CP2K: A HECToR dCSE Project

HECToR dCSE Support Technical Meeting

23-24/09/2009

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

CP2K: Contents

- Project Overview
- Introduction to CP2K
- Realspace to planewave transfer
- Fast Fourier Transforms
- Load Balancing
- Summary

CP2K: Project Overview

- A HECToR dCSE Project
  - “Improving the performance of CP2K”
- 6 months effort at 50% FTE (Aug 08 – Jul 09)
- Collaboration with:
  - Slater, Watkins @ UCL (HECToR Users)
  - VandeVondele et al @ PCI, University of Zurich (CP2K Developers)
- Stated aims:
  - 10-15% speedup on 64-256 cores
  - 40-50% speedup on 512-1024+ cores

CP2K: Introduction to CP2K

- CP2K is a freely available (GPL) Density Functional Theory code (+ support for classical, empirical potentials) – can perform MD, MC, geometry optimisation, normal mode calculations…
- Developed since 2000, many developers migrated from the CPMD project – mainly based in Univ Zurich / ETHZ / IBM Zurich
- 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!

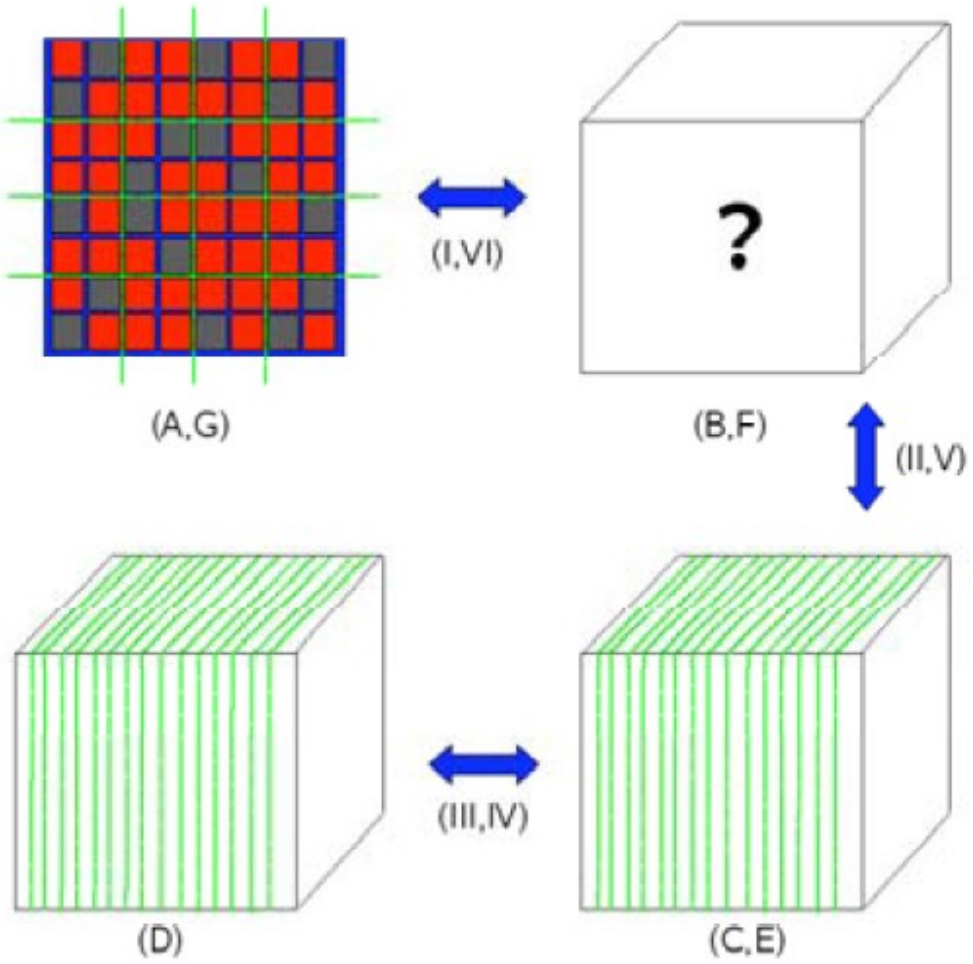
CP2K: Introduction to CP2K

- The Gaussian basis results in sparse matrices which can be cheaply manipulated e.g. diagonalisation during SCF calculation.
- The planewave basis (relying on FFTs) allows easy calculation of long-range electrostatics.
- 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: Introduction to CP2K

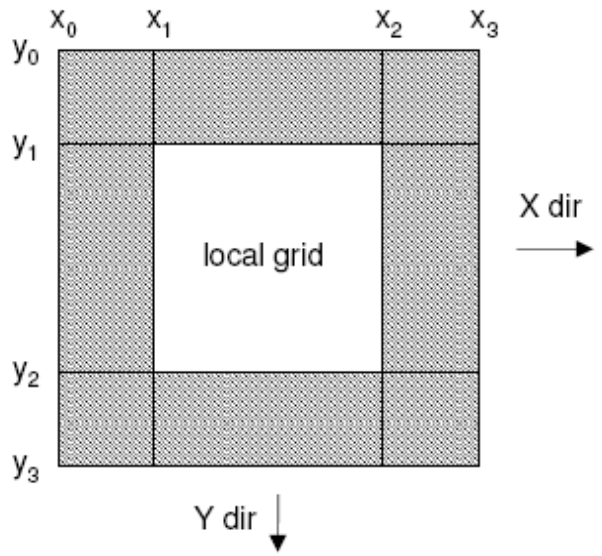
- (A,G) – distributed matrices
- (B,F) – realspace multigrids
- (C,E) – realspace data on planewave multigrids
- (D) – planewave grids

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



CP2K: Realspace to planewave transfer

- Gaussians are mapped by the ‘owner’ of the corresponding real space grid – but they may extend over the boundaries of this region, so a halo region is necessary



- Halos are swapped to ensure each process has all the contributions from all gaussians which overlap its local grid.
- Data is then redistributed onto planewave grids by MPI_Alltoallv

CP2K: Realspace to planewave transfer

- In a conventional halo swap (e.g. distributed 5-point stencil algorithms) the edges of the core region of a process are copied into the halos of the neighbouring processes, which need it for the next step of calculation
- 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

CP2K: Realspace to planewave transfer

- Optimisation:
  - Swapping the full width of the halo in all three directions is unnecessary – only the data that will end up in a core region matters
  - In fact, the halo regions are much larger than shown (e.g. for a 125^3 grid on 512 processors, the core region is 16x16x16, but the halo width is 18)
  - CrayPAT timing with regions showed that the buffer packing for the ‘X’ swap was most expensive, followed by ‘Y’, then by ‘Z’ even if the halos were the same size – this is due to the data lying contiguously in memory for the ‘Z’ swap
  - Performing the swap in the Z,Y,X direction, and reducing the size of halo sent each time gave a 100% speedup for this routine

CP2K: Realspace to planewave transfer

	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: Realspace to planewave transfer

- 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

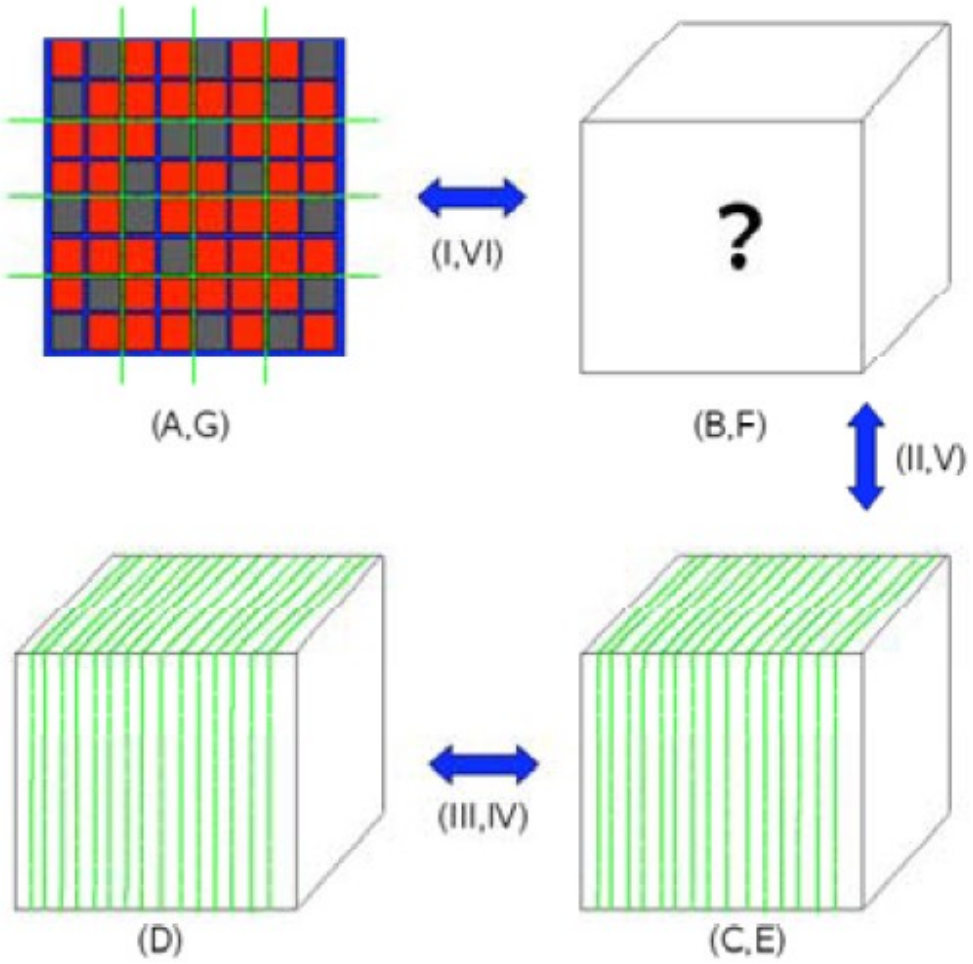
Comparison of bench_64 runtime before and after rs2pw optimisation

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

CP2K: Introduction to CP2K

- (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

- CP2K already has a data structure `fft_scratch` which stores buffers, coordinates etc. for reuse
- The MPI sub-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

CP2K: Fast Fourier Transforms

- Introduced a new type to the `fft_scratch` for storing library-dependent data
- Implemented storage and re-use of plans for FFTW 2 and 3 – for other libraries planning is a no-op
- This allowed the more expensive plan types to used…

CP2K: Fast Fourier Transforms

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

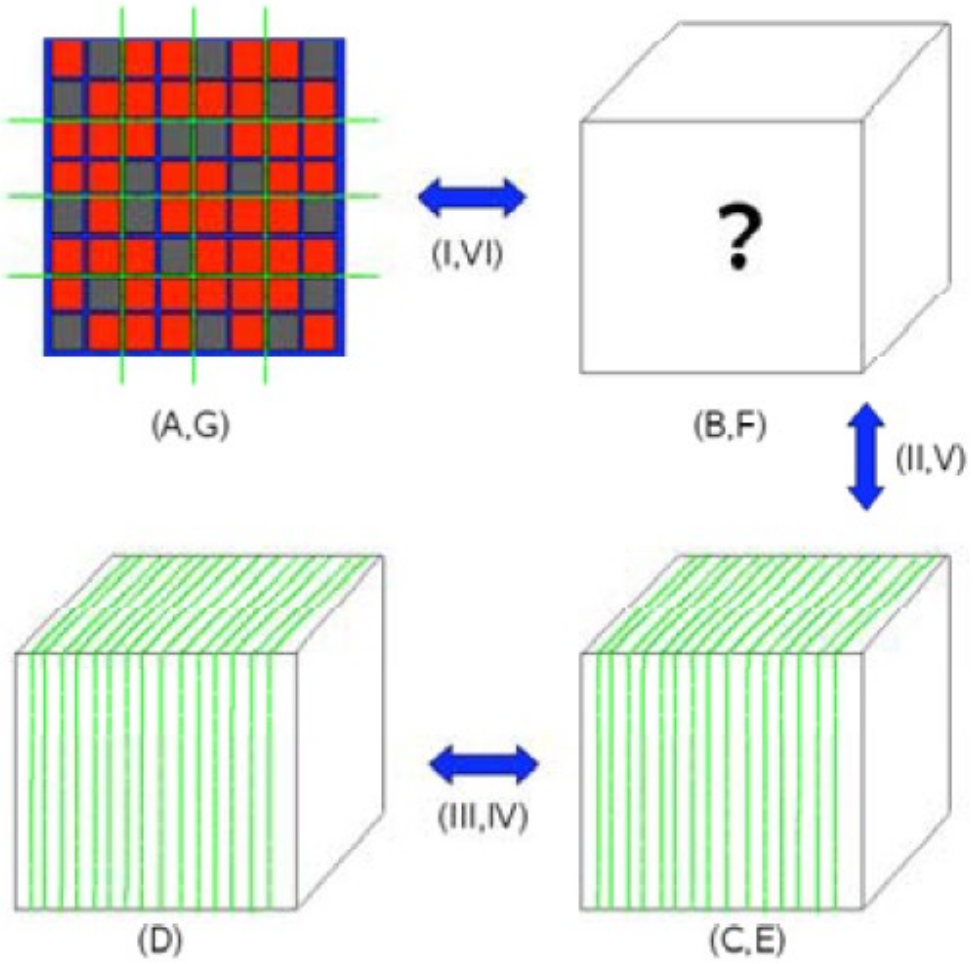
Time and speedup for 2000 3D FFTs using different plan types

- Choice of plan type is left up to the user and exposed as an option in the input file, defaulting to FFTW_ESTIMATE

CP2K: Introduction to CP2K

- (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

CP2K: Load balancing

- The existing load balancing scheme uses ‘tasks’ belonging to the replicated grid levels to load balance – these can be mapped by any process:

```
At the end of the load_balance_distributed
Maximum load:                  75667
Average load:                  68312
Minimum load:                  13060

At the end of the load_balance_replicated
Maximum load:                 123552
Average load:                 123457
Minimum load:                 123374
```

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

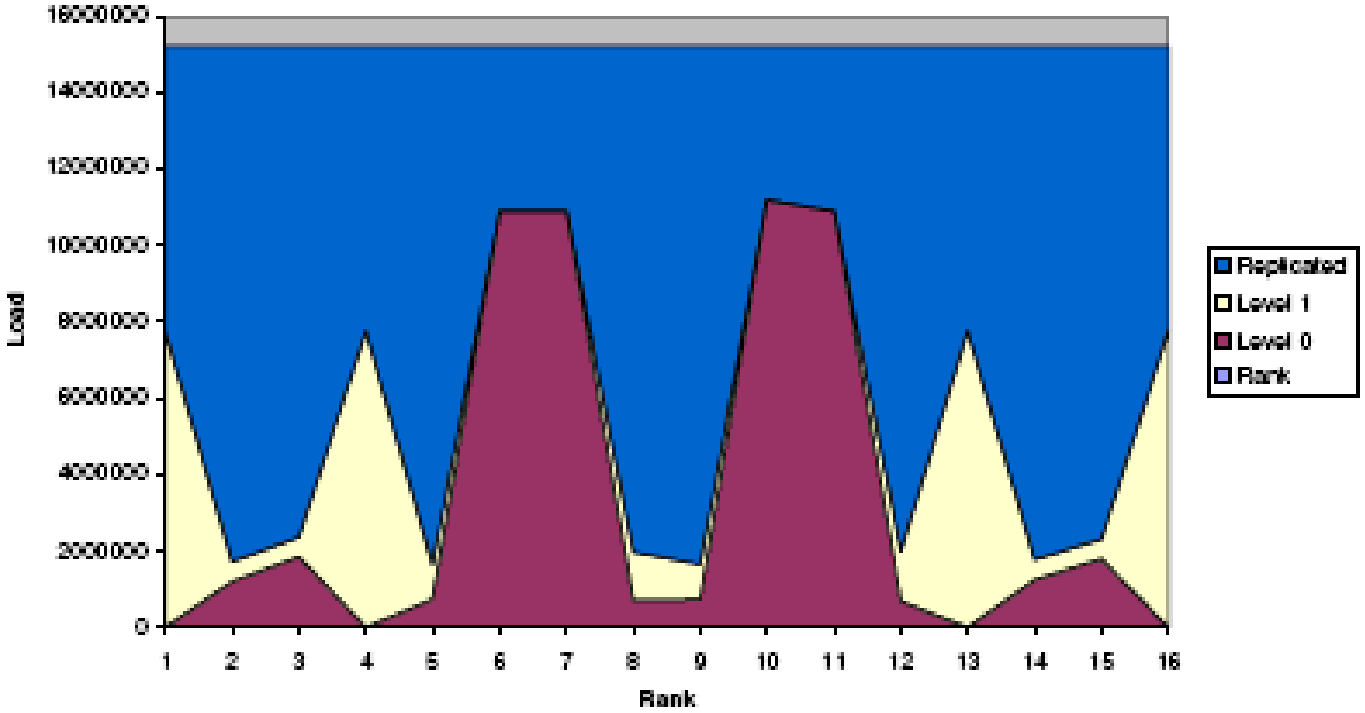
- For the example shown above the load on the most heavily loaded process is reduced by 30%, and there is now a load imbalance of 3:1

```
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
```

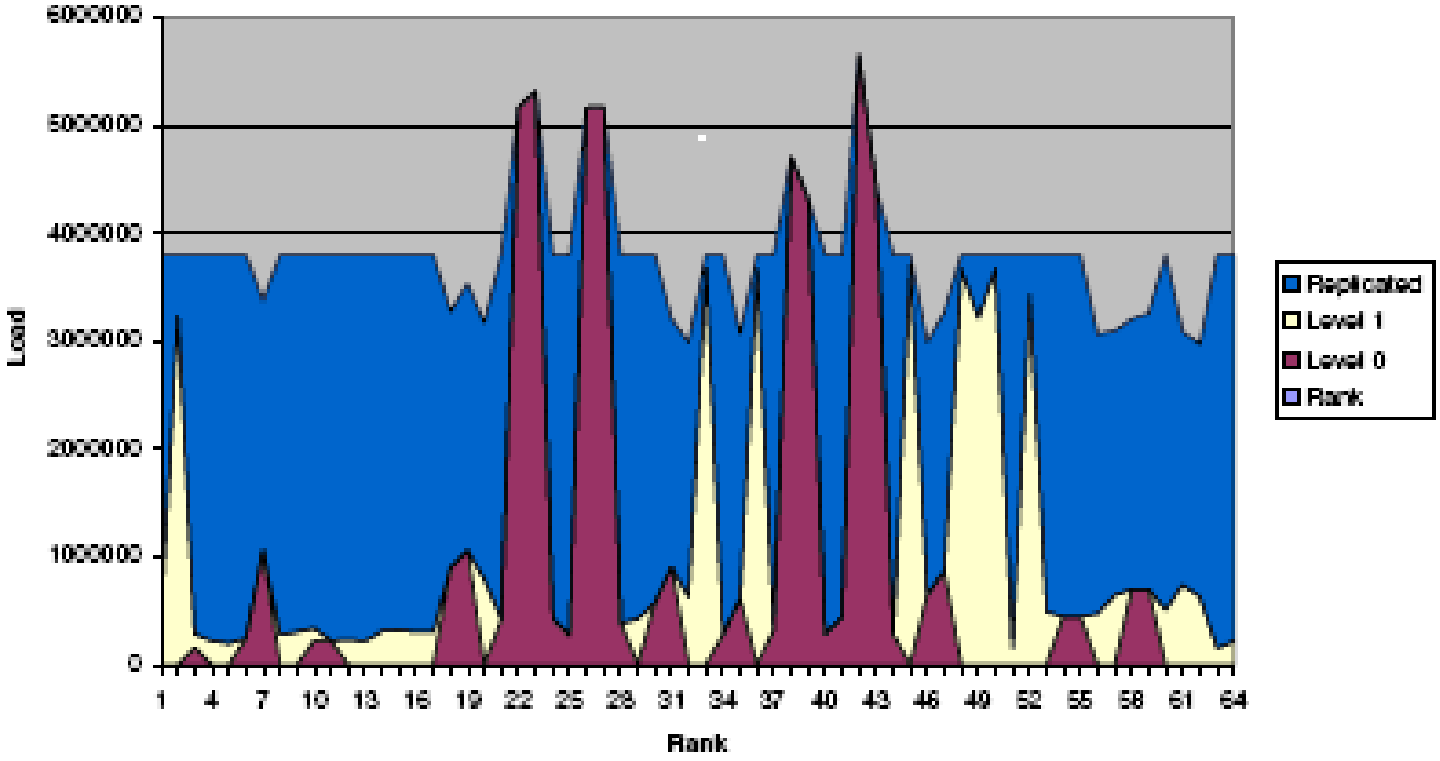
CP2K: Load balancing

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



CP2K: Load balancing

- But if there is a single region with load from one grid level larger than the average load then we still have some imbalance:



CP2K: Load balancing

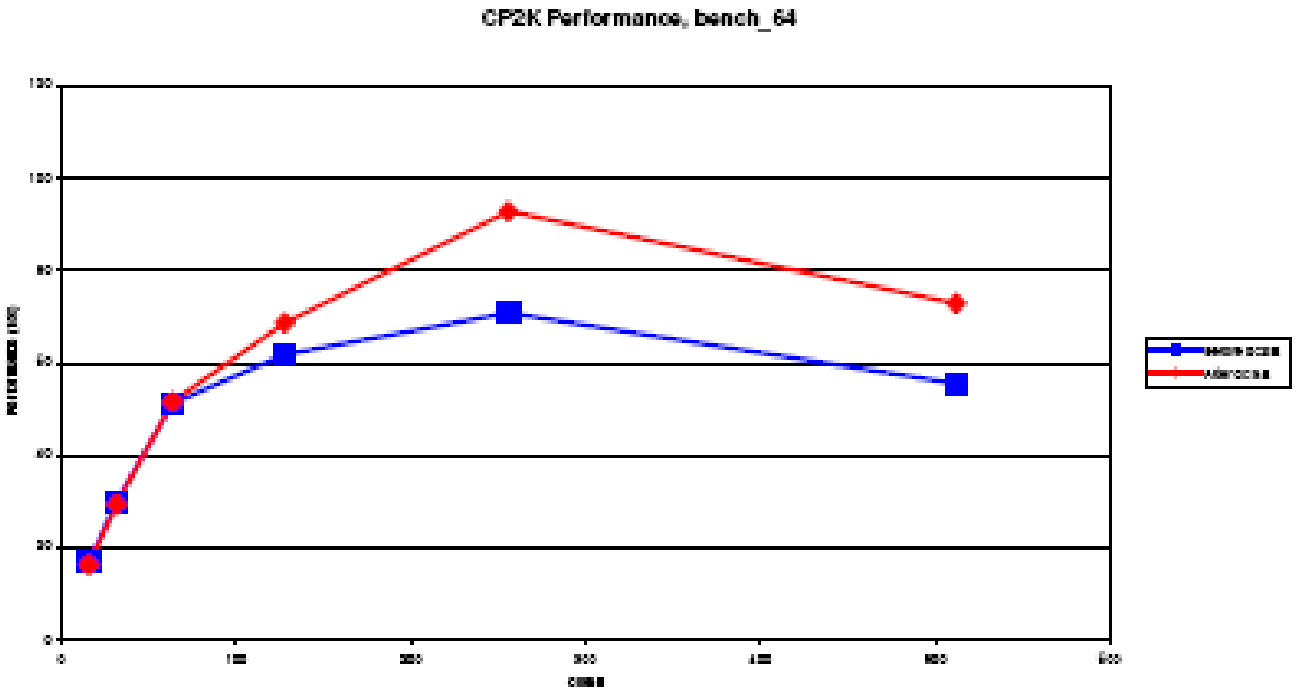
- The result: 25% speedup on 128 cores, 10% on 1024 cores

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

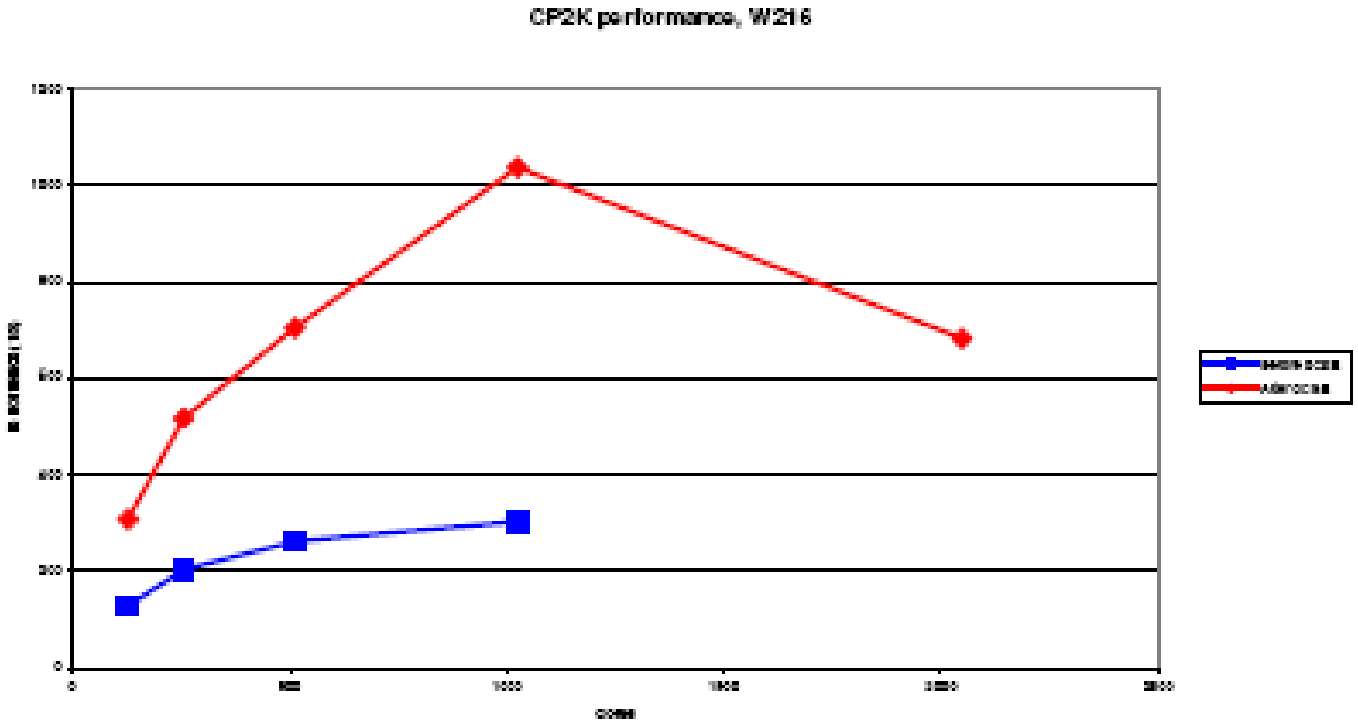
CP2K: Summary

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



CP2K: Summary

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

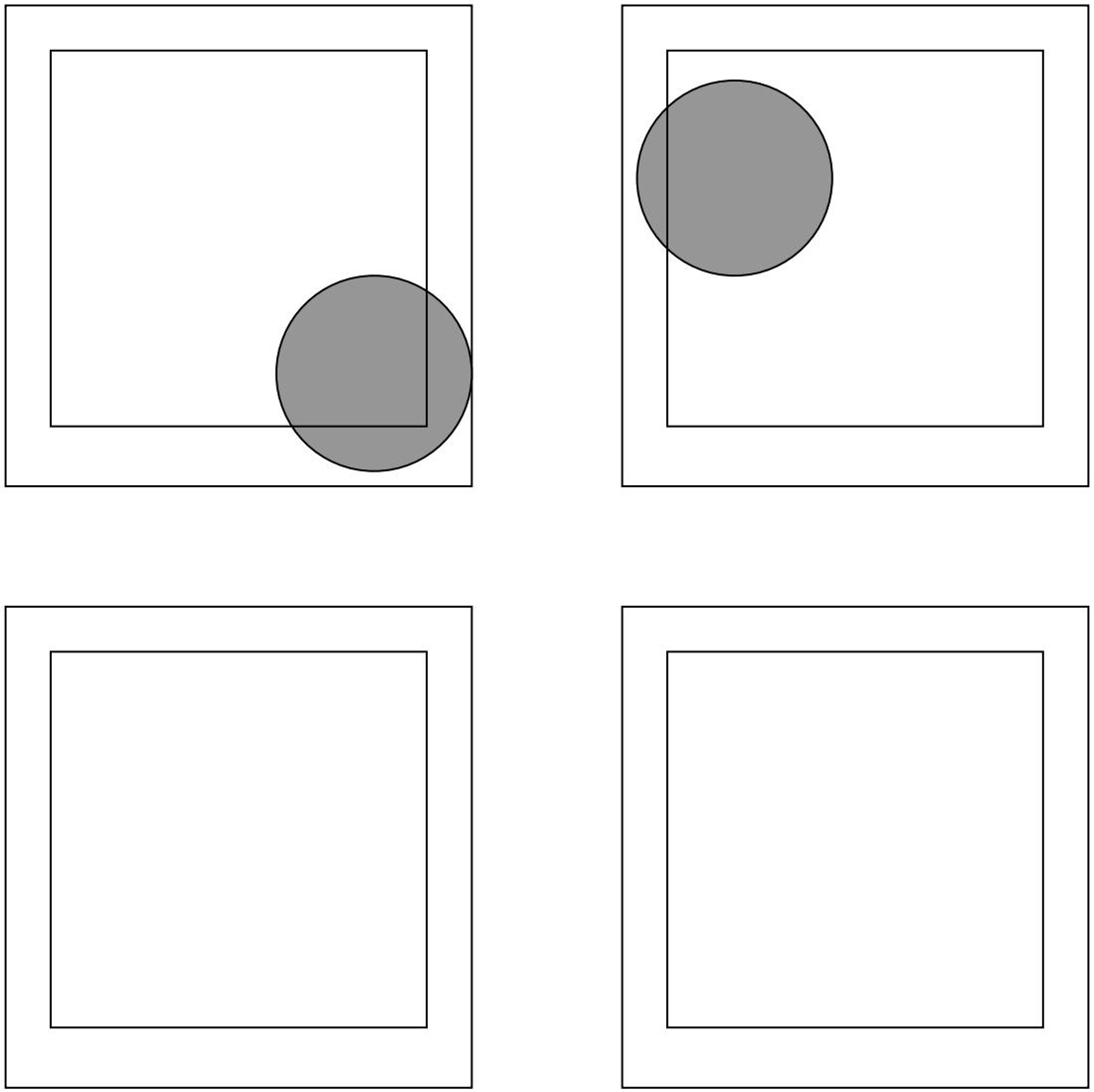


CP2K: Summary

- Project achieved the stated aims and more…
- Improvements are in CVS and in use on HPCx and HECToR
- NAG have funded an additional 6 months of dCSE support to implement hybrid OpenMP/MPI and address other bottlenecks

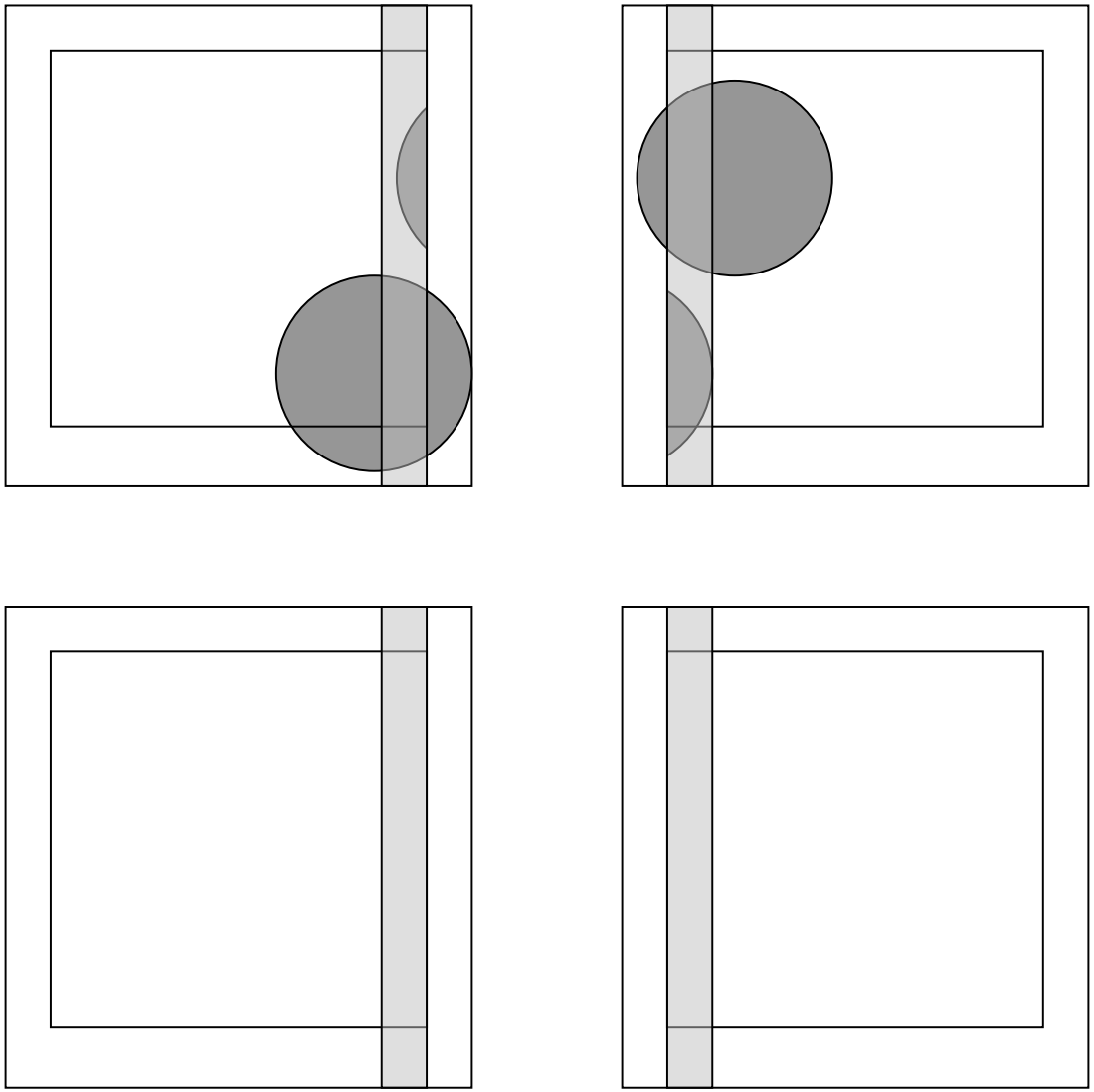
- Questions?

CP2K: Realspace to planewave transfer



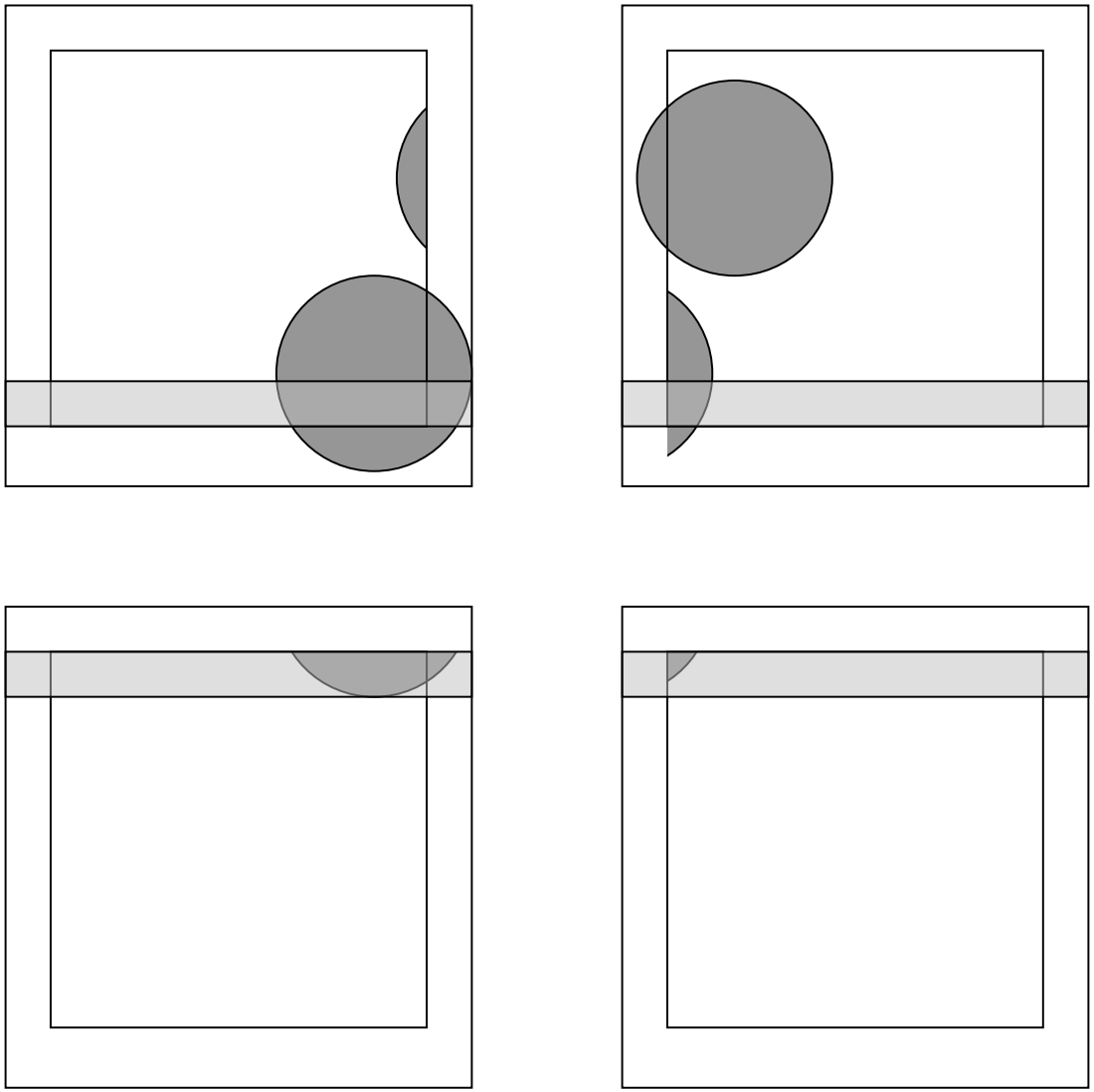
- Step 1 : Gaussians are mapped

CP2K: Realspace to planewave transfer



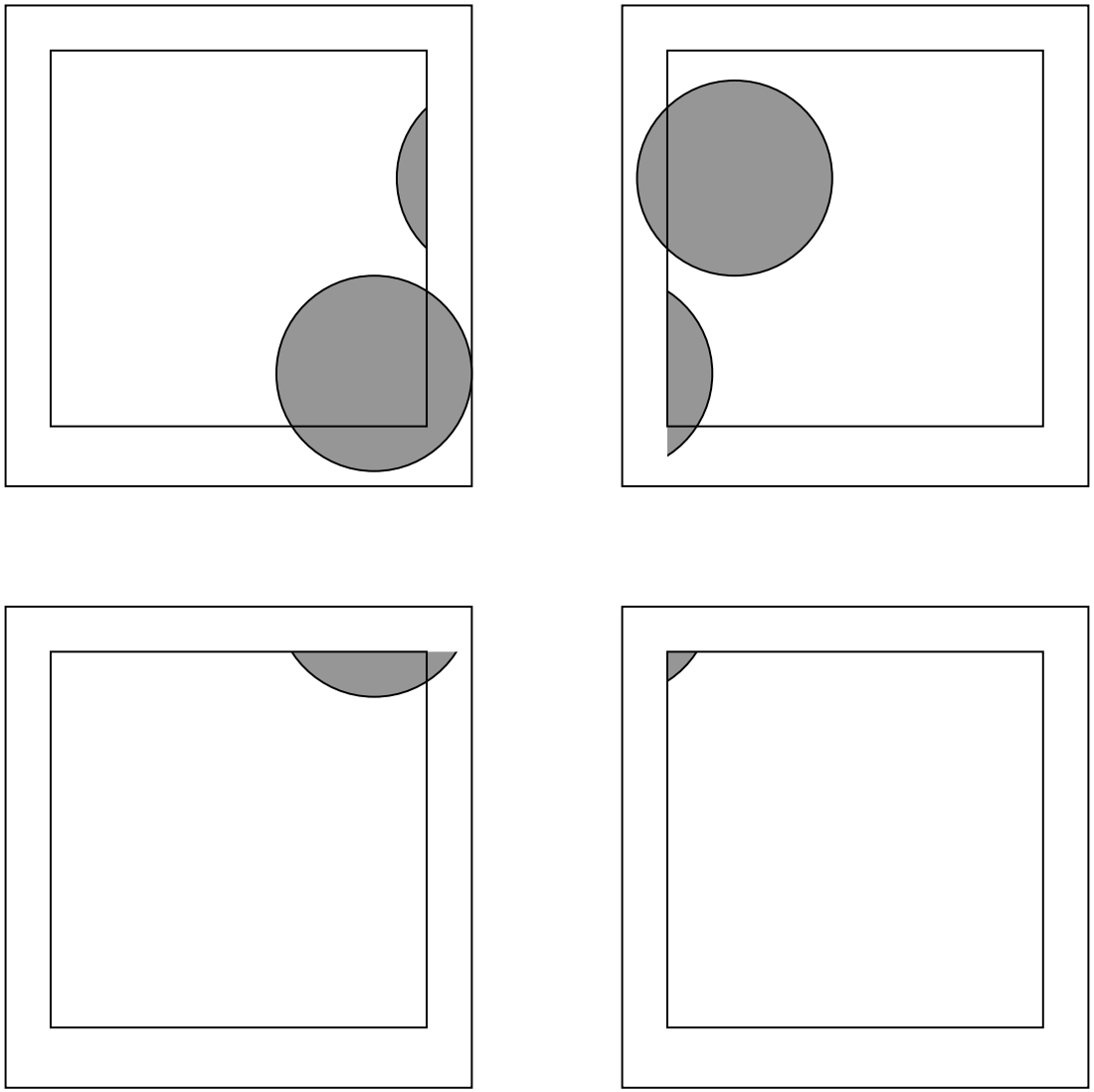
- Step 1 : Gaussians are mapped
- Step 2: Swap halos in X direction

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: 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!

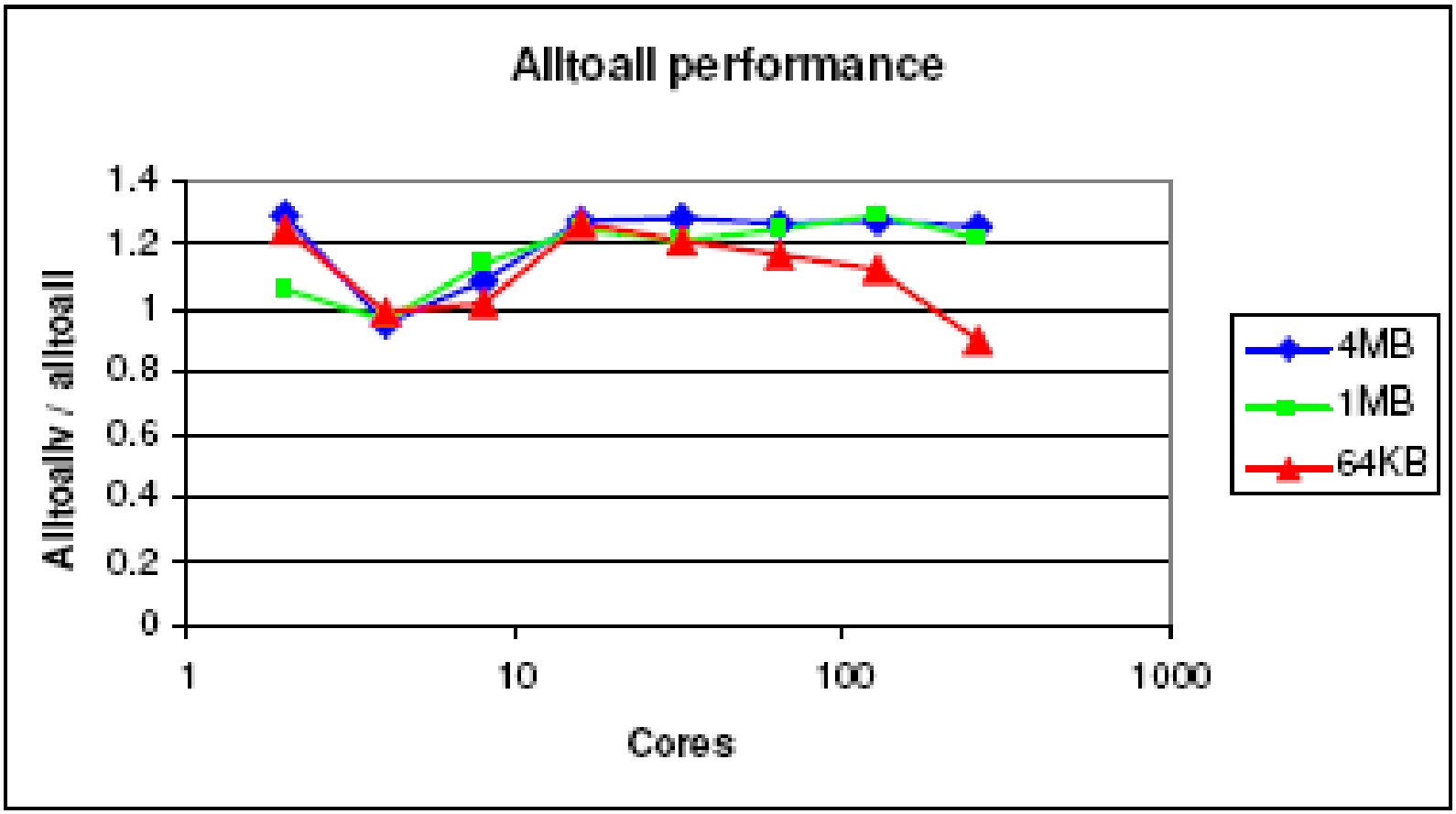
```
 Time % |      Time |Imb. Time |   Imb. |    Calls |Group
        |           |          | Time % |          | Function
        |           |          |        |          |  PE.Thread='HIDE'

 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

- MPI_Alltoallv is used for the transpose steps
- However, data is distributed evenly such that with a little padding we could use MPI_Alltoall
- This should give a 20-30% speedup as measure by Intel/ Pallas MPI benchmark

CP2K: Fast Fourier Transforms



CP2K: Fast Fourier Transforms

- In practise, only a 2% improvement was gained due to poor synchronisation
- But the code was not added to CVS due to the extra complexity of book-keeping code and buffer padding