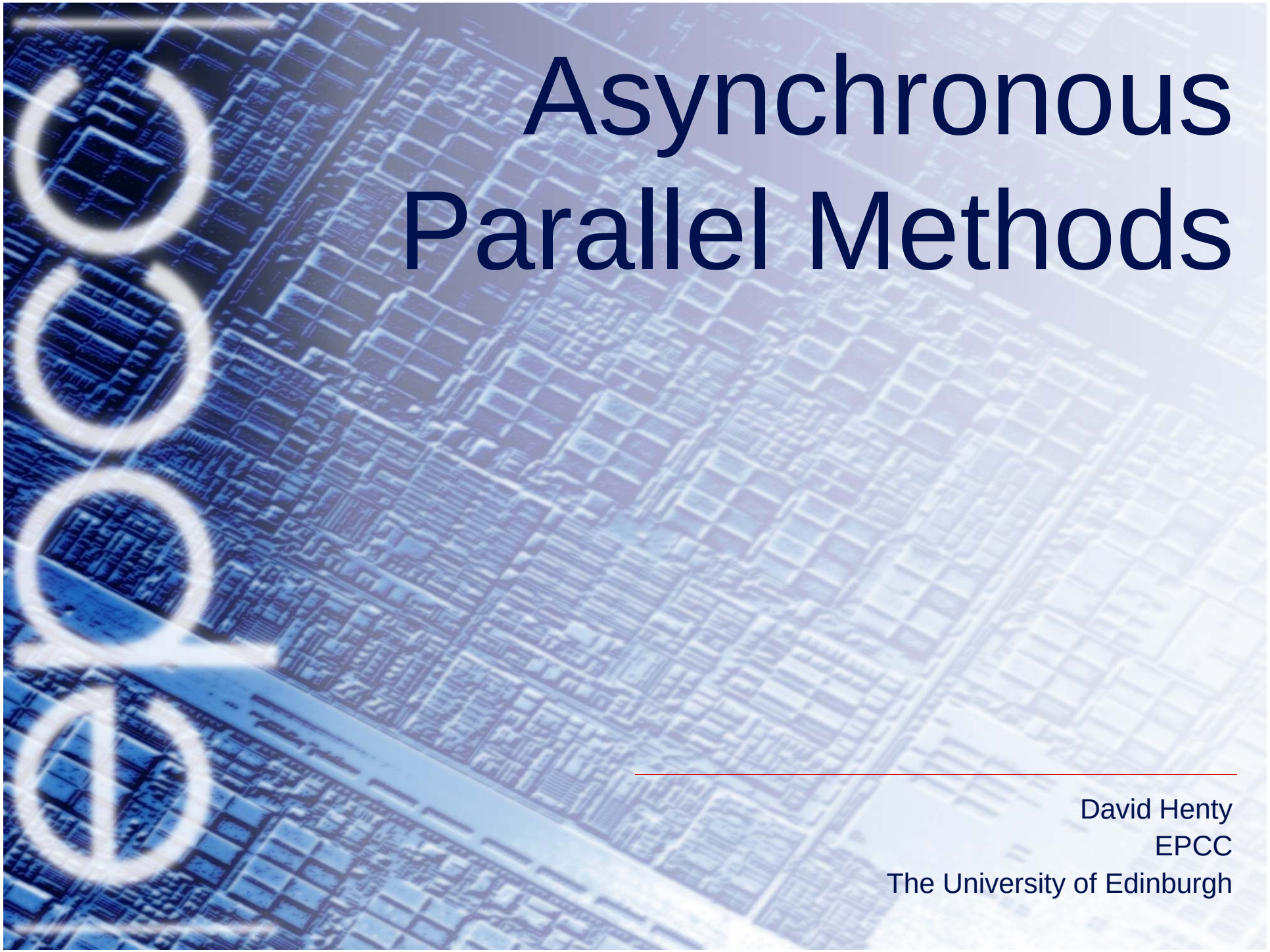


Asynchronous Parallel Methods

David Henty
EPCC
The University of Edinburgh

Overview

- What's the problem?
- What is an asynchronous method?
- Reducing synchronisation in existing models
- Asynchronous algorithms
- Removing all synchronisation
- Summary

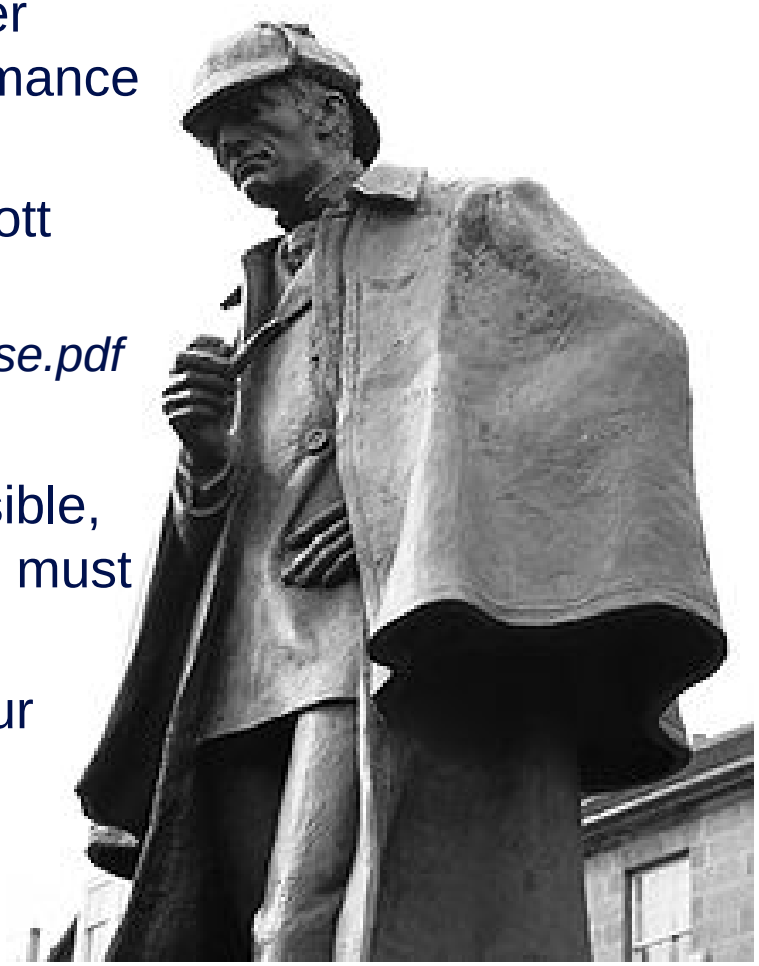
The Problem

- 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
  - *hpc.pnl.gov/people/fabrizio/papers/sc03_noise.pdf*

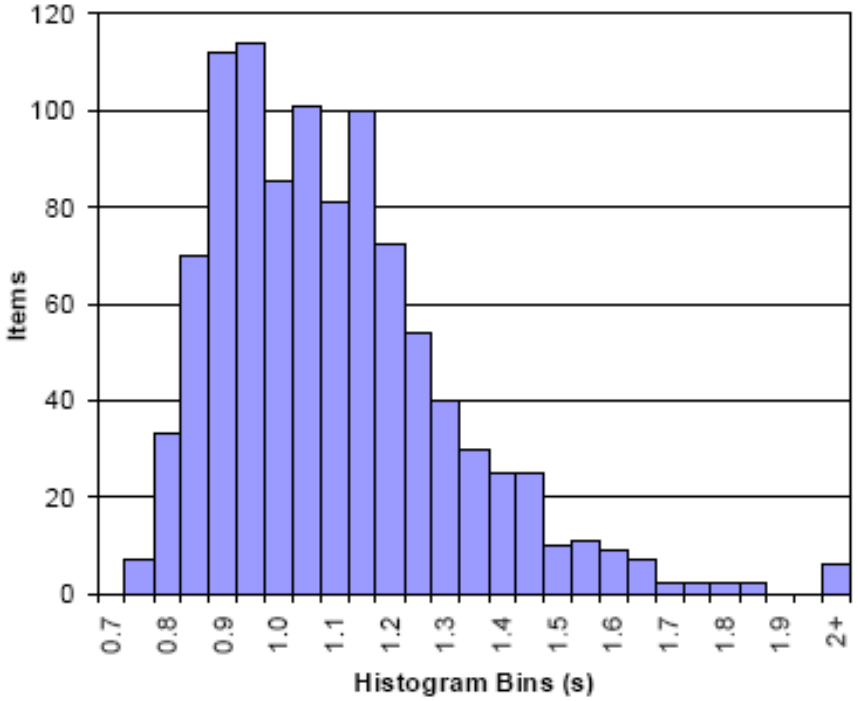
  - "[W]hen you have eliminated the impossible, whatever remains, however improbable, must be the truth."
  - Sherlock Holmes, *Sign of Four*, Sir Arthur Conan Doyle



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. MPP coursework: 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, e.g., Conjugate Gradient
  - can use different algorithms, e.g. Chebyshev iteration
  - again, more iterations but less synchronisation

Shared memory barriers

- (Almost) never required for MPI program correctness
- OpenMP
  - remove with nowait

```
OMP parallel
  OMP loop #1
  loop i=1:M
    calculation 1
  end loop #1
  OMP no wait
  OMP loop #2
  loop j=1:N
    calculation 2
  end loop #2
OMP end parallel
```

  - essential to check that this gives correct answers

PGAS barriers

- Simple CAF

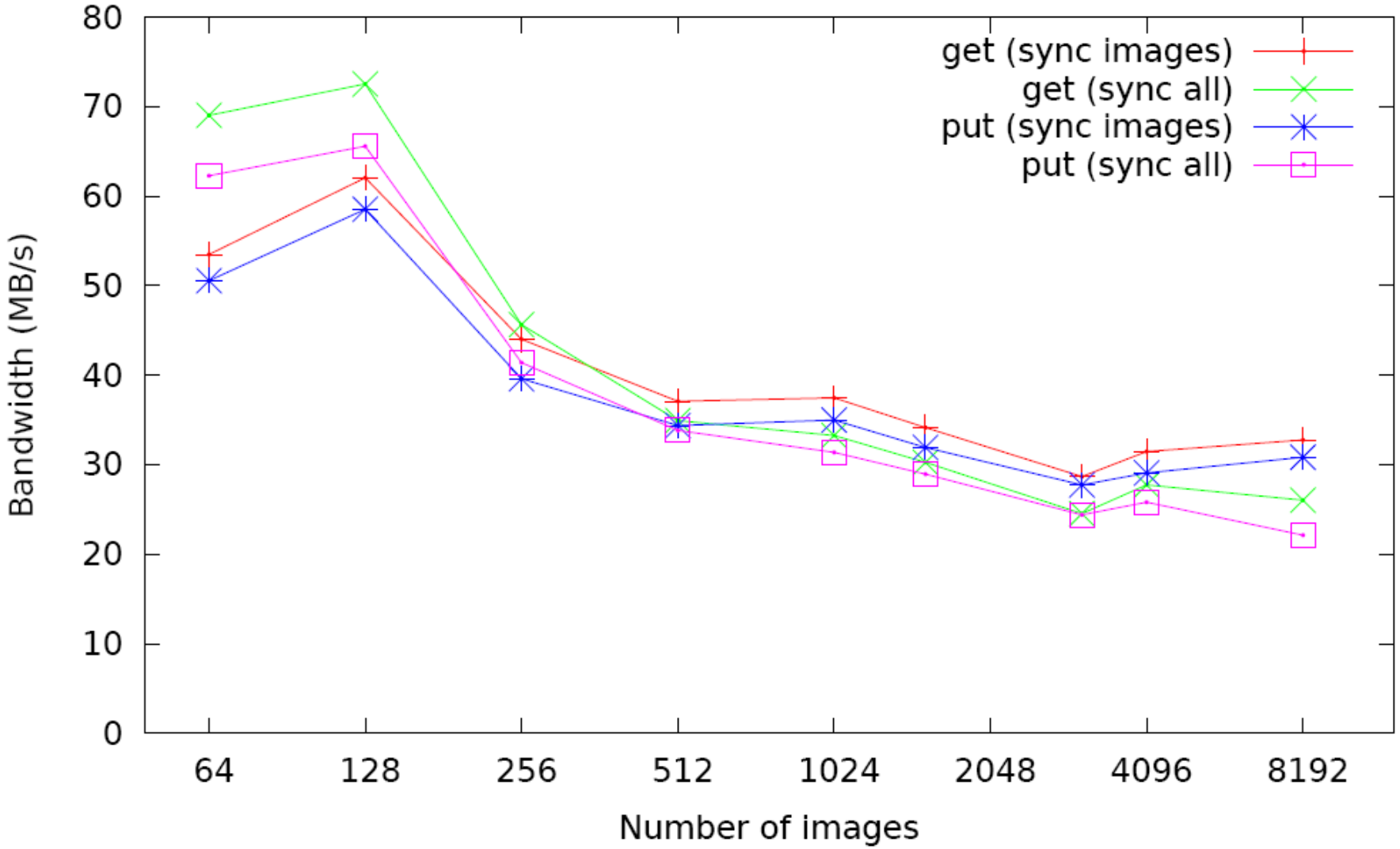
```
loop over iterations:
  update halos:
    remote reads or writes;
    sync all;

  update arrays;

  wait for updates to
  complete on all images:
     sync all;
end loop
```

- Only require synchronisation with immediate neighbours
  - replace sync all with sync images

3D Halo Swaps (10x10x10 array)



Point-to-point

- Do not impose unnecessary ordering of messages
  - e.g. pi calculation

```
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
  - e.g. MPP coursework

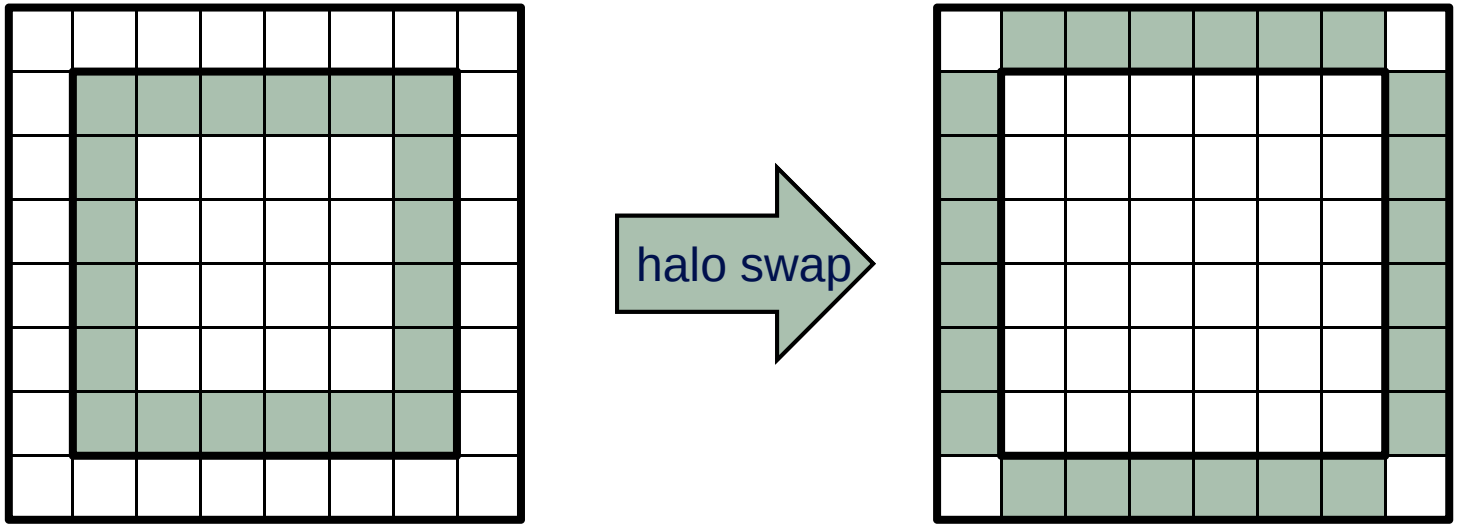
```
loop over directions:
  send up; recv down;
  send down; recv up;
end loop
```

```
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

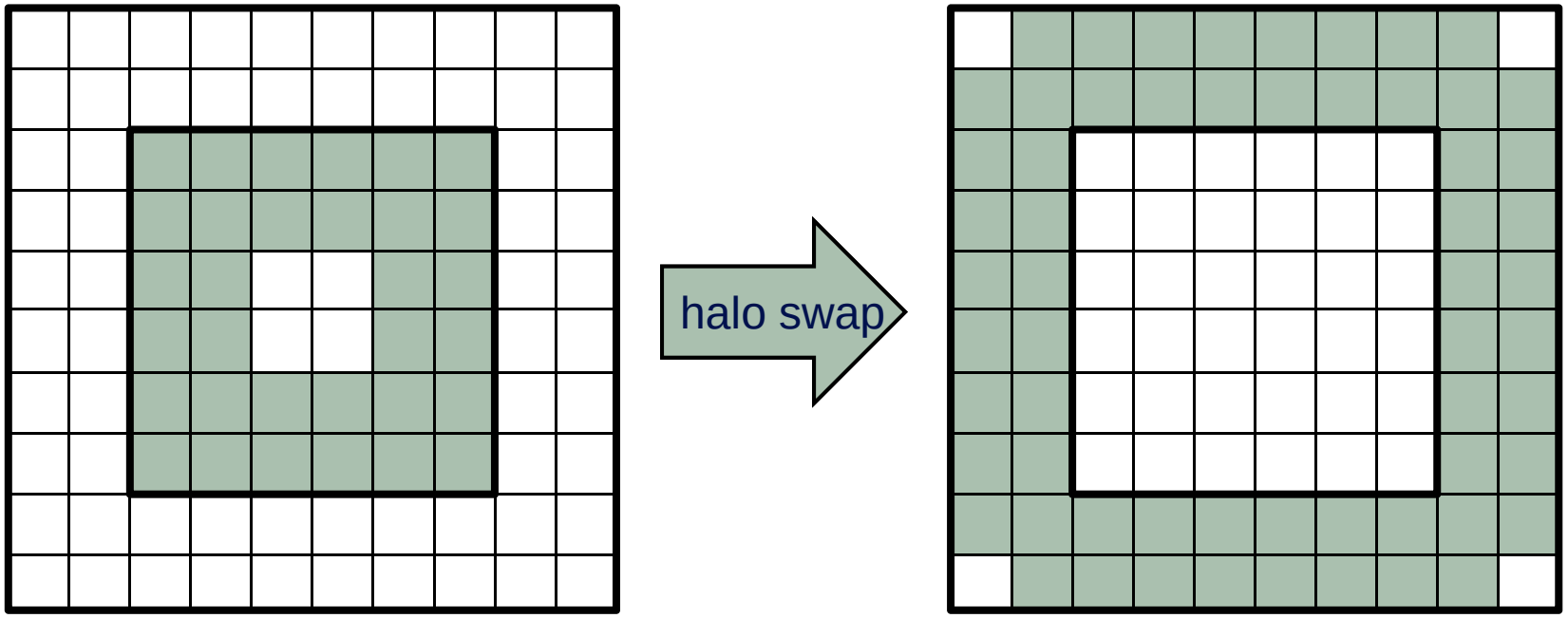
Deep halos

- Use less frequent communication
  - smaller number of larger messages; increased computation
- Normal halos on old(6,6)




```
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)                )
```

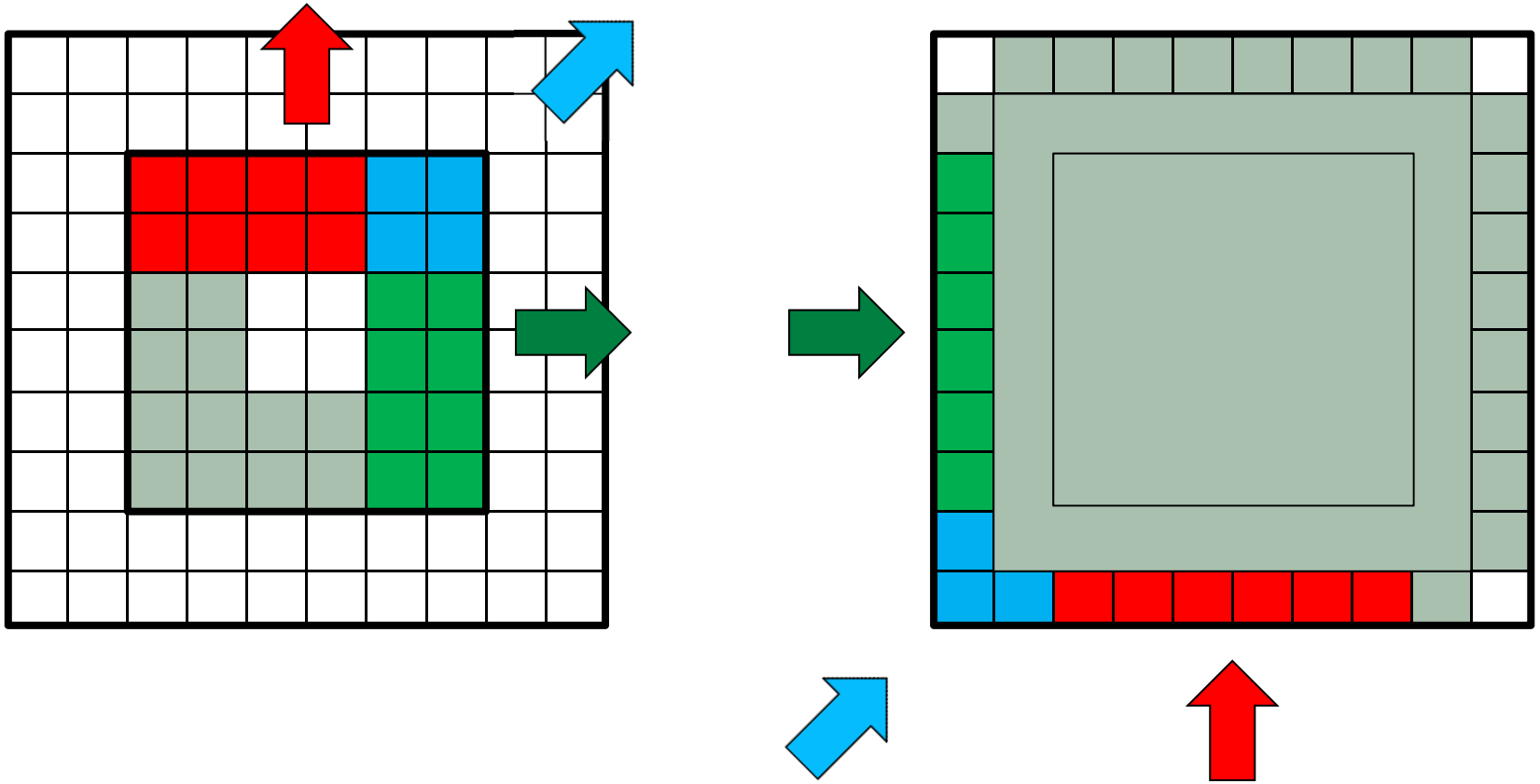
Halos of Depth *D*




```
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
  - and must swap halos of depth *D*-1 on edge(i,j)

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
- 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) a special case for 5-point stencil