# Understanding Hybrid MPI + OpenMP Performance

Holly Judge and Mark Bull

EPCC, University of Edinburgh, UK

# Hybrid MPI + OpenMP

- Hybrid MPI + OpenMP applications are becoming increasingly common on HPC systems
- Can both reduce memory usage and/or improve scalability
- Semantics are straightforward
  - especially if only one thread ever makes MPI calls
- But…we need to choose how many OpenMP threads to run per MPI process
  - performance is a complex trade-off between multiple sources of overhead

# Potential advantages of MPI + OpenMP

- Reducing memory usage
  - fewer copies of replicated data structures
  - less data in halo regions
- Exploiting additional levels of parallelism
  - easier do this by adding OpenMP than trying to it in pure MPI
- Reducing computation
  - some MPI codes replicate parts of the computation
- Reducing load imbalance
  - easier and cheaper to balance load between threads than between processes
- Reducing communication costs
  - don't communicate unused data
  - fewer ranks in collectives
  - fewer (but maybe larger) point-to-point messages

# Performance pitfalls

- Most hybrid applications are written (for simplicity) in master-only style – all MPI calls are outside of OpenMP parallel regions
  - OpenMP threads are necessarily idle during MPI communications
  - cache misses occur if master thread communicates data written/read by other threads
- Implicit point-to-point synchronisation via messages may be replaced by (more expensive) barriers.
  - loose thread-to-thread synchronisation is hard to do in OpenMP
- In a pure MPI code, the intra-node messages will often be naturally overlapped with inter-node messages
  - harder to overlap inter-thread communication with inter-node messages
- OpenMP can suffer from false sharing and NUMA effects
  - MPI naturally avoids these

# Typical trends

If we keep the total number of cores fixed and increase the number of threads per MPI process:

- Total time spent in MPI reduces 😀
- Load imbalance between processes reduces 😀
- Amount of computation may reduce 😀

- Thread idle time increases ☹
- Thread synchronization time increases (mostly barriers) ☹
- Load imbalance between threads may increase ☹
- Use of memory system may get less efficient (false sharing, NUMA effects) ☹

# Experimental setup

**Hardware**

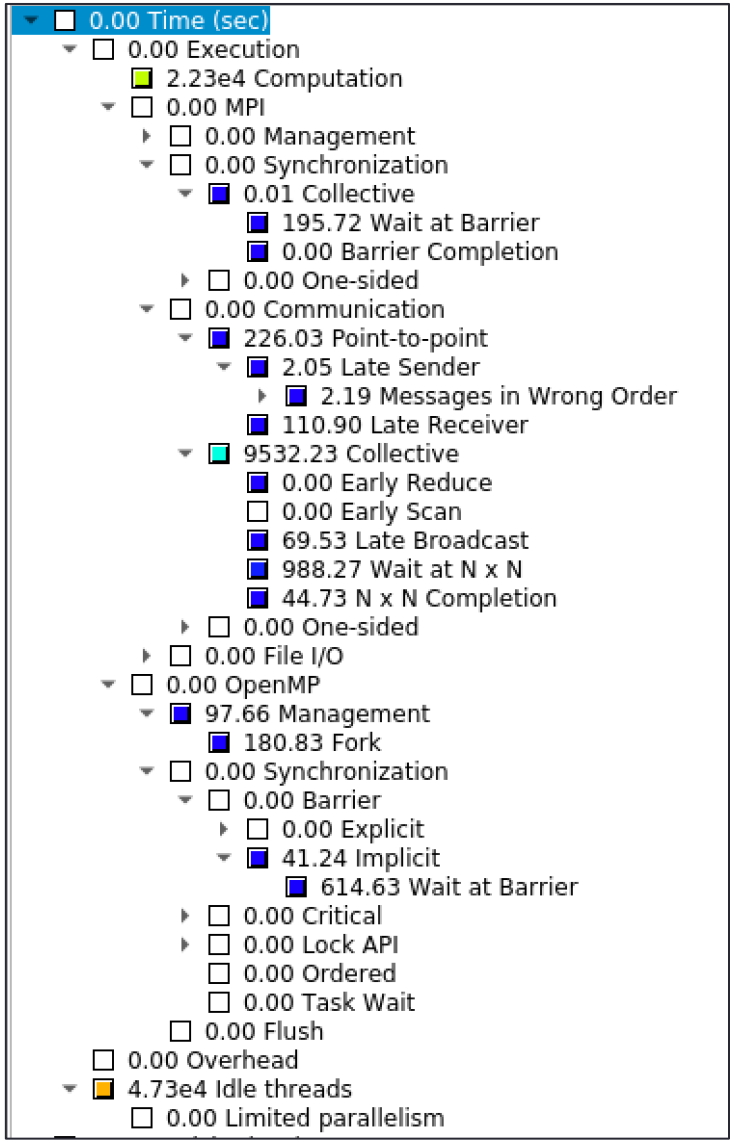
- HPE Cray EX system
- 2 x AMD EPYC 7742, 2.25 GHz, 64-core chips per node (128 cores/node)
- 4 NUMA regions per socket (1 NUMA region per 16 cores)
- 16MB L3 cache per 4 cores
- HPE Slingshot network


**Software**

- MPI: HPE Cray MPICH2
- OpenMP: GNU compilers
- Profiling tool: Scalasca

# Scalasca

Scalasca tracing experiments give a detailed hierarchical breakdown of MPI and OpenMP overheads

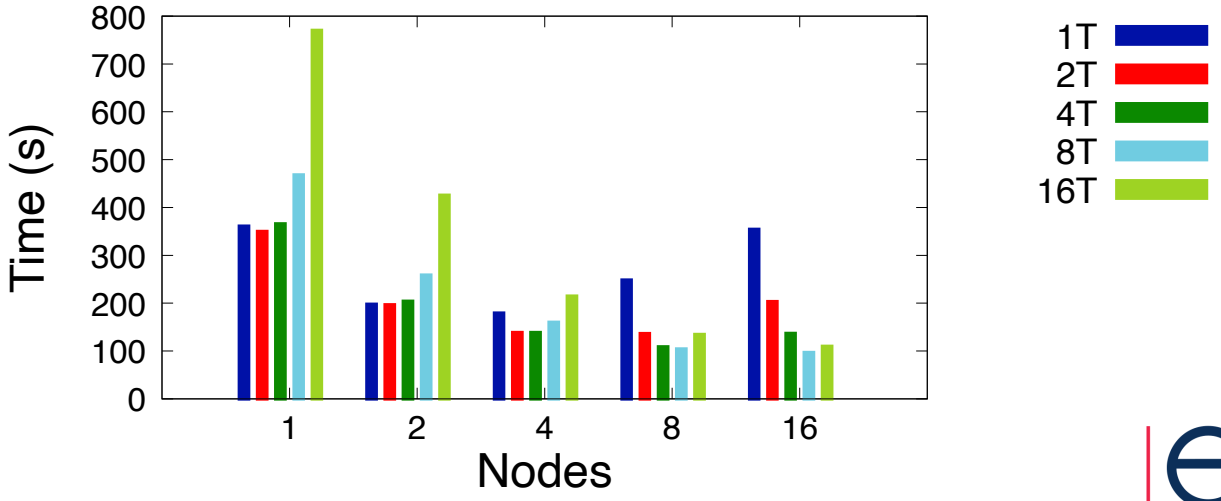



epcc

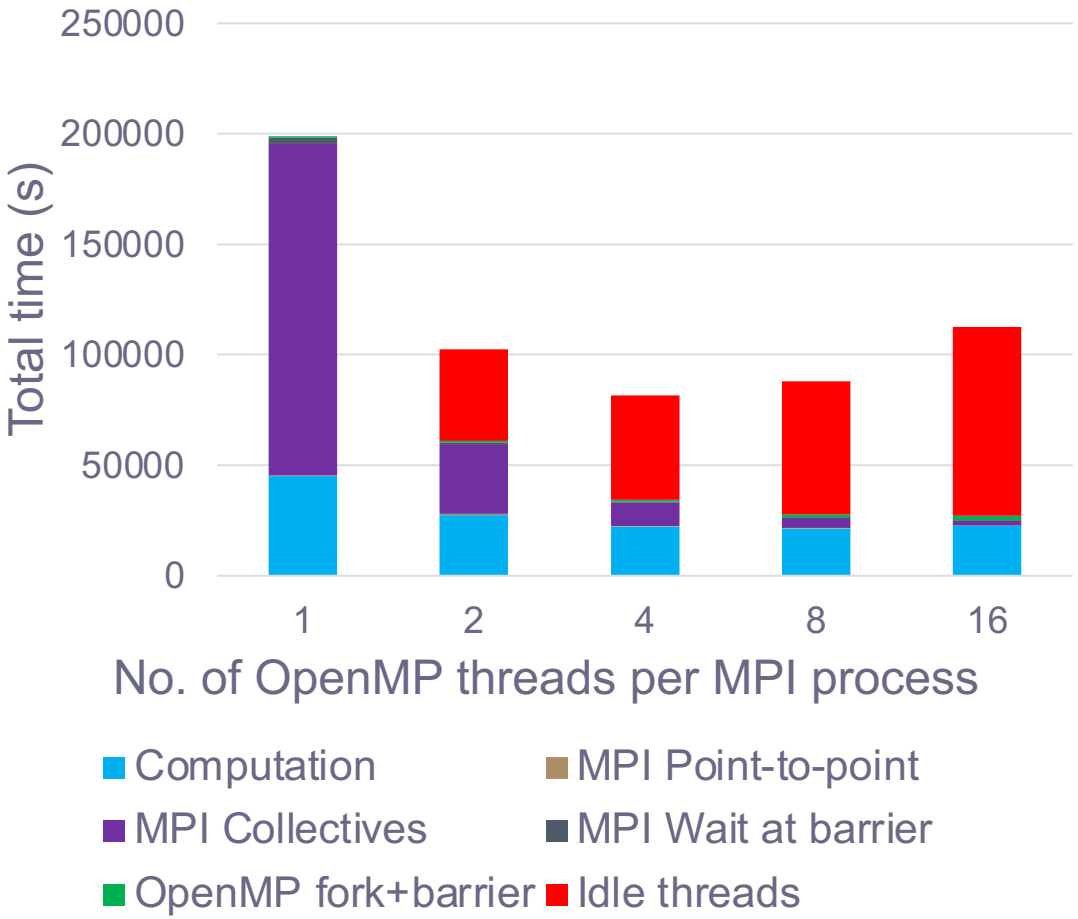
THE UNIVERSITY OF EDINBURGH

# Case study - CASTEP

- CASTEP - density functional theory software package for electronic structure calculations using plane waves
  - Version 20.11 - GCC version 10.2, Intel MKL 19, Cray-mpich 8.1.4, Cray-fftw 3.3.8.11
  - Al3x3 benchmark



# CASTEP – profiling




8 nodes (= 1024 cores)

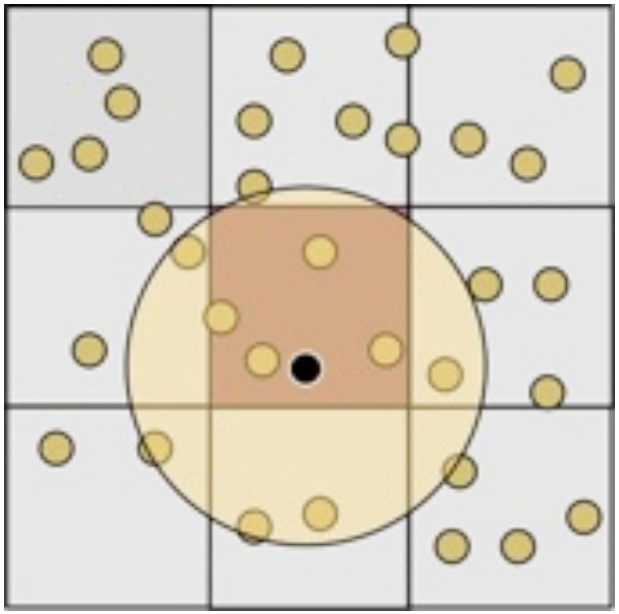
Threads give big savings in compute and comms times (mostly in FFT library)

But….

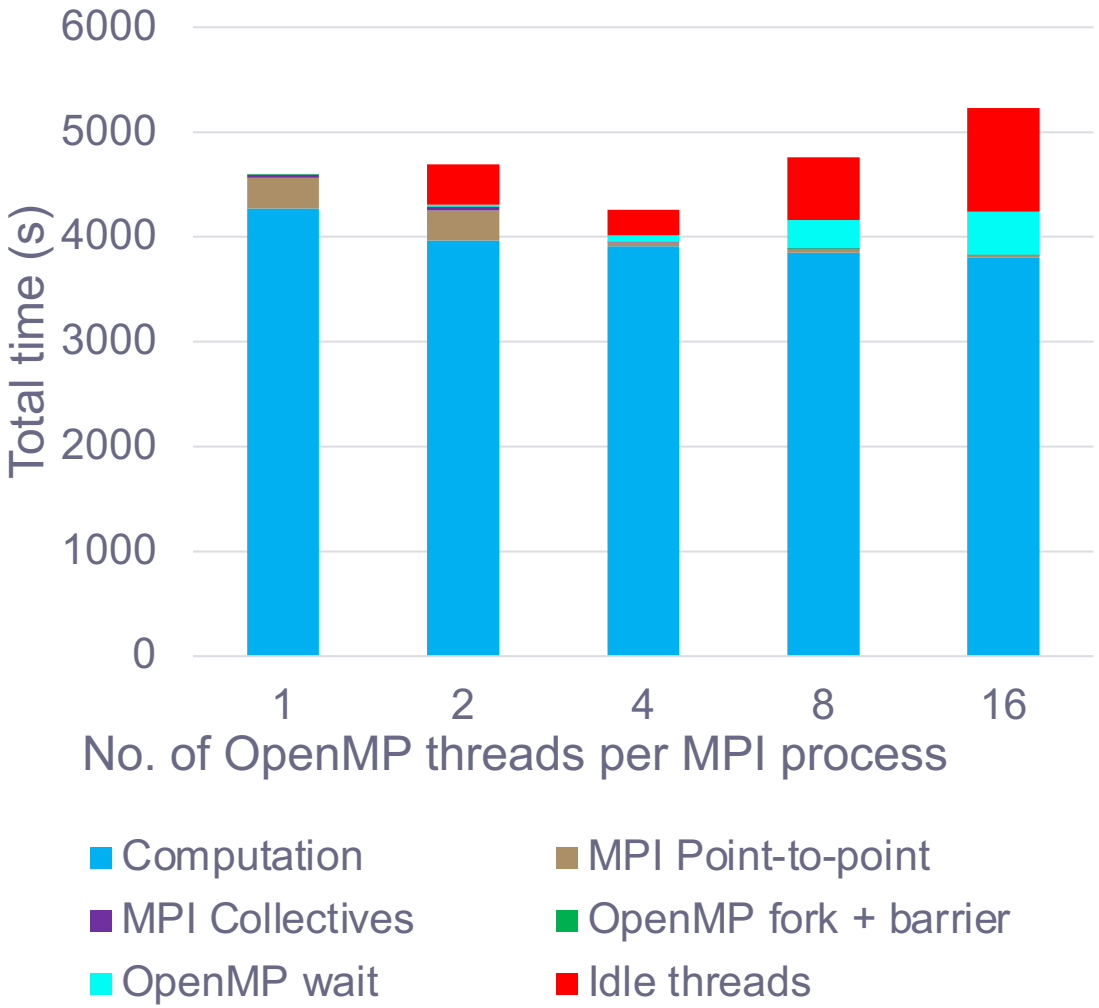
FFT library comms are single threaded, so idle time outweighs savings above 8 threads per process

# Case study - CoMD

- CoMD Proxy Application – classical molecular dynamics using link cells
  - GCC version 8.3, Cray-mpich 8.1.4
  - $128^3$ atoms



# CoMD – profiling




8 nodes (= 1024 cores)

Threads give some savings in compute time

Load imbalance across processes is reduced

But..

Load imbalance across threads appears

Idle time becomes significant

epcc

THE UNIVERSITY OF EDINBURGH

*Thanks for watching!*