Performance Optimisation on Xeon Phi

Adrian Jackson, Michele Weiland, Fiona Reid, Manos Farsarakis,
David Scott

adrianj@epcc.ed.ac.uk

epcc

PERFORMANCE DEOPTIMISATION ON XEON PHI

Adrian Jackson, Michele Weiland, Fiona Reid, Manos Farsarakis,
David Scott

adrianj@epcc.ed.ac.uk

epcc

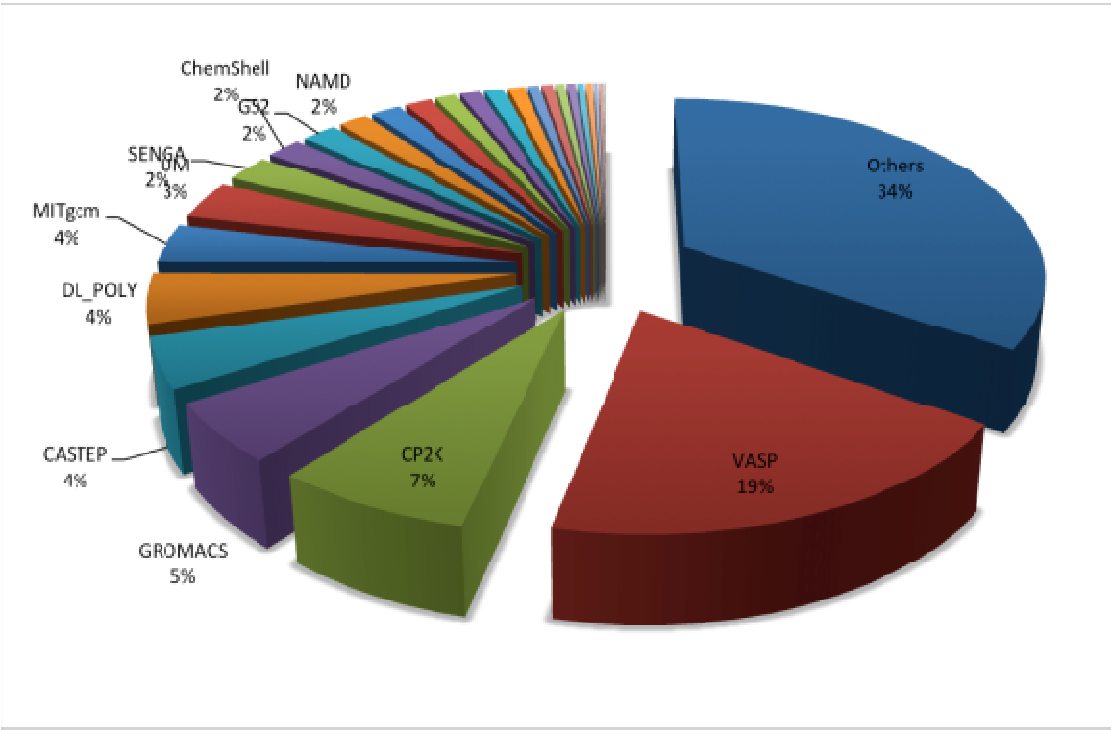
Intel's IPCC program

- Collaboration between Intel and leading Universities around the world
- "Intel® Parallel Computing Centers are universities, institutions, and labs that are leaders in their field, focusing on modernizing applications to increase parallelism and scalability through optimizations that leverage cores, caches, threads, and vector capabilities of microprocessors and coprocessors."

epcc

IPCC Proposal

- Two main aims
  - Port and optimise codes for Xeon Phi
  - Optimise codes for Xeon for ARCHER system



- Target ‘Grand Challenges’ codes which are also heavily used in the UK
  - codes we have strong knowledge of/working relationship with

epcc

Intel Xeon Phi

- Intel Larrabee: “A Many-Core x86 Architecture for Visual Computing”
  - Release delayed such that the chip missed competitive window of opportunity.
  - Larrabee was not released as a competitive product, but instead a platform for research and development (Knight’s Ferry).
- Knights Corner derivative chip
  - Intel Xeon Phi – co-processor
  - Many Integrated Cores (MIC) architecture. No longer aimed at graphics market
    - Instead “Accelerating Science and Discovery”
  - PCIe Card
  - 60 cores/240 threads/1.054 GHz
  - 8 GB/320 GB/s
  - 512-bit SIMD instructions
- Hybrid between GPU and many-core CPU

Intel Xeon Phi

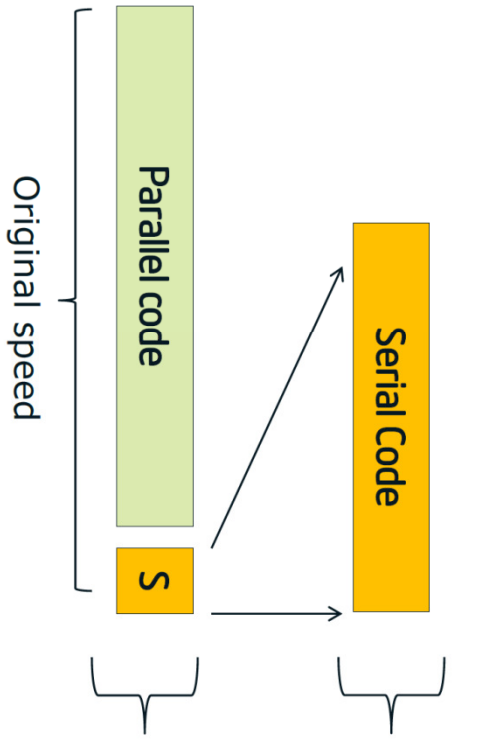
	3100 series	5100 series	7100 series
cores	57	60	61
Clock frequency	1.100 GHz	1.053 GHz	1.238 GHz
DP Performance	1 Tflops	1.01 TFlops	1.2 TFlops
Memory Bandwidth	240 GB/s	320 GB/s	352 GB/s
Memory	6 GB	8 GB	16 GB

- Usable in different ways
  - Offload kernels
  - “Native” direct run applications

Achievable Performance

- 1 to 1.2 TFlop/s double precision performance
  - Dependent on using 512-bit vector units
  - And FMA instructions
- 240 to 352 GB/s peak memory bandwidth
- ~60 physical cores
  - Each can run 4 threads
  - Must run at least 2 threads to get full instruction issue rate
  - Don’t think of it as 240 threads, think of it as 120 plus more if beneficial
- 2.5x speedup over host is good performance
  - Highly vectorised code, no communications costs
- MPI performance
  - Can be significantly slower than host

Serial code



Slide from Intel

The Serial Factor

Serial Factor =
Clock Factor * ILP Factor * Issue Factor
Where
Clock Factor = 2.6 /1.09

For FMA type calculations
ILP Factor*** = 2/2 =1

For non-FMA type calculations
ILP Factor = 2/1

FMA**
x4.77
slower

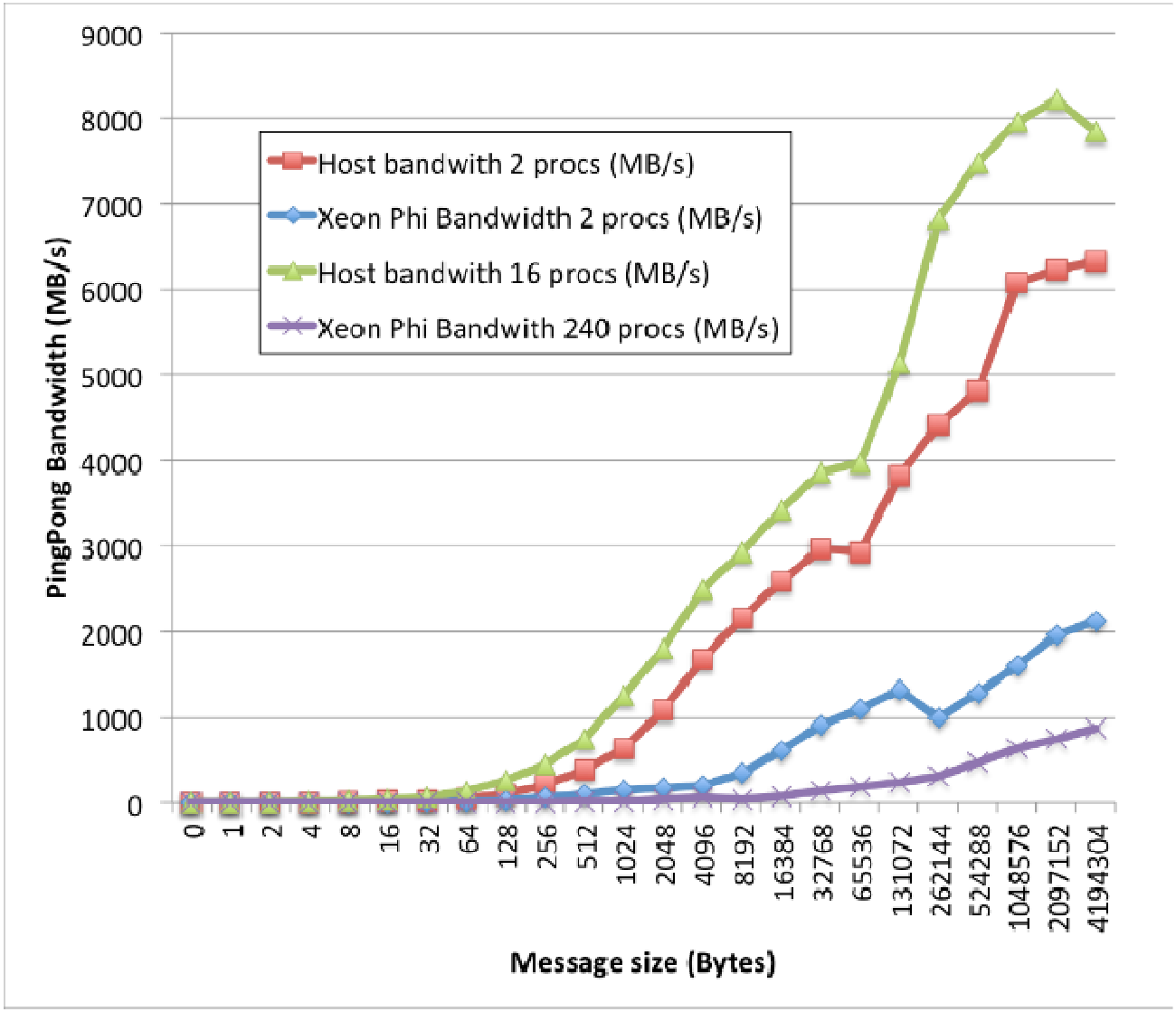
Issue factor =
Num cycles to issue instruction on Phi /
Num cycles to issue instruction on Xeon
= 2/1

Non-FMA
x9.54
slower

Note: in single threaded code Xeon Phi uses two cycles to issue an instruction
(in threaded mode it takes just one cycle)

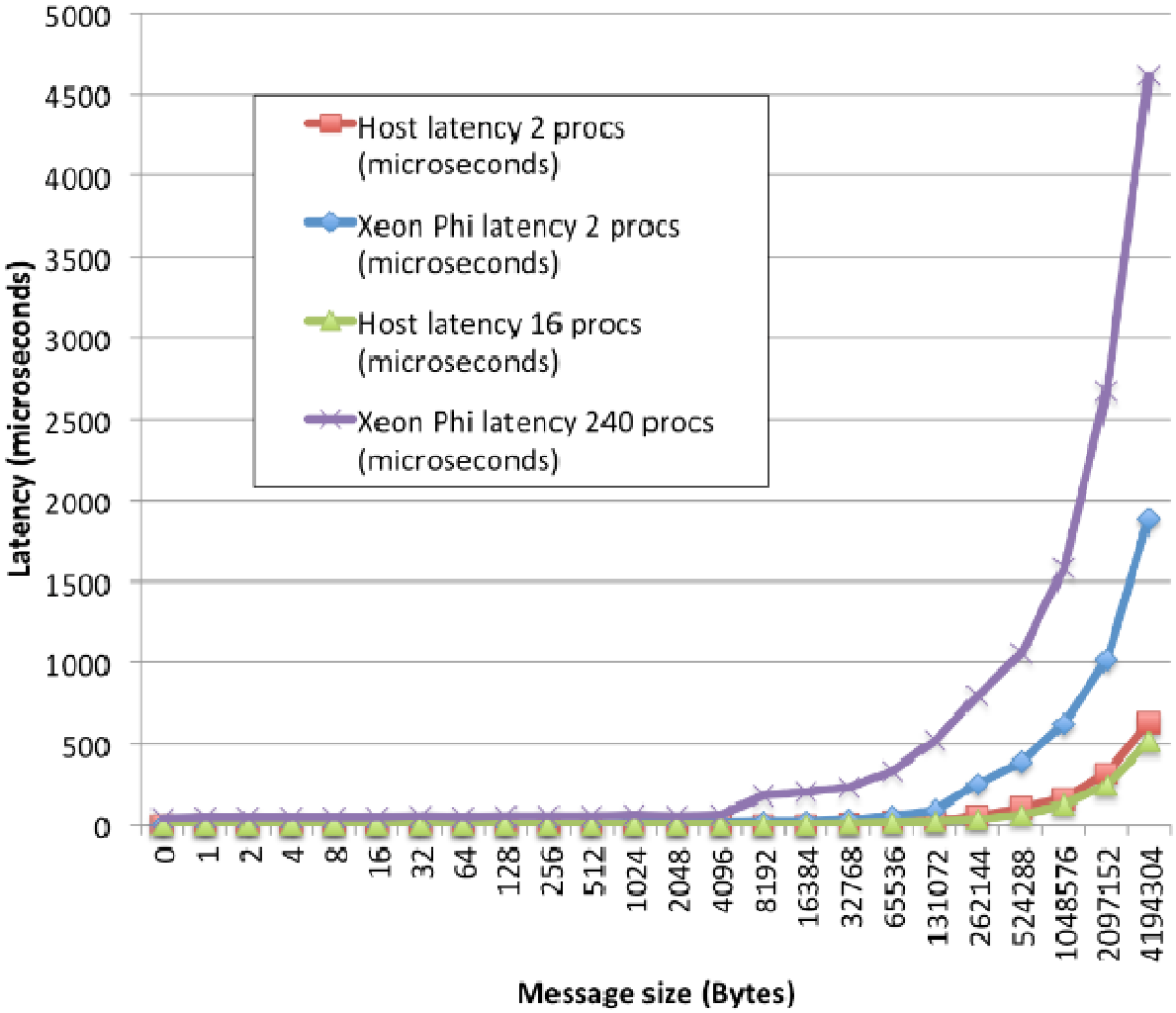
** FMA: source code is capable of using Fused Multiple Add when built for Xeon Phi

PingPong Bandwidth



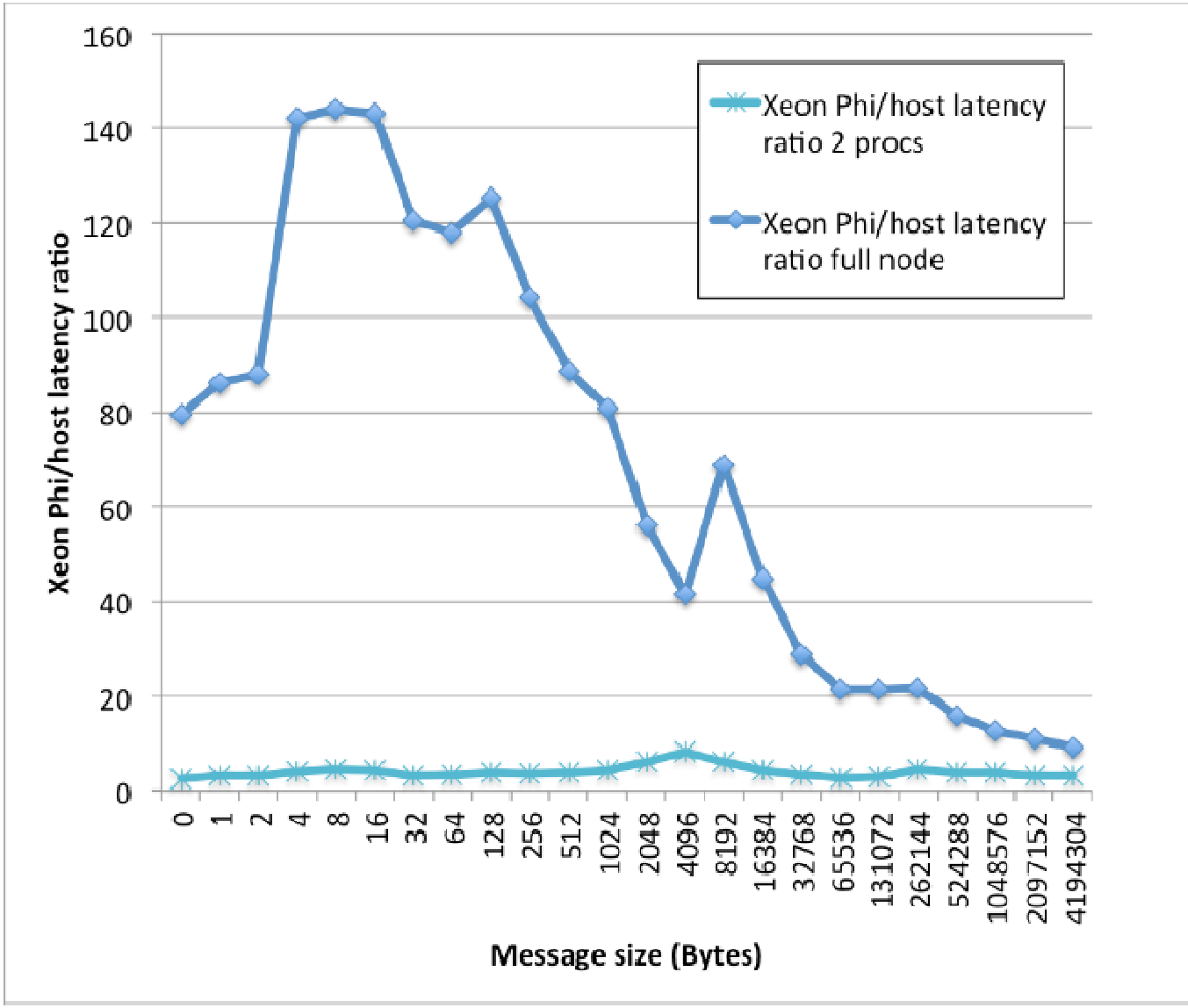
epcc

PingPong Latency



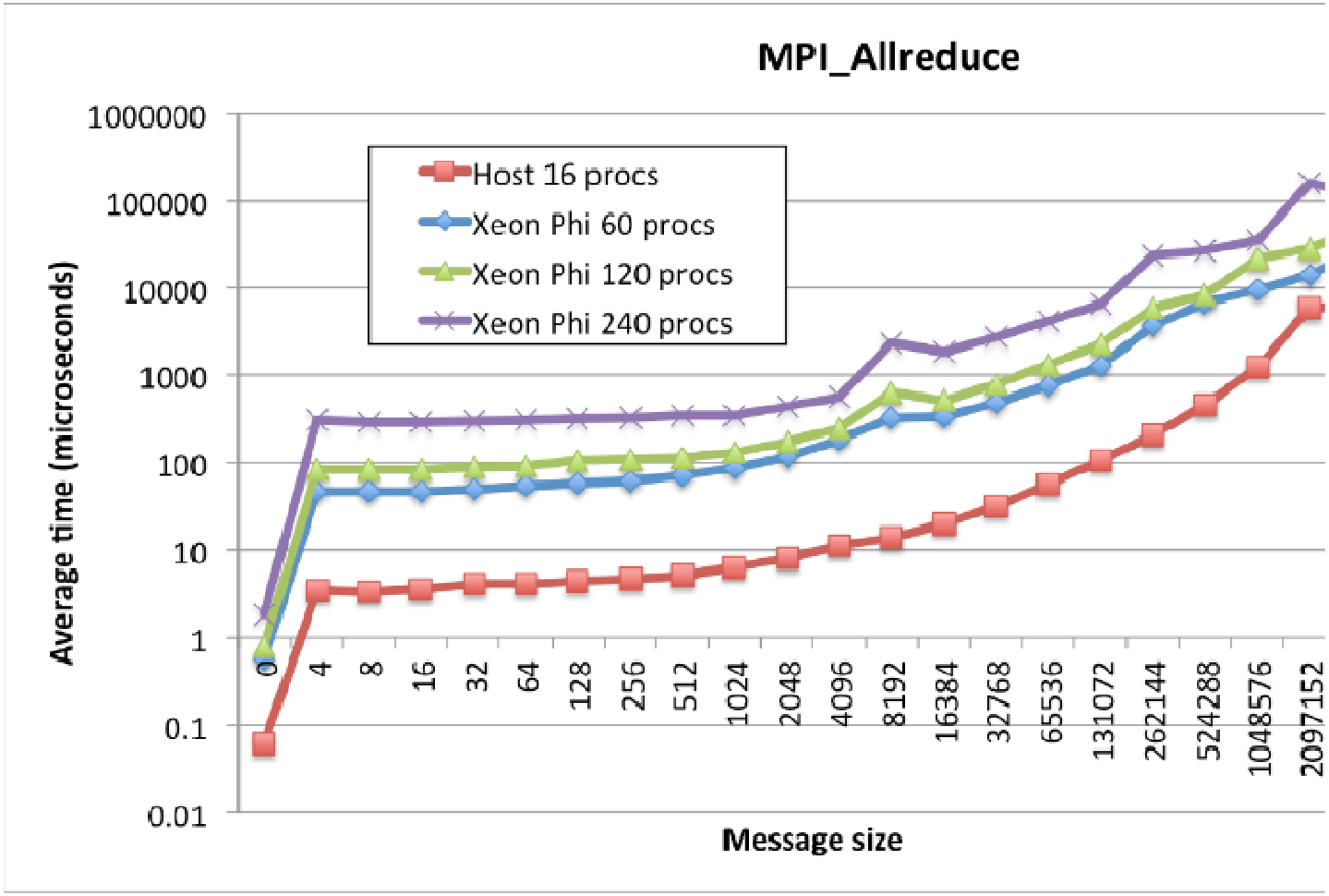
epcc

PingPong Latency



epcc

MPI_Allreduce



epcc

General approach

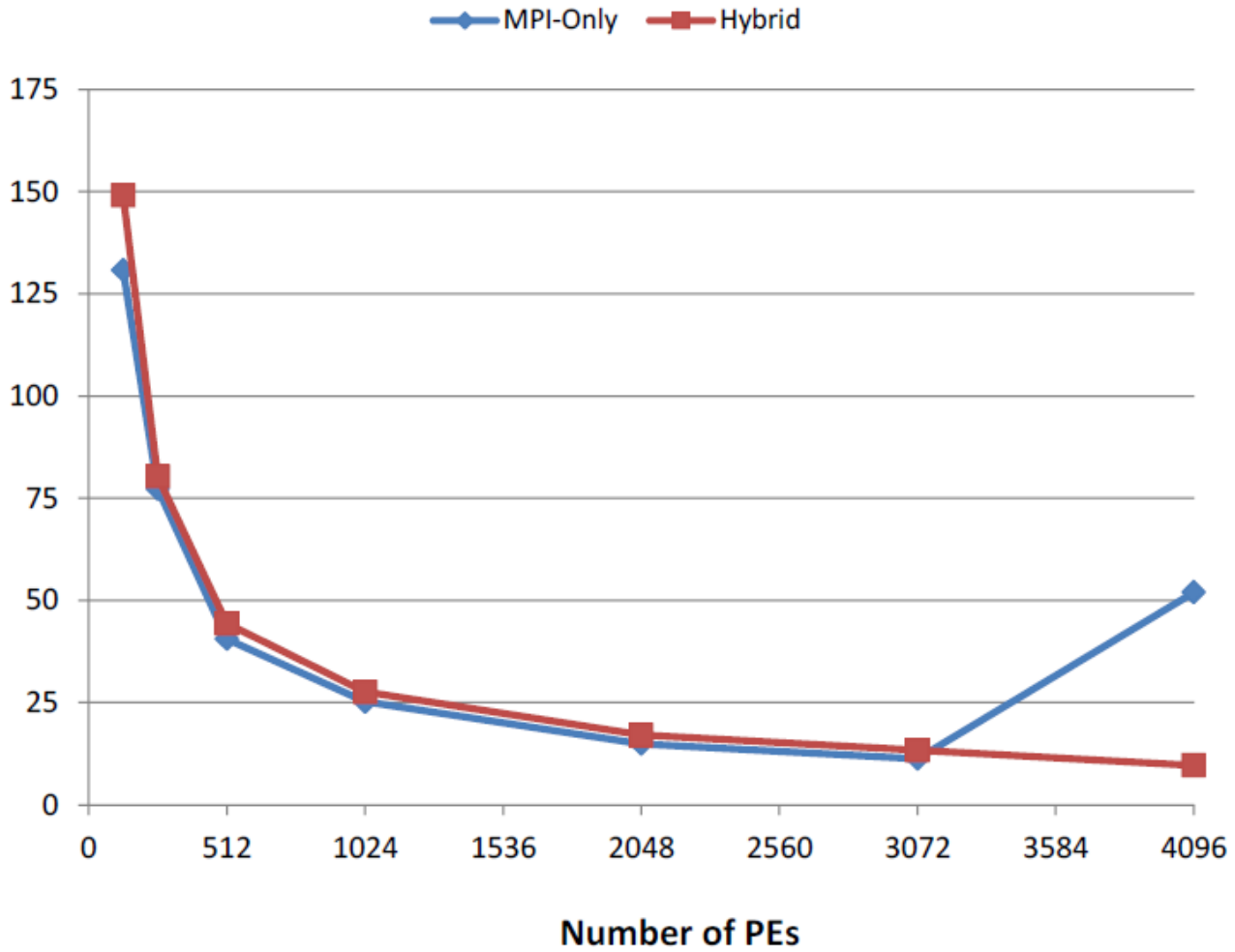
- Work on 3 codes
  - GS2, COSA, and CP2K
- All FORTRAN
- Investigate and optimise vectorisation of codes
  - Use profiler and compiler tools to evaluate vectorisation
  - Modify computationally expensive code to improve vectorisation
- Improve/implement hybrid parallelisation
  - Can help for both standard and phi systems
  - Reduce memory footprint
- Reduce serial code
  - I/O etc….

GS2

- Flux-tube gyrokinetic code
  - Initial value code
  - Solves the gyrokinetic equations for perturbed distribution functions together with Maxwell's equations for the turbulent electric and magnetic fields
  - Linear (fully implicit) and Non-linear (dealiased pseudo-spectral) collisional and field terms
  - 5D space – 3 spatial, 2 velocity
  - Different species of charged particles
- Advancement of time in Fourier space
- Non-linear term calculated in position space
  - Requires FFTs
  - FFTs only in two spatial dimensions perpendicular to the magnetic field
- Heavily dominated by MPI time at scale
  - Especially with collisions

Initial hybrid version

- Performance of existing version



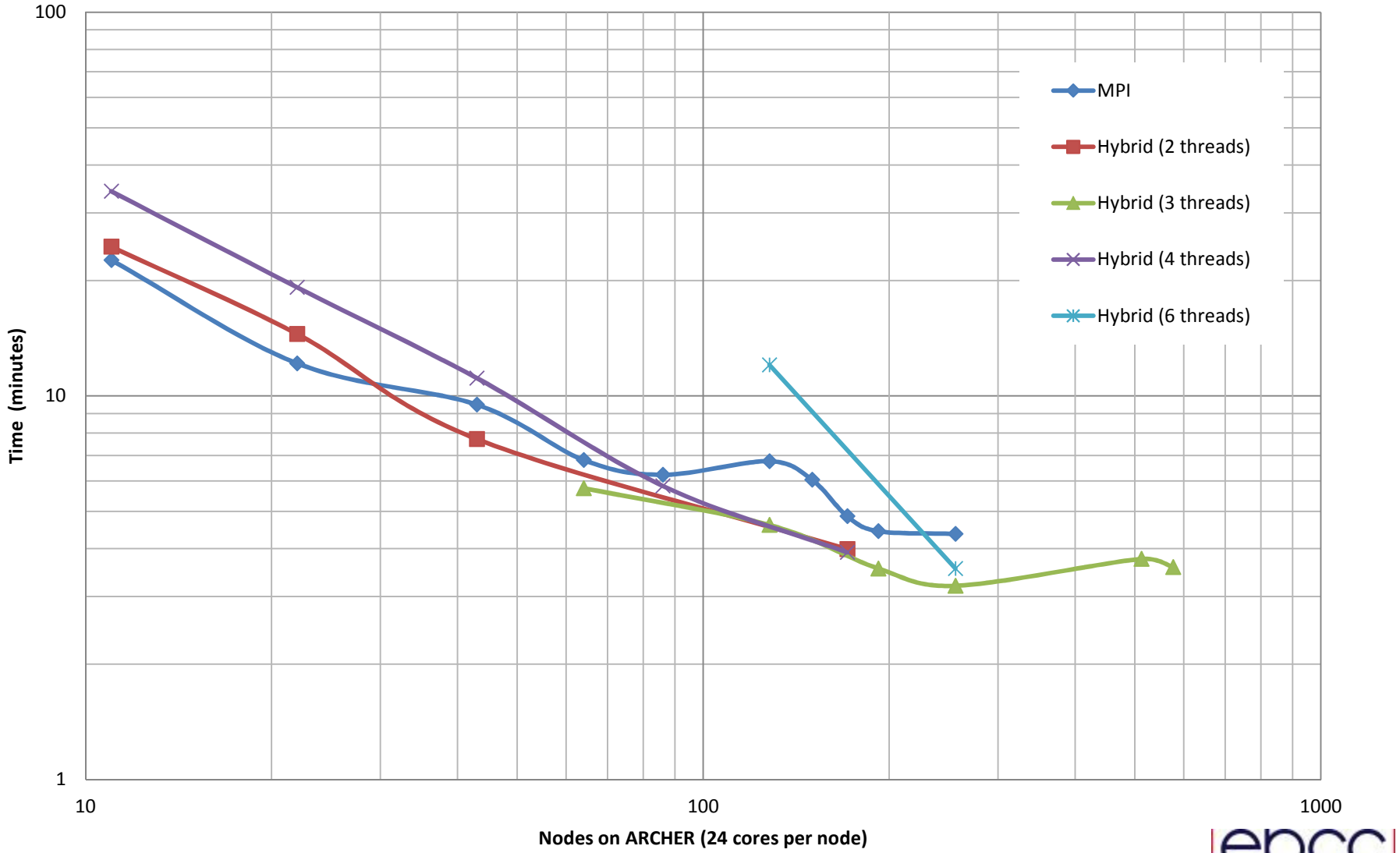
epcc

New hybrid implementation

- Still funnelled communication model
- OpenMP done at a higher level in the code
- Single parallel region per time step
  - Better can be achieved (single parallel region per run)
- Some code excluded but computationally expensive code all hybridised

MPI processes	OpenMP threads	Execution time (seconds)
192	1	16.54
96	2	18.34
64	3	16.46
48	4	30.86
32	6	28.3

Total runtime



epcc

Port to Xeon Phi

- Pure MPI code performance:
  - ARCHER (2x12 core Xeon E5-2697, 16 MPI processes): 3.08 minutes
  - Host (2x8 core Xeon E5-2650, 16 MPI processes): 4.64 minutes
  - 1 Phi (176 MPI processes): 7.34 minutes
  - 1 Phi (235 MPI processes): 6.77 minutes
  - 2 Phis (352 MPI processes): 47.71 minutes
- Hybrid code performance
  - 1 Phi (80 MPI processes, 3 threads each): 7.95 minutes
  - 1 Phi (120 MPI processes, 2 threads each): 7.07 minutes

Vector (de)optimisations

- Vector optimising work unsuccessful
  - A number of poorly vectorising targets identified
  - Code restructuring and directives not able to improve performance

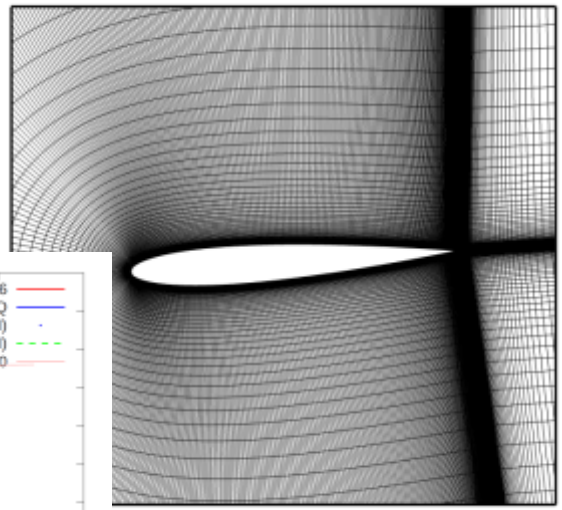
Function	Compiler flags	Compiler directives	Execution time
	-O2 (original)	-O2 (original)	16.46
invert_rhs_1	-align array64byte	attributes align, vector aligned	16.73
get_source_term	-align array64byte	attributes align, vector aligned	16.99
get_source_term	-align array64byte	attributes align, vector aligned (only for variables gexp1, gexp2 and gexp3)	16.72

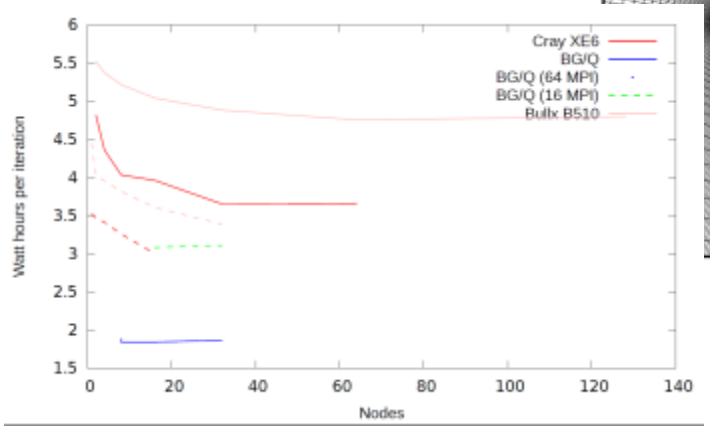
Complex number optimisation

- Much of GS2 uses FORTRAN Complex numbers
  - However, often imaginary and real parts are treated separately
  - Can affect vectorisation performance
- Work underway to replace with separate arrays
  - Initial performance numbers demonstrate performance improvement on Xeon Phi
  - 2-3% for a single routine when using separate arrays

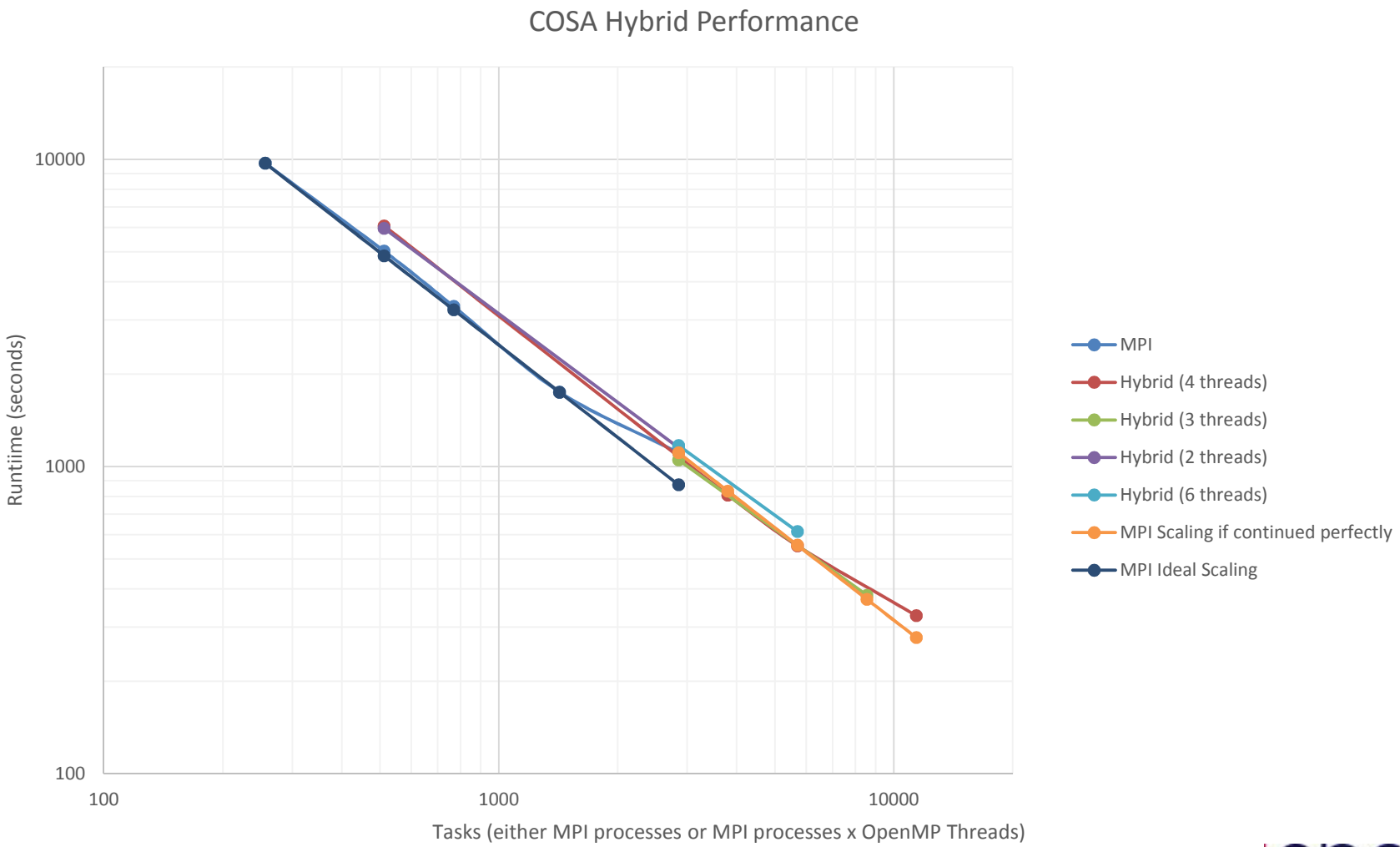
COSA

- Fluid dynamics code
  - Harmonic balance (frequency domain approach)
  - Unsteady navier-stokes solver
  - Optimise performance of turbo-machinery like problems
  - Multi-grid, multi-level, multi-block code
  - Parallelised with MPI and with MPI+OpenMP





epcc


COSA Hybrid Performance
Runtiime (seconds)
10000
1000
100
100
1000
10000
Tasks (either MPI processes or MPI processes x OpenMP Threads)
MPI
Hybrid (4 threads)
Hybrid (3 threads)
Hybrid (2 threads)
Hybrid (6 threads)
MPI Scaling if continued perfectly
MPI Ideal Scaling

Xeon Phi Performance

Configuration	Number of hardware elements	Occupancy	Runtime (s)
8 MPI processes	1/2	8/16	2105.71
16 MPI processes	2/2	16/16	1272.54
64 MPI processes	1/2	64/240	3874.45
64 MPI processes 3 OpenMP threads	1/2	192/240	2963.58
118 MPI processes 4 OpenMP threads	2/2	472/480	2118.05
128 MPI processes 3 OpenMP threads	2/2	384/480	1759.30

- Hardware:
  - 2 x Xeon Sandy Bridge 8-core E5-2650 2.00GHz
  - 2 x Xeon Phi 5110P 60-core 1.05GHz
- Test case
  - 256 blocks
  - Maximum 7 OpenMP threads

Serial optimisations

- Manual removal of floating point loop invariants divisions

```
do ipde = 1,4
      fac1 = fact * vol(i,j)/dt
end do


recip = 1.0d / dt
do ipde = 1,4
       fact1 = fact * vol(i,j) * recip
end do
```

- Provides ~15% speedup so far on Xeon Phi
  - No real benefit noticed on host
  - Changes the results

I/O

- Identified that reading input is now significant overhead for this code
  - Output is done using MPI-I/O, reading is done serially
  - File locking overhead grows with process count
  - Large cases ~GB input files
- Parallelised reading data
  - Reduce file locking and serial parts of the code
- One or two orders of magnitude improvement in performance at large process counts
  - 1 minute down to 5 seconds

Future work

Configuration	Number of hardware elements	Occupancy	Runtime (s)
8 MPI processes	1/2	8/16	2105.71
16 MPI processes	2/2	16/16	1272.54
128 MPI processes	1/2	128/240	1903.51
64 MPI processes 3 OpenMP threads	1/2	192/240	2214.56
128 MPI processes 3 OpenMP threads	2/2	384/480	1503.45

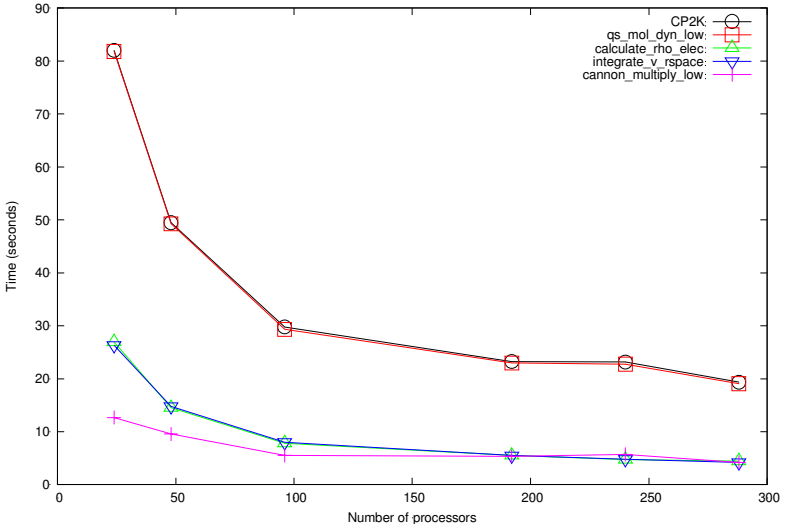
- Further serial optimisation
  - Cache blocking
- 3D version of the code now developed
  - Porting optimised and hybrid version to this

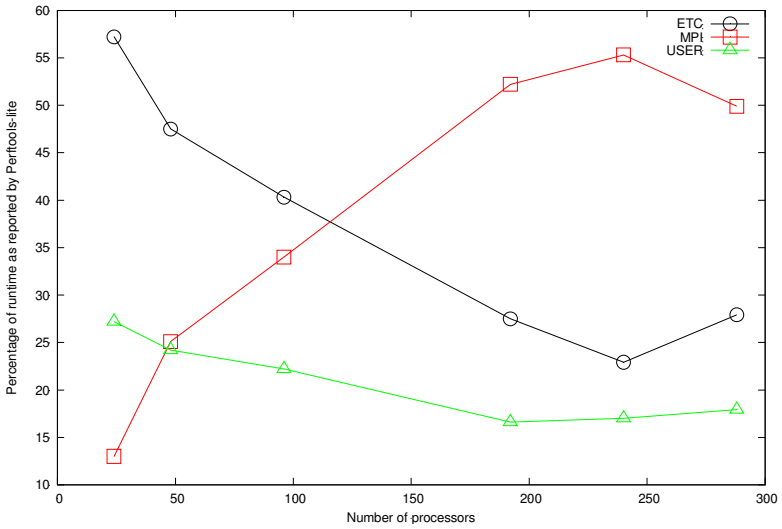
epcc

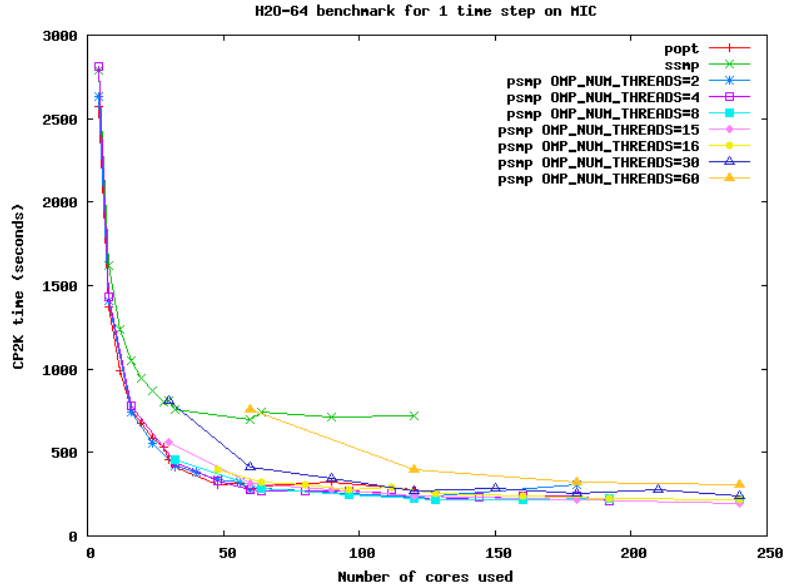
CP2K

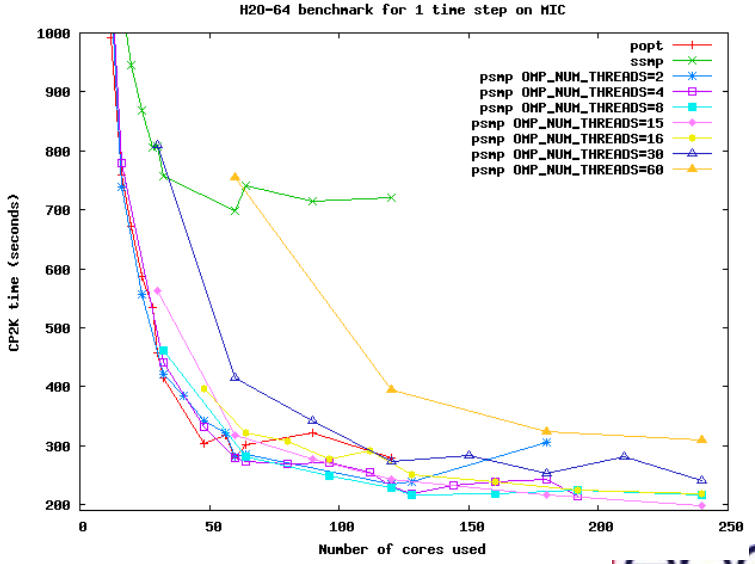
- Atomistic and molecular simulations of solid state, liquid, molecular, and biological system
- MPI and hybrid parallelisations implemented
- Heavily uses internal and external libraries for core computations
- Other sites working on Xeon Phi
  - Offload functionality
  - Investigating compiler optimisations
- EPCC has previously worked on a native mode Xeon Phi port
  - Performance not great, 50% compared to CPU version (16 cores), low memory requirement restricts accuracy
  - This work identified a number of vectorisation targets

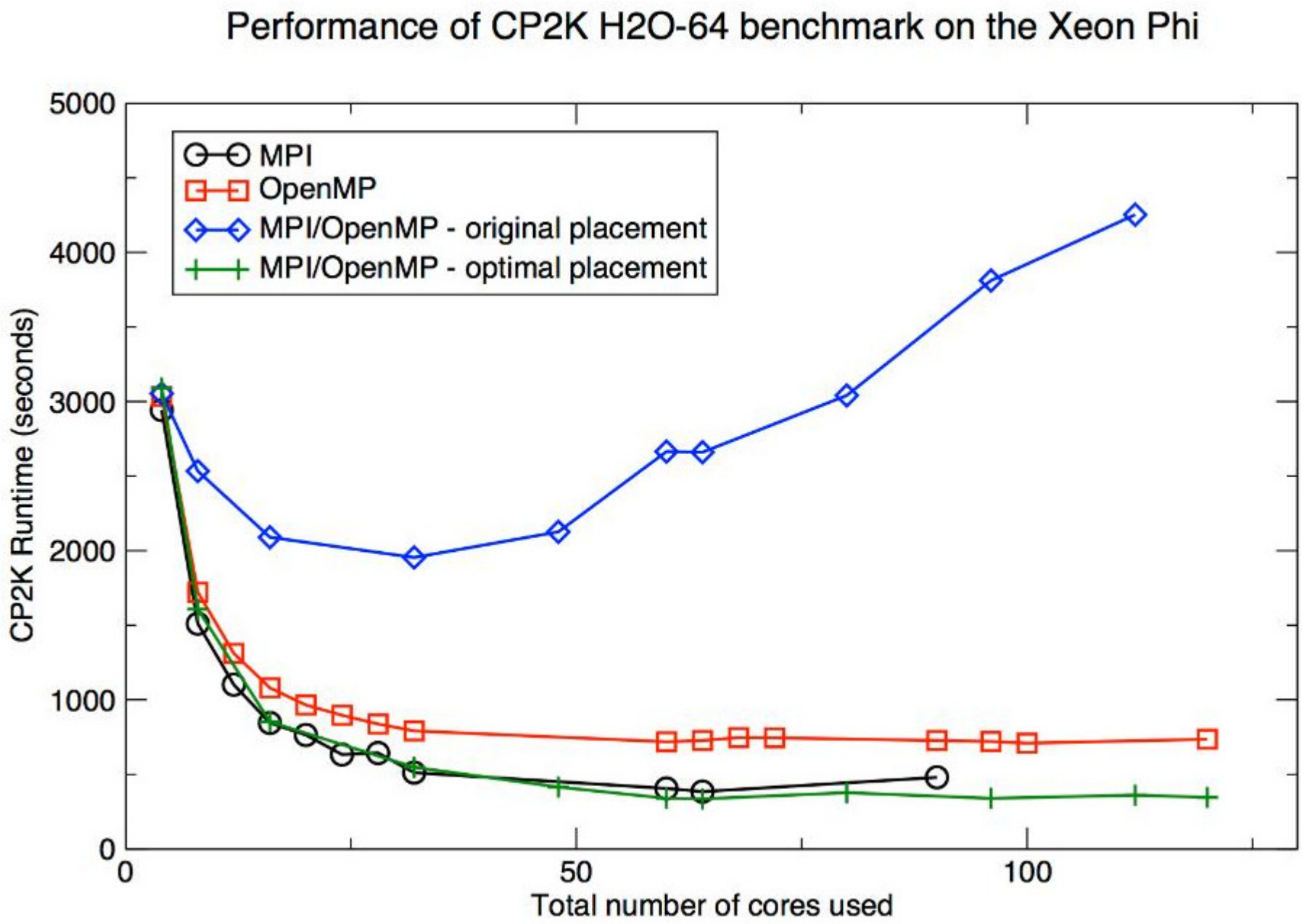
CP2K Performance










Performance of CP2K H2O-64 benchmark on the Xeon Phi
MPI
OpenMP
MPI/OpenMP - original placement
MPI/OpenMP - optimal placement
CP2K Runtime (seconds)
Total number of cores used
0
1000
2000
3000
4000
5000
0
50
100

CP2K Test Suite

- Regression test suite and continuous integration testing important to ensure CP2K maintains correctness
  - Important as it informs which compilers and libraries users can build the application with
- Ported test suite and framework to enable use of Intel compilers and MKL libraries
  - Build both on host and Xeon Phi
- Essential task before code modification could be undertaken
  - Picked up a number of bugs with the code and Intel compilers/libraries

Vector (de)optimisations

- Vector optimising work unsuccessful
  - CP2K uses auto-tuning library routines for core kernels
  - Vectorising these routines struggled due to code structure

Code version	Time (seconds)
Original code	2.423632
Adding !DIR$ IVDEP to loop over ig	2.472624
Attempt 1: Array syntax	2.438629
Attempt 1: Array syntax + !DIR$ IVDEP on loop over ig	*2.437631
Attempt 1: Array syntax + !DIR$ VECTOR ALWAYS on loop over ig	2.430630
Attempt 1: Array syntax + !DIR$ SIMD on loop over ig	2.463625
Attempt 1: Array syntax + !$OMP SIMD private(i,s) on loop over ig	2.484623
Attempt 1: Array syntax + align map and pol_x	2.479622
Attempt 1: Array syntax + align map and pol_x + !$OMP SIMD on loop over ig	2.524676
Attempt 2: use ivec(ig) array and array syntax	2.477623
Attempt 2: use ivec(ig) array and array syntax + !DIR$ IVDEP on loop over ig	2.473624
Attempt 2: use ivec(ig) array and array syntax + !DIR$ SIMD on loop over ig	2.580608
Attempt 2: use ivec(ig) array and array syntax + !$OMP SIMD private(i,s) on loop over ig	2.620602
Attempt 2: use ivec(ig) array and array syntax + localmap 1d array used to compute l	2.475624
Attempt 3: replace the ig loop with loops over countblocks and starti(ib) to stopi(ib)for each block of contiguous iterations	2.626000
Attempt 3: replace the ig loop with loops over countblocks and starti(ib) to stopi(ib)for each block of contiguous iterations + !DIR$ IVDEP on loop over i	2.625601
Attempt 3: replace the ig loop with loops over countblocks and starti(ib) to stopi(ib)for each block of contiguous iterations + !DIR$ VECTOR ALWAYS on loop over i	2.627600
Attempt 3: replace the ig loop with loops over countblocks and starti(ib) to stopi(ib)for each block of contiguous iterations + !DIR$ SIMD on loop over i	2.582607
Attempt 3: replace the ig loop with loops over countblocks and starti(ib) to stopi(ib)for each block of contiguous iterations + !$OMP SIMD private(s) on loop over i	2.634599
Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc	2.769579
Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ IVDEP on loop over i	2.760580
Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ VECTOR ALWAYS on loop over i	2.755581
Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ SIMD on loop over i	2.754581
Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !$OMP SIMD private(s) on loop over i	2.757580

Summary

- Working on large FORTRAN MPI (or hybrid) simulation codes
  - Already heavily optimised, no real low hanging fruit
- Single code base work highly favoured
  - Large scale codes won't maintain mixed source versions
  - Favours native mode parallelisation
- Hybrid parallelisations will help elsewhere
  - Obvious target for many MPI programs
- Intel compilers v15 has impacted performance across the board for our codes
  - Slower with v15 vs v14
- MPI across Xeon Phi's can heavily impact performance
  - Global comms dominated codes don't currently scale
  - Local comms codes can scale well

Acknowledgements

- Work supported by Intel Parallel Computing Centre program
- Some work also supported by PRACE and EPSRC Plasma HEC grant