Proceedings of the

EMerging Technology (EMiT) Conference 2016

EMiT
Emerging Technology Conference
2016
MANCHESTER 1824
The University of Manchester
BSC Barcelona Supercomputing Center
Centro Nacional de Supercomputación

2-3 June 2016
Barcelona Supercomputing Center, Spain

Edited by B.D. Rogers, D. Topping, F. Mantovani, M.K. Bane

http://emit.tech

June 2016

Foreword to the 2016 Emerging Technology (EMiT) Conference

Dear Delegate,

The Emerging Tech conference, EMiT, is now firmly established. In its third year we are delighted to be hosted by Barcelona Supercomputing Center and the Mont-Blanc project. BSC has long been held in very high regard, both for its innovative HPC and also its renowned work on advancing new tools for performance analysis and programming models. The presence of the Mont-Blanc project on the EMiT Organising Committee emphasises the growing importance of emerging technologies & techniques to improve the cost and energy efficiency of next generation HPC platforms.

This third EMiT conference follows the philosophy of those preceding by seeking out the challenges of hardware, software, tools and algorithms that we are expecting from over the horizon or are helping create ourselves.

The Organising Committee has overcome many challenges to bring EMiT2016 to fruition. Please join me in extending your appreciation to each of them.

I would also like to thank each keynote, everybody submitting papers (accepted or not), and each sponsor and stall, for showing support to the EMiT series.

We are looking towards EMiT 2017. If you are inspired by this year's conference to host or join the Organising Committee next year then please speak to us.

[illegible]

Michael Bane
Chair, EMiT2016 Organising Committee
http://highendcompute.co.uk
@highendcompute

Committee

Organising Committee

Dr Michael K. Bane	High End Compute
Dr Filippo Mantovani	Barcelona Supercomputing Center
Renata Gimenez	Barcelona Supercomputing Center
Dr David Topping	University of Manchester, School of Earth, Atmospheric & Environmental Sciences
Dr Stephen Longshaw	Science & Technology Facilities Council, STFC
Dr Benedict D. Rogers	University of Manchester, School of Mechanical, Aerospace & Civil Engineering (MACE)
Irfan Alibay	University of Manchester, School of Pharmacy
Ignas Daugalas	University of Manchester, IT Services
Shih-Chen Chao	University of Manchester, IT Services
Prof. Dave Emerson	Science & Technology Facilities Council, STFC

Printing

Copyright © Published by The Emerging Technology (EMiT) Conference
Manchester, U.K.
ISBN 978-0-9933426-3-9

CONTENTS

The Road to Exascale: It’s about the Journey not the Flops

William Sawyer
Exa2Green,
Swiss National Supercomputing Centre,
Switzerland

***Abstract*— One look at the www.top500.org proves it: The exponential growth in supercomputer power has now gone on for decades. Every time physical realities — for example the limits in processor frequency — impinge on this trend, human ingenuity finds workarounds, for example, the evolution of multi-core technology. The hounds have long been barking in the distance, heralding the demise of Moore’s Law and reminding us that memory bandwidth is not keeping pace with processing power. And now energy consumption is a key bottleneck: critics correctly point out that a computing center cannot have its own dedicated nuclear power plant. Certainly the HPC advances cannot go on forever.**

There is good reason to believe that the trend will continue to Exascale and beyond. Human ingenuity still has the upper hand: new stacked memory offers a new increase in bandwidth; multi-core has given way to many-core, and new processors are effectively reducing energy consumption per Flop. In this talk we present some milestones at the Swiss National Computing Centre. Since the installation of our flagship platform, Piz Daint, we have experimented with new numerical algorithms, developed domain-specific languages, and used new compiler features to refactor codes for its constituent Graphics Processing Units (GPUs) to attain dramatic improvements in time-to-solution and energy-to-solution. We draw the conclusion that HPC is alive and well on its way to Exascale and beyond.

Parallel finite element analysis using the Intel Xeon Phi

L. Margetts, J. D. Arregui-Mena
School of Mechanical, Aerospace and Civil Engineering
University of Manchester
Manchester, M13 9PL, UK
lee.margetts@manchester.ac.uk

W. T. Hewitt, L. Mason
The Hartree Centre
STFC Daresbury Laboratory
Daresbury, Cheshire, WA4 4AD, UK

***Abstract*—This paper describes the porting of the open source engineering software ParaFEM to the Intel Xeon Phi processor. The results of a preliminary performance study are presented for a new open source ParaFEM mini-app written especially for the purpose. The main findings of the study are that: (i) The original MPI-based software scales linearly on up to 56 of the 60 available cores; (ii) A new mixed mode MPI/OpenMP implementation boosts performance by a factor of 4 when using 4 threads per core on 1-16 cores (for the largest problem that fits in the Xeon Phi memory) and (iii) The best Xeon Phi solution time is ~2 times faster than the host; here comprising 2 x 12 core "standard" Intel Xeon processors. It appears that scaling beyond 16 cores and 4 threads per core is limited by the amount of work available to each of the 240 threads. The authors propose a number of strategies that can be explored to reduce the memory footprint of individual finite elements so that a much larger problem can be tackled. With more "parallelizable work" per thread, we expect to be able to further improve the performance of the mixed mode MPI/OpenMP implementation. This work will be of interest to researchers and engineers who may wish to evaluate the Intel Xeon Phi for scientific computing, particularly those using the finite element method.**

Keywords—Xeon Phi; finite element analysis; element by element; engineering simulation; mixed MPI/OpenMP; mini-app

I. INTRODUCTION

ParaFEM is a parallel library for general purpose finite element analysis that has been written mainly in modern Fortran and uses MPI for message passing. It is supplied with a set of driver programs or "mini-apps" for solving different types of engineering problem. The ParaFEM software comprises a set of libraries for general purpose finite element analysis [1], a library that uses MPI [2] for parallel processing and a set of driver programs or mini-apps for specific types of engineering problem. The mini-apps are concise parallel programs of 2-4 pages in length. The philosophy behind writing mini-apps rather than a monolithic program is that it makes it easier for scientists and engineers to modify the programs for their own use. It also enables parallel computing experts to quickly evaluate strategies for improving performance. The source code for the software is fully documented, both from the point of view of the algorithms used and the meaning of each variable name, in the popular text book "Programming the Finite Element Method" [3].

ParaFEM has been used on various HPC systems including those at the Hartree Centre; the UK's national HPC facilities ARCHER and N8 HPC; as well as systems belonging to the Partnership for Advanced Computing in Europe (PRACE).

Recently published work using ParaFEM involves a number of different scientific application areas including: the characterization of materials for fusion reactors [4]; assessing the structural integrity of nuclear power plants [5]; developing a multiscale modelling platform for fracture using a coupled cellular automata finite element strategy [6] and understanding microstructural deformation in bone [7].

ParaFEM uses a matrix free or element by element method [8]. This works very well in parallelising each stage of the finite element process: general housekeeping (building tables that relate nodes and elements to equations); generating the stiffness matrix for each element; solving the equations (using an element by element form of the Krylov solvers) and computing derived quantities such as stress or strain from the solution vector. No global matrix is ever created and this approach is widely known to be more memory efficient than global matrix assembly and factorization. The implementation therefore seems to suit the small memory footprint (per core) of the Intel Xeon Phi.

For large finite element problems that involve many load steps or time increments, more than 90% of the time is spent in the solver. Drilling down, time in the solver is dominated by large loop count element by element "do loops" of small matrix-vector multiplications. Each matrix represents a single finite element and each vector relates to the element part of the global solution vector. Getting good performance on the Xeon Phi (and other types of hardware) therefore requires optimising this part of the computation.

II. PORTING AND OPTIMISATION

A. Hardware

This research was carried out using the iDataPlex system hosted at The Hartree Centre in the UK. The system comprises 84 nodes, each with 2 x 12 core Intel Xeon processors (Ivy Bridge E5-2697v2 2.7GHz). 42 of the nodes have an Intel Xeon Phi 5110P accelerator. Each Xeon Phi has 60 cores running at 1.052GHz and can support 4x threads per core.

B. *Software*

A new mini-app was written to help evaluate the performance of ParaFEM on the Xeon Phi. This was given the name *xx16* and can be found in the ParaFEM repository on the Sourceforge platform [1]. *xx16* contains instructions to evaluate a range of implementation strategies as well as built-in timers and counters. The program outputs a high level report on the time spent in each major section of the program as well as an estimate the number of floating point operations per second. This research also made use of the Intel Vtune Amplifier XE performance monitoring software.

C. *Porting*

The ParaFEM software is built using a top level makefile that reads compiler specific flags from a machine specific include file. A new include file was written for the Intel compiler on the iDataplex system. Compared with using general purpose graphics processors, compiling for the Xeon Phi seems trivial. All that is needed to differentiate between compiling for a standard Xeon chip and the Xeon Phi is the addition of the compiler switch *–mmic* for the latter. The Xeon Phi runs its own installation of Linux locally, so an executable can be built to run on the card as easily as compiling for any other Linux system.

The Xeon Phi can be used in three different ways: (i) as a traditional accelerator, offloading computationally intensive instructions from the host to the Xeon Phi; (ii) as a standalone processor, running the executable entirely on the Xeon Phi or (iii) in mixed mode, as part of a heterogeneous system whereby the domain of a problem is subdivided over all available cores (in both the Xeon host and the Xeon Phi card). In this paper, the executable is run entirely on the Xeon host and entirely on the Xeon Phi card, enabling a direct comparison of the use of 2 x 12 Xeon processors and a single 60 core Xeon Phi card.

A number of different compiler options were evaluated and the best performance on the Xeon Phi was obtained using the following flags: *-O3* *-align array64byte* and *-opt-streaming-stores*, the latter specifically to improve performance in memory bandwidth dominated programs. The sequential version of the Intel maths kernel library (MKL) was used for the matrix vector multiplication. The parallel version of MKL gave no benefit here because of the small size of the arrays.

D. *Mixed Mode MPI/OpenMP*

Previously unpublished work (carried out by MSc/PhD students at more than one research institution) has shown that the element by element preconditioned conjugate gradient (PCG) solver available in the ParaFEM library does not benefit from mixed mode MPI/OpenMP on standard x86 processors. The documentation available for the Xeon Phi [9] is very persuasive regarding how the hardware has been specially designed for threading on each physical core. So, despite a poor track record for mixed mode MPI/OpenMP, we were encouraged to consider this strategy again. OpenMP directives were inserted into the elements loop for testing.

The elements loop is parallelized using MPI, with each MPI process (or physical core) operating on its own local set of finite elements. The mixed mode MPI/OpenMP code subdivides these local loops further, allowing us to test performance on up to 60 Xeon Phi cores with up to 4 threads per core, 240 way parallelism.

III. Preliminary Results

A number of different test runs have been carried out, but here we focus on the results of one specific finite element problem that best illustrates our preliminary findings. The analysis involves applying a vertical load to a patch on the surface of a cubic elastic domain; an engineering test problem described in the text book for use with program p121, a mini-app for the stress analysis of an elastic material [3]. In ParaFEM, there is a command line program called *p12meshgen* that can be used to quickly generate input decks of different sizes for this problem.

It should be noted that this section reports on the performance of a distributed do loop in the PCG solver that operates on a local set of finite elements assigned to each core. The results are representative of any general finite element analysis that uses the PCG solver, both from the point of view of the geometry of the model and the physics of the problem.

Every finite element stiffness matrix in the mesh is assumed to be unique, so each one has to be stored in memory. The “do loop” involves multiplying each stiffness matrix (here 60 by 60 double precision floating point numbers) by a vector. Relevant problems include those involving material or geometric nonlinearity (plasticity [10] and large deformations [7]); transient heat flow [4]; thermo-mechanical stress analysis [5] and forced vibrations [3].

The largest test problem (for the cubic domain) that could be stored in the 8GB memory of the Xeon Phi card, using this particular version of ParaFEM, was a mesh of 42,872 twenty noded hexahedral elements (521,780 equations to be solved).

A. *Speed-up*

Fig. 1 shows speed-up for the largest test problem. The original MPI implementation (labeled 1 thread per core) scales linearly with increasing number of cores, with a slight drop off between 56 and 60 cores.

The speed up for 4 threads per core was calculated with respect to the time taken using 1 MPI core (with 1 thread). This helps highlight the advantage of the mixed mode MPI/OpenMP implementation. The mixed mode program shows a striking improvement in performance compared with the MPI only execution. On 1 to 16 cores, the use of 4 OpenMP threads per core leads to a “perfect” factor of 4 reduction in run time. Core counts greater than 16 cores do not continue this trend.

B. *Percentage Peak Performance*

The Xeon Phi used in this paper has a theoretical peak performance of 2 teraflops for single precision and 1 teraflop for double precision floating point operations. Here, the finite element mini-app has been compiled for computation using double precision. Fig. 2 shows that the (best to date) MPI implementation only achieves ~2.5% of peak performance

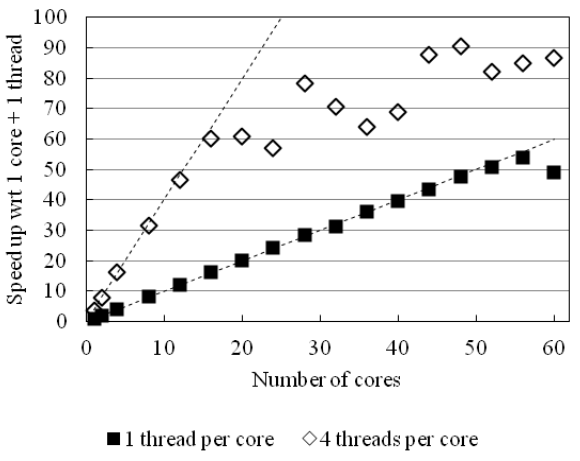


Fig. 1. Speed-up using 1 thread and 4 threads per core

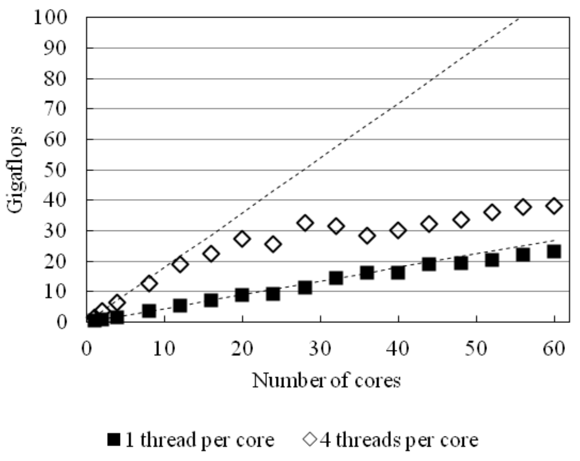


Fig. 2. Operations per second using 1 thread and 4 threads per core

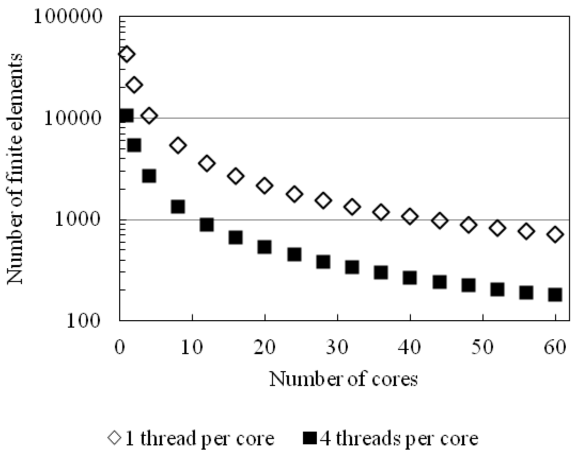


Fig. 3. Number of finite elements per thread

using 60 cores. In the mixed mode MPI/OpenMP case, there is a modest improvement to ~4% peak using the maximum number of threads and cores, i.e. a maximum of 240 processes. The dotted line in Fig. 2 shows that if the 4 threads per core data series continued scaling linearly, this part of the mini-app could be expected to achieve a more respectable ~10% peak performance.

C. Number of finite elements per thread

Fig. 3 shows the number of finite elements that each thread processes during the analysis. For the MPI only implementation (1 thread per core), performance scales linearly on up to 56 cores. This corresponds to ~765 elements per core. In the mixed mode MPI/OpenMP case, performance drops off markedly after 16 cores and 4 threads per core (64 threads), ~670 elements per thread. The number of elements reduces to ~178 elements per thread when the maximum 240 threads are used.

D. Comparison with the Xeon host

The best performing Xeon Phi implementation reported here runs around twice as fast as the 2 x 12 core Xeon processors on the host. At the moment, care needs to be taken in comparing the two platforms. There are further performance optimisations that can be implemented and tested on both.

IV. Discussion

One of the key features of the finite element method is that it can be used to predict a range of physical processes occurring in domains of arbitrary geometry. In 3D, individual hexahedral or tetrahedral elements do not need to be perfect. They can be distorted so that a mesh can fill any space. As long as the elements have good aspect ratios, then the finite element method will give reasonable results.

The characteristics of a finite element, such as the shape and material properties, are captured in the element stiffness matrix. In any general problem, it is standard practice to create and store a unique element stiffness matrix for every element in the mesh. In this paper, our results indicate that, for the largest general problem that can be stored in 8GB RAM, there is not enough work to keep all the available hardware of the Xeon Phi busy.

To address this problem, it is possible to reduce the storage requirements per element in certain problem specific cases. For example, when a mesh comprises identical elements, it is only necessary to store one stiffness matrix. In that case, the loop of matrix-vector computations can be replaced by a single matrix-matrix multiply. Typically meshes are not like this, as the nuclear model in Fig. 4 shows. This model has elements with different shapes on the face. Looking down the bore, each of the face elements sits on top of a column of identical elements. In this case, only elements on the top face need to be stored.

In some engineering problems, the stiffness elements are symmetrical, so only half needs to be stored. In the extreme case, when the elements are perfect hexahedra, some of the values in the symmetric part of the matrix are repeated.

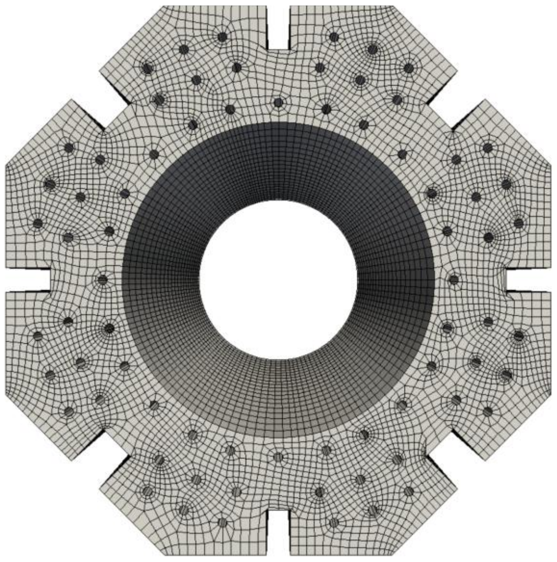


Fig. 4. Finite element mesh of a nuclear graphite brick comprising arbitrarily shaped hexahedral elements.

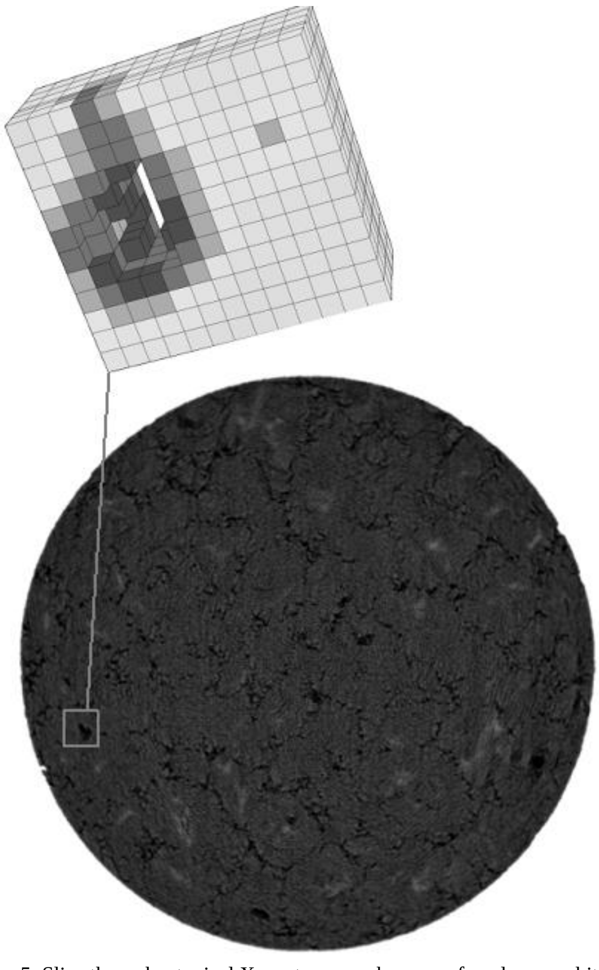


Fig. 5. Slice through a typical X-ray tomography scan of nuclear graphite and voxel mesh based on patch of X-ray tomography data.

For example, the stiffness matrix storage for 8 node hexahedral elements derived from voxel-based meshes (Fig. 5) can be reduced from 24 x 24 floating point numbers to just 5. All the cases described have been coded in program *xx16*.

One final remark is that here, we advocate looking for ways of reducing the storage required by each finite element - in order to deal with the difficulties that arise due to the small memory footprint per core (133MB) or per thread (33MB). The implication is that to achieve good performance, it may be better to have many "simple" low storage overhead finite elements in a mesh than a completely unstructured one where every element is unique. However, the story is not so simple. An area of active research in finite element analysis is the development of enriched elements to deal with special cases such as fracture; near incompressibility of the material or highly distorted elements [11]. These require an increased amount of storage and larger number of floating point operations per element. Getting the balance right, between computer performance and the accuracy of the engineering solution is therefore quite tricky!

Acknowledgment

This work was carried out in collaboration with the Intel Parallel Computing Centre hosted by The Hartree Centre, STFC Daresbury Laboratory Warrington, Cheshire, UK.

References

[1] ParaFEM website, http://parafem.org.uk. Accessed 1 May 2016.

[2] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 3.1, HLRS, 2015.

[3] I.M. Smith, D.V. Griffiths and L. Margetts, "Programming the Finite Element Method", Wiley, 2014.

[4] Ll.M. Evans, L. Margetts, V. Casalegno, L..M. Lever, J. Bushell, T. Lowe, A. Wallwork, P.G. Young, A. Lindemann, M.J.J. Schmidt and P.M. Mummery, "Transient thermal finite element analysis of CFC-Cu ITER monoblock using X-ray tomography data", Fusion Engineering and Design, 100, pp.100-111, 2015.

[5] J.D. Arregui Mena, L. Margetts, D.V. Griffiths, L.M. Lever, G.N. Hall and P.M. Mummery, "Spatial variability in the coefficient of thermal expansion induces pre-service stresses in computer models of virgin Gilsocarbon bricks", J. of Nuclear Materials, 465, pp.793-804, 2015.

[6] A. Shterenlikht and L. Margetts, "Three-dimensional cellular automata modelling of cleavage propagation across crystal boundaries in polycrystalline microstructures", Proceedings of the Royal Society A, Volume 471(2177), 2015.

[7] F. Levrero, L. Margetts, E. Sales, S. Xie, K. Manda and P. Pankaj, "Evaluating the macroscopic yield behaviour of trabecular bone using a nonlinear homogenisation approach", Journal of the Mechanical Behaviour of Biomedical Materials, 61, pp. 384-396, 2016.

[8] I.M. Smith and L. Margetts, "The convergence variability of parallel iterative solvers", Engineering Computations, 23(2), pp.154-165, 2006.

[9] J. Reinders and J. Jeffers, "High Performance Parallelism Pearls: Volume 2", Elsevier, 2015.

[10] I.M. Smith and L. Margetts, "Portable parallel processing for nonlinear problems", VII International Conference on Computational Plasticity, Barcelona, 2003.

[11] T.H. Ong, C.E. Heaney, C.K. Lee, G.R. Liu, H. Nguyen-Xuan, "On stability, convergence and accuracy of bES-FEM and bFS-FEM for nearly incompressible elasticity", Computer Methods in Applied Mechanics and Engineering, 285, pp. 315-345, 2015.

Evaluating the Maturity of OpenFOAM Simulations on GPGPU for Bio-fluid Applications

Ahmet Duran
Department of Mathematics
Istanbul Technical University (ITU)
Istanbul, Turkey
e-mail: aduran@itu.edu.tr

Senol Piskin
Department of Mechanical Engineering
Koc University
Istanbul, Turkey

Mehmet Tuncel
Department of Mathematics, ITU and
Informatics Institute, ITU
Istanbul, Turkey

Abstract
It is important to deal with the computational challenges for bio-medical fluid flow simulations and an *OpenFOAM* solver, *icoFoam*, for the large matrices coming from the simulation of blood flow in arteries on different HPC clusters. The flow problem produced various matrices as the time advances in simulation. In this study we examined the behaviour of the solvers for ill-conditioned matrices. We compared the CPU performance of the iterative solver *icoFoam* and the hybrid parallel codes (MPI+OpenMP) of a direct solver *SuperLU_DIST 4.0* (see [2]) at TGCC Curie (a Tier-0 system) thin nodes at CEA, France (see [5]). Moreover, we compared the performance of the hybrid parallel codes of MPI+OpenMP+CUDA versus MPI+OpenMP implementation of *SuperLU_DIST 4.0* at TGCC Curie (a Tier-0 system) hybrid nodes of CPU + GPU at CEA, France (see [5]). We discuss the performance, scalability and robustness of *OpenFOAM* on GPGPU cluster. We present our results regarding the speed-up of the solvers for the large matrices of size up to 20 million x 20 million. The authors thank to PRACE, GENCI and CEA for the opportunity to conduct our research in the frame of the Project 2010PA2505 awarded under the 18th Call for PRACE Preparatory Access.

I. INTRODUCTION

We investigated the challenges facing CFD solvers as applied to bio-medical fluid flow simulations and in particular the *OpenFOAM 2.1.1* solver, icoFoam, for the large penta-diagonal matrices coming from the simulation of blood flow in arteries with a structured mesh domain in PRACE-3IP project at TGCC Curie (a modern Tier-0 system) (see [1] and references therein). We generated a structured mesh by using blockMesh as a mesh generator tool. To decompose the generated mesh, we employed the decomposePar tool. After the decomposition, we used icoFoam as a flow simulator/solver tool. We achieved scaled speed-up for large matrices up to 64 million x 64 million matrices and speed-up up to 16384 cores on Curie thin nodes.

In this paper, we examined *OpenFOAM 2.2.2* "*icoFoam*" simulator with an iterative solver such as diagonal incomplete LU preconditioned bi-conjugate gradient in addition to direct solvers such as distributed *SuperLU 4.0* (see [2]). The flow problem produced various matrices as the time advances in simulation. The solution of the matrices obtained after each time step can be more challenging due to the changing structure of the matrices. This change may be caused by mess change or flow variable change. Generally the solution time of the matrices increases as the time advances in simulation.

It is challenging to discuss on the benefits or drawbacks of hybrid nodes. There are tradeoffs using GPU accelerators especially for the software packages or applications where it is not possible to fit the whole part into GPU. While it is expected to obtain a reduced time due to the accelerator, there would be communication over-head between the various processors and the GPU accelerators, as well. Therefore, it is important to obtain a feasible/optimal proportion of the tasks to MPI, OpenMP, and CUDA/OpenCL usages in emerging CPU+GPU systems. For example, it is not possible to do everything only in GPU for a complex algorithm like *SuperLU_DIST*. Therefore hybrid nodes like Curie hybrid nodes at CEA in France provide opportunity.

It would be interesting to discuss about the relative energy requirements for thin nodes versus hybrid nodes. A diversification of hardware solutions based on the application capability may be needed in order to attain a good efficiency (see [6] and [7]). While the compute partition of Curie thin nodes having total of 80,640 cores consumes 2132 kW, the partition of Curie hybrid nodes having total of 288 Intel® + 288 Nvidia processors uses 108.80 kW as the total power (see TOP500 Supercomputing sites [8] and the Green500 List [9]). The partition of Curie hybrid nodes outperforms the Curie thin nodes when the energy efficiency is compared in terms of performance per watt and the rates of computation are 1,010.11 MFLOPS/W and 637.43 MFLOPS/W, respectively.

The remainder of this work is organised as follows: In Section 2, the test environment and the flow of approach are described. In Section 3, thin nodes results of the CPU performance for the iterative solver *icoFoam* and the hybrid parallel codes (MPI+OpenMP) of a direct solver SuperLU_DIST 4.0 are compared. Moreover, simulation test results of hybrid node using MPI+OpenMP+CUDA versus MPI+OpenMP with *SuperLU_DIST 4.0* solvers are presented. Section 4 concludes this work.

II. Test Environment and Flow of Approach

OpenFOAM (see [10]) is an open source Computational Fluid Dynamics (CFD) toolbox. It is a software package with many tools for several main tasks of the simulation such as pre-processing (meshing), decomposition and solution. Here, the solver refers to not only linear system solver but also Navier Stokes solver and simulator.

The first four matrices in Table 1 are obtained at time 0.00005 (s) of the simulation where the time step size is 0.00005 (s), as in [1]. Unlike [1], the last six matrices in Table 1 are encountered at the third time step, at time 0.012 (s) of the simulation where the time step size is 0.004 (s). This is a relatively large time step size for such a very small mesh size. Thus, we obtained challenging ill-conditioned matrices. Almost 5 or 7 banded sparse matrix occurs at each time step and the matrices are described in Table 1.

The flowchart in Figure 1 shows the flow of approach in the paper.

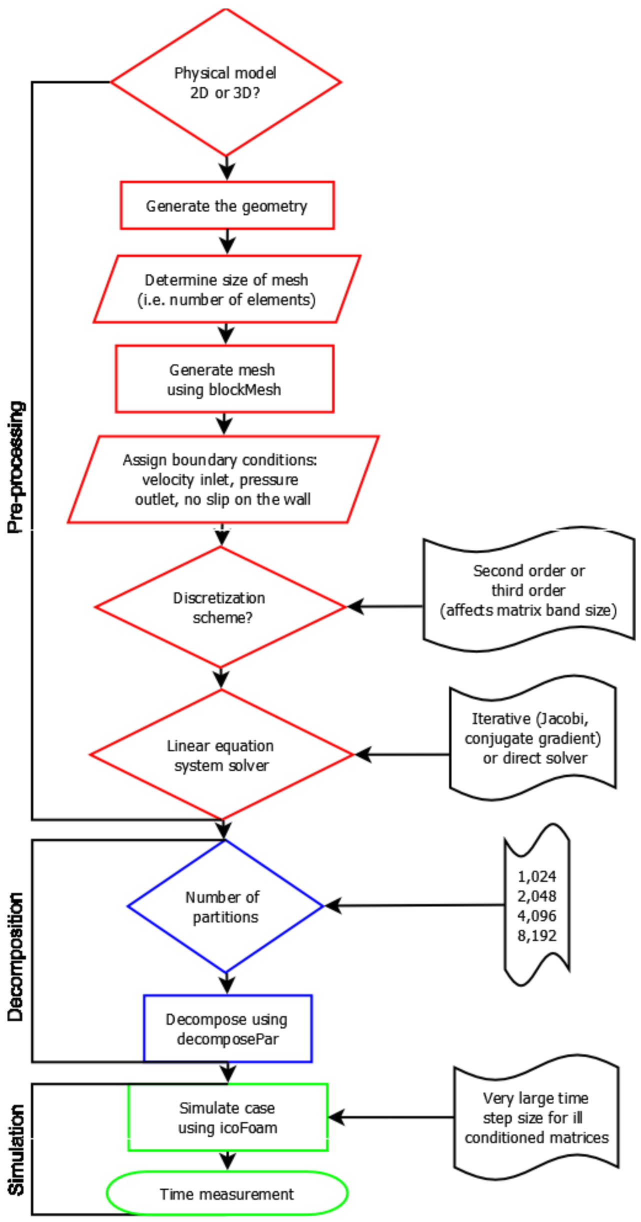

Pre-processing
Physical model 2D or 3D?
Generate the geometry
Determine size of mesh (i.e. number of elements)
Generate mesh using blockMesh
Assign boundary conditions: velocity inlet, pressure outlet, no slip on the wall
Discretization scheme?
Second order or third order (affects matrix band size)
Linear equation system solver
Iterative (Jacobi, conjugate gradient) or direct solver
Decomposition
Number of partitions
1,024
2,048
4,096
8,192
Decompose using decomposePar
Simulation
Simulate case using icoFoam
Very large time step size for ill conditioned matrices
Time measurement

Figure 1. Flowchart for the flow of the approach including the main tasks

III. Test Results

The tests were done for only a few time steps due to time limitations, while the real case runs are conducted for more than thousands of time steps. No single CPU solution was possible because of long waiting times, so, information regarding the pre-processing (meshing), partitioning etc. are given for parallel processing. The most time consuming part of the simulation was the decomposing of the mesh. For 8192 partitions, it took over 3 hours. The "Simple" decomposition method was preferred since the running cases were for a structured mesh. This technique simply splits the geometry into pieces by direction, such as 32 pieces in x direction and 32 pieces in y direction. Since the mesh is structured, mC_20M_n matrix means 20M of cells in the fluid domain.

TABLE I. Description of Matrices

	N	NNZ	NNZ/N	Origin
mC_8M	8,000,000	39,988,000	4.999	ITU Mathematics
mC_16M	16,000,000	79,984,000	4.999	ITU Mathematics
mC_6M_D	6,000,000	41,800,000	6.967	ITU Mathematics
mC_8M_D	8,000,000	55,760,000	6.970	ITU Mathematics
mC_8M_n	8,000,000	39,988,000	4.999	ITU Mathematics
mC_16M_n	16,000,000	79,984,000	4.999	ITU Mathematics
mC_20M_n	20,000,000	99,982,000	4.999	ITU Mathematics
mC_6M_n_D	6,000,000	41,780,000	6.963	ITU Mathematics
mC_8M_n_D	8,000,000	55,760,000	6.970	ITU Mathematics
mC_10M_n_D	10,000,000	69,660,000	6.966	ITU Mathematics

A. Thin Node Results

We compared the CPU performance of the iterative solver *icoFoam* and the hybrid parallel codes (MPI+OpenMP) of a direct solver SuperLU_DIST 4.0 (see [2]) at TGCC Curie (a Tier-0 system) thin nodes at CEA, France (see [5]). Fig. 2 and Fig. 3 show the wall-clock time comparisons of the solvers, excluding the refinement time, for mC_16M_n and mC_20M_n on Curie thin nodes, respectively. The iterative solver with a diagonal incomplete LU preconditioned bi-conjugate gradient outperforms the direct solver *SuperLU_DIST 4.0* for the simulation matrices.

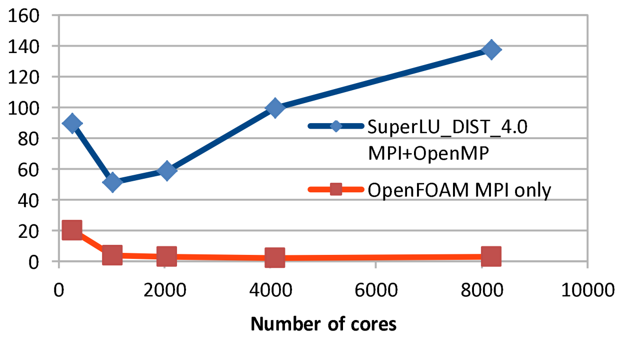


Figure 2. Wall-clock time comparison of the solvers for mC_16M_n on Curie thin nodes

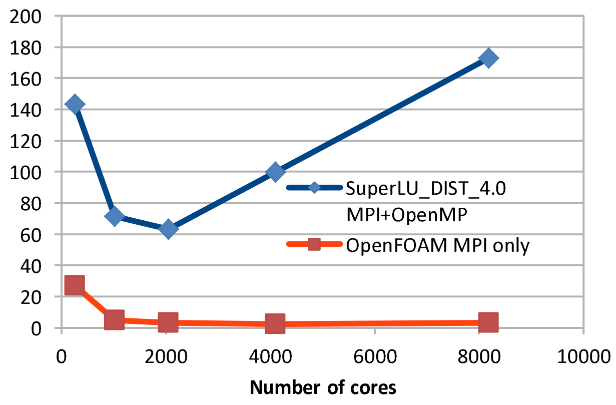


Figure 3. Wall-clock time comparison of the solvers for mC_20M_n on Curie thin nodes

B. Hybrid Node Results Using MPI+OpenMP+CUDA

TABLE II. THE CONFIGURATION OF MPI+OPENMP AND MPI+OPENMP+CUDA FOR THE DIRECT SOLVER

Testbed:CURIE/	**hybrid**	**hybrid**	**hybrid**	**hybrid**
SuperLU_DIST version	4	4	4	4
# of cores	64	256	512	1024
# of processes	16	64	128	256
# of threads per process	4	4	4	4
# of GPUs per process	1	1	1	1

We compared the performance of the hybrid parallel codes of MPI+OpenMP+CUDA (see [4]) versus MPI+OpenMP implementation of SuperLU_DIST 4.0 at TGCC Curie (a Tier-0 system) hybrid nodes of CPU + GPU at CEA, France (see [5]). Table 2 describes the corresponding configurations while we run the direct solver.

Table 3 shows the performance results for the ten simulation matrices described in Table 1. For example, Fig. 4 shows the comparison for the performances of MPI+OpenMP+CUDA and MPI+OpenMP implementations for mC_20M_n on Curie hybrid nodes. In Fig. 5, we observe a linear speed-up of the direct solver up to 512 cores for both implementations for mC_20M_n on Curie hybrid nodes.

Generally, we see that MPI+OpenMP implementation outperforms the hybrid of MPI+OpenMP+CUDA for this set of simulation matrices when we consider the wall clock times for the optimal number of cores because of several overheads coming from CUDA implementation for the direct solver algorithm. It is not possible to put everything only in GPU for *SuperLU_DIST*. Therefore, the tasks should be proportioned to MPI, OpenMP, and CUDA/OpenCL. In *SuperLU_DIST 4.0* (see [4]), cuBLAS library execution is one of the most time consuming tasks performed in GPU in order to gain from explicit parallelization. On the other hand, there are overheads such as data transfer on PCIe between host and device memory (CPU and GPU) and new data structure changes related to data packing and scattering. Moreover, *SuperLU* is a complex algorithm and it is challenging to select the right combination for better intra-node communications and inter-node communications within CPU+GPU heterogeneous systems, under current technology limitations (see [3]).

The last eight matrices in Table 3 are challenging large matrices because they are relatively denser or ill-conditioned. The error labelled Error 1 occurs for small number of cores. We meet with an error message labelled Error 2 related to buffer size during the factorization subroutine pdgstrf, for the hepta-diagonal matrices. Error 3 is a CUDA stream error related to setting cuBLAS library execution stream.

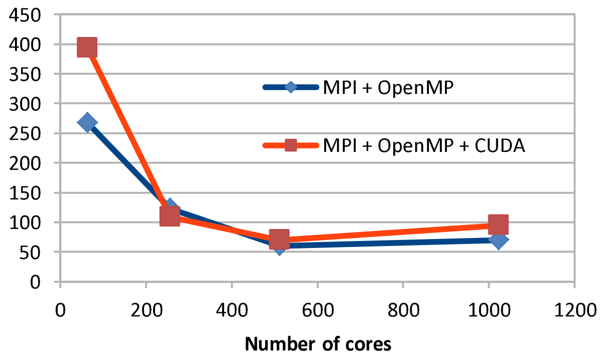


Figure 4. Wall-clock time of direct solver for mC_20M_n on Curie hybrid nodes

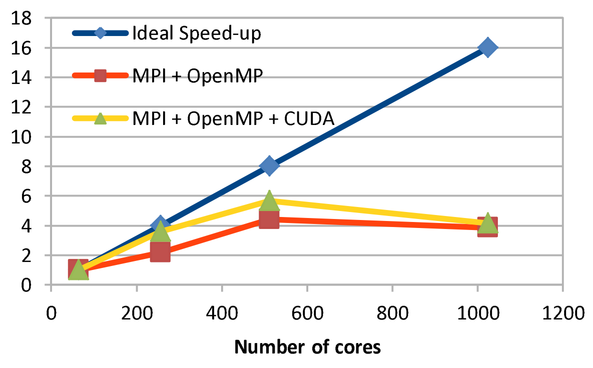


Figure 5. Speed-up of direct solver for mC_20M_n on Curie hybrid nodes

TABLE III. WALL CLOCK TIMES (S) OF SUPERLU_DIST 4.0 FOR THE LARGE PENTA-DIAGONAL MATRICES FOR 2D PROBLEMS AND HEPTA-DIAGONAL MATRICES FOR 3D PROBLEMS, DESCRIBED IN TABLE I, ON MPI+OPENMP VERSUS MPI+OPENMP+CUDA IMPLEMENTATIONS

Matrices	**/ Number of cores**	64	256	512	1024
mC_8M	MPI + OpenMP	99.96	34.70	28.78	37.89
	MPI + OpenMP + CUDA	94.70	39.10	43.70	60.72
mC_16M	MPI + OpenMP	230.30	83.19	47.73	59.02
	MPI + OpenMP + CUDA	236.83	87.23	60.00	81.41
mC_6M_D	MPI + OpenMP	Error 1	260.38	296.74	239.52
	MPI + OpenMP + CUDA	Error 1	Error 2	254.44	257.15
mC_8M_D	MPI + OpenMP	Error 1	1005.96	516.86	387.20
	MPI + OpenMP + CUDA	Error 1	680.25	Error 2	353.40
mC_8M_n	MPI + OpenMP	94.70	31.00	32.79	35.83
	MPI + OpenMP + CUDA	70.94	38.27	Error 3	61.34
mC_16M_n	MPI + OpenMP	181.53	75.93	49.53	58.61
	MPI + OpenMP + CUDA	233.22	75.58	61.42	83.61
mC_20M_n	MPI + OpenMP	266.82	122.59	60.30	69.49
	MPI + OpenMP + CUDA	393.49	108.90	69.60	94.99
mC_6M_n_D	MPI + OpenMP	1178.51	409.15	248.84	211.70
	MPI + OpenMP + CUDA	782.22	294.14	Error 2	222.04
mC_8M_n_D	MPI + OpenMP	Error 1	948.03	533.78	386.72
	MPI + OpenMP + CUDA	Error 1	682.02	Error 2	349.16
mC_10M_n_D	MPI + OpenMP	Error 1	877.92	465.60	373.09
	MPI + OpenMP + CUDA	Error 1	752.78	Error 2	Error 3

IV. CONCLUSION

We performed bio-medical fluid flow simulations for the large matrices coming from the simulation of blood flow in arteries in emerging CPU+GPU systems. The flow problem produced various challenging matrices during the simulation. We compared the CPU performance of the iterative solver icoFoam and the hybrid parallel codes (MPI+OpenMP) of a direct solver SuperLU_DIST 4.0 (see [2]) at TGCC Curie (a Tier-0 system) thin nodes at CEA, France (see [5]). We observe that the iterative solver with a diagonal incomplete LU preconditioned bi-conjugate gradient outperforms the direct solver *SuperLU_DIST 4.0* for the simulation matrices. Moreover, we compared the performance of the hybrid parallel codes of MPI+OpenMP+CUDA versus MPI+OpenMP implementation of SuperLU_DIST 4.0 at TGCC Curie (a Tier-0 system) hybrid nodes of CPU + GPU at CEA, France (see [5]). Generally, we notice that MPI+OpenMP implementation outperforms the hybrid of MPI+OpenMP+CUDA for the set of simulation matrices when we consider the wall clock times for the optimal number of cores because of several overheads coming from CUDA implementation for the complex direct solver algorithm. Furthermore, we met with several errors for the challenging simulation matrices. We believe that the technology developments in emerging CPU+GPU systems will increase the scalability of related complex algorithms by eliminating the bottlenecks coming from communication and right matching of system components required for special applications.

ACKNOWLEDGEMENT

This research was supported by the Project 2010PA2505 awarded under the 18th Call for PRACE Preparatory Access and we acknowledge that the results of this research have been achieved using the PRACE Research Infrastructure resource TGCC Curie (a modern Tier-0 system) based at CEA in France.

REFERENCES

[1] A. Duran, M.S. Celebi, S. Piskin, and M. Tuncel, “Scalability of OpenFOAM for bio-medical flow simulations,” Journal of Supercomputing, 71(3), 2015, pp. 938-951.

[2] X. S. Li, J. W. Demmel, J. R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki, SuperLU Users' Guide, Tech. Report UCB, Computer Science Division, University of California, Berkeley, CA, 1999, update: 2011

[3] M.S. Celebi, A. Duran, M. Tuncel and B. Akaydın, Scalable and improved SuperLU on GPU for heterogeneous systems, *PRACE (Partnership for Advanced Computing in Europe),* PRACE PN: *283493, PRACE-2IP white paper, Libraries,* WP 44, July 13, 2012.

[4] P. Sao, R. Vuduc, and X.S. Li, “A distributed CPU-GPU sparse direct solver,” Euro-Par 2014 Parallel Processing, Lecture Notes in Computer Science vol. 8632, 2014, pp. 487-498.

[5] http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm.

[6] N. Meyer, M. Lawenda, et al., Best Practices for HPC Procurement and Infrastructure, PRACE-2IP project, under Grant agreement No. RI-283493, Aug. 2013.

[7] J. David, JN Richet, E. Boyer, N. Anastopoulos, G Collet, GC Verdiere, et al., Best Practice Guide - Curie v1.17, PRACE, Nov. 2013.

[8] TOP500 Supercomputing sites, http://top500.org/

[9] The Green500 List, http://www.green500.org

[10] OpenFOAM main site. http://www.openfoam.com

Code modernization of DL_MESO LBE to achieve good performance on the Intel Xeon Phi.

Sergi Siso and Luke Mason
Intel Parallel Computing Centre, Hartree Centre
Science and Technology Facilities Council
Daresbury, United Kingdom
sergi.siso@stfc.ac.uk, luke.mason@stfc.ac.uk

Michael Seaton
Computational Chemistry Group
Science and Technology Facilities Council
Daresbury, United Kingdom
michael.seaton@stfc.ac.uk

Abstract

The Lattice Boltzmann Equations of DL_MESO have been re-implemented using a Two-Grid algorithm, threaded programming and vectorization in order to effectively utilize novel highly parallel architectures such as the one offered by the Intel Xeon Phi.

I. INTRODUCTION

DL_MESO LBE [1][2] is a general purpose mesoscopic simulation package which can simulate multi component lattice-gas systems using the Lattice Boltzmann Equation (LBE). It is used to model systems with multiple fluids and/or phases coupled to solute diffusion and heat transfer, as well as apply geometrically complex boundaries comparatively easily.

The main version of this code implements a SWAP algorithm [3], where data dependencies are circumvented by a strict processing order of the lattice nodes. The SWAP algorithm performs well using multiple low core count processors with MPI, however, performance in a modern architecture like the Intel Xeon Phi was disappointing.

In this work we have re-implemented the Lattice Boltzmann solver within DL_MESO using OpenMP threads and SIMD clauses and a double-buffered and computationally simpler algorithm. This implementation utilises twice as much memory but shows better performance (x5 on Xeon Phi and x1.7 in Xeon processors) and scalability characteristics. The new implementation has the advantage of being ready to use the features of other modern and future hardware architectures to come.

This paper starts describing the Xeon Phi architecture in section 2 and the DL_MESO Lattice Boltzmann physics in section 3. Then, it explains the SWAP algorithm implementation used in the original DL_MESO package, section 4, and the Two Grid implementation we have implemented for the new version of the code, section 5. Finally section 6 gives details of the vectorization while section 7 compares the performance and scalability of the two versions of the code.

II. INTEL XEON PHI

The target architecture for this project is the Intel Xeon Phi Knights Corner coprocessor. The Xeon Phi is a highly parallel architecture which consists of many small, power efficient, in-order cores, each of which has a 512-bit vector processing unit (SIMD unit) [4]. It is a x86 64bits architecture with a cache memory system similar to general purpose CPUs, therefore legacy x86 codes that use to run well on CPUs also run on the Xeon Phi, but due to its large number of threads and vector units length, it is challenging to port existing applications to this platform. Nevertheless, given the trends seen in CPU design in the recent years, porting applications to the Xeon Phi helps to prepare the applications to future generations of hardware.

Specifically, the results presented in this paper are obtained using the Intel Xeon Phi Coprocessor 5110P with the following specifications:

- 60 cores, 240 threads (4 threads/core)
- 1.053 GHz,
- 1 TeraFLOP double precision theoretical peak performance,
- 8 GB memory with 320 GB/s bandwidth,
- 512 bit wide SIMD vector engine,
- 32KB L1, 512KB L2 cache per core,
- Fused multiply-add (FMA) support.

The performance is compared with a compute node using two Intel Ivy Bridge Xeon E5-2697 v2 at 2.70GHz [5].

III. Lattice Boltzmann Method and DL_MESO

DL_MESO is a C++ general purpose mesoscopic simulation package which comes with two different simulation methods: Dissipative Particle Dynamics (DPD) and Lattice Boltzmann equations (LBE) [1][2]. The current work is concerned with the LBE method exclusively.

The LBE is a computational fluid dynamics method which has emerged from the lattice-gas automata and is used to simulate a multitude of flow problems. In LBE a fluid can be represented by using the probability of finding one of its particles at a given position in space and time with a given momentum, described by a density distribution function, $f(\vec{x},\vec{p},t)$, depending on the position, $\vec{x}$, the momentum, $\vec{p}$ and time, *t*. Over a single time-step Δt the distribution function at each lattice point $\vec{x}$ evolves initially by collisions

$$f_i(\vec{x},t^+) = f_i(\vec{x},t) + C_i \tag{1}$$

and then by propagation between neighbouring lattice sites

$$f_i(\vec{x}+\vec{e}_i\Delta t, t+\Delta t) = f_i(\vec{x},t^+) \tag{2}$$

where $\vec{e_i}$ is the velocity vector for lattice link $(\vec{\imath},t^+)$ denotes a time during the time-step after collisions have taken place and C_i is a collision operator acting on deviations from local equilibria.

DL_MESO implements several choices for collision the variable. The work presented in this paper we have used the single relaxation step Bhatnagar-Gross-Krook (BGK) [5]

$$C_i = -\frac{f_i - f_i^{eq}}{\tau} \tag{3}$$

with a relaxation time τ related to the kinematic viscosity of the fluid ν. The distribution function at local equilibrium for a particular point can be determined from its density and velocity

$$f_i^{eq} = \rho w_i\left[1 + 3(\vec{e_i}\cdot\vec{u}) + \frac{9}{2}(\vec{e_i}\cdot\vec{u})^2 - \frac{3}{2}\vec{u}^2\right] \tag{4}$$

The relaxation time is determined by

$$\nu = c_s^2\left(\tau - \frac{1}{2}\right)\Delta t \leftrightarrow \tau = \frac{\nu}{c_s^2\Delta t} + \frac{1}{2} \tag{5}$$

where c_s is the speed of sound for the lattice in use: this value is equal to $\Delta\vec{x}/(\sqrt{3}\Delta t)$ for square lattices following the D3Q19 form.

To deal with interactions between multiple phases and fluid species the Shan–Chen algorithm [9][10] is used in this work (DL_MESO has more options), which gives the force on fluid a as

$$F_a = -\psi_a(x)\sum_b g_{ab}\sum_i w_i\psi_b(\vec{x}+\vec{e_i})\vec{e_i} \tag{6}$$

g_{ab} is the interaction coefficient between species a and b, which can be related directly to the interfacial tension between the species σ_{ab}, and ψ_a is a pseudo-potential for species a, related to the fluid density of that species.

The advantage of DL_MESO over other packages implementing the Lattice Boltzmann equation is that it allows computing multiple components and/or fluid phases and coupling them with other physics like the heat transfer function or solute diffusion equations. This flexibility to plug in different physics makes DL_MESO LBE ideal to simulate applications such as the cavity flow problem [13], or subjecting an initially stationary fluid to a temperature difference between two solid boundaries [14], shown at figure 1.

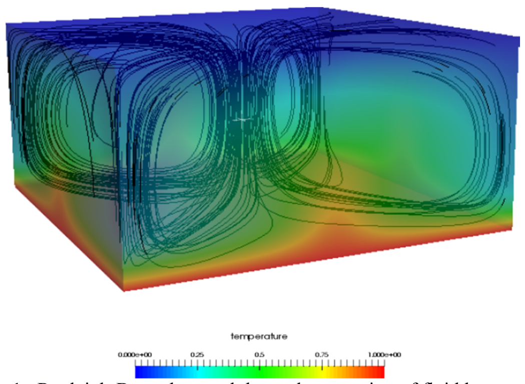


Figure 1. Rayleigh-Benard natural thermal convection of fluid between cold and hot plates.

The Lattice Boltzmann method is generally suitable for parallel computing and it is easy to code, however, the mentioned flexibility in the physics comes with some performance constraints. The necessity of computing the pseudo-potentials in each time step and therefore having non-local collision computations complicates the algorithm compared to other LBE implementations.

IV. SWAP Algorithm

There exist several algorithms to implement the Lattice Boltzmann equations and each of them offers different trade-offs in terms of memory utilization and performance [8]. Originally, DL_MESO LBE was implemented using a SWAP algorithm [3] with the MPI parallel programming model. In SWAP algorithm the data dependencies on the grid are avoided by a strict processing order of the lattice nodes and by explicitly exchanging some of the distribution values of the lattice node at hand with those of neighbouring nodes.

Because it removes the data dependencies, it requires only one lattice to store the state of the simulation during the runtime. Hence, it is very efficient in terms of the total memory we need to allocate for a given problem size.

However, the SWAP algorithm is challenging to implement in highly parallel architectures due to the strict order of processing of the different lattices. For instance, we need to ensure that each lattice of a neighbour particle has computed its collision before we swap them. In the original implementation of the code this involves five different loops, all going through the whole data-structure. This effectively creates multiple parallel barriers in each iteration and the cache utilization is poor. The vectorization of the arithmetic operations is also challenging because the inner loop has small trip counts and often we need to introduce peeling and remainder loops to accommodate the size of the loop to the vector registers.

Table 1 shows the routines of the original DL_MESO code where most execution time is spent in an execution on a Xeon processor, together with the L1, L3 and DRAM and the percentage of Floating Point operations performed at the vector units on that execution. This data was obtained using the Intel VTune Performance analysis tool [11].

Function	Time (%)	L1 Bound	L3 Bound	DRAM Bound	Vect FP (%)
fGetSpeedShanChe.	13.0	0.299	-	-	0
fSwapPair<double>	12.0	0.697	0.076	0.892	0
MPID_nem_sshm_.	10.2	0.425	0.033	0	0
fGetAllMassSite	7.40	0.241	0.504	0	100
fCalcInteraction_Sh.	7.00	0.151	0.007	0.04	60
fGetEquilibriumF	6.80	0.318	0.009	0	55

Table 1. Performance metrics of DL_MESO LBE SWAP implementation.

V. Two-Grid Algorithm

The Two-Grid Algorithm implemented in DL_MESO in the current work is a simpler approach. The probability function distributions are stored in two different grids, A and B. The pseudo-potentials and the collisions are performed with the values of the grid A and the post-collision values of each node are immediately streamed to their neighbours and stored in lattice B. At the end of the iteration lattice pointers to A and B are swapped and the iteration starts over

Although the Two-Grid Algorithm is not as sophisticated as the SWAP algorithm, it does not have compute order dependencies and allows a more natural parallel implementation from both: threads and vector instructions.

DL_MESO used the OpenMP capabilities to distribute the work between different threads using a static schedule and among the vector lanes using the SIMD directive available since OpenMP 4.0 [12]. Hence, the parallel implementation is just a few additional lines to the serial implementation of the code. This greatly reduces the complexity of the code because both implementations share the same code base.

The new code has the main advantage that the algorithm loops can be fused into just two, one for the pseudo-potentials pre-computations and another for the collision and streaming steps. Consequently, the data-structure is only traversed twice and each element that is reads has more computational steps to be done. This leads to a lower cache pressure as shown in Table 2. The table presents main function in the Two-Grid version of the code, again with the L1, L3, DRAM and vectorization usage provided by VTune.

Function	Time (%)	L1 Bound	L3 Bound	DRAM Bound	Vect FP (%)
solve_x_iterations	66.2	0.54	0.004	0.234	100

Table 2. Performance metrics of DL_MESO LBE Two-Grid implementation.

VI. Vectorization and Compile Time Parameters

To get a good performance from the explicit vectorization implementation we rely in the ability of the compiler to unroll some loops. Specifically, we want to unroll the two most inner loops in order to vectorise the 3rd nested loop, which has a considerably larger trip count. However, the compiler does not have enough information during compile time to properly execute such decisions. Therefore, our solution is to provide some of the simulation parameters at the compiling stage enabling the compiler to take smarter decisions.

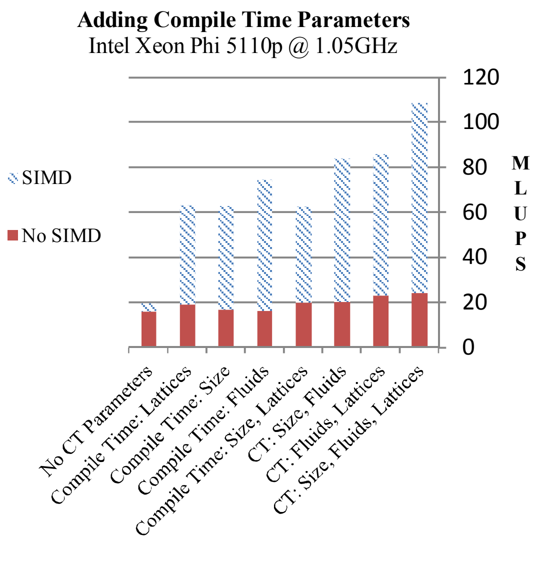


Figure 2. MLUPS of DL_MESO LBE simulation with different parameter set up at compile time.

Figure 2 presents the Million Lattice Update Per Second (MLUPS) for several combinations of parameters defined at compile time. The figure demonstrates that when we provide more parameters the compiler is able to produce significantly better vectorization of the code, and that leads to faster iterations.

VII. Performance and Scalability Comparison

Figure 3 shows how the performance on the Intel Xeon Phi is improved by more than five times with the new implementation compared to the SWAP algorithm. The performance on the Xeon processors is also improved by 1.5.

Considering the new performance executing a simulation on the Intel Xeon Phi is 70% faster than in a 2 by 12 core Xeon node. Also, this new implementation has the advantage of being ready to use the features of future hardware architectures where longer vector units and more cores are expected.

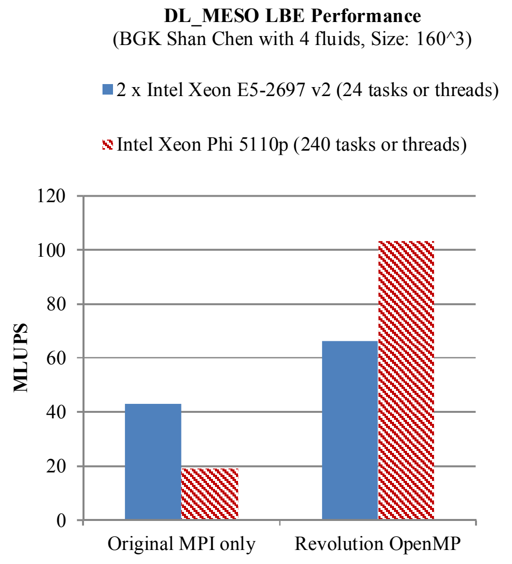


Figure 3. Performance comparison of the SWAP and Two-Grid algorithm in the Xeon and the Xeon Phi processors.

Figure 4 plots the scalability of both implementations in the Xeon Phi architecture. Note that the SWAP algorithm was implemented using MPI programming model while the Two-Grid algorithm is implemented using the OpenMP API for threaded programming. In both cases they scale reasonably well until 60 cores, one thread per core in the Xeon Phi 5110P, but after 60 cores the Two-Grid OpenMP version scales better by reaching x140 speed up at 240 threads while the SWAP implementation reaches 81 at 240 MPI processes.

References

[1] M. A. Seaton, R. L. Anderson, S. Metz and W. Smith, DL_MESO: highly scalable mesoscale simulations, Mol. Sim. 39 (10), 796–821 (2013)

[2] M. A. Seaton and W. Smith, DL_MESO User Manual

[3] K. Mattilaa, J. Hyväluomab, T. Rossia, M. Aspnäsc and J. Westerholmc, An efficient swap algorithm for the lattice Boltzmann method, Computer Physics Communications. February 2007

[4] Intel Corporation, Intel Xeon Phi Coprocessor System Software Developers Guide, 2012, http://software.intel.com/en-us/mic-developer.

[5] Intel® Xeon® Processor E5-2600 v2 Product Family. http://ark.intel.com/products/series/75291/Intel-Xeon-Processor-E5-2600-v2-Product-Family#@All

[6] J. Latt, How to implement your DdQq dynamics with only q variables per node (instead of 2q), Technical Report, Tufts University, Medford, USA, 2007

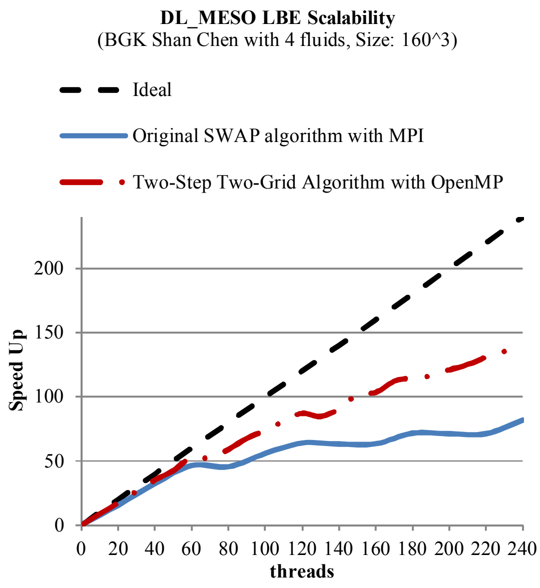


Figure 4. Parallel scalability comparison of the SWAP algorithm (MPI) and the Two-Grid algorithm (OpenMP) on the Intel Xeon Phi.

[7] P.L. Bhatnagar, E.P. Gross, M. Krook, A model for collision processes in gases. I. Small amplitude processes in charged and neutral one-component systems, Phys. Rev. 94 (3) (1954) 511–525.

[8] M. Wittmanna, T. Zeisera, G. Hagera, G. Welleinb. Comparison of different Propagation Steps for the Lattice Boltzmann Method, Computers and Mathematics with Applications

[9] X. Shan and H. Chen. Lattice Boltzmann model for simulating ows with multiple phases and components. Physical Review E: Statistical Physics, Plasmas, Fluids, and related interdisciplinary topics, 47(3):1815{1819, March 1993.

[10] X. Shan and H. Chen. Simulation of nonideal gases and liquid-gas phase transitions by the lattice Boltzmann equation. Physical Review E: Statistical Physics, Plasmas, Fluids, and related interdisciplinary topics, 49(4):2941{2948, April 1994.

[11] Intel® VTune™ Amplifier 2016. https://software.intel.com/en-us/intel-vtune-amplifier-xe

[12] OpenMP Application Program Interface Version 4.0 - July 2013

[13] S. Hou, Q. Zou, S. Chen, G. Doolen, and A.C. Cogley, Simulation of cavity flow by the lattice Boltzmann method, J. Comp. Phys. 118 (1995), pp. 329-347.

[14] Z. Guo, B. Shi, and C. Zheng, A coupled lattice BGK model for the Boussinesq equations, Int. J. Numer. Meth. Fl.39 (2002), pp. 325-342.

Back to the Future? High Performance Computing and ARM

Chris Adeniyi-Jones,
ARM Research Cambridge,
United Kingdom

Abstract—**Achieving Exascale levels of performance for HPC systems within a sustainable power budget is the widely held goal for many. Improving the efficiency of HPC systems is going to require novel technologies and new ways of using existing technology. ARM is a leading provider of Intellectual Property. Our range of IP products and our partnership business model enables our partners to address many different markets. I will highlight the Mont-Blanc project as an example of a different approach to HPC; what was done and what we learnt. I will also discuss the changes over the past five years and how new products are helping us prepare for the next five years of advances in high-performance computing.**

Low-Power, Fault-Resilient Communications in a Million-Core Neural Processing Architecture

Javier Navaridas, Mikel Luján, Luis A. Plana and Steve B. Furber
School of Computer Science, The University of Manchester
Oxford Road, Manchester, UK, M13 9PL

***Abstract*— The SpiNNaker neuromimetic architecture is a biologically-inspired massively-parallel architecture based on a custom-made multicore System-on-Chip (SoC). In order to deal with the challenging and atypical communication demands of spiking neural networks, SpiNNaker features a specialised, ad hoc communication infrastructure based around a custom-made multicast router. This paper summarizes the main research work around the peculiarities of SpiNNaker's architecture and interconnects focusing on the most exceptional features of the platform. We derived the main topological properties of the network, analyzed the effects of failures congestion and traffic burstiness in the stability of the interconnect and, finally, proposed a collection of multicast routing algorithms.**

Keywords—Interconnection Networks, Massively Parallel Systems, Multicast Routing Algorithms, Performance Evaluation.

I. Introduction

SpiNNaker is a bespoke massively-parallel architecture targeting the simulation, in biological real-time, of very large-scale *spiking neural networks,* with more than 10^9 neurons. To put this number in context, it roughly represents 10% of the human cortex. Spiking neural networks communicate by means of spike events which occur when a neuron is stimulated beyond a given threshold and discharges an electrical impulse. These spikes are communicated to all connected neurons, with typical fan-outs of the order of 10^3. At a realistic biological firing rate of 10Hz, there could be more than 10^{10} neuron firings per second, which can replicate up to 10^{13} communication events per second in the largest SpiNNaker configuration. Thus, an essential problem inherent to the simulation of spiking neural networks is how to distribute large numbers of small packets very widely amongst up to the million processors featured by SpiNNaker in an efficient way and with minimal latency. In this paper, we discuss the most important research that has been carried out around the custom-made interconnection network of SpiNNaker. In [1], we performed a theoretical analysis of the network, deriving theoretical properties such as the maximum throughput and distance-related characteristics of its topology. We also studied the fault tolerance capabilities of the system showing that the network is able to adapt and to remain stable in the presence of failures [3]. Later on, we investigated how the traffic burstiness inherent to the application affects network stability. As increasingly larger configurations of the system have become available, there has been a great effort on understanding how to better exploit SpiNNaker's multicast infrastructure [4, 5] in order to prevent the interconnection network from becoming the main performance and scalability bottleneck.

A. Machine Construction Progress

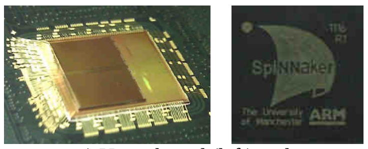
The construction of large-scale versions of SpiNNaker is ongoing and is expected to culminate with the million-core system in the following months. Some prototypes and production systems have already been designed and fabricated. Back in 2010 a first batch of test chips (two cores and a fully functional router) were produced and successfully demonstrated running spiking neural nets. This was followed in 2011 by the production of small quantities of full-fledged SpiNNaker chips with 18 ARM cores and the development of small boards, able to house four SpiNNaker chips and to support inter-board communications. These boards, due to their low-power design, have been used as control devices for robotic systems. As shown in Fig. 1, we have reached most milestones in the path towards the full-fledged SpiNNaker: a 48-chip board has been designed and large numbers of them have been produced and can be interconnected to construct increasingly large machines. One board forms a 10^3-core machine, one rack frame with 24 boards forms the 10^4 one, a cabinet with five of these frames (120 boards) forms the 10^5 machine. Very recently 5 of these cabinets have been installed. The final expansion to reach the full-fledged, million-core SpiNNaker machine will be to add 5 more cabinets together, which will be done in the next few months.

B. System software and libraries

Aside from the hardware, an extensive collection of system software and application libraries is already offered to operate SpiNNaker and new features are developed and released frequently. Among all the software involved in SpiNNaker, is especially important the PyNN frontend, a domain specific language devised to define spiking neural networks widely used within the neuroscientist community. The PyNN/ SpiNNaker combination exploits the system flexibility and decouples neural applications from the actual hardware, allowing users to rapidly develop and simulate spiking neural networks without any knowledge of the intricacies of the underlying system. Thanks to this transparency for the end-user, the adoption of SpiNNaker as a simulation platform is rapidly growing within the cognitive computing community.

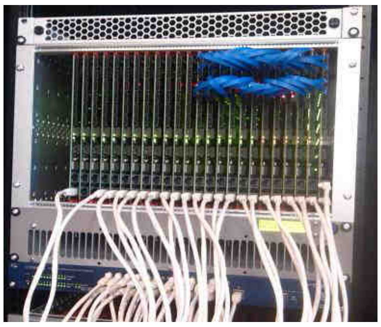
II. SpiNNaker Architecture

The main foundations of SpiNNaker's design philosophy were to reduce power consumption, to improve reliability by means of high redundancy and to provide a flexible architecture, general enough to run a wide range of applications. Each SpiNNaker SoC contains 18 low-power



a) Unpackaged (left) and packaged (right) SpiNNaker chips.

b) 48-chip production board (10^3 machine)



c) 24-board frame (10^4 machine)

d) 19" rack cabinets (halfway to 10^6 machine)

Fig. 1. Different sizes of SpiNNaker Machines, from a single chip to the recently installed 5-cabinet system.

general-purpose ARM968 cores, typically running an independent event-driven process, which responds to events generated by different on-chip modules: timer, communications controller and DMA controller, among others. The chip is packaged together with a 128 MB SDRAM, whose primary function is to store synaptic information. The chips are interconnected (see Fig. 2) using a two-dimensional triangular torus relying on a custom-made on-chip multicast router which handles both inter- and intra-chip communications. It has 18 ports for the cores plus six external ports to communicate with adjacent chips. To avoid the high complexity intrinsic to crossbar-based designs, the SpiNNaker router uses a simpler architecture in which ports are hierarchically merged into a single pipelined queue so that only one packet can use the routing engine at once. The routing engine is not expected to become a bottleneck as it has much higher bandwidth than the transmission ports (8 Gbps vs. 250 Mbps). The router supports point-to-point and multicast communications using small packets of 40 bits. The multicast engine reduces the pressure at the injection ports and the number of packets traversing the network and so is the main communication method during regular operation of the system.

Another interesting aspect of the interconnection architecture is the routing paradigm. Following the Address Event Representation protocol, packets do not contain any information about their destination(s), only an identifier of the neuron that has fired. The information necessary to deliver a neural packet to all the relevant cores and chips is compressed and distributed across the 1024-word routing table within each router. To minimize the impact of such an exiguous resource and allow the system to perform complex routing, routing tables offer a masked associative route look-up and routers are designed to perform a default routing—which requires no entry in the routing table—by sending the packet to the port opposite to the one the packet comes from, i.e. if a packet comes from the North it will be sent to the South. Routing tables are not intended to remain static during long simulations. They can be modified dynamically in real-time to accommodate scenarios of congestion or even failures in the interconnection network.

The flow-control is very simple: when a packet arrives to the routing engine, one or more output ports are selected and the router tries to transmit the packet through them. If the packet cannot be forwarded, the router will keep trying, and after a given period of time it will also test the clockwise emergency route. Finally, if a packet stays in the router for longer than a given threshold the packet will be dropped to avoid deadlock. This threshold is an arbitrary router configuration parameter which was thoroughly studied in [1].

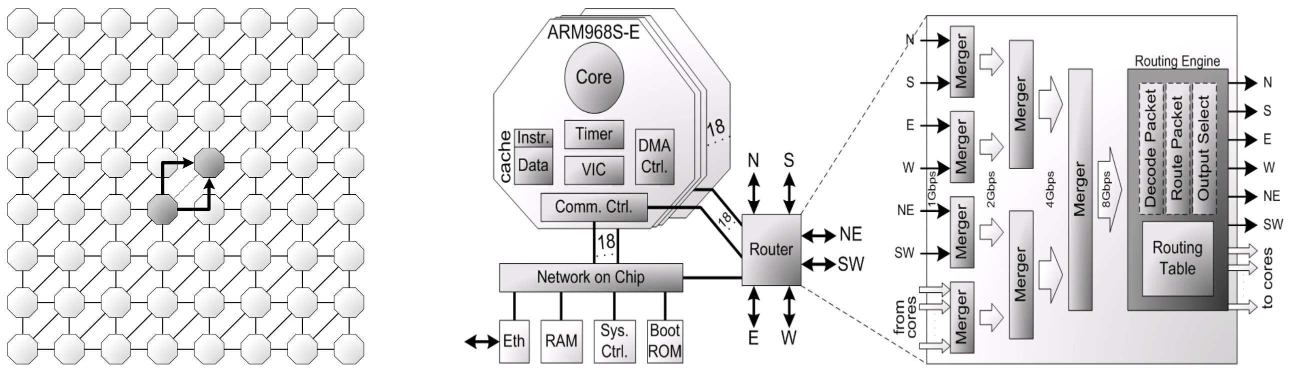


Fig. 2. Example of an 8×8 SpiNNaker topology (left) and a communication-centric diagram with the main components of a chip (right). The peripheral connections of the topology are not depicted for the sake of clarity. The regular route (slashed line) and the two emergency routes (thick arrows) between the shaded nodes are shown.

III. EVALUATING THE INTERCONNECTION NETWORK

A. *Topological Analysis*

One of the most important aspects of a network is its topology as it has a great effect on the overall performance both in terms of bandwidth and latency. As a first step to assess SpiNNaker network [1], we derived theoretical properties such as the maximum throughput (Θ), the average distance (δ) and the diameter (D) based on the nodes per dimension (*n*):

$$\Theta = \frac{16 \cdot n}{n^2} = \frac{16}{n} \, packets/cycle/node \tag{1}$$

$$D = \frac{n}{2} + \left\lfloor \frac{n}{6} \right\rfloor = \left\lfloor \frac{2 \cdot n}{3} \right\rfloor \tag{2}$$

$$\delta = \frac{\sum_{i=1}^{\frac{n}{2}} \left(6 \cdot i^2\right) - 3 \cdot \frac{n}{2} + \sum_{i=1}^{\left\lfloor \frac{n}{6} \right\rfloor} 2 \cdot \left(\frac{n}{2} + i\right) \cdot \max\left(1, 9 \cdot \left(\left\lfloor \frac{n}{6} \right\rfloor - i\right) + 3 \cdot \left(\frac{n}{2} \bmod 3\right)\right)}{n \cdot (n-1)} \tag{3}$$

B. *Network stability under failures*

Another interesting insight coming from our previous evaluations is the stability of SpiNNaker's interconnect in the presence of failures thanks to the emergency routing mechanism [1]. Fig. 3 shows the temporal evaluation of a SpiNNaker system when the emergency routing is either deactivated or activated. In these simulation results, we started with a fault-free network and every 5k cycles the number of faults is increased (to 1, 2, 4, 8, etc as per the top axis). These results show clearly that whereas performance metrics in the system without Emergency routing have large fluctuations as failures are added up, the system with emergency routing shows a very stable operation with small variations in max latency as we approach large number of failures. However, while the emergency routing mechanism is shown to be excellent when dealing with failures, it has been found more recently to be one of the contributors of the system vulnerability to congestion. Fig. 4 shows the network throughput as measured with and without the mechanism activated. It is clear that activating this mechanism may be counterproductive in scenarios of congestion (after about 0.1 load) because accepted load is reduced when the emergency routing is used. Although congestion is not expected to be a major problem in SpiNNaker given the small packet size and the traffic locality, the emergency routing mechanism is normally deactivated unless failures are detected in order to avoid network reaching saturation prematurely.

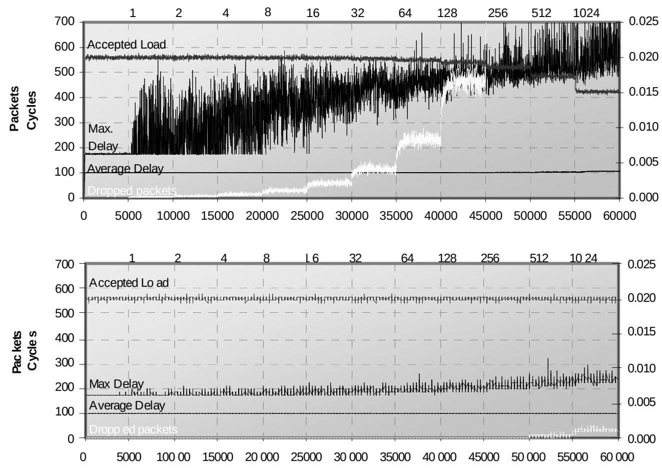


Fig. 3 Temporal evolution of the systems under uniform traffic at a given load of 0.02 packets/node/cycle. Without emergency routing (top) and with emergency routing (bottom). *Adapted from [2].*

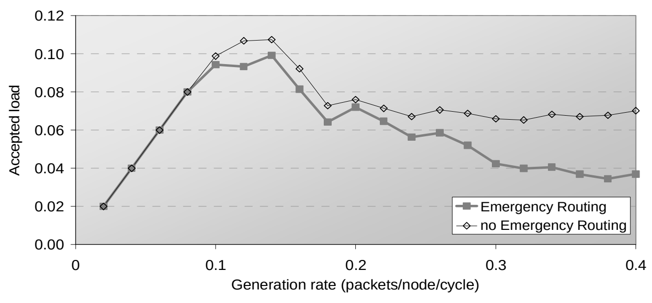


Fig. 4 Effects of emergency routing mechanism on network throughput.

C. *Effects of traffic burstiness*

SpiNNaker features a very distinctive execution model which generates traffic with some characteristics which are dissimilar from the typical workloads that can be found in other parallel computers such as datacentres or HPC systems. For this reason we later investigated the effects that these characteristics (traffic burstiness, locality and causality) had on the network [3]. The most important of these characteristics is, without any doubt, that the event-driven nature of the application tends to generate traffic in bursts because all the neural activity is checked at the beginning of each simulation step (typically with a resolution of 1 ms). Fig. 5 shows the effect that increasing the burstiness of the traffic – up to 50% of the traffic being in the form of 10-packet bursts – has on the packet dropped ratio of the system. It can be seen that the proportion of bursts barely affects the generation rate when the network starts dropping packets, just about a 5% earlier when half of the traffic is bursty. Note that this experiment was carried out in an extremely pessimistic configuration in which the traffic was mostly non-local and the injection rates in which packets are first dropped are about 4 times higher than the typical operation of the system. These results show that SpiNNaker will be able to perform in a stable manner when dealing with the anticipated network workloads.

D. *Multicast Routing*

With the relentless increase in the size of the available systems the use of SpiNNaker's multicast infrastructure is becoming increasingly important [4, 5] as route generation has

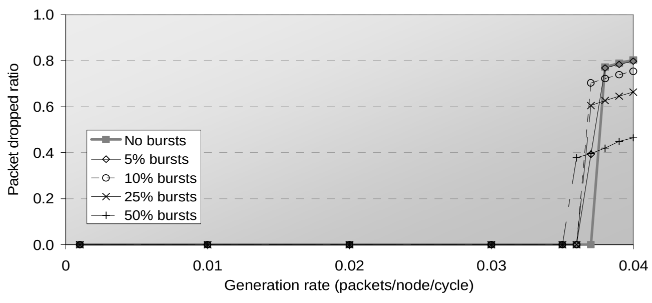
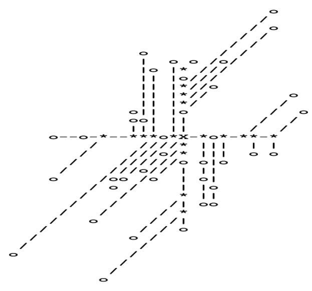
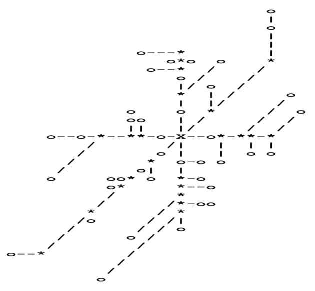
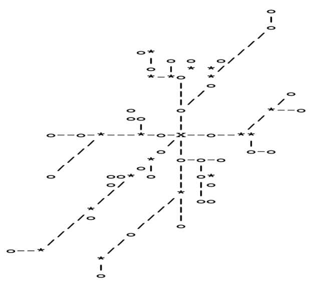


Fig. 5 Effects of burst in the packet dropped ratio. Bursts of 10 packets, *Adapted from [3].*


a) Dimension Order Routing (DOR)


b) Longest Dimension First Routing (LDFR)


c) Enhanced Shortest Path Routing (ESPR)

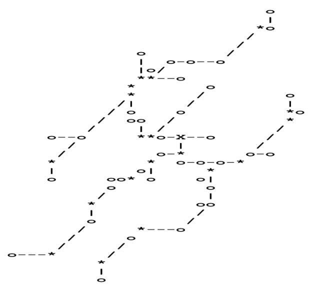
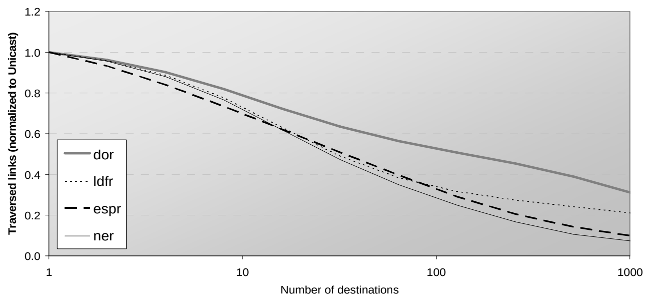

d) Neighbour Exploring Routing (NER)

Fig. 6. Example of multicast trees for each algorithm. Legend: ‘x’ source, ‘o’ destinations, ‘*’ entry in routing table, ‘-’ ‘|’ ‘/’ default route. *From [5]*

a great impact on the performance and efficiency of the network. In this line, we have proposed (Fig. 6.) a number of multicast routing algorithms providing alternatives to reduce the different aspects of the network which can limit scalability. DOR (Fig. 6a), one of the most frequently used algorithms for mesh-like topologies, is shown to be the worst alternative as it requires the highest network resources. LDFR (Fig. 6b) is a well-rounded solution as the multicast routes can be generated very quickly while keeping resource requirements low and balanced. ESPR (Fig. 6c) searches for connections using always shortest paths. NER (Fig. 6d) searches around without requiring shortest path. Both of them can reduce network use but they generate the routes slower than LDFR and so they may be relegated to those cases in which any of the scarce resources of the network limits system scalability.Fig. 7 shows the required network bandwidth and route generation time as the number of destinations is increased up to one thousand. These results were captured assuming the destinations are uniformly distributed; again, worst-case with little locality. First, we can see how the aggregated network bandwidth required to deliver a packet to each destination is reduced considerably when compared to a unicast alternative. Indeed, unicast can require over one order of magnitude more aggregated bandwidth than the best multicast alternative and 2-3× in most of the cases. NER is shown to require the lower aggregated bandwidth in most of the cases, only ESPR is better in the cases in which the number of destinations is reduced. Looking at the route generation time we can see how the search done by ESPR and NER burdens their generation time except in these cases in which there are many destinations so finding connections is likely and has a positive return as once a connection is made the rest of the route can be safely skipped.



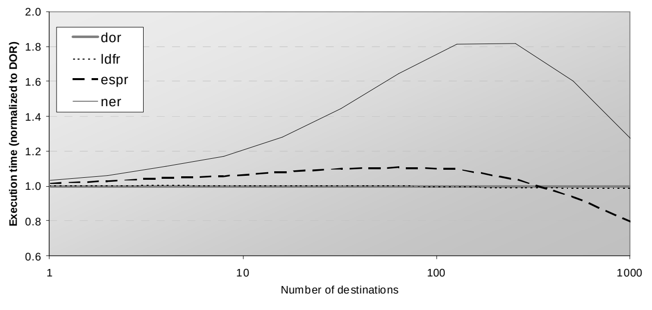


Fig. 7 Evaluation of the different routing algorithms with uniformly distributed destinations. Network utilization (top) and route generation time (bottom). *Adapted from [5].*

IV. Conclusions

In this paper we have highlighted the most important features and strengths of SpiNNaker interconnect. Our analysis shows that SpiNNaker’s low-spec, custom-made multicast router will be able to sustain the demand and operate in a stable manner in most foreseeable situations as regards traffic load, burstiness and network failures. Only in unlikely scenarios, with extremely low locality and pathological allocation of neurons, may the network become a bottleneck. Further we justify the router architecture by showing that its multicast nature substantially reduces required bandwidth when compared to unicast.

Acknowledgment

The design and construction of the SpiNNaker machine was supported by EPSRC (the UK Engineering and Physical Sciences Research Council) under grants EP/D07908X/1 and EP/G015740/1. Ongoing development is supported by the EU ICT Flagship Human Brain Project (FP7-604102), and our own exploration of the capabilities of the machine is supported by the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 320689. Dr Luján holds a Royal Society University Research Fellowship. Dr. Navaridas is supported by EPSRC, through Grant EP/K015699/1 and by the EU H2020 through Grant agreement No. 671553.

References

[1] J Navaridas, et al. “Understanding the Interconnection Network of SpiNNaker”. International Conference on Supercomputing (ICS’09), June 8 to 12, 2009, New York, USA.

[2] J Navaridas et al. “SpiNNaker: Fault tolerance in a power- and area-constrained large-scale neuromimetic architecture”, Parallel Computing, 39(11), Nov 2013, pp. 693-708, DOI: 10.1016/j.parco.2013.09.001.

[3] J Navaridas, et al. “SpiNNaker: Impact of Traffic Locality, Causality and Burstiness on the Performance of the Interconnection Network”. Computing Frontiers (CF’10), 2010,

[4] J Navaridas, et al. “Analytical Assessment of the Suitability of Multicast Communications for the SpiNNaker Neuromimetic System”. IEEE International Conference on High Performance Computing and Communications (HPCC 2012), June 25-27, 2012, Liverpool, UK.

[5] J Navaridas, et al. “SpiNNaker: Enhanced Multicast Routing”. Parallel Computing 45, Jun 2015, pp. 49-66 DOI: 10.1016/j.parco.2015.01.002

Energy versus performance on low power processors for HPC applications

Enrico Calore, Alessandro Gabbana, Sebastiano Fabio Schifano and Raffaele Tripiccione
INFN and Università degli Studi di Ferrara, Ferrara, Italy

Abstract—**Energy efficiency is becoming more and more important in the HPC field; high-end processors are quickly evolving towards more advanced power-saving and power-monitoring technologies. At the same time, low-power processors, designed for the mobile market, attract interest in the HPC area for their increasing computing capabilities, competitive pricing and low power consumption. In this work we compare energy and computing performances for different low-power platforms. As a benchmark we use an HPC Lattice Boltzmann application, instrumented with several software and hardware tools to monitor its power consumption; our analysis uses time-accurate measurements performed on all systems by a simple custom-developed current monitor. We consider the ARM Cortex-A15 CPU and the GK20A GPU of a NVIDIA Tegra k1, as well as an ARM Cortex-A53 CPU. We discuss several energy and performance metrics, evaluating the different energy-performance tradeoffs available on the tested architectures.**

I. Introduction

The computational performances of current HPC systems are increasingly limited by power consumption, and this is only expected to worsen in the foreseeable future. This is also relevant from the point of view of operating costs; indeed, large computing facilities are considering the option to charge not only running time but also energy dissipation. In response to these problems, high-end processors are quickly introducing more advanced power-saving and power-monitoring technologies [1]. On the other hand, low-power processors, designed for the mobile market, are gaining interest as it appears that they may eventually fill (or at least reduce) their performance gap with high-end processors and still keep a competitive edge on costs, thanks to the economies of scale associated to large production volumes of mobile devices [2], [3]. The power consumption problem is starting to be approached also from the software point of view, with developers focusing not only on performance, but also learning to optimize codes to achieve acceptable trade-offs between performance and energy efficiency [4], [5].

In this paper we address one facet of these issues. We analyze in details, using accurate measurements, the role played by hardware factors and by some software aspects in the energy-performance landscape of real-life HPC applications. Our application benchmark is a Lattice Boltzmann code widely used in CFD. As a hardware testbed we consider a low-power Tegra K1 SoC (System on a Chip), embedding a multi-core ARMv7 CPU and a GPU, and a 96boards HiKey system embedding a 64-bit multi-core ARMv8 CPU. We use three versions of our code, optimized for the two ARM CPUs and the NVIDIA GPU, with different configurations and compilation options. We then measure energy consumption and performance of the computationally intensive kernels in our code, using several clock frequency combinations, building a large database of measured data. We then analyze these results, also guided by a simple but effective model of the energy behavior of our test systems.

II. The hardware testbed

Our hardware setup is based on two development boards and a custom current monitoring system described in [6], able to acquire and store current values out-of-band [1].

A. Jetson TK1 Development Board

The Tegra K1 SoC, hosted on the Jetson TK1 board, has a CPU and a GPU on a single chip; the CPU is a NVIDIA "4-Plus-1", a 2.32GHz ARM quad-core Cortex-A15 and a low-power shadow core; the GPU is a NVIDIA Kepler GK20a with 192 CUDA cores (with 3.2 compute capability). Both units access a shared DDR3L 2 GB memory bank on a 64-bit bus running at up to 933 MHz. This system has several energy-saving features: cores in the CPU can be independently activated and the frequency of the CPU, GPU and memory system can be individually selected in a wide frequency range (CPU: $204 \cdots 2320.5$ MHz in 20 steps; GPU: $72 \cdots 852$ MHz in 15 steps; Memory: $12.75 \cdots 924$ MHz in 12 steps).

B. 96 Boards HiKey

The HiKey board is based around the HiSilicon Kirin 620 eight-core ARM Cortex-A53 64-bit SoC, running at 1.2GHz and embeds 1GB of 800MHz LPDDR3 DRAM on-board modules. The board has the standard 96Boards credit-card form factor and is powered by an 8-18V DC 2A power supply. Also this system has several energy-saving features: cores in the CPU can be independently activated and the frequency of the CPU can be selected between 208MHz and 1.2GHz, in 5 steps.

C. Power monitoring system

Both boards are powered by a 12V source, so their power consumption can be easily derived by current measurements. We have developed a simple system able to measure the current flowing into the boards with very good accuracy and time resolution (≈ 1 msec) and able to correlate measurements with the execution of specific software kernels. The setup uses an analog current to voltage converter (using a LTS 25-NP current transducer) and an Arduino UNO board; the latter

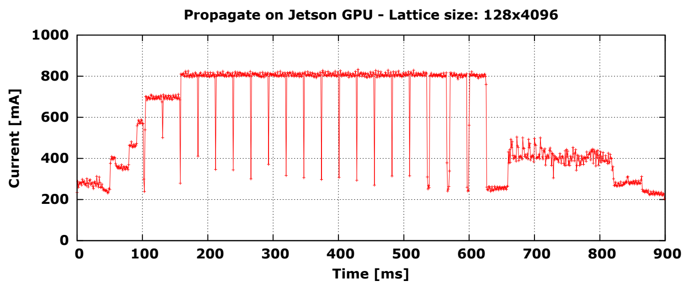


Fig. 1: Raw data collected by the current monitoring system as the Jetson GPU runs 20 iterations of a CUDA kernel. Current increases during the first iterations as the performance governor (in default mode) increases the clock frequency.

uses its embedded 10-bit ADC to digitize current readings and stores them in its memory. We synchronize the Arduino UNO and the instrumented application through a simple serial protocol built over an USB connection. With this setup, a generic application running on the boards only needs to trigger the Arduino UNO to start acquisition immediately before launching the kernel function to be profiled. After the function under test completes, acquired data is downloaded from the Arduino UNO memory, so it can be stored and analyzed offline. The monitor acquires current samples with 1 ms granularity; for increased accuracy, multiple consecutive readings (e.g. 5 in our case) are performed and averaged. This setup is able to correlate current measurements with specific application events with an accuracy of a few milliseconds, while minimally disrupting the execution of the kernel function to profile. Fig. 1 shows a typical raw current measurement of a CUDA kernel running 20 times on the Jetson GPU; it highlights the good time resolution and accuracy. Note the increasing values of the measured current during the first iterations, as the performance governor, left in in default mode in this example, scales the clock frequency.

Starting from N current samples $i[n]$ for the time interval T_S corresponding to the execution of a given kernel, different power metrics can be computed. The instantaneous power is $p[n] = V \times i[n]$ and the average power $P_{\text{avg}} = \frac{1}{N}\sum_{n=0}^{N-1} p[n]$. Another popular metric, the so-called *energy-to-solution* is defined as $E_S = T_S \times P_{avg}$.

III. The Application Benchmark

For our tests, we use a production-grade computational fluid-dynamics code based on the Lattice Boltzmann (LB) method. LB methods [7] – discrete in position and momentum spaces – are based on the synthetic dynamics of *populations* sitting at the sites of a discrete lattice. At each time step, populations hop from lattice-site to lattice-site and then *collide* among one another, that is, they mix and their values change accordingly. We consider a state-of-the-art LB code reproducing the thermo-hydrodynamical evolution in two dimensions of a fluid with the equation of state of a perfect gas ($p = \rho T$) [8], [9]. This model, extensively used for large scale simulations of convective turbulence (see e.g., [10], [11]), uses 37 populations (a D2Q37 method, in standard LB jargon). Populations ($f_l(\boldsymbol{x},t)$ $l = 1\cdots 37$), defined on a discrete and regular lattice and each having a lattice velocity $\boldsymbol{c}_l$, explicitly evolve in (discrete) time:

$$f_l(\boldsymbol{x},t+\Delta t) = f_l(\boldsymbol{x}-\boldsymbol{c}_l\Delta t,t) - \frac{\Delta t}{\tau}\left(f_l(\boldsymbol{x}-\boldsymbol{c}_l\Delta t,t) - f_l^{(eq)}\right) \tag{1}$$

Macroscopic variables, density ρ, velocity $\boldsymbol{u}$ and temperature T are defined in terms of the $f_l(x,t)$ and of the $\boldsymbol{c}_l$s (e.g., $\rho = \sum_l f_l$, $\rho\boldsymbol{u} = \sum_l \boldsymbol{c}_l f_l$) and the equilibrium distributions ($f_l^{(eq)}$) are in turn a function of these quantities. In suitable limiting cases, the evolution of the macroscopic variables obeys the thermo-hydrodynamical equations of motion of the fluid.

An LB simulation starts with an initial assignment of the populations, corresponding to an initial condition on some spatial domain, and iterates Eq. 1 for each point and for as many time-steps as needed. At each iteration two critical kernels are executed (all other routines having a negligible computational cost): i) propagate moves populations across the lattice, collecting at each site all populations that will interact at the next phase (collide). So, propagate moves blocks of memory locations across sparse memory addresses, corresponding to populations of neighbor cells; ii) collide performs all steps needed (per Eq. 1) to update population values at the new time step, using data gathered by propagate. This is the floating point intensive step of the code.

This code – an important HPC application per se – has two specific merits as a benchmark: i) it has a huge degree of easily identified parallelism, that can be exploited by different architectures (e.g., many-core CPUs or GPUs), and ii) one of the two key routines (propagate) is strongly memory-bound while the other (collide) is strongly compute-bound.

IV. Measurements

Our benchmark is based on codes implementing the LB algorithm described in the previous section and exploiting to a large extent the available parallelism. On the GPU we run an optimized CUDA code, developed for large scale CFD simulations on large HPC systems [12], [13]. On the CPUs we run two plain C versions using respectively 32-bit and 64-bit NEON SIMD *intrinsics* exploiting the vector unit of the ARM Cortex-A15 and Cortex-A53. We also use OpenMP for multi-threading within the CPU cores and OpenMPI for future testing purposes on multiple boards.

We have instrumented both critical kernels as described in Sec. II-C, and performed several test runs, monitoring the current profile at all times during the tests, accumulating a large database of measured data. On the software side, we have included runs with different numbers of OpenMP threads (for CPUs) and CUDA block sizes (for the GPU); on the hardware side we have logged data for most combinations of the adjustable clock frequencies, disabling automatic frequency scaling. The C code using NEON *intrinsics*, was run on the Cortex A15 manually forcing the use of the G cluster (i.e. the high performance quad-core). When running on the GPU, the CPU was forced to use the LP cluster (i.e. the

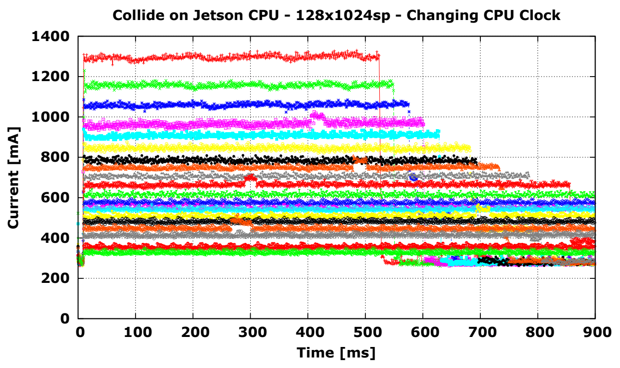


Fig. 2: Current measurements of the Cortex A15 CPU for the collide kernel on a lattice of 128×1024 points with 4 OpenMP threads. Each plot line refers to a different CPU clock frequency, between 204MHz (lowest green line) and 2.3GHz (highest red line). The memory clock stays at its maximum value.

low performance shadow core). We performed all tests in single and double floating point precision, but for the sake of a fair comparison, here we only consider single precision results; indeed, the Cortex-A15 is a 32-bit CPU (lacking double precision vector instructions) and the GK20a GPU does not have double precision floating point units.

As an example of data obtained by the various test runs, in Fig. 2 are shown current readings for the collide kernel for several values of the CPU clock (GPU and memory clocks are fixed); similar results are available for the propagate kernel, for most clock combinations and for all processors.

V. Results and Discussion

We consider *energy-to-solution* (E_S) and *time-to-solution* (T_S) – and the correlations thereof – relevant and interesting handles to explore tradeoffs between potentially conflicting time and energy requirements.

To better highlight the time/energy tradeoff, we plot E_S as a function of T_S, for all processors and for both kernels, see Fig. 3. Interestingly enough, E_S scales approximately linearly with T_S. A crude way to understand this behavior is as follows: as the processor executes a kernel, it consumes power in two ways: i) the power associated to the (constant in time) background current (including the leakage current of the processor and the current drawn by ancillary circuits on the board) and ii) the power associated to the switching activity of all gates of the processor as it transitions across different states while executing the program. The first term implies a constant power rate (P_0), while the second term implies an average energy dissipation CV^2 every time a bit in the state of the processor toggles during execution (V is the processor power supply and C is an average value of the output capacitance of each gate); this model derives directly from early power analyses found in classic books in VLSI design [14], and recently discussed in [15]. We are fully aware that the actual

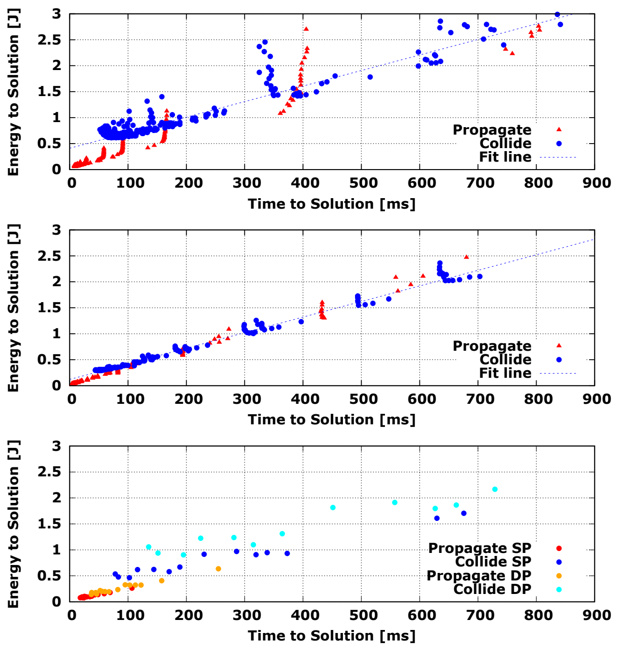


Fig. 3: Measured values of E_S vs. T_S on the Cortex A15 CPU (top), NVIDIA GK20a GPU (middle) and Cortex A53 (bottom), for the collide (blu) and propagate (red) kernels for several clock frequencies. Results of a fit of collide data to Eq. 2 are also shown (top and middle). For the Cortex A53 (bottom), we also include results for the double precision version of the code for collide (cyan) and propagate (orange).

situation is more complex, but we let this simple model guide our further analysis. We fit E_S as a function of T_S as follows:

$$E_S = E_0 + P_0 \times T_S \tag{2}$$

P_0 should be independent of the program under test, while E_0 should depend on the kernel and the processor executing it, as – to first approximation – it counts the number of state transitions that the processor has to go to execute the code, *irrespective* of the frequency at which they happen.

Deviations from the scaling behavior occur as clustered points of roughly equal T_S but different E_S. These abnormal energy costs are associated to a mismatch between the clock frequencies of the processor and the memory system.

Therefore, contrary to intuition, using low clock frequencies is an ineffective way to reduce energy dissipation, since a low clock frequency does not affect E_0 significantly while it increases T_S, causing an higher value for E_S. This phenomena is known as the "clock race to idle" [16] and is clearly visible in Fig. 3.

Summing up, running codes at very low frequencies is almost useless for all the tested processors (at least for our benchmarks); it would probably be more useful to add flexible (and fast) options to remove power from parts of the processor; efficient ways to save the state of the subsystem before shut down would be useful.

To first approximation, the best energy saving options for this class of processors, correspond to running the system at a frequency close to the highest possible value, and accurately tuning the match between the clock frequencies of the various subsystems. In the high frequency corner (lower-left in all panels of Fig. 3), an interesting tradeoff between energy- and time-to-solution can be looked for.

Taking into account these tradeoffs regions, different clock frequencies could be selected from a sub-set of all the available ones in order to tune the energy consumption with respect to an affordable worsening of the performances.

Different points could be chosen for the different architectures, but in order to perform a fair comparison one has to adopt a single metric [5]. There is no single well-defined criterion here, but one of the most used is the EDP (Energy Delay Product) cost model [17], which is the product of E_S and T_S. EDP values for the compute intensive (single precision) collide kernel in our application are shown in Tab. I, where for each processor we have selected the best corresponding figure.

Processor	E_S [J] per iter.	T_S [ms] per iter.	EDP [J s]
GK20A	0.30	42	0.013
ARM A15	0.67	58	0.039
ARM A53	0.52	77	0.040

TABLE I: Best EDP values, with corresponding *energy-to-solution* and *time-to-solution*, for the tested processors, running the (SP) collide kernel.

According to this metric and for the LBM code we have considered, the GPU is more efficient than the CPUs by a factor of three. Interestingly, we also see that the two different ARM CPUs have approximately the same EDP value.

VI. Conclusions and future works

Our analysis shows that a limited but not negligible optimization of the energy consumption is possible by carefully matching the values of clock frequencies to the characteristics of the code being executed.

Comparison among different architectures is not simple because of the large number of parameters involved and the lack of a well-defined metrics. Drawing a preliminary conclusion, the EDP values of Tab. I may provide a "general purpose" assessment of energy vs. performance merits across different processors.

For the future, we plan to extend our analysis in several ways: i) improving the current monitoring system, to have more information available and to measure how the various parts of the system contribute to energy dissipation; ii) applying our analysis to more advanced low-power systems supporting double precision floating point operations, such as the Jetson X1 board and high-end system like the Cavium ThunderX ARM Processors, also comparing with mainstream HPC accelerators, such as NVIDIA K80 GPUs; iii) considering not only hardware-based tuning, but also software options toward energy saving.

Acknowledgements

This work has been done in the framework of the COSA, COKA and Suma projects, supported by INFN. The Jetson TK1 and HiKey boards were awarded to E.C. for the best paper at UCHPC 2014 and 2015.

References

[1] D. Hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 2013, pp. 194–204.

[2] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez, "Tibidabo: Making the case for an ARM-based HPC system," *Future Generation Computer Systems*, vol. 36, no. 0, pp. 322 – 334, 2014.

[3] J. Choi, M. Dukhan, X. Liu, and R. Vuduc, "Algorithmic time, energy, and power on candidate HPC compute building blocks," in *Parallel and Distributed Processing Symposium, IEEE 28th Int.*, 2014, pp. 447–457.

[4] J. Coplin and M. Burtscher, "Effects of source-code optimizations on GPU performance and energy consumption," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU 2015, 2015, pp. 48–58.

[5] M. Wittmann, G. Hager, T. Zeiser, J. Treibig, and G. Wellein, "Chip-level and multi-node analysis of energy-optimized lattice Boltzmann CFD simulations," *Concurrency and Computation: Practice and Experience*, 2015.

[6] E. Calore, S. F. Schifano, and R. Tripiccione, "Energy-Performance Tradeoffs for HPC Applications on Low Power Processors," in *Euro-Par 2015: Parallel Processing Workshops*, ser. LNCS, 2015, vol. 9523, pp. 737–748.

[7] S. Succi, *The Lattice-Boltzmann Equation*. Oxford university press, Oxford, 2001.

[8] M. Sbragaglia, R. Benzi, L. Biferale, H. Chen, X. Shan, and S. Succi, "Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria," *Journal of Fluid Mechanics*, vol. 628, pp. 299–309, 2009.

[9] A. Scagliarini, L. Biferale, M. Sbragaglia, K. Sugiyama, and F. Toschi, "Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh–Taylor systems," *Physics of Fluids (1994-present)*, vol. 22, no. 5, p. 055101, 2010.

[10] L. Biferale, F. Mantovani, M. Sbragaglia, A. Scagliarini, F. Toschi, and R. Tripiccione, "Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions," *Physical Review E*, vol. 84, no. 1, p. 016305, 2011.

[11] ——, "Reactive Rayleigh-Taylor systems: Front propagation and non-stationarity," *EPL*, vol. 94, no. 5, p. 54004, 2011.

[12] J. Kraus, M. Pivanti, S. F. Schifano, R. Tripiccione, and M. Zanella, "Benchmarking GPUs with a parallel Lattice-Boltzmann code," in *Computer Architecture and High Performance Computing (SBAC-PAD), 25th Int. Symposium on*. IEEE, 2013, pp. 160–167.

[13] E. Calore, S. Schifano, and R. Tripiccione, "On Portability, Performance and Scalability of an MPI OpenCL Lattice Boltzmann Code," in *Euro-Par 2014: Parallel Processing Workshops*, ser. LNCS, 2014, vol. 8806, pp. 438–449.

[14] C. Mead and L. Conway, *Introduction to VLSI systems*. Addison-Wesley Reading, MA, 1980, vol. 802.

[15] N. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec 2003.

[16] G. Hager, J. Treibig, J. Habich, and G. Wellein, "Exploring performance and power properties of modern multi-core chips via simple machine models," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 2, pp. 189–210, 2014.

[17] J. H. Laros III, K. Pedretti, S. M. Kelly, W. Shu, K. Ferreira, J. Vandyke, and C. Vaughan, *Energy-Efficient High Performance Computing: Measurement and Tuning*. Springer London, 2013, ch. Energy Delay Product, pp. 51–55.

Embedded supercomputing at NVIDIA

Alex Ramirez
NVIDIA

***Abstract*— In 2007, NIDIA introduced its CUDA parallel programming language: an extension to C, C++, and Fortran that enables general purpose parallel computing on GPU accelerators. It has always been the plan for NVIDIA to enable "CUDA anywhere".**

Following that plan, NVIDIA introduced the Tegra K1 in 2014. Its first embedded multicore featuring 192 CUDA-capable cores based on Kepler microarchiteture, delivering over 300 GFLOPS (single precision). From there, NVIDIA produced the Tegra X1, upgrading the GPU to 256 Maxwell cores, and delivering 512 GFLOPS. The Tegra X1 was built into the dual-socket Drive PX embedded supercomputer, enabling development of smart self-driving cars. The NVIDIA DrivePX2 system features dual Tegra sockets coupled with two discrete Pascal GPU accelerators to deliver over 24 TFLOPS of performance in an embedded supercomputer meant to be under the hood of your next self-driving car.

In this talk I will describe the architecture and capabilities of these embedded supercomputers, as well as highlight some upcoming technologies (NVlink, UVM) that will enable integration of larger systems built from these embedded components.

Efficient HPC: Waste Not? Want Not?

Michele Weiland
ADEPT Project,
Edinburgh Parallel Computing Centre,
UK

***Abstract*— The focus of the HPC community is firmly on reaching its next big goal: the Exascale. There are many technical hurdles that need to be overcome to reach this goal in the next five to ten years, but there is an underlying theme of efficiency. HPC applications often only use a few percent of the peak performance a system can offer, wasting vast amounts of resources that could be exploited to achieve increased science throughput. The efficiency of Exascale systems in terms of power is also of great concern – we will need to achieve 50 GFlop/s per Watt if we want to stay within a 20MW power envelope. In this talk, I will present the work of two projects: Adept, which investigates and models power/energy expenditure in parallel systems; and NEXTGenIO, which exploits Intel 3D XPoint technology to eliminate the I/O bottleneck in large-scale parallel systems.**

Whole Systems Energy Transparency

Kerstin Eder
Department of Computer Science
University of Bristol

***Abstract*— Energy efficiency is now a major (if not the major) concern in electronic systems engineering. While hardware can be designed to save a modest amount of energy, the potential for savings are far greater at the higher levels of abstraction in the system stack. The greatest savings are expected from energy consumption-aware software. This talk emphasizes the importance of energy transparency from hardware to software as a foundation for energy-aware system design. Energy transparency enables a deeper understanding of how algorithms and coding impact on the energy consumption of a computation when executed on hardware. It is a key prerequisite for informed design space exploration and helps system designers to find the optimal tradeoff between performance, accuracy and energy consumption of a computation. Promoting energy efficiency to a first class software design goal is therefore an urgent research challenge. In this talk I will outline the first steps towards giving "more power" to software developers. We will cover energy monitoring of software, energy modelling at different abstraction levels, including insights into how data affects the energy consumption of a computation, and static analysis techniques for energy consumption estimation.**

Code_Saturne on POWER8 clusters: First Investigations

Charles Moulinec
and Vendel Szeremi
and David R. Emerson
STFC Daresbury Laboratory, UK

Yvan Fournier
EDF R&D, FR

Pascal Vezolle
and Ludovic Enault
IBM Montpellier, FR

Benedikt Anlauf
and Markus Bühler
IBM Böblingen, GE

***Abstract*—Code_Saturne is a Computational Fluid Dynamics (CFD) multi-physics software parallelised using MPI and some OpenMP. It has demonstrated extreme scalability on Argonne IBM Blue Gene/Q up to 3.0 million threads and on Jülich IBM Blue Gene/Q up to 1.8 million threads. Recent developments have been carried out to link the code to the PETSc library, in order to access more options to solve linear systems, but also to benefit from their latest developments on GPUs. Results showing Code_Saturne's performance for a canonical flow (lid-driven cavity) are presented using MPI only first, then hybrid MPI-OpenMP results are shown, before presenting some performance using PETSc on CPUs and on CPUs & GPUs.**

I. Introduction

Code_Saturne [1][2][3] is a Computational Fluid Dynamics (CFD) multi-physics software primarily developed by EDF R&D. The code is open-source since 2007 under GNU GPL and is one of the only 2 CFD codes of the PRACE Unified European Applications Benchmark Suite [4]. The code is parallelised using MPI and some OpenMP. It has demonstrated extreme scalability on the whole Jülich IBM Blue Gene/Q up to 1.8 million threads, joining then the JUQUEEN High-Q Club [5]. Recent developments have been carried out to link *Code_Saturne* to the PETSc library, in order to get access to more options to solve linear systems, but also to benefit from their latest developments on GPUs. Very first results are presented in this work.

II. Brief description of the IBM POWER8 archictecture

The target machines for this paper are IBM POWER8[1] clusters [6]. Two types of nodes are considered, S822LC and S824L and a short description is given in the following subsections.

A. S822LC node

A node is made of 2 processors, each of them having 10 cores running at about 2.92GHz. It also has 8 logical cores, allowing to use Symmetric Multi-Threading (SMT). Four on-chip memory controlers are available (SCM). A node has got about 256 GB RAM available.
Two NVIDIA K80 GPUs are attached to each node. Each of them has got 2 GPU GK210 with 12 GB memory and 2496 stream processors. These 2 K80 cards are similar to 4 K40 cards, in terms of computing capability but have similar performance as 2 K40 cards only, in terms of CPU-GPU data transfers.

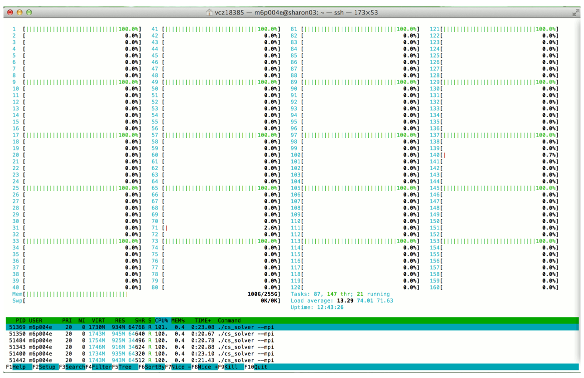
[1]Trademarks of IBM in USA and/or other countries

B. S824L node

A node is made of 2 processors, each of them having 12 cores running at about 3.0GHz. It also has 8 logical cores, allowing to use SMT. Eight on-chip memory controlers are available (DCM). A node has got about 256 GB RAM available.
Two NVIDIA K40 GPUs are attached to each node. Each of them has got 12 GB memory and 2880 stream processors.

C. CPU Binding

To efficiently use POWER8 nodes, CPU binding is required and the distribution of the cores is different from the one for x86 nodes, where the cores are consecutively numbered. The following examples show the core distribution when a single POWER8 node is used. For instance, a 20 MPI tasks simulation would run best on cores 0, 8, 16, ... (see Fig. 1), a 40 MPI tasks simulation on cores 0, 4, 8, 12, 16, ... (see Fig. 2), a 80 MPI tasks simulation on cores 0, 2, 4, 6, 8, 10, 12, 14, 16, ... (see Fig. 3) and a 160 MPI tasks simulations on cores 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ... (see Fig. 4).

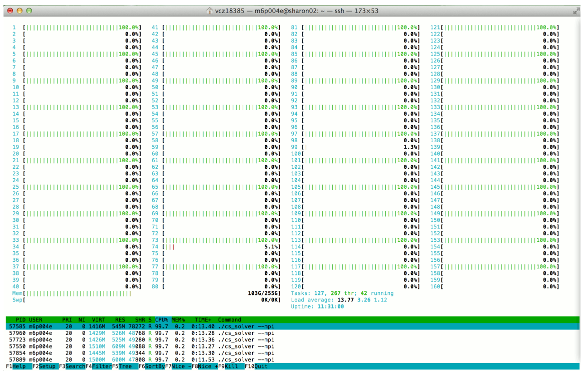


```
vcz18385 — m6p004e@sharon03: ~ — ssh — 173×53

1  [|||||||||||||||||||||||||||100.0%]   41 [|||||||||||||||||||||||||||100.0%]   81 [|||||||||||||||||||||||||||100.0%]   121[|||||||||||||||||||||||||||100.0%]
2  [                             0.0%]   42 [                             0.0%]   82 [                             0.0%]   122[                             0.0%]
3  [                             0.0%]   43 [                             0.0%]   83 [                             0.0%]   123[                             0.0%]
4  [                             0.0%]   44 [                             0.0%]   84 [                             0.0%]   124[                             0.0%]
5  [                             0.0%]   45 [                             0.0%]   85 [                             0.0%]   125[                             0.0%]
6  [                             0.0%]   46 [                             0.0%]   86 [                             0.0%]   126[                             0.0%]
7  [                             0.0%]   47 [                             0.0%]   87 [                             0.0%]   127[                             0.0%]
8  [                             0.0%]   48 [                             0.0%]   88 [                             0.0%]   128[                             0.0%]
9  [|||||||||||||||||||||||||||100.0%]   49 [|||||||||||||||||||||||||||100.0%]   89 [|||||||||||||||||||||||||||100.0%]   129[|||||||||||||||||||||||||||100.0%]
10 [                             0.0%]   50 [                             0.0%]   90 [                             0.0%]   130[                             0.0%]
11 [                             0.0%]   51 [                             0.0%]   91 [                             0.0%]   131[                             0.0%]
12 [                             0.0%]   52 [                             0.0%]   92 [                             0.0%]   132[                             0.0%]
13 [                             0.0%]   53 [                             0.0%]   93 [                             0.0%]   133[                             0.0%]
14 [                             0.0%]   54 [                             0.0%]   94 [                             0.0%]   134[                             0.0%]
15 [                             0.0%]   55 [                             0.0%]   95 [                             0.0%]   135[                             0.0%]
16 [                             0.0%]   56 [                             0.0%]   96 [                             0.0%]   136[                             0.0%]
17 [|||||||||||||||||||||||||||100.0%]   57 [|||||||||||||||||||||||||||100.0%]   97 [|||||||||||||||||||||||||||100.0%]   137[|||||||||||||||||||||||||||100.0%]
18 [                             0.0%]   58 [                             0.0%]   98 [                             0.0%]   138[                             0.0%]
19 [                             0.0%]   59 [                             0.0%]   99 [                             0.0%]   139[                             0.0%]
20 [                             0.0%]   60 [                             0.0%]   100[                             0.0%]   140[|                            0.7%]
21 [                             0.0%]   61 [                             0.0%]   101[                             0.0%]   141[                             0.0%]
22 [                             0.0%]   62 [                             0.0%]   102[                             0.0%]   142[                             0.0%]
23 [                             0.0%]   63 [                             0.0%]   103[                             0.0%]   143[                             0.0%]
24 [                             0.0%]   64 [                             0.0%]   104[                             0.0%]   144[                             0.0%]
25 [|||||||||||||||||||||||||||100.0%]   65 [|||||||||||||||||||||||||||100.0%]   105[|||||||||||||||||||||||||||100.0%]   145[|||||||||||||||||||||||||||100.0%]
26 [                             0.0%]   66 [                             0.0%]   106[                             0.0%]   146[                             0.0%]
27 [                             0.0%]   67 [                             0.0%]   107[                             0.0%]   147[                             0.0%]
28 [                             0.0%]   68 [                             0.0%]   108[                             0.0%]   148[                             0.0%]
29 [                             0.0%]   69 [                             0.0%]   109[                             0.0%]   149[                             0.0%]
30 [                             0.0%]   70 [                             0.0%]   110[                             0.0%]   150[                             0.0%]
31 [                             0.0%]   71 [|                            2.6%]   111[                             0.0%]   151[                             0.0%]
32 [                             0.0%]   72 [                             0.0%]   112[                             0.0%]   152[                             0.0%]
33 [|||||||||||||||||||||||||||100.0%]   73 [|||||||||||||||||||||||||||100.0%]   113[|||||||||||||||||||||||||||100.0%]   153[|||||||||||||||||||||||||||100.0%]
34 [                             0.0%]   74 [                             0.0%]   114[                             0.0%]   154[                             0.0%]
35 [                             0.0%]   75 [                             0.0%]   115[                             0.0%]   155[                             0.0%]
36 [                             0.0%]   76 [                             0.0%]   116[                             0.0%]   156[                             0.0%]
37 [                             0.0%]   77 [                             0.0%]   117[                             0.0%]   157[                             0.0%]
38 [                             0.0%]   78 [                             0.0%]   118[                             0.0%]   158[                             0.0%]
39 [                             0.0%]   79 [                             0.0%]   119[                             0.0%]   159[                             0.0%]
40 [                             0.0%]   80 [                             0.0%]   120[                             0.0%]   160[                             0.0%]
Mem[|||||||||||||||||||||||||||||                                    100G/255G]   Tasks: 87, 147 thr; 21 running
Swp[                                                                     0K/0K]   Load average: 13.29 74.01 71.63
                                                                                  Uptime: 12:43:26

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
51369 m6p004e    20   0 1730M  934M 64768 R 101.  0.4  0:23.08 ./cs_solver --mpi
51350 m6p004e    20   0 1743M  945M 64640 R 100.  0.4  0:20.67 ./cs_solver --mpi
51484 m6p004e    20   0 1754M  925M 34496 R 100.  0.4  0:20.78 ./cs_solver --mpi
51343 m6p004e    20   0 1746M  916M 34624 R 100.  0.4  0:20.88 ./cs_solver --mpi
51400 m6p004e    20   0 1734M  935M 64320 R 100.  0.4  0:23.10 ./cs_solver --mpi
51442 m6p004e    20   0 1743M  943M 64512 R 100.  0.4  0:21.43 ./cs_solver --mpi
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Fig. 1. Load of a single node with 20 MPI tasks.

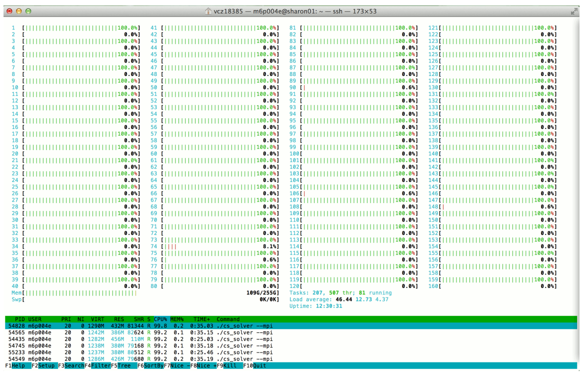


```
vcz18385 — m6p004e@sharon02: ~ — ssh — 173×53

1  [|||||||||||||||||||||||||100.0%]  41 [|||||||||||||||||||||||||100.0%]  81 [|||||||||||||||||||||||||100.0%]  121[|||||||||||||||||||||||||100.0%]
2  [                           0.0%]  42 [                           0.0%]  82 [                           0.0%]  122[                           0.0%]
3  [                           0.0%]  43 [                           0.0%]  83 [                           0.0%]  123[                           0.0%]
4  [                           0.0%]  44 [                           0.0%]  84 [                           0.0%]  124[                           0.0%]
5  [|||||||||||||||||||||||||100.0%]  45 [|||||||||||||||||||||||||100.0%]  85 [|||||||||||||||||||||||||100.0%]  125[|||||||||||||||||||||||||100.0%]
6  [                           0.0%]  46 [                           0.0%]  86 [                           0.0%]  126[                           0.0%]
7  [                           0.0%]  47 [                           0.0%]  87 [                           0.0%]  127[                           0.0%]
8  [                           0.0%]  48 [                           0.0%]  88 [                           0.0%]  128[                           0.0%]
9  [|||||||||||||||||||||||||100.0%]  49 [|||||||||||||||||||||||||100.0%]  89 [|||||||||||||||||||||||||100.0%]  129[|||||||||||||||||||||||||100.0%]
10 [                           0.0%]  50 [                           0.0%]  90 [                           0.0%]  130[                           0.0%]
11 [                           0.0%]  51 [                           0.0%]  91 [                           0.0%]  131[                           0.0%]
12 [                           0.0%]  52 [                           0.0%]  92 [                           0.0%]  132[                           0.0%]
13 [|||||||||||||||||||||||||100.0%]  53 [|||||||||||||||||||||||||100.0%]  93 [|||||||||||||||||||||||||100.0%]  133[|||||||||||||||||||||||||100.0%]
14 [                           0.0%]  54 [                           0.0%]  94 [                           0.0%]  134[                           0.0%]
15 [                           0.0%]  55 [                           0.0%]  95 [                           0.0%]  135[                           0.0%]
16 [                           0.0%]  56 [                           0.0%]  96 [                           0.0%]  136[                           0.0%]
17 [|||||||||||||||||||||||||100.0%]  57 [|||||||||||||||||||||||||100.0%]  97 [|||||||||||||||||||||||||100.0%]  137[|||||||||||||||||||||||||100.0%]
18 [                           0.0%]  58 [                           0.0%]  98 [                           0.0%]  138[                           0.0%]
19 [                           0.0%]  59 [                           0.0%]  99 [|                          1.3%]  139[                           0.0%]
20 [                           0.0%]  60 [                           0.0%]  100[                           0.0%]  140[                           0.0%]
21 [|||||||||||||||||||||||||100.0%]  61 [|||||||||||||||||||||||||100.0%]  101[|||||||||||||||||||||||||100.0%]  141[|||||||||||||||||||||||||100.0%]
22 [                           0.0%]  62 [                           0.0%]  102[                           0.0%]  142[                           0.0%]
23 [                           0.0%]  63 [                           0.0%]  103[                           0.0%]  143[                           0.0%]
24 [                           0.0%]  64 [                           0.0%]  104[                           0.0%]  144[                           0.0%]
25 [|||||||||||||||||||||||||100.0%]  65 [|||||||||||||||||||||||||100.0%]  105[|||||||||||||||||||||||||100.0%]  145[|||||||||||||||||||||||||100.0%]
26 [                           0.0%]  66 [                           0.0%]  106[                           0.0%]  146[                           0.0%]
27 [                           0.0%]  67 [                           0.0%]  107[                           0.0%]  147[                           0.0%]
28 [                           0.0%]  68 [                           0.0%]  108[                           0.0%]  148[                           0.0%]
29 [|||||||||||||||||||||||||100.0%]  69 [|||||||||||||||||||||||||100.0%]  109[|||||||||||||||||||||||||100.0%]  149[|||||||||||||||||||||||||100.0%]
30 [                           0.0%]  70 [                           0.0%]  110[                           0.0%]  150[                           0.0%]
31 [                           0.0%]  71 [                           0.0%]  111[                           0.0%]  151[                           0.0%]
32 [                           0.0%]  72 [                           0.0%]  112[                           0.0%]  152[                           0.0%]
33 [|||||||||||||||||||||||||100.0%]  73 [|||||||||||||||||||||||||100.0%]  113[|||||||||||||||||||||||||100.0%]  153[|||||||||||||||||||||||||100.0%]
34 [                           0.0%]  74 [|||                        5.1%]  114[                           0.0%]  154[                           0.0%]
35 [                           0.0%]  75 [                           0.0%]  115[                           0.0%]  155[                           0.0%]
36 [                           0.0%]  76 [                           0.0%]  116[                           0.0%]  156[                           0.0%]
37 [|||||||||||||||||||||||||100.0%]  77 [|||||||||||||||||||||||||100.0%]  117[|||||||||||||||||||||||||100.0%]  157[|||||||||||||||||||||||||100.0%]
38 [                           0.0%]  78 [                           0.0%]  118[                           0.0%]  158[                           0.0%]
39 [                           0.0%]  79 [                           0.0%]  119[                           0.0%]  159[                           0.0%]
40 [                           0.0%]  80 [                           0.0%]  120[                           0.0%]  160[                           0.0%]
Mem[||||||||||||||||||||||||||||||||                         103G/255G]  Tasks: 127, 267 thr; 42 running
Swp[                                                             0K/0K]  Load average: 13.77 3.26 1.12
                                                                         Uptime: 11:31:00

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
57585 m6p004e    20   0 1416M  545M 78272 R 99.7  0.2  0:13.40 ./cs_solver --mpi
57960 m6p004e    20   0 1429M  526M 48768 R 99.7  0.2  0:13.28 ./cs_solver --mpi
57723 m6p004e    20   0 1426M  525M 49280 R 99.7  0.2  0:13.36 ./cs_solver --mpi
57550 m6p004e    20   0 1510M  609M 49088 R 99.7  0.2  0:13.27 ./cs_solver --mpi
57854 m6p004e    20   0 1445M  539M 49344 R 99.7  0.2  0:13.30 ./cs_solver --mpi
57889 m6p004e    20   0 1500M  600M 47808 R 99.7  0.2  0:11.53 ./cs_solver --mpi
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Fig. 2. Load of a single node with 40 MPI tasks.

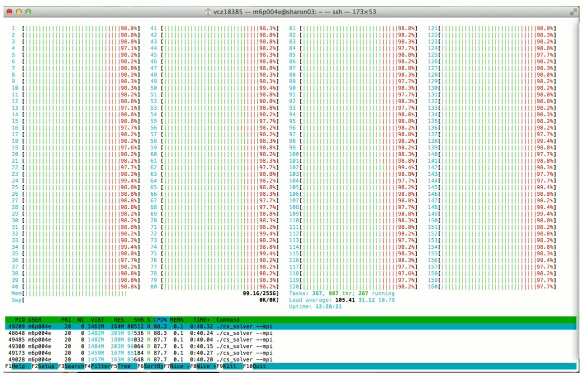


```
vcz18385 — m6p004e@sharon01: ~ — ssh — 173×53

1  [|||||||||||||||||||||||||100.0%]  41 [|||||||||||||||||||||||||100.0%]  81 [|||||||||||||||||||||||||100.0%]  121[|||||||||||||||||||||||||100.0%]
2  [                           0.0%]  42 [                           0.0%]  82 [                           0.0%]  122[                           0.0%]
3  [|||||||||||||||||||||||||100.0%]  43 [|||||||||||||||||||||||||100.0%]  83 [|||||||||||||||||||||||||100.0%]  123[|||||||||||||||||||||||||100.0%]
4  [                           0.0%]  44 [                           0.0%]  84 [                           0.0%]  124[                           0.0%]
5  [|||||||||||||||||||||||||100.0%]  45 [|||||||||||||||||||||||||100.0%]  85 [|||||||||||||||||||||||||100.0%]  125[|||||||||||||||||||||||||100.0%]
6  [                           0.0%]  46 [                           0.0%]  86 [                           0.0%]  126[                           0.0%]
7  [|||||||||||||||||||||||||100.0%]  47 [|||||||||||||||||||||||||100.0%]  87 [|||||||||||||||||||||||||100.0%]  127[|||||||||||||||||||||||||100.0%]
8  [                           0.0%]  48 [                           0.0%]  88 [                           0.0%]  128[                           0.0%]
9  [|||||||||||||||||||||||||100.0%]  49 [|||||||||||||||||||||||||100.0%]  89 [|||||||||||||||||||||||||100.0%]  129[|||||||||||||||||||||||||100.0%]
10 [                           0.0%]  50 [                           0.0%]  90 [|                          0.6%]  130[                           0.0%]
11 [|||||||||||||||||||||||||100.0%]  51 [|||||||||||||||||||||||||100.0%]  91 [|||||||||||||||||||||||||100.0%]  131[|||||||||||||||||||||||||100.0%]
12 [                           0.0%]  52 [                           0.0%]  92 [                           0.0%]  132[                           0.0%]
13 [|||||||||||||||||||||||||100.0%]  53 [|||||||||||||||||||||||||100.0%]  93 [|||||||||||||||||||||||||100.0%]  133[|||||||||||||||||||||||||100.0%]
14 [                           0.0%]  54 [                           0.0%]  94 [                           0.0%]  134[                           0.0%]
15 [|||||||||||||||||||||||||100.0%]  55 [|||||||||||||||||||||||||100.0%]  95 [|||||||||||||||||||||||||100.0%]  135[|||||||||||||||||||||||||100.0%]
16 [                           0.0%]  56 [                           0.0%]  96 [                           0.0%]  136[                           0.0%]
17 [|||||||||||||||||||||||||100.0%]  57 [|||||||||||||||||||||||||100.0%]  97 [|||||||||||||||||||||||||100.0%]  137[|||||||||||||||||||||||||100.0%]
18 [                           0.0%]  58 [                           0.0%]  98 [                           0.0%]  138[                           0.0%]
19 [|||||||||||||||||||||||||100.0%]  59 [|||||||||||||||||||||||||100.0%]  99 [|||||||||||||||||||||||||100.0%]  139[|||||||||||||||||||||||||100.0%]
20 [                           0.0%]  60 [                           0.0%]  100[                           0.0%]  140[                           0.0%]
21 [|||||||||||||||||||||||||100.0%]  61 [|||||||||||||||||||||||||100.0%]  101[|||||||||||||||||||||||||100.0%]  141[|||||||||||||||||||||||||100.0%]
22 [                           0.0%]  62 [                           0.0%]  102[                           0.0%]  142[                           0.0%]
23 [|||||||||||||||||||||||||100.0%]  63 [|||||||||||||||||||||||||100.0%]  103[|||||||||||||||||||||||||100.0%]  143[|||||||||||||||||||||||||100.0%]
24 [                           0.0%]  64 [                           0.0%]  104[                           0.0%]  144[                           0.0%]
25 [|||||||||||||||||||||||||100.0%]  65 [|||||||||||||||||||||||||100.0%]  105[|||||||||||||||||||||||||100.0%]  145[|||||||||||||||||||||||||100.0%]
26 [                           0.0%]  66 [                           0.0%]  106[|                          0.6%]  146[                           0.0%]
27 [|||||||||||||||||||||||||100.0%]  67 [|||||||||||||||||||||||||100.0%]  107[|||||||||||||||||||||||||100.0%]  147[|||||||||||||||||||||||||100.0%]
28 [                           0.0%]  68 [                           0.0%]  108[                           0.0%]  148[|                          0.6%]
29 [|||||||||||||||||||||||||100.0%]  69 [|||||||||||||||||||||||||100.0%]  109[|||||||||||||||||||||||||100.0%]  149[|||||||||||||||||||||||||100.0%]
30 [                           0.0%]  70 [                           0.0%]  110[                           0.0%]  150[                           0.0%]
31 [|||||||||||||||||||||||||100.0%]  71 [|||||||||||||||||||||||||100.0%]  111[|||||||||||||||||||||||||100.0%]  151[|||||||||||||||||||||||||100.0%]
32 [                           0.0%]  72 [                           0.0%]  112[                           0.0%]  152[                           0.0%]
33 [|||||||||||||||||||||||||100.0%]  73 [|||||||||||||||||||||||||100.0%]  113[|||||||||||||||||||||||||100.0%]  153[|||||||||||||||||||||||||100.0%]
34 [                           0.0%]  74 [|||                        8.1%]  114[                           0.0%]  154[                           0.0%]
35 [|||||||||||||||||||||||||100.0%]  75 [|||||||||||||||||||||||||100.0%]  115[|||||||||||||||||||||||||100.0%]  155[|||||||||||||||||||||||||100.0%]
36 [                           0.0%]  76 [|                          0.6%]  116[                           0.0%]  156[                           0.0%]
37 [|||||||||||||||||||||||||100.0%]  77 [|||||||||||||||||||||||||100.0%]  117[|||||||||||||||||||||||||100.0%]  157[|||||||||||||||||||||||||100.0%]
38 [                           0.0%]  78 [                           0.0%]  118[                           0.0%]  158[                           0.0%]
39 [|||||||||||||||||||||||||100.0%]  79 [|||||||||||||||||||||||||100.0%]  119[|||||||||||||||||||||||||100.0%]  159[|||||||||||||||||||||||||100.0%]
40 [                           0.0%]  80 [                           0.0%]  120[                           0.0%]  160[                           0.0%]
Mem[||||||||||||||||||||||||||||||||||                       109G/255G]  Tasks: 207, 507 thr; 81 running
Swp[                                                             0K/0K]  Load average: 46.44 12.73 4.37
                                                                         Uptime: 12:30:31

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
54828 m6p004e    20   0 1290M  432M 81344 R 99.8  0.2  0:35.03 ./cs_solver --mpi
54565 m6p004e    20   0 1242M  386M 82624 R 99.2  0.1  0:35.15 ./cs_solver --mpi
54435 m6p004e    20   0 1282M  456M  110M R 99.2  0.2  0:25.03 ./cs_solver --mpi
54745 m6p004e    20   0 1238M  380M 79168 R 99.2  0.1  0:35.18 ./cs_solver --mpi
55233 m6p004e    20   0 1237M  380M 80512 R 99.2  0.1  0:25.46 ./cs_solver --mpi
54549 m6p004e    20   0 1286M  426M 79680 R 99.2  0.2  0:35.19 ./cs_solver --mpi
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Fig. 3. Load of a single node with 80 MPI tasks.



```
vcz18385 — m6p004e@sharon03: ~ — ssh — 173×53

1  [||||||||||||||||||||||||||98.8%]  41 [||||||||||||||||||||||||||98.3%]  81 [||||||||||||||||||||||||||98.8%]  121[||||||||||||||||||||||||||98.8%]
2  [||||||||||||||||||||||||||98.8%]  42 [||||||||||||||||||||||||||98.8%]  82 [||||||||||||||||||||||||||98.2%]  122[||||||||||||||||||||||||||98.3%]
3  [||||||||||||||||||||||||||98.8%]  43 [||||||||||||||||||||||||||98.8%]  83 [||||||||||||||||||||||||||98.3%]  123[||||||||||||||||||||||||||98.2%]
4  [||||||||||||||||||||||||||97.1%]  44 [||||||||||||||||||||||||||98.2%]  84 [||||||||||||||||||||||||||97.7%]  124[||||||||||||||||||||||||||98.8%]
5  [||||||||||||||||||||||||||98.2%]  45 [||||||||||||||||||||||||||98.3%]  85 [||||||||||||||||||||||||||98.8%]  125[||||||||||||||||||||||||||97.7%]
6  [||||||||||||||||||||||||||98.2%]  46 [||||||||||||||||||||||||||98.8%]  86 [||||||||||||||||||||||||||98.2%]  126[||||||||||||||||||||||||||98.2%]
7  [||||||||||||||||||||||||||98.8%]  47 [||||||||||||||||||||||||||98.8%]  87 [||||||||||||||||||||||||||98.8%]  127[||||||||||||||||||||||||||98.3%]
8  [||||||||||||||||||||||||||98.3%]  48 [||||||||||||||||||||||||||98.3%]  88 [||||||||||||||||||||||||||98.3%]  128[||||||||||||||||||||||||||98.8%]
9  [||||||||||||||||||||||||||98.3%]  49 [||||||||||||||||||||||||||98.3%]  89 [||||||||||||||||||||||||||97.7%]  129[||||||||||||||||||||||||||98.2%]
10 [||||||||||||||||||||||||||98.3%]  50 [||||||||||||||||||||||||||99.4%]  90 [||||||||||||||||||||||||||98.3%]  130[||||||||||||||||||||||||||98.2%]
11 [||||||||||||||||||||||||||98.2%]  51 [||||||||||||||||||||||||||98.8%]  91 [||||||||||||||||||||||||||97.7%]  131[||||||||||||||||||||||||||98.8%]
12 [||||||||||||||||||||||||||98.8%]  52 [||||||||||||||||||||||||||98.8%]  92 [||||||||||||||||||||||||||98.3%]  132[||||||||||||||||||||||||||98.8%]
13 [||||||||||||||||||||||||||97.1%]  53 [||||||||||||||||||||||||||98.8%]  93 [||||||||||||||||||||||||||97.7%]  133[||||||||||||||||||||||||||98.2%]
14 [||||||||||||||||||||||||||98.8%]  54 [||||||||||||||||||||||||||98.2%]  94 [||||||||||||||||||||||||||98.8%]  134[||||||||||||||||||||||||||98.3%]
15 [||||||||||||||||||||||||||98.8%]  55 [||||||||||||||||||||||||||97.7%]  95 [||||||||||||||||||||||||||98.8%]  135[||||||||||||||||||||||||||98.2%]
16 [||||||||||||||||||||||||||97.7%]  56 [||||||||||||||||||||||||||98.2%]  96 [||||||||||||||||||||||||||98.2%]  136[||||||||||||||||||||||||||98.2%]
17 [||||||||||||||||||||||||||98.2%]  57 [||||||||||||||||||||||||||98.2%]  97 [||||||||||||||||||||||||||98.8%]  137[||||||||||||||||||||||||||97.7%]
18 [||||||||||||||||||||||||||98.2%]  58 [||||||||||||||||||||||||||98.3%]  98 [||||||||||||||||||||||||||98.2%]  138[||||||||||||||||||||||||||99.4%]
19 [||||||||||||||||||||||||||97.6%]  59 [||||||||||||||||||||||||||98.8%]  99 [||||||||||||||||||||||||||98.2%]  139[||||||||||||||||||||||||||98.8%]
20 [||||||||||||||||||||||||||98.2%]  60 [||||||||||||||||||||||||||98.2%]  100[||||||||||||||||||||||||||98.3%]  140[||||||||||||||||||||||||||97.7%]
21 [||||||||||||||||||||||||||98.2%]  61 [||||||||||||||||||||||||||98.3%]  101[||||||||||||||||||||||||||98.8%]  141[||||||||||||||||||||||||||98.8%]
22 [||||||||||||||||||||||||||97.7%]  62 [||||||||||||||||||||||||||97.7%]  102[||||||||||||||||||||||||||99.4%]  142[||||||||||||||||||||||||||98.3%]
23 [||||||||||||||||||||||||||98.2%]  63 [||||||||||||||||||||||||||98.8%]  103[||||||||||||||||||||||||||98.8%]  143[||||||||||||||||||||||||||97.7%]
24 [||||||||||||||||||||||||||99.4%]  64 [||||||||||||||||||||||||||98.2%]  104[||||||||||||||||||||||||||97.7%]  144[||||||||||||||||||||||||||97.7%]
25 [||||||||||||||||||||||||||98.8%]  65 [||||||||||||||||||||||||||98.8%]  105[||||||||||||||||||||||||||98.2%]  145[||||||||||||||||||||||||||99.4%]
26 [||||||||||||||||||||||||||98.8%]  66 [||||||||||||||||||||||||||98.3%]  106[||||||||||||||||||||||||||98.8%]  146[||||||||||||||||||||||||||98.8%]
27 [||||||||||||||||||||||||||98.8%]  67 [||||||||||||||||||||||||||97.7%]  107[||||||||||||||||||||||||||98.8%]  147[||||||||||||||||||||||||||98.2%]
28 [||||||||||||||||||||||||||98.8%]  68 [||||||||||||||||||||||||||97.7%]  108[||||||||||||||||||||||||||97.7%]  148[||||||||||||||||||||||||||99.4%]
29 [||||||||||||||||||||||||||98.2%]  69 [||||||||||||||||||||||||||98.8%]  109[||||||||||||||||||||||||||98.8%]  149[||||||||||||||||||||||||||99.4%]
30 [||||||||||||||||||||||||||98.2%]  70 [||||||||||||||||||||||||||98.3%]  110[||||||||||||||||||||||||||98.3%]  150[||||||||||||||||||||||||||98.8%]
31 [||||||||||||||||||||||||||98.8%]  71 [||||||||||||||||||||||||||98.2%]  111[||||||||||||||||||||||||||98.8%]  151[||||||||||||||||||||||||||98.2%]
32 [||||||||||||||||||||||||||98.2%]  72 [||||||||||||||||||||||||||99.4%]  112[||||||||||||||||||||||||||98.2%]  152[||||||||||||||||||||||||||98.8%]
33 [||||||||||||||||||||||||||98.2%]  73 [||||||||||||||||||||||||||98.2%]  113[||||||||||||||||||||||||||97.7%]  153[||||||||||||||||||||||||||98.2%]
34 [||||||||||||||||||||||||||99.4%]  74 [||||||||||||||||||||||||||98.8%]  114[||||||||||||||||||||||||||98.2%]  154[||||||||||||||||||||||||||98.8%]
35 [||||||||||||||||||||||||||98.8%]  75 [||||||||||||||||||||||||||99.4%]  115[||||||||||||||||||||||||||98.3%]  155[||||||||||||||||||||||||||98.3%]
36 [||||||||||||||||||||||||||97.7%]  76 [||||||||||||||||||||||||||98.2%]  116[||||||||||||||||||||||||||98.3%]  156[||||||||||||||||||||||||||99.4%]
37 [||||||||||||||||||||||||||98.2%]  77 [||||||||||||||||||||||||||98.2%]  117[||||||||||||||||||||||||||97.7%]  157[||||||||||||||||||||||||||97.7%]
38 [||||||||||||||||||||||||||98.8%]  78 [||||||||||||||||||||||||||98.2%]  118[||||||||||||||||||||||||||97.6%]  158[||||||||||||||||||||||||||98.2%]
39 [||||||||||||||||||||||||||98.8%]  79 [||||||||||||||||||||||||||98.3%]  119[||||||||||||||||||||||||||97.7%]  159[||||||||||||||||||||||||||97.7%]
40 [||||||||||||||||||||||||||98.8%]  80 [||||||||||||||||||||||||||98.2%]  120[||||||||||||||||||||||||||98.2%]  160[||||||||||||||||||||||||||97.7%]
Mem[|||||||||||||||||||||||||||||||                          99.1G/255G]  Tasks: 367, 987 thr; 267 running
Swp[                                                              0K/0K]  Load average: 105.41 31.12 10.79
                                                                          Uptime: 12:28:21

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
49209 m6p004e    20   0 1461M  164M 80512 R 88.3  0.1  0:40.32 ./cs_solver --mpi
48648 m6p004e    20   0 1482M  201M 97536 R 88.3  0.1  0:40.24 ./cs_solver --mpi
49485 m6p004e    20   0 1482M  188M 84032 R 87.7  0.1  0:40.04 ./cs_solver --mpi
49300 m6p004e    20   0 1484M  202M 96064 R 87.7  0.1  0:40.15 ./cs_solver --mpi
49173 m6p004e    20   0 1459M  167M 85184 R 87.7  0.1  0:40.27 ./cs_solver --mpi
49028 m6p004e    20   0 1457M  163M 83648 R 87.7  0.1  0:40.20 ./cs_solver --mpi
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Fig. 4. Load of a single node with 160 MPI tasks.

III. Description of the Code

Code_Saturne is an open-source CFD software package based on the finite volume method to simulate the Navier-Stokes equations. It can handle any type of mesh built with any cell/grid structure. Incompressible and compressible flows can be simulated, with or without heat transfer, and a wide range of turbulence models is also available. The velocity-pressure coupling is handled using a projection-like method. The default algorithm to compute the velocity is the Jacobi algorithm and the pressure is solved with the help of an algebraic multigrid (AMG) algorithm. Parallelism is handled by distributing the domain over MPI processes, with an optional second level of shared memory parallelism based on the OpenMP model. Several partitioning tools are available, i.e. geometric ones (Space-Filling Curve with Morton and Hilbert approach) and graph-based ones (METIS [7], ParMETIS [7], SCOTCH [8] and PT-SCOTCH [8]).

Code_Saturne can be used as a standalone package, but extra libraries may also be plugged in, as to read some of the supported mesh formats (CGNS, MED, CCM, for instance), to get access to graph-partitioners (METIS, ParMETIS, SCOTCH, PT-SCOTCH) or to additional sets of linear solvers (PETSc [9], for instance). The code (350,000 lines) is written in Fortran ($\sim$37%), C ($\sim$50%) and python ($\sim$13%). In this work, python is only used on the frontend, where *Code_Saturne* 'cases' are prepared, i.e. the executable is created, accounting from the usersubroutines, and a symbolic link is added pointing on the mesh in the native format.

MPI is used for communications between subdomains and OpenMP pragmas have been added to the most time-consuming parts of the code.

MPI-IO is used for output of postprocessing files, which by default uses the EnSight Gold format, also readable by ParaView, for dumping potential checkpointing files and meshes (*mesh_output* file) if requested by the user, and also for reading the *mesh_input* file and potential restart files.

IV. Test Case

The canonical test case of the lid-driven cavity is used for benchmarking. The configuration is a cubic box of size $1\times1\times1$ and the mesh is made of 13 million tetrahedral cells. Symmetry boundary conditions are set in the spanwise direction, inlet boundary conditions at the top of the box, with a constant horizontal velocity of 1 and wall boundary conditions for the 3 last faces of the box.
The Reynolds number based on the cavity edge size and the lid velocity is set to 100.

V. Settings

Code_Saturne version 4.2.1 is used, as it supports PETSc and allows testing the behaviour of the code on GPUs.
For the CPU simulations, the native Algebraic Multigrid algorithm is used as a preconditioner with a Conjugate Gradient-like algorithm as a solver for the Poisson pressure equation.
For the simulations involving GPUs, the Conjugate Gradient algorithm of the PETSc library is used to solve the Poisson pressure equation.

VI. Results

Performance studies are carried out, using different MPI distributions and compilers. They will be mentioned at the beginning of each subsection. The native Algebraic Multigrid algorithm is used as a preconditioner and a Conjugate

Gradient-like algorithm as a solver, in order to solve the Poisson pressure equation, except in the last tests dealing with GPUs, where the linear solver is changed to the Conjugate Gradient algorithm of the PETSc library, which supports CUDA. Fifty time-steps will be run, except for the PETSc simulations, where only 5 time-steps will be run, in order to limit compute time consumption, but still showing a realistic behaviour of the code.

Nomenclature: #C is the number of cores, #T the number of threads, T (s) the average time for a time-step to complete, E (%) the code efficiency and SP the speedup observed between two simulations using the same number of CPUs, but without and with GPU acceleration.

A. POWER8 nodes vs x86 node

The first test deals with simulations performed on 1 node of the S822LC (made of 20 cores) machine, the S824L (made of 24 cores) machine and an x86 Ivy Bridge E5-2697v2 2.7GHz node, made of 24 cores [10]. Table I gathers the timings obtained while running on half a node (i.e. using the two sockets half loaded and not only one of the 2 sockets) and a full node (without using hyper-threading).

TABLE I
COMPARISON OF 2 TYPES OF NODES OF POWER8 AND 1 NODE OF X86.

S822LC			S824L			x86		
#C	T (s)	E (%)	#C	T (s)	E (%)	#C	T (s)	E (%)
10	26.50	-	12	21.51	-	12	31.61	-
20	16.43	**81**	24	11.80	**91**	24	25.33	**62**

A speed-up is observed for all 3 cases, but the efficiency is much better on POWER8 nodes being over 80 % for S822LC and over 90 % for S824L, whereas it is quite poor for the x86 node (about 62 %). This is most likely explained by a better bandwidth in the case of the POWER8 nodes, but also by the presence of a L3 cache partitioned per core on POWER8 nodes, which reduces performance degradation when the system is fully loaded. All these very early tests on POWER8 nodes have been carried out in SMT8 mode, and performance of simulations using 1 process per physical core might be slightly improved in SMT1 mode, as in SMT8 mode, a single logical core does not have access to all store and load units. Note also that the same partition is used for S824L and x86.
It is also quite remarkable to see that the simulation on 1 node, using S824L is more than twice as fast as on the x86.

B. Pure MPI

The runs on the POWER8 nodes have been performed using IBM Parallel Environment (PE) and XL compilers.

Table II presents the timings obtained when using hyperthreading on a single node, in SMT 8 mode (the same as the one described for the CPU binding). A 15 % (resp. 18 %) gain is observed going from 20 to 40 threads (resp. 24 to 48 threads) on S822LC (resp. S824L). From 24 to 96 using the S822LC node, a gain of 22 % is achieved.

TABLE II
SIMULATIONS RESTRICTED TO 1 NODE (MPI ONLY WITH MULTI-THREADING - SMT8).

S822LC		S824L	
	PE/XL		PE/XL
#C	T (s)	#C	T (s)
20	16.43	24	11.80
40	14.35	48	9.94
80	14.38	96	9.62
160	14.75	192	11.10

C. MPI & OpenMP

The simulations are performed with 20 MPI tasks (physical cores) using 1 node of S822LC. The number of threads varies from 1 to 8 and the SMT8 mode is used. The performance of the OPENMPI + GNU distribution is presented.

TABLE III
SIMULATIONS RESTRICTED TO 1 NODE (MPI AND OPENMP - 20 MPI TASKS - SMT8).

S822LC		
	OPENMPI/GNU	
#T	T (s)	SP
1	18.49	1.00
2	16.01	1.16
4	14.20	1.30

Table III shows that a speed-up of 1.3 is achieved when using 4 threads (80 non-physical threads in total).

D. Comparison CPU and CPU+GPU

The objective of the last series of tests is to compare performance when using CPUs and CPUs & GPUs. These are preliminary tests on POWER8 nodes. Two K80s (each of them made of 2 GK210 GPUs) are available per node on the S822LC machine, and 2 K40s on the S824L machine. Figures 5 and 6 show the GPU loads for a 4 CPUs & 4 GPUs simulations and for a 2 CPUs & 2 GPUs simulation, using a S822LC and S824L node, respectively (cs_solver is the default name of *Code_Saturne*'s executable).

The second and fourth columns of Table IV show the timings obtained on CPUs only, as a function of the number of used CPUs and the third and fifth columns the timings observed when using the GPUs. Note that the values are displayed in italic when the GPUs are overloaded.
Using GPUs is clearly cheaper as long as they are not overloaded (running only 1 executable).

```
+------------------------------------------------------+
| NVIDIA-SMI 352.59     Driver Version: 352.59         |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K80           On   | 0000:03:00.0     Off |                    0 |
| N/A   51C    P0    66W / 149W |    477MiB / 11519MiB |     28%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K80           On   | 0000:04:00.0     Off |                    0 |
| N/A   42C    P0    79W / 149W |    477MiB / 11519MiB |     27%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla K80           On   | 0002:03:00.0     Off |                    0 |
| N/A   49C    P0    67W / 149W |    477MiB / 11519MiB |     32%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla K80           On   | 0002:04:00.0     Off |                    0 |
| N/A   41C    P0    78W / 149W |    477MiB / 11519MiB |     27%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0     44382    C   ./cs_solver                                    420MiB |
|    1     44383    C   ./cs_solver                                    420MiB |
|    2     44384    C   ./cs_solver                                    420MiB |
|    3     44385    C   ./cs_solver                                    420MiB |
+-----------------------------------------------------------------------------+
```

Fig. 5. K80 usage on the S822LC node.

```
+------------------------------------------------------+
| NVIDIA-SMI 352.59     Driver Version: 352.59         |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K40m          On   | 0002:01:00.0     Off |                    0 |
| N/A   35C    P0    61W / 235W |    144MiB / 11519MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla K40m          On   | 0006:01:00.0     Off |                    0 |
| N/A   35C    P0    63W / 235W |    144MiB / 11519MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID  Type  Process name                               Usage      |
|=============================================================================|
|    0     30297    C   ./cs_solver                                     87MiB |
|    1     30298    C   ./cs_solver                                     87MiB |
+-----------------------------------------------------------------------------+
```

Fig. 6. K40 usage on the S824L node.

TABLE IV
SIMULATIONS RESTRICTED TO 1 NODE (CPUS VS CPUS+GPUS - SMT8).

S822LC				S824L			
	CPU	CPU &GPU			CPU	CPU &GPU	
#C	T (s)	T (s)	SP	#C	T (s)	T (s)	SP
1	1022.18	630.54	1.62	1	1087.75	637.86	1.71
2	621.20	337.12	1.84	2	659.71	327.81	2.01
4	263.61	173.95	1.51	4	267.82	*189.04*	*1.42*
20	76.38	*109.75*	*0.70*	24	57.81	*104.82*	*0.55*

VII. CONCLUSIONS - FUTURE WORK

Code Saturne has been tested on 2 types of IBM POWER8 nodes, using MPI only, MPI OpenMP and also on GPUs, with the help of the PETSc library. Compared to current x86 systems the performance scales nearly linearly as a function of the number of cores per node, especially on the IBM S824L model. Using SMT mode with several MPI or OpenMP threads per physical core helps reducing the time to solution, but the main results come from the preliminary tests carried out on POWER8 nodes using PETSc on GPUs, where it is cheaper to use the CPU/GPU model than pure CPU, up to a number of CPUs equal to the number of GPUs available on a node. Using the PETSc library to make use of the GPU architecture provided interesting gain in compute time, but the full capability of each node was not exploited. New compilers with OpenMP 4.5 directives for GPU will be available in the next few months, and it will be then possible to offload some other parts of the code to GPUs. Another significant performance boost should be observed when the new NVIDIA TESLA P100 GPU architecture will be available. Beyond a dedicated high speed interconnect (NVLink) between the GPUs and POWER CPUs, *Code_Saturne* should be able to take advantage of other features, in terms of performance and implementation, like hardware Unified Memory, paging, Compute Preemption and stacked high bandwidth memory.

ACKNOWLEDGMENTS

The authors would like to thank the Hartree Centre for performing the x86 tests on their NextScale machine. They also would like to thank PRACE (PRACE-4iP - WP7.2B) for funding the performance analysis on GPUs and the UK Turbulence Consortium (EP/L000261/1) for the rest of the work.

REFERENCES

[1] F. Archambeau, N. Méchitoua, M. Sakiz; *Code_Saturne: a finite volume code for the computation of turbulent incompressible flows - industrial applications*. International Journal on Finite Volumes **1** (2004) 1–62.
[2] Y. Fournier, J. Bonelle, C. Moulinec, Z. Shang, A.G. Sunderland, J.C. Uribe; *Optimizing Code_Saturne computations on Petascale systems*. Computers & Fluids **45** (2011) 103–108.
[3] http://www.code-saturne.org
[4] http://www.prace-ri.eu/ueabs
[5] http://www.fz-juelich.de/ias/jsc/EN/Expertise/High-Q-Club/_node.html
[6] https://www.redbooks.ibm.com/portals/power
[7] http://glaros.dtc.umn.edu/gkhome/views/metis
[8] https://gforge.inria.fr/projects/scotch/
[9] https://www.mcs.anl.gov/petsc/
[10] http://community.hartree.stfc.ac.uk/wiki/site/admin/resources.html#systems2

Emerging Technologies for the Convergence of HPC and Big Data

Herbert Cornelius
INTEL

***Abstract*— As we see Moore's Law well alive, more and more parallelism is introduced into all computing platforms and on all levels of integration to achieve higher performance and energy efficiency. HPC is also reaching an inflection point with the convergence of (traditional) high performance computing and the emerging world of Big Data Analytics: High-Performance Data Analytics (HPDA) – using HPC for Big Data. We will discuss the evolution of general purpose Multi-and Many-Core processing, and for very specific applications and workloads augmented by FPGA acceleration in the future. In order to support the ever increasing compute performance, we will furthermore discuss new emerging memory and storage technologies and paradigms, as well as advanced communication technologies supporting new capabilities and capacities going forward.**

Are you getting the wrong answer, but fast? Challenges for reproducible research in HPC

Lorena Barba
School of Engineering and Applied Science George
Washington University in Washington, DC
More info: https://about.me/lorenabarba

Abstract—**On undertaking a full replication study of a previous publication by our own research group, we learned some new lessons about reproducible computational research. The previous study used an in-house code to solve the Navier-Stokes equations for flow around an object, using GPU hardware. As is common in CFD applications, we rely on an external library for solving linear systems of equations, in this case, the NVIDIA Cusp library. We later did a code re-write of the CFD solver in a distributed parallel setting, on CPU hardware. This version uses the PETSc library for solving linear systems of equations. In addition, we used two open-source CFD libraries: OpenFOAM and IBAMR (from New York University). Apart from the many things that can go wrong with discretization meshes and boundary conditions, we found that simply using a different version of an external library can lead to different results, for example. In view of this exercise, we tightened our reproducibility practices even more. Open data and code are the minimum requirement; we also require a fully automated workflow, systematic records of dependencies, environment and system, and only scripted visualizations (no GUI manipulation). We must also raise awareness of numerical non-reproducibility of parallel software and include this topic as part of the training of HPC researchers.**

High-Level Abstraction for Block Structured Application: A Lattice Boltzmann Exploration

Jianping Meng
Scientific Computing Department,
STFC Daresbury laboratory,
Warrington WA4 4AD, UK

Xiao-Jun Gu
Scientific Computing Department,
STFC Daresbury laboratory,
Warrington WA4 4AD, UK

David R. Emerson
Scientific Computing Department,
STFC Daresbury laboratory,
Warrington WA4 4AD, UK

Gihan R. Mudalige
Oxford e-Research Centre,
University of Oxford,
Oxford OX1 3QG, UK

István. Z. Reguly
Faculty of Information Technology
and Bionics, PPCU, Hungary

Mike B. Giles
Oxford e-Research Centre,
University of Oxford,
Oxford OX1 3QG, UK

***Abstract*—We explore the Oxford Parallel Library for Structured-mesh solvers (OPS) by developing a two-dimensional lattice Boltzmann application. OPS, an embedded domain specific language (EDSL) minimizes the programming effort for utilising various emerging hardware platforms by providing a concise abstraction and automatic code generation facilities. When using OPS the applications developer does not use any explicit calls to MPI and/or utilize OpenMP/CUDA where the EDSL generates the necessary parallelisations automatically. The resulting application demonstrates near optimal performance in tests for both multiple-CPU and multiple-GPU calculations.**

***Index Terms*—Heterogeneous system, High-level abstraction, Lattice Boltzmann method.**

Multi-core and many-core processors have gained popularity in high-performance computing including both mainstream CPUs and add-on accelerators. This trend however, presents a significant challenge for scientific software development since there are competing hardware platforms which often require different software frameworks. It is not only time-consuming but also risky to adapt to an emerging hardware platform.

One approach to tackle this challenge is to utilise high-level abstractions, which allows the decoupling of computations from their parallel implementation. Following this spirit, OPS was designed to ease the burden of developing block-structured applications [1]. With carefully designed abstractions, the library hides both the implementation details on various parallel systems and the complexity of the multi-block mesh technique such as domain decomposition and communications among blocks. It is therefore very attractive from an application developer's point of view. In this work, we will explore the library by developing a two-dimensional lattice Boltzmann code and report the experience [2].

The lattice Boltzmann method (LBM) is a simple to understand yet powerful mesoscopic CFD tool [3], which is particularly suitable for parallelisation and is often able to provide near-linear scaling. As a special approximation to the Boltzmann-BGK equation [4]–[7], its governing equation is

$$\frac{\partial f_\alpha}{\partial t} + \boldsymbol{C}_\alpha \cdot \nabla f_\alpha = -\frac{1}{\tau}(f_\alpha - f_\alpha^{eq}) + F_\alpha, \qquad (1)$$

which represents the evolution of the distribution function $f_\alpha(\boldsymbol{r}, t)$ for the αth discrete velocity $\boldsymbol{C}_\alpha$ at position $\boldsymbol{r} = (x, y, z)$ and time t. The effect of external body force is described by F_α, and the discrete equilibrium distribution function by $f_\alpha^{eq}(\boldsymbol{r}, t)$. In order to simulate incompressible and isothermal flows, it is common to use an equilibrium function with second order velocity terms, i.e.,

$$f_\alpha^{eq} = w_\alpha \rho [1 + \frac{\boldsymbol{U}\cdot\boldsymbol{C}_\alpha}{RT_0} + \frac{1}{2}\frac{(\boldsymbol{U}\cdot\boldsymbol{C}_\alpha)^2}{(RT_0)^2} - \frac{\boldsymbol{U}\cdot\boldsymbol{U}}{2RT_0}], \qquad (2)$$

which is determined by the density ρ, the fluid velocity $\boldsymbol{U}$, and the reference temperature T_0, where the gas constant is denoted by R. The weight factor is denoted by w_α for a discrete velocity $\boldsymbol{C}_\alpha$. The relaxation time τ is related to the fluid viscosity μ and the pressure p via the Chapman-Enskog expansion, i.e., $\mu = p\tau$. To get the macroscopic quantities, for example the density and velocity, we only need summation operations, i.e.,

$$\rho = \sum_\alpha f_\alpha, \text{ and, } \rho\boldsymbol{U} = \sum_\alpha f_\alpha \boldsymbol{C}_\alpha.$$

As shown, the governing equation (1) has a linear convection term, which is very easy to be numerically solved in parallel. In particular, a smart trapezoidal scheme can be used to achieve the stream-collision scheme [8],

$$\begin{gathered}\tilde{f}_\alpha(\boldsymbol{r} + \boldsymbol{C}_\alpha dt, t + dt) - \tilde{f}_\alpha(\boldsymbol{r}, t) = \\ -\tfrac{dt}{\tau+0.5dt}\left[\tilde{f}_\alpha(\boldsymbol{r}, t) - f_\alpha^{eq}(\boldsymbol{r}, t)\right] + \tfrac{\tau F_\alpha dt}{\tau+0.5dt},\end{gathered} \qquad (3)$$

by introducing

$$\tilde{f}_\alpha = f_\alpha + \frac{dt}{2\tau}(f_\alpha - f_\alpha^{eq}) - \frac{dt}{2}F_\alpha. \qquad (4)$$

At the same time, the macroscopic quantities become

$$\rho = \sum_\alpha \tilde{f}_\alpha, \text{ and, } \rho\boldsymbol{U} = \sum_\alpha \boldsymbol{C}_\alpha \tilde{f} + \frac{\rho \boldsymbol{G} dt}{2}. \qquad (5)$$

where the actually acceleration is denoted by $\boldsymbol{G}$.

For two dimensional flows, the D2Q9 lattice is commonly used where the nine discrete velocities ($\alpha = 1..9$) are

$$C_{\alpha,x} = \sqrt{3RT_0}[0, 1, 0, -1, 0, 1, -1, -1, 1], \quad (6)$$

$$C_{\alpha,y} = \sqrt{3RT_0}[0, 0, 1, 0, -1, 1, 1, -1, -1], \quad (7)$$

and the corresponding weights are

$$w_\alpha = [\frac{4}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}, \frac{1}{36}]. \quad (8)$$

To test the code performance, this lattice will be used for the simulations in this paper.

Since the scheme (3) actually assumes a regular grid, it is particularly suitable to be implemented using the OPS library which supports solvers over structured meshes. In this library, a concise abstraction framework is designed to ease the burden of application development. As a multi-block structured mesh will divide the whole geometry into blocks, the library provides an **ops_block** construct to describe this split conceptually. A quantity such as f needed for evolution can then be defined as an **ops_dat** construct on the block by using a function **ops_decl_dat**, which requires the input of the dimension of quantities and the mesh size (see Code example 1). The library hides the implementation detail for distributing blocks into different MPI ranks if in a distributed computing environment. Of course, a large block, which may not fit to one core due to either the computation or memory cost, will be also split automatically.

Code example 1: Define blocks and variables

```
ops_block* grid2D;
grid2D = new ops_block[block_num];
grid2D[ib] = ops_decl_block(2, blockname);
ops_dat* velo;
velo = new ops_dat[block_num];
velo[ib]=ops_decl_dat(grid2D[ib],2,
        size,base,d_m,d_p,(Real*)temp,"double", veloname);
```

Code example 2: Define kernels and parallel loops

```
void collision(const Real *f, const Real *feq, Real* fcoll,
        const Real* omega) {
       for (int l = 0; l < nc; l++) {
           fcoll[OPS_ACC_MD2(l, 0, 0)] = (1 - (*omega)) *
               f[OPS_ACC_MD0(l,
           0, 0)] + (*omega) * feq[OPS_ACC_MD1(l, 0, 0)];
       }
}
ops_par_loop(collision, "collision", grid2D[0], 2,
    iter_range_coll,
ops_arg_dat(f[0], nc, S2D_00, "double", OPS_READ),
ops_arg_dat(feq[0], nc, S2D_00, "double", OPS_READ),
ops_arg_dat(fcoll[0], nc, S2D_00, "double", OPS_WRITE),
ops_arg_gbl(&omega, 1, "double", OPS_READ));
```

The key part of a typical scientific code is a loop over all the grid points which updates evolutionary quantities such as f. There is an **ops_par_loop** API call in the library for this purpose. It accepts a so-called kernel function as its input and automatically distributes the operations described in the kernel function over grid points assuming that the loop iterations are independent (see Code example 2). OPS will automatically generate different parallelizations for this loop. For instance, if we aim to run the code on a GPU, then the call may be translated into a CUDA source file. Also, utilising this opportunity, the script may be able to do other optimisations such as vectorisation. Currently, the library supports CUDA, OpenACC, OpenMP and OpenCL so that the final code will be able to run on most of current hardware platforms. Moreover, the library also supports execution on Multi-GPUs/Intel Xeon Phis by using hybrid MPI/CUDA(OpenMP, OpenACC, OpenCL) programming.

Another important aspect is to deal with communications among connected blocks. For this purpose, the OPS library implements two constructs and three functions (see Code example 3). Firstly, an **ops_halo** construct is provided and can be defined by calling **ops_decl_halo**, which needs the connectivity information as input. Additionally, a **ops_halo_group** class can be used to group all the related halo points, which can be defined by using the call **ops_decl_halo_group**. Finally, to trigger the communications within the halo group, one just needs to call **ops_halo_transfer**, which will finish all the operations as necessary.

Code example 3: Define halos and trigger communications

```
ops_halo* halos;
ops_halo_group f_halos;
halos[1] = ops_decl_halo(velo[0], velo[1], halo_iter,
        base_from, base_to, dir, dir);
f_halos = ops_decl_halo_group(halo_num, halos);
ops_halo_transfer(f_halos);
```

By utilising the above abstractions, an application developer can focus on the scientific problem itself and the library will take care of almost all of implementation details and optimisation. In general, no explicit MPI/CUDA/OpenCL call will be necessary for the application code. However, due to the complexity of debuging a parallel code, the recommended working flow is to debug the application code in the non-optimised serial and MPI mode first, and ensure that the code can work correctly, then utilise the code generation tools to translate the code into an optimised form for various hardware.

Following the aforementioned work flow, we first develop a two-dimensional lattice Boltzmann code based on the OPS library [2], and then test the code on various platforms including an IBM Power S824L server, an IBM Power 8335-GTA server and an iDataPlex cluster by using a 4096×4096 mesh. In the Power S824L system, there are 24 physical cores running at 3.32GHz and two NVIDIA Tesla K40 GPU accelerators while there are 32 nodes with 16 CPU cores and two NVIDIA Tesla K80 GPU ccelerators per node for the IBM Power 8335-GTA server. The iDataPlex cluster is equipped with the Intel Ivy Bridge E5-2697v2 2.7GHz CPU. For the IBM Power S824L server, we use the IBM XL C/C++ V13.1.2 compiler for CPU and the NVIDIA CUDA 7.0 SDK for the K40 GPU. Moreover, OpenMPI 1.10.1 is used for message passing of both multiple-CPU and multiple-GPU calculations. While for the IBM Power 8335-GTA server, the tools sets are upgraded to the IBM XL C/C++ V13.1.3, the NVIDIA CUDA 7.5 and OpenMPI 1.10.2 respectively. For the iDataPlex cluster, the code is compiled by the Intel C/C++ V15.2.164, and the Intel MPI 5.03 is used.

To investigate the capability of using various hardware platforms and associated programming API, we first run a few tests on MPI, OpenMP, CUDA and hybrid MPI/OpenMP using the IBM Power S824L system. As discussed above, the provided Python script can automatically translate all the kernel functions into appropriate form and there is no need to manually write any MPI, OpenMP or CUDA code. The generated code can be compiled natively using relevant compilers, and is human readable which is very useful. As has been shown in Fig. 1, the various versions of compiled code run successfully on the system. Interestingly, the MPI version actually shows better performance than the OpenMP version, which is consistent to the finding in [1] (see e.g., Fig. 5).

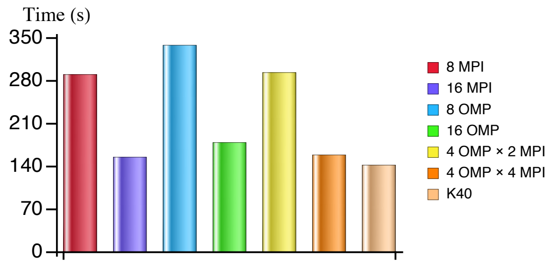


Fig. 1: Tests of MPI, OpenMP(OMP), CUDA and hybrid MPI/OpenMP capability using the IBM Power S824L system.

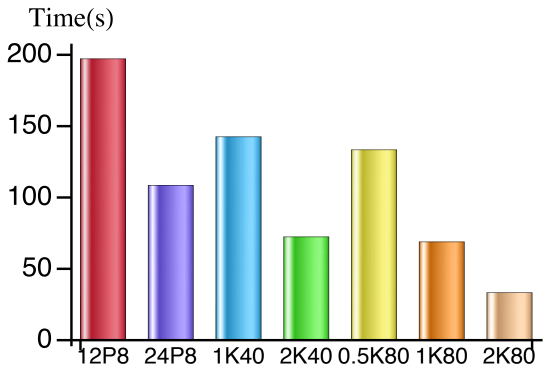


Fig. 2: Multiple-GPU tests using IBM Power 8 systems. “P8” means the Power S824L CPU core.

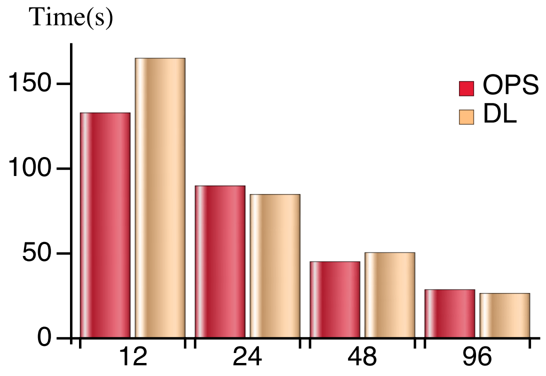


Fig. 3: Strong scaling tests using the iDataPlex cluster. “DL” means the LBM code in the DL_MESO package and “OPS” the present 2D LBM code.

Further tests have been conducted for the Multiple-GPU calculations, which is perhaps the most interesting capability from an application developer point of view. In Fig. 2, we presents the results on both two IBM systems although the CPU results are based on the Power S824L CPU. It is found that, while using ***double precision***, a single K40 GPU can achieve similar performance of 16 IBM Power S824L CPU cores. Consistently, “a half” K80 GPU has the similar performance. The code demonstrates nearly linear scalability for the current tests, which is important for large scale simulations.

The capability for multiple-CPU calculations is mainly tested based on the iDataPlex cluster. The results are shows in Fig. 3. For reference, we also present data of the LBM code in the DL_MESO (DL) package, which is developed by the Daresbury laboratory. However, we remind that the two codes have different functionalities and thereby different internal data structures, the results should only be understood as a rough reference. It can be seen that the code also demonstrates nearly linear scaling behaviour for multiple-CPU calculations. Considering that no MPI code is manually written for the OPS-based code, it appears that we have good chance to achieve a good balance of performance and productivity.

It is also of interest to look at the energy consumption. For this purpose, we choose to use the iDataPlex cluster for tests of CPU since the system will report the total energy consumption which is believed to be more reliable, although this system also take the energy consumption of idle CPU cores into account. For the K40 and K80 GPU, we calculate the consumption according to the computational time and its power specified by NVIDIA. As has been shown in Table I, the results are in general consistent with those of computational time. In particular, the K80 GPU uses roughly a half of energy of the K40 GPU, which is consistent to the TDP and performance specification of two GPUs.

Overall, it appears that the code does not achieve excellent speedups over one CPU core. This might be attributed two reasons: the first one is, as motioned above, we are using double precision; The second one is that the OPS based code is written in a way that is more easy to understand from the application point of view. For instance, the iteration for Eq. (3) has been divided into five major **ops_par_loop**s and there also several sub **ops_par_loop**s for such as the boundary treatment. This way may not be optimal for the GPU calculation since it may induce more data exchanges between CPU and GPU. In fact, it appears that a LBM code tends to be memorybound [9] so that the memory bandwidth can become a serious limiting factor. Therefore, the performance may be improved by using more GPU-oriented optimisation but there will be certainly a trade-off between performance and such as readability of the code. On the other hand, the performance reported here appears consistent with the discussion in Ref. [9] in which a Nvidia GTX280 GPU just gets 5X speedups over one Intel Core I7 CPU with for cores.

TABLE I: Energy consumption on iDataPlex and GPUs

Cores/GPU	DL (KWh)	OPS (KWh)
12	0.0072	0.0061
24	0.00397	0.00477
48	0.0028	0.00371
1K40	–	0.00931
1K80	–	0.00495

To summarise, the OPS library provides concise abstractions for writing block-structured applications. It is encouraging that the OPS based LBM code demonstrates nearly linear scaling behaviour in our simulations for both multiple-CPU and multiple-GPU calculations. The latter is also important because a single-GPU is not enough for certain tasks requiring either large memory or more computing power. With the reference of the LBM code in the DL_MESO package, we find that the OPS based code achieves nearly optimal performance while no MPI and/or CUDA/OpenMP code is manually written.

Acknowledgements

All authors would like to thank the UK Engineering and Physical Sciences Research Council for their support "Future-proof massively-parallel execution of multi-block applications" (EP/K038451/1 and EP/K038494/1). JPM and DRE also thank for the UK Consortium on Mesoscale Engineering Sciences (UKCOMES) under Grant EP/L00030X/1.

References

[1] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, and J. A. Herdman, "Performance analysis of a high-level abstractions-based hydrocode on future computing systems," in *Proceedings of the 5th international workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS '14)*, 2014.

[2] J. P. Meng, X.-J. Gu, D. R. Emerson, G. R. Mudalige, I. Z. Reguly, and M. B. Giles, "Block structured lattice Boltzmann simulation using ops high-level abstraction," in *Conference Abstracts: 28th International Conference on Parallel Computational Fluid Dynamics*, 2016.

[3] S. Chen and G. D. Doolen, "Lattice Boltzmann method for fluid flows," *Annu. Rev. Fluid Mech.*, vol. 30, no. 1, pp. 329–364, 1998.

[4] X. He and L.-S. Luo, "A priori derivation of the lattice Boltzmann equation," *Phys. Rev. E*, vol. 55, pp. R6333–R6336, Jun 1997.

[5] X. He and L.-S. Luo, "Theory of the lattice Boltzmann method: From the Boltzmann equation to the lattice Boltzmann equation," *Phys. Rev. E*, vol. 56, pp. 6811–6817, 1997.

[6] X. Shan and X. He, "Discretization of the velocity space in the solution of the Boltzmann equation," *Phys. Rev. Lett.*, vol. 80, pp. 65–68, Jan 1998.

[7] X. W. Shan, X. F. Yuan, and H. D. Chen, "Kinetic theory representation of hydrodynamics: A way beyond the Navier Stokes equation," *J. Fluid Mech.*, vol. 550, pp. 413–441, 2006.

[8] X. He, S. Chen, and G. D. Doolen, "A novel thermal model for the lattice Boltzmann method in incompressible limit," *J. Comput. Phys.*, vol. 146, no. 1, pp. 282–300, 1998.

[9] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 451–460.

Analyzing the Impact of Parallel Programming Models in NoCs of Forthcoming CMP Architectures

Iván Pérez*, Emilio Castillo†‡, Ramón Beivide*, Enrique Vallejo*,
José Luis Bosque*, Miquel Moretó†‡, Marc Casas†‡, Mateo Valero†‡

*Universidad de Cantabria (UC), Santander, Spain
†Barcelona Supercomputing Center (BSC), Barcelona, Spain
‡Universitat Politècnica de Catalunya - BarcelonaTech (UPC), Barcelona, Spain

***Abstract*—Modern chip multiprocessors (CMPs) are complex systems with an increasing amount of cores. It is expected to see more heterogeneous CMPs with thousands of cores in the future. To take advantage of such parallel systems, alternative parallel programming paradigms are gathering strength. With this scenario, we performed realistic simulations in order to compare a traditional parallel programming model (pthreads) with an alternative based on tasks (OmpSs). To identify the advantages and understand the architectural requirements of OmpSs we modeled two networks on-chip (NoCs), a complete graph and a mesh. The results show that OmpSs provides better performance despite the overhead introduced by its runtime, independently of the NoC. Such a higher performance can lead to an increase of the on chip network requirements.**

I. Introduction

As the number of compute units and diversity grows in CMPs, the importance of optimizing parallel computing grows consequently. Conventional threading programming models like pthreads [5] or OpenMP [7] (before version 3.0) are inappropriate to implement certain types of parallelism, requiring complex and inefficient mechanisms. In addition, the increasing development cost with the increasing size of these systems, implies poor productivities. To alleviate these problems, task parallelism [4] is emerging. Task parallel paradigms distribute the workload in tasks and link them by data dependencies. When these data dependencies are met, tasks can be executed asynchronously. In this work we used OmpSs [8]. OmpSs' runtime tracks data dependencies between tasks. When these dependencies are satisfied, the runtime maps the task in a cpu thread. This type of execution improves the load balancing distributing the tasks when the cpu threads are available.

There are works that compare both programming models or equivalents. This is the case of [8], which evaluates six micro-benchmarks written in OmpSs, matching or improving the performance of their counterparts written in OpenMP and OpenCL, in homogeneous and heterogeneous environments. The work presented in [9] shows performance trade-offs between the OmpSs and OpenMP 4.0 tasking and loop parallelism and also exhibits speedup improvements towards pthreads, using some PARSEC benchmarks. In [6], the benefits of task programming models are analyzed in 16-core system with realistic workloads for the PARSEC benchmark suite.

In this work we analyze the differences of pthreads and OmpSs from the side of the interconnect and memory hierarchy. The purpose is to observe the response of the programming model to architectural changes, in particular the network topology, identifying the benefits and inconveniences of using OmpSs. To carry out this study we present detailed full-system simulations of some PARSEC benchmarks in a 64-core system using two NoCs: a fully connected graph and a mesh.

II. Methodology

We have performed our experiments by running three PARSEC [2] benchmarks under Linux 2.6.28 on a customized gem5 [3] able to support OmpSs. Table I collects the benchmarks and input sets used for the experiments.

Table I: Benchmarks and input sets simulated.

Benchmark	Input set
Blackscholes	1,048,576 options
Bodytrack	1 frame, 1,000 particles (simmedium)
Ferret	64 queries, 13,787 images (simmedium)

We configure gem5 to model a 64 core system based on x86. The organization of the system is presented in Figure 1. Table II summarizes the most relevant simulation parameters.

We use the out-of-order (O3) cores and a dynamic voltage and frequency scaling model from 100MHz to 2GHz. We use Ruby to model the memory hierarchy and Garnet [1] for the on-chip interconnection (NoC), with a frequency for network and caches of 1 GHz. We use the standard router with a pipeline of 5 stages and a MESI coherence protocol based on directories with a private L1 cache and a shared L2 with one cache bank per core. The number of memory controllers is set to 16, allocated to the first and last rows of cores in the SoC. Two topologies are used for the NoC: a complete graph as the performance-reference network, and a mesh as the realistic one. Three virtual networks are used in both cases to avoid protocol-deadlock.

III. Results

In this section we present metrics that relate the network topology with the programming model.

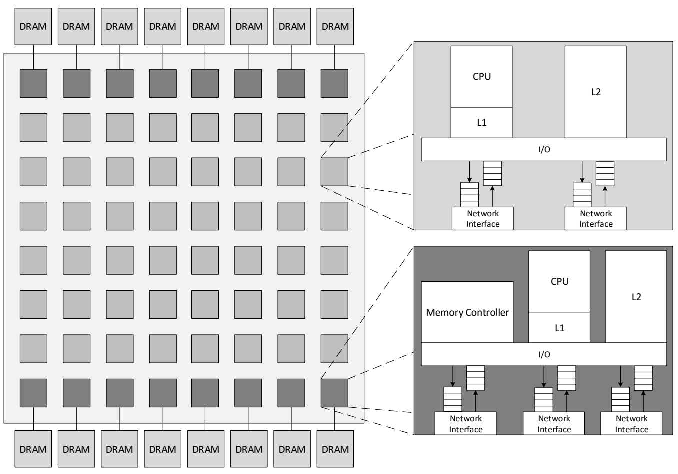


Figure 1: Modeled 64 core CMP with a 64 bank NUCA of shared cache. There are two types of nodes, one without memory controller (light gray) and the other with a memory controller (dark gray).

Table II: Simulation parameters

Parameter	Value
CPU units	64
CPU ISA	x86
CPU model	O3
CPU frequency	100 MHz - 2 GHz
Ruby frequency	1 GHz
Coherence protocol	MESI
Memory controllers	16
Network model	5-stage Garnet router
Topology	8x8 mesh and 64 complete graph
Virtual network (VN)	3
Virtual channels per VN	1
Buffers per port	10 flits
Flit size	16 B
Block size	64 B
Message control size	8 B
L1I Size	32 KB
L1D Size	64 KB
L1 Latency	1 Ruby cycle
L2 Size	64 banks of 512 KB
L2 Latency	15 Ruby cycles
DRAM type	DDR3-1600

A. *Performance*

The most significant metric to measure the performance of a system is execution time. Figure 2 shows execution times normalized to the complete graph with pthreads. As expected, the complete graph performs better than the mesh, thanks to its better distance and throughput parameters. Concerning the programming model, OmpSs consistently shows better results, especially with Bodytrack and Ferret. It is remarkable that the OmpSs versions of these benchmarks run on a mesh clearly outperform their pthread counterparts run on the complete graph. This illustrates the need of co-design techniques: a change on the programming model habilitates for much lower NoC costs without compromising performance.

Table III quantifies the relative speedup of the OmpSs versions with respect to the pthreads, one for each benchmark.

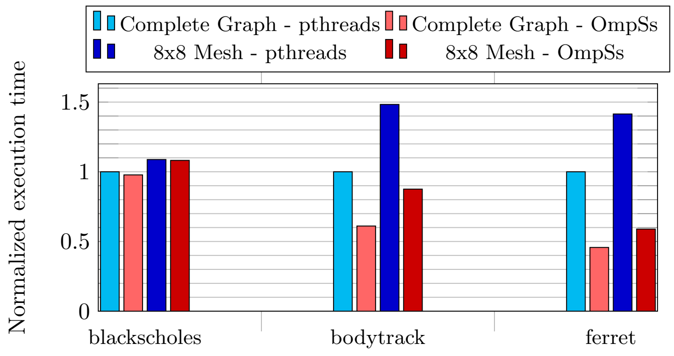


Figure 2: Execution time normalized to the complete graph with pthreads.

Table III: OmpSs speedup against pthreads.

Topology	Blackscholes	Bodytrack	Ferret
Complete Graph	1.023	1.637	2.187
Mesh	1.006	1.693	2.401

The speedup values are very similar for both topologies, and they are significantly larger for Bodytrack and Ferret. These two benchmarks are parallelized using a pipeline model, and the load-balancing and fine-grain task dependency tracking mechanisms implemented in OmpSs provide a more balanced execution and higher performance.

The performance improvement in Bodytrack when using OmpSs was already observed in [6], and the improvement of Ferret occurs due to a lack of scalability of the pthreads version with a large number of threads. Figure 3 shows a visualization of a trace of execution of the OmpSs version of Ferret in an 8x8 mesh. Bars of different colors represent different types of tasks; the pipeline is composed of five consecutive tasks. The dynamic assignment of tasks to cores makes that each set of pipeline tasks do not execute consecutively in the same core, but instead they are balanced dynamically as their operands become available. The pthreads version, by contrast, relies on a synchronization mechanism with significant overheads, which employs a huge number of threads, and results in reduced performance.

B. *Number of memory accesses*

The number of memory accesses in conjunction with the miss rate of each core are the two factors that will determine the amount of messages generated in the network that interconnects L2 banks (and memory controllers). Figure 4 shows the distribution of data memory accesses. Except for Blackscholes, OmpSs executions generate slightly more requests on average, caused by the larger number of instructions executed, which is presumably related with the runtime overhead.

C. *Miss rate*

Memory accesses that miss in some level of the hierarchy are the cause of the traffic injected in the NoC in our model. These messages comprise requests for a memory block, memory blocks data themselves or control messages generated

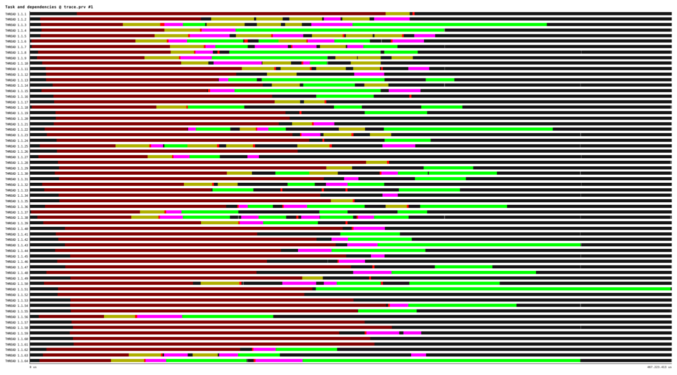


Figure 3: Visualization of the different tasks of the OmpSs version of Ferret in an 8x8 mesh.

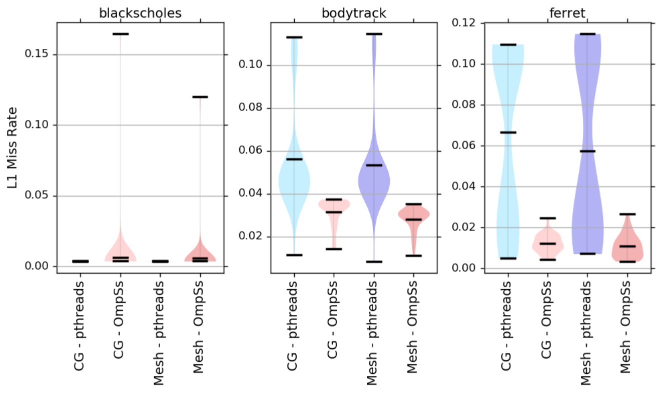


Figure 5: Distribution of L1-D miss rates.

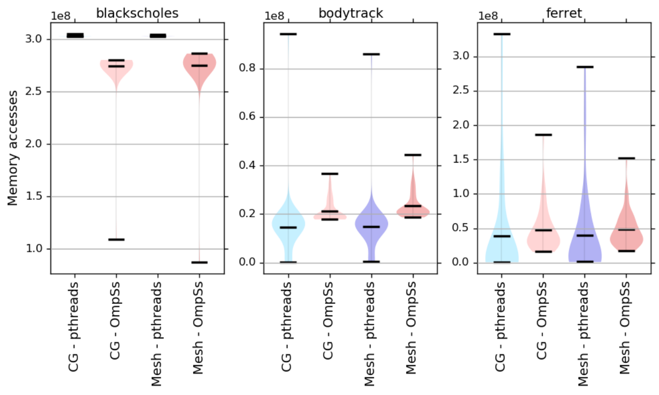


Figure 4: Distribution of data memory accesses per core. The three highlighted points of each distribution from top to bottom are the maximum, average and minimum memory accesses among cores.

by the coherence protocol. In this way, cache miss rates give an idea of the amount of traffic that the application will generate per memory access. As the L1 caches act as a NoC accesses filter, misses at L1 will provoke the most significant contribution to the traffic injected to the NoC. Figure 5 shows the distribution of the L1 cache miss rates in all the 64 cores.

OmpSs generally shows lower L1 miss rates than pthreads with the exception of Blackscholes, in which the slightly higher average miss rate of OmpSs comes from the large value corresponding to the main thread that creates tasks. This means that OmpSs generates less network traffic per memory access.

Also there are minimal differences between NoC topologies, but generally the mesh shows average lower miss rates which is a positive fact, reducing the total amount of traffic as described in the following subsection.

D. *Network Behaviour*

Figure 6 shows the amount of flits injected into the network and the injection rate. The results of injected flits are coherent with the L1 miss rates and the number of memory accesses showed previously. Because of the higher values of injection rate, it can be deduced that OmpSs stresses more the network in overall terms. These ratios are higher because of the shorter execution times and the similar amounts of injected flits. In addition, injection rates are lower for the 8x8 mesh, essentially due to the increased execution time.

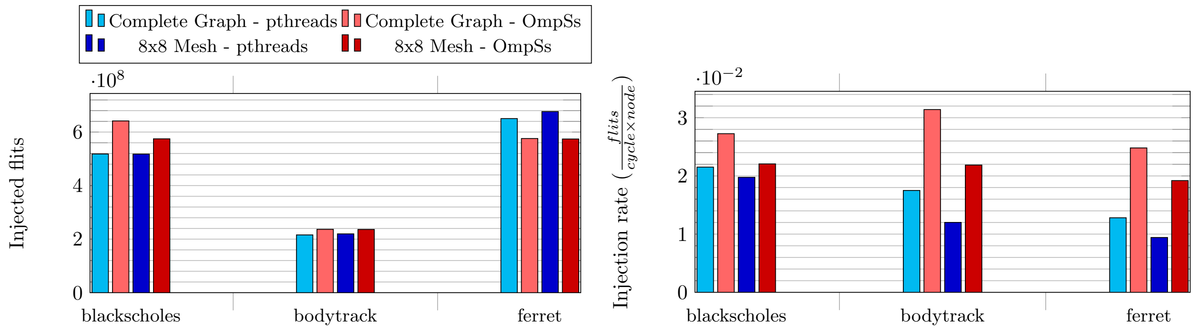
Figure 7 depicts the injection queueing latency and the network latency respectively. From these graphs, there are two points to remark:

- The non-linear part of the latency corresponding to contention at the routers is very low, supposing uniform traffic. This can be deduced knowing that base network latencies (latencies for zero load) are approximately 15 and 37 cycles for the complete graph and the mesh, respectively, when using the 5-stage Garnet model.
- Despite the low injected load, Blackscholes has high values for injection latency, particularly with pthreads, which means that messages are unable to enter the network easily. We observed that this is caused by a particular hotspot pattern with the pthreads version.

IV. Conclusions

These set of experiments presented in this work analyzes, at a first approach, the beneficial effects of using task-based programming models instead of traditional ones based on threads on modern CMPs. We have seen that OmpSs performs better, independently of the interconnection network selected. In general, OmpSs generates a higher volume of memory traffic despite better exploiting locality. Presumably, such higher traffic is caused by the overhead introduced by its runtime. OmpSs shorter execution times generate larger injection rates, causing a higher stress in the network.

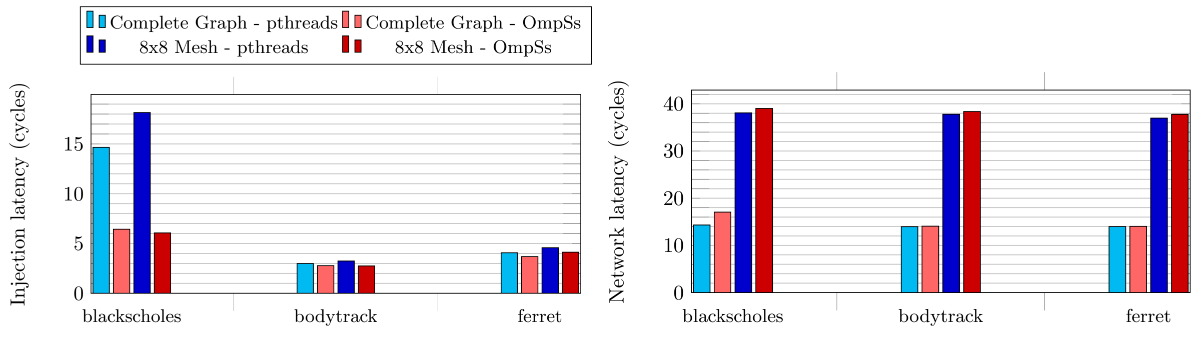
As future work, our intention is to evaluate both models under more resource-constrained networks, such as concentrated meshes. Such topologies reduce average distance and therefore latencies, what typically increases performance under modest concentration levels. However, because of the higher network load observed when using the OmpSs model, employing concentrated topologies might cause an increase of congestion and a reduction of performance with those programming models.



(a) Injected flits.

(b) Injection rate.

Figure 6: Injected flits into the network and injection rate defined as the number of flits per cycle and node, considering a node as all the injectors connected to the same router



(a) Injection latency.

(b) Network latency.

Figure 7: Injection and network latencies. *Injection latency* is defined as the delay from message creation to message injection into the network. *Network latency* is the delay of a message from its injection to its ejection to the destination node queue.

Acknowledgment

This work has been supported by the Spanish Government (Severo Ochoa grants SEV2015-0493, SEV-2011-00067), by the Spanish Ministry of Science and Innovation (contracts TIN2015-65316-P, TIN2012-34557, TIN2013-46957-C2-2-P), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272) and the European HiPEAC Network of Excellence. The research leading to these results has received funding from the European Communitys Seventh Framework Programme (FP7/2007-2013) under the Mont-Blanc Project (http://www.montblanc-project.eu), grant agreement n° 610402, and RoMoL project, GA 321253. M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047. M. Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243). E. Castillo has been partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2012/2254.

References

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 33–42. IEEE, 2009.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[5] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[6] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero. PARSECSs: Evaluating the impact of task parallelism in the parsec benchmark suite. *ACM Trans. Archit. Code Optim.*, 12(4):41:1–41:22, Dec. 2015.

[7] L. Dagum and R. Enon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[8] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[9] R. Vidal, M. Casas, M. Moretó, D. Chasapis, R. Ferrer, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero. Evaluating the impact of openmp 4.0 extensions on relevant parallel workloads. In *IWOMP*, pages 60–72, 2015.

Technology emerging from the DEEP & DEEP-ER projects

Estela Suarez
Jülich Supercomputing Centre (JSC)
Germany

Abstract— **Striving at pushing the applications scalability to the limits, the DEEP project proposed an alternative heterogeneous HPC architecture: It matches the different concurrency levels existing in large and complex simulation codes into the hardware. In this architecture, a standard HPC cluster is connected to the so-called "Booster": a cluster of autonomous accelerators. While the highly scalable parts of the application codes run on the energy efficient Booster, their low/medium scalable parts profit from the high single-thread performance of the Cluster side of the system.**

Extending this concept, the DEEP-ER project adds to the Cluster-Booster architecture a multi-level memory hierarchy exploiting non-volatile memory technologies, with the ultimate goal of improving the I/O and resiliency capabilities of the system to better support data intensive applications.

Additional to their hardware developments culminating in various prototype systems, both the DEEP and DEEP-ER projects have developed innovative software packages to support the newly introduced architecture, and extended existing programming environments preparing them for future heterogeneous Exascale systems.

This keynote will highlight some of the hardware and software technologies emerging from the European Exascale projects DEEP and DEEP-ER.

Accelerating High-Throughput Computing through OpenCL

Andrei Dafinoiu, Joshua Higgins, Violeta Holmes
High-Performance Computing Research Group
University of Huddersfield
Huddersfield, United Kingdom

Abstract—**As the computational trend diverges from standard CPU computing, to encompass GPUs and other accelerators, the need to integrate these unused resources within existing systems becomes apparent. This paper presents the implementation of an HTCondor pool with GPU execution capabilities through OpenCL. Implementation is discussed from both the system setup and the software design standpoint. The GPU landscape is investigated and the efficiency of the system is evaluated via Fast-Fourier Transform computations. Experimental results show that HTCondor GPU performance matches a dedicated GPU cluster.**

I. Introduction

In more recent years, the computing environment has seen a shift from the traditional CPU based computing onto a much diverse, and more parallel architecture such as GPUs. As of November 2015, two of the top ten supercomputers make use of GPU accelerated computing[1].

However, not all computational tasks are based on raw execution performance. Thus the emergence of High-Throughput Computing (HTC), a branch focusing on computational tasks that require the use of resources over an extended period of time.

The HTC community is not concerned about the execution rate of jobs, but rather, the parallelism of discrete jobs, given a much larger pool of resources. The interest is in how many jobs can be executed over a given amount of time, not the execution speed of a single job. HTC can take advantage of opportunistic resources, for example idle PCs in a University campus, to execute tasks, thus ”stealing” CPU time[2].

OpenCL is a heterogeneous programming framework for the development of applications that span across multiple architectures, including CPUs, GPUs, DSPs, and other accelerators. Whilst, from a computing standpoint, GPUs are being used as accelerators in supercomputers, the commodity, general purpose GPUs (GPGPUs) available in typical workstations may not have a significant impact on the fast processing required for High Performance Computing. In HTC, the fast performance of individual units is not the most important consideration, thus making HTC a better candidate for GPGPU based acceleration[3].

Previous work has shown that middleware vendors support the idea of GPU computing in HTC through built-in detection methods of GPU capabilities, as it is the case with newer versions of HTCondor[4]. Typically this relies on CUDA, the proprietary GPU computing framework developed by NVidia. However, CUDA is limited to newer NVidia GPUs, while also requiring special packages to be installed on the system in order to function[5]. In terms of general purpose PCs available on a university campus, only a small number of workstations may be capable of supporting CUDA. In order to exploit a much larger and more diverse mix of computing resources, OpenCL is a framework that can be used to exploit a much larger pool of devices than CUDA.

While attempts have been made to create computing environments using commodity GPUs, these implementations typically operate as dedicated clusters with the sole purpose of emulating typical GPU clusters, thus requiring large investments to develop[6].

A significant number of UK universities deploy HTC pools through HTCondor, including Oxford Unversity, Cambridge University, and Manchester University[7], however there is limited research output indicating the development of GPU integration within these pools.

This research aims to expand the capabilities of the existing HTC system by enabling use of the GPU resources in addition to the CPU resources of the pool, to be utilised during periods when they would otherwise be idle, supplementing the dedicated GPU computing cluster. To evaluate the ease-of-use, efficiency and flexibility of the OpenCL framework across the heterogeneous HTCondor pool, an FFT computation test case is implemented.

By increasing the amount of GPU resources available to researchers it is anticipated that the system would be more appealing, and GPU computing would be more accessible. This, in turn, would encourage some of the researchers using CPU intensive programs to shift to GPU instead, and improve utilisation of dormant resources.

II. Background

A. HTCondor

HTCondor is a workload management system used primarily for executing processes in an opportunistic environment, where tasks are distributed between resources as they become available, allowing the user to take advantage of otherwise idle resources. It features execution queues, job scheduling, priority, and resource discovery and management[2].

HTCondor employs cycle-stealing, the ability to use general workstations whilst they are unused for their specific purpose and would otherwise sit idle. HTCondor also contains a checkpoint system that allows it to migrate jobs to different machines once a machine starts to be used by the owner.

Submitted jobs are matched to resources by using the ClassAd mechanism, a framework that allows both jobs and machines to specify requirements and or preferences in regards to resource allocation.

The University of Huddersfield implements an HTCondor pool across all workstations available on campus, ranging from low-end library dedicated PCs to high-performance workstations in the design laboratories[7]. The university has a policy in place setting the minimum core count of a processor to 4, however no restrictions apply to the GPU components. The system is accessed via SSH authentication on the HTCondor headnode, and user data storage is offered via a GLUSTER based mirrored storage.

The HTCondor pool actively updates available resources, with offline machines being removed from the HTCondor pool. There are approximately 7000 CPU cores registered within the pool, however, the number of available cores changes constantly based on the availability of PCs on campus. On average, between 700 and 3000 are available for opportunistic jobs at peak times.

B. *OpenCL*

OpenCL is an open source framework for developing heterogeneous programs, created by a consortium of leading companies to develop standards for graphic acceleration and parallel computation, which has grown to include most commodity CPU/GPU providers, as well as specialised accelerators in the form of FPGAs and DSPs[3].

Since OpenCL has wide device support, workstation specifications are not as relevant at the programming stage of development, making OpenCL a viable solution for environments that contain more than one architecture or operating system. Its ability to operate using base drivers promises a much faster integration with existing systems.

Development of OpenCL applications is split into two different files. The host file is a C code that deals with the outer control logic of the system, dispatching work kernels to the compute units, controlling memory read/writes and executing the serial segments of the code. The kernel is also written in C, however it also incorporates OpenCL specific syntax. It represents the parallel component of the program, to be executed on the compute nodes, and can be compiled at run time by the host, to allow for a more diverse execution environment.

The flexibility of OpenCL allows for the creation of programs targeting machines with CPUs and GPUs (used individually or together), CPUs and other accelerators, or dedicated clusters that contain multiple GPUs (or CPUs) within each node.

III. OpenCL on HTCondor

In order to evaluate feasibility of OpenCL within an HTCondor pool, a test case was designed around the use of Fast-Fourier Transforms, as they are the basis for many computational algorithms in scientific software[8].

A. *Configuring Condor*

An HTCondor job must be scheduled for execution within a ’slot’. The default implementation advertises separate slots for each CPU core in a machine. The HTCondor ClassAd could be modified to add information about the GPU resource. HTCondor recommends that the GPU is appended to one of the CPU slots,[9] so that a machine may run both OpenCL and regular CPU jobs at the same time, as can be seen in Figure 1. During preliminary testing of this environment, it was determined that such an implementation slows down the OpenCL execution, for example where multi core CPU parallelism is employed, hence a different approach was considered.

Therefore, we configured a slot that allows an OpenCL program to block the entire machine, using all available resources. This was done alongside the existing slots, allowing a computer to either act as a CPU resource, advertising each CPU node individually, or as a single OpenCL entity, as shown in Figure 1. This dual setup is mutually exclusive, meaning that a machine that has started running a CPU job will be unavailable for OpenCL jobs, and vice versa.

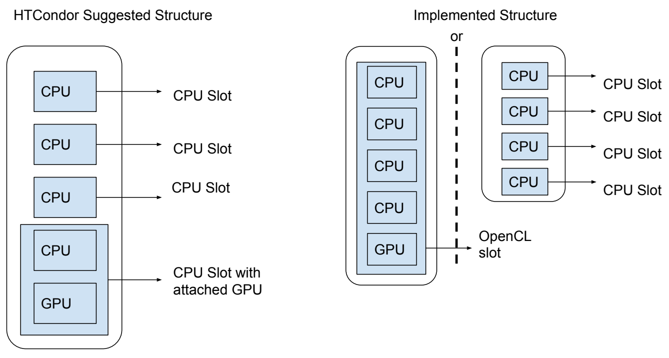


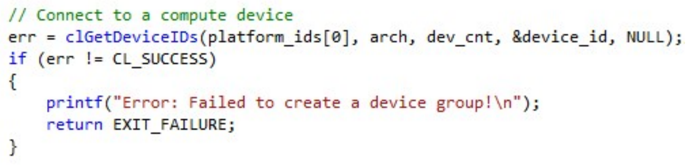
Fig. 1. HTCondor Slot Structure

B. *Implementing OpenCL*

The host file is created in order to setup the OpenCL environment, control execution and manage data transfers. Multiple segments of the host file are reusable across different OpenCL programs, mainly those that deal in environment setup and data transfers. Memory sizes, work-flow and optimisations depend on the targeted architecture and device. Within the host file, the command queue which controls execution is defined. It creates the execution kernels and enqueues them, either sequentially or parallel, also transferring memory buffers between devices and the host.

The kernel file, executed on the compute device, represents the parallel segment of the program. It encapsulates the computations to be executed and is directly controlled by the host. The kernel file does not change across different architectures or devices, and is only influenced by the type of parallelism used. The kernel file functions as a C function, that is executed inside a loop, either synchronously or asynchronously. Kernels can be task parallel, executing multiple functions simultaneously or data parallel, executing a single function on multiple data elements.

There are two methods of compiling the kernel file. The file can be compiled before execution, at the same time as the host file, reducing setup latency but limiting execution to the chosen architecture and device. Alternatively, the host file can be allowed to compile the kernel at run time to allow for execution across multiple devices and architectures. Figure 2 illustrates the compute device selection segment of the host code. The arguments passed to the function determine the selected device. As an example, assuming that a machine has both a CPU and a GPU, to execute the kernel on one or the other only requires a change in the arch variable in Figure 2. In reality, this results in a functional, yet unoptimised program. However, this example emphasises the flexibility of the OpenCL.



```
// Connect to a compute device
err = clGetDeviceIDs(platform_ids[0], arch, dev_cnt, &device_id, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to create a device group!\n");
    return EXIT_FAILURE;
}
```

Fig. 2. OpenCL host code excerpt

C. *GPU discovery and landscape*

HTCondor implements a function for GPU detection, however this relies primarily on CUDA and while also offering minimal support for OpenCL, returns insufficient device details to make optimisation decisions. For this reason, a program was designed to poll the target computer for available OpenCL devices and record their specifications.

Since OpenCL can run on CUDA drivers, there is no need to implement a separate CUDA program. The detection program was executed on 1000 machines, and, as seen in Table I, the GPU landscape discovered is quite diverse. Of the investigated machines, 300 failed to return information about their GPUs. This could be due to a lack of up-to-date drivers supporting OpenCL, or unsupported GPUs. The majority of devices are mid-range GPGPUs, mostly released around 2011. However, the program also identified a number of much newer generation NVIDIA GPUs, which feature a large number of cores that are more suited for efficient parallel execution.

D. *Application*

The Fast-Fourier Transform technique is used to convert time-domain signals into frequency-domain signals, and is widely adopted by researchers in engineering and science. For this reason it was chosen as a test-case for the system, which will be used in the aforementioned fields[8].

To better represent a real-world use case, GPU benchmarking was executed in a live environment, meaning that HTCondor resources became available only when idle, and jobs stopped when users returned to their machines. Tests were run using using the OpenCL Fast-Fourier Transform library on

TABLE I
GPU LANDSCAPE

GPU	**Nr**
AMD 5600	8
NVIDIA Quadro K600	42
NVIDIA GTX 610	40
NVIDIA GTX 670	75
NVIDIA GTX 750 Ti	77
NVIDIA GTX 970	133
AMD 6500	137
AMD 6400	189
Not detected	299
Total	**1000**

single dimensional FFTs[10]. To exploit the high number of available compute cores present in GPUs, FFT calculations were batched together until they filled the available memory size. The results are displayed in GFLOPs, calculated based on the FFTW benchmarking methodology[11]. To ensure the accuracy of the results, each FFT calculation was iterated 1000 times.

The challenge when using HTCondor is that the resource used for execution is unknown in advance, thus increasing the difficulty of optimizing the application for each individual GPU found through the discovery. For this experiment, the lowest GPU specification was used, to ensure execution across the entire system. For example, executing the program with a batch size lower than the GPU maximum will not exploit the best performance, however, using a batch size higher than the maximum will prevent execution.

Initial testing of CPU performance has shown between 0.6 and 1.7 GFLOPs on a standard Intel i5 CPU, shown in Figure 3. CPU benchmarking of the FFT implementation was not executed on a similar scale to the GPU since it would be beyond the scope of this work. More extensive CPU benchmarking has already been carried out by [12].

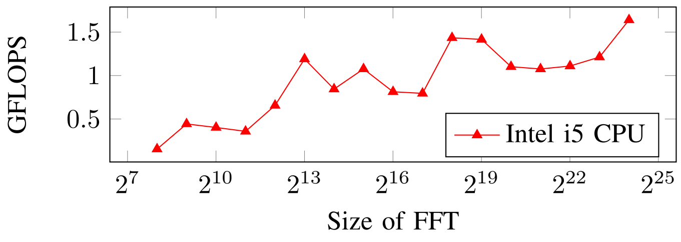


Fig. 3. CPU FFT execution

IV. Benchmarking Results

A. *Fast-Fourier Transforms on HTCondor GPUs*

The performance diversity of the GPU landscape can be noted in Figure 4, with the fastest GPU being an NVIDIA GTX 970 GPU, released in 2014, and the slowest being an AMD 5600 GPU, released in 2009. Newer generation GPUs benefit from higher clock speeds, more internal cores, and bigger memory buffers.

The average execution duration of the entire benchmark was 50 minutes. The first 40 % of jobs finished within 12 hours,

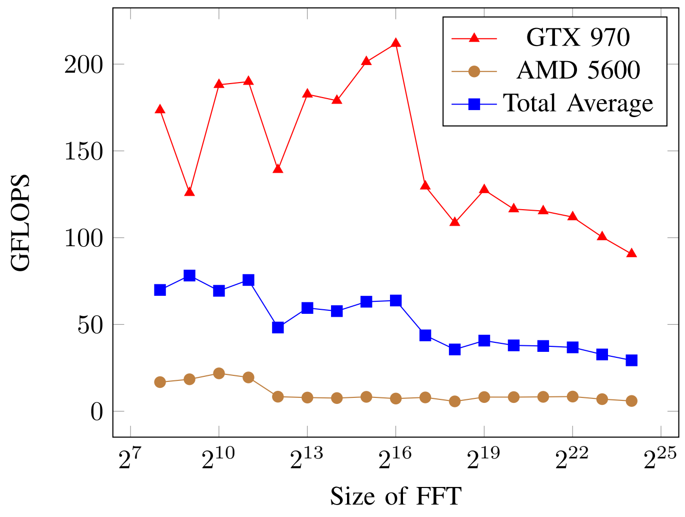


Fig. 4. GPU Benchmarking across 700 machines

with the other 60 % taking as long as 24 hours to complete. This is due to the fact that a job will be preempted when a real user accesses the machine.

B. Comparison with dedicated GPU cluster

In order to compare the available compute power of the GPUs inside HTCondor, the system needs to be compared against an existing GPU cluster. As such, the same benchmark program was executed on the NVIDIA C2050 GPU compute module inside VEGA, the dedicated GPU cluster at the UoH. In Figure 5, it can be seen that the average Condor GPU equals the performance of the single GPU compute module. However, the AMD GPUs are significantly slower overall, whilst some NVIDIA GPUs based on a newer architecture than the compute module achieve better performance.

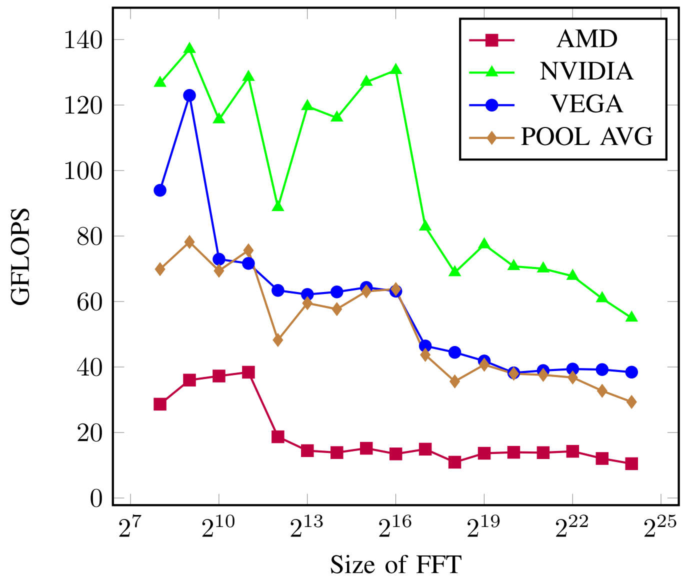


Fig. 5. Average Results by GPU Manufacturer

V. Summary

The successful integration of HTCondor with OpenCL has revealed a multitude of GPU resources that can be used to increase overall system performance for parallelizable applications, while also making GPUs more accessible in terms of physical usage and programming.

This work has shown that newer generation GPGPUs are able to match the performance of older dedicated GPU resources. However, due to the opportunistic nature of HTCondor, maximizing performance across such a system is difficult.

The flexibility and ease-of-use of OpenCL make it a promising framework for developing GPU applications across a diverse, and constantly evolving, environment.

VI. Future Work

Further develop the HTCondor implementation to facilitate access to GPU resources, by using the ClassAd system to advertise more GPU specific information for each machine, allowing users to optimise implementations for specific GPUs. Also, by exploiting the HTCondor ranking system, execution can be prioritised on machines with better capable GPUs foremost.

Improve performance of OpenCL applications by automating optimisations within the host file to increase performance within the HTCondor environment, whilst also documenting best-practise approaches and device-specific optimisations.

Acknowledgment

The authors would like to thank the QueensGate Grid at the University of Huddersfield

References

[1] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *IEEE Micro*, no. 5, pp. 7–17, 2011.

[2] M. Livny, J. Basney, R. Raman, and T. Tannenbaum, "Mechanisms for high throughput computing," *SPEEDUP journal*, vol. 11, no. 1, pp. 36–40, 1997.

[3] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 1-3, pp. 66–73, 2010.

[4] "condor gpu discovery." http://http://research.cs.wisc.edu/htcondor/manual/current/condor_gpu_discovery.html. Accessed: 2016-01-16.

[5] C. Nvidia, "Programming guide," 2008.

[6] S. Guba, M. Őry, and I. Szeberényi, "Harnessing wasted computing power for scientific computing," in *Large-Scale Scientific Computing*, pp. 491–498, Springer, 2013.

[7] D. Gubb, "Implementation of a condor pool at the university of huddersfield that conforms to a green it policy," Master's thesis, University of Huddersfield, 7 2013.

[8] V. U. Reddy, "On fast fourier transform," *RESONANCE*, vol. 3, no. 10, pp. 79–88, 1998.

[9] "Htcondorwiki: How to manage gpus in series seven." https://htcondor-wiki.cs.wisc.edu/index.cgi/wiki?p=HowToManageGpusInSeriesSeven. Accessed: 2015-12-03.

[10] "clfft: Opencl fast fourier transforms(ffts)." http://clmathlibraries.github.io/clFFT/. Accessed: 2016-02-11.

[11] "Fft benchmark methodology." http://www.fftw.org/speed/method.html. Accessed: 2015-12-13.

[12] C. Cullinan, C. Wyant, T. Frattesi, and X. Huang, "Computing performance benchmarks among cpu, gpu, and fpga," *Internet: www. wpi. edu/Pubs/E-project/Available/E-project-030212-123508/unrestricted/Benchmarking Final*, 2013.

Energy Efficiency Evaluation in Heterogeneous Computers

Borja Perez
Computer and Electronics Engineering Deptatment
Universidad de Cantabria
Email: perezpavonb@unican.es

Jose Luis Bosque
Computer and Electronics Engineering Deptatment
Universidad de Cantabria
Email: bosquejl@unican.es

Esteban Stafford
Computer and Electronics Engineering Deptatment
Universidad de Cantabria
Email: stafforde@unican.es

Ramón Beivide
Computer and Electronics Engineering Deptatment
Universidad de Cantabria
Email: beividej@unican.es

Abstract—**Heterogeneous systems have excellent properties both for performance and energy efficiency. However, the programming of these machines requires some effort to get the best results in massively data-parallel applications. This article presents a programming model with a set of load balancing algorithms that allow for a good workload distribution among all the devices of the system. In this way, none of the devices is idle, and all of them contribute computing power to the execution of the application. This not only reduces execution time, but also energy consumption, reaching a higher energy efficiency.**

I. Introduction

Current efforts towards increasing the computing power of modern machines is to include GPUs alongside general purpose processors. This is a solution adopted by desingers facing the challenge of the Exa-scale computing milestone, as the high processing power of GPUs allow these systems to increase the FLOPs per Watt ratio. Maintaining reasonable energy demands is a main concern in the Mont-Blanc project of the European Commision, which attempts to design an Exa-scale supercomputer based on low-power technologies [1].

The key of the energy efficiency of GPUs is that they are designed around massively parallel architectures which provide a staggering amount of computing power. However, this power can only be exploited through highly parallel workloads. Hence the need of general purpose processors and a way to conveniently program these heterogeneous systems.

Harnesing this heterogeniety is a considerable challenge by itself. The most common programming model, being used by CUDA and OpenCL, is the host-device model [2]. It dictates that the host processor starts the execution of an application and offloads highly parallel parts to the GPU. In general, the host waits for the completion of these code sections. Consequently, this model does not exploit the computing power of the host, as it remains idle while the GPU is in operation. Interestingly, the CPUs are still consuming a noticeable ammount of power during these periods, significantly reducing the energy efficiency of the system as a whole.

Maat [3] is an OpenCL library that hides from the developers all the different computing elements of a heterogeneous system. The applications can then be programmed using the host-device model, while Maat conveniently distributes the workload among all the available computing elements. It provides several load-balancing strategies to suit different workload scenarios.

To better understand the implications of the different strategies, this work provides an energy efficiency study of a heterogenous system with several CPUs and GPUs. The experimental results show that the H-guided strategy achieves very good results of energy efficiency, in terms of *Energy-Delay Product* (EDP), for all the applications analized.

II. Load Balancing Algorithms

Extracting the best performance out of heterogeneous systems is conditioned by an adequate use of all their resources. The following considerations are vital to making such an optimal workload distribution:

- Devices may have significant performance and architectural differences, unknown at design time.
- Accurately modelling the performance of the different devices is particularly difficult if the workload is irregular.
- Data communication with the devices is performed over slow interconnects.

The first two issues encourage the use of *dynamic* algorithms, which distribute the workload to the devices during runtime. Dynamic algorithms do not require prior information about the performance of the devices. However, the last consideration can impose a significant overhead.

For regular workloads, the first two issues are less relevant so a *static* algorithm is worth considering. This can give better results than the dynamic due to a minimzation of the communicantion overhead.

Trying to balance the above extremes, a third algorithm, named *h-guided*, has been implemented.

All three algorithms focus on the balancing of data parallel workloads, in which every device performs the same computation on a disjoint partition of the data. Thus, data are themselves the subjects of the distribution.

A. Static Balancing

This algorithm is based on dividing the load in as many portions as devices are available in the system. Then, it assigns a single portion to each of them. In order to obtain good performance, the load has to be divided in such a way as to make every device finish their computation simultaneously. Otherwise some devices will be idle while the computation is being completed. This is achieved by assigning each device a portion of work proportional to its computational power.

Let there be a heterogeneous system $H = \{D, P\}$, where $D = \{d_1, \cdots d_n\}$ is the set of devices and $P = \{P_1, \cdots P_n\}$ are the corresponding *computational powers*. P_i is defined as the amount of work that each device can complete in a time unit, including the necessary communication overhead. For a given application, these depend on the architecture of the devices, and are parameters that must be given to the model.

In order to perform the data partition, consider a given workload, that needs to process W *work-items* grouped in G *work-groups*. In OpenCL, each group can be executed concurrently as they do not require communication among them. Whereas threads within a group can synchronize among themselves. Consequently a work-group should not be split across devices, and it should be considered as the atomic distribution unit. All groups have the same size $L_s = \frac{W}{G}$, called *local work size*.

The response time of the heterogeneous system will be that of the last device to finish its work, $T_H = max_{i=1}^{N} T_{d_i}$. Therefore, the goal of the static method is to obtain a mapping of work-groups to devices, so that the workload is best balanced. This means, to find a tuple $\{\alpha_1, \cdots \alpha_n\}$, where α_i is the number of work-groups assigned to the device d_i, such that all the devices finish their work at the same time, and then the system response time is minimized:

$$T_H = T_{d_1} = \cdots = T_{d_n} \Rightarrow \frac{L_s \cdot \alpha_1}{P_1} = \cdots = \frac{L_s \cdot \alpha_n}{P_n}$$

Following the optimal algorithm proposed by O. Beaumont in [4] this can be done with complexity $O(n^2)$ in two steps:

- First, α_i is calculated so that $\frac{\alpha_i}{P_i}$ is almost constant $\forall\ i \in [1, \cdots n]$, and $\alpha_1 + \alpha_2 + \cdots + \alpha_n \leq G$:
$$\alpha_i = \left\lfloor \frac{P_i}{\sum_{i=1}^{n} P_i} \cdot G \right\rfloor$$
- Second, if $\sum_{i=1}^{n} \alpha_i < G$, then the remaining work-groups are assigned to the most powerful device. This amount of work is practically negligible, and does not disturb the load distribution.

This algorithm guarantees that the number of synchronization points is minimized, and performs well when facing regular loads, provided computational power of the devices are accurately known. However, it is not adaptable, so performance is not so good for irregular loads.

B. Dynamic Balancing

This algorithm divides the load in small, equally-sized packages, many more than the amount of available devices. The runtime orcherstation is carried out by a *master thread* that follows the next algorithm:

1) The master splits the number of work-groups G, in a set of p packages, all of them with the same size $Package\_size = \left\lfloor \frac{G}{p} \right\rfloor$. If G is not divisible by p, an extra package will have the remainder of the division.
2) The master launches one package on each device.
3) The master waits for the completion of any package.
4) When device d_i completes the execution of a package:
   a) The master stores results returned by the device.
   b) If there are outstanding packages the next package is assigned to device d_i.
   c) Else, if d_i is a GPU and there is a busy CPU d_i steals the package from the CPU.
   d) If none of this conditions is met, the master proceeds to step 5.
   e) The master returns to step 3.
5) The master ends as all the packages have been processed and their results are stored in the host.

This shows that the dynamic approach can addapt to different hardware and workload scenarios where the static can not. Attending to performance, the communication overhead must be taken into account. In the static, there is one package per device, but in the dynamic there are many more. Even if the data volume transferred to and from the devices is the same, the dynamic has a greater time dedicated to synchronization than the static.

C. H-guided Balancing

The h-guided algorithm strives to reduce the amount of synchronization points inherent to the dynamic scheme. It can be thought of a refinement of the latter, as it revolves about the same basic algorithm, only there is a difference in the size of the packages. These are not equal, but of diminishing size. On a first approach they can be computed as $Package\_size = \left\lfloor \frac{G_r}{n} \right\rfloor$. Where G_r is the number of pending work-groups and n is the number of available devices. This way, the packages are moderately big at the beginning of the execution and small at the end. This results in a reduction in synchronization points, while maintaining adaptability mostly at the end of the execution, when a finer-grained load distribution is needed. The size diminishes down to a minimum size that the algorithm must be provided with.

This solution has been used already in homogeneous systems, but when applied to heterogeneous machines it needs a further refinement. If there is a great difference in the computational power of the devices, a big package may be assigned to a slow device, delaying the completion of the whole program. To avoid this, the h-guided algorithm takes into account the computational power of the device, in a similar fashion to the static approach. Then, considering P_i as the computational power of device d_i, the size of the packages is calculated as:

$$Package\_size = \left\lfloor \frac{P_i}{\sum_{i=1}^{n} P_i} \cdot \frac{G_r}{n} \right\rfloor$$

III. Experimental Evaluation

The experiments presented in this section have been performed on a system with two GPUs, two CPUs and 64GBs of DDR3 memory. The CPUs are Intel Xeon E5-2670, with six cores that can run two threads at 2.0 GHz. The CPUs are connected via QPI, which results in OpenCL detecting them as a single device. Threfore, through the rest of this document, any reference to the CPU includes both Xeon E5-2670. The GPUs are two NVIDIA K20m, with 13 SIMD lanes and 5 GBytes of VRAM each.

Four applications have been chosen for the experiments. Two of them are part of the AMD APP SDK[5]. Both, NBody and MatMul, are regular applications in which different, equal-sized work units have the same running times. The other two applications, which are in-house implementations of known algorithms, are examples of irregular workloads, in which different, equal-sized work units may have different running times. First, RAP is an implementation of the Resource Allocation Problem, based on the one proposed by Acosta *et al.* [6]. It must be noted that there is a certain pattern in the irregularity of RAP, as each successive package represents a bigger amount of work than the previous. To test *Maat* using a truly irregular workload, a raytracing algorithm (RAY) was implemeted. This computes a realistic rendering of a scene by following light rays with independent threads. Thus, each of them represents an unpredictable amount of work, as the number of ray bounces depends on the objects of the scene.

The performance has been measured as the *speedup* with respect to the execution time using OpenCL and a single GPU. Due to the heterogeneoty of the system, the maximum speedup is not the number of devices, due to different computational power of each of them. Note that the CPUs are significantly less performant than the GPUs for the considered loads and that the performance difference between CPU and GPUs is application-dependent. To meassure the energy consumption, a monitor was developed that samples the power consumption of each device. This periodically measures the GPU power sensors through the NVIDIA Management Library (NVML) [7]. It also reads the Running Average Power Limit (RAPL) registers of the CPUs [8]. Finally, the energy efficiency is represented in terms of Energy-Delay-Product (EDP).

Figure 1 presents the speedup of each load balancing algorithm and benchmark referred to the performance of a single GPU. The speedup of the regular applications is limited by both the sensitivity to the number of packages and the performance differences between devices. The best algorithm for NBody is the static, although closely followed by the H-guided. However for MatMul, the poor performance in static can be blamed on the enormous difference in performance of GPUs and CPUs, making even the smallest work package too big for the CPU and so, it is better not to use the CPUs in this case. In the dynamic and H-guided methods the GPUs perform all the work, because the packages initially scheduled to CPUs are stolen. The analysis of the irregular applications, RAP and Ray, shows that the H-guided method obtains the best results. This is because the size of the packages assigned is proportional to the computing power of the device and inversely to the number of devices. Which means that more devices imply a greater amount of smaller packets, enabling a finer load balancing. This has a second consequence: the CPUs can contribute processing small and simple packages while the GPUs acomplish the big complex ones.

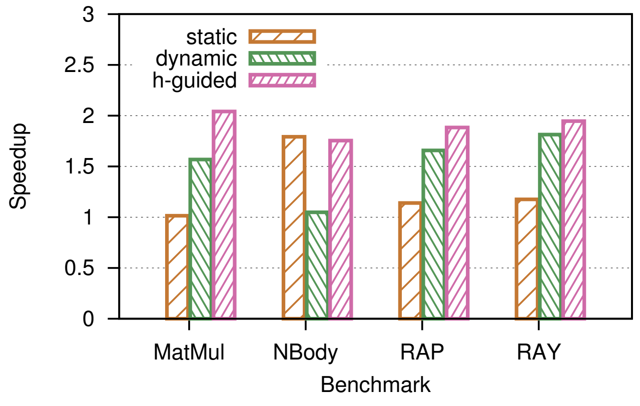


Fig. 1. Speedup for each algorithm relative to a single GPU.

Attending to energy considerations, Figure 2 shows the energy consumption of the system for each benchmark and load balancing algorithm (less is better). The bars on the graph are divided to indicate the energy corresponding to the CPU (12 cores) and the two GPUs. The bar labeled as ”base” corresponds to the results with one GPU. The first conclusion to highlight is that, despite using two GPUs, the total energy consumption of the heterogeneous system is less than the ”base”, except in the RAY benchmark. For all the rest, there is at least one load balancing algoritm that improves the ”base” energy consumption. This comes as a consequence of two improvements: the reduction in the execution time of the benchmarks, and that all the devices are contributing useful work. Thus, improving the energy efficiency of the system.

Energy saving is more noticeable in regular applications. In NBody the static algorithm reduces the base consumption in 22.5%. This is due to the regularity of the applications, the static algorithm can achieve an almost perfectly balanced distribution with minimal interaction between the devices. The H-guided saves up to 20% of the base energy in the MatMul benchmark. This is caused by regularity of the application and the small ammount of packages necessary to get a good balance. RAP, despite having greater speedup than NBody yields a smaller energy reduction, around 12.5%. This is caused by the fact that the irregularity of the application requires to work with a much larger number of work-packets to obtain a good load balancing. Which causes greater interaction between CPU and GPU, and increased energy consumption that really does not contribute to perform useful work. This situation is more pronounced in Ray Tracing, where no balancing algorithm makes the heterogeneous system consume less energy than the base system.

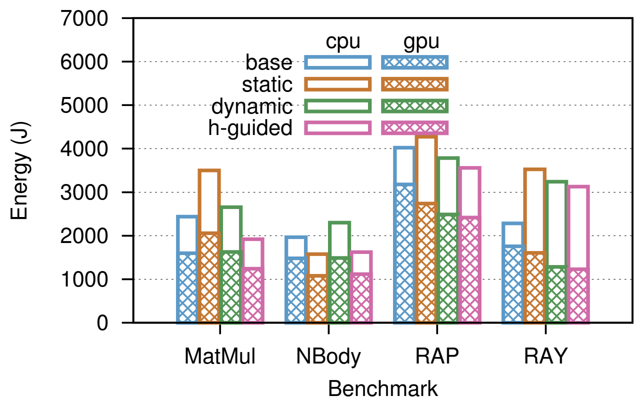


Fig. 2. Energy consumption for each algorithm and benchmark.

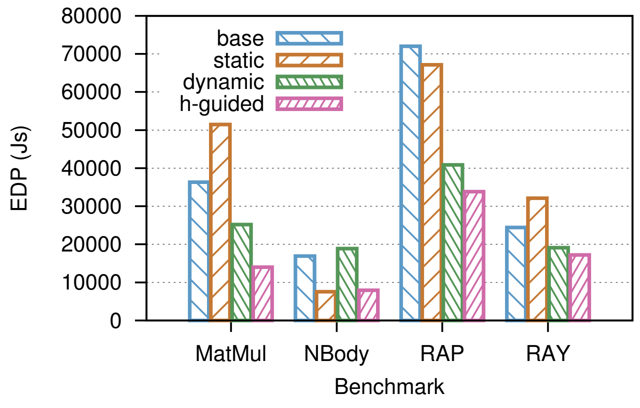


Fig. 3. Energy Delay Product (EDP) for each algorithm and benchmark.

Finally, Figure 3 shows the results of the energy efficiency in terms of the Energy-Delay Product. This metric combines the performance and energy metrics in one value. It can be seen that NBody with static as well as Matmul and RAP with the h-guided algorithm, obtain some important improvements in EDP with respect to the base system. However, the most interesting result is shown by the RAY benchmark. Despite showing an energy consumption 27% higher than the base system, the h-guided algorithm provided the best speedup, halving the computation time of the application. This also reduces the EDP for this benchmark, showing a considerable improvement, around 28% less than the base system.

IV. Conclusions

Nowadays, heterogeneous systems are commonly built by combining the computing power of general purpose CPUs with hardware accelerators (GPUs). This architecture provides very high performance thanks to the large number of cores available in the GPU coupled with a very low energy consumption, which maximizes energy efficiency. However, the exploitation of these systems pose a number of problems. For instance, the programming models, such as CUDA or OpenCL, are based on the Host-Device paradigm in which making a balanced load distribution is difficult.

By presenting a set of load balancing algorithms for massively data-parallel applications, this paper fulfills two important objectives. By harnesing the computing power of the whole heterogeneous machine, not only better performance is achieved, but energy efficiency is also improved.

The experimental results presented in this paper show that for both regular and irregular applications, there is always a load balancing algorithm that reduces the excution time. This is a logical consquence of the full system having greater computing power than a single GPU. Furthermore, the energy consumption of the machine with these algorithms is also reduced. In fact three of the four applications analyzed showed energy reductions. These savings are more notable in regular applications, because the interaction between CPU and GPU to obtain a balanced workload is lower. Finally, energy efficiency results, shown in terms of EDP, show that for all applications there is at least one load balancing algorithm that achieves substantial improvements.

The best overall results are obtained with the h-guided algorithm, yet the dynamic also gives very good results in irregular applications without prior knowledge of the power of the computing devices. The static algorithm is appropiate for homogeneous environments and regular applications.

Acknowledgment

This work has been supported by the Spanish Science and Technology Commission under contract TIN2013-46957-C2-2-P, the University of Cantabria, grant CVE-2014-18166, and the European HiPEAC Network of Excellence. The research leading to these results has received funding from the European Communitys Seventh Framework Programme (FP7/2007-2013) under the Mont-Blanc Project (http://www.montblanc-project.eu), grant agreement n° 610402.

References

[1] "Mont-Blanc project. Europen approach towards energy efficient high performance." last accesed April 2016. [Online]. Available: https://www.montblanc-project.eu/project/introduction

[2] B. R. Gaster, L. W. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL - Revised OpenCL 1.2 Edition*. Morgan Kaufmann, 2013.

[3] B. Pérez, J. L. Bosque, and R. Beivide, "Simplifying programming and load balancing of data parallel applications on heterogeneous systems," in *Proc. of the 9th Workshop on General Purpose Processing using GPU, Barcelona, Spain, March 12*, 2016, pp. 42–51.

[4] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, "A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers)," *IEEE Trans. Computers*, vol. 50, no. 10, pp. 1052–1070, 2001.

[5] "Amd accelerated parallel processing software development kit v2.9," last accesed November 2015. [Online]. Available: http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/

[6] A. Acosta, R. Corujo, V. Blanco, and F. Almeida, "Dynamic load balancing on heterogeneous multicore/multiGPU systems." in *HPCS*, W. W. Smari and J. P. McIntire, Eds. IEEE, 2010, pp. 467–476.

[7] NVIDIA, "NVIDIA Management Library (NVML)," last accesed April 2016. [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml

[8] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power management architecture of the 2nd generation Intel Core microarchitecture, formerly codenamed Sandy Bridge," in *IEEE Int. HotChips Symp. on High-Perf. Chips (HotChips˜2011)*, 2011.

INDEX

Copyright © Published by

Proceedings of the 2016 Emerging Technology Conference

2016
University of Manchester, Manchester, U.K.

ISBN 978-0-9933426-3-9