

# Asynchronous Parallel Methods

---

# 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](http://www.epcc.ed.ac.uk)”

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

# Overview

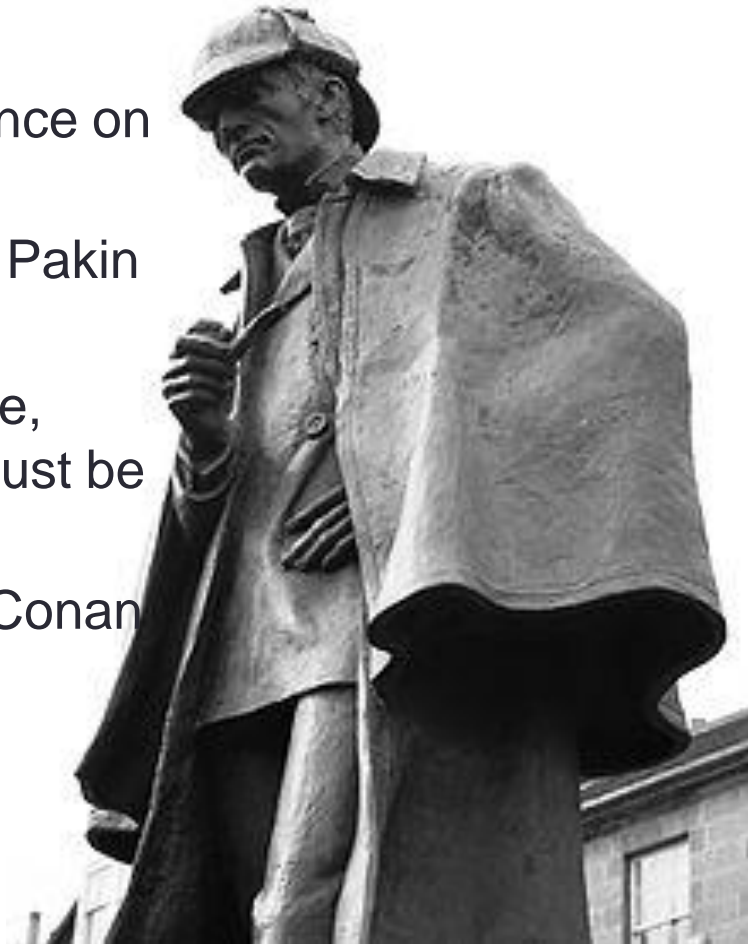
- What's the problem?
- What is an asynchronous method?
- Reducing synchronisation in existing models

# The Problem

- Synchronisations often essential for program correctness
  - waiting for an MPI receive to complete before reading from buffer
  - barriers at the end of an OpenMP parallel loop
  - ...
- But they cost time
  - and slow down the calculation
- Cost is usually not the synchronisation operation itself
  - it is waiting for other tasks to catch up with each other
  - all calculations have some load imbalance from random fluctuations
  - a real problem as we increase the number of cores
- Try to reduce synchronisation
  - and let things happen in their “natural” order

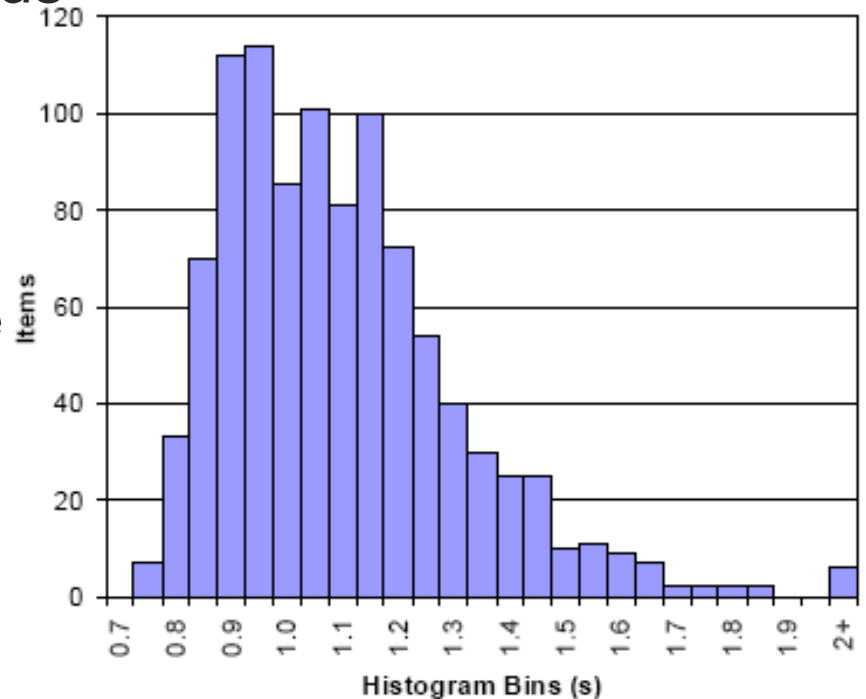
# Reference

- 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
```

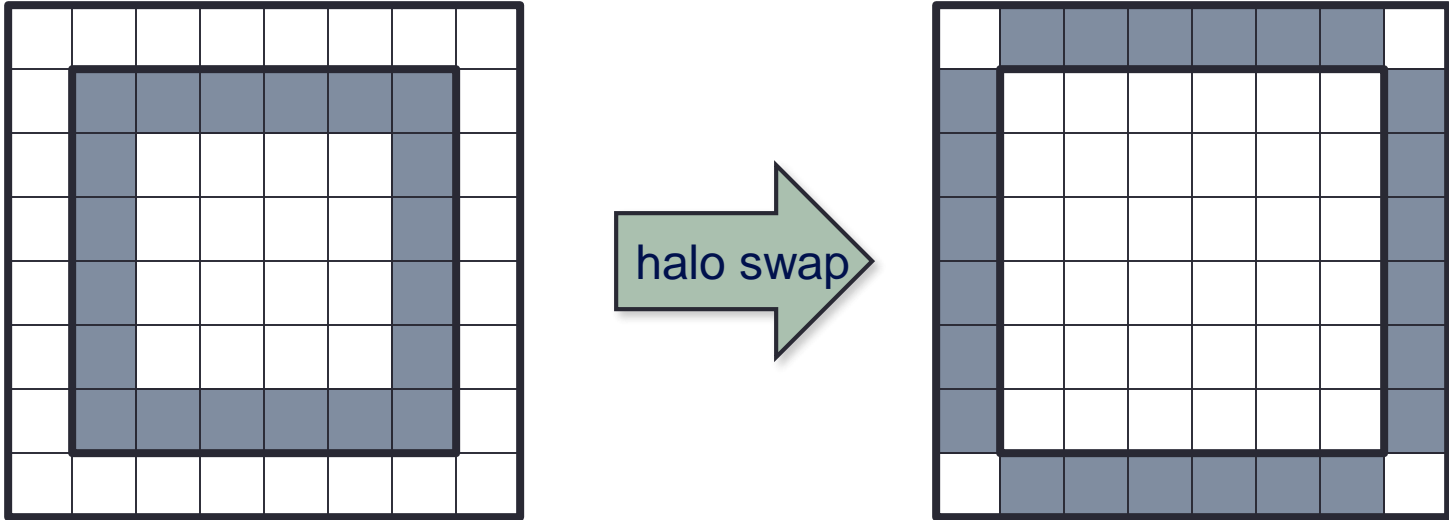
- Possible because array updates independent of global values
  - may not be true for other algorithms, e.g., Conjugate Gradient
  - again, more iterations but less synchronisation

# Barriers

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



# 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

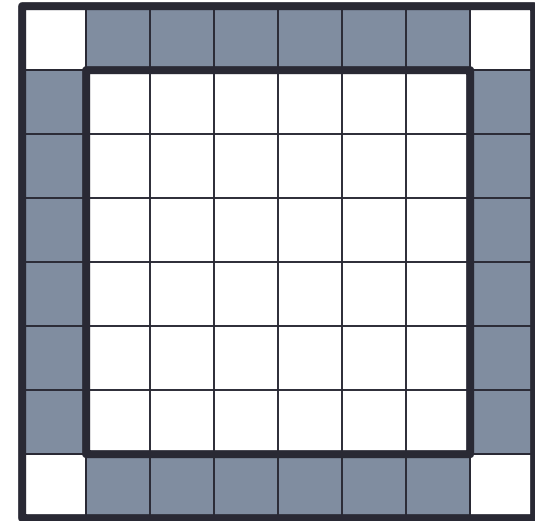
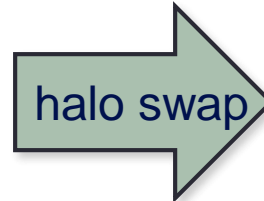
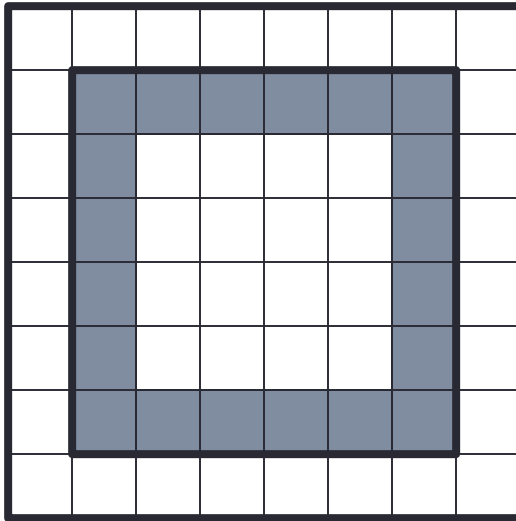
```
loop over directions:  
  send up; recv down;  
  send down; recv up;  
end loop
```

```
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

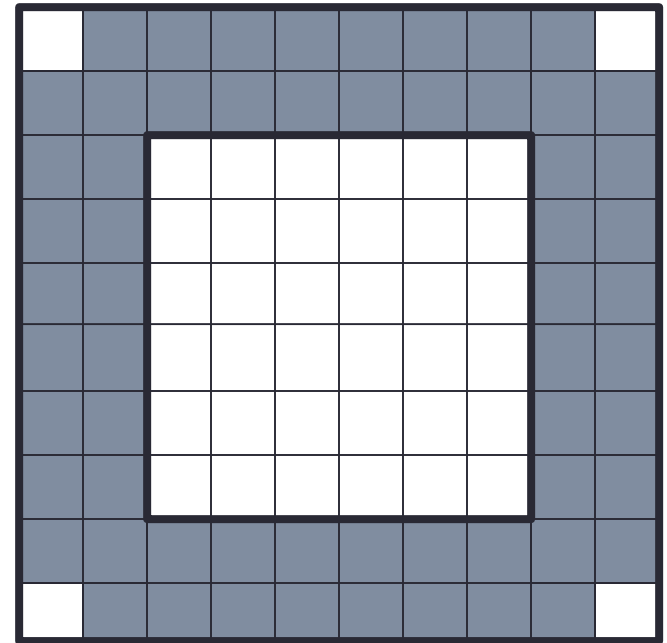
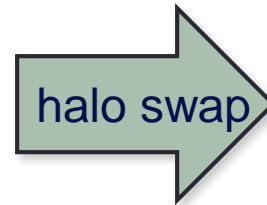
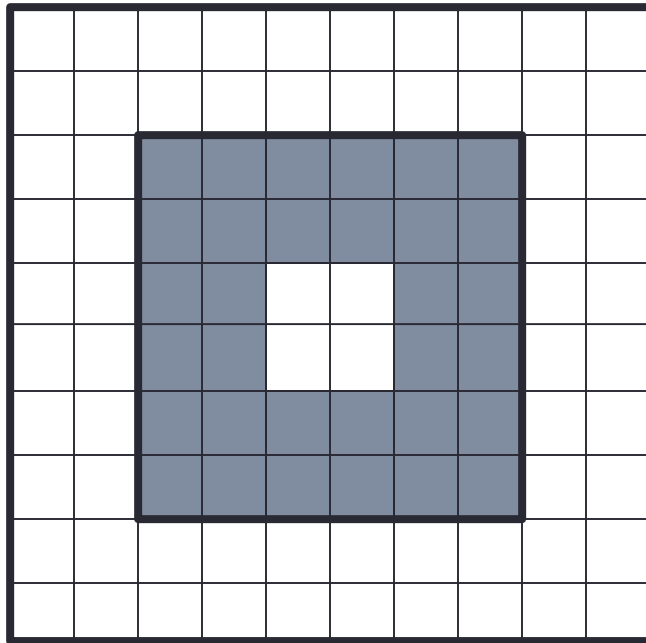
# Overlapping



```
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
```

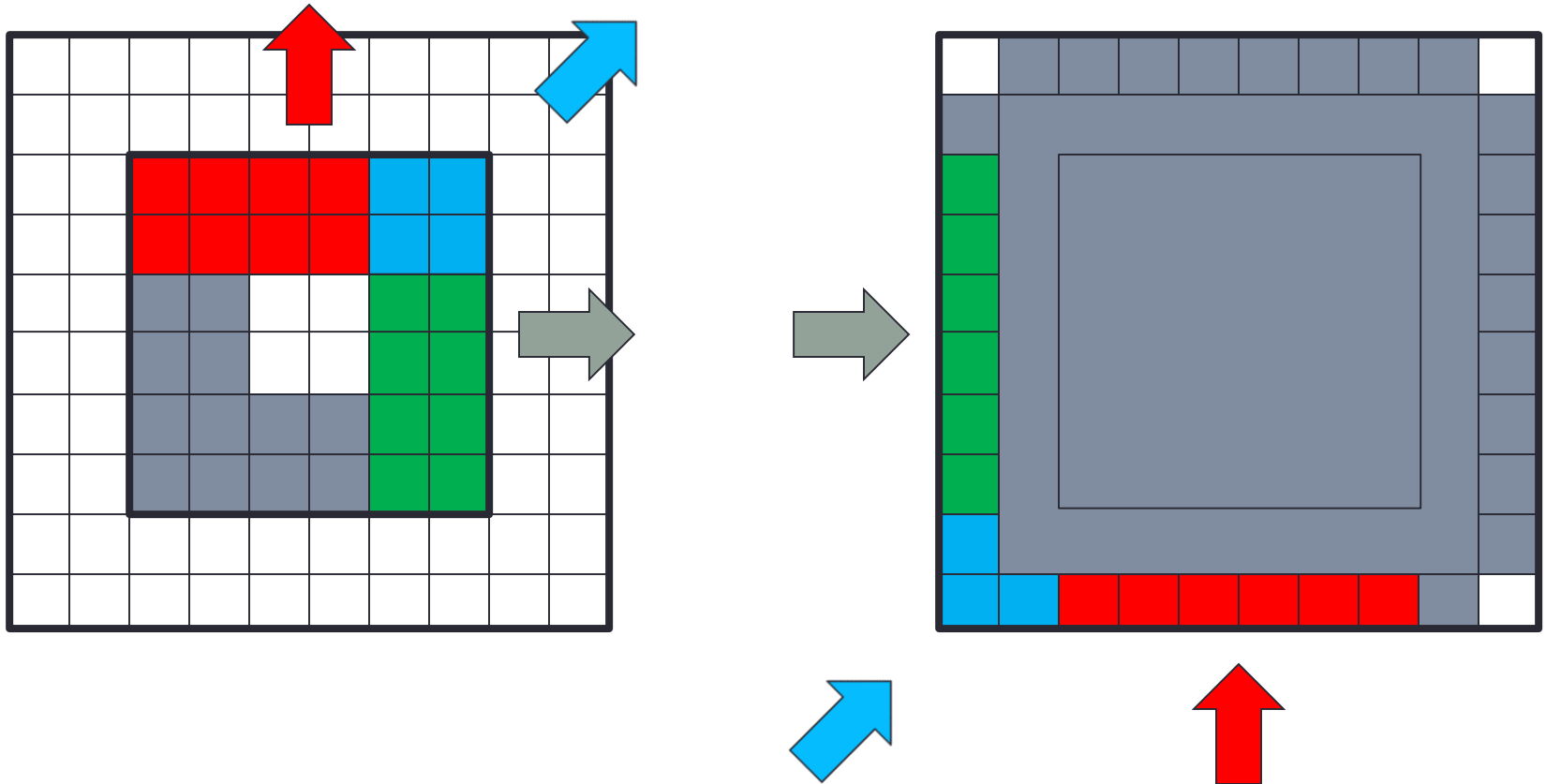
# Halos of Depth $D$ every $D$ iterations

- 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

# Implementation

- Do 8 non-blocking sends and 8 non-blocking receives
  - as opposed to only 4 for depth=1
  - ... or 26 vs 6 for three dimensions
  - when we wanted to send fewer messages!
- Can “carry” halos rather than explicit diagonal comms
  - ordered swaps: left/right after up/down ...
  - – ... but introduces more synchronisation
- Quite hard to implement in practice
  - $D=1$  is (thankfully) special case for 5-point stencil with no diagonals

# 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*