Improving the Performance of CP2K on the Cray XT

CUG 2010

27/05/2010

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

CP2K: Contents

- Introduction to CP2K
- MPI Optimisation
- Fast Fourier Transforms
- Load Balancing
- Introducing OpenMP into CP2K
- Summary

CP2K: Introduction

- Work funded by the HECToR Distributed Computational Science & Engineering (dCSE) Support programme
- In Collaboration with:
  - Slater, Watkins @ UCL (HECToR Users)
  - VandeVondele et al @ PCI, University of Zurich (CP2K Developers)
- Aug 08 – Jul 09
  - HECToR dCSE Project “Improving the performance of CP2K”
- Sep 09 – Aug 10
  - Follow on dCSE Project “Improving the scalability of CP2K on multi-core systems”
- Total of 1 FTE over 2 years

CP2K: Introduction

- Systems used during the projects
- EPCC, University of Edinburgh
  - HECToR 'Phase 1'
    - Cray XT4, 5664 2.8GHz dual-core CPUs
    - 2-way shared memory (OpenMP node)
  - HECToR 'Phase 2a'
    - Cray XT4, 5664 2.3GHz quad-core 'Budapest' CPUs
    - 4-way shared memory (OpenMP node)
- CSCS, Swiss National Supercomputing Centre
  - Rosa
    - Cray XT5, 3688 2.4GHz hexa-core 'Istanbul' CPUs
    - 12-way shared memory (OpenMP) node
    - Thanks to J. Hutter (Zurich) for access

CP2K: Introduction

- 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: Introduction

- 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: Introduction

- Developed since 2000, open source approach, ~20 developers – mainly based in Univ Zurich / ETHZ / IBM Zurich
- 600,000+ lines of Fortran 95, ~1,000 source files

- Employs a dual-basis (GPW[1]) method to calculate energies, forces, K-S Matrix in linear time
  - N.B. linear scaling in number of atoms, not processors!

1) J. VandeVondele, M. Krack, F. Mohamed, M.Parrinello, T. Chassaing, J. Hutter, Comp. Phys. Comm. 167, 103 (2005)

CP2K: Algorithm

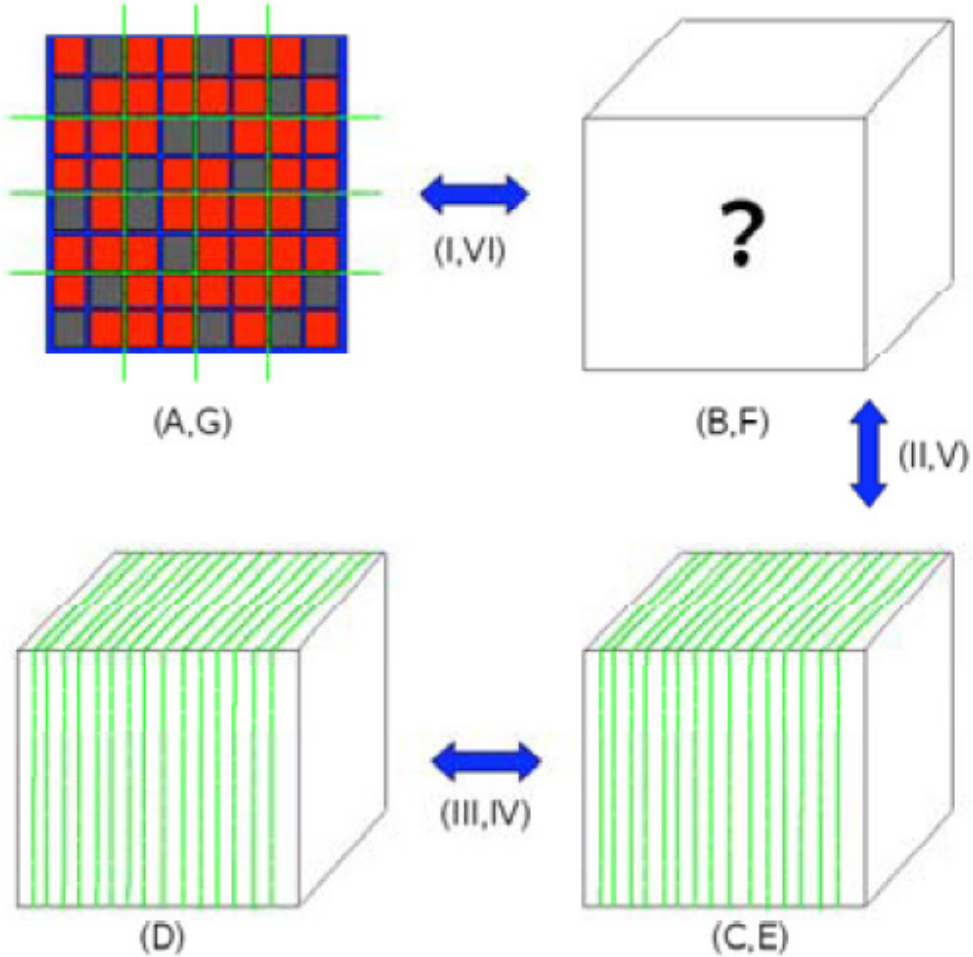
- The Gaussian basis results in sparse matrices which can be cheaply manipulated e.g. diagonalisation during SCF calculation.
- The Plane wave basis (relying on FFTs) allows easy calculation of long-range electrostatics.

- A key step in the algorithm is transforming from one representation to the other (and back again) – this is done once each way per SCF cycle.

CP2K: Algorithm

- (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)



CP2K: MPI Optimisation

- The rs2pw halo swap step becomes a bottleneck as the number of cores increases (e.g. on 512 cores, 125^3 grid, 90%+ of data is in the halo!)

- In CP2K, the halo region (containing Gaussian data mapped locally) of a process is sent and summed into the core region of a neighbouring process

- So, throw away any data that won't end up in any core region!

CP2K: MPI Optimisation

	Before	After
Avg. Message Size (bytes)	194688	91008
Time in SendRecv (s)	0.468	0.22
Time packing X bufs (s)	0.107	0.002
Time unpacking X bufs (s)	0.189	0.003
Time packing Y bufs (s)	0.060	0.005
Time unpacking Y bufs (s)	0.096	0.017
Time packing Z bufs (s)	0.054	0.054
Time unpacking Z bufs (s)	0.091	0.091

60 iterations of the rs2pw libtest, before and after optimisation

CP2K: MPI Optimisation

- Also added non-blocking MPI communication
- The result – a 14% speedup on 256 cores:

Cores	16	32	64	128	256	512
Before(s)	952	541	318	268	217	264
After(s)	938	519	296	247	190	235
Speedup(%)	2	4	7	9	14	12

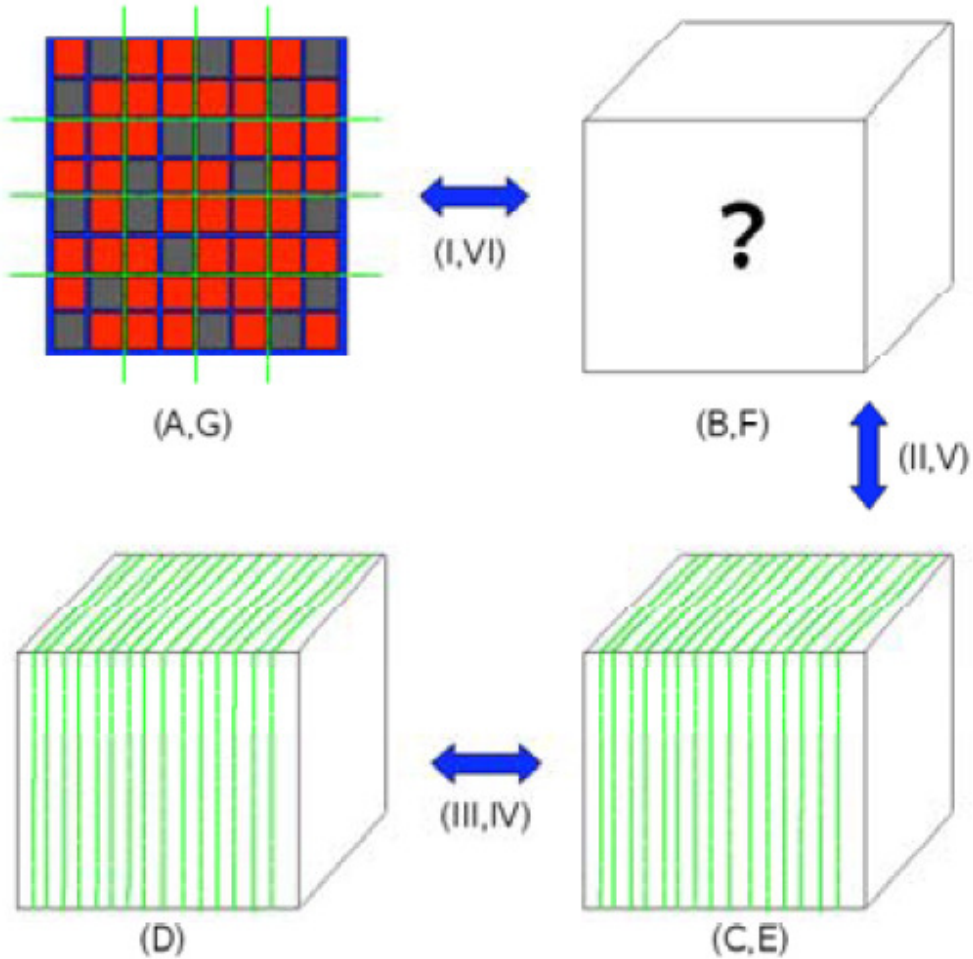
Comparison of bench_64 runtime before and after rs2pw optimisation

- bench_64 is a small test case of 64 water molecules, 40,000 basis functions, 50 MD steps

CP2K: Algorithm

- (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)



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

- Initial profiling of the 3D FFT using CrayPAT showed many expensive calls to MPI_Cart_sub to decompose the cartesian topology – called every iteration, generating the same set of sub-communicators each time!

Time %	Time	Imb. Time	Imb. Time %	Calls	Group Function PE.Thread='HIDE'
100.0%	19.588726	--	--	126389.0	Total
62.8%	12.298019	--	--	120362.0	MPI
37.1%	7.270134	0.741629	9.3%	4000.0	mpi_cart_sub_
24.4%	4.782975	1.257500	20.9%	4000.0	mpi_alltoallv_
0.7%	0.144511	0.006960	4.6%	2002.0	mpi_barrier_
0.2%	0.034614	0.003197	8.5%	24065.0	mpi_wtime_
0.1%	0.025250	0.002017	7.4%	70001.0	mpi_cart_rank_
0.1%	0.014001	0.001163	7.7%	4002.0	mpi_comm_free_
0.0%	0.008200	0.001827	18.3%	6002.0	mpi_cart_get_
0.0%	0.007483	0.001781	19.3%	6005.0	mpi_comm_size_
...					

CP2K: Fast Fourier Transforms

- CP2K already has a data structure `fft_scratch` which stores buffers, coordinates etc. for reuse
- The communicators, and a number of other pieces of data were added
- Number of MPI_Cart_sub calls reduced from 11722 to 5 (for 50 MD steps)

Cores	64	128	256	512
Before(s)	366	264	191	238
After(s)	363	250	177	213
Speedup(%)	1	6	8	12

Comparison of bench_64 runtime before and after FFT caching optimisation

- N.B. This speedup would increase for longer runs

CP2K: Fast Fourier Transforms

- Initially the FFTW interface did not use FFTW plans effectively
  - At each step a plan would be created, used, and destroyed.
- But at least the interface was simple, and consistent with the other FFT libraries
- Implemented storage and re-use of plans for FFTW 2 and 3 – for other libraries planning is a no-op

CP2K: Fast Fourier Transforms

- This allowed the more expensive plan types to used:

	Time(s)	Speedup(%)
Original Code	997	
FFTW_ESTIMATE	995	0.2
FFTW_MEASURE	989	0.8
FFTW_PATIENT	975	2.3
FFTW_EXHAUSTIVE	1081	

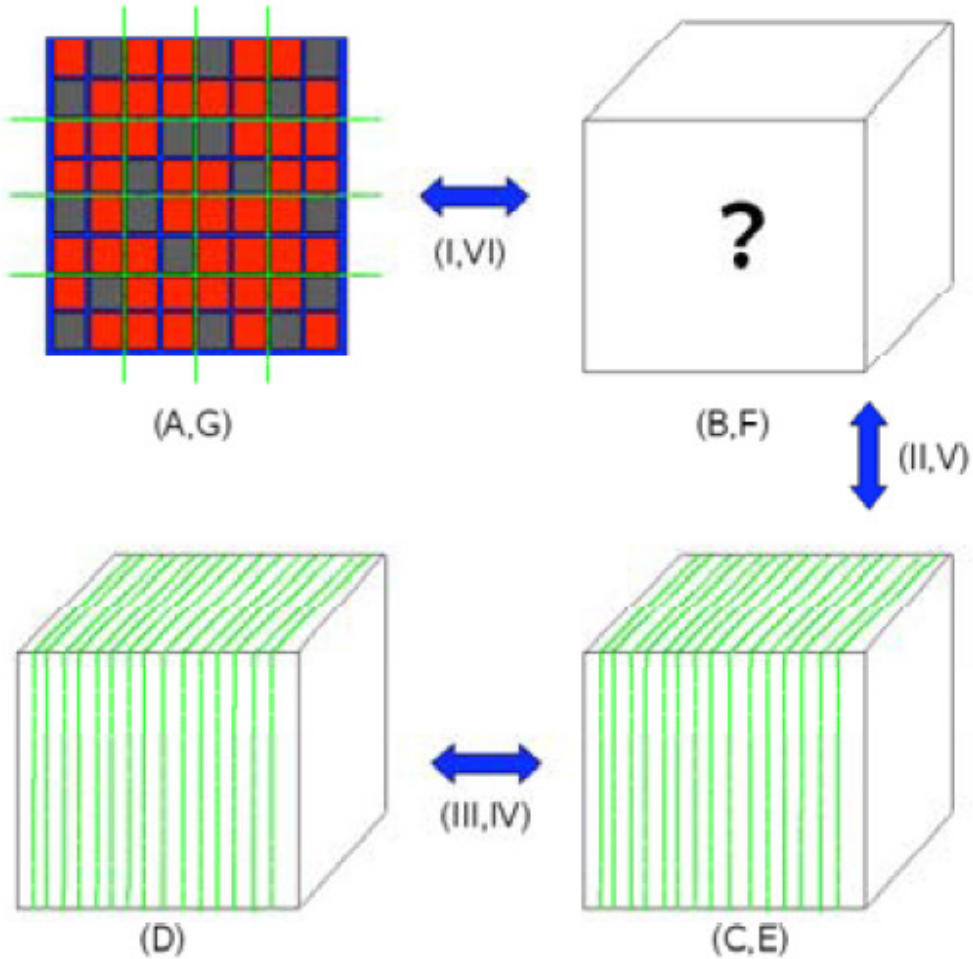
Time and speedup for 2000 3D FFTs using different plan types

- Choice of plan type is exposed to user via GLOBAL%FFTW_PLAN_TYPE input file option
- Default remains FFTW_ESTIMATE

CP2K: Algorithm

- (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)



CP2K: Load balancing

- The sparse matrix representing the electronic density has structure dependent on the physical problem
- For condensed-phase systems atoms are (relatively) uniformly distributed over the simulation cell
- Therefore the work of mapping Gaussians to the real space grid is fairly well load balanced

- What about interfaces, clusters, other non-homogeneous systems?

CP2K: Load balancing

- We used the ‘W216’ test case – a cluster of 216 water molecules in a large (34A^3) unit cell
- Severe load imbalance is encountered (6:1):

```
At the end of the load_balance_distributed
Maximum load:                 1738978
Average load:                  176232
Minimum load:                       0

At the end of the load_balance_replicated
Maximum load:                 1738978
Average load:                  475032
Minimum load:                  286053
```

CP2K: Load balancing

- To address this, a new scheme was used where each MPI process could hold a different spatial section of the real space grid at each (distributed) grid level
- Once the loads on each MPI process were determined (per grid level), underloaded regions would be matched up with overloaded regions from another grid level
- Replicated tasks would be used as before to finely balance the load

CP2K: Load balancing

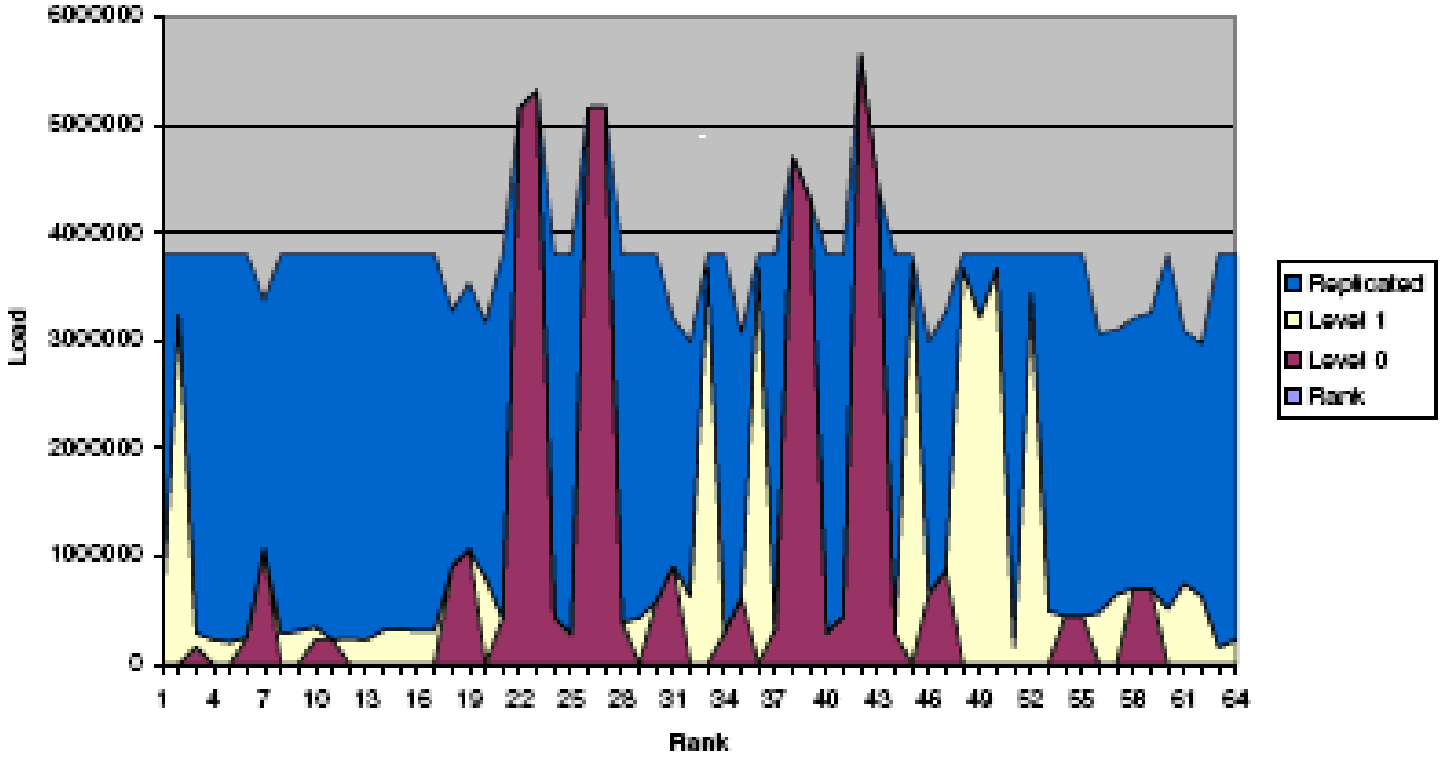
- For the example shown above the load on the most heavily loaded process is reduced by 30%, and there is now a load imbalance of 3:1

```
After load_balance_distributed
Maximum load:                1165637
Average load:                 176232
Minimum load:                      0

After load_balance_replicated
Maximum  load:                1165637
Average  load:                 475032
Minimum  load:                 317590
```

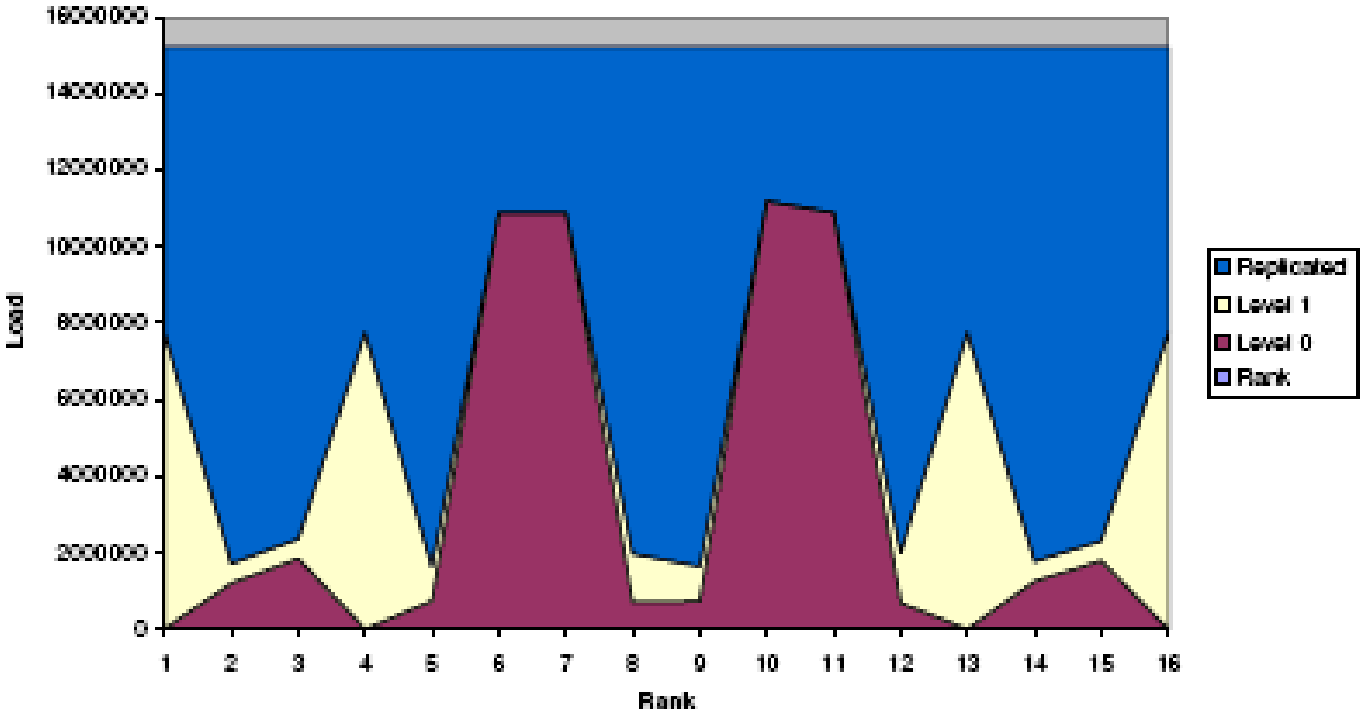
CP2K: Load balancing

- In this case, there are still a single region(s) of one grid level with more total work than the average across all grid levels…



CP2K: Load balancing

- …but if it is possible to balance the load, this method will succeed:



- Can add more closely spaced grid levels (and so decrease the size of the peaks) by decreasing FORCE_EVAL%DFT%MGRID%PROGRESSION_FACTOR

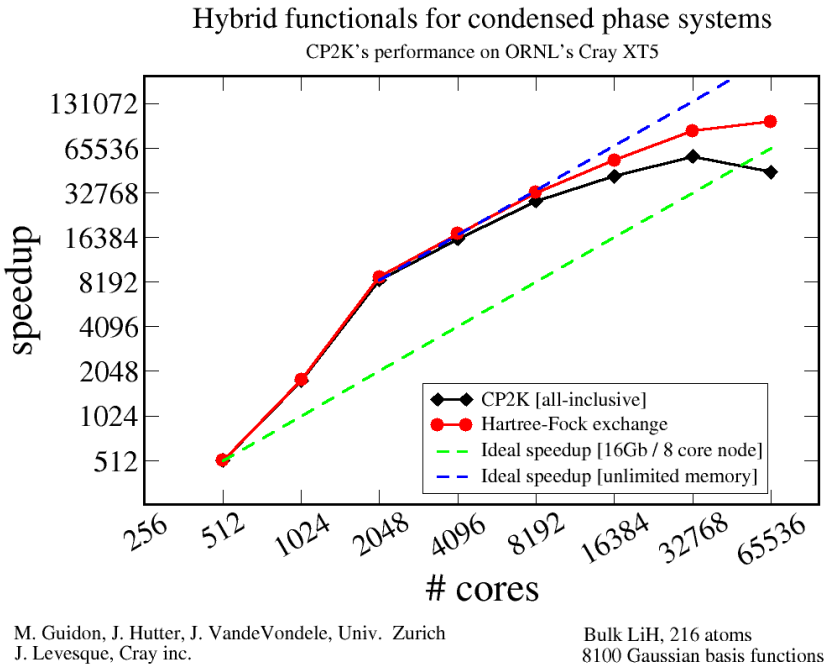
CP2K: Summary

- Overall speedup for bench_64 – 30 % on 256 cores (target was 10-15%)

- Overall speedup for W216 – 300 % on 1024 cores (target was 40-50%)

CP2K: Introducing OpenMP

- Follow-on dCSE Project to implement mixed-mode OpenMP and MPI parallelism (Sep 09 – Aug 10)
- Motivations:
  - extremely scalable Hartree-Fock Exchange (HFX[1]) code uses OpenMP to access more memory per task, and is limited to 32,000 cores by non-HFX part of the code
  - Cray XT architecture going increasingly multi-core -> minimise contention for network access by using OpenMP on node, MPI between nodes



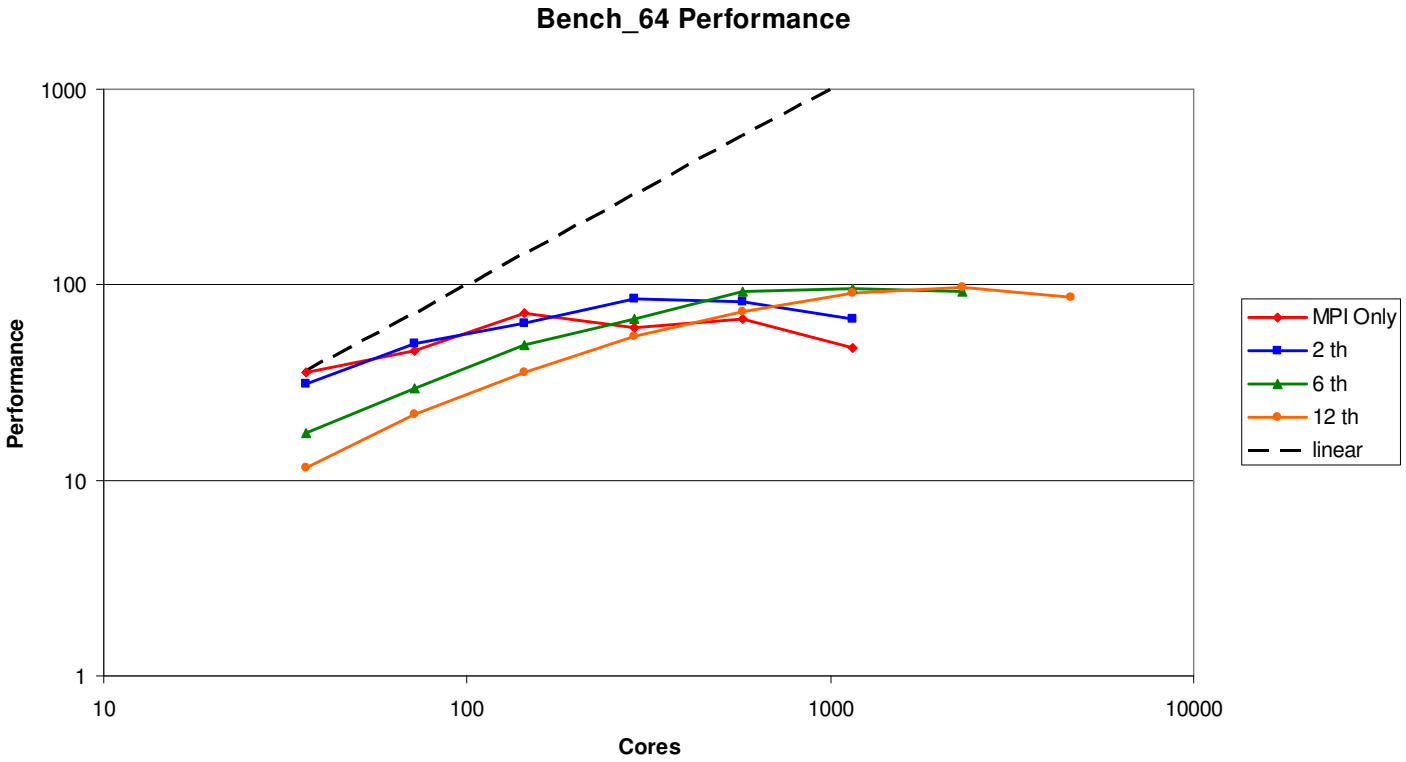
1) M. Guidon, J. Hutter, J. VandeVondele, J. Chem. Theory Compute. 5(11) (2009)

CP2K: Introducing OpenMP

- Taking a simple, targeted approach – OpenMP regions only used in areas of the code that are known to take up the majority of the runtime:
  - rs2pw transfer ✓
  - FFTs ✓
  - Mapping gaussians <-> realspace grids ✓
  - Functional Evaluation ✗ (not yet)

CP2K: Introducing OpenMP

- 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: Introducing OpenMP

- 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: Introducing OpenMP

- Some reasons to use mixed-mode OpenMP/MPI
  - Using multiple threads per task increases scalability by factor of nthreads
  - 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)
  - Benefits should increase as HECToR goes to 24-way multi-core (Phase 2b)
  - 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
  - Able to use OpenMP threads for matrix operations
  - In the code since Autumn 2009

CP2K: Summary

- In the last 2 years, CP2K performance has more than doubled in the 100s of cores region
- Scalability has been extended well into the 1,000s of cores (for smallish systems)
- Demonstrated scalability into the 10,000s of cores (for larger systems, and HFX calculations)

Questions?

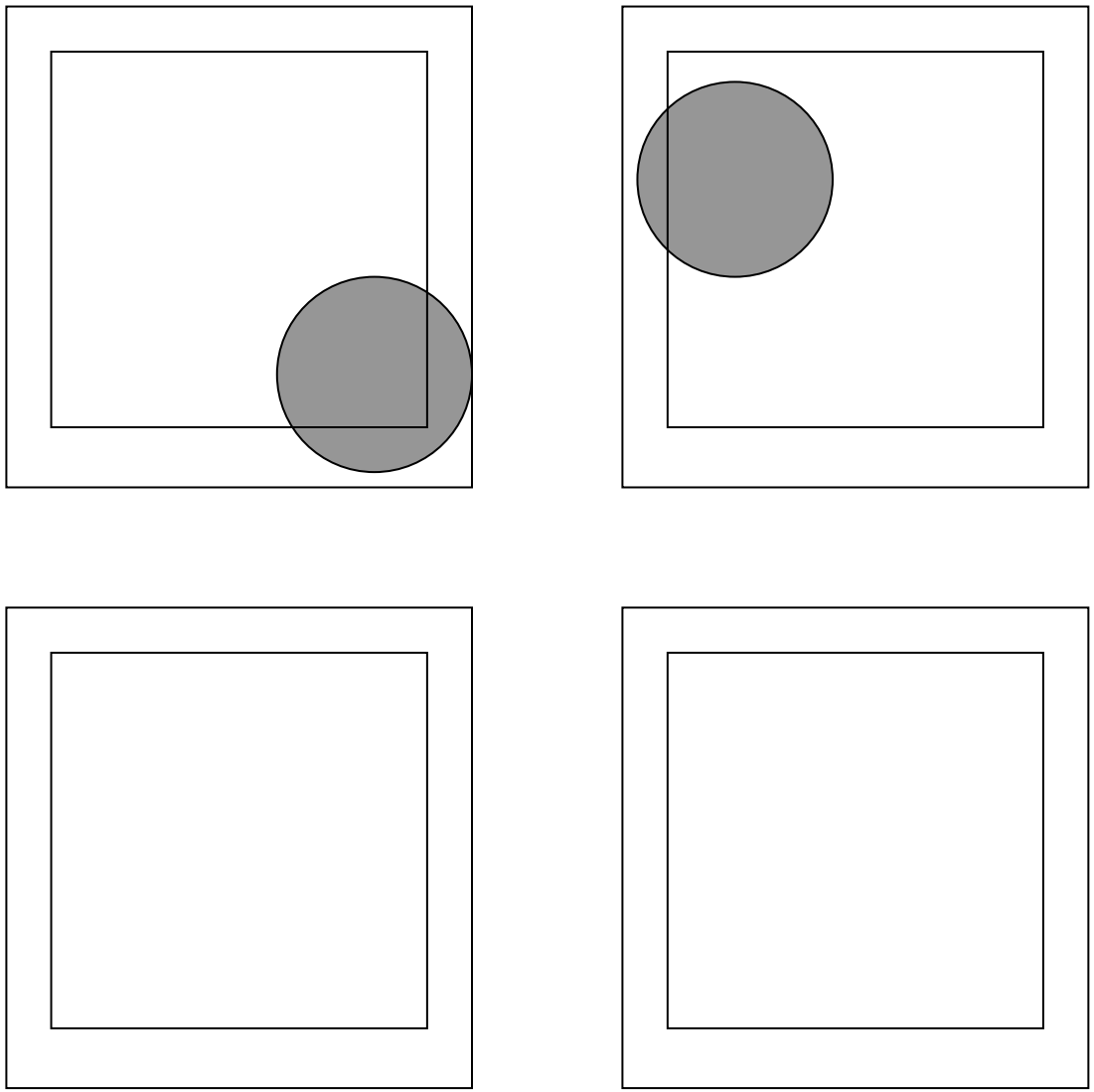
If you are interested in collaborating to improve the performance or functionality of scientific codes, please get in touch!

ibethune@epcc.ed.ac.uk

www.epcc.ed.ac.uk/research-collaborations

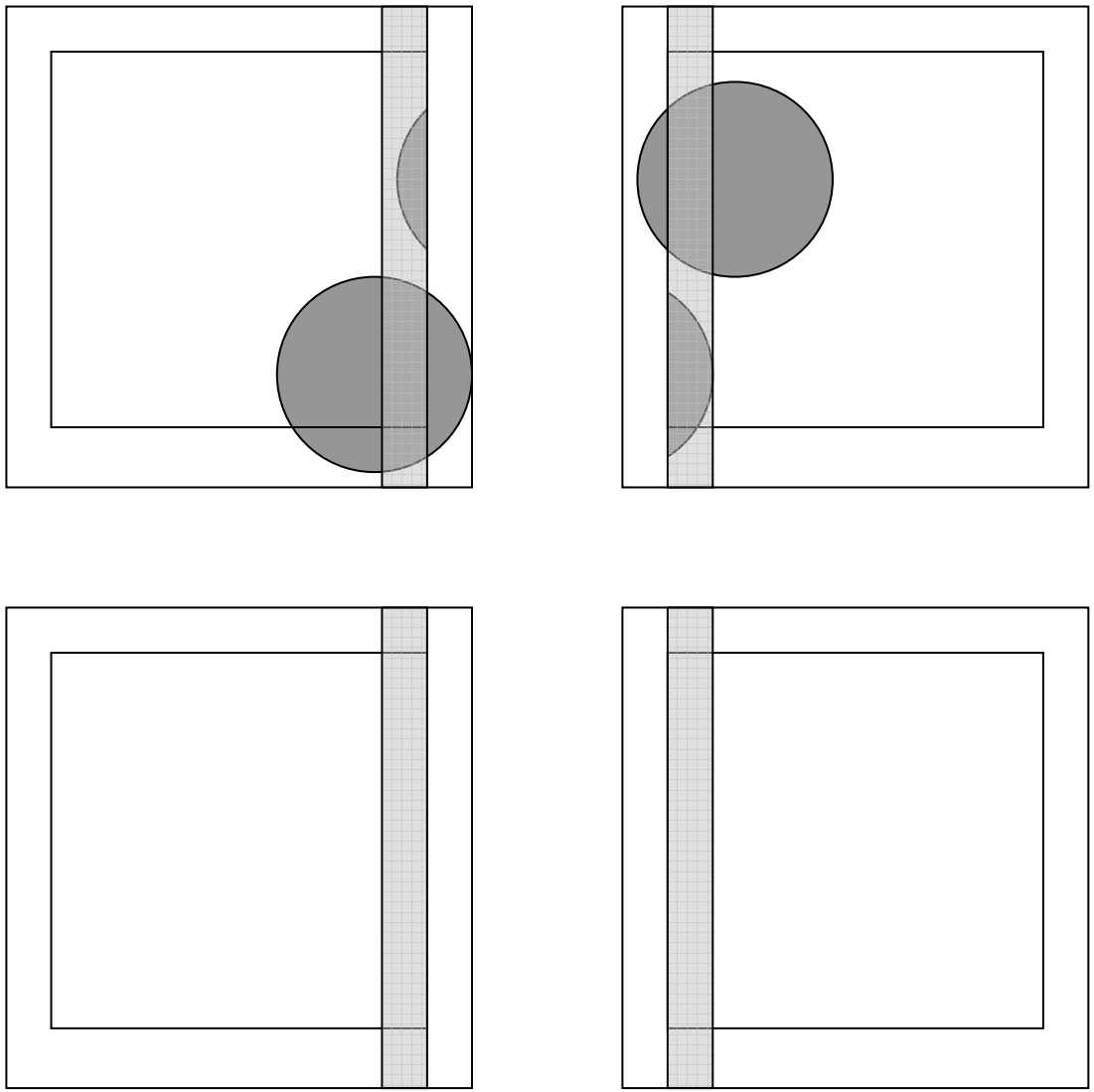
Supplementary slides

CP2K: Realspace to planewave transfer



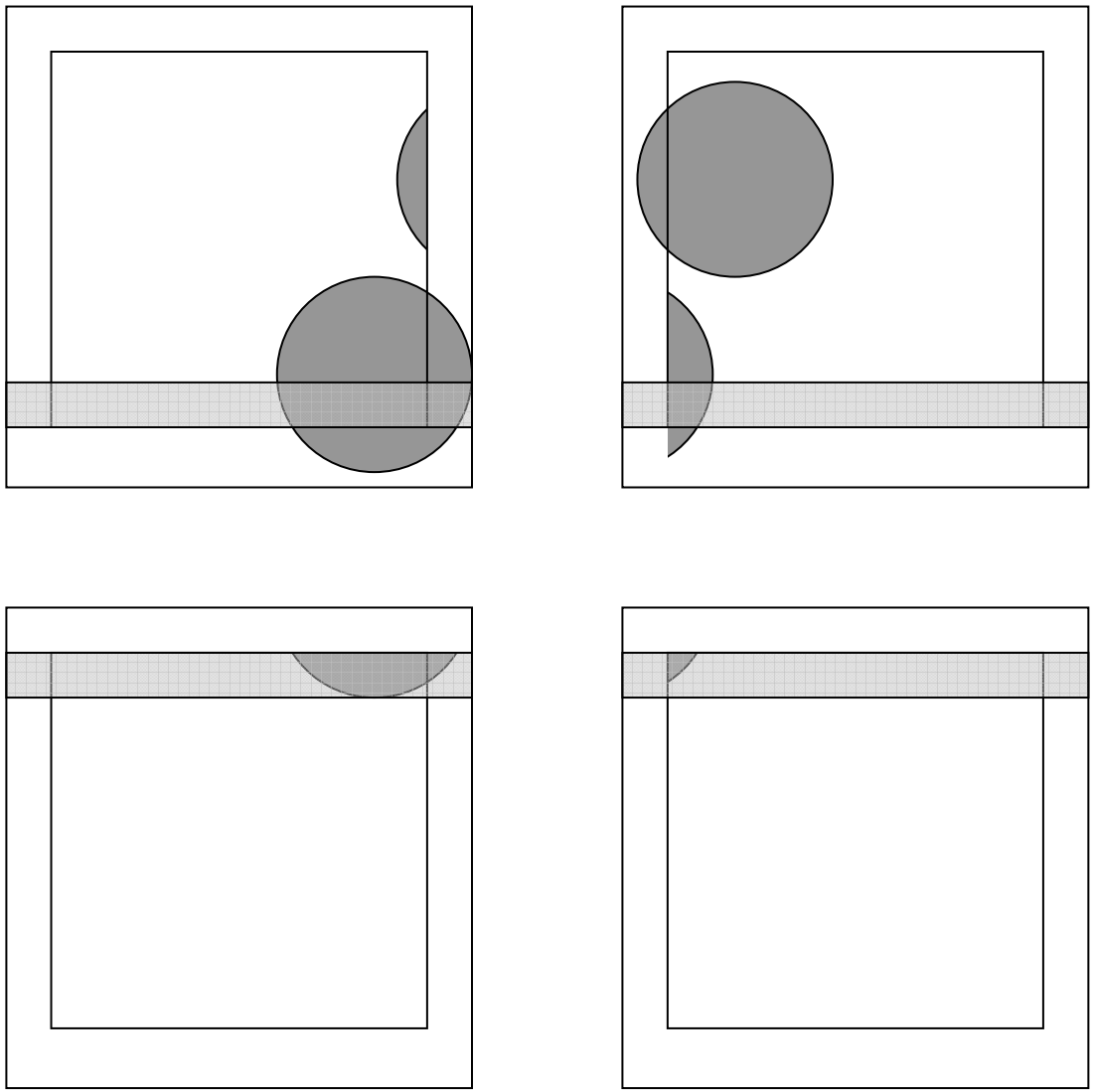
- Step 1 : Gaussians are mapped

CP2K: Realspace to planewave transfer



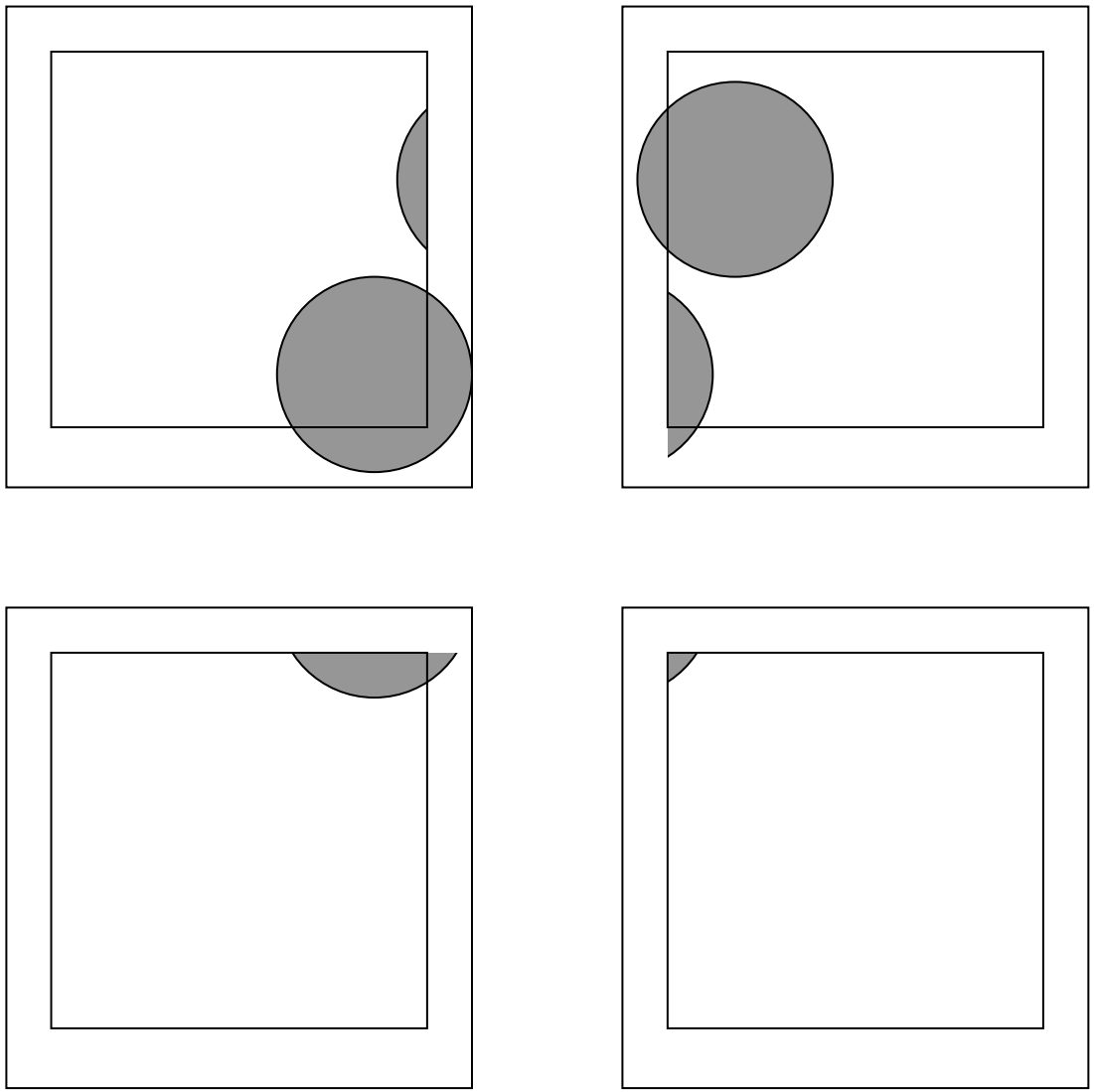
- Step 1 : Gaussians are mapped
- Step 2: Swap halos in X direction

CP2K: Realspace to planewave transfer



- Step 1 : Gaussians are mapped
- Step 2: Swap halos in X direction
- Step 3: Swap halos in Y direction

CP2K: Realspace to planewave transfer



- Step 1 : Gaussians are mapped
- Step 2: Swap halos in X direction
- Step 3: Swap halos in Y direction
- Step 4: Redistribute

CP2K: Load balancing

- The result: 25% speedup on 128 cores, 10% on 1024 cores

Cores	128	256	512	1024	2048
Before(s)	5998	3499	2448	1569	2565
After(s)	4800	2859	2096	1425	2166
Speedup(%)	25	23	16	10	18

Comparison of W216 runtime before and after rank reordering for load balance

CP2K: Fast Fourier Transforms