CP2K - Sparse Linear Algebra on 1000s of cores
A dCSE Project

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

January 17, 2012

Abstract

CP2K is a freely available atomistic and molecular simulation code, able to study of a wide range of molecular and bulk materials with methods including classical potentials, density functional theory (DFT), Hartree-Fock and post-HF methods. Following two earlier dCSE projects, we report here on an additional 6 months of work to optimisise the DBCSR sparse matrix multiplication library embedded within CP2K. Efficient and scalable sparse matrix operations are shown to benefit existing users of the code by reducing time to solution for typical simulations, and has enabled development of new algorithms including for the fully linear scaling DFT based on density matrix iterations.

Contents

1 Introduction **2**
1.1 CP2K 2
1.2 HECToR 2

2 Objectives and Results **2**

3 DBCSR Optimisation **5**
3.1 Node-local data multiply 6
3.1.1 Recursive multiplication 6
3.1.2 SMM 7
3.1.3 Threading 10
3.2 MPI data exchange 12
3.3 Data preparation for multiplication 13

4 Recommendations for users **15**

5 Conclusion **16**

6 Acknowledgment **16**

1 Introduction

1.1 CP2K

CP2K[1] is an open-source program written in Fortran 95 for atomistic and molecular simulation. The code is most well-known for it's implementation of the Quickstep[2] linear scaling DFT algorithm, but has been designed in an extensible and efficient manner thus allowing users a wide choice of simulation methods from classical, semi-empirical and DFT to Hartree-Fock and the recently added Møller-Plesset second order perturbation theory (MP2). CP2K consists of over 800,000 lines of code and is developed by a distributed team of researchers, mostly based in Switzerland, but also including the UK.

CP2K is the third most heavily-used code on HECToR (data from 21 Dec 2010 - 3 October 2011), consuming 8.48% (731193 AUs) of system usage, behind only CASTEP (9.21%) and VASP (12.11%). Therefore we believe the work done under dCSE funding[3][4] greatly impacts the scientific output of HECToR as a whole. Section 3.3 of this report advises HECToR users how they can achieve the best possible performance from CP2K.

1.2 HECToR

HECToR[7] (High End Computing Terascale Resource) is the UK National Supercomputing Service, and consists of Cray XE6 hardware. During this project, the Phase 2b and Phase 3 components of the system were used.

The Phase 2b XE6 system consists of 1856 compute nodes, each containing two 2.1 GHz 12-core 'Magny-cours' AMD Opteron processor and 32 GB of main memory, giving 1.33 GB per core. Each processor is coupled to a Cray Gemini routing chip, providing a high bandwidth and low latency 3D torus network. The peak performance of the system was 360 TF, and ranked 16th in the June 2010 Top 500 list.

For Phase 3, the processors have been replaced with 16-core 2.3 GHz 'Interlagos' AMD Opteron processors, giving a total of 32 cores per node, with 1 GB of memory per code. The addition of another 10 cabinets (928 compute nodes) in Jan 2012 will increase the peak performance to over 820 TF.

In addition, thanks to the support of Prof. Hutter and the HP2C project we have access to the Rosa (XT5 12-core), Palu (XE6 24-core) and Todi (XK6 16-core) systems at CSCS in Switzerland. This allowed continued testing of CP2K on a range of multi-core systems, and a development platform when HECToR was unavailable.

2 Objectives and Results

It should be noted that in contrast to prior CP2K dCSE projects where optimisation was carried out with the support of the development team, but on fairly independent sections of the code, in this project the optimisation was carried out at the same time as the DBCSR[5] library was under active development. It is therefore difficult to seperate the benefits of the dCSE-funded work from the wider development of DBCSR, although in the following section of the report we will detail exactly what work was carried out and show performance results for all changes made.

To quantify the performance of the code, we compare two versions of CP2K on the most recent HECToR hardware (Phase 3, XE6, 32 cores per node) - CP2K 2.1.390 (1st

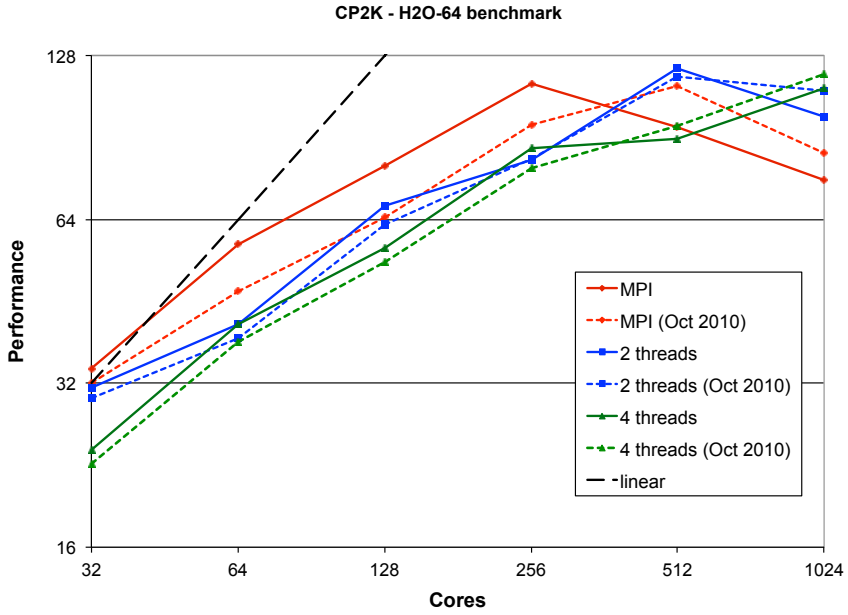

Figure 1: Performance of H2O-64 benchmark on HECToR comparing Oct 2010 and Dec 2011 versions of CP2K (Performance normalised to Oct 2010, 32 cores)

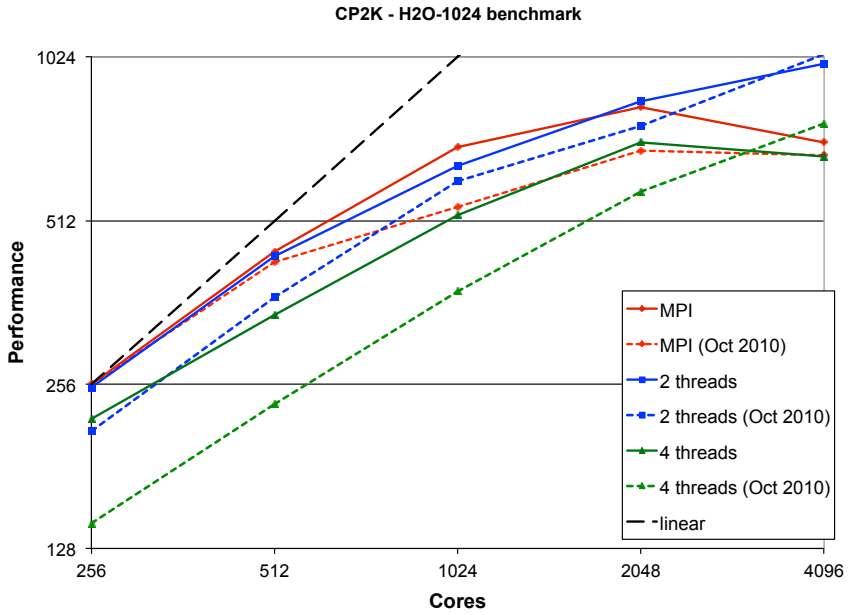

Figure 2: Performance of H2O-1024 benchmark on HECToR comparing Oct 2010 and Dec 2011 versions of CP2K (Performance normalised to Oct 2010, 256 cores)

Oct 2010), and CP2K 2.3.r12105 (22nd Dec 2011). The benchmark inputs H2O-64 and H2O-1024 are molecular dynamics runs using cubic cells of 64 and 1024 water molecules respectively. These represent typical and fairly large systems that might be studied

currently with CP2K. Both use the GTH Basis Set[9] (TZV2P) and the PBE[10] and PADE exchange-correlation functionals respectively. The performance of both versions of CP2K, for varying numbers of threads per MPI process is shown in figures 1 and 2.

Several performance objectives were stated in the project proposal:

- Mixed-mode OpenMP/MPI : we expected to bring performance up to par with pure MPI for compute dominated runs. For communication-dominated runs we expected approximately a 30% improvement over pure MPI.

  For the H2O-64 benchmark on 32 cores is mostly compute dominated (approximately 15% of the runtime is spent in communication). Here we see that the mixed-mode code still lags a little way behind pure MPI (9% for 2 threads, 42% for 4 threads). However, only about 40% of the runtime is spent in DBCSR, so there are still significant other areas of the code that do not have as good threaded performance.

  In the communication-dominated regime (e.g. on 512 cores, 39% is communication), we see that using 2 threads per MPI process gives a speedup of **22%** over pure MPI.

  The larger H2O-1024 benchmark spends about 70% of its runtime in DBCSR. Here the goal of achieving equal performance between mixed-mode and pure MPI has been achieved for compute-dominated runs (on 256 cores, 12% is communication). In particular, compared to the Oct 2010 version of the code, the performance with 2 threads per process has been increased by 18% and for 4 threads per process by 35%.

  Similarly to H2O-64, we see that as the total number of cores is increased and communication begins to dominate, the mixed-mode code begins to perform better (e.g. an improvement of **29%** over pure MPI on 4096 cores).

  Extensive testing of similar benchmark inputs ranging from 32 to 2048 water molecules has been performed by Joost VandeVondele on the Cray XE6 'Monte Rosa' system at CSCS. Analysis of these results showed that the absolute fastest time to solution for all problem sizes could be achieved using 2 OpenMP threads per process.

- In the compute-bound regime, we expected a 10% improvement.

  The H2O-64 results on 32 cores show an improvement over the Oct 2010 code of **7%** (pure MPI). For H2O-1024 there is little change in the peformance for pure MPI, but as noted above large improvements have been made for the mixed-mode code.

- In the communication-bound regime, we also expected a 10% improvement.

  It is difficult to draw general conclusions about performance in the communication-bound regime from the H2O-64 data. The best case performance (2 threads per process, 512 total cores) has been increased by **4%**, but in other cases the mixed-mode code performs slightly worse that before. For H2O-1024 at the highest cores count tested (4096) the newer code performs 6% faster than the Oct 2010 version for pure MPI, but 4% worse with 2 threads per process.

3 DBCSR Optimisation

DBCSR (Distributed Block Compressed Sparse Row) is a library embedded within CP2K which had been developed to provide a storage format and multipication operation for the sparse block-structured matrices used within CP2K. Many of the large matrices stored in CP2K (density matrix, overlap matrix, Kohn-Sham matrix etc.) are naturally sparse due to the localisation of the Gaussian basis set in space. The block structure arises from the fact that each atom may be represented by a number of basis functions. Thus for a system with N atoms, typical matrices would have N rows (or columns) of blocks, where each block itself comprises multiple rows (or columns) e.g. 1 for Hydrogen with a minimal basis set (SZV-GTH-MOLOPT), or 13 for Oxygen with a larger basis set (DZV-GTH-MOLOPT). Thus the entire matrix is composed of rows and columns of small dense blocks of varying size. Blocks are addressed using a CSR storage format where an indivdual block can be accessed via a pointer to the start of it's row, plus an offset into the row for that block. The matrices are distributed across MPI processes in a 2D grid.

Matrix multiplication is performed using Cannon's algorithm[8]. Briefly, to perform the multiplication $C = A * B$ where the matrices are distributed on a square grid of P processes, there are $\sqrt{P}$ steps in the algorithm. At each step the local data area is multiplied and acculumated into the result matrix, then row-wise and column-wise shifts (for matrix A and B respectively) are performed using non-blocking MPI. This operation is illustrated in figure 3.

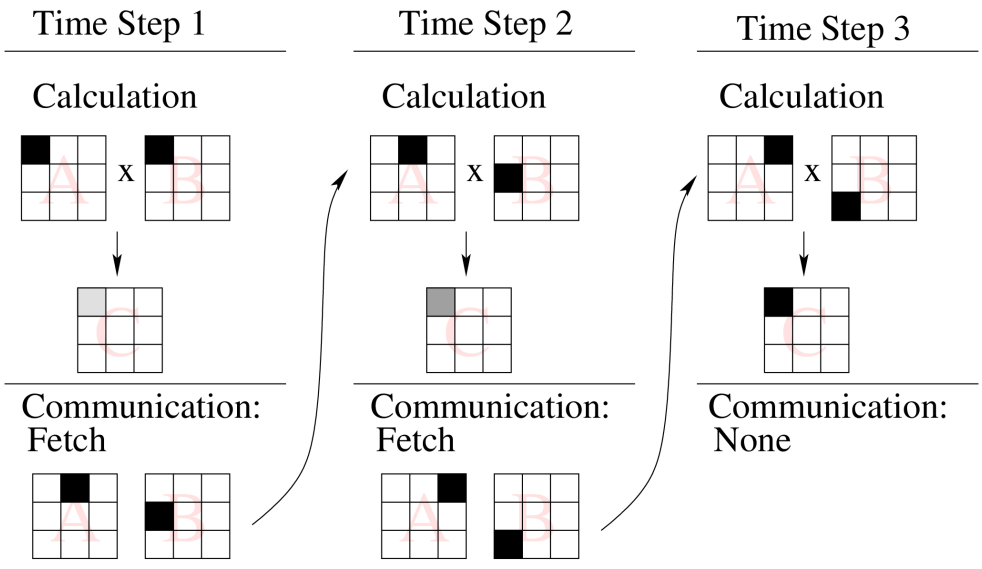

Figure 3: Example of Cannon's algorithm in DBCSR for 9 processes (Image courtesy Urban Borstnik)

The communication is performed using double-buffering - with `calc` and `comm` buffers to hold current and next matrix data and index - as illustrated by the following pseudocode:

```
do i=1,nsteps

  call mpi_waitall() - ensures communication from previous iteration
    is complete (new data has arrived in current calc buffer,
    comm buffer data has been sent)

  post mpi_irecv() and mpi_isend() for column and row shifts - data is
    sent from the current calc buffer, and received into the comm buffer

  perform C += A x B on current calc buffers

  comm and calc buffers are (pointer) swapped for next iteration

end do
```

In order to take advantage of OpenMP, DBCSR decomposes the rows of blocks of a matrix over the available team of threads. Load balancing is achieved by assigning rows to threads so that the total number of blocks per thread is approximately equal (since in a sparse matrix, each row may now have the same number of non-zero blocks). Matrix rows are then reordered so each thread's rows are contiguous in memory.

3.1 Node-local data multiply

Initially, when multiplying two matrices (or sub-matrices), DBCSR would perform a loop over rows of the left (A) matrix, and for each block, loop over all blocks in the corresponding column of the right matrix (B), accumulating the products into the appropriate block of C. The individual block multiplications were performed by DGEMM from the system-provided BLAS (or Fortran MATMUL if not available). In fact, as the blocks of the matrix are traversed, instead of multiplying individual blocks, the parameters for the multiplication (block pointers, sizes) are added to a stack, which is then processed once it reaches a certain size. This allows better use of cache by alternating between accessing the index (building the stack) and accessing the data (processing the stack) in a more granular fashion. A stack limit of 1000 is used as a default, but may be overridden by setting the `MM_STACK_SIZE` variable in the `GLOBAL / DBCSR` section of the input file

3.1.1 Recursive multiplication

The linear multiplication described above exhibits poor use of cache, since blocks of the B matrix will be accessed several times - once for every block in the corresponding column of A - but not necessarily soon afterwards (poor temporal locality). In addition, since the A matrix is accessed by row and B by column one of these will stride irregularly through memory (poor spatial locality). To overcome these problems a recursive multiplication scheme was implemented which makes divides up the matrices by halves until the block

size is below a preset limit. At this base case of the recursion, the multiplication parameters for the blocks in this section of the matrices are pushed onto the stack (which may be executed if it has grown large enough). Choosing when to stop recursing is a trade-off between overhead of further recursion steps, and the cost of poor memory locality by operating on too large a section of the matrix. The controlling parameter `norec` defines the maximum number of matrix blocks the recursion will terminate at. Experiments were performed to tune this parameter using dense matrices of size 2000x2000 and varying block sizes (see figure 4). The graph shows that relatively small numbers of blocks should be processed at once, indicating the recursion is not too expensive in most cases. A value of 512 was chosen as a reasonable compromise. The new algorithm was first tested as a standalone code before being integrated into the main CP2K code. Results in the graph are taken from the stand-alone code.

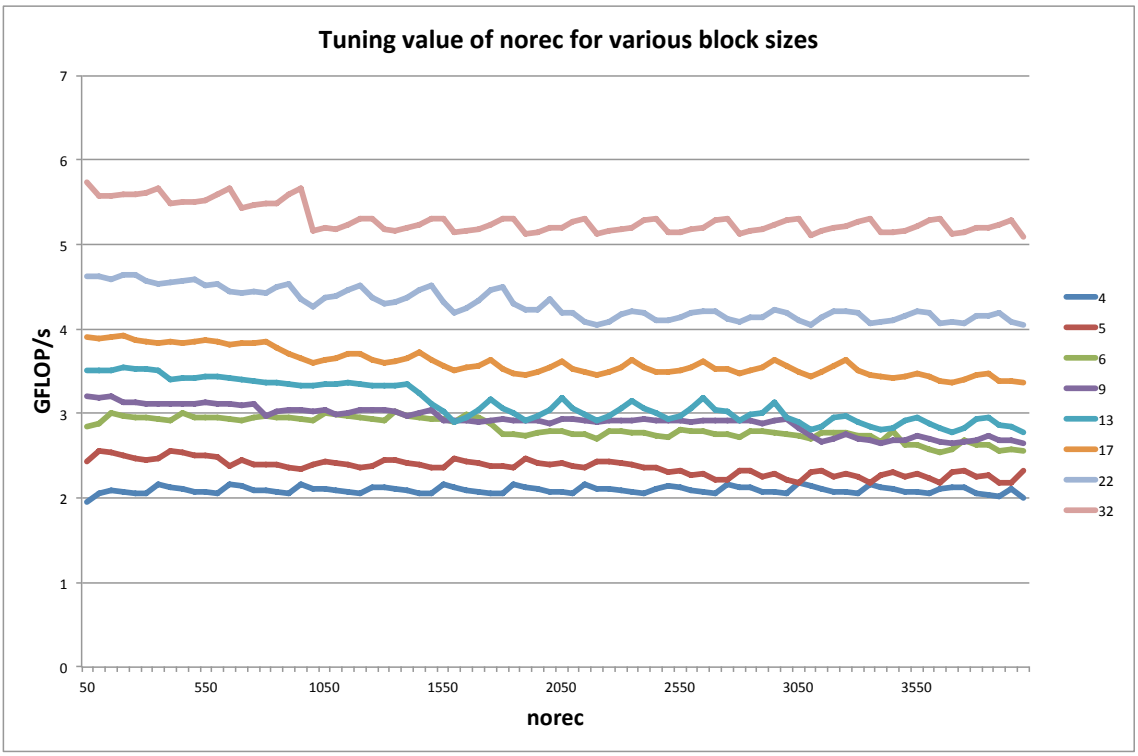

Figure 4: Tuning the recursion parameter (Performance for various block sizes)

3.1.2 SMM

At the same time, before introducing the recursive multiplication into the main code, another new feature was introduced - a specialised small matrix multiplication library 'libsmm'. In CP2K the sizes of the matrix blocks depend on the atomic species being simulation and the basis set. Typically, this results in small blocks with unusual sizes such as 1x4, 5x13, 9x22 etc. Libsmm takes an autotuning approach to producing specialised DGEMM routines for a specified set of small matrix sizes:

Firstly for a set of 'tiny' block sizes (typically 1 up to 12) a set of different loop permutations and unrollings is generated for each combination of block sizes (`m`,`n`,`k`),

where matrix C is `m` by `n`, A is `n` by `k` and B is `k` by `m`. E.g. for `m`=2, `n`=3, `k`=5 the canonical multiplication loop is:

```
DO j= 1 , 3 , 1
DO i= 1 , 2 , 1
DO l= 1 , 5 , 1
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+0)*B(l+0,j+0)
ENDDO
ENDDO
ENDDO
```

There are six possible loop permutations (ijl, jil, ilj, jli, lij, lji) and for each permutation it is possible to unroll any combination of the loops e.g. for the above loop ordering the first three possible unrollings are:

```
DO j= 1 , 3 , 1
DO i= 1 , 2 , 1
l= 1
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+0)*B(l+0,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+1)*B(l+1,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+2)*B(l+2,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+3)*B(l+3,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+4)*B(l+4,j+0)
ENDDO
ENDDO

j= 1
DO i= 1 , 2 , 1
DO l= 1 , 5 , 1
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+0)*B(l+0,j+0)
  C(i+0,j+1)=C(i+0,j+1)+A(i+0,l+0)*B(l+0,j+1)
  C(i+0,j+2)=C(i+0,j+2)+A(i+0,l+0)*B(l+0,j+2)
ENDDO
ENDDO

j= 1
DO i= 1 , 2 , 1
l= 1
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+0)*B(l+0,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+1)*B(l+1,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+2)*B(l+2,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+3)*B(l+3,j+0)
  C(i+0,j+0)=C(i+0,j+0)+A(i+0,l+4)*B(l+4,j+0)
  C(i+0,j+1)=C(i+0,j+1)+A(i+0,l+0)*B(l+0,j+1)
```

```
      C(i+0,j+1)=C(i+0,j+1)+A(i+0,l+1)*B(l+1,j+1)
      C(i+0,j+1)=C(i+0,j+1)+A(i+0,l+2)*B(l+2,j+1)
      C(i+0,j+1)=C(i+0,j+1)+A(i+0,l+3)*B(l+3,j+1)
      C(i+0,j+1)=C(i+0,j+1)+A(i+0,l+4)*B(l+4,j+1)
      C(i+0,j+2)=C(i+0,j+2)+A(i+0,l+0)*B(l+0,j+2)
      C(i+0,j+2)=C(i+0,j+2)+A(i+0,l+1)*B(l+1,j+2)
      C(i+0,j+2)=C(i+0,j+2)+A(i+0,l+2)*B(l+2,j+2)
      C(i+0,j+2)=C(i+0,j+2)+A(i+0,l+3)*B(l+3,j+2)
      C(i+0,j+2)=C(i+0,j+2)+A(i+0,l+4)*B(l+4,j+2)
    ENDDO
```

In the case where m,n or k are not prime, it is possible to partially unroll the loops (e.g. a 4-fold loop may be unrolled by 2 or by 4). For the largest block sizes considered (12,12,12) there are 540 permutations that are automatically generated. Each of these variants is then run (for long enough to perform 1 GFLOP) and the performance (GLOP/s) is recorded. The particular loop structure which generated this performance is then recorded.

Recognising that for larger block sizes, the number of combinations to be tested will grow rapidly, in the second stage of autotuning a different set of 'small' block sizes are used - these are the sizes that will be supported by the final library, and can be chosen by the user to match block sizes they expect to be used by CP2K as a result of the particular system they are simulating. By default the set of m,n,k are 1, 4, 5, 6, 9, 13, 16, 17 and 22.

Again for each possible combination of m,n,k, a set of 7 multiplication procedures are generated:

- tiny version (or canonical loop, if not available)
- Fortran MATMUL
- Library DGEMM
- 4 recursive variants with base cases terminating in good-performing tiny mults

Similarly to before, each of the permutations is then run (for 10 GLOPs) and the performance is measured. The best variant for each of the small sizes is then compiled into an individual object file and added to the library. Note that in some cases, particularly for large block sizes, library DGEMM may give the best performance. Because of this, CP2K still needs a BLAS library when linking with libsmm in order to resolve these fall-though cases. A wrapper routine is also included such that SMM can be called for any sizes of m,n,k and if these are not supported directly in the library, will call directly to BLAS. The entire library compilation is controlled by a set of scripts and makefiles, allowing it to run in parallel on a single node on the HECToR compute nodes. Compilation of the library took around 2 hours 15 minutes using the default set of 'tiny' and 'small' sizes.

Finally, the library routines are then tested against the supplied BLAS library, both to check performance and correctness. Results of this test are shown in figure 5, taken

from a compilation on the XE6 TDS system. Especially for small block sizes (or blocks where one or more dimensions is small) we find that libsmm outperforms the BLAS in Cray's libsci by up to 10 times. Similar results have been found comparing with e.g. MKL on an Intel platform. For larger block sizes, the performance tends towards Libsci BLAS indicating that a faster method could not be found. It should be noted that in the limit of very large blocks (1000x1000), DGEMM achieves around 12.8 GLOP/s, which is around 5.5 FLOPs/cycle, indicating that the library is making use of the AMD Bulldozer architecture's FMA4 instructions since for these tests only a single thread is running.

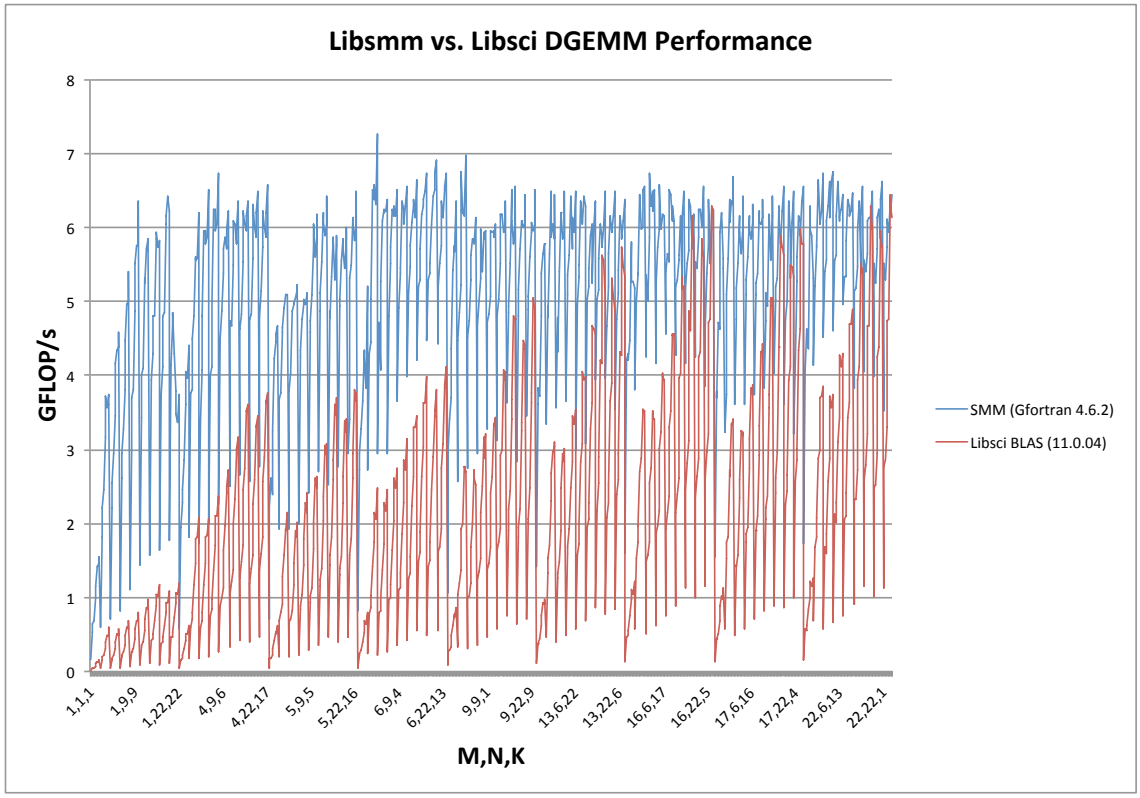

Figure 5: Comparing performance of SMM and Libsci BLAS for block sizes up to 22,22,22

Libsmm is distributed with the CP2K source package, and a version of the library optimised for the current HECToR Phase 3 'Interlagos' processors can be found in `/usr/local/packages/cp2k/2.3.15/libs/libsmm/`.

3.1.3 Threading

Recall that DBCSR matrices are decomposed by rows, which each row being 'owned' by a specific OpenMP thread. The current load balancing strategy (rows are assigned weighted by the block size of each row) results in some load imbalance since it does not take account of the sparsity of each row.

When investigating how to improve the load balance it was discovered that thread 0 was consistently taking longer than the other threads by up to 20% (even for artificial inputs which are perfectly load balanced). Careful inspection of the code revelead this was due to timing routines called by every thread which contained `!$omp master` directives.

	Row sizes only(CVS)		Row block counts	
Thread	Blocks	Time(s)	Blocks	Time(s)
0	25806281	895.52	23249480	921.14
1	23278623	868.15	23271941	813.60
2	21799888	758.59	23284712	788.97
3	26823318	1050.02	23261575	896.97
4	15986481	629.71	23254617	1003.18
5	25902624	995.68	23274890	813.44

Table 1: Comparison of load balancing strategies

These were removed from the recursive part of the multiplication, since timing detail at this level is not of interest to users, and can be obtained via profiling if necessary.

To achieve a better load balance using the total number of blocks in a (distributed row) as the criteria for assigning rows to threads was investigated. This achieves a good overall load balance of FLOPs for matrix multiplication (see table 1). However, due to the multiple steps involved in the Cannon's algorithm multiplication, when DBCSR is used for more than one MPI process, each local multipication is still not guaranteed to be load balanced, and in fact significant load imbalance is observed, leading to poor scaling of the multipication with larger thread counts. In addition, calculating the number of blocks per row requires an MPI_Allreduce per processor row which incurs some synchronisation cost. As a result, this load balancing method was not added to CVS.

Since this load imbalance is intrinsic to the idea of having threads 'own' fixed rows of the matrix, two further modifications to the threaded multiplication were proposed to allow threads to share access to the result (C) matrix in more flexible manner:

1. Using OpenMP 3 tasks, create a task for each leaf of the recursive multipication, ensuring that the recursion terminates such that there are many more tasks than threads. Then in each task, take a lock on the corresponding area of the C matrix to protect against concurrent update from another thread. Testing in the standalone code showed that generating and executing the tasks did not give too much overhead, especially when the number of thread is small i.e. within a single NUMA region so that memory access is not an issue (see figure 6, showing performance of the standalone test code). However, the addition of the locking overhead was roughly equivalent to the cost of the load imbalance, so this was not integrated into the main code.

2. In the process of adapting DBCSR to use GPUs (HP2C-funded work by Urban Borstnik, Univ. Zurich) an extra layer of indirection was added - queues of the multiplication parameter stacks. It was proposed that as the matrices are recusively divided, a queue is created for every resulting sub-matrix. Each thread fills the stacks corresponding to its parts of the matrix, following the current thread distribution, and puts them in the appropriate queue. The threads then process queues independently (preferentially by the thread that filled the queue), but with the possibility that if a thread has no work left in its queues it can start processing queues filled by other threads. This avoids the overhead of locking since only a single queue every writes to a given area of the C matrix, and load balancing is

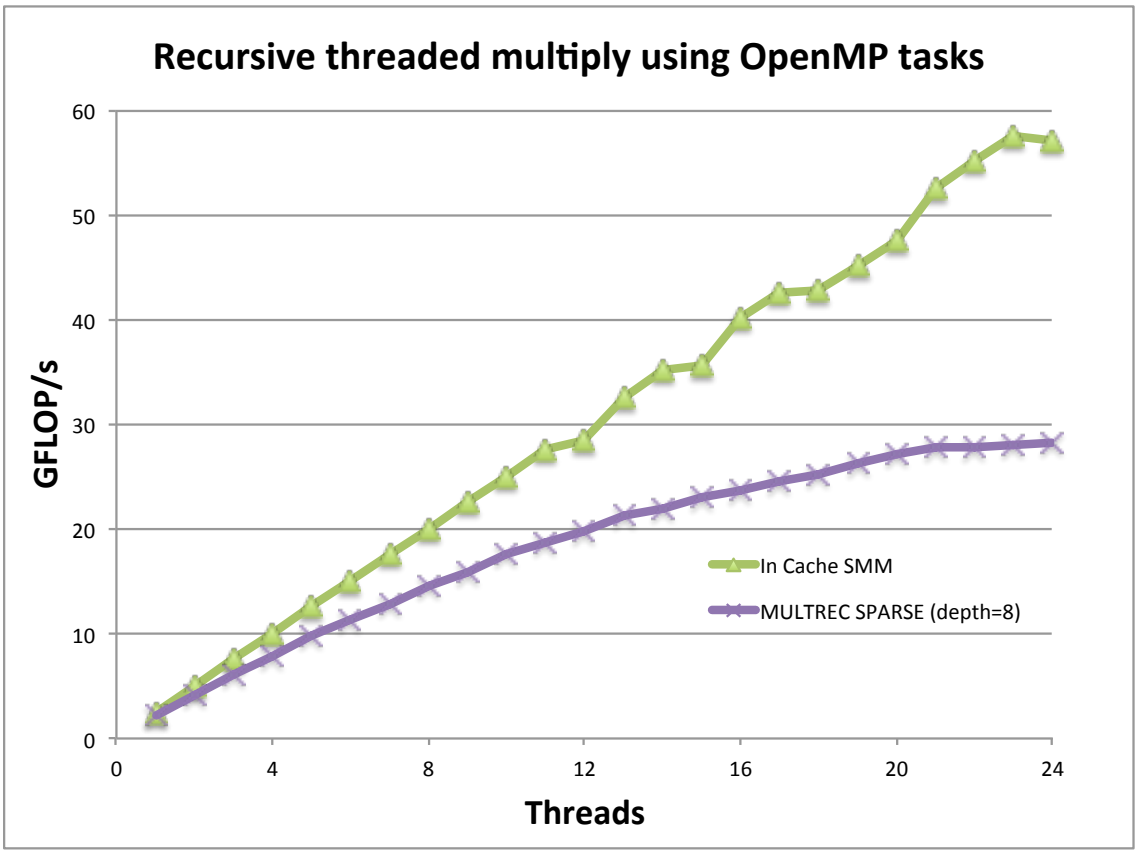

Figure 6: Performance of OpenMP tasked version against ideal SMM on 24-core XE6 (CSCS)

achieved by dynamic 'work-stealing'. This method also avoids the current overhead of merging each threads private 'work matrix' since all threads could now write directly to a shared data area. While this idea seems in principle to solve all the current load balance problems, it would very complex to implement and so was not attempted in the scope of this project

3.2 MPI data exchange

Craypat profiling indicated that the majority of time spent in MPI during the multiplication loop was in MPI_Waitall, with quite a large imbalance (some processes spending up to 6 times longer than the fastest), although the amount of data sent from one process to another is well balanced, as is the computation performed by each process between communications. Even though the existing double-buffering allows computation to take place while communication for the next iteration is ongoing, the MPI_Isend and MPI_Irecv calls are posted at the same time. If the recieves could be pre-posted far enough in advance it was hoped this might reduce the time spend in MPI_Waitall by reducing the need for additional data copies for unexpected messages (arriving before the corresponding recieve is posted). This was done by posting the recieve a whole iteration earlier than the corresponding send (and two whole iterations before it was

due to complete). Unfortunately, this did not reduce the time spent it MPI_Waitall as expected.

Another experiment to determine if the wait time was due to message latency (i.e. genuinely waiting because the message had not arrived from the remote process) was carried out. Here the local multiplication step was made arbitrarily long by sleeping for several seconds after actually carriying out the multiplication. This also did not affect the amount of time in MPI_Waitall, so it was determined that the cost of the waitall must be due to work that could only be done when MPI was called (i.e. requiring the MPI progress engine), rather than asynchronous DMAs or similar, although exactly what was occurring is not clear.

Given that this cost is fixed, we implemented a scheme whereby during the multiplication, the master thread would periodically poll MPI by making MPI_Testany calls. This has two advantages - firstly at least while the MPI processing was ongoing, other threads were doing useful work (rather than having the other N-1 threads idle during the Waitall). Secondly, since each local multiplication may be be somewhat load imbalanced, if thread 0 is underloaded then the MPI_Testany calls become essentially free as they simply take up some of the slack time that would be spent waiting for other thread to complete. To account for the case where the polling of MPI takes significantly longer than the available slack time, it is possible for the user to manually underload the master thread by a specified amount. This behaviour is controlled by two new variables in the input file (in the `GLOBAL / DBCSR` section):

- `USE_COMM_THREAD`

  During multiplication, use a thread to periodically poll MPI to progress outstanding message completions. This optional keyword cannot be repeated and it expects precisely one logical. Default value: .TRUE.

- `COMM_THREAD_LOAD`

  If a communications thread is used, specify how much multiplication workload (%) the thread should perform in addition to communication tasks This optional keyword cannot be repeated and it expects precisely one integer. Default value: 100

The performance of CP2K with and without the comm thread enabled is shown in figure 7. The benchmark used here is a 6144 atom calculation of liquid water using the new linear-scaling DFT[6] implementation, which is strongly dominated by DBCSR operations as it performs SCF via iterations on the density matrix. Here we see that using a Communications thread can give up to a 13% improvement when used with two threads up to 20% wthen using six threads, even on relatively modest numbers of processors. At 2304 cores and above, underloading the comm thread to 80% is also shown to give an improvement over the default settings.

3.3 Data preparation for multiplication

The third aspect of DBCSR that was investigated was the preparation step that occurs before every matrix multiplication, found in the DBCSR routine `make_images`. This essentially subdivides each local sub-matrix into a 2D-array of 'images', such that the

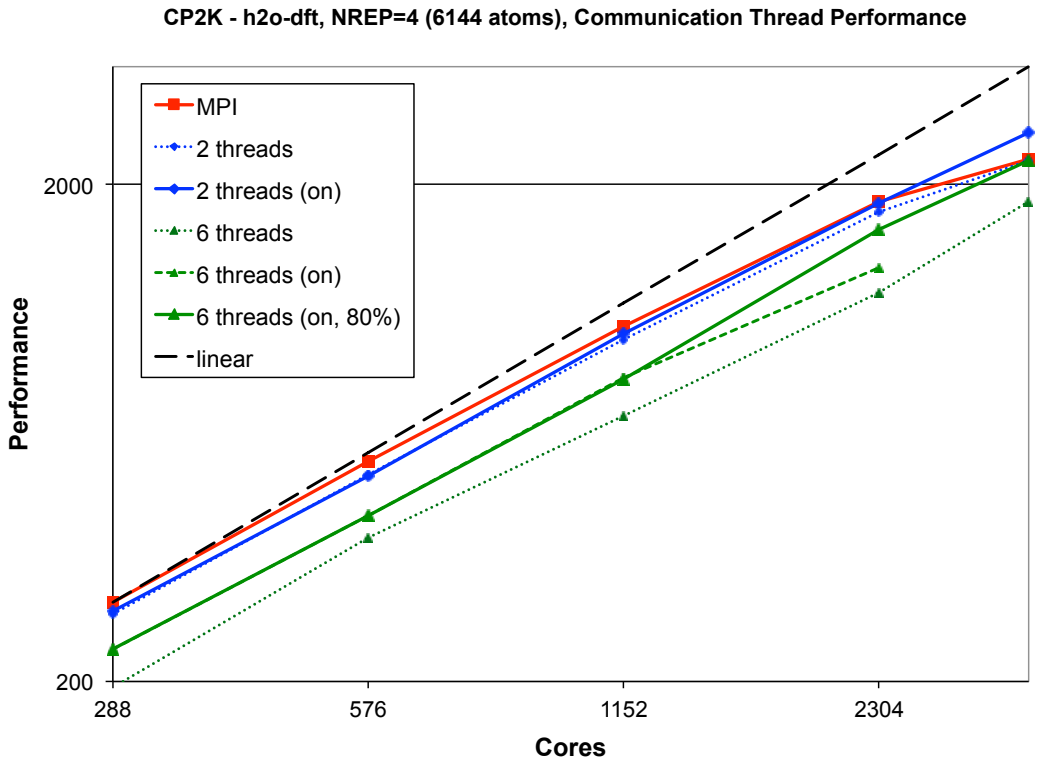

Figure 7: Performance of linear scaling DFT with and without communications thread enabled (24-core XE6, CSCS)

global set of images is square, and thus suitable for Cannon's algorithm. For example, when using 6 MPI processes (arranged in a 2x3 grid), each process will have an array of 3x2 images, giving a total of 6x6 images globally. In addition, the first column and row shifts (pre-shifts) of Cannon's algorithm are performed. If a matrix is symmetric, blocks which are stored only once in the initial matrix are desymmetrized and stored twice as they may be sent to different processes. Once the destination (either local or remote) is determined, data is copied into buffers and sent to the recieving process. The recieved data in general will have come from a variety of different processes, and so the blocks are sorted into the correct CSR order and the index is rebuilt using the `dbcsr_finalize` routine, normally used for merging work matrices from multiple threads together.

Using the CrayPAT API to profile sub-regions within this routine the largest contribution to the runtime comes from `dbcsr_finalize`. A special case of this routine was written to account for the fact that rather than merging data from several threads' work matrices, we instead have only a single work matrix which contains unsorted blocks. We also avoid having to account for the case were the matrix being merged into already has existing blocks, since all the blocks making up the new image come from the MPI recieve buffer. As a result of this, and a number of other smaller OpenMP optimisations, results in a speedup in cases where there is more than one image (i.e. the number of MPI processes is not a square) - 8% faster for 128 MPI x 2 OMP, and 43% faster for 32 MPI x 8 OMP.

Threads	1	2	4	8
sort+merge	0.99	1.35	1.89	1.84
sort only	0.99	1.65	2.67	2.78
Magny-Cours	0.99	1.54	2.43	2.48
Parallel data	0.99	1.44	2.37	2.94

Table 2: Speedup vs CP2K sort on 1 thread (10,000 elements)

The majority of the time taken by the new finalize routine, and the routine used when there is only one image is taken up by sorting the recieved blocks into CSR order (first by row, then by column). This is currently done using an efficient quicksort, but only uses a single thread. A threaded sort (parallel mergesort) was implemented following [11] using the existing quicksort as the base case an the parallel merge from [12].

This gives poorer than expected results for typical list sizes of 10,000 elements (see table 2). One reason for this is related to the shared 'module' structure of the AMD Interlagos processor. If we run on the Magny-Cours processor, where every core has it's own FPU, instruction unit etc. we see much better performance (see third row of the table). In addition since we are sorting an array that has been written by MPI (i.e. a single thread), there is a penalty in accessing the data. If the data were written in parallel, so each thread has it's own portion of the array in cache at the start of the sort, better performance is achieved. However, this is not possible in practice and only serves to put an upper limit on the performance of the parallel sort. Larger array sizes give greater speedups (e.g. 2.78x on 4 threads with 100,000 elements), however, with smaller sizes such as 1,000 there is no speedup at all. With more work it might be possible to construct a heuristic on when to use the parallel sort (and with how many threads), but for now the CP2K sort has been left unchanged.

4 Recommendations for users

In addition to providing recent versions of CP2K on HECToR (see http://www.hector.ac.uk/support/documentation/software/cp2k/ for details) containing the improvements described in this report, below are some general points of advice for users wishing to achieve best performance from CP2K on HECToR. Of course, nothing substitutes for careful benchmarking on a shortened version of the problem you are trying to solve (for example performing only a single MD step, or a reduced number of SCF cycles).

- Using the right number of MPI processes

  Firstly, try to choose a number of MPI processes that is a square, to maximise the performance of DBCSR (see section 3.2. Secondly, use whole nodes (multiples of 32 cores on HECToR Phase 3). Thirdly, if possible choose a number of processes that is a power of two. Following these guidelines, better performance might be obtained on 64, 256, 1024 MPI processes, depending on problem size.

- Use libsmm

  As discussed in section 3.1.1, using libsmm can give significant performance improvements over regular BLAS DGEMM for simulations involving small block

sizes (elements with low atomic number, small basis sets). The centrally-installed CP2K package is linked with libsmm, or if you wish to build your own version from source, pass `-D__HAS_smm_dnn` to the compiler, and link the library in `/usr/local/packages/cp2k/2.3.15/libs/libsmm/`

- Use OpenMP

  For large problem sizes in particular, or on large numbers of cores, using perhaps 2 or 4 OpenMP threads per MPI process may give better performance than pure MPI. OpenMP may also be of use for problems requiring large amounts of memory, where otherwise nodes would hve to be underpopulated. An OpenMP-capable binary of CP2K (`cp2k.psmp`) is provided in the CP2K package on HECToR. If using OpenMP, consider experimenting with the communication thread settings as described in section 3.1.3, which may provide further speed improvements. In some cases (particularly on small numbers of cores, where the compute to communication ratio is high, it may be beneficial to switch the comm thread off).

Finally, for any issues regarding CP2K on HECToR please contact the HECToR helpdesk `helpdesk@hector.ac.uk`.

5 Conclusion

After three consecutive rounds of dCSE funding we have shown that real improvements in the performance of CP2K have been delivered to HECToR users, in addition to the functionality of the code being extended over time by other developers. We consider that the extended funding of an HPC expert to work closely with both code developers and users is an excellent model for sustainable applications development. Evidence of this is the large number of users who have been assisted in using the code on HECToR, as well as further development projects including an MSc student[13] working on improving GPU support in the code, and 16 months of PRACE funding[14][15] to improve scalability into the petascale regime. Both these projects are also of benefit to HECToR users and would have been impossible without initial support from the dCSE scheme to build familiarity with the code.

6 Acknowledgment

The project was made possible thanks to ongoing support from Dr. Ben Slater (University College London, CP2K User), Prof. Joost VandeVondele (ETH Zurich, CP2K Developer), and Prof. Jurg Hutter (University of Zurich, CP2K Developer).

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR - A Research Councils UK High End Computing Service - is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. http://www.hector.ac.uk

References

[1] CP2K website - Open Source Molecular Dynamics, http://www.cp2k.org

[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. Chassasing and J. Hutter, Comp. Phys. Comm. 167 (2005) 103-128

[3] Improving the performance of CP2K on HECToR: A dCSE Project, Iain Bethune, 2009, http://www.hector.ac.uk/cse/distributedcse/reports/cp2k/cp2k_final_report.pdf

[4] Improving the scalability of CP2K on multi-core systems: A dCSE Project, Iain Bethune, 2010, http://www.hector.ac.uk/cse/distributedcse/reports/cp2k02/cp2k02_final_report.pdf

[5] U. Borstnik, J. VandeVondele and J. Hutter, In preparation, 2012.

[6] Linear scaling self-consistent field calculations with millions of atoms in the condensed phase. J. VandeVondele, U. Borstnik and J. Hutter, In preparation, 2012

[7] HECToR: UK National Supercomputing Service, http://www.hector.ac.uk

[8] Cannon, L. E. Ph.D. thesis, 1969, AAI7010025

[9] Separable dual-space Gaussian pseudopotentials. Goedecker, S; Teter, M; Hutter, J. PHYSICAL REVIEW B, 54 (3), 1703-1710 (1996). http://dx.doi.org/10.1103/PhysRevB.54.1703

[10] Generalized gradient approximation made simple. Perdew, JP; Burke, K; Ernzerhof, M. PHYSICAL REVIEW LETTERS, 77 (18), 3865-3868 (1996). http://dx.doi.org/10.1103/PhysRevLett.77.3865

[11] Efficient Implemention of Sorting on Multi-Core SIMD CPU Architecture, Chhugani et al, Intel Research, http://pcl.intel-research.net/publications/sorting_vldb08.pdf

[12] A Benchmark Parallel Sort for Shared Memory Multiprocessors. R. Francis and I. Mathieson, IEEE Transactions on Computers, 37:1619-1626, 1988.

[13] Optimising the DBCSR GPU implementation, J. Chetty, 2011, http://www.epcc.ed.ac.uk/wp-content/uploads/2011/11/JayChetty.pdf

[14] Million Atom KS-DFT with CP2K, I. Bethune, A. Carter, X. Guo and P. Korosoglou, 2011, http://prace-portal.cscs.ch/uploads/tx_pracetmo/CP2K.pdf

[15] Mixed-mode MPI/OpenMP for the CP2K User Community, I. Bethune, A. Carter, K. Stratford, P. Korosoglou and M-F. Iozzi, 2012, In preparation