On the trade-offs between energy to solution and runtime for real-world CFD test-cases.

Michael Bareford
EPCC
University of Edinburgh
Edinburgh, United Kingdom
m.bareford@epcc.ed.ac.uk

Nick Johnson
EPCC
University of Edinburgh
Edinburgh, United Kingdom
n.johnson@epcc.ed.ac.uk

Michèle Weiland
EPCC
University of Edinburgh
Edinburgh, United Kingdom
m.weiland@epcc.ed.ac.uk

ABSTRACT

This paper provides an insight into the optimisation of runtime and energy performance for two widely-used CFD codes. Energy efficiency is a hot-topic in HPC and methods to reduce the energy consumption of large machines are an active area of research. In this paper, we examine the energy efficiency and runtime performance of the SBLI and Nektar++ codes, using small but real-world test cases. The codes are instrumented in sufficient detail to reveal significant variability in energy usage during the course of the simulations. In addition, the test cases are run at different CPU frequencies and the consequences of changing this parameter, for both runtime performance (time to solution) and power performance (energy to solution) are presented.

Keywords

Energy Efficiency; Power Measurement; Cray XC30; CFD

1. INTRODUCTION

This paper describes the energy and performance profiling of two computational fluid dynamics (CFD) applications and test cases, one involving the flow of air around a wing, the other simulating the flow of blood through an aorta. The motivation is to establish the relationship between energy efficiency (i.e. energy to solution) and performance (i.e. runtime to solution). A naive assumption is that as runtime reduces, so does the energy to solution. For a perfect case, such as one where the number of iterations in a simulation is reduced, this may be valid. However, if the runtime decrease is due to improvements in algorithm implementation by the programmer, better vectorisation by the compiler or frequency scaling by the processor, then this is unlikely to hold. For example, doubling the processor frequency may result in more than double the power consumption; a simulation may as a result speed up to take half the original time to completion, but it may consume more energy than before the increase in CPU frequency.

One of the core Exascale challenges is understanding how to develop parallel software that minimises performance costs and power usage despite an increasingly complex hardware landscape [1]. An essential part of this research effort is the ability to instrument code so that we can accurately measure performance data during runtime. Such measurements can be taken directly by external sensors, a so called hardware-based approach, which has been followed by several researchers [19][17][4]. However, this strategy suffers from the restrictions associated with expensive bespoke hardware that can be complicated to setup and maintain. Alternatively, some HPC vendors offer software-based support for obtaining energy measurements; these typically take the form of simple APIs that allow access to performance counters (PMC) implemented at the hardware-component level [2][10][5], otherwise known as a bottom-up approach, which is the one followed by this paper. The top-down variant uses various system events (e.g., cache misses) as proxies for power measurement [6]; although this technique can effectively be applied to any platform there is a loss of accuracy and resolution: i.e., it is harder to determine which parts of the software/hardware are using the most energy.

The remainder of this paper is structured as follows. In Section 2, we introduce the two CFD applications, the associated measurement libraries and the real-world test-cases used in the paper. In Section 3, we present the results gained from performing the measurements on the ARCHER system. Finally, in Section 4 we present some concluding remarks on the implications of our findings and point towards further work which would lead to a fuller understanding of the subject.

2. APPLICATIONS & TEST CASES

Whilst it is possible to use synthetic benchmarks, in this paper we choose to make use of realistic test cases to measure real-world performance. We do this for two codes, SBLI [15] from The University of Southampton, and Nektar++ [7]; the main development of this code is co-ordinated by Imperial College London. The first application, SBLI, which stands for Shockwave Boundary Layer Interaction, is a Direct Numerical Simulation code, designed for large eddy simulations of transitional and turbulent flow. SBLI is a high-order fully parallellised finite difference code that solves the full 3D compressible Navier-Stokes equations; it has often been used to model turbulent flows for a range of Mach numbers...
The second code, Nektar++ [7], is an open-source spectral element code that combines the accuracy of spectral methods with the geometric flexibility of finite elements, specifically, hp-version FEM. This code supports several scalable solvers for many sets of partial differential equations, from (in)compressible Navier-Stokes to the bidomain model of cardiac electrophysiology [12][8].

The SBLI test case is an air-flow simulation that was run using a customised version of SBLI, based on v4.2.0, which featured a particular treatment for a trailing airfoil edge as specified by the NACA4412 simulation [14] (a 3-block airfoil test case). The code was instrumented such that the energy usage and power consumption could be measured at various points within each time step. Several calls to the monitoring library, PATMPI [3] were placed inside the main application loop (see the SBLI source file, main_3d.f). The following code fragment shows the straightforward and lightweight nature of the PATMPI interface.

```fortran
CALL pat_mpi_open (out_fn)
DO i = 1, nstep
    CALL pat_mpi_monitor (i, 1)
    ! application code...
    CALL pat_mpi_monitor (i, 2)
ENDDO
CALL pat_mpi_close()
```

Together, the three library calls (open, monitor and close) ensure that the minimum number of MPI processes are tasked with reading the hardware performance counters, and one of these monitoring processes (e.g., rank 0) is also tasked with the collation and output of the performance data. The counter values themselves are accessed via the CrayPAT API [11]. This approach means that controlling which counters are monitored does not require programmatic changes, either to the monitoring library or to the code under investigation. Instead, the monitored counters are specified by setting environment variables in the submission script.

```bash
#!/bin/bash
#PBS -N naca4412
#PBS -l select=4
... export PATRTSUMMARY = 1
export MYRTCTRCAT = PAT_CTRS_PM
export PATRTPERFCTR = PMPOWER:NODE, \
  PMENERGY:NODE
... aprun -b -n 96 ./dns3d.x+pat < Input.in
```

For example, in the script fragment above, two counters from the power management category [PAT_CTRS_PM] will be read. Each ARCHER node has one set of PM counters, which means that only one of the twenty four MPI processes that run on a fully-populated node is required to read the counter values. The two counters listed, PMPOWER:NODE and PMENERGY:NODE, record the instantaneous power (W) and cumulative energy (J) respectively — these measurements cover CPU, memory and any other hardware contained on the processor daughter card; any consumption due to the network controllers and beyond is excluded however. It is possible to specify other counters, assuming that the corresponding counter category is also specified in the submission script: e.g., to capture the DRAM_ENERGY, the PAT_CTRS_RAPL category must also be listed.

The Nektar++ test case is a simulation of the blood flow through an aorta using the unsteady diffusion equations with a continuous Galerkin projection — the advection type was set to weak discontinuous Galerkin. Here, a branch of the Nektar++ code (also based, coincidentally, on v4.2.0) was similarly instrumented such that energy and power measurements could be taken. For this code, it was necessary to convert the PATMPI monitoring library to a new C++ class, called PerformanceAnalyser, which was added to the LibUtilities/AnalysisUtils namespace, thereby integrating the CFD code with the CrayPAT API. Calls to the methods of the PerformanceAnalyser class were placed inside the test case, defined in the v_DoSolve method of the UnsteadySystem class, see the SolverUtils/UnsteadySystem.cpp file within the NEKTAR++ source code repository.

All Cray XC30 counters have a time resolution of 0.1 s [11], which falls below the runtimes per simulation time step recorded for the SBLI (0.2–0.3 s) and Nektar++ (0.8–0.9 s) test cases. Please note, all measurements were restricted to the main application loop, in order to prevent the performance data from being influenced by the costs associated with any setup procedures required before starting the simulation proper.

3. RESULTS

We present results obtained by running the two test-cases on the UK’s national HPC service ARCHER [9], a Cray XC30 platform hosted by EPCC. Each test-case can only be run with one application at this stage, a pointer for further work would be to run the NACA4412 case with Nektar++.

3.1 SBLI

The NACA4412 test case was run ten times across four compute nodes (comprising ninety six processors in total) on ARCHER. Each run comprised one thousand time steps and took approximately fifteen minutes to execute. For this work, the SBLI code was compiled with the CrayPAT API. Calls to the methods of the PerformanceAnalyser class were placed inside the test case, defined in the v_DoSolve method of the UnsteadySystem class, see the SolverUtils/UnsteadySystem.cpp file within the NEKTAR++ source code repository. This data was then further processed in order to calculate quantities on a per time step basis, such as average power drawn per node and total energy consumed.

From Figure 1 it can be seen that in general, the point-in-time power histograms are asymmetric about the maximum. Indeed, the minimum readings (262–272 W) are much lower than the peak value. These low readings are recorded during the first time step of the simulation; from then on, there is a noticeable increase in the power readings over the first 2–3 minutes, as can be seen in Figure 3 which may explain the asymmetry of the histograms. Please note, the normalised count used to plot the histograms is a simple re-scaling that sets the total of the histogram bin counts to one.
Figure 1: Histograms of average point-in-time power per node from the second, sixth and eighth simulation runs.

Figure 2: The normal distributions inferred from the histograms generated by all ten runs.

Figure 3: The point-in-time power readings for the third and eighth simulation runs.

The asymmetry can, in part, be explained by the differential nature of the workload. During the early part of a
simulation run, some parts of the setup may not exercise the processor as much as the main computational part. Hence, when control enters the main time step loop, the rise in workload prompts the power management to begin increasing the CPU frequency, this technique is otherwise known as dynamic voltage and frequency scaling (DVFS), which also raises the power consumption.

The eighth run appears to be somewhat unusual in that there are three bands of power readings as shown in Figure 3. It is the two lower bands (starting between 290 − 310 W) that cause the corresponding histogram to depart further from a normal distribution, see the green line of Figure 2.

Figure 4: Histograms of energy used per time step for the first and fourth simulation runs.

We now move to consider not the point-in-time power but the energy to solution. Over the ten runs, the mean energy usage was 1.128 ± 0.011 MJ over 876 ± 7.2 s. The variation in energy used per time step can be seen in Figure 4 and is rather noticeable, with each histogram featuring multiple peaks. One may have expected the variation to be small and further work is required to understand the variation. The plots of energy used per time step as seen in Figure 5 show a corresponding banded pattern.

The NACA4412 test simulation was run ten times for each of these settings. Figure 6 plots the mean energy usage and runtime for each set of runs; the standard deviations are indicated by the horizontal and vertical bars centred on each plot point. The green point denotes the CPU frequency associated with the lowest energy consumption. Moving from 1.2 to ≤ 3.5 GHz gives a reduction in runtime of roughly 28%; however four fifths of this speed-up has already been achieved by the time the frequency has risen to 1.5 GHz. If a CPU frequency is not specified as part of the job submission, the job will run so as to minimise execution time, i.e., by default, the highest CPU frequency is selected. Hence, selecting the green frequency of 1.6 GHz would give an energy

![Figure 5: The energy used per time step for the first and fourth simulation runs.](image)

On ARCHER as with many systems, it is possible to specify as part of a job submission, a CPU frequency that is then applied to all the CPUs owned by the assigned compute nodes. On this system, the allowed frequencies range from 1.2 to 2.5 GHz in steps of 100 MHz; there are also two higher frequencies, 2.7 GHz and a special turbo frequency labelled 2.701, where, for IvyBridge CPUs, this boost-able frequency can reach as high as 3.5 GHz. The exact frequency in this boost state is controlled by the hardware and based on thermal and other limiting properties.
reduction of \( \approx 34\% \).

Figure 6: The relationship between total energy usage, runtime and CPU frequency.

3.2 NEKTAR++

Like the above NACA4412 test-case, the aorta blood test was also run ten times over four compute nodes on ARCHER. Each run involved four thousand time steps and took approximately sixteen minutes to execute. The NEKTAR++ code was compiled with the GNU Programming Environment (gcc v4.9.2). Again, all readings are taken on a per node basis and then either averaged (power) or summed (energy) to provide an overall measurement.

Figure 7: Histogram of average point-in-time power per node from the first simulation run.

Figures 7 and 8 show the point-in-time power histograms for selected runs of this test-case, which are asymmetric about the maximum with more readings below the peak value than above, similar to the previous test-case. Figure 9 shows the resulting ensemble of histograms giving an indication of the variability between runs.

Figure 8: Histograms of point-in-time power measurements form the sixth and ninth simulation runs.

Figure 9: The normal distributions inferred from the histograms generated by all ten runs.

The minimum power readings of \( \approx 262 - 272 \text{W} \) are taken
from the first time step of the simulations. There is a noticeable increase in the power readings over the first 2-3 minutes, as can be seen in Figure 10. Hence, we surmise again that DVFS is the primary cause.

Figure 10: The point-in-time power readings for the first and third simulation runs.

As with the SBLI test-case, we now move to consider the energy consumption. Over the ten runs, the average energy usage was 1.061 ± 0.013 MJ over 838 ± 11.8 s. The variation in energy used per time step (Figure 11) is similar to that seen for the SBLI code, although for the NEKTAR++ cases, there appears to be a single high-energy time step (∼1.6 kJ) at the start of the simulation; the energy usage of subsequent iterations falls in the 200 – 400 J range. Figure 12 shows a view of the same runs showing the energy banding effect. This is interesting as it suggests that each time step consumes one of a few discrete amounts of energy.

Figure 11: Histograms of energy used per time step for the second and tenth simulation runs.

Following Figure 6, the aorta test simulation was run ten times for each possible CPU frequency. From Figure 13, it can be seen that changing CPU frequency from 1.2 to ≤3.5 GHz can reduce the runtime by 40%. However, four fifths of this speed-up has already been achieved by increasing the frequency to 2.0 GHz. The most energy efficient option is setting the frequency to 1.9 GHz which gives an energy reduction of ≈27% compared to leaving the CPU setting unspecified and allowing the system to vary the frequency dependant on load and temperature. Therefore, it is clear there is a trade-off between running at maximum frequency for the shortest possible runtime, which may be important to achieve maximum scientific throughput in a given temporal window, and aiming to minimise energy usage by slowing the CPU frequency, at a cost of a reduced throughput.
4. CONCLUSIONS

In this paper, we have presented two real-world test cases and analysed their performance with respect to time to solution (runtime performance) and energy to solution (energy efficiency). In addition, we have considered the average point-in-time power per node drawn during each run. Our results show that both ensembles of simulation runs are consistent as regards time to solution (runtime) and energy to solution (energy usage). Individual runs however can exhibit considerable variation in energy and power, such a complication is hidden when performance is measured as a single figure of energy to solution. The primary cause of this variation is likely to be due to the combined fluctuations of the CPU frequencies: each job ran across ninety-six cores.

When considering optimisation of a test-case, we have shown that changing the processor frequency effects a trade-off between time to and energy to solution. A balance can be achieved whereby a small increase in runtime can reduce the energy consumption. By profiling a representative test-case a user could establish a graph similar to those shown which would allow them to select between maximum throughput and minimum energy.

The variations between the power and energy histograms for simulation runs involving the same code (see Figures 2 and 9), suggest a corresponding variation in compute node performance and network load. Indeed, none of the SBLI test runs were assigned nodes from the same blade (the Cray XC30 hardware is arranged such that there are 4 nodes in a blade, 16 blades in a chassis and 3 chassis in a cabinet), which means performance is likely to have been affected by the activities of nodes assigned to other applications. Each of the CPU frequency plots (Figures 6 and 13) were generated from data produced by 160 runs; the intention here, being to average out the noise from the ARCHER network. Furthermore, each of those tests ran at a specified CPU frequency: by default, the boost setting is enabled which automatically introduces variability ($2.7 < f_{CPU} \leq 3.5$ GHz) into the results. Therefore, the fact that the SBLI and Nektar++ codes achieved their lowest energy consumption for different CPU frequencies implies that algorithmic choices in code design can influence the minimum in the CPU frequency curve.

Further work is required to gain a deeper understanding into the variations seen in this paper, specifically to understand if they are caused by algorithmic or system features, which will require control over CPU frequency and possibly node assignment too. Additionally, study of further test-cases may show if the energy-runtime curves are characteristic of the application or of the problem being solved.

Figure 12: The variation in energy usage over time for the second and tenth simulation runs.

Figure 13: The relationship between total energy usage, runtime and CPU frequency.
5. REFERENCES


