

# Monitoring the Cray XC30 Power Management Hardware Counters

Michael Bareford  
Edinburgh Parallel Computer Centre  
University of Edinburgh  
Edinburgh, UK  
Email: michael.bareford@epcc.ed.ac.uk

**Abstract**—This paper explores the usefulness of the data provided by the power management (PM) hardware counters available on the Cray XC30 platform. PM data are collected for two molecular modelling codes, DL\_POLY and CP2K, both of which are run over multiple compute nodes. The first application is built for three programming environments (Cray, Intel and gnu): hence, the data collected is used to test the hypothesis that the choice of compiler should not impact energy use significantly. The second code, CP2K, is run in a mixed OpenMP/MPI mode, allowing us to explore the relationship between energy usage and thread count.

The Cray-compiled DL\_POLY code had the lowest energy usage on average, 3-4% lower than the Intel and gnu results. In general, energy usage follows execution time. For the CP2K code, an energy-usage sweet spot of three threads per MPI process was revealed; for higher thread counts, execution times increase monotonically with energy usage.

**Keywords**—Cray XC30; Power Management; MPI; OpenMP

## I. INTRODUCTION

This document is concerned with how to record and analyse the data provided by the power monitoring hardware counters on ARCHER [1], a Cray XC30 platform. Basic descriptions of these counters can be obtained by running `papi_native_avail` on the compute nodes of the host system using the `aprun` command.

There are two groups of power monitoring counters, running average power limit (RAPL) and power management (PM), see the following two tables.

Table I  
RUNNING AVERAGE POWER LIMIT COUNTERS

Name (units)	Description (value)
UNITS:POWER	Scaling factor for power (3)
UNITS:ENERGY	Scaling factor for energy (16)
UNITS:TIME	Scaling factor for time (10)
THERMAL_SPEC (W)	Thermal specification (130)
MINIMUM_POWER (W)	Minimum power (64)
MAXIMUM_POWER (W)	Maximum power (200)
MAXIMUM_TIME_WINDOW (s)	Maximum time window (0.046)
PACKAGE_ENERGY (nJ)	Energy used by chip package
DRAM_ENERGY (nJ)	Energy used by package DRAM
PP0_ENERGY (nJ)	Power-plane zero package energy

The counters have a time resolution of 100 ms [2]. The term *package* relates to the compute node processor:

Table II  
POWER MANAGEMENT COUNTERS

Name (units)	Description
PM_POWER_CAP:NODE (W)	Compute node power cap
PM_POWER:NODE (W)	Compute node point-in-time power
PM_ENERGY:NODE (J)	Compute node accumulated energy
PM_POWER_CAP:ACC (W)	Accelerator power cap
PM_POWER:ACC (W)	Accelerator point-in-time power
PM_ENERGY:ACC (J)	Accelerator accumulated energy
PM_FRESHNESS	Counter update count

each node has two 12-core processors and therefore two packages. This means there is one set of RAPL counters per socket, whereas for PM counters, there is one set per node (or per two sockets). The PP0 package energy counter value is a summation of the energy used by all the cores (including processor caches) within a package [3]. Note, several of the hardware counters are by definition fixed (e.g., UNITS:POWER) and so only ever have one value, which is given in brackets at the end of the counter description.

Two molecular modelling codes (DL\_POLY v4.05 and CP2K v2.6.14482) are used to show how to access power management data for simulation jobs that run over multiple compute nodes. The former application achieves parallelism through MPI exclusively, and can be built for all three programming environments available on ARCHER (Cray, Intel and gnu). Here, the choice of compiler (crayftn v8.3.7, ifort v14.0.4.211, gfortran v4.9.2) should not impact energy use, nevertheless this paper will confirm whether this is indeed the case. The second code, CP2K, will be run in a mixed OpenMP/MPI mode (using the gnu environment): this provides a more useful case, allowing one to investigate whether or not adding more threads raises or lowers the energy use.

## II. CRYPAT

We begin with the Cray Performance and Analysis Tool (CrayPat) as the most straightforward way to obtain power management data. This tool requires that the application code is instrumented before it is submitted to the job launcher. Before the code is compiled the perftools software module must be loaded.

```
module load perftools
```

Then, after compilation, the application executable is instrumented by the `pat_build` command.

```
pat_build DLPOLY.Z
```

This produces a new executable with a `+pat` suffix. It is this instrumented program that is launched via the `aprun` command and controlled according to various properties supplied by the job submission script, an excerpt of which follows.

```
module load perftools

export PAT_RT_SUMMARY=1
export PAT_RT_PERFCTR=UNITS:POWER, /
UNITS:ENERGY,UNITS:TIME, /
THERMAL_SPEC,MINIMUM_POWER,MAXIMUM_
POWER,MAXIMUM_TIME_WINDOW, /
PACKAGE_ENERGY,PP0_ENERGY,DRAM_ENERGY, /
PM_POWER_CAP:NODE,PM_POWER:NODE,PM_
ENERGY:NODE, /
PM_FRESHNESS,PM_GENERATION,PM_
STARTUP,PM_VERSION

export OMP_NUM_THREADS=1

aprun -n 96 ./DLPOLY.Z+pat >& stdouterr
```

Note, the ACC power management counters are missing from the list assigned to the `PAT_RT_PERFCTR` environment variable. Although those three counters are returned by `papi_native_avail`, CrayPat reports an error, insisting that those performance counter event names do not exist. Having the summary flag set to one will generate one set of values for the counters listed in the submission script. If the code runs successfully, one of the output files will have a `.xf` extension, which can then be analysed via the `pat_report` command.

```
pat_report ./DLPOLY.Z+pat+43271-13s.xf
```

Something like the following output is then generated.

```
...
Program HW Performance Counter Data
...
THERMAL_SPEC 130 W
MINIMUM_POWER 64 W
MAXIMUM_POWER 200 W
MAXIMUM_TIME_WINDOW 0.046 s
PACKAGE_ENERGY 530.413 J
```

```
PP0_ENERGY 387.626 J
DRAM_ENERGY 58.021 J
PM_POWER:NODE 286 W
PM_ENERGY:NODE 767.219 J
PM_FRESHNESS 28.417
PM_STARTUP 1414408518017769 cycles
...
```

In the previous example, the `DL_POLY` code was run over four nodes, and so, by default the energies are averages, whereas the powers (also by default) are maximum values: e.g., `PACKAGE_ENERGY` is the average energy used by the eight processors, and `PM_POWER:NODE` is the maximum point-in-time power recorded (for one node) during the execution run. The `pat_report` tool has the flexibility to accept different aggregations [2].

Unfortunately, it doesn't seem possible to record multiple counter readings during execution on a per node basis. A CrayPat sampling job is specified by using the `-S` option with the `pat_build` command and by adding two extra options to the job script, see below.

```
export PAT_RT_EXPERIMENT=samp_pc_time
export PAT_RT_INTERVAL=1000
```

The interval is in microseconds. This setup does not generate a set of sampling data that could be used to investigate how energy use varies throughout the run; the only data generated specific to the selected counters appears to be in aggregate form only.

### III. PM MPI LIBRARY

Hart et al. 2014 [4] have provided a library that allows one to monitor the power management (hereafter PM) counters directly. This library, `pm_lib`, accesses the actual counter files, located at `/sys/cray/pm_counters`. There are five counters available, instantaneous power and cumulative energy for normal nodes and for accelerated nodes — these measurements cover CPU, memory and any other hardware contained on the processor daughter card; any consumption due to the Aries network controllers and beyond is excluded however. The fifth counter is the freshness counter, which should always be read before reading the other counters. After a set of readings have been taken, the freshness counter is read again and if the value is unchanged, we know we have a consistent set of measurements: i.e., the power and energy were taken at the same time. The counters have a temporal resolution of 100 ms. The library can be linked with Fortran and C/C++ programs: however, for codes running on several multi-core nodes it is necessary to extend the library in two significant ways. First, only one MPI process per node must use `pm_lib` to access the PM counters on that node; and second, only one MPI process (e.g., rank 0) should collate the data, writing it to a single file.

The result of these extensions is a new library called `pm_mpi_lib` [7], which has an interface featuring just three functions. When instrumenting a code for the first time, the area of interest is the main loop that controls how the simulation progresses. For DL\_POLY, this is the loop at the start of the `./VV/w_md_v.f90` source file, which can be instrumented as follows. A call to the `pm_mpi_lib` function, `pm_mpi_open`, should be placed immediately before the loop. This function establishes the MPI rank of the calling process and then obtains the unique number of the node on which the process is running via `MPI_Get_processor_name`. The node number is then used to split the `MPI_COMM_WORLD` communicator: i.e., the node number is used as the *colour* of the sub-communicator, which is effectively a communicator for all processes on the same node. Thus, an appropriate `MPI_Allreduce` can be used to determine which process has the lowest numerical rank value on any given node, thereby choosing the process that will monitor the PM counters. All such monitoring processes (one per node) then go on to create another sub-communicator, one that unites them all, which allows rank 0 to determine the number of monitors. In addition to these tasks, the monitoring processes open their respective PM counter files; rank 0 also gets the current MPI time.

The second `pm_mpi_lib` function call, `pm_mpi_monitor`, can be placed at various points within the application loop. As the name suggests, this function allows the monitoring processes to read the PM counters (all other processes simply wait at a MPI barrier). The monitoring processes go on to perform an `MPI_Allreduce`, thereby summing the counter values over all nodes; thus the total cumulative energy can be obtained. For the point-in-time power counters however, the sum is divided by the number of monitoring processes; this gives an average power reading. Rank 0 then writes this data to the output file, see the readme file in the `pm_lib_mpi` GitHub repository [7] for a description of the output format.

Finally, the third and last call, `pm_mpi_close`, is placed immediately outside the main application loop: this call ensures that each monitoring process closes its PM counter files and that rank 0 also closes the output file.

#### IV. RESULTS

The three functions of the `pm_mpi_lib` library have been used to instrument the DL\_POLY and CP2K codes mentioned earlier, such that the PM counters for cumulative energy and point-in-time power are read for each node (four for DL\_POLY and eight for CP2K). The data is then collated and output as a total energy and as an average power for each time step of the simulation.

##### A. Compiler Dependence

Figures 1 and 2 show histograms of energy use per simulation step, defined as one iteration of the loop at the start of the DL\_POLY source file, `./VV/w_md_v.f90`. There are three sets of histograms, one for each compiler environment: grey represents the Cray compiler, blue is Intel and red is gnu. For each run, the number of nodes assigned (4) and the input data set were the same: the latter was taken from the standard DL\_POLY test case 40 (ionic liquid dimethylimidazolium chloride), see Section 8.1 of the user manual [5]. Furthermore, the molecular simulation defined by the test case was extended by increasing the value of the steps parameter in the CONTROL file from 750 to 20 000.

The histograms do not correspond with a single normal distribution: in general, there is one main peak and two minor ones appearing at lower energies, see Figure 1. In addition, there are high-energy simulation steps that occur every one thousand iterations (Figure 2): this is how often the DL\_POLY restart files are written to disk.

The same DL\_POLY job was run six times for each compiler environment and an average performance was calculated in order to tease out any compiler dependence. Repeatedly running exactly the same executable with the same input data will produce some variation in energy usage: however, this variation is mitigated by the fact that all jobs were run on the ARCHER test development server, where resource contention is minimal.

The Cray-compiled code had the lowest energy usage ( $1.92 \pm 0.02$  MJ over  $1748 \pm 2.6$  s), followed by the Intel-compiled code ( $1.97 \pm 0.01$  MJ over  $1770 \pm 2.7$  s), and lastly, the gnu-compiled code ( $2 \pm 0.02$  MJ over  $1823 \pm 2$  s). The Cray energy usage is approximately 3% lower ( $1.92/1.97 \approx 0.97$ ) than the Intel result and 4% lower than the mean result for the gnu runs. Unsurprisingly, energy usage follows execution time. However, the Intel runs show a marked difference as regards the high-energy iterations: for the Cray and gnu runs, these high-energy iterations are a factor of sixteen higher than the energy used for 99.9% of the simulation steps, whereas for the Intel runs, this factor increases to around ninety-five. Note, the Intel-compiled code would be the most energy efficient, if the total for the high-energy iterations (0.18 MJ) was reduced such that it matched the high-energy totals achieved by the Cray and gnu runs: in fact, this hypothetical reduction would make the Intel runs the most efficient ( $\sim 5\%$  lower than the Cray energy usage).

The distributions of execution times per simulation step for the same runs as those featured in Figure 1 are also shown, in order to understand more fully the impact of execution times on energy consumption.

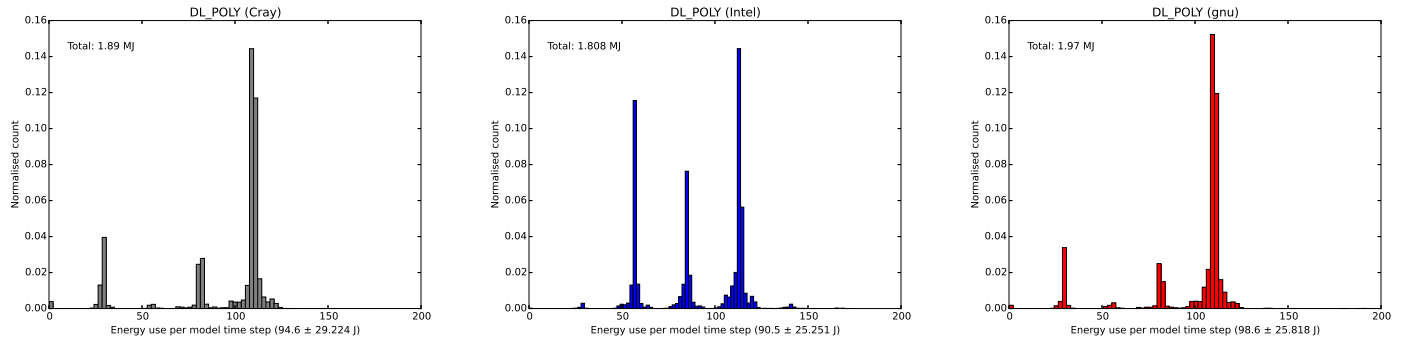


Figure 1. Histograms of energy used per DL\_POLY simulation step for energies less than 200 J. Plots are based on results obtained from the third run of six for each compiler environment, Cray (left), Intel (middle) and gnu (right).

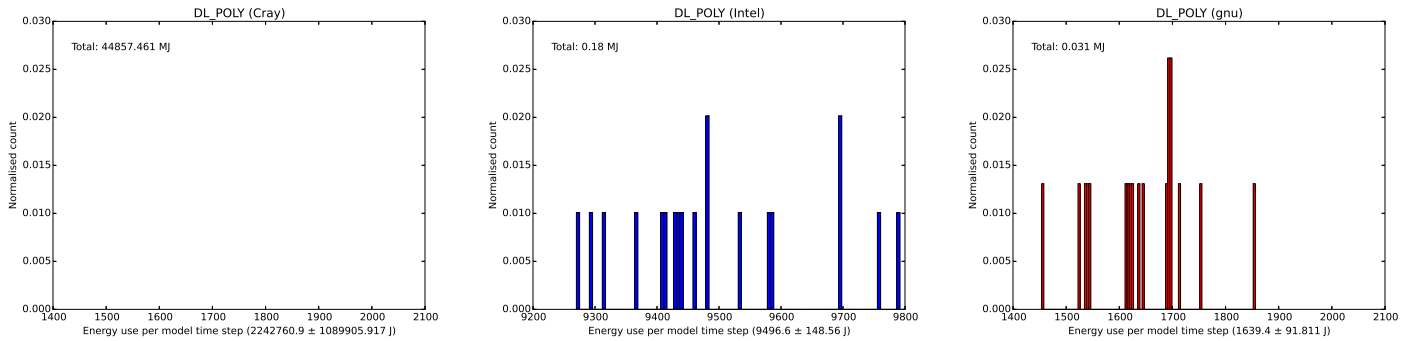


Figure 2. Histograms of energy used per DL\_POLY simulation step for energies greater than 200 J.

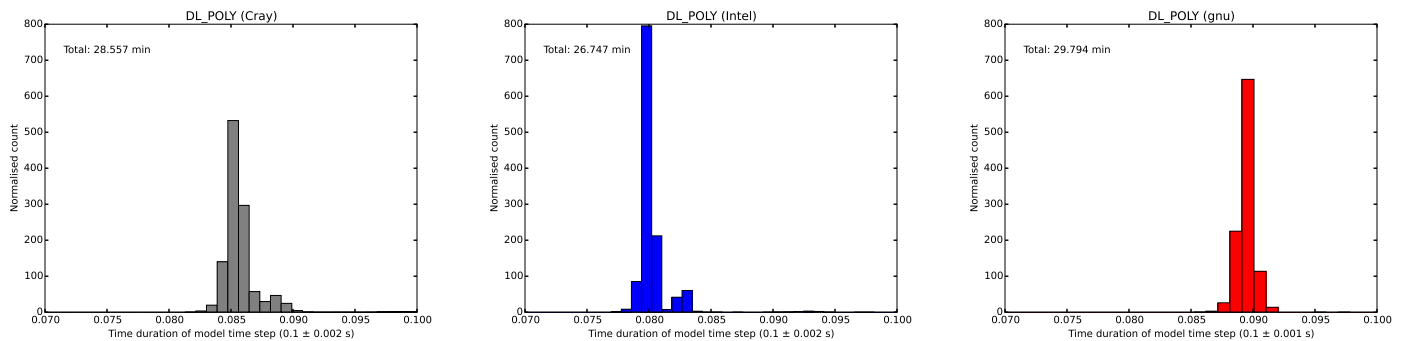


Figure 3. Histograms of time duration per DL\_POLY simulation step for times less than 0.1 s. Plots are based on results obtained from the third run of six for each compiler environment, Cray (left), Intel (middle) and gnu (right).

Again, the distributions are divided into two groups, short iterations (Figure 3) and long iterations (Figure 4) occurring every 1000 simulation steps (i.e., the restart dumps). The high energy steps consume more energy because those iterations take longer to execute; also, the Intel high-energy iterations run for the longest (by almost an order of magnitude).

Figure 5 presents example point-in-time power histograms. Unlike the cumulative energy results, these histograms correspond with a single normal distribution, giving a meaningful average and deviation, see the x-axis labels. Hence, the normal distributions inferred by the power histograms for all eighteen runs are also given (Figure 5, right).

The Cray and gnu compiled codes have similar dis-

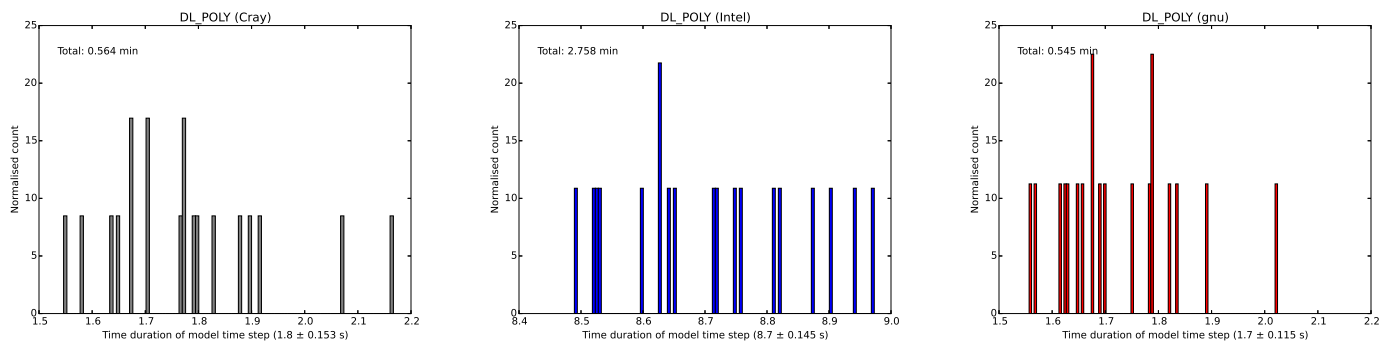


Figure 4. Histograms of time duration per DL\_POLY simulation step for times greater than 0.1 s.

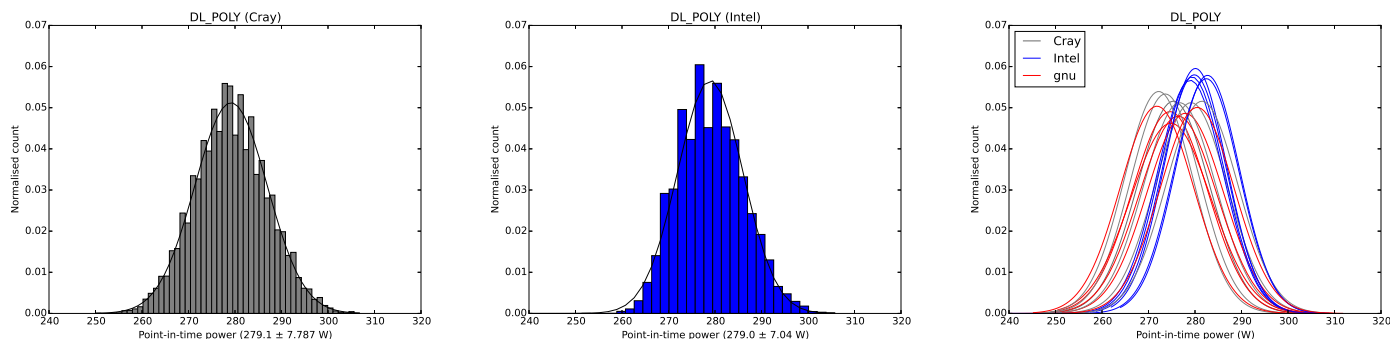


Figure 5. The point-in-time power histograms based on results obtained from the second run of six for the Cray (left) and Intel (middle) compiler, and the normal distributions inferred from these histograms for all eighteen runs (right).

tributions as regards power measurements: therefore, the difference in energy usage for these two codes can be attributed to execution time. Overall, the Intel runs show the highest mean power and we also know that the high-energy iterations for the Intel runs are the most time consuming. Nevertheless, the Intel energy usage is mitigated by the fact that the majority low-energy iterations execute the fastest on average: in the middle plot of Figure 4, the Intel distribution is furthest to the left. This mitigating factor is perhaps the reason why the Intel runs do not show the highest energy consumption.

### B. OpenMP Threads

This section examines the impact of OpenMP thread count on energy usage when running the gnu-compiled CP2K code with the input data set entitled H2O-1024.inp, which can be found in the `./tests/QS/benchmark/` folder that resides within the CP2K installation directory. A particular loop within the CP2K source file, `./src/motion/md_run.F`, was instrumented with the three `pm_mpi_lib` functions mentioned earlier. The loop in question is labelled `Real MD Loop` and exists within the `qs_mol_dyn_low` subroutine. In addition, the value of the

`MOTION.MD.STEPS` input parameter (i.e., the number of loop iterations) was changed from 10 to 100. The results were generated by running the CP2K code over eight nodes for a variety of threading options. Table III reveals an

Table III  
CUMULATIVE ENERGIES AND RUN TIMES FOR DIFFERENT CP2K  
THREADING OPTIONS

<b>aprun options</b>	<b>Energy [MJ]</b>	<b>Time [hr]</b>
-n 192 -N 24 -S 12 -d 1	52.263	5.4
-n 96 -N 12 -S 6 -d 2	49.727	5.94
-n 64 -N 8 -S 4 -d 3	45.052	5.27
-n 48 -N 6 -S 3 -d 4	48.819	6.26
-n 32 -N 4 -S 2 -d 6	54.284	7.47
-n 24 -N 3 -d 8	71.54	11.57
-n 16 -N 2 -S 1 -d 12	91.342	16.72

energy-usage sweet spot for 3 threads per MPI process; increasing the thread count further causes the performance to deteriorate. The `aprun` option, `-ss`, was also set for six of the seven cases: this option was not used for the eight thread case, since it is not possible to evenly divide three MPI processes over the two blocks of memory (or NUMA

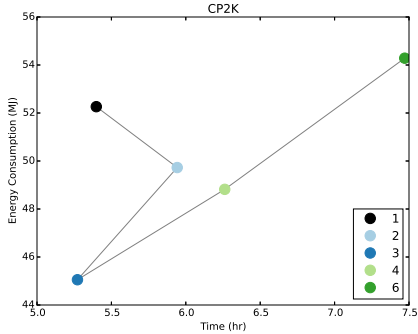


Figure 6. CP2K energy consumption against runtime for different OpenMP threading options (1-4 and 6 threads).

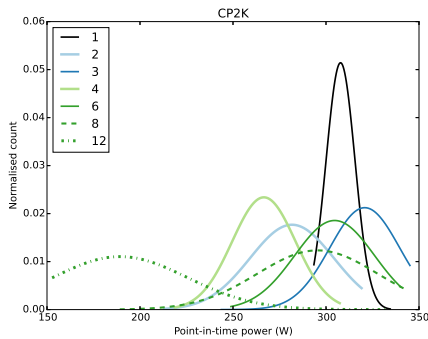


Figure 7. The normal distributions inferred from the point-in-time power histograms across all OpenMP threading counts used for the CP2K code.

regions) that exist on each compute node.

Figure 6 allows the reader to see the change in energy consumption and runtimes when increasing the thread count from one to six. Further increases in the thread count merely lengthen the runtime, thereby consuming more energy, see the last two rows of Table III. Interesting behaviour occurs for low thread counts, e.g., when changing from one to two threads, the energy use comes down even though the runtime has increased. At three threads, both the runtime and energy consumption are reduced — this is the sweet spot.

The deviation in power readings becomes much more pronounced when the number of threads per MPI process is greater than one, see Figure 7. Single-threaded execution gives a sharply defined peak at around 310W. The power distributions increase in width for thread counts of 2-6, and then become even wider for the highest thread counts of 8 and 10. From left to right, the distribution peaks correspond to 12, 4, 2, 8, 6, 1 and 3 threads: this means that specifying 12 threads per MPI process draws the lowest mean power, whereas 3 threads results in the highest power usage.

## V. CONCLUSIONS AND FURTHER WORK

The DL\_POLY results show the expected correlation between energy use and runtime. The choice of compiler

does produce noticeable differences in energy usage: the Cray-compiled code uses the least energy, followed by Intel then gnu. However, the results also show that the evenly-spaced restart write iterations (19 out of 20 000) run for significantly longer, and furthermore, this disparity is more extreme for the Intel runs. Closer examination of the data, reveals that the Intel runs would use the least energy, if the compiler options could be set such that the Intel high-energy iterations had runtimes comparable with the Cray and gnu results.

Energy use will depend on the number of threads per MPI process: using multiple threads can reduce runtimes and energy usage but not beyond a certain thread count. Three threads is the optimum thread count for CP2K running over eight nodes with the H2O-1024.inp data set. Further work could investigate the importance of node assignment within the ARCHER dragonfly topology [6] as regards energy consumption. For example, running with three threads per MPI process, one could compare energy usages for the following scenarios.

- 1) All eight nodes from the same chassis.
- 2) Four nodes from one chassis and four nodes from a different chassis.
- 3) Same as scenario two but involving a chassis from a different group.

The first scenario involves communication within the rank 1 network only, the second will also feature communication over the rank 2 network, whereas the final scenario will involve communication over the optical rank 3 network. It is assumed that runtimes will increase should more communication hops between nodes be required: hence, these different scenarios should result in different node energies being recorded. The usefulness of this work would be in understanding the energy cost of communicating via the rank 2 and/or rank 3 networks. This work would of course need to be done on ARCHER itself as opposed to the test and development server (tds), which comprises only one group containing two cabinets. Note, the CP2K tests involving thread counts of 4 and 12 were performed on nodes that resided in different cabinets; the other CP2K tests involved nodes within the same chassis, thereby utilising the rank 1 network only. Regarding the DL\_POLY results, 16 out of 18 runs were executed on nodes within the same chassis: the remaining two runs (one Cray and one gnu) featured a single node in the second tds cabinet.

Finally, it might be possible to alter the pm\_mpi\_lib library so that it uses CrayPat API calls, specifically, PAT\_record and PAT\_counters. The first function allows non-monitoring processes to turn off CrayPat monitoring, avoiding unnecessary overhead. The second is an alternative to reading the PM counter files directly: it reads the counters specified by the environment variable, PAT\_RT\_PERFCTR (see Section II). Thus, the addition

of new counters, such as ones from the RAPL group (Section I), would not necessarily require a rebuild of the library and application code.

#### ACKNOWLEDGMENT

The author would like to thank Harvey Richardson and Alastair Hart of the Cray Centre of Excellence, EPCC, Edinburgh.

#### REFERENCES

- [1] EPCC. (2015) Archer user guide. Web site. [Online]. Available: <http://www.archer.ac.uk/documentation/user-guide/>
- [2] C. Inc., *Using Cray Performance Measurement and Analysis Tools*, s-2376-60 ed. Seattle, WA, USA: Cray Inc., September 2012. [Online]. Available: <http://docs.cray.com/books/S-2376-60/>
- [3] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, Pittsburgh, PA, USA, September 2012, pp. 262–268. [Online]. Available: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6337489&queryText%3DMeasuring+Energy+and+Power+with+PAPI>
- [4] A. Hart, H. Richardson, J. Doleschal, T. Ilsche, M. Bielert, and M. Kappel, "User-level power monitoring and application performance on cray xc30 supercomputers," in *Proceedings of CUG2014*, Seattle, WA, USA, May 2014. [Online]. Available: [https://cug.org/proceedings/cug2014\\_proceedings/includes/files/pap136.pdf](https://cug.org/proceedings/cug2014_proceedings/includes/files/pap136.pdf)
- [5] I. T. Todorov and W. Smith, *The DL\_POLY\_4 User Manual*, 4th ed., Warrington, UK, January 2014. [Online]. Available: [http://www.ccp5.ac.uk/DL\\_POLY/MANUALS/USRMAN4.pdf](http://www.ccp5.ac.uk/DL_POLY/MANUALS/USRMAN4.pdf)
- [6] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," Seattle, WA, USA, 2012. [Online]. Available: <http://www.cray.com/Assets/PDF/products/xc/CrayXC30Networking.pdf>
- [7] M. R. Bareford, A. Hart, and H. Richardson. (2014) The pm mpi library. GitHub repository. [Online]. Available: [https://github.com/cresta-eu/pm\\_mpi\\_lib](https://github.com/cresta-eu/pm_mpi_lib)