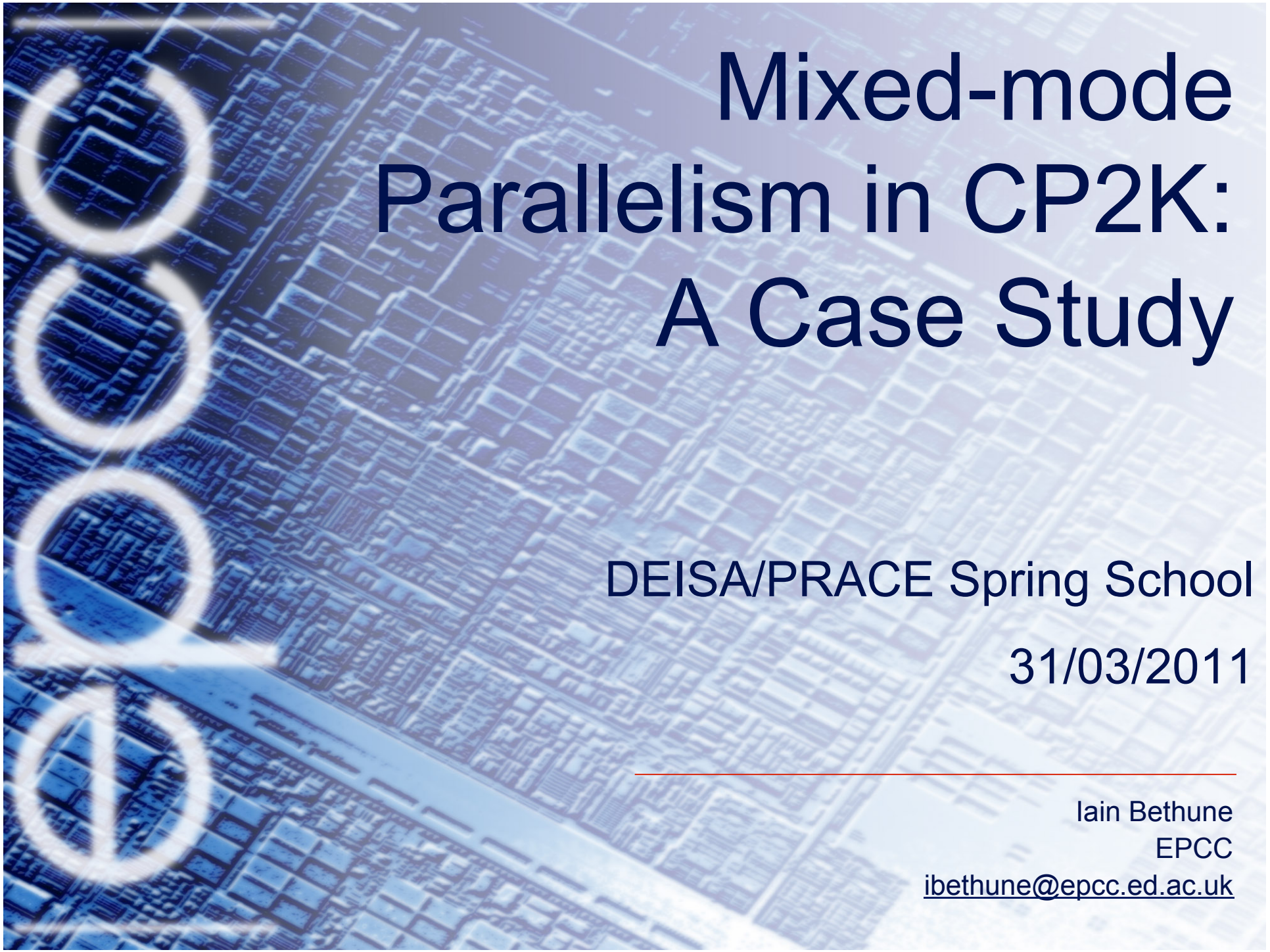


Mixed-mode Parallelism in CP2K: A Case Study

DEISA/PRACE Spring School

31/03/2011

Iain Bethune
EPCC
ibethune@epcc.ed.ac.uk

CP2K: Contents

- CP2K Overview
- OpenMP Strategy
- Fast Fourier Transforms
- Collocate and Integrate
- Miscellaneous Gotchas
- Performance Results
- Summary

CP2K: Overview

- CP2K is a freely available (GPL) Density Functional Theory code (+ support for classical, empirical potentials) – can perform MD, MC, geometry optimisation, normal mode calculations…

- The “Swiss Army Knife of Molecular Simulation” (VandeVondele)
- c.f. CASTEP, VASP, CPMD etc.

CP2K: Overview

- CP2K is a freely available (GPL) Density Functional Theory code (+ support for classical, empirical potentials) – can perform MD, MC, geometry optimisation, normal mode calculations…

- The “Swiss Army Knife of Molecular Simulation” (VandeVondele)
- c.f. CASTEP, VASP, CPMD etc.

CP2K: Overview

- Over 600,000 lines of FORTRAN 90/95
- Efficient MPI parallelisation – scales to 1,000s of tasks for large enough systems.
- Some existing OpenMP directives, but not kept up to date
- Actively developed by ~10 members of distributed development team (Uni. Zurich, ETH, EPCC, and more…)

CP2K: Overview

- Employs a dual-basis (GPW) method to calculate energies, forces, K-S Matrix in linear time
  - N.B. linear scaling in number of atoms, not processors!

- As a result, there are various (complicated) data structures and algorithms in the code:
  - Regular grids (1D, 2D, 3D domain decomposed)
  - Sparse Matrices
  - Dense Matrices
  - Distributed task lists

CP2K: OpenMP Strategy

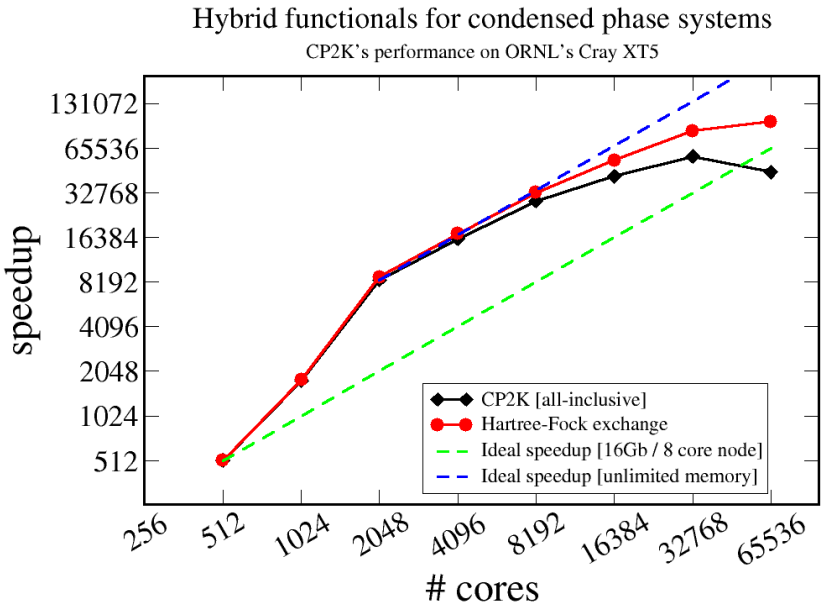
- Work reported here done under
  - HECToR dCSE Project “Improving the scalability of CP2K on multi-core systems” (50% FTE) Sep 09 – Aug 10
- Use OpenMP to parallelise only areas of the code which consume the most CPU time (Amdahl’s law)
  - Setting up a parallel region is relatively cheap, allowing micro-parallel regions
  - Can see this via CrayPat (omp-rtl trace group) in final lab session
- MPI Communication takes place on a single thread outside the parallel regions

CP2K: OpenMP Strategy

- Expect improved performance for the following reasons:
  - Reduce impact of algorithms which scale poorly with number of MPI tasks
    - E.g. When using T threads, switchover point from 1D decomposed FFT (more efficient) to 2D decomposed FFT (less efficient) is increased by a factor of T
  - Improved load balancing
    - Existing MPI load balancing algorithms do a coarser load balance, fine-grained balance done over OpenMP threads
  - Reduced number of messages significantly
    - Especially on pre-Gemini networks
    - For all-to-all communications, message count reduced by factor of T^2

CP2K: OpenMP Strategy

- Further motivations:
  - extremely scalable Hartree-Fock Exchange (HFX) code uses OpenMP to access more memory per task, and is limited to 32,000 cores by non-HFX part of the code
  - HPC architectures (e.g. Cray XT/XE going increasingly multi-core -> map well to architecture by using OpenMP on node, MPI between nodes



M. Guidon, J. Hutter, J. VandeVondele, Univ. Zurich
J. Levesque, Cray inc.

Bulk LiH, 216 atoms
8100 Gaussian basis functions

CP2K: Functional Evaluation

- One of the key routines in the code – evaluate a functional (and derivatives) over the electronic density:

```
CALL pbe_lda_calc(rho=rho, ..., e_rho=e_rho, ...)

...

SUBROUTINE pbe_lda_calc(rho, ..., e_rho, ...)

...

 DO ii=1,npoints
        my_rho = rho(ii)
        ...
        t6 = 0.1e1_dp / my_rho
        t7 = t5 * t6
        ...
        e_rho(ii) = e_rho(ii)+&
            scale_ex * (ex_unif * Fx + t208 * Fx + t108 * Fxrho) + &
            scale_ec * (epsilon_cGGA + my_rho * epsilon_cGGArho)
        ...
 END DO
...
END SUBROUTINE
```

- npoints is large (>1000), iterations are independent, and all cost the same -> easy to parallelise

CP2K: Functional Evaluation

```
!$omp parallel default(none), &
!$omp shared(rho,...,e_rho,...)
CALL pbe_lda_calc(rho=rho, ..., e_rho=e_rho, ...)
!$omp end parallel

...

SUBROUTINE pbe_lda_calc(rho, ..., e_rho, ...)

...

!$omp do
 DO ii=1,npoints
        my_rho = rho(ii)
        ...
        t6 = 0.1e1_dp / my_rho
        t7 = t5 * t6
        ...
        e_rho(ii) = e_rho(ii)+&
            scale_ex * (ex_unif * Fx + t208 * Fx + t108 * Fxrho) + &
            scale_ec * (epsilon_cGGA + my_rho * epsilon_cGGArho)
        ...
 END DO
!$omp end do
...
END SUBROUTINE
```

CP2K: Functional Evaluation

- Key points:
  - default(none) is good practice, but verbose. Compiler will force you to think about the sharing (or not) of all variables
  - Place parallel region outside function call to avoid specifying all the 100s of local variables, which are made private by default
  - The default schedule for the loop is a static schedule – each thread gets npoints/nthreads consecutive iterations

CP2K: Functional Evaluation

- Result

Threads	1	2	3	4	6	12	24
`pbe_lda_eval`	7.98	4.05	2.73	2.08	1.42	0.75	0.45
Speedup	1	1.97	2.92	3.84	5.62	10.64	17.73

Table 3: Times (in seconds) and speedup for PBE evaluation on HECToR Phase 2b

- 93% efficiency with 6 threads, 74% with 24 threads

CP2K: Fast Fourier Transforms

- CP2K uses a 3D Fourier Transform to turn real data on the plane wave grids into g-space data on the plane wave grids.
- The grids may be distributed as planes, or rays (pencils) – so the FFT may involve one or two transpose steps between the 3 1D FFT operations
- The 1D FFTs are performed via an interface which supports many libraries e.g. FFTW 2/3 ESSL, ACML, CUDA, FFTSG (in-built)

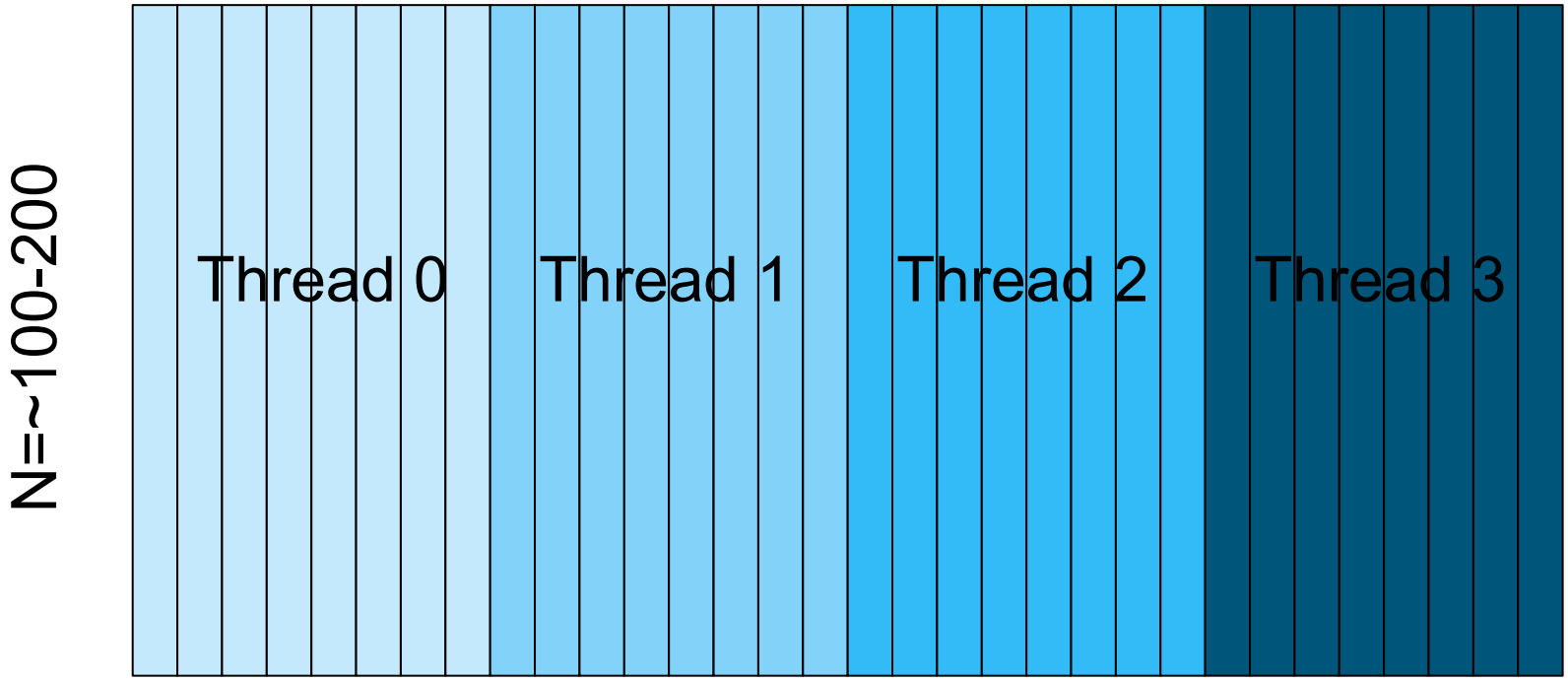
CP2K: Fast Fourier Transforms

- We can parallelise two parts with OpenMP
- 1D FFT – assign each thread a subset of rows to FFT
- Buffer packing – threads cooperatively pack the buffers which are passed to MPI
- Communication still handled outside a the parallel regions

CP2K: Fast Fourier Transforms

- Typically we have M 1D FFTs of length N to perform

M=~N/P in 1D or ~N/sqrt(P) in 2D)



- Ask FFTW to plan M/T FFTs of length N, each thread executes the plan starting at a different offset in the array

CP2K: Fast Fourier Transforms

- Some care needed:
  - If T does not divide M, we need to have two different plans, so some threads will do slightly more FFTs than others
  - Must ensure each thread's first element is 16-byte aligned to FFTW can safely use SSE instructions (~2x speedup). The only case this affects is if N is odd and we are using single-precision values (each element is complex number so 8 bytes). In this case we always divide up on pairs of FFTs to ensure alignment is retained.

CP2K: Fast Fourier Transforms

- Buffer (un)packing:

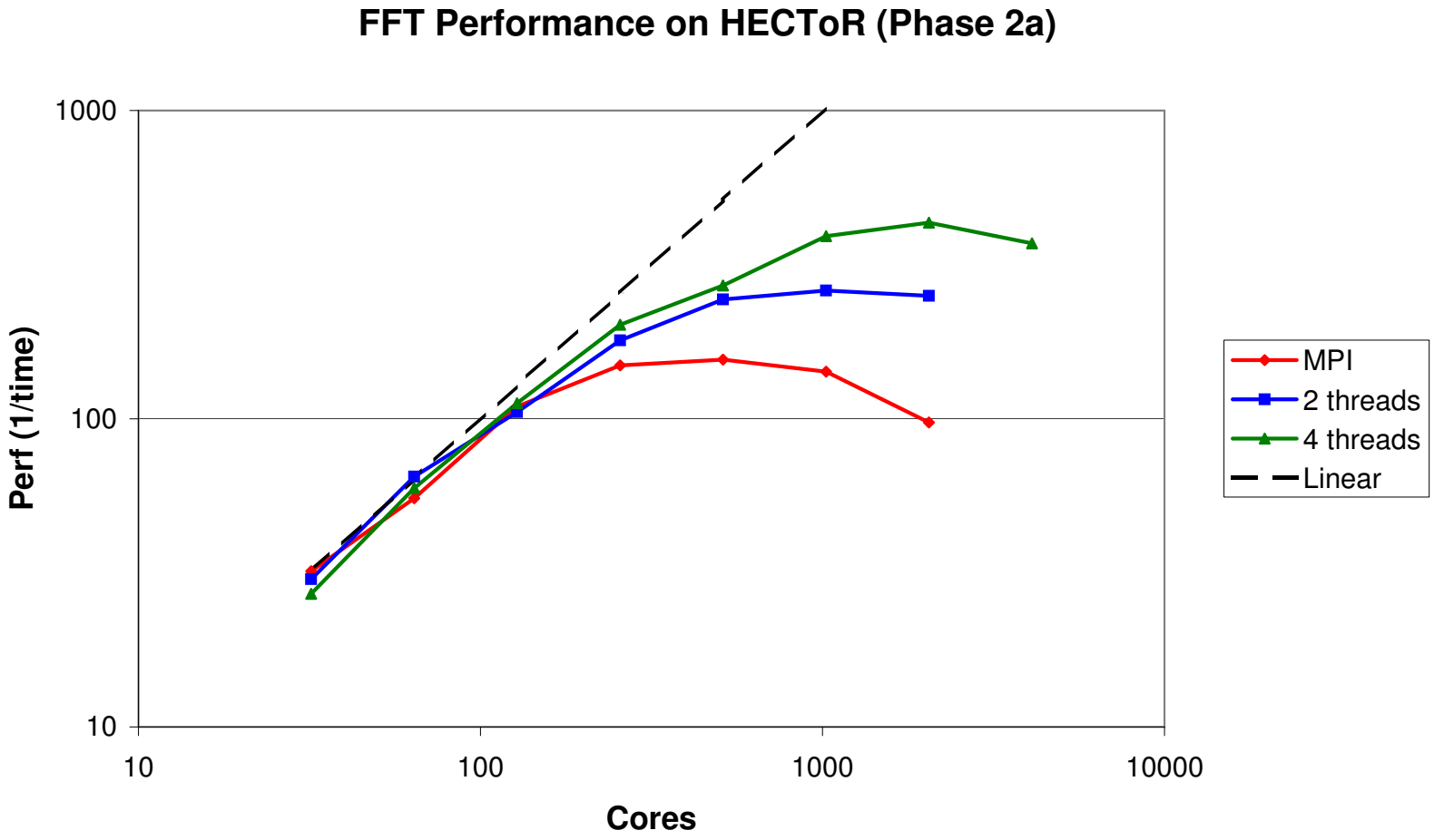
```
CALL mp_alltoall ( cin, scount, sdispl, rbuf, rcount, rdispl, sub_group )

!$omp parallel do default(none) __COLLAPSE2 &
!$omp             private(ip,ipl,nz,iz,is,ir) &
!$omp             shared(nx,ny,np,pgrid,boin,sout,rbuf)
    DO ixy = 1, nx * ny
       DO ip = 0, np - 1
          ipl = pgrid ( ip, 2 )
          nz = boin ( 2, 3, ipl ) - boin ( 1, 3, ipl ) + 1
          DO iz = 1, nz
             is = boin ( 1, 3, ipl ) + iz - 1
             ir = iz + nz * ( ixy - 1 )
             sout ( is, ixy ) = rbuf ( ir, ip )
          END DO
       END DO
    END DO
!$omp end parallel do
```

CP2K: Fast Fourier Transforms

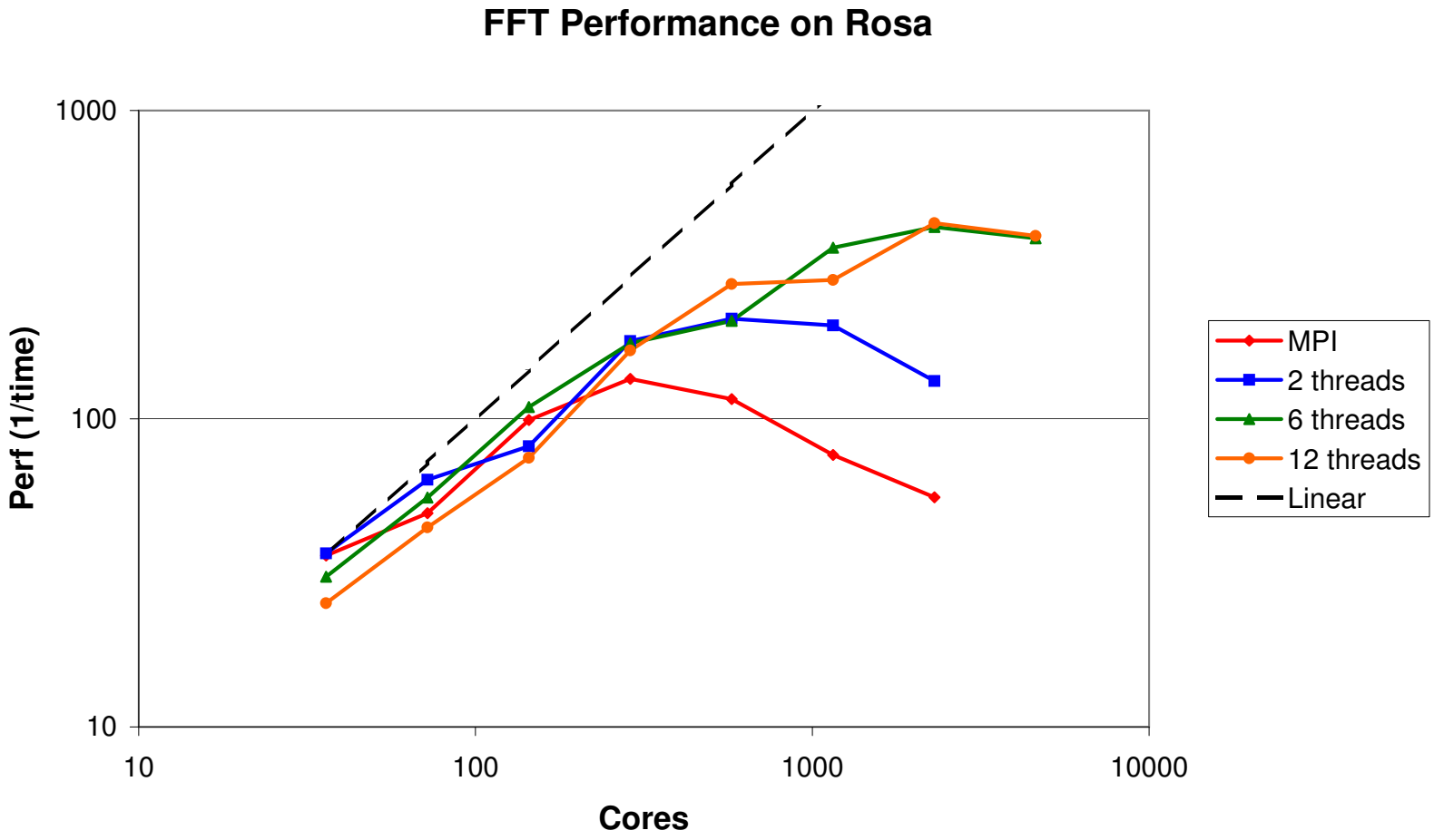
- Key points:
  - Outer loop over points (X*Y), inner loop over processors, for small P, there are many iterations in the outer loop, for large P, there may be less than T iterations in the outer loop.
  - OpenMP 3.0 provides the collapse clause that merges the loop nest and parallelises over the entire iteration space, so we will always have enough iterations to make good use of all threads.
  - For pre-3.0 compilers (e.g. Pathscale), we can define out the collapse clause:

```
#if defined(__HAS_NO_OMP_3)
#define __COLLAPSE2
#else
#define __COLLAPSE2 collapse(2)
#endif
```

CP2K: Fast Fourier Transforms



CP2K: Fast Fourier Transforms



CP2K: Collocate and Integrate

- Another computationally expensive step is mapping between a sparse matrix representation of the electron density (coefficients of 3D Gaussian basis functions) and the real-space grids, prior to FFT.

- Original implementation looped over a task list, and for each task, read some elements from the matrix, evaluated the function, and summed data onto the grids (Collocation)

- In reverse (Integration), for each task, a region of the grid is read, and part of the matrix is updated with the new values.

CP2K: Collocate and Integrate

- But:
  - Basis functions may overlap, so we need to ensure threads don't update the same area of the grids simultaneously (race condition)
  - Matrix library (DBCSR) requires that only a single thread update a single block of the matrix between 'finalization' calls (expensive)
  - We in fact have multiple grid levels (typically 4-6, coarse to fine), and the task list is ordered such that all tasks for a given grid level are consecutive, so that the grids (several MB) are retained in cache – we need to preserve this
- Solution:
  - Preprocess the task list and split it by grid level, and by atom pair (corresponding to individual matrix blocks)

CP2K: Collocate and Integrate

```
DO ipair = 1, SIZE(task_list}
  <process each task>
END DO
```

Becomes…

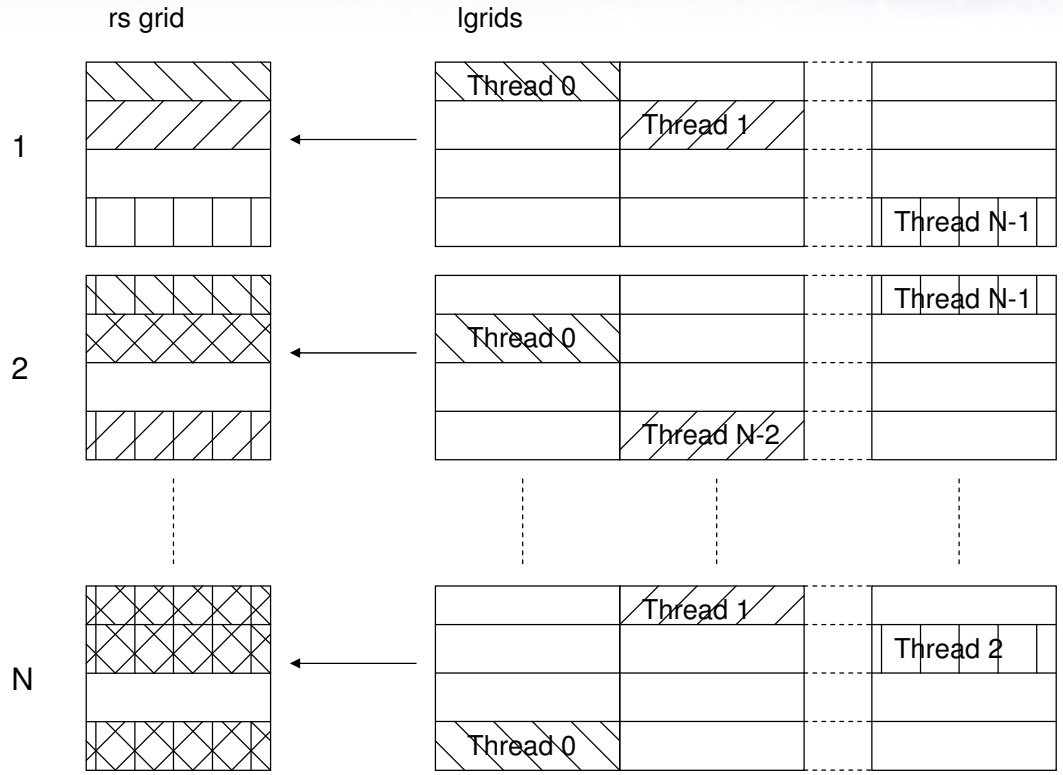
```
!$omp parallel
DO ilevel = 1, ngrid_levels
  !$omp do
  DO ipair = 1, task_list%npairs(ilevel)
    DO itask = task_list%taskstart(ilevel,ipair), task_list%taskstop(ilevel,ipair)
      <process each task>
    END DO
  END DO
  !$omp end do
END DO
!$omp end parallel
```

Integrate is now easy, just need to finalize the matrix at the end of each grid level loop

CP2K: Collocate and Integrate

- Collocate is more tricky…
  - Tried using OpenMP locks to protect updates to regions of the grid, but hard to be granular enough to avoid contention, and have low overhead setting up & destroying the locks
  - Eventually settled on giving each thread a thread-local copy of the grid, and then performing a manual reduction step after each grid level is completed.
  - There are various possible ways of doing the reduction, currently using a method where each thread sums its local grid into the shared grid a section at time.

CP2K: Collocate and Integrate



Threads	1	2	3	4	6	12	24
Collocate Speedup	1	1.9	2.9	3.7	5.5	8.2	10.4
Integrate Speedup	1	1.8	2.6	3.2	3.9	2.7	1

CP2K: Miscellaneous Gotchas

- How much should you trust your compiler?
  - In several places we have large array operations (zeroing, summation) that we would like to parallelise. Should be as simple as:

```
Real(kind=dp), dimension(:) :: a
!$omp parallel workshare
  a = 0.0
!$omp end parallel workshare
```

  - It turns out that for GNU OpenMP (prior to GCC 4.5), the `workshare` directive is implemented as a `single` !!! So we need to implement workshare ourselves…

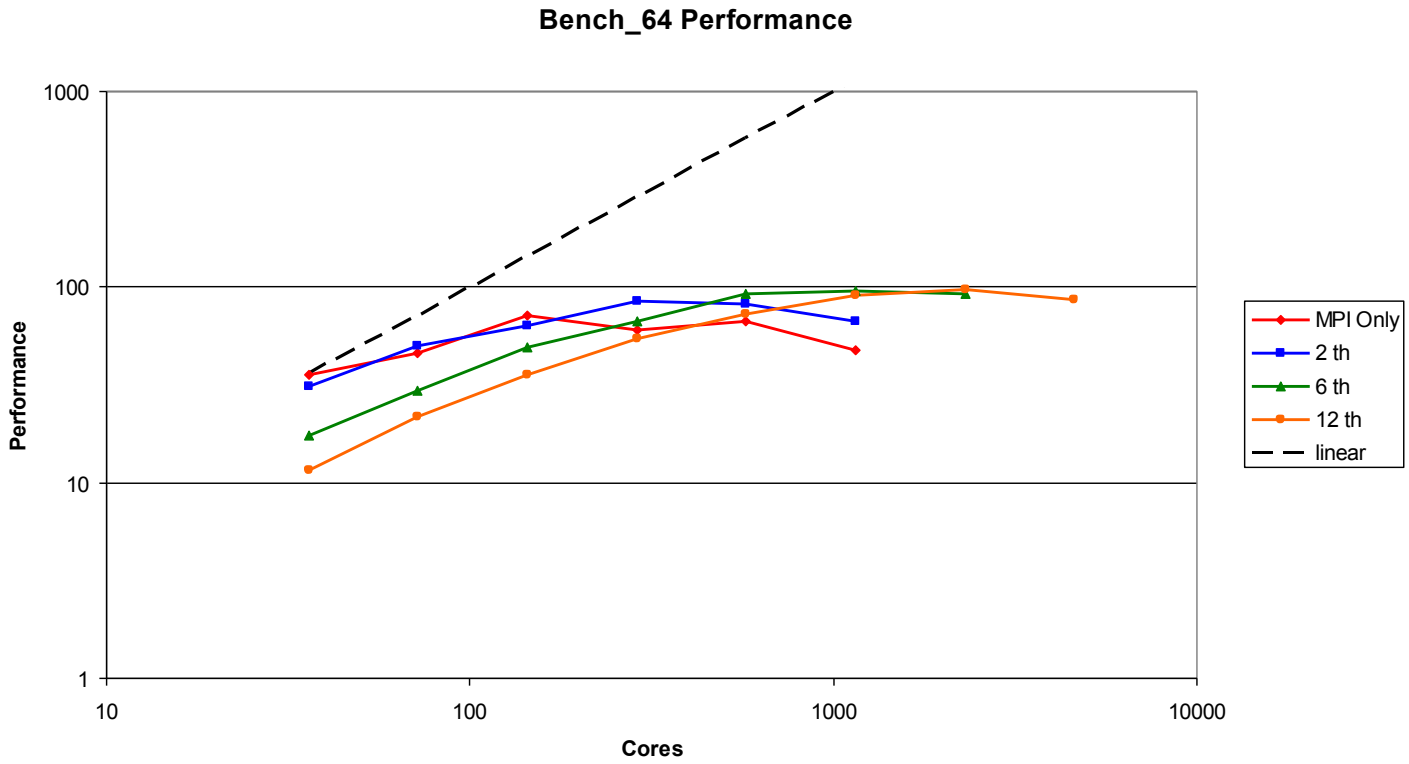
```
!$omp parallel default(none)&
!$omp          private(num_threads,my_id,lb,ub), &
!$omp          shared(buf)
!$  num_threads = MIN(omp_get_max_threads(), SIZE(buf))
!$  my_id = omp_get_thread_num()
    IF (my_id < num_threads) THEN
      lb = (SIZE(buf)*my_id)/num_threads
      ub = (SIZE(buf)*(my_id+1))/num_threads - 1
      buf(lb:ub) = 0.0
    END IF
!$omp end parallel
```

CP2K: Miscellaneous Gotchas

- Thread to core binding does not work as expected with GCC on Cray CNL.
  - By default, the operating system binds threads to cores. This is typically what we want for HPC codes, since we usually only have as many threads as cores.
  - However, if entering a parallel region with less threads e.g. using the num_threads clause, GCC terminates the unneeded threads, and when starting them up again later, binds them starting at core 0. So may eventually end up with all threads on a single core!
  - Work arounds:
    - Use –cc none aprun flag to allow threads to migrate to idle cores (but may suffer performance hit)
    - Avoid using num_threads, manually idle threads inside a parallel region

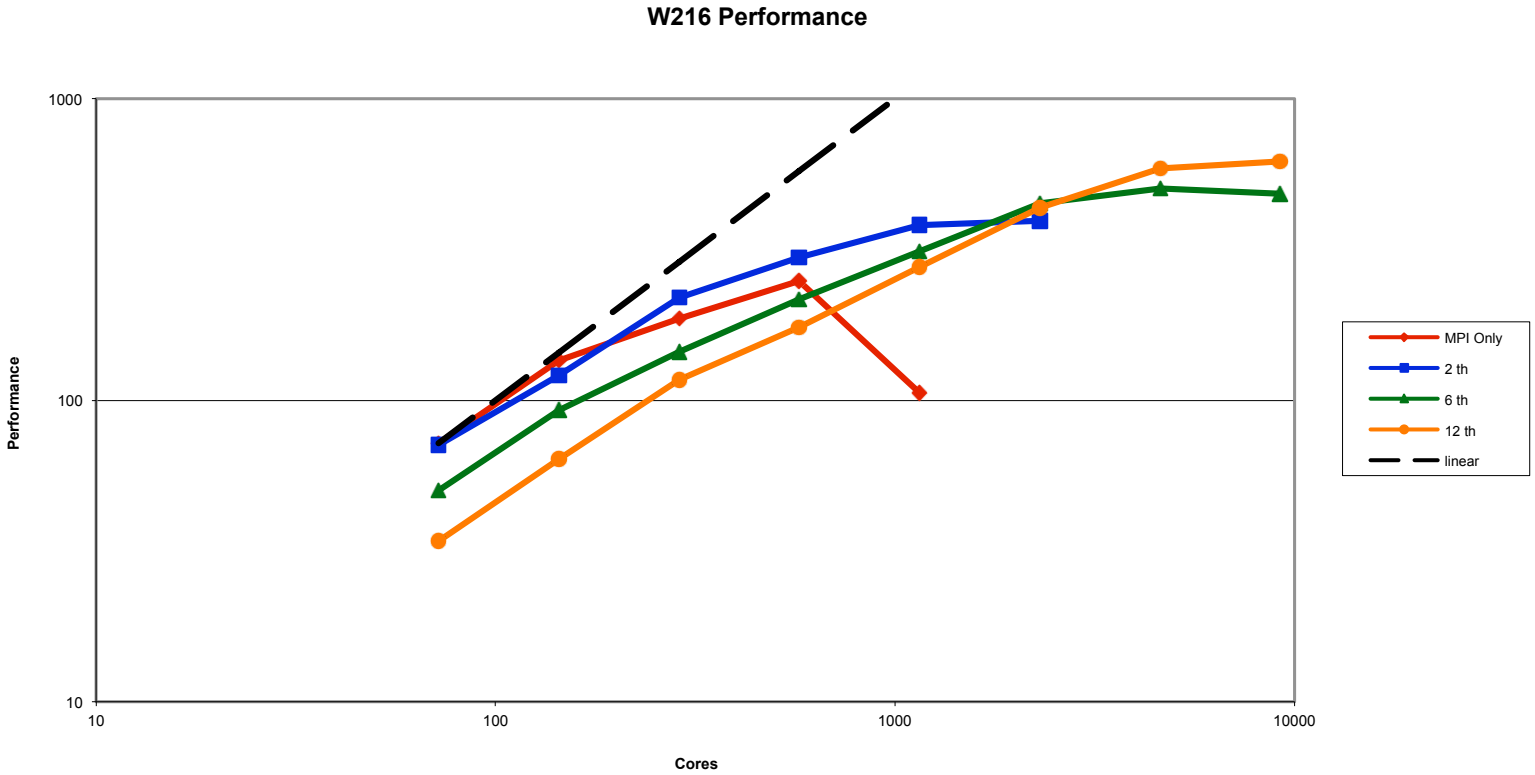
CP2K: Performance Results

- Results so far (H_2O-64):
  - Fastest pure MPI run = 85s on 144 cores
  - Fastest 2 threads/task = 72s on 288 cores
  - Fastest 6 threads/task = 64s on 1152 cores
  - Fastest 12 threads/task = 63s on 2304 cores



CP2K: Performance Results

- Results so far (W216):
  - Fastest pure MPI run = 1662s on 576 cores
  - Fastest 2 threads/task = 1047s on 2304 cores
  - Fastest 6 threads/task = 816s on 4608 cores
  - Fastest 12 threads/task = 665s on 9216 cores (and more?)



CP2K: Summary

- Benefits of mixed-mode OpenMP/MPI
  - Using multiple threads per task increases scalability by factor of T
  - Can get a faster time to solution (~25% at expense of more AUs)
  - Small runs may be slower with more threads (as the unthreaded sections are more significant)
  - Even greater speedup when used in load-imbalanced case (less MPI tasks -> better load balance)

- Also, new sparse matrix library DBCSR by Borstnik et al (Zurich)
  - High scalability (MPI)
  - Designed with OpenMP threads for matrix operations
  - Currently under revision, proposed use of OpenMP 3.0 tasks for threaded recursive sparse matrix multiply

CP2K: Summary

- PRACE WP7.2 Applications Enabling with Communities

  – Project Proposal submitted to improve and extend OpenMP parallelism within CP2K

  – Aim to demonstrate the value of using mixed-mode version of the code to user groups across Europe

  – Will use a set of real user-supplied test cases to benchmark changes and guide development.

  – If you use CP2K, or know a group who does, please get in touch:

ibethune@epcc.ed.ac.uk

CP2K: Summary

Thanks for listening!

Questions / Comments / Feedback?

ibethune@epcc.ed.ac.uk