Leveraging MPI RMA to optimise halo-swapping communications in MONC on Cray XC Architecture

Michael Bareford, ARCHER CSE Team
m.bareford@epcc.ed.ac.uk

With thanks to Nick Brown and Michèle Weiland
EPCC, University of Edinburgh

Contents

- Background
  - MONC, a code used by the UK Meteorological Office
  - RMA, stands for Remote Memory Access, part of the MPI standard.
- RMA Implementation
  - Initialisation of RMA memory window
  - Optimisation of epoch creation
  - Passive target synchronisation
- Results
  - ARCHER, Cray XC30
  - Weak and Strong Scaling

Met Office NERC Cloud (MONC) model

- MONC was developed for simulating clouds and atmospheric flows (it was a replacement for Large Eddy Model)
  - Written in Fortran 2003 and oriented around the concept of ***components***.
  - A model core is provided which contains general utility functionality, but all science and parallelism is provided by independent, separate components

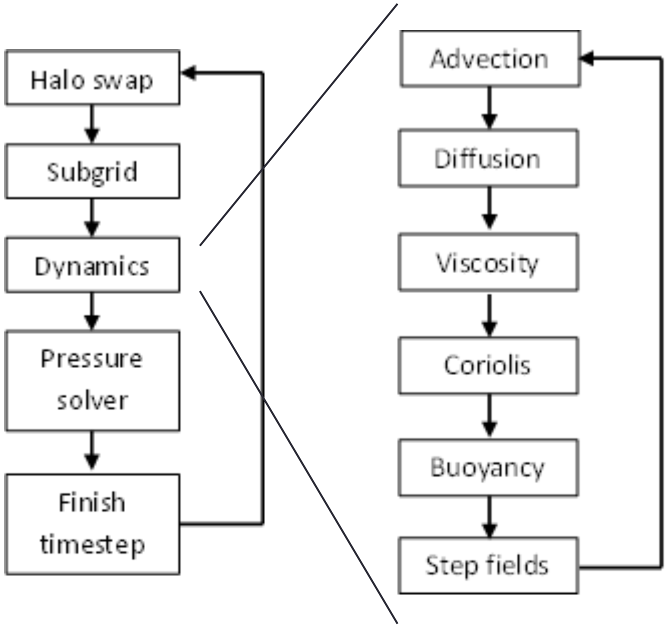
- MONC has been demonstrated up to 32,768 cores using MPI point-to-point (P2P) messaging for halo swapping.

Brown, N. et al EASC 2015

MONC basic operation

- MONC is made up of many loosely coupled components, which users combine via configuration file settings for specific runs.

- Halo Swap component provides an interface for any physics components that need to halo swap data.



- Underlying communication method (P2P or RMA) used for halo swapping can be separated from the data being swapped.

MONC Halo Communication - MPI P2P

per code run

Initiate Halo Communication

- Setup a halo-swapping context that is returned to component to allow halo swapping of specific fields --- operation involves the allocation of send/recv buffers for process neighbours.

per time step

Initiate Non-blocking Halo Swap

- Register non-blocking receives from neighbouring processes.
- Pack domain data into send buffers.
- Send packed data via non-blocking sends.

Complete Non-blocking Halo Swap

- Wait for all communications to complete.
- Unpack received data into the appropriate halo locations.

Finalise Halo Communication

- Model has completed execution --- clean up memory allocated for communication buffers.

MPI RMA Basics

- MPI Remote Memory Access (RMA) is a way of reading and writing data to the memory of other processes without having to go through the usual point to point semantics of inter-process communication.
- Memory is exposed between processes via ***windows*** (`mpi_win_create`).
- A communication is initiated by an ***origin*** process and involves accessing the memory (via MPI get/put) of a ***target*** process.
- All RMA communications are non-blocking and take place within ***epochs***, which control process synchronization.

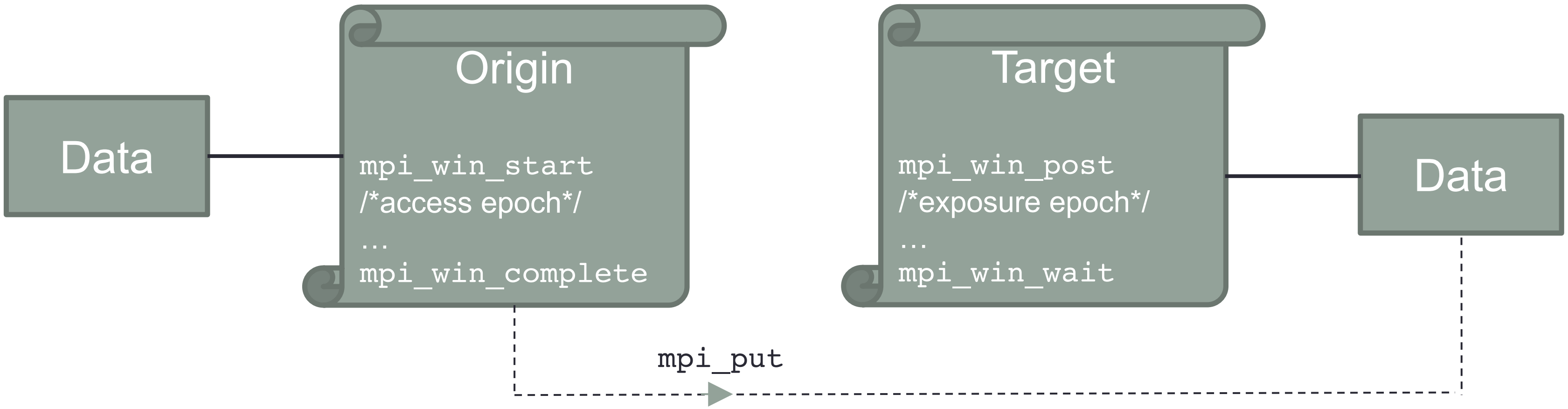
Active Target: Fence and PSCW (**P**ost**S**tart**C**omplete**W**ait)
Passive Target: Lock synchronisation

Fence Synchronisation - MPI RMA Active Target

- Call `mpi_win_fence` to open and close an epoch.
- Each process synchronises with every other process in the windows communicator.
- Can provide assertions to permit optimised operation, such as `MPI_MODE_NOPRECEDE` and `MPI_MODE_NOSUCCEED`, but MPI implementation can ignore these.
- Fence is the simplest method (similar to `mpi_barrier`) but some overhead due to all-to-all synchronization.

PSCW Synchronisation - MPI RMA Active Target

- Communication between an origin process and a target process requires the origin to enter an access epoch and the target to enter an exposure epoch.



- PostStartCompleteWait ensures that synchronisation occurs between communicating processes only (MPI group).

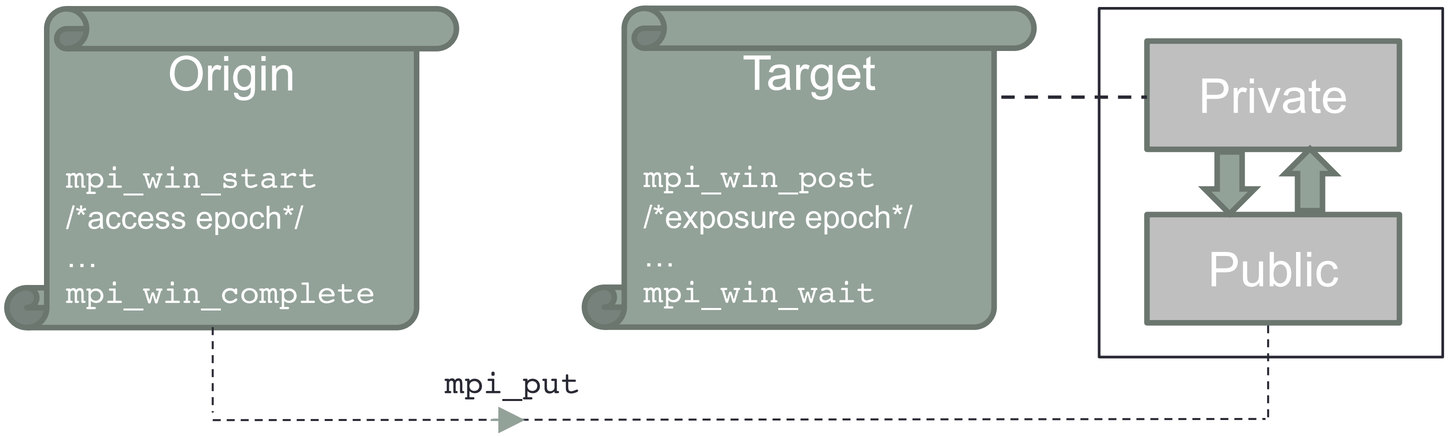
- Pertinent to MONC where comms are nearest neighbour rather than across all compute processes.

Lock Synchronisation - MPI RMA Passive Target

- Only the active process is involved in the synchronisation
  No interaction required by the target process.
- Origin process issues `mpi_win_lock` to start an access epoch, `mpi_win_unlock` closes the epoch.
- Locks can be shared or exclusive, i.e., only one process at a time can access the window of the target.

MPI RMA Memory Model

- MPI provides the concept of public and private copies of window data, the so called *separate* model.



- There is also the *unified* model, which requires a cache coherent machine: this allows certain synchronisations to be omitted.

- ARCHER MPI implementation (cray-mpich v7.5.5) supports the unified memory model.

MONC Halo Communication - MPI P2P

Initiate Halo Communication

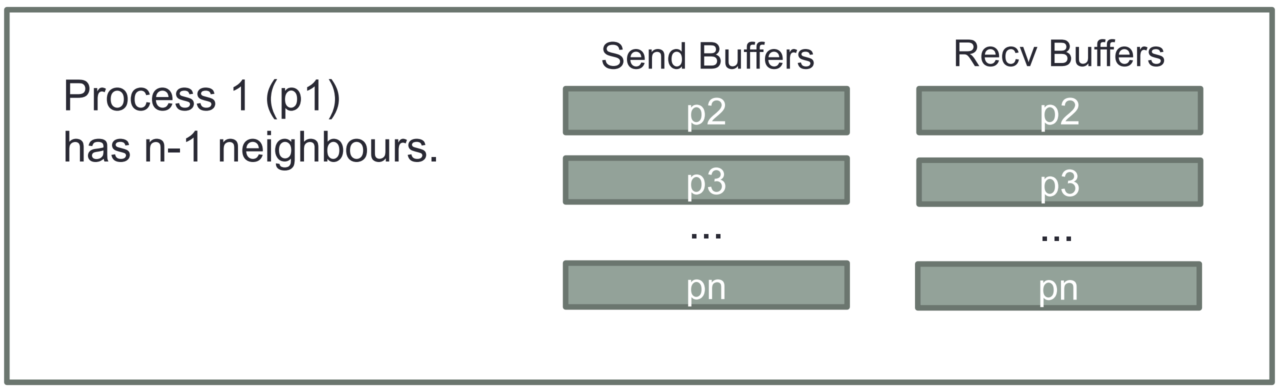
Initiate Non-blocking Halo Swap

Complete Non-blocking Halo Swap

Finalise Halo Communication

Initiate Halo Communication

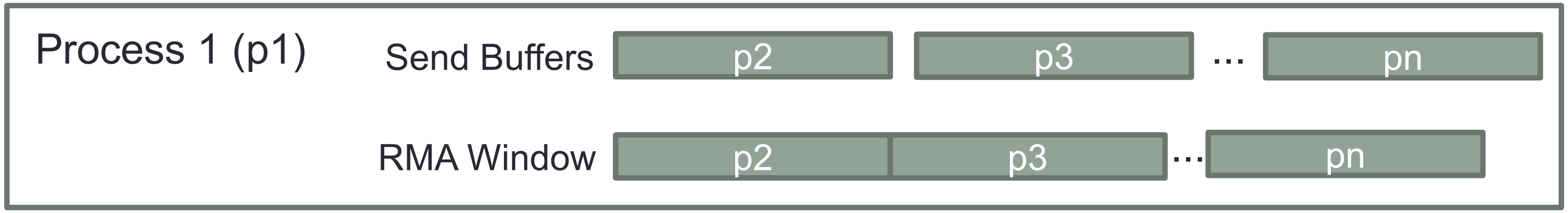
Setup a halo-swapping context that is returned to component to allow halo swapping of specific fields.



MONC Halo Communication - MPI RMA

Initiate Halo Communication

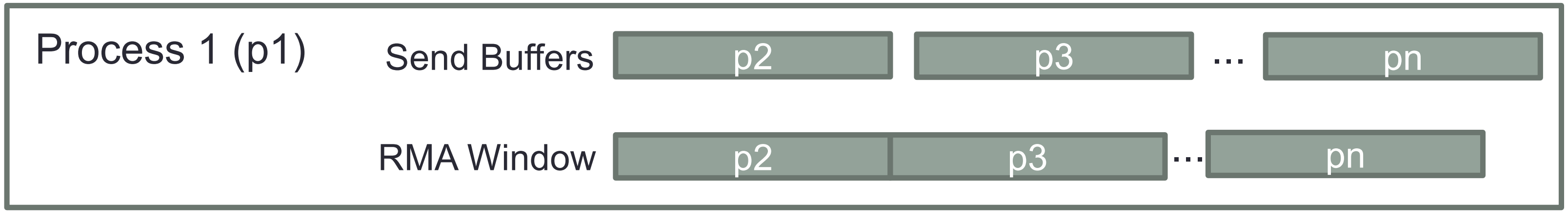
Create RMA window: a contiguous 1D buffer that equals total size of neighbour *send* buffers.



MONC Halo Communication - MPI RMA

Initiate Halo Communication

Create RMA window: a contiguous 1D buffer that equals total size of neighbour *send* buffers.



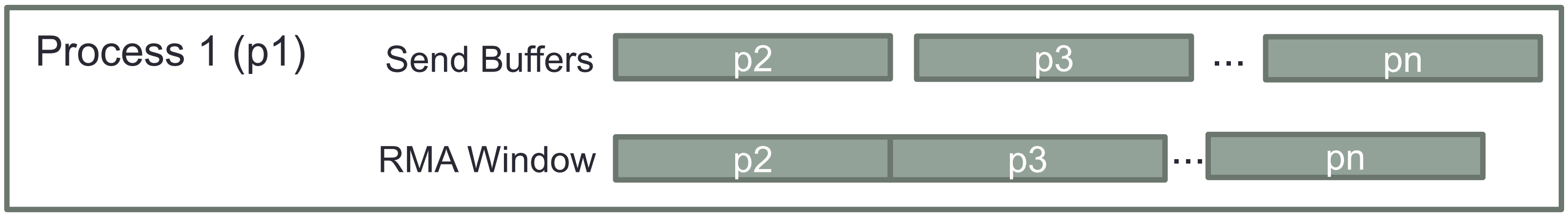
All processes (1-n) swap window offsets (using issends, irecvs followed by waitall).

For example, Process 1 sends the offset that it uses for each neighbour to that neighbour; and all other processes do the same.

MONC Halo Communication - MPI RMA

Initiate Halo Communication

Create RMA window: a contiguous 1D buffer that equals total size of neighbour *send* buffers.



All processes (1-n) swap window offsets (using issends, irecvs followed by waitall).

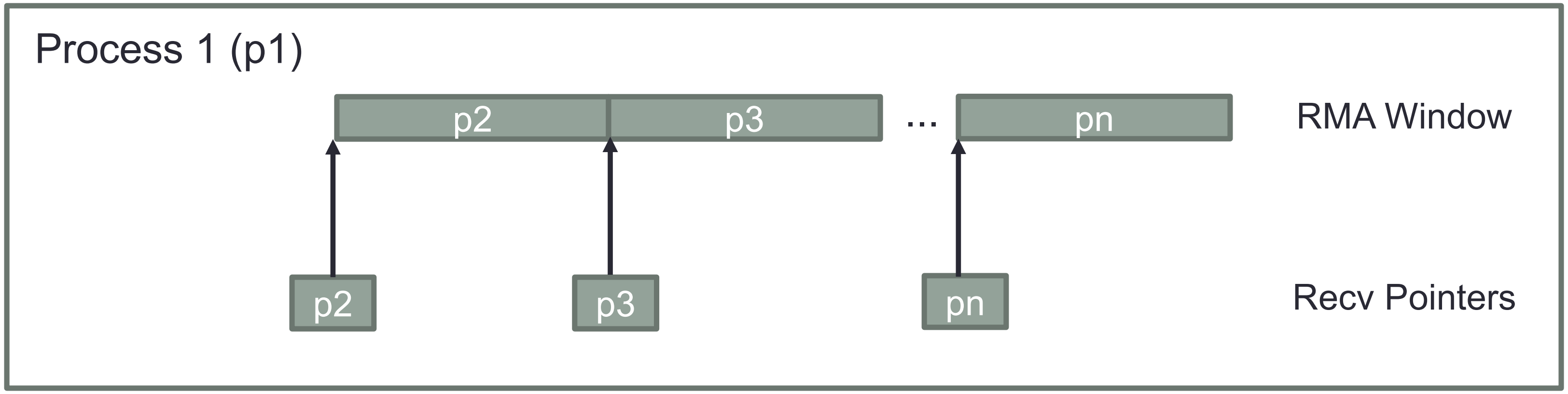
For example, Process 1 sends the offset that it uses for each neighbour to that neighbour; and all other processes do the same.

So, later when p3 does an `mpi_put` to p1 it can use the offset that p1 provided, i.e., when p3 writes into p1's window it writes to an offset reserved for p3.

MONC Halo Communication - MPI RMA

Initiate Halo Communication

What's happened to the neighbour *recv* buffers?



The neighbour recv buffers have been replaced by pointers into the RMA window. The data is directly unpacked from the RMA window into the halo cells.

Tricky, as Fortran does not directly support pointer arithmetic, so this work is done in C code instead. ISO C bindings provide the `c_ptr` derived type (used with `mpi_alloc_mem`) as well as the `c_f_pointer` subroutine.

MONC Halo Communication - MPI RMA Fence

Initiate Halo Communication

per code run

- Create the RMA window.

Initiate Non-blocking Halo Swap

per time step

- Pack domain data into RMA window
- `mpi_win_fence(no_precede, ...)`
- Send packed data via non-blocking RMA get.
- `mpi_win_fence(no_succeed, ...)`

`no_precede` may not block leading to data inconsistency

Complete Non-blocking Halo Swap

- Unpack received data into the appropriate halo locations.

Finalise Halo Communication

- Free the RMA window.

MONC Halo Communication - MPI RMA Fence

Initiate Halo Communication

per code run

- Create the RMA window.

Initiate Non-blocking Halo Swap

per time step

- Pack domain data into send buffers.
- `mpi_win_fence(no_precede, ...)`
- Send packed data via non-blocking RMA put.
- `mpi_win_fence(no_succeed, ...)`

Complete Non-blocking Halo Swap

- Unpack received data into the appropriate halo locations.

Finalise Halo Communication

- Free the RMA window.

MONC Halo Communication - MPI RMA Fence

Initiate Halo Communication

per code run

- Create the RMA window.
- `mpi_win_fence(no_precede,...)`

Initiate Non-blocking Halo Swap

per time step

- Pack domain data into send buffers.
- Send packed data via non-blocking RMA puts.

Complete Non-blocking Halo Swap

- `mpi_win_fence(no_succeed, ...)`.
- Unpack received data into the appropriate halo locations.
- `mpi_win_fence(no_precede, ...)`.

Finalise Halo Communication

- `mpi_win_fence(no_succeed, ...)`
- Free the RMA window.

MONC Halo Communication - MPI RMA PSCW

Initiate Halo Communication

per code run

- Create the RMA window and the neighbour MPI group.
- `mpi_win_post(...)`
- `mpi_win_start(...)`

Initiate Non-blocking Halo Swap

per time step

- Pack domain data into send buffers.
- Send packed data via non-blocking RMA puts.

Complete Non-blocking Halo Swap

- `mpi_win_complete(...)`
- `mpi_win_wait(...)`
- Unpack received data into the appropriate halo locations.
- `mpi_win_post(...)`
- `mpi_win_start(...)`

Finalise Halo Communication

- `mpi_win_complete(...)`
- `mpi_win_wait(...)`
- Free neighbour MPI group and RMA window.

MONC Halo Communication - MPI RMA Lock (First Attempt)

Initiate Halo Communication

per code run

- Create the RMA window.

Initiate Non-blocking Halo Swap

per time step

- Pack domain data into send buffers.
- For each neighbour...

```
mpi_win_lock(shared,...)
mpi_put(...)
mpi_win_unlock(...)
```

Complete Non-blocking Halo Swap

- Use `mpi_win_flush()` and `mpi_barrier` to ensure all comms have completed.
- Unpack received data into the appropriate halo locations.

Finalise Halo Communication

- Free the RMA window.

MONC Halo Communication - MPI RMA Lock

Initiate Halo Communication

per code run

- Create the RMA window.
- `mpi_win_lock_all(nocheck, ...)`

Initiate Non-blocking Halo Swap

per time step

- Pack domain data into send buffers.
- Send packed data via non-blocking RMA puts.

Complete Non-blocking Halo Swap

- Call `mpi_irecv` with empty message for all neighbours.
- **`mpi_win_flush_all(...)`**
- Call `mpi_isend` with empty message for all neighbours.
- Use `mpi_testall` to determine when irecv/isend comms have completed.
- **`if (SeparateRMAMemoryModel) mpi_win_sync(...)`**
- Unpack received data into the appropriate halo locations.

Finalise Halo Communication

- `mpi_win_unlock_all(...)`
- Free the RMA window.

Introducing ARCHER

Advanced Research Computing High End Resource

EPSRC

NERC SCIENCE OF THE ENVIRONMENT

CRAY THE SUPERCOMPUTER COMPANY

archer

www.archer.ac.uk

epcc

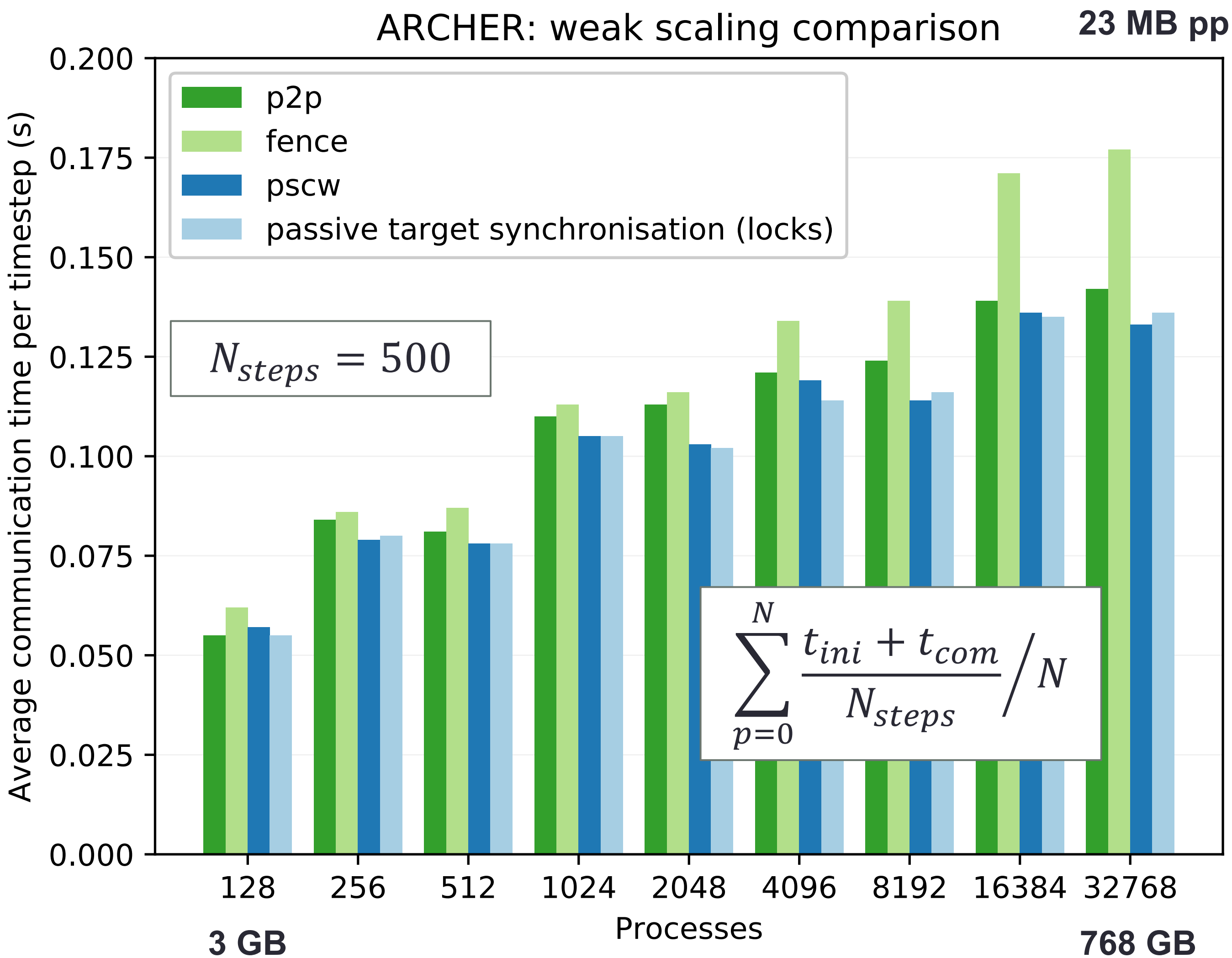
THE UNIVERSITY OF EDINBURGH

Introducing ARCHER

- Cray XC30 MPP, 4920 Compute Nodes
  - Dual Intel Xeon processors (Ivy Bridge), 24 cores, 64 GB
- Tests conducted on production system
- Supports Cray's Distributed Memory Application API (DMAPP)
  - A communication library (v1.6.0) that can call straight through to the underlying Aries networking ASIC on the Cray and implements many RMA operations directly in hardware.

MONC test case

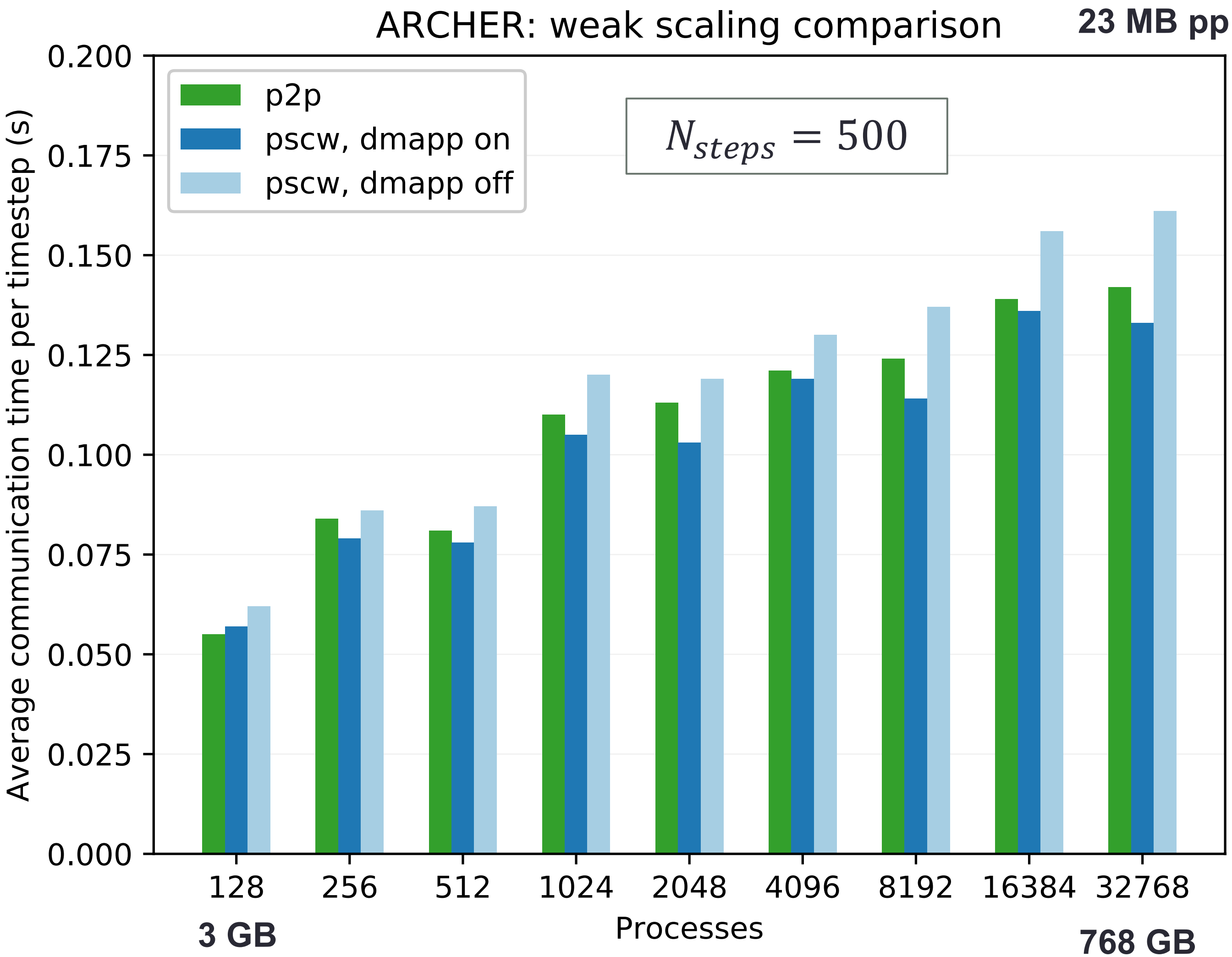
- A standard test case for marine stratocumulus cloud was used which contains 25 Q (moisture) fields, as well as fields for temperature, pressure and wind.
- All of these fields need to be swapped once per time step.
- MONC mode compiled using Cray Fortran Compiler v8.4.1 and Cray MPICH v7.5.5.
- Run 1 process per core.



ARCHER: weak scaling comparison
23 MB pp
Average communication time per timestep (s)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
p2p
fence
pscw
passive target synchronisation (locks)
$N_{steps} = 500$
$\sum_{p=0}^{N} \frac{t_{ini} + t_{com}}{N_{steps}} / N$
128
256
512
1024
2048
4096
8192
16384
32768
Processes
3 GB
768 GB

archer

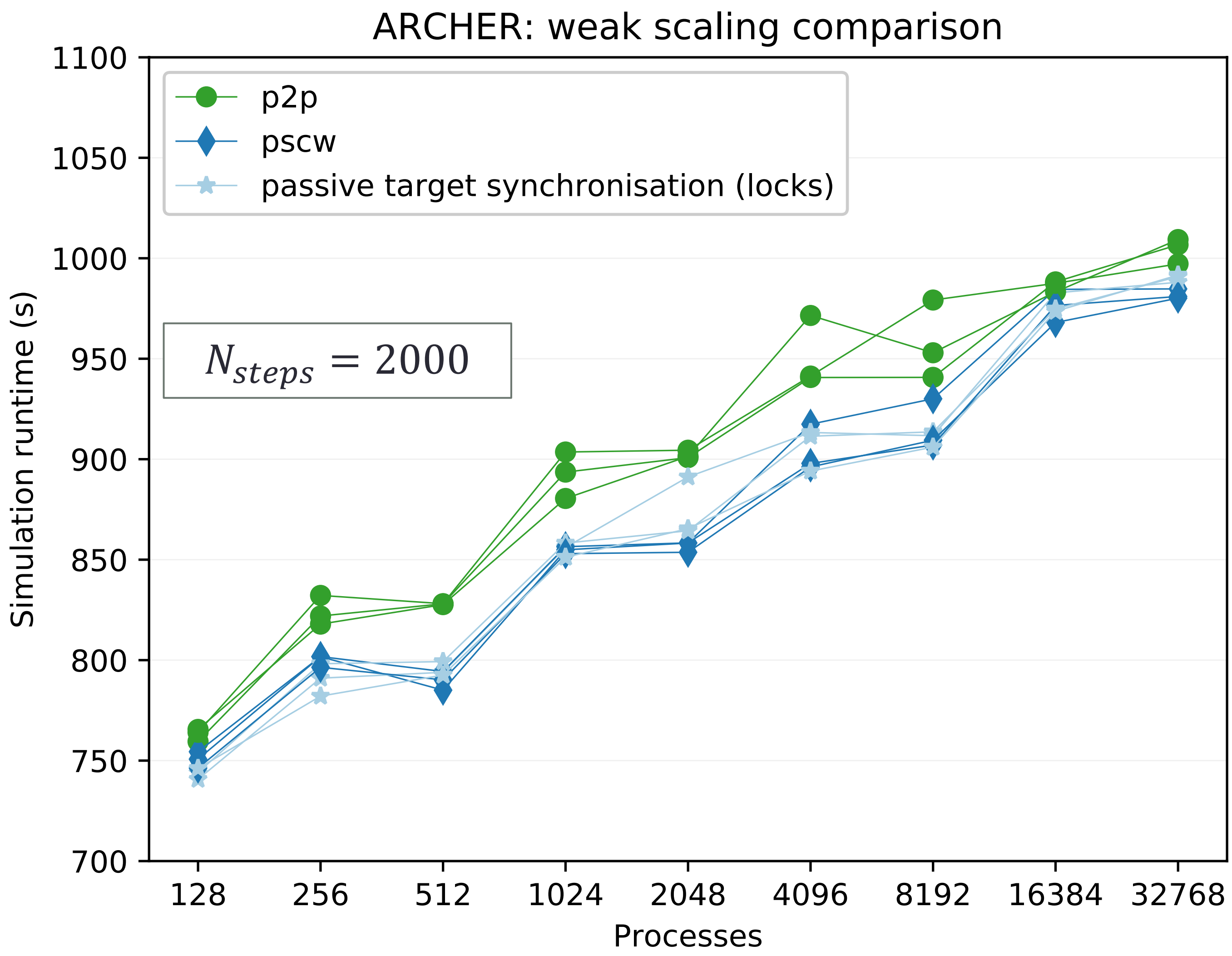
epcc



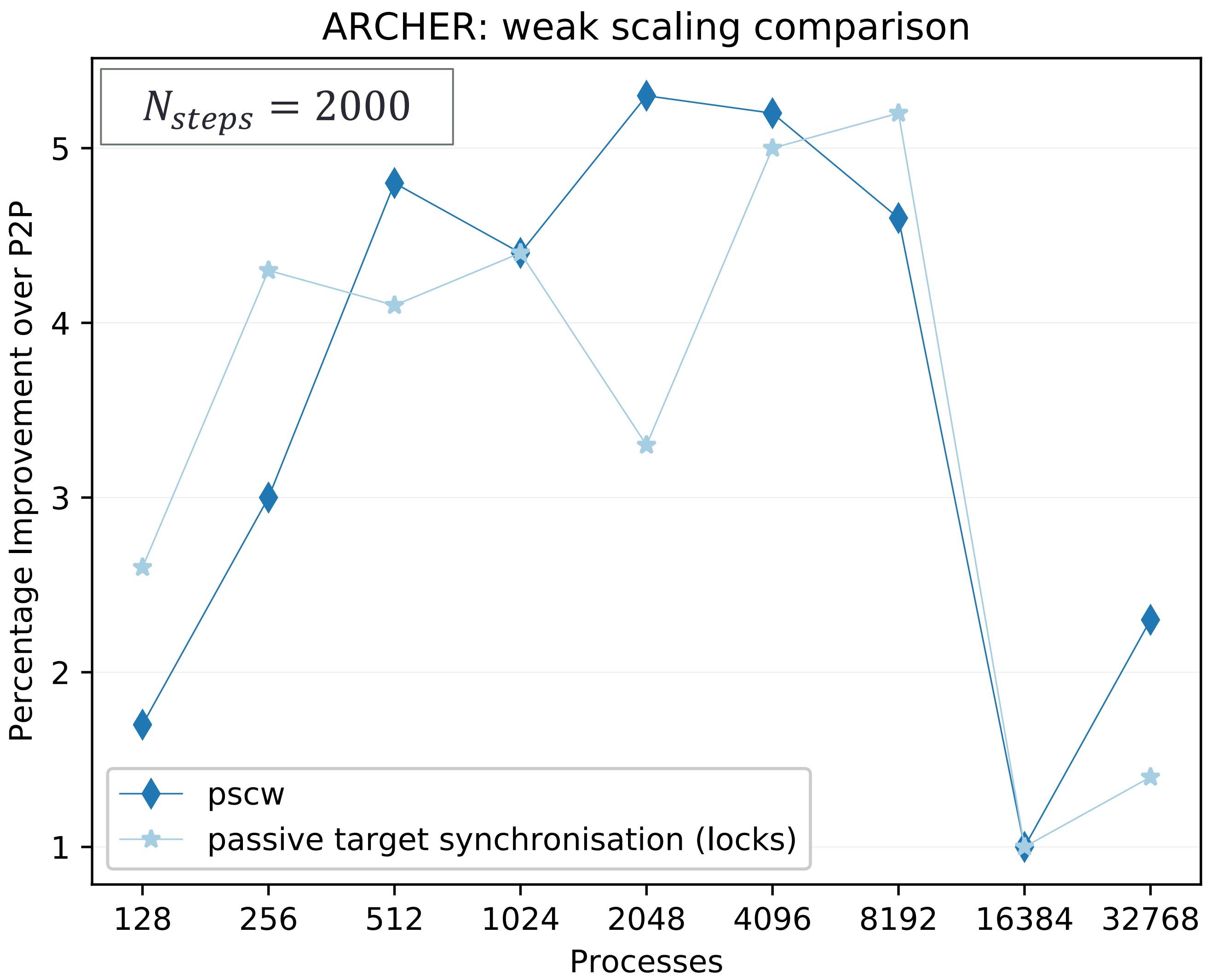
archer

epcc

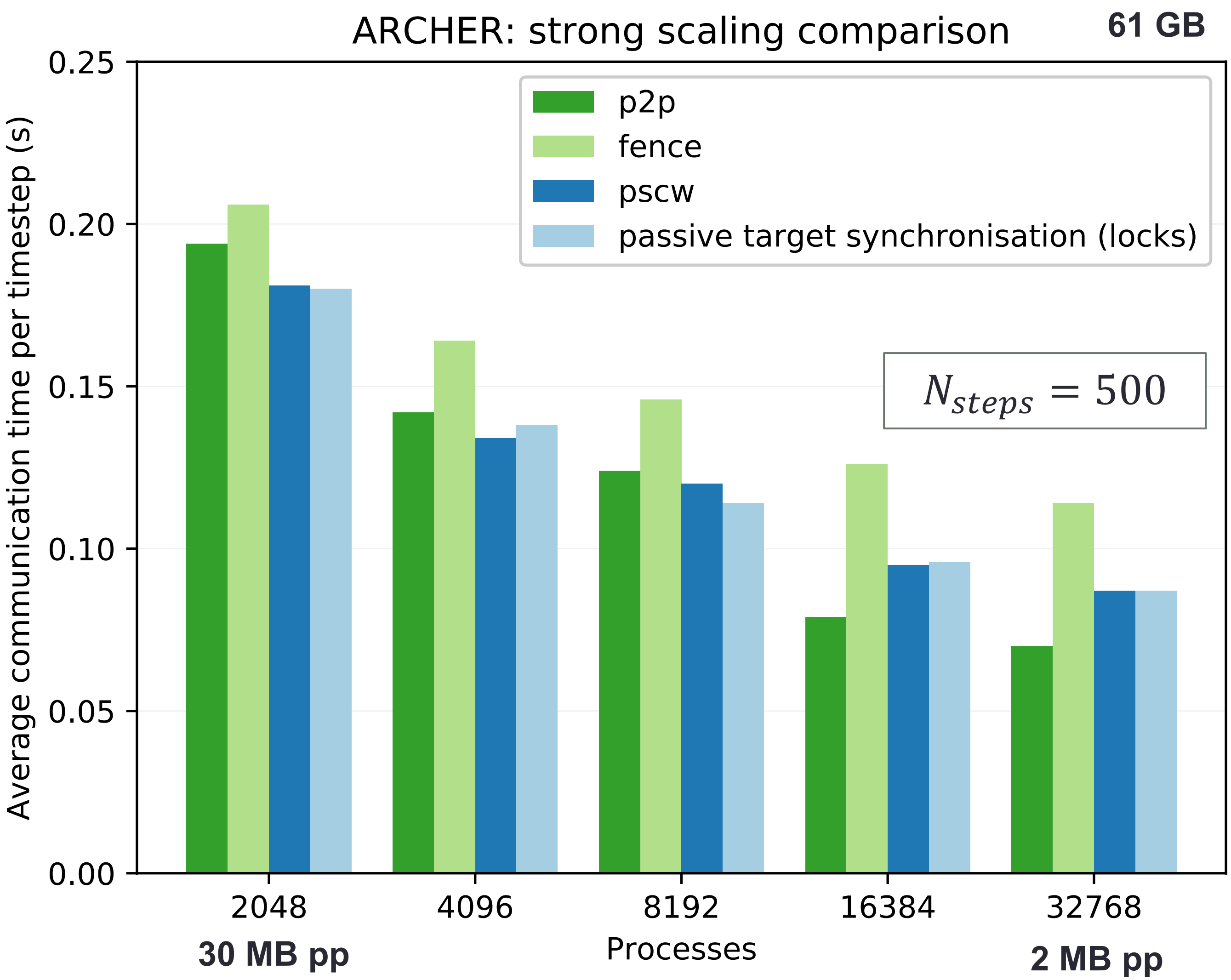
THE UNIVERSITY OF EDINBURGH



ARCHER: weak scaling comparison
p2p
pscw
passive target synchronisation (locks)
$N_{steps} = 2000$
Simulation runtime (s)
1100
1050
1000
950
900
850
800
750
700
128
256
512
1024
2048
4096
8192
16384
32768
Processes



ARCHER: weak scaling comparison
$N_{steps} = 2000$
Percentage Improvement over P2P
Processes
128
256
512
1024
2048
4096
8192
16384
32768
1
2
3
4
5
pscw
passive target synchronisation (locks)



ARCHER: strong scaling comparison
61 GB
Average communication time per timestep (s)
0.25
0.20
0.15
0.10
0.05
0.00
p2p
fence
pscw
passive target synchronisation (locks)
$N_{steps} = 500$
2048
4096
8192
16384
32768
30 MB pp
Processes
2 MB pp

Conclusions

Results show that RMA is worth pursuing, if you are careful with RMA implementation (slides 13-22) and use DMAPP.

Conclusions

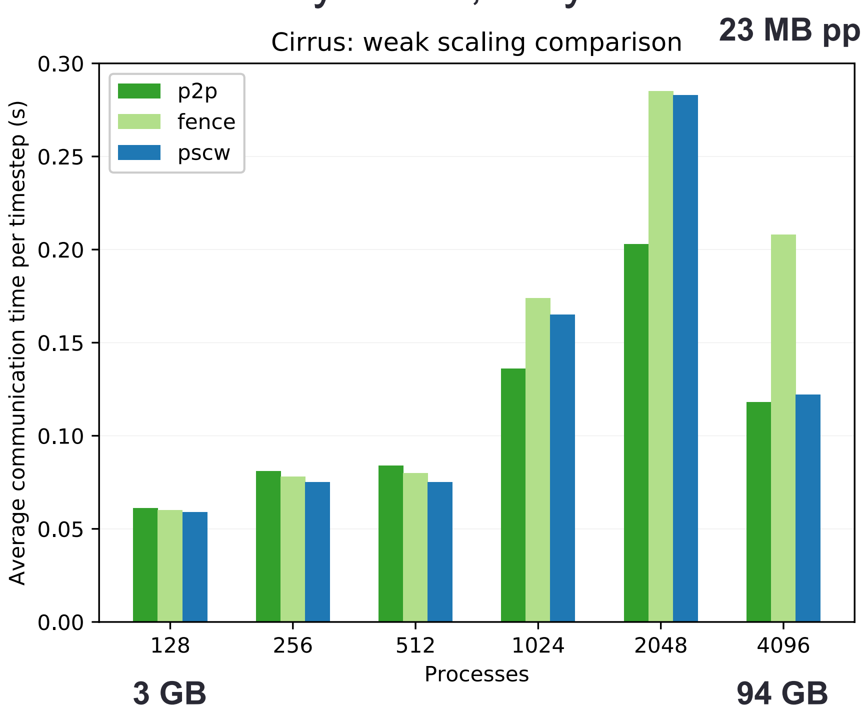
Results show that RMA is worth pursuing, if you are careful with RMA implementation (slides 13-22) and you use DMAPP.

However, what is achievable with ARCHER Cray XC30, may not be the case for other platforms.

Cirrus SGI ICE (20,160 cores)

Compute node: two 18-core Broadwell processes, 256 GB.

SGI MPT v2.14 does *not* implement passive target synchronisation.



Conclusions

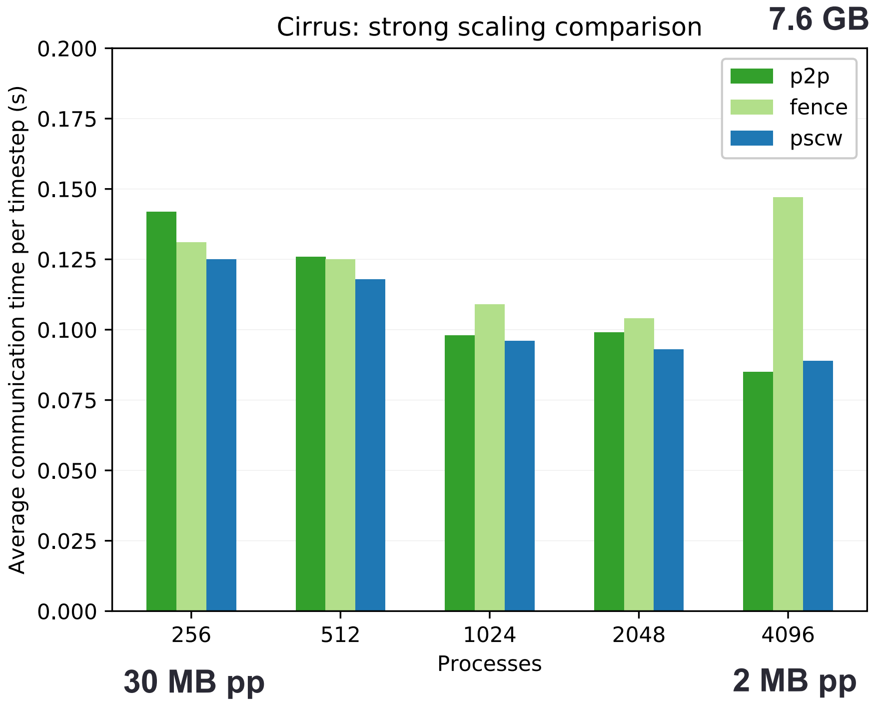
Results show that RMA is worth pursuing, if you are careful with RMA implementation (slides 13-22) and you use DMAPP.

However, what is achievable with ARCHER Cray XC30, may not be the case for other platforms.

Cirrus SGI ICE (20,160 cores)

Compute node: two 18-core Broadwell processes, 256 GB.

SGI MPT v2.14 does *not* implement passive target synchronisation.



Conclusions

Results show that RMA is worth pursuing, if you are careful with RMA implementation (slides 13-22) and you use DMAPP.

And of course MPI implementations will continue to evolve.
Expectation is that RMA performance will improve.

But, so could P2P...

ARCHER Cray XC30:
cray-mpich v7.7.0 module introduces “optimized message matching”.

`Initial results have shown improvements of as much as 16% in some micro-benchmarks`

`Cray Release Notes`