CP2K PARALLELISATION AND OPTIMISATION

Iain Bethune (ibethune@epcc.ed.ac.uk)

Overview

- Overview of Parallel Programming models
  - Shared Memory
  - Distributed Memory
- CP2K Algorithms and Data Structures
- Parallel Performance
- CP2K Timing Report

epcc

THE UNIVERSITY OF EDINBURGH

Parallel Programming Models

- Why do we need parallelism at all?
- Parallel programming is (even) harder than sequential programming
- Single processors are reaching limitations
  - Clock rate stalled at ~2.5 GHz (due to heat)
  - Full benefits of vectorisation (SIMD) can be hard to realise
  - Chip vendors focused on low-power (for mobile devices)

epcc

Parallel Programming Models

- But we need *more* speed!
  - Solve problems faster (strong scaling)
  - Solve bigger problems in same time (weak scaling)
  - Tackle new science that emerges at long runtimes / large system size
  - Enables more accurate force models (HFX, MP2, RPA …)
- Need strategies to split up our computation between different processors
- Ideally our program should run P times faster on P processors - but not in practice!
  - Some parts may be inherently serial (Amdahl's Law)
  - Parallelisation *will* introduce overheads e.g. communication, load imbalance, synchronisation…

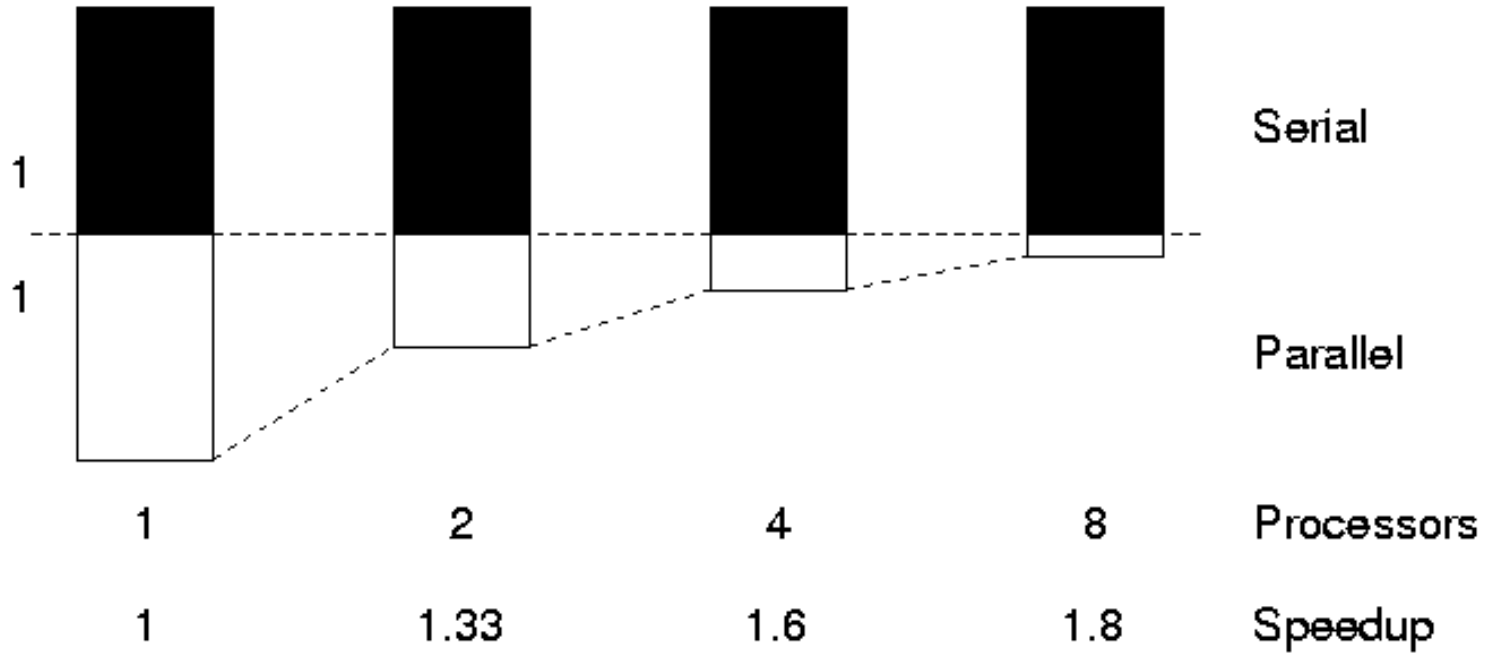
epcc

THE UNIVERSITY OF EDINBURGH

Parallel Programming Models

"The performance improvement to be gained by parallelisation is limited by the proportion of the code which is serial"

Gene Amdahl, 1967



epcc

THE UNIVERSITY OF EDINBURGH

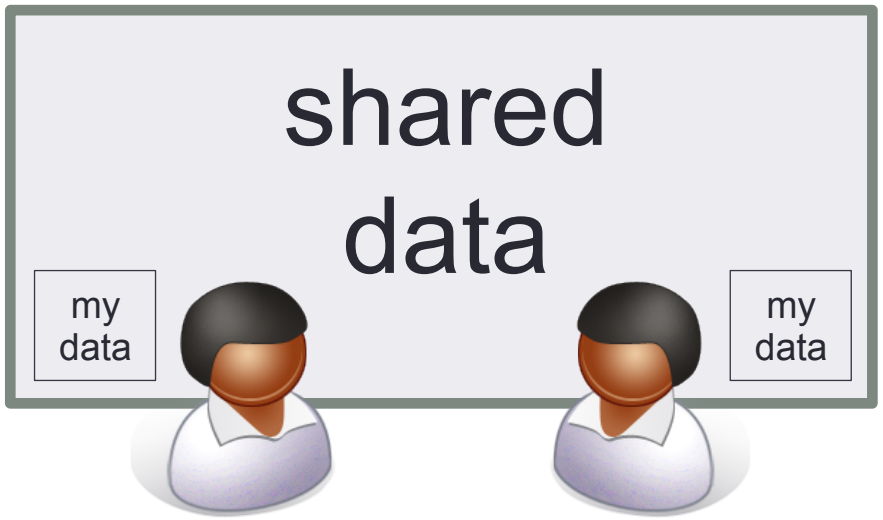
Parallel Programming Models

- Almost all modern CPUs are multi-core
  - 2,4,6… CPU cores, sharing access to a common memory
- This is Shared Memory Parallelism
  - Several processors executing the same program
  - Sharing the same address space i.e. the same variables
  - Each processor runs a single ‘thread’
  - Threads communicate by reading/writing to shared data
- Example programming models include:
  - OpenMP, POSIX threads (pthreads)

epcc

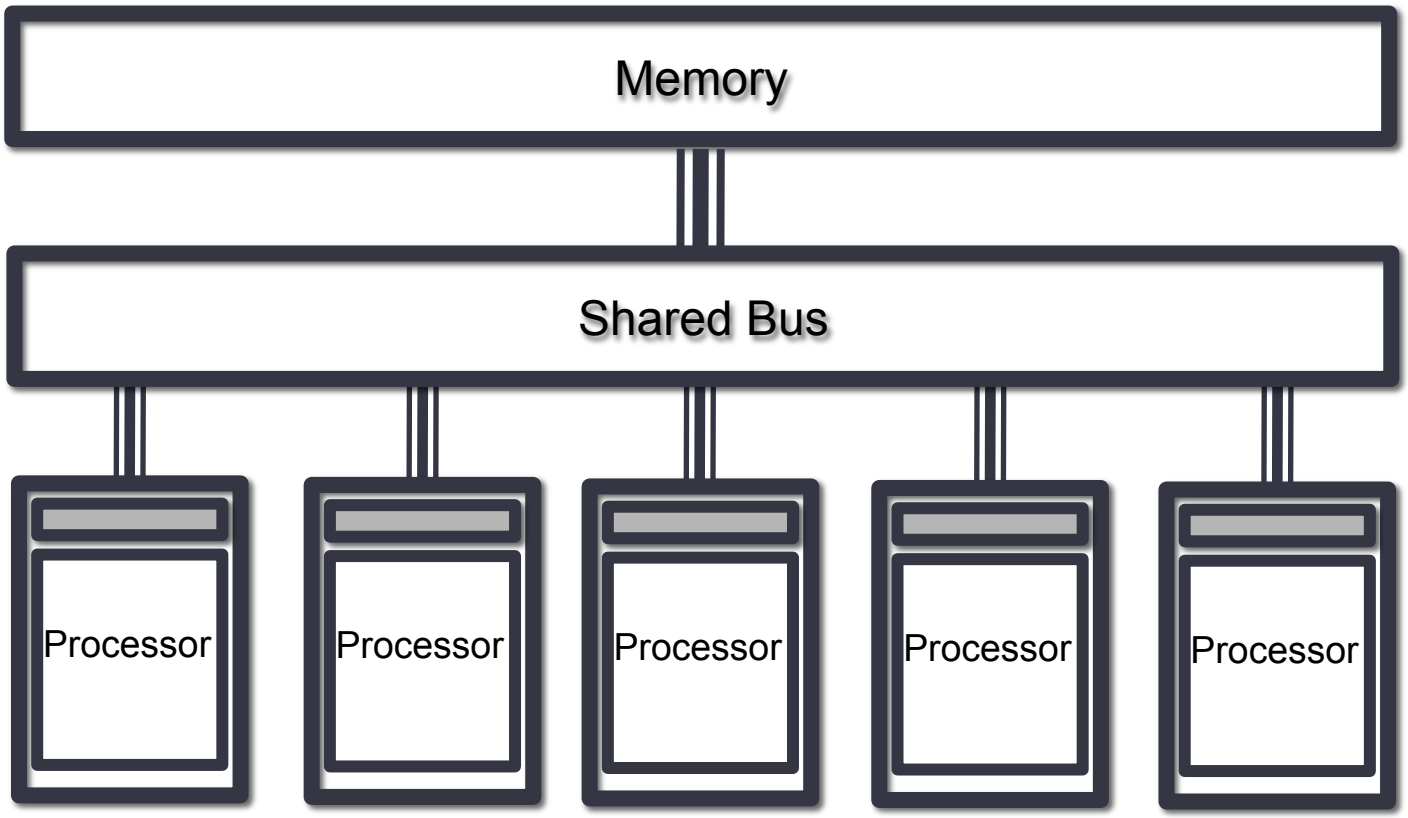
Analogy

- One very large whiteboard in a two-person office
  - the shared memory
- Two people working on the same problem
  - the threads running on different cores attached to the memory
- How do they collaborate?
  - working together
  - but not interfering
- Also need *private* data



Hardware

- Needs support of a shared-memory architecture



Parallel Programming Models

- Most supercomputers are built from 1000s of nodes
  - Each node consists of some CPUs and memory
  - Connected together via a network

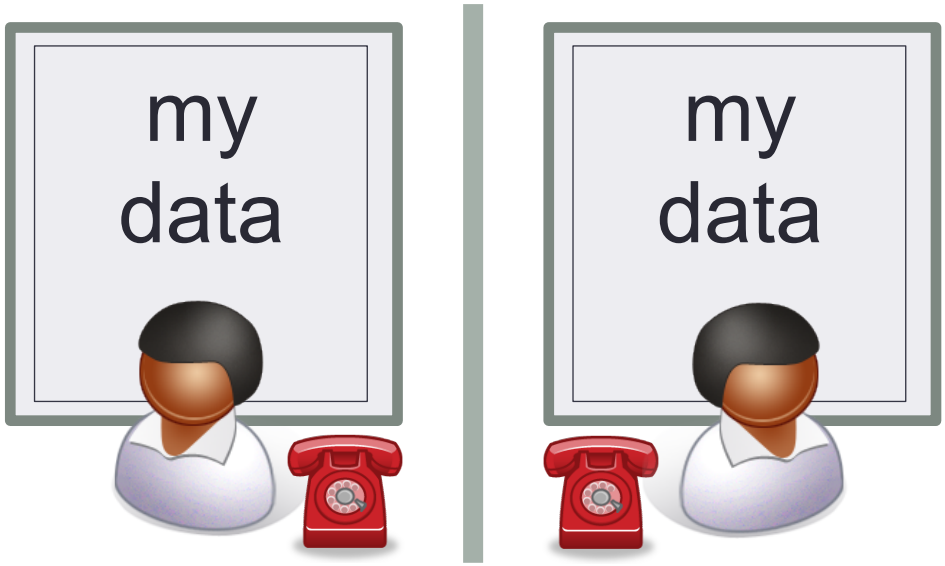
- This is Distributed Memory Parallelism
  - Several processors executing (usually) the same program
  - Each processor has it's own address space
  - Each processor runs a single 'process'
  - Threads communicate by passing messages
- Example programming models include:
  - MPI, SHMEM

epcc

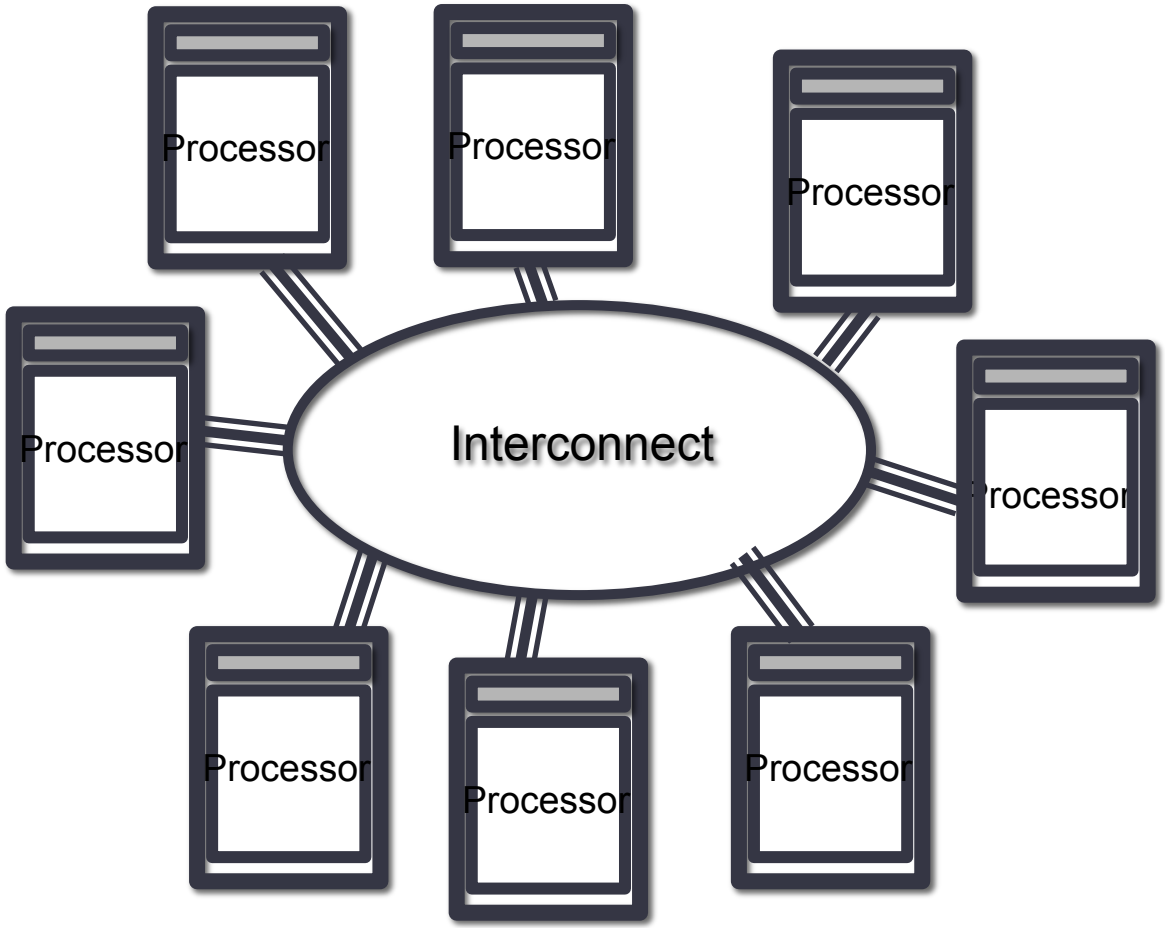
THE UNIVERSITY OF EDINBURGH

Analogy

- Two whiteboards in different single-person offices
  - the distributed memory
- Two people working on the same problem
  - the processes on different nodes attached to the interconnect
- How do they collaborate?
  - to work on single problem
- Explicit communication
  - e.g. by telephone
  - no shared data



Hardware



- Natural map to distributed-memory
  - one process per processor-core
  - messages go over the interconnect, between nodes/OS's

Parallel Programming Models

- Support both OpenMP *or* MPI (`ssmp` and `popt`)
  - Use OpenMP for desktop PCs with multi-cores *or*
  - MPI for clusters and supercomputers
  - Maybe also support for Accelerators (GPUs)

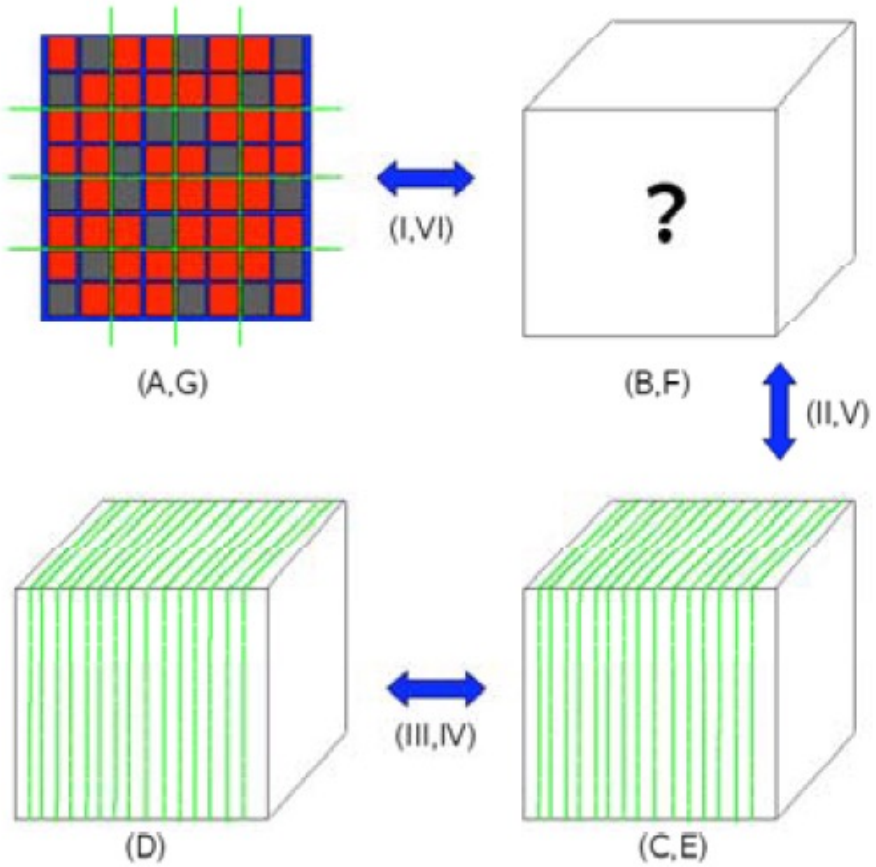
- May also combine MPI *and* OpenMP (`psmp`)
  - Called hybrid or mixed-mode parallelism
  - Use shared memory within a node (with several processors)
  - Use message passing between nodes
  - Usually only useful for scaling to 10,000s of cores!

epcc

CP2K Algorithms and Data Structures

- (A,G) – distributed matrices
- (B,F) – realspace multigrids
- (C,E) – realspace data on planewave multigrids
- (D) – planewave grids

- (I,VI) – integration/ collocation of gaussian products
- (II,V) – realspace-to-planewave transfer
- (III,IV) – FFTs (planewave transfer)

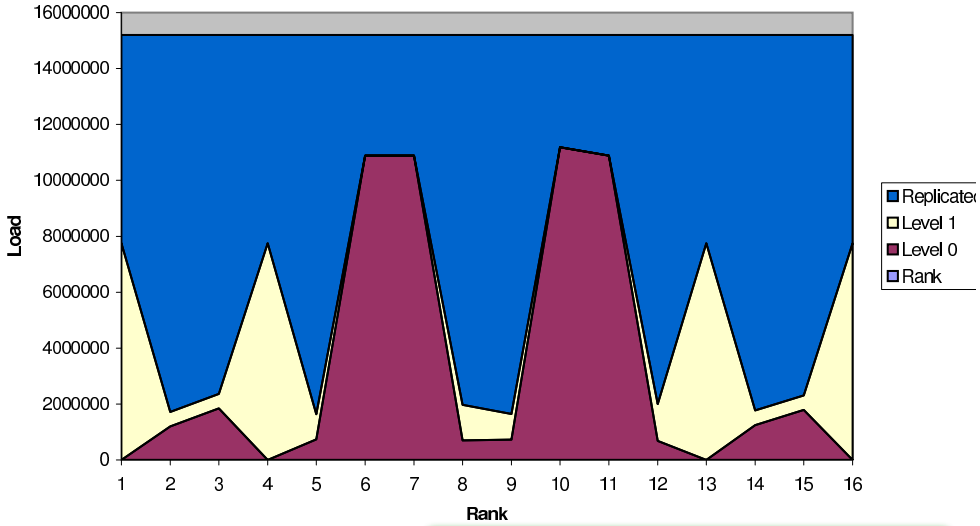


epcc

THE UNIVERSITY OF EDINBURGH

CP2K Algorithms and Data Structures

- Distributed realspace grids
  - Overcome memory bottleneck
  - Reduce communication costs
  - Parallel load balancing
    - On a single grid level
    - Re-ordering multiple grid levels
    - Finely balance with replicated tasks



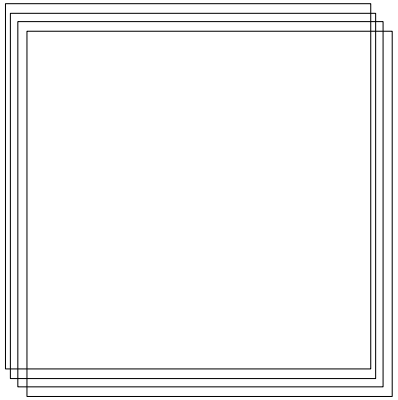
Level 1, fine grid, distributed

1	2	3
4	5	6
7	8	9

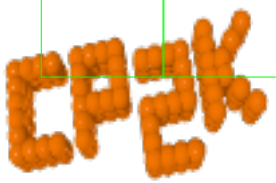
Level 2, medium grid, dist

5	6	8
3	1	7
9	4	2

Level 3, coarse grid, replicated



- `libgrid` for optimised collocate/integrate routines
- ~5-10% speedup typical



epcc

THE UNIVERSITY OF EDINBURGH

CP2K Algorithms and Data Structures

- Fast Fourier Transforms
  - 1D or 2D decomposition
  - FFTW3 and CuFFT library interface
  - Cache and re-use data
    - FFTW plans, cartesian communicators
- DBCSR
  - Distributed MM based on Cannon's Algorithm
  - Local multiplication recursive, cache oblivious

- `GLOBAL%FFTW_PLAN_TYPE MEASURE | PATIENT`
- Up to 5% Speedup possible

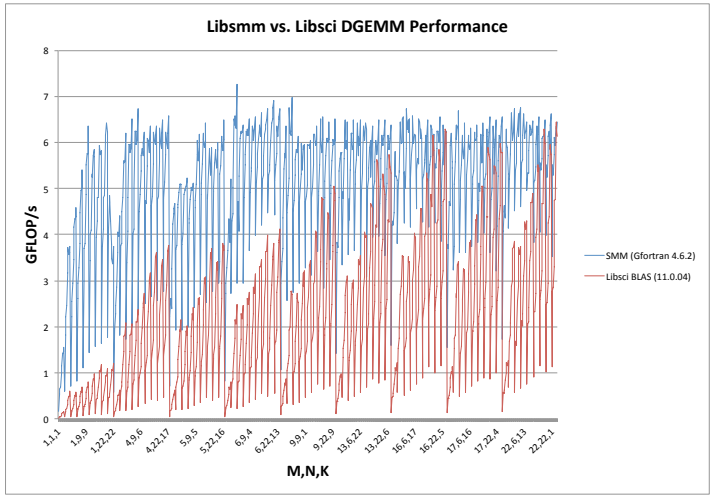


Figure 5: Comparing performance of SMM and Libsci BLAS for block sizes up to 22,22,22

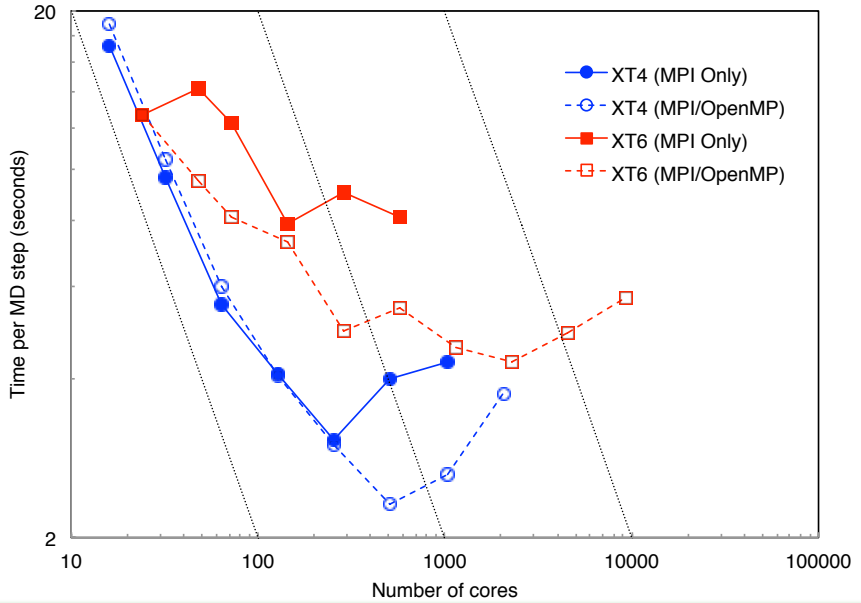
- `libsmm` for small block multiplications

epcc

THE UNIVERSITY OF EDINBURGH

CP2K Algorithms and Data Structures

- OpenMP
  - Now in all key areas of CP2K
  - FFT, DBCSR, Collocate/ Integrate, Buffer Packing
  - Incremental addition over time
    - Usually 2 or 4 threads per process
- Dense Linear Algebra
  - Matrix operations during SCF
  - GEMM - ScaLAPACK
  - SYEVD – ScaLAPACK / ELPA



- `-D__ELPA3` and link library to enable
- `GLOBAL %PREFERRED_DIAG_LIBRARY ELPA`
- Up to ~5x Speedup for large, metallic systems

cp2k

epcc

THE UNIVERSITY OF EDINBURGH

Measuring Parallel Performance

- Basic measure – wallclock time
  - How long did my calculation take from start to finish?
  - Depends on the number of processors!
  - Lower is better

- Application-specific measures
  - For MD, simulation time per wallclock time
  - e.g. ns / day
  - Using how many processors?
  - Higher is better

epcc

THE UNIVERSITY OF EDINBURGH

Measuring Parallel Performance

- Speed up
  - typically S(N,P) < P

$$S(N,P) = \frac{T(N,1)}{T(N,P)}$$

- Parallel efficiency
  - typically E(N,P) < 1

$$E(N,P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{PT(N,P)}$$

Where N is the size of the problem and P the number of processors

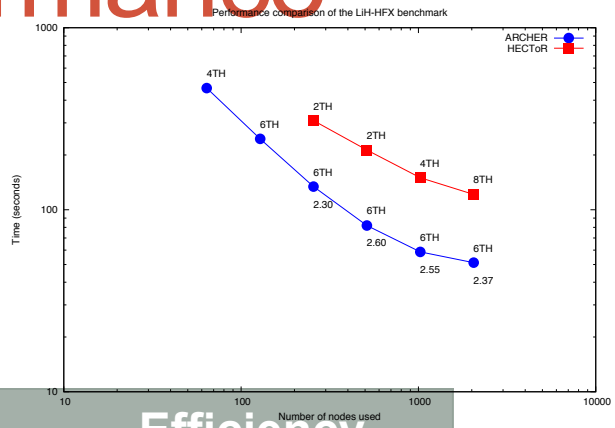
- Usually, consider E > 70% to be 'good' scaling
- If E < 1, then using more processors uses more compute time (AUs)
- Compromise between overall speed of calculation and efficient use of budget
  - Depends if you have one large or many smaller calculations

epcc

THE UNIVERSITY OF EDINBURGH

Measuring Parallel Performance

- CP2K 216-atom LiH-HFX benchmark



Nodes / cores	Wallclock time(s)	Cost (kAU)	Speedup (vs 64 nodes)	Efficiency (%)
64 / 1536	466	2.98	1	100%
128 / 3072	245	3.14	1.9	95%
256 / 6144	134	3.43	3.5	87%
512 / 12288	81.9	4.19	5.7	71%
1024 / 24576	58.7	6.01	7.9	50%
2048 / 49152	51.2	10.5	9.1	28%

epcc

THE UNIVERSITY OF EDINBURGH

Measuring Parallel Performance

- How to choose the number of CPUs for your simulation?
- Rely on *relevant* benchmark data
  - How many atoms, what level of theory?
  - What accuracy settings (basis set, numerical accuracy)
  - Similar type of HPC system?
- Benchmark data may (or may not) be available for your machine / problem size / method
- No substitute for testing with your own system

epcc

THE UNIVERSITY OF EDINBURGH

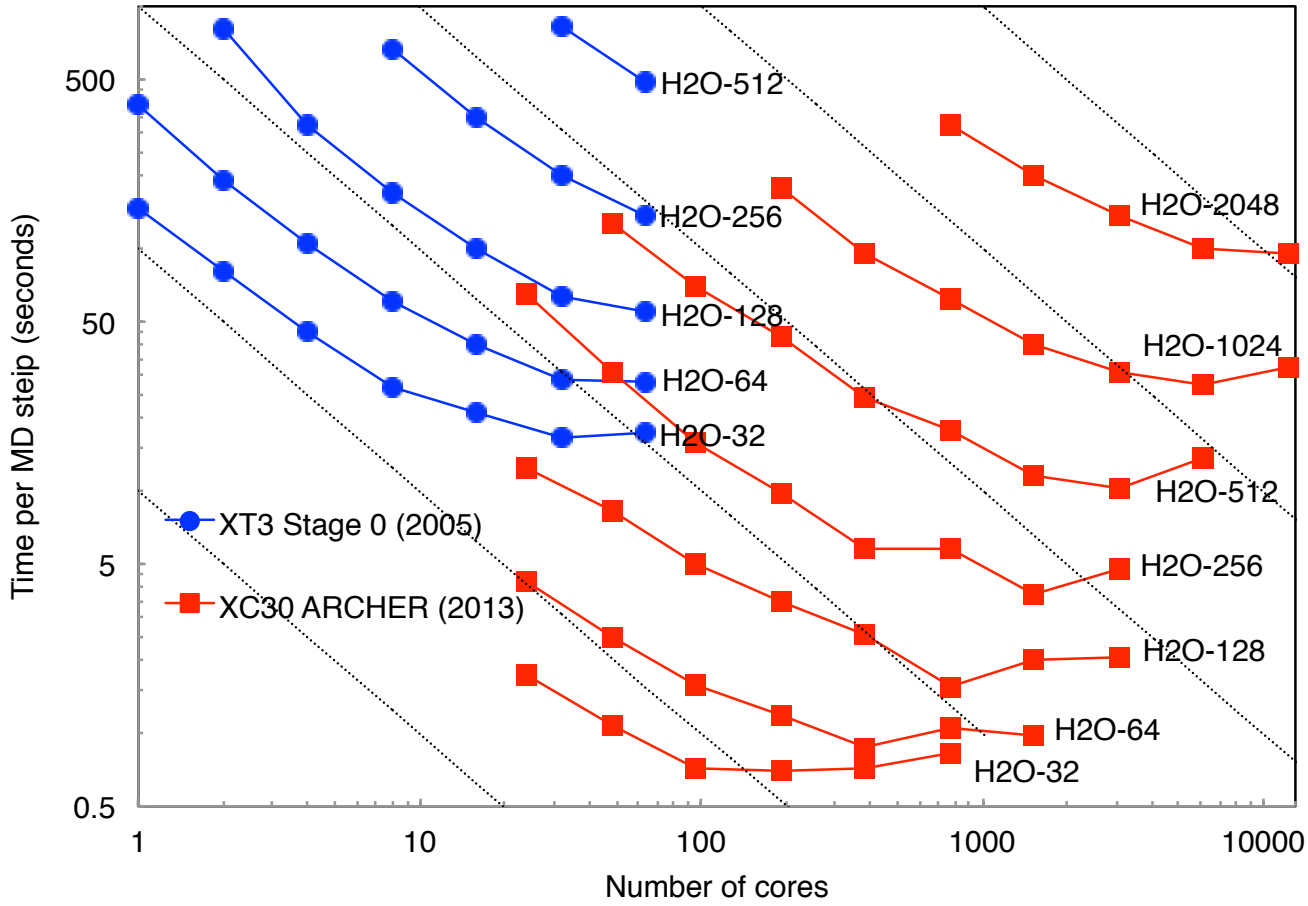
Measuring Parallel Performance

- Important factors for a benchmark calculation
- Use ‘production settings’
  - I/O turned on, chosen simulation parameters
  - Benchmark should closely reflect performance of real simulation
- Reduce the number of MD steps / SCF cycles
  - Long enough to ignore the effects of startup overheads
    - If there are ‘special’ things that only happen in the first step
  - Short enough to not waste CPU time
    - Aim for a few minutes

epcc

THE UNIVERSITY OF EDINBURGH

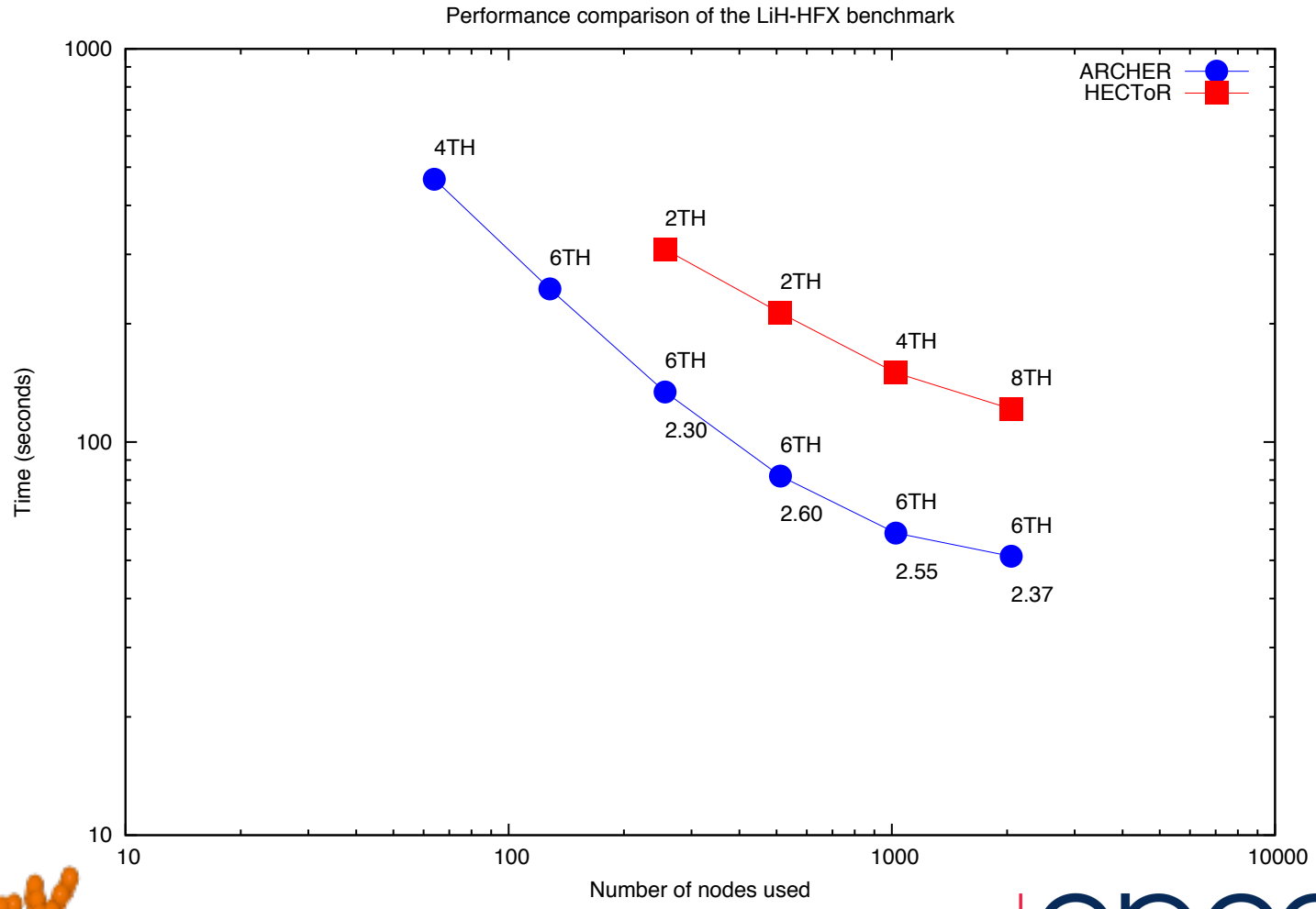
Parallel Performance : H2O-xx



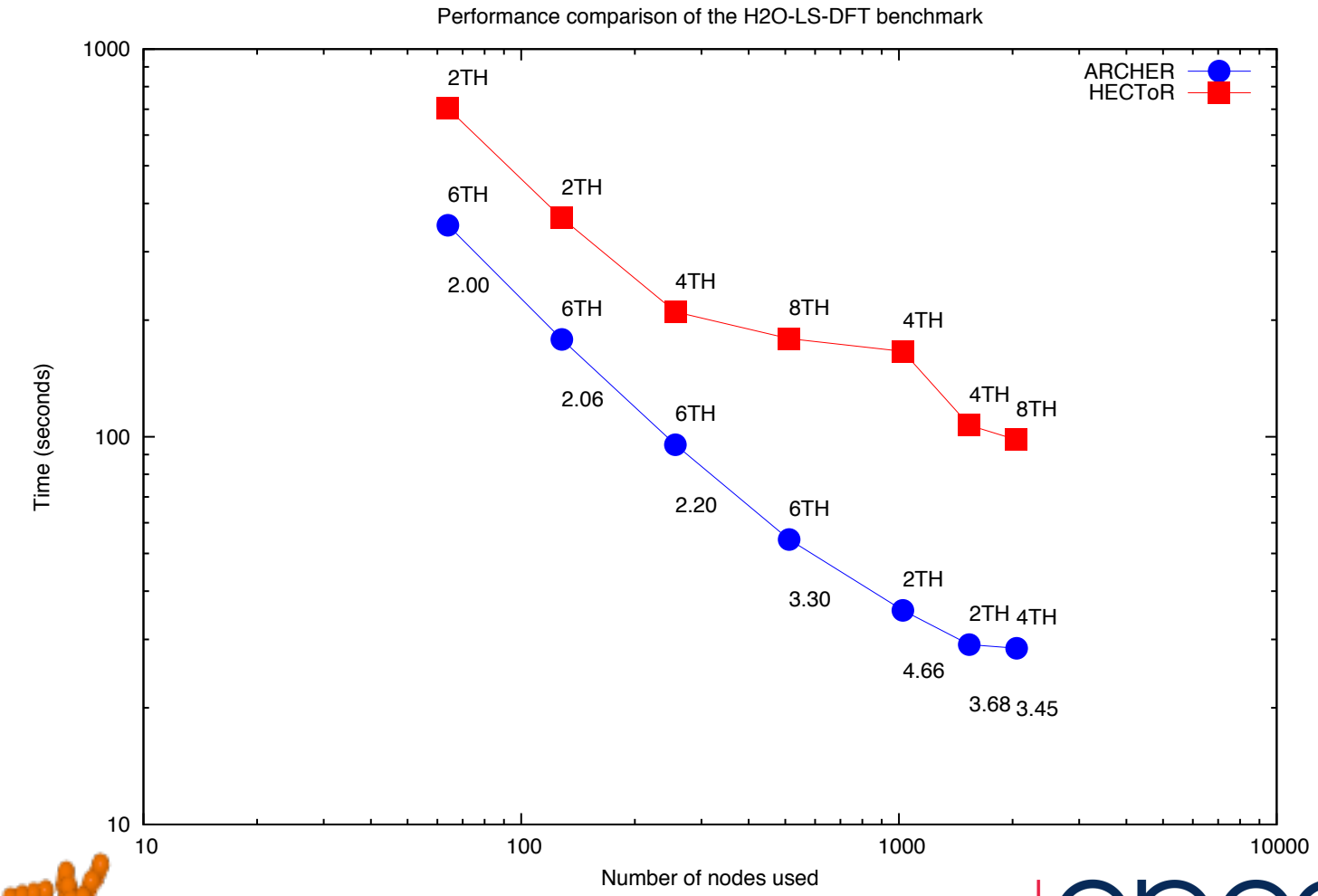
epcc

THE UNIVERSITY OF EDINBURGH

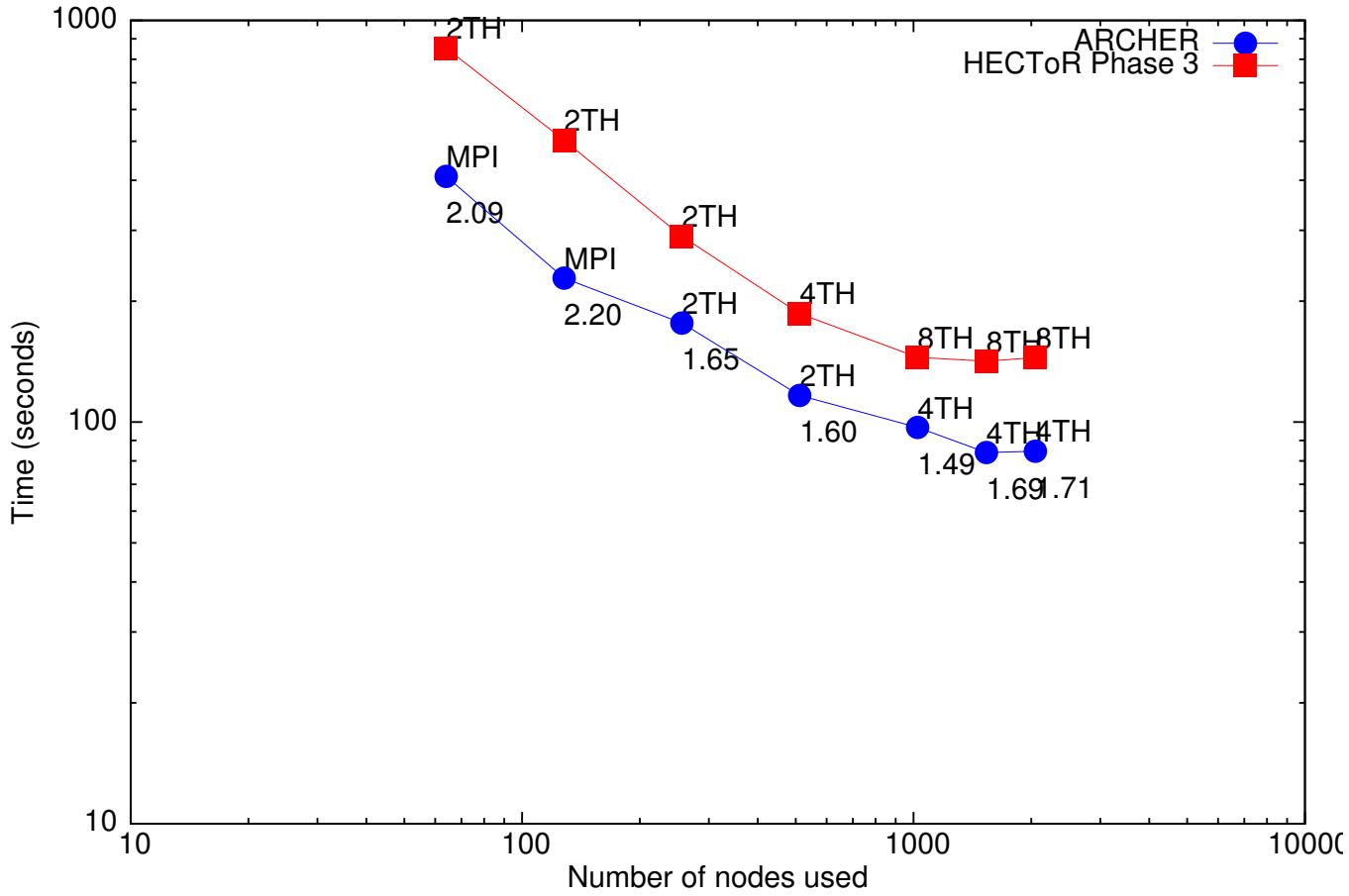
Parallel Performance: LiH-HFX

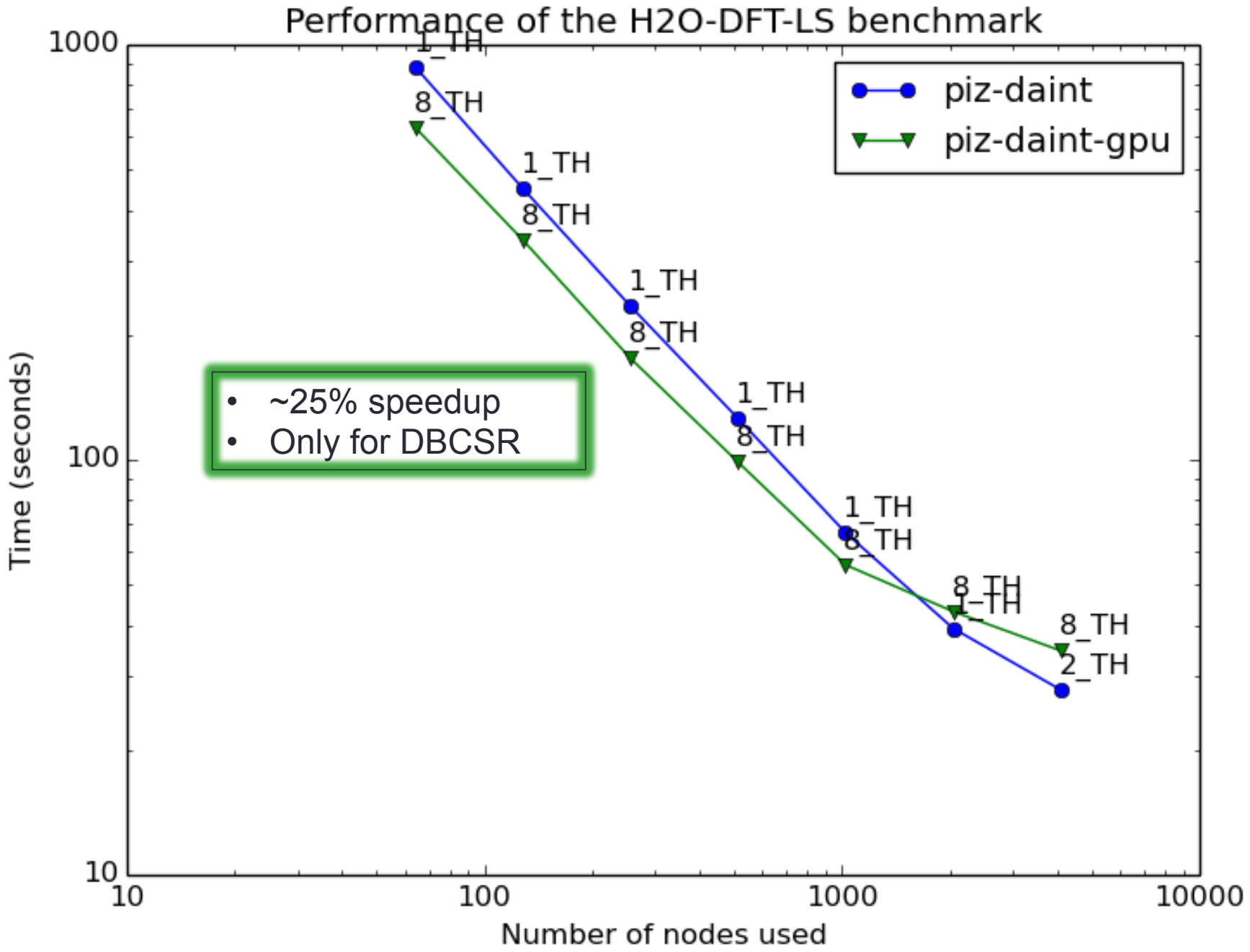


Parallel Performance: H2O-LS-DFT



Parallel Performance: H2O-64-RI-MP2




Performance of the H2O-DFT-LS benchmark
piz-daint
piz-daint-gpu
• ~25% speedup
• Only for DBCSR
Time (seconds)
Number of nodes used
1000
100
10
10
100
1000
10000
1_TH
8_TH
2_TH

CP2K Timing Report

- CP2K measures are reports time spent in routines and communication
  - timing reports are printed at the end of the run

```
-------------------------------------------------------------------------------
-                                                                             -
-                         MESSAGE PASSING PERFORMANCE                         -
-                                                                             -
-------------------------------------------------------------------------------

ROUTINE             CALLS  TOT TIME [s]  AVE VOLUME [Bytes]  PERFORMANCE [MB/s]
MP_Group                4         0.000
MP_Bcast              186         0.018             958318.              9942.82
MP_Allreduce         1418         0.619               2239.                 5.13
MP_Gather              44         0.321              21504.                 2.95
MP_Sync              1372         0.472
MP_Alltoall          1961         5.334          323681322.            119008.54
MP_ISendRecv       337480         0.177               1552.              2953.86
MP_Wait            352330         5.593
MP_comm_split          48         0.054
MP_ISend            39600         0.179              14199.              3147.38
MP_IRecv            39600         0.100              14199.              5638.21
-------------------------------------------------------------------------------
```

epcc

CP2K Timing Report

```
-------------------------------------------------------------------------------
-                                                                             -
-                                T I M I N G                                  -
-                                                                             -
-------------------------------------------------------------------------------
SUBROUTINE                       CALLS  ASD         SELF TIME        TOTAL TIME
                               MAXIMUM       AVERAGE  MAXIMUM  AVERAGE  MAXIMUM
CP2K                                 1  1.0    0.018    0.018   57.900   57.900
qs_mol_dyn_low                       1  2.0    0.007    0.008   57.725   57.737
qs_forces                           11  3.9    0.262    0.278   57.492   57.493
qs_energies_scf                     11  4.9    0.005    0.006   55.828   55.836
scf_env_do_scf                      11  5.9    0.000    0.001   51.007   51.019
scf_env_do_scf_inner_loop           99  6.5    0.003    0.007   43.388   43.389
velocity_verlet                     10  3.0    0.001    0.001   32.954   32.955
qs_scf_loop_do_ot                   99  7.5    0.000    0.000   29.807   29.918
ot_scf_mini                         99  8.5    0.003    0.004   28.538   28.627
cp_dbcsr_multiply_d               2338 11.6    0.005    0.006   25.588   25.936
dbcsr_mm_cannon_multiply          2338 13.6    2.794    3.975   25.458   25.809
cannon_multiply_low               2338 14.6    3.845    4.349   14.697   15.980
ot_mini                             99  9.5    0.003    0.004   15.701   15.942
-------------------------------------------------------------------------------
```

epcc

CP2K Timing Report

- Not just for developers!
  - Check that communication is < 50% of total runtime
  - Check where most time is being spent:
    - Sparse matrix multiplication - `cp_dbcsr_multiply_d`
    - Dense matrix algebra – `cp_fm_syevd` (`&DIAGONALISATION`), `cp_fm_cholesky_*` (`&OT`), `cp_fm_gemm`
    - FFT – `fft3d_*`
    - Collocate / integrate – `calculate_rho_elec`, `integrate_v_rspace`
- Control level of granularity

```
&GLOBAL
  &TIMINGS
    THRESHOLD 0.00001
  &END TIMINGS
&END GLOBAL
```

Default is 0.02 (2%)

epcc

THE UNIVERSITY OF EDINBURGH

Summary

- First look for algorithmic gains
  - Cell size, SCF settings, preconditioner, choice of basis set, QM/MM, ADMM…
- Check scaling of *your* system
  - Run a few MD steps / reduced `MAX_SCF`
- Almost all performance-critical code is in libraries
  - Compiler optimisation –O3 is good enough
  - Intel vs gfortran vs Cray – difference is close to zero
- Before spending 1,000s of CPU hours, build libsmm, libgrid, ELPA, FFTW3…
  - Or ask your local HPC support team ☺

epcc

THE UNIVERSITY OF EDINBURGH

CP2K Parallelisation and Optimisation

Questions?

epcc

THE UNIVERSITY OF EDINBURGH