MPI Internals

Advanced Message-Passing Programming

Reusing this material

Overview

- MPI Library Structure
- Point-to-point
- Collectives
- Group/Communicators
- Single-sided

MPI Structure

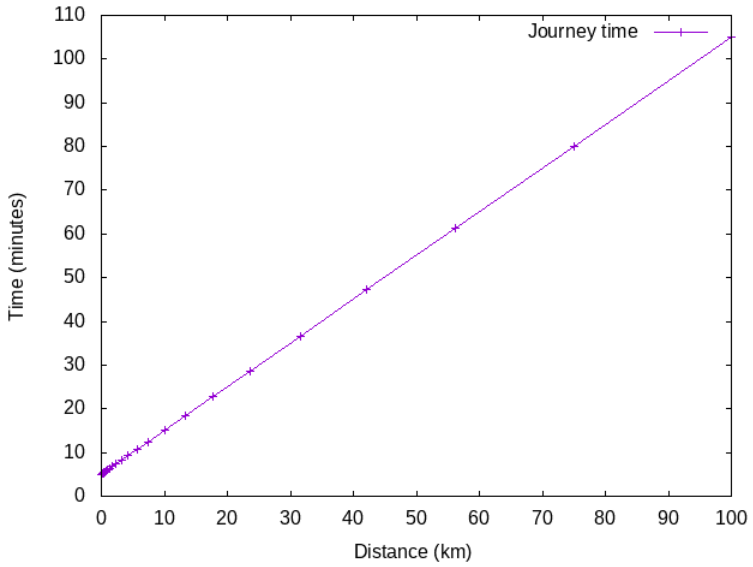
- Like any large software package MPI implementations need to be split into modules.
- MPI has a fairly natural module decomposition roughly following the chapters of the MPI Standard.
  - Point to Point
  - Collectives
  - Groups Contexts Communicators
  - Process Topologies
  - Process creation
  -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

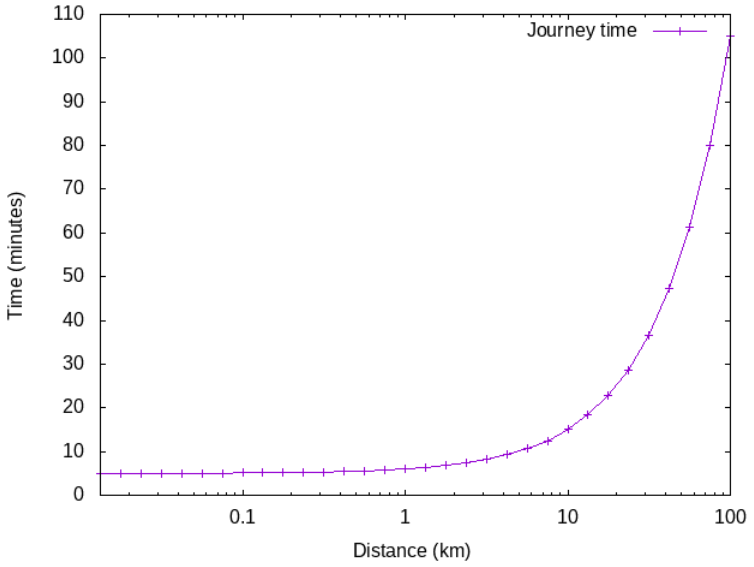
Point to Point

- 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 standard communications calls
  - Actually almost all libraries use the portable 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).

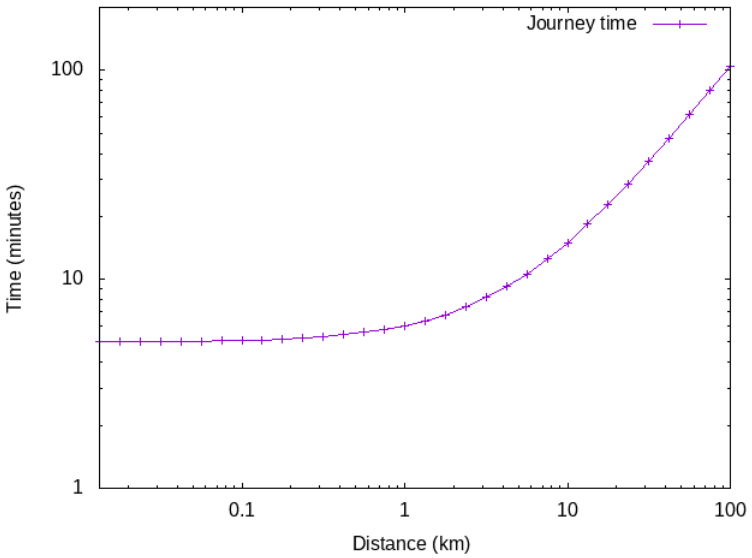
Simple Performance Model

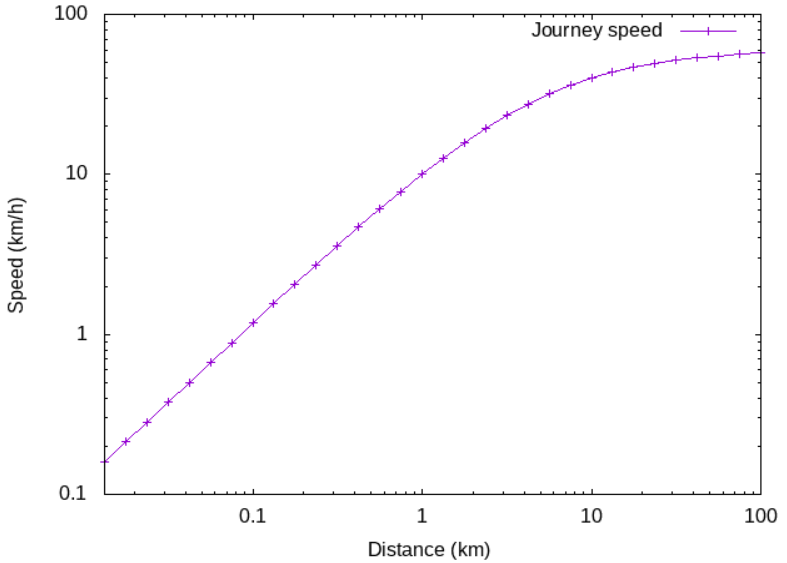
- Analogy: car journey
  - my car is parked in nearby garage: takes 5 minutes to walk there
  - after that, average 60 km/h when driving
  - how long does a journey take? what is my average speed?





Small and large distance behaviour

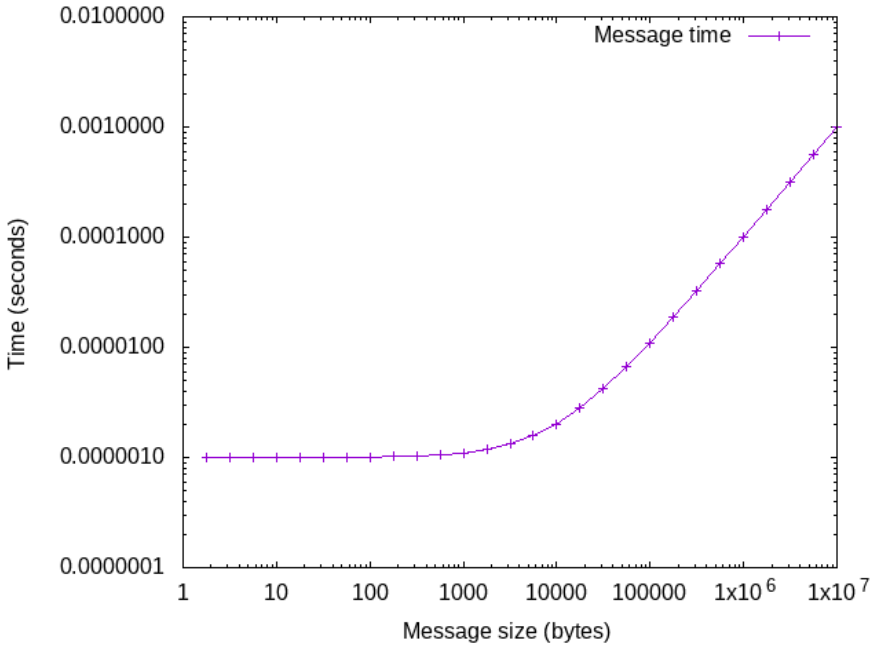


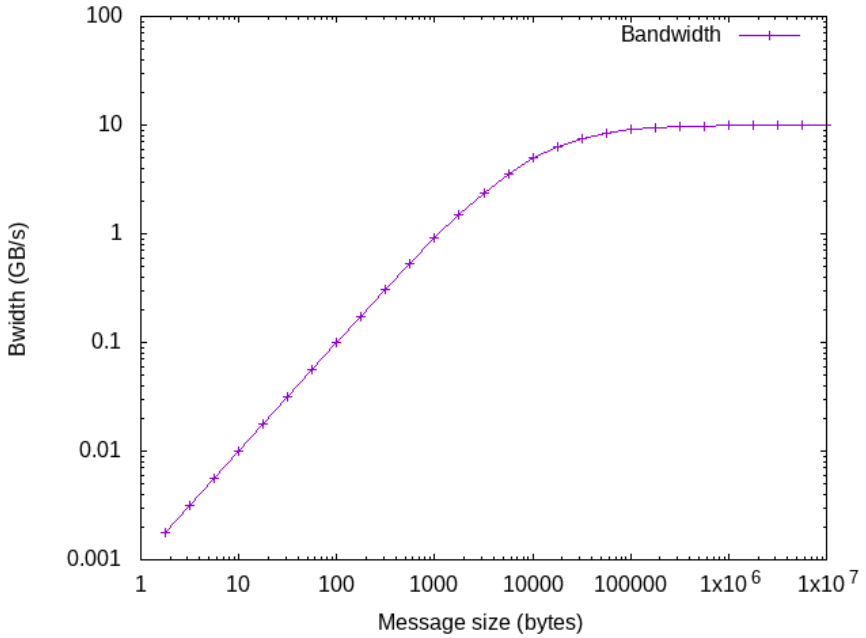


- Small distances take constant *time* of 5 minutes
  - long distances achieve a constant *speed* of 60 km/h
- Crossover at $N_{1/2}$ km with equal time driving as walking
  - $N_{1/2}$ = 60 km/h * 5 minutes = 5 km

For point-to-point MPI

- *Latency* is overhead of sending any message
  - time to get very short message onto network (e.g. 1 microsecond)
- *Bandwidth* is rate of transfer on the network (e.g. 10 GB/s)
  - here $N_{1/2}$ = 10 GB/s * 1 microsecond = 10 KB





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

MPI communication modes (ii)

- MPI defines multiple types of send
  - Ready
    - Ready sends can only be started if the receive is known to be already posted (it's 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 (but many implementations attempt to)
  - Resource limitations
    - 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.

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

Blocking/Non-blocking

- MPI defines both blocking and non-blocking calls.
- Most implementations will implement blocking messages as a special case of non-blocking
  - Low-level network protocols likely to be 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.

Point-to-point example

- MPI_Irecv(up_neighbour, ..., &requests[0])
- MPI_Irecv(down_neighbour, ..., &requests[1])
- MPI_Ssend(down_neighbour, ...)
- MPI_Ssend(up_neigbour, ...)
- MPI_Wait(&requests[0], ...)
- MPI_Wait(&requests[1], ...)

- This takes advantage of message order preservation to ensure correct matching

Persistence

- MPI standard also defines persistent communications
  - These are like non-blocking but can be re-run multiple times.
  - Advantage is that argument-checking/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.

- Many more optimisations possible in collectives
  - persistent collectives included in MPI 4.0

Point-to-point persistent calls

- MPI_Recv_init(up_neighbour, ..., &requests[0])
- MPI_Recv_init(down_neighbour, ..., &requests[1])
- MPI_Send_init(down_neighbour, ..., &requests[2]);
- MPI_Send_init(up_neighbour, ..., &requests[3]);

- loop over many iterations ...
  - MPI_Startall(4, requests);
  - ...
  - MPI_Waitall(4, requests, statuses);

- Ordering of individual calls not guaranteed
  - may need to be careful about message matching (e.g. use tags)

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.

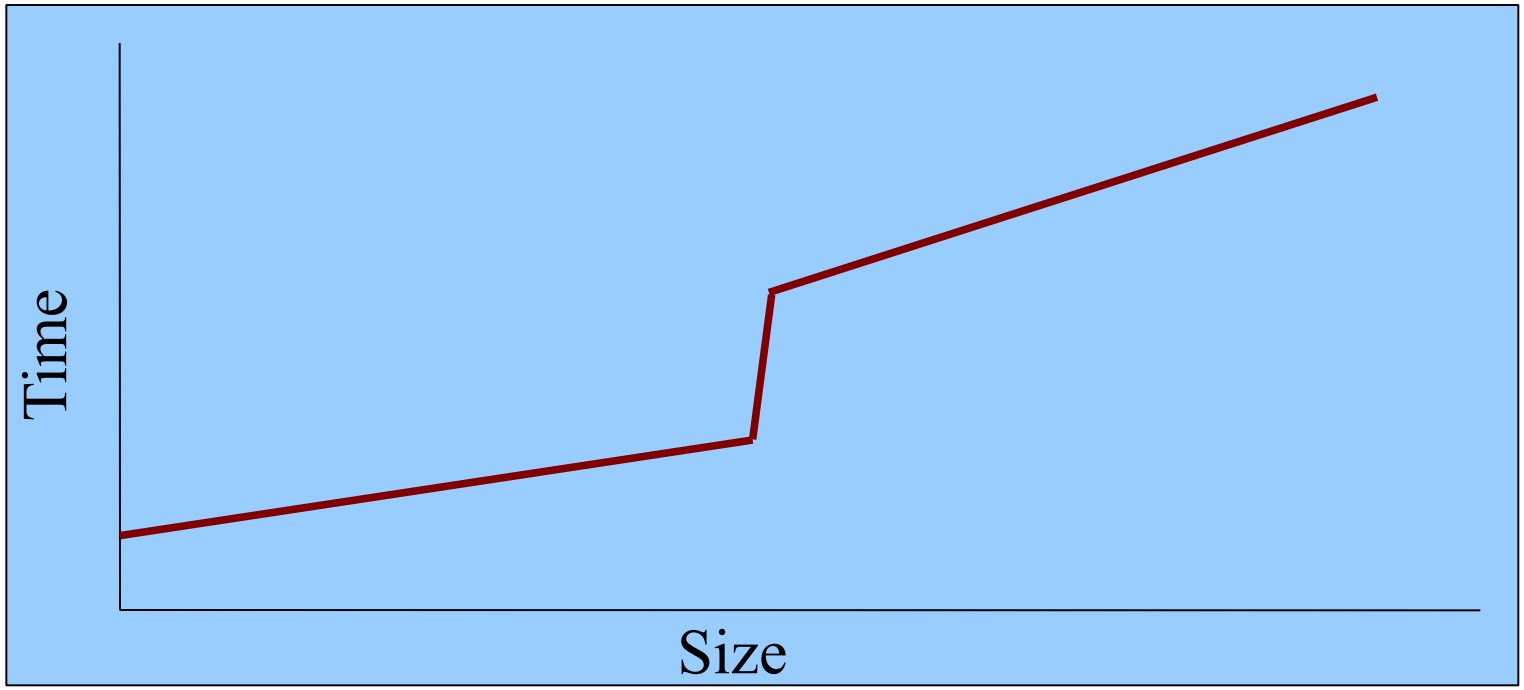
Rendezvous protocol

- 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 time 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.

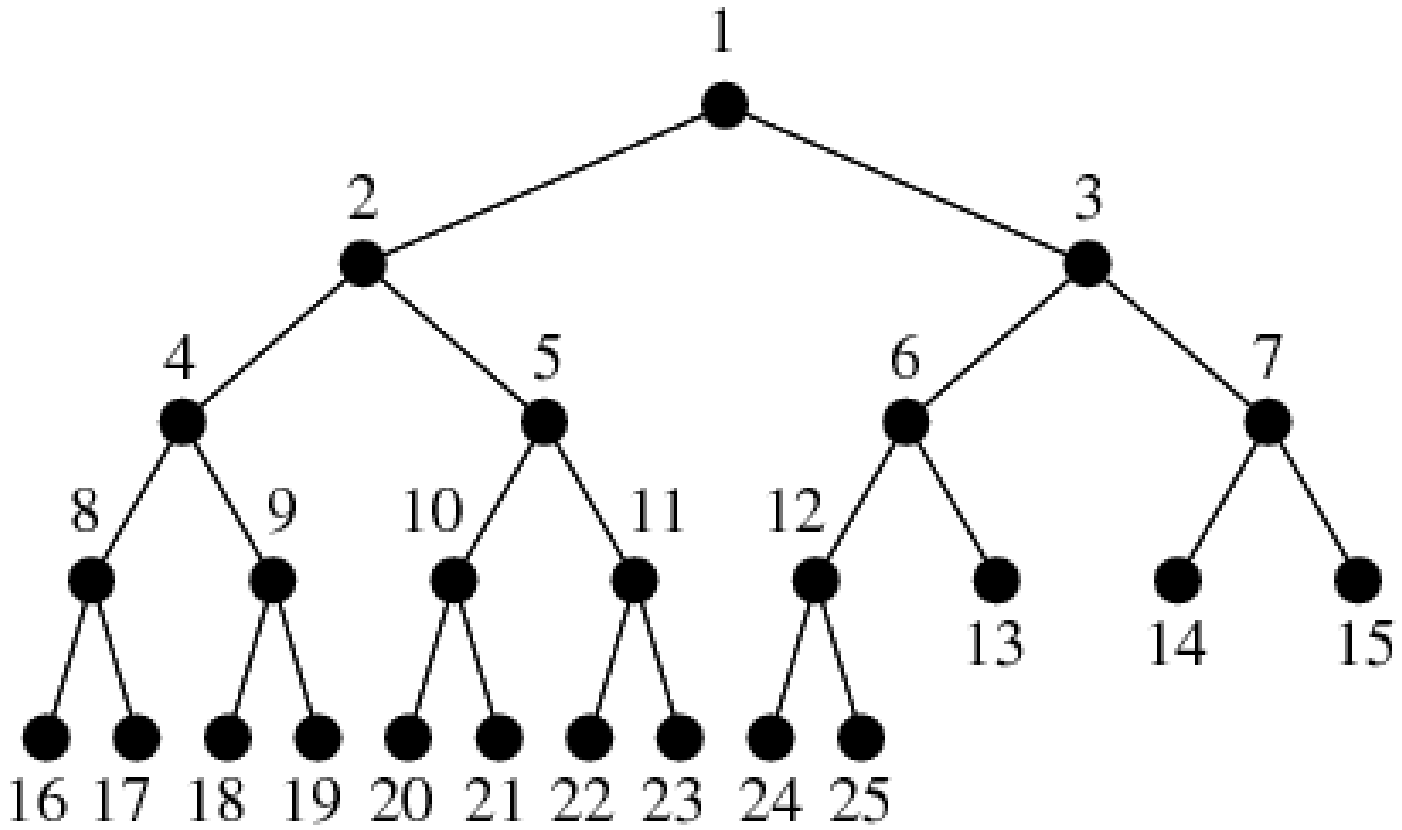
Collective Communication

- 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.

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.

- Process topologies perhaps most useful with neighbourhood collectives
  - communication pattern implied by the topology
  - e.g. “each processor exchanges data with all its neighbours”

Single Sided

- MPI-2 added single-sided communication routines.
  - memory “windows” replace communicators in function calls
  - added remote write (MPI_Put) and remote read (MPI_Get)
- Very complex set of APIs (also quite ambiguous in some places)
- For many applications, single-sided performed slower than normal point to point.

- Most major MPI implementations now support MPI-3 memory model and single-sided operations.
  - now allows direct read and write to memory windows on a node

Summary

- There are many ways of specifying MPI communication
- Which one is best (fastest) depends on the implementation
  - Which MPI library are you using?
  - Which hardware are you using?
  - Which options are you using?
- Performance portability is, therefore, really hard
  - Implement lots of different methods
  - Test all of them in each new situation
  - Pick the best one for each situation