Evaluating CP2K on Exascale Hardware: Intel Xeon Phi

Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Evaluating CP2K on Exascale Hardware: Intel Xeon Phi

Fiona Reid[a], Iain Bethune[a]*

[a]*EPCC, The University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh, EJ9 3JZ, UK*

Abstract

CP2K, a popular open-source European atomistic simulation package has been ported to the Intel Xeon Phi architecture, requiring no code modifications except minor bug fixes. Benchmarking of a small molecular dynamics simulation has been carried out using CP2K's existing MPI, OpenMP and mixed-mode MPI/OpenMP implementations to achieve full utilisation of the Xeon Phi's 240 virtual cores. Running on the Xeon Phi in native mode, CP2K is approximately 4x slower than utilising all 16 cores of a Xeon E5-2670 Sandy Bridge dual-socket node. Careful placement of processes and threads on the virtual cores of the Xeon Phi was found to be crucial in achieving good performance.

Analysis of the benchmark results has led to the identification of a number of bottlenecks which must be resolved to achieve competitive performance on the Xeon Phi, which will be carried out as a follow on to the work reported here.

Application Code: CP2K

1. Introduction

CP2K [1] is a popular, open-source program for atomistic simulation. It provides many levels of theory ranging from classical potential models, DFT using a mixed Gaussian and plane waves approach known as QUICKSTEP[2], to hybrid DFT and post-HF methods (MP2, RPA). In addition, many simulation methods are supported including, molecular dynamics, Monte Carlo, path integrals, nudged elastic band and free energy calculations. These features have made CP2K an increasingly popular tool across the fields of materials science, computational chemistry, solid-state physics and biochemistry. It is widely used throughout Europe, including on the current PRACE RI systems, as well as national and local-scale HPC systems.

As a result, it is of interest to evaluate the performance of CP2K on potential future HPC architectures to ensure the continued viability of the code as HPC providers start looking towards Exascale. One such new architecture is the Intel Xeon Phi [3] platform, which combines the high performance and low power of e.g. GPUs, with the ease of use of standard x86 CPUs. The Xeon Phi architecture is known as the Intel Many Integrated Core (MIC) co-processor, and throughout this report we will use Xeon Phi and MIC interchangeably.

To ease application porting, Intel supports several standard programming models on the Xeon Phi including the widely used MPI, OpenMP and OpenCL, as well as proprietary interfaces like Intel Thread Building Blocks and Cilk+. Since CP2K already has a mixed-mode MPI and OpenMP parallelisation strategy, we planned to port the code to the Xeon Phi Platform, and investigate the performance and scalability that can be achieved without significant code modification or tuning. As a result we have gained a good understanding of what further development is needed to enable the code to fully take advantage of the MIC architecture and make is suitable for deployment on large-scale Xeon Phi-based HPC systems in the future. Additional work within PRACE investigate the suitability of OpenCL and OpenACC exploiting accelerators within CP2K [4].

* Corresponding author. *E-mail address*: ibethune@epcc.ed.ac.uk

Test System

The following system, located at the Swiss National Supercomputing Centre (CSCS) in Lugano, was used to obtain the results presented in this report.

Dommic is a tightly clustered Linux system running CentOS 6.3. The system is made up of 6 Intel nodes comprising two eight-core Intel Xeon E5-2670 processors running at 2.6 GHz with 32 GB of memory per node, giving a total of 96 cores and 192 GB of memory. Each node allows shared memory jobs using up to 16 threads to be run, and MPI can be used both within and between nodes. It is also possible to run up to 32 shared memory threads per node with hyper-threading enabled, but this was found to give poor performance for CP2K. Each Dommic node has 2 Xeon Phi 5110P co-processor cards connected to it. Each Xeon Phi card contains 60 physical cores running 4 virtual threads per core giving access to a total of 240 threads per card. The clock speed of the cores is 1.053 GHz and each card has 8 GB of memory with a memory bandwidth of 320 GB/s. For more details on the hardware specifications please see [3]. In the remainder of this report the Xeon E5-2670 will often be referred to as the host because it hosts the Xeon Phi MIC cards.

The Dommic nodes have the Intel Cluster Studio XE 2013 software installed, which includes compilers, tools and debuggers for both the host nodes and the Xeon Phi cards. To date, only Intel compilers can be used to compile code specific for the MIC architecture and as such our first step was to compile and test CP2K with the Intel compiler suite on the host nodes.

2. Porting and testing

Porting CP2K to the Intel software stack

We initially compiled CP2K on Dommic (the host) using `gfortran` as the code is known to work well with the GNU compiler suite and thus it was determined to be a good starting point and means of comparing output should problems be encountered with the Intel build. Four different versions of the code were compiled which will subsequently be referred to as SOPT, SSMP, POPT and PSMP. These correspond respectively to, a pure serial version, a pure OpenMP version, a pure MPI version and a mixed mode MPI/OpenMP version. Each version was tested for correctness by running the CP2K regression test suite. The test suite comprises 2384 tests, sampling most features of the code, and can be run both in parallel and serial. More details on the regression tests can be found at [5]. These tests are run before any changes are committed to the code repository to ensure that the changes have not damaged the code base.

Once the GCC compiled version passed all the regression tests we moved on to building a version of CP2K with the Intel compiler suite. We began by building a host version, i.e. a version that will run only on the Xeon processors, rather than the MIC co-processor. The host nodes have more memory, tools, debuggers and libraries available and thus resolving any compiler or code issues should be considerably easier on the host compared with the Xeon Phi cards. Once the code runs on the host nodes compiling up a Xeon Phi specific version *should* be relatively straightforward.

Version 13.1.0 of Intel `ifort` compiler and Intel MPI version 4.1.0.030 were used throughout this work. Our initial runs of the Intel compiled host versions of CP2K failed a number of the regression tests. The SOPT and POPT versions generally only failed one test (`Fist/regtest-15/fixed_points.inp` described in item 3 below) but the SSMP and PSMP versions generally failed a much larger number (typically between 65-80 failures). Furthermore, on re-running the regression tests with the threaded versions (SSMP, PSMP) of the code we discovered that the results were not numerically stable. The difference between one test result and a new run of the tests was often much larger than would be expected from simple rounding errors that occur due to different orders of computation in the parallel versions. These numerical instabilities were caused by the non thread safety of the Intel FFT library (see item 4 below for more detail). To identify as many potential bugs as possible, debug versions of the code were also compiled with the "`-check all`" flag. This flag applies all compiler checks (e.g. bounds checking, temporary arguments, formatting, pointers, output conversion & uninitialized variables) and also removes any optimisation. Adding this flag caused a large number (>1000 initially) of the regression tests to fail and allowed a number of non critical but potentially damaging bugs to be fixed in the code.
We encountered a number of different issues (compiler bugs, library bugs and code bugs) when building CP2K with the Intel compiler suite. The main issues encountered and the solutions to them are summarised below:

1. Fixed several CP2K source files by adding missing `RECURSIVE` attributes to subroutines that form part of recursion loops. This was initially tested by running the code with the `–recursive` flag which forces the compiler to treat all routines as if there were recursive.

2. Bug fixes (source files changed: `src/qmmm_force_mixing.F` and `src/qs_collocate_density.F`) to ensure pointers were nullified prior to use. Unlike GCC, the Intel compiler doesn't set pointers to `NULL` before use and thus the code will crash if attempting to read from a pointer before it's been nullified or given an appropriate value.

3. The `Fist/regtest-15/fixed_points.inp` regression test fails when it attempts to read in the input file. The error message obtained was of the form:

```
forrtl: severe (8): internal consistency check failure, file for_get.c, line 1446
Image              PC                Routine            Line        Source
cp2k.sopt          000000000C06C1AE  Unknown               Unknown  Unknown
cp2k.sopt          000000000C06AC46  Unknown               Unknown  Unknown
cp2k.sopt          000000000C0289D2  Unknown               Unknown  Unknown
cp2k.sopt          000000000BFDD3CB  Unknown               Unknown  Unknown
cp2k.sopt          000000000C047556  Unknown               Unknown  Unknown
cp2k.sopt          000000000C006A4C  Unknown               Unknown  Unknown
cp2k.sopt          000000000C007BAD  Unknown               Unknown  Unknown
cp2k.sopt          000000000BFFA98F  Unknown               Unknown  Unknown
cp2k.sopt          000000000BFF96CC  Unknown               Unknown  Unknown
cp2k.sopt          00000000004A8320  cp_parser_methods        1350  cp_parser_methods.F
cp2k.sopt          000000000049DD84  cp_parser_methods         808  cp_parser_methods.F
cp2k.sopt          000000000090DD10  input_parsing_mp_         651  input_parsing.F
cp2k.sopt          00000000009064E1  input_parsing_mp_         306  input_parsing.F
cp2k.sopt          0000000000903C6B  input_parsing_mp_         226  input_parsing.F
cp2k.sopt          0000000000903C6B  input_parsing_mp_         226  input_parsing.F
cp2k.sopt          0000000000903C6B  input_parsing_mp_         226  input_parsing.F
cp2k.sopt          0000000000903C6B  input_parsing_mp_         226  input_parsing.F
cp2k.sopt          0000000000903C6B  input_parsing_mp_         226  input_parsing.F
cp2k.sopt          00000000007AAC39  input_cp2k_mp_cre         109  input_cp2k.F
cp2k.sopt          00000000004298DF  cp2k_runs_mp_cp2k         188  cp2k_runs.F
cp2k.sopt          00000000004411DF  cp2k_runs_mp_run_        1147  cp2k_runs.F
cp2k.sopt          000000000042808E  MAIN__                    307  cp2k.F
cp2k.sopt          0000000000423EDC  Unknown               Unknown  Unknown
libc.so.6          000000364EA1ECDD  Unknown               Unknown  Unknown
cp2k.sopt          0000000000423DD9  Unknown               Unknown  Unknown
```

   The line in the input file which the code fails to read is very long (218 characters). It turns out that CP2K has exposed a bug in the Intel Compiler, which has now been fixed in the June 2013 release of the Intel Composer suite.

4. When testing the Intel SSMP and PSMP versions on the host a large number (nearly 50%) of the regression tests initially failed with segmentation faults. These crashes occurred whenever more than one OpenMP thread was used. The reason was identified to be a problem in the Intel MKL library when using the FFTW3 interface. Essentially, the FFT execute functions are not by default thread-safe which is the case for a real FFTW library. The MKL documentation at [6] gives more detail on this and advises that one can set a parameter allowing multiple threads to share a plan, e.g.

```
#include "fftw3.h"
#include "fftw3_mkl.h"
fftw3_mkl.number_of_user_threads = 4; plan = fftw_plan_dft(...);
```

   However, this workaround can only be performed in C and as CP2K is a Fortran code it means that we can't use MKL with the FFTW3 interface until such a workaround exists for Fortran. This has been submitted to Intel as a feature request (Intel ID DPD200243422) and will appear a future release of MKL. As this Beta release was not available to us at the time of writing the results presented in this report have all be produced using one of FFTW 3.3.3 or FFTW 3.3.4 (see the discussion re. FFTW v MKL performance below). An additional check has also been added to CP2K such that the code will now abort if Intel MKL's FFTW3 wrapper is used in a multi-threaded build.

5. Fixed a bug in the `dbcsr_lib` part of the code, which resulted in incorrect data sizes being sent and received.

6. Fixed a bug in the `dbcsr_lib` part of the code to avoid array overruns when expanding a zero-length array.

7. Discovered that using the `–heap-arrays` (to ensure all arrays get placed on the heap) and the `–openmp` compiler flags together causes more than 50% of the regression tests to crash for the SSMP and PSMP versions if more than one thread is used. This issue has been reported to Intel for further investigation. The suspicion is that the `–heap-arrays` flag is somehow interfering with the OpenMP runtime such that variables that should be thread private (or on a thread's private stack) are getting placed on the heap and this results in multiple threads accessing the same data.
8. When the `libgrid` library was added to the code many previously working tests failed with segmentation faults. If building a threaded version of CP2K (i.e. SSMP or PSMP) then libgrid needs to be built with the `-openmp` flag.
9. In addition to the above, a large number of CP2K source files were altered to ensure that variables used in conditionals that always evaluate to false were initialised before use. This is not technically a bug, as the variable is never read before being written. However, the Intel compiler would report these issues with the "`–check all`" flag turned on and it was felt best to eliminate them so that any real bugs could be distinguished.

Due to the issues mentioned in item 4 regarding the thread-safety of MKL we have used FFTW 3.3.3 for all the host builds.

After making the changes detailed above, CP2K passed all except one regression test for each of the versions tested. The failing test is `fixed_points.inp` as described in item 3 above. The performance of the host version of CP2K was then benchmarked and the results can be found in section 3.

CP2K on Intel Xeon Phi

Once we had compiled and tested CP2K with the Intel software stack on the host we built a native version for the MIC architecture. In theory, this should be as simple as adding `–mmic` to the compiler flags and then rebuilding the code and any associated libraries. In practice it is somewhat more complex than this since the Xeon Phi cards do not have any compilers installed and thus we had to cross-compile from the host.

When building many codes or libraries the build process may generate and run executables in order to test or optimise the build. When building for the host this is fine as the build and host architectures are the same but when building for the Xeon Phi the binaries generated on the host cannot be run on the host as they contain MIC-specific instructions. Many build systems handle this problem by setting cross-compiler flags where the binaries are simply not executed. However, for some of the libraries that CP2K requires (e.g. `libxc` and `libsmm_dnn`) executing the binaries is a critical part of the build process as they generate some of the source code or are used for auto-tuning and an alternative approach must be found. Two different approaches were used when building such libraries. One approach (used to build `libxc`) involved compiling the code on the host and when the build fails due to attempting to run a Xeon Phi binary switching to the Xeon Phi and running the binary before resuming the compilation. The other approach (used to build `libsmm_dnn`) involved modifying the build scripts such that lines that execute binaries intended for the Xeon Phi are preceded by "`ssh mic0`" such that the binary is executed on the Xeon Phi and not on the host.

Prior to building the native version of CP2K the performance of the different FFT libraries was investigated on both the host and Xeon Phi cards. The libraries tested were Intel MKL (using the FFTW3 interface), FFTW version 3.3.3 and also FFTW 3.3.4, a pre-release version supplied by Matteo Frigo which had been optimised specifically for the Xeon Phi. This investigation was carried out in order to understand the impact of not being able use Intel MKL's FFTW3 interface due to the thread-safety issues discussed above. The performance of the different FFT libraries was measured using two simple benchmarks, which perform either 1D or 3D FFTs. Each benchmark performs `NREPS` forward and inverse FFTs and measures the time taken. The value of `NREPS` is chosen such that it is large enough to give reliable timing but small enough that it runs in a few minutes. The time taken to re-scale the transformed results is also computed. The results for the 1D and 3D benchmarks with problem sizes of N=128 and N=256 elements are given by Table 1 and Table 2 respectively. For the 3D benchmarks the problem size is N^3. These values of N were chosen as they are typical of FFT grid sizes used in CP2K.

N	host FFTW 3.3.3	host MKL	MIC FFTW-3.3.3	MIC FFTW-3.3.4	MIC MKL
128	0.0929	0.0928	12.1598	8.3509	8.4219
256	0.1981	0.1929	24.1600	17.1924	17.3632
**128	0.0542	0.0572	1.7499	0.3292	0.3891
**256	0.1218	0.1152	3.8131	0.6374	0.7139

Table 1: Time (seconds) for 100000 reps of the 1D FFT benchmark, ** indicates that the time taken for data re-scaling was subtracted and gives the cost of the FFT's alone. "Host" indicates that the results were obtained on the host processors and MIC indicates that the results were obtained on the Xeon Phi cards running in native mode.

N	host FFTW 3.3.3	host MKL	MIC FFTW-3.3.3	MIC FFTW-3.3.4	MIC MKL
128	1.8241	1.6153	58.9320	43.2592	36.3927
256	7.9546	8.6894	267.2964	222.3350	165.8646
**128	1.6725	1.3278	35.5704	19.1855	11.1212
**256	7.3357	6.1707	169.4342	117.3201	65.9620

Table 2: Time (seconds) for 20 reps (N=128) or 10 reps (N=256) of the 3D FFT benchmark. **indicates that the scaling was removed and gives the cost of the FFT's alone.

Examining the results in Table 1 for the 1D FFT benchmark we can see that for the host processors the choice of FFTW library for a 1D transform is not important, both Intel MKL and FFTW 3.3.3 have similar performance. However, this is not the case for the Xeon Phi. Here the best performance (scaling removed) is obtained with FFTW 3.3.4 with Intel MKL being up to 18% slower. The performance of the release version of FFTW 3.3.3 is up to 6 times slower (as it is not optimised for the MIC SIMD unit) and thus for 1D FFTs we should use FFTW 3.3.4.

Now looking at the results for the 3D FFT benchmark (Table 2) we see for both the host and Xeon Phi the Intel MKL library outperforms FFTW for both problem sizes. On the host MKL is up to 20% faster than FFTW 3.3.3. On the Xeon Phi again FFTW 3.3.3 gives by far the poorest performance with MKL being up to 78% faster than FFTW 3.3.4.

As mentioned previously, we were unable to use Intel MKL's FFTW3 interface due to the thread safety problems and thus all our CP2K benchmarks have used FFTW 3.3.3 on the host or FFTW 3.3.4 on the Xeon Phi as these versions were the ones found to give the best performance whilst allowing the multi-threaded (SSMP and PSMP) version of the code to run correctly. When the thread-safe version of Intel MKL's FFTW3 interface becomes available for Fortran then we could potentially expect an improvement in performance of the FFT computations of over to 70% to be obtained for 3D FFT containing 128^3 or 256^3 elements.

For simplicity, the native version of CP2K was initially built without any additional libraries (e.g. without `libint`, `libxc`, `libsmm_dnn`). This enabled us to later introduce these libraries one at a time to make sure they didn't cause any unexpected problems. Three versions of CP2K were compiled for the Xeon Phi, SSMP, POPT and PSMP. As serial execution will not give good performance on the Xeon Phi, the SOPT version was not compiled. The snippet below shows the differences between the CP2K 'arch file' for the host and Xeon Phi SSMP version when compiled without any additional libraries:

```
< FC       = ifort -openmp
< LD       = ifort -openmp
> FC       = ifort -openmp -mmic
> LD       = ifort -openmp -mmic
< FFTW_DIR = /users/fiona/fftw3_intel_threaded
> FFTW_DIR = /users/fiona/fftw3.3.4_intel_mic
<       -Wl,--start-group  $(MKLROOT)/lib/intel64/libmkl_intel_lp64.a \
<       $(MKLROOT)/lib/intel64/libmkl_sequential.a \
<       $(MKLROOT)/lib/intel64/libmkl_core.a -Wl,--end-group -lpthread -lm \
>       -Wl,--start-group  $(MKLROOT)/lib/mic/libmkl_intel_lp64.a \
>       $(MKLROOT)/lib/mic/libmkl_sequential.a \
>       $(MKLROOT)/lib/mic/libmkl_core.a -Wl,--end-group -lpthread -lm \
```

Essentially all the user needs to do when compiling a native version of CP2K for the Xeon Phi is to add the `-mmic` flag and ensure that any libraries linked by the code are also built for Xeon Phi. For the full arch files

used in this work see Appendix A. Appendix B contains further compilation details for CP2K, its associated libraries and regression test suite on the Intel host and Xeon Phi (native mode).

Once SSMP, POPT and PSMP binaries had been built for the Xeon Phi we ran the code through the regression test suite. The regression test script is written in `bash` but no `bash` shell is available on the Xeon Phi cards. The only shell available on the Xeon Phi cards is `sh` which means we would either have to convert the regression test script to `sh` (non trivial due to lack of arrays and other features in `sh`) or alter the script such that the regression tests are launched by prepending the execution line with "`ssh mic0`" as was done for the `libsmm_dnn` library, and run the script on the host. The tests were then run for the POPT and SSMP versions of the code. A large number of the tests initially failed. A summary of the test output for POPT and SSMP is given below:

POPT (30 MPI procs)

```
number of FAILED  tests 734
number of WRONG   tests 0
number of CORRECT tests 0
number of NEW     tests 1652
number of         tests 2386
```

SSMP (30 OpenMP threads)

```
number of FAILED  tests 431
number of WRONG   tests 0
number of CORRECT tests 0
number of NEW     tests 1955
number of         tests 2386
```

A few tests failed due to running out of time as we initially set a limit of 3000 seconds per test, but a large number failed with errors of the form:

```
forrtl: severe (154): array index out of bounds
```

Using a combination of print statements and the Phi enabled version of GDB described in [7] we narrowed the issue down to a call to `dfftw_execute_dft` (in FFTW) made from the subroutine `fftw3_execute_workshare_dft`.. A stack trace produced by the Xeon Phi enabled GDB gave the following:

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000acd6cc4 in t3fv_10 ()
(gdb) bt
#0  0x000000000acd6cc4 in t3fv_10 ()
#1  0x000000000acfaadf in apply ()
#2  0x000000000aba0531 in apply ()
#3  0x000000000a5a1d3a in fftw3_workshare_execute_dft (plan=263171968, plan_r=263172672,
split_dim=40, nt=16, tid=11, input=..., istride=1600, output=..., ostride=1600)
    at /project/g43/fiona/cp2k/cp2k/makefiles/../src/fft_lib/fftw3_lib.F:515
#4  0x000000000a5a437e in L_fftw33d__598__par_region0_2_218 () at
/project/g43/fiona/cp2k/cp2k/makefiles/../src/fft_lib/fftw3_lib.F:622
#5  0x00007ffff743367f in __kmp_invoke_microtask () from /lib64/libiomp5.so
#6  0x00007ffff741189a in __kmp_invoke_task_func (gtid=266547872) at
../../src/kmp_runtime.c:8610
#7  0x00007ffff7410366 in __kmp_launch_thread (this_thr=0xfe332a0) at
../../src/kmp_runtime.c:6821
#8  0x00007ffff7433a6b in __kmp_launch_worker (thr=0xfe332a0) at ../../src/z_Linux_util.c:864
#9  0x00007ffff7bc8bbe in start_thread () from /lib64/libpthread.so.0
#10 0x00007ffff6f1263d in clone () from /lib64/libc.so.6
```

The root cause of this problem is that the FFT arrays are not correctly aligned to use the MIC vector instructions (must be 64-byte rather than the default 16-byte aligned). This is a stricter alignment requirement than is supported by the Intel Compiler, so to achieve it, we must use the `fftw_malloc()` routine, which is only available via the Fortran 2003 interface to FFTW. To avoid (for now) re-implementing the FFT routines to this interface, we linked CP2K with FFTW 3.3.3 which works around the problem and allows many of the failing tests to run at the expense of a slower execution time.

After recompiling the code with FFTW 3.3.3 we re-ran the POPT regression tests again. The array out of bounds errors no longer occurred but 80 tests still failed. Of these, 10 tests failed with segmentation violations resulting from the numerical behaviour of the Xeon Phi causing DBCSR to incorrectly identify a full matrix as a sparse matrix, as the calculation of the matrix occupancy returns `1.0-EPSILON` rather than strictly `1.0`. We have fixed this bug in the current CP2K SVN, reducing the number of failing tests to 66. Several of the tests also failed due to being run on too many processors. The Xeon Phi regression tests were run on 30 MPI processes to reduce the runtime, but some of the regression tests are designed to work only on a maximum of two processors and thus need to be run separately from the rest of the tests[†]. Around 20 tests failed with the following error:

[†] This can be achieved by changing the number of processes used from 30 to 2 and then using the `–restrictdir` flag to the regression test suite to restrict the test runs to those from a particular directory.

```
*****************************************************************************
*** 13:45:35 ERRORL2 in ps_wavelet_types:ps_wavelet_create processor 0  ***
*** :: err=-300 Poisson solver for non cubic cells not yet implemented  ***
*****************************************************************************
CP2K| Poisson solver for non cubic cells not yet implemented
CP2K| Abnormal program termination, stopped by process number 0
```

Similarly to the root cause of the previous error, this is also caused by the numerical behaviour of the MIC, where calculation of the cell vectors from cell angles of 90° would result in cell lengths that did not have exactly identical floating point representation. This bug is also now fixed in the CP2K SVN. A further 20 tests remain which still fail when run on the Xeon Phi, and work is ongoing to resolve these.

Affinity testing on the Xeon Phi

Our initial testing of CP2K on the Xeon Phi gave very poor performance when the PSMP version of the code was used. When compared with the SSMP or POPT versions the PSMP version was found to take up to 8 times longer than expected. The reason for this was found to be due to the process and thread affinity that was used by default. It appears that the default affinity settings for mixed mode MPI/OpenMP codes may not also be ideal. From our tests we discovered that instead distributing the MPI processes and threads across all the physical cores we actually ended up with just 2 physical cores being used and these cores were heavily over-subscribed. The thread affinity that is used by a code can be verified by setting the following environment variable prior to executing the code:

```
export KMP_AFFINITY=verbose
```

When the code is run with this setting, the standard output/error will contain information regarding the actual placement of the processes and threads (if used) in your code. The resulting output is rather verbose and lists the affinity of all 240 virtual threads on the card and thus it's often simpler to `grep` for "`Internal`" or "`Internal thread`" in order to find out *where* the processes/threads you are actually using have been bound. With the PSMP version on 30 MPI processes each running 2 OpenMP threads the following output was obtained:

```
/apps/dommic/mic/intel/impi/4.1.0.030/mic/bin/mpirun -np 30 /users/fiona/cp2k.psmp
H2O-64_test.inp &> test.txt
grep "Internal" test.txt
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {1}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {5}
<SNIP>
```

In the code output, a total of 60 such lines appeared reporting the affinity of the 30 MPI processes and their 2 threads. For each MPI process, thread 0 is bound to proc set 1 (i.e. the first virtual thread on the first physical core) and thread 1 is bound to proc set 5 (i.e. the first virtual thread on the second physical core). This means that 30 threads are actually running on a single virtual thread of the first two physical cores with 58 physical cores remaining totally idle. This is clearly not what we want and explains the very poor performance that was initially observed when running the PSMP version. It should be possible to use the pre-set affinities, e.g. `compact`, `scattered`, `balanced` but with these settings appeared to be ignored and thus the we must explicitly specify the list of virtual threads that each MPI process should use. This can be done via the `proclist` option to `KMP_AFFINITY` for threaded codes or via the `I_MPI_PIN_PROCESSOR_LIST` environment variable for pure MPI codes. Examples of how to do this for the POPT, SSMP and PSMP versions of CP2K are given below:

POPT (4 MPI procs):

```
mpirun -np 4 -env I_MPI_PIN_PROCESSOR_LIST=1,5,9,13 cp2k.popt H2O-64.inp > popt_4_procs.out
```

SSMP (4 OpenMP threads):

```
export KMP_AFFINITY=verbose,granularity=fine,proclist=[1,5,9,13],explicit
export OMP_NUM_THREADS=4
cp2k.ssmp H2O-64.inp > ssmp_4_threads.out
```

PSMP (2 MPI procs and 2 OpenMP threads):

```
export OMP_NUM_THREADS=2
mpirun -prepend-rank -genv LD_LIBRARY_PATH path_to_the_mic_libs \
-np 1 -env KMP_AFFINITY verbose,granularity=fine,proclist=[1,5],explicit -env OMP_NUM_THREADS
${OMP_NUM_THREADS} $CP2K_BIN/cp2k.psmp H2O-64.inp : \
-np 1 -env KMP_AFFINITY verbose,granularity=fine,proclist=[9,13],explicit -env OMP_NUM_THREADS
${OMP_NUM_THREADS} $CP2K_BIN/cp2k.psmp H2O-64.inp &> psmp_2_procs_2_threads.out
```

For the POPT and SSMP versions setting the process or thread affinity is relatively straightforward. A list of virtual thread ID's is passed to the environment variable `I_MPI_PIN_PROCESSOR_LIST` or to the `proclist` part of `KMP_AFFINITY` and the MPI processes or threads are then bound to the ID's in the specified list. For the PSMP version each MPI process needs to have a separate entry listing (as shown above) where the threads associated with that particular process should be executed. For reference, on the Xeon Phi the 60 physical cores and their 4 virtual threads are numbered as follows:

1,2,3,4 – physical core 1
5,6,7,8 – physical core 2
…
233,234,235,236 – physical core 59
237,238,239,240 – physical core 60

Having discovered that the default affinity settings used in native mode may not always be optimal we carried out some further testing to determine the optimal affinity settings for the different versions of CP2K. In general we want to distribute the work across as many physical cores as possible whilst also maximising data and thread locality when running the PSMP version. We began by testing the POPT version with 30 MPI processes. We compared the performance of several different affinity settings; default, `compact`, `scattered` and `balanced`. With a compact affinity all the virtual threads on a physical core are used before moving on to the next physical core. So, with 30 MPI processes a compact affinity will fill the first seven physical cores with four processes with the eighth physical core having two processes. The mapping between MPI ranks and virtual thread ID's on the Xeon Phi cards is as follows:

```
MPI rank:           0 1 2 3       4 5 6 7       …       24 25 26 27   28 29
Virtual thread ID:  1 2 3 4       5 6 7 8       …       25 26 27 28   29 30
```

With a scattered affinity one virtual thread from each physical core is used in a round robin fashion. The mapping between MPI ranks and virtual thread ID's is:

```
MPI rank:           0 1 2  3  4  5  6  7 … 24  25  26  27  28  29
Virtual thread ID:  1 5 9 13 17 21 25 29 … 97 101 105 109 113 117
```

When using only 30 MPI processes scattered and balanced affinities are identical. In general scattered and balanced affinities only differ when more than 60 virtual threads are used. Essentially, scattered allocates thread IDs such that threads with numerically close IDs are on different physical cores whereas balanced places threads with similar IDs on the same core. Diagrams showing examples of compact, scattered and balanced affinity for OpenMP can be found in [8].

The performance results are summarised in Table 3. The H2O-64 benchmark was used for all the tests described in this section.

Affinity used	CP2K time (seconds)
POPT, default	579.120
POPT, compact	1192.891
POPT, scattered	582.907
POPT, balanced	580.302

Table 3: Performance of the POPT version of CP2K on the Xeon Phi with different affinity settings using 30 MPI processes.

From Table 3 we can see that balanced and scattered affinities are equivalent as expected. The default affinity also gives good performance. Using a compact affinity where less than half the physical cores get used takes more than 200% longer than scattered or balanced and thus should be avoided.

A similar such test was performed on the SSMP version of the code using 30 OpenMP threads. The results are presented in Table 4.

Affinity used	CP2K time (seconds)
SSMP, default	804.444
SSMP, compact	1329.221
SSMP, scattered	810.107
SSMP, balanced	805.167

Table 4: Performance of the SSMP version of CP2K on the Xeon Phi with different affinity settings using 30 OpenMP threads.

As with the POPT version scattered and balanced affinities give very similar performance with compact being considerably (~1.65 times) slower. Thus for SSMP using either a balanced or scattered affinity is best.

Finally we looked at the affinity for the PSMP version of the code. With this version of the code we expected that the location of each MPI process' threads would be important. We expected that it would be best to place threads from a particular MPI process on the same physical core such that they can share cache, i.e. using a balanced type affinity should be best. To investigate this we ran the PSMP version with 60 MPI processes each running 2 threads using a scattered and balanced affinity. The results are shown in Table 5.

Affinity used	CP2K time (seconds)
PSMP, scattered	409.090
PSMP, balanced	351.297

Table 5: Performance of the PSMP version of CP2K on the Xeon Phi with different affinity settings using 60 MPI processes each running 2 OpenMP threads.

From Table 5 it is clear that using a balanced affinity that keeps threads belonging to a particular MPI process close together gives better performance relative to having the threads and their processes far apart. Balanced affinity is around 16% faster than scattered for this particular example.

Having determined that a balanced affinity gives the best performance, the final test we carried out was to investigate whether the actually placement of any extra threads beyond a multiple of 60 matters. For example, with 64 MPI processes each running 2 threads a total of 128 virtual threads will be used. Is it better to distribute the remaining 8 virtual threads such that the first 8 physical cores have a total of 3 virtual threads or should we distribute them such that the first 4 physical cores have 4 threads each? Having 3 threads per core gives the most even distribution of work between the cores but having 4 threads on a core keeps the threads belonging to a particular MPI process close to each other. The results of this test are summarised in Table 6.

Affinity	CP2K time (seconds)
PSMP, scatter	579.478
PSMP, balanced, first 8 cores with 3 threads, remainder with 2	338.935
PSMP, balanced, first 4 cores with 4 threads, remainder with 2	518.202

Table 6: Performance of the PSMP version of CP2K on the Xeon Phi with different affinity settings using 64 MPI processes each running 2 OpenMP threads.

Given the poor performance when using compact affinity, the results in Table 6 are not surprising. The best performance is given when any extra threads are distributed as evenly as possible between the physical cores. Any benefit that may have been gained by having an MPI process' threads close by appears to be mostly lost by having a subset of the cores fully loaded.

In summary, for the POPT and SSMP versions of CP2K using either a scattered or balanced affinity are roughly equivalent. For the PSMP version a balanced affinity is best as it keeps the threads belonging to a particular MPI process physically close to one another. The optimal solution is to use a balanced type affinity where we ensure that the virtual threads are distributed as evenly as possible between the physical cores. If we use less than 60 virtual threads then having one of these running on each physical core is optimal. If using more than 60 virtual threads then the extra threads should be placed such that they are as close as is practical to their associated MPI process *but* the total number of virtual threads per physical core should be kept as small as possible.

3. Benchmark results

To investigate the performance of CP2K we used the H2O-64 benchmark, which is a molecular dynamics simulation of 64 water molecules in a 12.4Å^3 cell using QUICKSTEP DFT to evaluate forces. This is typical of routine calculations carried out on modern HPC systems, can scale to 100s of CPU cores, while still taking up relatively little memory in order to fit on a single Xeon Phi card. The performance of the benchmark was initially investigated on the host using a single 16-core node. The four different host versions of CP2K were tested; SOPT, POPT, SSMP, PSMP. A range of thread counts or MPI process counts up to sixteen was tested for the SSMP and POPT versions respectively. For the PSMP version the number of threads was fixed at 1, 2, 4 etc with the number of MPI processes being increased until reaching sixteen. In addition to this, the PSMP version was also run keeping the number of threads times the number of MPI processes fixed at sixteen to ensure that all cores on a node get used. Figure 1 shows the CP2K runtime plotted against the number of cores used for the host version of the code with Table 7 giving the actual timings. For the PSMP version using all sixteen cores the runtime is plotted against the number of OpenMP threads.

Code version	Number of MPI or OpenMP threads for SSMP version				
	1	2	4	8	16
SOPT	555.586	-	-	-	-
POPT	558.143	296.680	158.044	88.181	54.781
SSMP	554.202	307.965	177.223	116.564	111.166
PSMP (full node)	120.999	117.978	147.789	183.422	58.775
PSMP 1 thread	559.040	297.341	158.564	89.450	55.846
PSMP 2 threads	320.635	244.244	195.082	182.987	-
PSMP 4 threads	190.515	161.979	145.514	-	-

Table 7: Performance of the H2O-64 benchmark run on the host. The fastest runtime was obtained with the POPT version using 16 MPI processes.

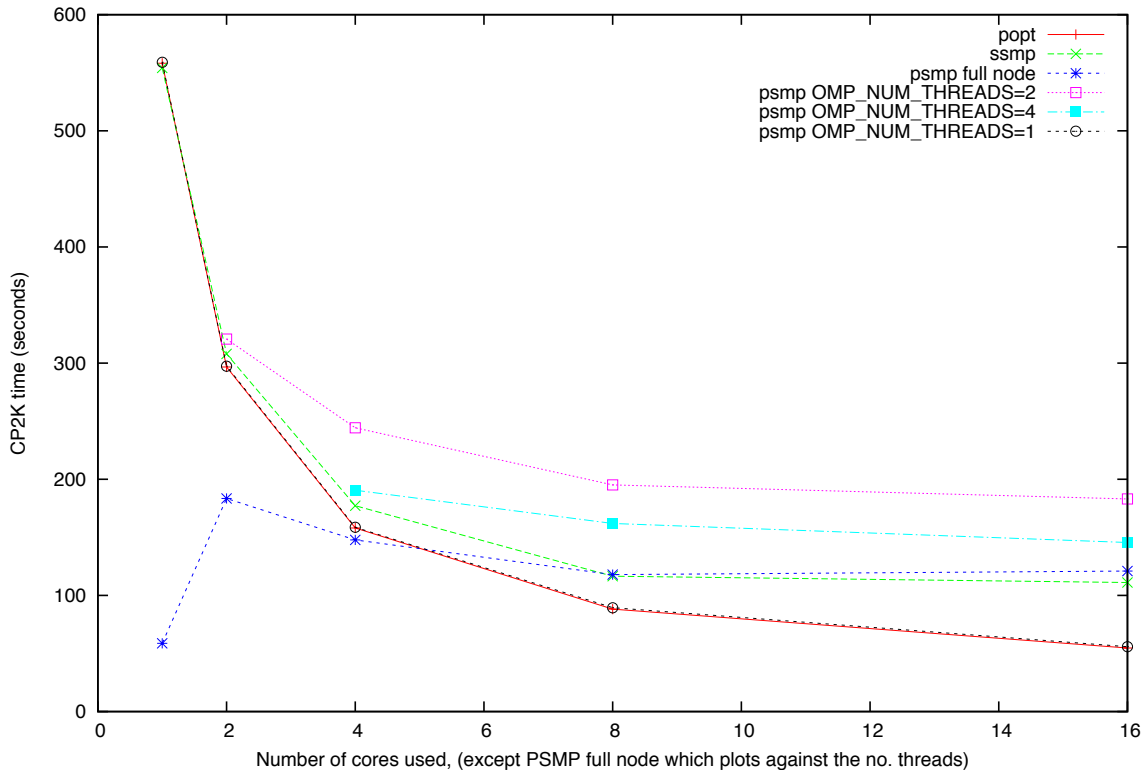

Figure 1: Performance of H2O-64 benchmark for 1 time step on a 16-processor Intel Xeon host node of Dommic.

From Figure 1 we can see that the best performance is obtained when running the pure MPI (POPT) version of the code on 16 processors. The runtime of the OpenMP (SSMP) version is generally higher than for the POPT version as it doesn't scale as well as the MPI version. The PSMP version run on a fixed thread count is slower

than both the SSMP or POPT versions for the equivalent number of cores. When running all sixteen cores the PSMP version gives equivalent performance to the POPT version when using 16 MPI processes. The other thread counts however never manage to do any better than the SSMP version. The performance of the POPT version and PSMP versions with 1 thread are identical as expected. The serial (SOPT) runtime was the same as the SSMP (1 thread) or POPT (1 process) versions.

Figure 2 and Figure 3 give the corresponding performance results obtained on the Xeon Phi. Figure 3 is a zoom in of Figure 2 that enables the important features to be observed. Some process or thread counts (e.g. POPT with more than 90 MPI processes, SSMP with more than 120 OpenMP threads) could not be run due to the memory limitations of the card - the Xeon Phi card has 8 GB of memory whereas the host nodes have 32 GB. Some of the data structures in CP2K (replicated real-space grids, DBCSR work matrices) scale with the number of MPI processes or with the number of OpenMP threads. As the Xeon Phi cards allow us to run up to 240 MPI processes or OpenMP threads it is not surprising that we can quickly run out of memory. Due to these memory limitations we were unable to run the PSMP version of the code in for all possible combinations of MPI processes and OpenMP threads. Instead we ran the PSMP version with 1, 2, 4, 8, 15, 16, 30, 32 and 60 threads per process increasing the number of MPI processes until we ran out of memory, used all 240 virtual threads or the performance started to degrade.

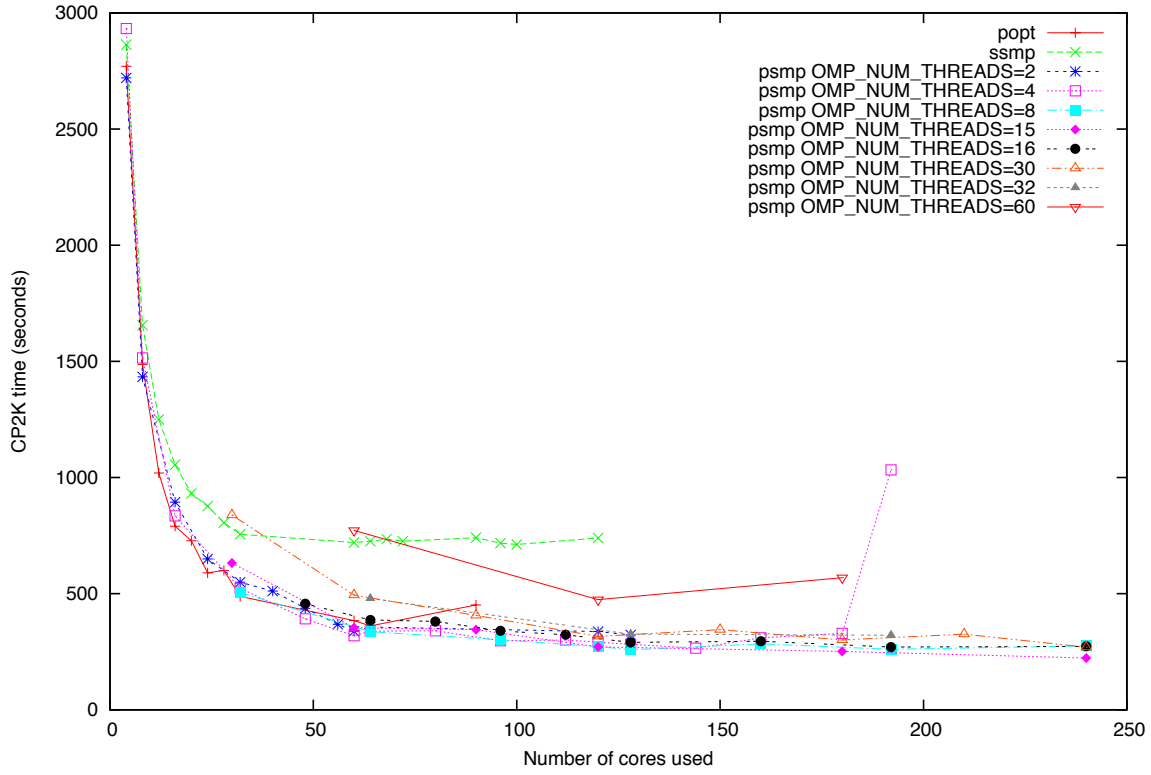

Figure 2: Performance of H2O-64 benchmark for 1 time step on Xeon Phi MIC card.

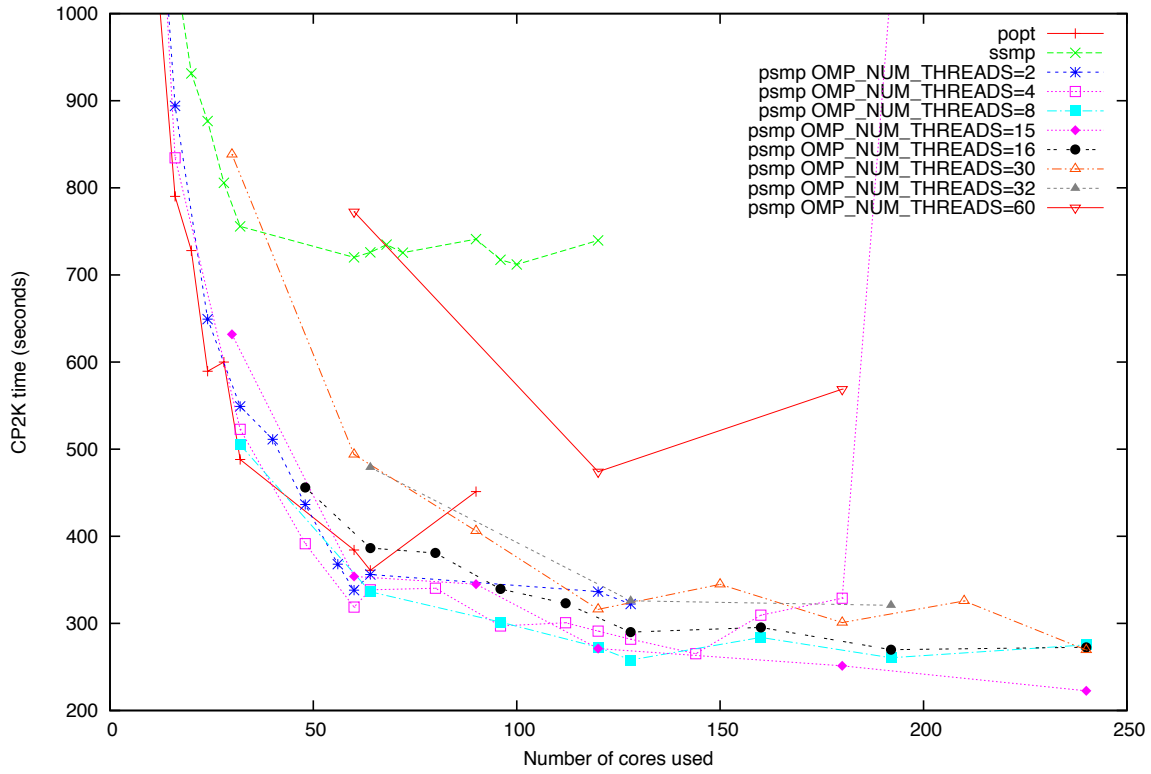

Figure 3: Performance of H2O-64 benchmark for 1 time step on Xeon Phi MIC card - zoomed in to show details

Comparing the performance of the host and Xeon Phi (i.e. comparing Figure 1 and Figure 2) we can see that the POPT and SSMP versions behave in a similar way. The POPT version is roughly twice as fast as the SSMP version for large numbers of cores. The main difference is that on the Xeon Phi the PSMP version is able to outperform both the SSMP and POPT version. This is because by varying the number of MPI processes and threads we are able to reduce the memory requirements, which allows more virtual threads to be used and thus better performance to be obtained. The jumpy behaviour of the Xeon Phi results is mainly due to the fact having different numbers of MPI processes results in different process decompositions for data structures within CP2K.

The best performance on the Xeon Phi card was obtained with the PSMP version running with 16 MPI processes each using 15 OpenMP threads. The runtime for this test was 223 seconds. Comparing this with the best performance obtained on the host node (55 seconds on 16 MPI processes) we find that the host node is a around 4 times faster than the Xeon Phi. Clearly, the code is not currently able to utilise the Xeon Phi to its full potential (for this particular benchmark problem), since the MIC has around 12x the peak FLOP/s of the Xeon E5-2670. The performance of CP2K on the Xeon Phi may be limited for several reasons: the lack of strong scaling in several parts of the code, potentially a lack of vectorization on the Xeon Phi native build and also the increasing memory footprint as the number of threads or processes increases which prevents all of a cards resources from being utilised. The performance of the different parts of CP2K will be investigated in more detail in Section 4, and we present a number of possible solutions to these issues in Section 5.

4. Performance analysis

Scalability of individual subroutines

The results presented in Section 3 focussed on the total runtime of CP2K as reported by the “CP2K” time in the timing report output by the code. This section looks at the performance and scaling of different parts of the code on both the host and Xeon Phi with a view to identifying areas of the code that may benefit from further optimisation or parallelisation. All results in this section were obtained using the H2O-64 benchmark running for one time step.

We begin by looking at the performance of the POPT version on the host processors. By default CP2K times a large number of subroutines and writes the timings obtained at the end the standard output from the code. An example of this output is given in Figure 4 for the POPT version of the code running on 4 processors.

```
SUBROUTINE                          CALLS  ASD         SELF TIME        TOTAL TIME
                                                 AVERAGE  MAXIMUM  AVERAGE  MAXIMUM
CP2K                                    1  1.0    0.038    0.048  158.044  158.045
qs_mol_dyn_low                          1  2.0    0.005    0.005  157.700  157.732
qs_forces                               2  3.5    0.003    0.003  157.663  157.663
qs_energies_scf                         2  4.5    0.000    0.000  153.002  153.003
scf_env_do_scf                          2  5.5    0.000    0.000  148.743  148.743
scf_env_do_scf_inner_loop              58  6.1    0.005    0.006  145.367  145.368
qs_rho_update_rho                      60  7.2    0.001    0.001   67.241   67.244
calculate_rho_elec                     60  8.2   57.535   60.212   67.240   67.243
qs_ks_build_kohn_sham_matrix           60  8.1    0.018    0.021   65.053   65.061
qs_ks_update_qs_env                    60  7.2    0.001    0.001   62.317   62.325
sum_up_and_integrate                   60  9.1    0.159    0.164   52.089   54.397
integrate_v_rspace                     60 10.1   48.431   50.740   51.931   54.243
cp_dbcsr_multiply_d                  1299 11.2    0.002    0.003   26.517   29.487
dbcsr_multiply_anytype               1299 13.2    0.154    0.173   26.435   29.403
velocity_verlet                         1  3.0    0.000    0.000   25.153   25.156
qs_scf_loop_do_ot                      58  7.1    0.000    0.000   20.173   20.185
cp_dbcsr_mult_NS_NR                   184 11.1    0.001    0.001   16.910   19.874
dbcsr_mm_cannon_multiply             1299 14.2    0.152    0.168   18.547   18.597
dbcsr_mm_cannon_multiply_multre      2598 15.2   16.564   16.584   16.566   16.585
ot_scf_mini                            58  8.1    0.004    0.005   16.311   16.318
density_rs2pw                          60  9.2    0.002    0.002    9.238   12.284
fft_wrap_pw1pw2                       602 10.8    0.005    0.006   10.733   11.025
fft_wrap_pw1pw2_140                   242 11.3    0.688    0.702    9.919   10.239
fft3d_ps                              602 12.8    6.788    7.207    8.320    8.663
dbcsr_make_images                    2540 14.2    0.006    0.006    5.563    8.568
make_images                          2540 15.2    0.358    0.363    5.558    8.563
rs_pw_transfer                        484 11.1    0.003    0.005    4.625    8.125
```

Figure 4: Sample timing output from CP2K running the POPT version on 4 MPI processes. Only routines taking more than 5% of the total runtime have been included.

The code reports the number of calls made to each subroutine, the average stack subroutine depth or ASD which records how many calls down the source tree the subroutine call was made from and the self time and total time. The total time gives the inclusive time for that subroutine, i.e. the time taken for the subroutine and all of its offspring, the self time gives the exclusive time for the routine excluding any offspring. The averages and maximum values are taken over all MPI processes. Subroutines that are of interest are therefore the ones which either have a high maximum self time with a similar value for the total time or a high total time with low value for the self time. If the total time and self time are identical then the subroutine has no child function calls.

Table 8 shows the time spent in the most costly routines for a 4 processor POPT run and a 16 processor POPT run. The speedup is computed relative to the 4 processor POPT result, thus ideal (perfect scaling) speedup would be 4.

	POPT,4	POPT,16	Speedup
Total run time	158.045	54.785	2.88
`calculate_rho_elec`	60.212	16.281	3.70
`integrate_v_rspace`	50.740	14.139	3.59
`dbcsr_mm_cannon_multiply`	18.597	7.249	2.57
`dbcsr_make_images`	8.568	1.454	5.89
`fft_wrap`	11.025	4.363	2.53
`build_*`	~5.0	2.139	2.33
Sum of subroutine timings	154.142	45.625	-

Table 8: Showing time spent in the most costly routines for the POPT version of CP2K running on the host. Note: the speedup is computed relative to the 4 MPI process result and thus ideal speedup would be 4. `Build*` refers to a set of routines `build_core_ppl, build_core_ppnl, build_kinetic_matrix, build_overlap_matrix` which are individually small, but can be significant collectively when large numbers of threads are used.

We can see that on the host the majority of CP2K's runtime is spent inside a relatively small number of routines and that these routines scale quite well given the small problem size of this benchmark case. The

dbcsr_make_images subroutine exhibits super-linear scaling suggesting that splitting the problem across more processors could well allow it to reside in cache.

We now consider the results from the Xeon Phi. Table 9 compares the POPT version run on 4 processes with POPT and SSMP run on 60 processes or threads respectively with the PSMP version run on 240 cores (using 16 MPI processes each running 15 OpenMP) threads as this was found to give the best performance on the Xeon Phi (see Section 3).

	POPT,4	POPT,60	Speedup	SSMP,60	Speedup	PSMP,240	Speedup
Total run time	2770	384	7.21	720	3.85	223	12.42
calculate_rho_elec	1166	73	15.97	123	9.48	46	25.35
integrate_v_rspace	1042	65	16.03	91	11.45	29	35.93
dbcsr_mm_cannon_multiply	186	79	2.35	48	3.88	55	3.38
dbcsr_make_images	160	24	6.67	21	7.62	19	8.42
fft_wrap	148	17	8.71	24	6.17	9	16.44
build_*	103	11	9.36	427	0.24	33	3.12
Sum of subroutine timings	2805	269	-	734	-	191	-

Table 9: Comparison of the cost of key routines for CP2K running on the Xeon Phi in native mode. The speedup is computed relative to the 4 MPI process POPT result in each case. For the POPT,60 and SSMP,60 results ideal scaling would be 15. For the PSMP,240 results, ideal scaling would be 240/4 = 60. The "Total time" can be greater than the CP2K time as the maximum runtime over all processors is used.

Table 9 shows the scaling of the POPT version of the code is good (speedup over 15) for the routines calculate_rho_elec and integrate_v_rspace. Most of the other routines do scale, although not so well (speedup of 6 to 9), the exception is dbcsr_mm_cannon_multiply, which has a speed up of only 2.35 relative to the 4 processor time. The scaling behaviour of the SSMP version is broadly similar to the POPT version except that the speedups are slightly lower overall. However, the SSMP version spends a very long time in the build_* routines relative to either the POPT or PSMP versions. Over half the SSMP runtime is spent in these routines, compared to the POPT,60 and PSMP,240 runs which only spend 3% and 12% respectively here. On examining the 4 thread SSMP results it appear that the build_* routines also cost ~430-440 seconds - they do not scale at all as these routines are not yet OpenMP parallelised (they are MPI parallelised though). In order to to run the SSMP or PSMP versions efficiently on the Xeon Phi (or indeed any architecture where these routines take a significant time) then these routines must be parallelised with OpenMP.

Finally, examining the PSMP speedups the most obvious feature is that the speedups are very much less than the ideal scaling target of 60. In some ways this is not so surprising, the PSMP version is using a combination of MPI and OpenMP and though we can use a much larger number of cores than with MPI or OpenMP alone, both of these are at the limit of scalability, and so speedup is relatively poor. Any improvements made to the scalability of the pure MPI or pure OpenMP parts of the code will benefit the PSMP version. With the PSMP,240 run the sum of the routines we reported individually only accounts for 85% of the total run time. Closer study of studying the code output, shows the majority of this time can be accounted for with two routines: cp_fm_syevd and cp_fm_syevd_base which take up to 23.643 and 23.091 seconds respectively. These are parallel eigensolvers implemented in ScaLAPACK, which do not appear to be to take advantage of many OpenMP threads.

Analysis of compiler vectorization

In order to get good application performance on the Xeon Phi two things are particularly important; one is to use as many virtual cores as possible, the other is to ensure your code is well vectorized such that you can take advantage of the vector processing unit (VPU) on the Xeon Phi card, which can execute 16 single precision or 8 double precision operations per clock cycle. Each of these operations can be a fused multiply-add (FMA) giving a maximum of 32 single precision or 16 double precision operations per cycle. Vectorized code will generally perform much better than non-vectorized code and thus where possible, application codes being executed on the Xeon Phi should be vectorized. Often the compiler will automatically do a good job of vectorizing code but it's important to check which parts of the code were vectorized and which were not.

To investigate the vectorization of each of the subroutines listed in Table 9 (e.g. the routines that were found to be the most costly) was recompiled with the –vec-report2 or –vec-report6 compiler flag. This provides diagnostic information at compile-time regarding the compiler's ability to vectorize (or not). Level 2 flag shows whether or not a loop was vectorized, the level 6 flag produces the maximum amount of detail including data

dependencies and other reasons for loops not being vectorized. An extract of the output using `–vec-report2` for the `calculate_rho_elec` subroutine is shown in Figure 5.

```
qs_collocate_density.F(1571): (col. 5) remark: loop was not vectorized: nonstandard loop
is not a vectorization candidate.
qs_collocate_density.F(1591): (col. 19) remark: LOOP WAS VECTORIZED.
qs_collocate_density.F(1591): (col. 19) remark: LOOP WAS VECTORIZED.
qs_collocate_density.F(1612): (col. 5) remark: loop was not vectorized: existence of
vector dependence.
qs_collocate_density.F(1681): (col. 7) remark: loop was not vectorized: existence of
vector dependence.
qs_collocate_density.F(1693): (col. 37) remark: loop was not vectorized: vectorization
possible but seems inefficient.
qs_collocate_density.F(1797): (col. 8) remark: loop was not vectorized: vectorization
possible but seems inefficient.
qs_collocate_density.F(1799): (col. 8) remark: loop was not vectorized: vectorization
possible but seems inefficient.
qs_collocate_density.F(1840): (col. 14) remark: loop was not vectorized: vectorization
possible but seems inefficient.
qs_collocate_density.F(1866): (col. 14) remark: loop was not vectorized: vectorization
possible but seems inefficient.
qs_collocate_density.F(1879): (col. 5) remark: loop was not vectorized: nonstandard loop
is not a vectorization candidate.
qs_collocate_density.F(1880): (col. 5) remark: loop was not vectorized: nonstandard loop
is not a vectorization candidate.
qs_collocate_density.F(1898): (col. 12) remark: loop was not vectorized: existence of
vector dependence.
qs_collocate_density.F(1925): (col. 20) remark: LOOP WAS VECTORIZED.
qs_collocate_density.F(1925): (col. 20) remark: loop was not vectorized: not inner loop.
qs_collocate_density.F(1925): (col. 20) remark: loop was not vectorized: not inner loop.
qs_collocate_density.F(1925): (col. 20) remark: loop was not vectorized: unsupported loop
structure.
qs_collocate_density.F(1925): (col. 20) remark: loop was not vectorized: not inner loop.
qs_collocate_density.F(1925): (col. 20) remark: loop was not vectorized: not inner loop.
qs_collocate_density.F(1928): (col. 5) remark: loop was not vectorized: nonstandard loop
is not a vectorization candidate.
```

Figure 5: Sample vectorization report output for the calculate_rho_elec subroutine with the `-vec-report2` flag used.

Considering Figure 5, with the exception of line 1681 the loops that cannot be vectorized have genuine data dependencies or are too small to be of great concern. The code at line 1681 is a single array syntax operation, which zeros parts of an array with the between lower and upper limits depending on the thread id. This should be vectorizable in principle but the compiler doesn't seem to be able to deal with this.

Examining the output for `integrate_v_rspace` the vast majority of the routines either have genuine data dependencies or contain array syntax operations. Lines 1487, 1490, 1496, 1498 seem to have the same issue discovered in `calculate_rho_elec` where the code looks as though it should be vectorizable but the compiler doesn't manage to unravel it.

For `dbcsr_mm_cannon_multiply` the vectorization report makes reference to a number of potential alignment issues where it may be possible to pad the arrays such that they are aligned on the 512-bit boundaries. With this subroutine the vectorization output doesn't always seem to match up with the correct line numbers in the code hindering the interpretation of the output somewhat.

With the subroutine `fft_wrap_pw1pw2` there are again a number of alignment issues and some confusing output where the vectorisation report says that the loop was vectorized and also that it failed to vectorize for the exact same line numbers, e.g. with reporting level 2, some sample output was as follows:

```
pw_methods.F(1422): (col. 11) remark: LOOP WAS VECTORIZED.
pw_methods.F(1422): (col. 11) remark: loop was not vectorized: not inner loop.
pw_methods.F(1422): (col. 11) remark: loop was not vectorized: not inner loop.
pw_methods.F(1422): (col. 11) remark: loop was not vectorized: vectorization
possible but seems inefficient.
pw_methods.F(1422): (col. 11) remark: loop was not vectorized: not inner loop.
pw_methods.F(1422): (col. 11) remark: loop was not vectorized: not inner loop.
```

For `build_core_ppl` there are some unaligned variables/arrays but nothing that should impact on the performance of the code significantly. For `build_kinetic_matrix` the `CASE` statement at line 338 cannot be vectorized. The compiler reports:

```
qs_kinetic.F(338): (col. 21) remark: loop was not vectorized: unsupported loop structure.
```

The code that this relates to is simply a `DO` loop calling a subroutine inside it, e.g.

```
336   DO ikind=1,nkind
337       atomic_kind => atomic_kind_set(ikind)
338       SELECT CASE (basis_set_id)
339       CASE (use_orb_basis_set)
340         CALL get_atomic_kind(atomic_kind=atomic_kind,orb_basis_set=basis_set_a)
341       CASE (use_aux_fit_basis_set)
342         CALL get_atomic_kind(atomic_kind=atomic_kind,aux_fit_basis_set=basis_set_a)
343       END SELECT
344       IF (ASSOCIATED(basis_set_a)) THEN
345         basis_set_list(ikind)%gto_basis_set => basis_set_a
346       ELSE
347         NULLIFY(basis_set_list(ikind)%gto_basis_set)
348       END IF
349     END DO
```

Figure 6: Code snippet from `build_kinetic_matrix` showing subroutine calls within a loop.

However, this loop is not computationally intensive, so vectorization is not important for performance.

The subroutine `build_matrix_overlap` also has a similar `DO` loop at line 390 that cannot be vectorized.

Overall the compiler seems to do a reasonable job of vectorizing the key routines in CP2K. Often the compiler fails to vectorize expressions containing "`%`" (expressions involving derived data types) , "`=>`" (pointers) or complex subroutine calls made from within a loop. Fortunately, most such expressions operate on small data structures and thus vectorization won't actually help. Loop nests with large trip counts involving significant floating point arithmetic are usually well vectorized.

5. Opportunities for optimisation

Based on the results of our benchmarking and analysis to date, we have identified a number of opportunities for optimisation which will be required to enable CP2K to take full advantage of the Xeon Phi architecture, and list these below. They are principally concerned with improving OpenMP scalability, as this will not only benefit the use of the code in native mode (as we have done), but will also be required in order to use the Xeon Phi offload model, where individual regions of code are executed on the MIC and the rest on the host CPU. OpenMP can be used to parallelise within the offloaded region, and MPI is used to parallelise the calculation over multiple compute nodes (and thus many Xeon Phis).

1. To avoid the linear memory scaling in `calculate_rho_elec` we propose to replace the existing scheme of threads writing to private array sections followed by an expensive parallel reduction. This can be done by using OpenMP locks or atomic updates to allow safe concurrent access to a single shared grid, with smaller buffer arrays to reduce lock contention. This should allow for scaling to larger numbers of OpenMP threads due to the reduction in memory footprint, and also improve performance by avoiding the reduction step.

2. To improve the scalability of the 3D FFT the existing partitioning of slabs to threads (a 1D decomposition) can be replaced by a 2D partitioning of individual FFT rows to threads. This will give better load balance in the case where the number of threads does not evenly divide one dimension of the FFT grid and allow for greater ultimate scalability

3. The `build_core_*` subroutines must be fully OpenMP parallelised to ensure they are no longer a bottleneck to high thread-count scalability, and could potentially be offloaded to the MIC.

4. MPI communication (`MPI_Alltoall`) may be overlapped with (threaded) computation in the `rs_distribute_matrix` subroutine. This is currently an expensive synchronisation point when running with large thread counts in addition to large numbers of MPI processes.

5. Our benchmarking also showed a significant serial overhead in `dbcsr_multiply_anytype` (a precursor step to the multiplication). This can be parallelised using OpenMP, and included in the offload region for DBCSR multiplication. The core perfomrnace and scalability of DBCSR is also a concern, but work in this area is currently underway at CSCS in collaboration with Cray.

We intend to implement these optimisations under a PRACE Preparatory Access project aimed at enabling applications on the newly installed Xeon Phi cluster EURORA based at CINECA in Italy.

6. Conclusion

In summary, we have ported CP2K to the Intel software stack, and fully tested the code on both Xeon and Xeon Phi hardware. All bugs found during testing on the host have now been fixed, and although a small number (around 1%) of regression tests still fail on the Xeon Phi, the vast majority of functionality in the code has been verified to work correctly. No further code changes regarding parallelisation or optimisation have been carried out at this stage as we investigated the ease with which an existing code could be ported.

Benchmarking of the code has been carried out using a small molecular dynamics simulation using MPI, OpenMP and mixed-mode MPI/OpenMP, and we have found that at for this calculation, using the full 16 cores of a Xeon E5-2670 processor still outperforms the unoptimised Xeon Phi ported code by a factor of 4. During this process we discovered that controlling the placement of MPI tasks and OpenMP threads on the virtual cores of the MIC is crucial, and that a balanced approach where threads are kept as close as possible to the parent process while ensuring that no physical core is overloaded gives the best performance. The porting and benchmarking activities reported here are the result of 4 person-months of work carried out under PRACE-3IP Task 7.2.

Detailed analysis of individual routines within CP2K has revealed a number of bottlenecks that must be resolved to enable the code to take full advantage of the performance offered by the Xeon Phi architecture. Work to implement these optimisations will be carried out within the scope of a PRACE Preparatory Access project targeting the new PRACE Xeon Phi cluster 'EURORA'. The results of this work are reported in a separate white paper [9].

References

[1] CP2K homepage; http://www.cp2k.org

[2] VandeVondele, J; Krack, M; Mohamed, F; Parrinello, M; Chassaing, T; Hutter, J. 2005 QUICKSTEP: Fast and accurate density functional calculations using a mixed Gaussian and plane waves approach. Comp. Phys. Comm. 167 (2): 103-128 http://dx.doi.org/10.1016/j.cpc.2004.12.014

[3] Intel Xeon Phi Specifications; http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[4] M. Uchronski, A. Kwiecien, and M. Gebarowski, Exploiting CP2K application for exascale computing with accelerators using OpenACC and OpenCL, PRACE White Paper, 2014, http://www.prace-ri.eu/IMG/pdf/wp155.pdf

[5] CP2K Regression Tester webpage; http://people.web.psi.ch/krack/cp2k/regtest/regtest.html

[6] Intel documentation regarding using FFTW3 wrappers in MKL; http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/GUID-D17B3AB4-BD4E-4652-94A7-BAD4130CCB4A.htm

[7] Debugging on Intel Xeon Phi; http://software.intel.com/en-us/articles/debugging-on-intel-xeon-phi-coprocessor-use-case-overview

[8] Intel Fortran Compiler XE 13.1 User and Reference Guides, section Balanced Affinity Type; http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/fortran-lin/index.htm - GUID-8FCD3720-6F73-429C-AE65-7144ED0B991A.htm

[9] F. Reid and I. Bethune, Optimising CP2K for the Intel Xeon Phi, PRACE White Paper, 2013, http://www.prace-ri.eu/IMG/pdf/wp140.pdf

Acknowledgements

This work was financially supported by the PRACE-3IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.

We thank CSCS for giving us access to their 'Dommic' Intel Sandy Bridge / Xeon Phi cluster and Dr. Matteo Frigo for providing MIC support for FFTW.

Appendix A: CP2K architecture files

This appendix contains the CP2K architecture specific files for building the PSMP version on the host and Xeon Phi cards in native mode.

```
# PSMP arch file for MPI/OpenMP version using Intel compilers on Dommic
CC       = cc
CPP      =
FC       = mpiifort -openmp
LD       = mpiifort -openmp
AR       = ar -r

FFTW_DIR = /users/fiona/fftw3_intel_threaded
LIBXC_DIR = /users/fiona/lib/libxc-intel
LIBINT_DIR = /users/fiona/lib/libint-intel
LIBSMM_DIR = /users/fiona/lib/cp2klibs-intel
LIBGRID_DIR = /users/fiona/lib/cp2klibs-intel

CPPFLAGS =
DFLAGS   = -D__INTEL -D__FFTSG -D__FFTW3 -D__LIBINT -D__LIBXC2 -D__parallel \
      -D__BLACS -D__SCALAPACK -D__HAS_smm_dnn -D__HAS_LIBGRID
CFLAGS   = $(DFLAGS)
# Version 13.1.0 of compiler
MKLROOT = /apps/dommic/intel/composer_xe_2013.2.146/mkl/
FCFLAGS  = $(DFLAGS) -O2 -g -traceback -fpp -free \
      -I$(FFTW_DIR)/include
FCFLAGS2  = $(DFLAGS) -O1 -g -traceback -fpp -free \
       -I$(FFTW_DIR)/include
LDFLAGS  = $(FCFLAGS) -static-intel
# Link with the intel64 libs for a host version use mic for MIC cards
LIBS     = $(FFTW_DIR)/lib/libfftw3.a $(FFTW_DIR)/lib/libfftw3_omp.a \
      $(MKLROOT)/lib/intel64/libmkl_scalapack_lp64.a \
      -Wl,--start-group  $(MKLROOT)/lib/intel64/libmkl_intel_lp64.a \
      $(MKLROOT)/lib/intel64/libmkl_sequential.a \
      $(MKLROOT)/lib/intel64/libmkl_core.a \
      $(MKLROOT)/lib/intel64/libmkl_blacs_intelmpi_lp64.a -Wl,--end-group \
      -lpthread -lm \
      $(LIBINT_DIR)/lib/libderiv.a $(LIBINT_DIR)/lib/libint.a -lstdc++ \
      -L$(LIBXC_DIR)/lib -lxc \
      $(LIBSMM_DIR)/libsmm_dnn.a $(LIBGRID_DIR)/libgrid.a

OBJECTS_ARCHITECTURE = machine_intel.o

thermostat_utils.o: thermostat_utils.F
                          $(FC) -c $(FCFLAGS2) $<

qs_dispersion_nonloc.o: qs_dispersion_nonloc.F
                          $(FC) -c $(FCFLAGS2) $<

qs_vxc_atom.o: qs_vxc_atom.F
            $(FC) -c $(FCFLAGS2) $<

et_coupling.o: et_coupling.F
            $(FC) -c $(FCFLAGS2) $<

cp_lbfgs_optimizer_gopt.o: cp_lbfgs_optimizer_gopt.F
            $(FC) -c $(FCFLAGS2) $<
```

Figure 7: PSMP arch file for Xeon host nodes.

```
# PSMP mixed mode MIC/OpenMP arch file for Intel compilers on MIC in native mode
CC       = cc
CPP      =
FC       = mpiifort -openmp -mmic
LD       = mpiifort -openmp -mmic
AR       = ar -r

FFTW_DIR = /users/fiona/fftw3.3.4_intel_mic
LIBXC_DIR = /users/fiona/lib/libxc-intel-mic
LIBINT_DIR = /users/fiona/lib/libint-intel-mic
LIBSMM_DIR = /users/fiona/lib/cp2klibs-intel-mic

CPPFLAGS =
DFLAGS   = -D__INTEL -D__FFTSG -D__FFTW3 -D__LIBINT -D__LIBXC2 -D__parallel \
      -D__BLACS -D__SCALAPACK -D__HAS_smm_dnn
CFLAGS   = $(DFLAGS)
# Version 13.1.0 of compiler
MKLROOT = /apps/dommic/intel/composer_xe_2013.2.146/mkl/
FCFLAGS  = $(DFLAGS) -O2 -g -traceback -fpp -free \
      -I$(FFTW_DIR)/include
FCFLAGS2  = $(DFLAGS) -O1 -g -traceback -fpp -free \
       -I$(FFTW_DIR)/include
LDFLAGS  = $(FCFLAGS) -static-intel
# Link with the intel64 libs for a host version use mic for MIC cards
LIBS     = $(FFTW_DIR)/lib/libfftw3.a  $(FFTW_DIR)/lib/libfftw3_omp.a \
      $(MKLROOT)/lib/mic/libmkl_scalapack_lp64.a \
      -Wl,--start-group  $(MKLROOT)/lib/mic/libmkl_intel_lp64.a \
      $(MKLROOT)/lib/mic/libmkl_sequential.a \
      $(MKLROOT)/lib/mic/libmkl_core.a \
      $(MKLROOT)/lib/mic/libmkl_blacs_intelmpi_lp64.a -Wl,--end-group \
      -lpthread -lm \
      $(LIBINT_DIR)/lib/libderiv.a $(LIBINT_DIR)/lib/libint.a -lstdc++ \
      -L$(LIBXC_DIR)/lib -lxc $(LIBSMM_DIR)/libsmm_dnn_MIC.a

OBJECTS_ARCHITECTURE = machine_intel.o

thermostat_utils.o: thermostat_utils.F
                $(FC) -c $(FCFLAGS2) $<

qs_dispersion_nonloc.o: qs_dispersion_nonloc.F
                         $(FC) -c $(FCFLAGS2) $<

qs_vxc_atom.o: qs_vxc_atom.F
            $(FC) -c $(FCFLAGS2) $<

et_coupling.o: et_coupling.F
            $(FC) -c $(FCFLAGS2) $<

cp_lbfgs_optimizer_gopt.o: cp_lbfgs_optimizer_gopt.F
            $(FC) -c $(FCFLAGS2) $<
```

Figure 8: PSMP arch file for Xeon Phi MIC cards in native mode.

Appendix B: Compilation instructions and key changes to scripts

This Appendix contains some compilation instructions for building CP2K and its associated libraries on the Intel host and Intel Xeon Phi MIC cards.

Building a host version of CP2K – additional points

The key details for building an Intel version of CP2K are described in Section 2. Any additional points that maybe helpful to users are included here.

The libraries used by CP2K (libxc, libint, libsmm_dnn, libgrid, FFTW) were all built using the instructions supplied with the library. Usually, changing the build scripts to use the appropriate Intel compiler was all that was required followed by the usual process of `./configure`; `make`; `make install` etc. The exceptions were FFTW 3.3.3 and libgrid. Both of these libraries had to be built with OpenMP to ensure thread safe execution. For FFTW 3.3.3 this was achieved by adding `–enable-openmp` to the configure line, for libgrid `–openmp` was added to the `FCFLAGS` flag in `config.in`.

Building a native version of CP2K on the Xeon Phi

Essentially the procedure is to start with an Intel specific host makefile and then add `–mmic` to the `FC` and `LD` flags and ensure any libraries used are Xeon Phi specific. More details can be found in Section 2 with a sample makefile given in Figure 8.

Compiling FFTW 3.3.3 for Xeon Phi in native mode

On the host, i.e. on dommic or dom41 load the appropriate compiler and MPI modules:

```
module load icc/2013.2.146 impi/4.1.0.030
```

Then use the following for configure:

```
./configure MPICC=mpiicc CC=icc CXX=icpc FC=ifort CFLAGS="-mmic -O3" CXXFLAGS="-
mmic -O3" FCFLAGS="-mmic -O3" --enable-openmp --prefix=/users/fiona/fftw3_intel_mic
--host=blackfin
```

The `–enable-openmp` flag is required so that a threaded version of FFTW 3.3.3 gets built. The `--host=blackfin` or `--host-x86` flag enables cross compilation. If you don't add this flag then the configure stage fails when it attempts to run an executable compiled for the Xeon Phi on the host e.g. you'll get an error of the form:

```
fiona@dom04:~/fftw-3.3.3> ./configure MPICC=mpiicc CC=icc CXX=icpc FC=ifort CFLAGS="-mmic -O3"
CXXFLAGS="-mmic -O3" FCFLAGS="-mmic -O3" --prefix=/users/fiona/fftw3_intel_mic
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking whether to enable maintainer-specific portions of Makefiles... no
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for gcc... icc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... configure: error: in `/users/fiona/fftw-3.3.3':
configure: error: cannot run C compiled programs.
If you meant to cross compile, use `--host'.
See `config.log' for more details
```

To build and install the library use:

```
make
make install
```

Compiling FFTW 3.3.4 for Xeon Phi in native mode

FFTW 3.3.4 is a version of FFTW that contains patches that improve the performance on the Xeon Phi. It was patched and supplied to us by Matteo Frigo (one of the FFTW developers). It is NOT a release version and is not distributed. To compile this version we used the instructions provided by Matteo:

```
CC='icc -mmic' LDFLAGS='-mmic' ./configure --enable-kcvi  --host=x86_64-k1om-linux
--enable-openmp --prefix=/users/fiona/fftw3.3.4_intel_mic
make
make install
```

As with FFTW 3.3.3 the `–enable-openmp` flag must be used. The `–enable-kcvi` flag enables Knights Corner specific vector instructions to be used.

Building libxc-2.0.1 for Xeon Phi in native mode

Building libxc for the MIC uncovered a bug in the supplied configure script. Basically when the `-mmic` flag is added one of the macros in `config.h` gets mangled. To build the library, use the following configure line:

```
./configure CC=icc CXX=icpc F77=ifort FC=ifort F90=ifort CFLAGS="-O3 -mmic"
FFLAGS="-O3 -mmic" CXXFLAGS="-O3 -mmic" --prefix=/users/fiona/libxc-intel-mic --
host=blackfin
```

The build will then fail as follows:

```
/bin/sh ../libtool  --tag=FC --tag=F77  --mode=compile ifort   -c -o libxc_la-
libxc.lo  `test -f 'libxc.f90' || echo './'`libxc.f90
libtool: ignoring unknown tag F77
libtool: compile:  ifort -c libxc.f90 -o libxc_la-libxc.o
mv -f .deps/libxc_la-mgga_x_tpss.Tpo .deps/libxc_la-mgga_x_tpss.Plo
mv -f .deps/libxc_la-mgga_x_br89.Tpo .deps/libxc_la-mgga_x_br89.Plo
xc_f.c(32): error: function returning function is not allowed
       (int *major, int *minor)
        ^
xc_f.c(34): error: identifier "major" is undefined
    XC(version)(major, minor);
```

The reason for the failure is that the macro that sets `FC_FUNC` gets mangled when the `-mmic` flag is used. If you remove the `-mmic` flag, re-run configure and then difference the resulting `config.h` files:

```
fiona@dom04:~/libxc-2.0.1> diff config.h config.h.test
27c27
< #define FC_FUNC(name,NAME) name ## _
---
> /* #undef FC_FUNC */
30c30
< #define FC_FUNC_(name,NAME) name ## _
---
> /* #undef FC_FUNC_ */
```

Basically the definitions for `FC_FUNC` and `FC_FUNC_` are omitted with the `-mmic` flag. Rather than attempting to fix the configure script the simplest option is just to append on these two macros to `config.h` as generated with `-mmic`. i.e. add the following two lines to `config.h`

```
#define FC_FUNC(name,NAME) name ## _
#define FC_FUNC_(name,NAME) name ## _
```

It's probably best to add them in close to the commented out `/* #undef FC_FUNC */`, i.e. around line 30. The build can then be completed with:

```
make
make install
```

Building libint-1.1.4 for the Xeon Phi in native mode

Libint-1.1.4 can also be built by adding the `-mmic` flag and cross compiling. However, part of the build process involves executing a binary and using that to generate source code for the next stage of the compile. If we cross-compile then any binaries are Xeon Phi specific and won't run on the host. There are several possibilities here. You can allow the make to fail and then run the binary on the Xeon Phi card directly, you can then continue the build process and repeat. Or you can change the build scripts such that the Xeon Phi specific binary is executed via "`ssh mic0 ./whatever_my_binary_is`"

The first option was used to build the code on dommic. The configure line used was:

```
./configure CXX=icpc CC=icc FC=ifort FFLAGS="-O3 -mmic" --with-cc-optflags="-O3 -
mmic" --with-cxx-optflags="-O3 -mmic" --with-libdirs="-O3 -mmic" --
prefix=/users/fiona/lib/libint-intel-mic --host=x86_64-unknown-linux-gnu
```

The `--with-*` flags were used to set the compiler flags as setting `CFLAGS`, `CXXFLAGS`, `LDFLAGS` was ignored by the configure process (it shouldn't be). The library was built in the following way (compilation was carried out on dommic and binaries run on the Xeon Phi cards):

On dommic:
1. Load compiler/MPI modules `module load icc/2013.2.146 impi/4.1.0.030`
2. Run configure as above
3. Make

```
make -j 16
```
build will fail with:
```
make[3]: Entering directory `/users/fiona/libint-1.1.4/src/lib/libint'
cd src; /users/fiona/libint-1.1.4/src/lib/libint/../../../src/bin/libint/build_libint
/bin/sh: /users/fiona/libint-1.1.4/src/lib/libint/../../../src/bin/libint/build_libint: cannot
execute binary file
```

On Xeon Phi card:
4. Run binaries

```
export OMP_NUM_THREADS=1
ulimit -s unlimited
cd /users/fiona/libint-1.1.4/src/lib/libint
cd src
/users/fiona/libint-1.1.4/src/lib/libint/../../../src/bin/libint/build_libint
```

The `build_libint` binary will run on the Xeon Phi card generating a number of source files, which can then be used to continue the build.

On dommic:
5. Resume build

```
make -j 16
```

The build will continue for a while but now fail for `build_libr12.` Repeat steps 4 and 5 as above cd-ing into the appropriate directory for `build_libr12` and `build_deriv`.

After `build_deriv` has been executed and its source code compiled the build should then complete successfully and a Xeon Phi specific version of libint-1.1.4 should be created.

On dommic:
6. Install the library

```
make install
```

Building libsmm_dnn for the Xeon Phi in native mode

To build libsmm_dnn for the Xeon Phi in native mode you need to follow the instructions supplied in the `README` file under `cp2k/tools/build_libsmm` directory. Depending on your particular system, you may need to change the `./do_generate_tiny` and `./do_generate_small` scripts.

To compile the version of libsmm_dnn used in this report. The supplied `config.in` file was modified as follows: (important lines below)

```
target_compile="ifort -O2 -funroll-loops -vec-report2 -warn -mmic -fno-inline-
functions -nogen-interfaces -openmp"
SIMD_size=64 # default is AVX size, this should be picked up automatically for MIC
builds
blas_linking="-static-intel -mkl=sequential"
host_compile="ifort -O2 "
tasks=40 # MIC
jobs=350 # MIC
```

For now it seems best not to use the `./do_all` script but instead to use the individual scripts `./do_generate_tiny`, `./do_generate_small` and `./do_generate_lib` in this order. Prior to running these scripts the following changes must be made to ensure that the post-processing of the output works properly:

For `./do_generate_tiny`:

```
<     res=`tail -n 1 ${out_dir}/$file`
>     res=`tail -n 2 ${out_dir}/$file | head -n 1`
```

For `./do_generate_small`:

```
<     res=`tail -n 1 ${out_dir}/$file`
>     res=`tail -n 2 ${out_dir}/$file | head -n 1`
```

If this change is not made to these scripts then the output files from the tiny and small matrix tests don't contain the required timing data and when you attempt to build the small tests you will get an error of the form:

```
make: Nothing to be done for `all'.
# Elements = 1^3 = 1 split in 350 jobs.
Launching elements 1 --> 1 (run_small_dnn_job1) .././small_gen.x `echo
small_find_1_1_1 | awk -F_ '{ print $3" "$4" "$5 }'` 1 1 64
../tiny_gen_optimal_MIC_dnn.out > small_find_1_1_1.f90
forrtl: severe (24): end-of-file during read, unit -5, file Internal List-Directed
Read
Image              PC                Routine            Line        Source
small_gen.x        0000000000485D0E  Unknown               Unknown  Unknown
small_gen.x        00000000004847A6  Unknown               Unknown  Unknown
small_gen.x        0000000000445372  Unknown               Unknown  Unknown
small_gen.x        000000000040EBEB  Unknown               Unknown  Unknown
small_gen.x        000000000040E152  Unknown               Unknown  Unknown
small_gen.x        0000000000422B4B  Unknown               Unknown  Unknown
small_gen.x        000000000042137C  Unknown               Unknown  Unknown
small_gen.x        0000000000406453  Unknown               Unknown  Unknown
small_gen.x        000000000040A614  Unknown               Unknown  Unknown
small_gen.x        0000000000402AAC  Unknown               Unknown  Unknown
libc.so.6          0000003C2F21ECDD  Unknown               Unknown  Unknown
small_gen.x        00000000004029A9  Unknown               Unknown  Unknown
make: *** [small_find_1_1_1.f90] Error 24
```

This error arises because the file `tiny_gen_optimal_MIC_dnn.out` contains just the matrix sizes tests and no timing data, e.g. the incorrect version looks like:

```
1 1 1
1 1 2
1 2 1
1 2 2
```

Whereas, we should have something more like:

```
1 1 1    3   1   1   1    2.480000       0.081
1 1 2    1   1   1   1    2.860000       0.140
1 1 3    1   1   1   1    3.240000       0.185
```

There should be 9 columns in the file.

The build needs to be invoked from a node that has a Xeon Phi card connected, e.g. dom41. Furthermore, the source and binary files need to accessible to the Xeon Phi cards and thus the simplest approach is to check out the `build_libsmm` directory to your home space and build the library from there. Our library was built with the following settings in `config.in` which determine the matrix sizes to be tested:

```
dims_tiny=`seq 1 24`
dims_small="1 4 5 6 9 13 16 17 22 23 24"
```

With these settings it took over a week to do the tiny matrix runs and a further 1-2 days to build the small tests. The final generation of the library and performance testing took another 1-2 days to complete. However, the reader is referred to the notes in `config.in`, as it may not actually be necessary to search over such a large number of matrix sizes. When building `libsmm_dnn` you may need to periodically move the executable (`*.x`) files created as part of the build as these take up a lot of disk space and can potentially cause you to go above your disk quota.

Finally, for checking the library is building okay, setting:

```
dims_tiny=`seq 1 4`
dims_small="1 4"
```

will build a library in less 30 minutes which lets you check that a valid library will be created from your scripts before starting a build that could take many days to complete.

Running the regression test suite on the Xeon Phi

Running the regression tests on the Xeon Phi is somewhat more complex than on the host. The regression tests are written in bash, the only shell available on the Xeon Phi cards is sh and thus either the script must be converted (complex and error prone) or it must be adapted such that you run the Xeon Phi binaries via remote ssh. In addition, the test files, binaries etc all need to be located on a file system the Xeon Phi card can actually "see", i.e. somewhere on `/users/fiona` and not on the `/project` filespace. The `/users` filespace has a quota limit of 10 GB per user and thus only files that the Xeon Phi card need to access can be placed here, e.g. the test files themselves, the test output and the CP2K binaries.

The most up to date copy of the CP2K regression tests can be checked out with:

```
cd /users/fiona/regtest_mic  # This directory must be accessible to the MIC cards
svn co svn://svn.code.sf.net/p/cp2k/code/trunk/cp2k/tests
```

We also created an `exe/` directory inside our top level directory into which we copied the CP2K binaries across from `/project` so that the Xeon Phi cards will be able to access the binaries, e.g.

```
mkdir exe
cd exe
mkdir dommic-intel-mic
cp path-to-cp2k-binaries/cp2k.*  dommic-intel-mic/.
```

It should be noted these steps are only needed if you have a limited amount of disk space visible from the Xeon Phi cards. If you have unlimited amounts of space you can keep your entire CP2K build, regression tests etc on a file system accessible to the Xeon Phi card.

Finally, the `do_regtest` script needs to be modified such the correct paths to binaries and tests get picked up. The key variables that need to be changed are:

```
dir_base=/users/fiona/regtest_mic  # must be accessible to the MIC card
cp2k_version=popt
dir_triplet=dommic-intel-mic
cp2k_dir=../../../../../../../users/fiona/regtest_mic
cp2k_exe_dir=/users/fiona/regtest_mic  # New variable to set the path to CP2K exes
export OMP_NUM_THREADS=1 # The Xeon Phi sets this to 240 by default, set to 1 for
# POPT version or to the number of threads required for SSMP or PSMP
```

Other changes that were made to the script were:

1. uncomment `cp2k_run_prefix` and set it to:

   ```
   cp2k_run_prefix="ssh mic0"
   ```

2. comment out the initial occurrence of `cp2k_prefix` in the script. Because you need to run on the Xeon Phi via ssh you need to resolve this later once the test directory, test name etc are known. If you don't inline the call later on the Xeon Phi card doesn't know where to find the current test.

3. Inside the loop over tests, i.e. inside:
   ```
   for ((itest=1;itest<=ntest;itest++));
   do
   ...
   done
   ```

   modify the `cp2k_prefix` assignment such that it looks something like (use `;` to separate commands):

```
cp2k_prefix=${cp2k_prefix:-"${cp2k_run_prefix} cd ${dir_out}/tests/${dir};
ulimit -s unlimited;
ulimit -t 3000;
ulimit -a > /users/fiona/regtest_mic/checkthreads.out;
echo "PWD= $PWD" >> /users/fiona/regtest_mic/checkthreads.out;
echo "pwd = $pwd" >> /users/fiona/regtest_mic/checkthreads.out;
echo "CWD = $CWD cwd = $cwd dir_out = $dir_out dir = $dir test = ${dir_out}/tests/${dir}" >>
/users/fiona/regtest_mic/checkthreads.out ;
echo "Running on $OMP_NUM_THREADS threads" >> /users/fiona/regtest_mic/checkthreads.out;
/apps/dommic/mic/intel/impi/4.1.0.030/mic/bin/mpirun -np 30
${cp2k_exe_dir}/exe/${dir_triplet}/cp2k.${cp2k_version}"}
```

`ulimit -s unlimited` = Sets the stacksize to unlimited so we don't run out of memory
`ulimit -t 3000` = reduce the time allowed per job to 3000 seconds. The default on the Xeon Phi cards is unlimited and thus slow tests can take a very long time (25,000 seconds or more).

In addition, various variables are printed out to file (`checkthreads.out`) to check that have been correctly set on Xeon Phi cards but these can be omitted once you have the tests working.

To run the test suite use the following command from a node with a Xeon Phi card connected to it:

```
./do_regtest –nocvs –quick &> mytest.output
```

This will run all the regression tests and may take more than 12 hours. If you just want to run a single set of tests the `–restrictdir` flag can be added followed by a test directory name. For example, to run all the tests in the QS directory you could use:

```
./do_regtest –nocvs –quick –restrictdir QS &> mytest.output
```