ARCHER MPI LIBRARY

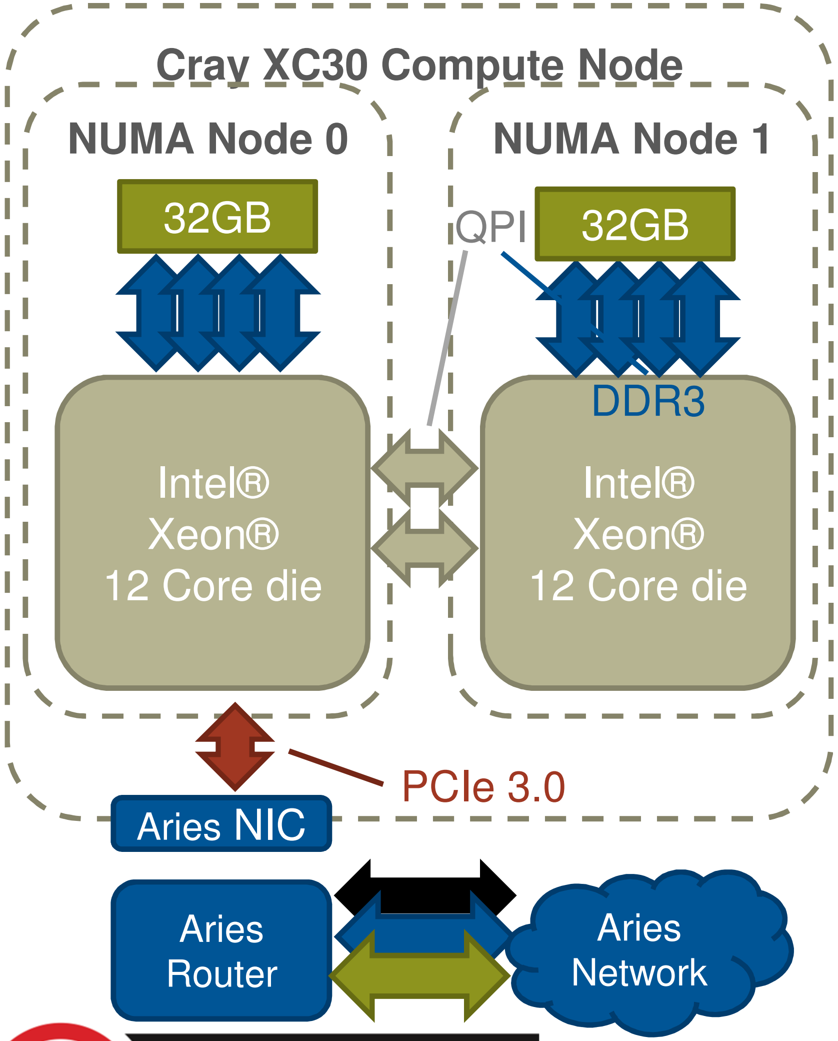
Slides contributed by Cray and EPCC

Architectural features of XC30

- Many levels of “closeness” between physical cores
  - leads to many levels of closeness between user proceses
  - e.g. many levels of “closeness” between MPI sender and receiver

- A reminder of what they are ….

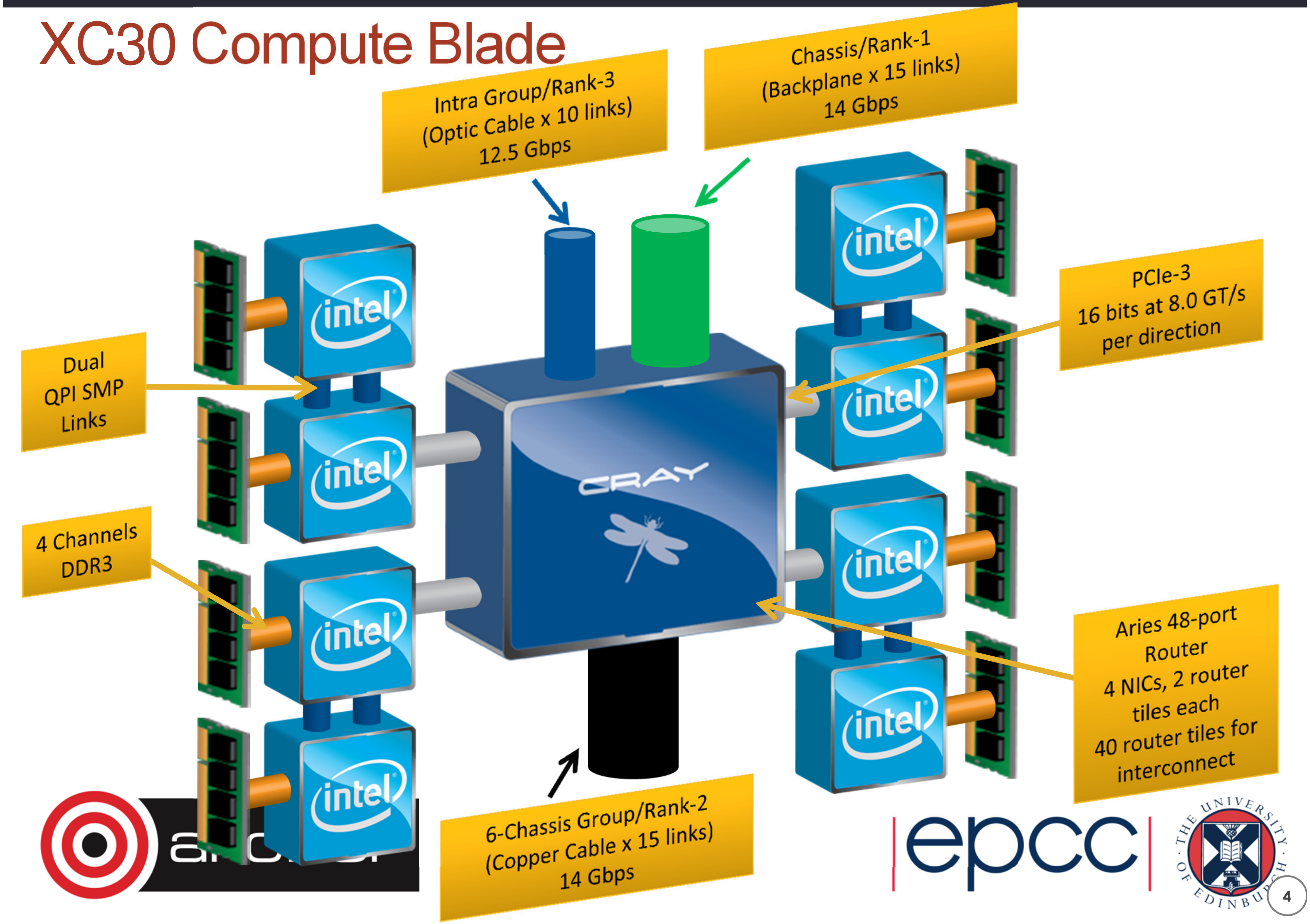
Cray XC30 Intel® Xeon® Compute Node



The XC30 Compute node features:

- 2 x Intel® Xeon® Sockets/die
  - 12 core Ivy Bridge
  - QPI interconnect
  - Forms 2 NUMA nodes
- 8 x 1833MHz DDR3
  - 8 GB per Channel
  - 64/128 GB total
- 1 x Aries NIC
  - Connects to shared Aries router and wider network
  - PCI-e 3.0

XC30 Compute Blade



Intra Group/Rank-3
(Optic Cable x 10 links)
12.5 Gbps

Chassis/Rank-1
(Backplane x 15 links)
14 Gbps

PCIe-3
16 bits at 8.0 GT/s
per direction

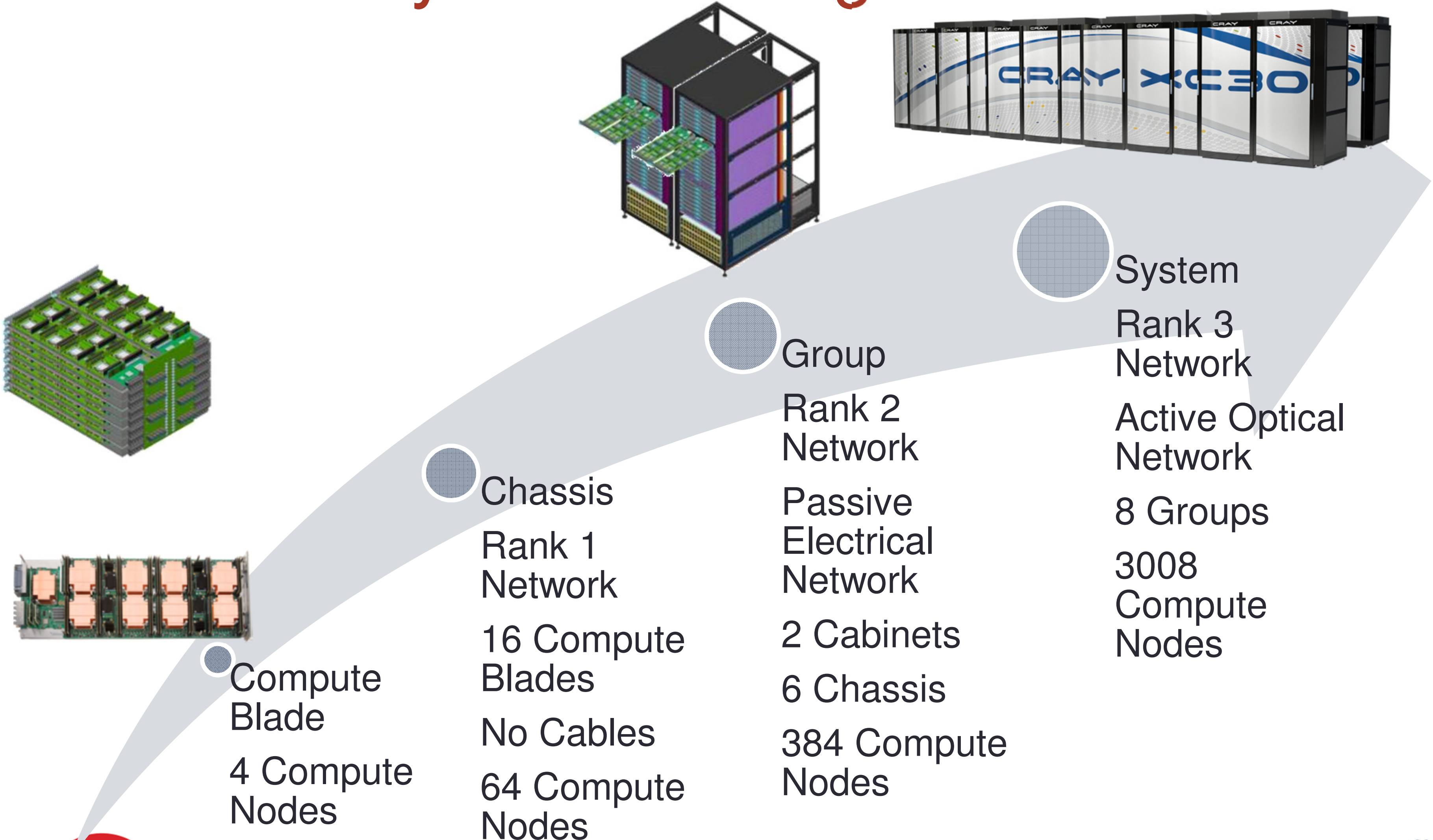
Dual
QPI SMP
Links

4 Channels
DDR3

Aries 48-port
Router
4 NICs, 2 router
tiles each
40 router tiles for
interconnect

6-Chassis Group/Rank-2
(Copper Cable x 15 links)
14 Gbps

ARCHER System Building Blocks



Compute Blade

4 Compute Nodes

Chassis

Rank 1 Network

16 Compute Blades

No Cables

64 Compute Nodes

Group

Rank 2 Network

Passive Electrical Network

2 Cabinets

6 Chassis

384 Compute Nodes

System

Rank 3 Network

Active Optical Network

8 Groups

3008 Compute Nodes

Architectural features relevant to MPI

- In principle expect hierarchy of MPI performance between

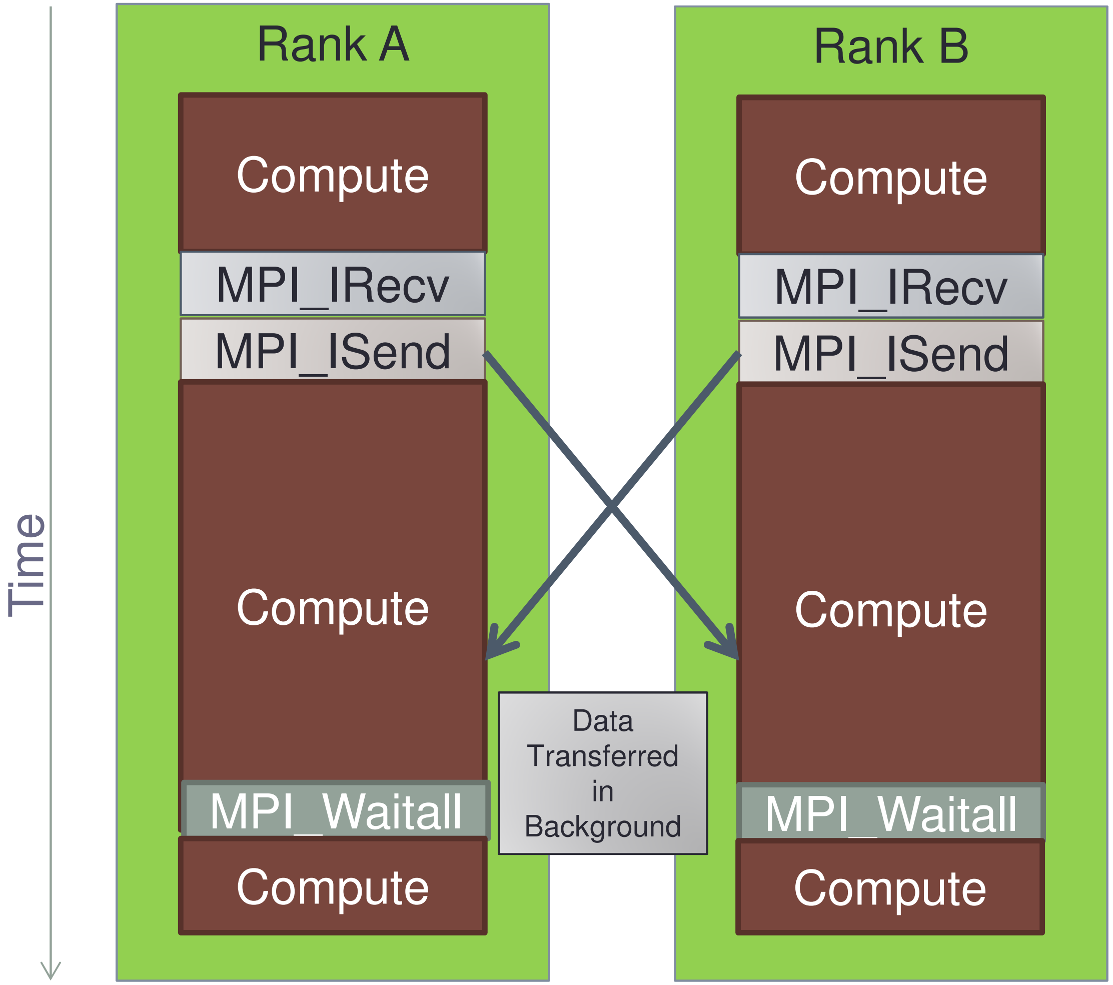
  1) two hyperthreads on the same core
  2) two cores on the same NUMA region but different cores
  3) two cores on the same node but different NUMA regions
  4) two cores on the same blade but different nodes
  5) two cores on the same chassis but different blades
  6) two cores on the same group but different chassis
  7) two cores in different groups

- In practice levels 4 – 7 are basically the same
  - As for HECToR, only really care if comms is on-node or off-node

MPICH2 and Cray MPT

- Cray MPI uses MPICH2 distribution from Argonne
  - Provides a good, robust and feature rich MPI
  - Cray provides enhancements on top of this:
    - low level communication libraries
    - Point to point tuning
    - Collective tuning
    - Shared memory device is built on top of Cray XPMEM
- Many layers are straight from MPICH2
  - Error messages can be from MPICH2 or Cray Libraries.

Overlapping Communication and Computation



The MPI API provides many functions that allow point-to-point messages (and with MPI-3, collectives) to be performed asynchronously.

Ideally applications would be able to overlap communication and computation, hiding all data transfer behind useful computation.

Unfortunately this is not always possible at the application and not always possible at the implementation level.

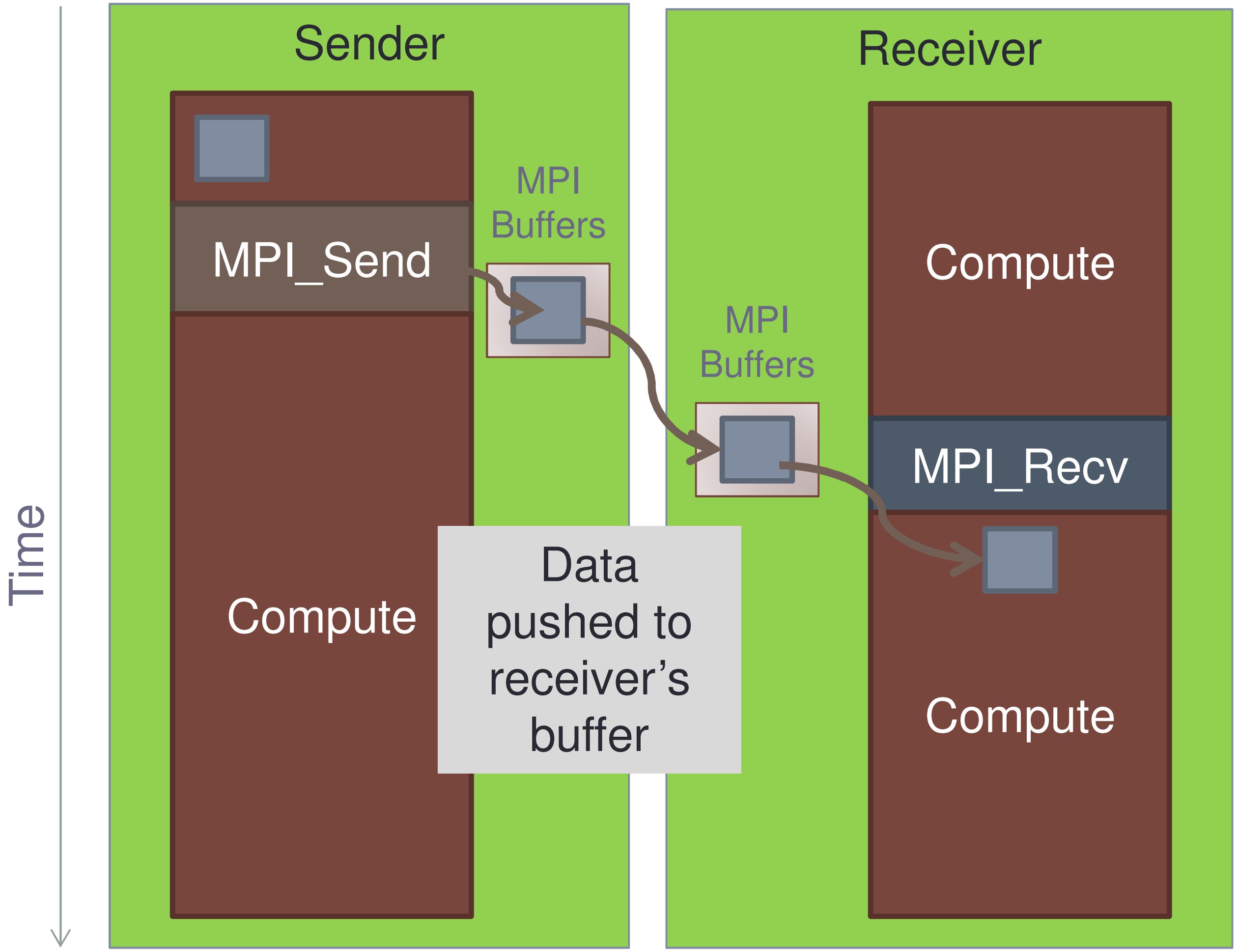
What prevents Overlap?

- Even though the library has asynchronous API calls, overlap of computation and communication is not always possible

- This is usually because the sending process does not know where to put messages on the destination as this is part of the MPI_Recv, not MPI_Send.

- Also on Gemini and Aries, complex tasks like matching message tags with the sender and receiver are performed by the host CPU. This means:
  - + Gemini and Aries chips can have higher clock speed and so lower latency and better bandwidth
  - + Message matching is always performed by one fast CPU per rank.
  - - Messages can usually only be “progressed” when the program is inside an MPI function or subroutine.

EAGER Messaging – Buffering Small Messages



Smaller messages can avoid this problem using the eager protocol.

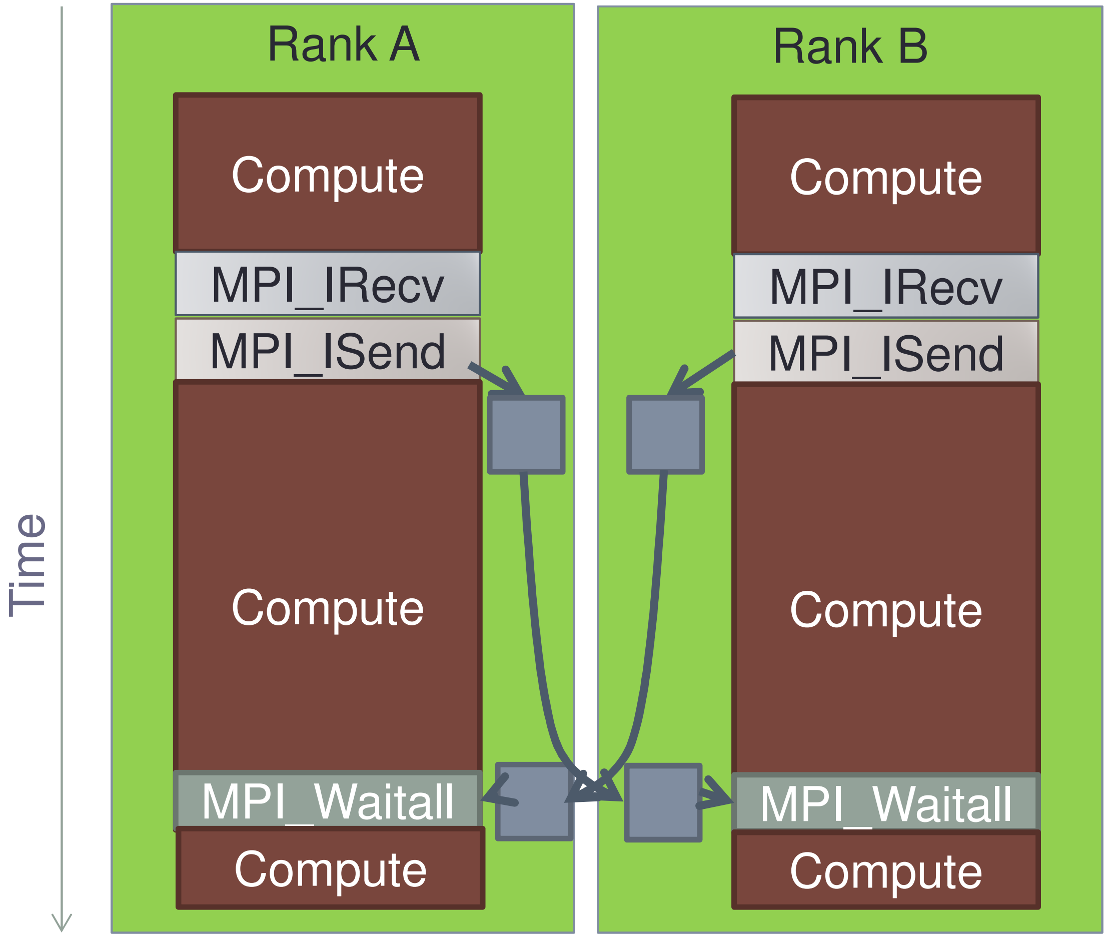
If the sender does not know where to put a message it can be buffered until the sender is ready to take it.

When MPI Recv is called the library fetches the message data from the remote buffer and into the appropriate location (or potentially local buffer)

Sender can proceed as soon as data has been copied to the buffer.

Sender will block if there are no free buffers

EAGER potentially allows overlapping

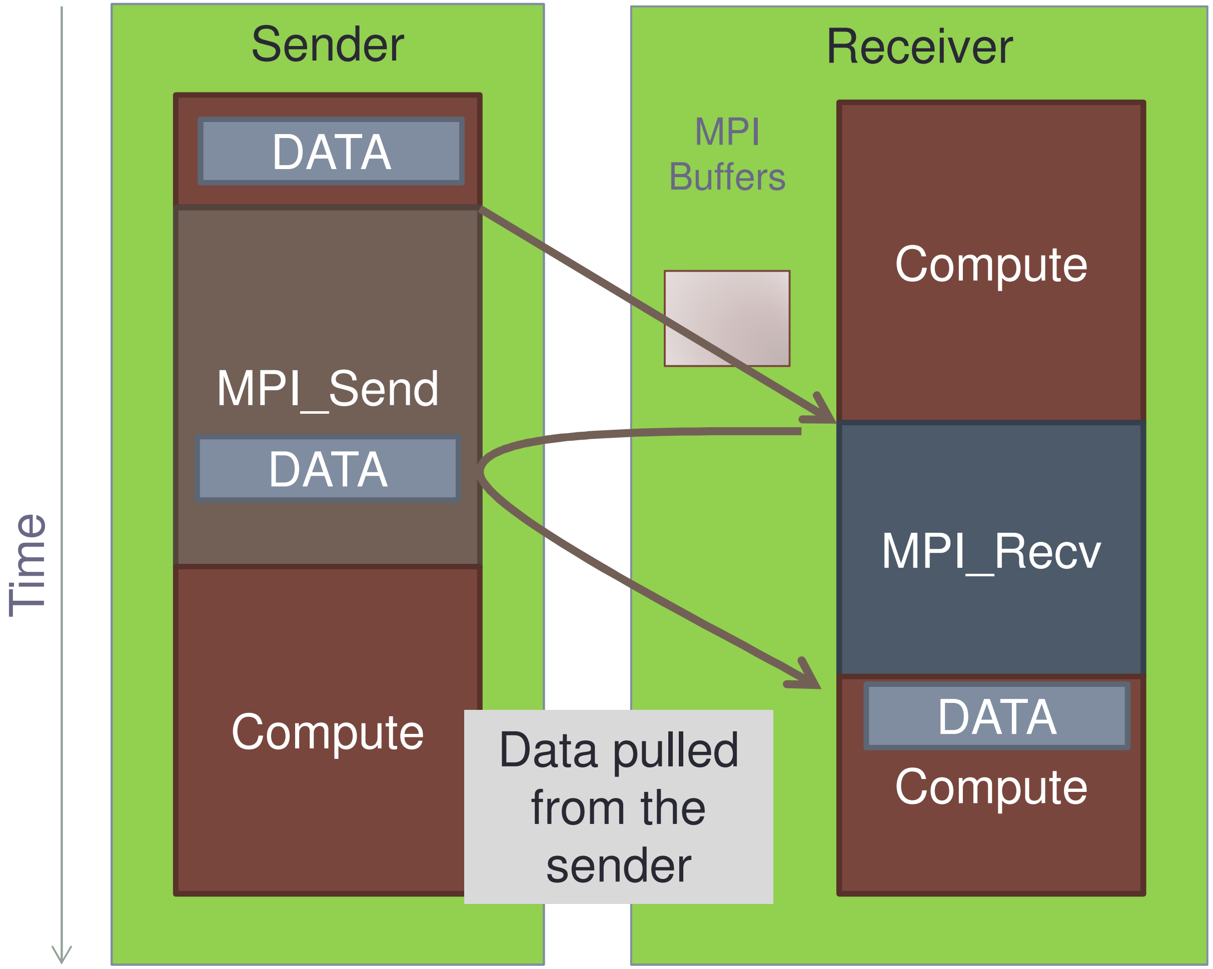


Data is pushed into an empty buffer(s) on the remote processor.

Data is copied from the buffer into the real receive destination when the wait or waitall is called.

Involves an extra memcopy, but much greater opportunity for overlap of computation and communication.

RENDEZVOUS Messaging – Larger Messages



Data pulled from the sender

Larger messages (that are too big to fit in the buffers) are sent via the **rendezvous** protocol

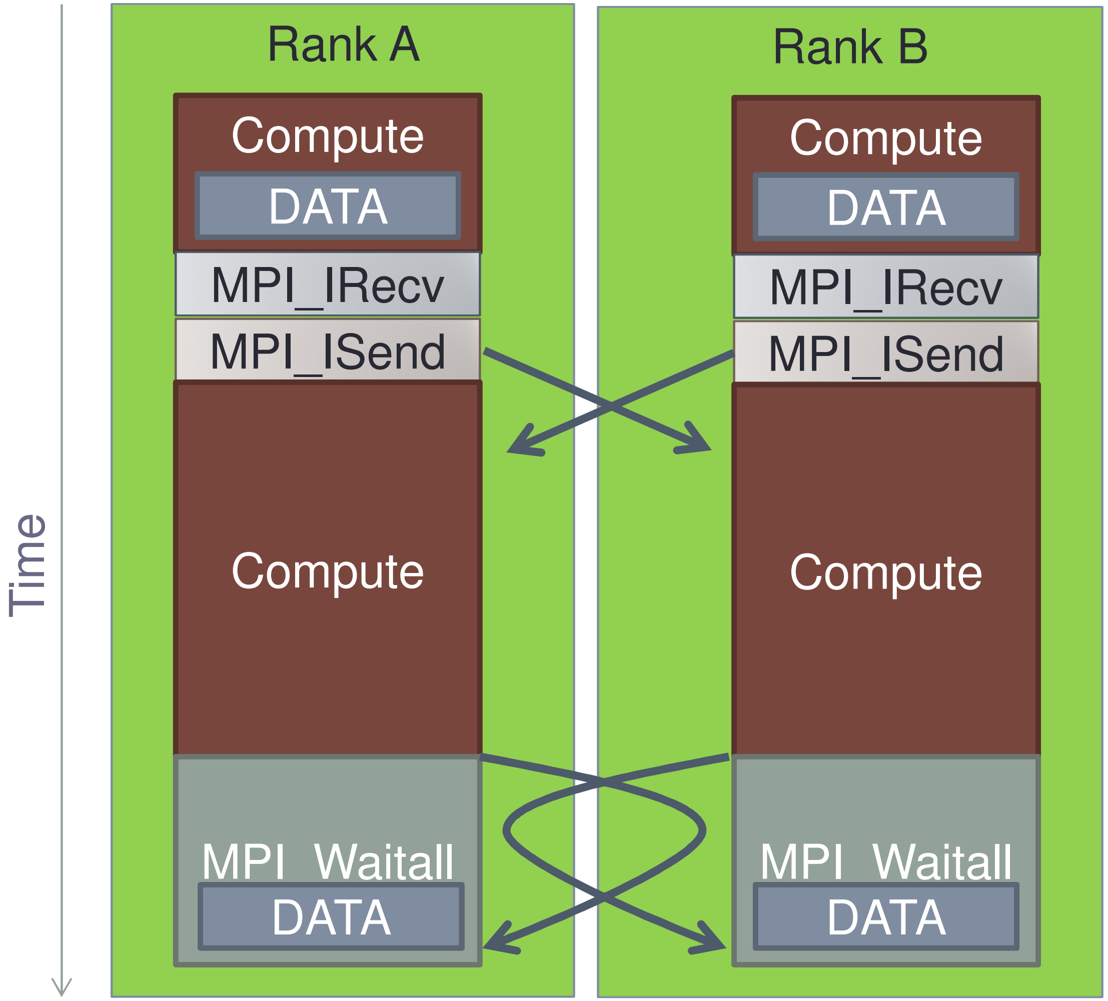
Messages cannot begin transfer until MPI_Recv called by the receiver.

Data is pulled from the sender by the receiver.

Sender must wait for data to be copied to receiver before continuing.

Sender and Receiver block until communication is finished

RENDEZVOUS does not usually overlap



With rendezvous data transfer often only occurs during the Wait or Waitall statement.

When the message arrives at the destination, the host CPU is busy doing computation, so is unable to do any message matching.

Control only returns to the library when MPI_Waitall occurs and does not return until all data is transferred.

There has been no overlap of computation and communication.

Making more messages EAGER

- One way to improve performance is to send more messages using the eager protocol.
- This can be done by raising the value of the eager threshold, by setting environment variable:
  `export MPICH_GNI_MAX_EAGER_MSG_SIZE=X`
- Values are in bytes, the default is 8192 bytes. Maximum size is 131072 bytes (128KB).
- Try to post MPI_IRecv calls before the MPI_ISend call to avoid unnecessary buffer copies.

Consequences of more EAGER messages

- Sending more messages via EAGER places more demands on buffers on receiver.
- If the buffers are full, transfer will wait until space is available or until the Wait.
- Buffer size can be increased using:
  `export MPICH_GNI_NUM_BUFS=X`
- Buffers are 32KB each and default number is 64 (total of 2MB).
- Buffer memory space is competing with application memory, so we recommend only moderate increases.