Performance of Fortran Coarrays on the Cray XE6

CUG 2012

David Henty
EPCC, The University of Edinburgh
d.henty@epcc.ed.ac.uk

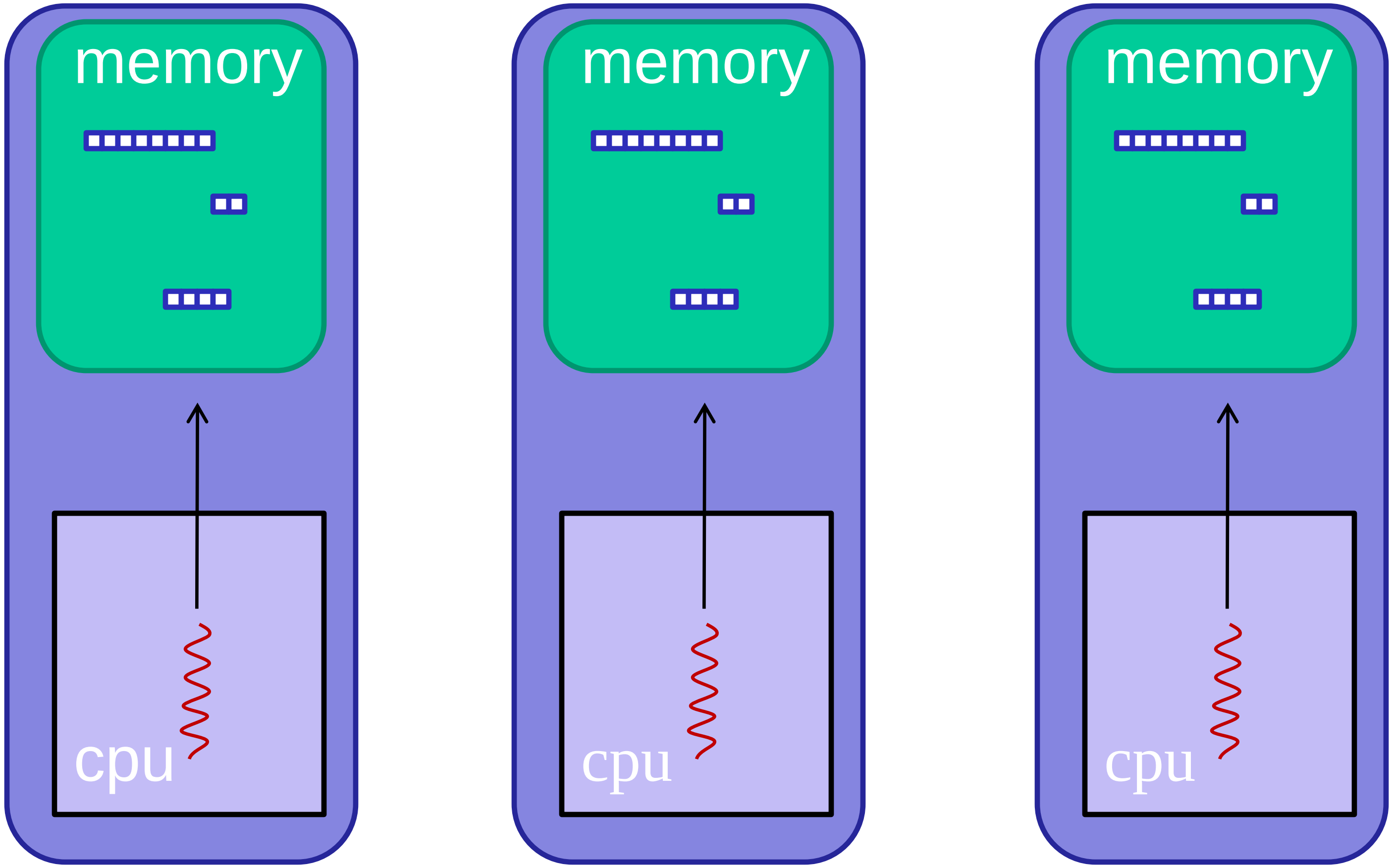
Contents

- Overview of PGAS
- MPI vs CAF vs UPC
- Fortran coarray syntax
- Benchmark design
- Platforms
- Results
- Conclusions
- Further work
- Acknowledgements

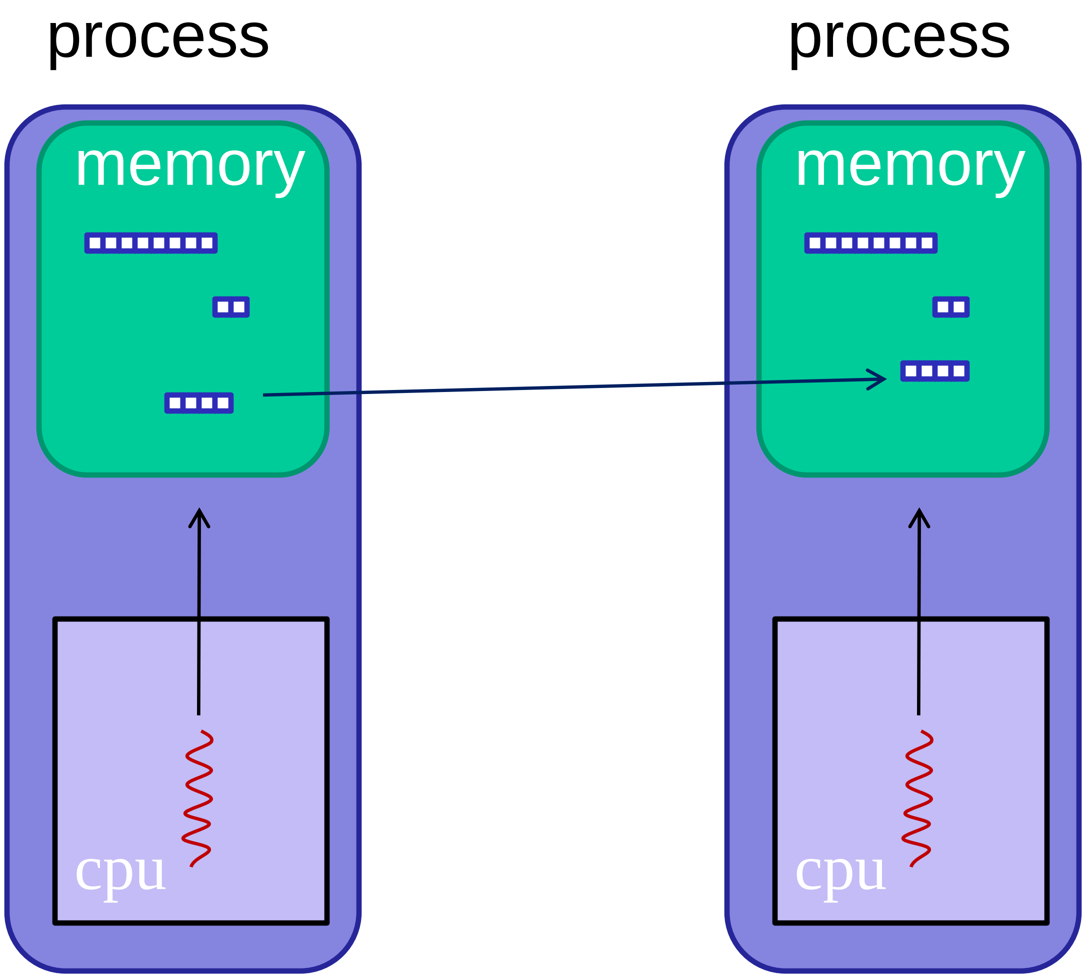
PGAS languages and exascale

- Issues foreseen for MPI at large scale
  - manycore memory footprint
  - excessive synchronisation
  - high latency
  - message matching overheads
  - …
- Partitioned Global Address Space models
  - private and shared data
  - direct read/write from/to remote memory
  - compiler support
  - no explicit buffering
- Two main implementations
  - Fortran Coarrays (part of Fortran 2008 standard)
  - Unified Parallel C

Message-Passing

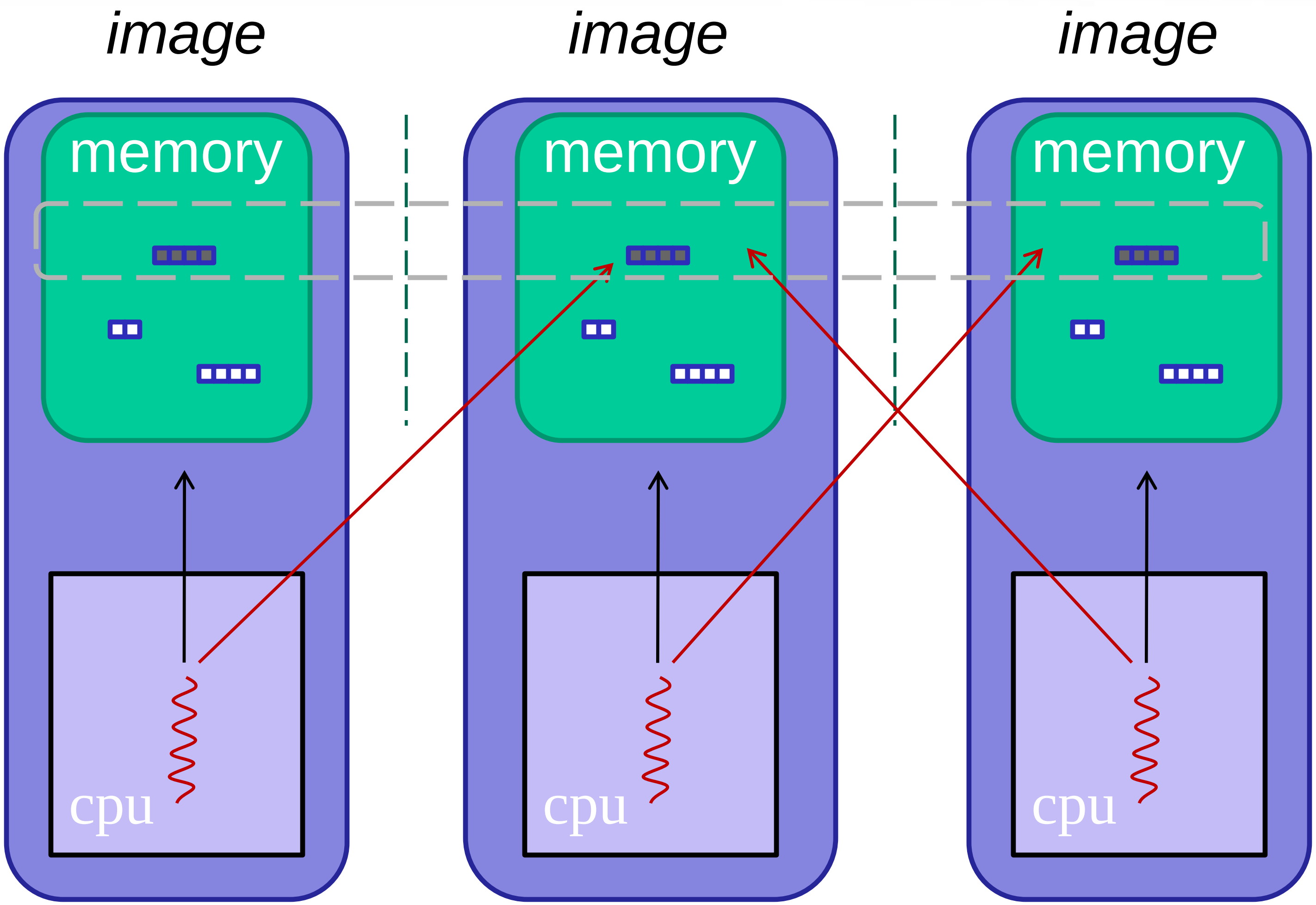


MPI

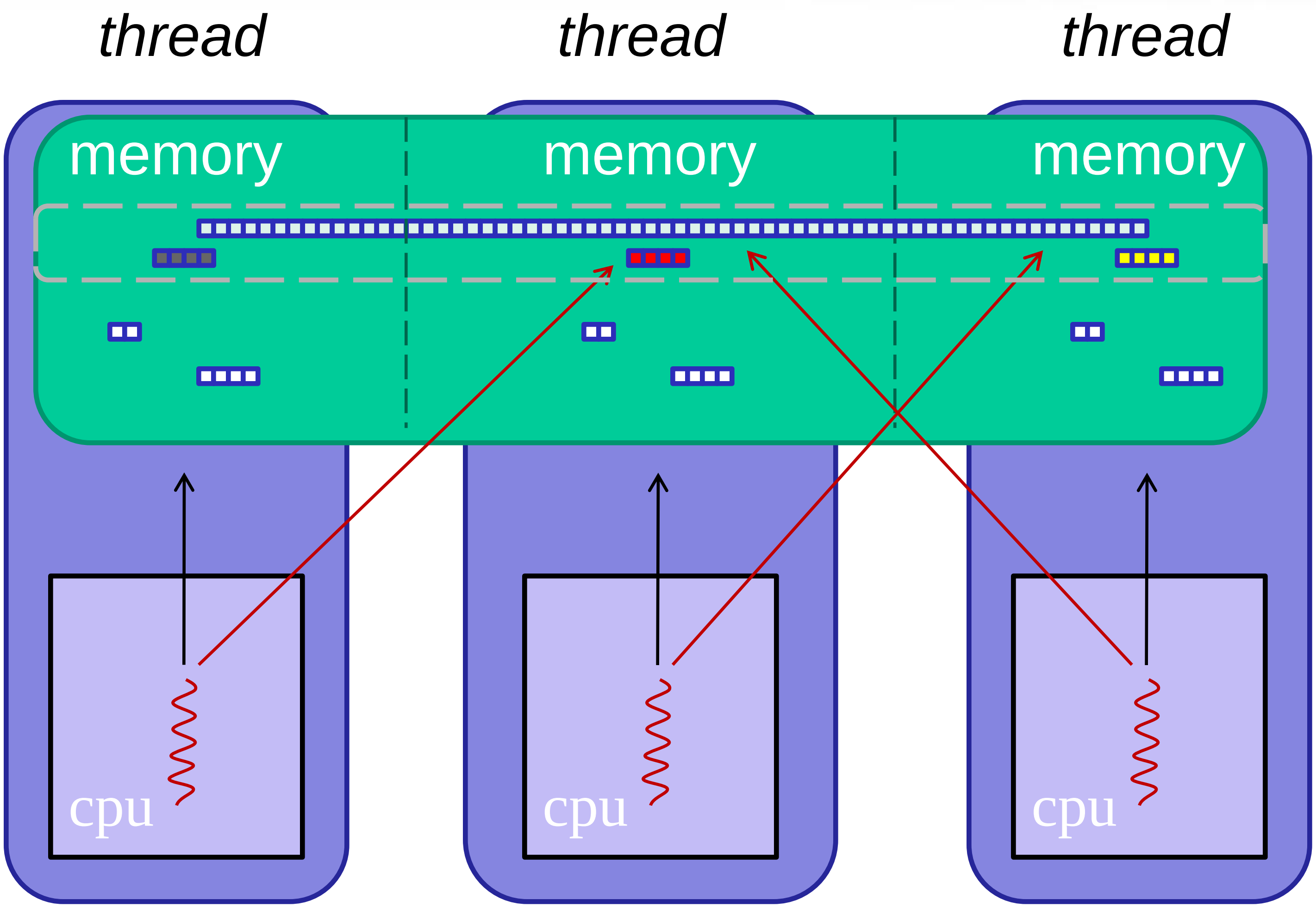


`MPI_Send(a,...,1,…)` **`MPI_Recv(a,...,0,…)`**

Fortran coarrays



UPC



Minimal syntax extension to Fortran

- SPMD execution model on multiple *images*
- Declare data to be remotely accessible

```
real, dimension(10), codimension[*] :: x
real x(10)[*]
```

- Access remote data directly

```
x(2)     = x(3)[7]   ! remote read from image 7
x(6)[4] = x(1)       ! remote write  to image 4
```

- Explicit synchronisation: global and point-to-point

```
sync all; sync images(imagelist)
```

- Supports multiple dimensions and codimensions

What to benchmark?

- Minimal syntax
  - plus a few synchronisation intrinsics
  - no collectives (yet …)
  - different to, e.g., MPI and UPC
- Need to decide on particular data access and communications patterns
- Several ways to describe same pattern
  - array syntax, explicit loops, inline or subroutine, …
  - important to understand compiler capabilities

Initial benchmark

- Single contiguous point-to-point read and write
- Multiple contiguous point-to-point read and write
- Strided point-to-point read and write
- All basic synchronisation operations
- Various representative synchronisation patterns
- Halo-swapping in a multi-dimensional regular domain decomposition

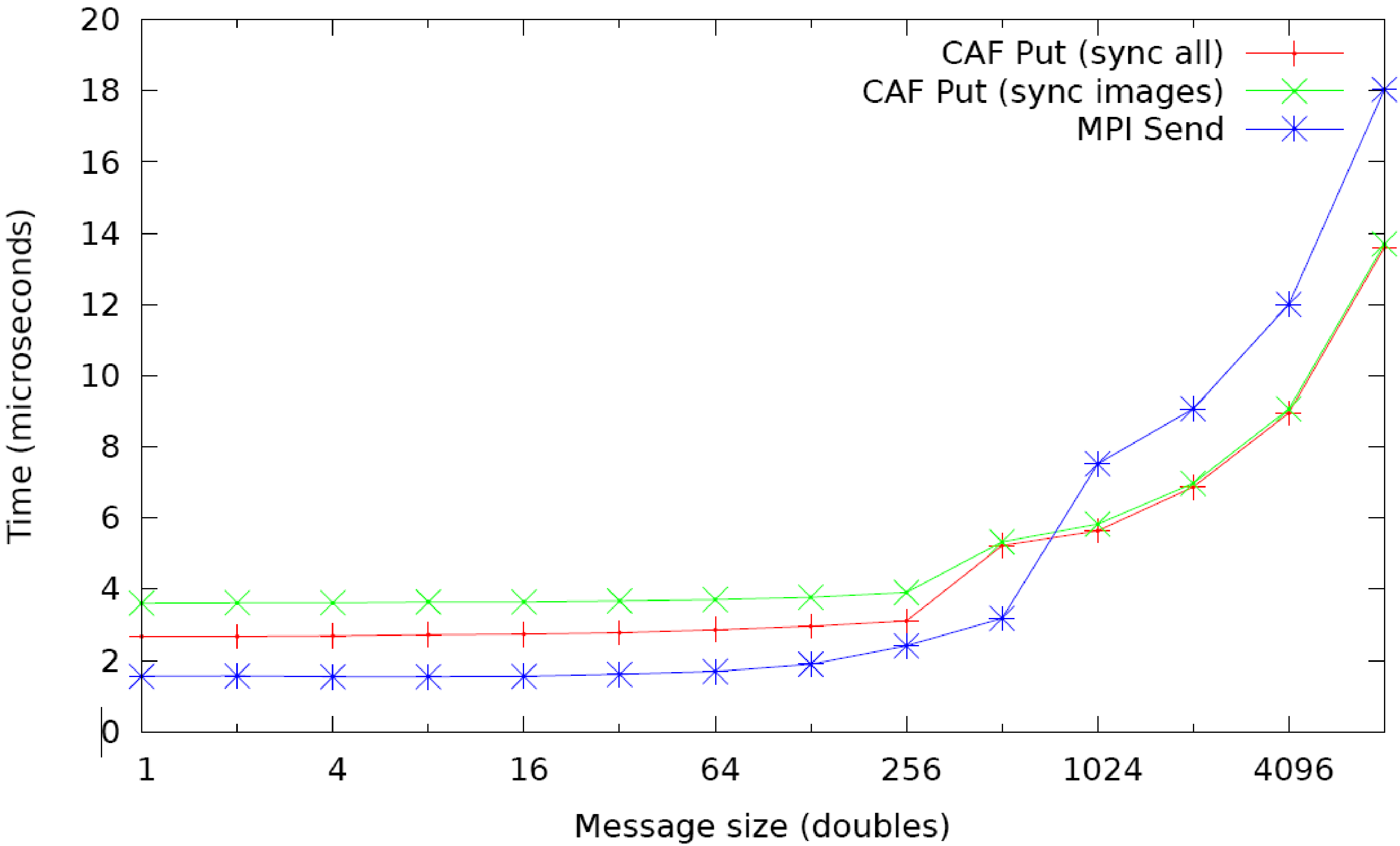
- All communications include synchronisation cost
  - use double precision (8-byte) values as the basic type
  - use Fortran array syntax (mostly)

- Synchronisations overhead measured separately as overhead = (delay + sync) – delay

Platforms

- HECToR
  - UK National Supercomputer
  - 90,000 cores, 2x16-core Interlagos nodes, GEMINI interconnect
  - plus internal Cray XE6 development system

Ping-Pong

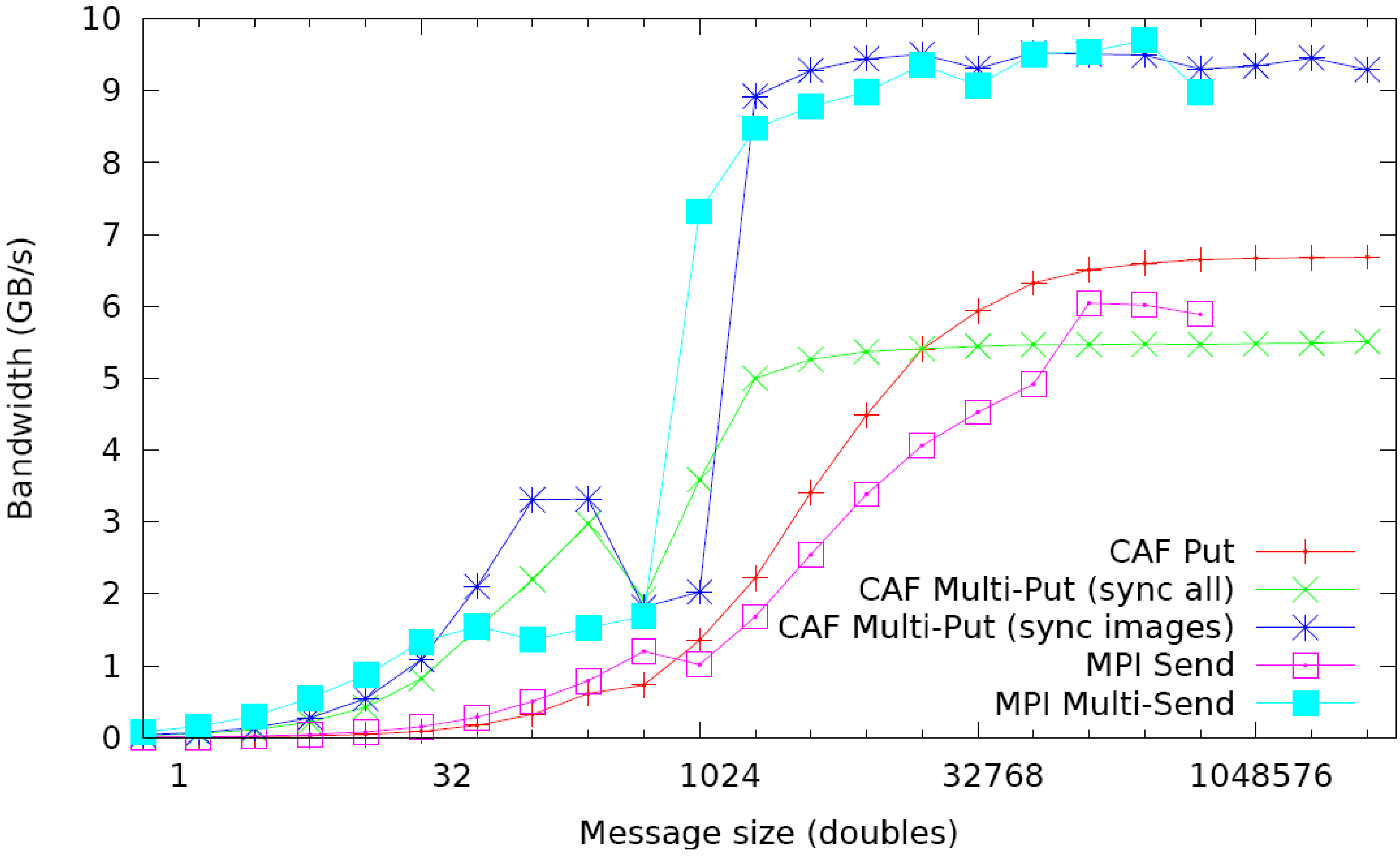


CAF Put (sync all)
CAF Put (sync images)
MPI Send
Time (microseconds)
Message size (doubles)
20
18
16
14
12
10
8
6
4
2
0
1
4
16
64
256
1024
4096

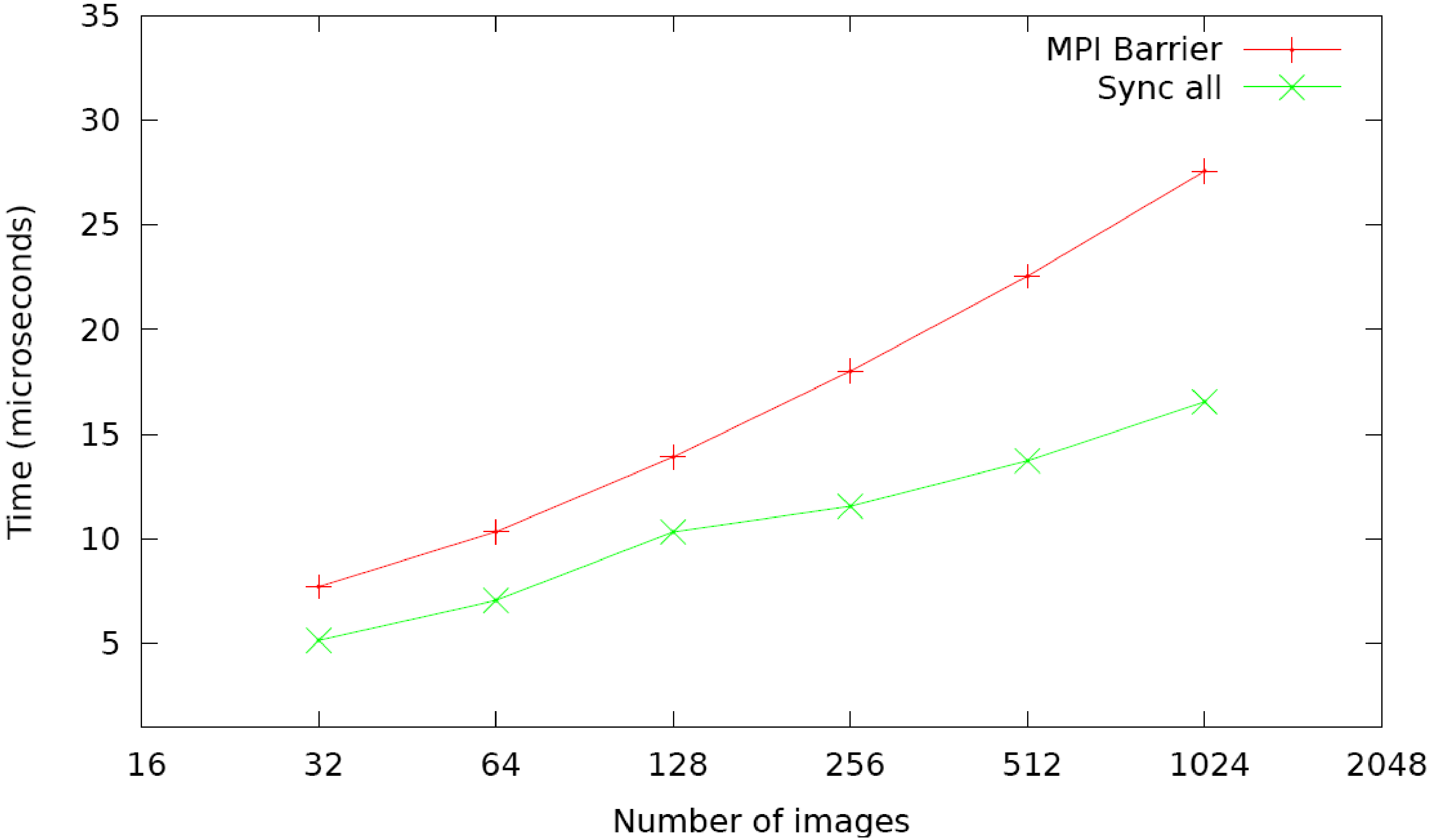
Synchronisation

- All coarray operations require explicit synchronisation calls for correctness
  - typically two!
- Ping-pong benchmark includes synchronisation
  - dominates the transfer time
  - actual transfers take about 1.3 μs per double (see later)
- Measure synchronisation explicitly in benchmark
  - global (sync all): 2.3 μs
  - pt-to-pt (sync images): 3.6 μs
- Not completely consistent, but implies latency worse than MPI
- However…
  - MPI synchronises per message pair
  - coarray synchronisation is between images
- Potential to issue many coarray ops between single sync pair

Multi Ping-Pong



Global Synchronisation



Strided Ping-Pong

Mode	time per double (μs)	total bandwidth (MB/s)
Single ping-pong		
Put (DO loop)	1.3	6.1
Put (array syntax)	0.3	25.3
Get (DO loop)	1.2	6.5
Get (array syntax)	1.2	6.6
Multi ping-pong		
Put (DO loop)	0.06	129
Put (array syntax)	0.02	364
Get (DO loop)	0.06	142
Get (array syntax)	0.06	141

Point-to-point synchronisation patterns

- Should be advantageous when running on many images
  - images must mutually synchronise

```
sync images(imagelist)
```

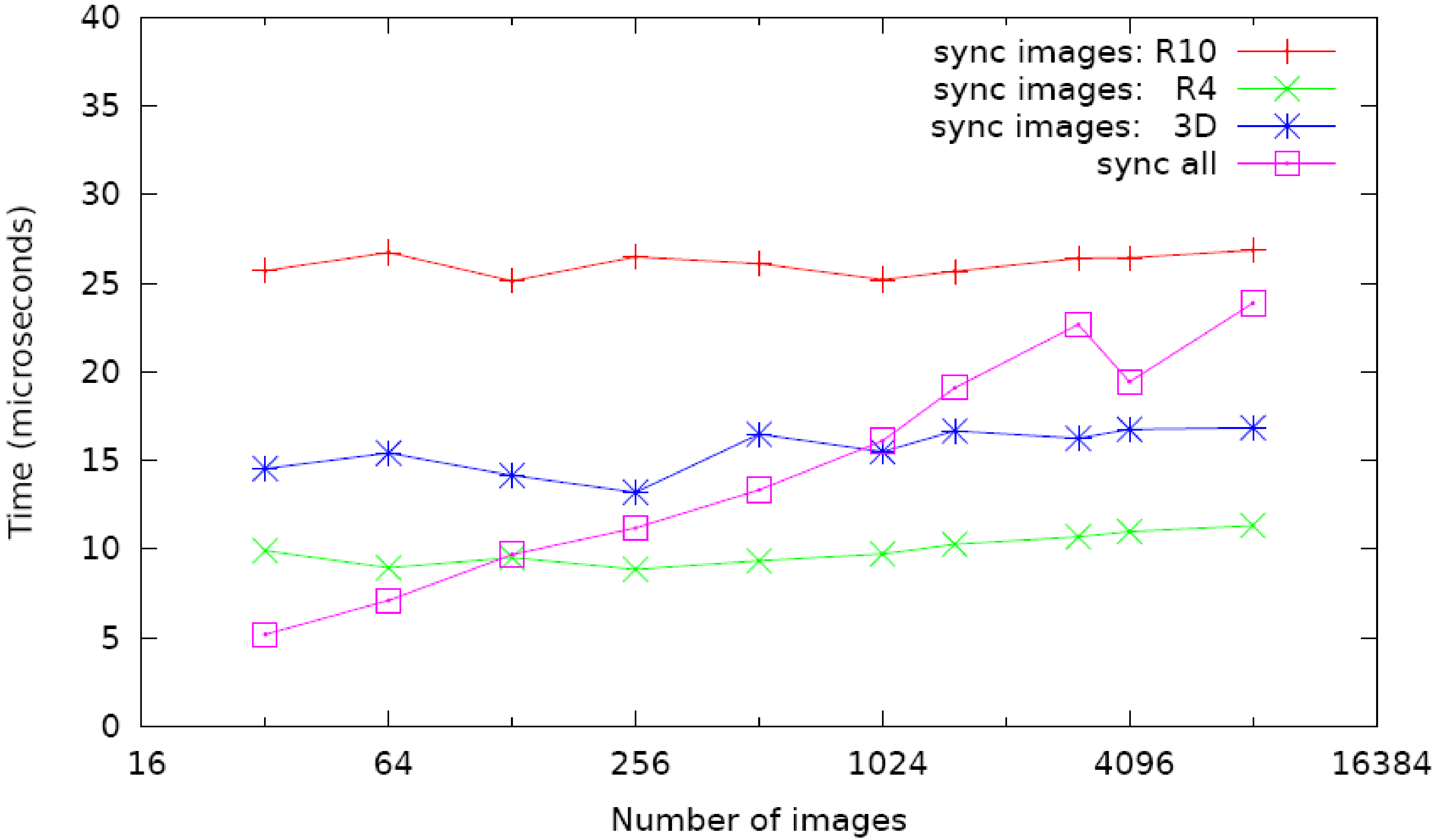
  - user must ensure they all match up

- L*X*
  - each image pairs with *X* images in a (periodic) line
  - e.g. L4 pairs with image-2, image-1, image+1 and image+2

- R*X*
  - same as L*X* but images are randomly permuted
  - ensures many off-node messages

- 3D
  - six neighbours in a 3D cartesian grid

Point-to-point Synchronisation Perfomance



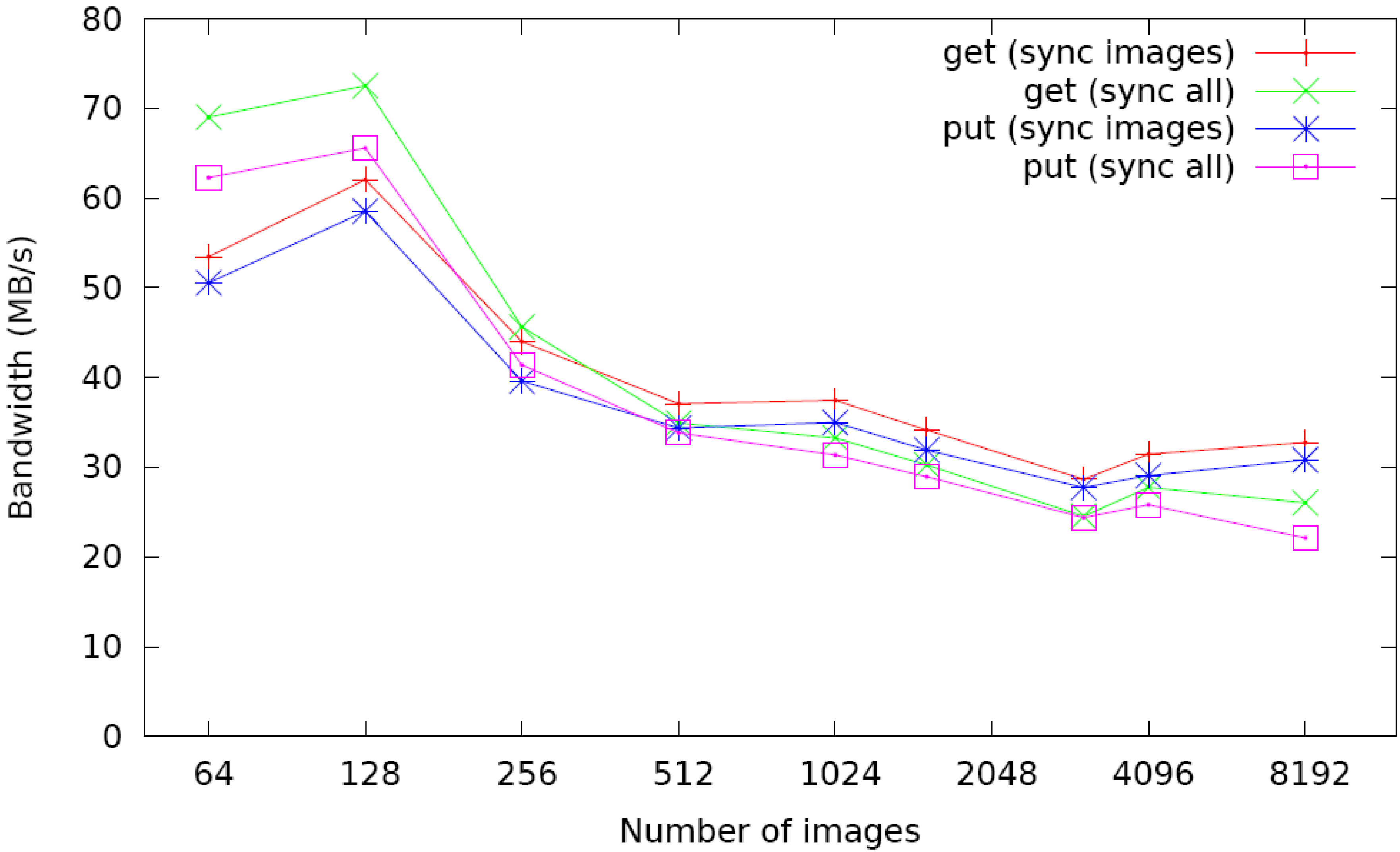
3D halo swap

- Fixed volume, weak scaling

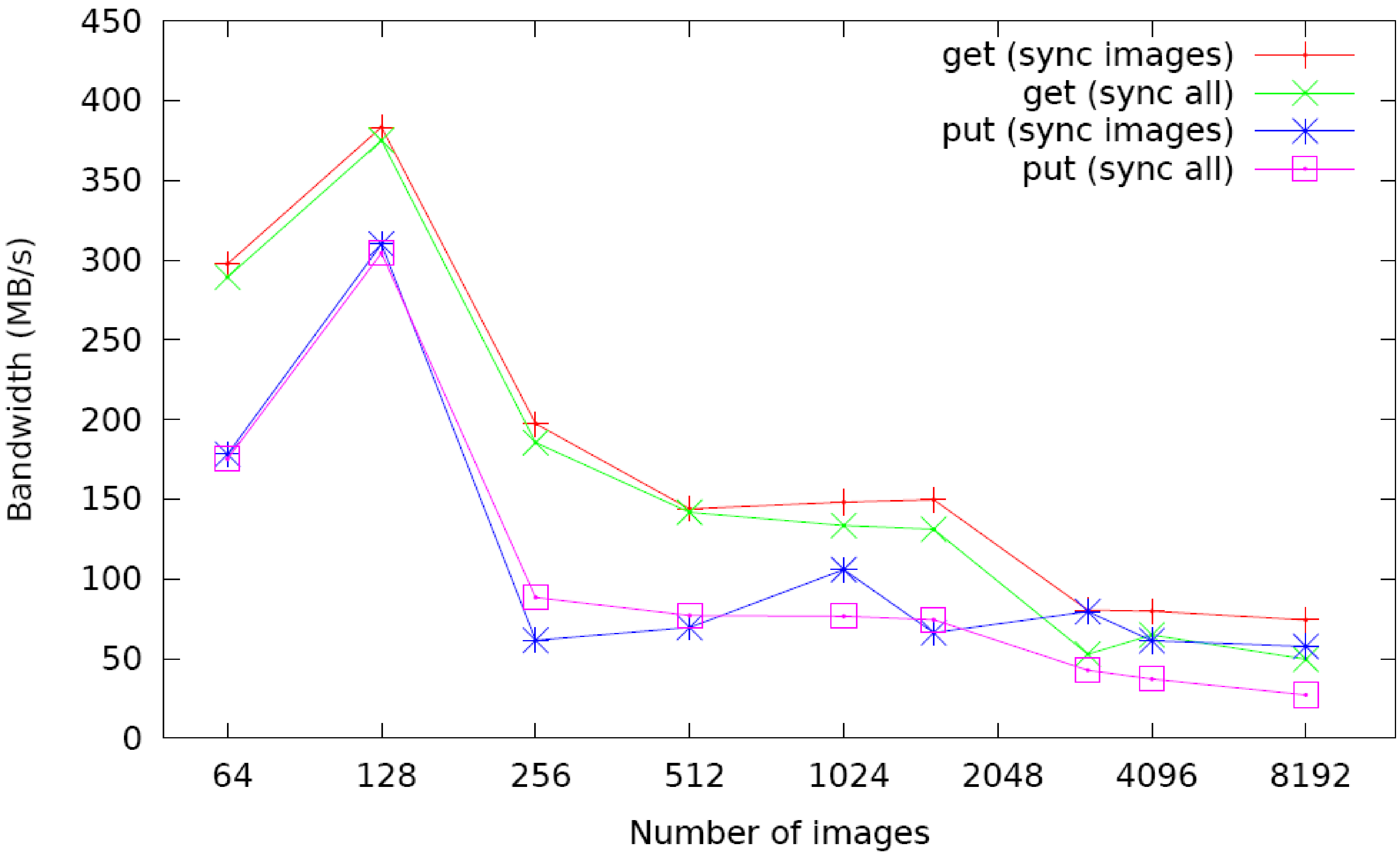
- 3D domain decomposition as suggested by MPI_Cart_dims
  – reversed for Fortran ordering

- Measure time for complete halo swap operation
  – remote read (get) and write (put)
  – sync images and sync all

3D Halo Swaps (10x10x10)



3D Halo Swaps (100x100x100)



Conclusions

- Simple benchmarks give good insight into performance characteristics of Fortran coarrays
- Cray compiler quite mature
  - good ability to pattern match (i.e. “vectorise” remote operations)
  - performance generally good on XE6
  - significant advantages from point-to-point synchronisation
  - but explicit buffering would still be faster than strided access
- Essential to have verification tests
  - benchmark has a “do verify” option which impacts performance
  - previous Cray compilers have sometimes had bugs
  - e.g. wrong data transferred in 3D halo swaps
  - earliest Intel compiler implemented `sync images` as a no-op!
  - Cray bugs fixed quickly but you do need to detect them …

Further work

- Increase scope of benchmark
  - overlap of communications and calculations (via compiler directive)
  - more complicated data transfer patterns (e.g. for locks)
  - include proposed coarray collectives
- Tidy up and make available for general release

Acknowledgements

- People
  - Harvey Richardson (Cray Inc)
- Funding
  - Edinburgh University Exascale Technology Centre
  - CRESTA project: www.cresta-project.eu
  - "This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703"
- Facilities
  - HECToR, the UK's national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme
  - Cray Inc

Backup: Lock and Critical