Improving the performance of CP2K on HECToR
A dCSE Project

I. Bethune
EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EH9 3JZ, UK

July 29, 2009

Abstract

This report presents the results of a HECToR dCSE project to improve the performance of CP2K, a freely available and popular Density Functional Theory code, on HECToR. Building on a recently implemented domain decomposition method, further optimisation of the code was performed, and significant performance gains were measured - around 30% on 256 cores (for a generally representative benchmark) and up to 300% on 1024 cores (for non-homogenous systems).

Detailed profiling of the code was also carried out, which has highlighted further opportunities to improve the performance of the code.

Contents

1 Introduction

Following on from prior work by Dr. Matt Watkins, a member of Dr. Ben Slater's research group at University College London, an application for dCSE support was made by Dr. Slater and Dr. Joost VandeVondele of the University of Zurich to further optimise the code for HECToR. 6 months of development effort was provided by Iain Bethune, an Applications Consultant at EPCC, between August 2008 and July 2009. All the objectives set out in the proposal were acheived, and the performance gains achieved are described in detail below.

1.1 CP2K

CP2K [1] is a freely available density functional theory (DFT) package, licensed under the GPL. The code is developed by a number of individuals and groups, and currently numbers over 50,000 lines of Fortran 95 code. Key developers include members of Prof. Jurg Hutter's group in the Physical Chemistry Institue of the University of Zurich, in particular Dr. VandeVondele who collaborated closely throughout the project.

The key difference between CP2K and other DFT codes is its implementation of the Quickstep algorithm, which uses a dual basis - atom-centred Gaussian functions to represent the wave-functions, and plane waves/regular grids for the electonic density. See the Quickstep paper [2] for full details of the algorithm. In practise this approach requires an efficient and scalable method for transforming between the two representations, which is the subject of most of the work in this project. Like most other DFT codes, CP2K also makes use of 3D Fast Fourier Transforms (FFTs) to convert from the real space to reciprocal space (plane wave) representations.

As CP2K is under constant development, the modifications made under this project were able to be committed to the main CVS repository directly after passing the regression test suite and are already available for users to download. A recent version of the code is installed as a 'package' on HECToR and HPCx.

1.2 HECToR

This dCSE project was carried out on HECToR, the UK National Supercomputing service. Full details of the system are available on the website [3]. During the project HECToR was in it's 'Phase 1' configuration - 5664 Cray XT4 nodes, each containing an AMD 2.8GHz dual-core Opteron processor with 6GB RAM shared between the two cores. However, the work is expected to also benefit the many other architectures that CP2K runs on, including other HPC systems based on Intel, AMD, and Power processors, as well as desktop and commodity cluster systems.

HECToR also contains an X2 Vector Unit consisting of 28 nodes with 4 vector processors per node. This part of the system was not used during this project.

2 Compiling CP2K for HECToR

At a minimum, CP2K requires a Fortran 95 compiler, BLAS and LAPACK libraries in order to compile, and an MPI-2 implementation and ScaLAPACK library to build a parallel version. Optionally, FFT libraries, and the the libint[4] library, required for Hartree-Fock exchange (HFX) calculations, can also be used. There is a single Makefile, and the code comes with over 100 'arch' files which customise the make procedure to a particular system. By customising the existing `CRAY-XT5.popt` arch file, the following settings were used, and saved as `CRAY-XT-CNL-pathscale.popt`:

```
CC       = cc
CPP      = cpp
FC       = ftn -freeform
LD       = ftn
AR       = ar -r
DFLAGS   = -D__PATHSCALE -D__XT5 -D__FFTW3 -D__FFTSG -D__LIBINT \
           -D__parallel -D__BLACS -D__SCALAPACK
CPPFLAGS = -traditional -C $(DFLAGS) -P $(FFTW_INCLUDE_OPTS)
FCFLAGS  = -O3 -OPT:Ofast
LDFLAGS  = $(FCFLAGS)
LIBS     = -L/work/z01/z01/ibethune/cp2k/libint/lib \
           /work/z01/z01/ibethune/cp2k/cp2k/libint_tools/libint_cpp_wrapper.o \
           -lderiv -lint -lstdc++

OBJECTS_ARCHITECTURE = machine_xt5.o
MODDEPS = no
```

To build the code the following was used:

```
module swap PrgEnv-pgi PrgEnv-pathscale
make ARCH=CRAY-XT-CNL-pathscale VERSION=popt
```

On HECToR, the linear algebra (BLAS, LAPACK, BLACS, ScaLAPACK) are provided in the libsci library which is loaded and linked by the default module environment. FFTW 3 is also available on HECToR and is found to give very good performance (see section 4). The `FFTSG` is an in-built set of FFT routines that can be used as a fallback where no FFT library is available on the system. Here, libint is built and linked in, but could easily be omitted by removing the `-D__LIBINT` definitation and the libraries in the `LIBS` variable.

2.1 Benchmarks

CP2K has many different uses including molecular dynamics, geometry optimisation, normal mode analysis, free energy calculations, path-integral runs, and Monte Carlo, using a variety of force evaluation methods such as Quickstep DFT, MM, and QM/MM. For the purposes of this project two benchmark systems were used which were considered representative of the type of simulations performed by many CP2K users and that exercised the major components of the Quickstep algorithm.

The first benchmark, "bench_64" is a molecular dynamics simulation of 64 water molecules in a cubic, periodic cell of side 12.42Å, described by a TZV2P basis set (40,000 functions) in an NVE ensemble at a temperature of 300K. The system is evolved for 50 timesteps of 0.5fs each. This simulation takes a few minutes to complete on 256 cores.

The second benchmark, "W216", is a larger system of 216 water molecules, in a 34Åcell, using a larger, molecularly optimized basis[5]. However, this system is not periodic, and the atoms are clustered in the centre of the simulation cell, so gives interesting load balancing properties which are investigated in the third part of the project. A run of 10 0.5fs MD steps takes around 20 mins on 1024 cores. This system is only slightly smaller than those being studied by real CP2K users, although typical runs would be of the order of 1000s of MD steps.

CP2K also includes a set of 'libtests' which isolate particular features of the code, rather like a unit test harness. Libtests exist for "RS_PW_TRANSFER" (realspace to planewave grid conversion), and "PW_TRANSFER" (FFTs) which were used extensively for checking correctness and measureing performance during development.

2.2 Compiler Performance

As CP2K is very portable, performance was investigated using the 3 main compiler suites available on HECToR - Portland Group (PGI), Pathscale, and GNU gfortran. For each of these compilers, a reasonably aggressive set of optimisation settings were chosen, and the bench_64 case was run on 64 cores, where computation dominates the communication cost. The following results were obtained:

Compiler	Optimisation flags	Time(s)
PGI 8.0.2	-fast	337s
Pathscale 3.1	-O3 -OPT:Ofast	318s
gfortran 4.3.2	-O3 -ffast-math -funroll-loops -ftree-vectorize	335s

Table 1: Comparison of HECToR compilers, using bench_64

As Pathscale was significantly faster than the other two compilers it was used for all benchmarking runs. However, gfortran was used for development as it was able to compile the code much more quickly than either Pathscale or PGI.

The compiler versions indicated in table 1 are all able to compile CP2K correctly. Earlier versions of the PGI compiler are known to cause numerical issues at runtime and so should be avoided.

3 Realspace to Planewave transfer

The first major work item in the project was to optimise the `rs2pw_transfer` routine (steps II and V in figure 1). As mentioned earlier, CP2K maintains two distinct bases - products of atom-cented gaussian functions to repesent the wave-function (stored as a sparse matrix), and planewaves (stored as distributed regular grids). In common with other DFT codes, CP2K performs a Self-Consistent Field (SCF) loop to find the ground state energy of the system. Each iteration therfore requires to convert back and forth between these two representations via the following steps (illustrated in figure 1):

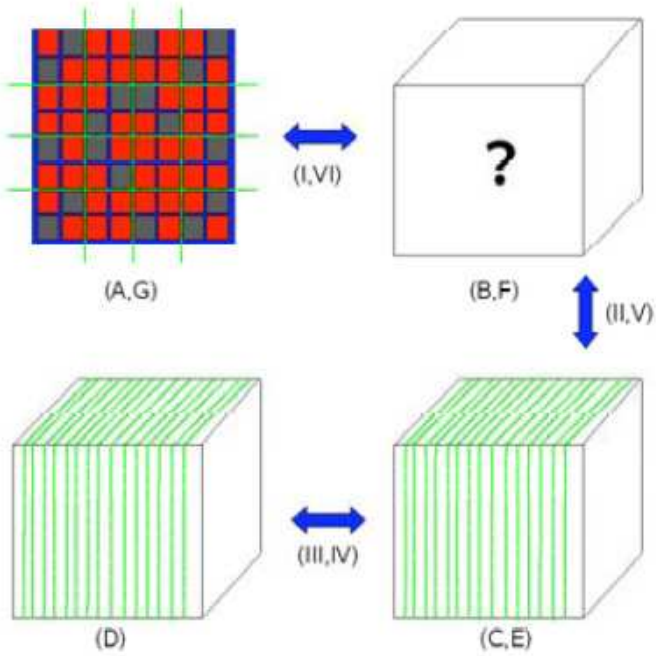

Figure 1: Transformation from gaussians to plane waves via real space grids

1. Steps I,VI: The matrix elements corresponding to products of gaussian functions are mapped onto a set of realspace multi-grids. Each level of the multi-grids may be fully distributed, so each MPI task is responsible for mapping the subset of gaussians which are centred on it's particular section of the grid, or replicated, in which case any task may map any given gaussian. These properties are important for the load balancing scheme described in section 5. This step is referred to as 'collocation', and when done in reverse 'integration'.

2. Steps II,V: The data stored on the realspace multi-grids is then transferred to a set of planewave multi-grids. The planewave grids in general are distributed differently to the realspace grids (either in slices or pencils, in order to suit a parallel FFT), so this step involves using MPI_Alltoallv to globally reshuffle the data into the right place on the distributed planewave grids. In addition to this there is a 'halo-swap' step required (before the redistribution for the realspace-to-planewave direction and afterwards for the planewave-to-realspace direction). This

is necessary because when a gaussian function is mapped to the grid, it may extend beyond the boundaries of grid which are local to the MPI task that performed the mapping. Therefore every process maintains a halo region which is wide enough to accommodate the largest possible gaussian, and after mapping is complete, these halo regions are swapped with the neighbours and summed into their local grid.

3. Steps III,IV: Once the data is on the planewave grids, a parallel 3D Fourier Transform is done to move from a realspace to recriprocal space representation. Further detials of the FFT algorithm is in section 4.

3.1 Halo Swapping

The halo swap step can be quite expensive, particularly as the number of MPI tasks are increased. For a fixed grid size, the amount of local grid space decreases as the grid is divided between more processes. The halo width, however, is fixed by the maximally-sized gaussian function, and so the halos grow proportionally larger compared to the local domain as more processors are added. For example, a 125^3 grid on 512 processors has a local domains size of 16x16x16, and a halo width of 18. As a result the halos make up 97% of the total grid data.

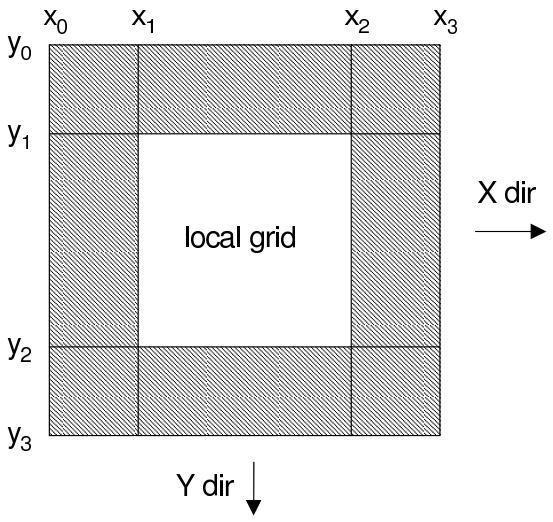

Figure 2: Realspace grid for a single task showing local and halo(shaded) parts of the grid - 2D only

In the original algorithm, the halos are first swapped with neighbours in the X direction, then the Y direction, then Z. See figure 2 for a 2-dimensional example. In each case the entire halo (e.g. the region $x_2 < x < x_3, y_0 < y < y_3$ which is passed to the right) was sent. The observation was made that since only the data in the local grid is required after the halo swap is complete, it is not necessary to send the entire width of the halo in each direction - any data which will end up in a halo can be discarded before it is sent. For example, in 2D, if the swap in the X direction is done first, then when the swap in the Y direction is performed only the region $x_1 < x < x_2$ needs to be

	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

Table 2: 60 iterations of the rs2pw libtest on 512 cores, before and after optimisation

sent. When done in 3 dimensions this gives substantial savings, shown in table 2 below, in terms of both the amount of data to be sent, and the amount of time taken packing buffers.

This data also shows that the packing and unpacking of the X buffers is more expensive than the Y buffers and the Z buffers. This is because when swapping in the Z direction, the halo data is contiguous in memory, and is more fragmented in Y and X directions. By applying the optimisation above and doing the halo swaps in the order Z, Y, X, a speedup of about 100% is obtained for this routine.

Due to the observation that swapping halos in the Z direction is cheapest, the heuristic for domain decomposition was modified. Originally, given a certain number of processors, the grid would be decomposed in order to minimise the total size of the halos relative to the local grids. In practise this means keeping the domains as close to cubic as possible. By adding small weightings to the heuristic calculation, CP2K will now favour decompositions which minimise the number of cuts in the Z direction, so maximising the halo size where the cost of swapping is cheapest. E.g. for a 128 MPI tasks CP2K would decompose the grid into 8x4x4 blocks, whereas before, 4x8x4, or 4x4x8 would be equally likely.

Since the buffer packing is relatively expensive, an approach using MPI derived types was also attempted. However, it was found not to provide any benefit, mainly due to the fact that when the halo data is recombined into the realspace grids, it is summed, rather than simply being copied into place, so a temporary receive buffer is still required, thus negating the main benefit of using derived types.

3.2 Non-blocking Communication

The second major change to the `rs2pw_transfer` routine was to replace the existing use of MPI_SendRecv with non-blocking MPI, thus allowing the packing of the buffers to be overlapped with communication. The following pseudocode outlines the approach taken:

```
calculate bounds for 'down' direction
allocate recv buffer
post MPI_IRecv
pack send buffer
send using MPI_ISend
```

```
calculate bounds for 'up' direction
allocate recv buffer
post MPI_IRecv
pack send buffer
send using MPI_ISend

MPI_Waitany on both outstanding recvs
unpack each buffer when it arrives

MPI_Waitall on both sends
```

This maximises performance by pre-posting recvs, which makes best use of Cray's 'Portals' underlying communication architecture, and by starting the send as early as possible, so that it can be overlapped with the packing of the other buffer.

In practice, this change was found to be performance-neutral. This is believed to be because the amount of time spent actually sending the data is much greater than the time spent packing the buffers, so there is actually very little time that could potentially be saved. As only 2 send/recv pairs are ever active at one time, this also limits the scope for overlapping. As it stands, `rs2pw_transfer` is in fact called in a loop over the grid levels, so there is the possibility that this would allow up a further factor of 4 (depending on the number of grid levels) more communications to be overlapped. However, the amount of extra book-keeping code that this would add, compared to a potentially modest performance gain meant that this was not attempted.

Nevertheless, because non-blocking communication is recommended by Cray as the optimal way to do point-to-point messaging, this change was committed to CVS as it may be of benefit where the halos are very large, or if there is poor network performance.

3.3 Benchmark Results

To test the results of these changes the bench_64 test case was used. The table below gives the runtimes on a range of processor counts on HECToR and the resulting speedup:

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

Table 3: Comparison of bench_64 runtime before and after rs2pw optimisation

4 Fast Fourier Transforms

The second major work item in the project was to profile and optimise the CP2K's Fourier Transform routines (steps III,IV in figure 1). CP2K has an FFT library module that provides a consistent interface to a number of popular FFT libraries including FFTW [7] 2 and 3, ESSL from IBM, ACML from AMD, and a CUDA implementation for GPUs. There is also in-built FFT library based on work by Goedecker et. al. [6]. These libraries provide 1D and 3D (serial) FFTs. As the planewave grids are distributed, CP2K requires a parallel 3D FFT, which involves three 1D FFTs, with global transpositions of the data using MPI_Alltoallv between the FFTs. For small numbers of processors (less than the number of planes in the grids), CP2K uses a 'plane', 'slice' or 'slab' decomposition for the grids, allowing the FFTs to be performed with only a single transpose step. For larger numbers of processors, a 'pencil' or 'ray' decomposition is used, which requires two transpose steps to be performed. This is more expensive, but can scale to larger numbers of processors than the plane decomposition.

4.1 Caching to amortize costs

Initally, CrayPAT was used to profile the FFT routines, using the PW_TRANSFER libtest as a micro-benchmark.

```
 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_
...
```

The most obvious item to address is the large number of calls to MPI_Cart_sub. This routine is used to partition the 3D cartesian communicator containing all the MPI tasks into multiple sub-communicators which are used to transpose the data in the FFT grids and is called once at every tranpose step. This operation is collective and blocking, so causes (unnecessary) synchronisation between all processes. However, in CP2K the grid layout and mapping of the grid to MPI tasks remains constant throughout the simulation, so the sub-communicators are also the same every time an FFT is performed. CP2K already provides a data structure `fft_scratch` which is used to cache data relating to

the FFT (data buffers, coordinates etc.), which was ideal for storing and reusing the MPI communicator handles. As a result it was possible to reduce the number of calls to MPI_Cart_sub from 11722 (for a 50 MD step run) to 5. As well as MPI_Cart_sub, a number of other related MPI operations could be moved into the FFT scratch cache, including rank and coordinate calculations. As a result of these changes, further speedups of 12% were achieved at 512 cores for the bench_64 test (see table 4). However, as the calls above were moved from the FFT loop into a one-off initialisation step, the performance gains would increase for longer runs.

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

Table 4: Comparison of bench_64 runtime before and after FFT caching optimisation

4.2 Transpose and MPI_Alltoallv

As part of the 3D FFT algorithm, the data on the plane wave grids has to be transposed in order that the correct data is available on each process for the next 1D FFT step. This communication is accomplised using MPI_Alltoallv. Alltoallv allows each process to send and receive differing amounts of data. However, in practice, the variation between processes is very small, typically only 1 extra row of the grid. For example, on a 125^3 grid divided over 8 processes 3 of the processes would have 15 planes, and 5 would have 16. Micro-benchmarking using the Intel MPI Benchmarks [8] showed that MPI_Alltoall performs better than MPI_Alltoallv on HECToR for the same volume of data transferred (see figure 3). Typical message sizes for the transposes range from 256KB up to several MB, so we would expect around a 20-30% speedup if we were able to use Alltoall instead of Alltoallv.

In order to allow this, the buffer packing and unpacking steps before and after the tranpose were modified to add padding to the data, so that each process would send the same amount of data. The padding would then be discarded when the buffer was unpacked at the receiver. In the above case, the 3 processes with only 15 planes would have 1 extra plane's worth of padding added. Since the fraction of additional padding that would be sent was smaller than the performance gain of using MPI_Alltoallv, a speedup was expected.

Initial results obtained using the FFT libtest were encouraging, with a speedup of 43% shown for a 125^3 grid on 256 processors. However, this result was not replicated when a full benchmark was run (e.g. bench_64), with only a 2% improvement in the FFT routines. After further investigation this appeared to be the result of poor synchronisation. The performance gain of using Alltoall only occurs when all processes in the communicator are well synchronised, such as in the libtest or Intel benchmark.

It was decided that this change should not be included into CP2K as the extra complexity of 'book-keeping' code to correctly pack and unpack the buffers was too much to justify such a small performance gain.

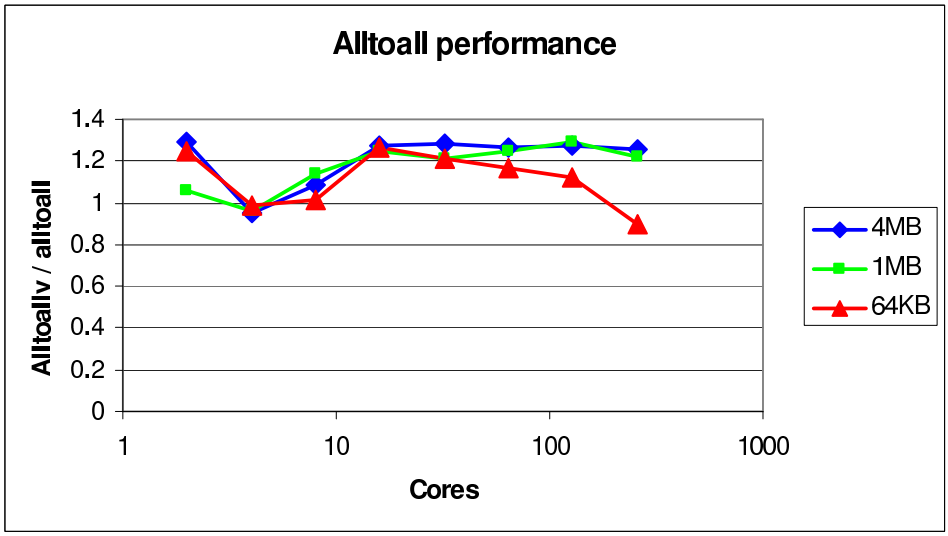

Figure 3: Graph of improvement of MPI_Alltoall over MPI_Alltoallv

4.3 FFTW Planning

As described earlier, CP2K has an FFT library interface which supports several FFT libraries. Of these, FFTW 3 generally gives the best performance. However, in order to maintain a consistent interface with all the libraries, CP2K does not make use of the facility in FFTW to reuse FFT plans when repeated FFTs are performed on the same arrays (or on arrays of the same layout for FFTW 2). Since planning was done before every FFT, only `FFTW_ESTIMATE`, the cheapest form of plan was able to be employed.

In a similar way to the earlier optimisations to the FFT routines, the code was modified so that a plan is created once, cached in the `fft_scratch`, and reused at each time step. In addition, now that the planning is only done at startup, it is possible to use the `FFTW_MEASURE` and for FFTW 3, `FFTW_PATIENT` and `FFTW_EXHAUSTIVE` plan styles. These plans take longer than the estimated plans, but should produce plans that allow individual FFTs to perform better. Because the choice of which plan style is best varies depending on the number of FFTs to be performed (i.e. number of SCF cycles) and even on the particular machine architecture, this choice is exposed to the user via a new input file option, which defaults to `FFTW_ESTIMATE`. Table 5 shows the runtimes for 2000 3D FFTs using each of the different types of plan. Even using `FFTW_ESTIMATE`, the code is slightly faster as only a single plan is made, rather than repeatedly creating and destroying plans each time. `FFTW_EXHAUSTIVE` does not show any benefit here, as the planning step takes around 10 times longer than for `FFTW_PATIENT`, but only gives a slightly better performance at runtime. For longer runs, the more expensive types of plans are expected to have a bigger benefit.

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

Table 5: Time and speedup for 2000 3D FFTs using different plan types

5 Load Balancing

The third and final major section of work looked at improving load balancing of the code, particularly in the region where gaussian basis functions held in sparse matrix representation are mapped to the real space grids (steps I,VI in figure 1). The real space to plane wave transfer and the FFT are well load balanced as the amount of work per process is proportional to the number of grid points, which are evenly distributed either as planes, pencils (plane wave grids) or blocks (real space grids). However, the task of mapping gaussians from the matrix to the grid is not so well load balanced. The matrix is distributed across all processors in a ScaLAPACK block-cyclic style. However, in general, each gaussian contained in a process' matrix block, may not be located in it's own section of the real space grids.

The existing load balancing scheme works by each process building a task list of the gaussians contained within it's local matrix block. Each particular gaussian will be mapped to one level of the real space grids, which may be either distributed or replicated. Thus the task list will be made up of 'distributed tasks' which must be completed by the process with the corresponding section of the real space grid, and 'replicated tasks' which may be completed by any process. A cost model exists in CP2K which is very accurate at predicting the computational cost of a particular task (see figure 4). This is used to estimate the load on each process due to distributed tasks and the replicated tasks are then assigned to processes in order to even out the load.

For homogeneous systems approximately the same number of atoms will reside in each process' section of the grid, so the distribution of gaussian mapping tasks will be well load balanced. For example, the bench_64 test case on 64 cores is able to achieve a nearly perfect load balance because there is large enough number of replicated tasks to remove the unbalanced load or distributed tasks:

```
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
```

However, for non-homogenous systems including interfaces and clusters (e.g. large molecules), some processes will have very few distributed tasks that they can process, as

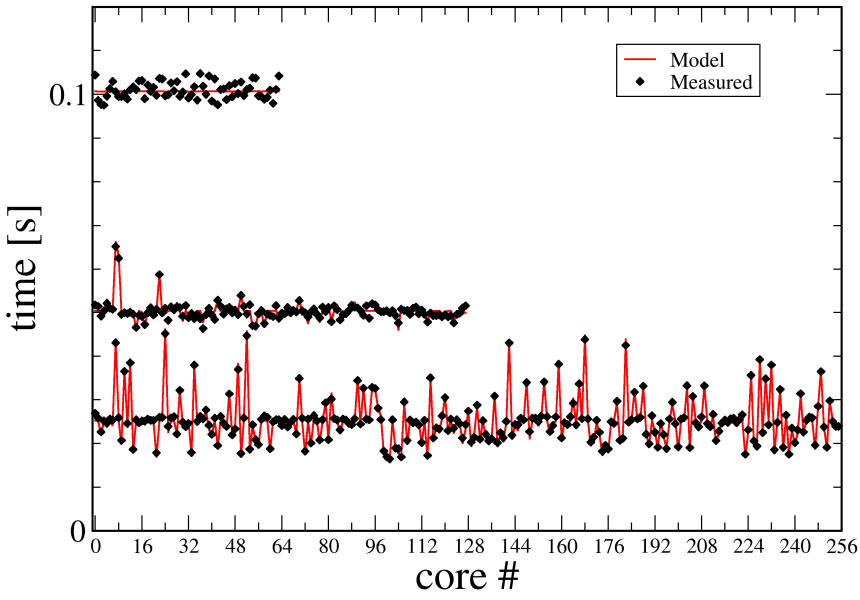

Figure 4: Excellent correlation between predicted and measured cost for task mapping (J. VandeVondele)

they may have few atoms in their local real space grid. If the number of replicated tasks is not enough, a poor load balance will result. To test this scenario, the W216 test case, described in section 2.1 was used. As shown, for W216 on 256 cores, there is a severe load imbalance in this region of the code, with the most heavily loaded processor having over 6 times as much work as the least loaded.

```
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
```

To address this, a scheme was proposed where instead of each process owning the same segment of the real space grid on each grid level, the mapping of grid blocks to processes could be varied at each level. Blocks which result in a heavy load on one grid level could be paired with blocks with low loads on another, thus distributing the load more evenly. This concept was introduced into the code as 'real' and 'virtual' ranks. Each process has a fixed real rank and a virtual rank for each grid level which is the rank of the process that would have held the particular section of the grid if there was no reordering. As the load balancing step is done before the grids are allocated, an appropriate mapping of real to virtual ranks is chosen based on the load of the distributed tasks, then the

grids are allocated on each process corresponding to their virtual ranks. There are a number of changes required in the realspace to planewave transfer routines to ensure that the reordered grid data is sent to the correct process for transferring to the plane wave grid, but this is facilitated by the use of a pair of mapping arrays `real2virtual` and `virtual2real` which are members of the real space grid data structure and are used to convert between the two orderings as needed.

For the same problem as above, using the new load balancing scheme, the load on the most overloaded process is reduced by 30%, and this is now only 3.5 times the load of the least loaded process. For this particular problem it is not possible to find a perfect load balance, as there is a single grid level block which has more load associated with it than then total average load. It is possible to overcome this by setting up the grid levels so that they are more closely spaced, and thus there is less load on each grid level. However, this comes at an increased memory cost for the extra grid levels and also affects the numerics of the calculation slightly (1 in 10^6). As shown in figures 5 and 6 if it is possible to balance the load perfectly, then this algorithm will succeed.

```
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
```

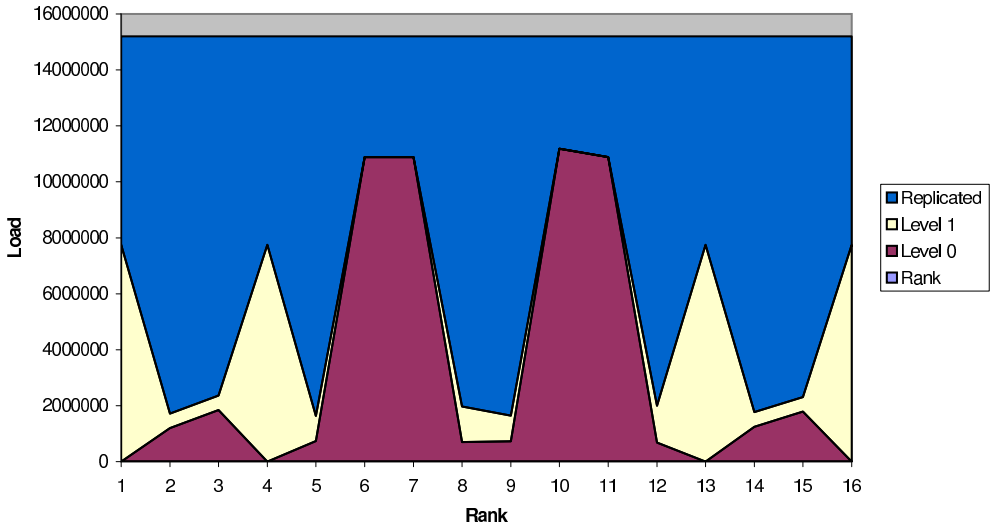

Figure 5: W216 load balance on 16 cores - perfect load balance achieved

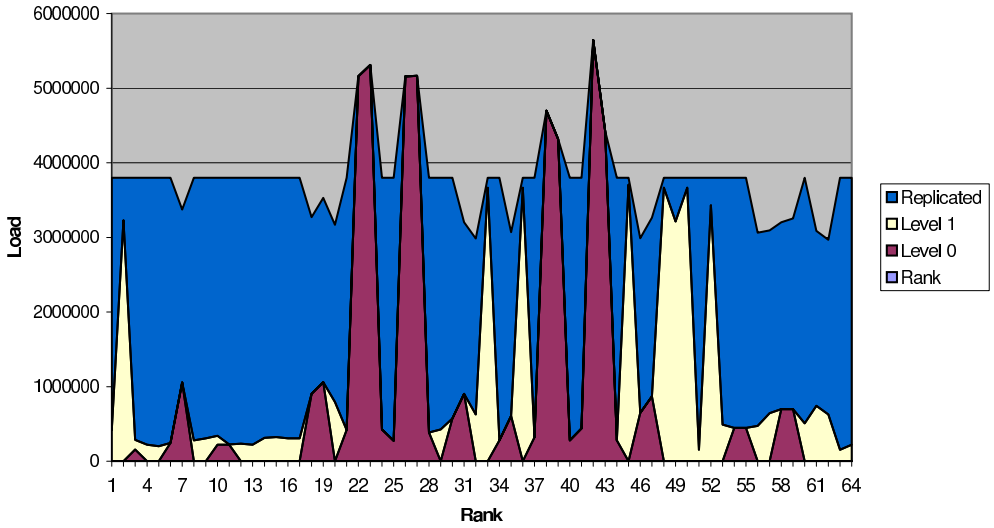

Figure 6: W216 load balance on 64 cores - several heavily loaded grid sections stop perfect local balance

5.1 Benchmark Results

The W216 test case was use to benchmark these changes. The table below gives the runtimes on a range of processor counts on HECToR and the resulting speedup:

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

Table 6: Comparison of W216 runtime before and after rank reordering for load balance

6 Final Benchmarks

Here the overall performance gains from the dCSE are shown (figures 7 and 8). To show performance more clearly, the reciprocal of the runtime is plotted. For bench_64, a speedup of 30% on 256 cores is achieved. An even greater speedup up 300% on 1024 cores is shown for W216, showing the effect of these optimisations on larger problems (and inhomogeneous systems in particular). This exceeds the original aims set out in the project proposal, which were:

> we expect the performance gain on 64-256 processors to be around 10-15%. Far more significantly, at the capability end 512-1024+ processor jobs are expected to increase in performance by around 40-50%.

It should be noted that in addition to the work performed within the dCSE project, other work was undertaken by the CP2K development group, which would affect the benchmarks of W216. In particular, the load balancing was modified to allow heavily loaded processes to shift some work to neighbouring processes, provided that their realspace grid halos still contain the entirety of a Gaussian to be mapped. This has the effect of reducing the highest peaks of load (see figure 6. However, the FFT and halo swap optimisation also has a significant effect, especially on larger numbers of cores. Due to the concurrent nature of development, and the fact that W216 is relatively expensive to run, these changes were not benchmarked at each step in development.

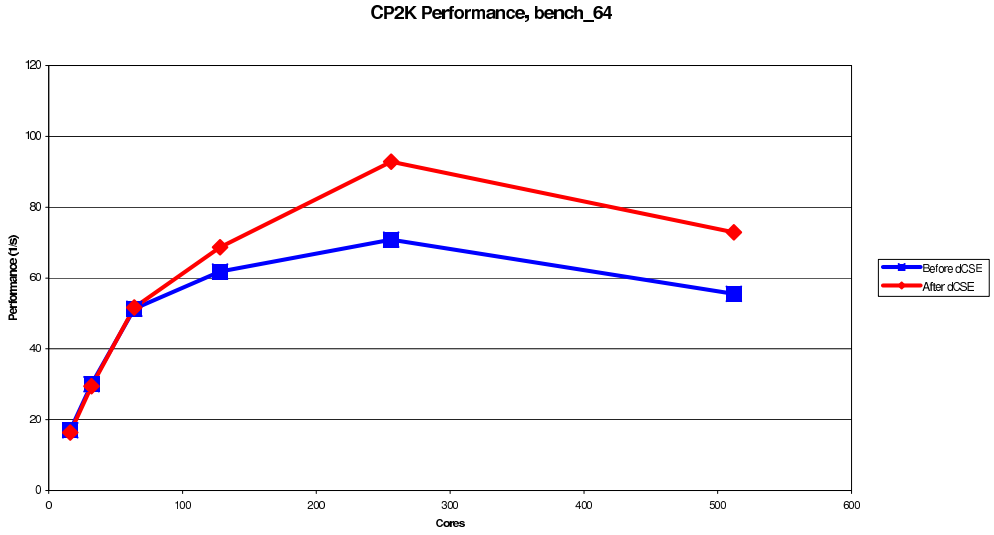

Figure 7: Overall performance gains on bench_64

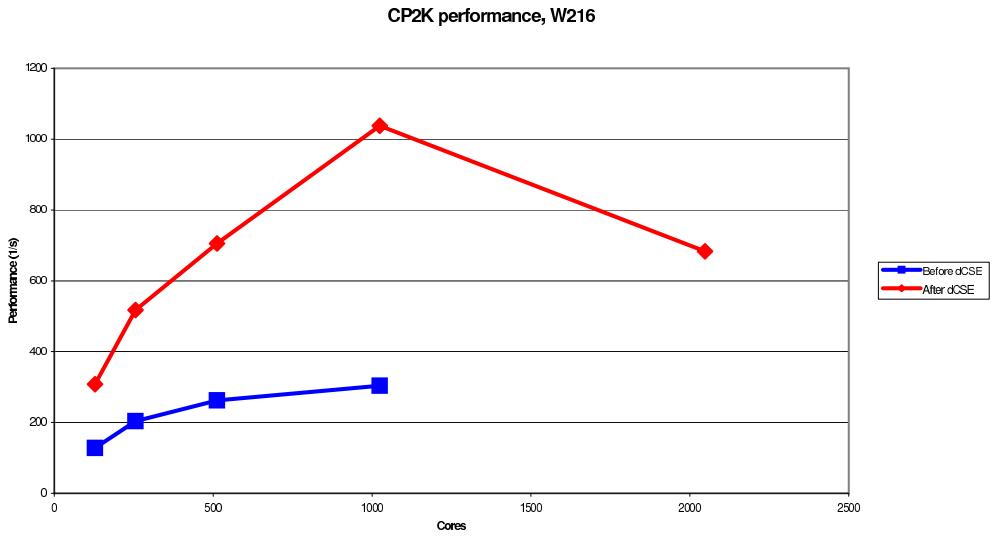

Figure 8: Overall performance gains on W216

7 Further Work

During the dCSE project, further data on CP2K's performance was gathered from CrayPAT profiles, as well as CP2K's own timing routines. This highlighted a number of other regions of the code that were not addressed in this project, but are still barriers to improved scalability of the code.

1. Dense matrix algebra (ScaLAPACK). CP2K makes use of a number of ScaLAPACK routines, principally PDGEMM (matrix multiplication) and PDSYEVD (solving an eigenvalue problem). These have been measured to have a parallel efficiency of around 20-30% on 2048 cores compared with 512 cores, which is the worst of all the major regions of the code. Currently these scale up to the point where there is about one atom per core. At this point communication costs dominate over computation, as shown by CrayPAT profiles.

2. Sparse matrix algebra. Several of the important matrices in CP2K (Kohn-Sham, overlap, density matrices) are sparse. CP2K has it's own routines for multiplication of sparse and full matrices, but these also scale poorly in the same way as the dense matrix routines above. Work is already underway by the CP2K development group at the University of Zurich to rewrite these routines for better scalability and performance.

It is believed that the scalability of the code can be further improved by introducing OpenMP 'hybrid' parallelism within a multi-core node while retaining the existing MPI communication between nodes. For a fixed number of processes, this will reduce the number of off-node messages to be sent, and is a better 'fit' to the increasingly wide-SMP nodes currently available on the Cray XT series. A proposal for further dCSE funding to implement this has been funded.

8 Conclusion

In summary, six months of HECToR dCSE funding has allowed detailed investigation of CP2K from a performance standpoint and substantial development of the performance-critical regions involved in the transformation from Gaussian to planewave bases which is central to the code. This resulted in performance gains which exceeded those set out in the initial project proposal.

These improvements are now available to CP2K users worldwide via CVS and are already included in the installed versions of CP2K on HECToR and HPCx. Knowledge of how to benefit from these improvements, and understanding of the behaviour of the code in general will be shared with the user community through this report on the HECToR website, and also through the September dCSE workshop in Oxford.

References

[1] CP2K website, http://cp2k.berlios.de

[2] Quickstep: fast and accurate density functional calculations using a mixed Gaussian and plane waves approach, J. VandeVondele, M. Krack, F. Mohamed, M.Parrinello, T. Chassaing and J. Hutter, Comp. Phys. Comm. 167, 103 (2005)

[3] HECToR website, http://www.hector.ac.uk

[4] Libint website,
http://www.files.chem.vt.edu/chem-dept/valeev/software/libint/libint.html

[5] Gaussian basis sets for accurate calculations on molecular systems in gas and condensed phases, J.VandeVondele and J. Hutter, J. Chem. Phys. 127, 114105 (2007)

[6] An efficient 3-dim FFT for plane wave electronic structure calculations on massively parrallel machines composed of multiprocessor nodes, S. Goedecker, M. Boulet and T. Deutsch, http://pages.unibas.ch/comphys/comphys/SOFTWARE/FFT/fft.ps

[7] The Design and Implementation of FFTW3, M. Frigo and S. Johnson, Proceedings of the IEEE 93 (2), 216.231 (2005).

[8] Intel®MPI Benchmarks 3.2, http://software.intel.com/en-us/articles/intel-mpi-benchmarks/