MPI Optimisation

Advanced Parallel Programming

David Henty, Iain Bethune, Dan Holmes
EPCC, University of Edinburgh

Overview

Can divide overheads up into four main categories:

- Lack of parallelism
- Load imbalance
- Synchronisation
- Communication

Lack of parallelism

- Tasks may be idle because only a subset of tasks are computing
- Could be one task only working, or several.
  - work done on task 0 only
  - with split communicators, work done only on task 0 of each communicator
- Usually, the only cure is to redesign the algorithm to exploit more parallelism.

Extreme scalability

- Note that sequential sections of a program which scale as O(p) or worse can severely limit the scaling of codes to very large numbers of processors.
- Let us assume a code is perfectly parallel except for a small part which scales as O(p)
  - e.g. a naïve global sum as implemented for the MPP pi example!
- Time taken for parallel code can be written as

$$T_p = T_s((1-a)/p + ap)$$

where Ts is the time for the sequential code and a is the fraction of the sequential time in the part which is O(p).

- Compare with Amdahl's Law

$$T_p = T_s((1-a)/p + a)$$

For example, take a = 0.0001

For 1000 processors, Amdahl's Law gives a speedup of ~900

For an O(p) term, the maximum speedup is ~50 (at p =100).

- Note: this assume strong scaling, but even for weak scaling this will become a problem for 10,000+ processors

WolframAlpha

- O(1) term in scaling with a=0.0001 assuming strong-scaling (Amdahl's law):
  - http://www.wolframalpha.com/input/?i=maximum+1%2F%28%281-0.0001%29%2Fp%2B0.0001%29+with+p+from+1+to+100000
- O(p) term in scaling with a=0.0001 assuming strong-scaling:
  - http://www.wolframalpha.com/input/?i=maximum+1%2F%28%281-0.0001%29%2Fp%2B0.0001+*+p%29+with+p+from+1+to+1000
- O(p) term in scaling with a=0.0001 assuming weak-scaling:
  - http://www.wolframalpha.com/input/?i=maximum+1%2F%28%281-0.0001%29%2B0.0001+*+p%29+with+p+from+1+to+10000
- O(log2(p)) term in scaling with a=0.0001 assuming strong-scaling:
  - http://www.wolframalpha.com/input/?i=maximum+1%2F%28%281-0.0001%29%2Fp%2B0.0001+*+log2%28p%2B1%29%29+with+p+from+1+to+100000
- O(log2(p)) term in scaling with a=0.0001 assuming weak-scaling:
  - http://www.wolframalpha.com/input/?i=maximum+1%2F%28%281-0.0001%29%2B0.0001+*+log2%28p%2B1%29%29+with+p+from+1+to+100000
- O(log2(p)/p) term in scaling with a=0.0001 assuming strong-scaling:
  - http://www.wolframalpha.com/input/?i=maximum+1%2F%28%281-0.0001%29%2Fp%2B0.0001+*+log2%28p%2B1%29%2Fp%29+with+p+from+1+to+100000

Load imbalance

- All tasks have some work to do, but some more than others....
- In general a much harder problem to solve than in shared variables model
  - need to move data explicitly to where tasks will execute
- May require significant algorithmic changes to get right
- Again scaling to large processor counts may be hard
  - the load balancing algorithms may themselves scale as O(p) or worse.
- We will look at some techniques in more detail later in the module.

- MPI profiling tools report the amount of time spent in each MPI routine
- For blocking routines (e.g. Recv, Wait, collectives) this time may be a result of load imbalance.
- The task is blocked waiting for another task to enter the corresponding MPI call
  - the other tasks may be late because it has more work to do
- Tracing tools often show up load imbalance very clearly
  - but may be impractical for large codes, large task counts, long runtimes

Synchronisation

- In MPI most synchronisation is coupled to communication
  - Blocking sends/receives
  - Waits for non-blocking sends/receives
  - Collective comms are (mostly) synchronising
- MPI_Barrier is almost never required for correctness
  - can be useful for timing
  - can be useful to prevent buffer overflows if one task is sending a lot of messages and the receiving task(s) cannot keep up.
  - think carefully why you are using it!
- Use of blocking point-to-point comms can result in unnecessary synchronisation.
  - Can amplify “random noise” effects (e.g. OS interrupts)
  - see later

Communication

- Point-to-point communications
- Collective communications
- Task mapping

Small messages

- Point to point communications typically incur a start-up cost
  - sending a 0 byte message takes a finite time
- Time taken for a message to transit can often be well modeled as

$$T = T_l + N_b T_b$$

where T_l is start-up cost or *latency*, N_b is the number of bytes sent and T_b is the time per byte. In terms of *bandwidth* B:

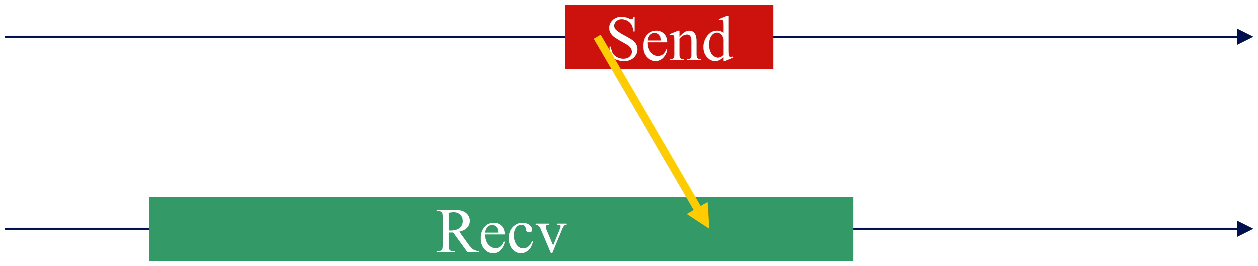
$$T = T_l + N_b/B$$

- Faster to send one large message vs many small ones
  - e.g. one allreduce of two doubles vs two allreduces of one double
  - derived data-types can be used to send messages with a mix of types

Communication patterns

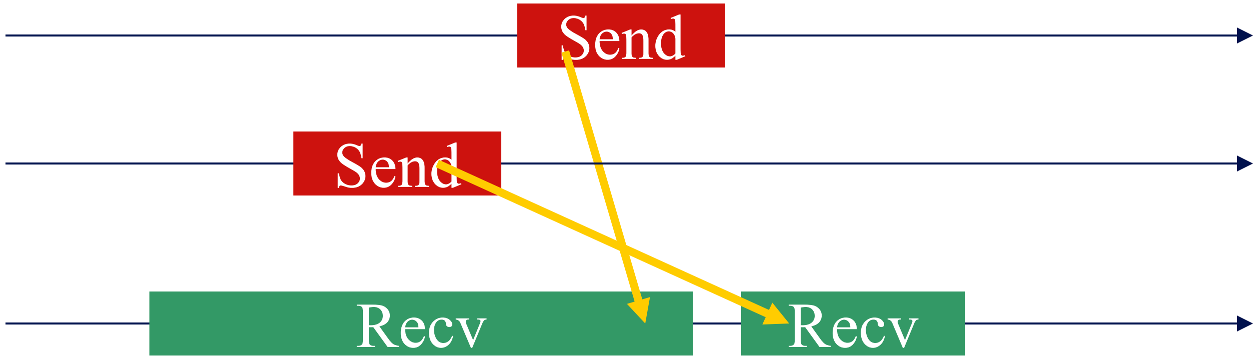
- Can be helpful, especially when using trace analysis tools, to think about communication patterns
  - Note: nothing to do with OO design!
- We can identify a number of patterns which can be the cause of poor performance.
- Can be identified by eye, or potentially discovered automatically
  - e.g. the SCALASCA tool highlights common issues

Late Sender

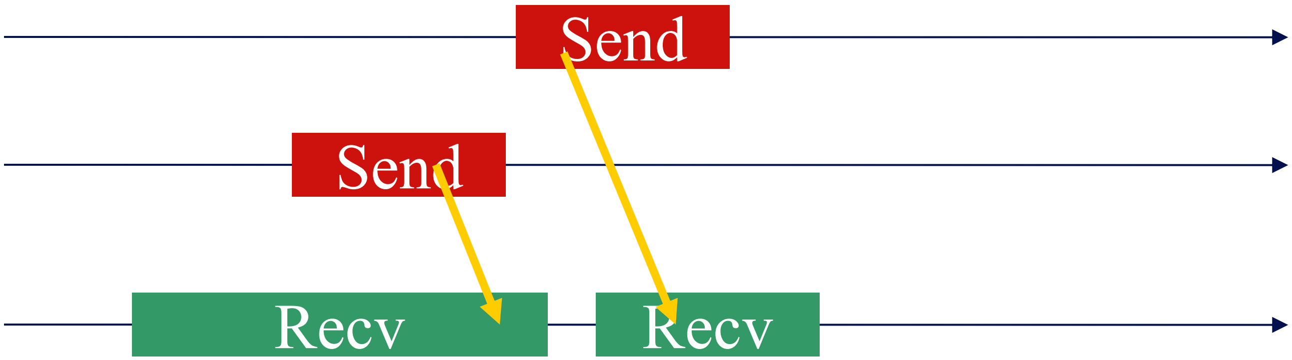


- If blocking receive is posted before matching send, then the receiving task must wait until the data is sent.

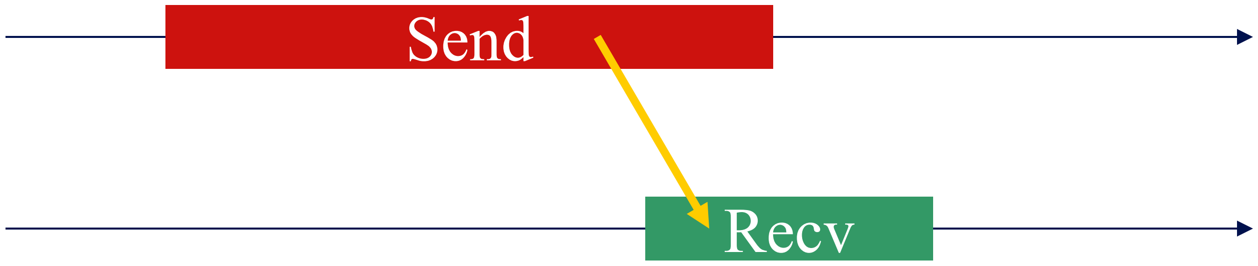
Out-of-order receives



- Late senders may be the result of having blocking receives in the wrong order.

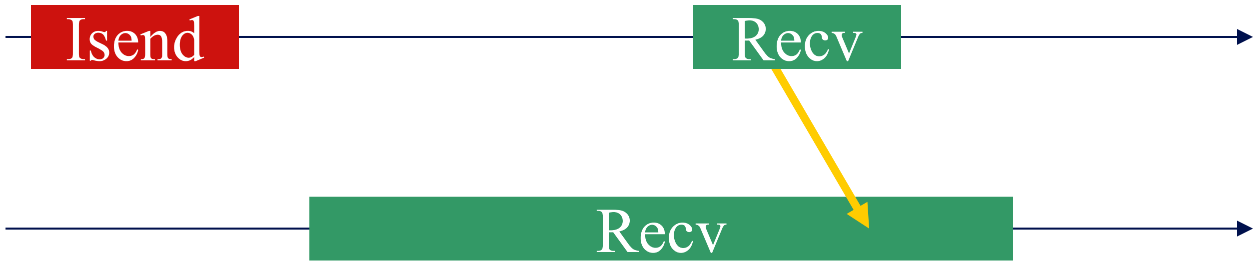


Late Receiver



- If send is synchronous, data cannot be sent until receive is posted
  - either explicitly programmed, or chosen by the implementation because message is large
  - sending task is delayed

Late Progress



- Non-blocking send returns, but implementation has not yet sent the data.
  - A copy has been made in an internal buffer
- Send is delayed until the MPI library is re-entered by the sender.
  - receiving task waits until this occurs

Non-blocking comms

- Both late senders and late receivers may be avoidable by more careful ordering of computation and communication
- However, these patterns can also occur because of “random noise” effects in the system (e.g. network congestion, OS interrupts)
  - not all tasks take the same time to do the same computation
  - not all messages of the same length take the same time to arrive
- Can be beneficial to avoid blocking by using all non-blocking comms entirely (Isend, Irecv, WaitAll)
  - post all the Irecv’s as early as possible

Halo swapping

```
loop many times:
     irecv up; irecv down
     isend up; isend down
     update the middle of the array
     wait for all 4 communications
     do all calculations involving halos
end loop
```

- Receives not necessarily ready in advance
  - remember your recv’s match *someone else’s* sends!

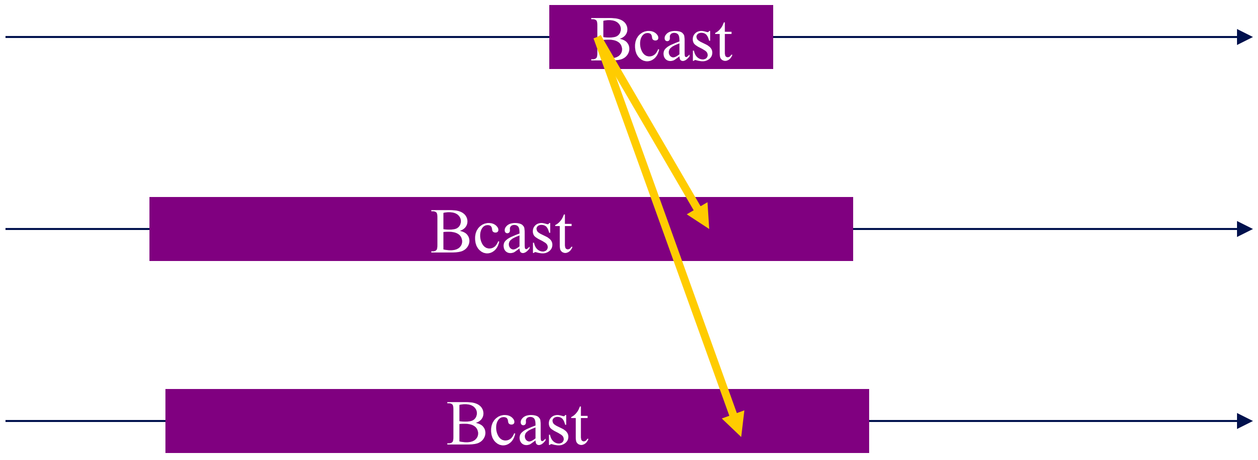
Halo swapping (ii)

```
loop many times:
      wait for even-halo irecv operations
      wait for odd-halo isend operations
      update "out" odd-halo using "in" even-halo
      irecv even-halo up; irecv even-halo down
      isend odd-halo up; isend odd-halo down
      update the middle of the array
      wait for odd-halo irecv operations
      wait for even-halo isend operations
      update "out" even-halo using "in" odd-halo
      irecv odd-halo up; irecv odd-halo down
      isend even-halo up; isend even-halo down
      update the middle of the array
end loop
```

Collective communications

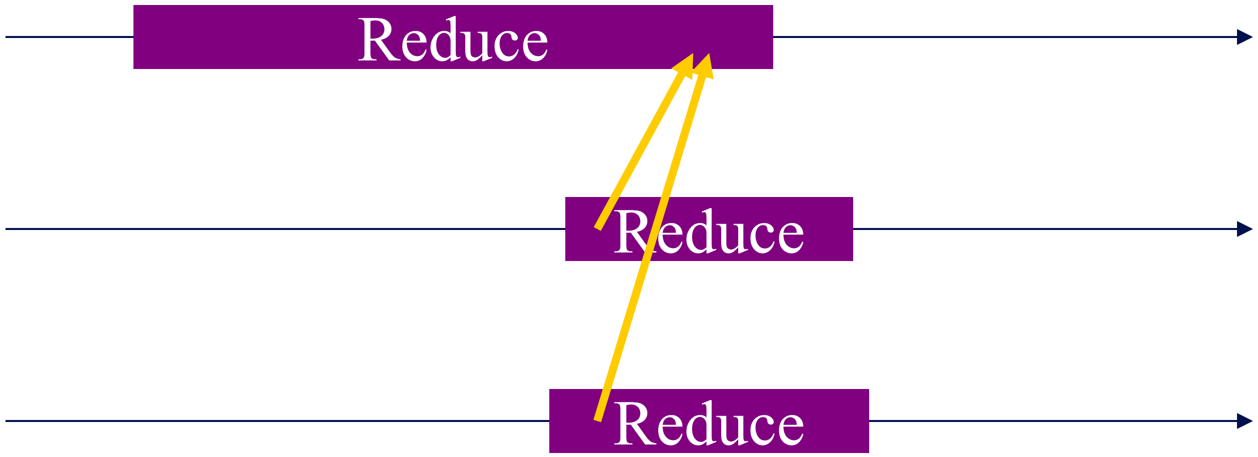
- Can identify similar patterns for collective comms...

Late Broadcaster



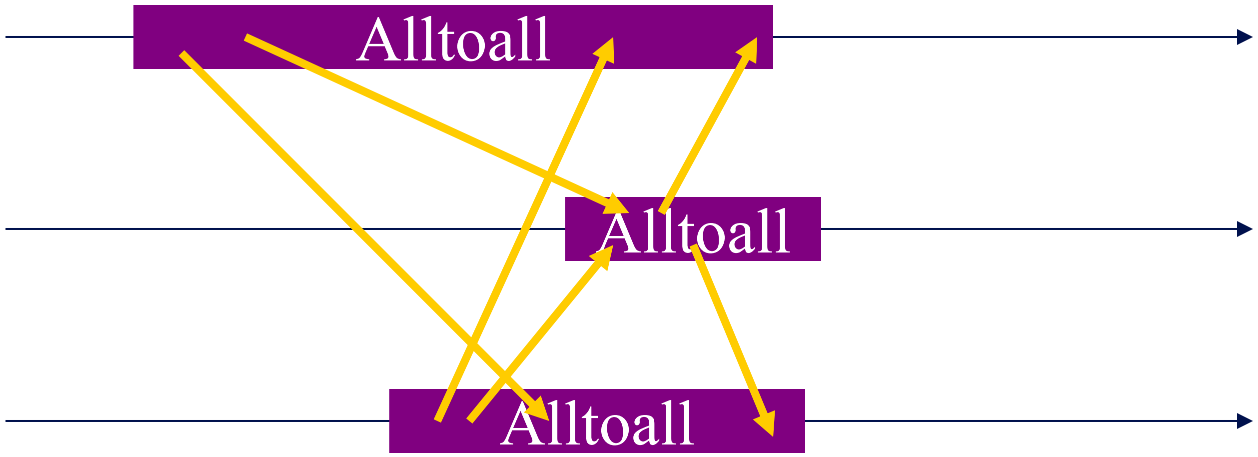
- If broadcast root is late, all other tasks have to wait
- Also applies to Scatter, Scatterv

Early Reduce



- If root task of Reduce is early, it has to wait for all other tasks to enter reduce
- Also applies to Gather, GatherV

Wait at NxN



- Other collectives require all tasks to arrive before any can leave.
  - all tasks wait for last one
- Applies to Allreduce, Reduce_Scatter, Allgather, Allgatherv, Alltoall, Alltoallv

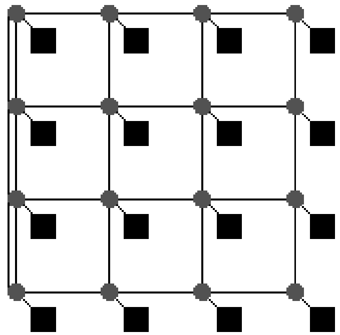
Collectives

- Collective comms are (hopefully) well optimised for the architecture
  - Rarely useful to implement them your self using point-to-point
- However, they are expensive and force synchronisation of tasks
  - helpful to reduce their use as far as possible
  - e.g. in many iterative methods, a reduce operation is often needed to check for convergence
  - may be beneficial to reduce the frequency of doing this, compared to the sequential algorithm
- Non-blocking collectives added in MPI-3
  - may not be that useful in practice …

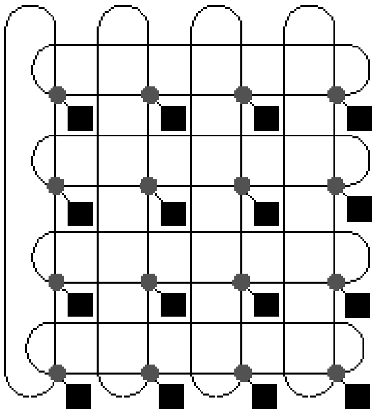
Task mapping

- On most systems, the time taken to send a message between two processors depends on their location on the interconnect.
- Latency may depend on number of hops between processors
- Bandwidth may also vary between different pairs of processors
- In an SMP cluster, communication is normally faster (lower latency and higher bandwidth) inside a node (using shared memory) than between nodes (using the network)

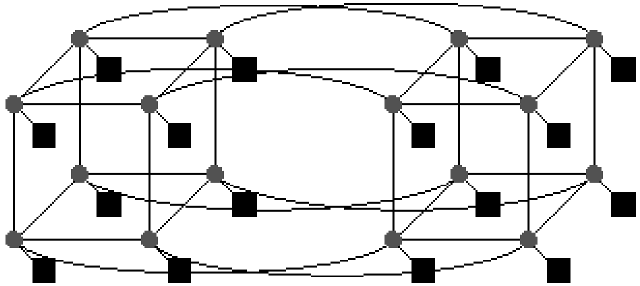
- Communication latency often behaves as a fixed cost + term proportional to number of hops.



a. 2D grid or mesh of 16 nodes



b. 2D torus of 16 nodes



c. Hypercube tree of 16 nodes ($16 = 2^4$ so $n = 4$)

- The mapping of MPI tasks to processors can have an effect on performance
- Want to have tasks which communicate with each other a lot close together in the interconnect.
- No portable mechanism for arranging the mapping.
  - e.g. on Cray XE/XC supply options to aprun
- Can be done (semi-)automatically:
  - run the code and measure how much communication is done between all pairs of tasks
  - tools can help here
  - find a near optimal mapping to minimise communication costs

- On systems with no ability to change the mapping, we can achieve the same effect by create communicators appropriately.
  - assuming we know how MPI_COMM_WORLD is mapped
- MPI_CART_CREATE has a reorder argument
  - if set to true, allows the implementation to reorder the task to give a sensible mapping for nearest-neighbour communication
  - unfortunately many implementations do nothing, or do strange, non-optimal re-orderings!
- … or use MPI_COMM_SPLIT