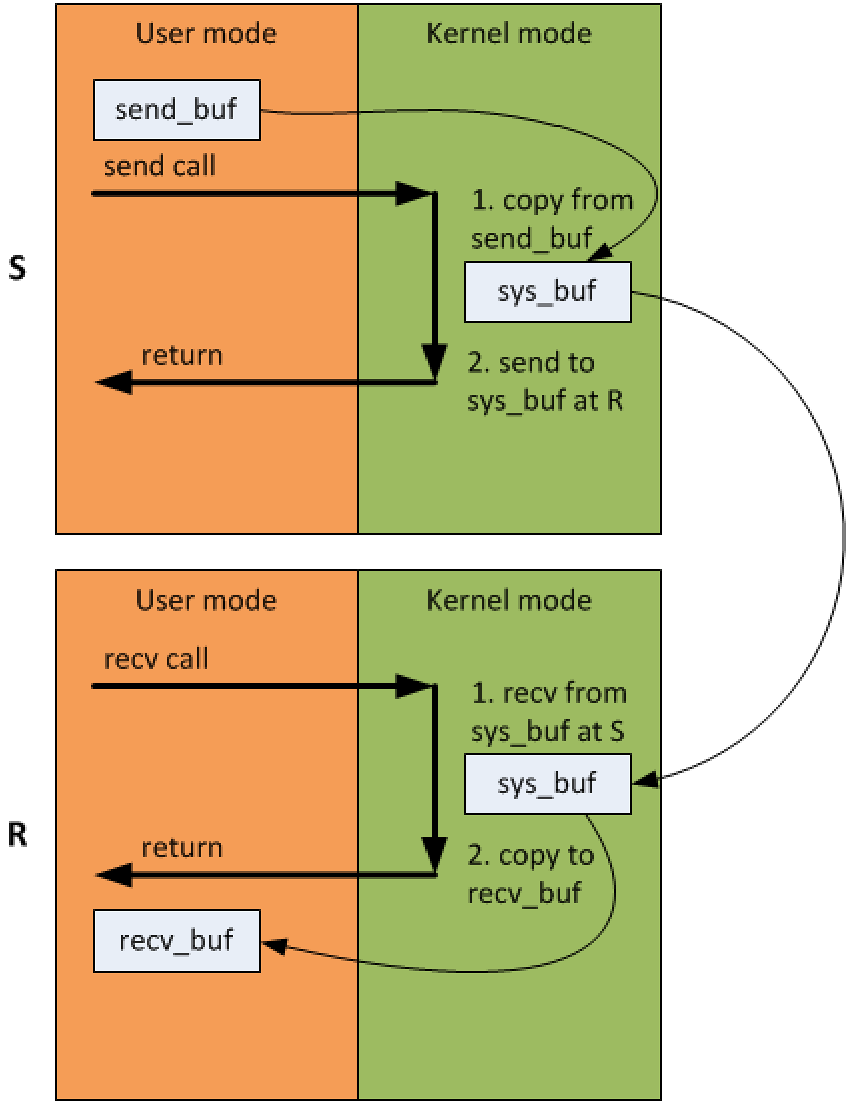


Tasks wait until data transfer is complete

Basic Buffering Concept

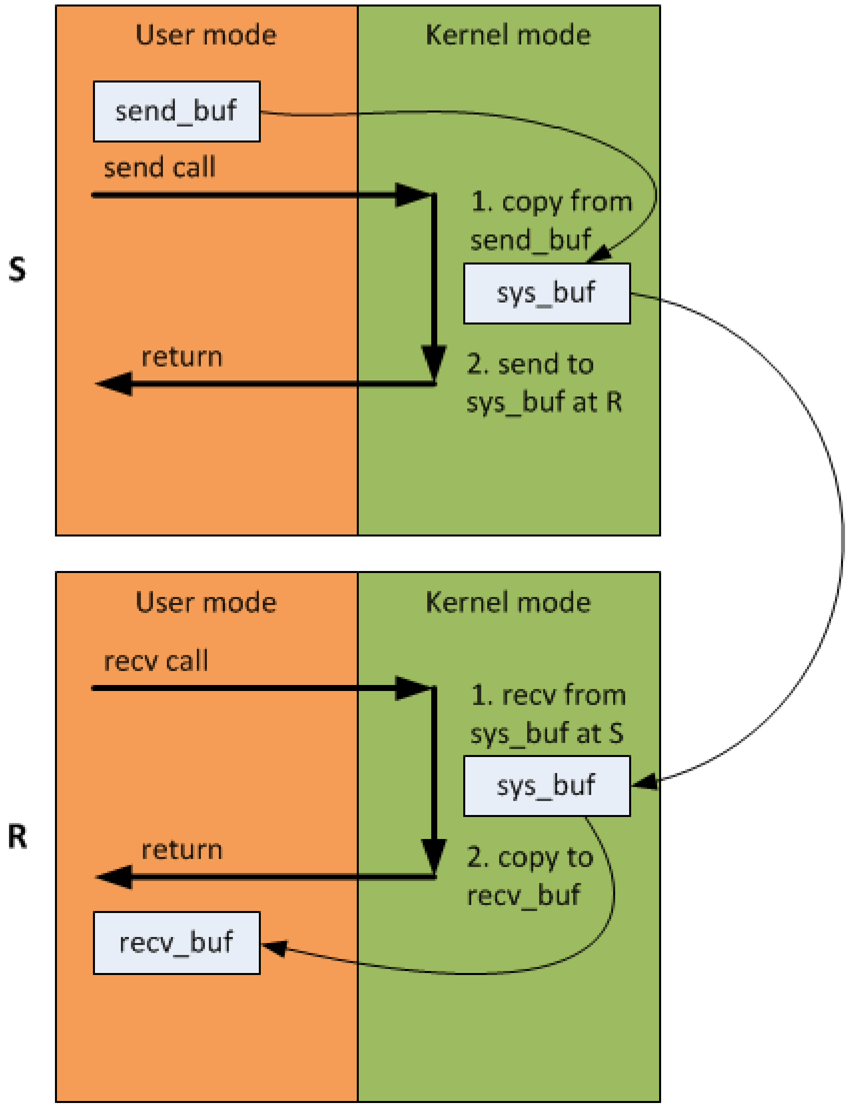
1. Sending process copies data to be sent from user buffer to system buffer
2. Data is sent over the network into remote system buffer
3. Receiving process copies data from system buffer to user buffer

- Notes
  - Return on sender side depends on selected communication mode
  - Special communication hardware may provide additional buffers



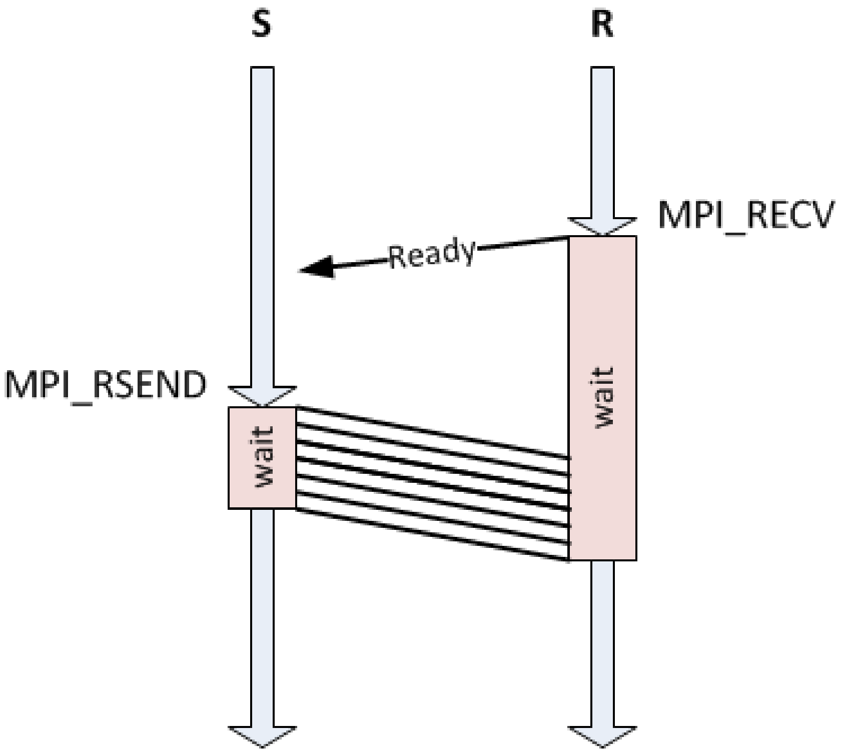
Basic Buffering Concept

- **Without** communication hardware buffers
  - Interrupts required
  - CPU on-loading
  - Example: Ethernet (plain)
- **With** communication hardware buffers
  - Minimized overhead
  - CPU off-loading
  - Example: Infiniband



Blocking Ready Send

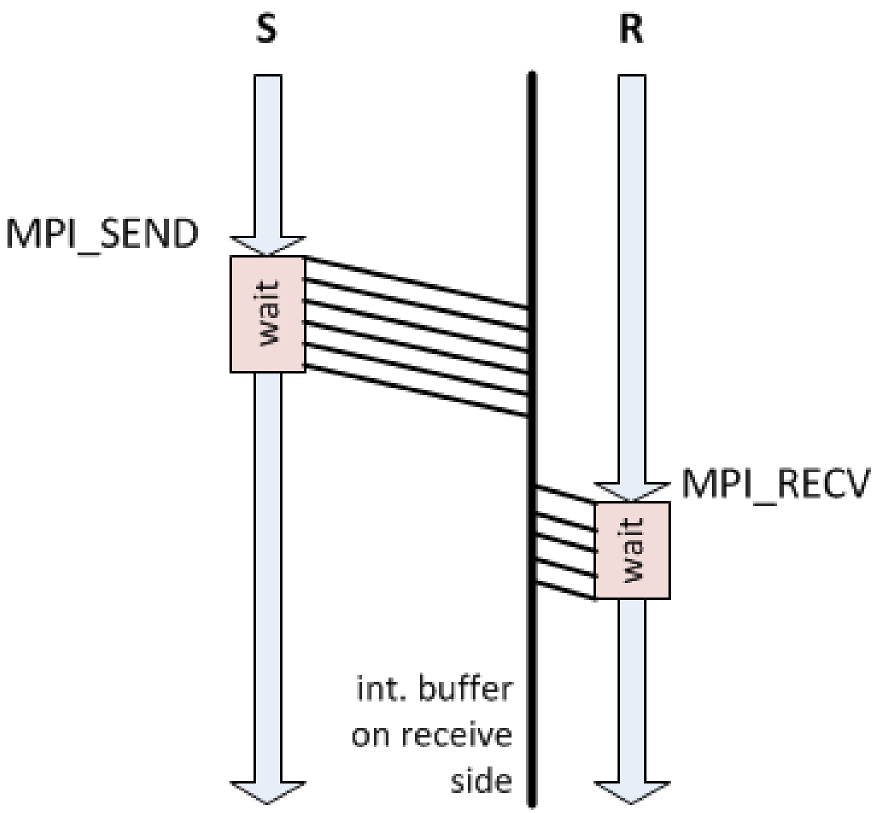
- Ready Send expects pre-posted receive
- If not, an error will incur and send returns
  - Programmer‘s responsibility to handle errors
- Overhead on sender side minimized
- Receive side still can incur significant overhead
  - *“No free lunch”*



Tasks wait until data transfer is complete

Blocking Standard Send – Eager

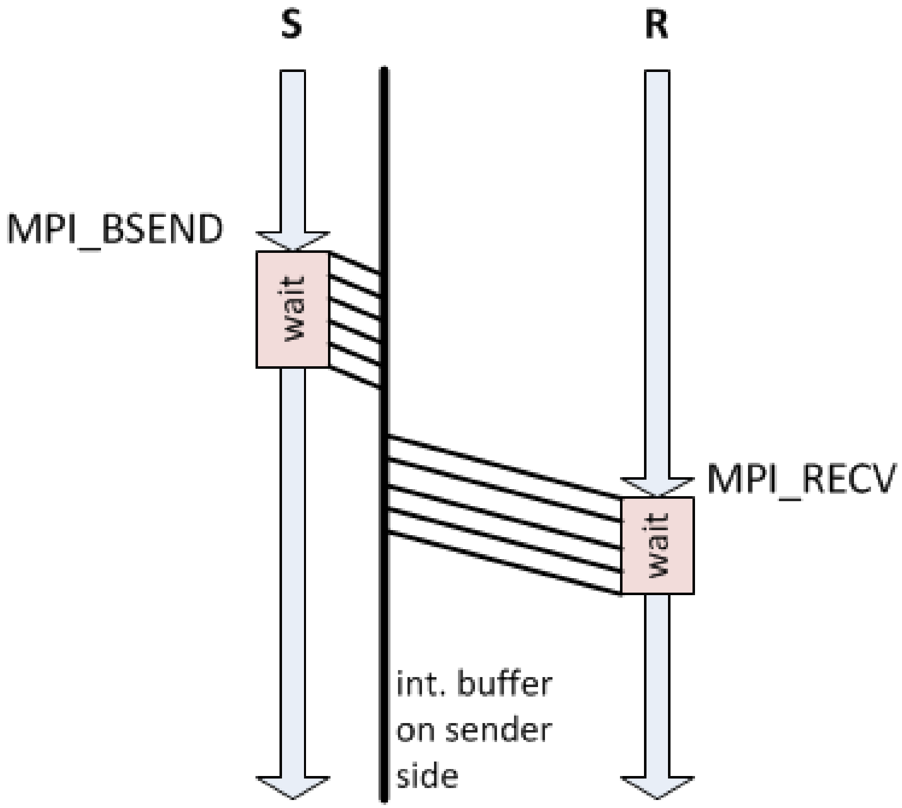
- Standard send is implementation dependent
  - Only typical aspects covered here…
- Typically it includes at least one threshold value
  - If data size is below threshold (small messages), it is copied into a buffer on the receiver side
- Receive call copies the data from system buffer to user buffer



Standard send if data size below threshold. (Eager protocol)

Blocking Buffered Send

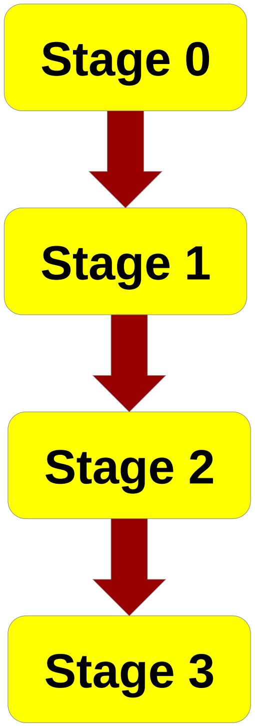
- User supplied buffers on sender side
- Message buffer can be immediately reused
- Low (time) overhead on sender side, large buffering requirements



Bufferd send using user supplied buffers on sender side

Note on Bounded Buffers

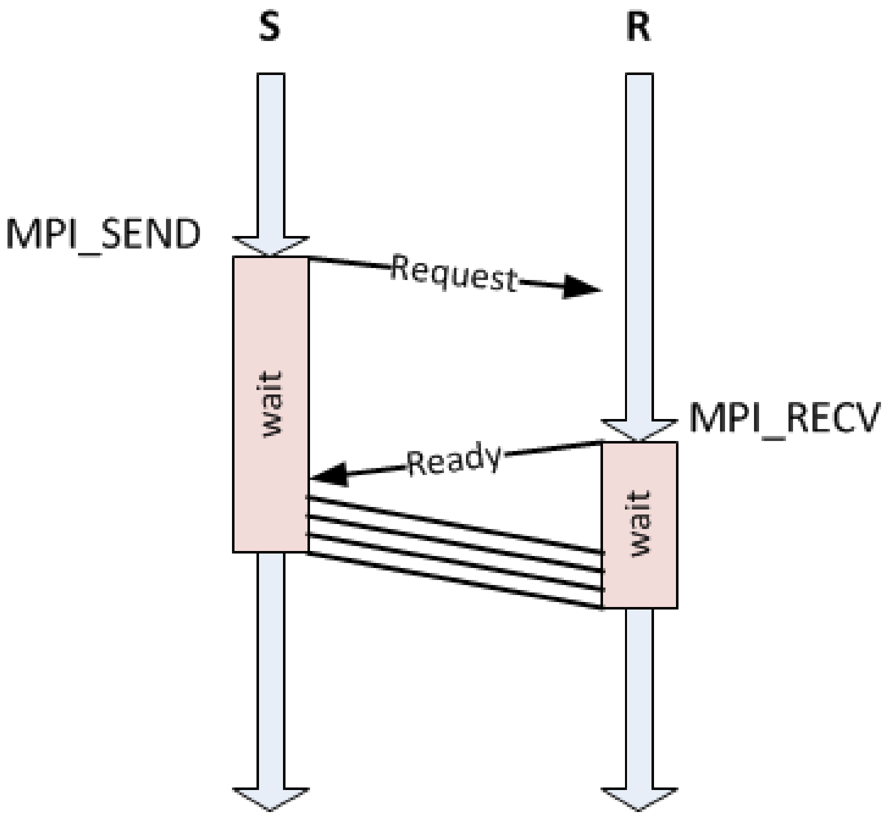
- Severe performance degradation can happen
  - Producer/Consumer with one being slower
  - Buffers are – independently of their size – always full or empty
- Slowest part of a chain limits overall performance
- Independent how large buffers are, there is always an upper bound
- Lower bound: at least one maximum sized message
- Buffer exhaustion or overflow can lead to program failure or stalling
  - Buffering is implementation-dependent



RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Blocking Standard Send - Rendezvous

- Again: standard send is implemention dependent
- If data size exceeds threshold: **synchronous send**
  - Rendezvous protocol
- Unnecessary copying avoided
- Longer idle times



Standard send if data size above threshold. (Rendezvous protocol)

Eager vs. Rendezvous

	Eager	Rendezvous
Send call	• Sender assumes that receiver can store the message	• No assumptions are made about receiver – send out request containing envelope
Receive call	• System buffers at receiver side • Receiver decides how to handle message	• Interpret request, search for buffer space (user- and/or system-level) • Send back response
Process count	• Severely limits buffer space	• Implies no limitations
Advantages	• Reduces synchronization delay, no rendezvous	• Scalable • No additional copying
Disadvantages	• Not scalable • Additional copies • Buffer space over-provisioned • Behaviour in the case of buffer exhaustion?	• Inherent synchronization due to handshaking • Best use for non-blocking sends - more programming complexity
Primary use	Small messages (< ***n*** kB)	Large messages (>= ***n*** kB)

Deadlock issues

- Additional buffers cannot avoid deadlocks!

```
P0              P1
RECV (P1, B)    RECV (P0, A)
SEND (P1, A)    SEND (P1, B)
```

- Blocking calls are more likely to deadlock
- Solutions:
  - Non-blocking ISENDs
  - Reordering

```
P0              P1
SEND (P1, A)    RECV (P0, B)
SEND (P1, B)    RECV (P0, A)
```

```
P0              P1
SEND (P1, A)    RECV (P0, A)
RECV (P1, B)    SEND (P1, B)
```

- Producing deadlocks is much easier than one might think

Notes on Blocking Send/Recv

- Blocking: MPI_SEND & MPI_RECV
- Blocking in the sense, that the function call won‘t complete until data has been completely copied from or to message buffer
  - **Send**: nothing is guaranteed about receiving!
  - Obviously a blocking **receive** also implies a corresponding send
- If MPI_SEND or MPI_RECV finish, then the message buffer is freed respectively valid
  - Nothing is guaranteed for non-blocking calls
  - Use wait or test operation to update corresponding MPI_STATUS

More Notes on Blocking Send/Recv

- Beside the high deadlock potential, there are also performance issues
- Make use of overlap between computation and communication!
  - Non-blocking calls as soon as user buffers are ready
  - Use time between first call and following wait for other useful work
    - Computation (pipelined fashion)
    - Other messaging
    - I/O, …
- Even more beneficial if special communication hardware support exists
  - Off-loading, e.g. DMA engines in the networking device

Notes on non-blocking Send/Recv

- Standard non-blocking send, below threshold (eager):
  - Sender side
    - Eventually: wait until buffer space at receiver available
    - Send & let receiver decide how to handle message
  - Receiver side
    - Copy to user buffer (if suitable receive already posted)
    - Copy to system buffer (otherwise)
  - **Non-local**
- Standard non-blocking send, above threshold (rendezvous):
  - Sender side
    - Enqueue request in local request queue & return
  - **Local**

Progress

- Who guarantees that non-blocking operations are futher processed?

- Completion of non-blocking operations might depend on subsequent calls
  - Implementation dependent

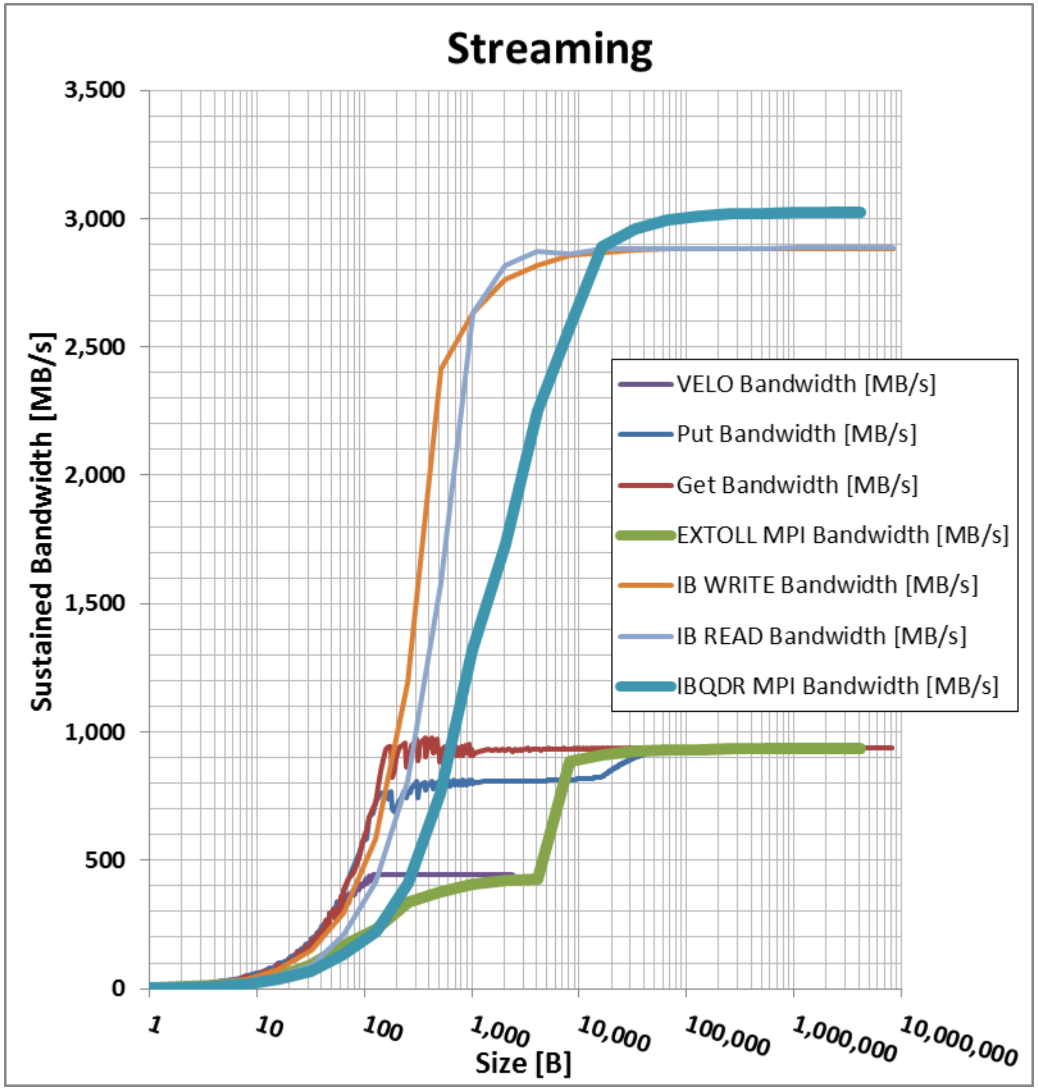
- Non-blocking sends typically only enqueue requests
- Non-blocking receives typically also enqueue requests, but also call internally the **progress function**
- All blocking operations call the progress function

Progress

- Internal request queue (or ring buffer)
  - For efficient decoupling between computation and communication
  - Case of exhaustion is implementation dependent
- Messages have to be processed in-order
  - In the queue there might be older requests with identical criteria (source, tag)
  - Always enqueue, even if a matching send/receive request is available
- Progress is one of the biggest overhead sources in MPI
  - Besides additional copying

MPI Overhead

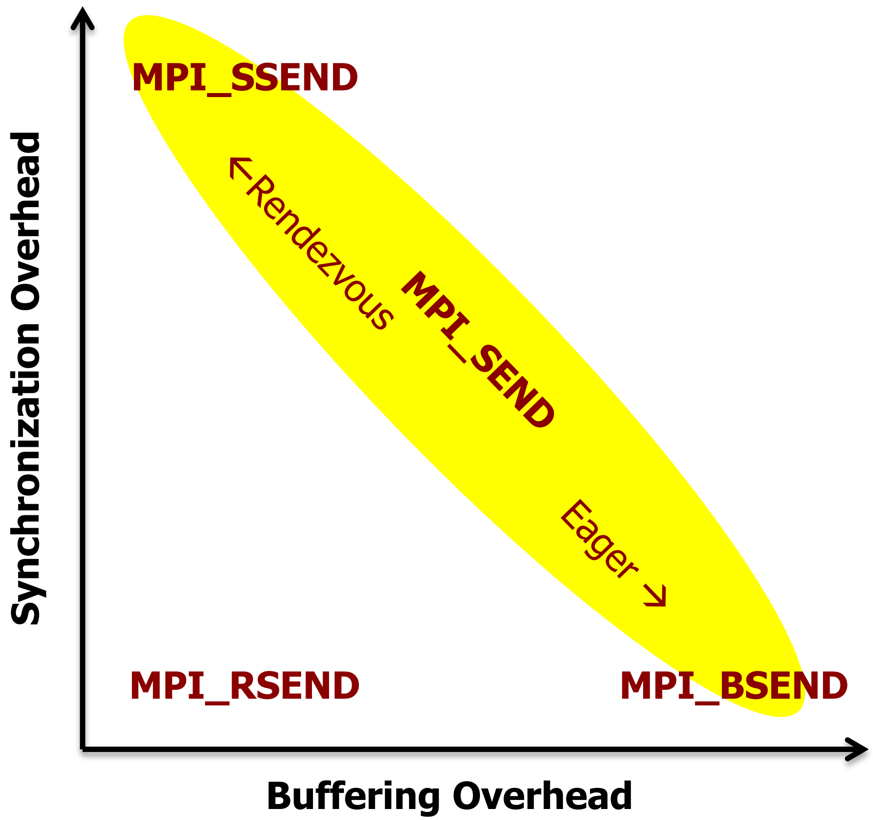
- Many guarantees
  - Send/receive semantics
- Huge overhead
  - Tag matching
  - Copying
  - Progress
- CPU2CPU example
  - Simple streaming test
  - EXTOLL R2 FPGA & OpenMPI
  - IBQDR & MVAPICH



Summary

- MPI send call determines communication mode
  - Standard, Buffered, Synchronous, Ready
- Blocking/non-blocking independent of this
  - Also applies to receive calls
- Typical functionality within communication
  - Rendezvous-protocol
  - Eager-protocol
- Trade-offs between copying and long idle times
  - Both are overhead sources and limit performance
  - Copying is data movement, data movement is extremely expensive
  - So are long idle times ☹
- Many aspects are implementation dependent
  - Allows to optimize for system-specific properties and characteristics

Summary



Communication Modes

1. **MPI_SSEND?**
2. **MPI_RSEND?**
3. **MPI_BSEND?**
4. **MPI_SEND?**

Too difficult?

A short excerpt

“We are probably the 1st generation of message-passing programmers.“

“We are also probably the last generation of message-passing programmers.“

[MPI – The complete reference, Vol. 2, The MPI Extensions, 1998]