A Benchmark-Driven Modelling Approach for Evaluating Deployment Choices on a Multicore Architecture

A. Osprey[1],[2], G. D. Riley[3], M. Manjunathaiah[4], and B. N. Lawrence[1],[2]
[1]Department of Meteorology, University of Reading, UK
[2]National Centre for Atmospheric Science (NCAS), Natural Environment Research Council, UK
[3]School of Computer Science, University of Manchester, UK
[4]School of Systems Engineering, University of Reading, UK

Abstract— *The complexity of current and emerging high performance architectures provides users with options about how best to use the available resources, but makes predicting performance challenging. In this work a benchmark-driven performance modelling approach is outlined that is appropriate for modern multicore architectures. The approach is demonstrated by constructing a model of a simple shallow water code on a Cray XE6 system, from application-specific benchmarks that illustrate precisely how architectural characteristics impact performance. The model is found to recreate observed scaling behaviour up to 16K cores, and used to predict optimal rank-core affinity strategies, exemplifying the type of problem such a model can be used for.*

Keywords: Performance modelling, shallow water model, Cray XE6, multicore

1. Introduction

Climate modelling is one of the grand challenge problems of current times. To gain a greater understanding of the climate system, scientists are adding complexity to their models through increased resolution, modelling of additional physical processes, and increasing numbers of ensemble experiments to quantify uncertainty, all requiring vast computational resources [1]. Thus climate modelling is one of the application areas making use of the top high performance systems across the globe, and to make best use of these expensive resources means adapting to the new architectures that are emerging.

Current supercomputing trends are for multicore processors and the use of co-processor accelerators, both of which have lead to increased levels of heterogeneity within high performance systems. Apart from the different types of processor that might coexist, for example CPUs and GPUs, modern multicore architectures often have several hierarchies inherent within the system. For the processor this may include cache-sharing, non-uniform memory access times and floating point unit sharing as with the AMD Bulldozer [2], and for the network this may be due to full connectivity between groups of nodes and then point to point connections between the groups, as in the IBM Power 7 Host Fabric Interface (HFI). This means that the user has many choices about how best to use the system in order to maximise floating point throughput and minimise data transfer costs. For example, it has been found that under-populating nodes, altering the domain decomposition or changing the rank-core affinity can all alter performance [3], [4], [5].

Currently, finding the best way to run an application on a given architecture, especially for highly complex scientific code such as the UK Met Office climate model, is often a lengthy process of trial and error [4]. Recent work has been done with automated optimisation techniques, such as simulated annealing, to guide the search for optimal performance tuning parameter values [6].

In this paper, we introduce a benchmark-driven predictive modelling approach that allows the rapid evaluation of different deployment choices, without the execution of costly full-model runs. The model is based on commonly used analytical modelling techniques, but driven solely by data from a series of application-specific benchmarks designed to capture the effects of various resource sharing scenarios. The use of benchmark data over analytical models of the architecture leads to a shorter model development cycle. Since both the application and architecture are complex, building accurate performance models may be time-consuming, and in many cases, unnecessary. Often it is sufficient to know that one choice performs better than another. The method is outlined in Section 3.

The approach is demonstrated on a complex modern architecture, the Cray XE6, that exhibits many of the heterogeneous features mentioned. A simple shallow water code is chosen as it displays one of the main patterns of behaviour seen in atmosphere and ocean models: the calculation of regular grids of data using a finite difference scheme requiring periodic boundary exchanges. The code uses an MPI parallelisation and straightforward rectangular 2-dimensional domain decomposition. The simplicity of the application means that effort can be focused on understanding the complexity of the architecture. The methodology could, however, be extended to explore application issues, such as parallelisation strategies and communication patterns.

The resulting model is described in Section 4 and used to predict run times for a series of scaling experiments. These predictions are then evaluated against measured results to quantify the model accuracy. The usefulness of the model is tested by exploring rank-core affinity strategies in Section 5. It is known that assigning approximately square domains to each rank minimises costly off-node communications [7], and this is reproduced by the model and confirmed by measured results.

2. Performance modelling

A performance model combines information about an application and underlying architecture to make an estimation of the expected wallclock run time. The application part of the model describes the amount and type of work that needs to be performed, at any granularity from subroutines to low-level operations, and the machine part supplies the times to complete each portion of work, which may be from benchmark measurements or a mathematical representation of features of the hardware. Typically for a scientific application, the work to be performed can be split into two broad categories: i) computations, the bulk of which will be loops of floating point calculations, and ii) communications, the MPI operations that transfer data between ranks. This requires machine models of the processor and network respectively.

There are many different means of building performance models [8]. Logically the process begins with the creation of the application model, which can be done automatically by a tool, or written by hand with expert knowledge and analysis of the code, each of which are discussed in the following sections.

2.1 Automated methods

Application models can be generated automatically by profiling tools which log the operations a program performs as it executes. The resulting trace file can then be replayed by a machine simulator which has the ability to emulate a different architecture to the one on which the application was originally run. An example of this is the PMaC prediction framework described by Snavely et al [9] and Carrington, Snavely and Walter [10]. The process cannot however predict changes to the number of processors or other application inputs. A similar but more flexible method (the WARPP toolkit) is described by Hammond et al [11].

Although automated tools require minimal human effort and limited knowledge of the code, fundamentally the code must exist and be executable. Often it is treated as a black box, therefore such methods are limited in their ability to explore deployment choices, as this involves parameterisation in terms of application inputs such as problem size.

2.2 Analytical methods

Analytical application models are generally constructed by hand, based on an abstract view of the program code. Typically for scientific applications, the processor and network are modelled with a mixture of analytical and empirical techniques. Performance models have been developed in this way for many applications including, in the climate science domain, the POP ocean code (Kerbyson and Jones [12]), HYCOM ocean code (Barker and Kerbyson [13]) and WRF weather prediction code (Kerbyson et al [14]). A systematic description is given by Hoefler et al [15]. First, the application input parameters, code kernels, communication patterns, and any overlap with computations are identified. These are used to compose the analytical application model. Next, empirical steps are taken to provide run times for the model. For computations, an expression of the sequential time for each of the kernels is derived by measuring the code for different problem sizes. Communication times are expressed as a LogGP model of point-to-point messages [16], with the parameters derived from benchmark experiments [17].

One of the main criticisms of detailed analytical application models is the human effort required to build them. Ultimately, the purpose of the modelling procedure should be to improve performance by aiding understanding, highlighting areas for optimisation, informing the tuning process and so forth. For these purposes a fully accurate model may not be required, and so automatic tool-based methods or simple coarse analytical models may be preferred. An example of this is the work of Dennis, Jessup and Waite [18] who use a prototyping tool (SLAMM) to compare the memory usage of several algorithmic implementations within POP. Although far more simplistic than the Kerbyson and Jones model [12], theirs is sufficient to substantially optimise the code.

3. Benchmark-driven modelling approach

Here we propose an updated analytical and empirical modelling approach that is suited to a complex and heterogeneous architecture such as the Cray XE6. There are three main differences to the methodology described by Hoefler et al [15].

Firstly, for modern architectures simple models of data transfers can become highly complicated. The number of parameters involved can quickly rise with the different communication protocols in use (large and small messages), links along which data can travel (on-node and off-node) and contention due to multiple cores per node accessing the same network interface. This can be seen in the work of Mudalige, Vernon and Jarvis [19] for a Cray XT4 system which is somewhat more simplistic than the Cary XE6 considered here. Since collecting benchmark data is a necessary part of the creation of these models, we propose skipping the modelling phase and simply interpolating from the data.

Secondly, part of the time spent in MPI transfers includes accessing the data from it's location in cache or memory. Although packing the data into a buffer may be done as a separate stage, Fortran 90 compilers are able to do this automatically, even with non-contiguous subsections of data. For data stored as a 2-dimensional array, retrieving subsections in one of the directions will lead to non-contiguous accesses. In this benchmarking approach, both contiguous and non-contiguous accesses are included in the transfer time, without having to be explicitly modelled.

Thirdly, it is insufficient to provide only the sequential computation time, as the model needs to account for the effects of resource-sharing such as multiple cores accessing the same cache. Therefore here, multiple benchmarks are run to account for each of these cases.

As in Hoefler et al [15] the first steps of this approach are to express the application analytically in terms of code kernels and communication patterns. The empirical steps, however, involve an additional stage which is to identify the resource sharing scenarios to be measured. Compute benchmarks are based on an instrumented version of the application. The communications benchmarks are bespoke versions of standard tools. Data from the benchmarks are then organised in a database which is accessed by a deployment model. This translates a given runtime scenario into a performance prediction interpolating from the measured results as necessary. Assuming applications follow the shared memory program multiple data (SPMD) paradigm, all cores execute the same code but over different data domains. Here it is also assumed that all cores are synchronised, thus only the maximum time per core is needed.

4. Performance model of a shallow water code

Using the benchmark-driven approach described, a model is constructed for a version of the NCAR shallow water code [20], [21] on the HECToR supercomputer, a Cray XE6 system based at the University of Edinburgh [22]. The following sections describe the shallow code (Section 4.1) and HECToR system (Section 4.2), after which the performance model is outlined (Section 4.3) and evaluated (Section 4.4).

4.1 The NCAR shallow water code

The NCAR shallow water code (herein 'shallow') uses a second-order finite-difference solver to evaluate the shallow water equations. Calculations are performed over a rectangular domain of size M by N with periodic boundary conditions in both directions to replicate the behaviour on a sphere whilst avoiding the use of poles. The version of shallow used here is a Fortran 90 implementation with a 2-d parallel domain decomposition. Local domains are sized m by n with arrays dimensioned as $m+1$ by $n+1$ to allow for a single halo row and column. There are 13 local array fields and at each timestep the code performs 10 array update loops, 3 array copies, and 7 exchanges of halo data.

Halo exchanges update the values at boundary cells from the domains held by neighbouring ranks, and in this version these are implemented with `MPI_Sendrecv` operations. As shallow uses double-precision real numbers, the total data volume sent and received each halo update will be $(m+n+2)\times 8$ bytes. This data volume only depends on the local data size and will not vary with the global data size or total number of ranks. Thus, under ideal conditions the wallclock communication time should remain constant for the same local array size (weak scaling). In reality however, the physical mapping of ranks to cores will affect the run time and, as the total size of the communicator increases, interference and load imbalance may increase.

As well as the transfer time between cores, additional time will be spent on overheads associated with initialising the exchange and loading the data from its location in cache or memory. Since the data is stored in 2-dimensional arrays, halos sent in the M-direction will require loading $n+1$ cachelines as the data in this dimension will be held non-contiguously in memory. Conversely, the halos in the N-direction will require only $(m+1)/8$ cachelines as this data will be contiguous.

4.2 HECToR

HECToR is the national UK supercomputing facility based at the University of Edinburgh and funded by the UK research councils. It is currently in Phase 3 of its lifespan which is a Cray XE6 system, consisting of 2816 compute nodes for a total of 90,112 cores.

Each compute node on HECToR comprises two sixteen-core AMD Opteron Interlagos chips, part of the Bulldozer family. The Interlagos chips are made up of two 8-core dies, each directly connected to their own 8 GB memory and consisting of four 'compute modules'. A module contains two integer cores that share a single floating point execution unit and 2 MB of L2 cache. Additionally, integer cores have their own 16 KB L1 data cache and all cores on the die share 6 MB of L3 cache with an extra 2 MB given over to maintaining cache coherency. Caches in the AMD Opteron series are exclusive with lower levels acting as victim caches for the higher levels. Cores operate at a frequency of 2.3 GHz and are capable of processing 8 double precision floating point operations per cycle. The nature of the shared floating point unit means that codes can obtain double the cache and memory space per task by running with only one of the integer cores and still have access to the full floating point capability. Diagrams representing the hardware can be found on the HECToR website [23].

All four dies on a node are connected to each other via HyperTransport (HT) links forming a non-uniform memory access (NUMA) node meaning that all dies can access the total 32 GB of memory. The links between dies vary, with 24-bits between dies on the same chip, 16 bits between opposite dies, and 8-bits between diagonally opposite dies.

In addition, each node has a single link to a Gemini Network Interface Controller (NIC) that connects nodes into a 3-dimensional torus [24].

4.3 Performance model

From the information in Sections 4.1 and 4.2, a performance model of shallow can be constructed. The application model consists of a series of identical timesteps, each comprising i) some volume of floating point compute work dependent on the local array size, and ii) several halo-exchanges with a single row and column transferred per rank. Array copies are not considered, as they only account for around 10% of the runtime. If needed they could be benchmarked in a similar way to the compute loops. The next step is to design and run the benchmark experiments to generate data with which to populate the model. The computation and communication models are described in the following sections.

Computation model

To benchmark the computations, an instrumented version of shallow is run over a set of 23 problem sizes ranging from L1 resident to memory resident. Timers are inserted around each block of compute loops with an MPI barrier before the timer call so that all cores are synchronised. This means that times are consistent from run to run with full resource sharing. In real application runs however, it is unlikely that cores would be synchronised, leading to less resource-sharing (increasing performance), but also more waiting time in the halo exchanges (decreasing performance).

The benchmark is run over a variety of cases that illustrate each resource-sharing scenario. These are: i) core pair mode, where only one of the integer cores in each module is in use, with no floating-point unit or L2 cache sharing, and ii) compact mode, where both integer cores are used, causing floating point and L2 cache sharing. Both cases are measured on a single die, with one to four modules to account for L3 cache sharing as well. In reality, the achieved performance may be reduced slightly by communication overheads, although this may be offset by increased performance due to the lack of interruption by timer and barrier calls.

Each experiment is repeated a total of 5 times to provide a mean flop rate, with the number of flops taken from the source code. The benchmark results are shown in Figure 1, with features of the architecture clearly identifiable. The per-core performance drops when both integer cores are in use, from a peak of 3.3 GF to 2.6 GF. This does, however, increase the per-module performance to 5.2 GF, showing the benefits of two feeds to the floating point pipelines. Performance differs little with the number of modules when the problem size fits in each module's L2 cache, with only a slight degradation when all modules are in use. As the problem size increases further, the performance decreases for each additional module due to contention for L3 cache, memory bandwidth and the translation lookaside buffer (TLB).

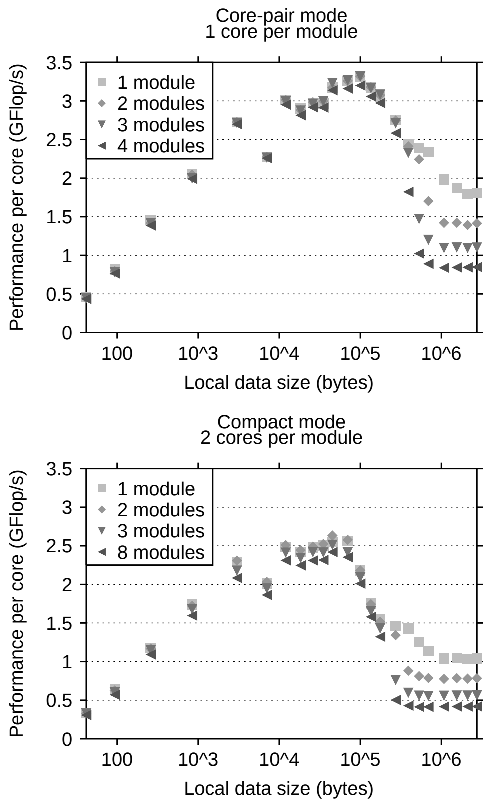

Fig. 1: Computational performance per core as measured by benchmark experiments. The local data size is based on storing 13 array fields.

Communication model

A bespoke benchmark code is used to measure the halo-exchanges. It is based on the Intel MPI Benchmarks (IMB) [25], but with greater control over the pairs of cores to transfer between, contiguous and non-contiguous memory access and location of data in cache or memory.

The benchmark takes as input a series of message lengths spaced roughly exponentially, with extra values around the boundaries between different MPI message protocols (for small and large messages). To ensure messages are accessed from the correct cache level, the benchmark takes two other inputs: the size of the cache to operate from ('the operating cache'), and the total size of all caches that are closer to the core ('the higher cache'). Message buffers are taken from an array of size roughly equal to the size of the operating cache. This is initially loaded into the processor, then pushed into the operating cache by loading a dummy array which is the same size as the higher cache. Several message transfers are performed and a mean value taken to reduce timer overhead and effects of timer granularity. To ensure that each message is actually taken from the operating cache, a different slice of the buffer is extracted each time, similar to the approach taken by the IMB. In this way both contiguous messages and non-contiguous messages are measured. It should be noted that these memory assumptions do not account for copies of data made by subroutines or held in MPI buffers, therefore data may be located further from the processor. The same assumptions are made for the application code, although the benchmark does not replicate all behaviour. Furthermore, as the benchmark performs multiple consecutive communications, any overheads associated with initialising MPI buffers will be lost, whereas in the application these costs may be more significant.

Experiments are performed over all types of connection along which data may be transferred. On the XE6, these can be categorised as between i) cores on the same module (shared L2 cache), ii) cores on the same die (shared L3 cache), iii) dies on a node (HT links), or iv) nodes in the torus (Gemini interconnect). To minimise complexity, means are taken over the different bandwidths between dies on a node and different numbers of hops between two nodes on the torus. This can be justified since, unless a very large communication volume is taking place, little difference is observed in the achieved bandwidths between dies. Furthermore, on the XE6 it is not known in advance which group of nodes the scheduler will select, thus the number of hops cannot be predicted. In most cases the scheduler selected nodes on the same or neighbouring NICs, yet frequently the nodes were as many as 13 hops away.

Along with the message lengths, cache usage, contiguous and non-contiguous accesses and connections to measure, it is also necessary to quantify the contention due to different numbers of cores communicating along the same link concurrently. Experiments are therefore run with 1 to 8 cores per die for transfers within and between dies, and from 1 to 32 cores for transfers between nodes. Selected results for off-node transfers and off-die transfers are shown in Figure 2, showing the per-transfer slow down when all cores communicate along the same link simultaneously. Non-contiguous transfers achieve approximately a factor of 8 lower bandwidth than contiguous transfers (as expected), and off-die transfers generally achieve a higher bandwidth than off-node transfers, except where only a single core per node is used.

4.4 Evaluation

The performance model of shallow combines the data from the computation and communication benchmarks to make a prediction about the total application runtime. The model was evaluated by comparing predictions to measured times for several examples. Three global problem sizes were used initially: 256×256, 512×512 and 1024×1024. Each of these was run with 4, 8, 16 and 32 cores per node on 1 to 16 nodes, up to a total of 512 cores. It should be noted that, up to 32 cores per node, only one integer core

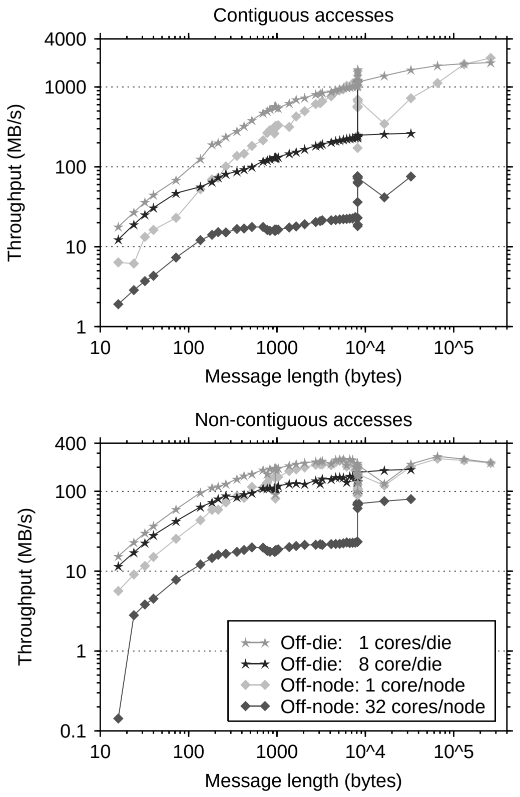

Fig. 2: Bandwidths as observed by halo-exchange benchmarks. Off-node values are means over 10 runs with pairs of nodes selected by the scheduler. Off-die values are means of two runs over each pair of dies within a node.

per module is used to make best use of the floating point units. These examples test how well the model captures the interactions within a node. It is also useful to look at larger problem sizes that scale out to thousands of cores. A second set of examples therefore takes two larger problem sizes, 2048×2048 and 4096×4096, and scales these out to 512 nodes with 16 or 32 cores per node, up to a total of 16,384 cores.

Model predictions and measured run times are shown in Figure 3. Shallow runs were performed 5 times, with the mean and two standard deviations either side shown, which assumes a normal distribution of run times. The modelled run times can be seen to accurately reproduce the measured behaviour. Model errors are defined as the difference between the predictions and mean run times for each set of problem sizes and number of cores per node. The median percentage errors, the form often quoted in the literature, range from 0.8% to 25%. Up to 20% is generally considered reasonable.

5. Rank-core mapping strategies

The usefulness of the model is further tested by predicting performance under various MPI rank to physical

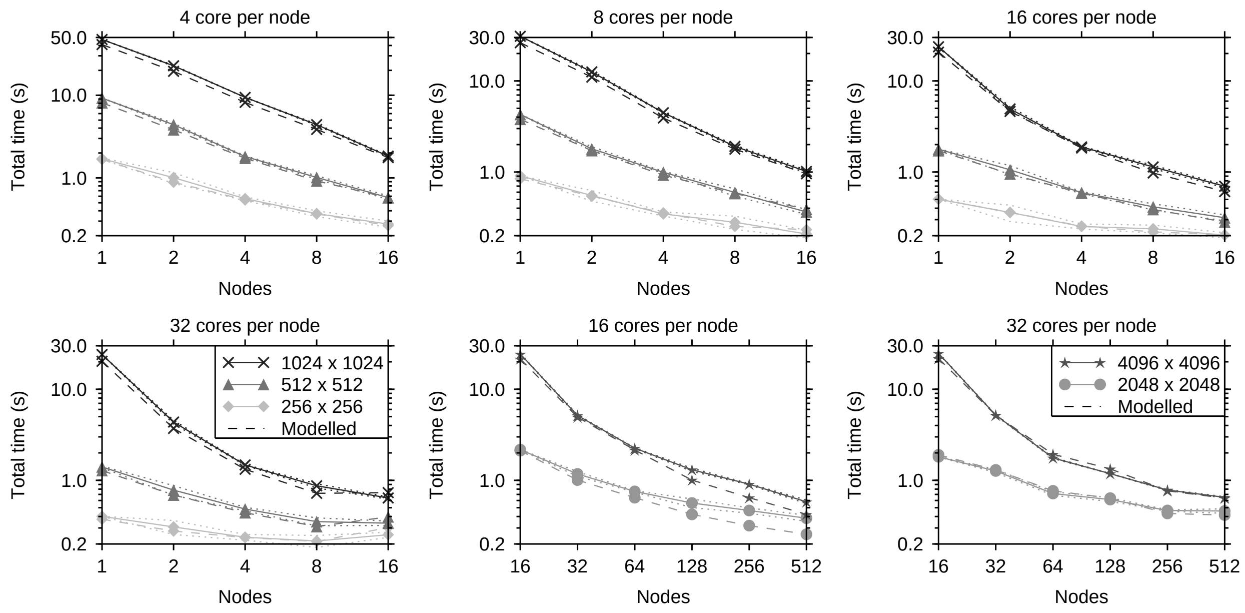

Fig. 3: Shallow model predictions of run time (dashed lines) versus measured times for a series of 5 runs, with the mean (solid lines) and two standard deviations on either side of the mean (dotted lines).

core mapping strategies. On HECToR, the default “SMP-style” mapping means that consecutive ranks fill nodes one at a time. For shallow, this corresponds to rows (or portions of a row) of the subdomain being mapped to each node. Intuitively, for nearest-neighbour communications it is better to assign rectangular subdomains to each node in order to minimise the off-node data transfer volume. Such a “custom” mapping can be generated automatically on HECToR with the `grid_order` tool. An alternative mapping strategy, "round-robin", assigns subsequent ranks to each different node in turn. For a small number of ranks this leads to columns of the subdomain being assigned to each node, however for large numbers of ranks each neighbour will reside on a different node, maximising the off-node transfer volume. These three options should produce distinct performance behaviour, and this hypothesis was tested using the predictive model and measured runs.

To test the hypothesis, the larger problem sizes from the evaluation runs were used (2048×2048 and 4096×4096 up to 512 nodes). As the computational work per rank remains the same, only the communication model is used. Shallow runs were repeated 5 times as before to quantify the variability in run time. Figure 4 shows the model predictions and the mean measured run time and two standard deviations either side. The model successfully predicts the order of performance in all cases, although the run times themselves are reproduced with varying degrees of accuracy. The custom mapping shows least variability and the best prediction. This is likely to be since it has the smalled volume of off-node transfers which are affected by network traffic. For a real climate model application the communication dependencies are more complex than just nearest neighbour and so a more sophisticated model would be required to evaluate the optimal mapping.

6. Conclusion

In this paper we have presented a benchmark-driven performance modelling approach, based on existing work but designed specifically to quickly evaluate application performance on complex architectures. Communication and computation work are both expressed as functions of benchmarked results rather than detailed analytical models, yet predict performance well enough to replicate scaling behaviour and identify the best of three different rank-core affinity strategies. The assumptions inherent to the model are discussed, along with potential sources of error which will be analysed further in upcoming work. In addition, similar models of shallow will be constructed for the IBM Power 7 and BlueGene/Q systems which display substantially different performance behaviour.

A similar approach to that defined here could also be applied to more complex kernels of climate science applications, to aid directly in performance optimisation. In such cases, the compute kernels may be larger, reducing inaccuracies due to timer overheads, and the communication may extend to other patterns beyond halo-exchanges.

References

[1] J. Shukla, T. N. Palmer, R. Hagedorn, B. Hoskins, J. Kinter, J. Marotzke, M. Miller, and J. Slingo, “Towards a new generation of world climate research and computing facilities,” *Bulletin of the American Meteorological Society*, vol. 91, no. 10, pp. 1407–1412, 2010.

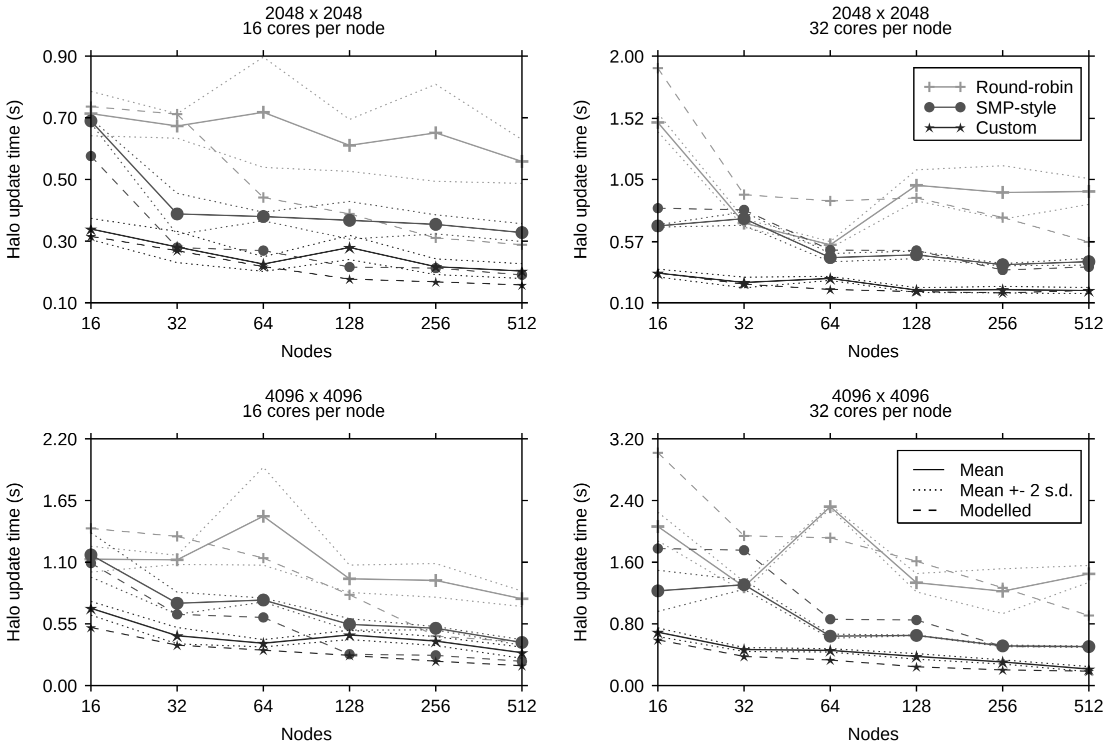

Fig. 4: Shallow model predictions of run time (dashed lines) versus measured times for a series of 3 runs, with the mean (solid lines) and two standard deviations on either side of the mean (dotted lines).

[2] D. Kanter, "AMD's Bulldozer microarchitecture," Real World Tech online article, August 2010, http://www.realworldtech.com/bulldozer/.

[3] I. Bermous, J. Henrichs, and M. Naughton, "Application performance improvement by use of partial nodes to reduce memory contention," *CAWCR Research Letters*, vol. 9, pp. 19–22, December 2012.

[4] T. Edwards, "Optimising UPSCALE on HERMIT," Cray CoE for HECToR, Report, May 2012.

[5] D. Whitaker, "Case study: AWP-ODC and MPI re-ordering," Part of presentation at the Cray XE6 performance workshop entitiled "Optimising Communication on the Cray XE6", November 2012, available: http://www.hector.ac.uk/coe/cray-xe6-workshop-2012-Nov/pdf/comms.pdf.

[6] T. Edwards, "Applying automated optimisation techniques to HPC applications," in *Geoengineering the future: Online proceedings of the 2013 Cray User Group*, May 2013.

[7] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," ser. PDP '09, 2009, pp. 427–436.

[8] S. Pllana, I. Brandic, and S. Benkner, "A survey of the state of the art in performance modeling and prediction of parallel and distributed computing systems," *International Journal of Computational Intelligence Research*, vol. 4, no. 1, January 2008.

[9] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for performance modeling and prediction," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002, pp. 1–17.

[10] L. Carrington, A. Snavely, and N. Wolter, "A performance prediction framework for scientific applications," *Future Gener. Comput. Syst.*, vol. 22, no. 3, pp. 336–346, 2006.

[11] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama, "WARPP: a toolkit for simulating high-performance parallel scientific codes," in *Simutools '09*, 2009, pp. 1–10.

[12] D. J. Kerbyson and P. W. Jones, "A performance model of the parallel ocean program," *Int. J. High Perform. Comput. Appl.*, vol. 19, no. 3, pp. 261–276, 2005.

[13] K. J. Barker and D. J. Kerbyson, "A performance model and scalability analysis of the HYCOM ocean simulation application," in *IASTED PDCS 2005*, November 2005.

[14] D. J. Kerbyson, K. J. Barker, and K. Davis, "Analysis of the weather research and forecasting (WRF) model on large-scale systems," in *ParCo 2007*, vol. 15, 2007, pp. 89–98.

[15] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *State of the Practice Reports*, ser. SC '11, 2011, pp. 6:1–6:12.

[16] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: incorporating long messages into the LogP model for parallel computation," *J. Parallel Distrib. Comput.*, vol. 44, no. 1, pp. 71–79, 1997.

[17] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm, "Netgauge: A Network Performance Measurement Framework," in *HPCC'07*, vol. 4782, Sep. 2007, pp. 659–671.

[18] J. M. Dennis, E. R. Jessup, and W. M. Waite, "SLAMM - automating memory analysis for numerical algorithms," *Electron. Notes Theor. Comput. Sci.*, vol. 253, no. 7, pp. 89–104, Sep. 2010.

[19] G. Mudalige, M. Vernon, and S. Jarvis, "A plug-and-play model for evaluating wavefront computations on parallel architectures," in *IPDPS 08*, April 2008, pp. 1–14.

[20] *NCAR HPC shallow water model tutorial*, UCAR, October 2006, http://www.cisl.ucar.edu/docs/hpc_modeling/.

[21] R. Sadourny, "The dynamics of finite-difference models of the shallow-water equations," *Journal of the Atmospheric Sciences*, vol. 32, pp. 680–689, 1975.

[22] *HECToR - UK National supercomputing service*, UoE HPCX Ltd, The University of Edinburgh, 2013, http://www.hector.ac.uk/.

[23] *Good Practice Guide: HECToR Phase 3 (32-core)*, UoE HPCX Ltd, The University of Edinburgh, 2012, http://www.hector.ac.uk/cse/documentation/Phase3/.

[24] *Gemini Network Whitepaper*, Revision 1.1 ed., Cray Inc., August 2010.

[25] *Intel MPI benchmarks: User guide and methodology description*, Document number 320714-007en ed., Intel Corporation, August 2011.