XEON PHI EXPERIENCES

Fiona Reid
Overview

• Introduction to EPCC – 5 mins
• Introduction to Xeon Phi – 10 mins
• Case study 1: porting CP2K to Xeon Phi – 30 mins

• Break

• Case study 2: optimising CP2K on Xeon Phi – 30 mins
• IPCC project at EPCC, our experiences so far – 15 mins
EPCC IN 2015
EPCC

- Supercomputing Centre at The University of Edinburgh
- 25 years old
- ~75 staff
  - highly experienced
  - wide range of skills
- Multidisciplinary
- Multi-funded
  - turnover ~£5 million
  - 95% from external sources
- Huge range of activities
EPCC Activities

- Visitor Programmes
- HPC Research
- Training
- Facilities
- European Coordination
- Technology Transfer
Supercomputing facilities

- Advanced Computing Facility (ACF)
- Opened 2005
- Purpose built, secure, world-class facility
- Houses wide variety of leading-edge systems and infrastructures
  - National services
    - HECToR
    - ARCHER
    - IBM BlueGene/Q
  - Local services
    - ECDF (provide hosting)
    - INDY – industry machine
    - EDIM1 – DIR machine
ARCHE at the ACF

Cray XC30

- 118,080 cores
- 2.55 PFlop/s
- 3.4 Pb memory

- Intel Xeon “Ivy-Bridge processors
- 2 x 12 cores per node
- £50 million investment by UK research councils
HPC Research

- HPC technology development
  - QCDOC / BG/Q
  - FPGA/ FHPCA Maxwell
  - DIR machine
- Languages and tools
  - OpenMP
  - Java for HPC
  - MPI
- New HPC directions
  - Exascale
  - Cloud Computing
- Standardisation leadership
  - OpenMP
  - MPI
  - OpenACC
Software and Services

• EPCC – what we do
  • Facilities access – for academia and industry
  • Performance optimisation
  • GPU/GPGPU/FPGA computing
  • Software engineering
  • Project management
  • Visitor programmes and training
  • Data integration and data mining
  • Numerical modelling and simulation
  • Future Internet
  • Cloud and distributed computing
  • Parallel application consultancy and design services
Dr Fiona J L Reid

• Applications Consultant at EPCC
  • Background in Geophysics, specifically seismology
  • Worked with HPC for over 11 years
  • Experience with a range of different types of applications, programming models and hardware
  • Involved in writing several benchmark suites
  • Lecturer on MSc in HPC

• email: fiona@epcc.ed.ac.uk
Intel Xeon Phi Hardware

• What is Xeon Phi?
  • First product from Intel’s Many Integrated Core (MIC) architecture
  • a.k.a. Knight’s Corner (KNC), following on from Knight’s Landing / Larrabee

• Xeon Phi 5110P:
  • 60 physical cores derived from original Pentium P54C design
  • + 64 bit addressing
  • + 512-bit SIMD (8xDP FMA per clock cycle)
  • 1.053 GHz -> 1.01 TFLOP/s
  • 4 hardware ‘virtual threads’ per core
  • 8GB memory, 320 GB/s memory bandwidth
Intel Xeon Phi Hardware

- At EPCC:
  - ‘phi’ node connected to Hydra
  - 2 x Intel Xeon E5-2670 CPUs (16 cores Sandy Bridge @ 2.0 GHz)
  - 64 GB main memory
  - 2 x Intel Xeon Phi 5110P
  - Intel Cluster Studio XE 2015
Intel Xeon Phi Hardware

- Our system lets you login directly into the Xeon Phi cards

[fiona@hydra ~] $ srun -p phi -w phi --pty bash -i
[fiona@phi ~] $ ssh mic0
[fiona@phi-mic0 ~] $ cat /proc/cpuinfo

processor : 0
vendor_id : GenuineIntel
cpu family : 11
Model : 1
model name : 0b/01
stepping : 3
cpu MHz : 1052.630
cache size : 512 KB
physical id: 0
siblings : 240
core id : 59
<SNIP>
Intel Xeon Phi Software

• The Xeon Phi cards run a cut down version of Linux
  • Most regular Unix commands are there but not everything
  • Older versions of the MPSS only had sh shell. Newer versions have bash. There is no csh shell.

• Need to ensure that file systems between host/Xeon Phi are cross mounted otherwise you need to copy binaries/libraries etc over to the Xeon Phi’s local storage

• Assuming /opt/intel/* is cross-mounted then the usual Intel setup scripts can be used to set paths, e.g.

  source /opt/intel/impi_5.0.1/mic/bin/mpivars.sh
  source /opt/intel/composer_xe_2015/mkl/bin/mklvars.sh mic
Programming Models

- Intel emphasise ease-of-use (compared with e.g. CUDA) by supporting a range of existing programming models:
  - MPI
  - OpenMP
  - Intel Thread Building Blocks (TBB)
  - Intel Cilk+
  - OpenCL
  - MIC support in MKL (BLAS, FFT …)

- Can cross-compile code for Xeon Phi using icc/ifort and \texttt{-mmic} flag
  - All compilation must be done on the host machine
  - You \textbf{cannot} compile on the Xeon Phi card
Programming Models

- Range of methods to access the Xeon Phi card
  - Native mode:
    - `ssh` directly into the card, card runs its own version of Linux OS
    - Run applications on the command line
    - Use any of the supported parallel programming models to make use of the 240 virtual threads available
  
  - Offload mode:
    - Similar to CUDA ‘accelerator’ model
    - Main thread(s) of execution on the CPU, specific regions offloaded to Xeon Phi using `#pragma offload` directive + clauses to control data transfer
    - Offload region should be parallelised with OpenMP or Cilk+
    - Can also offload MKL calls
    - May offload to multiple Xeon Phis if available
Programming Models

• Symmetric MPI mode:
  • Compile two versions of an MPI binary (host and Xeon Phi)
  • Configure MPI host list to include Xeon Host(s) and Xeon Phi coprocessor(s)
  • Launch application using mpirun (need to supply a host list)
  • Result – single MPI_COMM_WORLD containing heterogeneous cores!

• P.S. we have only tested this with simple “Helloworld” type codes
PORTING CP2K TO XEON PHI

Fiona Reid
Porting CP2K to Xeon Phi

- Work done under PRACE 3IP Task 7.2 “Exploitation of HPC Tools and Techniques”
  - Goal was evaluating the potential of novel approaches to HPC to benefit key European applications
  - We investigated the performance and ease of porting for a large, complex application (CP2K) on Xeon Phi
CP2K Overview

“CP2K is a program to perform atomistic and molecular simulations of solid state, liquid, molecular, and biological systems. It provides a general framework for different methods such as e.g., density functional theory (DFT) using a mixed Gaussian and plane waves approach (GPW) and classical pair and many-body potentials.”

From www.cp2k.org (2004!)
CP2K Overview

• CP2K is a powerful tool
  • DFT, Classical, Hybrid-DFT, Many-body correlation, QM/MM
  • MD, MC, Relaxation, NEB, Free Energy Tools
  • … and lots lots more

• CP2K is popular (and growing)
  • 2\textsuperscript{nd} most heavily used code on ARCHER
  • 10+ research groups already using CP2K on national service

• Open Source
  • GPL, www.cp2k.org
  • 1m lines of code, 2 commits per day
  • ~10 core developers
Porting CP2K to Intel compiler suite

- CP2K is usually compiled with the GNU compilers
  - Free and available everywhere
- To run on the Xeon Phi we need to use the Intel compilers
- Compilation on the Intel E5-2670 host was straightforward
  - Use `–openmp` flag and link to appropriate FFTW or MKL libraries
- 4 versions of code tested: SOPT, SSMP POPT and PSMP
- Initially lots of issues with regression tests failing 😞
  - 65-80 failed with PSMP and SSMP versions
  - Problems with numerical stability too
- Recompiled/tested with `–check all` flag
  - Turns off optimisation and turns on all compiler/runtime checks
Key issues fixed (host)

- Added **recursive** attribute to subroutines that form recursion loops

- Ensure all pointers are nullified before use – *gfortran* does this automatically *ifort* does not

- Compiler bug submitted to Intel re. reading v. long lines in input file

- Intel MKL FFT execute functions not threadsafe for Fortran. There is a workaround for C. Submitted as a bug to Intel. From MKL version 11.1.0 onwards this issue is re-solved.
  - Workaround: use FFTW 3.3.3 or 3.3.4 instead
  - Code now checks MKL version and stops if non threadsafe version used
Key issues fixed (host)

- Fixed several bugs in dbcsr_lib (zero length arrays and messages)
- Discovered that `-heap arrays` and `-openmp` flags are not compatible with each other - reported to Intel for investigation
- Libgrid must be compiled with `-openmp` flag
- Various other source files altered due to strictness of `-check all`
CP2K host performance

H2O-64 benchmark for 1 time step on a 16 processor HOST node

CP2K time (seconds)

POPT 16 procs = 55s

No. of cores used (except PSMP full node which plots against no. of threads)
Compiling CP2K for Xeon Phi

- Once host version was working we compiled for Xeon Phi
  - Should just need to add \texttt{--mmic} flag and link to Xeon Phi specific libs

- Various issues building dependent libraries
  - Sneaky cross-compilation tricks required
  - Can use \texttt{ssh mic0 ./myexe} to launch code on the Xeon Phi card

- Lack of threadsafe FFT in MKL meant we had to use FFTW
  - FFTW 3.3.3 not optimised for Xeon Phi and thus performance could be poor
  - We also had access to FFTW 3.3.4, a pre-release version which has, been specifically optimised for Xeon Phi
## FFTW performance on Xeon Phi

<table>
<thead>
<tr>
<th>N</th>
<th>host FFTW 3.3.3</th>
<th>host MKL</th>
<th>MIC FFTW-3.3.3</th>
<th>MIC FFTW-3.3.4</th>
<th>MIC MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>0.0929</td>
<td>0.0928</td>
<td>12.1598</td>
<td>8.3509</td>
<td>8.4219</td>
</tr>
<tr>
<td>256</td>
<td>0.1981</td>
<td>0.1929</td>
<td>24.1600</td>
<td>17.1924</td>
<td>17.3632</td>
</tr>
<tr>
<td><strong>128</strong></td>
<td><strong>0.0542</strong></td>
<td><strong>0.0572</strong></td>
<td><strong>1.7499</strong></td>
<td><strong>0.3292</strong></td>
<td><strong>0.3891</strong></td>
</tr>
<tr>
<td><strong>256</strong></td>
<td><strong>0.1218</strong></td>
<td><strong>0.1152</strong></td>
<td><strong>3.8131</strong></td>
<td><strong>0.6374</strong></td>
<td><strong>0.7139</strong></td>
</tr>
</tbody>
</table>

### 1D FFT performance, **scaling removed**

<table>
<thead>
<tr>
<th>N</th>
<th>host FFTW 3.3.3</th>
<th>host MKL</th>
<th>MIC FFTW-3.3.3</th>
<th>MIC FFTW-3.3.4</th>
<th>MIC MKL</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>1.8241</td>
<td>1.6153</td>
<td>58.9320</td>
<td>43.2592</td>
<td>36.3927</td>
</tr>
<tr>
<td>256</td>
<td>7.9546</td>
<td>8.6894</td>
<td>267.2964</td>
<td>222.3350</td>
<td>165.8646</td>
</tr>
<tr>
<td><strong>128</strong></td>
<td><strong>1.6725</strong></td>
<td><strong>1.3278</strong></td>
<td><strong>35.5704</strong></td>
<td><strong>19.1855</strong></td>
<td><strong>11.1212</strong></td>
</tr>
<tr>
<td><strong>256</strong></td>
<td><strong>7.3357</strong></td>
<td><strong>6.1707</strong></td>
<td><strong>169.4342</strong></td>
<td><strong>117.3201</strong></td>
<td><strong>65.9620</strong></td>
</tr>
</tbody>
</table>

### 3D FFT performance, **scaling removed**

- **1D**: on host, MKL and FFTW 3.3.3 ~ same, on Xeon Phi FFTW 3.3.3 up to 6 times slower => should use FFTW 3.4
- **3D**: on host MKL faster than FFTW 3.3.3, on Xeon Phi, FFTW 3.3.3 performs poorly with MKL up to 78% faster than FFTW 3.3.4
- **Should use MKL if thread safe version is available to you**
Affinity testing

- Initial runs on the Xeon Phi showed very poor performance for the PSMP version (~ 9 times slower than POPT)
- POPT version seemed to perform okay
- It looked like a problem with oversubscription but all our helloworld test codes placed threads where we expected
  - For each MPI process each thread placed on a unique virtual thread
- Other CP2K users had experienced similar problems
- 
- Lots of head scratching, experimenting, testing
- With CP2K you must set the affinity by hand via `mpirun` flags or `KMP_AFFINITY`
Affinity testing

• We have 60 physical cores (PC), each running 4 virtual threads

Compact

<table>
<thead>
<tr>
<th>PC</th>
<th>PC</th>
<th>PC</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

Compact: 0 1 2 3 4 5

Scatter

<table>
<thead>
<tr>
<th>PC</th>
<th>PC</th>
<th>PC</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

Scatter: 0 4 1 5 2 3

Balanced

<table>
<thead>
<tr>
<th>PC</th>
<th>PC</th>
<th>PC</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

Balanced: 0 1 2 3 4 5

• Various placement strategies tested
  • Compact – preserves locality but some physical cores end up with lots of work and some end up with none
  • Scatter – destroys locality but if < 60 virtual threads used is fine
  • Balanced – preserves locality and works for all thread counts
Affinity testing

For 2 MPI processes each running 2 OpenMP threads:

```bash
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 &> x
```

For every MPI process you say where its threads will be placed
With large numbers of processes this gets quite messy!
CP2K Xeon Phi performance

H2O-64 benchmark for 1 time step on Xeon Phi in native mode
CP2K Xeon Phi performance

H2O-64 benchmark for 1 time step on Xeon Phi (zoomed in)
### Profiling results from Xeon Phi

<table>
<thead>
<tr>
<th></th>
<th>POPT,4</th>
<th>POPT,60</th>
<th>Speedup</th>
<th>SSMP,60</th>
<th>Speedup</th>
<th>PSMP,240</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total run time</td>
<td>2770</td>
<td>384</td>
<td>7.21</td>
<td>720</td>
<td><strong>3.85</strong></td>
<td>223</td>
<td>12.42</td>
</tr>
<tr>
<td>calculate_rho_elec</td>
<td>1166</td>
<td>73</td>
<td>15.97</td>
<td>123</td>
<td>9.48</td>
<td>46</td>
<td>25.35</td>
</tr>
<tr>
<td>integrate_v_rspace</td>
<td>1042</td>
<td>65</td>
<td>16.03</td>
<td>91</td>
<td>11.45</td>
<td>29</td>
<td>35.93</td>
</tr>
<tr>
<td>dbcsr_mm_cannon_multiply</td>
<td>186</td>
<td>79</td>
<td><strong>2.35</strong></td>
<td>48</td>
<td><strong>3.88</strong></td>
<td>55</td>
<td><strong>3.38</strong></td>
</tr>
<tr>
<td>dbcsr_make_images</td>
<td>160</td>
<td>24</td>
<td>6.67</td>
<td>21</td>
<td>7.62</td>
<td>19</td>
<td>8.42</td>
</tr>
<tr>
<td>fft_wrap</td>
<td>148</td>
<td>17</td>
<td>8.71</td>
<td>24</td>
<td>6.17</td>
<td>9</td>
<td>16.44</td>
</tr>
<tr>
<td>build_*</td>
<td>103</td>
<td>11</td>
<td>9.36</td>
<td>427</td>
<td><strong>0.24</strong></td>
<td>33</td>
<td><strong>3.12</strong></td>
</tr>
</tbody>
</table>

- We compare the 3 versions of CP2K against the POPT 4 processor run.
- The `build_*` routines were found to scale very poorly with PSMP and SSMP versions of the code.
- Issues also identified in `dbcsr_*`
Code identified for potential optimisation

• A number of areas were identified for optimisation
  • Most relate to improving OpenMP scalability
  • Improving the OpenMP scalability will benefit both Xeon Phi + host
• The areas identified were:
  • Reduce memory used in calculate_rho_elec – threads currently write to private array sections, this can be done using OpenMP locks
  • Improve scalability of 3D FFT by replacing 1D decomp with 2D partitioning of individual FFT rows to threads
  • build_core_* routines must be parallelised with OpenMP
  • MPI communication (MPI_Alltoall) my be overlapped with (threaded computation) in rs_distribute_matrix
  • Serial overhead in dbcsr_multiply_anytype – this is currently being worked on at CSCS in collaboration with Cray
Code identified for potential optimisation

- A number of areas were identified for optimisation
  - Most relate to improving OpenMP scalability
  - Improving the OpenMP scalability will benefit both Xeon Phi + host
- The areas identified were:
  - Reduce memory used in `calculate_rho_elec` – threads currently write to private array sections, this can be done using OpenMP locks
  - Improve scalability of 3D FFT by replacing 1D decomp with 2D partitioning of individual FFT rows to threads
  - `build_core_*` routines must be parallelised with OpenMP
  - MPI communication (`MPI_Alltoall`) may be overlapped with (threaded computation) in `rs_distribute_matrix`
  - Serial overhead in `dbcisr_multiply_anytype` – this is currently being worked on at CSCS in collaboration with Cray
Key findings

• Porting harder than expected due to code & compiler bugs
  • Our efforts will hopefully benefit the CP2K community
• Correct process and thread placement can be crucial
  • KMP_AFFINITY=verbose is your friend, use it!
• The Xeon E5-2670 host code outperforms the unoptimised Xeon Phi code by a factor of 4 with the H2O-64 benchmark
  • Lack of strong scaling, vectorisation & parallelisation limit the Xeon Phi performance
• Identified areas of code for optimisation
• These were worked on under a PRACE Preparatory Access project targeting the PRACE Xeon Phi cluster ‘EURORA’.
OPTIMISING CP2K ON XEON PHI

Fiona Reid
Optimising CP2K on Xeon Phi

- Aim: to optimise CP2K on Xeon Phi by improving the performance of the Langasite benchmark

- For this study we used the EU (PRACE) funded Xeon Phi-based cluster at CINECA called EURORA
  - Also used the EPCC and CSCS Xeon Phi cards for testing/development work

EURORA

**EURORA** Xeon Phi-based cluster at CINECA

<table>
<thead>
<tr>
<th>Model: Eurora prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture: Linux Infiniband Cluster</td>
</tr>
<tr>
<td>Processors Type:</td>
</tr>
<tr>
<td>- Intel Xeon (Eight-Core SandyBridge) E5-2658 2.10 GHz (Compute)</td>
</tr>
<tr>
<td>- Intel Xeon (Eight-Core SandyBridge) E5-2687W 3.10 GHz (Compute)</td>
</tr>
<tr>
<td>- Intel Xeon (Esa-Core Westmere) E5645 2.4 GHz (Login)</td>
</tr>
<tr>
<td>Number of nodes: 64 Compute + 1 Login</td>
</tr>
<tr>
<td>Number of cores: 1024 (compute) + 12 (login)</td>
</tr>
<tr>
<td>Number of accelerators: 64 nVIDIA Tesla K20 (Kepler) + 64 Intel Xeon Phi (MIC)</td>
</tr>
<tr>
<td>RAM: 1.1 TB (16 GB/Compute node + 32GB/Fat node)</td>
</tr>
<tr>
<td>OS: RedHat CentOS release 6.3, 64 bit</td>
</tr>
</tbody>
</table>

- Mix of host CPUs - take care with this!
- Minimal interactive access to Xeon Phi cards – must use batch system => debugging rather tricky!
Langasite benchmark

- Langasites are a family of solid oxides composed of Lanthanum, Gallium and Germanium
- Applications in fuel cells
- Want to find different orderings of La, Ga & Ge with minimum and maximum energy conductivity structures
- Many structures to evaluate, thus a cluster of Xeon Phis suggested as a possible architecture
- Original benchmark supplied by Ben Slater, UCL
Langasite benchmark

• Runtime & memory usage initially too high for Xeon Phi
  • Host took >10 hours on 16 MPI processes requiring 20 GB memory
  • 1.25 GB per processor

• To resolve this we:
  1. Reduced maximum number of iterations for both GEO_OPT and CELL_OPT from 300 to 1
  2. Reduced number of SCF cycles from 35 and 15 to 1
  3. Used SZV–GTH instead of DZVP–GTH as basis set for Oxygen
  4. Reduced value of MGRID%CUTOFF from 600 to 50 – trades accuracy for reduced memory and runtime
  5. Added SAVE_MEM keyword
Langasite benchmark

• Steps 1. and 2. reduced the host runtime to 350 seconds on 16 MPI procs
• Step 3. further reduced host runtime to 117 seconds
• Steps 4. and 5. greatly reduced memory such that each MPI process required 422 MB
• Total memory required doesn’t scale quite linearly with number of procs
  • However we could still run up to 16 MPI processes on the Xeon Phi
Host performance

Performance of Langasite benchmark on HOST node of EURORA

Best performance:
POPT 16 processes: 83s
Xeon Phi performance

Performance of Langasite benchmark EURORA Xeon Phi

Best performance:
PSMP 16 proc, 8 threads: 671s
Observations

- Xeon Phi gave best performance using PSMP version with 16 MPI processors running 8 threads (671 seconds)
- Uses 128 virtual threads, only ~50% of available capacity
- Fastest host time was 83 seconds, host is ~8 times faster
- Comparing possible FLOP/s Xeon Phi has around 2.5x the peak FLOP/s of the Xeon E5-2678W 16 node core
- Code isn’t using Xeon Phi to its full potential!!
- Earlier study, identified routines for optimisation
  - We’re going to focus on the build_core_* routines
Optimisation of build_core_ppl

- `build_core_ppl` computes contributions to the core Hamiltonian matrix due to the interaction of particles via their pseudopotentials.
- The `build_*` routines are costly with large thread counts:
  - Not threaded, thus can be very bad on Xeon Phi.
- Past attempts to thread these didn’t scale beyond 2 or 4 threads.
- Initially parallelised by adding a parallel region around a loop over all particles in a neighbour list.
- Original approach:
  - Create new data structure to hold the data describing a task.
  - Task = one or more iterations of the main DO WHILE loop.
  - Before parallel region entire neighbour list iterated over in serial => list of tasks.
  - Add a new parallel loop which loops over the independent tasks.
  - Scaled to 2 or 4 threads but not beyond.
Optimisation of `build_core_ppl`

- We used a new approach
  - Modified the iterator code to enable the iterator to be used from within a parallel region
  - Involved adding optional arguments to the iterator, specifically the thread number from which the iterator is accessed
    • Allows us to access to elements in the list in a thread-safe manner
  - Should go faster – no need to pre-compute the list of neighbours
  - Should be perfectly load balanced – each thread keeps asking for iterations in a dynamic manner until all work is complete
  - Need to ensure that updates to any shared variables (e.g. `force`, `virial`, `h_block` variables) are protected
Optimisation of `build_core_ppl`

- After making the changes the code was tested using CP2K’s regression test suite
- Most development and testing was carried out on the host
  - Faster turnaround (interactive)
  - Better tools available for debugging
- If all tests passed on host, we tested on the Xeon Phi
Performance of `build_core_ppl`

Also tried splitting the force & virial updates into separate critical regions. No benefit from this.
Speedup of `build_core_ppl`!
Optimisation of `build_core_ppnl`

- Similar to `build_core_ppl`
- Has two separate DO WHILE loops over particle pairs using the same iterator
- First loop computes overlap integrals (`sap_int`) for use in second loop computing Hamiltonian matrix elements
- Means we need to protect updates to `sap_int` in addition to any other shared variables
Performance of `build_core_ppnl`
Speedup of `build_core_ppnl`
Fast Fourier transforms

- As mentioned previously, versions of MKL prior to 11.1.0 were not fully thread-safe.
- Looked at the performance of CP2K for different FFT libraries.
- Used a simple benchmark `fft.inp` which performs computations on a 125 x 125 x 125 element grid.
- Slightly bigger grid size than Langasite.
Fast Fourier transforms

PSMP fft.inp performance with 2 MPI processes on Europa MIC cards

Time (seconds)

Number of threads

- CP2K - MKL 11.1.0
- fft3d_ps - MKL 11.1.0
- CP2K - FFTW 3.3.3
- fft3d_ps - FFTW 3.3.3
Fast Fourier transforms

• The Langasite benchmark only spends about 1% of time in FFT computations
• Minimal impact on the performance when using MKL
• However other CP2K computations could be very different
  • Use MKL if you can
Other optimisations

- We also made a number of other changes to the code
- Improved the performance of `rs_distribute_matrix` overlapping local threaded data movement with the `MPI_Alltoall` call
- Also, various bug fixes, compiler bugs etc
Optimised host performance

Performance of Langasite benchmark on HOST node of EURORA

Best performance:
POPT 16 processes: 83s
Optimised Xeon Phi performance

Performance of Langasite benchmark EURORA Xeon Phi

Best performance: PSMP 16 proc, 15 threads: 451s
Summary

- Obtained a total speed up of 49%
- Despite this, it’s still faster to use the host!
- Need to parallelise more of the code – e.g. more OpenMP
- Improve vectorisation
- or make use of offload mode
- Our improvements have benefitted the host version too – means all users of CP2K benefit
IPCC PROJECT AT EPCC: EXPERIENCES SO FAR

Adrian Jackson and Fiona Reid
IPCC

• Intel Parallel Computing Centre

“Intel® Parallel Computing Centres are universities, institutions, and labs that are leaders in their field, focusing on modernizing applications to increase parallelism and scalability through optimizations that leverage cores, caches, threads, and vector capabilities of microprocessors and coprocessors” source https://software.intel.com/en-us/ipcc

• Currently at least 48 worldwide
• Full list at https://software.intel.com/en-us/ipcc
IPCC Proposal

- Two main aims
  - Port and optimise codes for Xeon Phi
  - Optimise codes for Xeon for ARCHER system

- Target ‘Grand Challenges’ codes which are also heavily used in the UK
  - codes we have strong knowledge of/working relationship with
IPCC Proposal

• Work on 2 simulation codes per year
  • Planned CP2K and CASTEP in first year
  • COSA and GS2 in the second year

• Actually…
  • Working on CP2K, COSA, and GS2 in the first year
  • Will target CASTEP in the second year (and any remaining work on the other codes as necessary)
General approach

- Investigate and optimise vectorisation of codes
  - Use profiler and compiler tools to evaluate vectorisation
  - Modify computationally expensive code to improve vectorisation
- Improve/implement hybrid parallelisation
  - Can help for both standard and phi systems
  - Reduce memory footprint
- Reduce serial code
  - I/O etc....
GS2

- Flux-tube gyrokinetic code
  - Initial value code
  - Solves the gyrokinetic equations for perturbed distribution functions together with Maxwell’s equations for the turbulent electric and magnetic fields
  - Linear (fully implicit) and Non-linear (dealiased pseudo-spectral) collisional and field terms
  - 5D space – 3 spatial, 2 velocity
  - Different species of charged particles
- Advancement of time in Fourier space
- Non-linear term calculated in position space
  - Requires FFTs
  - FFTs only in two spatial dimensions perpendicular to the magnetic field
- Heavily dominated by MPI time at scale
Vector (de)optimisations

- Vector optimising work unsuccessful
  - A number of poorly vectorising targets identified
  - Code restructuring and directives not able to improve performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Compiler flags</th>
<th>Compiler directives</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>invert_rhs_1</td>
<td>-O2 (original)</td>
<td>-O2 (original)</td>
<td>16.46</td>
</tr>
<tr>
<td>get_source_term</td>
<td>-align array64byte attributes align, vector aligned</td>
<td>16.73</td>
<td></td>
</tr>
<tr>
<td>get_source_term</td>
<td>-align array64byte attributes align, vector aligned</td>
<td>16.99</td>
<td></td>
</tr>
</tbody>
</table>
Complex number optimisation

• Much of GS2 uses FORTRAN Complex numbers
  • However, often imaginary and real parts are treated separately
  • Can affect vectorisation performance

• Work underway to replace with separate arrays
  • Initial performance numbers demonstrate performance improvement on Xeon Phi
  • 2-3% for a single routine when using separate arrays
COSA

- Fluid dynamics code
  - Harmonic balance (frequency domain approach)
  - Unsteady navier-stokes solver
  - Optimise performance of turbo-machinery like problems
  - Multi-grid, multi-level, multi-block code
  - Parallelised with MPI and with MPI+OpenMP
Work

- Focus on vectorisation optimisations
- Reasonable simulation: 20% in MPI, 80% in user code and blas/lapack (60% user code)
- Vector performance poor
  - Order of magnitude more vector scalar instructions than packed vector instructions
- Intel compiler reports problems vectorising time consuming loops
  - Vector dependencies, etc…
## Xeon Phi Performance

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Number of hardware elements</th>
<th>Occupancy</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 MPI processes</td>
<td>1/2</td>
<td>8/16</td>
<td>2105.71</td>
</tr>
<tr>
<td>16 MPI processes</td>
<td>2/2</td>
<td>16/16</td>
<td>1272.54</td>
</tr>
<tr>
<td>64 MPI processes</td>
<td>1/2</td>
<td>64/240</td>
<td>3874.45</td>
</tr>
<tr>
<td>64 MPI processes 3 OpenMP threads</td>
<td>1/2</td>
<td>192/240</td>
<td>2963.58</td>
</tr>
<tr>
<td>118 MPI processes 4 OpenMP threads</td>
<td>2/2</td>
<td>472/480</td>
<td>2118.05</td>
</tr>
<tr>
<td>128 MPI processes 3 OpenMP threads</td>
<td>2/2</td>
<td>384/480</td>
<td>1759.30</td>
</tr>
</tbody>
</table>

- **Hardware:**
  - 2 x Xeon Sandy Bridge 8-core E5-2650 2.00GHz
  - 2 x Xeon Phi 5110P 60-core 1.05GHz

- **Test case**
  - 256 blocks
  - Maximum 7 OpenMP threads
COSA

• Identified that reading input is significant overhead for this code
  • Output is done using MPI-I/O, reading is done serially

• Working to parallelise all I/O
  • Reduce file locking and serial parts of the code
  • Should improve performance on the Xeon Phi (and other platforms)

• 3D version of the code now developed
  • Porting optimised and hybrid version to this
CP2K

- Atomistic and molecular simulations of solid state, liquid, molecular, and biological system
- MPI and hybrid parallelisations implemented
- Heavily uses internal and external libraries for core computations
- Other sites working on Xeon Phi
  - Offload functionality
  - Investigating compiler optimisations
- EPCC has previously worked on a native mode Xeon Phi port
  - This work identified a number of vectorisation targets
**CP2K Vector (de)optimisations**

- Vector optimising work unsuccessful so far…
- CP2K uses auto-tuning library routines for core kernels
- Vectorising these routines struggled due to code structure

<table>
<thead>
<tr>
<th>Code version</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original code</td>
<td>2.423632</td>
</tr>
<tr>
<td>Adding !DIR$ IVDEP to loop over ig</td>
<td>2.472624</td>
</tr>
<tr>
<td>Attempt 1: Array syntax</td>
<td>2.438629</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ IVDEP on loop over ig</td>
<td>*2.437631</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ VECTOR ALWAYS on loop over ig</td>
<td>2.430630</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ SIMD on loop over ig</td>
<td>2.463625</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !SOMP SIMD private(i,s) on loop over ig</td>
<td>2.484623</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + align map and pol_x</td>
<td>2.479622</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + align map and pol_x + !SOMP SIMD on loop over ig</td>
<td>2.524676</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ IVDEP on loop over ig</td>
<td>2.477623</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !DIR$ SIMD on loop over ig</td>
<td>2.580608</td>
</tr>
<tr>
<td>Attempt 1: Array syntax + !SOMP SIMD private(i,s) on loop over ig</td>
<td>2.620602</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax</td>
<td>2.475624</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !DIR$ IVDEP on loop over ig</td>
<td>2.473624</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !DIR$ SIMD on loop over ig</td>
<td>2.580608</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + !SOMP SIMD private(i,s) on loop over ig</td>
<td>2.620602</td>
</tr>
<tr>
<td>Attempt 2: use ivec(ig) array and array syntax + localmap 1d array used to compute I</td>
<td>2.475624</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(i,b) to stop(i,b) for each block of contiguous iterations</td>
<td>2.626000</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(i,b) to stop(i,b) for each block of contiguous iterations + !DIR$ IVDEP on loop over i</td>
<td>2.625601</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(i,b) to stop(i,b) for each block of contiguous iterations + !DIR$ VECTOR ALWAYS on loop over i</td>
<td>2.627600</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(i,b) to stop(i,b) for each block of contiguous iterations + !DIR$ SIMD on loop over i</td>
<td>2.582607</td>
</tr>
<tr>
<td>Attempt 3: replace the ig loop with loops over countblocks and start(i,b) to stop(i,b) for each block of contiguous iterations + !SOMP SIMD private(s) on loop over i</td>
<td>2.634599</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc</td>
<td>2.769579</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ IVDEP on loop over i</td>
<td>2.760580</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ VECTOR ALWAYS on loop over i</td>
<td>2.755581</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !DIR$ SIMD on loop over i</td>
<td>2.754581</td>
</tr>
<tr>
<td>Attempt 4: as per attempt 3 but now split into two loops, one over ig and one over countblocks etc + !SOMP SIMD private(s) on loop over i</td>
<td>2.757580</td>
</tr>
</tbody>
</table>
Highlights

• Hybrid version of GS2 created
  • Beneficial for many different problems and machines
• Integration of Intel compilers and libraries into CP2K test suite
  • Enables a much wider range of software/hardware configurations to be targeted
• COSA Xeon Phi performance
  • COSA can scale across multiple Xeon Phi’s, enabling very large scale problems to be tackled in the future
• Developing training courses/materials
  • Strong addition to our training program for national services
• Facilitated a wide range of contacts/discussions
  • A number of external organisations (academic and industry) have approached us for collaborations based on our IPCC
Lessons learned to date/comments

• Compiler availability would improve development process
  • ANOther compiler and MPI library supporting the Phi would enable quicker investigation of bugs/errors
• All the codes we’ve worked on are FORTRAN
  • Heavily optimised production codes, makes increasing optimisation more difficult
• Single code base work highly favoured
  • Large scale codes won’t maintain mixed source versions
• Hybrid parallelisations will help elsewhere
  • Obvious target for many MPI programs
• Intel compilers v15 has impacted performance across the board for our codes
  • Slower with v15 vs v14
• MPI across Xeon Phi’s can heavily impact performance
  • Global comms dominated codes don’t currently scale
  • Local comms codes can scale well
Acknowledgements

• This work was funded by PRACE and Intel via the EPCC IPCC

• We acknowledge PRACE for awarding us access to the resource EURORA based in Italy at CINECA

• This work used ARCHER, the UK National Supercomputer Service (http://www.archer.ac.uk).

• The support of Prof. Jürg Hutter, Prof. Joost VandeVondele and staff at CSCS for providing access to HPC systems used for testing the code is gratefully acknowledged.

• Alfio Lazzaro is acknowledged for providing us with an optimised version of libsmm used on the ARCHER machine.