

MPI Optimisation

Advanced Message-Passing Programming



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

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.

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.

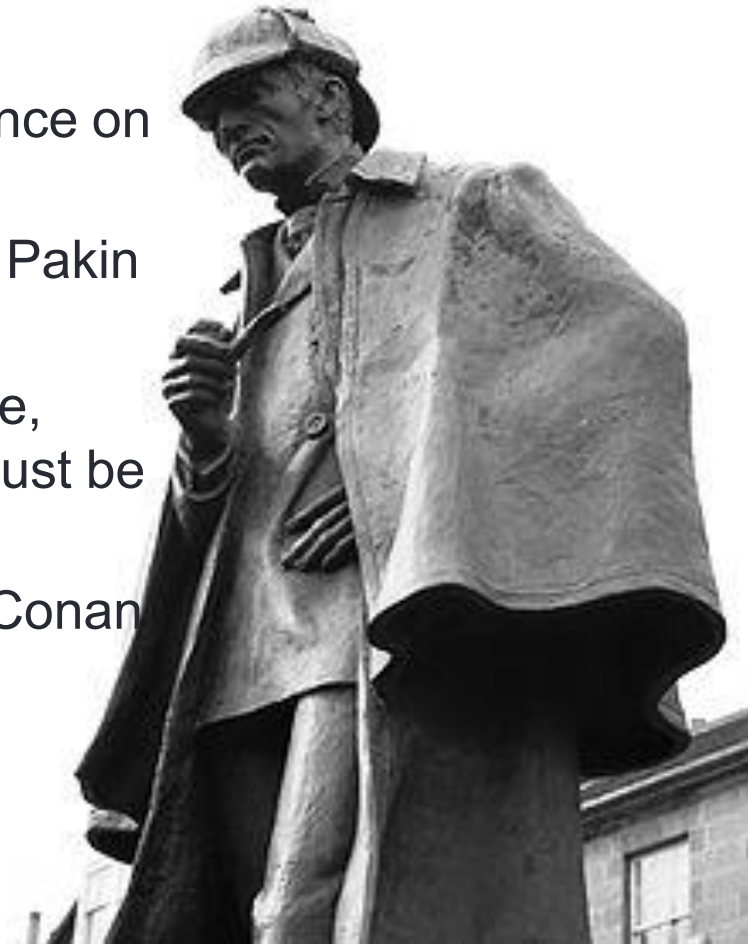
- MPI profiling tools report the amount of time spent in each MPI routine
- For blocking routines (e.g. Recv, Wait, collectives) this time may be a result of load imbalance.
- The task is blocked waiting for another task to enter the corresponding MPI call
 - the other tasks may be late because it has 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
- Use of blocking point-to-point comms can result in unnecessary synchronisation.
 - Can amplify “random noise” effects (e.g. OS interrupts)

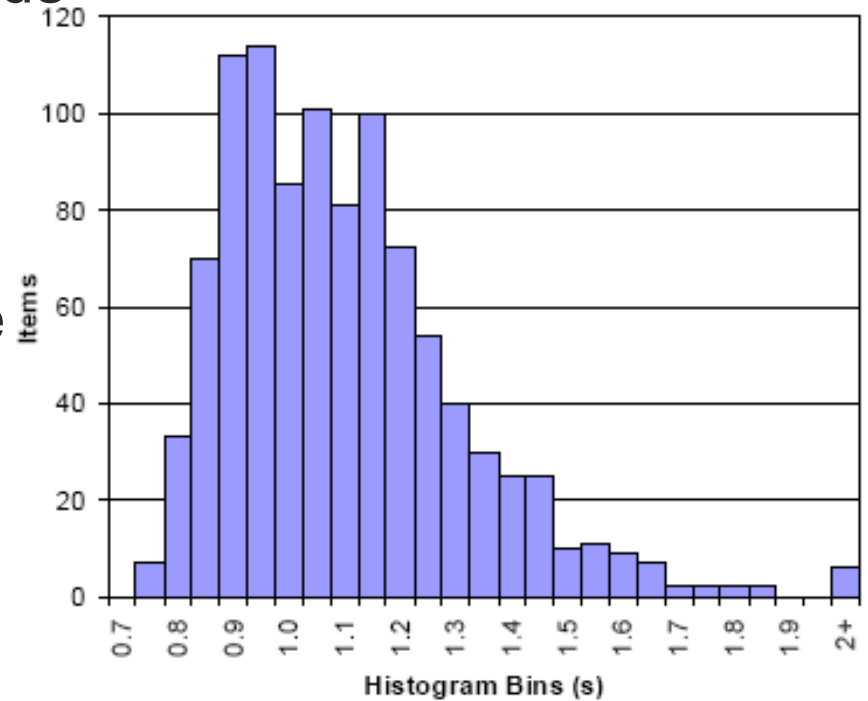
Reference

- See:
 - “The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q”
 - Fabrizio Petrini, Darren J. Kerbyson, Scott Pakin
 - <http://dx.doi.org/10.1145/1048935.1050204>
 - “[W]hen you have eliminated the impossible, whatever remains, however improbable, must be the truth.”
 - Sherlock Holmes, *Sign of Four*, Sir Arthur Conan Doyle



An example

- “Although SAGE [the application] spends half of its time in allreduce (at 4,096 processors), making allreduce seven times faster leads to a negligible performance improvement.”
- Collectives an extreme example
 - point-to-point is also an issue



SAGE time per iteration

Collectives

- Reduce frequency of calculation by a factor X
 - e.g. trade more calculation for fewer synchronisations

```
loop over iterations:  
    update arrays;  
    compute local delta;  
    compute global delta  
    using allreduce;  
    stop if less than  
    tolerance value;  
end loop
```

```
loop over iterations:  
    update arrays;  
    every X iterations:  
        local delta;  
        global delta;  
        can we stop?;  
  
end loop
```

- Possible because array updates independent of global values
 - may not be true for other algorithms, e.g., Conjugate Gradient
 - again, more iterations but less synchronisation

Barriers

- (Almost) never required for MPI program correctness
- Why?
 - because collectives do the appropriate synchronisation
 - because MPI_Recv is synchronous

Communication

- Point-to-point communications
- Collective communications

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

$$T_p = T_l + N_b T_b$$

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 :

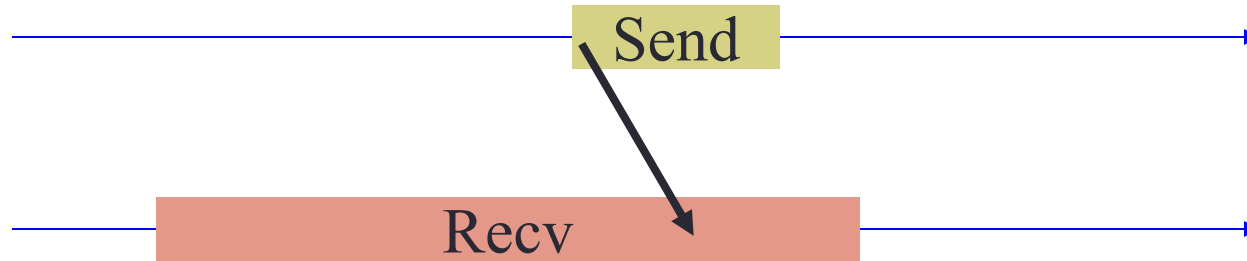
$$T_p = T_l + \frac{N_b}{B}$$

- 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

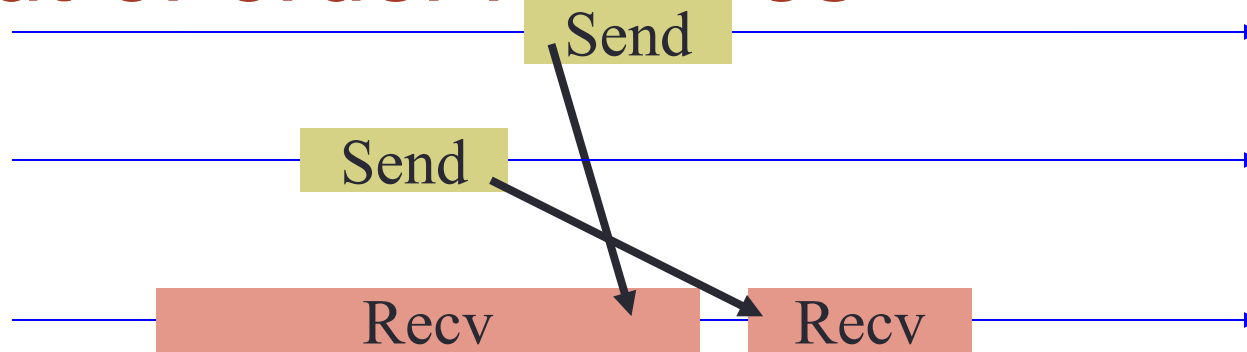
- Can be helpful, especially when using trace analysis tools, to think about communication patterns
 - Note: nothing to do with OO design!
- We can identify a number of patterns which can be the cause of poor performance.
- Can be identified by eye, or potentially discovered automatically
 - e.g. the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

Late Sender

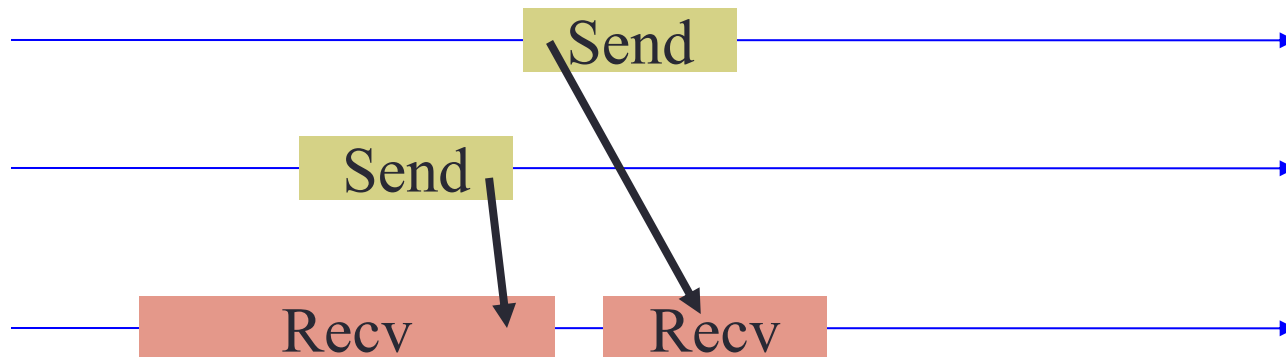


- If blocking receive is posted before matching send, then the receiving task must wait until the data is sent.

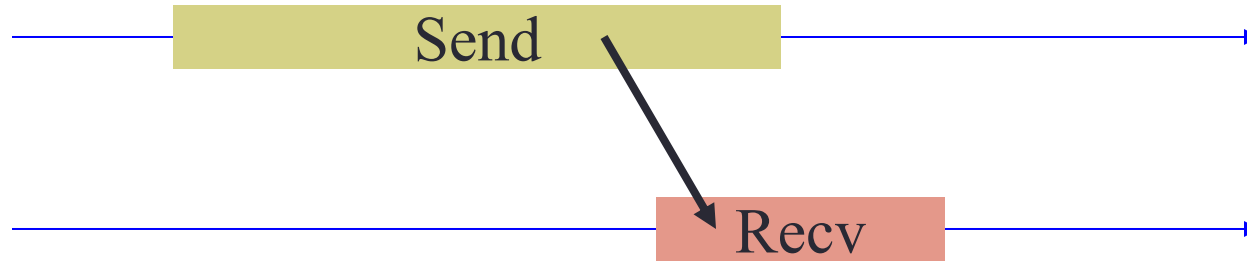
Out-of-order receives



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



Late Receiver

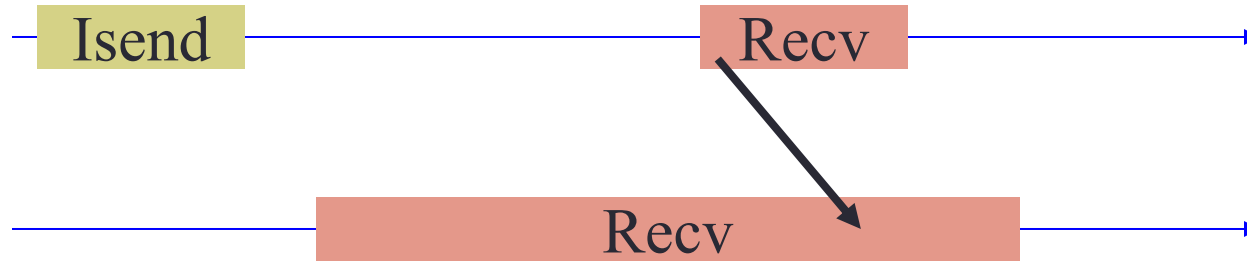


- If send is synchronous, data cannot be sent until receive is posted
 - either explicitly programmed, or chosen by the implementation because message is large
 - sending task is delayed

MPI Progression

- You probably think of MPI as running continuously
 - e.g. asynchronous / non-blocking comms happen “in the background”
 - communications and calculation overlap in time
- This is not generally true
 - MPI library is single-threaded by default, i.e. communications can only be processed when your program calls an MPI function
 - MPI treats calls as manual interrupts and will try to “progress” communications by matching outstanding sends and receives before actually doing what you have asked it to!
- If you issue a non-blocking send
 - it may be sent immediately if there is an existing receive
 - if not, it cannot be sent until the next explicit MPI call (which may be unrelated to the outstanding communication itself)

Late Progress

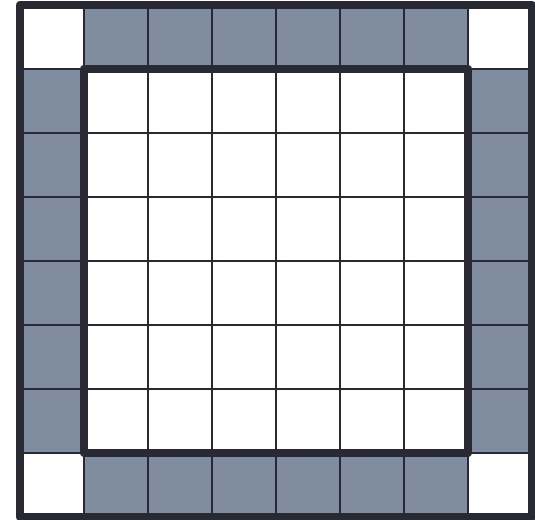
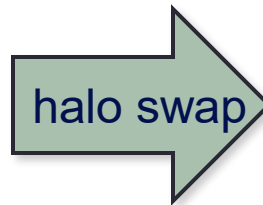
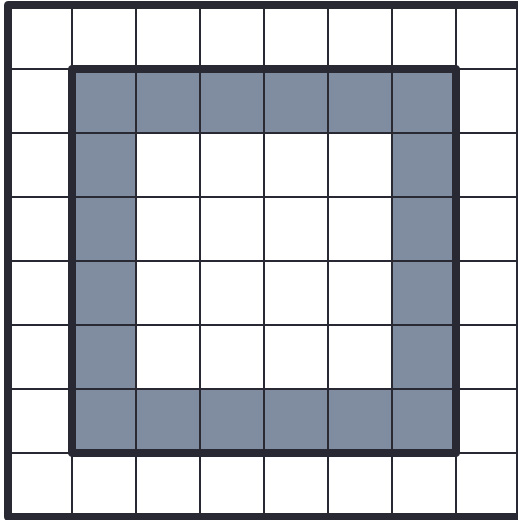


- Non-blocking send returns, but implementation has not yet sent the data.
 - A copy has been made in an internal buffer
- Send is delayed until the MPI library is re-entered by the sender.
 - receiving task waits until this occurs

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 delays by using all non-blocking comms entirely (Isend, Irecv, WaitAll)
 - post all the Irecv's as early as possible

Normal halo swapping



```
swap data into 4 halos: i=0, i=M+1, j=0, j=M+1
loop i=1:M; j=1:N;
  new(i,j) = 0.25*(    old(i-1,j) + old(i+1,j)
                    + old(i,j-1) + old(i,j+1)
                    - edge(i,j) )
```

Point-to-point

- Do not impose unnecessary ordering of messages

```
loop over sources:  
  receive value from  
  particular source;  
end loop
```

```
loop over sources:  
  receive value from  
  any source;  
end loop
```

- loop now just counts the correct number of messages
- Alternative
 - first issue a separate non-blocking receive for each source
 - then issue a single Waitall

Halo swapping

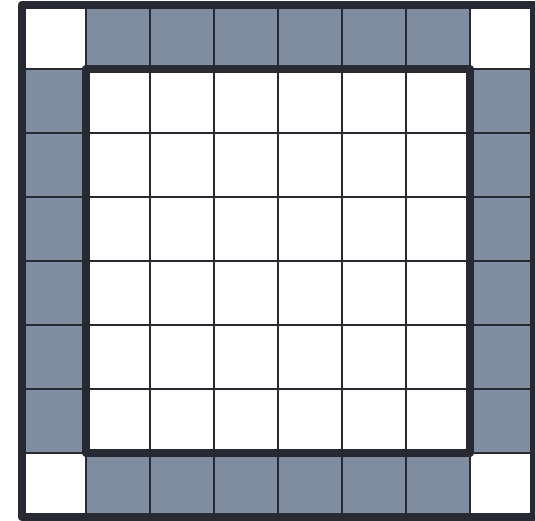
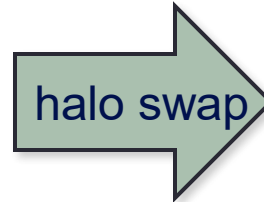
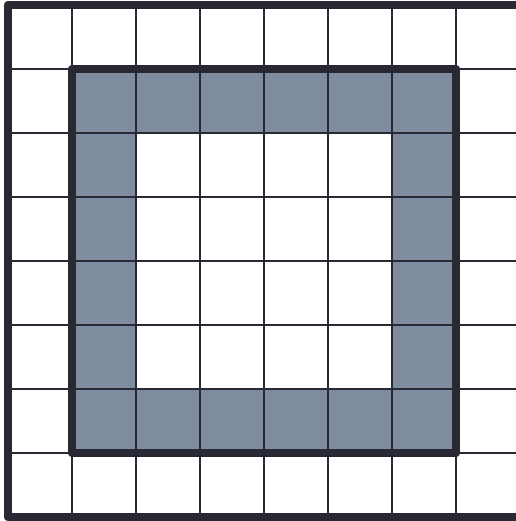
- Do not impose unnecessary ordering of messages

```
loop over directions:  
  send up; recv down;  
  send down; recv up;  
end loop
```

```
loop over directions:  
  isend up; irecv down;  
  isend down; irecv up;  
end loop  
wait on all requests;
```

- Extensions
 - can now overlap communications with core calculation
 - only need to wait for receives before non-core calculation
 - wait for sends to complete before starting next core calculation

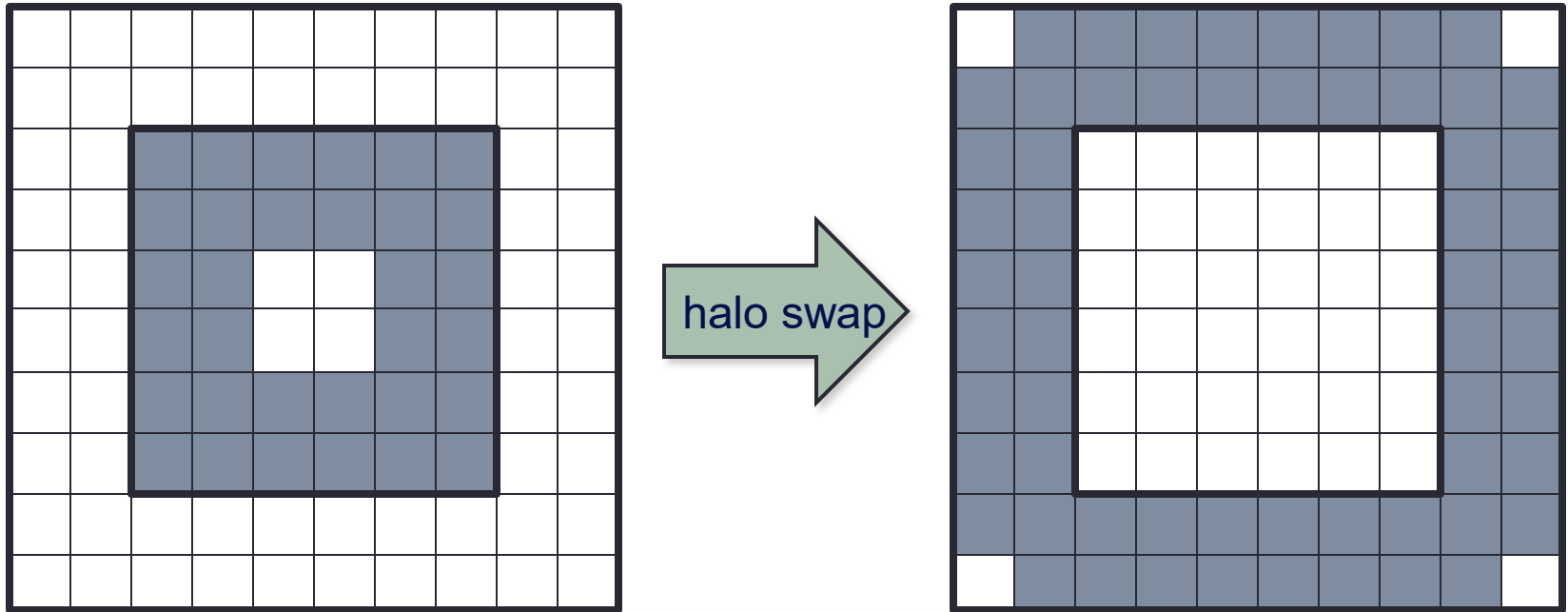
Overlapping



```
start non-blocking sends/recvs
loop i=2:M-1; j=2:N-1;
    new(i,j) = 0.25*(    old(i-1,j) + old(i+1,j)
                      + old(i,j-1) + old(i,j+1)
                      - edge(i,j) )
wait for completion of non-blocking sends/recvs
complete calculation at the four edges
```

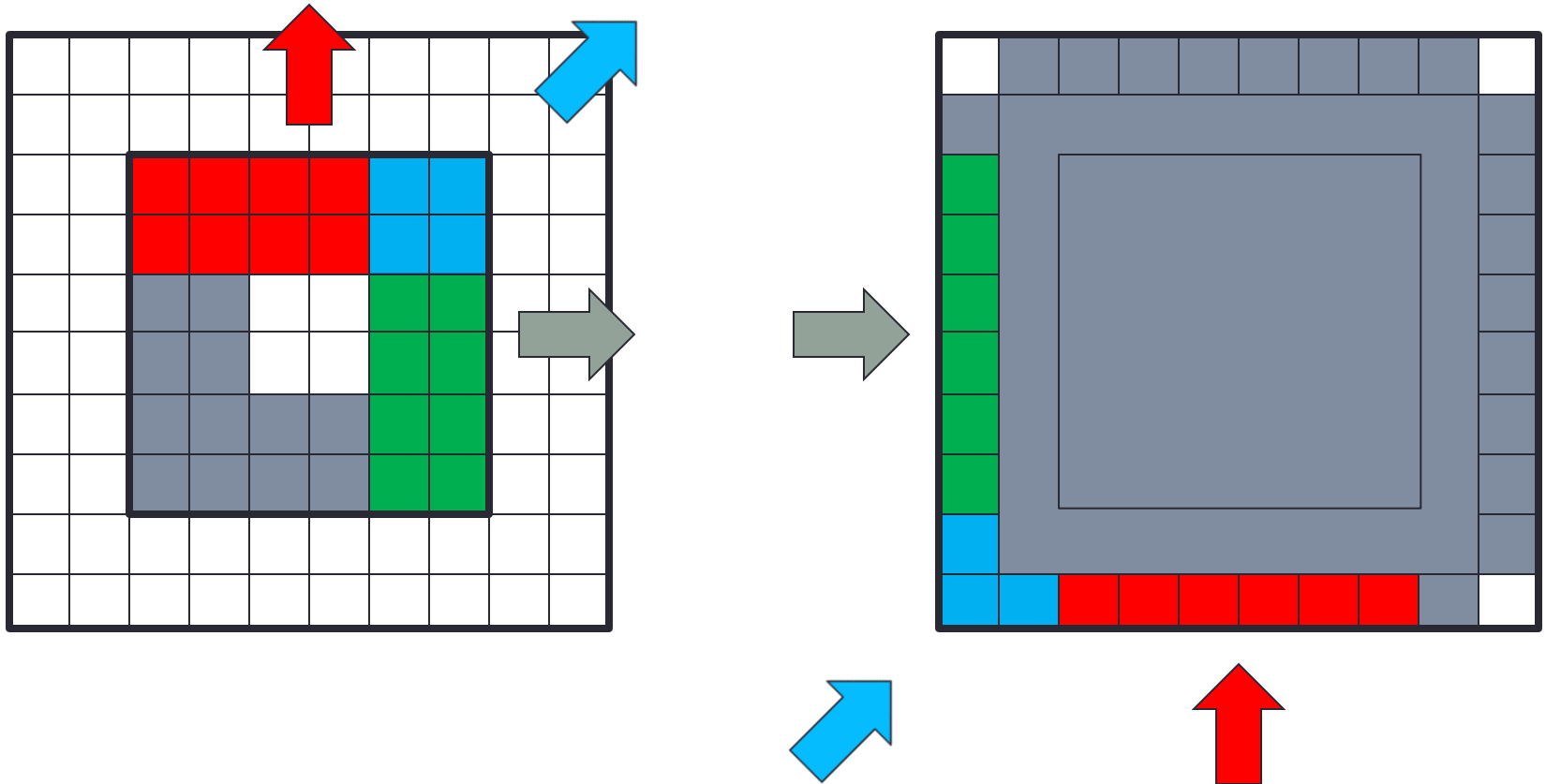

Halos of Depth D every D iterations

- Smaller number of larger messages; increased computation



```
loop d=D:1:-1
  loop i=2-d:M+d-1; j=2-d:N+d-1;
    new(i,j) = 0.25*( old(i-1,j) + old(i+1,j)
                     + old(i,j-1) + old(i,j+1)
                     - edge(i,j) )
```

Swap depth D every D iterations



- Need diagonal communications

Persistent communications

- Standard method: run this code every iteration

```
MPI_Irecv(..., procup, ..., &reqs[0]);  
MPI_Irecv(..., procdn, ..., &reqs[1]);  
MPI_Isend(..., procdn, ..., &reqs[2]);  
MPI_Isend(..., procup, ..., &reqs[3]);  
MPI_Waitall(4, reqs, statuses);
```

- Persistent comms: setup *once*

```
MPI_Recv_init(..., procup, ..., &reqs[0]);  
MPI_Recv_init(..., procdn, ..., &reqs[1]);  
MPI_Send_init(..., procdn, .... &reqs[2]);  
MPI_Send_init(..., procup, ..., &reqs[3]);
```

- Every iteration:

```
MPI_Startall(4, reqs);  
MPI_Waitall (4, reqs, statuses);
```

- Message ordering *not guaranteed to be preserved*
 - may need to use tags to correctly match messages

Neighbourhood Collectives

- Standard collectives are applied to whole communicator
 - e.g. `MPI_Allgather` collects data from all P processes
- Neighbourhood collectives apply to neighbouring processes
 - e.g. `MPI_Neighbor_allgather` only collects data from your neighbours
 - requires communicator to be constructed with a topology
- Regular grid
 - Cartesian topology via `MPI_Cart_create`
 - in 3D grid, gather from six nearest neighbours up, down, left, right, ...
- General communications pattern
 - requires a graph topology
 - each process connected to an arbitrary number of neighbours

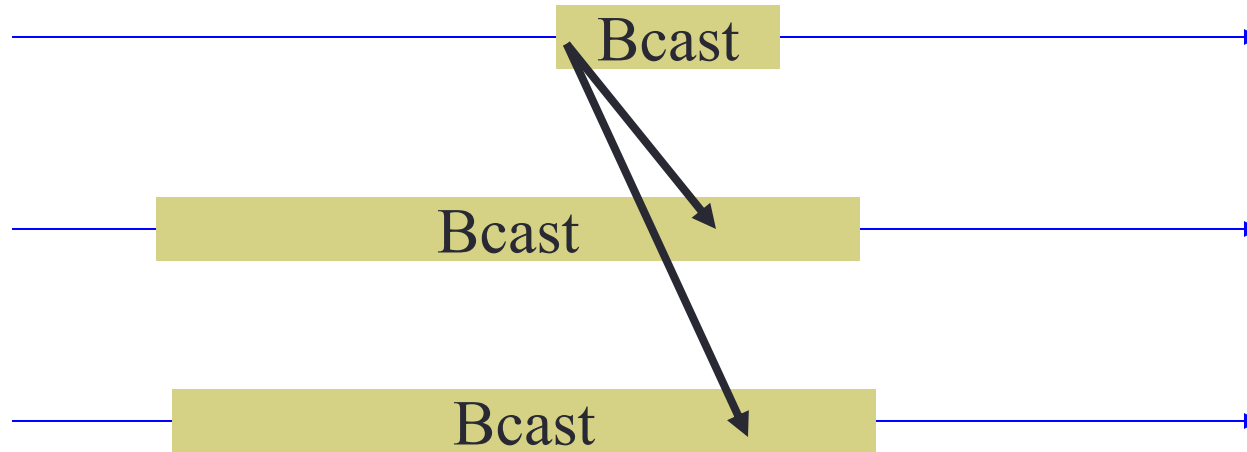
Use for halo swapping

- `MPI_Neighbor_alltoall` implements halo swapping
 - send and receive data with all your neighbours
- Simple 3D cartesian grid illustrated in halobench exercise
 - for multidimensional arrays need to play tricks with datatypes to send and receive correct data (see later talk)
- MPI library can implement in any way it chooses
 - hopefully efficiently!
 - code is much more elegant and compact than point-to-point

Collective communications

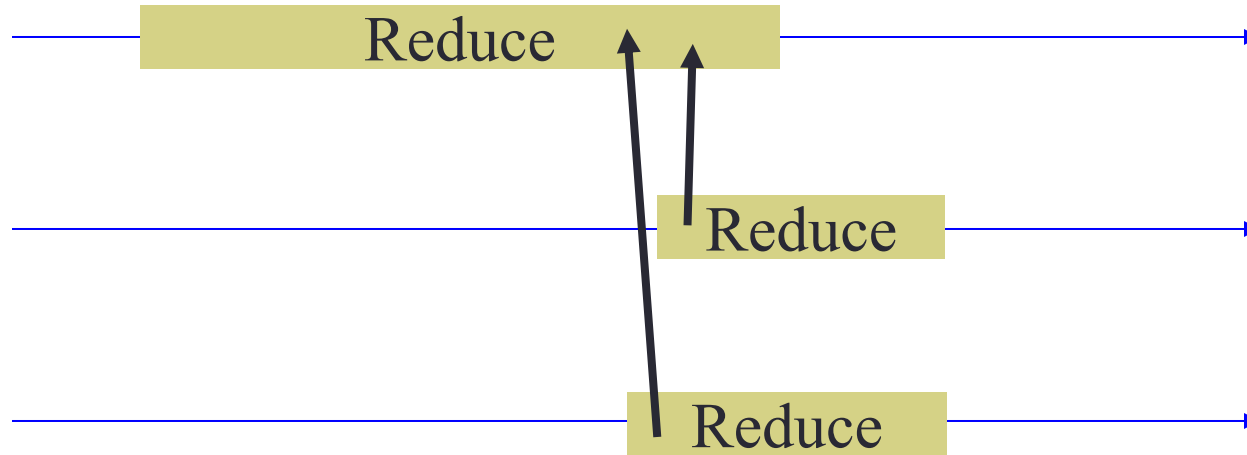
- Can identify similar synchronisation patterns for collective comms as for point-to-point...

Late Broadcaster



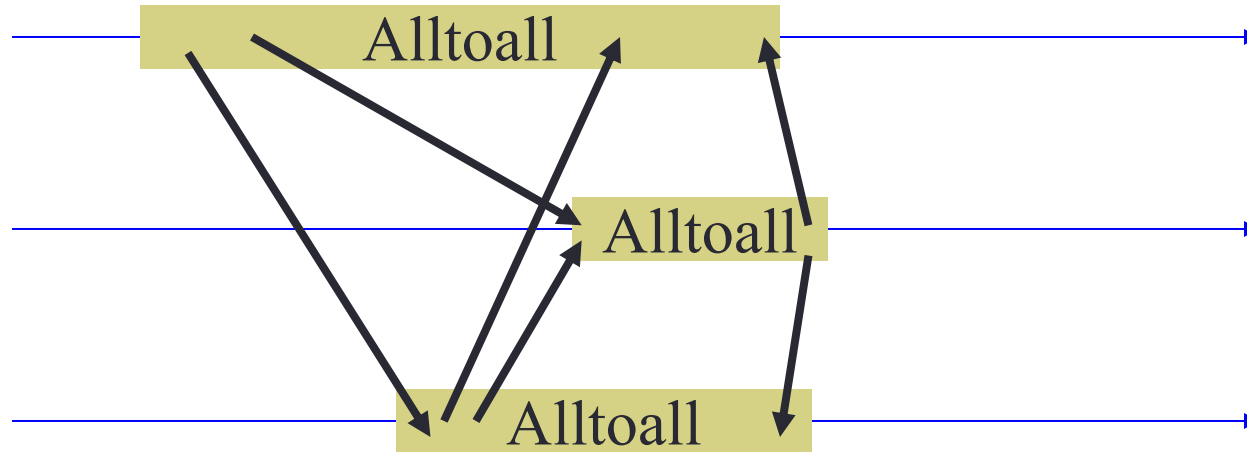
- If broadcast root is late, all other tasks have to wait
- Also applies to Scatter, Scatterv

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 ...

General advice

- Try to avoid imposing any non-essential message ordering
 - when messages are actually sent can change on every run
- Try to allow MPI to deal with messages as they happen
 - issue all sends and receives as non-blocking
 - issue receives as early as possible
 - complete with a single Waitall()
 - or set up as persistent comms
 - be careful about message ordering
 - or use neighbourhood collectives
- Avoid unnecessary collective calls e.g. by aggregation
 - do you really need any of those barriers?!?

Summary

Can divide overheads up into four main categories:

- Lack of parallelism
 - Cannot split work up into enough pieces
- Load imbalance
 - Pieces for each processor are not identical amount of work
- Synchronisation
 - Processors waiting for each other
- Communication
 - Inefficient patterns of communication