MPI and MPI on ARCHER

Reusing this material

archer

epcc | THE UNIVERSITY OF EDINBURGH

MPI Forum

- Main web site at http://meetings.mpi-forum.org/
- The MPI Forum contains representatives from many of the vendors and academic library developers.
- This is one reason the specification is so large:
  - MPI supports many different models of communication, corresponding to the various communication models supported by its predecessors.
- Much of the specification was driven by the library developers.
  - The API leaves a lot of scope for optimised versions on different hardware.
  - Many aspects of the MPI specification deliberately allow different implementations the freedom to work in different ways.
    - This makes it easy to port/optimise MPI for new hardware.
    - Application developers need to be aware of this when writing code.
    - Erroneous applications may work fine on one MPI implementation but fail using a different one.

History of MPI

- MPI is an “Application Programming Interface” (API) specification.
  - Its a specification not a piece of code.
  - There are many different implementations of the MPI specification.
- The MPI Standard is defined by the MPI Forum
  - Work started 1992
  - Version 1.0 in 1994 – basic point-to-point, collectives, data-types, etc
  - Version 1.1 in 1995 – fixes and clarifications to MPI 1.0
  - Version 1.2 in 1996 – fixes and clarifications to MPI 1.1
  - Version 1.3 in 1997 – refers to MPI 1.2 after combination with MPI-2.0
  - Version 2.0 in 1997 – parallel I/O, RMA, dynamic processes, C++, etc
  - --- Stable for 10 years ---
  - Version 2.1 in 2008 – fixes and clarifications to MPI 2.0
  - Version 2.2 in 2009 – small updates and additions to MPI 2.1
  - Version 3.0 in 2012 – neighbour collectives, unified RMA model, etc
  - Version 3.1 in 2015 – fixes, clarifications and additions to MPI 3.0

MPI-2 One-sided communication

- Separates data transmission from process synchronisation
- All communication parameters specified by a single process
- Definitions: “origin” calls MPI, memory accessed at “target”
- Initialise by creating a “window”
  - A chunk of local memory that will be accessed by remote processes
- Open origin “access epoch” (and target “exposure epoch”)
- Communicate: MPI_Put, MPI_Get, MPI_Accumulate
- Synchronise: passive target (or active target)
- Use data that has been communicated
- Tidy up by destroying the window – MPI_Win_free

MPI 3.0

- Major new features
  - Non-blocking collectives
  - Neighbourhood collectives
  - Improvements to one-sided communication
  - Added a new tools interface
  - Added new language bindings for FORTRAN 2008
- Other new features
  - Matching Probe and Recv for thread-safe probe and receive
  - Non-collective communicator creation function
  - Non-blocking communication duplication function
  - “const” correct C language bindings
  - New MPI_Comm_split_type function
  - New MPI_Type_create_hindexed_block function
- C++ language bindings removed
- Previously deprecated functions removed

MPI 3.0 – Changes to collectives

- Non-blocking versions of all collective communication functions added
  - MPI_Ibcast, MPI_Ireduce, MPI_Iallreduce, etc
  - There is even a non-blocking barrier, MPI_Ibarrier
  - They return MPI_Request like other non-blocking functions
  - The user code must complete the operation with (one of the variants of) MPI_Test or MPI_Wait
  - Multiple non-blocking collectives can be outstanding but they must be called in the same order by all MPI processes
- New neighbourhood collective functions added
  - MPI_Neighbor_allgather and MPI_Neighbor_alltoall (plus variants)
  - Neighbours defined using a virtual topology, i.e. cartesian or graph
  - Extremely useful for nearest-neighbour stencil-based computations
  - Allow a scalable representation for common usage of MPI_Alltoallv

MPI 3.0 – Changes to One-sided

- New window creation functions
  - New options for where, when and how window memory is allocated
- New atomic read-modify-write operations
  - MPI_Fetch_and_op and MPI_Compare_and_swap
- New “unified” memory model
  - Old one still supported, now called “separate” memory model
  - Simplifies memory consistency rules on cache-coherent machines
- New local completion semantics for one-sided operations
  - MPI_Rput, MPI_Rget and MPI_Raccumulate return MPI_Request
  - User can use MPI_Test or MPI_Wait to check for local completion

MPI Next – End-points and Fault-tolerance

- End-points proposal – improved support for hybrid programs
  - Allow threads to act like MPI processes
  - Allow multiple MPI ranks for a communicator in a single OS process
  - Example use-case: easier to map UPC thread id to MPI rank
- Fault-tolerance proposal – improved error-handling
  - Allow an MPI program to survive various types of failure
  - Node failure, communication link failure, etc
  - Notification: local process told particular operation will not succeed
  - Propagation: local knowledge of faults disseminated to global state
  - Consensus: vote for and agree on a common value despite failures
  - Low-level minimum functionality to support fault-tolerance libraries

MPI implementations

- There are many different implementations of the MPI specification.
- Many of the early ones were based on pre-existing portable libraries.
- Currently there are 2 main open source MPI implementations
  - MPICH
  - OpenMPI
- Many vendor MPI implementations are now based on these open source versions.

MPICH

- Virtually the default MPI implementation
  - Mature implementation.
  - Good support for generic clusters (TCP/IP & shared memory).
  - Many vendor MPIs now based on MPICH.
- Original called MPICH (MPI-1 functionality only)
- Re-written from scratch to produce MPICH-2 (MPI-2)
- Incorporated MPI-3 and renamed back to MPICH again
- Ported to new hardware by implementing a small core ADI
  - ADI = Abstract Device Interface.
  - Full API has default implementation using the core ADI functions.
  - Any part can be overridden to allow for optimisation.

OpenMPI

- New MPI implementation
  - Joint project between developers of some previous active MPI libraries
- Very active project
  - In its early days:-
    - Special emphasis on support for infiniband hardware
    - Special emphasis on Grid MPI
      - Fault tolerant communication
      - Heterogeneous communication
    - Sun switched to OpenMPI with clustertools-7
  - Current version supports MPI-3
  - Open Source project with large and varied community effort

MPI Libraries

- Most MPI implementations use a common “superstructure”
  - lots of lines of code
  - deals with whole range of MPI issues: datatypes, communicators, argument checking, …
  - will implement a number of different ways (protocols) of sending data
  - all hardware-specific code kept separate from the rest of the code, e.g. hidden behind an Abstract Device Interface
- To optimise for a particular architecture
  - rewrite low-level communication functions in the ADI
  - optimise the collectives especially for offload hardware
  - use machine-specific capabilities when advantageous
- Multi-core nodes
  - modern MPI libraries are aware of shared-memory nodes
  - already include optimisations to speed up node-local operations
  - uses multiple implementations of the same ADI in a single library

MPI Structure

- Like any large software package MPI implementations need to be split into modules.
- MPI has a fairly natural module decomposition roughly following the chapters of the MPI Standard.
  - Point to Point
  - Collectives
  - Groups Contexts Communicators
  - Process Topologies
  - Process creation
  - One Sided
  - MPI IO
- In addition there may be hidden internal modules e.g.
  - ADI encapsulating access to network

Point to Point

- Point to point communication is the core of most MPI implementations.
- Collective calls are usually (but not always) built from point to point calls.
- MPI-IO usually built on point to point
  - Actually almost all libraries use the same ROMIO implementation.
- Large number of point-to-point calls exist but these can all be built from a smaller simpler core set of functions (ADI).

MPI communication modes

- MPI defines multiple types of send
  - Buffered
    - Buffered sends complete locally whether or not a matching receive has been posted. The data is “buffered” somewhere until the receive is posted. Buffered sends fail if insufficient buffering space is attached.
  - Synchronous
    - Synchronous sends can only complete when the matching receive has been posted
  - Ready
    - Ready sends can only be started if the receive is known to be already posted (its up to the application programmer to ensure this) This is allowed to be the same as a standard send.
  - Standard
    - Standard sends may be either buffered or synchronous depending on the availability of buffer space and which mode the MPI library considers to be the most efficient. Application programmers should not assume buffering or take completion as an indication the receive has been posted.

MPI messaging semantics

- MPI requires the following behaviour for messages
  - Ordered messages
    - Messages sent between 2 end points must be non-overtaking and a receive calls that match multiple messages from the same source should always match the first message sent.
  - Fairness in processing
    - MPI does not guarantee fairness (though many implementations attempt to)
  - Resource limitations
    - There should be a finite limit on the resources required to process each message
  - Progress
    - Outstanding communications should be progressed where possible. In practice this means that MPI needs to process incoming messages from all sources/tags independent of the current MPI call or its arguments.
- These influence the design of MPI implementations.

Blocking/Non-blocking

- MPI defines both blocking and non-blocking calls.

- Most implementations will implement blocking messages as a special case of non-blocking
  - While the application may be blocked the MPI library still has to progress all communications while the application is waiting for a particular message.
  - Blocking calls often effectively map onto pair of non-blocking send/recv and a wait.
  - Though low level calls can be used to skip some of the argument checking.

Message Progression

```
if (rank == 1) MPI_Irecv  (&y, 1, MPI_INT, 0, tag, comm, &req);
if (rank == 0) MPI_Ssend(&x, 1, MPI_INT, 1, tag, comm);
MPI_Barrier(comm);
if (rank == 1) MPI_Wait(&req, &status);
```

- Potential problem if rank 1 does nothing but sit in barrier …
  - Especially if there is only one thread, which is the default situation

Persistence

- MPI standard also defines persistent communications
  - These are like non-blocking but can be re-run multiple times.
- Advantage is that argument-checking and data-type compilation only needs to be done once.
  - Again can often be mapped onto the same set of low level calls as blocking/non-blocking.

```
MPI_Send() {
  MPI_Isend(...,&r);
  MPI_Wait(r);
}
MPI_Isend(...,&r) {
  MPI_Send_init(..., &r);
  MPI_Start(r);
}
```

Derived data-types

- MPI defines a rich set of derived data-type calls.
- In most MPI implementations, derived data-types are implemented by generic code that packs/unpacks data to/from contiguous buffers that are then passed to the ADI calls.
- This generic code should be reasonably efficient but simple application level copy loops may be just as good in some cases.
- Some communication systems support some simple non-contiguous communications
  - Usually no more than simple strided transfer.
  - Some implementations have data-type aware calls in the ADI to allow these cases to be optimised.
  - Though default implementation still packs/unpacks and calls contiguous data ADI.

Protocol messages

- All MPI implementations need a mechanism for delivering packets of data (messages) to a remote process.
  - These may correspond directly to the user's MPI messages or they may be internal protocol messages.
- Whenever a process sends an MPI message to a remote process a corresponding initial protocol message (IPM) must be sent
  - Minimally, containing the envelope information.
  - May also contain some data.
- Many implementations use a fixed size header for all messages
  - Fields for the envelope data
  - Also message type, sequence number etc.

Message Queues

- If the receiving process has already issued a matching receive, the message can be processed immediately
  - If not then the message must be stored in a **foreign-send** queue for future processing.
- Similarly, a receive call looks for matching messages in the foreign-send queue
  - In no matching message found then the receive parameters are stored in a **receive** queue.
- In principle, there could be many such queues for different communicators and/or senders.
  - In practice, easier to have a single set of global queues
  - It makes wildcard receives much simpler and implements fairness

Message protocols

- Typically MPI implementations use different underlying protocols depending on the size of the message.
  - Reasons include, flow-control and limiting resources-per-message
- The simplest of these are
  - Eager
  - Rendezvous
- There are many variants of these basic protocols.

Eager protocol

- The initial protocol message contains the full data for the corresponding MPI message.
- If there is no matching receive posted when IPM arrives then data must be buffered at the receiver.
- Eager/Buffered/Standard sends can complete as soon as the initial protocol message has been sent.
- For synchronous sends, an acknowledge protocol message is sent when the message is matched to a receive. Ssend can complete when this is received.

Resource constraints for eager protocol

- Eager protocol may require buffering at receiving process.
- This violates the resource semantics unless there is a limit on the size of message sent using the eager protocol.
- The exception is for ready messages.
  - As the receive is already posted we know that receive side buffering will not be required.
  - However, implementations can just map ready sends to standard sends.

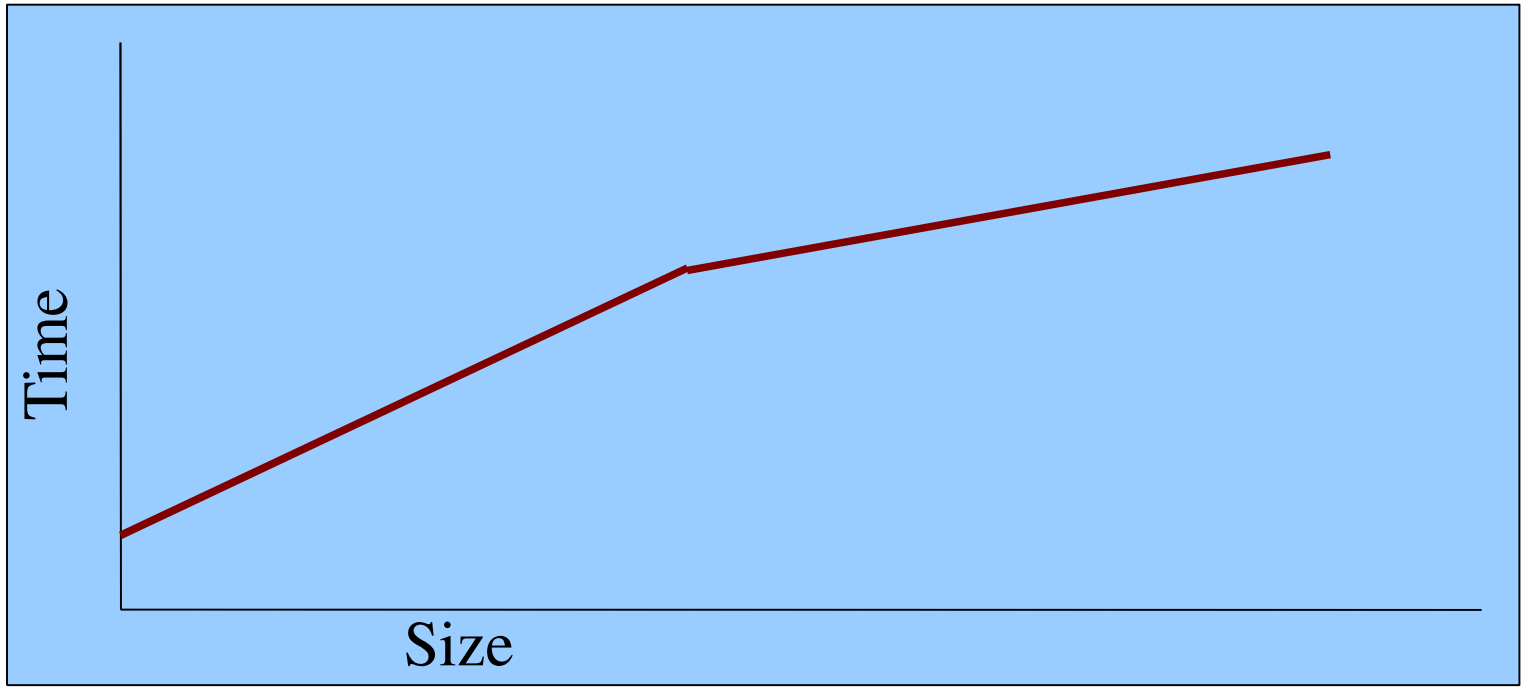
Rendezvous protocol

- IPM only contains the envelope information, no data.
- When this is matched to a receive then a ready-to-send protocol message is returned to the sender.
- Sending process then sends the data in a new message.
- Send acts as a synchronous send (it waits for matching receive) unless the message is buffered on the sending process.

- Note that for very large messages where the receive is posted late Rendezvous can be faster than eager because the extra protocol messages will take less time than copying the data from the receive side buffer.

MPI performance

- When MPI message bandwidth is plotted against message size it is quite common to see distinct regions corresponding to the eager/rendezvous protocols



Other protocol variants

- Short message protocol
  - Some implementations use a standard size header for all messages.
  - This header may contain some fields that are not defined for all types of protocol message.
  - Short message protocol is a variant of eager protocol where very small messages are packed into unused fields in the header to reduce overall message size.
- DMA protocols
  - Some communication hardware allows Direct Memory Access (DMA) operations between different processes that share memory.
  - Direct copy of data between the memory spaces of 2 processes.
  - Protocol messages used to exchange addresses and data is copied direct from source to destination. Reduces overall copy overhead.
  - Some systems have large set-up cost for DMA operations so these are only used for very large messages.

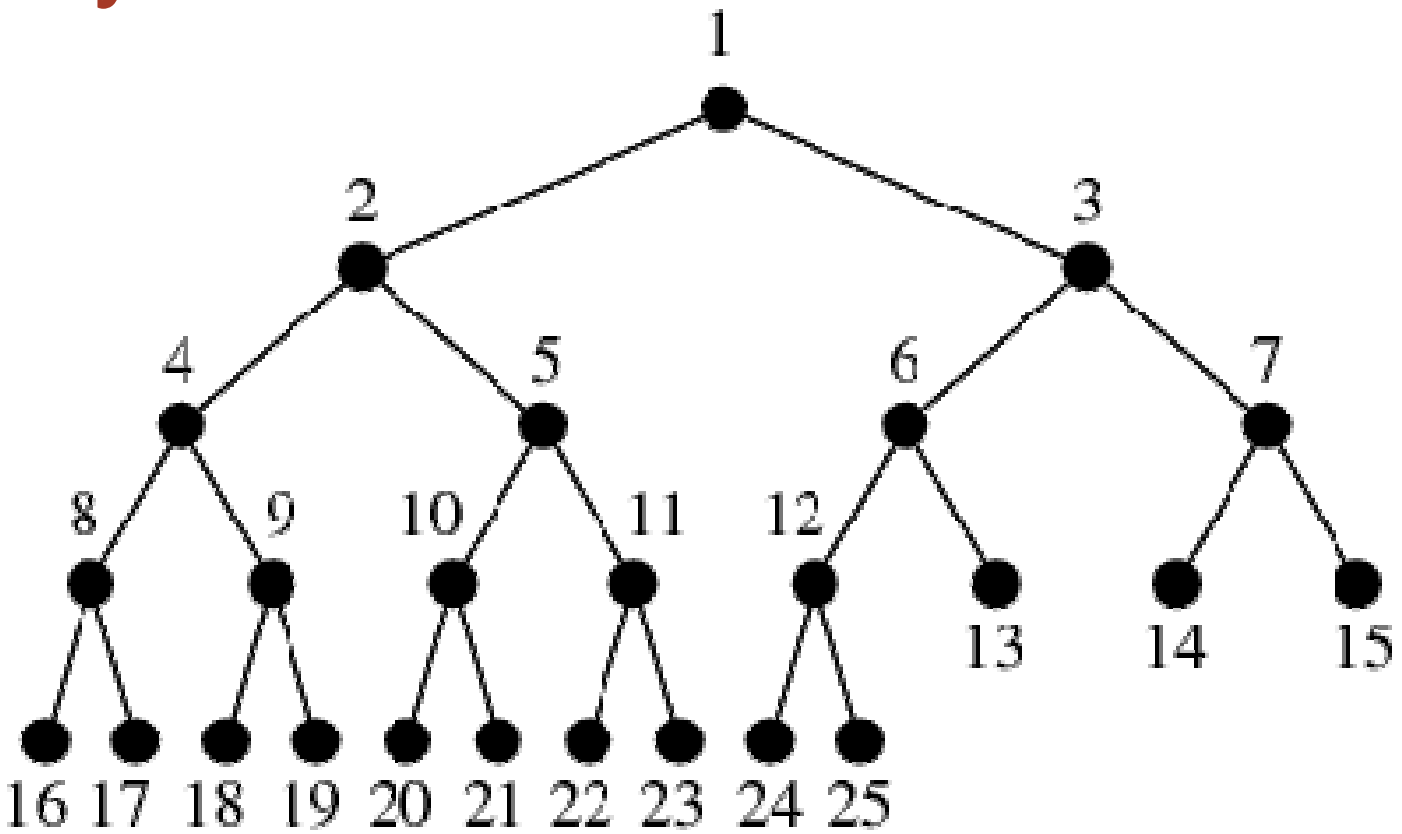
Collective Communication

- Collective communication routines are sometimes built on top of MPI point to point messages.
  - In this case the collectives are just library routines.
  - You could re-implement them yourself. But:
    - The optimal algorithms are quite complex and non-intuitive.
    - Hopefully somebody else will optimise them for each platform.
- There is nothing in the API that requires this however.
  - The collective routines give greater scope to library developers to utilise hardware features of the target platform.
    - Barrier synchronisation hardware
    - Hardware broadcast/multicast
    - Shared memory nodes
    - etc.

Collective algorithms

- There are many possible ways of implementing collectives.
  - Best choice depends on the hardware.
- For example consider MPI_Allreduce
- Good first attempt is to build a binary tree of processors.
  - Completes in $O(\log_2(P))$ communication steps.
    - Data is sent up the tree with partial combine at each step
    - Result is then passed (broadcast) back down tree.
  - $2 * \log_2(P)$ steps in total.
- If network can broadcast (includes shared memory) then result can be distributed in a single step.
- Note that most processors spend most of the time waiting.
  - For a vector all-reduce can be better to split the vector into segments and use multiple (different) trees for better load balance.
  - Also, what about a binomial tree or a hypercube algorithm?

Binary Tree



- From http://mathworld.wolfram.com/

Groups and Communicators

- Logically communicators are independent communication domains
  - Could use separate message queues etc. to speed up matching process.
  - In practice most application codes use very few communicators at a time.
- Most MPI implementations use a “native” processor addressing for the ADI
  - Often the same as MPI_COMM_WORLD ranks.
  - Communicators/Groups generic code at the upper layers of the library.
  - Need an additional hidden message tag corresponding to communicator id (often called a context id).

Process Topologies

- Topology code gives the library writer the opportunity to optimise process placement w.r.t. machine topology.
- In practice, some implementations use generic code and don't attempt to optimise.
- Major implementations make some attempt to optimise.
  - May not do a very good job in all situations.

Single Sided

- MPI-2 added single sided communication routines.
- Very complex set of APIs (also quite ambiguous in some places)
- Complexity is necessary for correctness with least restrictions
  - In practice, use simpler (and more restrictive) rules-of-thumb
- Most major MPI implementations now support MPI-3 memory model and single-sided operations.
  - Probably only want to use them if it makes programming easier.
- For many applications, single-sided will perform slower than normal point to point.

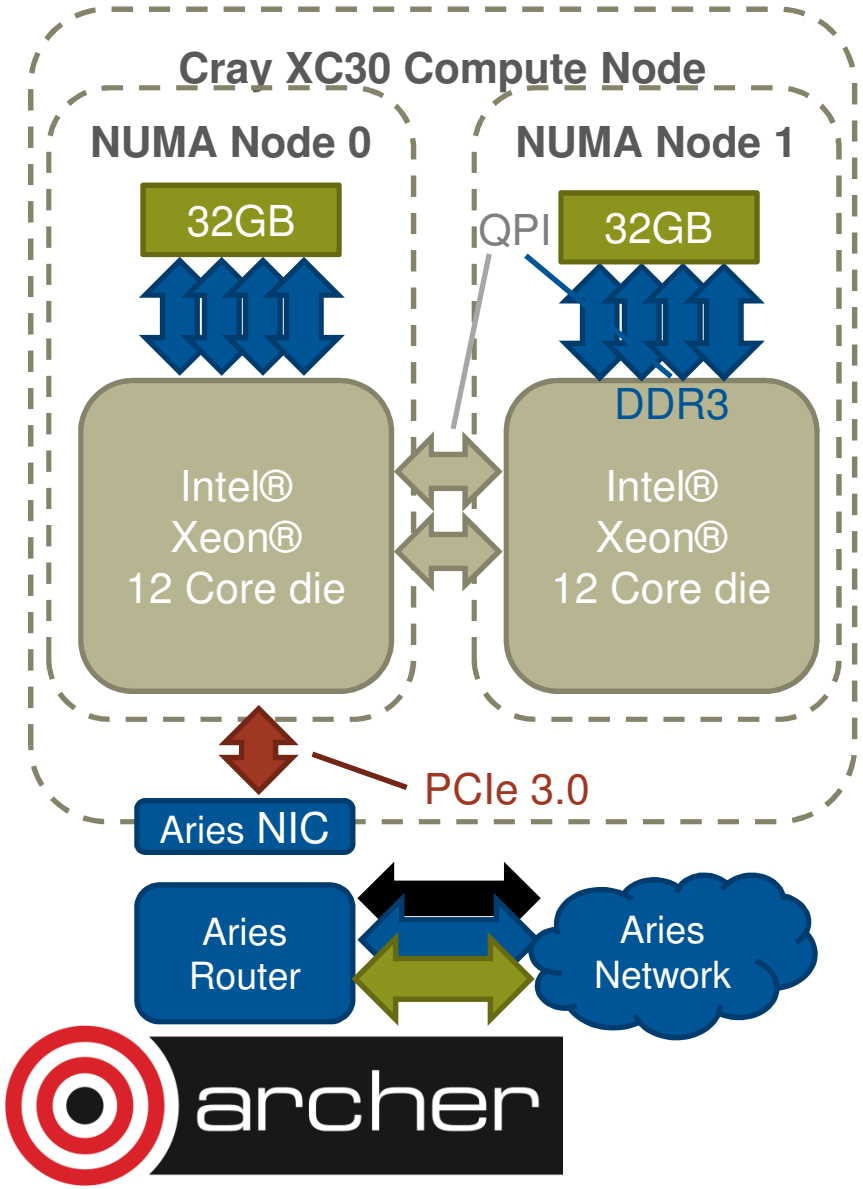
Implementing Single sided

- Like most of the MPI standard the single-sided calls allow great freedom to the implementer
- Single-sided operations must only target addresses in previously created “windows”
  - Creation operation is collective.
  - This is to allow MPI to map the window into the address space of other processors if HW allows.
- The single sided calls consist of RMA calls and synchronisation calls.
  - The results of the RMA calls are not guaranteed to be valid until synchronisation takes place.
  - In the worst case, MPI is allowed to just remember what RMA calls were requested then perform the data transfers using point-to-point calls as part of the synchronisation.
  - Naturally implementable if hardware supports RDMA, e.g. Infiniband

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

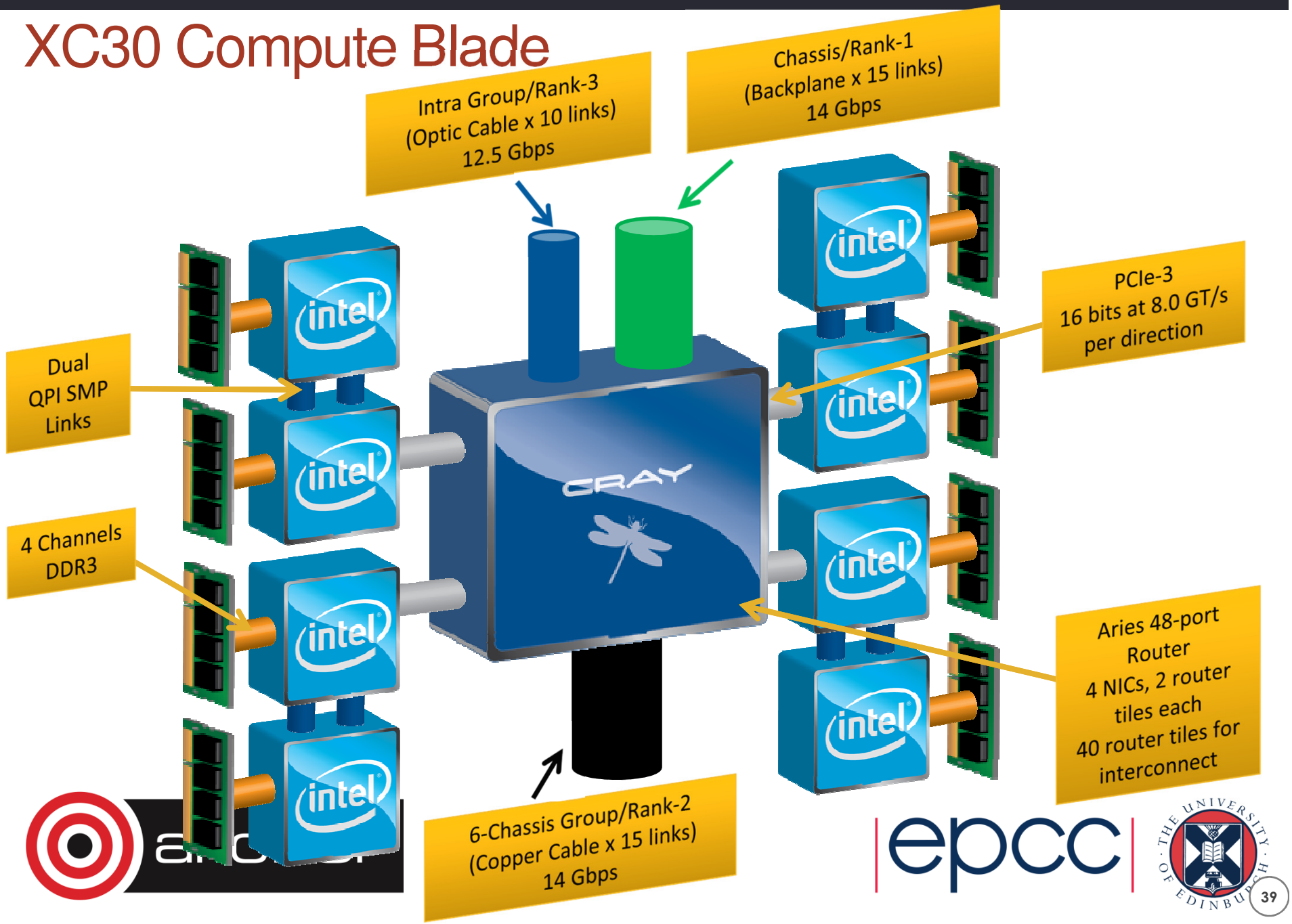


The XC30 Compute node features:

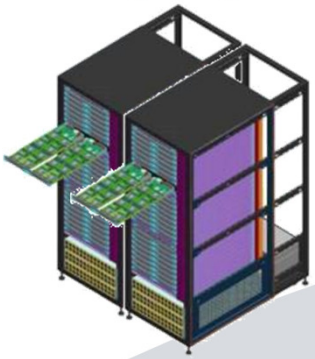
- 2 x Intel® Xeon® Sockets/die
  - 12 core Ivy Bridge
  - QPI interconnect
  - Forms 2 NUMA nodes
- 8 x 1833MHz DDR3
  - 8 GB per Channel
  - 64/128 GB total
- 1 x Aries NIC
  - Connects to shared Aries router and wider network
  - PCI-e 3.0

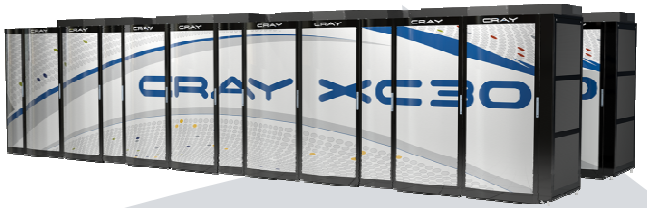
epcc | THE UNIVERSITY OF EDINBURGH

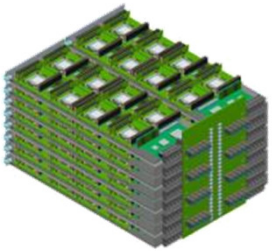
XC30 Compute Blade

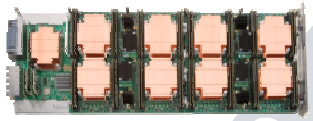


ARCHER System Building Blocks









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

archer

epcc

THE UNIVERSITY OF EDINBURGH

Architectural features relevant to MPI

- In principle expect hierarchy of MPI performance between

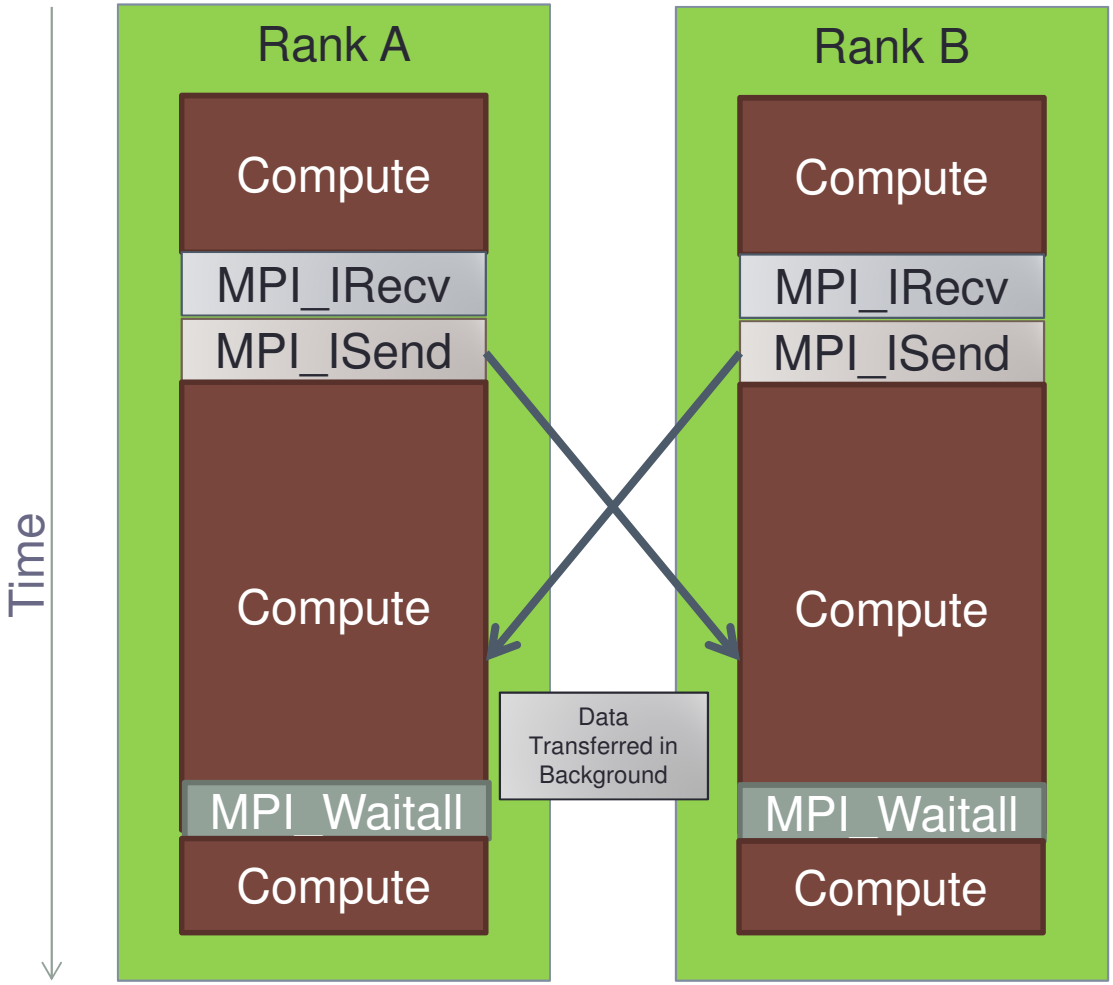
1) two hyperthreads on the same core
2) two cores on the same NUMA region but different cores
3) two cores on the same node but different NUMA regions
4) two cores on the same blade but different nodes
5) two cores on the same chassis but different blades
6) two cores on the same group but different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

- In practice levels 4 – 7 are basically the same
  - As with most system, 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.

archer

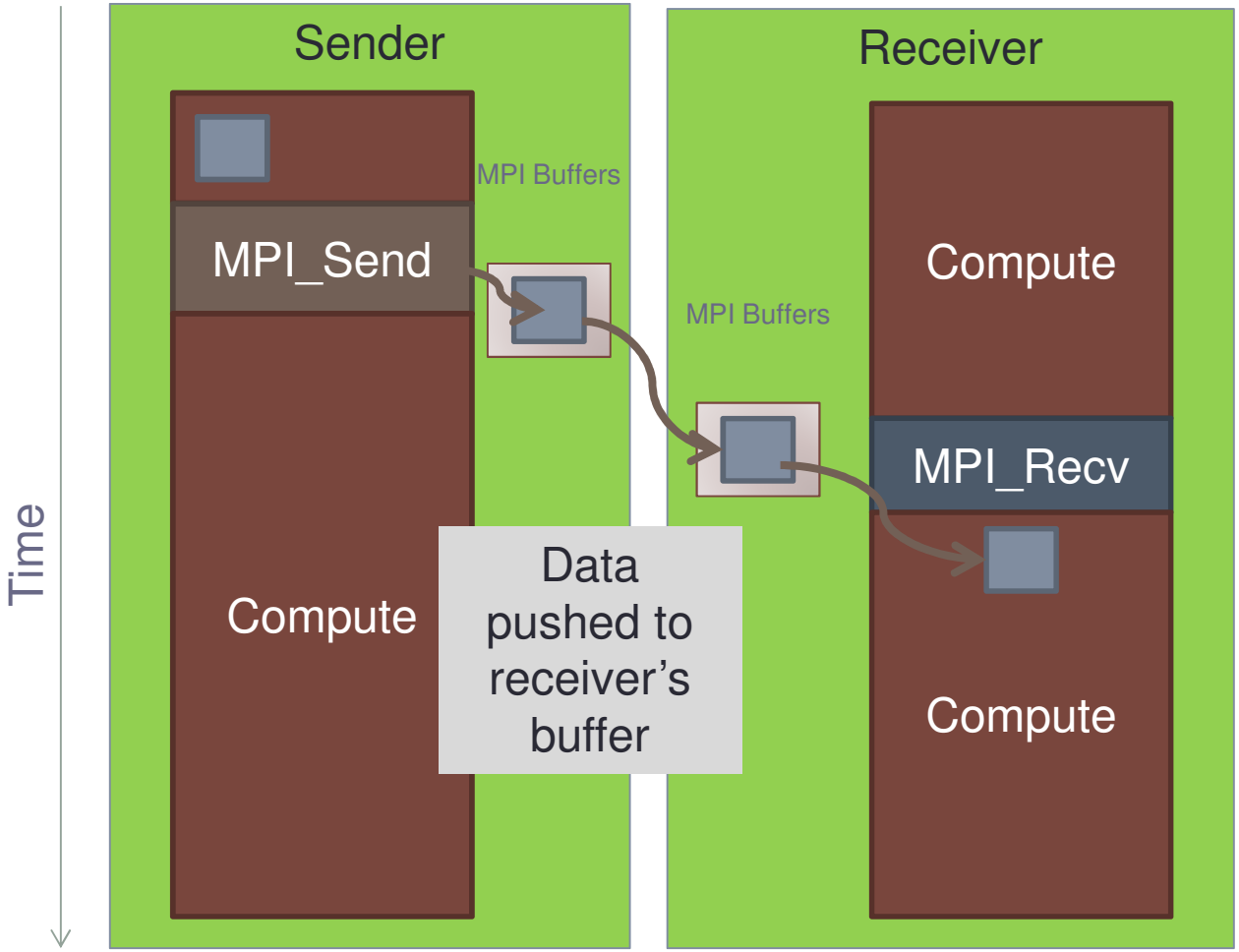
epcc

THE UNIVERSITY OF EDINBURGH

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 Aries, complex tasks like matching message tags with the sender and receiver are performed by the host CPU. This means:
  + 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.

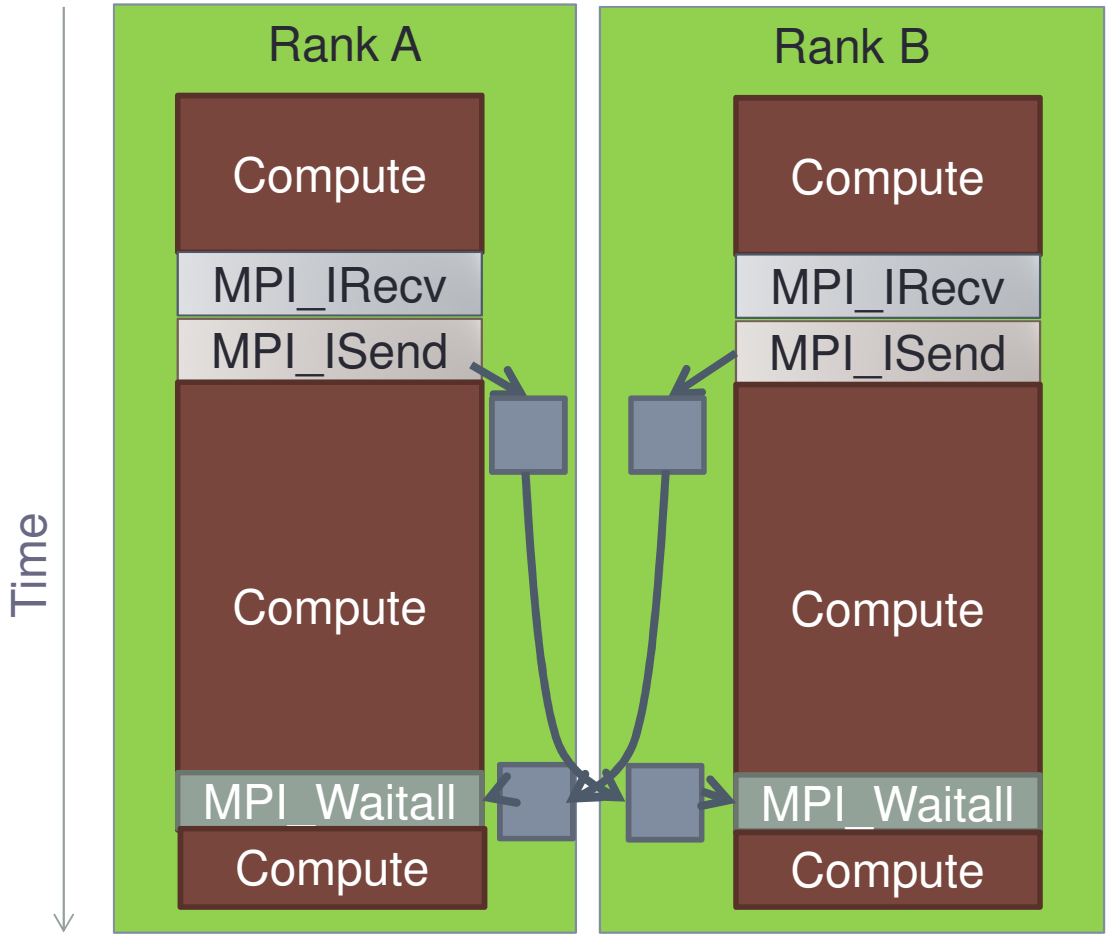
Sender will block if there are no free buffers

archer

epcc

THE UNIVERSITY OF EDINBURGH

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.

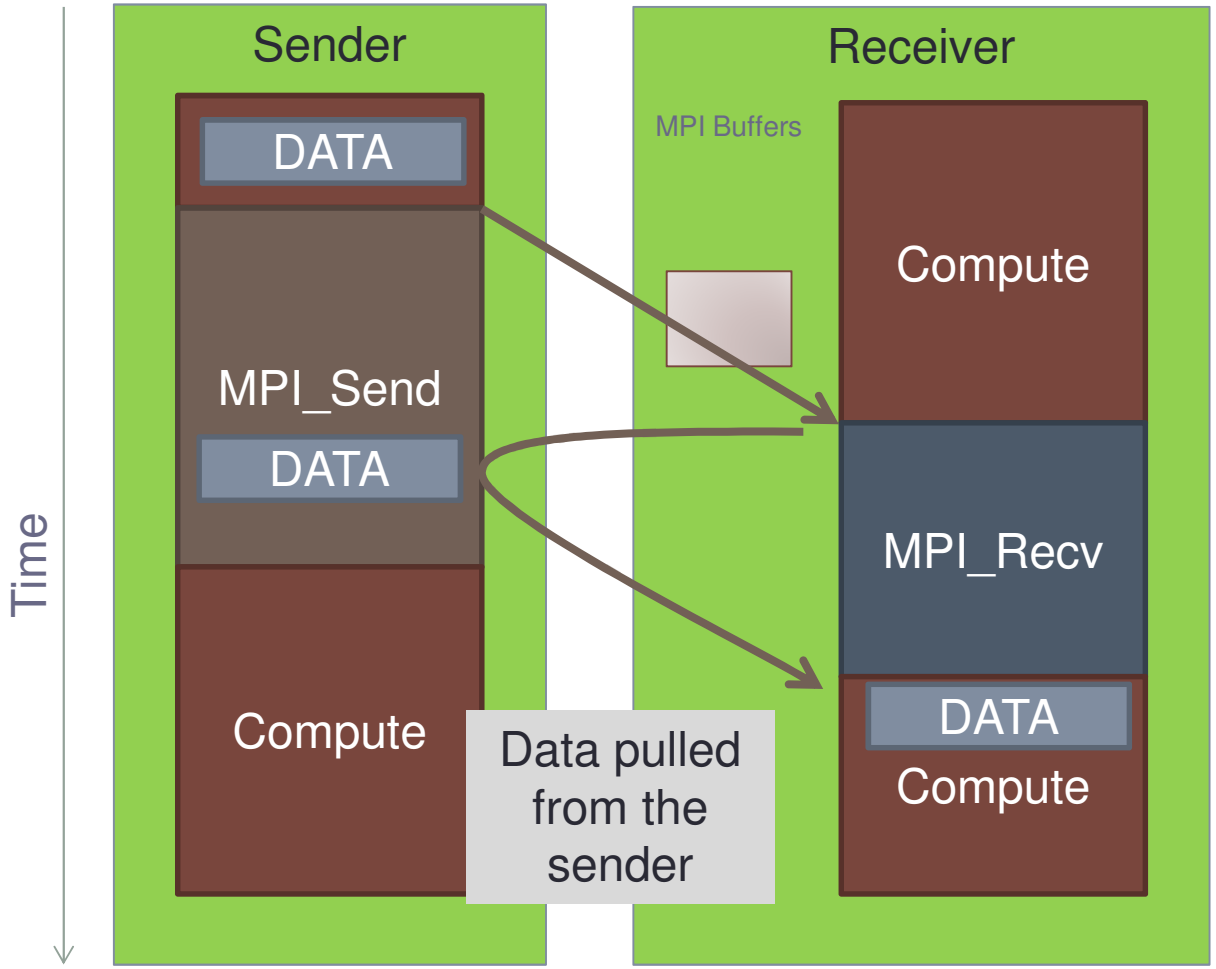
Involves an extra memcopy, but much greater opportunity for overlap of computation and communication.

archer

epcc

THE UNIVERSITY OF EDINBURGH

RENDEZVOUS Messaging – Larger Messages



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.

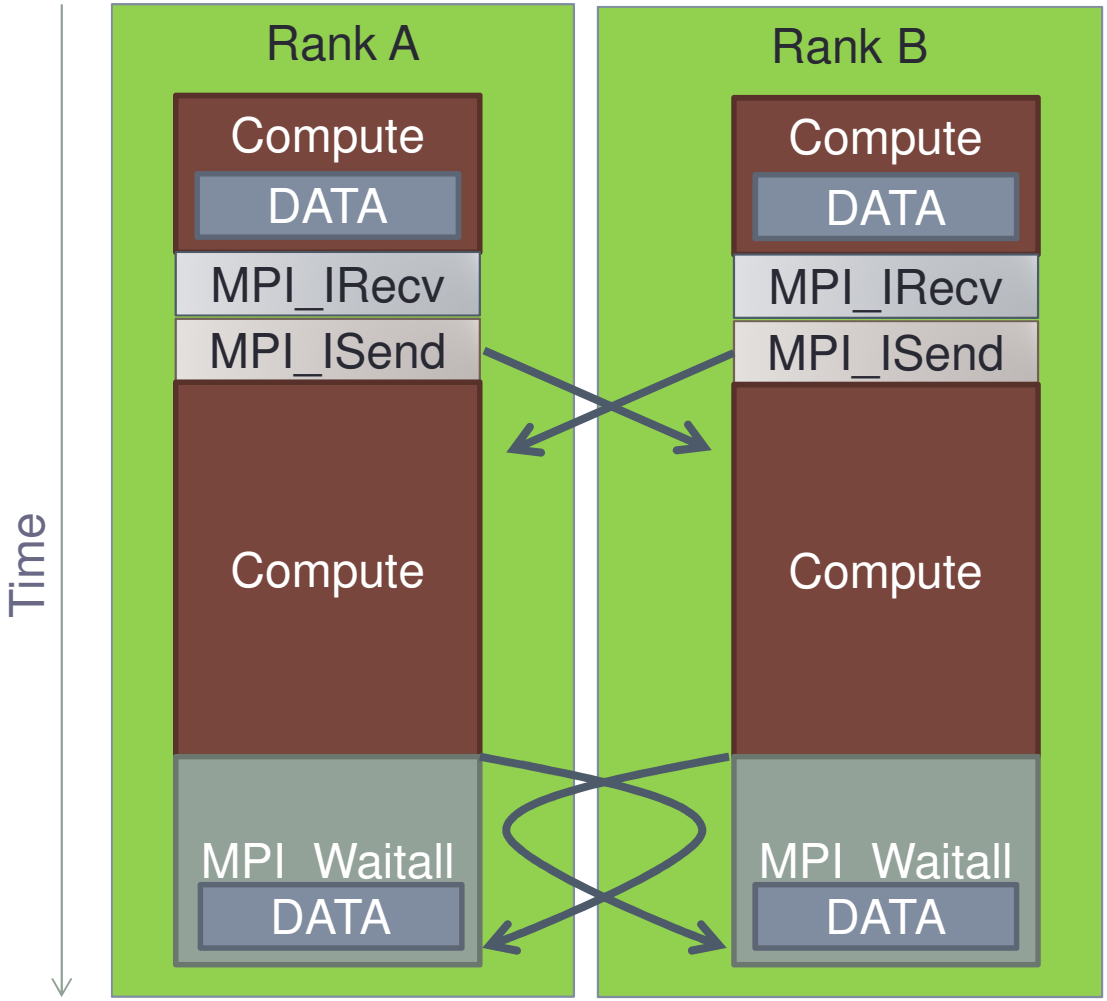
Sender and Receiver block until communication is finished

archer

epcc

THE UNIVERSITY OF EDINBURGH

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.

archer

epcc

THE UNIVERSITY OF EDINBURGH

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.