# PROCEEDINGS OF THE 3RD INTERNATIONAL CONFERENCE ON EXASCALE APPLICATIONS AND SOFTWARE

EASC 2015, 21st to 23rd April 2015, Edinburgh, UK

EDITED BY A GRAY, L SMITH & M WEILAND

Published by The University of Edinburgh

ISBN: 978-0-9926615-1-9

# Contents

Editors' Introduction ..... iii

Synthetic Program Analysis with Aspen
*Vetter & Meredith* ..... 1

From 11 to 14.4 PFLOPs: Performance Optimization for Finite Volume Flow Solver
*Hadjidoukas, Rossinelli, Hejazialhosseini & Koumoutsakos* ..... 7

The Impact of Process Placement and Oversubscription on Application Performance
*Wende, Steinke & Reinefeld* ..... 13

Radiative Transfer Modeling at High Performance Computers Using Self-Adjoint Transport Equation
*Olkhovskaya, Chetverushkin & Gasilov* ..... 19

Ensuring Efficiency of Exascale Supercomputer Centers
*Voevodin & Voevodin* ..... 24

Sniper: Simulation-Based Instruction-Level Statistics for Optimizing Software on Future Architectures
*Heirman, Isaev & Hur* ..... 29

Support Operators Technique for 3D Simulations of Dissipative Processes at High Performance Computers
*Gasilova, Poveshenko, Boldarev, Bagdasarov & Iakobovski* ..... 32

Efficiently Scheduling Task Dataflow Parallelism: A Comparison Between Swan and QUARK
*Vandierendonck* ..... 36

Portability and Performance of Nuclear Reactor Simulations on Many-Core Architectures
*Rahaman, Medina, Lund, Tramm, Warburton & Siegel* ..... 42

Exploiting Hierarchical Exascale Hardware using a PGAS Approach
*Fürlinger* ..... 48

Enabling Adaptive, Fault-tolerant MPI Applications with Dynamic Resource Allocation
*Szpindler* ..... 53

Evaluating Stencil Codes at Scale
*Modani, Ford, Johnson & Evangelinos* ..... 58

Exascale Computing for Everyone: Cloud-based, Distributed and Heterogeneous
*Inggs, Thomas, Hung & Luk* ..... 65

Flexible, Scalable Mesh and Data Management using PETSc DMPlex
*Lange, Knepley & Gorman* ..... 71

Prototyping for Exascale
*Vinter, Kristensen, Lund, Blum & Skovhede* ..... 77

ExaSHARK+GASPI: Reducing the burden to program large HPC systems since 2014
*Vander Aa, Chakroun, Wuyts, Rahn & Simmendinger* ..... 82

Towards Resilient Chapel
*Panagiotopoulou & Loidl* ..... 86

HPC and CFD in the Marine Industry: Past, Present and Future
*Mizzi, Kellet, Demirel, Martin & Turan* ..... 92

Swift: task-based hydrodynamics and gravity for cosmological simulations
*Theuns, Chalk, Schaller & Gonnet* ..... 98

Thread Parallelism for Highly Irregular Computation in Anisotropic Mesh Adaptation
*Rokos, Gorman, Jensen & Kelly* ..... 103

Paradigm Shift for EXASCALE Computing
*Matheou, Evripidou & Kyriacou* ..... 109

Large-scale Ultrasound Simulations Using the Hybrid OpenMP/MPI Decomposition
*Jaros, Nikl & Treeby* ..... 115

Algorithms in the parallel partitioning tool GridSpiderPar for large mesh decomposition
*Golovchenko, Kornilina & Yakobovskiy* ..... 120

Improving Performance Portability and Exascale Software Productivity with the Nabla Numerical Programming Language
*Camier* ..... 126

A highly scalable Met Office NERC Cloud model
*Brown, Weiland, Hill, Shipway, Maynard, Allen & Rezny* ..... 132

# Editors' Introduction

The 3rd International Conference on Exascale Applications and Software, EASC 2015, was hosted by EPCC, The University of Edinburgh (in cooperation with SIGHPC) from 21st-23rd April 2015 in Edinburgh.

The scale of today's leading HPC systems, which operate at the petascale, has put a strain on many simulation codes – both scientific and commercial. Only a small number of applications worldwide have, to date, demonstrated performance at the petaflop/s level. Many of the scientific challenges behind these codes are driving the need for the next generation of exascale HPC systems. Example scientific challenges originate from energy, climate, nanotechnology and medicine and are widely accepted to be of global significance. For simulation codes that are already struggling to scale up to petaflop levels, major investment is required to enable these codes to run at the exascale. Application optimisation and algorithmic modifications only represent part of this challenge. Systems of the scale envisaged present enormous challenges in terms of reliability, programmability, power consumption and usability. Programming models, libraries, languages, compilers and tools all need adaption and improvement. Applications must interact with many of these software aspects to be able to exploit exascale systems efficiently. The aim of this conference was to bring together all of the stakeholders involved in solving the software challenges of the exascale – from application developers, through numerical library experts, programming model developers and integrators, to tools designers.

The event was extremely successful in the dissemination of progress, the discussion of new ideas and the creation of new collaborations. This book contains a selection of proceedings from the conference, which we hope can disseminate the presented research to a wider audience.

Alan Gray, Lorna Smith and Michèle Weiland,
Editors

# Synthetic Program Analysis with Aspen

Jeffrey S. Vetter
Oak Ridge National Laboratory and
Georgia Institute of Technology
vetter@computer.org

Jeremy S. Meredith
Oak Ridge National Laboratory
jsmeredith@ornl.gov

## ABSTRACT

Our community is facing major challenges in the next decade: power, performance, resilience, and productivity. Emerging HPC systems have novel new features, like tightly-integrated heterogeneous computing and nonvolatile memory, that help solve one problem but at the cost of introducing considerable new complexity into the system design. Simply put, we see more complexity and uncertainty in emerging architectures than we have in the last two decades. Given this open design space, architects, applications scientists, and software designers need tools to estimate performance and resource requirements, while taking into account end-to-end design principles. In this paper, we demonstrate how our Aspen performance modeling language can be used to explore important properties of these application design spaces and inform the development of future architectures. We include examples from various applications, and show results from Aspen for idealized concurrency, memory capacity, computational intensity, and others.

## Keywords

Aspen, performance modeling, program analysis

## 1. INTRODUCTION

Our community is facing major challenges in the next decade: power, performance, resilience, and productivity. Although these challenges have been with us for some time, they are growing more acute as facilities and scientists are being confronted with a new level of complexity and required investment, deriving from the complex new architectures with a multitude of features and dynamically-controlled feedback. Poor decisions in architecting or procuring these next generation systems could have devastating consequences on scientists, sponsors, facilities, and vendors. In fact, many experts feel that we will experience the most uncertainty in computer architectures in two decades. Hence, it is imperative to have tools that facilitate end-to-end design [7] and design space exploration [10] of HPC systems.

### 1.1 Surveying the HPC Landscape

One need look no further than contemporary extreme scale systems being deployed and procured [11,12]. As more evidence, recent announcements of future HPC systems in DOE support this point. Titan, Aurora, and Summit will have entirely new features and configurations that have not be present in earlier systems. System software, programming environments, and applications must be improved to use these new systems. Given the current outlook as illustrated in Table 1, it is should be noted that no two architectures may be the same. Even if they have similar processors, the memory and storage hierarchy may be different. So, it is very important that these improvements be performance portable, in order to help hide this complexity.

In particular, several trends are already emerging: heterogeneous computing, nonvolatile memory, and small or no increases in storage bandwidth.

First, heterogeneous computing is apparent in many of today's top HPC systems. In earlier systems, such as Titan [2] and Tsubame2 [5], the addition of GPUs to systems gave performance improvements while keeping power constraints satisfied. As this capabilities evolves, we see tighter integrations of heterogeneous and special purpose capabilities onto general processors. For example, over the past several years, Intel has integrated GPUs, compression and encryption engines, random number generators, and other capabilities directly onto their main processors [1]. Although this functionality may be exposed to users in a number of ways, it will be imperative to provide portable solutions to HPC users. As seen in Table 1, both Titan and Summit will have heterogeneous ISAs within the node that scientists will need to carefully program and orchestrate. Meanwhile, the same applications will be expected to run on other platforms like Cori and Aurora.

Second, nonvolatile memory (NVM) systems in addition to alternative memory architectures are emerging as a solution to the limits of DRAM scaling, power, and cost [13]. Depending on the architectural solution, this change to the memory system could be more disruptive to applications teams than the change to heterogeneous computing. NVM devices have major differences from DRAM [13]: lower write durability; higher latencies and power costs for writes relative to reads; and, persistent state without the need for standby power. Again, as with heterogeneous systems, programming systems, system software, and architectures will need to hide these often subtle differences from applications. Although NVM devices have been transparently introduced into existing systems as replacements for hard-disk drives (HDD), they typically use existing I/O block-oriented interfaces, though the software stacks have been optimized. In the Summit and Aurora configurations, we must prepare applications for potentially tigher integration of these NVM devices with main memory and processors, bypassing the I/O interface [6,13].

In both of these cases, a number of open research questions about high-level system design remain. These ques-

Table 1: Contemporary HPC system configurations (estimates as of May 2015).

| | Titan | Cori | Summit | Aurora |
|---|---|---|---|---|
| Installation Date | 2012 | 2016 | 2017-8 | 2018 |
| Peak (PF) | 27 | >30 | 150 | >180 |
| Peak Power (MW) | 8.2 | <3.7 | 10 | 13 |
| Processor | AMD Opteron CPU + NVIDIA Kepler GPU | Intel Xeon Phi Knights Landing + Haswell (data) | POWER9 + NVIDIA Volta GPU | Intel Xeon Phi Knights Hill |
| Node Count | 18,688 | 9,300 (1,900 in data) | 3,400 | 50,000 |
| Node Main Memory (GB) | 32 | 64-128 GB DDR4; 16 GB High Bandwidth | 512 | >7PB all types, nodes |
| Node NVM (GB) | n/a | n/a | 800 | (incl above) |
| Storage Cap (PB) | 32, Lustre | 28, Lustre | 120, GPFS | > 150, Lustre |
| Storage BW (GBps) | 1,000 | 744 | 1,000 | > 1 |
| Interconnect | Gemini | Aries | Dual Rail EDR-IB | Intel Omni-Path |

tions include the amount of NVM versus DRAM memory, the number of latency-tolerant cores versus the number of throughput cores, and how much application data structures are a good fit for the characteristics of NVM when compared to DRAM?

Finally, storage systems continue to increase in capacity, but the aggregate storage bandwidth is only slowly increasing, if it is increasing at all. This trend will force users to consider other strategies for defensive checkpointing of application state (e.g., burst buffers), and post-processing and analysis of application output (e.g., in situ analysis). These changes could force major changes in application design, and perhaps the remainder of the architecture (e.g., increasing the amount of NVM for in situ analysis of time-series data [13]).

## 1.2 Contributions

To address these questions, we have developed a new methodology and tool for resource and performance prediction: Aspen. Aspen is a domain-specific programming language for performance modeling [8]. In this paper, we demonstrate how Aspen facilitates high level design and analysis of architectures and applications. Specifically, we make the following contributions.

1. We discuss the imminent challenges in the Introduction (§1).
2. We reinforce the importance of performance predictions and related tools in the coming wave of new HPC systems.
3. We discuss the range of performance prediction techniques and how they might help address these challenges.
4. We use our Aspen performance modeling language [8] to demonstrate how to draw insight into the end-to-end design of these new systems from important application metrics: computational intensity, memory usage, idealized concurrency, and others.

# 2. ASPEN

Aspen (Abstract Scalable Performance Engineering Notation) [8] is designed to allow simple construction of performance models through a domain-specific language which, like full programming languages, is flexible, supports composability. Aspen is a standard that makes it possible for scientists to share their work, including a formal methodology for application models and abstract machine models. An example of an Aspen kernel is shown in Listing 1; this example shows global parameters (the definition of $n$), a kernel definition (*FFT1D*) which contains computation ( the "execute" block) including defined parallelism ("n"), and a kernel definition (*FFT3Dstep*) which contains a parallel control flow ("map") which calls a number of one-dimensional FFTs in parallel.

The execute block is defined in terms of resource requirements, including bytes of memory and floating point operations. Traits for each resource requirement clarify how each resource is applied; for example, *simd* implies that the operations can use vector operations on a processor. Some resources specify sources/targets, such as *fftVol* being the source of bytes loaded as necessary to complete the operation.

There are several ways to generate a performance model for Aspen. They can be generated by hand, for instance; this manual process is the only feasible approach to model algorithms which are not yet codified. They can also be generated by hand even with source code available, but in this case, tools can remove some of the drudgery of performance modeling. Our COMPASS system [4] is just such a tool, using compiler-aided static analysis to generate Aspen models from source code.

# 3. RESULTS

## 3.1 Order Analysis

Using the COMPASS system, we generated performance models for a variety of benchmarks: Kernel Benchmarks (JACOBI, MATMUL, SPMUL, LAPLACE2D), NAS Parallel Benchmarks (CG), and Rodinia Benchmarks (BACKPROP, BFS, HOTSPOT, KMEANS, LUD, SRAD), and us-

```
param n = 1024

kernel FFT1D
{
   execute [n]
   {
     flops [5 * log2(n)]
         as dp, complex, simd
     loads [a * max(1, log(n)/log(Z))]
         of size [wordSize]
         from fftVol
   }
}

kernel FFT3Dstep
{
    map [n^2]
    {
        call FFT1D
    }
}
```

Listing 1: Aspen Kernel Example for 1D FFT

ing Aspen, we ran a performance prediction for each benchmark. As mentioned, one of the strengths of analytical performance modeling tools like Aspen is the ability to generate results *symbolically*. Here, we used Aspen to generate these equations and perform extra simplification on them, effectively treating all arithmetic involving constants as identity operations and simplifying them away.

For each benchmark, we identified key model parameters to leave as identifiers, but substituted values from application model and machine model parameters during the simplification process. Equations like $n*n+n*n*n$ get factored during the process into $(1+n)*(n*n)$, the constant elided, and finally simplified back to $n*n*n$. In essence, the results give us the order of the runtime (cf. Big O notation) in terms of key application parameters.

The results are shown in Table 2. As a concrete example, we see that the runtime of MATMUL (matrix multiply) returns an order of runtime of $N*M*P$; this is for a matrix multiply for matrices of size $N \times M$ and $M \times P$. In the case of square matrices (where $N == M == P$), this simplifies to $N^3$; a result we can easily validate against our expectations.

| Benchmark | Runtime Order |
|---|---|
| BACKPROP | $H*O+H*I$ |
| BFS | $nodes+edges$ |
| CFD | $nelr*ndim$ |
| CG | $nrow+ncol$ |
| HOTSPOT | $sim_time*rows*cols$ |
| JACOBI | $m\_size*m\_size$ |
| KMEANS | $nAttr*nClusters$ |
| LAPLACE2D | $n^2$ |
| LUD | $matrix\_dim^3$ |
| MATMUL | $N*M*P$ |
| NW | $max\_cols^2$ |
| SPMUL | $size+nonzero$ |
| SRAD | $niter*rows*cols$ |

Table 2: Order analysis, showing Big O runtime for each benchmark in terms of its key parameters.

## 3.2 Computational Intensity

The design of Aspen allows for the ability to represent and extract key features of computational kernels. One of the most common algorithmic features of interest in applications is *computational intensity*, the ratio of floating point operations to the number of bytes loaded. This can be used to gauge an algorithm's performance on architectures which are not defined in vast detail except for key features such as peak FLOPS rates and memory bandwidth. The well-known roofline plot is a common example — it shows system performance for a range of computational intensity values; if one is interested in measuring system performance in terms of FLOPS, one must have a sufficiently intensive algorithm.

To explore computational intensity for an application in detail, we took a COMPASS-generated model of the US DOE hydrodynamic proxy application, LULESH [3], and asked Aspen to return FLOPS:byte ratios for each major routine. The results are shown in Table 3. We see routines with both low and high computational intensity. Some even have zero; those routines simply exist to rearrange memory for future computation and perform no floating point operations at all.

| Method Name | FLOPS/byte |
|---|---|
| InitStressTermsForElems | 0.03 |
| CalcElemShapeFunctionDerivatives | 0.44 |
| SumElemFaceNormal | 0.50 |
| CalcElemNodeNormals | 0.15 |
| SumElemStressesToNodeForces | 0.06 |
| IntegrateStressForElems | 0.15 |
| CollectDomainNodesToElemNodes | 0.00 |
| VoluDer | 1.50 |
| CalcElemVolumeDerivative | 0.33 |
| CalcElemFBHourglassForce | 0.15 |
| CalcFBHourglassForceForElems | 0.17 |
| CalcHourglassControlForElems | 0.19 |
| CalcVolumeForceForElems | 0.18 |
| CalcForceForNodes | 0.18 |
| CalcAccelerationForNodes | 0.04 |
| ApplyAccelerationBoundaryCond | 0.00 |
| CalcVelocityForNodes | 0.13 |
| CalcPositionForNodes | 0.13 |
| LagrangeNodal | 0.18 |
| AreaFace | 10.25 |
| CalcElemCharacteristicLength | 0.44 |
| CalcElemVelocityGrandient | 0.13 |
| CalcKinematicsForElems | 0.24 |
| CalcLagrangeElements | 0.24 |
| CalcMonotonicQGradientsForElems | 0.46 |
| CalcMonotonicQRegionForElems | 0.21 |
| CalcMonotonicQForElems | 0.21 |
| CalcQForElems | 0.39 |
| CalcPressureForElems | 0.08 |
| Release | 0.04 |
| CalcEnergyForElems | 0.10 |
| CalcSoundSpeedForElems | 0.13 |
| EvalEOSForElems | 0.09 |
| ApplyMaterialPropertiesForElems | 0.09 |
| UpdateVolumesForElems | 0.13 |
| LagrangeElements | 0.22 |
| CalcCourantConstraintForElems | 0.14 |
| CalcHydroConstraintForElems | 0.20 |
| CalcTimeConstraintsForElems | 0.16 |
| LagrangeLeapFrog | 0.19 |

Table 3: Computational intensity (bytes loaded or stored per floating point operation) for methods in in the LULESH proxy application.

## 3.3 Memory Usage

Another algorithmic feature commonly of interest is the amount of memory used by an algorithm. More useful than merely the total memory used by the application, a more detailed per-kernel breakdown allows a number of further analyses. For example, armed with the size of all arrays touched by each kernel, and the CPU and GPU performance of each kernel, we can search for an optimal combination of kernels which should be offloaded to a GPU – specifically, the set which still fits in GPU memory and results in the best overall performance (including not just per-kernel CPU/GPU performance, but also any necessary PCI-Express transfer times for all arrays which are required on both host and device).

An example analysis of each major routine in LULESH is shown in Table 4. Note that exclusive memory usage is self-only, and inclusive memory usage includes any arrays used by any methods called directly or indirectly for each routine. Variables passed directly to a routine are not counted, and so some even computationally-intense routines have an exclusive memory footprint of zero.

| Method Name | Memory Usage | |
|---|---|---|
| | Exclusive | Inclusive |
| InitStressTermsForElems | 3.6e+06 | 3.6e+06 |
| CalcElemShapeFunctionDerivatives | 0 | 0 |
| SumElemFaceNormal | 0 | 0 |
| CalcElemNodeNormals | 0 | 0 |
| SumElemStressesToNodeForces | 1.7e+07 | 1.7e+07 |
| IntegrateStressForElems | 2.9e+07 | 2.9e+07 |
| CollectDomainNodesToElemNodes | 2.3e+06 | 2.3e+06 |
| VoluDer | 0 | 0 |
| CalcElemVolumeDerivative | 0 | 0 |
| CalcElemFBHourglassForce | 0 | 0 |
| CalcFBHourglassForceForElems | 6.6e+07 | 6.6e+07 |
| CalcHourglassControlForElems | 4.2e+07 | 7.0e+07 |
| CalcVolumeForceForElems | 0 | 7.3e+07 |
| CalcForceForNodes | 2.3e+06 | 7.3e+07 |
| CalcAccelerationForNodes | 5.4e+06 | 5.5e+06 |
| ApplyAccelerationBoundaryCond | 2.4e+06 | 2.4e+06 |
| CalcVelocityForNodes | 4.7e+06 | 4.7e+06 |
| CalcPositionForNodes | 4.7e+06 | 4.7e+06 |
| LagrangeNodal | 0 | 7.7e+07 |
| AreaFace | 0 | 0 |
| CalcElemCharacteristicLength | 0 | 0 |
| CalcElemVelocityGrandient | 192 | 192 |
| CalcKinematicsForElems | 1.1e+07 | 1.1e+07 |
| CalcLagrangeElements | 2.9e+06 | 1.1e+07 |
| CalcMonotonicQGradientsForElems | 1.3e+07 | 1.3e+07 |
| CalcMonotonicQRegionForElems | 7.3e+06 | 7.3e+06 |
| CalcMonotonicQForElems | 0 | 7.3e+06 |
| CalcQForElems | 0 | 1.9e+07 |
| CalcPressureForElems | 3.6e+06 | 3.6e+06 |
| Release | 7.3e+05 | 7.3e+05 |
| CalcEnergyForElems | 1.1e+07 | 1.2e+07 |
| CalcSoundSpeedForElems | 4.7e+06 | 4.7e+06 |
| EvalEOSForElems | 1.4e+07 | 1.7e+07 |
| ApplyMaterialPropertiesForElems | 2.6e+06 | 1.9e+07 |
| UpdateVolumesForElems | 1.5e+06 | 1.5e+06 |
| LagrangeElements | 0 | 3.7e+07 |
| CalcCourantConstraintForElems | 0 | 2.6e+06 |
| CalcHydroConstraintForElems | 0 | 1.1e+06 |
| CalcTimeConstraintsForElems | 2.2e+06 | 2.6e+06 |
| LagrangeLeapFrog | 0 | 1.0e+08 |

Table 4: Memory array bytes touched within a method (exclusive) and within either a method or its callees (inclusive) in the LULESH proxy application.

## 3.4 Ideal Concurrency

Ideal concurrency is the peak amount of parallelism available within an algorithm. Ideal concurrency can vary over time, and rise and fall based on the stage of a program's execution.

Aspen can measure ideal concurrency of an application independently of any particular machine. The process by which it detects this information is by walking the control flow of the program, and within each kernel, finding the total task and data parallelism expressed in the control constructs for that kernel, and multiplying that by the ideal concurrency of its callers. (A kernel called by two parent kernels may have two different ideal concurrencies.)

Again, note that this is *ideal* concurrency, and is thus a feature of the algorithm itself, but it does inform system design in that parallelism beyond the ideal concurrency will not result in an improvement to runtime. It also can inform algorithm optimization; if some phase of the application has a low concurrency, it is likely making poor use of the available resources.

An analysis of one application, the streaming sensor challenge problem featuring synthetic radar aperture (SAR) processing [9], is shown in Figure 1, for three different tile sizes. Note that larger tile sizes result in greater concurrency in some phases, but lower concurrency in others. The ability to extract and display this information concisely in Aspen is key to helping application developers understand performance implications on a variety of scalable systems.

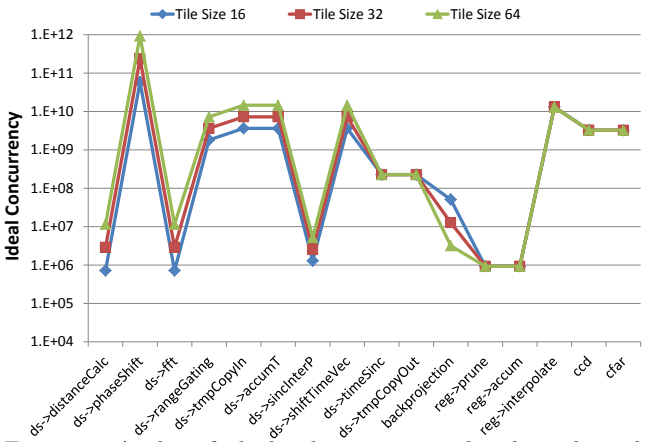



Figure 1: A plot of idealized concurrency by chronological phase in the digital spotlighting application model.

## 3.5 Performance Profiling

A crucial aspect to analytical performance analysis is the ability to generate predictions without needing source code or even access to the target hardware. One logical manifestation of this idea is the ability to generate program analyses which mimic that of real performance analysis tools.

For example, one very commonly used performance analysis tool is the GNU Profiler, *gprof*. One of the synthetic program analysis tools we created with Aspen, then, was a gprof-like tool. An example of this is seen in Figure 2. Note that the types of analyses needed to achieve this output include counts of calls to each kernel, per-kernel runtimes, both exclusive (self-only) and inclusive of called routines, and call-graph analysis.

```
Flat profile:

  %    cum    self        self      total
 time  sec    sec  calls ms/call   ms/call   name
-----------------------------------------------------------
86.91 370.76 370.76  30 12358.52   12358.52  fft3d.localFFT
10.03 413.54  42.78  20  2139.09    2139.09  fft3d.exchange
 3.06 426.57  13.03  20   651.71     651.71  fft3d.shuffle
 0.00 426.57   0.00  10     0.03       0.03  exchange
 0.00 426.57   0.00  10     0.03       0.03  buildNList
 0.00 426.57   0.00  10     0.01       0.01  ljForce
 0.00 426.57   0.00  30     0.00       0.00  integrate
 0.00 426.57   0.00  10     0.00   42657.18  fft
 0.00 426.57   0.00  10     0.00   42657.18  fft3d.main
 0.00 426.57   0.00   1     0.00  426572.70  main

Call graph:

index  %time     self  children       name
-----------------------------------------------------------
[  1]  100.0     0.00   426.57        main [1]
                 0.00     0.00            buildNList [8]
                 0.00     0.00            exchange [7]
                 0.00   426.57            fft [2]
                 0.00     0.00            integrate [10]
                 0.00     0.00            ljForce [9]
-----------------------------------------------------------
                 0.00   426.57            main [1]
[  2]  100.0     0.00   426.57        fft [2]
                 0.00   426.57            fft3d.main [3]
-----------------------------------------------------------
                 0.00   426.57            fft [2]
[  3]  100.0     0.00   426.57        fft3d.main [3]
                42.78     0.00            fft3d.exchange [5]
               370.76     0.00            fft3d.localFFT [4]
                13.03     0.00            fft3d.shuffle [6]
-----------------------------------------------------------
                 0.00   426.57            fft3d.main [3]
[  4]   86.9    370.76     0.00        fft3d.localFFT [4]
-----------------------------------------------------------
                 0.00   426.57            fft3d.main [3]
[  5]   10.0    42.78      0.00        fft3d.exchange [5]
-----------------------------------------------------------
                 0.00   426.57            fft3d.main [3]
[  6]    3.1    13.03      0.00        fft3d.shuffle [6]
-----------------------------------------------------------
                 0.00   426.57            main [1]
[  7]    0.0     0.00      0.00        exchange [7]
-----------------------------------------------------------
                 0.00   426.57            main [1]
[  8]    0.0     0.00      0.00        buildNList [8]
-----------------------------------------------------------
                 0.00   426.57            main [1]
[  9]    0.0     0.00      0.00        ljForce [9]
-----------------------------------------------------------
                 0.00   426.57            main [1]
[ 10]    0.0     0.00      0.00        integrate [10]
```

Figure 2: Synthetic *gprof* output from molecular dynamics application model.

## 3.6 Sensitivity Analysis

Another strength of analytical modeling techniques is their ability to easily generate predictions for hardware which does not (yet) exist. One can easily create abstract machine models for new systems with proposed architectural parameters and quickly evaluate existing application models on these new systems. To investigate this application of Aspen, we used models of two US DOE proxy applications, the hydrodynamics application LULESH and molecular dynamics application CoMD, and generated predictions using a model of a contemporary hardware system. We then automatically explored a range of hardware parameters (e.g., clock speeds, bus widths, latencies) to see how sensitive the application was to each parameter.

Here, we calculate sensitivity ($S$) as the ratio of improvement in application runtime to the improvement in modified hardware parameter.

$$S = \frac{I_{runtime}}{I_{param}}$$

Each of these improvements is itself a ratio, for example:

$$I_{runtime} = \frac{runtime_{orig}}{runtime_{new}}$$

$$I_{param} = \frac{param_{orig}}{param_{new}}$$

Note that while a decrease in runtime is always an improvement, hardware parameters are sometimes improved with a decrease in value (e.g., latency), and others are sometimes improved with an increase in value (e.g., clock speed), and so we invert $I_{param}$ based on the type of parameter so that a higher value of $I_{param}$ is always an improvement.

As a concrete example of sensitivity, suppose that we double clock speed (i.e., $I_{param} = 2.0$), and see a decrease in runtime by a factor of 2× (i.e., $I_{runtime} = 2.0$). This results in a sensitivity of 100% for this application to clock speed.

The results for LULESH and CoMD are shown in Figure 3. We see, for instance, that both applications are largely bottlenecked by memory performance on both CPU and GPU, as memory subsystem improvements showed the greatest application performance improvement (i.e., showed the highest sensitivity). However, we also predict that LULESH is more sensitive to memory latency while CoMD is more sensitive to memory bandwidth. This type of information is useful to predict architectural changes which would result in the greatest benefit to exist applications in, for example, the hardware/software codesign process.

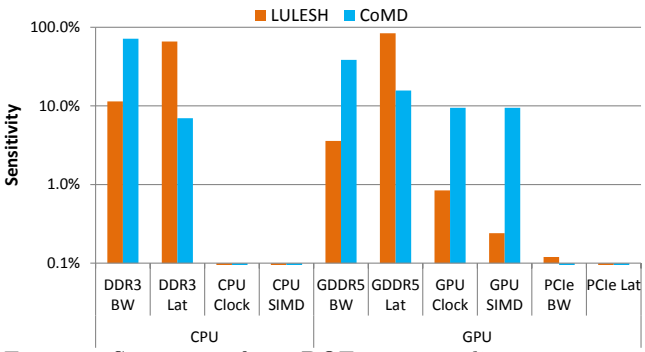



Figure 3: Sensitivty of two DOE proxy applications to various hardware parameters.

# 4. CONCLUSIONS

In this paper, we have reviewed major challenges for extreme-scale HPC in the next decade: power, performance, resilience, and productivity. Emerging HPC systems have novel new features, like tightly-integrated heterogeneous computing and nonvolatile memory, that help solve one problem but at the cost of introducing considerable new complexity into the system design. Simply put, we see more complexity and uncertainty in emerging architectures today than we have in the last two decades. Given this open de-

sign space, architects, applications scientists, and software designers need methods and tools to estimate performance and resource requirements, while taking into account end-to-end design principles. In this paper, we demonstrated how our Aspen performance modeling language can be used to explore important properties of these application design spaces and inform the development of future architectures. We included examples from various applications, and showed results from Aspen for idealized concurrency, memory capacity, computational intensity, and others.

## 5. ACKNOWLEDGMENTS

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research.

## 6. REFERENCES

[1] "5th generation intel core processors based on the mobile u-processor line." [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/platform-briefs/5th-gen-core-mobile-u-processor-platform-brief.pdf

[2] A. Bland, W. Joubert, D. Maxwell, N. Podhorszki, J. Rogers, G. Shipman, and A. Tharrington, "Titan: 20-petaflop cray XK6 at oak ridge national laboratory," in *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series, J. S. Vetter, Ed. Boca Raton: Taylor and Francis, 2013, vol. 1, p. 900.

[3] I. Karlin, A. Bhatele, B. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "LULESH programming model and performance ports overview," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep. LLNL-TR-608824, 2012.

[4] S. Lee, J. S. Meredith, and J. S. Vetter, "COMPASS: A framework for automated performance modeling and prediction," in *ACM International Conference on Supercomputing (ICS)*. Newport Beach, California: ACM, 2015.

[5] S. Matsuoka, T. Aoki, T. Endo, H. Sato, S. Takizawa, A. Nomura, and K. Sato, "TSUBAME2.0: The first petascale supercomputer in japan and the greenest production in the world," in *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series, J. S. Vetter, Ed. Boca Raton: Taylor and Francis, 2013, vol. 1, p. 900.

[6] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," Oak Ridge National Laboratory, Technical Report ORNL/TM-2014/633, 2014.

[7] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, 1984.

[8] K. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *SC12: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2012.

[9] K. Spafford, J. S. Vetter, T. Benson, and M. Parker, "Modeling synthetic aperture radar computation with aspen," *International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 255–262, 2013.

[10] K. L. Spafford and J. S. Vetter, "Automated design space exploration with aspen," *Scientific Programming*, vol. 2015, p. 10, 2015. [Online]. Available: http://dx.doi.org/10.1155/2015/157305

[11] J. S. Vetter, Ed., *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series. Boca Raton: Taylor and Francis, 2013, vol. 1.

[12] ——, *Contemporary High Performance Computing: From Petascale Toward Exascale*, 1st ed., ser. CRC Computational Science Series. Boca Raton: Taylor and Francis, 2014, vol. 2.

[13] J. S. Vetter and S. Mittal, "Opportunities for nonvolatile memory systems in extreme-scale high-performance computing," *Computing in Science & Engineering*, vol. 17, no. 2, pp. 73–82, 2015.

# From 11 to 14.4 PFLOPs: Performance Optimization for Finite Volume Flow Solver

Panagiotis E. Hadjidoukas
phadjido@mavt.ethz.ch

Diego Rossinelli
diegor@mavt.ethz.ch

Babak Hejazialhosseini
hbabak@mavt.ethz.ch

Petros Koumoutsakos
petros@ethz.ch

Chair of Computational Science, D-MAVT ETH Zürich, 8092 Switzerland

## ABSTRACT

CUBISM-MPCF is a compressible, two-phase flow solver that has performed unprecedented flow simulations, employing 13 trillion computational elements to study cavitation collapse of a cloud composed of 15'000 bubbles. The code had been deployed on 1.6 million cores of the Sequoia IBM BlueGene/Q supercomputer, reaching initially 11 PFLOPs, corresponding to 55% of its nominal peak performance. This paper reports, for the first time, the techniques used to extend the performance of the code by 30% reaching 14.4 PFLOPs on BlueGene/Q systems. The achieved 72% of the peak performance constitutes to date the best performance for flow simulations in supercomputer architectures. Our techniques take advantage of the underlying hardware capabilities and were applied through all levels in the software abstraction aiming at full exploitation of the inherent instruction/data-,thread- and cluster-level parallelism. The software advances by two to three orders of magnitude the state-of-the-art both in terms of time to solution and geometric complexity of the flow. We believe that the present methods are relevant to all grid based solvers and as such they may serve to enhance the capabilities across different areas of simulation based science.

## Keywords

High performance computing, flow simulations, supercomputers

## 1. INTRODUCTION

Vehicles operating with liquid fluids are the most dominant form of transportation and they account for more than 20% of the world's energy resources. Their energy efficient operation is of paramount importance as further reduction in $CO_2$ emissions requires improving the efficiency of internal combustion engines which in turn implies high-pressure fuel injection systems. Precise fuel injection control and enhanced fuel-air mixing implies high liquid fuel injection pressures. In such conditions, liquid fuel can undergo vaporization and subsequent re-condensation in the combustion chamber. Clusters of vapor bubbles incepted in such flow conditions are referred to as "cloud cavitation". Their collapse induces pressure peaks up to two orders of magnitude larger than the ambient pressure [10]. When such pressures are exerted on solid walls they can cause material erosion of the combustion chamber and limit the lifetime of the fuel injectors. The damaging effects of cloud cavitation collapse are also detrimental to the operation of marine propellers and turbomachinery yet they can be harnessed in medical applications ranging from kidney lithotripsy to drug delivery [7].

Realistic simulations require two phase flow solvers capable of capturing interactions between multiple deforming bubbles, pressure waves and shocks and their interaction with turbulent flow fields. CUBISM-MPCF is a high throughput software (ACM Gordon Bell Prize 2013) [8] that addresses challenges critical to flow simulations in terms of floating point operations, memory traffic and storage capacity. The software has been designed to take advantage of the features of the IBM BlueGene/Q (BGQ) platform to simulate cavitation collapse dynamics using up to 13 trillion computational elements. The performance of the software has been shown to reach an unprecedented 14.4 PFLOP/s on 1.6 million cores corresponding to 72% of the peak on the 20 PFLOP/s Sequoia supercomputer. Furthermore, the software introduces a first of its kind efficient wavelet based compression scheme, in order to decrease the I/O time and the footprint of the simulations. The scheme delivers compression rates up to 100 : 1 and takes less than 1% of the total simulation time.

As collapsing bubbles cover about 50% of the computational domain, we chose a uniform resolution over an adaptive mesh refinement [2] or a multi resolution technique [11] for the discretization of this flow field. By performing simulations that resolve collapsing clouds with up to 15'000 bubbles, CUBISM-MPCF improved by two orders of magnitude the previous state of the art, set by Adams and Schmidt [1]. Considering uniform resolution solvers, simulations of noise propagation of jet engines were performed on Sequoia using similar number of computational elements but with significantly lower performance in terms of time to solution and percentage of the peak [3]. Regarding performance, an earlier version of the present software achieved 30% of the nominal peak on 47k cores of Cray XE6 Monte Rosa [4] for studies of shock-bubble interactions.

In this work, we first discuss our key software design decisions for addressing simulation challenges with regard to floating point operations and memory traffic. Then, we present and evaluate optimization techniques that allowed us to improve the initial performance of CUBISM-MPCF from 55% to 72% of the theoretical peak on BGQ systems, which is translated to the increase from 11 to 14.4 PFLOP/s on Sequoia. These techniques take advantage of the underlying hardware capabilities and were applied at the three

abstraction layers of the software (*core, node, and cluster*), aiming at full exploitation of the inherent instruction/data-, thread- and cluster-level parallelism.

The paper is organized as follows: in Section 2 we briefly present the governing equations and their numerical discretization. In Sections 4 and 3 we present CUBISM-MPCF and summarize the main features of the BGQ platform, respectively. In Section5 we present the optimization techniques that improved the efficiency of our solver. Detailed performance results of the improved solver are presented in Section 6. Finally, we conclude in Section 7.

## 2. EQUATIONS AND DISCRETIZATION

Cavitation dynamics involve a complex interplay of physical processes associated with compressibility, convective and viscous dissipation effects. We simulated cavitation in inviscid, compressible, two-phase flows using a finite-volume discretisation of the governing Euler equations. The evolution of density, momenta and the total energy of the flow is described with the following system of equations:

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0, \\ \frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u}\mathbf{u}^T + p\mathbb{I}) &= \mathbf{0}, \\ \frac{\partial (E)}{\partial t} + \nabla \cdot ((E+p)\mathbf{u}) &= 0. \end{aligned} \tag{1}$$

The evolution of the vapor and liquid phases is determined by another set of advection equations:

$$\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = 0, \tag{2}$$

where $\phi = (\Gamma, \Pi)$ with $\Gamma = 1/(\gamma - 1)$ and $\Pi = \gamma p_c/(\gamma - 1)$. The specific heat ratio $\gamma$ and the correction pressure of the mixture $p_c$ are coupled to the system of equations (1) through a stiffened equation of state of the form $\Gamma p + \Pi = E - 1/2\rho|\mathbf{u}|^2$.

We discretize these equations using a finite volume method in space and evolving the cell averages in time with an explicit time discretization. Each simulation step involves three kernels: DT, RHS and UP. The DT kernel computes a time step that is obtained by a global data reduction of the maximum characteristic velocity. The RHS kernel entails the evaluation of the Right-Hand Side (RHS) of the governing equations for every cell-average. The UP kernel updates the flow quantities using a Total Variation Diminishing (TVD) scheme. Depending on the chosen time discretization order, RHS and UP kernels are executed multiple times per step.

The spatial reconstruction of the flow field is carried out on velocity and pressure ([6]) while their zero jump conditions across the contact discontinuities are maintained by reconstructing special functions of the specific heat ratios and correction pressures. Quantities at the cell boundaries are reconstructed through the fifth-order Weighted Essentially Non-Oscillatory (WENO) scheme [5]. In order to advance the system, we compute the numerical fluxes by using the HLLE (Harten, Lax, van Leer, Einfeldt) scheme [12]. The evaluation of RHS requires information exchange of adjacent subdomains due to the WENO scheme. The RHS evaluation includes five stages/microkernels: a conversion stage from conserved to primitive quantities (CONV), a spatial reconstruction (WENO) using neighboring cells, evaluation of the numerical flux (HLLE) at the cell boundaries, summation of the fluxes (SUM) and a final stage for writing back the results (BACK).

**Table 1: BGQ node performance table.**

| | |
|---|---|
| Cores | 16, 4-way SMT, 1.6 GHz |
| Memory | 16KB L1, 32MB L2, 16GB DDR3 |
| Peak performance | 204.8 GFLOP/s |
| L2 bandwidth | 185 GB/s (measured) |
| DDR3 bandwidth | 28 GB/s (measured) |

## 3. HARDWARE PLATFORM

Our target platform was the IBM Blue Gene/Q supercomputer. This platform is based on the BGQ compute chip which is equipped with 16 symmetric cores operating at 1.6 GHz. A per-core Quad floating-point Processing Unit implements the QPX instruction set and has a SIMD-width of 4. Each core supports 4 hardware threads, offering a maximum concurrency of 64 on a single BGQ node. A 16 KB L1 data cache is shared across the hardware threads of a single core. Each core accesses the shared L2 data cache through a crossbar. L2 is organized in 16 slices of 2 MB and memory addresses are scattered across these slices. An L1 cache prefetching unit aims at hiding possible latencies from the L2 data cache and DDR memory.

Table 1 summarizes the main performance features of a single BGQ node. Node boards consist of 32 compute nodes and are grouped in 32 to form a rack, with a nominal compute performance of 0.21 PFLOP/s. BGQ nodes are placed in a five-dimensional network topology, with a network bandwidth of 2 GB/s for sending and 2 GB/s for receiving data, respectively. Due to the relatively low ridge point of the platform, kernels that exhibit operational intensities higher than 7.3 FLOP/off-chip Bytes are compute-bound.

## 4. SOFTWARE LAYOUT

CUBISM-MPCF is designed to minimize compulsory memory traffic by using low-storage time stepping schemes that reduce the overall memory footprint and high-order spatiotemporal discretization schemes that decrease the total number of steps. In its current version, the solver employs a third-order low-storage TVD Runge-Kutta time stepping scheme, combined with a fifth order WENO scheme. To avoid degradation of operation intensity, we employ data reordering and cache-aware techniques. Data reordering is achieved by grouping the computational elements into 3D blocks of contiguous memory, organized in an AoS format. To effectively operate on blocks we consider SIMD-friendly temporary data structures, in SoA format, that allow for extensive use of vector intrinsics In addition, to increase temporal locality we employ computation reordering techniques when evaluating the RHS.

CUBISM-MPCF is written in C++ and parallelized using the MPI and OpenMP programming models. It is conceptually decomposed into three layers: *cluster, node, and core.*

The *cluster layer* is responsible for the domain decomposition and the inter-rank information exchange based on MPI. The computational domain is decomposed into subdomains across the ranks in a cartesian topology with a constant subdomain size. The subdomains are further decomposed into constant-sized blocks of data, which are divided into halo

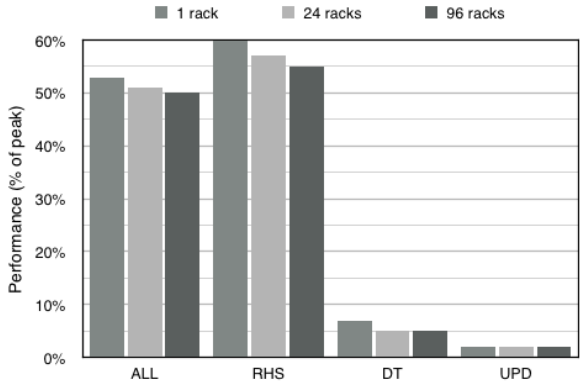



**Figure 1: Initial performance**

and interior ones during the evaluation of the RHS. The cluster layer uses non-blocking point-to-point communications to exchange ghost information for the halo blocks and dispatches the prepared blocks for computation to the node layer. The time for processing the blocks is one order of magnitude larger than the data transfer time, which allows for the efficient communication/computation overlap.

The multithreaded *node layer* is responsible for coordinating the work within the ranks. We enforce optimal thread-data affinity through a depth-first thread placement layout. The work associated to each block is exclusively assigned to one thread based on the OpenMP dynamic scheduling policy. This hides potential imbalances during the evaluation of the RHS, while the incurred runtime overhead is amortized by the work per block, which is in the order of 10 ms (per thread). To evaluate the RHS of a block, the assigned thread loads the block data and ghosts into a per-thread dedicated buffer. For a given block, the intra-rank ghosts are obtained by loading fractions of the surrounding blocks, whereas for the inter-rank ghosts data is fetched from a global buffer. The node layer relies on the core layer for the execution of the compute kernels.

The *core layer* is responsible for the execution of the compute kernels, namely RHS, UP, DT, as well as for the forward wavelet transform (FWT) kernel of the compression scheme. On IBM BGQ (and Cray XE6/XC30) platforms, this layer benefits from QPX and SSE/AVX intrinsics to expose more data-level parallelism. The RHS kernel makes use of light-weight ring buffers that are designed to minimize the memory consumption and maximize the temporal locality. Due to its spatial access pattern and computational irregularities, the RHS is not straightforward to vectorize: it involves AoS/SoA conversions, data reshuffling for stencil operations and conditional branches. This, together with kernel micro-fusion, increases the instruction-level parallelism.

For a more detailed description of CUBISM-MPCF we refer to Ref. [8].

# 5. PERFORMANCE IMPROVEMENTS

Fig 1 summarizes the initial overall performance (ALL) of CUBISM-MPCF as well as the performance of its individual kernels (RHS, DT, UP), obtained on 1, 24 and 96 BGQ racks. From 1 to 24 racks, the performance of the RHS kernel decreases from 60% to 57% of the peak. The DT kernel exhibits a 2% performance loss while the UP kernel, which does not involve any communication, remains unaffected. Another 2% loss in the performance is observed for the RHS kernel on the 96 BGQ racks, achieving 11 PFLOP/s.

The optimization techniques described in this section allowed us to improve by 31% the sustained performance of our simulations. The highest sustained peak performance reached 14.43 (previously 10.99) PFLOP/s, which corresponds to 72% (previously 55%) of the nominal peak of Sequoia, the IBM BGQ system at the Lawrence Livermore National Laboratory.

We did not introduce any algorithmic changes in our software but instead, we focused on its fine-tuning and the exploitation of the BGQ hardware. Our improvements result from addressing three major performance challenges: computation/communication overlap, memory management, and load imbalance.

### *Computation/Communication overlap*

Despite the use of non-blocking MPI calls and the adequate processing time of inner blocks before calling `MPI_Waitall()`, the cluster layer of the initial version suffered a 5% performance loss from 1 to 96 BGQ racks mainly due to inefficient communication/computation overlap observed for a large number of compute nodes.

Initially, we modified our code to post all the asynchronous `MPI_Irecv()` calls before the `MPI_Isend()` calls. We managed to eliminate communication overheads by activating the asynchronous progress communication at the PAMID layer of the BGQ MPI implementation. This mechanism leads to practically zero overhead for the non-blocking point-to-point communication in the RHS kernel. In addition to setting the appropriate environment variables, we had to apply a new workflow for the rank/thread configuration of 1/64 on each BGQ node. More specifically:

- The main thread issues `MPI_Irecv/Isend` calls for the halo blocks.
- It then encounters a parallel region with 63 OpenMP threads, where each thread processes a single inner block. The number of assigned blocks per thread can be adjusted to allow for full overlap of communication with computation.
- The main thread calls `MPI_Waitall()` after this parallel region.
- The rest of the inner blocks and the halo ones are processed by all 64 OpenMP threads in a subsequent parallel region using a dynamically scheduled for loop.

The necessity of this scheme is attributed to the non-preemptive scheduling of software threads, which does not allow for core oversubscription. Due to the 'free' hardware thread that advances MPI communications in the background, the total communication time of `MPI_Waitall()` is negligible. In turn the performance loss of the RHS kernel at the cluster layer is minimized and becomes practically **independent** of the number of compute nodes used for the simulation.

### *Memory management*

In order to reduce L1 Data Cache Misses (DCM) and register spilling, we applied the following techniques:

1. Linear stream prefetching of data was activated with the confirmed mode and depth equal to one (default is two). A stream is confirmed when there are two L1 cache misses within 128 bytes.

2. Deactivation of compiler-based unrolling of a loop that invokes the QPX-based WENO kernel. The initially used `pragma unroll(4)` compiler directive was affecting performance due to register spilling.

3. Faster loading of ghost data both at the node and cluster layer due to more efficient unpacking of the received data. This was achieved by extensive use of the built-in `__bcopy()` function.

### *Load imbalance*

The load imbalance at the cluster layer was originally manifested by significant times spent at `MPI_Allreduce`. Besides the varying times of `MPI_Waitall`, load imbalance was introduced by the intra-node scheduling scheme for block processing and the boundary conditions at the cluster level. we made the following code modifications:

1. Besides fine tuning of computation/communication overlap, the adopted block processing workflow improves load balancing because blocks are distributed evenly to the OpenMP threads of the two parallel regions mentioned above. For instance, for a typical cubic subdomain of $16^3 = 4096$ blocks per node, 63 blocks are initially processed and then 4033 blocks are dynamically distributed among 64 threads. In contrast, the previously used workflow first assigns 2744 inner blocks and then 1352 halo blocks to 64 threads, increasing the possibility that some OpenMP threads to become idle.

2. Boundary conditions are enforced by using loop unrolling and copying memory with the `__bcopy()` function. This minimizes per-block overheads and combined with the faster loading of ghost data leads to more uniform block-processing time across the compute nodes and, thus, in better load balancing.

### *Additional fine tuning options*

We observed minor performance improvements (<0.5%) in our solver for the following options:

- Use of the same optimization flag ("`-O3`") for all the three layers of the software: the core layer was previously compiled with "`-O5`" .

- Decrease of the stack size of OpenMP threads from 1MB to 512KB.

### *Numerical accuracy*

In addition to these advances, we extended the level of accuracy in our simulations, by introducing an additional second pass in the Newton-Raphson scheme used for the computation of the reciprocals. The previously employed single-pass scheme achieves 13.79 PFLOP/s of peak performance for the RHS Kernel. The two-pass scheme increases the computational intensity with respect to the single-pass, at the expense of slightly higher time-to-solution (12%). This, combined with the better exploitation of memory subsystem and the communication and load imbalance advances allowed us to reach 14.43 PFLOP/s for the RHS kernel.

## 6. PERFORMANCE RESULTS

We compiled CUBISM-MPCF with the same software stack and version of the IBM XL C/C++ compiler (v12.1) and used the IBM Hardware Performance Monitor (HPM) Toolkit for BGQ for measuring performance figures.

### *Cluster layer*

We repeated the initial runs for identical cloud simulations and problem sizes, using 4096 blocks on each compute node with $32^3$ computational elements in each block. Performance measurements for the fraction of the peak and the achieved PFLOP/s on the 96-rack Sequoia BGQ system (98304 nodes, 1572864 cores) are presented in Table 2 (top and bottom respectively). In summary, the overall performance has been improved from 10.14 to 13.1 PFLOP/s for the two-pass scheme and to 11.3 PFLOP/s for the single-pass scheme.

On the 96 racks of Sequoia, our simulations operate on $N_c$=13.2 trillion grid points and therefore, each core processes 8.39 million points every 15.2 seconds. By dividing the time per step with the number of points per core, we compute that the normalized time is equal to $T_w$=1.81. $T_w$ was introduced by Bermejo-Moreno et al. [3] to characterize the performance of their *Hybrid* solver and the best values they achieved for their turbulence simulations on Sequoia range between 16.3 and 39.0, while the achieved performance did not exceed 6.4% of the peak. By projecting the performance of [1] on the BGQ platforms and assuming perfect scaling, we compute that $Tw$ =29.7.

### *Node layer*

We assess the performance of the node layer by performing simulation runs on a single BGQ chip. Although this layer performs ghost reconstruction across the blocks, it completely avoids explicit communication and synchronization overheads due to MPI. In contrast to the cluster layer, the 4096 blocks are dynamically scheduled to the 64 OpenMP threads using a single parallel for loop.

The percentage of the peak achieved by the node layer is depicted in Table 3. We observe that the overall performance of both schemes for the node layer, both in fraction of the peak and time per simulation step, is close to the corresponding performance achieved for the cluster layer on the 96 racks. We observe 0.6% and 5.6% absolute performance loss for the RHS and DT kernels, while UP is not affected as it involves local computations. The minimal performance loss for RHS demonstrates the effectiveness of our computation-communication overlap scheme. With regard to DT, the difference between the two layers is that the cluster layer involves a global MPI reduction operation. In addition, load imbalance during the processing of blocks cannot be fully eliminated due to different boundary conditions. The loss in the overall performance is close to that of the RHS kernel, while the time required for a single simulation step increases increases by 0.3 seconds for both cases.

The third row in Table 3 shows the performance of a scheme where both reciprocal/divisions and square roots computations are performed using the native QPX-based `vec_swdiv()` and `vec_swsqrt()` functions. This scheme increases the performance of the DT kernel but delivers lower performance to RHS, which affects both the overall performance and mostly the time to solution. Based on the observed performance and accuracy of our simulations, we concluded that `vec_swdiv()` follows the two-pass scheme.

**Table 2: Performance in fraction of the peak (top) and improvements in attained PFLOP/s (bottom) as well as time to solution for the initial and the updated version of the software and two accuracy levels.**

| | ALL | RHS | DT | UP | TtS (sec) |
|---|---|---|---|---|---|
| Initial (single-pass) | 50.4% | 54.6% | 4.9% | 2.4% | 18.3 |
| Updated (single-pass) | 61.1% | 68.5% | 10.2% | 2.3% | 15.2 |
| Updated (two-pass) | 64.8% | 71.7% | 13.2% | 2.3% | 17.0 |
| Initial (single-pass) | 10.14 | 10.99 | 0.98 | 0.49 | 18.3 |
| Updated (single-pass) | +1.16 | +2.80 | +1.07 | -0.02 | -3.1 |
| Updated (two-pass) | +2.96 | +3.44 | +1.67 | -0.02 | -1.3 |

**Table 3: Achieved performance of the node layer.**

| | ALL | RHS | DT | UP | TtS (sec) |
|---|---|---|---|---|---|
| Updated (single-pass) | 61.9% | 69.1% | 15.8% | 2.3% | 14.9 |
| Updated (two-pass) | 65.5% | 72.3% | 19.9% | 2.3% | 16.7 |
| Updated (native) | 64.9% | 71.1% | 20.4% | 2.3% | 17.6 |

### *Core layer*

We evaluate the performance of the WENO kernel, the most time consuming stage of the RHS. Table 4 shows the performance of the two accuracy schemes (single and two-pass) for the reciprocal and for both of the QPX non-fused and fused WENO implementations. The fused WENO implementations reach 77.6% and 80.5% of the peak performance of the BGQ core (12.8 GFLOP/s), which are within 1% of their maximum theoretical performance as defined by their density of FMA operations. Kernel fusion improves the performance of WENO by 8% and 22% with respect to the attained GFLOP/s and processor cycles respectively.

### *Simulations*

We initialize the simulation with spherical bubbles modeling the state of the cloud right before the beginning of collapse, while radii of the bubbles are sampled from a lognormal distribution corresponding to a range of 50-200 microns. For the bubble distributions, we choose a resolution such that the smallest bubbles are still resolved with 50 points per radius. Material properties, $\gamma$ and $p_c$, are set to 1.4 and 1 bar for pure vapor, and to 6.59 and 4096 bar for pure liquid. Initial values of density, velocity and pressure are set to 1 $kg/m^3$, 0, 0.0234 bar for vapor and to 1000 $kg/m^3$, 0, 100 bar to model the pressurized liquid. We chose a CFL of 0.3, leading to a time step of $1ns$ for a total of 40'000 steps. The simulations were performed in mixed precision: single precision for the memory representation of the computational elements and double precision for the computation.

Recent simulation results are presented in Fig. 2 providing an unprecedented level of detail for the evolution of bubble interfaces and pressure isosurfaces in a collapsing cloud of cavities over a solid wall. In Fig. 2(left) we visualize the liquid/vapor interface as well as the pressure field and the solid wall for a simulation at $t = 0.6$. We also monitor the maximum pressure in the flow field and on the solid wall, the equivalent radius of the cloud ($\sqrt[3]{3V_{\text{vapor}}/4\pi}$). At $t = 0.3$, we observe initial asymmetric deformation of the bubbles while a few bubbles have undergone the final stage of their collapse. At $t = 0.6$ a large number of bubbles have collapsed with larger collective pressure hot spots within the flow field. At a later stage, the highest pressure is recorded over the solid wall to be about 20 times larger than the ambient pressure (Fig. 2, right). We consider that this pressure is correlated with the volume fraction of the bubbles, a subject of our ongoing investigations. We also observe that the equivalent radius of the cloud (blue line in the same figure) undergoes an expansion after $t = 0.6$ implying that some packets of vapor grow larger, indicating bubble rebound, before undergoing their final collapse.

## 7. CONCLUSION AND OUTLOOK

We have presented CUBISM-MPCF, a large-scale compressible, two-phase flow simulation software designed for studies of cloud cavitation collapse. The software is built upon algorithmic and implementation techniques that address the challenges posed to classical flow solvers by contemporary supercomputers, namely the imbalance between the compute power and the memory bandwidth as well as the limited I/O bandwidth. The present flow simulations on 1.6 million cores of Sequoia achieve an unprecedented 14.4 PFLOP/s corresponding to 72% of its peak. The simulations employ 13 trillion computational elements to resolve 15'000 bubbles improving by two orders of magnitude the state of the art in terms of geometric complexity.

We consider that the techniques reported here in can be adopted to enhance the performance of all finite volume and finite difference flow solvers that employ uniform grid sizes. Our goal is to enhance this software with wavelet adapted grids for multiresolution flow simulations [9]. We envision that large scale simulations of cloud cavitation collapse will enhance engineering models and form the foundation for complete simulations of high performance fuel injection systems. On-going research in our group focuses on coupling material erosion models with the flow solver for predictive simulations in engineering and medical applications.

## Acknowledgments

We wish to thank Dr. A. Curioni (IBM), Dr. C. Bekas (IBM), Dr. A. Bertsch (LLNL) and Dr. S. Futral (LLNL) for their valuable help in conducting the experiments on Sequoia, and Dr. R. Walkup (IBM) for providing us the HPM toolkit. This work was supported by DOE INCITE and PRACE Awards.

**Table 4: Achieved performance of the WENO kernel (per-core).**

| Scheme | Version | Performance (GFLOP/s) | Peak fraction [%] |
|---|---|---|---|
| Single-pass | Baseline | 9.22 | 72.0% |
| | Fused | 9.93 | 77.6% |
| Two-pass | Baseline | 9.62 | 75.2% |
| | Fused | 10.30 | 80.5% |

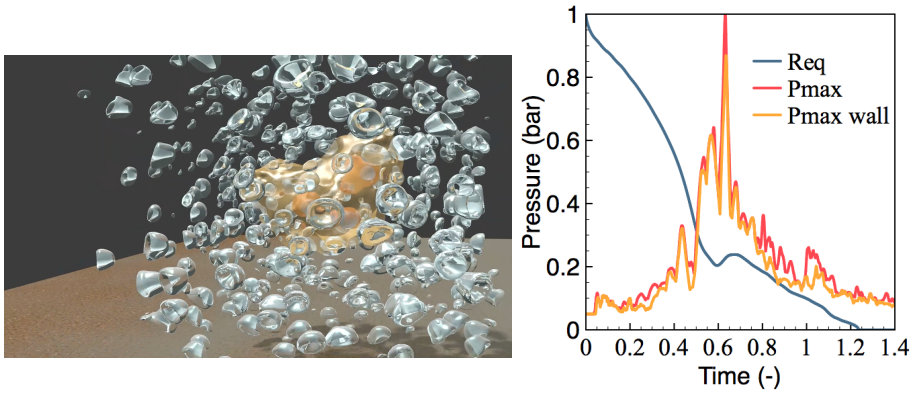



**Figure 2: (Left) Volume and isosurface rendering of the pressure (high/low in yellow/orange) and the interfaces of the bubble (translucent white) at late stages of the collapse of an array of clouds. (Right) Temporal evolution of the maximum pressure in the field and on the solid wall and temporal evolution of the normalized equivalent radius of the cloud (solid blue line)**

## 8. REFERENCES

[1] N. A. Adams and S. J. Schmidt. Shocks in cavitating flows. In *Bubble Dynamics and Shock Waves*, pages 235–256. Springer, 2013.

[2] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53(3):484–512, 1984.

[3] I. Bermejo-Moreno, J. Bodart, J. Larsson, B. M. Barney, J. W. Nichols, and S. Jones. Solving the compressible navier-stokes equations on up to 1.97 million cores and 4.1 trillion grid points. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 62:1–62:10, New York, NY, USA, 2013. ACM.

[4] B. Hejazialhosseini, D. Rossinelli, C. Conti, and P. Koumoutsakos. High throughput software for direct numerical simulations of compressible two-phase flows. *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, pages 16:1–16:12, 2012.

[5] G. Jiang and C. Shu. Efficient implementation of weighted ENO schemes. *J. Comput. Phys.*, 126(1):202–228, 1996.

[6] E. Johnsen and T. Colonius. Implementation of WENO schemes in compressible multicomponent flow problems. *J. Comput. Phys.*, 219(2):715–732, 2006.

[7] T. Kodama and K. Takayama. Dynamic behavior of bubbles during extracorporeal shock-wave lithotripsy. *Ultrasound Med Biol*, 24(5):723 – 738, 1998.

[8] D. Rossinelli, B. Hejazialhosseini, P. Hadjidoukas, and et al. 11 pflop/s simulations of cloud cavitation collapse. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 3:1–3:13, New York, NY, USA, 2013. ACM.

[9] D. Rossinelli, B. Hejazialhosseini, W. M. van Rees, M. Gazzola, M. Bergdorf, and P. Koumoutsakos. MRAG-I2D: multi-resolution adapted grids for remeshed vortex methods on multicore architectures. *J. Comput. Physics*, 288:1–18, 2015.

[10] D. P. Schmidt and M. L. Corradini. The internal flow of diesel fuel injector nozzles: A review. *International Journal of Engine Research*, 2(1):1–22, 2001.

[11] O. Vasilyev and C. Bowman. Second-generation wavelet collocation method for the solution of partial differential equations. *J. Comput. Phys.*, 165(2):660–693, DEC 10 2000.

[12] B. Wendroff. Approximate riemann solvers, godunov schemes and contact discontinuities. In *Godunov Methods*, pages 1023–1056. Springer, 2001.

# The Impact of Process Placement and Oversubscription on Application Performance

## A Case Study for Exascale Computing

Florian Wende Thomas Steinke Alexander Reinefeld
Zuse Institute Berlin (ZIB), Takustraße 7, 14195 Berlin
{wende,steinke,ar}@zib.de

## ABSTRACT

With the upcoming transition from petascale to exascale computers radically new methods for scalable and robust computing are required. Computing at the speed of exascale, that is, more than $10^{18}$ floating point operations per second, will only be possible on systems with millions of processing units. Unfortunately, the large number of functional components like computing cores, memory chips and network interfaces will greatly increase the probability of failures, and it can thus not be expected that an exascale application will complete its execution on exactly the same resources it was started. In this paper, we investigate the impact of unfavorable process placement and oversubscription of compute resources on the performance and scalability of typical application workloads like CP2K, MOM5 and BQCD. We provide results on two HPC architectures, a Cray XC40 with proprietary Aries network routers and dragonfly topology, and an InfiniBand cluster.

## Keywords

Fault-tolerance, Process placement, Oversubscription, Application performance, Hyper-Threading, Concurrent program execution

## 1. INTRODUCTION

Current petascale computer installations comprise $10^5$ compute nodes that are connected by custom network interconnects with a hierarchical abstraction of the entire machine down to the single-node level. For exascale computing a couple of architectural issues—also affecting today's installations already—need to be approached, two of which are the scalability of the network in terms of latency and bandwidth across the entire machine, and the steadily increasing number of compute resources within single nodes contrasted with only slightly increasing network and main memory bandwidths.

Alongside scaling the hardware to exascale, user applications need to adapt as well, e.g., by utilizing hybrid MPI plus threading in order to minimize inter-process communication, heterogeneous programming involving hardware accelerators as one means to approach exascale, vectorization (SIMD), parallel I/O, and sophisticated load balancing on a final note. For many existing code bases—possibly even those that are believed to be well optimized—it might be expected that without appropriate adaption it will not be possible to use the hardware efficiently. Small imbalances in the program execution, due to MPI and I/O, for instance, can result already in a certain amount of compute resources run idle [1].

A further issue is component failure, which becomes more likely with increasing number of functional units and size of the installation. Restarting an exascale application after component failure is expected to work efficiently with in-memory checkpointing together with, e.g., erasure-coding [2]. Thereby, the restart needs to happen almost immediately after the crash on an adapted and possibly reduced node allocation. One central question that arises in that context is whether the restarted application can utilize that allocation efficiently if the latter is unfavorable? and if not, what can be done to compensate for that? For exascale computing this question is relevant twice, first because of an increased component failure rate and possibly unfavorable resource allocation at restart, which then in turn might cause additional imbalances throughout the program execution, and second, because of the possibility that even an optimized application cannot fully utilize the available compute resources of modern processors.

We address these points in this paper targeting two different HPC systems, which together with the workloads CP2K, MOM5 and BQCD will be introduced in Section 2 and 3, respectively. In Section 4, we investigate the impact of unfavorable process placements for two of these applications. Section 5 approaches the resource utilization issue, including results for oversubscription and concurrent program execution.

## 2. TARGET HPC SYSTEMS

Experiments for which results are reported in this paper have been carried out on two current HPC systems: a Cray XC40 supercomputer and an InfiniBand cluster. Both systems are briefly described subsequently.

### 2.1 Cray XC40 with Aries Interconnect

The Cray XC40 integrates the Aries interconnect together with standard Intel Xeon x86 processors. The XC40 at ZIB comprises a total of 1128 compute nodes, each of which with two Intel Xeon E5-2680v3 (Haswell) 12-core CPUs and 64 GiB main memory. The installation runs SLES 11 with the latest Cray software stack. The compute resources are hierarchically organized as follows [3]: a *blade* contains four compute nodes and one Aries router; 16 blades form one *chassis*; Three chassis make up a *cabinet*, and two cabinets form an *electrical group*. Within the electrical groups communication happens over a two-dimensional all-to-all copper-based network: communication over the chassis' backplane in one dimension, and in the other dimension all-to-all communication within subgroups given by corresponding nodes in the six chassis belonging to the same electrical group. Electrical groups are connected by optical links in all-to-all fashion (Figure 1).

***Network Characteristics***: Node-to-node latencies and bandwidths for different placements of processes along the layers of the node hierarchy of the XC40 are listed in Table 1. Values are for both (a) $n = 1$ (MPI plus threaded applications, for instance) and (b) $n = 24$ (pure MPI jobs) pairs of communicating processes that are placed on different nodes in all cases. For (b) the QPI (QuickPath Inter-

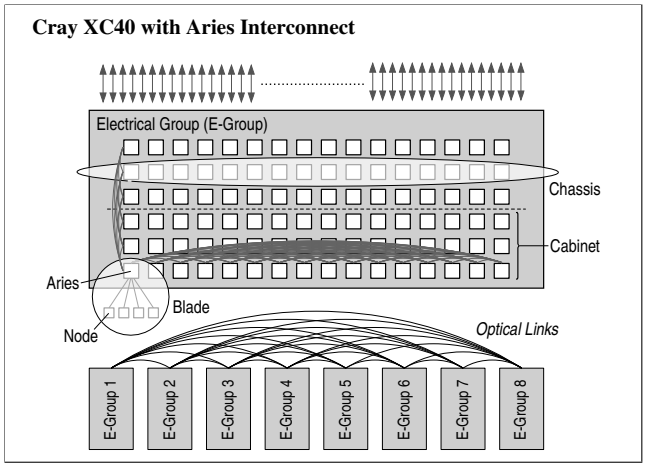



***Figure 1:*** *Schematic of the Cray Aries Interconnect. Communication within electrical groups happens over a two-dimensional all-to-all copper-based network. Electrical groups are connected via optical links in all-to-all fashion.*

connect) latency adds to the network latency twice, as the network interface is connected to CPU socket 0 but not 1.

According to Table 1 latencies only slightly decrease from intra-blade to inter-electrical-group communication, and from $n = 1$ to $n = 24$ groups. Extending the XC40 by additional electrical groups does not lower the latencies further. Bandwidths are almost homogeneous across the entire machine. However, it takes multiple MPI processes to saturate the network.

## 2.2 InfiniBand Cluster

The system comprises 32 compute nodes, each of which with four Intel Xeon E5-4650v2 (Ivy Bridge) 10-core CPUs, 512 GiB main memory and two Mellanox ConnectX-3 InfiniBand (IB) FDR ports (the nodes run SLES 11). All nodes are connected via two IB FDR switches, thereby forming a flat network with at most one hop for communicating processes.

Latencies reach down to 1.1 µs in case of $n = 1$ process per node, and about 7.5 µs for $n = 40$ (Intel MPI pingpong benchmark 4.0 with Intel MPI 5.0.2 and DAPL fabric). Bandwidths saturate ($n = 40$) at about 8.8 GiB/s for 1 MiB and 2 MiB packages, and 5.5 GiB/s for packages larger than 128 MiB—with the OFA dual-rail fabric, we got comparable values.

# 3. WORKLOADS

We investigate the impact of unfavorable process placements and oversubscription on the following workloads: CP2K, MOM5 and

***Table 1:*** *Latencies $\ell$ (µs) and per-link bandwidths b (GiB/s) of the XC40 network for n pairs of communicating MPI processes placed along the different layers of the node hierarchy. Values have been determined with the Intel MPI pingpong benchmark 4.0 with arguments* `-multi 0 -msglog 26:28 -map n:2 -off_cache -1`. *$\ell_{\text{min}}$, $\ell_{\text{avg}}$ : Minimum resp. average transfer time over {0,1,2,4}-byte packages in µs. b : Averaged bandwidth over {64,128,256}-MiB packages in GiB/s.*

| Node-to-Node: *n* processes per node | *n*=1 | | *n*=24 | | |
|---|---|---|---|---|---|
| Communicating processes in... | $\ell_{\text{min}}$ | *b* | $\ell_{\text{min}}$ | $\ell_{\text{avg}}$ | *b* |
| same blade, different node | 1.64 | 8.24(2) | 1.75 | 2.08(1) | 9.18(1) |
| same chassis, different blade | 1.78 | 8.17(2) | 1.92 | 2.29(1) | 9.34(1) |
| same cabinet, different chassis | 1.76 | 8.12(6) | 1.86 | 2.23(1) | 9.33(1) |
| same electrical group, different cabinet | 1.78 | 8.08(7) | 1.90 | 2.26(1) | 9.33(1) |
| different electrical group | 2.31 | 7.60(9) | 2.50 | 2.82(2) | 9.45(1) |

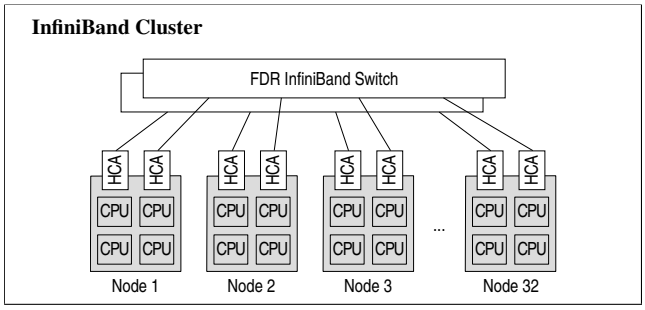



***Figure 2:*** *Schematic of the InfiniBand cluster. Nodes comprise four 10-core CPUs each, two Mellanox ConnectX-3 InfiniBand (IB) FDR ports, and are connected via two FDR IB switches.*

BQCD. All three applications are frequently used by a major fraction of ZIB's user community within HLRN.[1]

Code compilation is carried out for MOM5 and BQCD using optimized libraries and the Intel Fortran / C compiler version 15.0.2 on both the XC40 and the IB cluster. The CP2K code is compiled with the Intel Fortran / C compiler version 13.1.3 using a Haswell- respectively Ivy Bridge-optimized version of libsmm, and Intel's MKL. Codes are built against Cray MPI on the XC40 and Intel MPI 5.0.2 on the IB cluster.

***CP2K*** is an MPI + OpenMP parallel program to perform atomistic and molecular simulations of solid state, liquid, molecular, and biological systems. It implements density functional theory (DFT) using a mixed Gaussian and plane waves approach (GPW) and classical pair and many-body potentials [4].
*We use the H2O-1024 input in the CP2K branch with 5MD steps.*

***MOM5*** (Modular Ocean Model) is an MPI parallel program to perform numerical ocean simulation that is utilized for research and operations from the coasts to the globe [5].
*We use input files for simulating the Baltic Sea with three nautical miles resolution and adapt the simulated time to our needs.*

***BQCD*** (Berlin Quantum Chromodynamics) is a Hybrid Monte Carlo (HMC) MPI + OpenMP parallel program for the simulation of lattice QCD with dynamical Wilson fermions [6].
*We use the MPP benchmark input in the BQCD branch with a* $48 \times 48 \times 48 \times 80$ *lattice on the XC40, and a* $48 \times 50 \times 48 \times 80$ *lattice on the IB cluster. For cache sensitive runs, we use a local (per-process) lattice of size* $24 \times 3 \times 4 \times 4$.

Note: All three codes have been compiled as MPI-only versions.

# 4. PROCESS PLACEMENT

With a notably increased number of functional units over recent and current supercomputers as well as complex software stacks, exascale systems will be subject to failure rates with potentially multiple interrupts per hour due to hardware or software crashes. To recover program execution efficiently, the implementation of checkpoint/restart needs to be addressed on both the software and the hardware level. As part of our research on the development of a fast and fault-tolerant, microkernel based systems infrastructure, we developed an in-memory checkpointing mechanism that writes erasure-coded checkpoints to RAM disks [2]. The checkpoints are application-triggered and hence only few state information needs to be written which allows very frequent checkpointing. In case of component failures, the state information of the crashed processes is re-assembled from the saved erasure-encoded blocks and the processes are restarted on other (non-faulty) resources (Figure 3).

[1] North-German Supercomputing Alliance, `www.hlrn.de`.

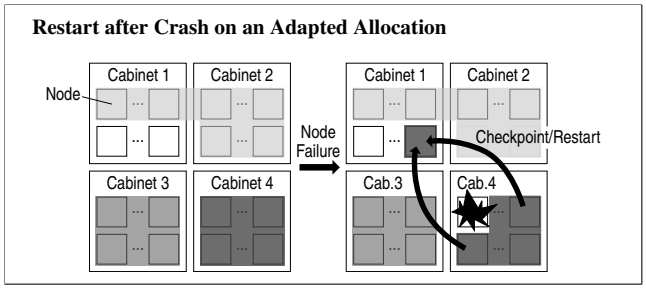



***Figure 3:*** *Restart of an application after a crash. The node allocation after the crash includes one node in a different cabinet (unfavorable). Node allocations are marked by semi-transparent gray-colored overlays.*

## 4.1 Communication Behavior

To study the impact of unfavorable process placements on the application performance, we first determine communication characteristics of CP2K and MOM5 for given inputs. For that purpose, both applications have been instrumented for sampling and tracing experiments on the XC40 using the Cray Performance Analysis Tool (CrayPAT). On the IB cluster none of the nodes is distinguished over the others, because of the flat network

For ***CP2K***, Figure 4 contains average values of the message sizes and message counts as well as information about the distribution of differently sized messages. "Small" messages (< 256 B) are those for which the data transfer is latency bound. Almost all "medium"-sized messages (256 B..1 MiB) can be assigned to FFT routines. "Large" messages (>1 MiB) are rare and have their origin in the user routines.

From the communication matrix in Figure 4, a (next-to-) nearest neighbor communication scheme can be extracted (see the minor diagonal pixels; the origin is in the top-left corner). Furthermore, MPI processes are organized into smaller groups corresponding to the block structure along the diagonal. It can be deduced that processes 0..7 are somehow distinguished as they receive data from all other processes (gray bar on the left). When scaling up the number of MPI processes, a gather-scatter scheme becomes obvious with lower process IDs as the root(s). Placing these distinguished processes far away from the other processes may result in degraded program performance, particularly as the majority of the messages is latency bound and hence sensitive to being placed, e.g, in different electrical groups (see Table 1).

Similar to CP2K, ***MOM5*** exhibits a (next-to-) nearest neighbor communication pattern. Some neighbor communication paths are

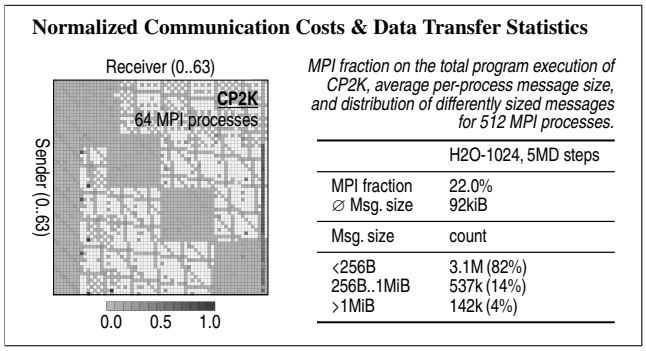



***Figure 4:*** *Communication characteristics of CP2K for H2O-1024 input: (left) communication matrix for 64 MPI processes, (right) message size statistics for 512 MPI processes.*

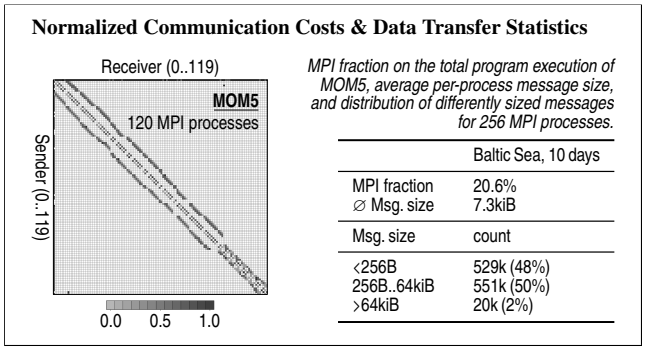



***Figure 5:*** *Communication characteristics of MOM5 for a Baltic Sea input setup with 10 days of simulated time: (left) communication matrix for 120 MPI processes, (right) message size statistics for 256 MPI processes.*

not present, which might be due to the input itself (here the geometry of the Baltic Sea area), or because of Cray PAT excluded data that is below the threshold for being factored into the plot. The vertical and horizontal lines at the left and top of the matrix indicate that process 0 is distinguished. In fact, process 0 gathers simulation data from all other processes throughout the execution (vertical line). The horizontal line corresponds to a broadcast operation with process 0 as the root. Unlike CP2K, only half the number of messages is latency bound (Figure 5). This remains true when increasing the number of MPI processes. We therefore expect only minor impact on the program performance in case of unfavorable placements.

## 4.2 Unfavorable Process Placements

Table 1 illustrates that on the XC40 only the network latency varies slightly when placing communicating processes along the node hierarchy, whereas bandwidths remain almost constant. Unfavorable placements hence are those where processes are "far away," that is, they are placed in different electrical groups. Subsequently, we consider process placements across four cabinets in two electrical groups. The distribution of the processes is encoded into the string "$n_1 - n_2 - n_3 - n_4$," where $n_i$ refers to the number of processes placed in cabinet $c_{i=1,2,3,4}$—the pairs $c_1$ and $c_2$, and $c_3$ and $c_4$ form an electrical group each. For both CP2K and MOM5 the execution is with 512 MPI processes. We use at most 16 instead of 24 MPI processes per node (symmetrically distribution across the two CPU sockets) to reduce the pressure on shared per-node resources, but still have a setup that is meaningful for execution on the XC40.

According to Figure 6 there are two levels of performance decrease for CP2K, and one level for MOM5. For both applications process 0 is distinguished. For CP2K, processes 1..15 are highly involved in collective operations, too. Separating these processes from the others, that is, placing them into another electrical group (corresponding to "0-1-0-511" .. "0-16-0-496"), causes about 8.4% and 3.6% longer program execution of CP2K and MOM5, respectively. For CP2K the performance decrease changes from 8.4% to 3.5% when balancing the total of 512 processes among the four cabinets. The value of 3.6% for MOM5 remains unchanged.

All experiments have been carried out with almost no other users on the XC40. However, to account for variations of the program execution times, each data point in Figure 6 is the mean value of six independent and temporarily separated runs. Within statistical errors the average execution times for the different placements are well distinguished. For both CP2K and MOM5, and for the input considered, the placement of the processes has only little effect on

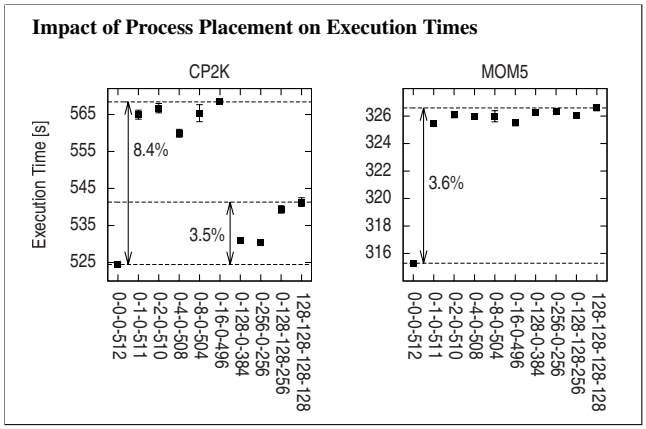



***Figure 6:*** *Impact of process placement on the program execution time of CP2K (H2O-1024 input) and MOM5 (Baltic Sea input with 1 month simulated time). The number of processes in each of four cabinets is encoded into the string "$n_1-n_2-n_3-n_4$," where $n_i$ refers to the number of processes in cabinet i. In all cases: $n_1+n_2+n_3+n_4=512$.*

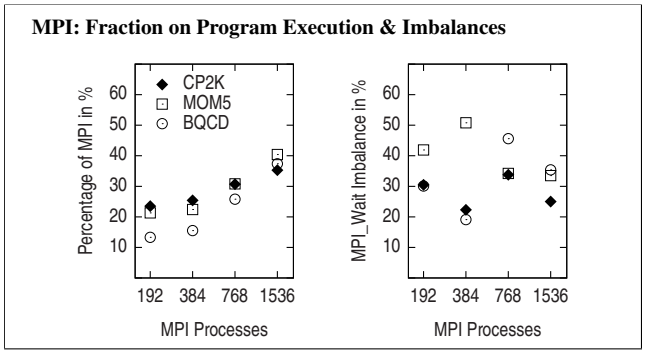



***Figure 7:*** *Fraction of MPI on the program execution time, and imbalance of* `MPI_Wait` *for CP2K, MOM5 and BQCD with different numbers of MPI processes.*

the program performance, and will be covered up by about 10% variation of the runtime in case of multiple jobs running on the machine side by side.

Regarding application checkpointing, component or node failures can be approached by restarting the execution on an adapted node allocation without concerning too much about performance losses because of an "unfavorable" allocation. The process placement at restart, however, should incorporate information about distinguished processes which should not be isolated from the other processes because of potentially longer program execution (Figure 6). This can be achieved, for instance, via rank re-ordering.

# 5. RESOURCE (UNDER)UTILIZATION

The execution of an MPI application for a given input will be dominated by MPI when increasing the number of MPI processes (strong-scaling). Figure 7 illustrates the fraction of MPI on the program execution time for the workloads CP2K, MOM5 and BQCD. For all three applications calls to `MPI_Wait` have the highest percentage on MPI (more than 50% for MOM5 and CP2K, and about 30% for BQCD). For the latter, we included the imbalance $(T_{\text{avg}} - T_{\text{min}})/T_{\text{max}}$ (reported by CrayPAT) into Figure 7—note that the impact of "stragglers" can result in high values here. Both the increase of MPI and the non-negligible fraction of `MPI_Wait` together with imbalances around 40% suggest the assumption that compute resources will be underutilized then. In a wider sense one may ask the question whether it is possible at all with current (HPC) codes to scale to thousands or millions of processors and at the same time make efficient use of the hardware.

## 5.1 Oversubscription

The idea behind oversubscription (of CPU resources) is to place more threads or processes (just processes hereafter) on the CPU cores than there are hardware threads available for execution in order to exploit modern CPU's parallel processing capabilities more effectively as is possible with only one process per hardware thread [7]. Processes that run idle then can be interleaved with others that are ready for execution, thereby reducing CPU idle cycles. On the other hand, CPU resources like the caches are shared among these processes, potentially causing increased cache miss rates.

Another way to achieving a better utilization of the CPU resources is simultaneous multithreading (SMT) like Intel's Hyper-Threading (HT) [1]. While caches are shared among the two hardware threads per CPU core, HT adds two architectural states as well as duplicated register sets to each core. However, the two hardware threads share one CPU core for execution. In a certain sense, using HT is akin to oversubscribing the CPU's compute resources partly.

Subsequently, we distinguish the following cases

- no oversubscription is used for program execution (no-OS),
- Hyper-Threading is utilized (HT-OS), and
- hardware threads are two-fold oversubscribed (2x-OS).

The mapping of processes to hardware threads in current Intel Xeon processors with Hyper-Threading is illustrated in Figure 8. For a given number of processes *N*, HT-OS and 2x-OS require only half the number of compute nodes compared to no-OS. The question is: *Does the execution time increase by a factor two or more for HT-OS and 2x-OS, respectively?* If not, CPU resources run idle in the no-OS case.

Figure 9 illustrates the strong-scaling behavior of CP2K, MOM5 and BQCD on the XC40 and the IB cluster. In case of no-OS 24 respectively 40 processes reside on the nodes of the XC40 and IB cluster. For HT-OS and 2x-OS the number of processes per node is 48 and 80 on XC40 and the IB cluster, respectively.

While BQCD scales perfectly with the number of MPI processes on both systems, for CP2K and MOM5 the scaling is acceptable only on the XC40. The growth of MPI on the program execution when increasing the number of MPI processes, as well as imbalances in the user and the MPI portion prevents better scaling. On both the XC40 and the IB cluster the increase of the execution time of CP2K and MOM5 hence is lower than a factor two in case of HT-OS. Unlike 2x-OS, the two processes per CPU core are interleaved by the CPU itself in hardware, thereby reducing idle cycles effectively. In case of 2x-OS, the operating system switches between the two processes per hardware thread, which happens with milli-second granularity. In addition to a less effective reduction of idle cycles, the context switch adds some overhead, e.g., cache invalidation. All in all, 2x-OS does not work for CP2K and MOM5.

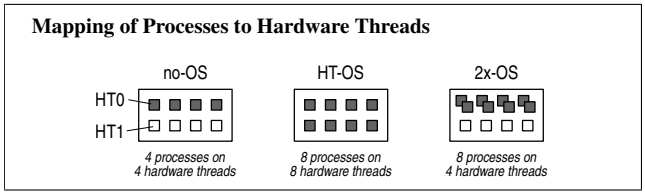



***Figure 8:*** *Mapping of processes to hardware threads for no-OS, HT-OS and 2x-OS. The CPU considered here supports Hyper-Threading and has four CPU cores.*

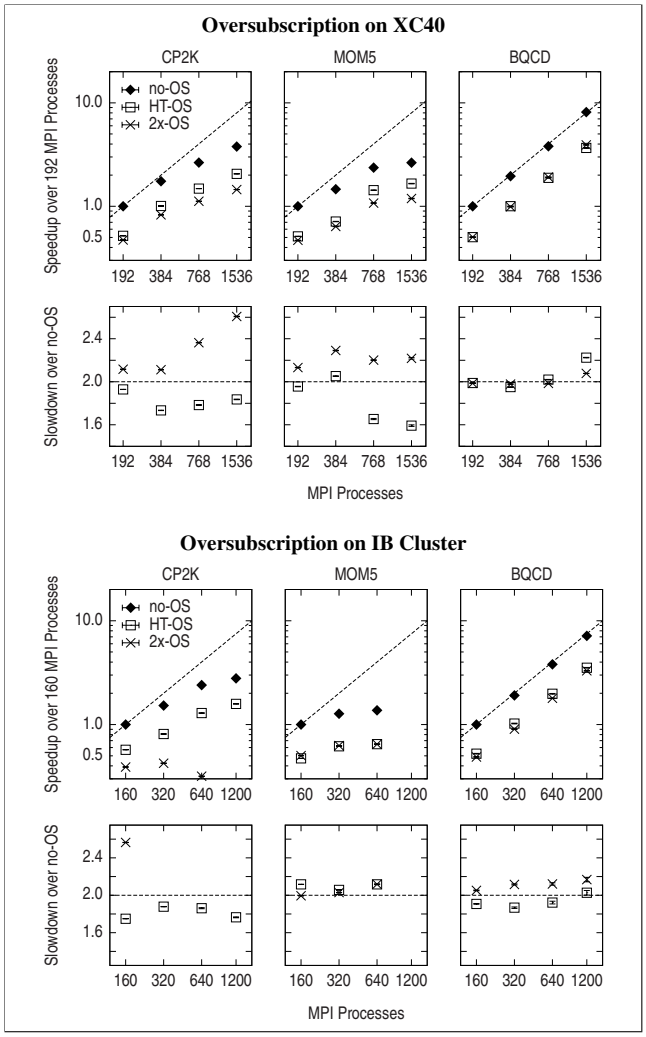



***Figure 9:*** *Strong-scaling of CP2K, MOM5 and BQCD for different numbers of MPI processes. In case of no-OS (see text) 24 respectively 40 MPI processes reside on the compute nodes of the XC40 and the IB cluster. With HT-OS and 2x-OS only half the number of compute nodes is required with 48 respectively 80 processes per node.*

For BQCD, the utilization of the CPU is already high. Using the HT-OS scheme on the XC40 results in a factor two higher execution times compared to no-OS. The difference between HT-OS and 2x-OS is immaterial on the XC40 for BQCD. Using HT-OS on the IB cluster gives only slightly increase performance over no-OS, whereas 2x-OS results in an effective performance loss.

**Cache utilization:** Figure 10 illustrates hit rates for the L1 + L2 data cache and the shared L3 cache on the XC40 for CP2K, MOM5 and BQCD (values have been determined with CrayPAT and PAPI). While the L3 cache is shared among all cores of the CPU, the L1 and L2 (data) cache is core-exclusive. Placing two processes on each of the CPU cores results for HT-OS in an effective reduction of the L1 and L2 cache sizes and potentially in decreased hit rates due to mutual cache pollution. In case of 2x-OS two processes are mapped to one hardware thread and context switching is performed by the operating system. The interleaving of the process executions in that case happens on a much coarser granularity, thereby rendering the L1 and L2 cache more or less process-exclusive. Hit rates for the L1 + L2 data cache therefore are lower for HT-OS, while those for 2x-OS are almost equal to no-OS hit rates.

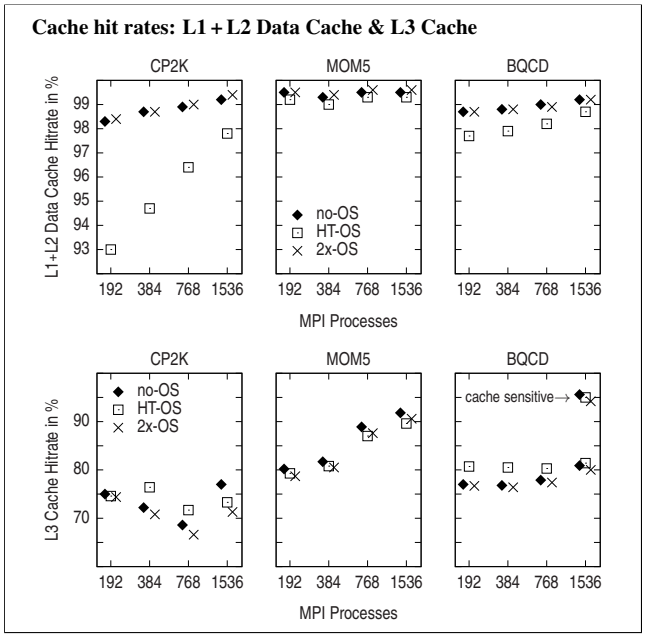



***Figure 10:*** *Cache hit rates for L1+L2 data cache and the shared L3 cache of the Intel Xeon processors on the XC40. Values have been determined with CrayPAT and PAPI.*

L3 cache hit rates vary only slightly between no-OS, HT-OS and 2x-OS. For CP2K and BQCD the HT-OS scheme slightly improves the L3 cache utilization. An interesting result is that even for BQCD and a cache sensitive input ($24 \times 3 \times 4 \times 4$ per-process lattice) the L3 hit rate does not reduce measurably when using HT-OS or 2x-OS.

## 5.2 Concurrent Program Execution

According to Figure 9 both CP2K and MOM5 show an effective performance gain when using the HT-OS scheme: the execution time on $N/2$ compute nodes with 48 processes on the XC40 increases by less than a factor two compared to the execution on $N$ nodes with 24 processes. However, using $N$ compute nodes with 48 processes does not lower the program execution time. That is, for a given node allocation none of CP2K, MOM5 and BQCD can benefit from using both hardware threads per core—a standing reason for that is the increased number of MPI processes in that case, causing additional MPI communication overhead. But, *can two independent applications run concurrently on the hardware threads in less time than executing them one after another?*

In a first step we overlap each application with itself using HT-OS, that is, two program instances with the same characteristics and bottlenecks run simultaneously (one on HT0 and the other on HT1), giving execution times $T_1$ and $T_2$. We state, that if the concurrent execution is faster than the sequential execution of the two program instances, that is, if $T_{||} = \max(T_1, T_2) < T_{\text{seq}}$, the application is a potential candidate for being placed beside another application. Results for CP2K, MOM5 and BQCD are illustrated in Figure 11: CP2K + CP2K, MOM5 + MOM5 and BQCD + BQCD. The cache-sensitive input for BQCD is marked by the star ("$\star$"). For all applications we can note a speedup $T_{\text{seq}}/T_{||}$ larger than one except for BQCD with the cache-sensitive input. These observations meet our expectations as in the previous sub-section we noted that none of CP2K and MOM5 can fully utilize the compute resources on both the XC40 and the IB cluster. For these two applications we actually observe speedups up to 20%, whereas for BQCD it is at most

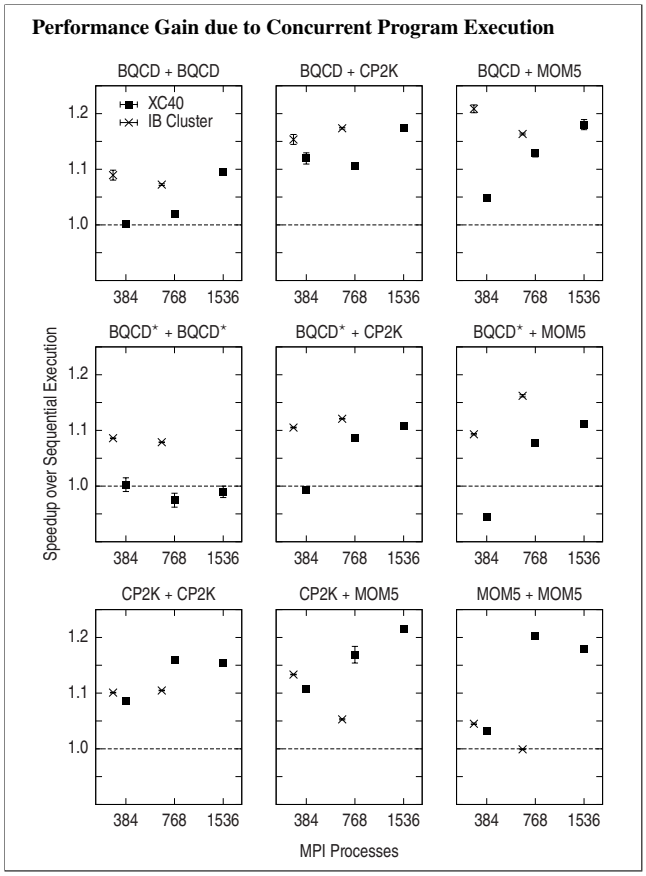



***Figure 11:*** *Performance gain due to a concurrent execution of CP2K, MOM5 and BQCD. In the concurrent case two applications run simultaneously on the same node allocation: one on HT0 and the other on HT1. The speedup values are the ratio of the sequential execution time for running the two applications one after another, and the concurrent execution time (see text). For BQCD we additionally consider a cache-sensitive setup, marked by the star ("⋆"). For all combinations we use 24 and 40 MPI processes per node and application on the XC40 and the IB cluster, respectively.*

10% for the input that is not sensitive to the cache. For the cache-sensitive input the concurrent execution on the XC40 takes slightly longer than the sequential execution. On the IB cluster we do not see any differences between the two BQCD inputs.

Figure 11 also illustrates the results for the concurrent execution of different applications. For the respective runs, we adapted either the number of MC steps for BQCD or the simulated time for MOM5 so as to achieve $T_1 \approx T_2$, that is, almost 100% overlapping program execution—$T_{\text{seq}}$ has been determined for these adapted inputs. Again for all combinations, except those involving BQCD and the cache-sensitive input, the performance gain over the sequential execution is larger than one. The tendency is towards a larger gain with increasing number of MPI processes. One reason for that is the decrease of the resource utilization when increasing the number of processes, as suggested by Figure 7.

# 6. CONCLUSION

In this paper we presented the impact of unfavorable process placements and oversubscription on the program performance for workloads CP2K, MOM5 and BQCD and selected inputs. Our results on the XC40 show that the increase of the runtime when placing distinguished processes far away from the remaining ones is within 10% compared to all processes close to each other. Within variations of the program execution times due to other users' workloads, this value is more or less insignificant. Our oversubscription experiments show that two processes per hardware CPU thread cause more than a factor two performance degradation compared to no oversubscription. Using Hyper-Threading (HT), however, the program performance of all workloads could be improved. For the concurrent execution of two applications, one on HT0 and the other on HT1, we found an overall performance gain in the majority of cases. Our investigations suggest the assumption that in the strong-scaling case MPI wait cycles together with imbalances due to access to shared compute resource cause an increase of CPU idle cycles, that, e.g., via HT, can be effectively reduced. Further investigations, however, are necessary to better understand, e.g, the role of caches in these scenarios, which we already started in this paper.

# Acknowledgment

This work was supported by the DFG Priority Program "Software for Exascale Computing" (SPPEXA, SPP 1648), project FFMK, "A fast and fault tolerant microkernel-based system for exascale computing." We also thank the HLRN Supercomputing Alliance for their support with computer time.

# 7. REFERENCES

[1] Drysdale, G. and Valles, A. C. and Gillespie, M., "Performance Insights to Intel Hyper-Threading Technology," November 2009. [Online]. Available: https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology

[2] Peter, K. and Reinefeld, A., "Consistency and fault tolerance for erasure-coded distributed storage systems," in *Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date*, 2012, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/2286996.2287002

[3] Alverson, B. and Froese, E. and Kaplan, L. and Roweth, D., "Cray XC Series Network," Cray Inc., Tech. Rep., 2012.

[4] Hutter, J. and Iannuzzi, M. and Schiffmann, F. and VandeVondele, J., "CP2K: Atomistic Simulations of Condensed Matter Systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, 2014. [Online]. Available: http://dx.doi.org/10.1002/wcms.1159

[5] Griffies, S. M., *Elements of the Modular Ocean Model (MOM)*, NOAA Geophysical Fluid Dynamics Laboratory, Princeton, USA, 2012.

[6] Nakamura, Y. and Stüben, H., "BQCD—Berlin Quantum Chromodynamics Program," *PoS*, vol. LATTICE2010, p. 040, 2010.

[7] Iancu, C. and Hofmeyr, S. and Blagojevic, F. and Zheng, Y., "Oversubscription on Multicore Processors," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–11.

[8] Wende, F. and Steinke, Th. and Reinefeld, A., "The Impact of Process Placement and Oversubscription on Application Performance: A Case Study for Exascale Computing," ZIB, Takustr. 7, 14195 Berlin, Tech. Rep. 15-05, 2015.

[9] Christmann, C. and Hebisch, E. and Weisbecker, A., "Oversubscription of Computational Resources on Multicore Desktop Systems," in *Proceedings of the 2012 International Conference on Software Multicore Engineering, Performance, and Tools*, ser. MSEPT'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 18–29. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31202-1_3

# Radiative Transfer Modeling at High Performance Computers Using Self-Adjoint Transport Equation

Olga Olkhovskaya
M. V. Keldysh Institute of Applied Mathematics RAS
4, Miusskaya sq., Moscow, 125047 Russia
+7 499 972 38 55
olkhovsk@gmail.com

Boris Chetverushkin
M. V. Keldysh Institute of Applied Mathematics RAS
4, Miusskaya sq., Moscow, 125047 Russia
+7 499 978 13 14
office@keldysh.ru

Vladimir Gasilov
M. V. Keldysh Institute of Applied Mathematics RAS
4, Miusskaya sq., Moscow, 125047 Russia
+7 499 972 38 55
vgasilov@keldysh.ru

## ABSTRACT

We present a numerical tool for large-scale 3D radiative transport simulations related to high energy density plasmas (HEDP). Angular non-uniformity of photon distribution function can be accounted in second-order self-adjoint transport equation. We apply DG procedure to self-adjoint transport equation which leads to a set of elliptic-type equations. They may be solved independently giving good opportunity for using of any paradigm of parallelization. Note that accurate simulation requires the value of tens to hundreds for both M (the number of spectral groups) and N (the number of quadrature points on a sphere). Spatial discretization yields linear system with a symmetric positive definite matrix allowing application of effective linear solvers (Krylov solvers or Chebyshev - Richardson iterations). The energy balance is calculated via numerical radiative fluxes which are restored from discrete photon distributions by means of special quadratures for radiative fluxes. The numerical algorithm was incorporated into the scientific CFD code MARPLE3D (Keldysh Institute of Applied Mathematics – KIAM). We employed mixed element computational meshes (hexahedral, tetrahedral, prismatic cells and their combinations) up to tens million cells. Numerical experiments demonstrated good scalability at KIAM RAS K-100 scalable GPGPU-based hybrid computing system. We have obtained robust numerical procedures suitable for multiscale simulations in finely discretized computational domains. It's a promising technique for upcoming exaflop computing.

## Keywords

Radiative transport, high-temperature plasmas, numerical simulation, unstructured mesh, high-performance computing.

## 1. INTRODUCTION

Due to the expected advent of computer systems with a performance level of about a hundred petaflops, and up to one exaflops in the foreseeable future, we have an additional motivation for the development of new numerical methods. Along with the traditional requirements of high precision, algorithm uniformity, solution monotonicity, "low-cost" calculation performance, etc. a special attention should be paid to the efficient use of high-performance computing systems. Special important direction of this kind of work is a design of algorithms proper for hybrid CPU-GPU parallel systems.

The matter of our research is to provide a parallel technique for solution of the radiative transfer equation possessing such qualities which can satisfy requirements put forward due to necessity of a renovation of those program tools which are developed for a numerical analysis of experiments now carried out worldwide in the field of high energy density plasmas.

Here we present a new parallel technique of three-dimensional modeling of energy transfer caused by a thermal radiation in a gaseous matter heated up to appropriate temperature. The word "appropriate" means that the radiative energy fluxes significantly affect the energy balance in matter. The technique has the necessary qualities for use in software for predictive modeling of high-density energy plasmas. Meeting the challenges of a given subject area related to the consideration of a large set of non-linear processes ("multiphysics") is very difficult in the sense of providing high quality relevant computer models and high cost in terms of resources required and the performance of computers, which is a consequence of the different scales of structures and strong interdependence of hydrodynamic, thermal and radiative processes in the high-temperature plasma. These tasks are actively studied in connection with investigations of extreme states of matter in laboratory and natural conditions, the development of small-scale technologies, biomedical applications and others.

One of the most popular approaches to the calculation of the thermal radiation of the dense high-temperature plasma is the solution of the transport equation in the form of "diffusion" of radiation [1]. Diffusion model, along with the linear dependence of the flow of radiant energy from the temperature gradient environment includes the exact energy balance equation, which provides it with a wide range of applicability. However, the validity of using the diffusion model is justified only for the media in a state of local thermodynamic equilibrium [1, 2]. Various modifications of the "gradient" approximation for the flow of energy through the semi-empirical correction factors make the diffusion model significantly depending on specific conditions.

Having in mind the possibilities of modern high-performance systems, we can consider the direct solution of the transport equation of the photons, in the general case of three-dimensional space and two-dimensional angular variables. The general solution of the transport equation [1] can be used to calculate the radiation intensity along the characteristics ("rays"). To achieve a good quality of such a calculation it is required to link all pairs of cells in the computational domain by the rays of different families (the angular variables), so they can "share" the photon flux. Otherwise there is a risk of loss of accuracy in the numerical "beam effect" when the photons emitted in some intensely

radiating subdomain does not affect the energy balance in the other subdomains [2]. Experimental evaluation conducted for various unstructured computational grids show that acceptable accuracy calculation of grid-characteristic method is very costly. In addition, the corresponding algorithm does not scale well in the case of the parallel solution of a general system of equations of radiation plasmodynamics geometric domain decomposition.

Other known approaches to solving the problems of radiation in hydrodynamics (methods of spherical harmonic, moments, discrete ordinates, etc.) are also not free from the said drawbacks to a greater or lesser extent. If one needs a full account of heterogeneity of the environment, or the angular distribution of the radiation intensity, these methods are either not applicable or quite costly [2, 3].

For these reasons, with the aim to develop software for predictive modeling of high energy density plasma with highly variable emissivity and absorption coefficients, it's advisable to look for a replacement of diffusion and grid-characteristic methods for calculation of the radiation field.

Computational algorithm with good scalability, which, besides all, accounts for the angular dependence of the photon distribution function (or emission intensity), and significantly reduces the negative impact of "beam effect" can be built, if the original problem for the first order transport equation is replaced by the problem for the second-order equation with self-adjoint differential operator. And such passing from the original equation to the transformed one is possible. In 1951, V. Vladimirov [4] proposed a variational principle for velocity transport equation and studied classical correct formulation of the problem for the transport equation of the second order. In 1986, B. Chetverushkin [2] proposed to use similar approach for solving problems of radiative heat transfer, and showed in numerical experiments with model tasks practical suppression of the "beam effect" at moderate computational cost when solving transport problems using this method. In this paper, we propose the development of the technique [2] for the implementation on multiprocessor computers.

## 2. SELF-ADJOINT EQUATION OF RADIATIVE TRANSFER

Generally radiative transfer processes are analyzed by means of a stationary transport equation for the spectral radiation intensity

$$I_\nu\left(\mathbf{r},\mathbf{\Omega}\right)d\nu d\Omega = h\nu c\cdot f\left(\mathbf{r},\nu,\mathbf{\Omega},t\right)d\nu d\Omega,$$

which can be written as:

$$\lambda_\nu\left(\mathbf{r}\right)\left(\mathbf{\Omega}\,\nabla\right)I_\nu\left(\mathbf{r},\mathbf{\Omega}\right)+I_\nu\left(\mathbf{r},\mathbf{\Omega}\right)=J_\nu\left(\mathbf{r}\right), \quad (1)$$

where we use common notations:

$f$ is a photon distribution function which arguments are the radius vector of the observation point, photon frequency $\nu$, direction of a photon movement $\mathbf{\Omega}$ and time $t$, $\lambda = 1/\chi$ is the photon mean free path (i.e. the reciprocal of the opacity coefficient $\chi$), and $J_\nu$ is emissivity coefficient per unit volume of a substance (the amount of energy emitted per unit time in one steradian). In many cases of practical importance it is possible to assume that the emissivity depends only on a thermodynamic state of a substance, and almost has no dependence on the radiative intensity. Optical properties of a radiative medium are strongly depend on its density and temperature, and can vary for different frequency bands of the emitted/absorbed photons. Having the radiative intensity distribution over space and angle variables one can account for the radiative heat transfer, radiative losses or contributions into the energy balance of a plasma substance using expressions for the radiation energy density $U$ and radiation flux $\mathbf{W}$:

$$U=\frac{1}{c}\int_0^{4\pi}\int_0^{\infty} I d\nu d\Omega,\ \mathbf{W}=\int_0^{4\pi}\int_0^{\infty} I d\nu \mathbf{\Omega} d\Omega.$$

The effect of radiation on the energy of matter can be described by introducing a source term into the energy balance equation:

$$Q_{Rad}=-\operatorname{div}\mathbf{W}.$$

In certain cases (astrophysical problems, etc.) plasmodynamic model also includes the effect of the momentum carried by the photons, but this is a question we have not discussed, because it has no direct relation to the contents of this work.

An essential element of the radiation field description in multicharged plasmas is a multigroup approach. It is used as a tool for reasonably adequate representation of opacities and emissivities corresponding to different parts of the spectrum. Sometimes a multigroup model is used in certain computational procedures aimed at iterative refinement of photon absorption coefficients depending on the calculated intensity distribution (see [3] for quasi-diffusion or Eddington factor method). The essence of this widespread approach is the superposition principle, which is valid for the equation (1) by virtue of its linearity.

A total range of most representative photon frequencies (necessary to achieve the goals of modeling) is divided into M intervals (frequency groups) $\nu_1=0<...<\nu_i<...<\nu_M<\infty$, the transport equation (1) is solved for each group of this partition. To solve the equation (1) we use integral values of the intensity and spectral energy for the group $i$:

$$I_{[i]}=\int_{\nu_i}^{\nu_{i+1}} I_\nu d\nu,\ U_{[i]}=\int_{\nu_i}^{\nu_{i+1}} U_\nu d\nu,\ (1\le i\le M).$$

It is assumed that the opacity (absorption) and emissivity coefficients within each group are independent of the photon energy:

$\chi_\nu(T,\rho,\nu)=\chi_{[i]}(T,\rho)$ when $\nu_i<\nu<\nu_{i+1}$.

The total intensity of the radiated energy is calculated by summing over spectral groups:

$$I(\mathbf{r},\Omega,t)=\int_0^{\infty} I_\nu d\nu=\sum_{i=1}^{M} I_{[i]}.$$

For correct calculation of the energy balance we need, as a rule, just a few tens of spectral bands. In situations where the principal studied object is the emitted radiation spectrum essentially finer spectrum representation may be required, i.e. for more precise analysis we may need several hundred or even thousand spectral groups. Calculations for some spectral group are implemented independently from the others. The calculation formulas for all spectral intervals are identical and differ only in the values of emissivity and opacity coefficients.

A self-adjoint equation (2) can be obtained via replacing a photon emissivity function in (1) according the following formula

$$I(\mathbf{r},\mathbf{\Omega}) = \varphi(\mathbf{r},\mathbf{\Omega}) - \frac{1}{\kappa(r)}(\mathbf{\Omega}\nabla)\varphi(\mathbf{r},\mathbf{\Omega})$$

The auxiliary function $\varphi$ is symmetric: $\varphi(\mathbf{r},-\mathbf{\Omega}) = \varphi(\mathbf{r},\mathbf{\Omega})$, and satisfies the differential equation

$$-\mathrm{div}\left(\frac{1}{\chi(\mathbf{r})}\mathbf{D}(\mathbf{\Omega})\mathrm{grad}\varphi(\mathbf{r},\mathbf{\Omega})\right) + \\ +\chi(\mathbf{r})\varphi(\mathbf{r},\mathbf{\Omega}) = \chi(\mathbf{r})J(\mathbf{r}) \qquad (2)$$

where $\mathbf{D}(\mathbf{\Omega}) = \mathbf{\Omega}\otimes\mathbf{\Omega}$ $\left(D_{ij} = \Omega_i\Omega_j\right)$ is a dyadic or tensor product of two vectors.

Boundary condition proper to (2) takes the appearance:

$$\mathbf{n}\frac{1}{\chi(\mathbf{r})}\mathbf{D}(\mathbf{\Omega})\mathrm{grad}\varphi(\mathbf{r},\mathbf{\Omega}) - (\mathbf{\Omega}\cdot\mathbf{n})\varphi(\mathbf{r},\mathbf{\Omega}) = 0$$

when $(\mathbf{\Omega}\cdot\mathbf{n}) < 0$, $\mathbf{n}$ is the outward normal to the boundary (zero income flux).

To make differencing in the space of angular variables we use quadrature formulas by V. I. Lebedev, developed for calculations on the spherical surfaces [5], which are a set of angular directions $\{\boldsymbol{\omega}_k\}_{k=1}^N$ with weights $\alpha_k$. To construct a difference scheme we assume that within each solid angle, resulting from such partition, a value of $\varphi$ does not depend on the angular variables. Thus we get a set of N self-adjoint equations for N variables $\varphi_k$ with its own tensor $\mathbf{D}(\omega_k)$ for each quadrature node $\omega_k$.

Specific radiation energy is calculated by numerical integration by means of these quadrature nodes

$$U(\mathbf{r}) = \sum_{k=1}^{N}\alpha_k\varphi_k(\mathbf{r}) \approx \int_{4\pi} I(\mathbf{r},\mathbf{\Omega})d\Omega,$$

and radiation flux is calculated as

$$\mathbf{W} = \sum_{k=1}^{N} w_k\omega_k\frac{I(\mathbf{r},\omega_k) - I(\mathbf{r},-\omega_k)}{2} = \\ = -\frac{1}{\chi(\mathbf{r})}\sum_{k=1}^{N}\alpha_k\mathbf{D}(\omega_k)\mathrm{grad}\varphi_k(\mathbf{r}) \approx \int_{4\pi}\mathbf{\Omega}I(x,\mathbf{\Omega})d\Omega.$$

Source term in the energy balance equation can be found from the equation (2), without resorting to numerical differentiation:

$$Q_{Rad} = -\mathrm{div}\mathbf{W} = \chi(\mathbf{r})\left(J(\mathbf{r}) - U(\mathbf{r})\right).$$

Additionally, this model allows calculating the intensity of the radiation in the desired direction:

$$I(\mathbf{r},\omega_k) = \varphi_k(\mathbf{r}) - \frac{1}{\chi(\mathbf{r})}\omega_k\mathrm{grad}\varphi_k(\mathbf{r}).$$

For the numerical implementation of the considered model it is necessary to solve a set of ($M\times N$) independent elliptic type equations ($M$ is the number of spectral bands, $N$ is the number of quadrature points on the sphere). Every equation is solved independently from the others, which makes good opportunity for parallel computations of various types, including hybrid systems using traditional central processors and graphics accelerators. One should appreciate that to ensure adequate calculation accuracy the value of M and N can be several tens or even hundreds. A spatial difference approximation to the equation (2) results in a system of linear equations with a symmetric positive defined matrix, which allows solving the system using effective iteration methods, e.g. a technique based on Krylov subspaces, or explicit Richardson iterations with Chebyshev set of parameters.

Each of these elliptic equations has the appearance

$$-\left[\frac{\partial}{\partial x}\frac{1}{\chi}\left(\Omega_x^2\frac{\partial\varphi}{\partial x} + \Omega_x\Omega_y\frac{\partial\varphi}{\partial y} + \Omega_x\Omega_z\frac{\partial\varphi}{\partial z}\right) + \right. \\ +\frac{\partial}{\partial y}\frac{1}{\chi}\left(\Omega_x\Omega_y\frac{\partial\varphi}{\partial x} + \Omega_y^2\frac{\partial\varphi}{\partial y} + \Omega_y\Omega_z\frac{\partial\varphi}{\partial z}\right) + \qquad (3) \\ \left.+\frac{\partial}{\partial z}\frac{1}{\chi}\left(\Omega_x\Omega_z\frac{\partial\varphi}{\partial x} + \Omega_y\Omega_z\frac{\partial\varphi}{\partial y} + \Omega_y^2\frac{\partial\varphi}{\partial z}\right)\right] + \chi\varphi = \chi J,$$

where ($\Omega_x$, $\Omega_y$, $\Omega_z$) are the direction cosines of the corresponding angular direction, and $\chi$ and $J$ are individual for each spectral interval.

To discretize the equation (3) we use one variant of the known Galerkin method with discontinuous basis functions for the variables defined in the grid cells. In the case of a regular rectangular grid this procedure provides a difference scheme on a template with 27 nodes. If we use a grid with irregular structure (unstructured) a template includes all cells having a common node with the central cell.

An important feature of our algorithm is a preliminary transformation of rotation, which is applied to the elliptic equation (3): ($\Omega_x$, $\Omega_y$, $\Omega_z$) → (1, 0, 0). By means of this transformation we exclude from the equation (3) mixed derivatives with respect to spatial variables, thereby the structures corresponding to its difference analogue are greatly simplified which significantly contributes to the acceleration of iterative processes of solving the equation (3).

## 3. NUMERICAL RESULTS

The technique of radiative transport calculation has been implemented as a part of an object-oriented code MARPLE3D [6] developed in KIAM RAS for numerical investigation of problems in the field of radiative magnetogasdynamics at massively parallel systems of a cluster type. Numerical experiments on testing of the technique have been carried out at a scalable GPGPU hybrid cluster K-100 (KIAM, Moscow).

Let's consider some test examples used for a accuracy/convergence study with respect to spatial and angular variables. Test studies were performed using 3D meshes of different element types (tetrahedrons, cubes, triangular prisms). Calculations of integral values (radiation energy per unit volume, radiative flux, etc.) were done via quadrature formulas with angular directions varied from 1 to 21 in the octant. For solving appropriate linear equation systems we have used BICGSstab -

biconjugate gradient method with stabilization, in combination with symmetric Gauss-Seidel preconditioner.

Here we present some results of numerical experiments. The test problem relates to the radiation produced by a body having the form of an infinite cylinder of a given radius, filled with a substance which emissivity and the absorption coefficient are constant, and outside of a cylinder these coefficients are equal to zero. We have calculated radiation intensity and radiation energy flux. Here numerical results are compared versus analytical solutions (see Figure 1).

Further improvement of the numerical solution accuracy without increasing the number of angular directions can be achieved by making refinement of the spatial grid. Owing to the rotation transformation marked above the number of iterations decreased by a factor of 2 or 3 as compared with the original equation including cross derivatives.

We have found that the proposed method correctly reproduces the limiting cases: completely isotropic radiation (no "beam effect" – see Figure 2) and propagation of a "laser beam" with the acute δ-shaped angular distribution in absorbing media. In the latter case the use of the rotation transformation not only reduces the number of iterations, but also eliminates the "numerical diffraction" effect and allows adequate simulating the extremely anisotropic radiation distributions (see Figure 3).

## 4. CONCLUSIONS

Angular non-uniformity of photon distribution function can be accounted effectively by means of a second-order self-adjoint transport equation. DG procedure to self-adjoint transport equation leads to a set of (MxN) elliptic-type equations. They may be solved independently giving good opportunity for using of any parallelization paradigm. Accurate simulation requires the value of tens to hundreds for both M (the number of spectral groups) and N (the number of quadrature points on a sphere). Spatial discretization yields linear system with a symmetric positive definite matrix allowing application of effective linear solvers.

Numerical experiments demonstrate good scalability at KIAM RAS K-100 scalable GPGPU based hybrid computing system. Best performance is achieved in configuration 1CPU+1GPU. Speedup is 1.5 for the entire linear system solution.

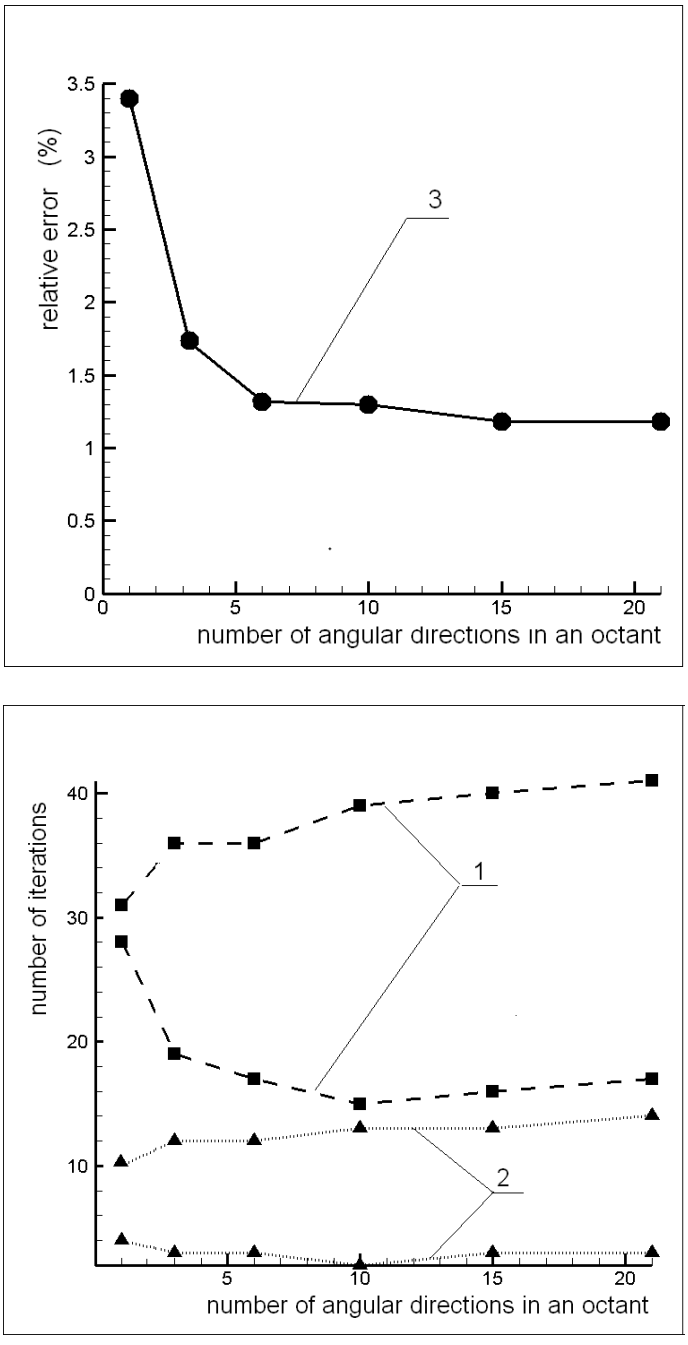



**Figure 1. Convergence of the solutions with increasing number of angular directions: 1 - maximum and minimum (depending on direction) the number of iterations of the original form of the equation (3); 2 - maximum and minimum (depending on direction) after the number of iterations of rotation;**
**3 - relative error in the solution.**

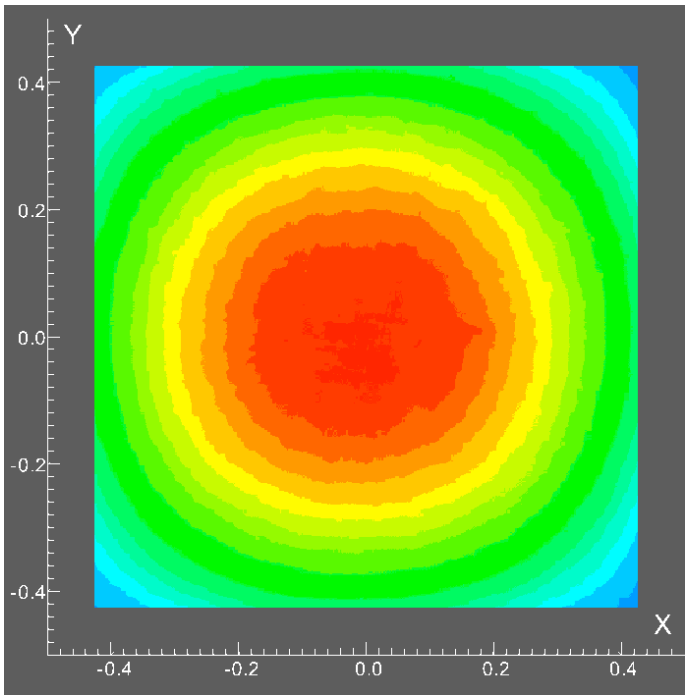



**Figure 2. Isotropic radiation of a solid sphere R = 0.1: $\chi = 10$, $J = 10$ when r < R; $\chi = 1$, $J = 0.00025$ when r > R. 21 angular directions in an octant.**

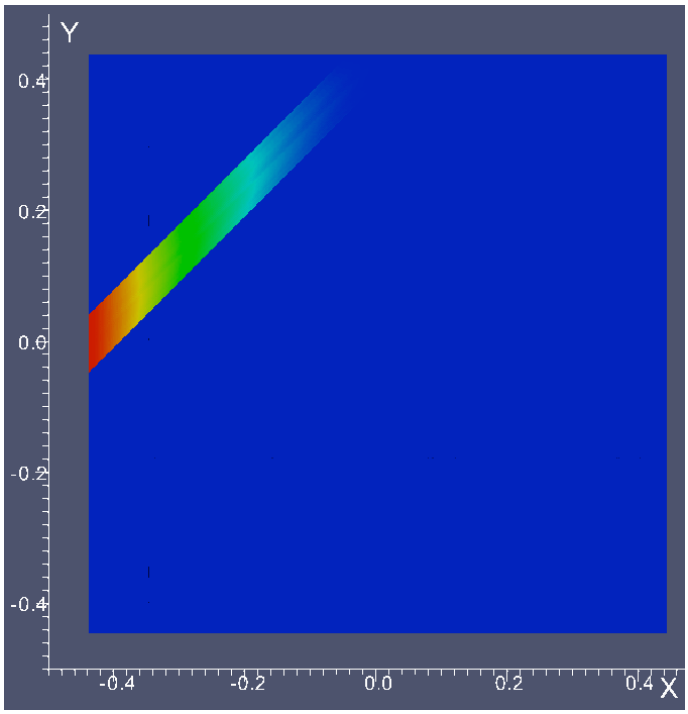



**Figure 3. Uniform beam propagation in preferential direction. $\chi = 1$, $J = 0$ inside the domain; incident beam: direction $\Omega = (0.707, 0.707, 0)$, $J = 100$, d = 0.05, center at the boundary point (-0.5, 0, 0).**

The advantages of the algorithm:

– Symmetrical self-adjoint equation replaces initial unsymmetrical transport equation. Thus the solution of the problem is equivalent to finding the minimum of a certain functional [4].

– Scalability of the method is determined by the choice of linear solver. The state-of-the-art linear solvers are efficient and permanently developing.

We have obtained robust numerical procedure suitable for multiscale simulations in finely discretized computational domains. It's a promising technique for upcoming exaflop computing.

## 5. ACKNOWLEDGMENTS

The work was supported by RFBR grants 14-01-00678 and 15-01-06195.

The computations were carried out at supercomputers K-100 (KIAM RAS), Lomonosov (RCC MSU), MVS-100K (JSCC RAS).

## 6. REFERENCES

[1] Ya. B. Zel'dovich and Yu. P. Raiser. Physics of Shock Waves and High-Temperature Hydrodynamic Phenomena. Moscow, Fizmatlit, 2008.

[2] B. N. Chetverushkin Mathematical modeling of problems in the radiative gas dynamics. Moscow, Science, 1985.

[3] S. T. Surzhikov Thermal radiation of gases and plasma. M.: N. Baumann Technical State University, 2004.

[4] V. S. Vladimirov, Mathematical problems of one-velocity transport theory. Proceedings of the Steklov Institute of Mathematics. VI Steklov. Moscow, 1961, p. 158.

[5] V. I. Lebedev. About squaring at a spherical surface// Computational Mathematics and Mathematical Physics, Volume 16, № 2 - Moscow, 1976, pp 293-306.

[6] V. Gasilov et al. Towards an Application of High-performance Computer Systems to 3D Simulations of High Energy Density Plasmas in Z-Pinches. In: Applications, Tools and Techniques on the Road to Exascale Computing. IOS Press, “Advances in parallel Computing”, 2012, Vol. 22, p. 235-242.

# Ensuring Efficiency of Exascale Supercomputer Centers

Vladimir Voevodin
RCC MSU
119234, Leninskie Gory, 1, bld. 4
Moscow, Russia
+7-495-939-51-66
voevodin@parallel.ru

Vadim Voevodin
RCC MSU
119234, Leninskie Gory, 1, bld. 4
Moscow, Russia
+7-495-939-52-16
vadim@parallel.ru

## ABSTRACT
In this paper, we describe a set of systems aimed to ensure efficiency of supercomputer centers at every level of concern. This includes application and cluster performance monitoring and analysis, user and resource management, supercomputer reliability and viability. Each system is addressed to solve one particular field of research, but they are designed to be interrelated allowing for more holistic and deep efficiency analysis overall.

## Keywords
Exascale, supercomputer, high-performance computing, performance, efficiency of applications, efficiency of supercomputers, reliability, monitoring, visualization, user management, resource management, efficiency analysis.

## 1. INTRODUCTION
We are used to the colossal abilities of supercomputers and expect corresponding returns from them. These expectations may be justified, but real life is not always so favorable. Everyone is aware of how poorly supercomputers perform on real-life applications: in most cases it's at just a small percentage of the peak performance characteristics. But only a few actually suspect what the efficiency of a supercomputing center is in general [1]. While a supercomputer is about as efficient as a steam engine for a single application, the efficiency of an entire supercomputing center is only a fraction of that figure. While losses may be barely visible at each specific step, they increase manifold as multiple user applications are processed. No detail is too small here, and every element of a supercomputing center must be reviewed thoroughly – from the task queuing policy and the application flow structure to system software setup and the efficient operation of engineering infrastructure.

The efficiency of supercomputing applications is only one aspect of the issue. An equally important issue today is control over the proper operation of the computing environment of supercomputing systems. The main reason is the unprecedented growth of parallelism degree. Thousands of users and applications, hundreds of thousands of computing nodes, processors, accelerators, ports, cables, software and hardware components, millions of processing cores, processes, events, messages… And all this has to work in complete harmony as a single system. A task scheduler hangs – and powerful resources are wasted idling. An error occurred in a single InfiniBand cable resulting in many broken and resent packets – and application performance drops (and this usually goes unnoticed by users). Skillful control over the state of supercomputer components is needed to promptly detect and isolate failures and errors. This control is a daunting task, due to the immense set of components to be monitored. But the main challenge is to ensure complete and continual monitoring. Such efficiency-related issues are important today, but their importance will grow significantly for future exascale supercomputers [2].

To ensure the efficiency of large supercomputing centers, we use an approach based on a set of interrelated software systems, technologies and instruments. It is designed for a coordinated analysis of the entire hierarchy of a supercomputing center: from hardware and operating system to users and their applications. This analysis requires going through both system level (CPU-related data, network performance, I/O operations, memory subsystem, etc.) and the upper level of entire supercomputing center (users, applications, queues, task flows, supercomputer sections, etc.).

The paper is organized as follows. Next section includes brief description some of the most interesting researches and tools for the discussed and related problems. In section 3 we propose set of software systems intended for ensuring efficiency, with each system described in separate subsection. Finally, last section contains conclusions and acknowledgments.

## 2. RELATED WORK
Our suggested approach touches upon a number of issues related to the efficiency of supercomputer systems. These can be logically broken down into several areas – monitoring, system reliability and viability, efficiency analysis of individual user applications and the entire supercomputer system. Each of these areas has been thoroughly studied individually. We will cite several important works that touch upon related issues in each case. It should be mentioned that all these systems are intended to address one of the selected areas, while our ultimate goal is to create a single set of interrelated solutions that helps analyzing all aspects of cluster system's efficiency.

There are many systems for distributed monitoring of the status of a computing system. Some of the most common names in the HPC area are open source solutions such as Ganglia, Nagios, Zabbix, Zenoss [3], as well as various commercial software suites, like HP Network Node Manager [4], ClustrX [5], etc.

A large set of issues is related to comprehensive behavior analysis of individual applications. Different methods are used for this purpose (tracing, profiling, emulating program execution, etc.), and a number of effective tools have been developed which apply these approaches in the HPC area. These include, for example, Intel Vtune and Intel Parallel Studio XE [6], Scalasca, Vampir, Score-P [7], Valgrind, and many others.

In addition to analysis of the behavior of individual programs, it is also needed to analyze the efficiency of the entire system. In

particular, the issue of analyzing the task flow structure to check for optimization is of interest. Performing this analysis requires, for example, a flow analysis method [8] (comparing to threshold values), even though this approach is often limited to visual analysis, and is provided by modern scheduling tools such as LoadLeveler [9], Torque, Cleo[10] and Bright Cluster Manager [11].

Two other issues stand somewhat apart. The first is the issue of ensuring the reliability and viability of the system, which is not directly related to efficiency but can affect it substantially. Many theoretical aspects are shown in [12]. From a practical point of view, the monitoring systems mentioned above are responsible for tracking the current status of the cluster. However, other issues are equally important, such as failure predictions [13,14], or various approaches to identifying the true causes of failures [15,16,17].

Another issue not directly related to efficiency, but also of great importance, is that of visualizing the status of the supercomputer in general, or its individual components. Existing solutions include the aforementioned monitoring systems, which often can visualize the data, and specialized tools and libraries for data visualization, such as RAW, D3 [18], canvasjs, Gephi [19] and others.

## 3. THE SET OF SOFTWARE SYSTEMS

System-level data is collected by **the total monitoring system**. Ensuring the efficient functioning of a supercomputing center requires monitoring absolutely everything that happens inside the supercomputer, and that task alone requires sifting through tons of data. Even for the relatively small “Lomonosov” supercomputer at the Moscow State University (1.7 Pflops peak, 100 racks, 12K nodes, 50K cores) [20], the necessary data arrive at the rate of 120 Mbytes/s (about 30 different metrics analyzed for each node, measured at the frequencies of 0.01-1.00 Hz). It means 3+ Pbytes for 365 days of a year. This is an incredibly large amount of data which must be collected with a minimum impact on application performance. A total monitoring system can be built by reaching a reasonable compromise on two key issues: what data must be stored in the database, and when should it be analyzed.

Basically monitoring data is used in other systems, but some analysis can also be made based only on data itself. For example, fig.1 shows distribution of usually abnormal LoadAVG values for different partitions of Chebyshev supercomputer. LoadAVG measures the amount of work being performed on one node during some period, so in most cases this value does not exceed 8 (since each node has 8 cores) or 16 (if HyperThreading is used). It can be seen that surprisingly often LoadAVG exceeds value of 16, which means that some side activity happens rather often. This is a reason for administrator to pay attention to this issue.

A modern supercomputing center is not only about managing supercomputers per se – this function has been studied well enough already – but about efficiently organizing a multitude of adjacent issues: managing software licenses, user quotas, project registration and maintenance, technical support, tracking warranty and post-warranty service and repairs, and many other tasks. All these issues are closely linked to one another and, given the number of components, it is clear how hard it is to efficiently organize work flows and maintain the entire supercomputing center in an up-to-date condition.

**The OctoShell system** combines data on all critical components for efficiently operating a supercomputing center, describing their current state and connections between components: the hardware and software components being utilized, users, projects, quotas, etc.

Here is one example. License of product A will be expired in 1 month. Is it necessary to extend it? Looking at this product usage stats in the Octoshell system, we can see that product A was being used by hundreds of users, and that means it is in demand. More detailed analysis of user activity shows that 50% of users run this product on more than 1000 cores, which means we need to include a lot of tokens to this license. Moreover, studying run times of this product in Octoshell, we can see there are some runs finished within several seconds, which usually indicates an error has occurred. In this case either correctness of product setup must

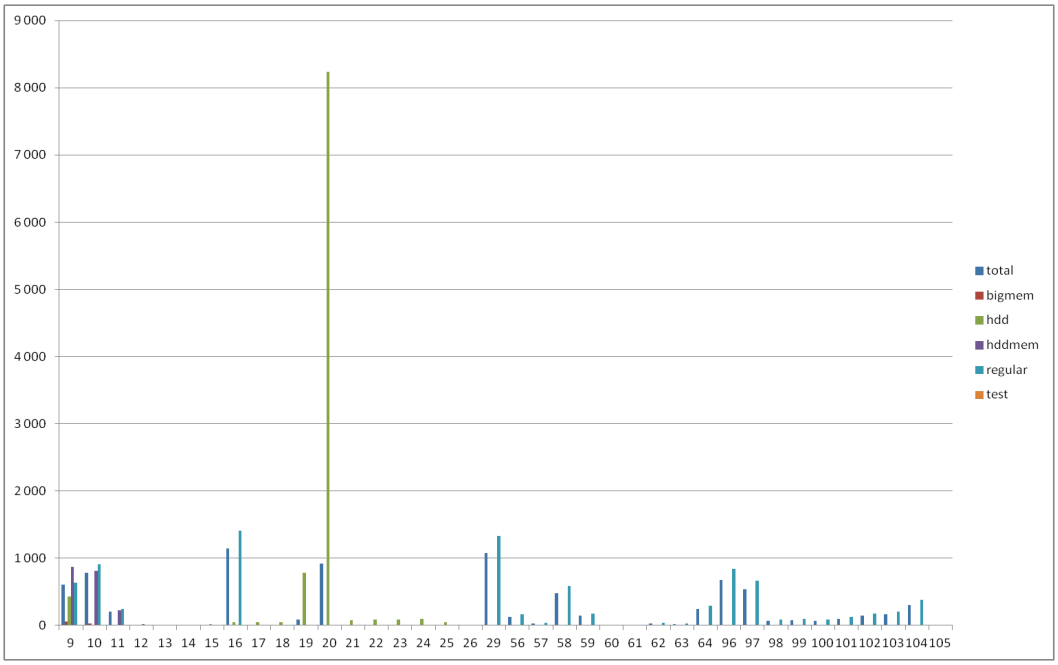



**Fig. 1. Distribution of abnormal LoadAVG values**

be checked or more proper product usage manual should be provided.

The practice of supercomputing center maintenance defines a set of strict technology and tool requirements for supporting the functioning of supercomputer systems. This includes maintaining high performance for the supercomputing infrastructure, constant monitoring of potential emergencies, continual performance monitoring for all critical software infrastructure modules, automatic decision-making on eliminating emergency situations and increasing the performance of supercomputer operations, guaranteed operator notification about the current status of the supercomputer and actions taken by the automatic maintenance system, and others. Until all these requirements are met, neither the efficient operation of a supercomputing center, nor the safety of its hardware can be guaranteed.

The goal of **the Octotron project** is to design an approach that guarantees the reliable autonomous operation of large supercomputing centers. The approach is based on a formal model of a supercomputing center, describing the proper functioning of its components and their interconnections. The supercomputer can continually compare its current state with the information from the model. If practice (monitoring data of the current supercomputer state) deviates from theory (the supercomputer model), Octotron can perform one of the predefined or dynamically selected actions, such as notifying the operator via email and/or SMS, disabling the malfunctioning device, restarting a software component, displaying an alert on the systems administrator screen, etc. No human is capable of monitoring millions of components and processes inside a supercomputer, but the supercomputer itself can do this. For example, fig. 2 shows number of suspicious events that were caught using our Octotron system in Lomonosov supercomputer within 1 month. It reaches almost 400 events during one 12h interval, which makes it almost impossible for operational manual respond.

Importantly, this approach guarantees not only reliable operation of the existing fleet of systems at a supercomputing center, but also ensures maintenance continuity when moving to a new generation of machines. Indeed, once an emergency situation arises, it is reflected in the model, along with the root causes and symptoms of its existence, and an adequate reaction is programmed into the model. It may never happen again in the future, but if it does, it will immediately be intercepted and isolated before it has any effect on the supercomputer operation.

The main purpose of **the Situational screen** is to give system administrators full and prompt control over the state of the supercomputing center. The screen provides detailed information on what is happening inside the supercomputer, and provides updates on the status of hardware and system software, task flow, individual user activity and/or the performance of individual supercomputing applications. The situational screen and its underlying instruments are designed to meet a number of strict requirements: the need to reflect all key performance parameters of the supercomputing systems, grant of complete control over their status, scalability, expandability, configuration and minimization of the impact of the situational screen on the supercomputer performance.

Based on many years of experience in administrating supercomputer systems, we can point out the most important components which the situational screen must reflect:

- supercomputer hardware: computing modules, hosts, secondary servers, storage, networks, engineering infrastructure elements. Here it is important to know both the overall characteristics and the status of individual components;
- system software: the status for the entire supercomputer and individual nodes, conducting audits with specified criteria for any number of nodes;
- user task flows: tasks queued, tasks being executed, current and historical data, statistics on various partitions of the supercomputer, statistics on application package usage;
- activity of a particular user: what is being executed now, what is queued, overall performance over a recent time period, etc.;
- status of any working application: statistics on performance, locality, scalability and system-level monitoring data.

**The OctoStat statistics collection system** provides highly valuable information on the performance of supercomputing systems with regards to the flow of tasks queued for execution. Task queue length, waiting time before execution, the distribution of processors needed for applications of different classes, statistics by partitions of the supercomputer, by users or application packages, intensity of application flow for execution at different times... All of this – and many other metrics – need to be analyzed and used to optimize quotas, priorities and strategies for distributing valuable supercomputing resources. Moscow State University Supercomputing Center uses OctoStat to generate daily statistics on the supercomputer usage, enabling prompt decision-making.

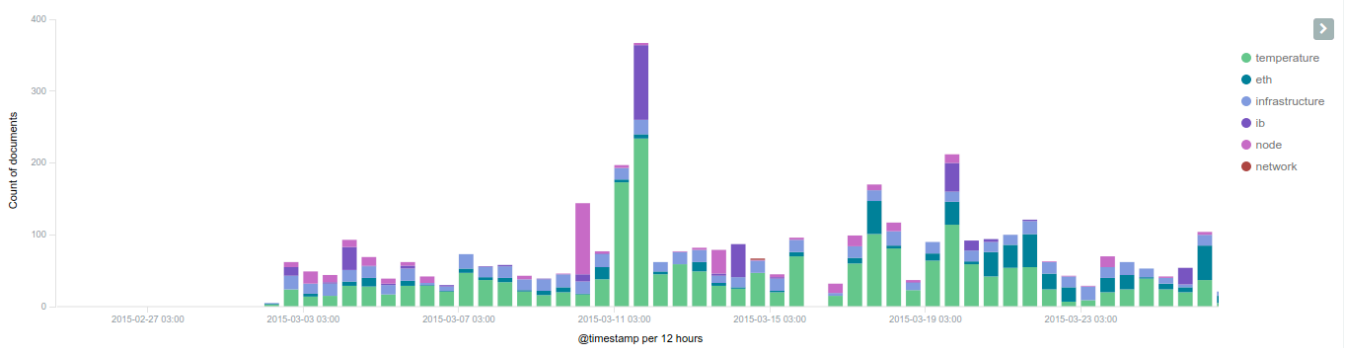



**Fig. 2. Timeline of event count of different types (within 1 month)**

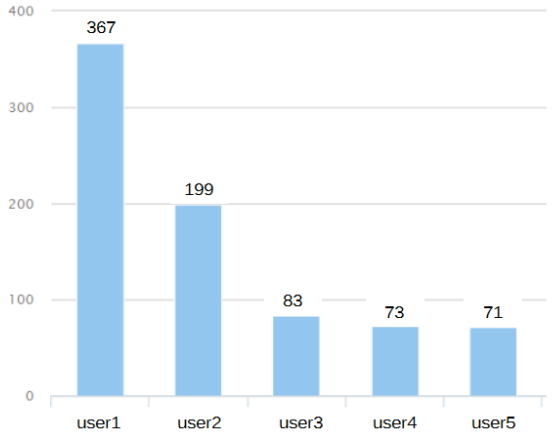



**Fig. 3. Top 5 users with worst wait_time/run_time ratio**

Fig. 3 shows top list of users with worst (wait time/run time) ratio. This means these users have to wait in queue really long comparing to time their task need to run. There could be several different reasons for that (or even all of them) – heavy workload of supercomputer, not optimal tasks manager work, or maybe their tasks finish really fast. Nevertheless, this situation is worth administrator's attention.

OctoStat is also intended to analyze how behavior of task flow matches the architecture of supercomputing system. On fig. 4 the timeline of average Infiniband interconnect usage in different partitions of Chebyshev supercomputer is shown. The peak throughput is 20 Gbps, but in most cases average throughput does not exceed 80 Mbps. This indicates that such fast interconnect is usually not in demand – only a few application fully use the potential of current Infiniband interconnect. This can be taken into account while designing new systems in order to build more cost effective supercomputer – with less expensive interconnect but instead, for example, with more computing nodes or memory per node.

One key issue when analyzing the efficiency of supercomputing centers is the user application runtime performance analysis. Performance of a supercomputer on real-life applications today is quite low, amounting to just a small percentage of the peak performance characteristics. As parallelism increases, this figure is bound to decline. There are a multitude of reasons for this decline in efficiency, and we use supercomputing system hardware and software monitoring data to identify them. A lot of data is required for the analysis: CPU load, cache misses, flops, number of memory references, LoadAVG, IB usage, I/O usage, etc. This data is used to build a runtime profile for each application, which is presented as a set of graphs demonstrating the changes in monitoring data during the application execution. This profile, along with a number of aggregate system characteristics (we call it **JobDigest of an application**) gives a good first estimate of the application performance and its features. If any issues are observed with the application overall performance, additional analysis of the monitoring data is performed to identify the causes. Particular attention is paid to analyzing application properties such as efficiency, data locality, performance, and scalability, which are extremely important for the supercomputing systems of the future.

One good example of JobDigest is provided on fig. 5, with timelines of flops, L1 cache misses per second and number of memory reads per second. Here we can see that during first part of the task there was normal behavior – amount of computations (flops) and memory usage (l1 mis and memory reads) is rather common, as well as network activity (not shown here). But in

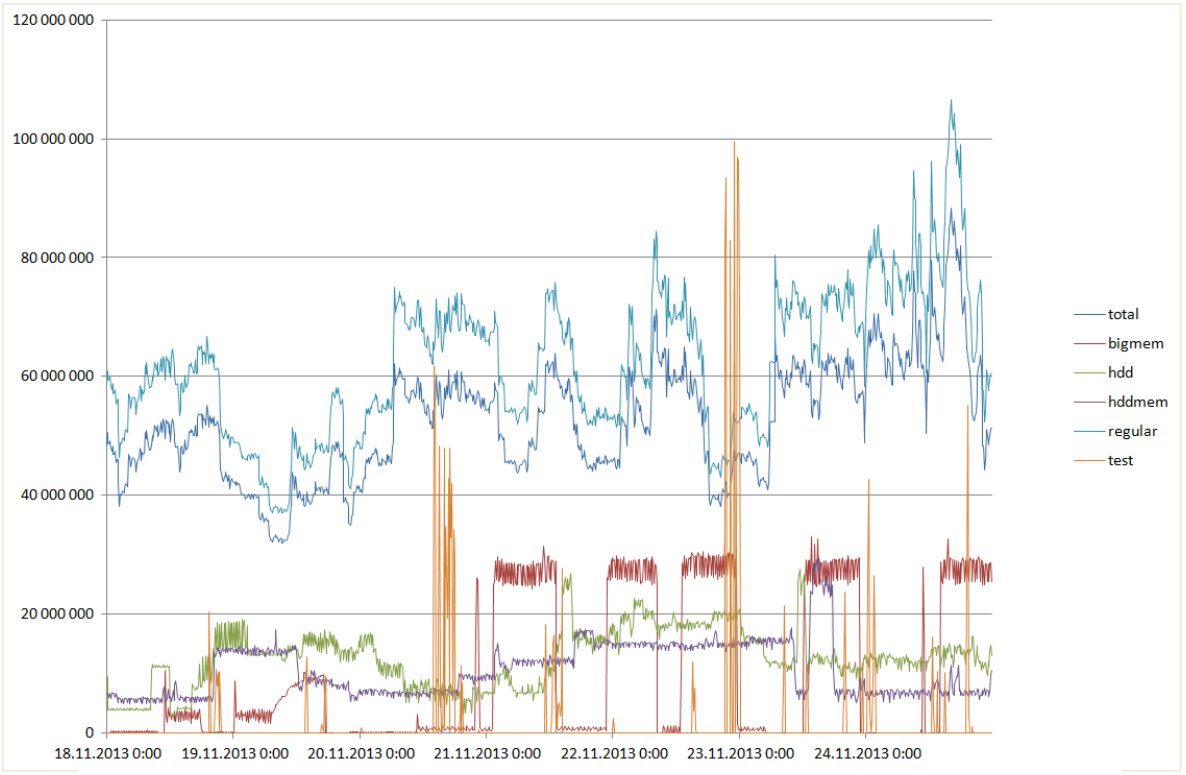



**Fig. 4. Intensity of Infiniband usage for different partitions (Chebyshev supercomputer)**

second part all activity besides memory reads drop nearly to zero. No data sent over network, no cache misses, no flops – generally this means that task fell into infinite loop, asking only for loop counter. So it is necessary to inform user that created this task about abnormal task behavior. It is worth mentioning that in this case average system characteristics could not help to find this anomaly because of rather long first part.

All the systems described above are closely linked to one another, ensuring high efficiency for large supercomputing centers. Of course, they were designed while taking all key aspects of maintenance and usage into account for existing supercomputers. At the same time, the architecture of these systems was designed to be able to adopt large scale, complexity, high degree of parallelism and performance of the forthcoming exascale supercomputers.

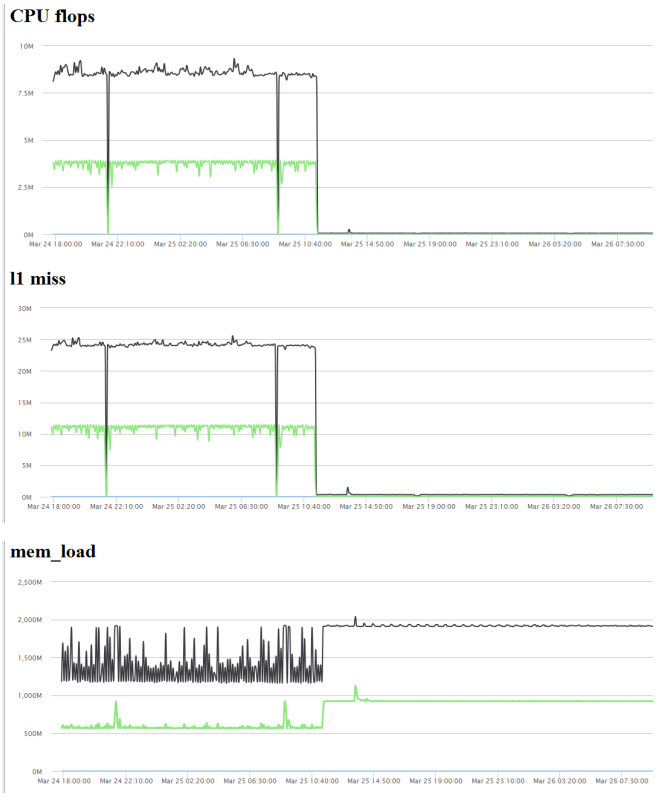



**Fig. 5. Timeline of flops, L1-cache misses and memory reads for a particular task**

## 4. CONCLUSIONS

A set of software systems presented is this article is intended to help users as well as administrators of supercomputers to ensure efficiency of every level of supercomputing center usage – from individual users tasks to the whole cluster. This includes a range of different efficiency issues, concerning not only performance monitoring and analysis but also visualization and reliability problems. This set is being actively developed and used by a team from research computing center at the Moscow State University.

**Acknowledgments.** The results were obtained with the financial support of the Ministry of Education and Science of the Russian Federation, Agreement N 14.607.21.0006 (unique identifier RFMEFI60714X0006).

## 5. REFERENCES

[1] Voevodin Vladimir. Medical practice: diagnostics, treatment and surgery in supercomputing centers. Presentation at the International Advanced Research Workshop on High Performance Computing, Cetraro, Italy, 2014. Available at: http://www.hpcc.unical.it/hpc2014/pdfs/voevodin.pdf. Date accessed: 08 May. 2015.

[2] Cappello, Franck et al. Toward Exascale Resilience: 2014 update. Supercomputing frontiers and innovations, [S.l.], v. 1, n. 1, p. 5-28, jun. 2014. ISSN 2313-8734. Available at: http://superfri.org/superfri/article/view/14. Date accessed: 08 May. 2015. doi:http://dx.doi.org/10.14529/jsfi140101.

[3] List of different monitoring tools. http://www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html

[4] HP Network Node Manager main page. http://www8.hp.com/us/en/software-solutions/network-node-manager-i-network-management-software/index.html

[5] ClustrX description. http://www.t-platforms.com/products/software/clustrxproductfamily.html

[6] Intel Parallel Studio XE main page. https://software.intel.com/en-us/intel-parallel-studio-xe

[7] Score-P main page. http://www.vi-hps.org/projects/score-p/

[8] Feitelson D. G., Rudolph L. Toward convergence in job schedulers for parallel supercomputers //Job Scheduling Strategies for Parallel Processing. – Springer Berlin Heidelberg, 1996. – С. 1-26.

[9] LoadLeveler description. http://www-03.ibm.com/software/products/en/tivoliworkloadschedulerloadleveler

[10] Cleo description and source code. http://sourceforge.net/projects/cleo-bs/

[11] Bright Cluster Manager main page. http://www.brightcomputing.com/Bright-Cluster-Manager

[12] Snir M, Wisniewski R, Abraham J, Adve S, Bagchi S, Balaji P, Belak J, Bose P, Cappello F, Carlson B and others. 2014. Addressing Failures in Exascale Computing. International Journal of High Performance Computing Applications

[13] Hoffmann, G. A. 2006. Failure Prediction in Complex Computer Systems: A Probabilistic Approach. Shaker Verlag

[14] Murray, J., Hughes, G., and Kreutz-Delgado, K. 2003. Hard drive failure prediction using non-parametric statistical methods. Proceedings of ICANN/ICONIP.

[15] Lu K. et al. Iaso: an autonomous fault-tolerant management system for supercomputers //Frontiers of Computer Science. – 2014. – Т. 8. – №. 3. – С. 378-390.

[16] Chuah E. et al. Diagnosing the root-causes of failures from cluster log files //High Performance Computing (HiPC), 2010 International Conference on. – IEEE, 2010. – С. 1-10.

[17] Bodik P. et al. Fingerprinting the datacenter: automated classification of performance crises //Proceedings of the 5th European conference on Computer systems. – ACM, 2010. – С. 111-124.

[18] D3 site. http://d3js.org/

[19] Gephi main page. http://gephi.github.io/

[20] V. Sadovnichy, A. Tikhonravov, Vl. Voevodin, and V. Opanasenko "Lomonosov": Supercomputing at Moscow State University. In Contemporary High Performance Computing: From Petascale toward Exascale (Chapman & Hall/CRC Computational Science), pp.283-307, Boca Raton, USA, CRC Press, 2013.

# Sniper: Simulation-Based Instruction-Level Statistics for Optimizing Software on Future Architectures

Wim Heirman
wim.heirman@intel.com

Alexander Isaev
alexander.isaev@intel.com

Ibrahim Hur
ibrahim.hur@intel.com

Intel Corporation
Leuven, Belgium

## ABSTRACT

In this paper we address the problem of optimizing applications for future hardware platforms. By using simulation—traditionally a tool for hardware architects—applications, libraries and compilers can be optimized before hardware is available, allowing new machines to start doing useful scientific work more quickly. However, traditional processor simulators are not very user-friendly and are, due to their extreme level of detail, too slow to run applications with large input sets or allow for interactive use. In contrast, the Sniper many-core simulator uses higher abstraction level models, trading off some accuracy for a much higher simulation speed. By adding instrumentation into the simulator that can annotate performance information at fine granularity, down to individual instructions, it becomes a valuable tool for software optimization on future architectures.

## 1. SHIFT-LEFT OF SOFTWARE DEVELOPMENT

Performance projections of future systems are crucial for both software developers and processor architects. Developers of applications, runtime libraries and compilers need predictions for tuning their software before the actual systems are available, and architects need them for architecture exploration and design optimization. Simulation is one of the most commonly used methods for performance prediction, and developing detailed simulators constitutes a major part of processor design. Traditionally, simulators were only used in the exploration and design phases of product development. This means software development and optimization have to wait until (prototype) hardware becomes available, see Figure 1(a). This puts the software development effort on the critical path towards bringing products to market (from the point of view of the vendor), or delays the point at which new machines can start running optimized science codes (for the HPC user).

Recently, much effort has been put into enabling software developers to start work early; at least before final hardware is available, and ideally to make application optimization part of the hardware exploration process—enabling a true co-design of hardware and software where both can be optimized in combination (Figure 1, b). The lack of available hardware requires early software development and optimization to be done using some form of performance simulation. Creating detailed, usually cycle-accurate simulators is part of the hardware development and validation effort. However, most detailed simulators, while very accurate, are too slow to simulate meaningful parts of applications—especially in the context of many-core systems with large caches. Instead, these simulators typically run short traces of code and require great care and often manual effort to both select these traces and provide adequate warmup of structures with long-living state such as caches and branch predictors.

(a) Traditional flow


(b) Enabling early software development

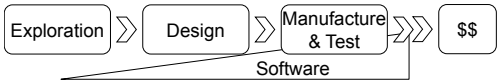



**Figure 1: Shift-left of software development enables quicker time-to-market.**

## 2. HIGH-LEVEL SIMULATION

By trading off some accuracy for the ability to run larger parts of the application, higher abstraction level simulation can play a valuable role in both software tuning and architecture exploration. Simulation speed can be increased by not modeling some hardware components that are known to be a bottleneck (e.g., instruction caches in many HPC codes), or by moving away from structural models that try to model exactly what each hardware component is doing and instead using analytical models such as interval simulation [4] or instruction-window centric models [2] for the processor core, or queuing theory for on-chip networks.

*Sniper* [1] is an x86 many-core simulator that combines many of these techniques, in addition to being built on a parallel simulation framework which can make use of modern multi-core hardware. These properties result in an acceptable accuracy (around 20% average absolute error compared to Nehalem hardware) but much improved simulation speed (around 1 MIPS, which is around 1000× faster than typical industrial detailed simulators). This brings interactive (overnight) runs of representative parts of an application within reach, greatly speeding up the optimization cycle.

## 3. ACCURATE PERFORMANCE METRICS

On real hardware, many performance counters are available that can give valuable insight into how codes are behaving. Cache miss rates are especially valuable, as these often

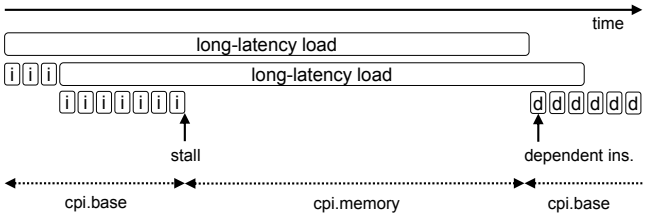



**Figure 2: Overlapped execution of cache misses and independent instructions on out-of-order processors.**

indicate long pauses in the execution of instructions by the processor leading to low performance (typically expressed in instructions per clock cycle, IPC). In the simulator, the behavior of structures such as caches is modeled in detail so extracting statistics such as hit rates is trivial.

However, the use of miss rates as indicators for application performance can be misleading, as indicated in Figure 2 which plots the execution timeline of a typical section of code when running on modern hardware. Load operations that miss in the processor caches usually take many tens or even hundreds of clock cycles, whereas loads that hit in cache or compute instructions take only a handful of cycles. One could therefore assume that the length of time taken to execute a section of code is proportional to the number of instructions, increased by the number of cache misses multiplied by the typical latency of a cache miss. But this does not take into account the fact that out-of-order processors can continue executing independent instructions, including potentially other long-latency loads, while waiting for the original cache miss to be serviced.

To alleviate this problem, hardware architects often employ the concept of the CPI stack [3]. This is a stacked bar graph which breaks up an application's execution time into a number of components, and is normalized to cycles per instruction (for a CPI stack) or to the total number of clock cycles (for a cycle stack). Each component in the stack denotes the *penalty* caused by a different hardware component, taking into account the fact that many miss events may overlap. In the case of the execution shown in Figure 2, all time spent executing compute instructions is accounted for in the *base* component (denoting the execution time assuming the processor would be capable of reaching its maximum performance all the time) while only the stall time, when no instructions other than the cache misses are in progress, is accounted for in the memory penalty component. This way, each clock cycle of execution is assigned to that hardware component that was on the critical path of execution. Solving a given stall that is visible on the cycle stack will therefore be guaranteed to lead to increased performance. In contrast, ignoring the fact that much of the cache miss latency is overlapped would overestimate its effect, potentially leading programmers to spend time to reduce cache misses or other miss events that are not performance critical.

While it is only very recently becoming possible to measure CPI stack components using hardware performance counters [6], they are natively supported in the Sniper simulator [5]. Figure 3 plots an example CPI stack obtained from running an FFT workload on a simulated dual-socket, eight-core Nehalem machine (with one software thread pinned to each core), and illustrates some interesting performance effects. Comparing the behavior of threads 0—3 with that of

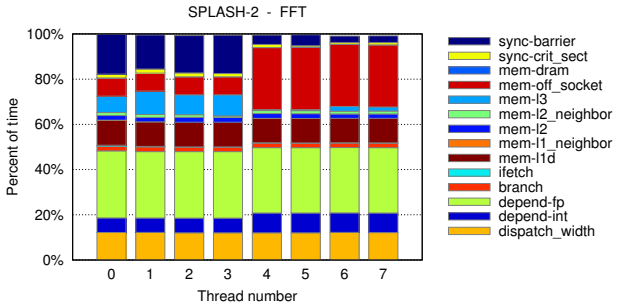



**Figure 3: Normalized cycle stacks for each core executing the `fft` benchmark when running the small input set on eight cores.**

threads 4—7, one can see that the first four threads spend around 20% of their time in the *sync-barrier* component, denoting they were stalled in a software barrier. This behavior may be surprising as all threads perform the same amount of work. Looking at the other components, it becomes clear that the difference in execution speed can be explained by non-uniform memory access (NUMA) behavior as all cores operate on data that is available in the first socket's level-3 cache to which the first four cores have faster access: cores 0—3 have some amount of *mem-l3* and only little *mem-off_socket* time denoting mostly local L3 accesses, while cores 4—7 have a significant *mem-off_socket* penalty.

## 4. FINE-GRAINED STATISTICS

To increase insight into the behavior of different parts of the code, we extended an internal version of Sniper to collect hardware events and timing effects at a per-instruction granularity. As in the whole-program case, comparable statistics can in some cases be obtained on existing systems using hardware performance counters, but these suffer from a number of drawbacks: many hardware counters have inaccuracies such as double-counting under certain conditions, skidding (meaning that events are not always associated with the correct instruction), sampling errors (instruction pointers are typically only sampled when a counter overflows), or a lack of insight into how hardware events contribute to execution time. In contrast, our instruction-level statistics are based on the concept of cycle stacks and can assign an execution time cost to each individual instruction. Event counts are added as well to aid in understanding what hardware component causes the time penalty.

Figure 4 provides an example for a snippet of AVX-512 code. The third instruction (`vaddpd`) goes out to DRAM (it performs 6.93 DRAM accesses per 1,000 executions) and hence has a high performance impact (it is responsible for 7.24% of total execution time). For HPC workloads it is often important to distinguish instructions that contribute to useful work (floating point operations) from those that manage data and control flow (loads and stores, address and loop index calculations, comparisons and branches, etc.). To this end the *ops* column plots the number of FP operations executed by each instruction, taking into account masked elements: the fourth instruction (`vfmadd231pd`) is a fused multiply-add which performs two operations on each of eight vector elements, but on average 21% of the elements are masked off leading to a useful operation count for this

```
eip   instruction                                 cycles   ops  mask  dram
--
4050b vpcmpd k2{k1}, zmm5, zmm4, 0x2               1.03%
40510 vmovupd zmm8, zmmword ptr [r11+r10*1]        3.51%
40517 vaddpd zmm7, zmmword ptr [r11+r14*1]         7.24%   8.0        6.93
4051e vfmadd231pd zmm8{k2}, zmm7, zmm6             1.84%  12.6   21%
40524 vmovupd zmmword ptr [r11+r10*1]{k2}, zmm8    1.03%
```

**Figure 4: Per-instruction statistics.**

```
Site #1:  Location:       main      fft.c:251  { trans = malloc(...); }
          Hit-where:      Loads           :      1433601  ( 14.3%)
                            L1            :      1384287  ( 96.6%)
                            L2            :        43560  (  3.0%)
                            dram          :         5754  (  0.4%)
          Total allocated: 2.0MB (2.0MB average)
Site #2:  Location:       main      fft.c:250  { x = malloc(...); }
          Hit-where:      Loads           :      1433601  ( 14.3%)
                            L1            :      1026277  ( 71.6%)
                            L2            :       326618  ( 22.8%)
                            dram          :        80706  (  5.6%)
          Total allocated: 2.0MB (2.0MB average)
```

**Figure 5: Per-array statistics for `fft`.**

instruction of 12.6 double-precision operations on average.

# 5. DATA-CENTRIC STATISTICS

Large data-parallel workloads are often limited by cache capacity and memory bandwidth. While gaining insight into which instructions cause cache misses can help in tracking down those data structures that are responsible for poor cache use, often it can be more insightful to be able to look at individual data structures directly. To this end, Sniper can collect cache statistics on a per data structure basis. In a simulator, implementing such functionality is relatively straightforward: application calls to *malloc* and other memory allocation library functions are intercepted, and the address ranges for each data type (determined by the call stack leading up to the *malloc* call) are recorded. Each memory access made by the core can then be tagged with its allocation site and cache statistics are accumulated per site.

An example can be seen in Figure 5. Two allocation sites are detected in the `fft` benchmark, corresponding to the *trans* and $x$ variables in the source code. Whereas *trans* has good cache behavior and can be serviced mostly out of the L1 cache, $x$ experiences many cache misses—and could be a candidate for moving into high-bandwidth memory, or algorithmic optimizations such as blocking.

# 6. REFERENCES


[1] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, Nov. 2011.

[2] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28:1–28:25, Aug. 2014.

[3] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. Smith. A top-down approach to architecting CPI component performance counters. *IEEE Micro*, 27(1):84–93, 2007.

[4] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, Feb. 2010.

[5] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 38–49, Nov. 2011.

[6] A. Yasin. A top-down method for performance analysis and counters architecture. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, Mar. 2014.

# Support Operators Technique for 3D Simulations of Dissipative Processes at High Performance Computers

Irina Gasilova
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
125047, Russia, Moscow
igasilova@gmail.com

Yuri Poveshenko
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
125047, Russia, Moscow
hecon@mail.ru

Aleksey Boldarev
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
125047, Russia, Moscow
boldar@imamod.ru

Gennady Bagdasarov
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
125047, Russia, Moscow
gennadiy3.14@gmail.com

Mikhail Iakobovski
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
125047, Russia, Moscow
lira@imamod.ru

## ABSTRACT

This paper describes an efficient method for 3D computational fluid dynamics (CFD) simulations on high performance computers. The problems related to the robustness and stability of numerical algorithms for distributed simulations are considered. On the example of common scalar divergence problem the rotationally-invariant finite difference scheme based on support operators technique is built and essential properties of the scheme are depicted. The results of the numerical simulations of the implemented in MARPLE3D package algorithm and the results of the scaling of the developed software on cluster are presented.

## Keywords

Support operators technique; finite differencing; unstructured meshes; high performance computing.

## 1. INTRODUCTION

Nowadays due to the rapid developments in the area of high performance computing there appeared the possibility to perform predictive numerical simulations in the area of CFD in the complex-shaped domains on the meshes of huge dimensions. These meshes are very finely discretized thus one can simulate highly multiscale physical processes which on its turn requires simulations using robust and stable numerical algorithms. These algorithms should possess a number of properties such as: conservativity, monotonicity, stability in a wide range of flow parameters, high resolution, positivness/nonpositiveness, self-adjointness of difference operators, etc [3], which lead to the solutions with guaranteed quality. The support operators method is remarkable as it allows building approximations to differential operators using general meshes (block-structured, unstructured, mortar etc.) while the resulting difference operators preserve not only basic properties mentioned above, but, additionally, they provide rotationally-invariant difference schemes [2]. It's important to pay special attention to the rotational invariance while working with systems describing deformations and dissipations in gas or liquid media, e.g. Navier-Stokes equations.

## 2. SUPPORT OPERATORS TECHNIQUE

In the area $O$ with the bounadry $\partial O$ let's consider a common scalar-divergence boundary problem with, for example, Dirichlet boundary condition:

$$\begin{cases} \mathrm{div}\mathbf{X}_u = f(\mathbf{r}) \\ \mathbf{X}_u = K\nabla u \end{cases}, u|_{\partial O} = u_*(x) \qquad (1)$$

Here $u$ – scalar (temperature, pressure and so on), $\mathbf{X}$ – arbitrary vector, $\mathbf{X}_u$ – flows, induced by the gradient of function $u$ in the media with the conduction properties, defined by tensor $K$. Traditional approach to the approximation of differential operators consists in the independent approximation of differential operators using the Gauss Theorem:

$$\int_O \mathrm{div}\mathbf{X}\, dV = \int_{\partial O} \mathbf{X}\, d\mathbf{S}, \quad \int_O \mathrm{grad}u\, dV = \int_{\partial O} u\, d\mathbf{S}$$

Support operators method – is a technique with the consistent approximation of differential operators. The system of equations (1) is considered along with the following identical integral equation:

$$\int_O (\mathbf{X}, \nabla u)\ dv + \int_O u \cdot \mathrm{div}\mathbf{X}\, dv = \int_{\partial O} u\,(\mathbf{X}, d\mathbf{s}) \qquad (2)$$

One operator (**div** or **grad**) is approximated directly; the other one is approximated in the way, that it satisfies the difference analogue of the integral equation (2).

## 3. DIFFERENCE SCHEME

To construct a difference scheme it is needed to introduce a difference mesh in the computational domain and to define mesh functions which will approximate functions of the continuous argument. Depending on the physical origin of the values and on the way of their definition on the mesh sometimes it is more convenient to approximate one or another differential operator directly. In the example given in this paper the support operators scheme was built using **grad** as a support operator.

## 3.1 Difference meshes and operators of support operators technique

In the computational domain $O$ let's introduce the difference mesh of general type, which consists of nodes ($\omega$), formed by nodes cells-polygons ($\Omega$), bases ($\varphi$), edges ($\lambda$), linked to edges faces ($\sigma(\lambda)$) – boundaries of the balance node domains $d(\omega)$. Closed around node $\omega$ surfaces $\sigma(\lambda(\omega))$ form node domains $d(\omega)$ (Fig. 1).

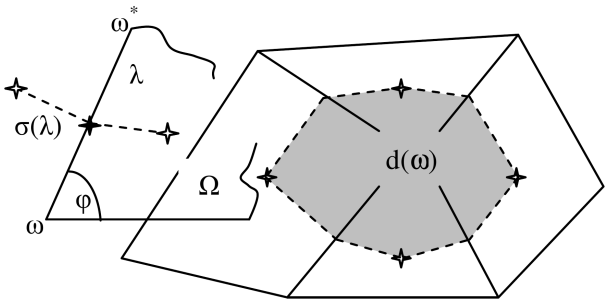



**Figure 1: Metrical meshes of the support operators technique.**

To the nodes of the grid $\omega$ we assign unknown mesh function $u$. In this case in the natural way operator **grad** is approximated.

Bases $\varphi$ are formed by the system of the initial (covariant) basis vectors $\mathbf{e}(\lambda)$, formed by edges. The centers of cells $\Omega$ and edges $\lambda$ are considered to be the arithmetic averages of radius vectors of nodes $\omega$ by which they are formed. Curve, which links these centers (of the two adjacent by edge cells or of the cell with the boundary edge $\partial\lambda$), represents by itself a surface:

$$\sigma(\lambda) = \sum_{\varphi(\lambda)} v_\varphi \mathbf{e}_\varphi(\lambda),$$

which is oriented in the same way as the basis vector $\mathbf{e}(\lambda)$. Here $\sum_{\varphi(\lambda)}$ means summing up by all bases $\varphi$, in the formation of which the edge $\lambda$ participated, $\mathbf{e}_\varphi(\lambda)$ – are basis vectors of the reciprocal (contravariant) bases with respect to the initial bases, formed by basis vectors $\mathbf{e}(\lambda)$.

Bases $\varphi(\lambda)$ are by pairs included into the cells $\Omega(\lambda)$ adjoint to the edge $\lambda$. Metrical calibration of the difference mesh consists in the choice of the volumes of basis (with the natural normalization condition $\sum_{\varphi(\Omega)} v_\varphi = V_\Omega$).

Calibration defines the structure of the closed adjacent mesh for the different classes of grids. These are triangle and quadrangle 2D meshes, tetrahedral, hexahedral, prismatic, etc. 3D meshes, and also their mortar sewed combinations with adaptation (introducing new nodes in the cells $\Omega$) with the preservation of self-adjointness and fixed sign properties of the corresponding "divergent-gradient" operations of vector analysis of the continuous boundary problems. All the following statements are of an overall character; the concrete choice of local basis volumes is illustrated on the example of triangle-quadrangle mesh.

Basis volume is given by the following formulas:

$$v_\varphi = \frac{1}{6}|\mathbf{e}(\lambda_1) \times \mathbf{e}(\lambda_2)|$$

for the triangular cell $\Omega$, which contains basis $\varphi$ and

$$v_\varphi = \frac{1}{4}|\mathbf{e}(\lambda_1) \times \mathbf{e}(\lambda_2)|$$

for the quadrangular cell, considering $\lambda_1(\varphi)$ and $\lambda_2(\varphi)$ are edges, which form basis $\varphi$.

## 3.2 Approximations of the differential operators

On the edges of the mesh let's choose the positive direction (Fig. 2):

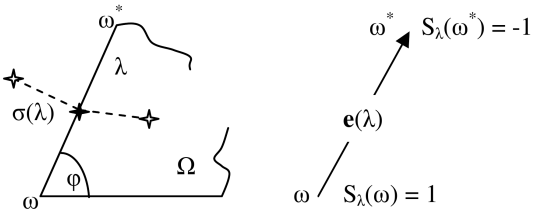



**Figure 2: Construction of bases.**

Divergence of the gradient field DIV : $(\varphi) \to (\omega)$ is defined by the approximation of the Gauss Theorem on $d(\omega)$:

$$\begin{cases} \mathrm{DIV}X = \sum\limits_{\lambda(\omega)} s_\lambda(\omega)\boldsymbol{\tau}_X(\lambda) \\ \boldsymbol{\tau}_X(\lambda) = \sum\limits_{\varphi(\lambda)} v_\varphi\left(\mathbf{e}_\varphi(\lambda), X_\varphi\right) \end{cases}$$

Where $\sum_{\lambda(\omega)}$ means summing up by all edges $\lambda$, that have common node $\omega$.

Mesh vector field $\mathbf{X}$ is defined by its representations in the bases $X_\varphi$. Let us denote by $()_\nabla$ approximation of the correspondent differential operators; in this case, taking (1) into account, we have:

$$\begin{aligned} &\left(\int (\mathbf{X}\,\nabla u)\,dv\right)_\nabla = \\ &= -\left(\int_O u\,\mathrm{DIV}\mathbf{X}\,dv - \int_{\partial O} u\,(\mathbf{X},\,d\mathbf{s})\right)_\nabla = \\ &= -\sum_\omega (u_\omega, \mathrm{DIV}\mathbf{X}) = \sum_\varphi (\mathbf{X}_\varphi,\ \mathrm{GRAD}u) \end{aligned}$$

Gradient vector field GRAD : $(\omega) \to (\varphi)$ is given by its components in the bases:

$$\begin{cases} \mathrm{GRAD}u = \sum\limits_{\lambda(\varphi)} \Delta_\lambda u \mathbf{e}'_\varphi(\lambda), \\ \Delta_\lambda u = -\sum\limits_{\omega(\lambda)} s_\lambda(\omega) u_\omega = u^*_\omega - u_\omega \end{cases}$$

Assuming in the bases $\varphi$ under $\mathbf{X}_\varphi$ vector field $\mathbf{X}_{u\varphi} = K_\varphi \mathrm{GRAD}_u$, we obtain self-adjoint non- negative operator $-\mathrm{DIV}\mathbf{X}_u : (\omega) \to (\omega)$ or $-\mathrm{DIV}K\mathrm{GRAD} : (\omega) \to (\omega)$. Here

the flow vector field $\mathbf{X}_u$ is given by its components in the bases $\mathbf{X}_{u\varphi}$. It is defined by the gradient properties of the scalar mesh function $u$, defined in the nodes $\omega$, and mesh tensor conduction field $K$, given by its representations in the bases $K_\varphi$. This operator will be strictly positive, if at least in one boundary node of the closed difference mesh the Dirichlet boundary value problem is defined, i.e. in this boundary node scalar mesh function goes to zero.

# 4. SOFTWARE IMPLEMENTATION

The described above algorithm was implemented inside the implicit heat diffusion solver within the MARPLE3D (Magnetically Accelerated Radiative Plasma Explorer) package [1]. MARPLE3D is a team work oriented for the solution and numerical simulations of CFD multiscale physical problems in domains with complex shapes at systems performing distributed computations. The code is implemented in C++ language. Parallel implementation is supported by MPI and CUDA.

The sizes of the meshes rule out their handling by a serial code. Thus distributed algorithms are used in all stages of the problem solution:

- distributed mesh generation
- partitioning and repartitioning of meshes (ParMetis)
- parallel solution of the problem
- parallel analysis of the results (ParaView)

The fact that the mesh is partitioned and distributed should be taken into account on the algorithmic level. Computations on the distributed mesh are supported by the original data structures, so called "fictive blocks", margins, which consist of several layers of cells and through which data exchange is performed. Implicit schemes lead to the distributed systems of linear equations (some equations involve the unknown values from the neighboring subdomains). For the solution of these systems, the package Aztec is used.

# 5. NUMERICAL RESULTS

The developed solver was tested on the propagation heat wave problem [4].

Let's consider a non linear heat diffusion equation in the following form:

$$\frac{\partial T}{\partial t} = \frac{\partial}{\partial s}\left(\kappa_0 \cdot T^\alpha \cdot \frac{\partial T}{\partial s}\right),$$

where $T$ – unknown temperature, $\kappa_0$, $\alpha$ – free coefficients, $s \equiv x\,|\,y\,|\,z$. This equation with the following initial and boundary conditions:

$$T(s,\ 0) = \begin{cases} \left[\dfrac{\alpha D}{\kappa_0}\,(s_0 - s)\right]^{\frac{1}{\alpha}}, & s \leqslant s_0, \\ 0, & s > s_0, \end{cases}$$

$$T(0,\ t) = \left[\frac{\alpha D}{\kappa_0}\,(Dt + s_0)\right]^{\frac{1}{\alpha}}, \qquad t > 0,$$

has an analytical solution in the form of propagation with the constant velocity wave:

$$T(s,\ t) = \begin{cases} \left[\dfrac{\alpha D}{\kappa_0}\,(Dt + s_0 - s)\right]^{\frac{1}{\alpha}}, & s \leqslant s_0 + Dt, \\ 0, & s > s_0 + Dt, \end{cases}$$

where $D$ – unknown temperature, $s_0$ – free parameter.

Simulations were held on 3D, 2D and 1D structured and unstructured meshes with the typical size of $s = 3$. Test parameters: $\alpha = 2.0$, $\kappa = 0.5$, $s_0 = 0.5$, $D = 5.0$; simulation time: $t$ from 0 to 0.4 with constant time step $\tau = 2 \cdot 10^{-4}$.

On the graphic represented on the Fig. 3 you can see results of one of such simulations: a one dimensional temperature front propagating along the axis $s$, both analytical and numerical solutions. From the graphic you can see that numerical solution is in a very good agreement with analytical solution, there is a tiny divergence on the front of the wave, that is a very peculiar area of the simulation. Despite how finely discretized mesh is taken in this area inhomogeneity will always persist cause the reason of this inhomogeneity is inside the physical task itself. On the front of the wave gradient of temperature goes to infinity and heat conduction coefficient $\kappa$ equals to zero. Thus we have a mathematical indeterminicity: infinity multiplied by zero. Solutions in such points are close to discontinuous. If solutions are obtained by non-monotonic although stable schemes, they "fall apart" in such points and give oscillations. Support operators method allows us to build robust numerical scheme that overcomes such problems and gives a stable solution close to analytical one.

# 6. SOFTWARE SCALING

On the test task described above there was also performed weak scaling of the developed software. The scaling was done on the Edison cluster of the NERSC (The North American Electric Reliability Corporation) supercomputer center. Simulations were performed on the amount of cores from 60 to more than 3000 cores. Mesh consistes of 8 million hexahedral cells. The results of the scaling are given on the graphic, represented on the Fig. 4. From the graphic you can see that efficient parallel computing is reached when the amount of cells per core varies from 30000 to 50000 cells. This was the case of ideal run of one heat diffusion solver. In case of multiple solvers run these estimation may shift.

# 7. CONCLUSIONS

The paper deals with the support operators method of constructing difference approximations using 3D unstructured grids. Differencing via support operator method guarantees preserving of properties of original differential operators, including rotational invariance. The method allows building robust numerical procedures suitable for multiscale simulations requiring very finely discretized computational domains (billions of grid cells) – and it is just a case that requires the use of high performance computing. The support operator technique is developed for meshes formed with hexahedral, tetrahedral, prismatic cells and their various combinations. The appropriate numerical algorithms are incorporated into the scientific CFD code MARPLE3D. The code versatility enables its applications to diverse CFD problems

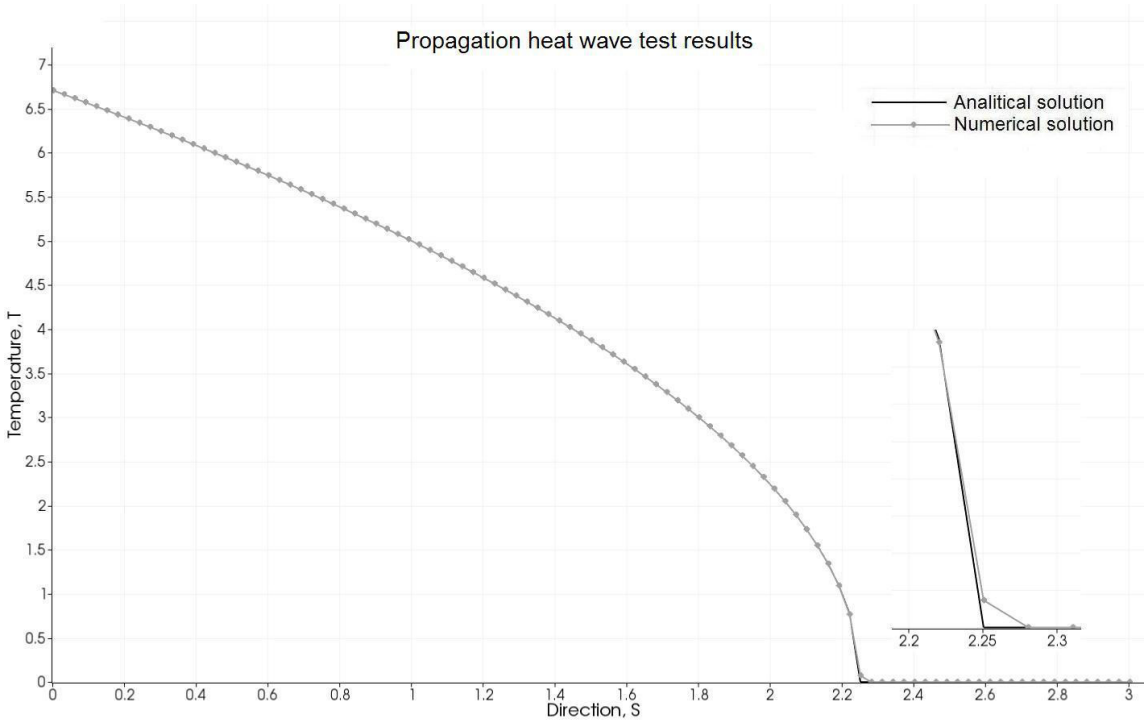



**Figure 3: Propagation heat wave front at the moment $t = 0.4$.**

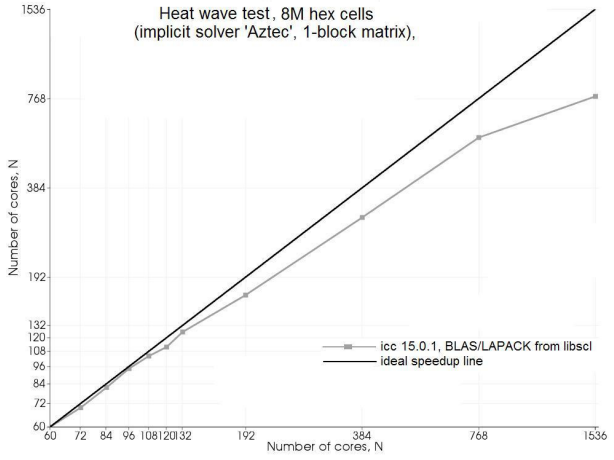



**Figure 4: Scaling results of developed solver.**

in regions with complex shapes. As it follows from numerical experiments, MARPLE3D ensures quite good scalability when performing the calculations using a large number of processors. The data models and features of parallel implementation are discussed. Examples of physical problems solved via HPC are presented.

## 8. ACKNOWLEDGMENTS

The work was supported by the RFFI grant No. 14-01-31154.

## 9. REFERENCES

[1] V. A. Gasilov and et Al. *The package of applied programs MARPLE3D for the modelling at high performance computers pulsed magnetically accelerated plasma*. Preprint KIAM RAS No. 20, 125047, Russia, Moscow, 2011.

[2] A. V. Koldoba, Y. A. Poveshchenko, I. V. Gasilova, and E. Y. Dorofeeva. Numerical schemes of the support operators method for elasticity theory equations. *Mathematical Modeling*, 24(12):86–96, November 2012.

[3] A. A. Samarskii, A. V. Koldoba, Y. A. Poveshchenko, V. F. Tishkin, and A. P. Favorskii. *Difference Schemes on Irregular Meshes*. JSC 'Kriteriy', 220027, Minsk, pr. F.Scoriny, 1999.

[4] A. A. Samarskii and I. M. Sobol. The examples of the numerical computing of temperature waves. *Journal of Computational Mathematics and Mathematical Physics*, 3(4):702–718, June 1963.

# Efficiently Scheduling Task Dataflow Parallelism: A Comparison Between Swan and QUARK

Hans Vandierendonck
Queen's University Belfast
United Kingdom
h.vandierendonck@qub.ac.uk

## ABSTRACT

Increased system variability and irregularity of parallelism in applications put increasing demands on the efficiency of dynamic task schedulers. This paper presents a new design for a work-stealing scheduler supporting both Cilk-style recursively parallel code and parallelism deduced from dataflow dependences. Initial evaluation on a set of linear algebra kernels demonstrates that our scheduler outperforms PLASMA's QUARK scheduler by up to 12% on a 16-thread Intel Xeon and by up to 50% on a 32-thread AMD Bulldozer.

## 1. INTRODUCTION

The many-core roadmap for processors dictates that the number of cores on a processor chip increases at an exponential rate. Moreover, cores tend to operate at different speeds due to process variability and thermal constraints. As such, parallel task schedulers in the exascale era must make dynamic (runtime) scheduling decisions [1].

The task dataflow notation has been studied widely as a viable approach to facilitate the specification of highly parallel codes [2, 3, 4]. Task dataflow dependences specify an ordering of tasks (they leverage a task graph), which by its nature exposes a higher degree of parallelism than barrier-based models where threads wait periodically for all running tasks to complete. Dynamic schedulers are, however, prone to result in less performance than static schedulers due to runtime task scheduling overhead.

This work investigates a new design for a task dataflow dynamic scheduler. The key design goal is to minimize runtime overhead without affecting the task dataflow programming interface. The scheduler supports programs mixing recursive divide-and-conquer parallelism and task dataflow parallelism. This hybrid design simplifies, for instance, the exploitation of parallelism across multiple kernels called in succession. The scheduler combines the efficiency of Cilk's work stealing scheduler [5] for recursively parallel programs with the efficiency of the steal-half queue [6] for programs generating large numbers of simultaneously ready tasks.

We evaluate our design experimentally and compare against PLASMA's QUARK [2] scheduler on a set of level-3 BLAS kernels with irregular parallelism. In comparison to QUARK, our scheduler reduces end-to-end execution time of several linear algebra kernels by up to 12% on an Intel Xeon (Sandy Bridge) and by up to 50% on an AMD Bulldozer.

## 2. RELATED WORK

Several approaches to task dataflow scheduling have been experimented with. Several authors have implemented Tomasulo's algorithm in software [7, 8]. QUARK [2] is a work-stealing scheduler tuned to linear algebra problems. QUARK attempts to optimize data locality. QUARK records and enforces dependences using the starting address of a task argument. As such it is dependent on a fixed argument size. SMPSs [9] uses a comparable work stealing design with many design decisions that are similar from a high level point of view. An SMPSs extension for strided and sparse access patterns incurs a high performance penalty by scanning across all outstanding tasks when scheduling a task [10].

PARSeC/DAGuE is a distributed task scheduler [3]. It pre-computes and distributes the task graph in order to obtain low overhead scheduling.

StarPU [4] schedules tasks according to predicted task latency. Task latency is predicted using performance models selected by the programmer.

Swan [11] is a task dataflow scheduler built as an extension to Cilk [12]. As such, it fully supports nested parallelism. Contrary to other approaches, Swan attempts to keep the task graph small during execution and only expands it when necessary to discover parallelism. In the initial design, task graphs were retained centrally with the parent procedure. In this paper, a distributed storage of the task graph among the worker threads is proposed.

OpenSTREAM [13] is focused on stream parallelism. As such, the scheduler is organized the matching of producers with consumers.

XKaapi [14] employs a number of pattern-specific scheduling heuristics in order to reduce scheduling overhead. Others have similarly proposed heuristics to reduce the overhead of specific parallel patterns [15].

## 3. SWAN

Swan is a task based programming model that extends the Cilk language with dataflow annotations and dataflow-driven execution [11, 16]. In this language, the *spawn* keyword is inserted before a function call to indicate that the call may proceed in parallel with the continuation of the calling procedure. The *sync* keyword indicates that the execution of the procedure should be delayed until all spawned procedures have finished execution.

Figure 1 shows an example Swan program that implements matrix multiply. The various components of the programming model are explained below.

### 3.1 Objects

Objects are special program variables of type *versioned*

```
1 typedef float (*block_t )[16]; // 16x16 tile
2 typedef versioned<float[16][16]> vers_block_t ;
3 typedef indep<float[16][16]> in_block_t ;
4 typedef inoutdep<float[16][16]> inout_block_t ;
5
6 void mul_add(in_block_t A, in_block_t B, inout_block_t C) {
7     block_t a = (block_t)A; // Recover pointers
8     block_t b = (block_t)B; // to the raw data
9     block_t c = (block_t)C; // from the versioned objects
10    // ... serial implementation on a 16x16 tile ...
11 }
12
13 void matmul(vers_block_t * A, vers_block_t * B,
14                 vers_block_t * C, unsigned n) {
15     for( unsigned i=0; i < n; ++i ) {
16         for( unsigned j=0; j < n; ++j ) {
17             for( unsigned k=0; k < n; ++k ) {
18                 spawn mul_add( (in_block_t)A[i*n+j],
19                                ( in_block_t )B[j*n+k],
20                                ( inout_block_t )C[i*n+k] );
21             }
22         }
23     }
24     sync;
25 }
```

Figure 1: Square matrix multiplication expressed in a language supporting runtime tracking and enforcement of task dependences.

that express inter-task dependences. Objects may be passed as arguments to tasks using annotated task arguments that express the side-effects of the task on that argument. Annotated task arguments can only accept objects as arguments, not constants or generic variable types.

An object may be renamed, which means that its address is changed by the runtime system. The runtime system performs renaming to increase parallelism. The runtime system also makes sure that latent pointers to renamed objects are properly translated to the appropriate version of the object before accessing memory.

The runtime systems associates metadata to each object, e.g. to perform dependence analysis and to recover its most recent version after renaming. The runtime system stores this metadata side-by-side with the object in order to speedup the retrieval of metadata.

We further stipulate that all arguments passed to a task are unique objects. This is to avoid circular dependences of a task on itself.

## 3.2 Memory Usage Annotations

The arguments of spawned procedures may be annotated with *memory usage information*, i.e. how the argument is accessed by the task. The memory usage may be *input*, *output*, *input/output*, *commutative in/out* or *reduction*. An input argument is read but not written to. An output argument is written and may be read, but it is always written before it is read. Consequently, its value upon initiation of the task is irrelevant. An input/output argument (or in/out for short) may be read and written and it may be read before it is written.

A commutative in/out annotation extends the in/out semantics with the notion that consecutively spawned tasks may be executed in any order, but may not execute concurrently. Reordering is subject to the absence of other inter-task dependences. The runtime system guarantees that commutative tasks do not execute concurrently by associating a lock with each object to enforce mutual exclusion.

Our model also supports reductions, details of which have been previously published [17]. We will not discuss the support for reductions here as they pose no specific constraints for the purposes of this work.

Hyperqueues extend the programming model with queue usage annotations such as push and pop [18]. These annotations are not fundamentally different than the annotations listed above as they allow to use the same dependence tracking and scheduling techniques as discussed above.

## 3.3 Execution Model

The Swan execution model is an extension of the Cilk execution model. Swan behaves identical to Cilk in the absence of task arguments with memory usage annotations. The execution model differs when dataflow dependences between tasks are specified. These dataflow dependences are restricted within a procedure body. In other words, a task can depend only on a sibling, i.e., another task spawned by the parent of the first task. The dataflow dependences are furthermore determined by the order of the spawn statements in the procedure body. It is assumed that a sequential thread of execution steps through the procedure and, in the process, encounters a sequence of spawn statements. This sequence, together with the memory usage annotations, defines dependences between the spawned tasks. A dependence states that a pair of tasks *must* execute in the order that they were spawned. These tasks are added one by one to the task graph, where nodes represent dynamic task instances and edges represent task dependences.

The task graph is a directed acyclic graph (DAG) because tasks can only depend on tasks that appear before them in (serial) program order. At any moment, the roots of the DAG are tasks that are either *executing* or that are *ready to execute*. We call the list of root tasks that are ready to execute the *ready list*. It provides direct access to the ready tasks when one is needed.

Note that a Swan program may have up to one dataflow task graph per procedure body. Execution of the program may proceed by executing tasks from multiple task graphs concurrently. Swan uses random work stealing to balance execution between task graphs dynamically, depending on the degree of parallelism in each task graph.

# 4. SCHEDULING

The Swan scheduler is responsible for deciding what task is executed next by each worker thread. Like Cilk, the Swan scheduler is symmetric, i.e., all workers execute the same scheduling algorithm.

On encountering a spawn statement, the scheduler first checks that all dependences have been satisfied. If so, the scheduler proceeds as in the Cilk case, pursuing a work-first execution. A stack frame is pushed on the worker's deque (*double-ended queue*), which is managed like a call stack.

If dependences are not satisfied, then the task is not started for execution at this point. Instead, it creates a pending frame, a new type of frame in the Swan scheduler that represents an uninitiated task. The pending frame is inserted into the task graph that corresponds to the stack frame that

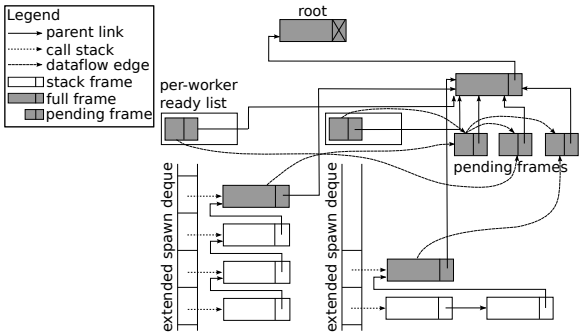



Figure 2: The Swan runtime data structures for two worker threads.

is currently at the tail of the spawn deque.[1] The scheduler then resumes execution of the stack frame at the tail of the spawn deque. Figure 2 shows the position of the ready list and the task graph in the scheduler.

The Swan scheduler applies three operations to task graphs: issue, release, and get-task. The *issue* operation registers that a task is accessing its operands (which are objects in Swan) and records the memory usage annotation. If other tasks are registered on the same objects, then the dependences are deduced and recorded, which amounts to linking the task in the task graph. If no dependences are present, then the task is executed immediately, or inserted in the ready list. The *release* operation unregisters a task, i.e., dependent tasks are notified that dependences are released and, if applicable, the dependent tasks are moved to the ready list. Finally, the *get-task* operation retrieves a runnable task from the ready list.

When a spawn deque becomes empty after completing a procedure, the scheduler first attempts to execute a ready task on the worker's ready list. This choice ensures that task graphs are kept small, as completing a task is likely to wakeup other pending tasks.

If the scheduler cannot identify ready tasks on the local ready list, it attempts a provably-good steal of the parent task. If the provably-good steal is unsuccesful, then random work stealing is attempted. Random work stealing is again designed to pick up ready pending tasks. First, a random worker is selected called the *victim*. If the victim has a non-empty ready list, then half of the tasks on the ready list are transfered to the stealing worker. This strategy minimizes work stealing [6]. One of the stolen tasks is moved to the worker's spawn deque and executed. If the vicitim's ready list is, however, empty then the scheduler tries to steal the top frame on the victim's spawn deque as in the Cilk scheduler. If all of this fails, another random victim is selected and the algorithm is repeated.

## 5. PLASMA INTERFACE

For the purpose of the evaluation in this paper, we tightly integrated Swan in the PLASMA system such that it can be used as a replacement of the QUARK dataflow scheduler [2]. We have implemented a number of dynamically scheduled level-3 BLAS kernels with irregular parallelism.

[1]As DAGs are restrained to a single procedure body, spawned procedures may be not ready for execution only if the parent procedure is executed in parallel, which requires it to be at the tail of the deque.

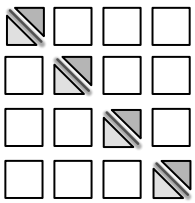


Figure 3: PLASMA matrix parts

Our implementation respects the PLASMA API. This integration enables a one-to-one comparison between Swan and QUARK as it is not affected by various implementation decisions such as data layout, library interfaces, etc.

PLASMA parallelizes level-3 BLAS kernels by decomposing them as blocked matrix operations. Hereto, matrices are decomposed in blocks, assuming an internally tuned block size. QUARK uses the starting addresses of matrix blocks to track dependences: a matrix block is shared between two tasks only if they both take the starting address of that block as an argument. Some tasks only access part of a matrix block and QUARK takes this into account. The commonly occurring parts are the lower triangular part, the diagonal and the upper triangular part of a matrix block (Figure 3). While typically only matrix blocks on the diagonal of the matrix are split in parts, it is necessary to allow per-part dependences for all matrix blocks as PLASMA supports the creation of sub-matrices which describe an arbitrary subset of the matrix. As such, the diagonal blocks on a sub-matrix may be non-diagonal blocks in the main matrix.

To integrate with PLASMA, we define the equivalent of a PLASMA descriptor (which describes the matrix layout) and PLASMA-specific dependence types that record dependences on matrix blocks (Figure 4). The Swan descriptor of a PLASMA matrix consists of the PLASMA descriptor and a 2D-array of dependence tokens. The dependence tokens consists of metadata to record actions of spawned tasks but contrary to normal variables, they do not contain data. Instead, the data is taken from the matrix. The tokens record up to three dependences to account for individual usage of the lower and upper triangular parts and the diagonal of a matrix block. The class `subobj_metadata` records such metadata and applies up to 3 times the standard Swan dependence tracking algorithm, depending on what parts of a matrix block are used by a task.

Input, output and input/output dependences can be obtained from the Swan descriptor using the `get_indep()`, `get_outdep()` and `get_inoutdep()` methods (only indep's are shown in Figure 4, other dependence types are defined similarly). These dependences hold a pointer to the corresponding token and the starting address of the matrix block's data.

Given the definition of the PLASMA matrix descriptor and dependence types in Swan, linear algebra kernels can be expressed in Swan and scheduled using task dataflow parallelism. Figure 5 shows how the `dormqr` function is declared and how it is used. `dormqr` accesses the lower-triangular part of a block A (indicated by the additional template argument `sub::lo`) and accesses blocks T and C in full. After instantiating the matrix descriptors in PLASMA and Swan formats, the appropriate matrix blocks, annotated with usage information, are obtained using the `get_`*X*`dep()` methods.

```
1  struct sub {
2     enum parts_id_t {
3        lo = 1, diag = 2, up = 4,
4        lodiag = lo | diag,
5        updiag = diag | up,
6        all = lo | diag | up };
7  };
8  template<typename T, sub::parts_id_t _parts=sub::all>
9  class plasma_indep {
10     static const sub:: parts_id_t parts = _parts;
11     subobj_metadata<sub> * meta; // dependence tracking
12     T * addr;
13
14     static plasma_indep<T,_Part>
15     create( subobj_metadata<sub> * meta, T * addr ) { ... }
16
17 public:
18     const T * get_addr() const { return addr; }
19 };
20
21 template<typename T>
22 class swan_desc {
23     PLASMA_desc desc;
24     subobj_metadata<sub> * tokens;
25
26 public:
27     swan_desc( const PLASMA_desc & _desc ) {
28        // Copy PLASMA_desc and setup 2D array of tokens
29     }
30     T * get_addr( int m, int n ) const {
31        return plasma_getaddr( desc, m, n );
32     }
33     template<sub::parts_id_t part = sub:: parts_id_t :: all>
34     plasma_indep<T,part> get_indep( int m, int n ) const {
35        return plasma_indep<T,part>::create(
36           get_token( m, n ), get_addr( m, n ) );
37     }
38 private:
39     subobj_metadata<sub> * get_token( int m, int n ) {
40        return tokens [...];
41     }
42 };
```

Figure 4: Swan interface to PLASMA descriptor

```
1  void
2  dormqr( ..., // dimensions, transforms
3          plasma_indep<double,sub::lo> A,
4          plasma_indep<double> T,
5          plasma_inoutdep<double> C );
6
7  PLASMA_desc A = ...;
8  PLASMA_desc T = ...;
9  swan_desc<double> As( A );
10 swan_desc<double> Ts( T );
11 spawn dormqr( ...,
12               As.get_indep<sub::lo>(k, k),
13               Ts.get_indep(k, k), // defaults to sub::all
14               As.get_inoutdep(k, n), ... );
```

Figure 5: Usage of Swan/PLASMA descriptor

# 6. EVALUATION

We compare the performance of Swan and QUARK to schedule three linear algebra kernels with irregular parallelism: Cholesky factorization, QR factorization and LU factorization with partial pivoting. Our comparison is performed on two machines: a dual-socket 2 GHz Intel Xeon Sandy Bridge E5-2650 (2x8 threads) and a dual-socket 2.1GHz AMD Opteron 6272 (2x16 threads). In the AMD processor, every pair of cores shares a floating-point unit.

We use PLASMA 2.6.0, gcc 4.9.2 and CentOS 6.5 on both machines. On Intel we use Intel MKL 11.1.2 for the basic single-threaded BLAS kernels. On AMD we use ACML 5.3.1.

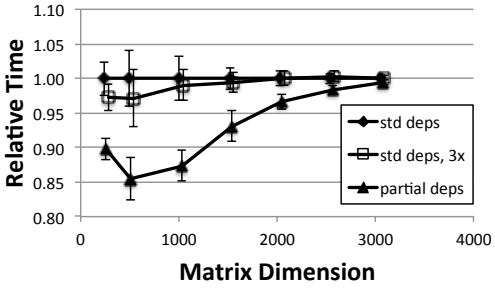



Figure 6: Overhead of tripple dependence tracking.

## 6.1 Dependence Tracking on Object Parts

Firstly, we validate the design of tracking dependences on object parts (one dependence chain per part of a matrix block). This analyses is performed exclusively using Swan on the Intel machine. Figure 6 compares three scenarios. The first scenario ("std deps") measures the performance of QR factorization while assuming that tasks access full matrix blocks. Only one dependence is recorded per matrix block. In the second scenario ("std deps, 3x"), we make the same assumption but we record 3 dependences per block, one for each part. The parallelism in the first two versions is identical. In the third version ("partial deps"), again three dependences are recorded per usage of a full matrix block, but the QR algorithm correctly records dependences on parts of matrix blocks. As such, the parallelism is higher in the third scenario, although dependences are recorded three times in the majority of cases. We furthermore vary the matrix dimension (the block size is kept constant to PLASMA's default of 128).

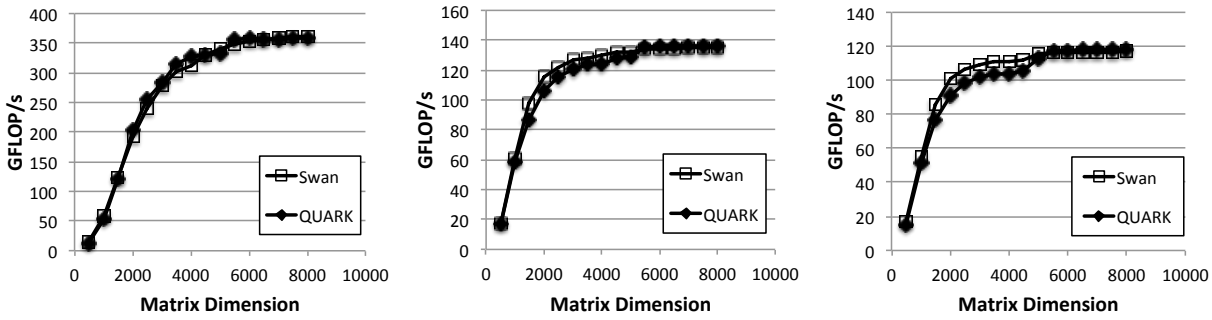
Figure 6 demonstrates that tracking dependences three times per task argument incurs little overhead. In fact, it results in a minor speedup. However the standard deviation, depicted using error bars, shows that this speedup is not statistically significant. Annotating partial usage of matrix blocks results in reduced execution time. We note this improvement for matrices with dimensions 500 to 2000, which in practice means that the degree of parallelism must be low in comparison to the block size and number of threads.

We conclude that our implementation enables increased parallelism without significant performance overhead in cases where only full matrix blocks are accessed.

## 6.2 Evaluation on Sandy Bridge

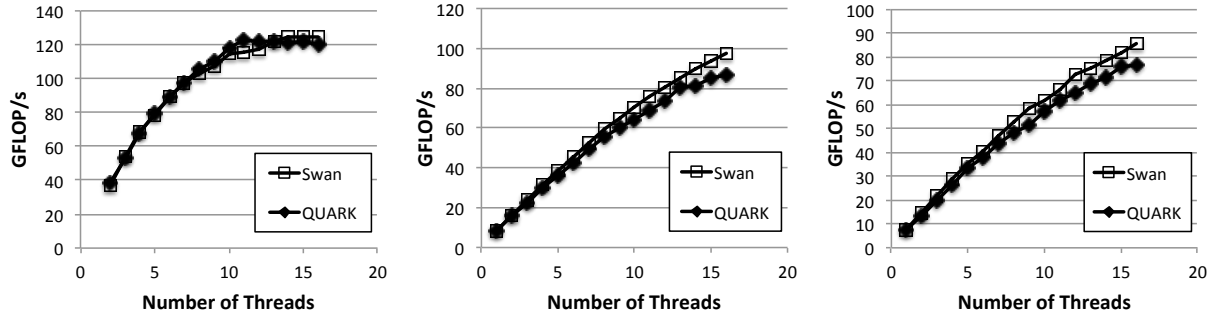
Figure 7 shows the performance of cholesky, QR and LU with partial pivoting when executing on 16 threads. Cholesky decomposition performs nearly equally with Swan and QUARK. Performance of QR, however, is between 3.8% and 12.5% faster with Swan than with QUARK for matrix dimensions up to 4000. LU is between 5.8% and 11.9% faster with Swan for matrix dimensions up to 4500.

Figure 8 shows that the performance differences grow with




(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 7: Performance comparison of Swan and QUARK for varying matrix dimension on Sandy Bridge using 16 threads.




(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 8: Performance comparison of Swan and QUARK for varying thread count and a $1500x1500$ matrix on Sandy Bridge.

an increasing thread count for a $1500x1500$ matrix. At the highest thread count, Swan executes Cholesky, QR and LU faster by 3.8%, 12.5% and 11.9%, respectively.

## 6.3 Evaluation on Bulldozer

Figure 9 shows the performance of the kernels when using the full Bulldozer machine (32 threads). We do not find noteworthy performance differences in this comparison for Cholesky. On LU, Swan outperforms QUARK by 5.6%–10.6% for matrix dimensions between 3000 and 5000. On QR, Swan is significantly faster, over 12% and up to 22.8% for matrix dimensions larger than 2500.

Investigating the variation with thread count (Figure 10), we see a marked difference between Swan and QUARK on the three kernels. Note that we applied thread pinning for both runtimes such that no floating-point units are shared between threads when 16 threads or less are used. Swan is able to quickly utilize most of the available performance, while performance increases more slowly after 16 threads. In contrast, QUARK needs to utilize all threads to reach close to peak performance. On 16 threads, Swan outperforms QUARK by 44–53%.

# 7. CONCLUSION

Swan is a versatile scheduler that has been proven in distinct scenarios, including pipeline parallelism, recursive parallelism and in this paper for linear algebra computations. The scheduler is optimized to schedule both recursive (divide-and-conquer) parallelism and task dataflow parallelism. In a one-to-one comparison with PLASMA, we demonstrate performance benefits up to 10% on a range of matrix dimensions on an Intel Sandy Bridge machine. Moreover, we demonstrate up to 22.8% performance improvement on a fully utilized AMD Bulldozer machine.

# 8. ACKNOWLEDGMENT

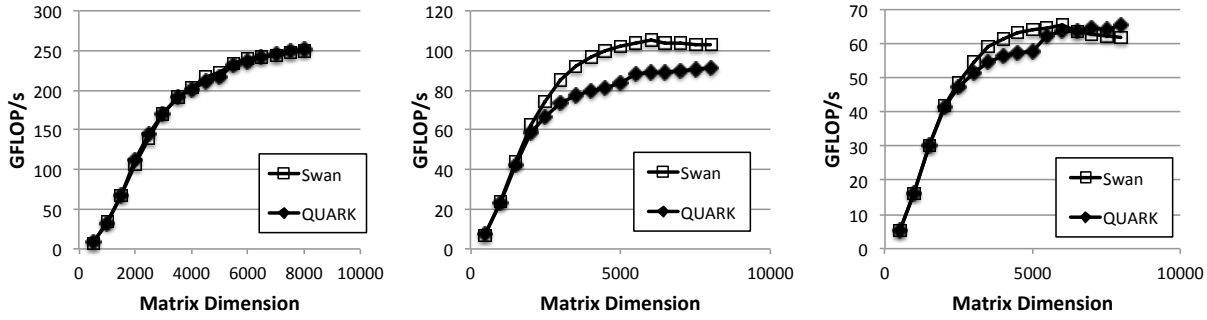
This work is partly supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the NovoSoft project (Marie Curie Actions, grant agreement 327744), the ASAP project (grant agreement 619706) and by the EPSRC under project EP/L027402/1.

# 9. REFERENCES

[1] J. Dongarra, P. Beckman, and et al, "The international exascale software project roadmap," *International Journal of High Performance Computer Applications*, vol. 25, no. 1, pp. 3–60, 2011.

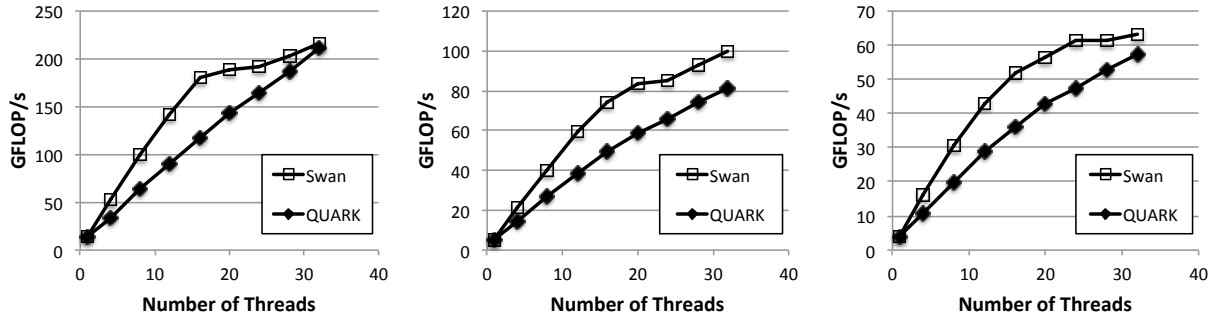
[2] J. Kurzak, P. Luszczek, and et al, *Multithreading in the PLASMA Library*. CRC Press, 2013, ch. 3, pp. 119–142.

[3] G. Bosilca, A. Bouteiller, and et al, "Dague: A generic distributed dag engine for high performance




(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 9: Performance comparison of Swan and QUARK for varying matrix dimension and maximum threads on Bulldozer.




(a) Cholesky (spotrf) (b) QR (dgeqrf) (c) LU (dgetrf_incpiv)

Figure 10: Performance comparison of Swan and QUARK for varying thread count and a $4500x4500$ matrix on Bulldozer.

computing," *Parallel Comput.*, vol. 38, no. 1-2, pp. 37–51, Jan. 2012.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2010.

[5] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *FOCS*. 1994, pp. 356–368.

[6] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," in *PODC*, 2002, pp. 280–289.

[7] E. Chan, F. G. Van Zee, and et al, "Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *PPoPP*, 2008, pp. 123–132.

[8] J. Kurzak and J. Dongarra, "Fully dynamic scheduler for numerical computing on multicore processors," University of Tennessee, Tech. Rep. UT-CS-09-643, Jun. 2009, LAPACK Working Note 220.

[9] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multicore architectures," in *CLUSTER*, Sep. 2008, pp. 142–151.

[10] ——, "Handling task dependencies under strided and aliased references," in *ICS*, 2010, pp. 263–274.

[11] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, Oct. 2011, pp. 1–11.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multi-threaded language," in *PLDI*, 1998, pp. 212–223.

[13] A. Pop and A. Cohen, "OpenSTREAM: Expressiveness and data-flow compilation of OpenMP streaming programs," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 53:1–53:25, Jan. 2013.

[14] T. Gautier, F. Lementec, V. Faucher, and B. Raffin, "X-kaapi: A multi paradigm runtime for multicore architectures," in *ICPP*, 2013, pp. 728–735.

[15] A. Tzannes, G. C. Caragea, R. Barua, and U. Vishkin, "Lazy binary-splitting: a run-time adaptive work-stealing scheduler," in *PPoPP*, 2010, pp. 179–190.

[16] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "Parallel programming of general-purpose programs using task-based programming models," in *Proc. of the Workshop on Hot Topics in Parallelism*, May 2011, p. 6.

[17] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos, "Analysis of dependence tracking algorithms for task dataflow execution," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 61:1–61:24, Dec. 2013.

[18] H. Vandierendonck, K. Chronaki, and D. S. Nikolopoulos, "Deterministic scale-free pipeline parallelism with hyperqueues," in *SC*, 2013, pp. 32:1–32:12.

# Portability and Performance of Nuclear Reactor Simulations on Many-Core Architectures

Ronald Rahaman
Argonne National Laboratory
Mathematics and Computer Science Division
Lemont, Illinois
rahaman@anl.gov

David Medina
Rice University
Department of Compuational and Applied Mathematics
Houston, Texas
dsm5@rice.edu

Amanda Lund
Argonne National Laboratory
Mathematics and Computer Science Division
Lemont, Illinois
alund@anl.gov

John Tramm
Massachusetts Institute of Technology
Department of Nuclear Science and Engineering
Cambridge, Massachusetts
jtramm@mit.edu

Tim Warburton
Rice University
Department of Compuational and Applied Mathematics
Houston, Texas
timwar@rice.edu

Andrew Siegel
Argonne National Laboratory
Mathematics and Computer Science Division
Lemont, Illinois
siegela@mcs.anl.gov

## ABSTRACT

High-fidelity simulation of a full scale nuclear reactor core is a computational challenge that has yet to be met but is predicted to be achievable on exascale-class supercomputers through established hardware-specific programming models (such as OpenMP and CUDA). Recently-developed, hardware-agnostic programming models offer opportunities to express multi-threaded parallelism in a portable fashion and allow a single, more-unified code base to run on many divergent high-performance computing architectures. Though the benefits of portability are clear, questions remain as to what practical performance tradeoffs apply to real world applications. In the present study, we port two existing proxy applications that represent key algorithms in nuclear reactor simulations to the hardware-agnostic language of OCCA. Performance and efficiency of the OCCA ports are compared to the native OpenMP versions on CPU and CUDA versions on GPU architectures. This study attempts to quantify tradeoffs between performance and portability of real world applications, specifically on exascale-class simulations for nuclear industry, using newer programming models.

## Keywords

nuclear reactor simulation, OCCA, OpenMP, CUDA, GPU

## 1. INTRODUCTION

In the foreseeable future, increased on-node parallelism will continue to be essential for improving peak FLOP throughput while constraining power usage [1, 10]. This trend is evinced by upcoming HPC systems that rely heavily on many-core accelerators and, in several cases, feature considerably fewer compute nodes than their predecessors. In nuclear reactor simulations, several methods of computational neutron transport are well-positioned to exploit these architectural trends. For Monte Carlo methods and the method of characteristics (MOC), communication costs have been greatly minimized to allow excellent strong scaling across nodes [5, 11]. Current research is increasingly targeting on-node optimizations for many-core CPUs with wide vector units and accelerators such as NVIDIA GPUs and Intel Xeon Phi Coprocessors [2, 8, 9, 13, 16]. With increased on-node performance, longstanding objectives in nuclear reactor simulation, such as high-fidelity simulations of full-scale 3D reactor models, are within reach [3, 6].

In response to the variety of many-core architectures, a variety of programming models have also been introduced, including CUDA, OpenACC, recent OpenMP 4.0 extensions, and research projects such as OCCA [7]. Though these models are solutions to the same problem (programming diverse architectures with a common set of abstractions), they may require different modifications to existing algorithms and applications. Thus for future application development, it is essential to know what modifications are needed to a accommodate a given programming mode and whether those modifications are performance portable. In this study, we examine portability and performance trade-offs for many-core implementations of XSBench and SimpleMOC, which are proxy applications for MC and MOC neutron transport, respectively [4, 14]. We evaluate ports with OpenMP on CPUs and CUDA and OpenACC on GPU. We evaluate hardware-agnostic OCCA ports on both kinds of devices.

## 2. BACKGROUND

### 2.1 Computing Environments

We report results for two generations of Intel Xeon CPUs: a dual-socket Sandy Bridge E5-2650 node (2.00 GHz, 8 cores per socket), and a dual-socket Haswell E5-2699 node (2.30 GHz, 18 cores per socket). Multithreaded results were run with the maximum available hyperthreads (32 threads for the Sandy Bridge node, 72 thread for the Haswell). OpenMP and OCCA ports of our applications were run on the CPUs. We also report results on the NVIDIA Tesla K40m GPU. CUDA, OpenACC, and OCCA ports were on the GPU.

### 2.2 OCCA

OCCA is an open-source library used to program current multi-core/many-core architectures. Different devices (such

as CPUs, GPUs, Intel's Xeon Phis, etc.) are abstracted using the same offload-model. Application development and programming for the devices is done through a C-based kernel language (OKL). Different devices can targeted by the same kernel through run-time parsing with the OCCA parser and compilation with the specified backend (Pthreads, OpenMP, CUDA, OpenCL, etc.). Herein, we compile OKL kernels with the OpenMP backend for CPU and the CUDA backend for GPU.

## 2.3 Computational Neutron Transport

### 2.3.1 Neutron cross sections

The methods described here make use of *neutron cross-sections*, which quantify the interactions of free neutrons with matter. *Microscopic cross-sections* ("micro XS") describe the interaction of an incident neutron with a single target nuclide (such as U-235, U-238, etc.). Micro XS values depend on the energy of the incident neutron, the identity of the target nuclide, and the interaction of interest (such as absorbtion, elastic scattering, fission, etc.). *Macroscopic cross-sections* ("macro XS") describe the interactions of a neutron as it travels through a homogeneous medium composed of many nuclides. For a given material, interaction, and neutron energy, a macro XS can be expressed as the denisty weighted average of the component micro cross-sections for all nuclides in the material.

In neutron transport simulations, cross-section data may be represented in a number of ways, including continuous-energy cross-sections or energy groups. Continuous-energy micro cross-sections are pointwise defined on a non-uniform energy grid for each isotope and interaction type. Continuous values can be interpolated between gridpoints. With energy groups, the micro XS values are much more coarsely discretized. When using continuous-energy cross-sections, the macro XS is typically computed at runtime, in order to keep memory requirements manageable. When using energy groups, macro XS values can be precomputed.

### 2.3.2 Neutron transport methods

Herein, we describe simulations using two different neutron transport methods, both of which have easily exploitable parallelism and are very promising for exascale. In *Monte Carlo* methods, many independent neutrons are simulated in order to obtain estimators for physical quantities of interest (such as neutron flux). In the *method of characteristics* (MOC), the spatial domain is decomposed into many independent 2D tracks, and neutron fluxes are deterministically attenuated across those tracks. Monte Carlo transport uses continuous-energy cross-sections, whereas MOC uses energy groups; hence, Monte Carlo methods can be more accurate. Furthermore, Monte Carlo methods are almost capable of simulating high-fidelity, full-scale 3D reactor models, whereas 3D MOC is still in early development. However, Monte Carlo simulations are generally memory bound, whereas MOC simulations have high computational intensity and are much easier to vectorize. In fact, 3D MOC is being specifically developed to leverage modern SIMD (single-instruction, multiple-data) architectures. To study these methods, we make use of two proxy applications, XSBench and SimpleMOC, which faithfully model the on-node computational workload of MC and MOC, respectively.

# 3. MONTE CARLO METHODS

## 3.1 Algorithm and Performance Issues

In Monte Carlo transport, each neutron is independently simulated until it is absorbed by a material or leaks from the boundaries of the domain (Algorithm 1, lines 2 - 9). To simulate a neutron's interactions, the macro XS must be calculated (lines 3 - 7) for each interaction type (lines 4 - 7) by looking up the material- and energy-dependent micro XS gridpoints (line 5), interpolating a continuous micro XS value (line 6), and adding the density-weighted micro XS to the macro XS (line 7). The macro XS is used to sample the distance to the next interaction and the result of the interaction (line 8); and finally, this information is used to update the neutron's position and energy (line 9). The proxy application, XSBench, executes the macro XS lookup kernel (lines 3 - 7) for a distribution of lookups.

A neutron's trajectory through phase space is mostly unpredictable, so lookups of the energy- and material-dependent micro XS are effectively stochastic and exhibit poor locality of reference. On CPU, XSBench suffers from a high last-level cache miss rate (up to 65%). When coupled with high latency to main memory, this results in a huge proportion (over 90%) of CPU cycles that are stalled on memory resources [15]. Multithreading with OpenMP can partially mask this latency, but at high thread counts, the available bandwidth becomes saturated and limits additional performance gains. On GPUs, high-bandwidth on-device memory offers the possibility to push performance bounds, provided that memory utilization is also high [13].

**Algorithm 1** Monte Carlo Method

```
1: for all neutrons do
2:     repeat
3:         for all nuclides in material do
4:             for all interaction types do
5:                 lookup bounding micro xs gridpoints
6:                 interpolate micro xs
7:                 accumulate micro xs into macro xs
8:         sample interaction
9:         update neutron position and energy
10:    until neutron is absorbed or leaks
```

## 3.2 Parallel Implementations

In parallel implementations, many independent neutrons are tracked simultaneously. In XSBench, this computational workload is abstracted by performing many simultaneous macro XS lookups. Finer-grained, nested parallelism may be implemented by parallelizing the inner loop over nuclides (line 3 in Algorithm 1). However, it has been demonstrated that coarser-grained parallelism over only the outermost loop is a more effective use of available cores on CPU [12]. The most efficient OpenMP and OpenACC implementations both follow this scheme (Algorithm 2).

Likewise, in the GPU implementation of XSBench, each CUDA thread performs an independent macro XS lookup (Algorithm 3). Because threads within the same warp will request lookups for different materials with different numbers of nuclides, the flow-of-control will diverge within warp. Divergent thread execution limits the number of concurrent threads that are able execute in SIMT (single-instruction

**Algorithm 2** Monte Carlo Method with OpenMP and OpenACC

```
1  #pragma omp parallel private(...), shared(...)
2  #pragma acc data copyin(...), copy(...)
3  {
4    ...
5  #pragma omp for schedule(dynamic)
6  #pragma acc kernels loop gang, vector
7    for (i = 0; i < n_lookups; i++) {
8      ...
9  #pragma acc loop seq
10     for (nuc=0; nuc < n_nucs(material); nuc++) {
11       // Get (hi, lo), the bounding micro xs gridpoints
12       f = (hi->energy - p->energy) / (hi->energy - lo->energy);
13       macro_xs[0] += conc * (hi->micro_xs[0] - f * (hi->micro_xs[0] - lo->micro_xs[0]);
14       macro_xs[1] += conc * (hi->micro_xs[1] - f * (hi->micro_xs[1] - lo->micro_xs[1]);
15       // Get macro_xs for all interactions...
```

multiple-thread), thus limiting core utilization. Divergence also limits the number of simultaneous pending load requests, thus limiting bandwidth utilization. The effective bandwidth can be improved by several optimizations, including explicit prefetching instructions shown here (lines 6 and 7) as well as optimizations discussed elsewhere [13].

A single OCCA implementation can express optimal parallelism for both the CPU and GPU. OCCA expresses paraellism with "outer" and "inner" loops, each of which may be multidimensional. In XSBench, one-dimensional outer and inner loops are tightly nested, and independent macro XS calculations are performed in the innermost loop. When compiled for the OpenMP backend, the outer loop is parallelized, and the worksharing is effectively the same the same as the native OpenMP version. When compiled for the CUDA backend, the outer and inner loops are mapped to threads and blocks, respectively. OCCA syntax also allows for bandwidth optimizations on GPU, including the use of directLoad() for prefetching (which is ignored by the OpenMP backend).

Results for all implementations are shown in Figure 3.2. The OCCA implementations are compiled with the GNU OpenMP backend on CPU and with the CUDA backend on GPU. With both backends, the performance of the OCCA implementation is competitive with the native OpenMP or CUDA implemenations. Thus, the single OCCA kernel can optimally express parallelism on both CPU and GPU.

# 4. METHOD OF CHARACTERISTICS

## 4.1 Algorithm

MOC implements a ray-tracing scheme, whereby the spatial domain is discretized into tracks and neutron fluxes are attenuated along those tracks (Algorithm 5). For this implementation of 3D MOC, the geometry is assumed to be axially extruded and can be represented by a single radial plane, which is a superposition of all radial detail. Because of this, tracks along different polar angles (line 2) and axially-stacked planes (line 4) are independent. Material boundaries further subdivide the tracks into segments (line 3). The proxy app, SimpleMOC-kernel, iterates over a distribution of segments and energy groups, so the kernel consists of two nested loops.

**Algorithm 5** Method of Characteristics

```
1: for all 2D tracks do
2:     for all polar angles do
3:         for all material segments do
4:             for all axially-stacked stacked planes do
5:                 for all energy groups do
6:                     Attenuate Flux
```

## 4.2 Parallel Implementations

Unlike Monte Carlo, MOC has a high computational intensity and benefits from nested parallelism on the CPU (Algorithm 6). In the outer loop, each OpenMP thread attenuates fluxes within different segments (line 1). The inner loop, in which fluxes are attenuated over different energy groups, represents about 50 FLOPs with very little divergent execution. Hence, the fluxes can be attenuated in SIMD over multiple energy groups. On CPU, the inner loop can be fissioned into 12 simpler loops to assist the compiler's vectorization process; two of these loops are illustrated here (lines 3 and 9). The Intel compiler is able to vectorize all 12 loops, whereas the GNU compiler can only vectorize 3 loops.

The GPU implementations can also exploit nested parallelism (Algorithm 7). In CUDA, segments are mapped to blocks (using a two-dimensional grid, lines 2 and 3) and energy groups are mapped to threads. Within a warp, fluxes are attenuated in SIMT over multiple energy groups (lines 4-7). Since thread execution within a warp is minimally divergent, instruction throughput can remain high.

Currently, we have implemented a single OCCA kernel that is similar to the CUDA implementation (Algorithm 8). The kernel is expressed with a two-dimensional outer loop over segments (lines 1 and 2) and a single inner loop over energy groups (line 5).[1] When compiled for the CUDA backend, the outer and inner loops are mapped to blocks and threads, respectively, as in the native CUDA version. When compiled for the OpenMP backend, the outer loops are coalesced and mapped to OpenMP threads. However, the single inner loop is not successfully vectorized by the backend compiler (GNU), based on the backend's vectorization report.

[1]The OpenACC kernel (not shown) has a similar structure to the OCCA kernel.

**Algorithm 3** Monte Carlo Method with CUDA

```
1  __global__ void lookup_kernel(...) {
2    global_id = blockIdx.x + gridDim.x + threadIdx.x;
3    if (global_id >= n_lookups) return;
4    for (nuc=0; nuc < n_nucs(material); nuc++) {
5      // Get (hi, lo), the bounding micro xs gridpoints
6      __ldg(lo->energy); __ldg(lo->micro_xs[0]; __ldg(lo->micro_xs[1]); ...
7      __ldg(hi->energy); __ldg(hi->micro_xs[0]; __ldg(hi->micro_xs[1]); ...
8      f = (hi->energy - p->energy) / (hi->energy - lo->energy)
9      macro_xs[0] += conc * (hi->micro_xs[0] - f * (hi->micro_xs[0] - lo->micro_xs[0]);
10     macro_xs[1] += conc * (hi->micro_xs[1] - f * (hi->micro_xs[1] - lo->micro_xs[1]);
11     // Get macro_xs for all interactions ...
```

**Algorithm 4** Monte Carlo Method with OCCA

```
1  occaKernel void lookup_kernel(...) {
2    for (outer_id = 0; outer_id < outer_dim; outer_id++; outer0) {
3      for (inner_id = 0; inner_id < inner_dim; inner_id++; inner0) {
4        global_id = outer_id * outer_dim + inner_id;
5        if (global_id >= n_lookups) return;
6        for (nuc=0; nuc < n_nucs(material); nuc++) {
7          // Get (hi, lo), the bounding micro xs gridpoints
8          directLoad(lo->energy); directLoad(lo->micro_xs[0]; directLoad(lo->micro_xs[1]); ...
9          directLoad(hi->energy); directLoad(hi->micro_xs[0]; directLoad(hi->micro_xs[1]); ...
10         f = (hi->energy - p->energy) / (hi->energy - lo->energy)
11         macro_xs[0] += conc * (hi->micro_xs[0] - f * (hi->micro_xs[0] - lo->micro_xs[0]);
12         macro_xs[1] += conc * (hi->micro_xs[1] - f * (hi->micro_xs[1] - lo->micro_xs[1]);
13         // Get macro_xs for all interactions ...
```

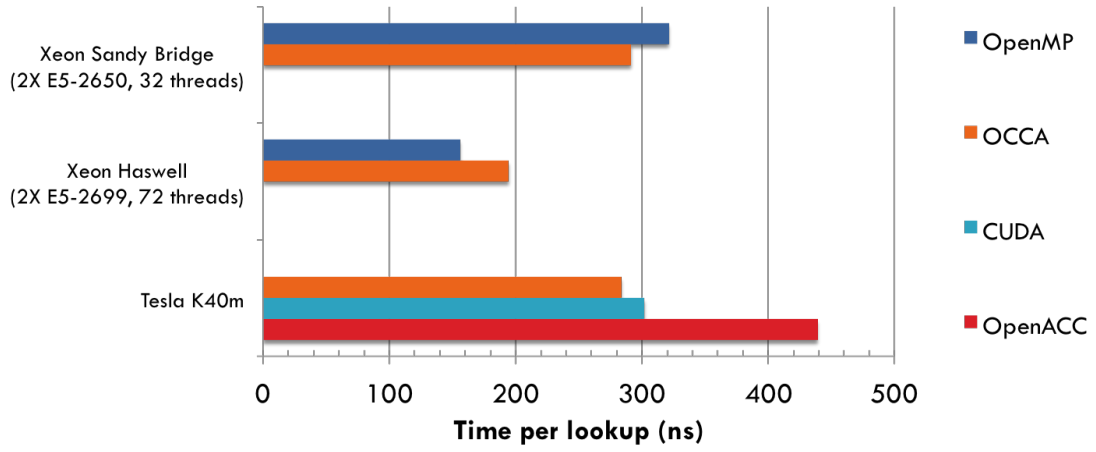



**Figure 1: Performance of XSBench on target architectures and programming models.**

Figure [?] demonstrates the performance portability of the OCCA kernel, compared to the other implementations. On Tesla K40m, the OCCA kernel is competitive with both native CUDA and OpenACC implementations. However, on the Sandy Bridge and Haswell Xeon CPUs, the OCCA kernel (compiled with GNU) is slower than the native OpenMP versions compiled with either GNU or Intel. The lack of vectorization in the OCCA version may be partially responsible for the performance gap, since the performance gap narrows when the native OpenMP versions are compiled without vectorization.

In SimpleMOC-kernel, it is nontrivial to write a single OCCA kernel that expresses optimal SIMD on CPU and SIMT on GPU. Expressing loop fission in the OCCA kernel may allow successful vectorization on CPU, since it was necessary in the native OpenMP version, but the implications for GPU performance are unclear. We are currently developing another OCCA kernel to explore these issues. We believe that similar problems would arise when writing a unified kernel GPU/CPU kernel in OpenMP or OpenACC.

# 5. CONCLUSIONS

In this work, we have presented many-core implementations of two methods for neutron transport, as represented by proxy applications. The Monte Carlo application, XSBench, does not make significant use of SIMD/SIMT operations and performs best without nested parallelism. In this case, a hardware-agnostic OCCA kernel can perform optimally on both CPU (compared to OpenMP) and GPU (compared to CUDA and OpenACC). The method of characteristics application, SimpleMOC-kernel performs best with nested parallelism by utilizing SIMD on CPU and SIMT on GPU. However, hardware-specific optimizations, such as loop-fission, were necessary to get peak performance on CPU. These optimizations were not trivial to express in a single OCCA kernel. In this case, the hardware-agnostic OCCA kernel performed optimally on GPU but sub-optimally on CPU. This work demonstrates the issues that arise when attempting to code a unified application kernel for multiple devices.

# 6. ACKNOWLEDGMENTS

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

# 7. REFERENCES

[1] N. Attig, P. Gibbon, and T. Lippert. Trends in supercomputing: The European path to exascale. *Computer Physics Communications*, 182(9):2041–2046, 2011.

[2] R. M. Bergmann and J. L. Vujić. Algorithmic choices in WARP âĂŞ A framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs. *Annals of Nuclear Energy*, 77:176–193, 2015.

[3] F. B. Brown. Recent Advances and Future Prospects for Monte Carlo. *Progress in Nuclear Science and Technology*, 2:1–4, 2011.

[4] G. Gunow, J. Tramm, B. Forget, K. Smith, and T. He. SimpleMOC - A Performance Abstraction for 3D MOC. In *ANS MC2015 – Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, Nashville, Tennessee, 2015. American Nuclear Society.

[5] B. M. Kochunas. *A Hybrid Parallel Algorithm for the 3-D Method of Characteristics Solution of the Boltzmann Transport Equation on High Performance Compute Clusters*. PhD thesis, University of Michigan, 2013.

[6] W. R. Martin. Challenges and prospects for whole-core Monte Carlo analysis. *Nuclear Engineering and Technology*, 44(5):151–160, 2012.

[7] D. S. Medina, A. St-Cyr, and T. Warburton. OCCA : A unified approach to multi-threading languages. *arXiv*, pages 1–25, 2014.

[8] D. Ozog, A. D. Malony, and A. Siegel. Full-core PWR Transport Simulations on Xeon Phi Clusters. In *ANS MC2015 – Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA), and the Monte Carlo (MC) Method*, page 112, Nashville, Tennessee, 2015. American Nuclear Society.

[9] C. M. N. a. Pereira, A. C. a. Mól, A. Heimlich, S. R. S. Moraes, and P. Resende. Development and performance analysis of a parallel Monte Carlo neutron transport simulation program for GPU-Cluster using MPI and CUDA technologies. *Progress in Nuclear Energy*, 65:88–94, 2013.

[10] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439–443, 2013.

[11] P. K. Romano and B. Forget. Parallel Fission Bank Algorithms in Monte Carlo Criticality Calculations. *Nucl. Sci. Eng.*, 170:125–135, 2012.

[12] A. R. Siegel, K. Smith, P. K. Romano, B. Forget, and K. G. Felker. Multi-core performance studies of a Monte Carlo neutron transport code. *International Journal of High Performance Computing Applications*, 28(1):87–96, July 2014.

[13] A. R. Siegel, J. Tramm, T. Scudiero, and P. K. Romano. A strategy for accelerating Monte Carlo criticality calculations using GPUs. *Annals of Nuclear Energy*, pages 1–14, 2014.

[14] J. R. Tramm and A. R. Siegel. Memory Bottlenecks and Memory Contention in Multi-Core Monte Carlo Transport Codes. In *Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013 (SNA + MC 2013)*, Paris, France, Oct. 2013. La Cite des Sciences et de l'Industrie.

[15] J. R. Tramm, A. R. Siegel, T. Islam, and M. Shulz. XsBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto, Japan, 2014. The Westin Miyako.

[16] X. G. Xu, T. Liu, L. Su, X. Du, M. Riblett, W. Ji, and F. B. Brown. An Update of ARCHER, a Monte Carlo Radiation Transport Software Testbed for Emerging Hardware Such as GPUs. In *Transactions of the American Nuclear Society*, volume 108, pages 433–434, 2013.

**Algorithm 6** Method of Characteristics with OpenMP

```
#pragma omp for
 for (int seg = 0; seg < n_segments; seg++) {
#pragma vector // SIMD ops for attenuating flux
   for (int g=0; g < n_energy_groups; g++) {
     tau[g] = sigT[g] * ds;
     sigT2[g] = sigT[g] * sigT[g];
   }
   ...
#pragma vector  // More SIMD ops for attenuating flux
   for (int g=0; g < energy_groups; g++) {
     flux_integral[g] = (q0[g] * tau[g] + (sigT[g] * ... ))
   }
   ...
```

**Algorithm 7** Method of Characteristics with CUDA

```
__global__ void kernel(...) {
  int blockId = blockIdx.y * gridDim.x + blockIdx.x;
  if (blockId >= n_segments) return;
  float tau = sigT * ds;
  float sigT2 = sigT * sigT;
  ...
  float flux_integral = (q0 * tau + (sigT * ...))
  ...
```

**Algorithm 8** Method of Characteristics with OCCA

```
for (int outerId1 = 0; outerId1 < outerDim1; outerId1++; outer1) {
  for (int outerId0 = 0; outerId0 < outerDim0; outerId0++; outer0) {
    int outerId = outerId1 * outerDim0 + outerId0;
    if (outerId >= n_segments) return;
    for (innerId0 = 0; innerId0 < n_egroups; innerId0++; inner0) {
      float tau = sigT * ds;
      float sigT2 = sigT * sigT;
      ...
      float flux_integral = (q0 * tau + (sigT * ...))
      ...
```

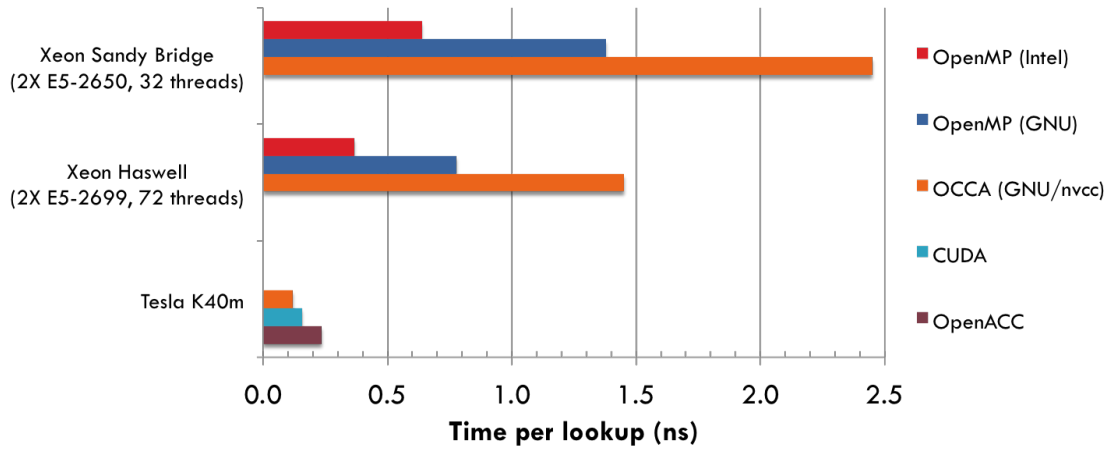



**Figure 2: Performance of SimpleMOC-kernel on target architectures and programming models.**

# Exploiting Hierarchical Exascale Hardware using a PGAS Approach

Karl Fürlinger
Department of Computer Science
Ludwig-Maximilians-Universität (LMU) München
Munich, Germany
Karl.Fuerlinger@nm.ifi.lmu.de

## ABSTRACT

Hardware architectures that enable Exascale-level performance are expected to break some long-held programmability assumptions on the node level and will come with a plethora of additional challenges that make the productive development of efficient applications difficult. One critical issue is data locality, which will become even more important than it is today. A shift towards data-centric programming models will be required to exploit the full potential of these machines. We present an overview of our work in progress on DASH, a data-structure oriented PGAS library implemented in C++, with which we attempt to address some of the challenges posed by upcoming hardware architectures by focusing on flexible data layout and by supporting a hierarchical locality model.

## Keywords

Exascale challenges, PGAS, data-oriented programming, multilevel locality.

## 1. INTRODUCTION

A number of daunting challenges have been identified on the way to Exascale computing [2]. Hardware architecture (particularly on the node-level) must change to achieve the desired performance and energy efficiency goals and this will have profound implications for the way in which high performance applications have to be written. *Locality of data access* will become an even more important aspect than it is today, as hardware vendors are forced to abandon node-wide cache coherence, several types of RAM (3D-Stacked, DRAM, NVRAM) are included, and complex on-chip networks interconnect are employed.

To accommodate these radical changes in hardware, programming models must transition from being *compute-centric* to being *data-centric* [16]. We argue that PGAS (partitioned global address space) approaches are particularly well suited for this transition and describe DASH, our own data-oriented PGAS approach realized in the form of a C++ template library as a step towards this goal.

In this paper we describe our ongoing work within the DASH project as well as our plans for the future and we show how PGAS approaches are well suited for the expected hardware characteristics of Exascale class machines – especially if a data-structure oriented approach is followed instead of a compute-centric one.

The rest of this paper is organized as follows: In Section 2 we summarize some of the challenges that have been identified for Exascale computing and describe how they impact programming of upcoming systems, especially on the node level. In Section 3 we describe DASH and our plans to address some of the identified challenges. Section 4 is dedicated to related work and in Section 5 we conclude and describe directions for future work.

## 2. EXASCALE CHALLENGES

It is expected that the hardware architectures employed in Exascale systems will differ significantly from current systems, especially on the node level. Below we summarize a selection of the most important challenges on the way to Exascale that have been identified in literature [3, 2] and we then describe how we plan to address these challenges with DASH in Sect. 3.

**Core Heterogeneity:** There will be different types of compute cores on a single chip: latency optimized "fat cores" and throughput optimized "thin cores", along with the possibility of on-chip accelerator cores similar to those found on current PCIe accelerator cards.

**Performance Heterogeneity:** Even among homogenous cores, several factors will lead to inherent performance variability of the compute characteristics, such as near-threshold voltage operation, automatic hardware error correction, and hardware frequency throttling to avoid thermal hotspots.

**Complex On-Chip Networks:** Cores will be connected by significantly more complex on-chip networks and locality to memory and the other cores will thus become a much stronger concern than in current machines.

**Noncoherence:** The memory system hierarchy will be subdivided into *coherence domains*. Cache coherence will only be provided between subsets of cores. Manual management of data coherence will be required between those coherency domains. It will thus become necessary to manage vertical locality more explicitly.

**Deep Memory Hierarchies:** The memory hierarchy will become deeper with different bandwidth and latency characteristics on different levels. This will most likely include a configurable scratchpad memory that must be manually managed. Thus, for some applications, a radical shift in

data structure layouts will be required to achieve energy efficiency and performance.

**Stratified Main Memory:** A variety of technologies will be available for main memory. Besides regular DRAM it is expected that systems will come equipped with high-bandwidth/low capacity 3D-stacked RAM as well as low-bandwidth/high capacity non-volatile RAM (NVRAM).

**Fault Tolerance and Recovery:** Traditional checkpoint-restart based approaches for fault-tolerance will not work at extreme scales due to the sheer size of the machines. More localized approaches will be necessary, where data can be recovered and computation be resumed on a local level.

**Code Modernization:** A lot of legacy software exists that will be ill suited for the hardware landscape of the Exascale era. Many applications will require substantial rewrites to enable them to achieve a significant fraction of available performance. Worse, as different vendors develop their specific design for Exascale, more than one such rewrite might be necessary.

While existing applications will be able to run on upcoming systems, for example with the help of automatic caching systems that take advantage of the additional levels of the memory hierarchy, the full potential of the hardware will only be unlocked when the changes on the hardware side are reflected in the programming model. Finding the right abstractions for these new programming models is a motivation for the DASH project.

## 3. DASH

DASH [8] is a data-structure oriented C++ template library that implements a PGAS model by relying on a one-sided communication substrate which is accessed through an intermediate runtime layer called DART (the DASH runtime). DASH can be used within a shared memory node as well as between nodes and it provides an iterator-based interface that is similar to the data containers of the C++ Standard Template Library (STL). In DASH, data elements are distributed over several *units* (the individual participants in a DASH program) and by using techniques such as operator overloading, the DASH distributed data containers can be used in much the same way that local data structures are used. DASH is currently being developed as part of the priority program for software for Exascale systems (SPPEXA)[1], funded by the German research foundation (DFG). A second phase of funding (covering some of the planned features outlined in Sect. 3.2) is currently under review.

The rest of this section discusses some of the features of DASH and how they can help address some of the challenges faced when transitioning to Exascale hardware.

### 3.1 Programming with DASH

Fig. 1 shows a simple example stand-alone DASH program. DASH follows the SPMD (single program, multiple data) model of execution and the number of units executing a

[1] `http://www.sppexa.de`

```
1  int main(int argc, char* argv[])
2  {
3    dash::init(&argc, &argv);
4
5    dash::Array<int> a(1000);
6
7    if( dash::myid()==0 ) {
8      // global access and standard algorithms
9      std::sort(a.begin(), a.end());
10   }
11
12   // local access using local iterators
13   std::fill(a.lbegin(), a.lend(), 23+myid);
14
15   dash::finalize();
16 }
```

Figure 1: An example stand-alone DASH program.

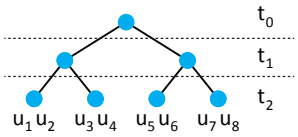
program is specified at program launch time. Sets of units can be grouped into *teams* and teams form the basis for all memory allocation and collective synchronization and communication operations. Teams are arranged in a hierarchy in DASH. The default team `dash::Team::All()` comprises all the units in the program and a new team can only be formed as a sub-team of an existing team. In line 5, a distributed `dash::Array a` is allocated. The array `a` contains 1000 integers and is distributed over the memory of all units in the program. The way in which the distribution occurs can be controlled by specifying a distribution `dash::Pattern`. Predefined patterns are `BLOCKED`, `CYCLIC` and `BLOCKCYCLIC`, if a pattern is not specified explicitly, `BLOCKED` is the default.

DASH follows a *global-view* approach, which means that the data structures in DASH represent global objects and can be accessed using global indices; i.e., for the same index `i` the expression `a[i]` refers to the same array element, regardless of which unit evaluates the expression. Global-view programming is an important productivity feature, because it allows distributed data structures to be treated much like regular local data structures or STL containers. In fact, DASH arrays can be used with standard STL algorithms, as shown in line 9 of Fig. 1.

For performance reasons it is however critically important to also enable efficient local access and indexing. This is achieved in DASH by the provision of a local proxy object (`.local`) and by supporting local iterators `.lbegin(), .lend()` that operate only on the local part of an array. An example for this mode of usage is shown in line 13 of Fig. 1.

Besides local and global access, DASH supports hierarchical views on data and hierarchical iterators. Hierarchical views exploit the arrangement of teams in a tree. For example `a.hview<2>()` defines the hierarchical view of `a` two levels "below" the allocation team.

This situation is shown in Fig. 2 where eight units are arranged in a two level hierarchy.




```
t0=dash::Team::All();
t1=t0.split(2);
t2=t1.split(2);
```

Figure 2: An example team hierarchy. Eight units $(u_1, \ldots, u_8)$ are arranged in a two-level hierarchy. The function `split(n)` splits the parent team into two equal-sized sub-teams.

```
1  dash::Team& t0 = dash::Team::All();
2  dash::Team& t1 = t0.split(2);
3  dash::Team& t2 = t1.split(2);
4
5  dash::Array<int> a(1000);
6
7  if(myid==3) {
8    // access on level of t1
9    auto hv1 = a.hview<1>();
10   for( auto el: hv1 ) { cout<<el; }
11
12   // access on level of t2
13   auto hv2 = a.hview<2>();
14   for( auto el: hv2 ) { cout<<el; }
15
16   // local access
17   auto hv3 = a.hview<-1>();
18   for( auto el: hv3 ) { cout<<el; }
19 }
```

In this example, `hv1` is the hierarchical view of the array `a` on the level of $t_1$. For unit 3 this represents the indices `a[0],...,a[499]`, assuming a blocked distribution. Similarly, `hv2` is the hierarchical view of `a` on the level of $t_2$. For unit 3 this represents `a[250],...,a[374]`. The `hview<>` template class also offers two specializations. `hview<0>` represents the whole global array and `hview<-1>` represents the local part of the data only.

## 3.2 Addressing Exascale Challenges

In this section we describe some of the existing and planned features for DASH and how they address the Exascale challenges outlined in Sect. 2.

**Global Address Space:** Similar to other PGAS approaches, DASH works both on top of shared memory and distributed memory hardware. The loss of cache coherence across full nodes is thus not a fundamental problem – the underlying runtime system can be adapted to use the most efficient form of synchronization and communication for each level in the hierarchy.

**Hierarchical Locality:** DASH was designed from the ground up to support hierarchical locality. Groups of units are organized in teams, which form a hierarchical structure. Teams build the basis for memory allocation, synchronization, and communication operations. Using the team concept, DASH allows for a more fine-grained control over data access patterns with multiple hierarchical locality levels. Instead of the normal two-level distinction (local/remote) it is possible in DASH to access data at a certain "distance" by using hierarchical locality iterators, as shown in Sect. 3.1.

**Memory and Execution Spaces:** In its present version, DASH does not provide an explicit execution model and implicitly fulfills memory allocation requests by using the node's main memory (DRAM). Future versions will relax these two restrictions by developing the notion of *explicit memory and execution spaces*.

A memory space is the representation of a unit's memory allocation capabilities from a physical memory of a certain type. In the abstract machine models envisioned for Exascale architectures [2] this could for example be high bandwidth 3D stacked memory, NVRAM, conventional DRAM, or even an explicitly managed scratchpad memory. Representing memory spaces as different types within the C++ template library (e.g., `dash::array<int, SCRATCH> a(1000);`) allows the compile-time specialization of the various allocation options with no runtime overhead.

Similar in spirit to memory spaces, execution spaces represent a unit's compute capabilities. Depending on the particular hardware characteristic, a unit might be able to launch tasks to a subset of the fat cores, thin cores and/or accelerator cores available on a node and this choice can similarly be encoded in template parameters to allow for a compile-time specialization.

**Data Redundancy:** DASH already realizes a virtualization layer for the access to data items stored in the separate physical memory of several nodes. Provided with a fault-tolerant foundation for the runtime system, it is conceptually easy to extend this virtualization concept to support multiple storage locations for a single data item so as to support a fault-tolerance mechanism through the redundant storage of data. I.e., if the node holding the primary copy of a data element becomes unavailable, it can be reconstructed from another node that has a copy.

DART, the DASH runtime layer, is currently based on MPI-3 RMA (remote memory access) operations and MPI is presently not a suitable basis for a fault-tolerant implementation of DASH. This situation might however change in the future and other candidate one-sided communication substrates with stronger support for fault-discovery and tolerance such as GASPI [10] are currently evaluated within our project.

**Persistent Data:** Data must be the primary concern for programmers in the Exascale era. It is rare that the useful lifetime of a dataset encompasses just a single application. Often a workflow of applications is combined to generate and analyze simulation results. In current practice, applications are often coupled via writing and reading of intermediate files - simply because this is the least common denominator between applications.

To improve upon this situation, we propose a persistent data dock as a solution to the important problem of application coupling and data interchange while simultaneously providing an incremental path to code modernization even for non C++ applications.

A schematic illustration of the planned DASH data dock is shown in Fig. 3. While a regular PGAS application reads and writes its own data structures using PGAS data access primitives, the DASH *data dock* generalizes this idea to allow

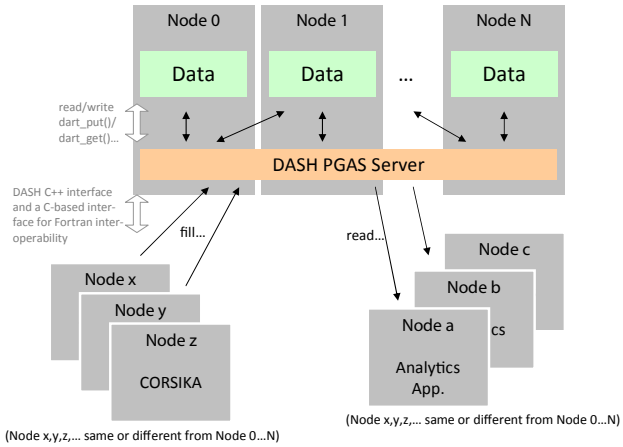



Figure 3: A schematic illustration of the DASH data dock.

other applications access to the data using the concept of a DASH PGAS "server". While this model conceptually allows several applications to access the data at the same time, the more common approach will be to hand off ownership of the data from one application of the workflow to the next.

Note that in Fig. 3 nodes a,b,c,... and x,y,z,... can be identical (or sub-sets) of nodes $0, 1, \ldots, N$. In the case of an execution on the same hardware, fast direct access is supported while remote data access requires some kind of network-based access. In both cases C++ applications can make use of the advanced C++ data access methods provided by DASH, while inter-operability with FORTRAN is realized by a simplified C-based interface.

# 4. RELATED WORK

Traditional PGAS approaches typically come in the form of a library (e.g., OpenSHMEM [15], Global Arrays [13]) or language extension (Unified parallel C, UPC [1], Co-Array Fortran, CAF [12, 14]). Those solutions usually don't address hierarchical locality and offer only a two-level (local/remote) distinction of access costs. Typically these approaches also only offer one-dimensional arrays as their basic data-type out of which more complex data structures can be constructed – but the work to do that falls on the individual programmer.

More modern PGAS languages such as Chapel [5] and X10 [6] address hierarchical locality (e.g., in the form of locales or hierarchical place trees [17]) but using these approaches requires a complete re-write of the application. Given the enormous amounts of legacy software, complete rewrites of large software packages are unlikely to happen.

Data structure libraries place their emphasis of providing data containers and operations on them. Kokkos [7] is a C++ template library that realizes multi-dimensional arrays with compile-time polymorphic layout and allows parallel operations executed over their data items. Kokkos is an efficiency-oriented approach trying to achieve performance portability across various manycore architectures. While Kokkos is limited to shared memory nodes and does not address multi-level machine organization, a somewhat similar approach is followed by Phalanx [9], which also provides the ability to work across a whole cluster using GASNet as the communication backend. Both approaches can target multiple back-ends for the execution of their kernels, such as OpenMP for the execution on shared memory hardware and CUDA for execution on GPU hardware.

STAPL [4] is a C++ template library for distributed data structures supporting a shared view programming model. STAPL is not a bona-fide PGAS approach (it does neither offer the abstraction of a global address space nor has it the concept of global pointers) but provides distributed data structure and a task-based execution model. STAPL offers flexible data distribution mechanisms that do however require up to three communication operations involving a directory to identify the home node of a data item. PGAS approaches in HPC usually forgo the flexible directory-based locality lookup in favor of a statically determinable location of data items in the global address space.

Recently, C++ has been used as a vehicle for realizing a PGAS approach in the UPC++ [18] and Co-array C++ [11] projects. While our previous work on the DASH runtime is based on MPI, UPC++ is based on GASNet. Porting an existing MPI application will therefore be more straightforward using DASH. Co-array C++ follows a strict local-view programming approach and is somewhat more restricted than DASH and UPC++ in the sense that it has no concept of teams to express local synchronization and communication.

# 5. CONCLUSION AND FUTURE WORK

The first public release of DASH is currently under preparation. This version will feature a one dimensional array (`dash::array`) as the basic data structure and will include an MPI-3 based runtime system that uses RMA (remote memory access) operations as a one-sided communication back-end []. The immediate next steps for the project are the development of a multi-dimensional matrix datatype and the exploitation of on-node shared memory facilities in the runtime system. Longer term plans for the project focus on the support of more irregular data structures such as lists and hash tables as well as the provision of a task-based execution model.

## Acknowledgments

We gratefully acknowledge funding by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA).

# 6. REFERENCES

[1] The UPC consortium: UPC language specification v1.2. June 2005. Technical Report, Lawrence Berkeley National Laboratory.

[2] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. Abstract machine models and proxy architectures for exascale computing. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 25–32. IEEE Press, 2014. Extended version

available online at `http://www.cal-design.org/publications/publications2`.

[3] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[4] A. Buss, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, L. Rauchwerger, et al. STAPL: standard template adaptive parallel library. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, page 14. ACM, 2010.

[5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, August 2007.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM Sigplan Notices*, 40(10):519–538, 2005.

[7] H. C. Edwards and C. R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop (XSW), 2013*, pages 18–24. IEEE, 2013.

[8] K. Fürlinger, C. Glass, A. Knüpfer, J. Tao, D. Hünich, K. Idrees, M. Maiterth, Y. Mhedeb, and H. Zhou. Dash: Data structures and algorithms with support for hierarchical locality. In *Euro-Par 2014 Workshops (Porto, Portugal)*, 2014.

[9] M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Nov. 2012. Salt Lake City, UT, USA.

[10] GASPI - global adress space programming interface webpage `http://www.gaspi.de`. Retrieved Jan. 2012.

[11] T. A. Johnson. Coarray C++. In *7th International Conference on PGAS Programming Models*, 2013. Edinburgh, Scotland.

[12] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, and G. Jin. A new vision for coarray Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models (PGAS '09)*, New York, NY, USA, 2009. ACM.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[14] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[15] S. W. Poole, O. Hernandez, J. A. Kuehn, G. M. Shipman, A. Curtis, and K. Feind. Openshmem - toward a unified rma model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer US, 2011.

[16] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, B. Goglin, C. Edwards, C. J. Newburn, D. Padua, D. Unat, E. Jeannot, F. Hannig, T. Gysi, H. Ltaief, J. Sexton, J. Labarta, J. Shalf, K. Fürlinger, K. O'Brien, L. Linardakis, M. Besta, M.-C. Sawley, M. Abraham, M. Bianco, M. Pericàs, N. Maruyama, P. H. J. Kelly, P. Messmer, R. B. Ross, R. Cledat, S. Matsuoka, T. Schulthess, T. Hoefler, and V. J. Leung. Programming Abstractions for Data Locality. Research report, PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center (CSCS), Lugano, Switzerland, Nov. 2014.

[17] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 172–187. Springer-Verlag, 2010.

[18] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS extension for C++. In *28th IEEE International Parallel & Distributed Processing Symposium*, 2014.

# Enabling Adaptive, Fault-tolerant MPI Applications with Dynamic Resource Allocation

Maciej Szpindler
Interdisciplinary Centre for Mathematical and Computational Modelling
University of Warsaw
m.szpindler@icm.edu.pl

## ABSTRACT

Dominant execution mode for the most of HPC applications using MPI or hybrid parallel model rely on static resource allocation. While this is explicitly mapped to both MPI usage scenarios and runtime systems, it entails restrictions for efficient dynamic execution. This limitation becomes critical when fault tolerance of MPI applications is concerned. Also some classes of applications require dynamic process creation. This paper discusses enabling of dynamic resource allocation for adaptive execution and fault tolerance in MPI applications. Existing functionality of the MPI library and underlying layers and required interactions are described. Implementation and common usage scenarios are presented. Practicality is discussed, showing performance and limitations on synthetic experiments.

## 1. INTRODUCTION

Recent HPC systems build their computational capabilities and performance on extreme number of processing elements - either multicore nodes, hyperthreaded cores, accelerated nodes or other constructs. This requires applications to be extremely parallel and scalable to run efficiently on those systems and benefit from their peak performance. On the other hand, in real usage several applications share one large system at the same time. It is quite obvious since such systems are expensive and consume a lot of resources even when idle. Parallel applications need fair and optimal sharing of the computational resources which is usually provided by resource manager or queuing system.

One of the common constraints for programmers in the described shared computing environment is static resource allocation. Although MPI and other parallel programming models have constructs that allow dynamic process creation and management, it is not easily manageable in a shared multi-task system. Newly created processes must use already allocated resources which leads to either waste of resources that need to be preallocated or processes oversubscription which likely results in performance degradation. Moreover fault tolerance of parallel applications becomes a real need. Common scenarios for process fault recovery assume a repair stage which implies dynamic creation of new processes.

In this work a simple model for the complete dynamic multi-process execution is proposed. The approach uses existing methods for resource allocation resizing in the Slurm environment. It allows dynamic multi-process applications in a shared environment. The latter gives more opportunities for better resource utilization and allows programmers to create more flexible applications that may allocate resources dynamically and exclusively which is usually required to achieve full performance and essential when addressing fault recovery.

This paper is organized as follows: section 2 is a study of a current context for most popular MPI implementations and their integration with scheduling systems. MPICH and OpenMPI projects are addressed and Slurm system as resource manager. Section 3 describes application of the dynamic resource re-allocation for failure recovery of the MPI jobs. An analysis of MPI communicator reconstruction and process failure recovery is discussed. Performance study for synthetic scenarios are described in section 4.

The key contributions of the described work include:

- A simple model for MPI user application with batch scheduler interaction.
- An implementation of the proposed model.
- An analysis of the available MPI communicator repair mechanisms and its integration with resource allocation.
- A study of a failure recovery performance for simple model application.

## 2. PROCESS AND RESOURCE MANAGEMENT INTERACTION

Most HPC systems that support multiple applications execution separate resource allocation and actual application execution. Queueing system or resource manager is usually responsible for allocation which is preceding step to application startup. Application processes management is more system-wide task and possibly many components are involved. MPI implementations introduce a form of process encapsulation for better execution control. Nevertheless all these components interact closely as shown on figure 1.

### 2.1 MPI level

MPI standard [7] defines functionality for process creation and management. There are library calls that allow dynamic process creation. It is MPI runtime responsibility to create

and start new processes within already initialized application instance. It allows a user to build dynamic applications that can spawn new processes whenever it is required.

While dynamic process creation is initialized within user MPI application, the actual process management is usually delegated to lower level system mechanism. It may depend on a specific MPI library implementation but most of modern implementations rely on the quasi-standardized API called Process Management Interface (PMI) [1]. There are two existing generations of the API (PMI and PMI2). In the case of MPICH, MPI process spawning model is delegated to the PMI infrastructure. For two mainstream MPI libraries, MPICH [8] is providing PMI layer implementation tightly integrated within its own process manager called *hydra* while OpenMPI [9] has a similar approach with closely related project called *PMIx*.

What MPI standard does not define is how the application processes are pinned to the available hardware. It is up to system levels (either operating system scheduler or dedicated resource manager like queuing system) how these processes are handled and what resources would be allocated to the MPI application. The reason for this is the variety of the systems and underlying execution models that MPI is intended to support. This approach is the state-of-the art of the parallel computing on the most of available HPC systems. As a consequence, most of the MPI applications are used in the following scenario: pre-allocate sufficient amount of resources (cpus, memory, other), start application when resources are available, wait for completion. This scenario enforces that resources are fixed throughout application run time, what is also the most practical approach.

## 2.2 Slurm integration

One of the commonly used resource managers is Slurm [10]. It allocates required resources that allow user run application in a batch job scheme. It has modular design and is organized with plugins that control job creation and execution. One of the functionalities of the system is to extend an existing allocation with additional resources. The opposite operation with resources reduction is also available. Slurm provides API for its functionality which implements some subset of the PMI interim process management layer. This allows communication between MPI methods from the user application and job management layers handled by the Slurm system. That allows to manage dynamic resource allocation, in a form of resizable user jobs, available from user level. Integration that would make it easily accessible and usable from user application requires control flow from application through the MPI and PMI levels to Slurm API. In our approach we allocate new resources and create new processes which enable an MPI application to utilize these resources dynamically within one Slurm job.

The proposed approach is based on an MPI process spawning model which only allows for a blocking mode of process creation. While resource allocation is usually not immediate in a system with many users and their jobs, the non-blocking mode is more practical. A basic implementation which extends the MPI standard has been proposed which partly follows previous ideas on non-blocking process creation considered on the MPI-Forum [6].

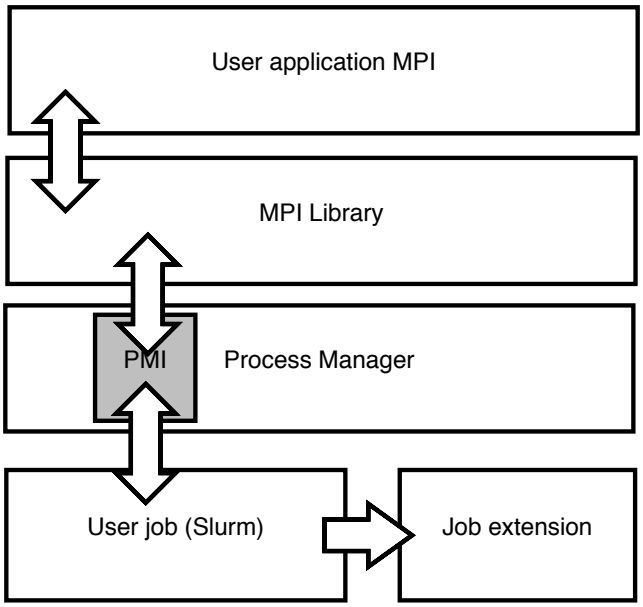



**Figure 1: Schematic diagram of interaction between process and resource managing layers.**

Additional information is passed from the MPI spawn call using info object. This set of key-value pairs provide the runtime system on how to start new processes, as standard defines. Such information can be also request on extra resources allocation. Appropriate key parameter need to be then parsed and interpreted by runtime system - PMI library in this case. This is the place where integration has been implemented: PMI reaction to the process creation together with resource allocation request resulting in Slurm job extension.

## 2.3 Modes of dynamic resource allocation

Availability of resources that need to be allocated depends on the load of the system and Slurm site configuration. In case when application must wait for these resources and their actual availability time is uncertain different approaches are possible. Allocation request may block execution until allocation is granted or given timeout reached. This is the simplest, blocking mode, implemented with a use of the `slurm_allocate_resources_blocking` API function. Another mode is immediate in which resources are allocated only if already available. Otherwise the request is rejected and error is returned. It is allocated using immediate constraint of the Slurm allocation request. The latest mode considered is non-blocking, but not really immediate, when execution control is returned but allocation request is pending and its completion need to be probed with additional function call or signalled with callback function. This mode has been implemented using small extension to the standard MPI functionality.

# 3. FAULT-TOLERANCE WITH MPI

Various approaches have been explored in the context of fault tolerance and failure recovery in the MPI model. Specific MPI implementations [4] and other mechanisms [5] have been proposed but without successful adoption in a form of

standard inclusion. Recent proposal of fault tolerance primitives called User Level Failure Mitigation [6] (ULFM) found wide recognition and is considered for future MPI standard version inclusion. ULFM extensions are partly implemented in both OpenMPI and MPICH projects and first analyses of MPI applications with ULFM based fault tolerance have been published [2][3].

One of the most common issues regarding failures in the MPI model is process faults handling. One may expect that in case of process failure global consistency of all applications components is preserved or may be recovered and application is able to continue. This requires a mechanism to recover MPI communicators that experienced process failure. Communicators are abstract construct that stores processes groups and respective communication context. ULMF contains primitives enabling communicator failure detection, acknowledgement and revoke operations. Recovery employs *shrink* operation that excludes all failed processes and creates new, possibly reduced, communicator. Complete restore that would recreate the failed processes requires user choice and implementation.

## 3.1 Communicator reconstruction

Reconstruction of communicator, that contains failed processes, may be realized at least in two scenarios. First, with direct application of ULFM *shrink* operation, which results in a new, consistent but reduced in size communicator. Second, similar to the first but with additional operations which fully reconstruct pristine communicator. The latter scenario requires to restart failed processes by applying MPI process spawning. This is the place where dynamic resource allocation is very helpful.

## 3.2 User-Level Failure Mitigation approach

ULMF primitives allow to implement fault handling that is most appropriate for user application. While failure is already detected by either acknowledge operation or by revoking the failed communicator, one can reconstruct the communicator. The task of restoring spawned processes to the state before failure need to be assured by the application and is not a part of the ULMF model.

## 3.3 Common scenarios

Two scenarios simulating real failure cases have been used. First analyses a single process failure. Synthetic case when chosen process dies has been used to explore costs of the complete communicator reconstruction. This corresponds to the situation when: either single process fails while other communicator members continue to run; or situation when node that executes single multithreaded process dies but other nodes participating in the communicator remain unaffected (top of the figure 2). Realization is a simple MPI application with main loop performing iterations of cyclic, non-blocking point to point communication between processes arranged in a ring in a single communicator. In each iteration communicator is tested against process failure. If any process is reporting communication failure then communicator is revoked and eventually reconstructed. Re-created process is spawned together with newly allocated resource (processor). Application is then ready to be fully restored by either withdrawal to the last successfully completed iteration stored in memory or checkpointed to the local file.

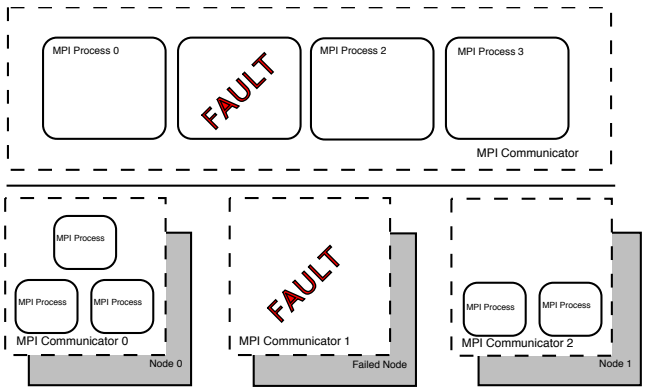



**Figure 2: Common failure scenarios addressed.**

An other scenario that has been modelled is a system node failure. It mimics the application, that uses separated inter-node communication context and intra-node synchronization, encountering node failure (bottom of the figure 2). This approach is natural for two level parallelism with MPI+MPI model that is composed of MPI communication across the nodes and MPI shared memory windows within the node. It allows to reduce communication costs and eventually optimize message exchange on the MPI internal level. MPI provides convenient functionality that partitions global communicator into a disjoint subgroups of given type and shared memory islands specifically. This exposes intra-node shared memory regions for local processes.

In this case dedicated intra-node communicator has been used. While computing node fails, respective communicator disappears and fault-tolerant application need to handle with corrupted communicator. With a choice of ULFM model, one must decide on detection technique. Two approaches aiming at distributed detection, not involving all participating processes, have been studied.

First approach relies on the MPI inter-communicators. This allows easy detection of the remote node failure locally. While ULFM implementation provided by MPICH does not support inter-communicators as of begining of 2015, it was not tested. Latter approach does not involve inter-communicators. Instead of global failure testing, which is not scalable as all processes are involved, more distributed attempt has been proposed with a special communicators structure.

Synthetic mini-application has been used to study intra-node communicator reconstruction. It makes two level reduction: locally over shared memory node and globally over nodes using MPI reduction. Every time global reduction raises fault error, local communicator associated with the failed node is detected and eventually re-created with a new node allocated dynamically. At this stage application is ready to be revoked to the desired point of execution using checkpoints. The use of checkpoints obviously introduces significant memory footprint and synchronization overhead. Other choices are possible in case when memory load is critical factor for application performance.

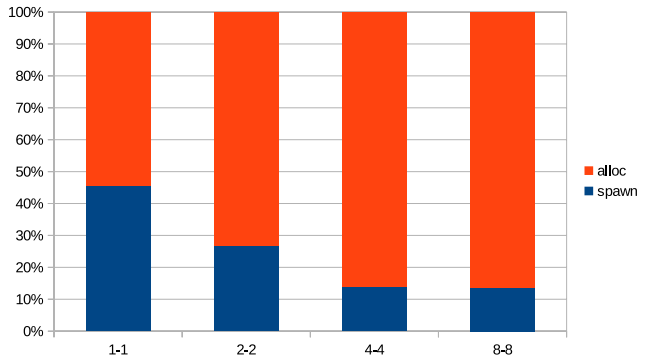



**Figure 3: Relative cost of the spawn and allocate operations for increasing number of processes (N-M: number of parents and children).**

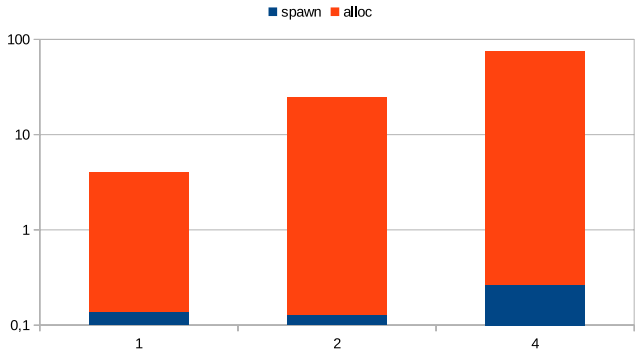



**Figure 4: Time cost in seconds of the spawn with allocing additional node. Note the logarithmic scale.**

Both scenarios have been explored on generic x86 cluster with multicore nodes running Slurm version 14.03.5 and MPICH version 3.2a2.

## 4. PERFORMANCE AND SCALABILITY

It has been already shown that communicator reconstruction with recreating failed processes introduces significant overhead [2]. This is associated with `MPI_Comm_spawn` implementation that have not received broader attention and serious optimizations since it was introduced.

Considering dynamic allocation one must expect further overheads due to user job resizing which involves many, possibly slow, system components. Overheads in the case of synthetic scenarios has been measured in *immediate* allocation mode.

Immediate allocation has been implemented using native Slurm request feature. The feature causes to immediately allocate resources if currently available or to raise an error in the other case. The non-blocking allocation scheme described in the previous section means immediate return to execution after the allocation request but not necessarily immediate availability of the resources. Thus blocking neither non-blocking allocation mode was studied in case of performance while it depended on external conditions and availability of the resources.

First experiment measured costs of a simple process spawning with extending allocation. Newly allocated resources have used either the same computing node or allocated additional node to spawn new process on it. Figure 3 shows overheads for *spawn* with *allocate* with immediate allocation on the same (local) node.

Second set of experiments explored node failure scenario. Performance of the intra-node communicator reconstruction with a use of dynamically allocated nodes has been analysed. Cost of the node failure detection tends to depend on the size of the intra-node communicator (number of local processes) only and does not affect scalability. Further overheads are observed for dynamic allocation of nodes, due to user job resizing which involves many, possibly slow, system components. Experimental results have been collected using ”immediate” allocation mode and figure 4 shows linear grow in time required to allocate new node for each of the nodes used in opposite to nearly costant cost of remote process spawn (logarithmic scale is used for visibility).

## 5. SUMMARY

Various techniques for MPI fault tolerance and recovery have been proposed. While the preferred approach would formalize there is still need to enable mechanisms for efficient application execution after successful recovery. Dynamic resource allocation is one of possible approaches. Moreover applications that require variable system load and computing power during execution can benefit from such construct. Is has been shown how to enable dynamic allocation of resources for MPI application by tighter integration of system runtime components. While this proved to be practical and should not introduce major issues for portability, performance and overhead costs still need to be addressed to improve the proposed solution.

## 6. REFERENCES

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur. PMI: A scalable parallel process-management interface for extreme-scale systems. In *Recent Advances in the Message Passing Interface*, pages 31–41. Springer, 2010.

[2] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184, 2013.

[3] W. Bland, K. Raffenetti, and P. Balaji. Simplifying the recovery model of user-level failure mitigation. In *Proceedings of the 2014 Workshop on Exascale MPI*, pages 20–25. IEEE Press, 2014.

[4] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent advances in parallel virtual machine and message passing interface*, pages 346–353. Springer, 2000.

[5] W. Gropp and E. Lusk. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[6] MPI Forum. Message Passing Interface (MPI) Forum Home Page. http://www.mpi-forum.org/.

[7] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0, September 21 2012. Available at: http://www.mpi-forum.org.

[8] MPICH: High-Performance Portable MPI. Available at: http://www.mpich.org/.

[9] Open MPI: Open Source High Performance Computing. Available at: http://www.open-mpi.org.

[10] Slurm Workload Manager. Available at: http://slurm.schedmd.com.

# Evaluating Stencil Codes at Scale

Manish Modani
STFC Daresbury Lab.,
Warrington, U.K.
mamodani@in.ibm.com

Rupert W. Ford
STFC Daresbury Lab.,
Warrington, U.K.
rupert.ford@stfc.ac.uk

Michael Johnson
IBM Research, Dublin,
Republic of Ireland
michaelj@ie.ibm.com

Constantinos Evangelinos
IBM Research,
Yorktown, USA
cevange@us.ibm.com

## ABSTRACT

Stencil-based codes are widely used in Scientific Computing and are considered to be good candidates for running at scale as their communication depends on their local neighbours and is therefore independent of the number of parallel tasks. A 2-dimensional stencil kernel has been written to determine whether stencil-based codes do indeed scale on todays supercomputers. This kernel is able to perform halo communication that represents the communication a stencil code would require, for a range of data sizes and parallel tasks. It is shown that this kernel does not scale well on an IBM BGQ (Blue Joule, 64K cores), a Cray XK7 (Titan, 16K cores) and a Cray XC30 (ARCHER, 16K cores) even if the suggested topology mapping tools are used. In weak scaling tests, performance degradations of up to x26, x34 and x28 respectively, are observed. A new task-to-topology mapping scheme is presented for a torus network which maximises intra-node communication and minimises network contention. Weak scaling performance results on the IBM BGQ show that when using this scheme there is no performance degradation for all data sizes and number of parallel tasks. The resultant time savings give a reduction in energy consumption of up to 64%.

**Keywords:** stencil code, 2-dimensional, weak scaling, mapping, performance, torus, energy.

## 1. INTRODUCTION

In the modern era of computations, most of the parallel applications tasks require to communicates with neighbors (e.g. share data with each others). For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Similarly, in the environmental sciences (e.g. weather/climate codes), boundary points need to be updated from neighbouring grid points etc. Communication among the neighbours is one of the biggest bottleneck in the modern simulation code's performance. Even on the current machines, cost to exchange the data among the neighbour's is a critical issue and the push towards the Exascale (when the machines will have larger number cores > 100000) will only exacerbate this (Modani and Porter 2015). In the present study, the performance of Stencil-based code, with the assumption that stencil codes scale, is analyzed on two different architectures e.g. (IBM/CRAY).

The theoretical scalability of stencil codes can be easily understood if one considers a weak scaling example. with weak scaling, using a stencil code with a regular mesh and regular parallel partition, each task will communicate with the same number of "neighbour" tasks and communicate the same amount of data to these enighbour tasks irrespective of the number of tasks being used. As the computation per task remains the same in a weak scaling code, the compute to communicate ratio is the same irrespective of the number of tasks. This indicates that the execution time, in case of weak scaling, should remain constant with the increase in number of task (see the straight line in Figure 2).

Stencil (nearest-neighbor) computations are widely used in scientific computing, including structured grids as well as implicit and explicit partial differential equation (PDE) solvers in domains including thermo/fluid dynamics, climate modeling and electromagnetics (Yongpeng and Frank, 2012).

In this work, a stencil kernel has been written to test the scalability of 2-dimensional stencil-based codes. The next section describe the development of the stencil benchmark. The MPI implementation, execution and the observed performance on the IBM BGQ (Blue Joule), Cray XK7 (Titan) and Cray XC30 (ARCHER) are described in Sections 3 and 4 respectively. Analysis of the observed poor scalability is discussed in Section 5. The proposed mapping scheme for improved scalability and energy consumption is explained in Sections 6 and 7 respectively. Finally conclusions are made in Section 8.

## 2. BENCHMARK DEVELOPMENT

A parallel two-dimensional stencil kernel has been written to determine how well stencil-based codes scale on modern supercomputers. This kernel performs the halo communication that a stencil code would require, but does no computation. As shown in Figure 1, each rank communicates with its neighbours in the X (east, west), and Y (north, south) directions respectively. Periodic boundary conditions are implemented so that a ring is made in the X and Y directions. For example, in the case of a 4x4 decomposition (shown in Figure 1), ranks 1, 5, 9 and 13 communicate to 4, 8, 12 and 16 respectively. Similarly, in the Y direction, ranks 1, 2, 3 and 4 communicate to 13, 14, 15 and 16 respectively.

The benchmark makes use of the Message Passing Interface (MPI) library to communicate between tasks. To avoid deadlock in the MPI communication, a red-black algorithm is employed. All Red (even) ranks send and then receive their messages in the X and Y directions. All Black (odd) ranks first receive and then send their messages in the X and Y directions. The halo communication is repeated 2000 times.

The X, Y (north, east, south, west) communication pattern is representative of a 2-dimensional partitioning of a 5-point stencil on a regular grid. The particular pattern and sizes were chosen to represent what is used by a large number of Atmosphere models in Climate and Weather Forecasting. Such models have a very large number of points in the horizontal and relatively few in the vertical (as the atmosphere height is small

compared to the area of the earth). Therefore these grids are invariably parallelised in the horizontal, leaving the vertical to run sequentially within a partition. Thus a partition contains a set of columns. The number of columns per partition will vary depending on the problem size and number of cores but is typically between 50 and 100 columns for cache based architectures, with each column having around 100 levels. The number of columns per partition is likely to need to be larger for GPU-based architectures. To cover these cases, halo sizes were chosen which range from 3,200 bytes per halo (100 levels * 4 columns * 8 bytes) to 204,800 bytes (100 levels * 256 columns * 8 bytes). Whilst the sizes were primarily designed to analyse the scalability of halo communication for Earth System Models, the halo sizes and performance results are relevant to other disciplines as well.

The problem sizes were chosen to fully populate the nodes in the machines. Multiples of 16 were used on the IBM BGQ and Cray XK7. As ARCHER has 12 cores per node different sizes were required in this case.

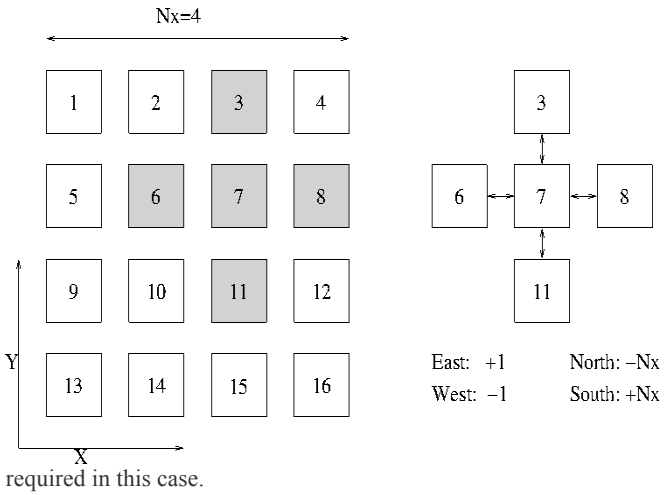



**Fig 1: 5-point stencil on regular grid. Stencil-communication.**

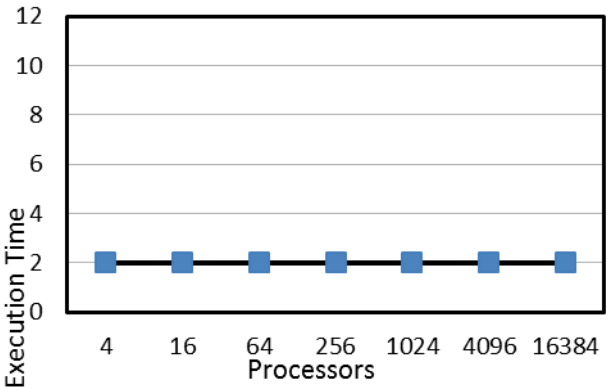



***Fig 2: Execution time in Ideal case for weak scaling.***

## 3. MACHINES

The benchmark was run on three architectures, an IBM Blue Gene/Q a Cray XK7 and a Cray XC30.

The IBM Blue Gene/Q system (Blue Joule) is a primary computational resource at the Hartree Centre, at STFC's Daresbury Laboratory in the U.K. The machine consists of 6 racks, each rack containing 1,024 nodes, and each node has 16-cores, 64 bit, 1.60 GHz A2 PowerPC processor. This equates to a total of 98,304 cores. Each node is equipped with 16 GB of system memory, providing a total of 96 TB of distributed memory across the system. Blue Joule nodes are interconnected by a five-dimensional "torus" network used for general purpose message-passing and multi-cast communication. Special purpose networks are provided for global collective and interrupt operations.

The Cray XK7 system (Titan) is installed at the Oak Ridge National Laboratory (ORNL), U.S.A and has a peak performance of 20 petaflops. contains 18,688 nodes, with each holding a 16-core AMD Opteron 6274 processor and an NVIDIA Tesla K20 GPU accelerator. The Gemini network connects the Titan nodes into a direct 3D torus interconnect network. Gemini uses wormhole flow control internally.

The Cray XC30 system (ARCHER), is the latest UK National Supercomputing Service. There are 4920 compute nodes in the current phase and each compute node has two 12-core Intel Ivy Bridge series processors giving a total of 118,080 processing cores. Each node has a total of 64 GB of memory with a subset of large memory nodes having 128 GB. The Cray Aries interconnect links all compute nodes in a Dragonfly topology. In the Dragonfly topology, 4 compute nodes are connected to each Aries router; 188 nodes are grouped into a cabinet; and two cabinets make up a group. The interconnect consists of 2D all-to-all electric connections between all nodes in a group with groups connected to each other by all-to-all optical connections. The number of optical connections between groups can be varied according to the requirements of the system. ARCHER has 84 optical links per group giving a peak bisection bandwidth of over 11,090 GB/s over the whole system.

## 4. *OBSERVATIONS*

In this analysis, timings are limited to the measurement of the halo communication costs with the time consumed in MPI Init, MPI_Reduce and MPI_Finalize being excluded.

The benchmark code was run on a range of cores whilst keeping the amount of communication per partition the same i.e. performing weak scaling tests, for a range of double integer sizes from 4*100 to 256*100 per core. As the ammount of communication per partition stays constant for weak scaling one would expect the time to remain constant (i.e. be independent of the number of cores) if the benchmark scaled with the number of cores.

The results in Figures 3 and 4 demonstrate that stencil communication does not show weak scaling properties when running on Blue Joule and Titan. **In fact, the performance is up to 26 times worse on Blue Joule and 34 times worse on Titan as the number of cores increases**.

On ARCHER the runs were limited to one size only (256*100) due to limited compute-time availability. Figure 5 also shows

that he **communication time increases by up to 26 times** (from 0.5 to 13 secs) with the number of cores.

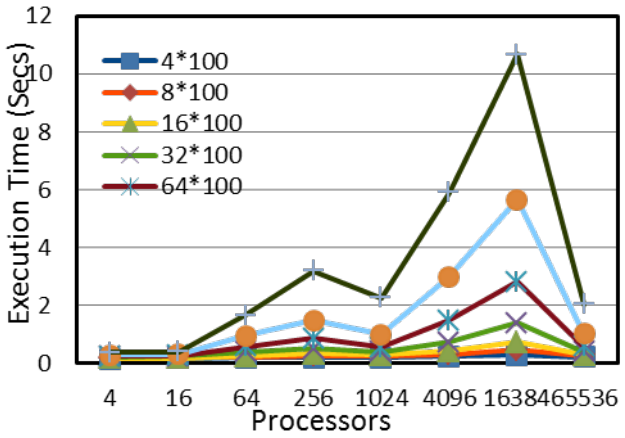



**Fig. 3: Weak scalability on BGQ (for default mapping)**

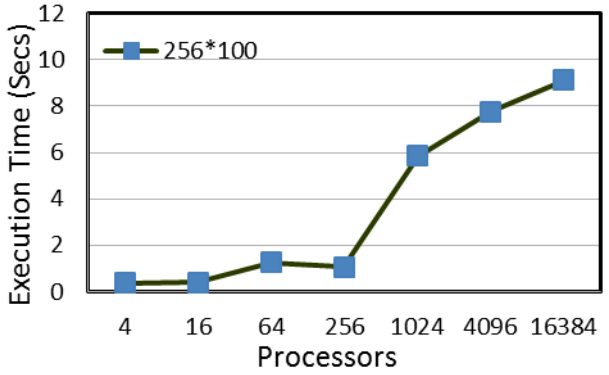



**Fig 4: Weak scalability on Titan (for default mapping)**

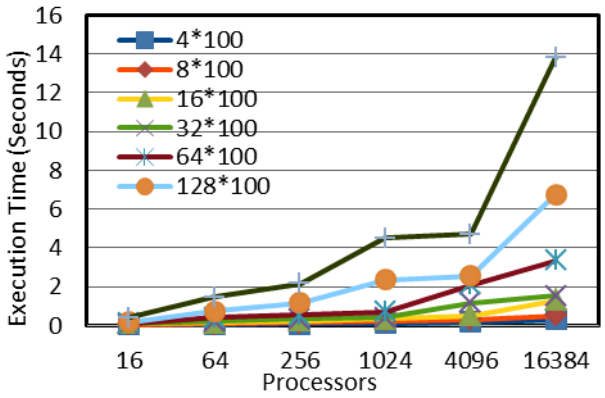



**Fig 5: Weak scalability on ARCHER (for default mapping)**

Codes that use a predictable communication pattern (as in the present case) can reorder the mapping of ranks to nodes to maximize the amount of intra-node communication via the shared memory system, which has a higher bandwidth and lower latency than inter-node communication over the network (*http://www.archer.ac.uk*). This can result in significantly improved communication (and thus application) performance and can be achieved entirely through environment variables and a single additional input file. The IBM Cartesian topology utility and Cray Performance Analysis Toolkit (CrayPAT) helps in generating the optimized rank file for BGQ and Cray systems respectively.

The benchmark code was also run using the utility/tool-generated rank files on Blue Joule and ARCHER respectively for the 256*100 problem size. Results (presented in Figure 6) for Blue Joule show that even with the suggested rank-reordering, the communication cost for weak scaling increases with the number of cores (from 0.4 to 9 secs). On ARCHER (see Figure 7), the communication cost becomes constant for higher core counts, however, the time increases significantly from 16 cores (0.2 secs) to 576 cores (5 secs).

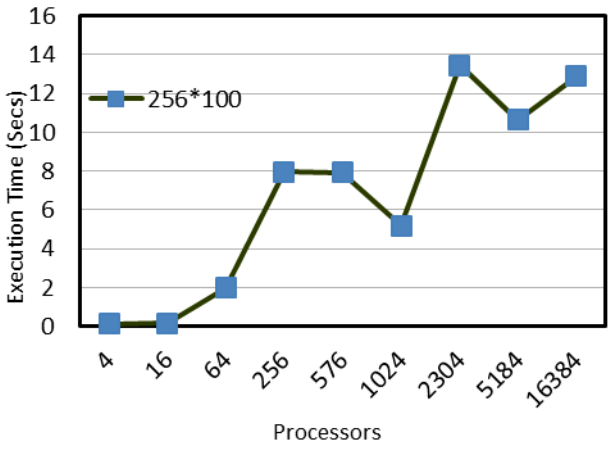



**Fig 6: Weak scalability on Blue Joule with utility e.g. Cartesian topology suggested rank file.**

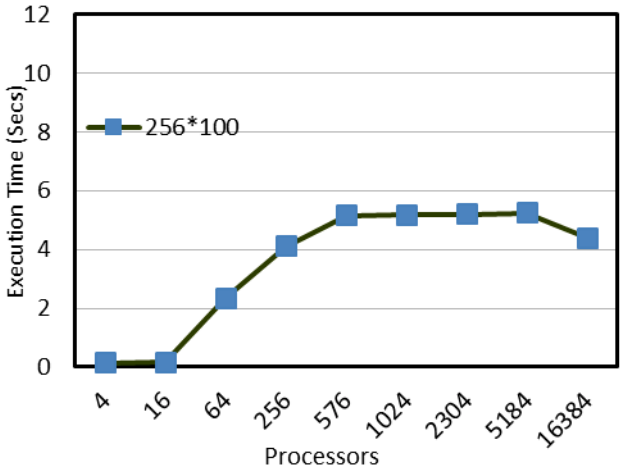



***Fig 7: Weak scalability on ARCHER with Craypet suggested rank file.***

## 5. ANALYSIS

The degradation in performance for the stencil code benchmark, which was observed even when using the supplied mapping utilities, was analyzed on the IBM BGQ.

The communication in the X and Y direction was analysed independently. It was observed that the communication in the X direction scales well. This is because in the majority of the communication is within a node and when communication is between nodes then the messages typically travel a single network hop with no other message traveling on this link. For the periodic boundary points the messages may need to travel more hops and will potential interfere with other communication by sharing links, but those are few.

However, in the Y direction, for the majority of the time the messages are being sent/received between different nodes and may require multiple network hops. In the case of boundary conditions, the neighbour's could be very far away and messages may need to travel many hops on the BGQ Torus network (Appendix-I).

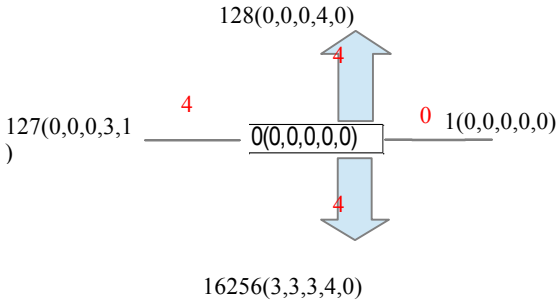



**Fig 8a: Default Rank Layout and hops to travel (in Red).**

As an illustration, the communication for rank 0 and rank 11311 is shown in Fig. 8 (a & b). The figure illustrates the required number of hops to communicate for 128x128 grids point in X & Y directions, 16384 mpi tasks and 4x4x4x8x2 Blue Joule's layout in A,B,C,D & E directions respectively.

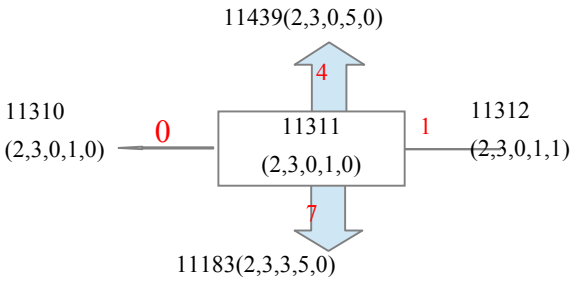



**Fig 8b : Default Rank Layout and hops to travel(in Red).**

Rank 0 (boundary point), in order to communicate with its neighbours, needs to travel a total of 4 hops in the x direction and 8 hops in the y direction respectively. Similarly, for rank 11311, it needs to travel a total of 1 hops in the x direction and 11 hops in the y direction respectively.

# 6. DEVISED MAPPING

Given the above analysis a new task-to-topology mapping scheme was devised. The objective of the devised mapping was twofold: (i) maximize communication within a node and (ii) minimize network contention by reducing the number of hops required by messages. The mapping scheme does not require any change to the code itself, rather it involves the mapping of tasks to cores when the associated job is submitted.

## 6.1 Intra-Node communication:

The MPI ranks are placed in such a way that maximises the communication within a node. This is explained below with an example.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 240 | 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 | 251 | 252 | 253 | 254 | 255 |
| 224 | 225 | 226 | 227 | 228 | 229 | 230 | 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 |
| 208 | 209 | 210 | 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 | 221 | 222 | 223 |
| 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 |
| 176 | 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 |
| 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 |
| 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 |
| 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 |
| 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |
| 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

**Fig9 a: Default Rank Layout. Each colour represent one compute node.**

A BGQ compute node has 16 cores. Hence, a block of 4x4 tasks is made to fit within the node to maximise communication within the node (and therefore minimise the communication between nodes). In the case of a 256 core run (i.e. 16 (EW) x 16 (NS)), there were previously a total 32 communication outside of each node (Fig 9 a & b). However, with the new mapping this reduces to 16.

As next generation systems are expected to have relatively "fat" nodes this approach should continue to be beneficial.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 204 | 205 | 206 | 207 | 220 | 221 | 222 | 223 | 236 | 237 | 238 | 239 | 252 | 253 | 254 | 255 |
| 200 | 201 | 202 | 203 | 216 | 217 | 218 | 219 | 232 | 233 | 234 | 235 | 248 | 249 | 250 | 251 |
| 196 | 197 | 198 | 199 | 212 | 213 | 214 | 215 | 228 | 229 | 230 | 231 | 244 | 245 | 246 | 247 |
| 192 | 193 | 194 | 195 | 208 | 209 | 210 | 211 | 224 | 225 | 226 | 227 | 240 | 241 | 242 | 243 |
| 140 | 141 | 142 | 143 | 156 | 157 | 158 | 159 | 172 | 173 | 174 | 175 | 188 | 189 | 190 | 191 |
| 136 | 137 | 138 | 139 | 152 | 153 | 154 | 155 | 168 | 169 | 170 | 171 | 184 | 185 | 186 | 187 |
| 132 | 133 | 134 | 135 | 148 | 149 | 150 | 151 | 164 | 165 | 166 | 167 | 180 | 181 | 182 | 183 |
| 128 | 129 | 130 | 131 | 144 | 145 | 146 | 147 | 160 | 161 | 162 | 163 | 176 | 177 | 178 | 179 |
| 76 | 77 | 78 | 79 | 92 | 93 | 94 | 95 | 108 | 109 | 110 | 111 | 124 | 125 | 126 | 127 |
| 72 | 73 | 74 | 75 | 88 | 89 | 90 | 91 | 104 | 105 | 106 | 107 | 120 | 121 | 122 | 123 |
| 68 | 69 | 70 | 71 | 84 | 85 | 86 | 87 | 100 | 101 | 102 | 103 | 116 | 117 | 118 | 119 |
| 64 | 65 | 66 | 67 | 80 | 81 | 82 | 83 | 96 | 97 | 98 | 99 | 112 | 113 | 114 | 115 |
| 12 | 13 | 14 | 15 | 28 | 29 | 30 | 31 | 44 | 45 | 46 | 47 | 60 | 61 | 62 | 63 |
| 8 | 9 | 10 | 11 | 24 | 25 | 26 | 27 | 40 | 41 | 42 | 43 | 56 | 57 | 58 | 59 |
| 4 | 5 | 6 | 7 | 20 | 21 | 22 | 23 | 36 | 37 | 38 | 39 | 52 | 53 | 54 | 55 |
| 0 | 1 | 2 | 3 | 16 | 17 | 18 | 19 | 32 | 33 | 34 | 35 | 48 | 49 | 50 | 51 |

**Fig 9b : Proposed Layout. Each colour ~~represent~~ one compute node.**

## 6.2 Reduction in Network Contention:

It is possible to reorder ranks such that communication from any rank to its neighbour will travel a maximum of one torus hop. On the IBM BGQ a hop is one portion of the path between the communication source and destination. Data packets pass through bridges, routers and gateways on the way to their destination. Each time packets are passed to the next device, a hop occurs.

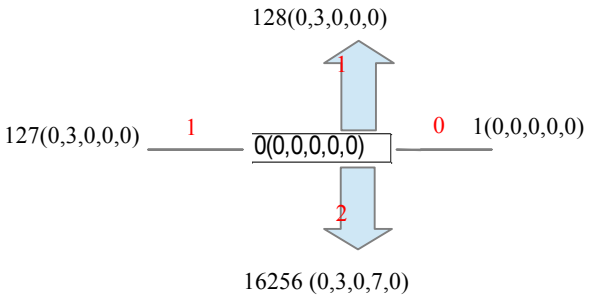



**Fig10a : Proposed Rank Layout and hops to travel(in Red).**

The IBM BGQ Torus network allowed us to devise a general 2d stencil mapping algorithm which limits communication to a single hop including periodic boundary conditions. As each communication is only 1 hop, only communication from the source node to the target node uses the specific link associated with the hop. Therefore there is no contention between messages from different nodes (or the same node going to a different destination). As shown in Figure 10 (a & b), for a 128x128 grid size (16384 mpi tasks with a 4x4x4x8x2 layout), the maximum communication cost in each dimension is 1 hop for Rank 0 and rank 11311.

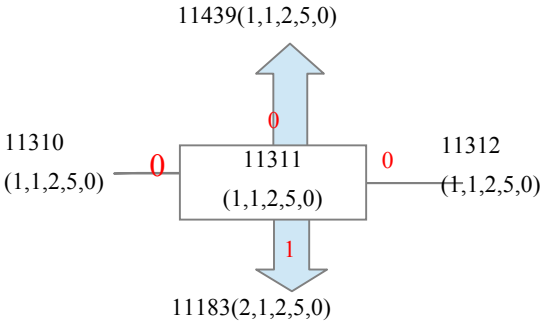



**Fig10 b : Proposed Rank Layout and hops to travel(in Red).**

The results obtained for the devised mapping are show in Figure 11. The figure shows that, for all of the problem sizes, the communication cost remains constants with the number of cores.

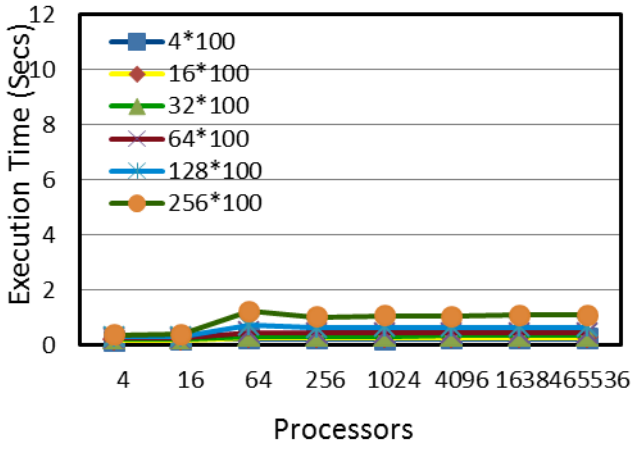



**Fig 11: Weak scalability on BGQ (with devised mapping).**

## 7. ENERGY MEASUREMENT & USE

The energy costs of the benchmark were also captured. To perform this analysis the benchmark needed to be run for a longer duration. Therefore, in this analysis the halo communication is repeated 12,000 times (6 times more then previous runs). The results are shown here for the halo size 256*100 and the benchmark was run on 16384 cores (1 rack, i.e. 16 nodes boards) of the IBM BGQ. The results reported from the node board (including rank 0) were captured for the default and optimised mapping cases.

The power profile for the default mapping is shown in Figure 12a. Execution time for this job was approximately 229 seconds and the average power was 1,750 Watts for the node board. This implies that the total energy consumed by the job is 16*1,750*229/1000 = 6412 KiloJoules (kJ).

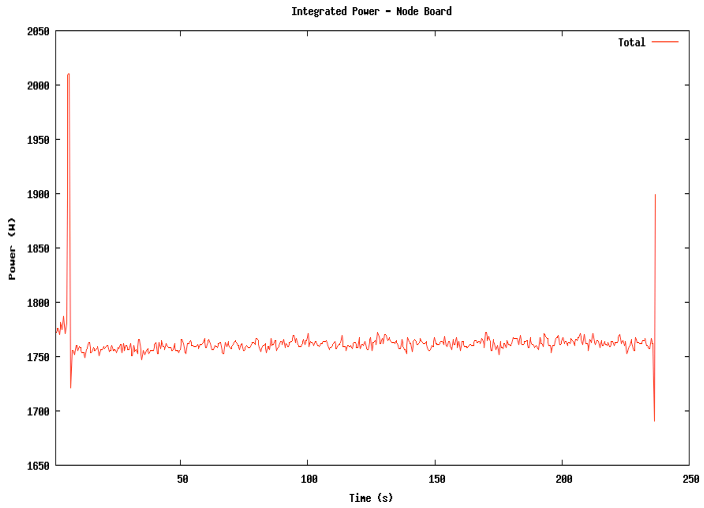



**Fig 12a: Energy consumption for default mapping.**

The power profile for the optimized run is shown in Figure 12b. The job completed in 84 secs. No differences in power consumption were observed for the two different mappings. Therefore the total energy consumption was 16*1,750*84=2352 kJ, a 64% improvement.

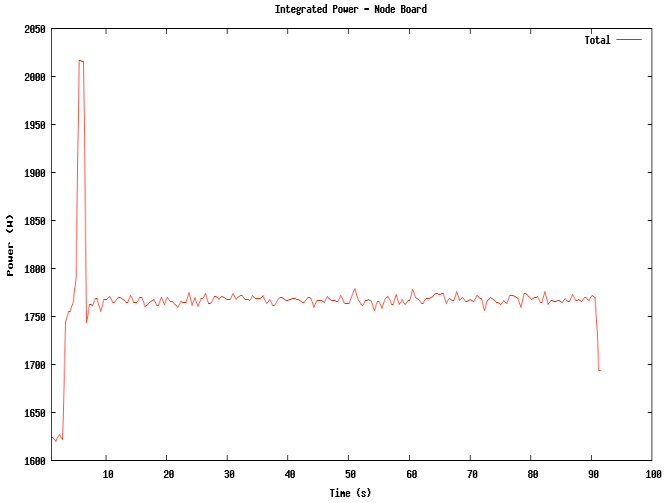



**Fig 12b: Energy consumption for devised mapping.**

The average power distribution not varies between the default and proposed mapping. Hence from this data the default mapping does not increase the power required by a node-board. A power increase might have been expected due to e.g. extra operations required to route packages through intermediate nodes, but this does not appear to be the case. The 64% energy saving purely comes from the proposed mapping taking less time than the default.

# 8. CONCLUSIONS

Weak scaling analysis was performed for a 2D halo benchmark on Blue Joule (up to 65K cores), Titan (up to 16K cores) and ARCHER (up to 16K cores) for a range of message sizes (3200 to 204800). It was observed that the communication time increases with number of cores for both the default and system-suggested (Cartesian topology & Craypat) mapping tools.

The proposed task-to-toppology mapping scheme maximizes the intra-node communication and removes network contention by reducing the number of hops to be traveled to communicate with neighbours. The results obtained with the devised mapping shows that communication costs remain constant with the number of cores for all the message sizes on Blue Joule. It is also shown that the energy consumption with the proposed mapping is reduced by 64%.

The Blue Gene Q (Cartesian topology) mapping tool is being updated to include this new mapping scheme.

# 8. ACKNOWLEDGEMENTS

The authors acknowledge the support provided by the Hartree Centre, ORNL and the ARCHER Team in allowing us to use their HPC systems.

# 9. REFERENCES

Yongpeng Zhang, Frank Mueller, “Auto-Generation and Auto-Tuning of 3D Stencil Codes on GPU Clusters” in International Symposium on Code Generation and Optimization (CGO'2012), San Jose, CA, Apr 2012.

Manish Modani and Andrew Porter (2015) “I/O at Scale in the Weather Research & Forecast Model”, Advances in Engineering Software (In Press).

http://www.archer.ac.uk

https://www.ornl.gov/

http://www.stfc.ac.uk/2512.aspx

# APPENDIX

## 5 Dimensional Torus for BGQ:

The Blue Gene/Q has a five-dimensional torus, or mesh network topology—for partitions smaller than the 512 node, the torus degenerates to a mesh, but the concepts discussed here remain the same—with direct links between the nearest neighbors in the ±A, ±B, ±C, ±D, and ±E directions. Note that the E dimension is always 2. Each node is therefore directly connected to nine neighbors.

The graphical representation for a node board of 32 compute nodes (e.g. 2x2x2x2x2) of BGQ is shown in Fig.13. The Torus dimensions for node# 0, 17, 20, 28, 29 and 31 are given as (0,0,0,0,0), (1,0,0,1,0),(1,1,1,1,1), (1,0,1,0,0), (1,0,0,0,0) and (1,1,1,0,0) respectively. Therefore, if node#0 needs to communicate with node#17, the message need to travel 2 hops (e.g. 1 hop in A and 1hop in D direction respectively). Similarly if node#0 sends message to node#20, the message needs to travel 5 hops (1 hop in each dimension e.g. A,B,C,D & E).

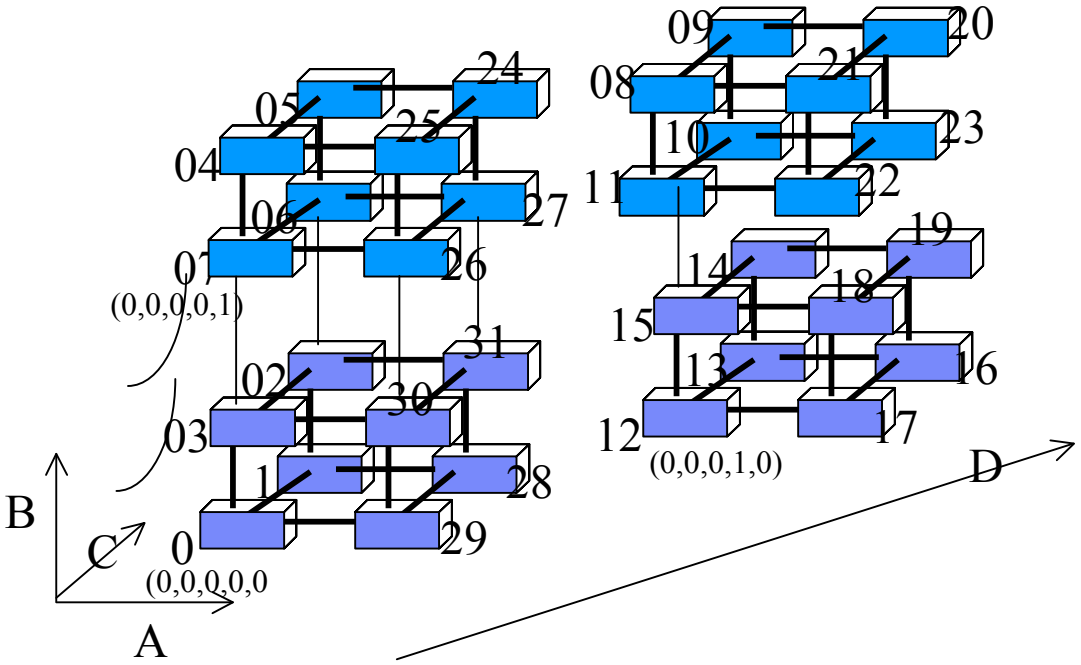



**Fig 13: BGQ 5D Torus for the node board of 32 computed nodes (e.g. 2x2x2x2x2). The dimensions for node# 17, 20, 28, 29 and 31 are given as (1,0,0,1,0),(1,1,1,1,1), (1,0,1,0,0), (1,0,0,0,0) and (1,1,1,0,0) respectively.**

# Exascale Computing for Everyone: Cloud-based, Distributed and Heterogeneous

Gordon Inggs and David B. Thomas
Circuits and Systems Group
Department of Electrical and Electronic Engineering
Imperial College London
London
United Kingdom
{g.inggs11,d.thomas1}@imperial.ac.uk

Eddie Hung and Wayne Luk
Custom Computing Group
Department of Computing
Imperial College London
London
United Kingdom
{e.hung,w.luk}@imperial.ac.uk

## ABSTRACT

Widespread adoption of exascale computing will most likely be heterogeneous and distributed, provided by Infrastructure-as-a-Service (IaaS) providers. This is because heterogeneous architectures, such as Graphic Processing Units (GPUs) and custom accelerators implemented in Field Programmable Gate Arrays (FPGAs), provide orders of magnitude greater performance than multicore CPUs. The attraction of IaaS is that it offers users a pay-per-use model while still taking advantage of economies of scale. There are however challenges in taking advantage of the heterogeneous and IaaS computing trends. Firstly, computational tasks need to be expressed in a form that can be executed on these IaaS-based, heterogeneous architectures. Secondly, these tasks need to be partitioned across the available resources so as to take advantage of the relative strengths of different architectures. A domain specific approach provides solutions to these problems by providing a portable, efficient execution framework, predictive performance modelling and automatic partitioning across platforms. An experiment using 16 multicore CPU, GPU and FPGA platforms to evaluate a large workload of 128 Option Pricing tasks found that a domain specific approach to partitioning, based upon domain knowledge performance models, outperforms naive heuristic approaches by two orders of magnitude. These results suggest that domain specificity provides a clear path for anyone, and everyone, to access exascale computing.

## Keywords

Exascale, Distributed, Cloud, IaaS, Heterogeneous Computing, Domain Specific

## 1. INTRODUCTION

We argue that widespread adoption of exascale computing will be in the form of distributed computing resources provided on a utility basis i.e. by a Infrastructure-as-a-Service Computing provider. Furthermore, these resources will be heterogeneous, comprised of not only Von Neumann machine CPUs, but also massively parallel compute devices such as Graphics Processing Units (GPUs) and accelerators implemented using reconfigurable computing devices such as large Field Programmable Gate Arrays (FPGAs). A further consideration is the growing range of hybrid devices that incorporate these elements within a single device such as Intel's Xeon Phi and Xilinx's Zynq system-on-chips.

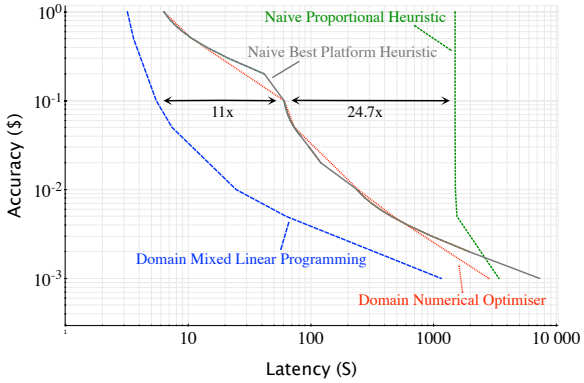



**Figure 1: Comparison of different methods for allocating tasks to distributed, heterogeneous computing platforms. We introduce the domain-based approaches.**

As of January 2015, a hypothetical exascale system would cost $\approx 82\ \$/s$ in either the Amazon Web Services (AWS) [14] or the Google Compute Engine IaaS offerings. An IaaS system is attractive as it allows users to share the total cost of ownership while also taking advantage of economies of scale. However, the millions of CPUs required is far beyond that which could be made available to a single user currently. If however Cloud-based GPUs are considered, currently only hundreds of thousands of devices would be required for exascale.

A further challenge is that partitioning the workload upon millions of CPU would be a considerable computational problem in its own right. However considering GPUs, hundreds of thousands of devices would be required, making the partitioning problem more tractable.

The use of IaaS FPGAs would also be advantageous as these devices typically provide throughput acceleration comparable to, and often exceeding high-end GPUs as well as order of magnitudes energy saving over CPUs and GPUs. Although the use of IaaS-based FPGAs is nascent, Microsoft has recently had success accelerating cloud-based applications such as their Bing search engine [12], while vendors such as Maxeler now provide publicly accessible cloud FPGA

offerings. Increasingly these reconfigurable devices also support the same level of design abstraction, supporting languages such as OpenCL, C and C++, making them usable by high performance software engineers with limited embedded hardware design expertise.

We claim that domain specific abstractions can realise the opportunities of the heterogeneous IaaS by providing three features: *Portability* - the capability to execute a single task description upon a wide range of platforms efficiently. *Predication* - characterisation models of the domain specific computation outputs that allow for the relationship between a particular task and platform to be quantified. *Partitioning* - the structure of the application domain can unify the predictive models, allowing for various techniques to be applied so as to produce workload partitions.

To illustrate our claims, in this paper we have applied this approach to the demanding domain of computational finance. We show how the fundamental concepts within the domain can be captured as types and operations, which can be executed upon a range of heterogeneous platforms. We also describe the financial domain metrics of latency and accuracy and show how they can be modelled. Finally, we apply domain knowledge to partitioning, showing that the domain metric models can be used to formulate an optimisation problem that can be solved for optimal partitions.

Our initial experiments on evaluating our domain specific approach upon heterogeneous cluster support this claim. Our cluster was comprised of 16 CPU, GPU and FPGA computing platforms, 6 of which were located in the AWS cloud. Using our open source application framework for financial option pricing, the Forward Financial Framework[1], we have validated all three properties for a PetaFLOP scale workload. The implementations automatically generated by our framework deliver comparable performance to state-of-art, platform-specific ones. The predictive domain latency and pricing accuracy models are within 10% of the true values in the worst case. Finally, the workload partitioning achieved using the Mixed Integer Linear Programming (MILP) approach shows a 100% or greater improvement over a heuristic-based approach to workload partitioning.

# 2. BACKGROUND

## 2.1 Domain Specific Heterogeneous Computing

An important finding in recent years is that domain specific abstractions enable improved performance in the heterogeneous computing context [3, 8]. As alluded to in the introduction, empirical studies of software engineering [11] have found that a small set of design patterns within an application domain are executed disproportionately more frequently than others, often following a Power Law distribution. Indeed, application domains are often identified by grouping these patterns together [13]. By supporting the efficient, heterogeneous acceleration of these disproportionately influentially patterns, significant gains can be realised automatically for programs restricted to a particular domain.

[1] `https://github.com/Gordonei/ForwardFinancialFramework`

We call this property portable performance. Previous works have shown this portable performance property in practice through Domain Specific Languages (DSL), as shown by Chafi et al [3], or application frameworks, as per our own previous work [8]. However putting this approach into practice remains a challenge, requiring system developers with domain expertise to create domain specific abstractions [8] that support heterogeneous execution. Chafi et al's [3] approach advocates the use of language virtualisation, providing both a framework for creating implicitly parallel domain specific languages as well as a runtime for supporting applications created using such languages.

## 2.2 Partitioning across Distributed Computing

When considering the allocation of tasks to heterogeneous computing resources, the general scenario considered in the literature, i.e. [2, 10, 5, 4] is a set of independent or atomic tasks being partitioned across multiple heterogeneous platforms. It is assumed that a task will occupy any of the computing resource completely if allocated to that resource. It is also commonly assumed that the partitioning is being performed statically, in advance of the execution of any of the tasks.

In this scenario, the general objective is to minimise the makespan. The makespan is the latency between when the first task is initiated until the last result returned for the task set. As the tasks are being evaluated on multiple platforms, the makespan is equivalent to the longest time it takes for any of the platforms to return the results of the tasks allocated to it.

Surveying the literature, there are three suggested approaches to the partitioning problem:

*Naive Heuristics* [2, 7]: a simple algorithmic rule is applied to allocate tasks to the available resources. Under specified circumstances such a rule might achieve a provably optimal allocation of tasks, and there is usually a worst case bound on the quality of the solution relative to the optimal solution.

*Numerical Optimisation* [5, 10]: a feasible task-platform allocation is improved using numerical optimisation techniques such as the Simplex algorithm, simulated annealing or genetic algorithms using predictions of the task performance for a given allocation.

*Integer Linear Programming* [4]: the optimisation problem formulated above can be solved using integer linear programming techniques, which in addition to applying the numerical optimisation above use a dual formulation of the problem to prove the optimality of the solution.

# 3. DOMAIN SPECIFIC APPROACH

We now describe the domain specific approach as applied to the real world application domain of derivatives pricing, within the larger subject area of computational finance. Computational finance is concerned with the determination and management of risk, an attempt to quantify the inherent operational unknowns in an uncertain world.

Derivatives pricing is an important activity in modern commerce, with the volume of products currently being traded

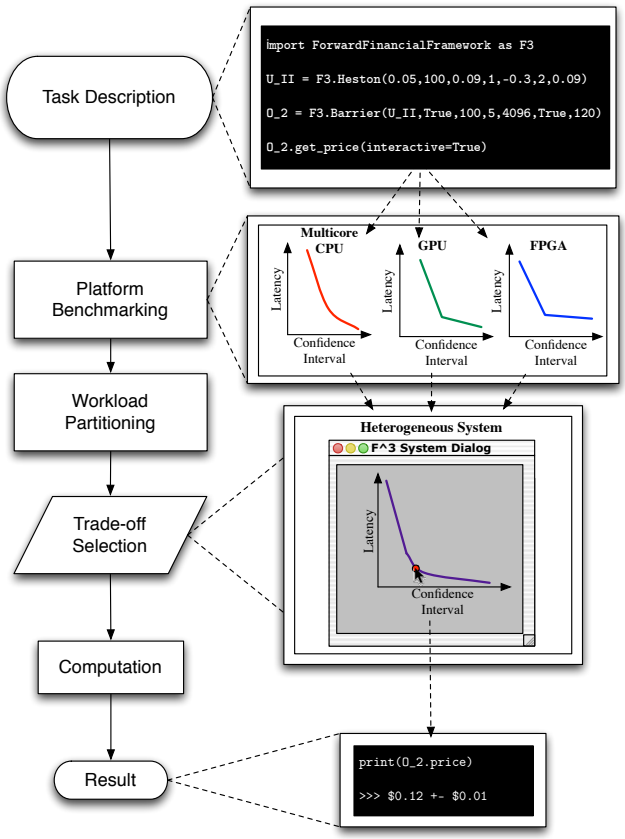



**Figure 2: Domain Specific Approach to distributed, heterogeneous computing. Illustrated using the example of the domain of derivatives pricing and the Forward Financial Framework($F^3$).**

amounting to trillions of dollars. The financial modelling techniques used are extremely computational intensive, and as a result banks are a major consumer of high performance computing. The use of clusters of heterogeneous computing technologies such as multicore CPUs and GPUs is widespread.

## 3.1 Portable Execution

### *3.1.1 Domain Types and Operations*

Each derivative pricing task can be subdivided into two components, the derivative contract or product, such as an option, which is being valued and the underlying asset from which that derivative derives its value [6]. The underlying asset encapsulates the probabilistic model, such as the Black-Scholes or Heston, being used to model the behaviour of the asset under consideration, for example a stock or commodity price. The derivative product embodies the details of the option contract both during the lifetime of the option as well at its expiration.

The pricing domain's sole operation is finding the price of a domain entity. Only applying the price operation to derivative types is of interest, as by definition underlyings can provide their price at any point in time. Multiple, varied techniques such as Monte Carlo or Tree-based methods could be used to implement the pricing operation, provided the end result is the price of the derivative product under consideration.

The popular Monte Carlo technique for option pricing uses random numbers to create potential scenarios or paths for the underlying asset based upon a model of its price evolution. The average outcome of these paths is then used to approximate the most probable option value.

### *3.1.2 Implementation*

We have created the Forward Financial Framework ($F^3$), an open source financial domain application framework to support the portable execution of Monte Carlo pricing upon multicore CPUs, GPUs and FPGAs. At the high level, $F^3$ allows for option pricing tasks to be described using a library of Python objects. This single task description can then be executed automatically by the framework upon a variety of backends: GCC and POSIX threads for multicore CPUs, OpenCL for GPUs, Xeon Phis and Altera FPGAs and the Maxeler tools for FPGAs. Previous work [8, 9] has described the framework in greater detail, as well as demonstrating that its automatically generated implementations are close to or better performing than those created by device experts.

## 3.2 Prediction

We have developed models for the domain metrics of latency and price accuracy for the pricing operation in our domain, as implemented using the Monte Carlo algorithm in $F^3$.

*Latency Model*: the latency between when a pricing operation is initiated and when it returns a price is fundamentally important within the financial domain [6]. This is because the time at which prices or information are available to market actors fundamentally affects the market as a result of behaviour in response to this information. Minimising the latency of the pricing operation is often desirable, as this confers first-mover advantage to pricer.

We have used a linear latency model:

$$L(n) = \beta n + \gamma \tag{1}$$

Where the coefficient of the number of paths ($n$), is $\beta$. This represents the time spent per Monte Carlo simulation. $\gamma$ represents the constant component of the task latency. This would capture the time spent initialising the computation within the operating system of the target platform, as well as any time spent communicating the task to and returning the result from the platform.

*Accuracy Model*: in the financial domain, the accuracy of a computed price is expressed in probabilistic terms. When using the Monte Carlo technique, often the 95% confidence interval is used, which gives the size of the finite interval around the computed price for which there is a 95% confidence that the true value lies within that interval. As small a confidence interval as possible is desired, as this means for the given confidence level the pricing result is close to the true value, and hence less risk has to be accounted for.

The accuracy model that we used is based upon the convergence of the Monte Carlo algorithm, which is given by the

inverse square root of the number of simulations, i.e. $\frac{1}{\sqrt{n}}$, scaled by a coefficient ($\alpha$).

The accuracy model is:

$$C(n) = \frac{\alpha}{\sqrt{n}} \tag{2}$$

*Combined Model*: to relate the two domain metrics of latency and accuracy, we can then solve for number of paths parameter, $n$, and use it to relate Equations 1 and 2, i.e.

$$\begin{aligned} L(C) &= \frac{\delta}{C^2} + \gamma \\ \text{where} \quad \delta &= \beta\alpha^2 \end{aligned} \tag{3}$$

## 3.3 Partitioning

### 3.3.1 Formulating the Partitioning Problem

In Equation 4 the combined model described in Equation 3 has been applied in a constrained integer linear program to minimise the makespan of $\tau$ Tasks implemented upon $\mu$ Platforms. The optimisation is performed over a binary matrix, $\boldsymbol{A}$, where $A_{i,j} = 1$ represents an allocation of task $j$ to platform $i$. The vector $\boldsymbol{C}$ gives the required accuracy for each task, while $\boldsymbol{\delta}$ and $\boldsymbol{\gamma}$ are matrices that provide the proportional and setup components for each task upon each platform. Hence, the element-wise division and addition operation, $\boldsymbol{\delta} : \boldsymbol{C}^2 + \boldsymbol{\gamma}$, captures the combined model in Equation 3

$$\begin{aligned} \underset{\boldsymbol{A}\in\{0,1\}^{\mu\times\tau}}{\text{minimise}} \quad & \boldsymbol{G_L}(\boldsymbol{A},\boldsymbol{C}) \quad \boldsymbol{C} \in \mathbb{R}_+^{\tau} \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1, 2, \ldots, \tau \end{aligned} \tag{4}$$

where:

$$\begin{aligned} \boldsymbol{G_L}(\boldsymbol{A},\boldsymbol{C}) &= \max(\boldsymbol{F_L}(\boldsymbol{A},\boldsymbol{C})) \\ \boldsymbol{F_L}(\boldsymbol{A},\boldsymbol{C}) &= ((\boldsymbol{\delta} : \boldsymbol{C}^2 + \boldsymbol{\gamma}) \circ \boldsymbol{A}) \cdot \mathbf{1} \\ & \boldsymbol{\delta} \in \mathbb{R}_+^{\mu\times\tau}, \boldsymbol{\gamma} \in \mathbb{R}_+^{\mu\times\tau} \end{aligned}$$

Two reduction functions, $\boldsymbol{G_L}$ and $\boldsymbol{F_L}$, are defined. $\boldsymbol{F_L}$, the task reduction function, reduces all of the tasks latencies to a vector of length $\mu$, where each entry corresponds to the total latency for that platform for the given allocation. $\boldsymbol{G_L}$, the platform reduction function, reduces all of the platform latencies to a single scalar value, in this instance the makespan.

We have investigated three approaches to partitioning work automatically, one from each of the earlier identified categories.

### 3.3.2 Heuristic Allocations

The first allocation approach we propose is intuitive: The best platform heuristic allocates all of the tasks to the single platform that completes all the tasks with the shortest latency. The second instance of this approach, the proportional allocation heuristic as given in equation 5, is a minor refinement of this, allocating work in proportion to the total latencies, $L$, of all of the platforms.

$$\vec{A} = \left[ \frac{\frac{1}{\vec{L}_i}}{\sum_{o=0}^{\mu} \frac{1}{\vec{L}_o}} \right] \quad i = 1, 2, \ldots, \mu \tag{5}$$

This approach only requires an estimate of the total latency of all tasks upon each platform. As is to be expected, the best platform heuristic performs well when there is a single platform within the set that can complete the tasks significantly faster than the others available. The proportional allocation heuristic is more general, working well when the magnitude of $\boldsymbol{\gamma}$ is significantly smaller than the magnitude of $\boldsymbol{\delta} : \boldsymbol{C}^2$.

### 3.3.3 Numerical Optimiser-based Allocation

The second approach builds upon the first, using either the best platform or proportional allocation heuristics as a starting, however it also uses the metric model-predicted setup latency($\boldsymbol{\gamma}$) and proportional latency ($\boldsymbol{\delta} : \boldsymbol{C}^2$) information of each task upon each platform. $G_L$ is specified as the objective function for a simulated annealing optimisation algorithm with a final "polishing" step performed using the simplex algorithm. By using the task-platform information, this approach should improve upon the intutive heuristics described above.

### 3.3.4 Integer Linear Programming-based Allocation

Similar to the Numerical Optimiser approach, the ILP approach uses the formulation of the domain partitioning problem, as given in 3.3.1, as the input to a formal linear programming framework, SCIP [1]. SCIP applies various optimisation techniques as well as a variety of heuristic approaches to solve this constrained program.

# 4. EVALUATION

## 4.1 Experimental Setup

An overview of the heterogeneous platforms that we used in our evaluation are described in Table 1. The first two dimensions of platform heterogeneity is device type and vendor - we have made use of a wide array of multicore CPUs, GPU and FPGA-based computational platforms from many different manufacturers. The final dimension is the diversity of interconnections used between the computational platforms.

The pertinent computational characteristics of the platforms are also described in Table 1. We describe the compute capabilities of the experimental platforms using an option pricing benchmark[2] and $F^3$'s implementations. As the Monte Carlo algorithm being used is amenable to parallel execution, it is unsurprising that GPUs provide the best application performance. We have also provided the network latency for each platform. We expect the former to dominate the proportional coefficient of the latency models, $\beta$, while the latter will determine the constant coefficient, $\gamma$. Not reflected is energy consumption, for which the FPGA platforms are notably more than an order of magnitude more efficient than the others.

[2] `http://www.uni-kl.de/en/benchmarking/option-pricing/`

**Table 1: Overview of Experimental Heterogeneous Computing Platforms**

| Device Category | Device Designation | Device | Network Location | Application Performance (GFLOPS) | Network RTT (mS) |
|---|---|---|---|---|---|
| CPUs | Desktop | Intel ® Core ® i7-2600 | Localhost | 5.916 | 0.024 |
| | Local Server | AMD ® Opteron ® 6272 | LAN | 27.002 | 0.380 |
| | Local Pi | ARM ® 11 76JZF-S | LAN | 0.049 | 2.463 |
| | Remote Server | Intel ® Xeon ® E5-2680 | WAN | 11.523 | 3300.000 |
| | AWS Server EC1 | Intel ® Xeon ® E5-2680 | WAN | 12.269 | 88.859 |
| | AWS Server EC2 | Intel ® Xeon ® E5-2670 | WAN | 4.913 | 88.216 |
| | AWS Server WC1 | Intel ® Xeon ® E5-2680 | WAN | 12.200 | 157.100 |
| | AWS Server WC2 | Intel ® Xeon ® E5-2670 | WAN | 4.926 | 159.578 |
| | GCE Server | Intel ® Xeon ® | WAN | 6.022 | 111.232 |
| GPUs | Local GPU 1 | AMD® FirePro ® W5000 | LAN | 212.798 | 0.269 |
| | Local GPU 2 | Nvidia® Quardo ® K4000 | LAN | 250.027 | 0.278 |
| | Remote Phi | Intel® Xeon Phi ® 3120P | WAN | 70.850 | 3300.000 |
| | AWS GPU EC | Nvidia® Grid ® GK104 | WAN | 441.274 | 88.216 |
| | AWS GPU WC | Nvidia® Grid ® GK104 | WAN | 406.230 | 159.578 |
| FPGAs | Local FPGA 1 | Xilinx® Virtex ® 6 475T | LAN | 114.590 | 0.217 |
| | Local FPGA 2 | Altera® Stratix ® V D5 | LAN | 161.074 | 0.299 |

**Figure 3: Relative error of latency models for a fixed runtime target and varying benchmark time**

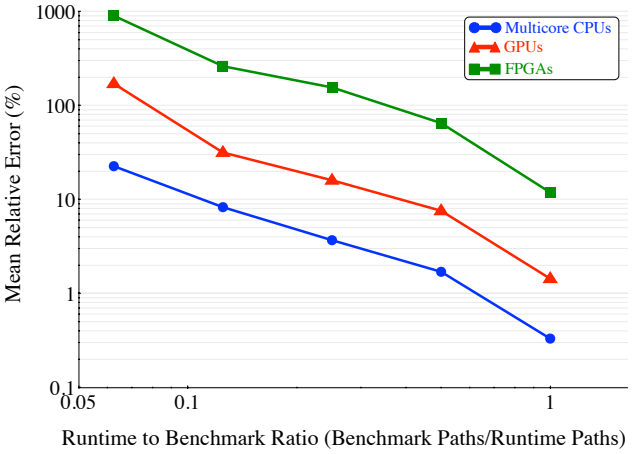
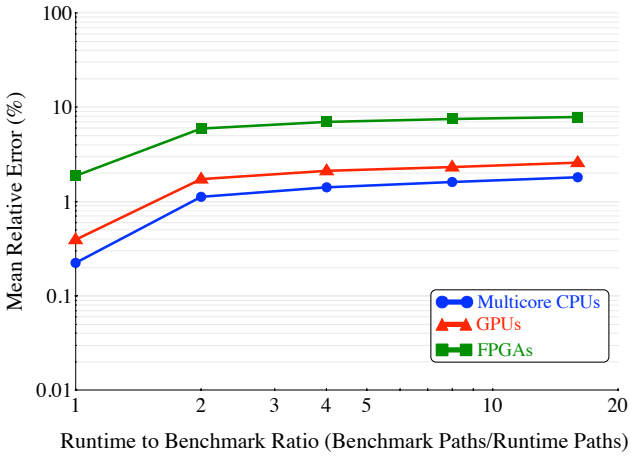



**Figure 4: Relative error of latency models for a fixed benchmark time and varying runtime targets**




We used a workload of 128 option pricing tasks, with European, Barrier, Double Barrier, Digital Double Barrier option varieties, as well as Heston and Black-Scholes model-based underlyings. The fixed parameters, such as the proprieties of underlying model, were generated using uniform random numbers within the values from the aforementioned option pricing benchmark[2]. A rejection procedure was utilised to keep the relative magnitude of the pricing tasks similar.

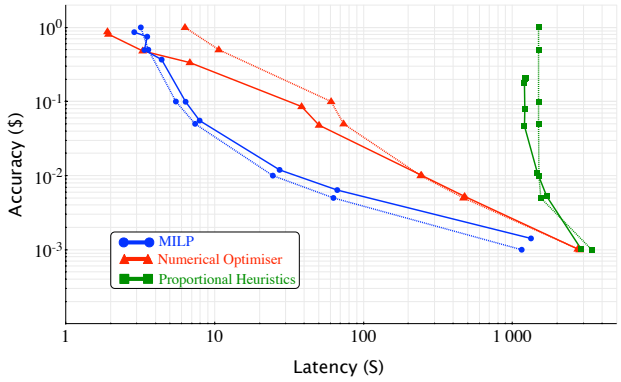
## 4.2 Results

*Domain Model*: The latency model results are given in Figures 3 and 4. The latency models are evaluated using the geometric mean of the relative error of the platforms in the three device categories.

Figure 3 illustrates that as a longer benchmarking procedure is performed relative to the fixed runtime of 4.69 seconds per task (10 minutes for all of the tasks) being predicted by the model, the models became more accurate. This suggests that the models benefit from additional information, which suggests the task-device dynamic is being captured.

Figure 4 shows how the models scale as the runtime prediction target is increased for a fixed benchmarking time (again, 600 seconds in total or 10 minutes for all of the tasks). The results demonstrate that for a runtime target of more than an order of magnitude greater than the benchmarking procedure, the latency models scale well, with only a modest increase in error for problem runtimes an order of magnitude greater than the benchmarking time.

**Figure 5: Resulting pareto curves from different partitioning approaches. Dotted line is the output of the domain partitioners, while the solid line is the actual performance**




*Domain Partitioner*: Figure 5 illustrates our evaluation of how different partitioning approaches based upon the model data match what is achieved in reality closely. Furthermore, both the numerical optimiser and MILP-based partitioner are orders of magnitude more efficient than that suggested by the naive proportional heuristic for much of the accuracy range analysed.

## 5. CONCLUSION

Fundamentally, we view the diverse performance seen in heterogeneous computing as a design feature, as opposed to a challenge that must be overcome. We have sought a means to take advantage of the relative strengths of each platform in a complimentary fashion.

In this paper we have described and demonstrated in practice that a domain specific approach to heterogeneous computing offers portability, prediction and partitioning. We assert that these three features are necessary for the distributed, heterogeneous exascale computing systems of the future. We used a case study from computational finance to illustrate how these three features features are supported. Using the same case study, we verified these features experimentally, using a large workload of option pricing tasks upon a cluster of heterogeneous computing resources.

## Acknowledgements

We are grateful for the funding support from the South African National Research Foundation and Oppenheimer Memorial Trust. We are also grateful for the support from the Nallatech, Altera, Xilinx, Intel and Maxeler university programs.

## 6. REFERENCES

[1] T. Achterberg. Scip: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] T. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Muthucumaru, A. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *J. Parallel Distrib. Comput.*, 61(6):810–837, June 2001.

[3] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A Domain-Specific Approach to Heterogeneous Parallelism. In *PPOPP*, pages 35–46, 2011.

[4] N. Fisher, J. Anderson, and S. Baruah. Task Partitioning upon Memory-Constrained Multiprocessors. *11th IEEE Int. Conf. Embed. Real-Time Comput. Syst. Appl.*, pages 416–421, 2005.

[5] D. Grewe and M. F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using OpenCL. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 6601 LNCS, pages 286–305, 2011.

[6] J. C. Hull. *Options, Futures and Other Derivatives*. Pearson, 8th edition edition, 2011.

[7] O. H. Ibarra and C. E. Kim. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *J. ACM*, 24(2):280–289, Apr. 1977.

[8] G. Inggs, D. Thomas, and W. Luk. A heterogeneous computing framework for computational finance. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*, pages 688–697, 2013.

[9] G. Inggs, D. Thomas, and W. Luk. A Domain Specific Approach to Heterogeneous Computing: From Availability to Accessibility. In *Proc. First Int. Work. FPGAs Softw. Program. (FSP 2014)*, Aug. 2014.

[10] Q. Kang, H. He, and H. Song. Task assignment in heterogeneous computing systems using an effective iterated greedy algorithm. *J. Syst. Softw.*, 84(6):985–992, June 2011.

[11] B. A. Nardi. *A small matter of programming: perspectives on end user computing*. MIT press, 1993.

[12] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of International Symposium on Computer Architecture*, pages 13–24, 2014.

[13] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages. *ACM SIGPLAN Not.*, 35(6):26–36, June 2000.

[14] J. A. Varia and S. A. Mathew. Overview of AWS. Technical report, Amazon, 2014.

# Flexible, Scalable Mesh and Data Management using PETSc DMPlex

Michael Lange
Department of Earth Science and Engineering
Imperial College London, UK
m.lange@imperial.ac.uk

Matthew G. Knepley
Computation Institute
University of Chicago, USA
knepley@gmail.com

Gerard J. Gorman
Department of Earth Science and Engineering
Imperial College London, UK
g.gorman@imperial.ac.uk

## ABSTRACT

Designing a scientific software stack to meet the needs of the next-generation of mesh-based simulation demands, not only scalable and efficient mesh and data management on a wide range of platforms, but also an abstraction layer that makes it useful for a wide range of application codes. Common utility tasks, such as file I/O, mesh distribution, and work partitioning, should be delegated to external libraries in order to promote code re-use, extensibility and software interoperability. In this paper we demonstrate the use of PETSc's DMPlex data management API to perform mesh input and domain partitioning in Fluidity, a large scale CFD application. We demonstrate that raising the level of abstraction adds new functionality to the application code, such as support for additional mesh file formats and mesh reordering, while improving simlutation startup cost through more efficient mesh distribution. Moreover, the separation of concerns accomplished through this interface shifts critical performance and interoperability issues, such as scalable I/O and file format support, to a widely used and supported open source community library, improving the sustainability, performance, and functionality of Fluidity.

## Keywords

Mesh, topology, partitioning, renumbering, Fluidity, PETSc

## 1. INTRODUCTION

Scalable file I/O and efficient domain topology management present important challenges for many scientific applications if they are to fully utilise future exascale computing resources. Although these operations are common to many scientific codes they have received little attention in optimisation efforts, resulting in potentially severe performance bottlenecks for realistic simulations that require and generate large data sets. Moreover, due to a multitude of formats and a lack of convergence on standards for mesh and output data in the community there is only limited interoperability and very little code reuse among scientific applications for common operations, such as reading and partitioning input meshes. Thus developers are often forced to create custom I/O routines or even use application-specific file formats, which further limits application portability and interoperability.

Designing a scientific software stack to meet the needs of the next-generation of simulation software technologies demands, not only scalable and efficient algorithms to perform data I/O and mesh management at scale, but also an abstraction layer that allows a wide variety of application codes to utilise them and thus promotes code reuse and interoperability. Such an intermediate representation of mesh topology has recently been added to PETSc [3], a widely used scientific library for the scalable solution of partial differential equations, in the form of the DMPlex data management API [14].

In this paper we demonstrate the use of PETSc's DMPlex API to perform mesh input and domain topology management in Fluidity [17], a large scale CFD application code that already uses the PETSc library as its linear solver engine. By utilising DMPlex as the underlying mesh management abstraction we not only add support for new mesh file formats, such as Exodus II [20], CGNS [18], Gmsh [9], Fluent Case [2] and MED [1], to Fluidity, but also enable the use of domain decomposition methods, data migration, and mesh renumbering techniques at run-time. Moreover, the separation of concerns allows PETSc parallel data management and HDF5 support to be independently optimized for the target platform, removing this complexity from the application code. Our refactoring of Fluidity provides significant performance benefits due to improved cache locality and mesh distribution during simulation initialisation, which we demonstrate with performance benchmarks performed on Archer, a Cray XC30 architecture.

## 2. BACKGROUND

The key challenge in designing software for large scale systems lies in the composition of abstractions and the definition of clearly defined yet flexible interfaces between them. Code reuse and inter-disciplinary cooperation necessitate deeper software stacks and thus configuration and extensibility will play a key role in designing the software stack of the future [5]. In this paper we therefore focus on the interaction between applications and their supporting libraries to provide the infrastructure for efficient data management at exascale.

### 2.1 Fluidity

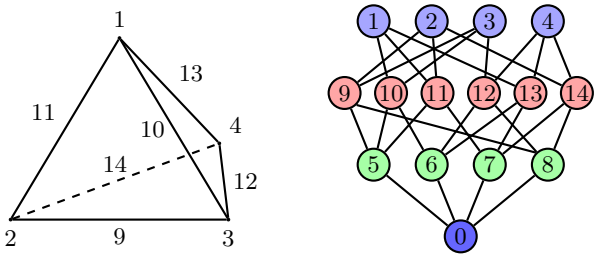
The primary user application in our work is Fluidity, an open source unstructured finite element code that uses mesh adaptivity to accurately represent a wide range of scales in a single numerical simulation without the need for nested grids. Fluidity is used in a number of different scientific areas including geophysical fluid dynamics, computational fluid dynamics, ocean modelling and mantle convection. Fluidity implements various finite element and finite volume discreti-

sation methods and is capable of solving solving the Navier-Stokes equation and accompanying field equations in one, two and three dimensions.

Previous optimisation efforts have highlighted that file I/O, in particular during model initialisation, presents a severe performance bottleneck when running on large numbers of processes [11]. The primary reasons for this are a off-line domain partitioning and the need to store each partition using a file-per-process strategy.

## 2.2 DMPlex

PETSc's ability to handle unstructured meshes is centred around DMPlex, a data management object that encapsulates the topology of unstructured grids to provide a range of functionalities common to many scientific applications. As shown in Figure 1, DMPlex stores the connectivity of the associated mesh as a layered directed acyclic graph (DAG), where each layer (stratum) represents a class of topological entities [14, 16]. This flexible yet efficient representation provides an abstract interface for the implementation of mesh management and manipulation algorithms using dimension-independent programming.




Vertex and edge numbering Topological connectivity

Figure 1: DAG-based representation of a single tetrahedron in DMPlex.

DMPlex stores data by associating data with points in the DAG, allowing an arbitrary data size for each point. This can be efficiently encoded using the same AIJ data structure used for sparse matrices. This scheme is general enough to encompass any discrete data layout over a mesh. The association with points also means that data can be moved using the same set of scalable primitives that are used for mesh distribution.

DMPlex's internal representation of mesh topology also provides an abstraction layer that decouples the mesh from the underlying file format and thus allows support for multiple mesh file formats to be added generically. At the time of writing DMPlex is capable of reading input meshes in Exodus II, CGNS, Gmsh, Fluent-Case and MED formats. Moreover, DMPlex provides output routines that generate solution output in HDF5-based XDMF format, while also storing the DMPlex DAG connectivity alongside the visualisable solution data to facilitate checkpointing [3].

In addition to a range of I/O capabilities DMPlex also provides parallel data marshalling through automated parallel distribution of the DMPlex [15] and the pre-allocation of parallel matrix and vector data structures. Mesh partitioning is provided via internal interfaces to several partitioner libraries (Chaco, Metis/ParMetis) and data migration is based on PETSc's internal *Star Forest* communication abstraction (PetscSF) [3]. Additionally, DMPlex is designed to provide the connectivity data and grid hierarchies required by sophisticated preconditioners, such as geometric multigrid methods and "Fieldsplit" preconditioning for multi-physics problems, to speed up the solution process [4, 6].

## 2.3 Mesh Reordering

Mesh reordering techniques represent a powerful performance optimisation that can be utilised to increase cache coherency of the matrices required during the solution process [10, 12, 21]. The well-known Reverse Cuthill-McKee (RCM) algorithm, which can be used to reduce the bandwidth of CSR matrices, is implemented in PETSc allowing DMPlex to compute the required permutation of mesh entities directly from the domain topology DAG. The resulting permutation can then be applied to any discretisation derived from the stored mesh topology to improve the cache coherency of the associated CSR matrices.

# 3. FLUIDITY-DMPLEX INTEGRATION

Initial mesh input has been a scalability bottleneck in Fluidity due to the off-line mesh partitioning step. As illustrated in Figure 2a, the current preprocessor module uses Zoltan [8], which use ParMetis [13] for graph partitioning, to partition and distribute the initial simulation state to the desired number of processes before writing the partitioned mesh and data to disk, allowing the main simulation to read the pre-partitioned data in a parallel fashion.

Fluidity's parallel mesh initialisation routines, however, rely on a file-per-process I/O strategy that require large numbers of individual files when running the application at scale. This has been shown to put significant pressure on the meta-data servers in distributed filesystems, such as Lustre or PVFS, which ultimately has a detrimental effect on scalability when using sufficiently large numbers of processes [11].

## 3.1 Parallel Simulation Start-up

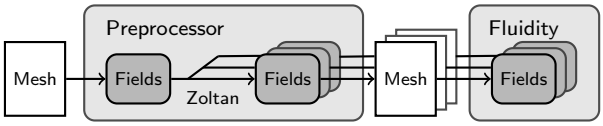
One of the objectives of this work, in addition to enhacing functionality and usability, is to alleviate Fluidity's start-up bottleneck by utilising DMPlex's mesh distribution capabilities to perform mesh partitioning at run-time. For this purpose, as shown in Figure 2b, a DMPlex topology object is created from the initial input mesh and immediately partitioned and distributed to all participating processes, allowing Fluidity's initial coordinate field to be derived from the DMPlex object in parallel. From the initial coordinate mesh all further discretisations and fields in the simulation state are then derived using existing functionality.

Using DMPlex as an intermediate representation for the underlying mesh topology has the following advantages:

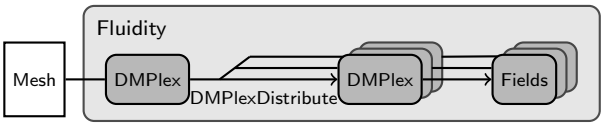
- Run-time mesh distribution and load balancing removes the need to store the partitioned mesh on disk, thus removing two costly I/O operations and reducing Fluidity's disk space requirements.
- Communication volume during startup is reduced, since only the topology graph is distributed. This is in con-

trast to the preprocessor, which partitions and distributes a fully allocated Fluidity state with multiple fields.

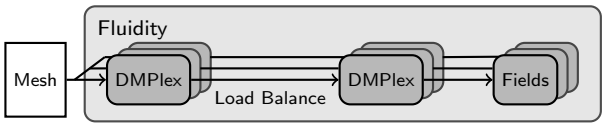
- Support for multiple previously unsupported mesh file formats is inherited from DMPlex, increasing application interoperability.




(a) Original Fluidity start-up based on off-line pre-processing.




(b) DMPlex-based start-up where an initial DMPlex object is distributed at run-time.




(c) Potential future workflow, where DMPlex performs the initial mesh read in parallel before a parallel load balancing step.

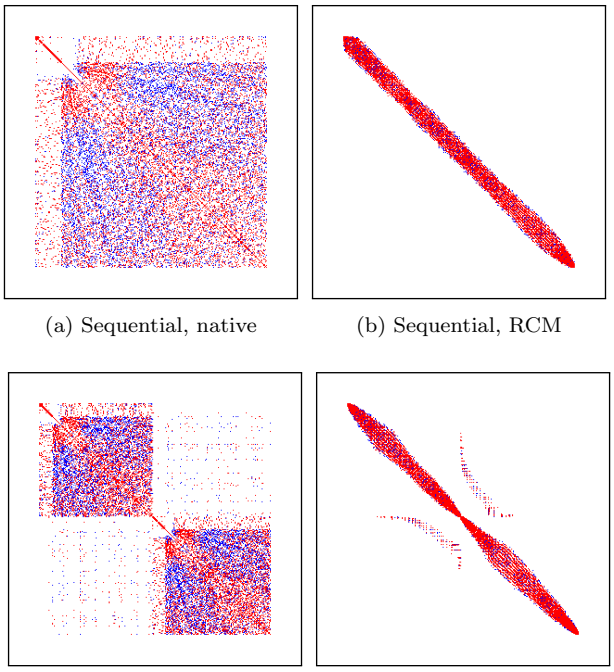
Figure 2: Workflow diagram for Fluidity simulation start-up.

A key point to note about the DMPlex-based mesh initialisation approach is that by delegating the initial mesh read to PETSc any mesh format reader added to DMPlex in the future will automatically be inherited by the application code. Moreover, future performance optimisations, such as parallisation of the initial mesh file read, will also be available in Fluidity without any further changes to the application. Such an envisaged scenario is shown in Figure 2c, where an already parallel DMPlex object is created from the initial file, followed by a load balancing step before deriving the parallel Fluidity state.

## 3.2 Mesh Renumbering

One of the key components of the DMPlex integration is the derivation of Fluidity's initial coordinate mesh object from the distributed DMPlex, which includes the derivation of the data mapping required for halo exchanges in parallel. DMPlex is able to provides such a mapping from local non-owned degrees-of-freedom (DoFs) to their respective remote owners. However, since Fluidity halo objects require remote non-owned DoFs in the solution field to be located contiguously at the end of the solution vector ("trailing receives" assumption), a node permutation is required when deriving Fluidity data structures from the mapping provided by DMPlex.

As a consequence, further node ordering permutations may be applied during the derivation of the initial field discretisation, such as the RCM renumbering provided by DMPlex. An optional renumbering step can be performed locally after the initial mesh distribution and added to the mesh initialisation routine with very little programming effort. As a result of Fluidity's run-time derivation of field discretisations from the underlying coordinate mesh, the RCM data layout of the initial reordering is inherited by all fields in the simulation, as shown in Figure 3. Moreover, as new mesh renumbering schemes are incorporating into PETSc, they will be automatically available to the application code.



(a) Sequential, native (b) Sequential, RCM

(c) 2 MPI ranks, native (d) 2 MPI ranks, RCM

Figure 3: Effects of RCM reordering on the structure of the assembled pressure matrix.

# 4. RESULTS

The following benchmark tests were performed on the UK national supercomputer, a Cray XE30 with 4920 nodes connected via an Aries interconnect and a parallel Lustre filesystem [1]. Each node consists of two 2.7 GHz, 12-core Intel E5-2697 v2 (Ivy Bridge) processors with 64GB of memory.

The benchmark runs simulate the flow past a square cylinder for 10 timesteps using a $P_1^{\mathrm{DG}} - P_2$ discretisation [7], where a second-order pressure field is solved using Fluidity's multigrid algorithm and paired with a discontinuous first order velocity field that uses a GMRES solver with SOR preconditioning. The mesh used has been generated with the Gmsh mesh generator [9] and is shown in Figure 4.

## 4.1 Mesh Initialisation

Figure 5 shows a comparison of the simulation start-up cost between the DMPlex-based implementation and the original preprocessor approach on 4 nodes (96 cores) with increasing mesh sizes up to approximately 3 million elements (weak scaling). The original start-up cost is hereby quantified as the sum of preprocessor and simulation initialisation times.

A clear improvement in overall start-up performance is shown in Figure 5a, although no significant improvement in direct

[1] `http://www.archer.ac.uk/`

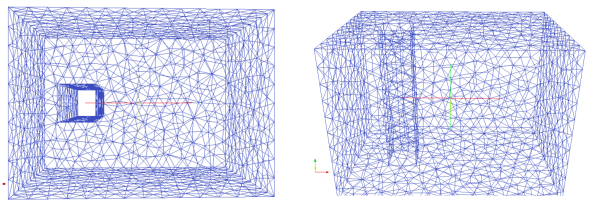


Figure 4: Three-dimensional benchmark mesh used to model flow past a cylinder.

file I/O, as shown in Figure 5b, can be determined. This is unsurprising, as file I/O is still completely dominated by the initial sequential read, although potential gains can be expected at larger scales due to removing the intermediate reads and writes of the partitioned mesh.

As highlighted in Figure 5c, the majority of the observed overall performance gains can be attributed to significantly improved mesh distribution via DMPlex. It is important to note here that DMPlex partitions and migrates only the mesh topology graph and its associated coordinate values, in contrast to the original preprocessor module that distributes fully assembled fields. As a result less data needs to be communicated during the mesh migration phase, resulting in significantly increased start-up performance.

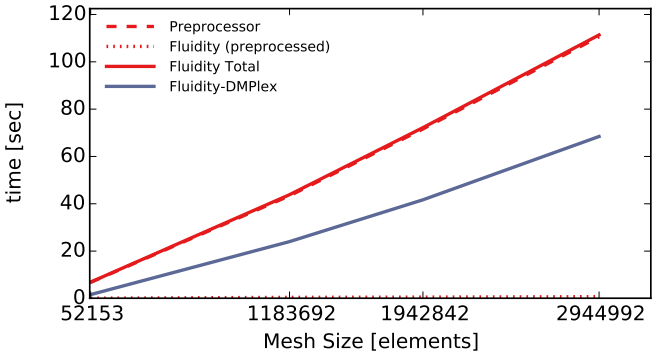
## 4.2 Mesh Renumbering

The overall simulation performance, including the effects of the mesh reordering derived from DMPlex, are evaluated in Figure 6. This benchmark compares the performance of both implementations by running 10 timesteps of the full simulation using a mesh with approximately 3 million elements on up to 96 cores. The results, shown in Figure 6a, indicate a consistent performance improvement of the DMPlex-based model with native mesh ordering over the preprocessor approach that increases with growing numbers of processes due to a smaller start-up overhead.

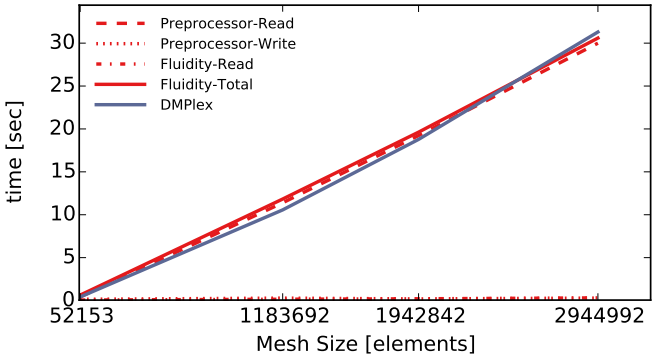
The effect of RCM mesh reordering is best demonstrated by analysing the two most expensive components of the simulation: the pressure field solve (Figure 6b) and the assembly of the velocity matrix (Figure 6c). Both components exhibit significant performance increases with RCM reordering on small numbers of processors that diminish as the simulation approaches the strong scaling limit. However, the benefits for overall simulation performance (Figure 6c) with RCM reordering decrease between 24 and 96 processes due to the fixed-cost start-up overhead of generating the permutation outweighing the solver and assembly benefits.

# 5. DISCUSSION

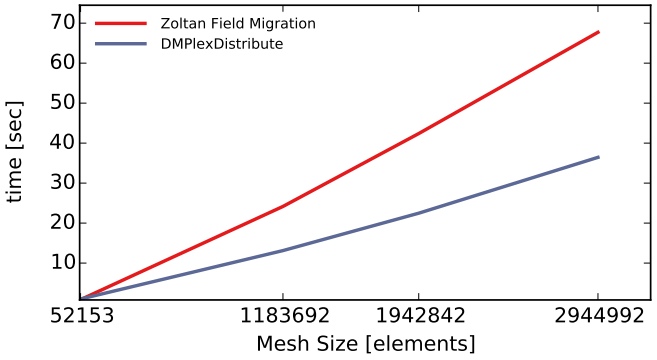
Achieving scalable performance with production-scale scientific applications on future exascale systems requires appropriate levels of abstraction across the entire software stack. In this paper we report progress on the integration of DMPlex, a library-level domain topology abstraction, with the application code Fluidity in order to delegate a set of common mesh and data management tasks to a widely used library. We highlight the increased interoperability achieved through the inheritance of new mesh file format readers and demonstrate improved model initialisation performance




(a) Overall simulation start-up

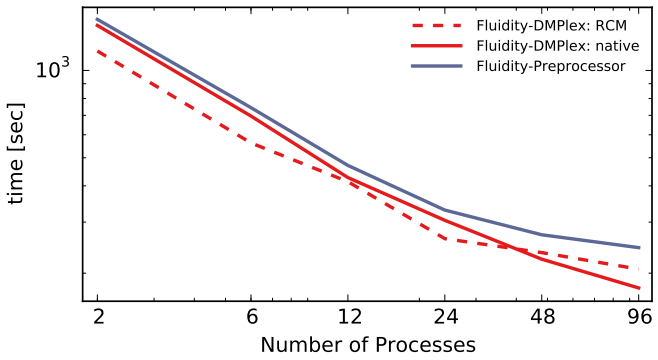



(b) Mesh file I/O

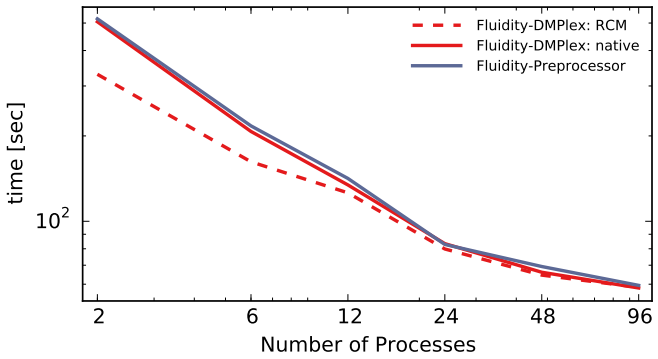



(c) Mesh distribution

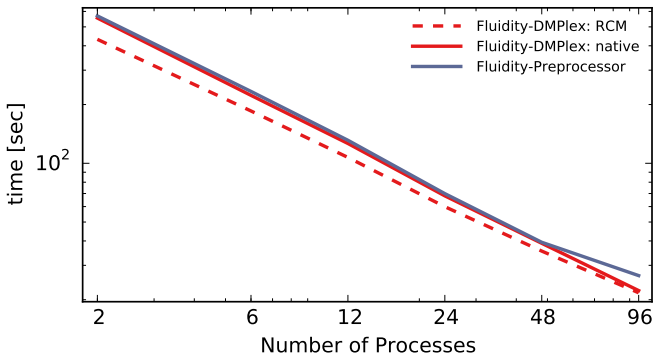
Figure 5: Comparison of total start-up cost between preprocessor and DMPlex-based approaches.




(a) Overall simulation performance.




(b) Pressure solve performance.




(c) Velocity assembly performance.

Figure 6: Full simulation performance for 10 timesteps on approximately 3 million elements.

through run-time mesh distribution routines provided by DMPlex.

The key benefit of the restructured model initialisation workflow, however, lies in the fact that responsibility for supporting various mesh file formats and optimising mesh file I/O now lies with the underlying library. This entails that any future extensions, such as new file formats or parallel mesh reader implementations, are automatically inherited by Fluidity, as well as other applications using PETSc, such as Firedrake [19] where we have also employed these abstractions. Moreover, by utilising a centralised mesh management API other types of mesh-based performance optimisations become available to the application, as highlighted by the seamless addition of the RCM renumbering feature.

## 5.1 Future Work

A key contribution of this work lies in the fact that it enables future extensions and optimisations. Most crucially perhaps is the development of a fully parallel mesh input reader in PETSc in order to overcome the remaining sequential bottleneck during model initialisation. This change, however, requires a new default mesh format for Fluidity due to the inherently sequential nature of the Gmsh file format, which again highlights the need for abstraction when optimising mesh management.

In addition to the optimisation of mesh input and model initialisation, further integration of DMPlex throughout Fluidity is desirable to utilise DMPlex's advanced I/O features, such as the HDF5-based Xdmf output format. For this purpose closer integration is required, where additional discretisation data needs to be passed to PETSc to perform all the necessary field I/O. Moreover, DMPlex's mesh and data distribution utility may also be used to provide load balancing after mesh adaptation.

# 6. ACKNOWLEDGMENTS

This work was supported by the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk), EPSRC grants EP/M019721/1 and EP/L000407/1 and the Intel Parallel Computing Center program through grants to both the University of Chicago and Imperial College London. MGK acknowledges partial support from DOE Contract DE-AC02-06CH11357. We would also like to thank Frank Milthaler for providing the test configurations used for benchmarking.

# 7. REFERENCES

[1] Med data model. `http://www.code-aster.org/outils/med/`.

[2] ANSYS. FLUENT reference manual, 2015. Software Release Version 6.3.

[3] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[4] J. Brown, M. Knepley, D. May, L. McInnes, and B. Smith. Composable Linear Solvers for Multiphysics. In *Parallel and Distributed Computing*

*(ISPDC), 2012 11th International Symposium on*, pages 55–62, June 2012.

[5] J. Brown, M. G. Knepley, and B. F. Smith. Run-time extensibility and librarization of simulation software. *IEEE Computing in Science and Engineering*, 2015.

[6] P. Brune, M. Knepley, and L. Scott. Unstructured Geometric Multigrid in Two and Three Dimensions on Complex and Graded Meshes. *SIAM Journal on Scientific Computing*, 35(1):A173–A191, 2013.

[7] C. J. Cotter, D. A. Ham, and C. C. Pain. A mixed discontinuous/continuous finite element pair for shallow-water ocean modelling. *Ocean Modelling*, 26(1):86–90, 2009.

[8] K. D. Devine, E. G. Boman, R. T. Heaphy, U. V. Çatalyürek, and R. H. Bisseling. Parallel hypergraph partitioning for irregular problems. *SIAM Parallel Processing for Scientific Computing*, February 2006.

[9] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[10] F. Günther, M. Mehl, M. Pögl, and C. Zenger. A Cache-Aware Algorithm for PDEs on Hierarchical Data Structures Based on Space-Filling Curves. *SIAM Journal on Scientific Computing*, 28(5):1634–1650, 2006.

[11] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland. Developing a scalable hybrid MPI/OpenMP unstructured finite element model. *Computers & Fluids*, 110(0):227 – 234, 2015. ParCFD 2013.

[12] G. Haase, M. Liebmann, and G. Plank. A Hilbert-order multiplication scheme for unstructured sparse matrices. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):213–220, 2007.

[13] G. Karypis and V. Kumar. ParMETIS: Parallel graph partitioning and sparse matrix ordering library. Technical Report 97-060, Department of Computer Science, University of Minnesota, 1997. http://www.cs.umn.edu/ metis.

[14] M. G. Knepley and D. A. Karpeev. Mesh Algorithms for PDE with Sieve I: Mesh Distribution. *Sci. Program.*, 17(3):215–230, Aug. 2009.

[15] M. G. Knepley, M. Lange, and G. J. Gorman. Unstructured overlapping mesh distribution in parallel. *Submitted to ACM TOMS*, 2015.

[16] A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, 4:283–295, 2009.

[17] M. D. Piggott, G. J. Gorman, C. C. Pain, P. A. Allison, A. S. Candy, B. T. Martin, and M. R. Wells. A new computational framework for multi-scale ocean modelling based on adapting unstructured meshes. *International Journal for Numerical Methods in Fluids*, 56(8):1003–1015, 2008.

[18] D. Poirier, S. R. Allmaras, D. R. McCarthy, M. F. Smith, and F. Y. Enomoto. The cgns system, 1998. AIAA Paper 98-3007.

[19] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. Kelly. Firedrake: automating the finite element method by composing abstractions. *Submitted to ACM TOMS*, 2015.

[20] L. A. Schoof and V. R. Yarberry. EXODUS II: a finite element data model. Technical Report SAND92-2137, Sandia National Laboratories, Albuquerque, NM, 1994.

[21] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious Mesh Layouts. *ACM Trans. Graph.*, 24(3):886–893, July 2005.

# Prototyping for Exascale

Brian Vinter
Niels Bohr Institute
University of Copenhagen
Denmark
vinter@nbi.ku.dk

Mads R. B. Kristensen
Niels Bohr Institute
University of Copenhagen
Denmark
madsbk@nbi.ku.dk

Simon A. F. Lund
Niels Bohr Institute
University of Copenhagen
Denmark
safl@nbi.ku.dk

Troels Blum
Niels Bohr Institute
University of Copenhagen
Denmark
blum@nbi.ku.dk

Kenneth Skovhede
Niels Bohr Institute
University of Copenhagen
Denmark
skovhede@nbi.ku.dk

## ABSTRACT

Prototyping is a common approach to developing new scientific codes. Through prototypes a scientist builds confidence that an idea can in fact work for a scientific challenge, and the prototype also acts as the definitive design for a final implementation. As supercomputing approaches exaflops performance, the teraflops platforms that are available for prototyping becomes increasingly distant from the target performance, and new tools are needed to help close the performance gap between high productivity prototyping and high performance end-solutions. In this work we propose Bohrium, an open-source just-in-time compiler, as a possible solution to the prototyping problem. We will show how the same Python program can execute seamlessly on single-core, multi-cores, GPGPU and cluster architectures, and thus eliminating the need for parallel programming in the prototype stage. We will show how the same, unmodified, Python implementation of a Jacobi solver, a Black-Scholes pricing, an $O(n^2)$ complexity n-body simulation, and a Shallow-Water simulation scales to a 32-core machine with 50.1, 29.8, 17.3, and 44.4 speedups compared to the NumPy execution, while the same Python benchmarks run on a NVidia GTX 680, achieves speedups of 55.7, 43.0, 77.1, and 140.2, and a eight node cluster with gb-ethernet interconnect (256 cores in total), obtain speedups of 4.1, 7.9, 6.6 and 6.4, compared to a single 32 core node.

## 1. INTRODUCTION

While achieving exascale-computing in itself is a huge technical task, bringing scientific users to a competence level where they can utilize an exascale machine is likely to pose problems of the same scale. While large codes, maintained by a research community, is likely to make the transition from peta- to exascale as a natural evolution in the code, smaller teams will be hard pressed to make the move to exascale. One of the challenges that face researchers that write their own codes is that of prototyping. Today most teams will move from idea to code via a prototype, typically in Matlab, IDL, Python or a similar high productivity programming language.

Prototyping is an essential tool for testing the scientific hypothesis in small scale before spending more time on an actual implementation, since many scientific expressions do not easily translate into algorithms, and issues such as numerical stability, etc. are often not investigated formally. Thus the actual workflow for scientific codes is often iterative as shown in figure 1 below. Scientists will test their idea in a high productivity environment, using a small dataset, typically a ratio of one to a thousand, on a conventional, but large, computer, before moving on to an actual supercomputer for the real scale experiments.

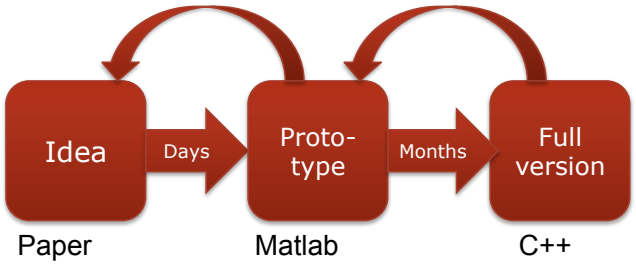



**Figure 1: Prototyping workflow**

With respect to exascale computing this approach poses a significant challenge, while exascale machines will be build by scaling supercomputers to a core count two orders of magnitude, in the order of 100 million, no such explosion in high-end servers is guaranteed, or even likely. Thus while researchers today, are expected to move three orders of magnitude, from teraflops to petaflops, when moving from prototype to final implementation, prototyping is likely to remain at teraflop when supercomputers move to exaflops, and researchers will have to move six orders of magnitude. Even scaling three orders of magnitude as is done today is non-trivial and often problems arise that were not detected by the prototype, when this challenge is increased another three orders of magnitude it is unlikely that the current approach will be sustainable.

Thus it makes sense to investigate new, scalable, approaches to prototyping. The successful prototyping tool must be highly productive and allow descriptive representations of an algorithm both of which are met by todays use of Matlab. A future prototyping tool must however be much faster than Matlab, it must have a better single core performance, and it must be able to run on multicores, accelerators, and

cluster-computeres alike in order to satisfy the requirement of doing prototyping at the petaflops-scale in order to reduce the distance to state-of-the-art exaflops machines to three orders of magnitude, as it is today.

In the following, we introduce the Bohrium just-in-time compiler which, combined with the Numerical Python library NumPy[5], may provide a solution to next-generation-prototyping. Bohrium is not developed for prototyping, but rather for rapid solutions on parallel hardware, but non the less it matches the requirements that we believe are essential for prototyping in for the exascale.

# 2. THE BOHRIUM RUNTIME SYSTEM

The open-source project Bohrium[1] is a runtime system for high-performance high-productivity development[4, 3]. Bohrium provides the mechanics to couple an array-programming language or library with an architecture-specific implementation seamlessly.

Bohrium consists of a number of components that communicate by exchanging a hardware agnostic array bytecode. Components can be architecture-specific but they are all interchangeable since all uses the same bytecode and communication protocol. This design makes it possible to combine components in a setup that match a specific execution environment without changing the original user application. Bohrium consist of the following three component types (Fig. 2):

**Bridge** The role of the Bridge is to integrate Bohrium into existing languages and libraries. The Bridge generates array bytecode that corresponds to the user-code.

**Vector Engine Manager (VEM)** The role of the VEM is to manage data location and ownership of arrays. It also manages the distribution of computing jobs between potentially several Vector Engines and thus multiple processors.

**Vector Engine (VE)** The VE is the architecture-specific implementation that executes array bytecode.

When using the Bohrium framework, at least one implementation of each component type must be available. However, the exact component setup depends on the runtime system and what hardware to utilize, e.g. executing NumPy on a single machine using the CPU would require a Bridge implementation for NumPy, a VEM implementation for a machine node, and a VE implementation for a CPU. Now, in order to utilize a GPU instead, we can exchange the CPU-VE with a GPU-VE without having to change a single line of code in the NumPy application. This is a key feature of Bohrium: the ability to change the execution hardware without changing the user application.

## 2.1 Configuration

To make Bohrium as flexible a framework as possible, Bohrium manage the setup of all the components at runtime through a configuration file. The idea is that the user or system administrator can specify the hardware setup of the system through an configuration file (Fig. 3). Thus, it is just a matter of editing the configuration file when changing or moving to a new hardware setup and there is no need to change the user applications.

[1]Available at http://www.bh107.org.

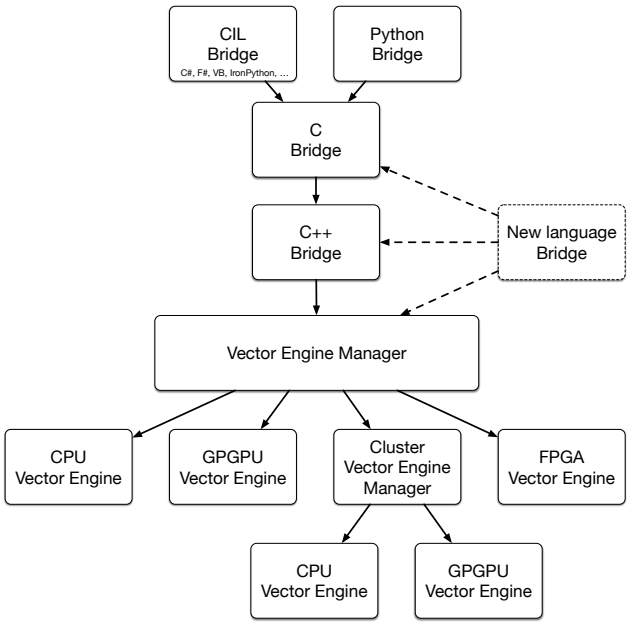



**Figure 2: Component Overview**

## 2.2 Vector Bytecode

A vital part of Bohrium is the array bytecode that constitutes the link between the high-level user language and the low-level execution engine. The bytecode is designed with the declarative array-programming model in mind where the bytecode instructions operate on input and output arrays. To avoid excessive memory copying, the arrays can also be shaped into multi-dimensional arrays. These reshaped array views are then not necessarily comprised of elements that are contiguous in memory. Each dimension comprises a stride and size, such that any regularly shaped subset of the underlying data can be accessed. We have chosen to focus on a simple, yet flexible, data structure that allows us to express any regularly distributed arrays. Figure 4 shows how the shape is implemented and how the data is projected.

The aim is to have an array bytecode that support data parallelism implicitly and thus makes it easy for the bridge to translate the user language into the bytecode efficiently. Additionally, the design enables the VE to exploit data parallelism through SIMD[2] and the VEM through SPMD[3].

## 2.3 Bridge

The Bridge component is the *bridge* between the programming interface. In order to interface with the frontend language, the language-specific bridge component translates array operations into Bohrium array bytecode lazily. That is, the bridge collects array operations until it encounter a language condition, in which case it sends the collected

[2]Single Instruction, Multiple Data
[3]Single Program, Multiple Data

```
# Bridge for NumPy
[numpy]
type = bridge
children = node

# Vector Engine Manager for a single machine
[node]
type = vem
impl = libbh_vem_node.so
children = gpu

# Vector Engine for a GPU
[gpu]
type = ve
impl = lbbh_ve_gpu.so
```

**Figure 3: This example configuration provides a setup for utilizing a GPU on one machine by instructing the Vector Engine Manager to use the GPU Vector Engine implemented in the shared library `lbhvb_ve_gpu.so`.**

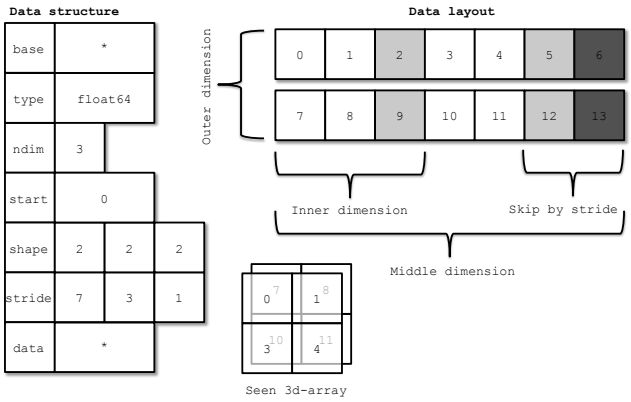



**Figure 4: Descriptor for n-dimensional array and corresponding interpretation**

```
import numpy as np

def solve(height, width, epsilon=0.005):
  grid = np.zeros((height+2,width+2),np.float64)
  grid[:,0]   = -273.15
  grid[:,-1]  = -273.15
  grid[-1,:]  = -273.15
  grid[0,:]   = 40.0
  center = grid[1:-1,1:-1]
  north  = grid[:-2,1:-1]
  south  = grid[2:,1:-1]
  east   = grid[1:-1,:-2]
  west   = grid[1:-1,2:]
  delta  = epsilon+1
  while delta > epsilon:
    tmp = 0.2*(center+north+south+east+west)
    delta = np.sum(np.abs(tmp-center))
    center[:] = tmp
```

**Figure 5: Python implementation of a heat equation solve that uses the finite-difference method to calculate the heat diffusion. Note that in order to utilize Bohrium, we use the command line argument "`-m`", e.g. "`python -m npbackend heat2d.py`"**

operations to the underlaying Bohrium components. Consequently, Bohrium only handles a subset of the frontend language – namely the array operations. The frontend language handles all non-deterministic aspect of program, such as conditional branches and loops, by itself.

An example of a Bohrium bridge is the Python/NumPy-bridge that seamlessly integrates with NumPy. The bridge is a drop-in replacement of NumPy thus without changing a single line of code, it is possible to utilize Bohrium (Fig. 5).

## 2.4 Vector Engine Manager

In order to utilize scalable architectures fully, distributed memory parallelism is mandatory. The Cluster component in Bohrium is currently quite naïve; it uses the bulk-synchronous parallel model[6] with static data decomposition and no communication latency hiding. We know from previous work than such optimizations are possible[2].

Bohrium implements all communication through the MPI-2 library and use a process hierarchy that consists of one master-process and multiple worker-processes. The master-process executes a regular Bohrium setup with the Bridge, Cluster-VEM, Node-VEM, and VE. The worker-processes, on the other hand, execute the same setup but without the Bridge and thus without the user applications. Instead, the master-process will broadcast array bytecode and array meta-data to the worker-processes throughout the execution of the user application.

Bohrium use a data-centric approach where a static decomposition dictates the data distribution between the MPI-processes. Because of this static data decomposition, all processes have full knowledge of the data distribution and need not exchange data location meta-data. Furthermore, the task of computing array operations is also statically distributed which means that any process can calculate locally what needs to be sent, received, and computed. Meta-

data communication is only needed when broadcasting array bytecode and creating new arrays – a task that has an asymptotic complexity of $O(\log_2 n)$, where $n$ is the number of nodes.

## 2.5 Vector Engine

The Vector Engine (VE) is the only component type that actually performs array operations. Bohrium implements two VEs, the CPU-VE and GPU-VE, that utilizes multi-core CPUs and GPGPUs respectively. Through the use of Just-In-Time (JIT) compilation, both VEs compiles the array bytecode received from the Node-VEM into architecture specific binary kernels. In order to utilize multi-core CPUs, the CPU-VE feeds the JIT-compiler with OpenMP annotated ANSI C source code whereas the GPU-VE generates OpenCL source code in order to utilize GPGPUs from both Nvidia and AMD.

# 3. BENCHMARKS

In order to evaluate the performance of Bohrium, we will perform a series of benchmarks that compares Bohrium against Python/NumPy. For each benchmark, we report the mean of five execution runs all within 10% deviation from the mean. We use 64-bit double floating-point precision for all calculations and all speedup results are strong scaling where the data size is fixed. The benchmarks consist of the following four applications:

**Black Scholes** The Black-Scholes pricing model is a partial differential equation, which is used in finance for calculating price variations over time[1]. This implementation uses a Monte Carlo simulation to calculate the Black-Scholes pricing model.

**Heat Equation** simulates the heat transfer on a surface represented by a two-dimensional grid, implemented using jacobi-iteration with numerical convergence (Fig. 5).

**N-Body Nice** The Nice variation of the newtonian n-body simulation is used to model larger galaxies with a large number of asteroids. The mass of the asteroids is small enough that their gravitational pull is insignificant. Thus, only the force of the planets are applied to the asteroids. The planets exchange forces similar to a regular n-body simulation.

**Shallow Water** simulates a system governed by the Shallow Water equations. The simulation commences by placing a drop of water in a still container. The simulation then proceeds, in discrete time-steps, simulating the water movement. The implementation is a port of the MATLAB application by Burkardt[4].

### *Multi-Core Processor*

Figure 6 shows that results of running the four applications on 32 CPU-cores (Table 1). Beside comparing Bohrium versus Python/NumPy, the results of the Heat Equation includes two handwritten parallel implementations – one in ANSI-C and one in C++11 – both using OpenMP. The results clearly shows that the CPU-VE of Bohrium achieve a significant performance boost compared to Python/NumPy and is even competitive to handwritten compiled C/C++ code.

[4]http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/

| | |
|---|---|
| Processor: | AMD Opteron 6272 |
| Clock: | 2.1 GHz |
| Cores: | 32 |
| L3 Cache: | 16MB |
| Memory: | 128GB DDR3 |
| Compiler: | GCC 4.6.3 |
| Network: | Gigabit Ethernet |
| Software: | Linux 3.13, Python 2.7, NumPy 1.8.2 |

**Table 1: Multi-Core Processor Specification**

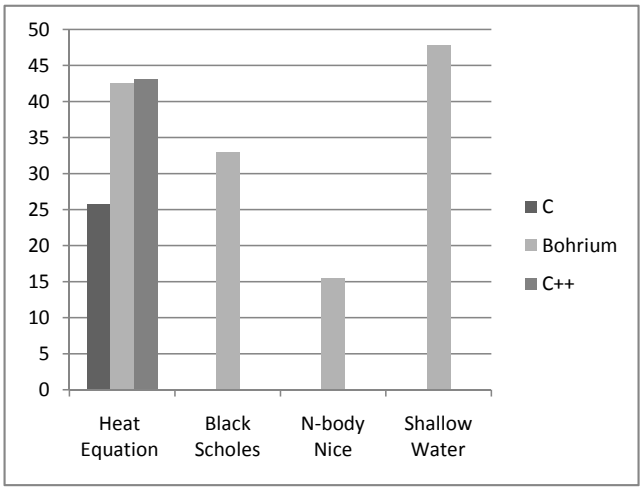



**Figure 6: Relative speedup utilizing 32-cores compared to a sequential Python/NumPy execution.**

### *GPGPU*

Figure 7 shows that results of running the four applications on a GPGPU (Table 2). Compared to the multi-processor, the GPGPU takes the performance boost even further. Noticeable is the Black Scholes results with more than 300 times speedup compared to Python/NumPy.

### *Cluster*

Figure 8 shows that results of running the four applications on an eight-node cluster where each node is the multi-code processor from Table 1 connected through Gigabit Ether-

| | |
|---|---|
| Processor: | Intel Core i7-3770 |
| Clock: | 3.4 GHz |
| Cores: | 4 |
| L3 Cache: | 16MB |
| Memory: | 128GB DDR3 |
| Compiler: | GCC 4.6.3 |
| Network: | Gigabit Ethernet |
| | |
| GPGPU: | AMD HD 7970 |
| Clock: | 1000 MHz |
| Memory: | 3GB GDDR5 |
| -bandwidth: | 288 GB/s |
| Software: | Linux 3.13, Python 2.7, NumPy 1.8.2, OpenCL 2.1 |

**Table 2: GPGPU Specification**

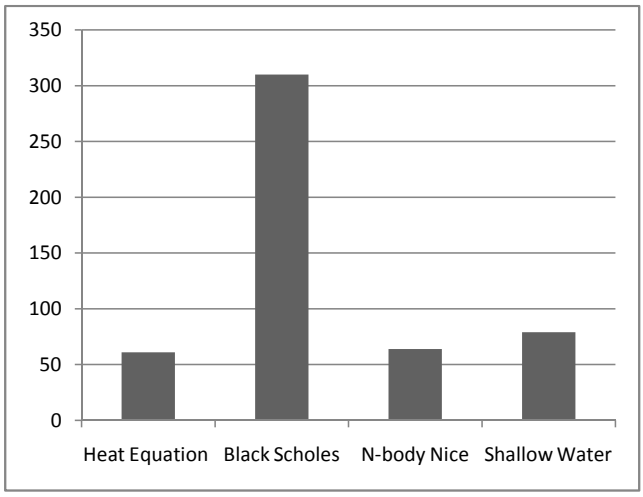



**Figure 7: Relative speedup utilizing GPGPU compared to a sequential Python/NumPy execution.**

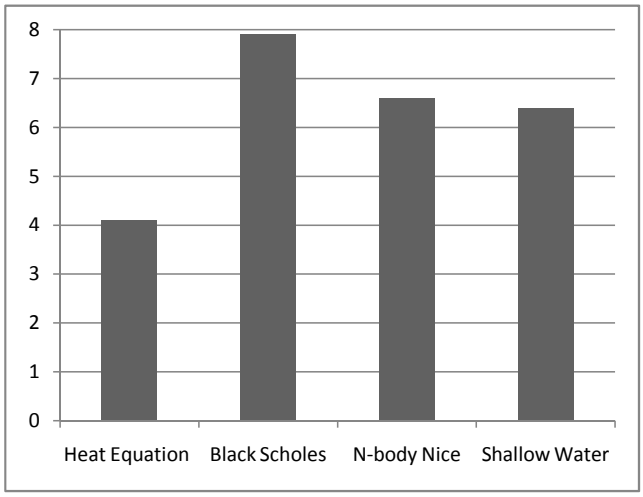



**Figure 8: Relative scalability on an eight-node cluster. We compared the utilization of one node (32-cores) against the utilization of eight node (256-cores).**

net. We run a MPI-process per cluster node and use the CPU-VE benchmark (here, we use an older version of the CPU-VE implementation than the one previously used) in Bohrium to utilize the 32-cores on each node. In order to show scalability, we compare 32-cores executions with 256-cores executions. The scalability goes from 50% to 95% speedup utilization.

# 4. CONCLUSIONS

In this work we have shown how Python/Numpy in combination with the Bohrium just-in-time compiler, offers an attractive work-to-performance ratio. While better performance can be had from expert implementations, a Python/Numpy implementation is fully on-par with other high-producitivity languages with respect to the the effort the scientist has to put in, and the performance is on-par, or close to, that of an straight forward C++ implementation. The end result is that a scientist may move seamlessly from a laptop version of a code to a large, heterogeneous, parallel machine, without any changes to the code. In fact, we will show that the scientist can continue to work from a laptop, with interactive graphics if needed, while the contracted array operations are all executed on a remote machine, including supercomputers, granted that the scientist is willing to wait online for the job to be scheduled at the SC site. The descriptive approach not only makes scientists more productive, but also reduces the number of errors as no explicit parallelism is expressed, and synchronization requirements are fully derived from the descriptive implementation of the algorithm.

# 5. REFERENCES

[1] F. Black and M. Scholes. The pricing of options and corporate liabilities. *The journal of political economy*, pages 637–654, 1973.

[2] M. Kristensen and B. Vinter. Managing communication latency-hiding at runtime for parallel programming languages and libraries. In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 546–555, 2012.

[3] M. R. Kristensen, S. A. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: a virtual machine approach to portable parallelism. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 312–321. IEEE, 2014.

[4] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter. Bohrium: Unmodified NumPy Code on CPU, GPU, and Cluster. In *4th Workshop on Python for High Performance and Scientific Computing (PyHPC'13)*, 2013.

[5] T. E. Oliphant. Python for Scientific Computing. *Computing in Science and Engg.*, 9(3):10–20, May 2007.

[6] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

# ExaSHARK+GASPI: Reducing the burden to program large HPC systems since 2014

Tom Vander Aa, Imen Chakroun, Roel Wuyts
ExaScience Lifelab at imec
Kapeldreef 75
Leuven, Belgium
Tom.VanderAa@imec.be

Mirko Rahn
Fraunhofer ITWM
Fraunhofer Platz 1
Kaiserslautern, Germany
rahn@itwm.fraunhofer.de

Christian Simmendinger
T-Systems for Research
Curiestraße 5
Stuttgart, Germany
christian.simmendinger@t-systems.com

## ABSTRACT

Several trends in HPC systems make it challenging to quickly and easily develop applications that perform well. One important trend is an increased number of levels in HPC systems: more levels of memory and interconnect, more levels of parallelism (SIMD, multi-threading, multi-core,... ). A second trend, caused by the first is an explosion of software solutions to exploit all these levels.

This paper is about how ExaSHARK - a library for handling n-dimensional distributed arrays - combined with the GASPI PGAS language aims to reduce the increasing programming burden while still providing good performance. ExaSHARK offers its users a global array like usability while its underlying runtime builds on GASPI to take optimal advantage of the PGAS paradigm.

We will present first result and challenges on using GASPI as the main underlying programming model for ExaSHARK. These result show that by using ExaSHARK the application can take advantage of the PGAS library without having to know it is underneath (code portability). On the other hand it is clear that to get good performance, we need to change the application's and ExaSHARK's communication patterns to better exploit the asynchronous nature of GASPI (no performance portability).

## Keywords

Regular Grids, Communication libraries, ExaScale, Software for communication optimization, PGAS, GPI, MPI

## 1. INTRODUCTION

High Performance Computing (HPC) architectures are expected to change dramatically in the next decade with the arrival of exascale computing, high performance computers that offer 1 exaFlop ($10^{15}$) of performance. Because of power and cooling constraints, large increases in individual core performance are not possible and as a result on-chip parallelism is increasing rapidly. The expected hardware for an exascale machine node will therefore need to rely on massive parallelism both on-chip and off-chip, with a complex distributed hierarchy of resources [6]. Programming a machine of such scale and complexity will be very hard unless some appropriate, workable layers of abstraction are introduced to bridge the gap between problem specification and efficient code at the cluster, node and chip levels.

Exascale machines will be mainly necessary for scientific and industrial simulations where scientists try to understand more complex phenomena by using simulations that run at ever higher resolutions and on ever longer time scales. One of the primary data structures in many of these scientific simulations is a *regular multidimensional array*. Indeed, a number of simulations can be modeled as time-discrete evolution on structured multidimensional array. Regular arrays are also at the core of many numerical algorithms.

In this paper we describe how to combine ExaSHARK, a library for handling n-dimensional distributed arrays, with the GASPI PGAS language to reduce the increasing programming burden while still providing good performance. ExaSHARK offers its users a global array like usability while its underlying runtime builds on GASPI to take optimal advantage of the PGAS paradigm. Next to GASPI, ExaSHARK can be configured to also include any of the aforementioned programming models.

The structure of this paper is as follows. Section 2 presents the ExaSHARK and GASPI libraries. Section 3 shows first results on ExaSHARK+GASPI and Section 4 are the conclusions.

## 2. EXASHARK AND GASPI

As this paper is about combining ExaSHARK and GASPI, this section aims to describe both libraries in more details, as well as the work undertaken to integrate them.

### 2.1 ExaSHARK

ExaSHARK is an open source middleware [3], offered as a library, targeted at reducing the increasing programming

burden on heterogeneous current and future exascale architectures. ExaSHARK handles matrices that are physically distributed block-wise, either regularly or as the Cartesian product of irregular distributions on each axis. The access to the global array is performed through a logical indexing. The programmer can define halos/ghost regions in the global array.

Our major architectural drivers for a scalable structured grid-based library are efficiency, portability and ease of coding. ExaSHARK is portable since it is built upon widely used technologies such as MPI and C++ as a programming language. It provides simple coding via a global-arrays-like interface which offers template-based functions (dot products, matrix multiplications, unary expressions). The functionalities offered by ExaSHARK are efficient since they use asynchronous and specific communication patterns.

The main properties of ExaSHARK are:

- **Data distribution** ExaSHARK is based on the PGAS parallel programming model which is convenient for expressing algorithms with large and random data access. Each process is assumed to have fast access to a portion of each distributed matrix, and slower access to the remainder.
- **Communication layer** ExaSHARK supports a plethora of lower level programming models and libraries with the aim of being adequate to exascale systems which are highly heterogeneous. Application developers may use pure MPI and/or hybrid MPI + OpenMP threads to target coarse and medium-grained parallelism.
- **Expression templates over global arrays** ExaSHARK offers many high-level functions traditionally associated with arrays, eliminating the need for programmers to write these functions themselves. Examples are basic mathematical operators, unary functions, standard global arrays operations such as dot products and matrix vector multiplication,and so on. These functionalities are implemented using expression templates [9].
- **Inter-operability and interfacing with external software** ExaSHARK can interface with external libraries when needed by the user's application. For example, developers can use the functionality of the Intel's Math Kernel Library (MKL) for optimized math routines [5] or PETSc (Portable, Extensible Toolkit for Scientific Computation) [1] for advanced numerical methods for partial differential equations and sparse matrix computations.

## 2.2 GASPI

The Global Address Space Programming Interface (GASPI [4]) is the specification for a PGAS style programming model for C/C++ and Fortran. The API consists of a set of basic routines. Its communication layer can take full advantage of the hardware capabilities to utilize remote direct memory access (RDMA) for spending no CPU cycles on communication.

The GASPI API provides one-sided communication calls and at its core consists of no more than there functions:

- **gaspi_write** Schedules the write of a block of local data to a remote memory.
- **gaspi_notify** Sends a notification to a remote node. While GASPI is very asynchronous, the notification is guaranteed to arrive at the remote side, *after* all the writes to that node have been completed.
- **gaspi_wait_notify** Waits for a notification and hence for the reads preceding this notification.

While GASPI has more functions for ease of programming, these three are the main functions that are needed by any program. The GASPI library has been optimized to work in a multi-threaded environment and is interoperable with MPI. The latter means a single application can use MPI for parts of its communication and GASPI for other parts.

Several popular proto-type applications and benchmarks have been implemented with GASPI and for these GASPI shows excellent scaling properties [8, 7].

## 2.3 ExaSHARK + GASPI

This section outlines how the communication layer of ExaSHARK can be ported to use GASPI. A first implementation of ExaSHARK on GASPI has been done, i.e. all functionality is there and we currently have a working but non-optimal version of Shark using GASPI.

The porting consists of replacing calls to MPI in the communication primitives of ExaSHARK with GASPI calls.

The effort depends on how compatible GASPI is with respect to the ExaSHARK functionality and varies depending on the functionality being ported. We list the different degrees below to serve as lessons learnt:

- **Trivial**: Some aspects can be trivially translated from MPI to GASPI. For example: An MPI_Win_allocate is translated to a gaspi_segment_create
- **Easier**: Some aspects are easier to do in GASPI than in MPI. For example: The concept of notifications in GASPI makes the handling of asynchronous communication much easier than with MPI.
- **Minimal Thinking**: Aspects that GASPI handles in a different way require some thinking to implement. For example: GASPI does not have vector data-types. It only communicates flat data. This means that in cases a sub-array is communicated this happens in MPI by use of an hvector data-type, which can contain blocks with stride. In GASPI the individual contiguous blocks have to be communicated individually.
- **Significant Effort**: MPI is feature-rich and some parts in ExaSHARK are easy to express in MPI but require significant effort in GASPI. The underlying reason is that GASPI is intentionally kept simpler than

MPI. For example: GASPI does not support remote accumulation because this would require additional buffering at the remote site, which does not scale well.

- **Problematic**: GASPI is still young compared to MPI and some use-cases are not yet well supported. One use-case is the use of GASPI in a library (in this case ExaSHARK). For example: GASPI uses integers to identify memory segments. It is non-trivial to make sure these identifies do not overlap between different libraries using GASPI in the same user application.

The next section presents first results obtained from the new ExaSHARK+GASPI library.

## 3. EXPERIMENTS AND RESULTS

As first experiment we choose to evaluate the scaling of ExaSHARK+GASPI on a simple 2D Jacobi heat [2] stencil application in an ExaScale environment. ExaScale systems will likely have many more cores but the system memory size will not scale as much [6]. For grid-based applications, such as the ones ExaSHARK support, this will lead to fewer grid points per core, and thus much more boundary points.

The graph in Figure 1 shows strong scaling results for the heat application on a $1024 \times 1024$ size grid. The X-axis lists the number of nodes and number of threads per node, the Y-axis the speedup relative to two nodes. Four implementations of the 2D heat application are compared:

- **nocomm**: a functionally incorrect communication-free implementation;
- **gpi**: ExaSHARK+GASPI with one GASPI process per node and one OpenMP process per core;
- **mpi+openmp**: ExaSHARK with one MPI process per node one OpenMP process per core;
- **mpi+ppn**: ExaSHARK with one MPI process per core.

The results clearly show the above example does not scale. Several potential reasons have been identified.

- ExaSHARK+GASPI uses a bulk synchronous communication scheme. Significant overhead is due to synchronization and load imbalance.
- In ExaSHARK+GASPI the communication functions are not multi-threaded, while the application itself is. Due to Amdahl's law the time spent in communication layer increases with the number of threads.
- The 2D heat application is memory-bandwidth bound. It contains relatively little computations that can be overlapped with communications.

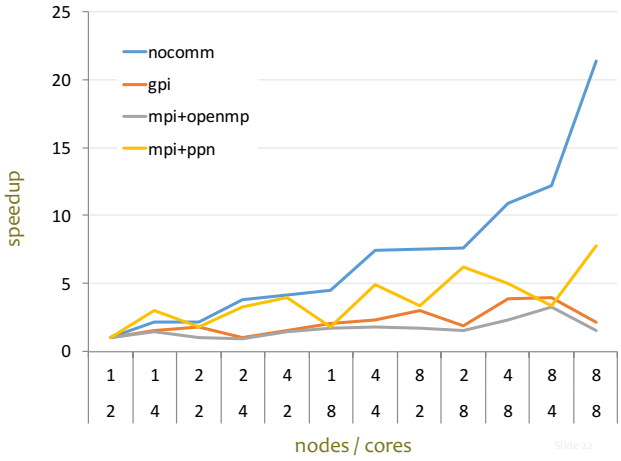



**Figure 1: Strong scaling for a 2D heat application on a $1024 \times 1024$ size grid. Speedup vs. number of nodes and cores for four versions.**

## 4. CONCLUSIONS

While ExaSHARK is a modern and high-level library with support for multiple programming paradigms and natural syntax (thank to c++), while ExaSHARK performs reasonably well at node-level, there is much room for improvement to make it scale well. Two ideas for improvement are:

- Make ExaSHARK thread-safe to be able to do multi-threaded communication.
- Support for much more fine-grained synchronization to scale to modern many-core systems.

More specifically for the ExaSHARK and GASPI combination, we can see that both GASPI itself and the GASPI+ExaSHARK combination are still young and much improvements in the coding and the library are possible. One example of this is to use ExaSHARK+MPI for the communication patterns of ExaSHARK where this is more suited, and use ExaSHARK+GASPI for those parts where asynchronicity and performance are more needed.

## 5. ACKNOWLEDGMENTS

This work is part of the European project EXA2CT. EXA2CT aims at discovering solver algorithms that can scale to the huge numbers of nodes at ExaScale, developing an ExaScale programming model that is usable by application developers and offering these developments to the wider community in open-source proto-applications as basis to develop real ExaScale applications.

## 6. REFERENCES

[1] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.

[2] J. M. Cecilia, J. M. García, and M. Ujaldon. CUDA 2D Stencil Computations for the Jacobi Method. In K. Jónasson, editor, *PARA (1)*, volume 7133 of *Lecture Notes in Computer Science*, pages 173–183. Springer, 2010.

[3] I. Chakroun, T. Vander Aa, B. De Fraine, T. Haber, R. Wuyts, and W. Demeuter. ExaShark: A scalable hybrid array kit for exascale simulation. In *23rd High Performance Computing Symposium (HPC 2015)*, 2015.

[4] D. Grünewald and C. Simmendinger. The GASPI API specification and its implementation GPI 2.0. In *7th International Conference on PGAS Programming Models*, 2011.

[5] Intel. Using intel math kernel library for matrix multiplication. https://software.intel.com/en-us/mkl_11.2_tut_c_pdf.

[6] P. Kogge and J. Shalf. Exascale computing trends: Adjusting to the "new normal" for computer architecture. *Computing in Science and Engineering*, 15(6):16–26, 2013.

[7] F.-J. Pfreundt and D. Stoyanov. Hybrid-parallel sparse matrix-vector multiplication and iterative linear solvers with the communication library GPI. In *WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS*, 2014.

[8] C. Simmendinger, J. Jägersküpper, R. Machado, and C. Lojewski. A PGAS-based implementation for the unstructured CFD solver TAU. In *PGAS 2011: Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[9] T. Veldhuizen. Expression templates. In *C++ Report*, volume 7, pages 26–31, June 1995.

# Towards Resilient Chapel

## Design and implementation of a transparent resilience mechanism for Chapel

Konstantina Panagiotopoulou
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Edinburgh
kp167@hw.ac.uk

Hans-Wolfgang Loidl
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Edinburgh
H.W.Loidl@hw.ac.uk

## ABSTRACT

The exponential increase of components in modern High Performance Computing (HPC) systems poses a challenge on their resilience: predictions of time between failures on *ExaScale* systems range from hours to minutes, yet the prevalent HPC programming model today does not tolerate faults. In this paper, we describe the design and prototype implementation of transparent resilience support for Chapel [1], a parallel HPC language with focus on scalability, portability and productivity, following the Partitioned Global Address Space (PGAS) [2] programming model.

## Keywords

Resilience, Fault tolerance, PGAS, Chapel, Runtime systems

## 1. INTRODUCTION

The use of manycore commodity hardware in a hierarchical structure on modern HPC systems, gives rise to rapidly deteriorating Mean Time Between Failures (MTBF) [3, 4] rates. This leads HPC systems to waste capacity on relaunching entire executions after failures. Many real world applications, from molecular dynamics to simulation algorithms take anywhere from a few hours to days to complete. Over this time horizon, failures can no longer be ignored on ExaScale architectures. Resilience [5] is the ability of a system to maintain state awareness and an accepted level of operational normalcy in response to disturbances, including threats of an unexpected and malicious nature. We address cases of failure, in particular hardware failure of one or multiple nodes, during execution on a distributed setup. Our goal is for Chapel programs to terminate successfully in the presence of failures.

In this work we embed support for resilience into Chapel's runtime system (RTS) using detection and recovery mechanisms together with automatic task adoption. Our goal is to provide transparent built-in resilience to the end user (Chapel programmer), without the requirement of extra programming effort. Our design employs data redundancy, retaining information about exported work and a dedicated resilience communication protocol to recover from failures. This design provides flexibility and scalability on a small set of assumptions, following similar design principles to Resilient X10's *Place-Zero Based Finish* [6]. Resilient task parallelism support is enhanced by employing non-blocking communication and by enabling asynchronous recovery of failed tasks on non-failed locations/nodes in the system.

## 2. CHAPEL OVERVIEW

Computation in Chapel is expressed using *tasks* that can be executed in parallel. The target architecture is abstracted via the concept of *locales*: a unit with storage and execution capabilities, such as a multi-core processor. A multi-locale program starts execution on Locale 0 and scales out to other locales, building the locale tree. Below, we provide a brief overview on aspects of relevance to the support for resilience.

### 2.1 Language Constructs

The `on` construct is the main language mechanism for task migration, used by the Chapel programmer to explicitly control locality of the executed task. The migrated task is a logical continuation of the initial task at a different place in the system. The body of an `on` block is viewed as a single blocking task by the controlling thread of the parent locale.

The three language constructs that create parallel tasks are `begin`, `cobegin` and `coforall` statements. Unstructured parallelism introduced by `begin` creates a new task, for execution on a new thread, while the rest of the program continues. The result of the `begin` statement is returned at a later point in the execution. Completion is tracked either by an enclosing explicit synchronisation block or the implicit synchronisation of the *main* function.

With the block-structured task creation employed by `cobegin` a new task is created for each statement in the block, while `coforall`, `cobegin`'s equivalent in loop form, creates one task per iteration. For `cobegin` and `coforall`, there exists an implicit synchronisation point at the end of the block/loop, while the tasks are launched asynchronously. The control flow returns when all tasks have reached the synchronisation point and the program continues.

### 2.2 Runtime Structure

Chapel's runtime system (RTS) is the lower level of the software stack, supporting language constructs and program activities. The runtime is composed of a set of layers, written in C, and standardised interfaces, written in Chapel, and relies on third-party services (Figure 1). The main layers are: communication, tasking, threading and memory.

### 2.3 The GASNet Communication Layer

Chapel uses *GASNet* [7] as the default instantiation of the communication layer. GASNet is a network- and language-independent interface for Global Address Space languages. It is portable and supports multiple low-level communication

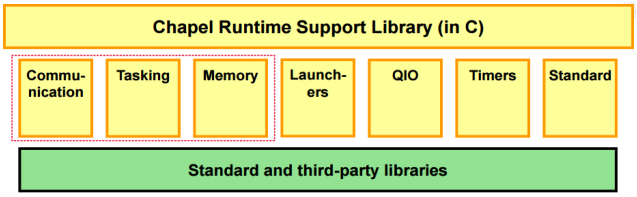


Figure 1: Chapel's Runtime Structure

conduits (e.g UDP, MPI). The *Active Messages* (*AM*) [8] interface, used on top of the conduits, is formulated as logically paired request and reply operations. Most GASNet core functions will return zero on success or an error code. If any node of a GASNet job crashes, aborts, or suffers a fatal hardware error, GASNet attempts to terminate the remaining nodes in a timely manner to prevent creation of orphaned processes.

## 2.4 Internal Synchronisation Constructs

Tasks are added to *task lists* for execution and all threads dispatched for the list's parallel scope can be rescheduled and reused.

```
class _EndCount {
    var i: atomic int,
    taskCnt: taskCntType,
    taskList: _task_list = _nullTaskList;
}
```

Listing 1: The endCount class

Internally, Chapel uses `endCount` objects (Listing 1) to track the completion of parallel tasks. An `endCount` is allocated at the beginning of each synchronised block and its atomic counter is increased before launching a new task. A reference to the `endCount` is passed to the wrapper of each task and the counter is decremented on completion. The controlling thread of the block waits on completion of the task(s) (i = 0) and frees the corresponding `endCount`. The main function itself is governed by an `endCount` object.

# 3. DESIGN OUTLINE

System failures may occur due to hardware fault, software crash or communication loss. We generalise the concept of node failure as anything that prevents a node from communicating with other nodes in the system. In the context of Chapel, using the *flat* locale model (one locale per node), we realise node failures as locale failures.

Parallelism and locality are orthogonal in Chapel: the constructs discussed in Section 2.1 can be arbitrarily combined. Scale-out task parallelism is expressed as combination of `on` constructs (blocking fork operation) and task parallel constructs (`begin`, `cobegin`, `coforall`), resulting in non-blocking fork operations. Our design focuses on the RTS, particularly on the communication layer and the internal modules. We examine task migration, expressed both with blocking and non-blocking fork operations.

With our support for resilience in the RTS, we aim for detection and possible recovery from failures. Our design objective is to allow programs to complete execution in the presence of failures. We make use of data redundancy techniques (Section 4.2), identifying the appropriate data to store and a resilient storage (internal to the RTS) for this purpose.

*Assumptions.* We base our design on the following set of assumptions:

- Locale 0 is failure free and acts as resilient storage for redundancy and data retrieval;
- resilience is only supported during execution of the user's code; we argue that errors during initialisation cannot lead to a fault-free execution and we consider such errors fatal;
- a failing locale explicitly notifies of its failure and the delivery of these notifications is guaranteed; we plan to modify this behaviour in the future with the use of an out-of-band signalling mechanism;
- the body of a task on a failing locale needs to execute till completion or not execute at all; due to the difficulty of tracking and shutting down tasks on the lower layers (tasking and threading);
- we require that the communication network does not fail or that any such failures lead to fatal errors

# 4. IMPLEMENTATION

## 4.1 System Specification

For our implementation we use Chapel's version 1.9.0 built with *gasnet* (v1.22.0); *flat* locale model; *fifo* tasks over POSIX threads; *default* memory (standard C *malloc* commands) and *intrinsics*. We use the GNU compiler suite (gcc v4.4.7) and the *amudprun* launcher on a 64-bit Linux platform.

## 4.2 Data Structures

We implement three new data structures for storing the *status* of locales and the *task descriptors* of migrated tasks.

`failed_table`: We implement an array of length equal to the number of locales in the configuration (*numLocales*). The array is stored on Locale 0 and records failures detected during execution. We store tuples of node id's and status variables (`failed_t` struct). The array is updated on reception of a `FAIL` (or `TIMEOUTNB`) signal from remote locales or on local detection of a failure.

`transit_msg_list` and `transit_arg_list`: We introduce two linked lists to capture the descriptors of migrated tasks and we store context information (functions and data) for each remote task in the execution. Data redundancy is employed for use in task relaunching, in the occurrence of failure. The lists are stored on Locale 0 and are updated through the handlers of `IN_TRANSIT` and `IN_TRANSIT_DEL` signals. The main operations on the lists are *append*, *delete* and (currently) linear-time *lookup*. We aim to replace the linked lists by a hash table with fixed lookup overhead in future work.

## 4.3 Communication Protocol

Remote fork operations are implemented using the `on` construct, which is realised as switching to a different (from the current) locale to execute a task. The `on` construct belongs primarily to the communication layer, but the tasking layer plays a subsidiary role to support the acknowledgement-based system.

The communication layer handles the transfer of the task (and data) to and from the remote location while the child locale launches the body of the `on` statement by calling the `chpl_task_startMovedTask()` function of the tasking layer. The target locale runs a synthetic task; calls the body of the `on` statement and on return it sets a flag on the initiating locale. The control on the parent locale waits on this flag before proceeding.

Listing 2 demonstrates a distributed serial program, using a blocking fork while Listing 3 demonstrates a distributed parallel program, using a non-blocking fork operation. Fork operations are also used during initialisation of multi-locale executions to establish communication between Locale 0 and all other locales in the configuration (e.g broadcasting of global variables, GASNet initialisation).

```
writeln(here.id);       // locale 0
on Locales[1] do
  writeln(here.id);     // locale 1
writeln(here.id);       // locale 0
```

Listing 2: Distributed serial Chapel program

```
begin on Locales[1] do
  writeln(here.id);     // locale 1
on Locales[2] do begin
  writeln(here.id);     // locale 2
writeln(here.id);       // locale 0
```

Listing 3: Distributed parallel Chapel program

*Blocking Fork.* Figure 2a demonstrates the functionality of the blocking fork function (`chpl_comm_fork()`), called on the parent locale. A wrapper (`fork_t`), capturing the function and arguments, is sent for remote execution via a `FORK` signal. On the child locale, the signal is handled in the `AM_fork()` handler, which schedules the task for local execution and replies with `SIGNAL` (acknowledgement). The signal is received on the parent and the locale exits the block-wait.

In Figure 2b we demonstrate the resilient version of the function[1]. We add a `TIMEOUT` signal (Figure 2b, Message 4b) for the child locale to notify the parent of local failure. Due to the asynchronous nature of the UDP protocol, we choose not to use a heart-beat function in order to avoid possible overheads. Furthermore, we add a UNIX signal handler to track status changes of the locales. On receipt of the `TIMEOUT` signal, the parent notifies Locale 0 (`FAIL`, Figure 2b, Message 5b) to record the newly detected failure.

In order to make use of the status information, available on Locale 0, the parent requests an update on the child's status before launching the fork operation. This is a proactive detection mechanism to help save extra communication, since the child locale may have been detected dead in a previous operation. We handle the exchange of information between the parent and Locale 0 using `FAIL_UPDATE_REQUEST/REPLY` signals (Figure 2b, Messages 1 & 2).

In both cases, recovery is handled on the parent locale by executing the task wrapper. The motivation behind this design is the availability of the evaluation context (function and arguments) on the parent locale. Failure of the parent locale, is handled on its immediate living ancestor (as a child failure) one level higher on the locale tree.

*Non-Blocking Fork.* In the non-blocking fork operation (Figure 2c) the parent locale launches the remote fork operation and exits without waiting for an acknowledgement. Instead, the `endCounts` mechanism (Section 2.4) is employed to track completion of the remote task(s).

Figure 2d demonstrates the non-blocking fork operation with the modifications to support resilience. Here we use the two linked lists introduced in Section 4.2. We use the first list, `transit_msg_list`, to capture the core part of the task descriptor with information on the parent and child locale and the function ID, while the second list, `transit_arg_list`, is used to store the variable-sized arguments of the task.

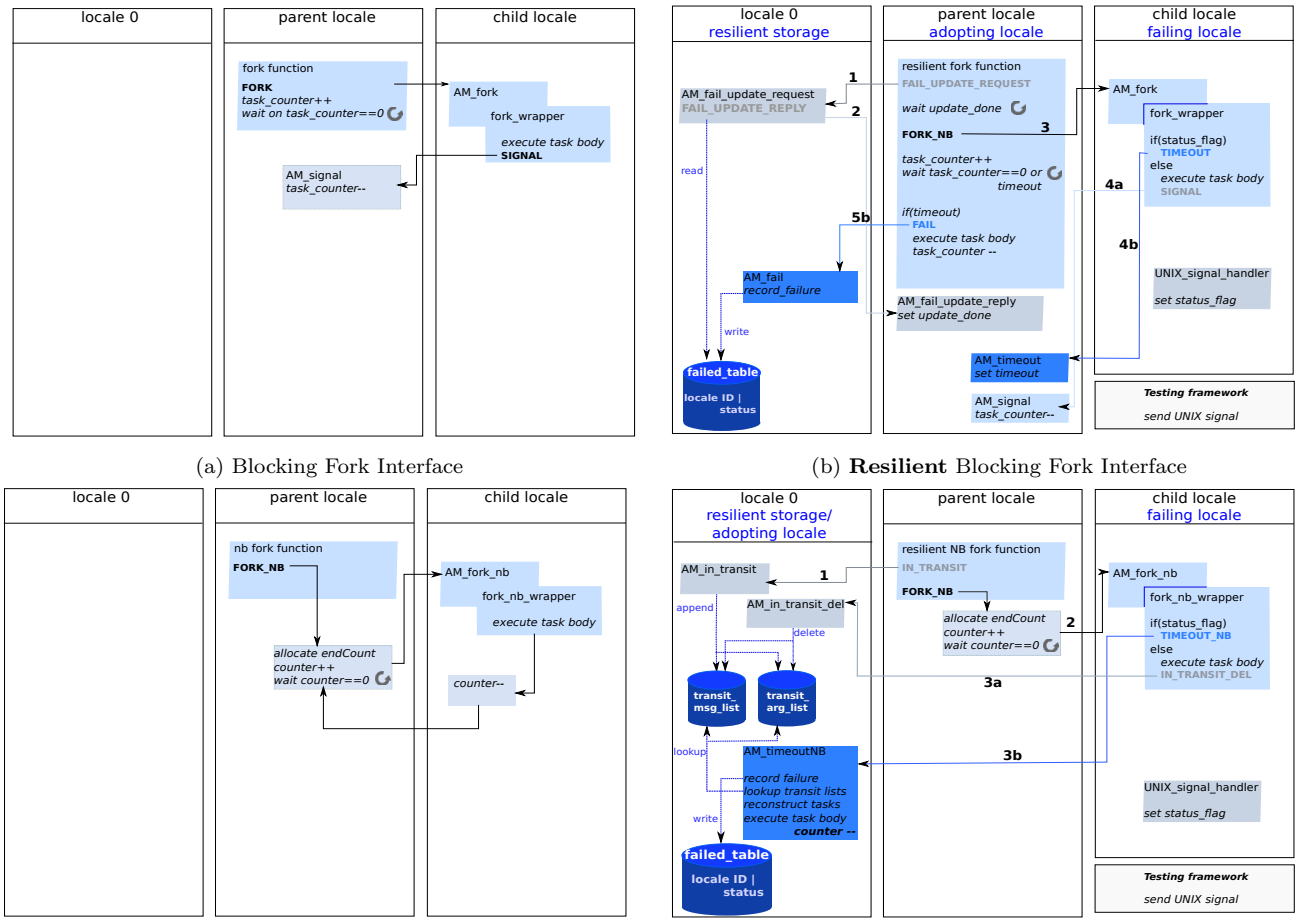
The parent sents the context to Locale 0 before launching the non-blocking fork operation (`IN_TRANSIT`, Figure 2d, Message 1). The arguments are copied using `memcpy`; an expensive but essential operation in order to retrieve data safely since it is unsafe to use pointers to the local memory of a locale that may fail (parent locale). The memory is freed on execution of the task, on reception of the `IN_TRANSIT_DEL` (Figure 2d, Message 3a) signal from the child locale or locally, if recovery took place.

In the resilient version, in the occurrence of failure, the failing locale notifies Locale 0 (Figure 2d, Message 3b). In turn, Locale 0 employs mechanisms to reconstruct the task from its in-transit lists; it then records the failure of the child in its local array and re-executes (recovers) the task.

The motivation for employing recovery on Locale 0 is based on the non-blocking nature of the operation. In contrast to the blocking case, the parent has possibly completed the fork operation by the time detection of the failure occurs. As a result, all memory references of the context of the remote task have been discarded. Furthermore, storing information on the parent locale would be unsafe as the parent is also susceptible to failures and it could lead to permanent loss of the essential information to relaunch the lost task.

For load balancing purposes we may choose to re-execute the task either on a new thread on Locale 0 or re-launch the task in a non-blocking manner on a remote locale (possibly the parent locale), known to be alive. Our current implementation adopts the first strategy, to avoid creating more communication and data copying, resulting by a new non-blocking fork operation (see Implementation Limitations, Section 7).

[1]The implementation is available for downloading at `http://www.macs.hw.ac.uk/~kp167/resilience/`




(a) Blocking Fork Interface

(b) **Resilient** Blocking Fork Interface

(c) Non-Blocking Fork Interface

(d) **Resilient** Non-Blocking Fork Interface

Figure 2: Resilient Communication Protocol: Design Overview of Blocking and Non-Blocking Fork Operations

# 5. RESULTS

## 5.1 Testing Framework

Our testing framework is signal-based to flexibly simulate locale failures and assess functionality of the prototype implementation. We use the following two testing modes: **all**, simulating failure of every locale except Locale 0 (stress test) and **rand**, simulating failure of a random number of locales. We assess both functionality and overhead of our prototype implementation, while a current limitation is the inability of the framework to simulate failures at different times during execution and the requirement to allow a short time frame in the beginning of the execution to send the signals.

Our experiments were performed on a 32-node Beowulf cluster (256 cores) connected via Gigabit ethernet network, with each node consisting of: two quad-core Xeon E5506 2.13GHz, 12GB of main memory and three-layered cache memory topology. For our tests, we performed 30 iterations.

## 5.2 Functionality Tests

Figure 3 demonstrates functionality of the resilient blocking fork implementation using the *constructed programs*, shown in Table 1, below. We use the *Monte Carlo Pi approximation* as the remote long-running task and we run experiments with **all** and **rand** modes. We get the expected success rates of 100% on both testing modes confirming functionality, when excluding the failures of the testing framework.

Failures of the testing framework are marked as *missed* (Figure 3). When launching a Chapel program, usually the first node on the *hostlist* acts as Locale 0, but this is not specified beforehand. These cases, reaching 30%, are the result of simulating failure on Locale 0. Complying to our assumption that Locale 0 is failure-free, these signals are discarded.

## 5.3 Overhead

In Figure 4, we demonstrate the overhead of our resilient blocking fork implementation on the *constructed programs* of Table 1. We use the regular (non-resilient) Chapel implementation without failures as baseline and we note overheads between 0.29% and 1.29% for our resilient implementation without failures. This overhead is due to the additional communication and management of the added data structures. The overhead rates on failure (all, rand) depend highly on the structure of the locale-tree (e.g. shallow/deep nesting).

# 6. RELATED WORK

In the context of PGAS languages, *Resilient X10* [6] is a complete implementation of X10 with added support for resilience. It uses distributed termination detection mecha-

**simpleons.chpl**

```
on Locales[1] do
    monteCarlo();
on Locales[2] do
    monteCarlo();
```

**simpleontest.chpl**

```
on Locales[1] do {
  monteCarlo();
  on Locales[2] do
    monteCarlo();
}
```

**three_on.chpl**

```
on Locales[1] do {
  monteCarlo();
  on Locales[2] do{
    monteCarlo();
    on Locales[3] do
      monteCarlo();
  }
}
```

**two_two.chpl**

```
on Locales[1] do {
  monteCarlo();
  on Locales[2] do
    monteCarlo();
}
on Locales[3] do{
  monteCarlo();
  on Locales[2] do
    monteCarlo();
}
```

**back.chpl**

```
on Locales[1] do {
  monteCarlo();
  on Locales[2] do{
    monteCarlo();
    on Locales[1] do
      monteCarlo();
  }
}
```

Table 1: Constructed Programs: used for functionality and overhead testing

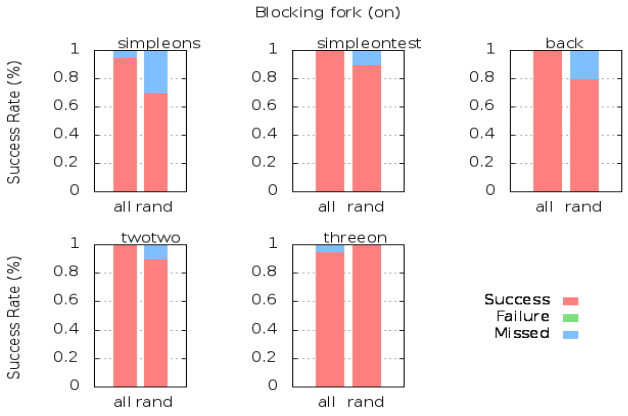



Figure 3: Resilient Blocking Fork - Functionality Tests

nisms, currently supported for the sockets implementation of the communication layer. Resilient X10 uses, in the latest version, X10-level resilient storage with data redundancy on multiple nodes, allowing fatal errors when all the nodes that store copies fail. This introduces extra communication costs but also allows performance tuning by controlling the manner that data is stored.

*Erlang* [9], is a declarative language with built-in resilience, applying a scheme of relaxed fault tolerance and following the *"let it fail"* design pattern where recovery is only em-

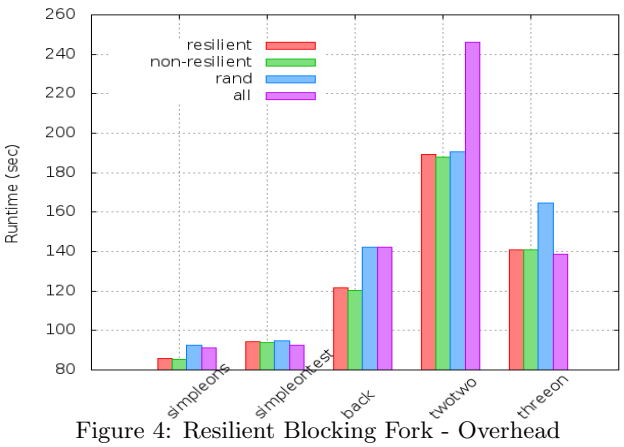



Figure 4: Resilient Blocking Fork - Overhead

ployed where possible with the use of process links, monitors and supervisor tasks. Furthermore, *Spark* [10] employs *resilient distributed datasets* (RDD's); a read-only, memory-persistent collection of objects partitioned across machines that can be rebuilt. Spark expands on the work of MapReduce [11] and Dryad [12] in the direction of acyclic data flows, supporting applications that reuse data across multiple operations.

Finally, we identify significant research on technical aspects, such as efficient data-redundancy and message logging [13], fault tolerance for work-stealing computations [14] and distributed termination detection mechanisms [15].

## 7. CONCLUSIONS

We have presented the design and initial implementation of basic resilience support embedded in Chapel's runtime system. Our design is completely transparent, using automatic task adoption and data redundancy techniques based on a small set of realistic assumptions.

Our preliminary results focus on functionality of the blocking fork operation with 100% success rates, indicating normal program termination and compliance with the task adoption strategy. The implementation also succeeds in cases of deeply nested fork operations; though this is not a common pattern in Chapel programs. Our results on constructed programs show negligible overhead in the failure-free case, mainly induced by communication and data structures' management.

The implementation of the non-blocking fork operation is currently work in progress, as we explore alternatives to the existing termination detection mechanism for parallel tasks. The most promising direction is *Parental Responsibility Termination Detection* as described in [15].

*Design Limitations.* For applications with strict performance requirements, our strategy of strict resilience may be unsuitable compared to a relaxed resilience scheme where a system could trigger graceful exit on reaching the threshold of parallelism or runtime.

*Implementation Limitations.* We identify fixed-length *vulnerability windows* between the reception of the `FORK` signal on the child locale and the call of the wrapper function, including copying of the computation's context. To address this issue we aim to embed local status monitors to the communication task on each locale; exploiting the existing dedicated thread used for AM polling.

Currently, we do not provide functionality and overhead results for the case of the non-blocking fork. The resilient non-blocking fork implementation, as described in Section 4.3, is functional concerning execution of recovered tasks but fails to track completion on recovery. The main reason is that termination detection is implemented on module level and is controlled by the compiler and not on RTS level.

Ongoing work explores the ideas presented in [15], concerning integration of termination detection mechanisms on the RTS level (communication protocol) for the resilient version of the non-blocking fork operations. This system adds the requirement for asynchronous acknowledgements for all messages (migrated tasks) and assumes failures of parent-children locales as unrecoverable. We also explore alternative solutions, such as adding hooks to the compiler generated code or exploring the memory to identify and copy the corresponding `endCount` objects.

Finally, we do not currently handle user-level distributed data structures. While this will be an important topic to address in the future, here we focus on the backbone of the resilient functionality.

*Future Work.* In future work, we will address advanced distributed task adoption strategies and the integration of resilience with Chapel's default data distributions. We expect that this will require a load balancing policy, to tackle the issue of task recovery occurring closer to the root of the locales tree, with increasing failure numbers.

Finally, we aim to explore further the idea of *evolving systems* [16], enabling "resurrection" of nodes after failure or allowing new nodes to join at later points in the execution, provided that the nodes have been configured as part of the communicable segment during initialisation.

# 8. REFERENCES


[1] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[2] T. Stitt, "An Introduction to the Partitioned Global Address Space (PGAS) Programming Model," tech. rep., 2010.

[3] X. Xie, X. Fang, S. Hu, and D. Wu, "Evolution of Supercomputers," *Frontiers of Computer Science in China*, vol. 4, no. 4, pp. 428–436, 2010.

[4] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Journal of Physics: Conference Series*, vol. 78, pp. 12–22, IOP Publishing, 2007.

[5] C. G. Rieger, D. I. Gertman, and M. A. McQueen, "Resilient Control Systems: Next Generation Design Research," in *Proceedings of the 2nd Conference on Human System Interactions*, HSI'09, (Piscataway, NJ, USA), pp. 629–633, IEEE Press, 2009.

[6] D. Cunningham, D. Grove, B. Herta, A. Iyengar, K. Kawachiya, H. Murata, V. Saraswat, M. Takeuchi, and O. Tardieu, "Resilient X10: efficient failure-aware programming," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 67–80, ACM, 2014.

[7] D. Bonachea, "GASNet Specification Version 1.1," *Univ. California, Berkeley, Tech. Rep. UCB/CSD-02-1207*, 2002.

[8] V. E. Thorsten, C. D. E, G. S. Copen, and S. K. Erik, "Active Messages: a mechanism for integrated Communication and Computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pp. 256–266, ACM, 1992.

[9] J. Armstrong, *Making reliable distributed systems in the presence of software errors*. PhD thesis, The Royal Institute of Technology Stockholm, Sweden, 2003.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX conference on Hot topics in Cloud Computing*, pp. 10–10, 2010.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 59–72, ACM, 2007.

[13] E. Meneses, X. Ni, and L. V. Kalé, "A message-logging protocol for multicore systems," in *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pp. 1–6, IEEE, 2012.

[14] W. Ma and S. Krishnamoorthy, "Data-driven Fault Tolerance for Work Stealing Computations," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, 2012.

[15] J. Lifflander, P. Miller, and L. Kale, "Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection," in *ACM SIGPLAN Notices*, vol. 48, pp. 13–22, ACM, 2013.

[16] G. R. R. Justo and P. R. F. Cunha, "An architectural application framework for evolving distributed systems," *Journal of systems architecture*, vol. 45, no. 15, pp. 1375–1384, 1999.

# HPC and CFD in the Marine Industry: Past, Present and Future

Kurt Mizzi
University of Strathclyde,
100 Montrose Street,
Glasgow
+44 (0)141 548 4911
kurt.mizzi@uni.strath.ac.uk

Paula Kellett
University of Strathclyde,
100 Montrose Street,
Glasgow
+44 (0)141 548 3237
paula.kellett@strath.ac.uk

Yigit K. Demirel
University of Strathclyde,
100 Montrose Street,
Glasgow
+44 (0)141 548 4275
yigit.demirel@strath.ac.uk

Richard Martin
University of Strathclyde,
107 Rottenrow East,
Glasgow
+44 (0)141 548 3265
richard.martin@strath.ac.uk

Osman Turan
University of Strathclyde,
100 Montrose Street,
Glasgow
+44 (0)141 548 3211
o.turan@strath.ac.uk

## ABSTRACT

This paper explores the use of Computational Fluid Dynamics (CFD) applications on High Performance Computing (HPC) platforms from the perspective of a user engaged in Naval Architecture research. The paper will consider the significant limitations which were imposed on research boundaries prior to present HPC capabilities, how this impacted development in the field and the implications for industry. One particular example is the costly experimental testing which, due to resource constraints, is generally restricted to model scale. It will then present an overview of the numerical simulation capabilities using current HPC performance and capability.

With the increase of computational power and capacity, CFD simulations are proving to be more accurate and reliable. Being relatively cheaper and more time efficient, numerical methods are becoming the preferred choice within the industry compared to traditional experimental tests. Nevertheless, certain experimental procedures cannot be numerically replicated with the current levels of computational capacity.

The future needs and challenges of research and development will be outlined and discussed, highlighting the significant impact exascale computing will have in the field.

## Keywords

Computational Fluid Dynamics (CFD), HPC Application, HPC Capability, Capacity Implications

## 1. INTRODUCTION

The aim of this paper is to provide an understanding of the importance of HPC and its contribution to research development in the area of marine research. Several examples of research fields which are extremely important in the marine sector, both academically and also crucially for the industry, will be used to further these discussions. More specifically, the ability to directly simulate ship resistance, the effects of fouling on ship performance, and underwater noise predictions at model and full scale will be discussed. HPC capabilities that allow the use of CFD as an optimization tool, with the example of Propeller Boss Cap Fin (PBCF) designs, will also be outlined.

Although High Performance Computing has come a long way, current HPC power still limits certain analyses and research, even though some of the current capability would never have been thought to be possible even a few years ago. A typical example would be propeller cavitation which is best predicted using a Detached Eddy Simulation (DES) solver that requires high computational capability. This indicates that computational capability dictates research boundaries as well as processes, which can be significant in an industry which tends to be conservative and reactive.

Although maximising HPC capabilities broadens the horizons for research and analysis, one common problem is convincing industry to rely on numerical simulations. Although these methods may be beneficial financially, while enabling the generation of high levels of useful data, quality assurance of the product is always a prime concern and therefore such new procedures need to be well proven and presented. This paper will also discuss earning the trust of industry because ultimately it is industry that drives and funds research in this sector, enabling it to progress.

## 2. CFD AND ITS IMPORTANCE IN NAVAL ARCHITECTURE

Generally, during the initial stages of current ship design processes, various design analyses are carried out using numerical approaches typically known as CFD methods. Before CFD procedures were available, all investigations were carried out experimentally in facilities, such as the one shown in Figure 1, which were very time consuming and expensive and which therefore constrained research to the analysis of a limited number of designs. Apart from the complexity of involving a number of stakeholders in experiments, they also incorporate a number of assumptions, errors and tolerances, the most common being the issue of scaling. All experimental studies are carried out at model scale, introducing scaling errors in the extrapolation of the results to full scale. The ability to simulate at full scale is very important as it removes the errors and correction factors associated with model scale testing due to scaling effects, and in cases where improvements in the range of 2-3% are being sought, it is very beneficial to have the sources of error minimised as far as possible. These errors arise due to the fact that the viscous effects of water cannot be scaled, and will therefore be the same at both model and full scale.

CFD technology is capable of predicting various parameters such as resistance, motion, free surface capturing, manoeuvring performance etc. some of which is very hard to predict in experimental procedures due to the need for instantaneous visualization or sophisticated measurement tools.

However, as previously explained, the industry still considers experimental investigation to be more reliable. Therefore, to save on costs, the design process is initially carried out using simulation based design (CFD) analyses followed by experimental studies at the final stages for validation.

High fidelity CFD tools have enhanced research, pushing boundaries in all aspects of the marine industry. For this reason,

researchers should focus on developing and improving next generation simulation based design in order to open new horizons and opportunities in marine research. This can generally be achieved by improving the numerical solvers as well as maximising HPC capabilities.

**Figure 1. Experimental Tests**

## 3. PAST AND PRESENT COMPUTATIONAL LIMITS

In the very early stages of the application of CFD to Naval Architecture problems, limited computational power and capacity restricted most analyses to potential flow methods. Typically, these methods did not take into account the viscous effects of the fluid which resulted in a loss of accuracy and an over simplification of the problem.

Therefore, the use of more simplified approaches was common and this in turn meant that the capture of complex flow phenomena was much more difficult. In time, with the advance in computational power, Reynolds-Averaged Navier-Stokes (RANS) equations based CFD simulations became more feasible allowing the use of more complex and representative turbulence models. These simulations were time consuming and computationally expensive thus constraining the cell count for meshes to the extent that only model scale could be simulated, which produced similar errors to those observed in experimental results. This also prevented detailed representation of the geometry and proper physical predictions. In addition, simulations took longer to converge, resulting in the fact that analytical and parametric studies were limited and optimization procedures were not really an option. Full scale simulations are still not as common as model scale and when they are applied, certain methods and approaches are typically used (such as mesh refinement regions) to limit the overall size of the mesh and computational domain.

Recent improvements have also led to the introduction of Large Eddy Simulation (LES) and Direct Eddy Simulation (DES) methods which are less common again, but can provide more accurate results through the application of fewer assumptions in certain cases. These kinds of simulation are required for applications such as cavitation, which is discussed later, however they remain time consuming and computationally expensive in comparison to RANS approaches. One promising approach is to blend both the RANS and DES models to produce a compromise between accuracy and execution time.

With the advance towards Exascale Computing, more advanced numerical methods may be introduced. There will likely be a move towards Direct Numerical Simulation (DNS) approaches, where the use of assumptions to simplify the problem are removed, making the results yet more accurate but significantly increasing the calculations that need to be carried out at each time-step.

A comprehensive discussion on the recent advances in CFD, and potential future trends and developments, is presented in [1].

## 4. CURRENT STATE OF THE ART

The following sections will present the current state of the art in CFD using three case studies from industry-relevant naval architecture problems.

### 4.1 Naval Architecture Case Studies

The examples discussed in this section outline topics in naval architecture which have been addressed at University of Strathclyde using current CFD capabilities, particularly the commercial Star-CCM+ software. Their advances over past capability as well as their limitations will be highlighted.

#### *4.1.1 Ship Resistance and Fouling*

Shipping has been, and still is, one of the most important methods of transport, with more reliance and importance now being placed on this mode of transport as a consequence of advancements in shipping technology and the ability of ships to hold and store increasing capacities of goods. However, these improvements and features bring some problems to the industry due to an increase in fuel consumption, which is detrimental to the environment and which erodes company revenues. Although other forms of fuel power exist, such as wind energy and solar power, carbon-based fuel is currently the only way for ships to run effectively. For this reason, minimising fuel consumption is crucial for shipping companies.

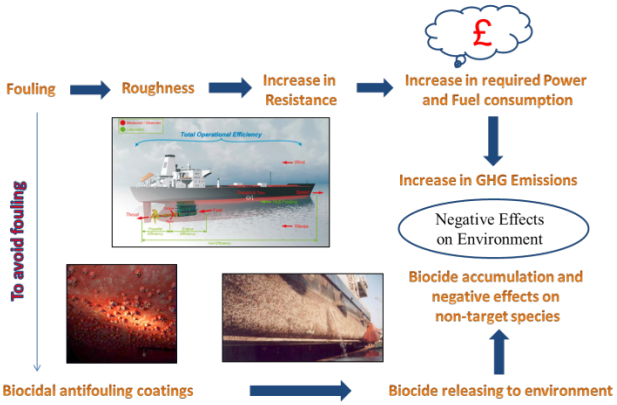



**Figure 2. Fouling Analyses Motive**

A major challenge is to relate technologies, such as antifouling coatings and the effect of biofouling, to ship resistance and fuel consumption, in order to evaluate their effects on energy efficiency and hence $CO_2$ emissions (see Figure 2). While retrofitting existing ships with new antifouling paints will improve their energy efficiency, it is equally important to accurately model the potential effects of biofouling on ship resistance and to demonstrate the importance of the mitigation of such effects through scientific research.

Two different CFD models were therefore proposed as outlined in [2] and [3] for the prediction of the roughness effects of antifouling coatings and biofouling on ship resistance. It is worth pointing out that the total number of cells of such CFD simulations is very high since the roughness of the hull surface requires to be represented in the order of micrometers (μm). Additionally, the time-step used in such simulations has to be kept

very small due to the very complex and unstable nature of the phenomenon. The capability of the computer system is therefore of great importance for researchers and engineers to be able to carry out reliable and feasible CFD analyses.

The effect of the total cell numbers on the results of such analyses are highlighted in the following table. For each mesh configuration, the frictional resistance coefficients ($C_F$) of a flat plate of ship length, coated with an antifouling coating are listed in Table 1 [2].

**Table 1: $C_F$ results of antifouling coatings at different mesh configurations [2]**.

| Mesh configuration | Total No. of Cells | $C_F$ (CFD) |
|---|---|---|
| Coarse | $1.8 \times 10^6$ | 0.001574 |
| Medium | $2.5 \times 10^6$ | 0.001576 |
| Fine | $4 \times 10^6$ | 0.001584 |
| Very Fine | $5.5 \times 10^6$ | 0.001584 |

From Table 1 it is evident that the results converged very well provided that the total numbers of cells are sufficiently high. Such simulations could not have been run without the existence of High Performance Computers.

It is significant that the modelling used in these simulations was based on available experimental data and was achieved using particular assumptions on the flow properties. Although the proposed CFD model is a reasonable method to predict these effects, the effects of biofouling on ship resistance can only be precisely predicted using a means of geometrical modelling of relevant organisms. Unfortunately, geometrical modelling of such small organisms in detail, along with the other complexities of the ships' systems such as a rotating propeller, is beyond the scope of current HPC capability. Additionally, spatial inhomogeneity of fouling on ship hulls is another challenge for modelling which is still beyond the reach current computational capabilities.

### *4.1.2 Ship Radiated Underwater Noise*

A topic which has recently become important in the marine industry is underwater noise from anthropogenic noise sources, in particular shipping, and its impact on marine wildlife. As outlined in [4], CFD has been used to predict the underwater noise of a moving ship and rotating propeller at given locations. In the past, simulations would not have allowed for a rotating propeller to be modelled. This meant that a steady state approximation was therefore applied to an unsteady problem. However, these kinds of simulations are now feasible, as in Figure 3, and have been carried out at full scale, with the benefit of removing the additional errors associated with scaling. However, given the limits of current computational capability, the vessel had to be simulated moving in calm and very deep water, with the propeller operating in a non-cavitating condition. Therefore, propeller noise and flow noise could be captured but the significant contribution to underwater noise from cavitation could not be addressed in this case.

Cavitation is the phenomenon whereby the water at the propeller effectively boils due to the pressure differential cause by the rotation of the propeller above a certain speed. Almost all propellers will cavitate at certain operational conditions. This phenomenon is of considerable concern in the marine industry as it leads to reduced propeller efficiency, high levels of underwater noise, and in more extreme cases, damage to the propeller and hull. An example of a cavitating propeller can be seen in Figure 4 below. With current CFD capabilities, the higher complexity solvers and the fine computational meshes required for simulations of cavitation only allow them to be carried out at model scale, and generally with only partial hull geometry and a calm water surface represented. The significant additional demands in simulating this phenomenon in more realistic conditions are beyond current computational capability.

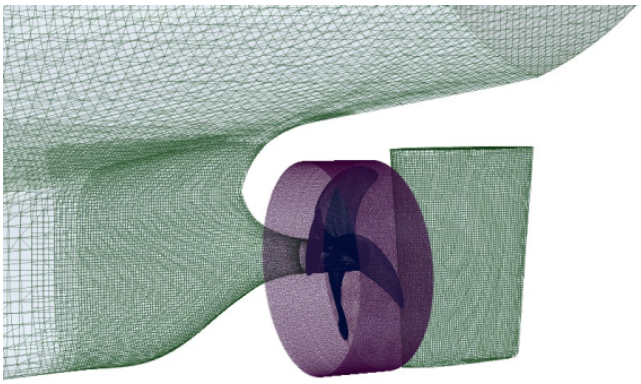


**Figure 3. Rotating Propeller Simulation**

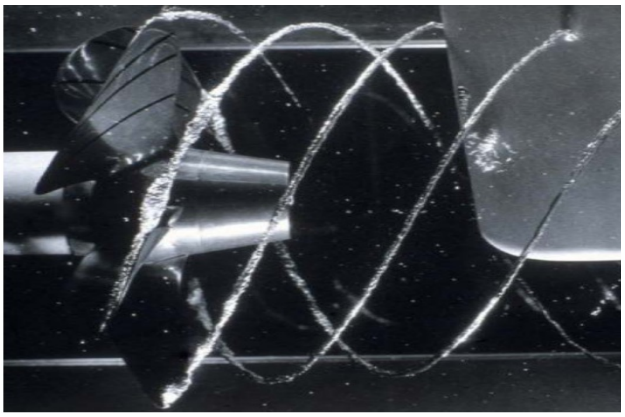


**Figure 4. Propeller Cavitation** [1]

### *4.1.3 Propeller Boss Cap Fins Optimisation*

As outlined above, there are various methods available to improve the propulsion efficiency of a vessel. With recent development in CFD procedures and HPC capability, designs with improved propulsion efficiency can be achieved by carrying out hull, propeller and retrofit device design optimisation procedures. In particular, one well established retrofit technology is the Propeller Boss Cap Fin (PBCF), which is a post swirl fin that is installed onto the boss cap of the propeller as demonstrated in Figure 7.

Due to the limitation of available tools, previous research was typically conducted by analysing different PBCF design parameters independently and seeking the local optimum by assessing different case studies. However, research was recently carried out seeking to optimise the PBCF design in order to find the global optimum by taking into consideration a number of related parameters [5]. This was made possible by the available capacity of ARCHIE-WeSt, the High Performance Computer at University of Strathclyde, and use of the available software

namely Star-CCM+ (the numerical CFD solver) and Friendship-Framework as an optimiser. These advanced numerical tools together with a large-scale computational resource allowed the analysis of 120 different PBCF designs with the optimal fin producing an open water efficiency improvement of 1.3%, which is very significant in this field. Results for the open water efficiency for the different designs can be seen in Figure 5 below. Similar methodology and approaches can be applied to different energy saving devices or case studies to suit different requirements.

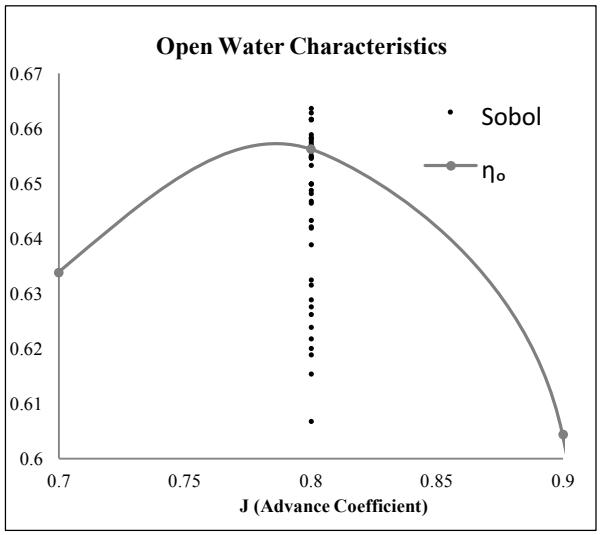



**Figure 5. PBCF Optimisation Study**

However, optimisation options and procedures are endless, for example, defining one or more constraints or seeking a single or multi-objective approach. Processes might also be computationally expensive and time consuming and therefore careful selection of a robust and efficient system must be taken into consideration. A common preferred approach is to run various designs with a less demanding numerical model, followed by further optimisation on selected designs using more accurate simulations. This demonstrates that optimisation procedures could be further exploited with the development of appropriate tools and resources. With increasing computer capacity, future enhancements could be extended by adding more design variables or, for example, a multi-objective optimisation approach could be used to seek a ship geometry providing maximum energy efficiency and reduction in hub vortex cavitation. Additionally it is conceivable that more detailed analyses could be carried out which might result in different optimal fin geometries altogether (Figure 6).

A particular limitation of these studies is the lack of a suitable cavitation model in the numerical simulation which would require more advanced modelling, numerical approaches and greater computational power. Since cavitation adversely affects propeller characteristics, more effort will be focused on implementing a cavitation model in the open water simulation.

The use of automation in the optimization procedure reduces time and user interaction which can be considered as a benefit, however it also provides less control for the user and requires constant monitoring for quality assurance purposes. Nevertheless, it saves a great deal of time allowing the extensive study of a system or technology within a limited time scale.

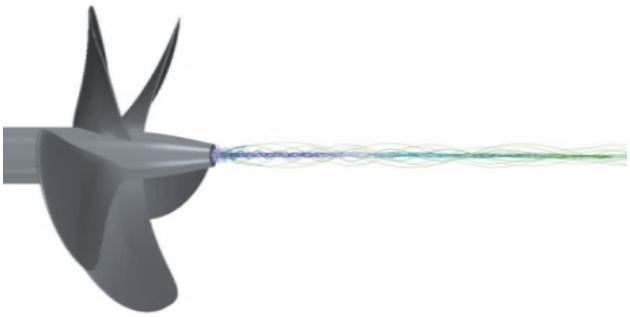


**Figure 6. Hub Vortex Cavitation**

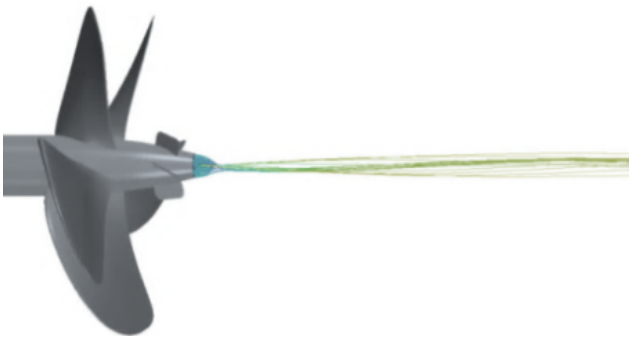


**Figure 7. PBCF**

## 5. FUTURE POSSIBILITIES

As has been outlined in the sections above, the advances that have already taken place in computational capability and capacity have enabled much more complex and realistic simulations to be carried out. This has resulted in more accurate predictions in much shorter timescales. Logically this then implies that further increases in computing capability to exascale and even beyond will extend the possibilities for researchers and industry to further improve the simulation of real world problems. Access to much greater capability will allow for the use of larger and more refined meshes within CFD simulations, which in turn leads to more accurate solutions of problems. Increased capacity will also allow for increasingly complex scenarios to be solved, as they will enable more parameters to be calculated directly rather than being assumed or simplified.

A good example of this is the subject of ship propeller cavitation, as discussed in Section 4.1.2 and 4.1.3. At present simulation of the phenomenon is typically only carried out at model scale and with only part of the ship's hull present in order to try to reduce computational complexity wherever possible. However the availability of much higher levels of computational capability would mean that simulations could be carried out in much more realistic conditions with a full scale propeller and full hull present, and operating in a seaway which is representative of an actual sea state. Such simulations would enable researchers, and more crucially industry, to gain a much better understanding of the phenomenon and its implications for real operational conditions in order to take more informed decisions on how to address it.

Further advances could also be directed towards using LES numerical models for cavitation, simulating cavities and eddies. Although it is found to be more accurate for certain engineering applications, LES is not currently very common as it is very computationally expensive.

Although optimization methods are available, their use is not common practice due to the computational capacity required. Hull and propeller optimization using RANS simulations would greatly benefit the industry allowing the analyses of multiple designs at a reasonable expense when compared to experimental procedures. Looking further into the future there is likely to be a venture into optimization methods in real operational conditions and optimization using LES or DNS numerical models.

As explained above, the increase of computational power, both in terms of capability and capacity, introduces endless possibilities to the marine industry and research in general.

However looking at short term improvements, the ability to create mesh configurations with higher cell numbers would increase accuracy, allow more full scale simulations and would make it easier for engineers to satisfy the validation and verification requirements (V&V) of their simulations which in turn would create increased credibility for their work and generate more trust from within industry.

One other significant improvement would be to allow more interaction between the HPC and the user i.e. creating a better graphical user interface allowing for interactive simulations. Some CFD engineers find it best to monitor their work while the job is running ensuring that the simulation is running adequately, and sometimes also making modifications during the run. This is commonly practiced when carrying out minor jobs on a personal computer. However, when running jobs on a HPC, although some visual elements are allowed, this is somewhat limited. Extending and expanding visual capabilities in HPC would definitely help engineers carry out their work more efficiently.

However in the present economic climate, researchers and industry are not at liberty to research and develop whichever techniques and capabilities they choose over an undefined timescale. Nor would access to such computational capability be granted for free. The following sections will discuss the implications of access to such capability from the perspective of research and of industry.

## 5.1 Technical Implications

Stern et al. outlined three HPC challenges in state of the art CFD simulations; System Memory, Interconnection and Input/output [6]. They explain that 10% of the current system memory (RAM), which is generally 2GB, is dedicated for system usage. In high fidelity complex simulations, a good portion of this is used to store the data to be solved, leaving limited memory for the solver. In some cases, marine applications require the processing of vast amounts of data which do not fit within a single computer node, which may generally be equipped with tens of processor cores. Future advances, such as Exascale computers, may allow transition from a multi–core to many-core systems equipped with many more cores on a single node. Therefore the continuous increase of number of processors within one node will allow more complex simulations with higher volumes of data to be solved within a single shared-memory system. On the other hand, due to the associated costs, a decrease in memory size per core is predicted and therefore it is vital for the CFD engineer to minimise memory usage in CFD simulations and processes.

The interconnection between the nodes is also an issue that requires attention. The network bandwidth and communication latency are the determining factors for network interconnect performance. Therefore while computer engineers are likely to focus on avoiding or reducing latency and increasing network bandwidth, flow simulation engineers should develop solvers that require less communication. In addition, high fidelity flow simulations require processes to read and write vast amounts of data which may result in a great number of nodes or cores to processing in parallel the input/output data. This might make data handling unmanageable and will also be affected by interconnect bandwidth and the performance of the available storage. Consequently, this could deteriorate simulation performance and could be costly.

Such issues require attention not only from the computer engineering perspective but also from the software developers with the former aiming to maximise HPC capability and the latter minimising the need for high data volumes, handling and communication. Together they should aim to improve the system performance and the scalability of the simulations.

## 5.2 Implications for Research

The direction that research takes is as much dictated by political agendas, funding sources and industry requirements as it is by computational capability. Political agendas tend to dictate “hot topics” which are perceived to be of upmost importance to society at a given juncture. This in turn influences the funding which becomes available for research or facilities which support development in these key areas. These developments can then have an impact in two ways: by producing new techniques or technologies which become available to industry, or through highlighting concerns which impact on industry where they lead to standards and regulations. The process then comes full circle as industry turns to the research community, seeking support in dealing with these new discoveries.

In the particular case of significant advancements in computational capability to Exascale Computing, access to such facilities and the funds to develop applications to take advantage of them would not necessarily be directly linked to the advancements themselves. Instead, it would most likely arise as a means of improving current knowledge and understanding of a key politically-supported topic, or investigating a particular concern raised by industry. This in effect means that just because the capability becomes available, the required developments in CFD and hence the advantages and possibilities outlined within this paper would not necessarily follow directly. For this reason, it is of great importance to all parties that key developments and their possible implications in other fields gain suitable publicity and are appreciated by as wide an audience as possible.

## 5.3 Implications for Industry

The marine industry has typically been conservative and reactive, meaning that new developments and techniques are not immediately trusted. Even the current state of the art capability in CFD simulation is not widely trusted, and where the results are accepted, they inevitably need to be well supported by costly experimental testing results. Due to the perceived complexity of CFD, where it is not understood, it is not trusted. It is therefore of vital importance that where the capability for sophisticated simulations is available, all possible steps are taken to ensure that the processes are transparent, well validated and can prove to provide accurate and high quality results. Validation and verification (V&V) procedures, e.g. [7] and [8], is a research topic in its own right, with engineers constantly improving the quality of CFD results with continuous developments in tools and computational power leading to new CFD methods.

If this can be achieved and if trust can be developed in the use of these approaches, the benefits for industry would be significant. As has already been outlined, model scale experimental tests are limited in what can be recreated and are also subject to inherent scaling errors. Carrying out such tests is time-consuming and costly, and any design changes would require a new model to be built and a new set of experiments to be run. By contrast, future CFD simulations could be run at full scale, removing any inherent scaling errors. As computational capacity increases, the timescales and costs involved will continue to reduce. Therefore, design changes could be made and analysed more easily, and simulations of realistic scenarios could be conducted. This latter advantage would have a significant benefit in terms of ship safety and, in what is probably more appealing to industry, efficiency. Improved efficiency means a lower fuel bill and the ability to predict the efficiency of a vessel in a range of realistic operational conditions rather than at just one design condition. This will lead to more informed decisions on design and installation, producing vessels that are much better adapted to their typical operating profile than at present.

The challenges which lie ahead for the industry in arriving at this point are significant, with many aspects to consider, but if it can be achieved, the benefits will be universal in the field.

## 6. CLOSING REMARKS

Numerical solvers have developed significantly over the past few years. This has allowed marine, and more specifically ship hydrodynamics, research to progress in parallel. The continuous improvement of HPC capabilities and the move towards Exascale Computing will allow research methods to be exploited even further, opening up new possibilities previously not possible due to the constraints of the available computing capabilities. With this in mind, numerical solvers should be further developed taking into consideration the power and capacity of Exascale Computing platforms that will open up new windows of opportunity. In time this will result in more robust, accurate, scaleable, high fidelity, state of the art simulations, even when employing optimization procedures.

## 7. ACKNOWLEDGMENTS

Results were obtained using the EPSRC funded ARCHIE-WeSt High Performance Computer (www.archie-west.ac.uk). EPSRC grant no. EP/K000586/1.

## 8. REFERENCES

[1] F. Stern, Z. Wang, J. Yang, M. M. H Sadat-Hosseini, S. Bushan, M. Diaz, S. H. Yoon, P. C. Wu, S. Mo, R. S. Thodal and J. L. Grenestedt, "Recent Progress in CFD for Naval Architecture and Ocean Engineering," in *11th International Conference on Hydrodynamics* , 2014.

[2] Y. K. Demirel, M. Khorasanchi, O. Turan, A. Incecik and M. P. Schultz, "A CFD model for the frictional resistance prediction of antifouling coatings," *Ocean Engineering Vol. 89,* pp. 21-31, 2014.

[3] Y. K. Demirel, M. Khorasanchi, O. Turan and A. Incecik, "Prediction of the effect of hull fouling on ship resistance using CFD," in *17th International Congress on Marine Corrosion and Fouling (ICMCF)*, Singapore, 2014.

[4] P. Kellett, O. Turan and A. Incecik, "A Study of Numerical Ship Underwater Noise Prediction," *Ocean Engineering Vol.66,* pp. 113-120, Jul. 2013.

[5] K. Mizzi, Y. K. Demirel, C. Banks and O. Turan, "PBCF design optimisation and propulsion efficiency impact," in *RINA International Conference on Influence of EEDI on Ship Design*, London, 2014.

[6] F. Stern, J. Yang, Z. Wang, H. Sadat-Hosseni, M. Mousaviraad, S. Bhushan and T. Xing, "Computational Ship Hydrodynamics: Nowadays and Way Forward," in *29th Symposium on Naval Hydrodynamics*, Gothenburg, Sweden, 2012.

[7] I. B. Celik, U. Ghia, P. J. Roache, C. J. Freitas, H. Coleman and P. E. Raad, "Procedure for Estimation and Reporting of Uncertainty Due to Discretization in CFD Applications," *Journal of Fluids Engineering,* vol. 130, no. 7, 2008.

[8] F. Stern, R. Wilson and J. Shao, "Quantitative V&V of CFD simulations and certification of CFD codes," *International Journal for Numerical Methods in Fluids,* vol. 50, pp. 1335-1355, 2006.

# Swift: task-based hydrodynamics and gravity for cosmological simulations

Tom Theuns
Institute for Computational Cosmology
Department of Physics
Durham University
Durham DH1 3LE, UK

Aidan Chalk
School of Engineering and Computing Sciences
Durham University
Durham DH1 3LE, UK

Matthieu Schaller
Institute for Computational Cosmology
Department of Physics
Durham University
Durham DH1 3LE, UK

Pedro Gonnet
School of Engineering and Computing Sciences
Durham University
Durham DH1 3LE, UK
and
Google Switzerland GmbH
Brandschenkestr. 110
8002 Zurich, Switzerland

## ABSTRACT

Simulations of galaxy formation follow the gravitational and hydrodynamical interactions between gas, stars and dark matter through cosmic time. The huge dynamic range of such calculations severely limits strong scaling behaviour of the community codes in use, with load-imbalance, cache inefficiencies and poor vectorisation limiting performance. The new SWIFT code exploits task-based parallelism designed for many-core compute nodes interacting via MPI using asynchronous communication to improve speed and scaling. A graph-based domain decomposition schedules interdependent tasks over available resources. Strong scaling tests on realistic particle distributions yield excellent parallel efficiency, and efficient cache usage provides a large speedup compared to current codes even on a single core. SWIFT is designed to be easy to use by shielding the astronomer from computational details such as the construction of the tasks or MPI communication. The techniques and algorithms used in SWIFT may benefit other computational physics areas as well, for example that of compressible hydrodynamics. For details of this open-source project, see `www.swiftsim.com`

## Keywords

Task-based parallelism, Asynchronous data transfer

## 1. INTRODUCTION

The main aim of cosmological simulations of the formation of structures in the Universe is to understand which physical processes play in role in how galaxies form and evolve. For example, what determines whether a galaxy becomes a spiral or an elliptical? What is the origin of the morphology-density relation - the observation that elliptical galaxies cluster much more strongly than spirals? What sets the colours of galaxies? How does the rate of galaxy formation evolve over cosmic time? What is the nature of high-redshift galaxies? A better understanding of these processes will be required to take full advantage of the rich data sets being collected now, or promised by future observatories such as the James Webb space telescope[1], ESO's Extremely-Large telescope [2] or the Square Kilometre Array [3].

Such cosmological simulations start from initial conditions motivated by observations of the cosmic microwave background (CMB). The CMB provides a directly observable imprint of the small density fluctuations that will eventually grow due to gravity into galaxies and clusters of galaxies today. In an expanding universe, regions which are slightly over-dense become denser and eventually collapse due to the self-gravity of their dark matter. These collapsing 'halos' accrete gas that cools radiatively and makes stars. The simulations follow the build-up of the dark matter halos and the accretion, shock-heating, and radiative cooling of the gas onto halos.

The gas densities above which stars form are orders of magnitude higher than the typical density in a galaxy and this large dynamic range is one of the most challenging aspects of these computations. The radiation and winds of recently formed stars, and the energy injected by super nova explosions, strongly limit the rate at which a galaxy's gas is turned into stars. As a result, only $\sim 17$ per cent of all gas in the Universe has been converted into stars to date [3]. The tremendous dynamic range in mass, length and time, between gas accreting onto a halo and turning into stars prevents simulations to model these crucial processes in detail. 'Subgrid' schemes are therefore used to model processes that cannot (yet) be resolved numerically, not unlike what is done in other multi-scale calculations such as for example weather or climate modelling. Limiting the impact of these subgrid models by actually resolving some of the underlying physics is a tremendously exciting and computationally demanding

[1] `http://www.jwst.nasa.gov/`
[2] `http://www.eso.org/public/teles-instr/e-elt/`
[3] `https://www.skatelescope.org/`

challenge for the exascale era.

Current cosmological simulations often take months to run on hundreds to many thousands of cores. For example the recent EAGLE simulation [11] took 45 days to run on 4000 cores of the Durham Data Centric Cluster, part of the DIRAC infrastructure[4], and the simulation suite used nearly 40M core hours on the CURIE machine using a PRACE[5] allocation of computer time. Such long run times are currently limiting scientific progress.

This paper discusses the SWIFT code that is designed to overcome some of the limitations of community codes widely used in cosmology, in particular improving load-balance, cache-usage, and vectorisation. It also intends to shield the astronomer who intends to implement and test subgrid schemes from the underlying computational details.

## 2. COSMOLOGICAL GAS DYNAMICS

This section provides a brief overview of the equations being integrated. Calculations are performed in co-moving coordinates $\mathbf{x}$ say for position, related to physical coordinates $\mathbf{r}$ by the time-dependent scale factor $a(t)$, $\mathbf{r} = a\mathbf{x}$ (see for example [10]), but we will ignoring these details here. Performing these calculations using a Lagrangian scheme where the fluid is represented by a set of particles that move with the fluid's speed is very advantageous, because the flow speeds are very large due to the large (gravitational) motions of forming galaxies.

Smoothed particle hydrodynamics (SPH, [4, 9]) is such a Langrangian scheme in which values for fluid variables are interpolated from a disordered particle distribution using kernel interpolation. For example the density $\rho$ and pressure gradient $\nabla p$ at the location $\mathbf{r}_i$ of particle $i$ are computed with equations of the form

$$\rho(\mathbf{r}_i) = \sum_j m_j W(\frac{|\mathbf{r_i}-\mathbf{r_j}|}{h_i}), \quad (1)$$

$$\nabla p(\mathbf{r}_i) = \sum m_j \left(\frac{p(\mathbf{r}_i)}{\rho(\mathbf{r}_i)^2}+\frac{p(\mathbf{r}_j)}{\rho(\mathbf{r}_j)^2}\right)\nabla W(\frac{|\mathbf{r_i}-\mathbf{r_j}|}{h_i}) \quad (2)$$

where $m_j$ is the mass of particle $j$ and $W$ is a bell-shaped kernel with compact support, $W(q) = 0$ for $q > 1$. The smoothing length $h_i$ is computed such that a given weighted number of particles contributes to the sum. The pressure is found from the density and temperature using an equation of state. Note that we need to evaluate the density for each particle before we can compute the pressure gradient. Several variations of Eq. (2) exist, we use this particular form here to illustrate the type of sums to be computed, SWIFT implements the more accurate version used in GADGET 2 [12].

Gravitational accelerations are calculated as,

$$\mathbf{a}_i = -\mathrm{G}\sum_{j\neq i}\frac{m_j}{|\mathbf{r_i}-\mathbf{r_j}|^3}(\mathbf{r}_i-\mathbf{r}_j), \quad (3)$$

with extra terms (not discussed here) to represent periodic images such that the simulated volume is periodically replicated (the Ewald summation familiar from solid state physics).

[4] http://www.stfc.ac.uk/1263.aspx
[5] http://www.prace-ri.eu/

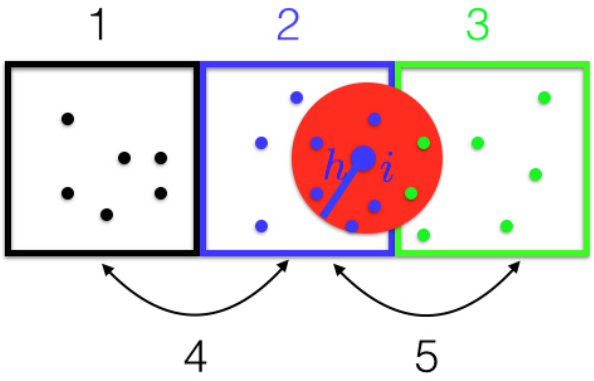



**Figure 1: Illustration of neighbour finding on a mesh. Five tasks are indicated, numbers 1-3 compute densities from pairs of particles in cells 1-3, whereas tasks 4 and 5 compute densities between particles pairs in neighbouring cells.**

Given the initial state of the system, specified by position and velocities of all particles, particles are marched forward in time using velocities to update positions and accelerations to update velocities. Most of the calculation time is spent in evaluating the hydrodynamical and gravitational forces. The popular GADGET [12] and GASOLINE [13] codes use a tree to find neighbours for evaluating the sum in Eqs. (1-2). These codes split the gravitational force from Eq. (3) into a contribution from nearby particles evaluated using a tree following [1], and contribution from distant particles evaluated using a mesh, as in the P3M scheme described in detail in [7], see [2] for the application in cosmology. The particles are distributed over the computational volume using a space-filling curve to attempt to preserve locality which reduces MPI communication needed if neighbour particles are not held on the same MPI task. Such 'domain decomposition' also takes significant compute time. How these issues are handled in SWIFT is described next.

## 3. TASK-BASED CALCULATIONS

### 3.1 SPH

SWIFT identifies potential neighbours by organising particles in cubic cells as illustrated in Fig.1 (drawn in 1 dimension for simplicity). By choosing the cell size of the mesh to be larger than the smoothing length $h$ of all particles in that cell guarantees that particles within $h_i$ of the fat blue particle in the figure can be found either in the same cell (blue, labelled '2'), or in one of the two neighbouring cells (black and green, labelled '1' and '3' respectively). Given the large dynamic range in $h$, such a mesh needs to be adaptive. The density calculation of Eq. (1) for particle $i$ now involves three steps: find neighbours of $i$ in each of the three cells (in the figure, these are particles within the red circle with radius $h_i$).

In SWIFT, each of these calculations is executed by separate **tasks**. In the simple case illustrated in Fig.1 there are two types: tasks that involve evaluating Eq. (1) for pairs of particles in the *same* cell (labelled 1-3), and tasks that involve

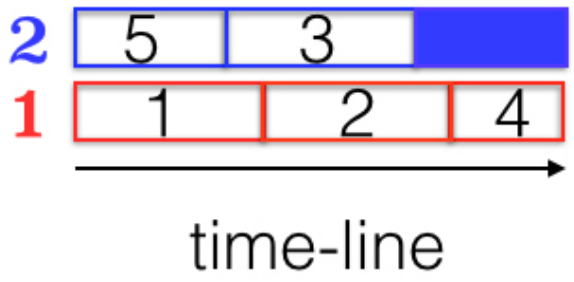



**Figure 2: Execution of the 5 tasks (labelled 1-5) illustrated in Fig.1, by two threads (labelled 1 and 2 and coloured red and blue, respectively) with conflicts. Thread 1 starts executing task 1, while thread 2 executes task 5, locking tasks 2, 3 and 4. When thread 2 completes task 5, it immediately starts executing task 3. Thread 1 can execute task 2 locking task 4 when task 1 is completed. However thread 2 cannot start executing task 4 as long as task 2 is not completed, since tasks 2 and 4 conflict.**

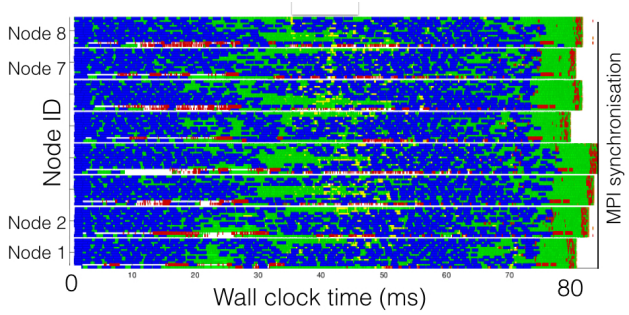



**Figure 3: Task time-line for SWIFT SPH calculation, running on 8 nodes (thick bands) with 12 cores (thin bands) each. Different colours corresponds to different tasks, for example red refers to communication. As the calculation progresses, each core is executing tasks mostly independent of other cores, with little idle time lost due to MPI synchronisation at the end of the time step.**

evaluating Eq. (1) for pairs of particles in neighbouring cells (labelled 4 and 5). To avoid race conditions, some tasks cannot be performed simultaneously, in this particular case tasks 4 and 5 conflict with each other, 4 conflicts with 1 and 2, and 5 with 2 and 3 . The task scheduling in SWIFT therefore should be able to handle both conflicts and dependencies.

How these 5 tasks could be executed by two threads is illustrated in Fig. 2. At the start, threads pick tasks independently, locking those tasks that conflict with them. In this example, thread 1 executes task 1, and thread 2 executes task 5 (locking tasks 2 and 3). When thread 2 completes task 5 it unlocks tasks 2 and 3, and starts executing task 3 (locking task 4). When task 3 is finished, thread 2 is idle because the remaining task 4 conflicts with task 2 being executed by thread 1. One of the threads (in the illustration thread 1) finishes off the work.

The efficiency of the tasks themselves can be improved by *sorting* [6, 5]. Indeed, consider again the fat blue particle $i$ in Fig.1 when task 5 is executed. If we were to sweep through the green particles in cell 3 *from left to right*, we would find that the fourth green particle no longer contributes to the density since it is outside the red circle. There is therefore no reason to even check if any of the other green particles is inside the red circle, since these are even further away from particle $i$ in the horizontal direction.

The SWIFT SPH implementation contains several similar 'kernels' that calculate the interaction between two particles (for example individual terms in Eq. (1) or in Eq (2)). Exposing these basic routine to the user greatly simplifies adapting the code to the user's wishes, for example in making changes to the basic SPH algorithm. This kernel is called for a range of particles that are in the same cell. Cache-misses are minimised by making sure these particles are nearly contiguous in memory. Bunching particles in cells is then also advantages for vectorising, either using intrinsics, or by using pragma's that allow the compiler to known that these calculations can be vectorised.

With sorting tasks, density tasks, and pressure gradients tasks (and gravity tasks, described next) combined for all cells, a science run will typically contain hundreds or even millions of tasks. Individual threads on a many-core node can thus all be executing tasks as long as these do not conflict with each other, using task stealing to grab a new task as soon as their current task is completed. Once a thread grabs a new task, it blocks those tasks that conflict with it. In addition to conflicts, the SWIFT task engine also handles task *dependencies* - for example the density of particles in a cell and its neighbouring cell should have been computed before pressure gradients can be computed.

Running this task-based parallelisation across MPI tasks introduces relatively minimal additional complexity. If neighbouring cells are assigned to different MPI tasks, SWIFT will generate extra communication tasks that exchange the contents of individual cells using asynchronous communication. The distribution of particles (or rather cells) across MPI tasks is based on the total costs of tasks - assigning similar work to each MPI task - while aiming to minimise

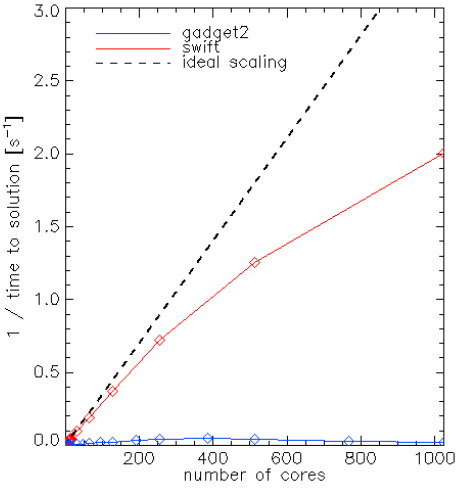



**Figure 4: Time to solution strong scaling test of the SPH implementation in SWIFT compared to GADGET 2 for a realistic particle distribution with 51 million particles taken from a cosmological volume. Scaling is shown from 1 to 1024 cores (64 nodes with 16 cores each). SWIFT uses 16 threads per core, GADGET 2 uses MPI also within a node. SWIFT reaches 60 per cent parallel efficiency for strong scaling from 1 to 1024 cores.**

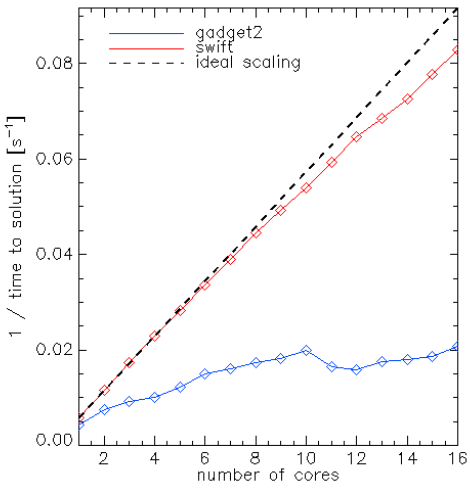



**Figure 5: Time to solution strong scaling test of the Barnes-Hut gravity implementation in SWIFT compared to GADGET 2 for a 10M highly-clustered particle distribution on a single node. Increasing the thread count from 1 to 16 reduces the time to solution in SWIFT by a factor 14, a 90 per cent efficiency (red line). Increasing the number of MPI-tasks for GADGET-2 from 1 to 16 decrease the time to solution by factor of 5.**

communication that results from spatially non-contiguous particle distributions. Generating and scheduling the inter-dependent tasks is performed in a similar way as is done in the QUICKSCHED library [5] using the METIS library [8] to partition tasks over MPI tasks. An example is shown in Fig.3 (a realistic version of Fig. 2), which shows a time-line of how 8 nodes of 12 threads each execute a set of tasks using MPI across nodes. Running on a realistic particle distribution, SWIFT achieves 60 per cent parallel efficiency in a strong scaling test increasing the core count from one to 1024 (see Fig. 4, see also [5]).

Using cells to organise particles spatially and identify potential neighbours may at first sight seem very different from using a tree as in the GADGET 2 or GASOLINE codes. However the algorithms are actually surprisingly similar once one limits the depth of the tree to cells that contain $\sim 100$ particles as is the case in SWIFT. How to find neighbouring *cells* in SWIFT is actually also performed using a tree.

## 3.2 Gravity

Currently SWIFT implements the Barnes-Hut tree code algorithm [1] for evaluating the gravitational acceleration from Eq. (3), with some modifications described below. The Barnes-Hut algorithm divides the simulation volume spatially and recursively in smaller cells. Such a division is very well suited for evaluating gravitational interactions. Indeed consider a particle $i$ at some distance from a tree node. A good approximation for the contribution of that node to $\mathbf{a}_i$ can be obtained using a multipole expansion, for example representing all the particles in the node by their monopole, as long as the distance particle-node is large compared to the extent of the node. If the distance is small, the node is split in its daughter cells, and the algorithm recurs.

This Barnes-Hut algorithm decreases the computational cost of evaluating $\mathbf{a}_i$ for all particles from order $N^2$ to order $N \log(N)$ [1]. Note that two particles that are spatially close are likely to execute nearly identical tree walks. In practise most of the compute time is now spent in the tree walk (rather than evaluating actual accelerations).

We implemented three optimisations of this algorithm in SWIFT. Firstly we limit the depth of the tree from leaf nodes that contain a single particle (as in GADGET) to cells with $\sim 100$ particles. This is because the tree walk is not very efficient for *small* numbers of particles.

Secondly we do not start a tree walk for each particle from the root node, but rather walk the tree walk for *nodes*. For each set of nodes, we decide whether they are sufficiently distant to compute forces using multipoles, or they should be split in their daughter nodes recursively. Doing so results in a list of tasks, those in which particles in one node interact with the multipole of another node, or those where all particles in one node interact with all particles in a nearby node. The latter task is implemented efficiently using the same task-based approach as used for SPH in the previous section.

Thirdly we use quadrupoles rather than monopoles. This increases time to solution minimally yet make the accelerations more accurate.

The speed and scaling of the tree implementation in SWIFT is compared to that of GADGET 2 in Fig. 5, in which $\mathbf{a}_i$ is

calculated for each of 10M particles taken from the same snapshot of an EAGLE simulation as used in Fig. 4 (a very clustered distribution of particles). The speed of SWIFT is close to that of GADGET 2 when run on a single core, and the scaling up to 16 threads is close to ideal (parallel efficiency of 90 per cent). The public version of GADGET 2 does not have multi-threading, and the scaling shown is when increasing the number of MPI tasks using one core per task.

# 4. CONCLUSIONS

We have implemented smoothed particle hydrodynamics (SPH) and a Barnes-Hut tree-code for self-gravity in the cosmological hydrodynamics code SWIFT. By grouping nearby particles in cells, the calculation is broken-up into very many short and inter-dependent tasks, whereby a single task processes particles within a cell, or between pairs of cells. Task dependencies and conflicts are encoded in the application. Using cells improves cache efficiency and simplifies vectorisation. The tasks are distributed across nodes, with individual threads using task-stealing within a node, and communication being performed asynchronously between nodes. We find that such task-based parallelism is well suited to take advantage of the multiple levels of parallelism of modern many-core super computers. Applied to a realistic particle distribution, SWIFT's SPH implementation reaches a parallel efficiency of 60 per cent in a strong scaling test when increasing core count from 1 to 1024, and better than 90 per cent on a single 16-core node for gravity. Individual physics routines, for example those that evaluate interactions between two particles, are implemented in simple kernels to shield the physicist from the intricacies of tasks or MPI communications. SWIFT is an open-source project, `www.swiftsim.com`.

## Acknowledgments

This work used the DiRAC Data Centric system at Durham University, operated by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (`www.dirac.ac.uk`). This equipment was funded by BIS National E-infrastructure capital grant ST/K00042X/1, STFC capital grant ST/H008519/1, and STFC DiRAC Operations grant ST/K003267/1 and Durham University. DiRAC is part of the National E-Infrastructure. This work was supported by the Science and Technology Facilities Council [ST/F001166/1] and the European Research Council under the European Union's ERC Grant agreements 267291 Cosmiway, by the Interuniversity Attraction Poles Programme initiated by the Belgian Science Policy ([AP P7/08 CHARM]), and by INTEL through establishment of the ICC as an INTEL parallel computing centre (IPCC).

# 5. REFERENCES

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449, Dec. 1986.

[2] G. Efstathiou, M. Davis, S. D. M. White, and C. S. Frenk. Numerical techniques for large cosmological N-body simulations. *ApJS*, 57:241–260, Feb. 1985.

[3] M. Fukugita, C. J. Hogan, and P. J. E. Peebles. The Cosmic Baryon Budget. *ApJ*, 503:518–530, Aug. 1998.

[4] R. A. Gingold and J. J. Monaghan. On the fragmentation of differentially rotating clouds. *MNRAS*, 204:715–733, Aug. 1983.

[5] P. Gonnet. Efficient and Scalable Algorithms for Smoothed Particle Hydrodynamics on Hybrid Shared/Distributed-Memory Architectures. *SIAM Journal on Scientific Computing*, 37:95–121, Apr. 2015.

[6] P. Gonnet, M. Schaller, T. Theuns, and A. B. G. Chalk. SWIFT: Fast algorithms for multi-resolution SPH on multi-core architectures. *ArXiv e-prints*, Sept. 2013.

[7] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. 1988.

[8] G. Karypis and V. Kumar. Fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

[9] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *AJ*, 82:1013–1024, Dec. 1977.

[10] P. J. E. Peebles. *Principles of Physical Cosmology*. 1993.

[11] J. Schaye, R. A. Crain, R. G. Bower, M. Furlong, M. Schaller, T. Theuns, C. Dalla Vecchia, C. S. Frenk, I. G. McCarthy, J. C. Helly, A. Jenkins, Y. M. Rosas-Guevara, S. D. M. White, M. Baes, C. M. Booth, P. Camps, J. F. Navarro, Y. Qu, A. Rahmati, T. Sawala, P. A. Thomas, and J. Trayford. The EAGLE project: simulating the evolution and assembly of galaxies and their environments. *MNRAS*, 446:521–554, Jan. 2015.

[12] V. Springel. The cosmological simulation code GADGET-2. *MNRAS*, 364:1105–1134, Dec. 2005.

[13] J. W. Wadsley, J. Stadel, and T. Quinn. Gasoline: a flexible, parallel implementation of TreeSPH. *New Astronomy*, 9:137–158, Feb. 2004.

# Thread Parallelism for Highly Irregular Computation in Anisotropic Mesh Adaptation

Georgios Rokos
Imperial College London
London, United Kingdom
georgios.rokos09@imperial.ac.uk

Gerard J. Gorman
Imperial College London
London, United Kingdom
g.gorman@imperial.ac.uk

Kristian Ejlebjerg Jensen
Imperial College London
London, United Kingdom
kristianejlebjerg@gmail.com

Paul H. J. Kelly
Imperial College London
London, United Kingdom
p.kelly@imperial.ac.uk

## ABSTRACT

Thread-level parallelism in irregular applications with mutable data dependencies presents challenges because the underlying data is extensively modified during execution of the algorithm and a high degree of parallelism must be realized while keeping the code race-free. In this article we describe a methodology for exploiting thread parallelism for a class of graph-mutating worklist algorithms, which guarantees safe parallel execution via processing in rounds of independent sets and using a deferred update strategy to commit changes in the underlying data structures. Scalability is assisted by atomic fetch-and-add operations to create worklists and work-stealing to balance the shared-memory workload. This work is motivated by mesh adaptation algorithms, for which we show a parallel efficiency of 60% and 50% on Intel®Xeon® Sandy Bridge and AMD Opteron™ Magny-Cours systems, respectively, using these techniques.

## Keywords

anisotropic mesh adaptivity, irregular data, shared-memory parallelism, manycore, parallel worklist algorithm, topology mutation, graph colouring, work-stealing, deferred update

## 1. INTRODUCTION

Finite element/volume methods (FEM/FVM) are commonly used in the numerical solution of partial differential equations (PDEs). Unstructured meshes, where the spatial domain has been discretised into simplices (*i.e.* triangles in 2D, tetrahedra in 3D), are of particular interest in applications where the geometric domain is complex and structured meshes are not practical. Simplices are well suited to varying mesh resolution throughout the domain, allowing for local coarsening and refinement of the mesh without hanging nodes. On the other hand, this flexibility introduces complications of its own, such as management of mesh quality and computational overheads arising from indirect addressing.

Computational mesh resolution is often the limiting factor in simulation accuracy. Being able to accurately resolve physical processes at the small scale coupled with larger scale dynamics is key to improving the fidelity of numerical models across a wide range of applications (*e.g.* [15, 20]). A difficulty with mesh-based modelling is that the mesh is generated before the solution is known, however, the local solution error is related to the local mesh resolution. Overly coarse meshes lead to low accuracy whereas over-refined meshes can greatly increase the computational cost.

Mesh adaptation methods provide an important means to minimise computational cost while still achieving the required accuracy [16, 12]. In order to use mesh adaptation within a simulation, the application code requires a method to estimate the local solution error. Given an error estimate it is then possible to compute a solution to a specified error tolerance while using the minimum resolution everywhere in the domain and maintaining element quality constraints.

Previous work has described how adaptive mesh methods can be parallelised for distributed-memory systems using MPI (*e.g.* [12, 10]). However, there is a continuous trend towards an increasing number of cores per compute node in the world's most powerful supercomputers and it is assumed that the nodes of a future exascale system will each contain thousands of cores [7]. Therefore, it is important that algorithms are developed with very high levels of parallelism and using thread-parallel programming models, such as OpenMP [5], that exploit the memory hierarchy. However, irregular applications are hard to parallelise effectively on shared-memory architectures for reasons described in [13].

In this article we take a fresh look at anisotropic adaptive mesh methods and parallelise them using new scalable techniques suitable for modern multicore and manycore architectures. These concepts have been implemented in the open source framework PRAgMaTIc (Parallel anisotRopic Adaptive Mesh ToolkIt)[1]. The remainder of the paper is laid out as follows: §2 gives an overview of the mesh adaptation procedure; §3 describes the new irregular compute methodology used to parallelise the adaptive algorithms; and §4 illustrates how well our framework performs in 2D and 3D benchmarks. We conclude the paper in §5.

## 2. MESH ADAPTIVITY BACKGROUND

In this section we give an overview of anisotropic mesh adaptation, focusing on the element quality as defined by an error metric and the adaptation kernels which iteratively improve local mesh quality as measured by the worst local element.

[1] http://meshadaptation.github.io/

## 2.1 Error control

Solution discretisation errors are closely related to the size and the shape of the elements. However, in general meshes are generated using *a priori* information about the problem under consideration when the solution error estimates are not yet available. This may be problematic because (a) solution errors may be unacceptably high and (b) parts of the solution may be over-resolved, thereby incurring unnecessary computational expense. A solution to this is to compute appropriate local error estimates and use them to dynamically control the local mesh resolution at runtime. In the most general case this is a metric tensor field so that the resolution requirements can be specified anisotropically; for a review of the procedure see [11].

## 2.2 Element quality

As discretisation errors are dependent upon element shape as well as size, a number of measures of element quality have been proposed, out of which, in the work described here, we use the quality functionals by Vasilevskii *et al.* for triangles [22] and tetrahedra [1], which indicate that the ideal element is an equilateral triangle/tetrahedron with edges of unit length measured in metric space.

## 2.3 Overall adaptation procedure

The mesh is adapted through a series of local operations: edge collapse, edge refinement, element-edge swaps and vertex smoothing. While the first two of these operations control the local resolution, the latter two are used to improve the element quality. Algorithm 1 gives a high level view of the anisotropic mesh adaptation procedure as described by Li *et al.* [12]. The inputs are $\mathcal{M}$, the piecewise linear mesh from the modelling software, and $\mathcal{S}$, the node-wise metric tensor field which specifies anisotropically the local mesh resolution requirements. The process involves the iterative application of coarsening, swapping and refinement to optimise the resolution and quality of the mesh. The loop terminates once the mesh optimisation algorithm converges or after a maximum number of iterations has been reached. Finally, mesh quality is fine-tuned using some vertex smoothing algorithm, which aims primarily at improving the worst-element quality. Smoothing is left out of the main loop because it is an expensive operation and it is found empirically that it is efficient to fine-tune the worst-element quality once mesh topology has been fixed.

**Algorithm 1** Mesh optimisation procedure.

```
Inputs: M, S.
repeat
    (M*, S*) ← coarsen(M*, S*)
    (M*, S*) ← swap(M*, S*)
    (M*, S*) ← refine(M*, S*)
until (max. number of iterations or convergence)
(M*, S*) ← smooth(M*, S*)
return M*
```

## 2.4 Adaptation kernels

A brief description of the four mesh optimisation kernels follows. Figure 1 shows 2D examples to demonstrate what each kernel does to the local mesh patch, but the same operations are applied in an identical manner in 3D. For more details on the adaptive algorithms the reader is referred to the publications by Li *et al.* [12] (coarsening, refinement, swapping) and Freitag *et al.* [8, 9] (smoothing).

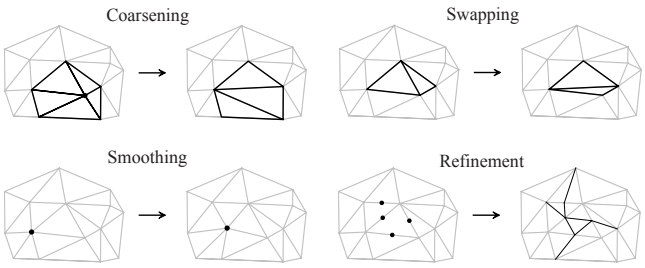



**Figure 1: Examples of the four adaptive kernels.**

### 2.4.1 Coarsening

Coarsening is the process of lowering mesh resolution locally by collapsing an edge to a single vertex, thereby removing all elements that contain this edge, leading to a reduction in the computational cost.

### 2.4.2 Refinement

Refinement is the process of increasing mesh resolution locally by (a) splitting of edges which are longer than desired (as indicated by the error estimation) and (b) subsequent division of elements using refinement templates [6].

### 2.4.3 Swapping

Swapping is done in the form of flipping an edge shared by two elements, considering the quality of the swapped elements - if the minimum quality is improved then the original mesh elements are replaced with the edge-swapped elements.

### 2.4.4 Smoothing

The kernel of vertex smoothing relocates a central vertex so that the local mesh quality is increased. Common heuristic methods are the quality-constrained Laplacian smoothing [8] and the more expensive optimisation-based smoothing [9].

### 2.4.5 Propagation

The operations of coarsening, swapping and smoothing often need to be propagated to the local mesh neighbourhood. When a kernel is applied onto an edge/vertex, neighbouring edges/vertices need to be reconsidered for processing because the topological/geometrical changes that occurred might give rise to new configurations of better quality. Therefore, these adaptive algorithms keep sweeping over the mesh until no further changes are made.

# 3. IRREGULAR COMPUTE METHOD

To allow fine grained parallelisation of mesh adaptation we based our methodology on graph colouring, following a proposal by Freitag *et al.* [10]. However, while this approach avoids updates being applied concurrently to the same neighbourhood, data writes will still incur significant lock and synchronisation overheads. For this reason we incorporate a deferred update strategy, described below, to minimise synchronisations and allow parallel writes. Additionally, we make use of atomic operations to create parallel worklists

in a synchronisation-free fashion and, finally, try to balance the workload among threads using work-stealing [3].

## 3.1 Hazards

There are two types of hazards when running mesh optimisation algorithms in parallel: *topological hazards*; and *data races*. The former refers to the situation where an adaptive operation results in invalid or non-conforming edges and elements. For example in coarsening, if some vertex $V_B$ collapses onto another vertex $V_A$, then $V_A$ cannot collapse onto some other vertex at the same time. Data races can occur when two threads try to update the same adjacency list of a vertex concurrently. For example in coarsening, two neighbours of some vertex $V_A$ can collapse onto $V_A$ concurrently, then $V_A$'s adjacency list has to be updated to reflect the changes made by the coarsening operations. Concurrent access to $V_A$'s adjacency list may lead to race conditions.

## 3.2 Colouring

Topological hazards for all adaptive algorithms are avoided by colouring a graph whose nodes are defined by the mesh vertices and edges are defined by the mesh edges. The adaptive algorithm then processes the mesh in batches of independent sets. The fact that topological changes are made to the mesh means that colouring is invalidated frequently and the mesh has to be re-coloured before proceeding to the next iteration of the adaptive algorithm. Therefore, we need to use a fast and scalable colouring algorithm (see [18]).

## 3.3 Deferred Update

Colouring does not eliminate data races when updating adjacency lists. A 2-distance colouring was not considered here as it is expensive and increases the chromatic number, effectively reducing the exposed parallelism. Instead, in a shared-memory environment with $N$ threads, each thread allocates a private collection of $N$ lists. When the adjacency list of some vertex $V_i$ has to be updated, the thread executing the adaptive kernel does not commit the update immediately; instead, it pushes the operation back into the list for thread $tid = ID(V_i)\%N$, where $ID(V_i)$ is the integer identifier of $V_i$. After processing an independent set and before proceeding to the next one, every thread $T_i$ visits the private collections of all threads, locates the list reserved for $T_i$ and commits the updates stored there. This way, it is guaranteed that one and only thread will update the adjacency list of any given vertex. We call this technique the *deferred update*. Code Snippet 1 demonstrates a typical usage scenario. An important advantage of this strategy is that we always read the most up-to-date data when executing an adaptive kernel (as if we used an "as we go" write-back scheme), eliminating the risk of mesh data corruption in coarsening, refinement and swapping and having a faster-converging Gauss-Seidel-style iteration process in smoothing.

## 3.4 Worklists and Atomic-Capture

There are many cases where it is necessary to create a worklist of items which need to be processed, *e.g.* for propagation of adaptive operations. New work items generated locally by a thread need to be accumulated into a global worklist over which the next invocation of the adaptive kernel will iterate. The classic approach based on prefix sums [2] requires thread

```
typedef std::vector<Updates> DefUpdList;
int N = omp_get_max_threads();

// N collections of deferred-update lists
std::vector< std::vector<DefUpdList> > defUpd(N);

#pragma omp parallel
{
  int tid = omp_get_thread_num();
  // Allocate one list for each thread.
  defUpd[tid].resize(N);

  // Process the independent set in parallel
  #pragma omp for
  for(int i=0; i<nVerticesInSet; ++i){
    update = execute_kernel(i);
    // To be committed by thread i%N.
    defUpd[tid][i%N].push_back(update);
  }

  // Commit updates tid is responsible for.
  for(int i=0; i<N; ++i)
    commit_all_updates(defUpd[i][tid]);

  // Proceed to the next independent set...
}
```

Code Snippet 1: Example of the deferred update scheme.

synchronisation and was found limiting in terms of scalability. A better method is based on atomic fetch-and-add on a global integer which stores the size of the worklist needed so far. Every thread increments this integer atomically while caching the old value. This way, the thread knows where to copy its private data and increments the integer by the size of this data, so the next thread to access the integer knows in turn where to copy its private data. An example of using this technique via OpenMP's atomic-capture clause [14] is given in Code Snippet 2, where it is shown that no thread synchronisation is needed to generate the global worklist (note the `nowait` clause). The overhead/spinlock associated with atomic-capture operations was found to be insignificant.

```
int worklistSize = 0;
std::vector<Item> globalWorklist(prealloc_size);

#pragma omp parallel
{
  std::vector<Item> private_list;

  #pragma omp for nowait
  for(all items which need to be processed){
    new_item = do_some_work();
    private_list.push_back(new_item);
  } // No need to synchronise at end of loop.

  int idx;
  #pragma omp atomic capture
  {
    idx = worklistSize;
    worklistSize += private_list.size();
  }

  memcpy(&globalWorklist[idx], &private_list[0],
         private_list.size() * sizeof(Item));
}
```

Code Snippet 2: Creating a worklist using atomic-capture.

## 3.5 Work-stealing

Work-stealing [3] is a sophisticated technique aiming at balancing workload among threads while keeping scheduling

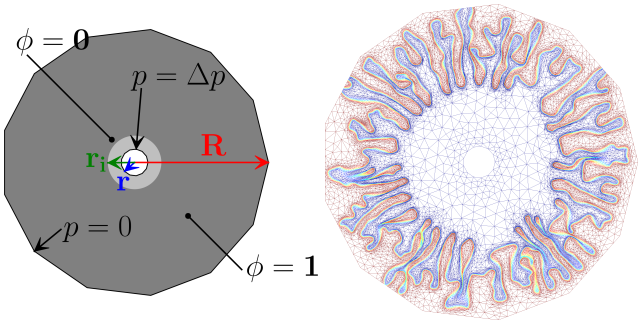



**Figure 2: The initial condition for the viscous fingering (left) and a snapshot of a simulation (right).**

overhead as low as possible. For-loop scheduling strategies provided by the OpenMP runtime system were found to be inadequate, either incurring significant scheduling overhead or leading to load imbalances. As of version 4.0, OpenMP does not support work-stealing for parallel for-loops so we created a novel scheduler [19] which differs from other proposals in two ways: it engages a heuristic to help the thief find a suitable victim to steal from; and uses POSIX signals/interrupts to accomplish stealing in an efficient manner.

## 4. EXPERIMENTAL RESULTS

We will show adaptivity results for viscous fingering in 2D and structural compliance minimisation in 2D and 3D, followed by performance evaluation.

### 4.1 Viscous Fingering

Viscous fingering is a limiting process for enhanced oil recovery technologies. It happens whenever one fluid displaces another with a higher viscosity [17], typically in a porous media. A typical setup and simulation is shown in Figure 2, with the blue fluid having a viscosity $e^2$ times lower than the red fluid. The initial saturation is unperturbed and it is thus the length scale of the initial mesh that triggers the instability. Mesh adaptation is driven by the Hessian of the pressure combined with the Hessian of the saturation $\phi$ [4].

### 4.2 Structural Optimisation

Structural compliance minimisation is concerned with the problem of finding stiff and lightweight mechanical components [21], often in the context of linear elasticity. The setup for a classical cantilever problem with support to the left and a load to the right is shown in Figure 3 (top left). The question is how to form the stiffest possible link between the two boundaries, given a certain amount of isotropic material. The problem is ill-posed unless a minimum length scale is imposed for the design, because the optimal structure is a composite. In fact, one can see a tendency towards microstructured areas when a small minimum length scale $L_{\min} = 10^{-3} L_{\text{char}}$ is used as illustrated in Figure 3 (bottom left). Note how the many straight and parallel connections can be efficiently resolved with anisotropic elements. Mesh adaptation is driven by the Hessians of the design and the topological derivative [21]. A Helmholtz filter is applied to both design and derivative to smooth out features smaller than $L_{\min}$, before the Hessians are computed.

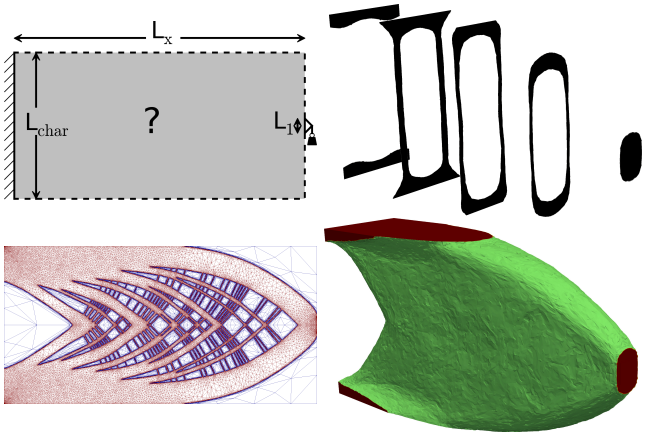



**Figure 3: The setup for structural compliance minimisation (top left) and the result for the case of a small minimum length scale (bottom left). Red corresponds to solid areas and blue to void. The result of compliance minimisation in 3D is shown in terms of the cross-sectional view (top right) and the solid/void interface (bottom right).**

The two dimensional setup is also extruded to three dimensions, as plotted in Figure 3 (top right and bottom right). Note that the large planar areas with little curvature are well resolved by the anisotropic elements. The increased dimensionality leads to a much simpler topology even though the optimisation is performed with $L_{\min} = 5 \cdot 10^{-3} L_{\text{char}}$.

In order to evaluate the parallel performance, a synthetic solution $\psi$ is defined to vary in time and space:

$$\psi(x, y, t) = 0.1 \sin\left(50x + \frac{2\pi t}{T}\right) + \arctan\left(-\frac{0.1}{2x - \sin\left(5y + \frac{2\pi t}{T}\right)}\right)$$

where $T$ is the period. This is a good choice as a benchmark as it contains multi-scale features and a shock front, *i.e.* the typical solution characteristics where anisotropic adaptive mesh methods excel. An isotropic mesh was generated on the unit square using approximately $200 \times 200$ triangles and the adaptation benchmark was run with a requirement for $\approx 550k$ elements. The same example was extruded in 3D, where an isotropic mesh was generated in the unit cube using approximately $50 \times 50 \times 50$ tetrahedra and the adaptation benchmark was run with a requirement for $\approx 210k$ elements. 3D swapping has not been parallelised, therefore the corresponding results have been omitted.

The code was compiled using the Intel compiler suite (version 14.0.1) and with the `-O3` optimisation flag. We used two systems to evaluate performance: (a) a dual-socket Intel® Xeon® E5-2650 system (Sandy Bridge, 2.00GHz, 8 cores per socket, 2-way hyper-threading) running Red Hat® Enterprise Linux® Server release 6.4 (Santiago) and (b) a quad-socket AMD Opteron™ 6176 system (Magny-Cours, 2.3GHz, 12 cores per socket) running Ubuntu 12.04.5. In all cases, thread-core affinity was used. Figures 4 and 5 show the average (over 50 time steps) execution time per time step and parallel efficiency against the number of threads using single-socket (SS), dual-socket (DS) and quad-socket (QS) configurations. On the Intel® Xeon® system we also enable hyper-threading (HT) to make use of all 40 logical cores.

Running on one socket, our code achieves a parallel efficiency of over 60% on Intel®Xeon® and around 50% on AMD Opteron™. Smoothing scales better than the other algorithms as it is more compute-intensive, which favours scalability, reaching an efficiency of over 50% even in the quad-socket case. When we move to more sockets, NUMA effects become pronounced, which is expected as common NUMA optimisations such as pinning and first-touch for page binding are ineffective for irregular computations. Nonetheless, the achievable efficiency is good considering the irregular nature of those algorithms.

## 5. CONCLUSION

In this paper we examined the scalability of anisotropic mesh adaptivity using a thread-parallel programming model and explored new parallel algorithmic approaches to support this model. Despite the complex data dependencies and inherent load imbalances we have shown it is possible to achieve practical levels of scaling using a combination of a fast graph colouring technique, the deferred-update strategy, atomic-based creation of worklists and for-loop work-stealing. In principle, this methodology facilitates scaling up to the point where the number of elements of an independent set is equal to the number of available threads.

## Acknowledgments

This project has been supported by Fujitsu Laboratories of Europe Ltd and the EPSRC (grant numbers EP/I00677X/1 and EP/L000407/1).

## 6. REFERENCES

[1] A. Agouzal, K. Lipnikov, and Y. Vassilevski. Adaptive generation of quasi-optimal tetrahedral meshes. *East-West Journal*, 7(4):223–244, 1999.

[2] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Nov. 1990.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 356–368, Washington, DC, USA, 1994. IEEE Computer Society.

[4] L. Chen, P. Sun, and J. Xu. Optimal anisotropic meshes for minimizing interpolation errors in $L^p$-norm. *Mathematics of Computation*, 76(257):179–204, 2007.

[5] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[6] H. L. De Cougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46(7):1101–1125, 1999.

[7] J. Dongarra. What you can expect from exascale computing. In *International Supercomputing Conference (ISC'11), Hamburg, Germany*, 2011.

[8] L. Freitag and C. Ollivier-Gooch. A comparison of tetrahedral mesh improvement techniques, 1996.

[9] L. A. Freitag and P. M. Knupp. Tetrahedral mesh improvement via optimization of the element condition

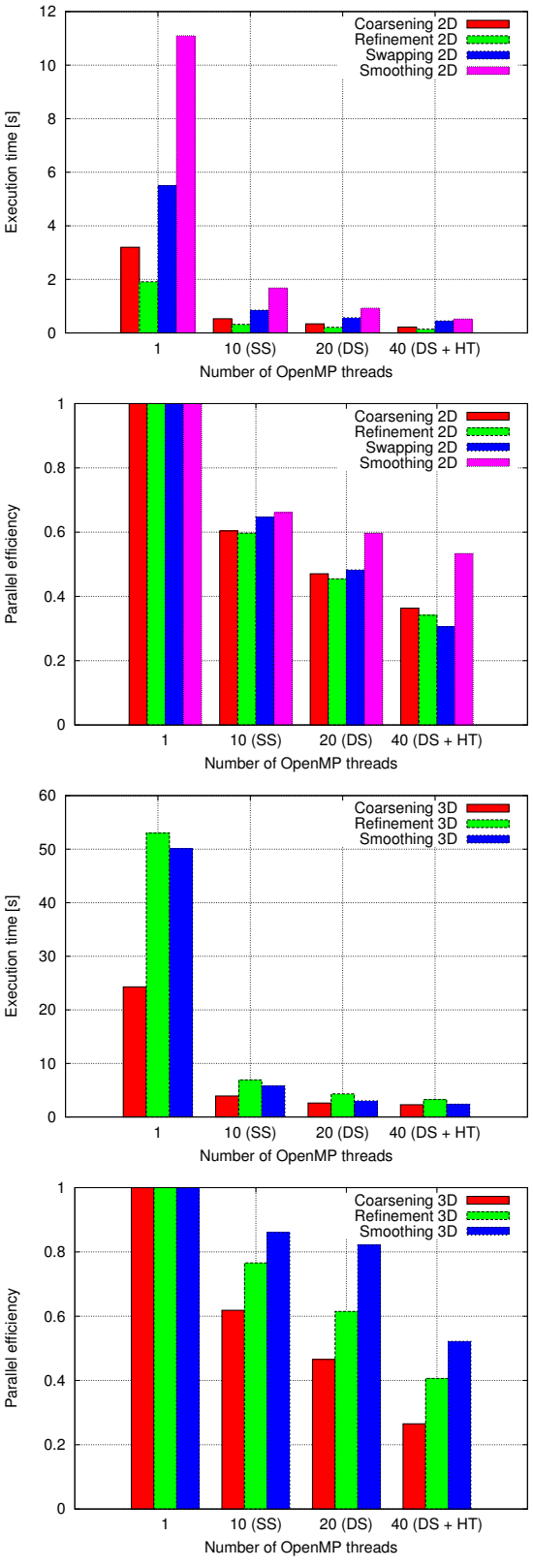



**Figure 4: Execution time and parallel efficiency of 2D and 3D synthetic benchmarks on the 2x8-core Intel®Xeon® Sandy Bridge system.**

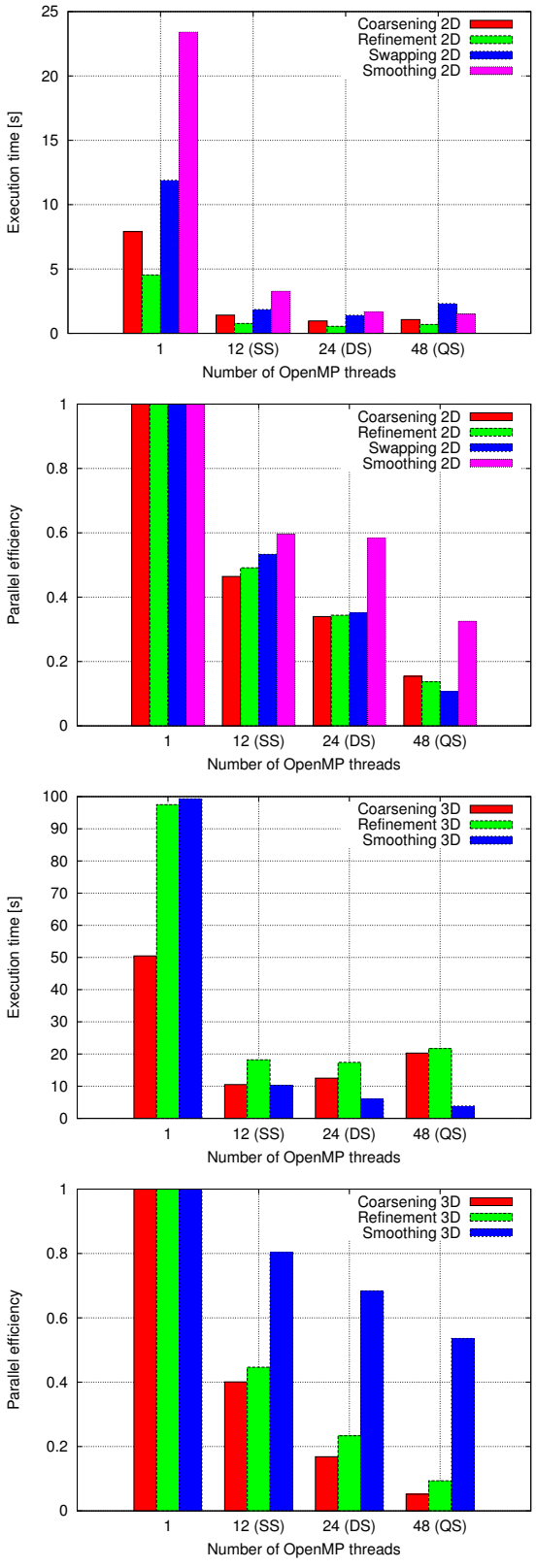



**Figure 5: Execution time and parallel efficiency of 2D and 3D synthetic benchmarks on the 4x12-core AMD Opteron™ Magny-Cours system.**

number. *International Journal for Numerical Methods in Engineering*, 53(6):1377–1391, 2002.

[10] L. F. Freitag, M. T. Jones, and P. E. Plassmann. The Scalability Of Mesh Improvement Algorithms. In *IMA Volumes in Mathematics and its Applications*, pages 185–212. Springer-Verlag, 1998.

[11] P.-J. Frey and F. Alauzet. Anisotropic mesh adaptation for cfd computations. *Computer methods in applied mechanics and engineering*, 194(48):5068–5082, 2005.

[12] X. Li, M. Shephard, and M. Beall. 3d anisotropic mesh adaptation by mesh modification. *Computer methods in applied mechanics and engineering*, 194(48-49):4915–4950, 2005.

[13] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[14] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0, July 2013.

[15] C. Pain, M. Piggott, A. Goddard, F. Fang, G. Gorman, D. Marshall, M. Eaton, P. Power, and C. De Oliveira. Three-dimensional unstructured mesh ocean modelling. *Ocean Modelling*, 10(1-2):5–33, 2005.

[16] M. D. Piggott, P. E. Farrell, C. R. Wilson, G. J. Gorman, and C. C. Pain. Anisotropic mesh adaptivity for multi-scale ocean modelling. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 367(1907):4591–4611, 2009.

[17] S. Pramanik, G. Kulukuru, and M. Mishra. Miscible viscous fingering: Application in chromatographic columns and aquifers. In *COMSOL conference, Bangalore*, 2012.

[18] G. Rokos, G. J. Gorman, and P. H. J. Kelly. A Fast and Scalable Graph Coloring Algorithm for Multi-core and Many-core Architectures. *pre-print*, 2015. `http://arxiv.org/abs/1505.04086`.

[19] G. Rokos, G. J. Gorman, and P. H. J. Kelly. An Interrupt-Driven Work-Sharing For-Loop Scheduler. *pre-print*, 2015. `http://arxiv.org/abs/1505.04134`.

[20] J. Southern, G. Gorman, M. Piggott, and P. Farrell. Parallel anisotropic mesh adaptivity with dynamic load balancing for cardiac electrophysiology. *Journal of Computational Science*, 2011.

[21] K. Suresh. A 199-line matlab code for pareto-optimal tracing in topology optimization. *Structural and Multidisciplinary Optimization*, 42(5):665–679, 2010.

[22] Y. Vasilevskii and K. Lipnikov. An adaptive algorithm for quasioptimal mesh generation. *Computational mathematics and mathematical physics*, 39(9):1468–1486, 1999.

# Paradigm Shift for EXASCALE Computing

George Matheou
Department of Computer Science,
University of Cyprus, Cyprus
geomat@cs.ucy.ac.cy

Paraskevas Evripidou
Department of Computer Science,
University of Cyprus, Cyprus
skevos@cs.ucy.ac.cy

Costas Kyriacou
Computer Science and Engineering Department,
Frederick University, Cyprus
eng.kc@fit.ac.cy

## ABSTRACT

In this paper we propose a paradigm shift for exascale computing by using a Hybrid Data-Flow/Control-Flow model of execution. Programming of High Performance Computers is mainly done through parallel extension of the sequential model like MPI and OpenMP. Even though these extensions facilitate high productivity parallel programming, they suffer from the inability to tolerate long latencies.

The Data-Flow model of execution enforces only a partial ordering as dictated by the true data-dependencies. This is very beneficial for parallel processing because it allows to exploit the maximum parallelism. Furthermore it tolerates synchronization and communication latencies. We believe that a paradigm shift to a hybrid Data-Flow and Control-Flow system will improve the performance of High Performance Computing (HPC).

Data Driven Multithreading (DDM), a threaded Data-Flow programming/execution model, could be the platform for the HPC paradigm shift. Our work on DDM showed that DDM can efficiently run on state-of-the-art sequential machines, resulting in a Hybrid Data-Flow/Control-Flow system.

Evaluation results of DDM implementations on a variety of platforms showed that DDM can indeed tolerate synchronization and communication latency. When comparing DDM with OpenMP, DDM performed better for all benchmarks used. This is primarily due to the fact that DDM effectively tolerates latency. Similar results were also obtained when comparing DDM implemented on a Cell processor, with CellSs and Sequoia.

## Keywords

Parallel Programming, Data-Flow, Data Driven Multithreading

## 1. INTRODUCTION

High Performance Computers and Supercomputers target large problems that have a high degree of parallelism. Programming of such machines is mainly done through parallel extensions of the sequential models like MPI and OpenMP. These extensions do facilitate high productivity parallel programming, but also suffer from the limitations of the sequential synchronization and their inability to tolerate long latencies. Arvind and Iannucci [4], Data-Flow proponents, have been warning us since the 1980's about the two fundamental issues in Multiprocessing: "long memory latencies and waits due to synchronization events". In their quest to develop an exascale supercomputer, the United States, through its DARPA agency, commissioned a study [14, 15] in 2007 to determine what kind of technologies will be needed to build such a supercomputer. Peter Kogge, the leader of the DARPA/USA study group for exascale computing, confirmed that the communication and synchronization latencies of the sequential model are getting out of hand for HPC/Exascale machines. Furthermore, Kogge, states that the power consumption of an exascale computer, will be around 500 MW. Michael Flynn stated in his keynote speech at FPL 2012 [9] that "We have multi-threaded, superscalar cores with limited ILP; worse yet, most of the die area (80%) is devoted to two or three levels of cache to support the illusion of sequential model". Thus, a paradigm shift for exascale computing is necessary.

Data-Flow is a programming/execution model that provides tolerance to communication and synchronization latencies. Data-Flow has been proposed by a number of researchers as an alternative to the Control-Flow model [7, 11, 4]. A paradigm shift to Data-Flow based systems can reduce the power consumption by architectural and organization optimizations. Deterministic prefetching to scratch-pad memories, driven by Data-Flow, can reduce the SDRAM needs of the multi/many-core systems considerably. Furthermore, Data-Flow systems do not need complex modules such as out-of-order execution, thus, the real-estate needs and the power consumption could be reduced. The partial ordering of the Data-Flow model reduces significantly the synchronization latencies to only the true data-dependencies. Data-Flow does not need cache coherence because it adheres to the single assignment semantics. Thus, there is no need for barriers and critical sections. In short, the Data-Flow model can address some of the major issues that can help towards the development of an exascale supercomputer.

SWARM (SWiFt Adaptive Runtime Machine) aims in Scalable Performance Optimizations for multi-core/multi-node systems. SWARM is based on the codelets model which was developed by the Data-Flow group of an earlier Data-Flow based project, the EARTH project [12]. SWARM divides a program into tasks, with runtime dependencies and constraints that can be executed when all runtime dependencies and constraints are met. The runtime schedules the tasks for execution based on resource availability. SWARM is object oriented with a two-level threading system: the first

level is heavy-weight bound to processing resources, and the second level it consists of light-weight threads that run non-preemptively. SWARM was compared with the current supercomputing state-of-the-art: OpenMP and MPI. Results demonstrated that SWARM can outperform OpenMP even in embarrassingly parallel applications such as the Barrnes-Hut. When compared with MPI on the Graph500 benchmark, SWARM showed consistent speedups up to 14.5 on four supercomputers: Sandia Redsky with 8 processors per node, and TACC Lonestar, Intel Endeavor, ORNL Jaquar with 12 processors per node.

Data-Driven Multithreading (DDM) [17] is an execution model that allows Data-Driven scheduling on sequential processors. The core of the DDM model is the Thread Scheduling Unit (TSU) which schedules threads dynamically at runtime based on data availability. In DDM, a program is divided into a number of threads. For each thread it collects meta-data that enable the TSU to manage the dependencies among the threads and determine when a thread can be scheduled for execution. Data-Driven scheduling enforces only a partial ordering as dictated by the true data-dependencies which is the minimum synchronization possible. This is very beneficial for parallel processing because it exploits the maximum possible parallelism. Furthermore, DDM can be implemented on state-of-the-art Control-Flow machines, running in a threaded Data-Flow mode, with the programmer or the system being able to switch between the Control-Flow and the Data-Flow modes of execution. Thus, resulting in a Hybrid Data-Flow/Control-Flow system.

The DDM model was evaluated by four different software implementations: the Data-Driven Network of Workstations ($D^2$Now) [17], the Thread Flux Parallel Processing Platform (TFlux) [21], the Data-Driven Multithreading Virtual Machine (DDM-VM) [2], and DDM++, the latest software DDM implementation. DDM++ is an object oriented implementation of DDM that takes advantage of object oriented techniques such as maintainability, re-usability, data-abstraction and encapsulation. DDM++ programs are developed faster and easier because there is no need of using macros, like in the previous DDM implementations. In the previous DDM systems, the threads' code was embodied in the main function of the program, and macros were used for executing them, using GOTO commands. In DDM++ the threads' code is embodied in standard C functions. It is noteworthy that the DDM applications developed in DDM-VM are more than two times larger than in the DDM++ implementation. DDM was also evaluated by two hardware implementations where the TSU with an 8-core processor were built on an FPGA.

Evaluation results of DDM on a variety of platforms showed that DDM can indeed tolerate synchronization and communication latencies. DDM outperformed OpenMP for all benchmarks. Similar results were obtained when comparing DDM implemented on a cluster of four Cell processors, with CellSs and Sequoia.

Our work on CacheFlow, the memory hierarchy system for DDM, showed that the TSU is aware of the threads scheduled for execution in the near future, and hence the data that will be needed in the near future. This enables the implementation of optimized cache placement and replacement policies, resulting in the need of smaller caches, or the replacement of the cache with a small scratch-pad memory. Furthermore, taking advantage of the near future memory references results in more efficient cache replacement policies that reduce bus/network traffic and power consumption.

Data-Flow provides tolerance to communication and synchronization latencies, and has the potential of making the processor smaller and more power efficient. Thus, it makes sense to consider a shift to the Data-Flow paradigm now.

# 2. DATA-DRIVEN MULTITHREADING

The Data-Driven Multithreading (DDM) [17] is a non-blocking multithreading model that allows data-driven scheduling on sequential processors. A DDM thread (called DThread) is scheduled for execution after all of its required data have been produced, thus no synchronization or communication latencies are experienced after a DThread begins execution. In the DDM model, DThreads have producer-consumer relationships. DThreads' instructions are executed by the CPU sequentially in a Control-Flow manner. This allows the exploitation of Control-Flow optimizations, either by the CPU at runtime or statically by the compiler.

In DDM, a program consists of the DThreads code, the Thread Templates and the Dependency Graph. A Thread Template holds the meta-data of a DThread. The latter describes the consumer-producer dependencies amongst the DThreads. DDM is utilizing the Thread Scheduling Unit (TSU), a special module responsible for scheduling the DThreads in a data-driven manner. The TSU uses the Thread Templates and the Dependency Graph to schedule DThreads for execution when all of their producer-threads completed their execution. This ensures that all data needed by a DThread is available, before it is scheduled for execution.

## 2.1 DDM Implementations

The DDM model was evaluated by several software implementations. The first implementation, the $D^2$Now [17], was targeting Networks of Workstations. It has illustrated the major components of DDM such as the TSU and CacheFlow [16]. The evaluation was done using execution driven simulations. That was followed by two other implementations, the TFlux [21] and the DDM-VM [2]. Both TFlux and DDM-VM were targeting data-driven concurrency on sequential multiprocessors. DDM-VM supported distributed multi-core systems for both homogeneous and heterogeneous systems. TFlux also developed the TFlux directives and a source-to-source compiler. The TFlux compiler was gradually extended to support all DDM systems. The latest software DDM implementation, the DDM++, is an object oriented implementation that takes advantage of object oriented techniques such as maintainability, re-usability, data-abstraction and encapsulation.

DDM was also evaluated by two hardware implementations. In the first one, the TSU was implemented as a hardware peripheral in the Verilog language and it was evaluated through a Verilog-based simulation [18]. The results show that the TSU module can be implemented on an FPGA device with a moderate hardware budget. The second one [19] was the full

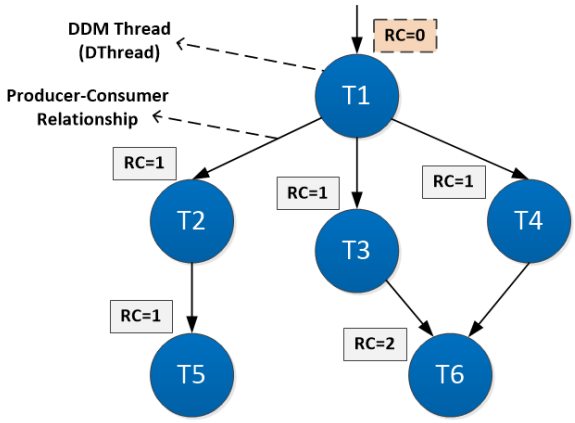



**Figure 1: Example of a DDM Dependency Graph.**

hardware implementation with an 8-core system. A software API and a source-to-source compiler were provided for developing DDM applications. For evaluation purposes, a Xilinx ML605 Evaluation Board with a Xilinx Virtex-6 FPGA was used. This implementation showed that data-driven execution can be implemented on sequential multi-core systems with very small hardware budget and negligible overheads.

## 2.2 The DDM Dependency Graph and Thread Context

The DDM Dependency Graph is a directed graph where the nodes represent the DThreads and the arcs represent the data dependencies amongst the DThreads. Each DThread is paired with a special value called Ready Count (RC) that represents the number of its producers. A simple example of a Dependency Graph is shown in Figure 1 which is composed of six DThreads. The RC values are depicted as shaded values next to the nodes. The DThreads T2, T3 and T4 have one producer, the T1, as such their RC is set to 1. The DThread T5 has also RC=1 since it has only one consumer, the DThread T2. The T6's RC is equal to 2 because it has two producers. The RC value is initiated statically and is dynamically decremented by the TSU each time a producer completes its execution. A DThread is deemed executable when its RC value reaches zero, such as the DThread T1 of the Figure 1.

The Context attribute is a value that enables multiple instances of the same DThread to co-exist in the system and run in parallel. This is essential for programming constructs such as loops and recursion. This idea was based on the U-Interpreter's tagging system [5] which provides a formal distributed mechanism for the generation and management of the tags at execution time. This system was used in Dynamic Data-Flow architectures to allow loop iterations and subprogram invocations to proceed in parallel via the tagging of data tokens [11].

The Context attribute in the previous DDM implementations was implemented as a 32-bit integer value. In DDM++, we extend it into a 96-bit value in order to provide more flexibility to the programmers. Figure 2 depicts a simple example of using multiple instances of the same DThread

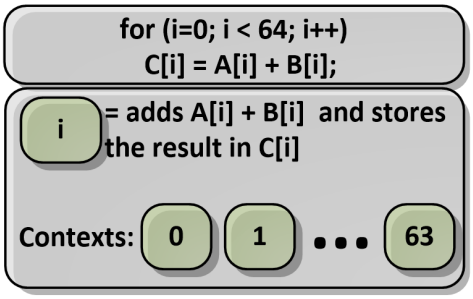



**Figure 2: Example of using multiple instances of the same DThread.**

through the Context attribute. The for-loop shown on the top of the figure is fully parallel, thus it can be executed by only one DThread. Each instance of the DThread is identified by the Context and it executes the inner command of the for-loop. The for-loop is executed 64 times, thus 64 instances are created with Contexts from 0 to 63.

# 3. PERFORMANCE ANALYSIS AND PERFORMANCE EVALUATION

The DDM model was evaluated, in the past, by a number of implementation, software and hardware. To demonstrate the benefits of DDM we show an overview of the performance evaluation results achieved by three different implementations: the Data-Driven Network of Workstations ($D^2$Now) with CacheFlow, the Data-Driven Multithreading Virtual Machine (DDM-VM) [3] and DDM++.

## 3.1 $D^2$Now Cacheflow:

$D^2$Now has shown that scheduling based on data availability can be used to exploit cache management policies that reduce significantly cache misses. Such policies include firing a thread for execution only if its data is already placed in the cache. Furthermore, two optimizations have been developed. The first optimization, called the False Conflict Avoidance, prevents the prefetcher from replacing cache blocks required by the threads waiting to be executed. The second optimization, called the Thread Reordering, attempts to exploit locality by scheduling the multiple invocations of threads in a sequential order on a single processor. We call this cache management policy, the CacheFlow policy.

Figure 3 shows the effect of the CacheFlow optimizations on the average cache miss rate between eight applications, as well as the average speedup achieved on a 32-node system. As shown in Figure 3, the baseline DDM configuration shows a higher miss rate than the sequential as expected (increase from 6.9% to 9.8%), which corresponds to a 42% increase for the average of all applications. This reflects the loss of locality for both the code and data. The basic Prefetch CacheFlow implementation reduces the miss rate from 9.8% to 3.1% (68%) compared to the baseline DDM. It is important to notice that the reduction, achieved by the basic Prefetch CacheFlow, results in miss rate values lower than the original sequential execution. The use of the other two CacheFlow optimizations, False Conflict Avoidance and Thread Reordering, results in further reductions on the miss rate, which becomes 2.0% and 1.4%, respectively.

Employing CacheFlow with both optimizations resulted in a speedup increase from 19.7 to 26.

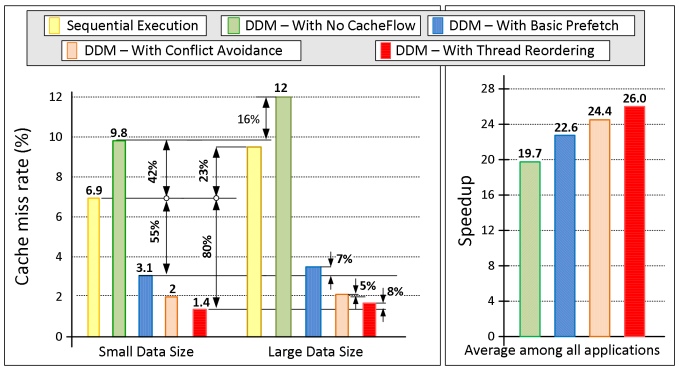



**Figure 3: Cacheflow Policies optimization results.**

## 3.2 DDM-VM$_c$:

DDM-VM [3] developed two virtual machines, one for Homogeneous (DDM-vm$_s$) and one for Heterogeneous (DDM-VM$_c$) systems. Both virtual machines support distributed execution and have achieved good results [3]. DDM-VM$_c$ has targeted the CELL processor. An automated prefetching mechanism, called S-Cacheflow, was developed which moves in Local Store of the SPEs the data required by the thread that will be executed. When a thread terminates, it moves that data to the shared memory of the PPE. DDM-VM$_c$ achieves very good results. For the Matrix Multiplication it achieved an average of 88% of the theoretical peak performance for the 2048 size and an average of 86% and 76% for the 1024 and 512 sizes respectively[2].

Figure 4 shows the comparison of DDM-VM$_c$ versus CellSs for the Matrix Multiplication (MatMult) and Cholesky. DDM-VM$_c$ outperforms CellSs for the entire range for both applications. DDM-VM$_c$ achieves an average improvement of 80% for the 512 size, 28% for 1024 and 19% for the 2048 size for MatMult. An improvement of 213% for 512, 99% for 1024 and 23% for 2048 is achieved for Cholesky. We attribute this to the fact that CellSs schedules annotated tasks at run-time, based on data-dependencies, by building the dependency graph at runtime. On the other hand, in our model we create the dependency graph statically which minimizes the scheduling overheads. Moreover, CellSs makes only a part of the graph available to the scheduler and consequently a fraction of the concurrency opportunities in the applications is visible at any time. DDM-VM$_c$ achieves the best improvement vs. CellSs for the smaller problem sizes, which indicates that it introduces less overhead for exploiting concurrency.

DDM-VM$_c$ was also compared with Sequoia [8] and it achieved an average improvement of 25% for Conv2D and 93% for Matrix Multiplication (Figure 5). Sequoia is a programming language that facilitates the development of memory hierarchy aware parallel programs. It provides a source-to-source compiler and a runtime system for the Cell. Unlike DDM-VM$_c$, Sequoia requires the use of special language constructs and focuses on portability. DDM-VM$_c$ focuses on high performance.

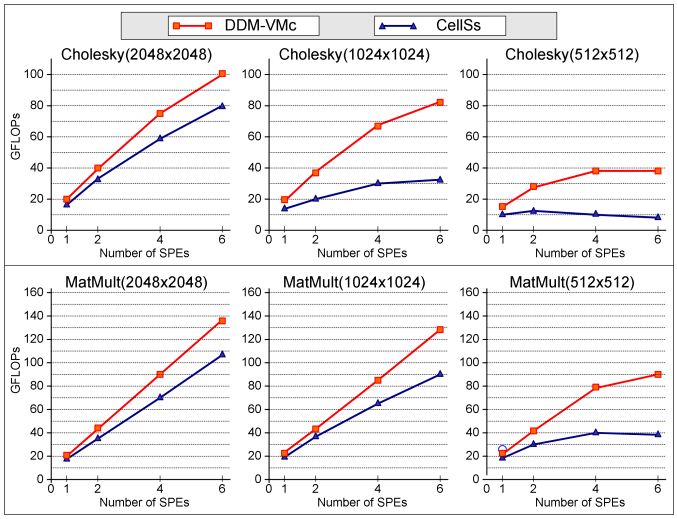



**Figure 4: DDM-VM$_c$ versus CellSs.**

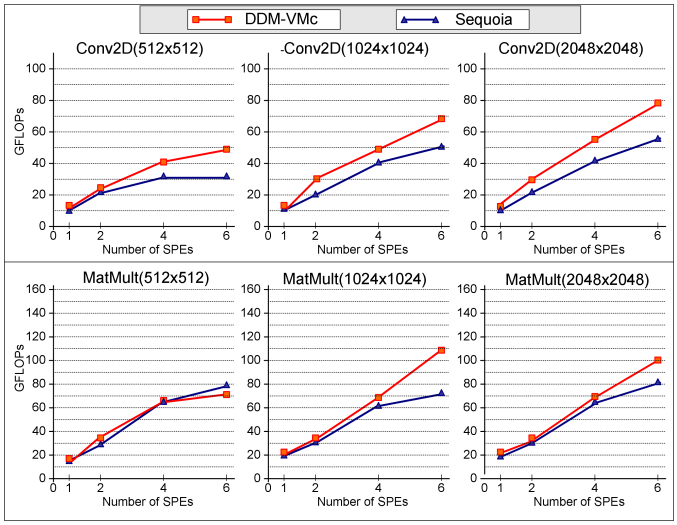



**Figure 5: DDM-VM$_c$ versus Sequoia.**

## 3.3 DDM++:

The latest implementation of the DDM is the DDM++. To evaluate the DDM++ architecture we have used an HP server machine with 2 AMD Opteron 6276 processors running at 1.4GHz. Each processor is an 8-core 64 bit Clustered Multi-Threaded (CMT) with the capacity of running 16 threads simultaneously, resulting in a total of 32 cores. Each core has a 16KB 4-way set associative L1 data cache, a 64K 2-way set associative L1 instruction cache and a 2MB 16-way set associative L2 cache. Also, the system utilizes 12MB 64-way set associative L3 cache. The server has a 48GB DDR3 RAM clocked at 1333MHz. Out of the 32 cores, one is used to implement the TSU function, while the rest are used as the computation units. This corresponds to a single cluster DDM++ machine.

For the performance evaluation of our system we use a suite of three different benchmarks: the Blocked Matrix Multiplication (BMMULT), the Blocked Cholesky Factorization

(Cholesky) and the Blocked LU Decomposition (LU), and compared the speedup results achieved, to those achieved by running the same benchmarks using their OpenMP optimized implementations. For all three benchmarks we have used different problem sizes.

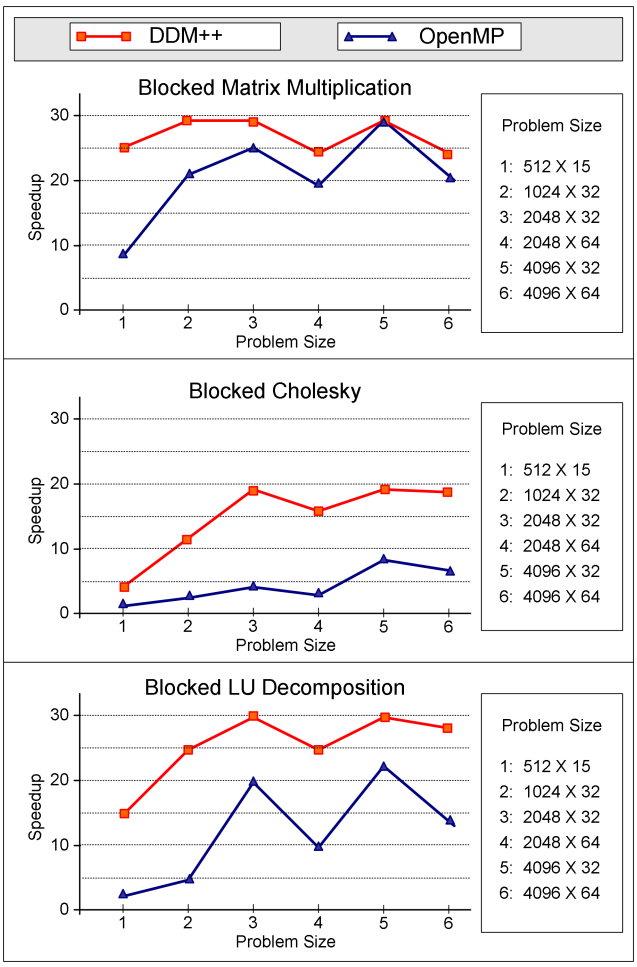



**Figure 6: DDM++ vs OpenMP.**

Figure 6 shows the speedups achieved for all three benchmarks for the different problem sizes. In the case of BMMULT, for small problem sizes, DDM++ achieves better performance than OpenMP. This is primarily due to the thread creation overheads of the OpenMP that are very high compared to the thread execution time. It should be noted that in DDM, thread switching overheads are very low, since it is mainly required to initialize the data frame pointers and load the program counter with the address of the first instruction of the thread. For medium and large problem sizes both implementations achieved good performance since Matrix Multiplication does not have complex data dependencies.

In contrast to the BMMULT, Cholesky is a complex application with strict data dependencies. As shown in Figure 6, DDM++ achieves much better performance results compared to the OpenMP implementation. This is due to the DDM ability to tolerate synchronization latency by decoupling the synchronization from computations, and allowing the TSU core to operate asynchronously from the computation cores. Furthermore, the speedup achieved by DDM++ does not vary significantly with the problem size. This is due to the low parallelization and thread switching overheads of the DDM model, as opposed to the OpenMP results, where speedup is reduced significantly as problem size decreases.

The LU Decomposition benchmark has similar characteristics with the Cholesky benchmark, with respect to the complexity and data dependencies. As shown in Figure 6, the results achieved for LU are similar to those of the Cholesky, in the sense that DDM++ achieves better performance than OpenMP, and that speedup does not vary significantly with the problem size. The reason for this is the ability of the DDM model to tolerate synchronization latency and amortize parallelization overheads. It should be noted that the speedups achieved for the LU benchmark are significantly higher than those of the Cholesky.

The results achieved for all benchmarks show that DDM++ effectively leverages the decoupling of synchronization and execution for the maximum tolerance of synchronization overheads.

## 4. RELATED WORK

A number of threaded Data-Flow architectures and execution models have been proposed within the context of multi/many-core systems. Scheduled Data-Flow (SDF) [13] is a non-blocking decoupled memory/execution multithreaded architecture, where a program is partitioned into non-blocking computation threads and memory-access threads.

The Decoupled Threaded Architecture-Clustered (DTA-C) [10] is an architecture that is based on the SDF architecture with the addition of the concept of clustering resources. As the name implies, the architecture is composed of a set of clusters or tiles. The Explicit Data Graph Execution Architecture (EDGE) [6] proposes an ISA that supports direct instruction communication that expresses the Data-Flow graph the compiler generates.

The Fuce processor [1] is based on the Data-Flow-like continuation based multithreading model. A thread is defined as a block of instructions that work on registers (except for loads and stores) and is executed without interruption until completion.

Star Superscalar (StarSs) [20] is a parallel programming platform that targets symmetric multiprocessors and multicores, the Cell processor and GPUs. It schedules annotated tasks at run-time based on data-dependencies. Unlike the approach adopted by our work, where we build the dependency graph statically, StarSs always builds its task dependency graph at run-time. This approach incurs extra overheads as it resolves the dependencies at run-time even if they can be resolved at compile-time.

## 5. CONCLUSIONS

In this paper we make the case that a paradigm shift to a parallel execution model, based on Data-Flow, can facilitate the development of an efficient Exascale Supercomputer. We propose that hybrid Data-Flow/Control-Flow systems can

combine their strong points in the quest for an exascale computing. Supercomputing is a niche market that strives for highest possible performance. Such a niche market could easily accept a paradigm shift if it can improve performance.

Hybrid Data-Flow/Control-Flow systems combine the best characteristics of each model. Data-Flow is a parallel model that enforces a partial ordering as determined by true data dependencies. This enhances the parallelism present in a program and avoids barriers and critical sections, which reduces the synchronization overheads and thus increases the utilization of the machines.

Data-Flow has tolerance to latency and provides single assignment semantics that avoid side effects. Threaded Data-Flow systems, like DDM, provide Data-Flow concurrency among threads and execute the threads in a Control-Flow matter. Such systems do not need out-of-order execution or cache coherence mechanisms. Furthermore, they can implement a light-way memory hierarchy with automated deterministic prefetching into scratch-pad memories. Thus, reducing further the hardware complexity and the power consumption.

# 6. REFERENCES


[1] S. Amamiya and et al. Fuce: The continuation-based multithreading processor. In *Proceedings of the 4th Int.Conference on Computing Frontiers*.

[2] S. Arandi and P. Evripidou. Programming multi-core architectures using data-flow techniques. pages 152–161. IEEE, July 2010.

[3] S. Arandi and P. Evripidou. DDM-VMc: the data-driven multithreading virtual machine for the cell processor. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 25–34. ACM, ACM Press, 2011.

[4] Arvind and D. E. Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1):225–253, June 1986.

[5] Arvind and Gostelow. The U-interpreter. *Computer*, 15(2):42–49, Feb. 1982.

[6] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[7] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag.

[8] K. Fatahalian and et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[9] M. Flynn. Dataflow supercomputers, keynote speech fpl2012. 2012.

[10] R. Giorgi, Z. Popovic, and N. Puzovic. Dta-c: A decoupled multi-threaded architecture for cmp systems. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 263–270. IEEE, 2007.

[11] J. R. Gurd, C. C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[12] H. H. J. Hum and et al. A design study of the earth multiprocessor. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT '95, pages 59–68, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[13] K. M. Kavi, R. Giorgi, and J. Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *Computers, IEEE Transactions on*, 50(8):834–846, 2001.

[14] P. Kogge. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, University of Notre Dame, 2008.

[15] P. Kogge. The tops in flops (Next generation supercomters). *IEEE Spectrum*, 48(2):48–54, Feb. 2011.

[16] C. Kyriacou, P. Evripidou, and P. Trancoso. Cacheflow: A short-term optimal cache management policy for data driven multithreading. In *Euro-Par 2004 Parallel Processing*, pages 561–570. Springer, 2004.

[17] C. Kyriacou, P. Evripidou, and P. Trancoso. Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1176–1188, Oct. 2006.

[18] G. Matheou and P. Evripidou. Verilog-based simulation of hardware support for data-flow concurrency on multicore systems. In *SAMOS XIII, 2013*, pages 280–287. IEEE, 2013.

[19] G. Matheou and P. Evripidou. Architectural support for data-driven execution. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):52, 2015.

[20] J. Planas and et al. Hierarchical task-based programming with starss. *International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

[21] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. TFlux: a portable platform for data-driven multithreading on commodity multicore systems. pages 25–34. IEEE, Sept. 2008.

# Large-scale Ultrasound Simulations Using the Hybrid OpenMP/MPI Decomposition

Jiri Jaros
Faculty of Information Technology
Brno University of Technology
Bozetechova 2
612 66 Brno, CZ
jarosjir@fit.vutbr.cz

Vojtech Nikl
Faculty of Information Technology
Brno University of Technology
Bozetechova 2
612 66 Brno, CZ
inikl@fit.vutbr.cz

Bradley E. Treeby
Dept. of Medical Physics and Biomedical Engineering
University College London
Malet Place Eng Bldg
London WC1E 6BT, UK
b.treeby@ucl.ac.uk

## ABSTRACT

The simulation of ultrasound wave propagation through biological tissue has a wide range of practical applications including planning therapeutic ultrasound treatments of various brain disorders such as brain tumours, essential tremor, and Parkinson's disease. The major challenge is to ensure the ultrasound focus is accurately placed at the desired target within the brain because the skull can significantly distort it. Performing accurate ultrasound simulations, however, requires the simulation code to be able to exploit several thousands of processor cores and work with datasets on the order of tens of TB. We have recently developed an efficient full-wave ultrasound model based on the pseudospectral method using pure-MPI with 1D slab domain decomposition that allows simulations to be performed using up to 1024 compute cores. However, the slab decomposition limits the number of compute cores to be less or equal to the size of the longest dimension, which is usually below 1024.

This paper presents an improved implementation that exploits 2D hybrid OpenMP/MPI decomposition. The 3D grid is first decomposed by MPI processes into slabs. The slabs are further partitioned into pencils assigned to threads on demand. This allows 8 to 16 times more compute cores to be employed compared to the pure-MPI code, while also reducing the amount of communication among processes due to the efficient use of shared memory within compute nodes.

The hybrid code was tested on the Anselm Supercomputer (IT4-Innovations, Czech Republic) with up to 2048 compute cores and the SuperMUC supercomputer (LRZ, Germany) with up to 8192 compute cores. The simulation domain sizes ranged from $256^3$ to $1024^3$ grid points. The experimental results show that the hybrid decomposition can significantly outperform the pure-MPI one for large simulation domains and high core counts, where the efficiency remains slightly below 50%. For a domain size of $1024^3$, the hybrid code using 8192 cores enables the simulations to be accelerated by a factor of 4 compared to the pure-MPI code. Deployment of the hybrid code has the potential to eventually bring the simulation times within clinically meaningful timespans, and allow detailed patient specific treatment plans to be created.

## Keywords

Ultrasound simulations; 2D domain decomposition; OpenMP/MPI Hybrid programming; Performance evaluation; Supercomputing, k-Wave toolbox.

## 1. INTRODUCTION

The simulation of ultrasound wave propagation through biological tissue has a wide range of practical applications. Recently, high intensity focused ultrasound has been applied to functional neurosurgery as an alternative, non-invasive treatment of various brain disorders such as brain tumours, essential tremor, and Parkinson's disease. The technique works by sending a focused beam of ultrasound into the tissue, typically using a large transducer. At the focus, the acoustic energy is sufficient to cause cell death in a localised region while the surrounding tissue is left unharmed. The major challenge is to ensure the focus is accurately placed at the desired target within the brain because the skull can significantly distort it.

Performing accurate ultrasound simulations, however, requires the simulation code to be able to operate on large domains and deliver the results in a clinically meaningful time. Apart from the physical complexity, the main obstacle in implementing new ultrasound treatment planning procedures in clinical practice is the computational complexity. Considering the domain of interest encompassing the ultrasound transducer and the treatment area (normally on the order of centimetres in each Cartesian direction), and the size of the acoustic wavelength (on the order of hundreds of micrometers at the maximum frequency of interest), we have to simulate the wave propagation over hundreds or thousands of wavelengths. A sufficiently fine discretisation of the simulation domain which avoids numerical dispersion and instability can easily lead to grid sizes exceeding $10^{12}$ elements. Storing all the necessary acoustic quantities for such a large simulation domain in computer memory requires petabytes of memory and its processing reaches the order of exascale.

We have recently developed a pure-MPI pseudospectral simulation code using 1D domain decomposition that has allowed us to run reasonable sized simulations using up to 1024 compute cores [3]. However, this implementation suffers from the maximum parallelism being limited by the largest size of the 3D grid used. At the age of exascale, more and more systems will have numbers of processing cores far exceeding this limit. For example, a realistic ultrasound simulation performed by the k-Wave toolbox might use a grid size of $1024^3$. Here, the 1D pure-MPI decomposition would only scale up to 1024 cores at most leading to calculation times exceeding clinically acceptable times (in this case between 30 and 72 hours). In contrast, top supercomputer facilities dispose with several hundred thousand compute cores and could provide the simulation result within an hour, if efficiently employed.

The second problem arising from limited parallelism is the total amount of memory that can be used to store simulation data. Not

scaling the code to larger core counts holds the simulation domain size below $4096^3$, which is not enough for some clinical applications (e.g., the use of shocked waves to vaporise a piece of tissue which can produce hundreds of harmonics).

This paper presents an improved implementation that exploits a 2D hybrid OpenMP/MPI decomposition. The 3D grid is first decomposed by MPI processes into slabs. The slabs are further partitioned into pencils assigned to threads on demand. This is supposed to (i) exploit shared memory within nodes and limit inter-process communication, (ii) employ 8 to 16 times more compute cores, (iii) increase the overall memory capacity while reducing the communication time.

# 2. DISTRIBUTED IMPLEMENTATION OF ULTRASOUND SIMULATIONS

The k-Wave toolbox [8] is designed to simulate ultrasound wave propagation in soft-tissues and bone, modelled as fluid and elastic media, respectively. In the k-Wave toolbox, the k-space pseudospectral method is used to solve the system of governing equations described in detail by Treeby in [9]. These equations are derived from the mass conservation law, momentum conservation law, and an empirically derived acoustic pressure-density relation that accounts for acoustic nonlinearity, absorption, and heterogeneity in the material properties [9].

The k-space and pseudospectral methods gain their advantage over finite difference methods due to the global nature of the spatial gradient calculations [4]. This permits the use of a much coarser grid for the same level of accuracy. However, the global nature of the gradient calculation, in this case using the 3D fast Fourier transform (FFT), introduces additional challenges for the development of an efficient parallel code. Specifically, the FFT requires a globally synchronising all-to-all data exchange. This global communication can become a significant bottleneck in the execution of spectral models. Fortunately, considerable effort has already been devoted to the development of distributed memory FFT libraries that show reasonable scalability of up to tens of thousands of processing cores [2], [5], [7].

The distributed implementation was written in C++ as an extension to the open-source k-Wave acoustics toolbox [8]. The standard message passing interface (MPI) was used to perform all interprocess communications, the MPI version of the FFTW library was used to perform the Fourier transforms [2], and the input/output (I/O) operations were performed using the HDF5 library [1]. To maximise performance, the code was also written to exploit single instruction multiple data (SIMD) instructions such as SSE or AVX. A detailed description can be found in [3]. The simulation time loop can be broken down into several phases:

1. The gradient of acoustic pressure is calculated by the Fourier collocation spectral method. This operation requires one forward 3D FFT and a few element-wise operations.
2. The acoustic particle velocity (a 3D vector) is calculated based on the acoustic pressure gradient using three inverse 3D FFTs and a few element-wise operations.
3. The gradients of particle velocity for each spatial dimension are calculated using three forward and three inverse 3D FFTs interleaved by several element wise operations.
4. The acoustic density is updated based on the particle velocity gradients using several element-wise operations.
5. The acoustic pressure field is updated based on the particle velocity gradients, acoustic density, and the non-linearity and absorption operators. This step includes two forward and two inverse 3D FFTs, and several elementary element-wise operations such as multiplication, addition, division, etc.
6. The desired acoustic quantities are sampled in regions of interest and either stored on the disk as time-varying series or further processed to calculate e.g. maximum, average, RMS, etc.

There are two important features of the time loop that should be highlighted. First, there are only two places where communication among MPI processes is required. It is within the 3D FFT while performing the distributed matrix transposition, and while the data is being sampled, collected, and stored using the parallel HDF5 library. To reduce the communication burden, pairs of forward and inverse FFTs do not bring the data into the original shape in between, instead a transposed shape is used to reduce the amount of communication to one half [3]. Moreover, the output data is collected and stored using chunks enabling buffering and staging of I/O operations. The second observation is that the simulation time loop is dominated by the FFT calculation. This accounts for nearly 60-80% (the higher number of processes, the higher proportion) of the execution time while the rest of the element-wise operations and the I/O only contribute by 40-20% [3]. Moreover, the FFT itself spends the vast majority of its time waiting for data being transmitted and transposed over the network.

The following subsections describe two different decompositions of the 3D simulation space we have developed: the 1D pure-MPI decomposition and the 2D Hybrid OpenMP/MPI decomposition.

## 2.1 Pure-MPI Decomposition

The pure-MPI decomposition is based on the 1D slab decomposition natively supported by the FFTW library. In this case, the 3D domain is partitioned along the $z$ axis and every MPI process receives a given number of 2D slabs. In practice, all 3D matrices (acoustic pressure, velocity, density, etc.) are partitioned and distributed this way while several other support data structures are either partitioned and scattered or simply replicated [3]. The communication phase consists of one `MPI_Alltoall` communication performed as a part of the FFT, see Fig 1.

It has to be noted, that this decomposition provides reasonable scaling as long as the number of MPI processes is smaller than the $z$ dimension size of the simulation domain. It also allows easy deployment on many supercomputing systems and eliminates problems with proper thread pinning, memory affinity, and so on. However, the disadvantage, apart from the limited number of processes to be used, is the communication overhead. With a growing number of MPI processes, the messages get smaller and smaller, while the number of messages grows with $P^2$. This eventually leads to network congestion and bandwidth decrease caused by the high latency of routing small messages.

## 2.2 Hybrid OpenMP/MPI Decomposition

The hybrid OpenMP/MPI decomposition tries to alleviate the disadvantages of the pure MPI decomposition by introducing a second level of decomposition and further breaking the 1D slabs up into pencils. In contrast to pure-MPI 2D decompositions, the smallest chunk an MPI process can receive still remains a 1D slab. Thus, the total number of MPI processes inherits the same limit as the 1D decomposition presented above. However, in this case, MPI processes are not mapped and bound to all compute cores, but only to one core per socket or node. Once a process is mapped on a socket/node, it spawns several OpenMP threads to process a given number of pencils from the allocated slab/slabs. Considering that

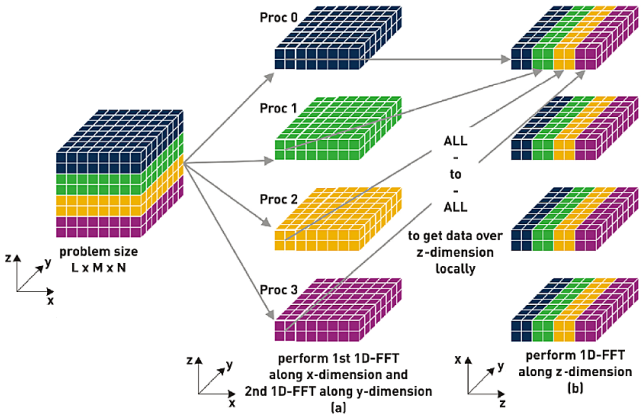



**Figure 1: 1D domain decomposition and communication patterns within a 3D FFT.**

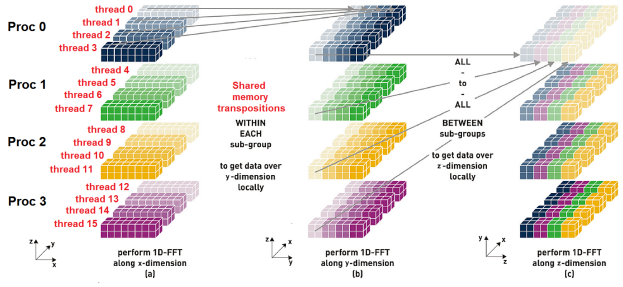



**Figure 2: 2D domain decomposition and communication patterns within a 3D FFT.**

many current supercomputers comprise of shared memory nodes typically integrating two sockets of 8 cores, we are able to scale the simulation up by a factor of 8 or 16. Moreover, the OpenMP threads can employ shared memory to significantly reduce the amount of inter-process communication and help in exploiting local caches.

It should be noted, that the 2D decomposition requires two communication phases to be carried out (one transpose along the $y$ axis followed by another one along the $z$ axis). Pure-MPI approaches typically implement this by a sequence of `MPI_Alltoall` communication over the $y$ and $z$ axis [7], [5]. Since the whole 1D slab is always placed on one socket/node, the hybrid implementation can efficiently employ the shared memory to perform the first transposition. The second transposition is carried out the same way as the 1D decomposition (see Fig 2), however, with a fewer number of processes (fewer and bigger messages, higher bandwidth, etc.).

The hybrid OpenMP/MPI simulation code was implemented in a very similar way to the pure-MPI one. The FFT calculation is based on the FFTW library tuned to be able to work with the 2D decomposition. We used our custom implementation presented in [6]. In a nutshell, it uses OpenMP FFTW kernels to perform series of 1D FFTs, a multi-threaded local transposition accelerated by SIMD instructions, and a distributed transposition offered by the FFTW library to carry out the communication part. This implementation has proved its superiority over pure-MPI approaches and enables better scaling than the original FFTW library (see [6] for more detail).

The element wise operations implemented in various steps of the simulation time loop were merged into a small number of kernels to maximize the temporal locality, written to utilise SIMD extensions, and run in parallel using the OpenMP library. To ensure correct thread and memory affinity, the First Touch Strategy was used.

# 3. EXPERIMENTAL RESULTS

The experimental evaluation of the hybrid decomposition was performed on two supercomputing systems, Anselm and SuperMUC. Anselm is a Czech supercomputer operated by the IT4Innovations National Supercomputing Center in Ostrava, Czech Republic. Anselm is an Intel-infiniband cluster based on Sandy Bridge processors (2x8 core Intel E5-2665 at 2.4GHz and 64GB RAM per node) interconnected by a 40Gb Fat-tree infiniband interconnection. The maximum number of cores we could use was 2048.

SuperMUC is a German supercomputer operated by Gauss Centre for Supercomputing and Leibniz Supercomputer Centre in Munich, Germany. SuperMUC is also an Intel-infiniband cluster based on similar Sandy Bridge CPUs (2x8 core Intel Xeon E5-2680 at 2.7 GHz and 32GB RAM per node) interconnected by a 40Gb Fat-tree infiniband network. The maximum number of cores we could use was 8192.

Comparing the hardware configuration, both systems are very similar and should produce very close results. The software stack on the other hand is different and allows us to check different compilers and MPI libraries. On Anselm, we used a GNU software stack comprising of a GNU C++ compiler (g++-4.8), the OpenMPI library in version 1.8.4, FFTW 3.3.3, and HDF5 1.8.13. The schedule manager is based on the OpenPBS software. SuperMUC on the other hand is based on an Intel software stack including an Intel Compiler 2015, Intel MPI in version 5.0, FFTW 3.3.3 and HDF5 1.8.12. The schedule manager is based on LoadLeveler.

## 3.1 Test configurations

One of the most important issues rising when working with a hybrid OpenMP/MPI code is the proper mapping of MPI processes and threads to cores, sockets and nodes. Improper setting can significantly deteriorate performance by allowing the threads to migrate among cores/sockets and losing the memory affinity. Since the default behaviour of MPI is to bind one process per core, spawning new threads by this process often leads to the threads being bound to the same core. As a consequence, one core is heavily overloaded while others are kept idle. The setting for three test configurations was as follows:

1. **Pure-C** (pure-MPI code, core level mapping) - This configuration uses the pure-MPI code implementing the 1D decomposition compiled without the OpenMP extension. This code is the reference for comparison. No special care has to be taken to run this code.

2. **Hybrid-S** (hybrid code, socket level mapping) - This configuration uses the hybrid OpenMP/MPI code implementing the 2D decomposition compiled with the OpenMP library. The code starts one MPI process per socket and then spawns 8 threads per process. On Anselm, the code was launched with `mpirun -map-by socket -bind-to socket ./executable`, the number of threads was set by environmental variable `OMP_NUM_THREADS=8` pinned by `GOMP_CPU_AFFINITY="0-15"`. On SuperMUC, the LoadLeveler automatically sets all necessary environmental variables when specifying task per nodes equal to 2.

3. **Hybrid-N** (hybrid code, node level mapping) - This configuration uses the hybrid OpenMP/MPI code implementing the 2D decomposition. The code starts one MPI process per node and then spawns 16 threads per process. On

Anselm, the code was launched with `mpirun -map-by node -bind-to none ./executable`, the number of threads was set by `OMP_NUM_THREADS=16` and thread binding by `GOMP_CPU_AFFINITY="0-15"`. On SuperMUC, the LoadLeveler automatically sets all necessary environmental variables when specifying task per nodes equal to 1.

The performance was investigated by a few simulation cases calculating the propagation of nonlinear waves in heterogeneous and absorbing media with a source driven by a sine wave. The domain sizes were chosen to equal $256^3$, $512^3$, and $1024^3$ grid points. We did not test larger domains due to extensive simulation cost and the allocation limits. However, we expect better scaling with large simulation domains. The number of simulation timesteps varied from 100 to 1000 in order to get stable results and run the simulation for a reasonable timespan. The overall simulation run was, however, much longer due to the necessity of FFTW plan creation, which could take up to 30 minutes [3].

## 3.2 Strong Scaling

The strong scaling plots describe how the execution time decreases with increasing number of compute resources. The size of the problem is fixed. Fig. 3 and Fig. 4 show strong scaling for simulation domains of $256^3$ and $512^3$ grid points, respectively, and the number of compute cores growing from 16 (1 node) up to 2048 cores (128 nodes) on the Anselm supercomputer. The curves show the average execution time per one time step of the pure-MPI and two hybrid versions.

It can be seen that the simulation time decreases linearly, slowly reaching a plateau at the end (2048 cores). This is given by the size of the simulation grid, which is simply too small to keep all cores busy; one core only has 8k or 65k grid points to calculate. We can also conclude that the hybrid implementation is not so efficient for small core counts and the Pure-C code beats the hybrid ones almost twice. The clue is hidden in the communication part (the amount of computation is the same in all cases). In the Pure-C code, all cores participate in the communication transposing its part of the grid. However, the hybrid codes only use the master thread to communicate while the others are sleeping. Since the messages are quite big at low core counts, the loss in concurrency affects the performance by a great deal. For the smallest simulation domain size of $256^3$, the hybrid decomposition seems to be inefficient. The Hybrid-S code offers a factor of two in performance, however, when using 8 times more resources. The efficiency is thus very low. For a bigger domain of $512^3$, the hybrid codes scale much better and catches up with the Pure-C code at 128 cores (Hybrid-S version) or 512 cores (Hybrid-N version). The real strength of the hybrid code becomes evident beyond the scaling capability of the Pure-C code (512 cores). The Hybrid-S configuration offers more than 2.3 times higher performance when running on 2048 cores (efficiency of 57% compares to 512 cores).

The same test was also performed on SuperMUC, see Fig. 5. Since having a much bigger allocation here, we used a grid size of $1024^3$ and executed the simulation with core counts ranging from 64 to 8192. Again, the Pure-C code is faster for lower core counts while the hybrid implementations win at the other side of the range. An interesting peak occurs for 2048 cores (Hybrid-S) and 4096 cores (Hybrid-N) where the performance is much lower than expected. This peak was also observed on other grid sizes always at the position where the number of cores is twice as high as the size of $z$ dimension for Hybrid-S version, and four times higher for the Hybrid-N version. When investigating of this phenomenon, we tried different FFTW planning flags (patient and exhaustive), various compiler flags, MPI versions, and pinning strategies, however,

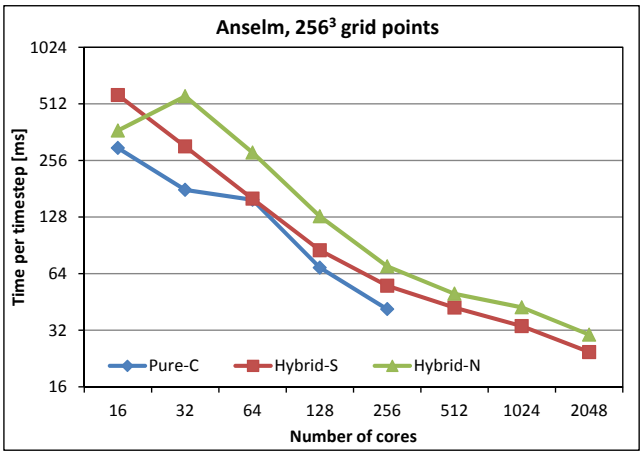



**Figure 3: Strong scaling on Anselm, simulation grid of $256^3$.**

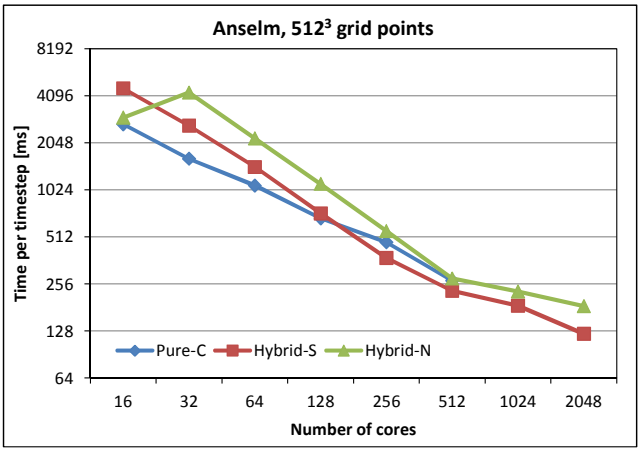



**Figure 4: Strong scaling on Anselm, simulation grid of $512^3$.**

we did not succeed in eliminating this behaviour. We suspect that it has something to do with the critical message size where MPI changes the policy of transmitting messages (sync. vs buffered), or that FFTW is unable to find a good communication plan.

To support this hypothesis we took a simulation flat profile, see Table 1. The peaks in execution time directly correspond to the communication share. In a typical run, the communication share is about 50%, while in those exceptional cases the communication share springs up to 75%. The profile confirmed our hypothesis that the distributed transposition is not done optimally and a custom routine needs to be implemented to ensure the correct behaviour. This table also reveals that the hybrid OpenMP/MPI decomposition bounds the communication at a reasonable level of 50%, even for high core counts.

Fortunately, at least one of the hybrid versions works correctly

**Table 1: Communication share for various core counts and hybrid implementations on SuperMUC (grid $1024^3$).**

| core count | Hybrid-S (MPI share) | Hybrid-N (MPI share) |
|---|---|---|
| 1024 | 51.60% | 46.24% |
| 2048 | 71.48% | 48.95% |
| 4096 | 52.84% | 74.38% |

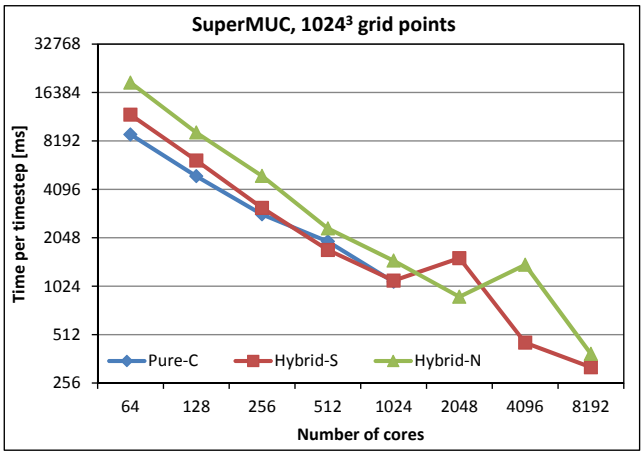



**Figure 5: Strong scaling on SuperMUC, simulation grid of $1024^3$.**

at a given core count and the user has the ultimate choice. Finally, we would like to note that Hybrid-S version offers almost 4 times higher performance over Pure-C, which yields efficacy of almost 50%, which is not so bad considering the code is proven to be communication and memory bound.

## 4. CONCLUSIONS

This paper has presented our first attempt to improve scaling of large-scale ultrasound simulations using the hybrid OpenMP/MPI decomposition. The main goal was to enable the code to employ a number of compute cores exceeding the limit imposed by the standard 1D decomposition (the size of the $z$ dimension). By introducing a second level of decomposition and breaking the 1D slabs assigned to MPI processes into pencils computed by OpenMP threads, as well as eliminating the need for another inter-process transposition by the shared memory, we have been able to accelerate the simulation by a factor of 4. This was achieved on SuperMUC when using 8192 compute cores to compute ultrasound wave propagation over a simulation domain discretised into $1024^3$ grid points. We also managed to keep the communication overhead at an acceptable 50%.

We also observed curious behaviour for some configurations (number of processes and threads) where the simulation time abruptly increased. This may be attributed to the inability of the FFTW to find an optimal communication plan at this configuration. We can also conclude, that the scaling gets better for bigger simulation domains. While for domain sizes of $256^3$ grid points, the hybrid decomposition does not bring much improvement due to the small amount of work, large domains of $1024^3$ and bigger appear to benefit from the additional compute resources very well.

In our future work, we would like to test the code for bigger grid sizes, introduce custom communication plans, and further optimise the simulation code.

## 5. ACKNOWLEDGEMENTS

The project is financed from the SoMoPro II programme. The research leading to this invention has acquired a financial grant from the People Programme (Marie Curie action) of the Seventh Framework Programme of EU according to the REA Grant Agreement No. 291782. The research is further co-financed by the South-Moravian Region. This work reflects only the author's view and the European Union is not liable for any use that may be made of the information contained therein. This work was also supported by the research project "Architecture of parallel and embedded computer systems", Brno University of Technology, FIT-S-14-2297, 2014-2016.

This work was supported by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033).

The authors gratefully acknowledge the assistance of Sebastian Lehrack from the LMU Faculty of Physics, and the Gauss Centre for Supercomputing e.V. (www.gauss-centre.eu) for funding this project by providing computing time on the GCS Supercomputer SuperMUC at Leibniz Supercomputing Centre (LRZ, www.lrz.de).

## 6. REFERENCES

[1] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An Overview of the HDF5 Technology Suite and Its Applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, New York, NY, USA, 2011. ACM.

[2] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[3] J. Jaros, A. P. Rendell, and B. E. Treeby. Full-wave nonlinear ultrasound simulation on distributed clusters with applications in high-intensity focused ultrasound. *The International Journal of High Performance Computing Applications*, 2015(2):1–19, 2015.

[4] T. D. Mast, L. P. Souriau, D.-L. D. Liu, M. Tabei, A. I. Nachman, and R. C. Waag. A k-space method for large-scale models of wave propagation in tissue. *IEEE Trans. Ultrason. Ferroelectr. Freq. Control*, 48(2):341–354, 2001.

[5] P. Michael. PFFT-An extension of FFTW to massively parallel architectures. *Society for Industrial and Applied Mathematics*, 35(3):213–236, 2013.

[6] V. Nikl and J. Jaros. Parallelisation of the 3D Fast Fourier Transform Using the Hybrid OpenMP/MPI Decomposition. In *Mathematical and Engineering Methods in Computer Science*, LNCS 8934, pages 100–112. Springer International Publishing, 2014.

[7] D. Pekurovsky. P3DFFT: A Framework for Parallel Computations of Fourier Transforms in Three Dimensions. *SIAM Journal on Scientific Computing*, 34(4):C192–C209, Jan. 2012.

[8] B. E. Treeby and B. T. Cox. k-Wave: MATLAB toolbox for the simulation and reconstruction of photoacoustic wave fields. *Journal of Biomedical Optics*, 15(2):021314, 2010.

[9] B. E. Treeby, J. Jaros, A. P. Rendell, and B. T. Cox. Modeling nonlinear ultrasound propagation in heterogeneous media with power law absorption using a k-space pseudospectral method. *The Journal of the Acoustical Society of America*, 2012(131):4324–4336, 2012.

# Algorithms in the parallel partitioning tool GridSpiderPar for large mesh decomposition

Evdokia N. Golovchenko
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
Moscow, 125047, Russia
+7(499)791-2917
ge03@imamod.ru

Marina A. Kornilina
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
Moscow, 125047, Russia
+7(499)250-7823
mary@imamod.ru

Mikhail V. Yakobovskiy
Keldysh Institute of Applied Mathematics RAS
Miusskaya sq., 4
Moscow, 125047, Russia
+7(499)250-7823
lira@imamod.ru

## ABSTRACT
The problem of load balancing arises in parallel mesh-based numerical solution of problems of continuum mechanics, energetics, electrodynamics etc. on high-performance computing systems. The program package for parallel large mesh decomposition GridSpiderPar was developed. We compared different partitions into microdomains, microdomain graph partitions and partitions into subdomains of several meshes ($10^8$ vertices, $10^9$ elements) obtained by means of the partitioning tool GridSpiderPar and the packages ParMETIS, Zoltan and PT-Scotch. Balance of the partitions, edge-cut and number of unconnected subdomains in different partitions were compared as well as the computational performance of gas-dynamic problem simulations run on different partitions.

## Keywords
High-performance computing, graph partitioning, mesh decomposition.

## 1. INTRODUCTION
The problem of load balancing arises in parallel mesh-based numerical solution of problems of continuum mechanics, energetics, electrodynamics etc. on high-performance computing systems. Geometric parallelism is commonly used in most of applications for large-scale 3D simulations of the problems listed above. It implies that every branch of an application code processes a subset of computational mesh (a subdomain). In order to increase processors efficiency it is necessary to provide rational domain decomposition, taking into account the requirements of balanced mesh distribution among processors and reduction of interprocessor communications, which depend on the number of bonds between subdomains.

The number of processors to run a computational problem is often unknown. It makes sense, therefore, to partition a mesh into a great number of microdomains which then are used to create subdomains. Microdomains are also used for efficient parallelization of domain decomposition methods (for example, Schwarz method) [1]. And microdomain partitions allow to increase the coefficient of mesh data compression in large mesh storage.

Graph partitioning methods implemented in state-of-the-art parallel partitioning tools ParMETIS, Jostle, PT-Scotch and Zoltan are based on multilevel algorithms consisting of three phases: graph coarsening, initial partitioning and uncoarsening with refinement of the partitions. That approach has a shortcoming of making subdomains with longer frontiers or irregular shapes. In particular these methods can form unconnected subdomains. Such worsening of subdomain quality adversely affects the performance of subsequent computations. It may result in a larger number of iterations to achieve convergence of iterative linear system solving methods. Furthermore, unconnected subdomains have longer frontiers where in the subdomain composition algorithm value recalculations are required and this algorithm can't be applied to narrow frontiers [2].

Another shortcoming of present graph partitioning methods is generation of strongly imbalanced partitions. This shortcoming is the most prominent in partitions made by ParMETIS, where number of vertices in some subdomains can be twice as large as in others. The imbalance can cause significant performance problems, especially in exascale computing.

Moreover, partitions into large number of microdomains can't always be obtained by the present graph partitioning methods.

To solve above mentioned problems the program package for parallel large mesh decomposition GridSpiderPar was developed.

## 2. PARALLEL PARTITIONING ALGORITHMS
Two algorithms were implemented in the GridSpiderPar package: a parallel geometric algorithm of mesh partitioning and a parallel incremental algorithm of graph partitioning. The devised parallel algorithms support two main stages of large mesh partitioning. They are a preliminary mesh partitioning among processors and a parallel mesh partitioning of high quality. Both work with unstructured meshes with up to $10^9$ elements.

### 2.1 Parallel incremental algorithm
The parallel incremental algorithm presented here is based on the serial incremental algorithm of graph partitioning [3].The main advantage of this algorithm is creation of principally connected subdomains.

The parallel incremental algorithm consists of the following stages:

- Distribution of vertices among processors according to the results of the geometric decomposition performed by the parallel geometric algorithm, described below.
- Redistribution of small groups of vertices (Figure 1).

- Local partitioning of vertices into subdomains on each processor using the serial incremental graph partitioning algorithm. Figure 2 shows that after local partitioning borders of some subdomains coincide with the processors borders.
- Redistribution of the groups of bad subdomains. Each group of bad subdomains is collected on a single processor.
- Local repartitioning of the groups of bad subdomains using the serial incremental graph partitioning algorithm. In Figure 2 (right) some subdomains spread to the other processors.

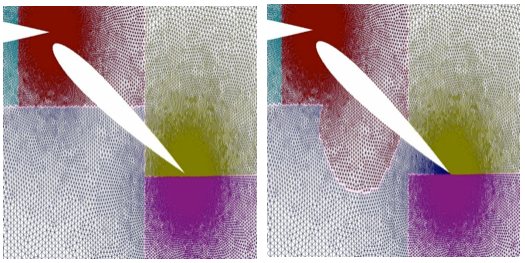


**Figure 1. Geometric distribution of vertices among processors (left) and redistribution of small groups of vertices between processors (right).**

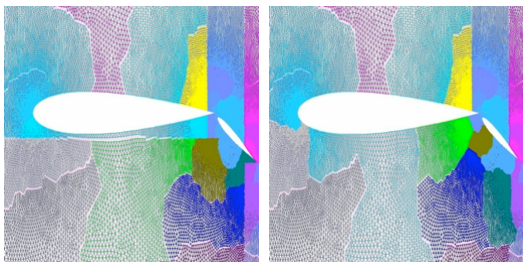


**Figure 2. Local partitioning (left) and bad subdomains groups repartitioning (right).**

At the beginning of the local partitioning of vertices into subdomains for each subdomain one vertex is chosen randomly as a seed. Further decomposition is performed within an iterative process, each step of which does the following:

- Incremental growth of subdomains and diffusion of the border vertices between subdomains (Figures 3, 4).

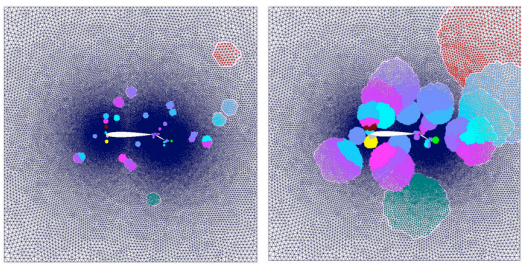


**Figure 3. Incremental growth of subdomains.**

- Local refinement of subdomains using KL / FM local refinement algorithm. Subdomain boundaries become smoother. However, some subdomains are unconnected (Figure 4).

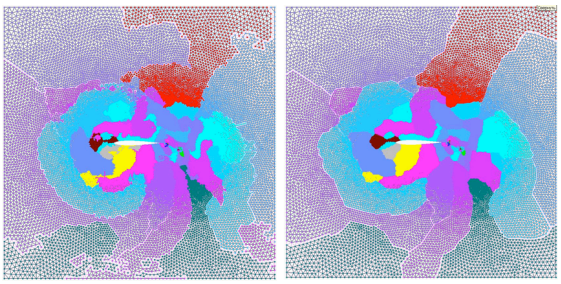


**Figure 4. Diffusion of the border vertices between subdomains (left) and local refinement (right).**

- Subdomains quality control. If the quality matches the specified one, the partition is found. In this case we exit the loop, otherwise - go to the next step.
- Some part of the vertices in bad subdomains is released and switch to the first step is made (Figure 5). Bad subdomains are the subdomains of poor quality and their neighbors. In case of unconnected subdomains only the largest part is preserved in the subdomain.

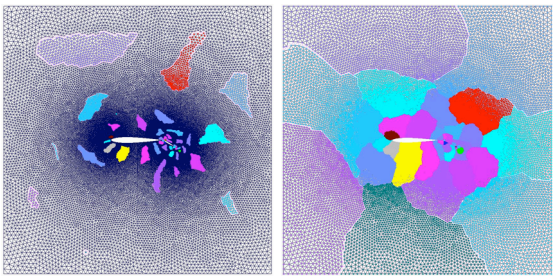


**Figure 5. Releasing of some part of the vertices in bad subdomains (left) and the resulting partition (right).**

Subdomain quality is examined as follows (Figure 6). All vertices in a subdomain which belong to its border or graph border are considered as the first layer of the subdomain. Vertices in this subdomain adjacent to the vertices in the first layer and not belonging to the first layer are considered as the second layer of the subdomain. Other layers are defined by analogy. Layers continuity is examined and the noncontinuous layer with the least number is found. Quality of the subdomain is considered good if the number is greater or equal to a threshold value.

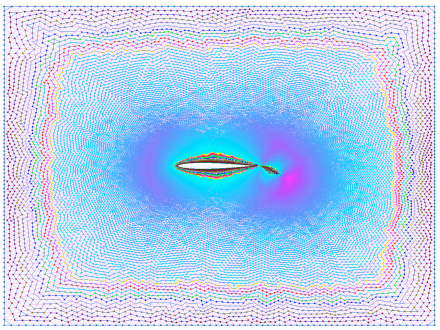


**Figure 6. Subdomain layers.**

## 2.2 Parallel geometric algorithm

The parallel geometric algorithm of mesh partitioning is based on the recursive coordinate bisection method [4, 5]. At each stage of the recursive bisection an area is divided into two parts. The

resulting subareas are split the same way until there is only one subdomain left in each subarea. The algorithm can be described as follows:

- Random initial distribution of vertices among processors (for example, according to the serial numbers of vertices).
- Recursive coordinate bisection of the vertices among processors (Figure 7):
  - The algorithm makes cut of the box comprising the mesh. The cut is made orthogonal to the coordinate axis along which the box is the most elongated.
  - The group of processors is divided into two, then each group splits its vertices block in the same way. To divide the vertices block the parallel sorting [6] according to the selected coordinate axis (and the other axes to split cutting plane) is used.

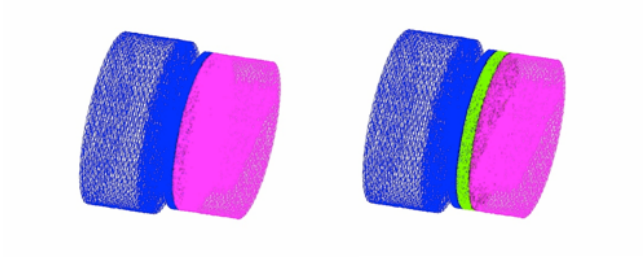


**Figure 7. Geometric partitioning into 7 subdomains on 3 processors. First and second stages of the partitioning - distribution of the vertices among processors.**

- Local recursive coordinate bisection of the vertices among subdomains. Further partitioning into subdomains is performed locally on each processor (Figure 8).

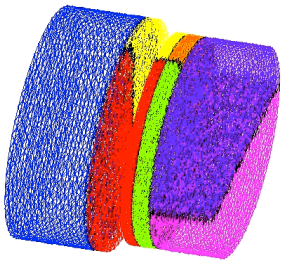


**Figure 8. The result of the geometric partitioning into 7 subdomains on 3 processors.**

The main advantage of this algorithm is that difference in numbers of vertices in resulting subdomains is no more than one vertex.

A similar algorithm is included in Zoltan [5]. The distinction of our algorithm is that it makes cuts of the cutting plane along other coordinate axes to distribute vertices from the cutting plane among subdomains (Figure 9). The algorithm from the Zoltan package distributes these vertices randomly, that increases edge-cut of the resulting partitions.

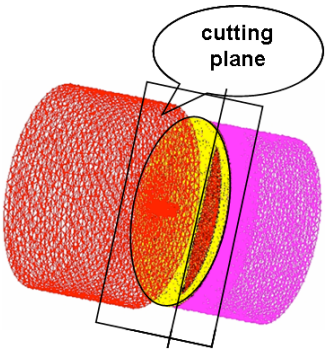



**Figure 9. Cutting plane splitting.**

More detailed description of the devised algorithms can be found in the paper [7].

# 3. EXPERIMENTAL RESULTS

## 3.1 Microdomain and subdomain partitions

We compared different partitions into microdomains, microdomain graph partitions and partitions into subdomains of several meshes with $10^8$ vertices, $10^9$ edges (Figure 10) obtained by means of the partitioning tool GridSpiderPar and the packages ParMETIS, Zoltan and PT-Scotch. Balance of the partitions, edge-cut and number of unconnected subdomains in different partitions were compared.

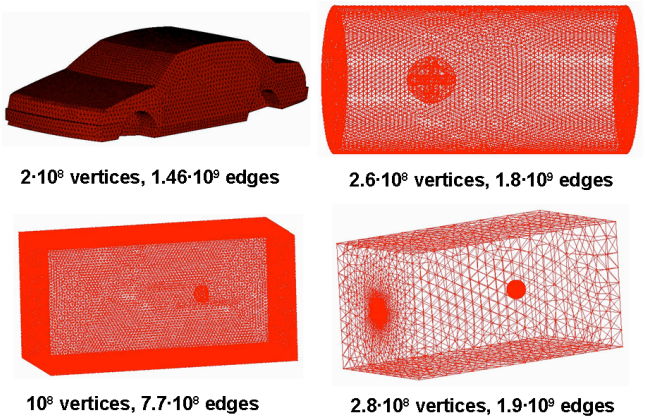



**Figure 10. Tetrahedral meshes.**

The methods in the comparison are:

- IncrDecomp – the incremental algorithm of graph partitioning from the GridSpiderPar package.
- PartKway - multilevel k-way graph partitioning algorithm from the ParMETIS package.
- PartGeomKway - multilevel k-way graph partitioning algorithm from the ParMETIS package, making initial partitioning using a space-filling curve method.
- PT-Scotch – multilevel diffusion algorithm from the PT-Scotch package.
- GeomDecomp – the recursive coordinate bisection method from the GridSpiderPar package.
- RCB – recursive coordinate bisection method from the Zoltan package.

First, all the meshes were partitioned into 25600 microdomains (Table 1 and Table 2). The results in Table 1 show that imbalance in the partitions made by the methods incorporated in the ParMETIS package amounts to 60%, in the partitions made by PT-Scotch – 8%, whereas almost in all the partitions made by IncrDecomp imbalance is less than 1%. And in the partitions made by geometric methods difference in numbers of vertices in resulting microdomains is no more than one vertex, as it was supposed. Hereinafter, by imbalance is meant relative maximum deviation from an average number of vertices in subdomains.

**Table 1. Imbalance in 25600 microdomains, %**

| Methods | Mesh1 | Mesh2 | Mesh3 | Mesh4 |
|---|---|---|---|---|
| **graph partitioning** | | | | |
| **IncrDecomp** | 3,5 | **0,1** | 0,3 | 0,2 |
| PartKway | 53,4 | 59,8 | 58,6 | **64,3** |
| PartGeomKway | 48,7 | 50,4 | **62,4** | 56,5 |
| PT-Scotch | **8,3** | 8,3 | 8,3 | 8,3 |
| **geometric methods** | | | | |
| **GeomDecomp** | **0,01** | 0,01 | 0,02 | 0,01 |
| RCB | **0,01** | 0,01 | 0,02 | 0,01 |

We can see in Table 2 that number of unconnected microdomains in the partitions made by the methods from ParMETIS amounts to 69 in 25600 microdomains as in the partitions made by the geometric methods. For PT-Scotch this number amounts to 7. Usually, only small number of microdomains in partitions made by PT-Scotch is unconnected but PT-Scotch doesn't guarantee the connectivity of microdomains. And almost in all partitions made by IncrDecomp all microdomains are connected.

**Table 2. Number of unconnected microdomains in 25600**

| Methods | Mesh1 | Mesh2 | Mesh3 | Mesh4 |
|---|---|---|---|---|
| **graph partitioning** | | | | |
| **IncrDecomp** | **0** | 0 | 0 | 1 |
| PartKway | **69** | 35 | 37 | 29 |
| PartGeomKway | **67** | 34 | 28 | 37 |
| PT-Scotch | **7** | 0 | 2 | 4 |
| **geometric methods** | | | | |
| **GeomDecomp** | **62** | 38 | 16 | 33 |
| RCB | **64** | 43 | 14 | 44 |

Second, microdomain graphs were constructed for the partitions made by the methods from ParMETIS and GridSpiderPar. Vertex weights in the microdomain graphs correspond to the number of vertices in the microdomains. The microdomain graphs were partitioned into 512 subdomains on 1 processor using methods PartGraphRecursive and PartGraphKway from the package METIS, PartKway from ParMETIS and IncrDecomp from GridSpiderPar. And all the meshes were partitioned directly into 512 subdomains using the methods PartKway and ParGeomKway from ParMETIS, the diffusion method from PT-Scotch and GeomDecomp from GridSpiderPar.

The results of the comparison in Table 3 show that imbalance in the partitions made directly by the methods from ParMETIS amounts to 50%, in the partitions made by PT-Scotch – 5%. Imbalance in the microdomain graph partitions didn't depend on the different imbalance in the microdomains and was about 5%. We assume that it's connected with the small number of microdomains in one subdomain (50) and insufficient sensitivity of the graph partitioning algorithms to vertex weights. The least imbalance was in the microdomain graph partitions made by the methods from METIS.

**Table 3. Imbalance in 512 subdomains, %**

| Methods | Mesh1 | Mesh2 | Mesh3 | Mesh4 |
|---|---|---|---|---|
| **graph partitioning** | | | | |
| PartKway | 12,9 | 20,6 | 17,6 | **28,4** |
| PartGeomKway | 31,1 | 35,7 | 44,2 | **51,4** |
| PT-Scotch | **4,9** | 1,7 | 2,8 | 2,9 |
| **geometric methods** | | | | |
| **GeomDecomp** | **0** | 0 | 0 | 0 |
| **microdomain graph partitioning** | | | | |
| Simple average | 5,3 | **5,4** | 3,7 | 5,1 |

Run time of IncrDecomp is several times greater than run times of the other algorithms. Run time of GeomDecomp is the same as run times of the other geometric algorithms. But since the algorithms were devised for static decomposition and run times of physical tasks are much greater than run times of partitioners the increase of partitioning time isn't so essential.

## 3.2 Gas-dynamic problem simulations run on different partitions

Testing of partitions obtained by the tools GridSpiderPar, ParMETIS, Zoltan and PT-Scotch was performed using simulations of gas-dynamic problems.

First test problem was model simulation of turbulent plasma flow in the ITER divertor (Figure 11). Situated along the bottom of the vacuum vessel, it is designed to extract heat and impurities from the plasma, in effect acting like a giant exhaust system.

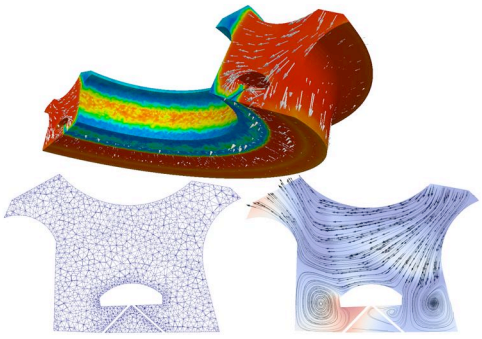


**Figure 11. Plasma flow simulation in the ITER divertor.**

Computational mesh (divertor) contained $2{,}8 \cdot 10^6$ tetrahedrons. Full RMHD system with dissipation and turbulent viscosity was solved using tabulated equations of state, opacities and emissivities. Explicit and implicit schemes were used. The computations were carried out on 256 processors.

Another problem was the near-earth explosion simulation (Figure 12). Two hexahedral computational meshes (boom and boomL), refined in the vicinity of blast area and consisting of parallelepipeds with different aspect ratio, were used: boom with $6{,}1 \cdot 10^7$ parallelepipeds and boomL with $1{,}16 \cdot 10^8$ parallelepipeds. The computations were carried out on 4096 and 10080 processors respectively. Full RMHD system with radiative and conductive heat transfer was solved. Turbulent flows were not taken into account. Explicit and implicit schemes were used.

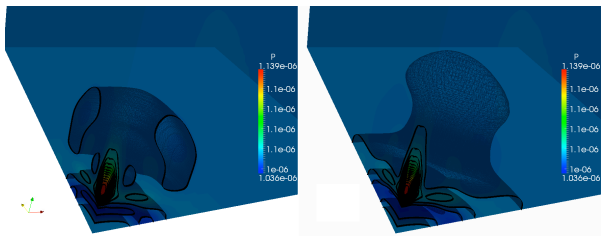


**Figure 12. Pressure spatial variations at t=1000 ms.**

Dual graphs were constructed for each test mesh with number of vertices $2{,}8 \cdot 10^6$ – $1{,}2 \cdot 10^8$ and number of edges $2{,}3 \cdot 10^7$ – $1{,}0 \cdot 10^9$. Dual graph partitions were obtained using the methods from the tools GridSpiderPar, ParMETIS, Zoltan and PT-Scotch.

Two methods were added to the previous list:

- RIB – recursive inertial bisection method from the Zoltan package.
- HSFC – method based on a Hilbert space-filling curve splitting from the Zoltan package.

For readability in Figures 13 – 15 the method IncrDecomp is marked by I, PartKway – by PK, PartGeomKway – by PGK, PT-Scotch – by PTScotch and GeomDecomp – by G. The methods IncrDecomp and GeomDecomp are highlighted by differing colours, the graph partitioning algorithms and the geometric algorithms are separated.

Figures 13 and 14 represent imbalance in the subdomain partitions in the sense of lack of vertices in subdomains and overflow of vertices in subdomains. Lack of vertices in subdomains in the partitions obtained by the methods from ParMETIS amounts to 80%, overflow of vertices – to 5%. Imbalance in the partitions made by PT-Scotch in the two cases is about 5% and in the partitions made by IncrDecomp imbalance is less the 0,1%.

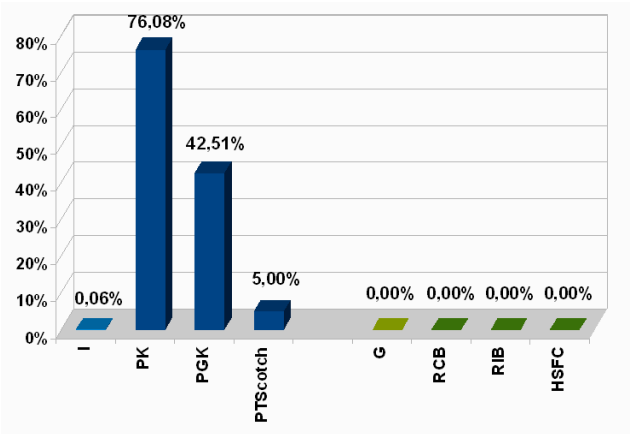



**Figure 13. Imbalance in subdomains: lack of vertices (boom).**

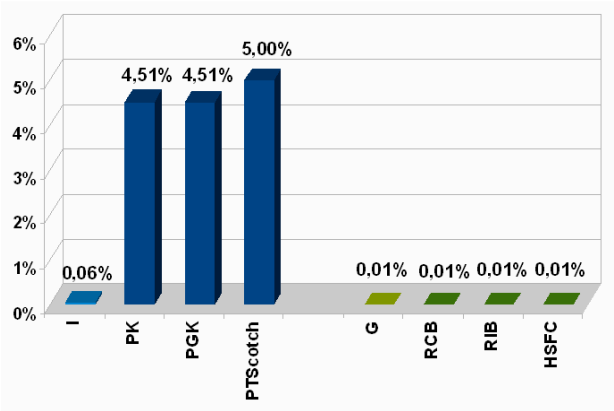



**Figure 14. Imbalance in subdomains: overflow of vertices (boom).**

The least edge-cut was achieved in the partitions obtained by the ParMETIS methods or made by IncrDecomp.

We used each test mesh partitions for the set of numerical experiments using the MARPLE3D research code designed in KIAM RAS [8] for multiphysics simulations in the field of radiative plasma dynamics. Computational performance of the simulations with MARPLE3D code run on different partitions was compared. All problems were run on the different partitions for the same time and numbers of time steps done within that time were measured.

Comparison results (Figure 15) show that IncrDecomp method is ahead of the other graph partitioning methods, and GeomDecomp method slightly outruns the other geometric methods. Number of time steps on partitions obtained by the graph partitioning methods is greater than on partitions obtained by the geometric methods since geometric methods don't take into account interprocessor communications.

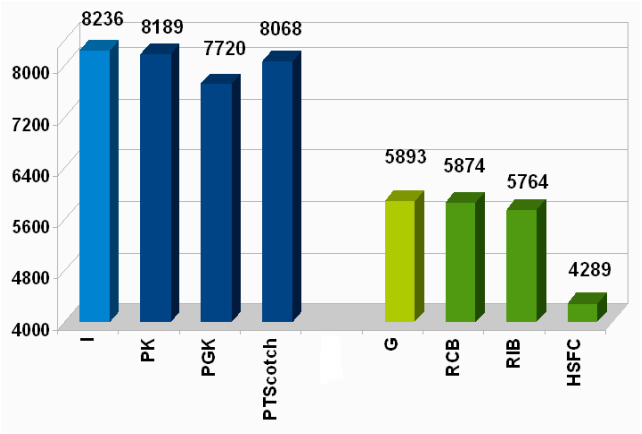



**Figure 15. Number of time steps done within 1 hour (divertor).**

## 3.3 Testing of microdomain graph partitions on near-earth explosion simulation problem

The dual graph boomL with $1{,}2 \cdot 10^8$ vertices and $1{,}0 \cdot 10^9$ edges was partitioned into different number of microdomains (from 24576 to 196608) and directly into 3072 subdomains by the devised algorithm IncrDecomp from the GridSpiderPar package. Microdomain graphs were constructed with vertex weights corresponding to the number of vertices in the microdomains. The

microdomain graphs were partitioned into 3072 subdomains by the method IncrDecomp.

The abovementioned near-earth explosion simulation problem was run on the different partitions for the same time (5 hours) and numbers of time steps done within that time were measured.

**Table 4. Testing of microdomain graph partitions on the near-earth explosion simulation problem**

| **Mesh info** | **name: BoomL 116 214 272 hexahedrons** | | | | |
|---|---|---|---|---|---|
| **Micro-domains** | **3072** | 24576 | 49152 | 98304 | **196608** |
| **Micro-domains in subdomain** | **1** | 8 | 16 | 32 | **64** |
| **Imbalance, %** | **9,1** | **62,5** | 37,5 | 18,7 | **7,9** |
| **Cut edges** | **5,31· $10^7$** | 6,46· $10^7$ | 6,65· $10^7$ | 6,88· $10^7$ | **6,82· $10^7$** |
| **Neigh-bouring subdomains (max.)** | **28** | 25 | 25 | 23 | **21** |
| **Time steps** | **1107** | 833 | 880 | 949 | **999** |

We can see in Table 4 that the partitions contain from 1 (direct partitioning into 3072 subdomains) to 64 microdomains in one subdomain. Imbalance in the microdomain graph partitions decreases with increase of the number of microdomains in one subdomain and for 64 microdomains in one subdomain the imbalance is less than in the direct partition into 3072 subdomains. Maximal number of neighbouring subdomains also decreases with increase of the number of microdomains in one subdomain. Maximal number of neighbouring subdomains influences on the number of interprocessor communications. Number of cut edges increases, because the microdomain graphs didn't take into account the number of cut edges between the microdomains. Number of time steps done within the same time increases with increase of the number of microdomains in one subdomain and for 64 microdomains in one subdomain the number of time steps is near to the number of time steps done on the direct partition into 3072 subdomains.

So we can say that with the sufficient quantity of microdomains in one subdomain microdomain graph partitions are not worse than direct graph partitions that is verified by small deceleration of the gas-dynamic problem simulation. In addition, microdomain graph partitioning takes much less time than direct graph partitioning.

# 4. CONCLUSION

The program package for parallel large mesh decomposition GridSpiderPar was developed. Two algorithms were implemented in the GridSpiderPar package: a parallel geometric algorithm of mesh partitioning and a parallel incremental algorithm of graph partitioning. The devised parallel algorithms support two main stages of large mesh partitioning: preliminary mesh partitioning among processors and parallel mesh partitioning of high quality. Both work with unstructured meshes with up to $10^9$ elements. The main advantage of the second algorithm is creation of principally connected subdomains.

We compared different partitions into microdomains, microdomain graph partitions and partitions into subdomains of several meshes ($10^8$ vertices, $10^9$ elements) obtained by means of the partitioning tool GridSpiderPar and the packages ParMETIS, Zoltan and PT-Scotch. Balance of the partitions, edge-cut and number of unconnected subdomains in different partitions were compared as well as the computational performance of gas-dynamic problem simulations run on different partitions. The obtained results demonstrate advantages of the devised algorithms.

The work was supported by RFBR grants 13-01-12073-ofi_m, 14-01-00663-a, 14-07-00712-a and 15-07-04213-a. The computations were carried out on supercomputers MVS-100K (JSCC RAS), Lomonosov (RCC MSU) and Helios (IFERC).

# 5. REFERENCES

[1] Barry Smith, Petter Bjorstad and William Gropp. 1996. Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations. *Cambridge University Press.* 225 p.

[2] Ilyushin A.I., Kolmakov A.A., Menshov I.S. 2012. Constructing parallel numerical model by means of the composition of computational objects. *Mathematical Models and Computer Simulations*. Vol. 4. Issue 1. 118-128 (in Russian).

[3] M.V. Iakobovski. Incremental algorithm of the graph decomposition. 2005. *Bulletin of Nizhniy Novgorod University, "Mathematical modelling and optimal control".* Issue 1(28). 243–250 (in Russian).

[4] R. Preis, R. Diekmann. PARTY: A Software Library for Graph Partitioning. 1997. *Advances in Computational Mechanics with Parallel and Distributed Processing.* CIVIL-COMP PRESS. 63–71.

[5] E. Boman, K. Devine, U. Catalyurek, D. Bozdag, B. Hendrickson, W. F. Mitchell, J. Teresco. Zoltan Parallel Partitioning, Load Balancing and Data-Management Services. Developers Guide, Version 3.3. Copyright 2000–2010. *Sandia National Laboratories.*

[6] M. V. Iakobovski. Parallel algorithms for sorting of large data volumes on distributed memory systems. 2004. *The book of scientific articles (edited by Lyudmila A.Uvarova), "Mathematical modeling: modern methods and applications".* 153–163.

[7] Evdokia Golovchenko, Elizaveta Dorofeeva, Irina Gasilova and Alexey Boldarev. Numerical Experiments with New Algorithms for Parallel Decomposition of Large Computational Meshes. 2014. *Parallel Computing: Accelerating Computational Science and Engineering (CSE). Advances in Parallel Computing. IOS Press*. Vol. 25. 441-450.

[8] V. Gasilov, A. Boldarev, S. Dyachenko, O. Olkhovskaya and others. Towards an Application of High-Performance Computer Systems to 3D Simulations of High Energy Density Plasmas in Z-Pinches. 2012. *IOS Press: Advances in Parallel Computing.* Vol. 22. 235–242.

# Improving Performance Portability and Exascale Software Productivity with the $\nabla$ Numerical Programming Language

Jean-Sylvain Camier
CEA, DAM, DIF, F-91297 Arpajon, France.
Jean-Sylvain.Camier@cea.fr

## 1. ABSTRACT

Addressing the major challenges of software productivity and performance portability becomes necessary to take advantage of emerging extreme-scale computing architectures. As software development costs will continuously increase to deal with exascale hardware issues, higher-level programming abstractions will facilitate the path to go. There is a growing demand for new programming environments in order to improve scientific productivity, to ease design and implementation, and to optimize large production codes.

We introduce the numerical analysis specific language Nabla ($\nabla$) which improves applied mathematicians productivity, and enables new algorithmic developments for the construction of hierarchical composable high-performance scientific applications. One of the key concept is the introduction of the hierarchical logical time within the high-performance computing scientific community. It represents an innovation that addresses major exascale challenges. This new dimension to parallelism is explicitly expressed to go beyond the classical single-program multiple-data or bulk-synchronous parallel programming models. Control and data concurrencies are combined consistently to achieve statically analyzable transformations and efficient code generation. Shifting the complexity to the compiler offers an ease of programming and a more intuitive approach, while reaching the ability to target new hardware and leading to performance portability.

In this paper, we present the three main parts of the $\nabla$ toolchain: the frontend raises the level of abstraction with its grammar; the backends hold the effective generation stages, and the middle-end provides agile software engineering practices transparently to the application developer, such as: instrumentation (performance analysis, V&V, debugging at scale), data or resource optimization techniques (layout, locality, prefetching, caches awareness, vectorization, loop fusion) and the management of the hierarchical logical time, which produces the graphs of all parallel tasks. The refactoring of existing legacy scientific applications is also possible by the incremental compositional approach of the method.

## 2. INTRODUCTION

Nabla ($\nabla$) is an open-source [4] Domain Specific Language (DSL) introduced in [6] whose purpose is to translate numerical analysis algorithmic sources in order to generate optimized code for different runtimes and architectures. The objectives and the associated roadmap have been motivated since the beginning of the project with the goal to provide a programming model that would allow:

- **Performances**. The computer scientist should be able to instantiate efficiently the right programming model for different software and hardware stacks.
- **Portability**. The language should provide portable scientific applications across existing and fore-coming architectures.
- **Programmability**. The description of a numerical scheme should be simplified and attractive enough for tomorrow's software engineers.
- **Interoperability**. The source-to-source process should allow interaction and modularity with existing legacy codes.

As computer scientists are continuously asked for optimizations, flexibility is now mandatory to be able to look for better concurrency, vectorization and data-access efficiency, even at the end of long development processes. The self-evident truth is that it is too late for these optimizations to be significantly effective with standard approaches. The $\nabla$ language constitutes a proposition for numerical mesh-based operations, designed to help applications to reach these listed goals. It raises the level of abstraction, following a bottom-up compositional approach that provides a methodology to co-design between applications and underlying software layers for existing middleware or heterogeneous execution models. It introduces an alternative way, to go further than the bulk-synchronous way of programming, by introducing logical time partial-ordering and bringing an additional dimension of parallelism to the high-performance computing community.

The remainder of the paper is organized as follow. Section 3 gives an overview of the $\nabla$ domain specific language. Section 4 introduces the hierarchical logical time concept. The different elements of the toolchain are described in section 5. Finally, section 6 provides some evaluations and experimental results for Livermore's Unstructured Lagrange Explicit Shock Hydrodynamics (Lulesh) [15] proxy application on different target architectures.

## 2.1 Related Work

Because application codes have become so large and the exploration of new concepts too difficult, domain specific languages are becoming even more attractive by offering the possibility to explore a range of optimizations.

**Loci**[18] is an automatic parallelizing framework that has been supporting the development of production simulation codes for more than twenty years. The framework provides a way to describe the computational kernels using a relational rule-based programming model. The data-flow is extracted, transformations are applied to optimize the scheduling of the computations; data locality can also be enhanced, improving the overall scalability of a Loci application.

**SpatialOps** which provides the **Nebo**[11] EDSL is an embedded C++ domain specific language for platform-agnostic PDE solvers [21]. It provides an expressive syntax, allowing application programmers to focus primarily on physics model formulation rather than on details of discretization and hardware. Nebo facilitates portable operations on structured mesh calculations and is currently used in a number of multiphysics applications including multiphase, turbulent reacting flows.

**Liszt**[10] is a domain-specific language for solving PDE on meshes for a variety of platforms, using efficient different parallel models: MPI, pthreads and CUDA. The design of computational kernels is facilitated: the data dependencies being taken care of by the compiler.

**Terra**[9] is a low-level language, designed for high performance computing, interoperable with Lua [20]. It is a statically typed, compiled language with manual memory management and a shared lexical environment.

**Scout**[19] is a compiled domain-specific language, targeting CUDA, OpenCL and the Legion[5] runtime. It does not provides a source-to-source approach, but supports mesh-based applications, in-situ visualization and task parallelism. It also includes a domain-aware debugging tool.

**RAJA**[14, 16] is a thin abstraction layer consisting of a parallel-loop construct for managing work and an IndexSet construct for managing data. The composition of these components allows architecture specific details of programming models, vendor compilers, and runtime implementations to be encapsulated at a single code site, isolating software applications from portability-related disruption, while enhancing readability and maintainability.

Thanks to the source-to-source approach and its exclusive logical time model, ∇ is able to target some of the above listed languages. It allows developers to think in terms of more parallelism, letting the compilation process tools perform appropriate optimizations.

**Listing 1: Libraries and Options Declaration in ∇**

```
with ℵ, slurm;

options{
  Real option_dtfixed          = -1.0e-7;
  Real option_δt_initial       = 1.0e-7;
  Real option_δt_courant       = 1.0e+20;
  Real option_δt_hydro         = 1.0e+20;
  Real option_ini_energy       = 3.948746e+7;
  Real option_stoptime         = 1.0e-2;
  Bool option_rdq              = false;
  Real option_rdq_α            = 0.3;
  Integer option_max_iterations = 8;
  Bool option_only_one_iteration = false;
};
```

# 3. OVERVIEW OF THE NABLA DSL

This section introduces the ∇ language, which allows the conception of multi-physics applications, according to a logical time-triggered approach. Nabla is a domain specific language which embeds the C language. It follows a source-to-source approach: from ∇ source files to C, C++ or CUDA output ones. The method is based on different concepts: no central *main* function, a multi-tasks based parallelism model and a hierarchical logical time-triggered scheduling.

## 3.1 Lexical & Grammatical Elements

To develop a ∇ application, several source files must be created containing *standard functions* and specific *for-loop* function, called *jobs*. These files are provided to the compiler and will be merged to compose the application. The compilation stages operate the transformations and return *source files*, containing the whole code and the required data. An additional stage of compilation with standard tools must therefore be done on this output.

**Listing 2: Variables Declaration in ∇**

```
nodes{
  Real3 ∂tx;
  Real3 ∂t2x;
  Real3 nForce;
  Real nMass;
};
cells{
  Real p;
  Real3 ϵ;
  Real3 cForce[nodes];
  Real delv_xi;
};
```

To be able to produce an application from ∇ source files, a first explicit declaration part is required. Language *libraries* have to be listed, *options* and the data fields -or *variables*- needed by the application have to be declared. Libraries are introduced by the `with` token: additional keywords will then be accessible, as well as some specific programming interfaces. For example, the `aleph` (ℵ) library provides the `matrix` keyword, as well as standard algebra functions to fill linear systems and solve them. The *options* keyword allows developers to provide different optional inputs to the application, with their default values, that will be then accessible from the command line or within some data input files. Listing 1 provides an example of libraries and options declaration in ∇.

Application data fields must be declared by the developer: these *variables* live on *items*, which are some mesh-based numerical elements: the *cells*, the *nodes*, the *faces* or the *particles*. Listing 2 shows two declarations of variables living on *nodes* and *cells*. Velocity ($\partial tx$), acceleration ($\partial t2x$) and force vector (`nForce`), as well as the nodal mass (`nMass`) for *nodes*. Pressure (`p`), diagonal terms of deviatoric strain ($\epsilon$) and some velocity gradient (`delv_xi`) on *cells*.

Different data types are also available, such as `Integer`, `Bool`, `Real` or three-dimension vector types `Real3`, allowing the insertion of specific directives during the second compilation stage. Unicode letter and some additional mathematical operators are also provided: the Listing 3 gives some operators that are actually supported and particularly used in reduction statements, assignment, conditional, primary and multiplicative expressions.

**Listing 3: Additional ∇ Expressions**

```
<?= >?= ?= @ ∀ ℵ ∧ ∨ ∞ ² ³ √ ∛ ½ ⅓ ¼ ⅛ ⋆ · × ⊗
```

## 3.2 Functions and Jobs Declaration

In order to schedule standard *functions* and *jobs*, the language provides some new syntactic elements, allowing the design of logical time-triggered multi-tasking applications. Data-parallelism is implicitly expressed by the declaration of *jobs*, which are functions with additional attributes. The first attribute is the *item* on which the function is going to iterate: it can be a for-loop on *cells* or *nodes* for example. *Input* and *output* variables the *job* will work with are also to be provided. Finally an attribute telling *when* the *job* shall be triggered can also be given: this introduces the logical-time triggered model of parallelism that is presented in section 4.

**Listing 4: ∇ Job Declaration, a *for-loop* on *nodes***

```
nodes void iniNodalMass(void)
  in (cell calc_volume)
  out (node nodalMass) @ -6.9{
  nodalMass=0.0;
  ∀ cell nodalMass += calc_volume/8.0;
}
```

Listing 4 is a *for-loop*, iterating on the *nodes*, set by the developer to be triggered at the logical time '`-6.9`'. This job uses in its body the '∀' token, which starts another for-loop, for each *cell* the current node is connected to.

**Listing 5: ∇ Job Declaration, another on *cells***

```
cells void temporalComputeStdFluxesSum(void)
     in (cell reconstructed_u, node u,
         cell reconstructed_p,
         cell CQs, cell AQs)
    out (cell momentum_fluxes_Σ,
         cell total_energy_fluxes_Σ) @ 16.0 {
    foreach node{
      const Real3 Δu = reconstructed_u-u;
      Real3 FQs = AQs ⋆ Δu;
      FQs += reconstructed_p*CQs;
      momentum_fluxes_Σ -= FQs;
      total_energy_fluxes_Σ -= FQs·u;
    }
}
```

The job in Listing 5 illustrates an explicit scheme [8]. It is a for-loop on *cells*, triggered at the logical time '`+16.0`', and with an inner connectivity loop on each of the cell's nodes. Two mathematical operators are used in this job: the vector dot product '·' and the matrix vector product '⋆'.

**Listing 6: ∇ Job Declaration Statement**

```
cells void calcEnergyForElems1(void)
  in (cell e_old, cell delvc,
      cell p_old, cell q_old, cell work)
  inout (cell e_new) @ 7.1 {
  e_new = e_old-½*delvc*(p_old+q_old)+½*work;
  e_new = (e_new<option_emin)?option_emin;
}
```

Listing 6 comes from the proxy application Lulesh [17], during the equation of state phase. It is a *cell* job, working on some of its variables and set to be launched at logical time '`7.1`'. It shows the use of the '`?`' binary operator, which changes the ternary standard C '`?:`', by allowing to omit the '*else*' ('`:`') statements, meaning here '*else unchanged*'.

**Listing 7: ∇ Implicit Job: filling a *matrix***

```
nodes void δNodes(void)
  in (face δ,
      node θ,
      node node_area,
      node node_is_an_edge,
      face Cosθ, face sdivs) @ 3.4 {
  Real δn,Σδ=0.0;
  if (node_is_an_edge) continue;
  foreach face {
    Node other=(node[0]==this)?node[1]:node[0];
    δn=δ/node_area;
    Σδ+=1.0/(Cosθ*sdivs);
    ℵ matrix addValue(θ,this,θ,other,-δn);
  }
  Σδ*=δt/node_area;
  ℵ matrix addValue(θ,this,θ,this,1.0+Σδ);
}
```

The job in Listing 7 is a for-loop on each node of the domain. It is set to be triggered at '`+3.4`'. The aleph (ℵ) library token and its programming interface is used to fill the matrix. The degrees of freedom are deduced by the use of pairs of the form: *(item,variable)*. Two degrees of freedom are used here: *(θ,this)* and *(θ,other)*.

More simple jobs can be expressed, like the one presented in Listing 8. It is one of the two reductions required at the end of each iteration in the compute loop of Lulesh. *δt_hydro* is the global variable and the *δt_cell_hydro* is the one attached to each cell. It is here a `minimum` reduction over all the *cells*.

**Listing 8: ∇ Reduction Statement**

```
∀ cells δt_hydro <?= δt_cell_hydro @ 12.22;
```

The different '`@`' attributes are then gathered and combined hierarchically by the toolchain, in order to create the logical time triggered execution graph, used for the scheduling of all *functions* and *jobs* of the application. Next section introduces the composition of such logical time statements.

# 4. HIERARCHICAL LOGICAL TIME

The introduction of the hierarchical logical time within the high-performance computing scientific community represents an innovation that addresses the major exascale challenges. This new dimension to parallelism is explicitly expressed to go beyond the classical single-program-multiple-data or bulk-synchronous-parallel programming models. The task-based parallelism of the ∇ jobs is explicitly declared via logical-timestamps attributes: each function or job can be tagged with an additional '`@`' statement. The two types of concurrency models are used: the control-driven one comes from these logical-timestamps, the data-driven model is deduced from the *in*, *out* or *inout* attributes of the variables declaration. These control and data concurrency models are then combined consistently to achieve statically analyzable transformations and efficient code generation.

By gathering all the '`@`' statements, the ∇ compiler constructs the set of partially ordered jobs and functions. By convention, the negative logical timestamps represent the initialization phase, while the positive ones compose the compute loop. You end up with an execution graph for a single ∇ component.

**Table 1: ∇ Logical Time Diagrams: [a] is the totally-ordered time-diagram from a typical mini-application ported to ∇ with consecutive *for-loops*; [b] is the diagram of a better partially-ordered numerical scheme. Colors stand for the *job items*.**

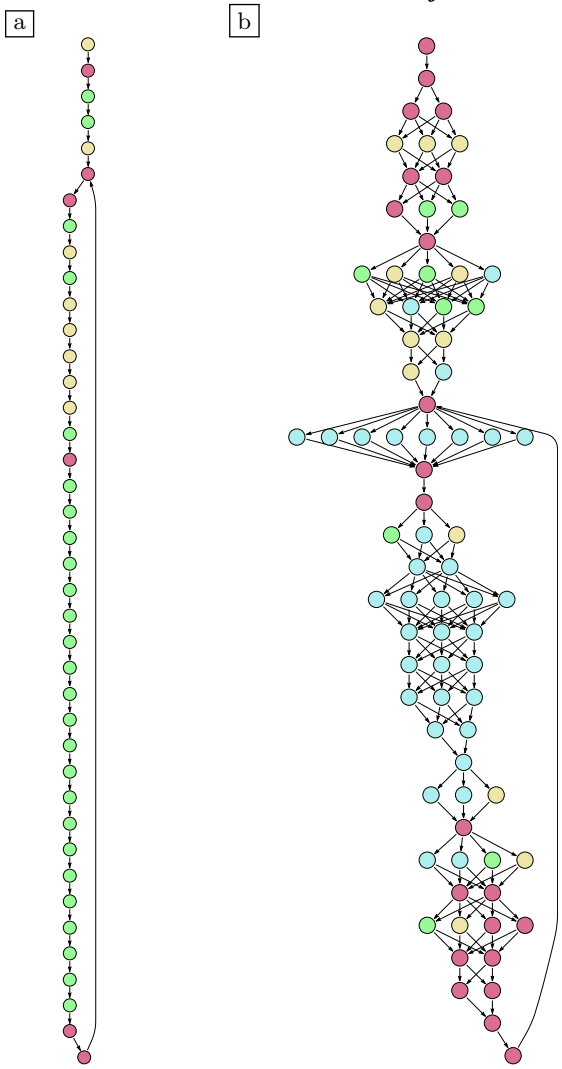



Table 1 presents two kinds of execution graphs: the first one (a) is taken from a typical proxy application, as the second one (b) comes from a new implicit numerical scheme [7], designed and written in ∇ entirely. No additional parallelism lies in the first totally-ordered diagram, whereas the second one exposes a new dimension that can be exploited for scheduling.
Each ∇ component can be written and tested individually. A nested composition of such logical-timed components becomes a multi-physic application. Such an application still consists in a top initialization phase and a global computational loop, where different levels of ∇ components can be instantiated hierarchically, each of them running there own initialization/compute/until-exit parts. This composition is actually done by the compiler with command line options; the need of frontend tools will rapidly be crucial as applications grow bigger.

# 5. THE NABLA TOOLCHAIN

The ∇ toolchain is composed of three main parts, illustrated in Figure 1. The frontend is a Flex [1] and Bison [2] parser that reads a set of ∇ input files. The middle-end provides a collection of software engineering practices, transparently to the application developer. Instrumentation, performance analysis, validation and verification steps are inserted during this process. Data layout optimizations can also take place during this phase: locality, prefetching, caches awareness and loop fusion techniques are in development and will be integrated in the middle-end.

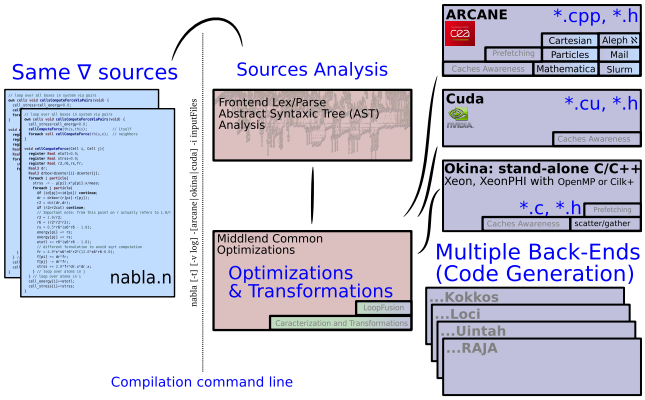



**Figure 1: The three parts of the ∇ Toolchain: the Sources Analysis (Frontend), the Optimizations & Transformations (Middle-end) and the Generation Stages (Backends).**

The backends hold the effective generation stages for different targets or architectures:

- **ARCANE** [13]. It is a numerical code framework for high-performance computing. It provides multiple strategies of parallelism: MPI, threads, MPI+threads and the Multi-Processor Computing framework (MPC) [24].
- **CUDA** [25, 22]. CUDA is a programming model to target NVIDIA's graphics processing unit (GPU). The ∇ compiler generates homogenous source files: all the numerical kernels run exclusively on the GPU. Initial speedups have been achieved with only minimal initial investment of time for this backend: further optimization opportunities are to be identified and deployed.
- **OKINA**. This standalone backend comes with the ∇ toolchain. C/C++ source files are generated: the code is fully-vectorized by the use of *intrinsics* classes. The choice of underlying data structures is possible for different hardwares: specific layouts with their associated *prefetch* instructions or the Kokkos [12] abstraction layer, are two examples. For now, only OpenMP 4.0 [23] and Cilk+ [3] can be used as underlying parallel execution runtimes.

As a demonstration of the potential and the efficiency of this approach, the next section presents the Lulesh benchmark, implemented in ∇. The performances are evaluated for a variety of hardware architectures.

## 6. ∇-LULESH EXPERIMENTAL RESULTS

The Lulesh benchmark solves one octant of the spherical Sedov blast wave problem using Lagrangian hydrodynamics for a single material. Equations are solved using a staggered mesh approximation. Thermodynamic variables are approximated as piece-wise constant functions within each element and kinematic variables are defined at the element nodes. For each figure, cells-updates-per-$\mu s$ for different kind of runs are presented. On Figures 2 and 3, yellow bars show the reference performances of the downloadable version, the violet ones are obtained with the hand-optimized OpenMP version and finally, blue ones are the performances reached from the ∇-Lulesh source files and the OKINA backend.

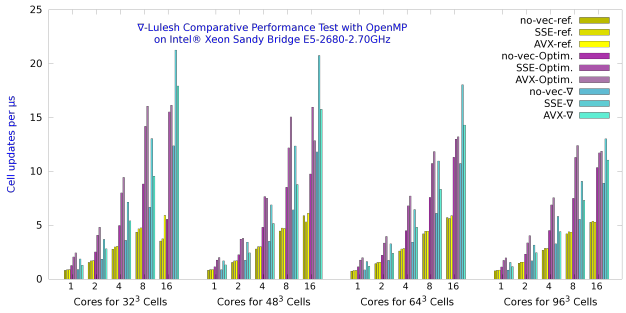



**Figure 2: Reference (ref.), Optimized (Optim.) and ∇ Lulesh Performances Tests on Intel Xeon-SNB with the C/C++ Standalone OKINA+OpenMP Backend and no-vec., SSE or AVX Intrinsics. Higher is better.**

Figure 2 shows the results obtained on Xeon Sandy Bridge E5-2680@2.7GHz architectures. Different kinds of vectorization are represented for each run: no-vectorization, only SSE and full AVX. The ∇-Lulesh version presents a similar level of performances as the optimised one, despite the `scatter` and `gather` instructions generated by the backend which are emulated by software on this architecture.

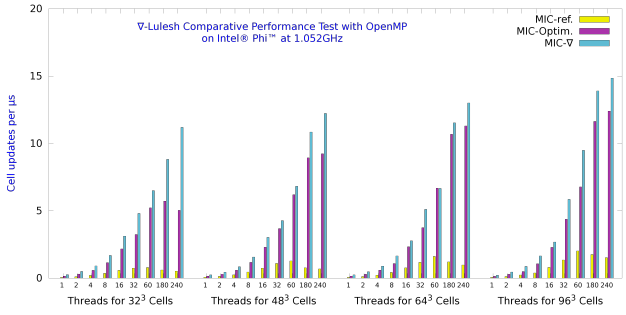



**Figure 3: Reference (ref.), Optimized (Optim.) and ∇ Lulesh Performances Tests on Intel Xeon PHI with the C/C++ Standalone OKINA+OpenMP Backend and AVX512 Intrinsics**

Figure 3 shows the performances obtained on Intel Xeon-PHI processors with the AVX512 vectorization intrinsics. In this example, the `scatter` and `gather` operation codes, supported by the hardware, are not emulated anymore and a higher level of performances is reached, better than the hand-tuned version.

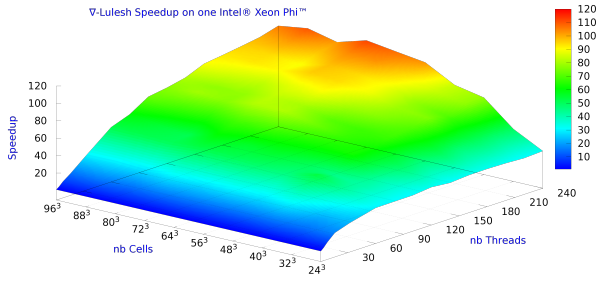



**Figure 4: ∇-Lulesh Speedups on Intel Xeon PHI: A speedup of more than one hundred is reached for a mesh of 125000 Elements with 240 Threads**

Figure 4 shows the speedup with the OKINA+OpenMP backend that is reached for different runs. The number of cells are presented on the X-axis, the number of threads used on the Y-axis. The 3D-surface renders in hot colors where the application starts taking advantage of hyper-threading on this architecture. A speedup of more than a hundred is reached for a mesh of more than one hundred thousands of cells on more than two hundreds of threads.

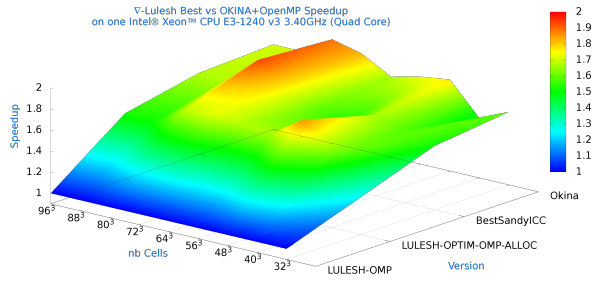



**Figure 5: ∇-Lulesh Speedups on a quad core Intel Xeon Haswell: the OKINA+OpenMP backend vs other OpenMP versions**

Finally, Figure 5 presents the performance results on a single Intel Xeon Haswell E3-1240v3 at 3.40GHz of different OpenMP versions of Lulesh. The `LULESH-OMP` one can be downloaded and is the reference in this test. The `LULESH-OPTIM-OMP-ALLOC` is an optimized version and the `BestSandy` is the fastest that can be found on the web site: it stands for the best candidate with OpenMP. The last one is the ∇-Lulesh version with OKINA. The 3D-surface renders again in hot colors the best speedups that are reached for different mesh sizes and for the different versions. The results of the OKINA backend are as good as the `BestSandyICC` ones: the back of the surface stays on the same level of speedup.

These results emphasize the opportunity for domain-specific languages. Doing so opens up a potential path forward for enhanced expressivity and performance. ∇ achieves both portably, while maintaining a consistent programming style and offering a solution to the productivity issues.

# 7. DISCUSSION AND FUTURE WORK

The numerical-analysis specific language Nabla (∇) provides a productive development way for exascale HPC technologies, flexible enough to be competitive in terms of performances. The refactoring of existing legacy scientific applications is also possible by the incremental compositional approach of the method. Raising the loop-level of abstractions allows the framework to be prepared to address growing concerns of future systems. There is no need to choose today the best programming model for tomorrow's architectures: ∇ does not require to code multiple versions of kernels for different models.

Nabla's source-to-source approach and its exclusive logical time model will facilitate future development work, focusing on new backends: other programming models, abstraction layers or numerical frameworks are already planned.

The generation stages will be improved to incorporate and exploit algorithmic or low-level resiliency methods by coordinating co-designed techniques between the software stack and the underlying runtime and operating system.

∇ is open-source, ruled by the French CeCILL license, which is a free software license, explicitly compatible with the GNU GPL.

# 8. REFERENCES

[1] Flex: The Fast Lexical Analyzer, flex.sourceforge.net.

[2] GNU Bison, www.gnu.org/software/bison.

[3] Intel Cilk Plus, www.cilkplus.org.

[4] ∇-Nabla, www.nabla-lang.org.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[6] J. Camier. ∇-Nabla: A Numerical-Analysis Specific Language for Exascale Scientific Applications. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2014.

[7] J.-S. Camier and F. Hermeline. A Monotone Non-Linear Finite Volume Method for Approximating Diffusion Operators on General Meshes, To Appear.

[8] G. Carré, S. Del Pino, B. Després, and E. Labourasse. A cell-centered lagrangian hydrodynamics scheme on general unstructured meshes in arbitrary dimension. *J. Comput. Phys.*, 228(14):5160–5183, Aug. 2009.

[9] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A Multi-stage Language for High-performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM.

[10] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[11] C. Earl, M. Might, A. Bagusetty, and J. C. Sutherland. Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations.

[12] H. C. Edwards and D. Sunderland. Kokkos Array Performance-portable Manycore Programming Model. Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM), 2012.

[13] G. Grospellier and B. Lelandais. The Arcane Development Framework. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 4:1–4:11, New York, NY, USA, 2009. ACM.

[14] R. Hornung and J. Keasler. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, 2014.

[15] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, December 2012.

[16] J. Keasler and R. Hornung. Managing Portability for ASC Applications. In *SIAM Conference on Computational Science and Engineering*, 2015.

[17] Lawrence Livermore National Laboratory. Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254.

[18] E. A. Luke. Loci: A Deductive Framework for Graph-Based Algorithms. In *Computing in Object-Oriented Parallel Environments, Third International Symposium, ISCOPE 99, San Francisco, California, USA, December 8-10, 1999, Proceedings*, pages 142–153, 1999.

[19] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis, and J. Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '14, pages 1–10, Piscataway, NJ, USA, 2014. IEEE Press.

[20] F. P. Miller, A. F. Vandome, and J. McBrewster. *Lua (Programming Language)*. Alpha Press, 2009.

[21] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland. Graph-based software design for managing complexity and enabling concurrency in multiphysics pde software. *ACM Trans. Math. Softw.*, Nov. 2012.

[22] NVIDIA. CUDA C Programming Guide, 2014.

[23] OpenMP Application Program Interface, version 4.0. OpenMP Architecture Review Board, 2013.

[24] M. Pérache, H. Jourdren, and R. Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, Berlin, Heidelberg, 2008.

[25] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

# A highly scalable Met Office NERC Cloud model

Nick Brown
EPCC, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh
nick.brown@ed.ac.uk

Michele Weiland
EPCC, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh

Adrian Hill
UK Met Office, FitzRoy Road, Exeter, Devon

Ben Shipway
UK Met Office, FitzRoy Road, Exeter, Devon

Chris Maynard
UK Met Office, FitzRoy Road, Exeter, Devon

Thomas Allen
UK Met Office, FitzRoy Road, Exeter, Devon

Mike Rezny
UK Met Office, FitzRoy Road, Exeter, Devon

## ABSTRACT

Large Eddy Simulation is a critical modelling tool for scientists investigating atmospheric flows, turbulence and cloud microphysics. Within the UK, the principal LES model used by the atmospheric research community is the Met Office Large Eddy Model (LEM). The LEM was originally developed in the late 1980s using computational techniques and assumptions of the time, which means that the it does not scale beyond 512 cores. In this paper we present the Met Office NERC Cloud model, MONC, which is a re-write of the existing LEM. We discuss the software engineering and architectural decisions made in order to develop a flexible, extensible model which the community can easily customise for their own needs. The scalability of MONC is evaluated, along with numerous additional customisations made to further improve performance at large core counts. The result of this work is a model which delivers to the community significant new scientific modelling capability that takes advantage of the current and future generation HPC machines.

## Keywords

MONC, LEM, Large Eddy Simulation, Met Office

## 1. INTRODUCTION

Large Eddy Simulation is a computational fluid dynamics technique used to efficiently simulate and study turbulent flows. In atmospheric science, LES are often coupled to cloud microphysics and radiative transfer schemes, to create a high resolution modelling framework that is employed to develop and test physical parametrisations and assumptions used in numerical weather and climate prediction. In the UK, the Met Office Large Eddy Model (LEM) is the principal LES that is used within the Met Office and academia. It includes a detailed cloud microphysics representation and a version of the operational radiative transfer scheme. The LEM was initially developed in the 1980s and, whilst the scientific output from the model is cutting edge, the code itself has become outdated. Hard coded assumptions made about parallelism, which were sensible 20 years ago, are now the source of severe limitations and this means that the model does not scale beyond 512 cores. This prevents scientists from carrying out very high resolution modelling on the latest HPC machines.

As machines become larger, and significantly different from the architectures that a code was initially designed for, it can sometimes be easier to re-write poorly performing applications rather than attempt to modernise them through re-factoring. The Met Office NERC Cloud model (MONC) is a complete re-write of the LEM, providing the atmospheric scientific community with a tool for modelling atmospheric flows, turbulence and clouds at very high resolutions and/or near real time. The fact that this aspires to be a community code, along with the desire to future proof to as great an extent as possible, has heavily influenced the design of the code. We have adopted a "plug in" architecture, described in section 3.1, where the model is organised as a series of distinctive, independent, components which can be selected at run-time. This approach, which we discuss in detail, not only allows for a variety of science to be easily integrated but it also supports development targeting different architectures and technologies by simply replacing one component with another.

Other innovative aspects of the code are presented and in particular our approach to data analysis and processing (section 3.2), which is a major feature of the model. These models analyse their raw data to produce higher level information, for instance the average temperature in a cloud or tracking of how specific clouds move through the atmosphere. The existing LEM, like many codes, performs data analysis inline as part of the model timestep which, along with the I/O operation time, is a major bottleneck. In contrast MONC uses the notion of an I/O server, where typically one core per processor is dedicated to handling and analysing the data produced by the model, which is running on the remaining cores. In this manner, MONC can act in a "fire and forget" fashion, asynchronously sending data to the "local" I/O server and continuing on with the next timestep whilst it is being processed.

Based upon the innovative approaches adopted, we present in section 4 performance and scalability results of MONC and discuss some of the lessons learnt, both in terms of large scale parallelism and software engineering techniques, that have become apparent in order to reach the level of scalability and performance demanded by the community. Section

5 draws some conclusions and considers future work.

## 2. BACKGROUND

The Large Eddy Model (LEM) [2] has been an instrumental tool, used by the weather and climate communities, in modelling clouds and atmospheric flows. Since its inception in the late 1980s, it has been a fundamental tool in the development and testing of the Met Office Unified Model (UM) boundary layer scheme [9][10], convection scheme [14][13] and cloud microphysics [1][7]. Given the solid scientific basis of the LEM, as established by the inter-comparison studies of [8][12][15][4], the community continues to heavily rely on this code as a means for furthering the state of the art. The model was initially developed for single processor, scalar machines and then vectorised to take advantage of the Cray C90, with much of the resulting loop and array structure still present in the current code base. It was not until the mid 1990s when the Met Office took delivery of a Cray T3E, their first parallel machine, that the code was parallelised. Since then, from a software point of view, some perfective maintenance has been performed to allow the LEM to run on later generations of machines but the same basic assumptions and principals have remained unchanged since the code was parallelised or even first written thirty years ago.

Even for very simple test cases, the LEM scales very poorly beyond 512 cores. A major reason for this is the fact that the LEM only decomposes in one dimension into slices (in X), there are a minimum of two slices per process and due to the way the grid is decomposed, choices for the global domain size in the X direction are severely limited by those in the Y direction. These restrictions place severe limitations upon the model and, whilst one might wish to extend the size in X to gain more parallelism, this also requires an extension in Y which increases the amount of data held locally and often means that the code reaches memory limits. The result is that the community are often forced to run the code unpacked, where all the memory of a node is used but not all the cores, this is a waste of additional compute resource and explicitly required in order to work around the limitations of the current model. Whilst MPI is used for parallelism, the calls are indirect and go via an abstraction layer called GCOM. This is a throwback to the fact that, in the mid 1990s when the model was first parallelised, MPI was not the de-facto standard that it is now and hence it was quite sensible to decouple the communication technology from the actual model. More recently this layer has become more of a hindrance than a help not least because generations of scientists have misunderstood the semantics of the different communication calls. For example, global barriers can often be found intertwined with point-to-point communications without differentiating between memory re-use in buffered and non-blocking sends.

The other important aspect to consider, from a software engineering perspective, is code maintainability. The LEM is written in a mixture of FORTRAN 66, 77 and 90, employing a variety of old fashioned programming constructs such as global variables, gotos and equivalence blocks. This is further exacerbated by the fact that scientists have modified the same files without the enforcement of code standards, so the style is very inconsistent throughout and changes abruptly. Code management is done via a system called nupdate where the code is organised into a snapshot at a specific version called the base, and user code which contains modifications for patching or specific simulations. These files contain, in addition to the code itself, a series of commands such as deleting lines of code from a specific file in the base, modifying existing code or inserting code. These are all fed into nupdate which effectively pre-processes everything into an intermediate, unstructured form which is then compiled. From a user's point of view, one of the major problems is that compiler messages bear no resemblance to their view of the code which can make debugging very difficult to achieve.

This combination of poor scalability, poor performance and antiquated software engineering techniques has meant that the community are now finding it more and more difficult to effectively use this model for the science that they wish to investigate. Modern machines such as ARCHER, a Cray XC30 (the UK national supercomputing service), and the Cray XC40 that the UK Met Office are taking delivery of in 2015 have hundreds of thousands of cores. Many of the problems that the scientific community wish to tackle require parallelism at this level, however the existing LEM can only take advantage of a fraction of the overall capabilities of these machines and as such requires extensive modernisation.

## 3. MONC

In order to support the current and next generation of science we had a choice between refactoring the existing LEM or using the well validated and trusted underlying science of the LEM as the basis for an entirely new model which shares no code. As a result of the common science and scientific assumptions the original LEM can be used for comparison. Because of the many fundamental issues with the LEM, not just in terms of parallelisation but also how the code is written and managed, we elected to follow the re-write avenue. Whilst keeping the same science the re-write route allowed us to use, from day one, modern software engineering and parallelisation techniques. The new model, called the Met Office NERC Cloud model (MONC) is written in Fortran 2003 with MPI for parallelisation and a number of other third party tools, such as Fruit [3] for unit testing and Doxygen [6] for documentation. There are two important aims for this code, firstly to provide a community model which is easy and accessible for non HPC experts to modify and extend without having to worry about impacts upon other unrelated areas of the code. Secondly performance and scalability are a major concern for our development of the model and in order to support the scientific community's desired problems the code is firmly targeted at the peta- and exascale.

There is a requirement for the model to support multiple compilers, initially the Cray, GNU, Intel and IBM compilers although this list is subject to change in the future. Whilst compiler implementation of the Fortran 2003 standard has reached maturity in some areas this is not universal and other aspects are not as commonly interpreted or well tested by all. Therefore a unit testing framework, which automatically compiles code and runs the tests using these different technologies is critically important for ensuring specific compiler support and code correctness throughout the development process.

## 3.1 Architecture

MONC has been designed around pluggable components where the majority of the code complexity, including all of the science and parallelisation, are contained within these independent units. They are managed by a registry and at run-time the user selects, via a configuration file, which components to enable. The aim was to make it trivial for a user to add their own components. To encourage this a standard means of definition and interaction with the model has been specified. The majority of a component's functionality is contained within optional callback procedures, which are called by the model at three stages: upon initialization, for each timestep and upon model completion. There are no global variables in MONC, but instead a user derived type is used to represent the current state of the model and this is passed into each callback which may modify the state. Using this approach means that the model's current state is represented in a structured manner and the type represents a single point of truth about the model's status at any point in time.

Figure 1 illustrates the outline of a MONC component, the function *test_get_descriptor* provides a descriptor of the component which contains its name, version number and (optionally) populated procedure pointers that represent the callbacks. It can be seen that in this component callbacks have been provided for model initialisation and timestepping. The *initialisation_callback* and *timestep_callback* procedures are the actual callbacks themselves and the current model's state is provided via the *current_state* argument which is of a Fortran derived type and similar to C structs. This *model_state_type* derived type contains the current status of the model in a structured manner which the callback procedures may modify. This component is contained within a Fortran module and is picked up by the MONC build system at compile time, and enabled by the user via *test_component_enabled=.true.* in the configuration file. The MONC registry, which manages these components, also allows for the user to provide more detailed configuration, for instance, determining the order in which components are run for each different callback.

Alongside the numerous components representing scientific, parallelism or miscellaneous functionality there is also a model core. This core contains a minimal amount of code to start the model and both manage and support the components themselves. The way in which the core manages components is via a registry, which stores central information about each component and a list of procedure pointers for initialisation, timestepping and finalisation which are called iteratively rather than having to parse each component for every callback. Whilst components are entirely independent from each other and strictly do not interact, it was identified early on in the development process that they often share some common functionality requirements such as the need for logging, data conversions or mathematical functionality. Therefore a series of utilities have been added to the model core, exposed via an API, which provide common functions that components might require and this saves one reinventing the wheel each time a new component is added.

The model core is mature and the project restrict who may check code in, it is well documented and unit tested to provide a solid foundation for the model. In summary the benefits of adopting a component based architecture for MONC are:

- *Trivial to add new components*: Following the standard format these are picked up, included in the model and then simply enabled in the configuration file. Because components are independent and share no code or variables then they simply plug in and out.
- *Can add immature components without polluting the rest of the code base*: Due the independent nature of these facets, new functionality can be developed without having to modify existing code. This is important as it allows for additional science to be developed, tested and checked into the code repository without impacting other areas of MONC.
- *Simple run-time configuration to customise the model*: A component represents some aspect of the model such as scientific functionality. By adopting this high level approach it is very obvious what functionality is represented in each component and users can easily turn off aspects which are of no interest to their specific run. It is also trivial for users to develop replacement components for areas that they wish to modify or improve. These plug-in via a structure manner. In existing models functionality can often be found a number of levels down in the code, and it can be not only difficult to find calls to disable but also to understand how this might impact the rest of the model.
- *Conceptual simplicity*: From a code point of view the running of the model and how each component works via its own callback procedures is a simple concept to understand.

The model core also contains an options database, which acts as a centralised store for all model configuration options. When the model is started this database is populated, either from a text configuration file for new simulations or an existing model checkpoint file for continuing simulations. The utilities API of the model core exposes functions to components so that they can check for and retrieve information from this database. Upon a model checkpoint write this centralised store is written to the checkpoint file which allows for simple model restarting.

## 3.2 I/O server

In addition to the simulation itself which produces raw (prognostic) results, lower level data is transformed into higher level (diagnostic) information. This data analysis is a crucial aspect of these models. Traditional approaches inline the data analytical aspect with the rest of a model and run it within in a specific timestep after prognostic data has been generated. However this is not optimal, not just because the data analytics involves significant I/O so the model can be stalled waiting for filesystem access, but also because data analysis work commonly involves intensive communications, for instance when calculating the average values of a global field, and ideally one would overlap this with compute.

```
module test_component
  type(component_descriptor_type) function test_get_descriptor()
    test_get_descriptor%name="test_component"
    test_get_descriptor%version=0.1
    test_get_descriptor%initialisation=>initialisation_callback
    test_get_descriptor%timestep=>timestep_callback
  end function test_get_descriptor

  subroutine initialisation_callback(current_state)
    type(model_state_type), target, intent(inout) :: current_state
    ...
  end subroutine initialisation_callback

  subroutine timestep_callback(current_state)
    type(model_state_type), target, intent(inout) :: current_state
    ...
  end subroutine timestep_callback
end module test_component
```

**Figure 1: Component standard interface**

MONC uses an IO server where some of the processes, instead of running the model, are instead dedicated to handling the diagnostic and IO aspects. Typically one core in a processor will run the IO server and this supports the remaining cores running the model. MONC then asynchronously "fires and forgets" the raw prognostic data to the IO server for handling. The user configures the IO server via a structured XML configuration file such that the IO server instructs its MONC processes about the specific type of data required and when. Generic actions for handling this data are included with the IO server, which can be added to if required, and are configured in a high level fashion by the user via the IO server XML configuration file. An example of this data analysis to produce two diagnostic outputs; the mean value of a field at each vertical level and secondly the maximum value of a field at each level. The same, *horizontal reduction* action is used by, the first instance configured with the *mean* operator and the second instance configured with the *max* operator. Their high level configuration is all that is required, with the action and underlying framework taking care of the tricky and lower level details such as having to perform inter IO server communications once local values have been computed. The MONC IO server uses a threading approach, where a pool will supply a thread for handling communications from a model process.

There are a number of alternative IO server implementations in use by the community and integration with our own IO server is not mandatory. At the current time of writing, no existing third party IO servers are entirely satisfactory for the diagnostics that the community required from MONC. However, it is important to future proof the model and from the MONC model's point of view it is simply a component, *io_bridge* which will interface with our IO server. Replacement components, such as *xios_bridge* can be written to, for instance, interface with the XIOS [11] IO server instead. This illustrates an important aspect of the model, where following this pluggable pattern has meant that intricate aspects, such as the handling of diagnostics, is trivial to replace rather than being hard coded in the LEM and other traditional approaches.

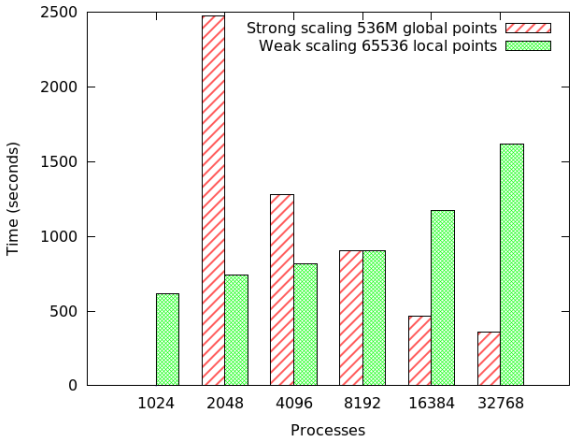



**Figure 2: MONC scaling experiment**

## 4. PERFORMANCE AND SCALING

Performance and scalability testing has been conducted with the dry boundary layer test case which models a dry, neutral boundary layer with a constant geostrophic wind. Experiments have been run on the UK national super computing service, ARCHER, a Cray XC30. Each run has modelled 10000 simulation seconds and involves dynamics, pressure solving and the subgrid scheme. The grid is Cartesian, where the size in the vertical (Z) is 64 and that of X and Y is $n^2$, where the value of $n$ is determined by the desired global size.

Figure 2 illustrates MONC scaling. From the strong scaling results it can be seen that, as the number of processes is increased, the run-time for the simulation decreases. However, there is only a small run-time improvement (100 seconds) between running on 16384 and 32768 cores. The weak scaling results, involve 65536 grid points per process (z=64, x=y=32) and provide a clearer picture of the scaling behaviour at larger core counts. The weak scaling run-time results, up until 8096 cores, are reasonably flat however when weak scaling at 16384 cores (global problem size of

1.07 billion global grid points, z=64, x=y=4096) there is a sharp increase in the run-time which is continued at 32768 cores (2.1 billion global grid points, z=64, x=8192, y=4096.) From these results it can be seen that the code, configured in this manner will run at up to 32768 cores and 2.1 billion grid points, although there is some inefficiency which is impacting the run-time as one reaches the larger core counts.

The results presented in figure 2 use an FFT method for solving pressure terms and analysis at 16k and 32k cores showed that this was taking up a large percentage of the overall run-time. Dealing with pressure terms boils down to solving the Poisson equation and the traditional method involves performing a forward FFT, then in Fourier space solving a vertical ODE before performing a backwards FFT from the spectral domain back to the spatial one. The MONC FFT solver decomposes via a pencil, 2D, decomposition and uses the Fastest Fourier Transformation in the West (FFTW) [5] library for the actual FFT computational kernel. However, each FFT requires global all-to-all communications and as one scales up the fact that each process must communicate with every other process for every FFT becomes a bottleneck.

An iterative solver has also been developed, which solves the Poisson equation using a Krylov subspace method (ILU preconditioned BiCGStab.) The major benefit of this approach is that the only global communication required is a reduction to construct the norm of the residual vector, and all other communications are localised to nearest neighbours for halo swapping. These different solvers have been developed as MONC components, which plug in and out as directed by the user configuration file, and a weak scaling comparison between using an FFT solver and an iterative solver to handle the pressure terms for the dry boundary layer test case are illustrated in figure 3.

The choice between solvers amounts to a trade off between the lower amount of computation but global all-to-all communication of the FFT solver and more significant amount of computation but less communication of the iterative solver. This can be clearly seen in figure 3 where for smaller numbers of cores the FFT solver is more efficient. For instance at 1024 processes solving pressure terms via the FFT solver is 130 seconds faster than using the iterative solver. However as one increases the amount of parallelism this performance gap decreases until the iterative solver overtakes the FFT solver at larger core counts and at 32768 cores the iterative solver reduces the overall run-time by 600 seconds compared to using the FFT solver. The fact that the FFT solver performs so well up until 8096 cores was a surprise to us and this is due to a combination of the very efficient interconnect that can be found on the Cray XC30 along with the highly tuned computation kernels in FFTW.

The results presented so far have all involved the model working in double precision. Whilst some areas of the model must work at this level of precision the pressure solvers do not necessarily need to, especially when solving to 1e-4 which we use in this paper. Instead running the solvers in single precision will not only result in much smaller amounts of data being sent as messages between processes to improve the communication aspects, but will also effectively double

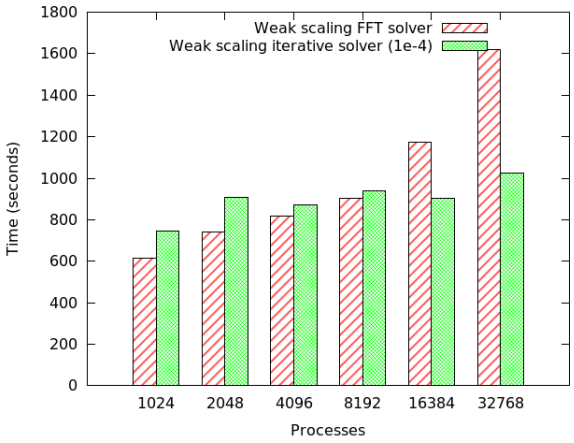



**Figure 3: FFT vs iterative solver weak scaling**

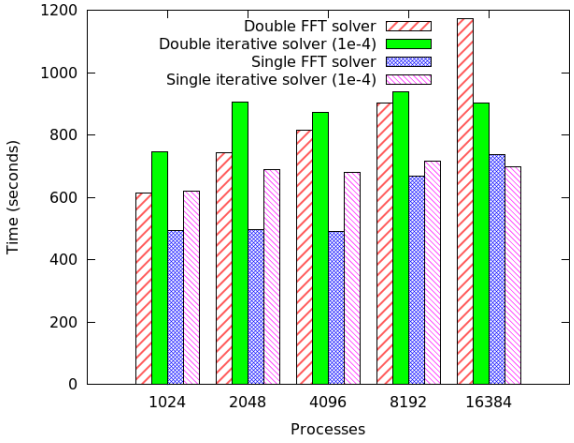



**Figure 4: Double vs single precision weak scaling**

the number of elements that can be held in the cache hence improving the computational side of things too. The FFT and iterative solver components were rewritten in single precision, plugged into the model and the weak scaling dry boundary layer test case was rerun on up to 16384 cores. Figure 4 illustrates a comparison between the two solvers running at single and double precision for the dry boundary layer test case. It can be seen that single precision provides a performance improvement for both the FFT and iterative solvers but the run-time pattern is similar for single precision as they do for double precision; the FFT solver looks favourable initially and then starts to degrade once the cost of communication becomes significant. At 16384 cores by adopting a single precision iterative solver over the traditional FFT solver for pressure terms, this has resulted in an run-time reduction of 476 seconds. It can be clearly seen that single precision, if a weak stopping criteria can be tolerated, does make a difference and is an important optimisation that can be easily applied with our plugable component architecture.

# 5. CONCLUSIONS AND FURTHER WORK

This paper has described the MONC model, from a software engineering and architectural point of view, which delivers a

step change in scalability, performance and capability compared to the existing LEM model. We have described the component based architecture and discussed how this forms the basis for a flexible and extensible code base which the community can easily add their own science to. This simple conceptual view of the model allows the user to easily configure MONC for their own requirements and ensures that runtime is not being wasted in areas not required for a specific simulation. Crucial to performance is how one handles the data analysis aspects of the model and our approach, using an IO server approach to effectively separate this from the raw science, has been introduced. We have demonstrated scalability up to 32678 cores and discussed some of the crucial factors that impact performance at this core count and how the architecture of the model is suited for allowing users to trivially experiment with these aspects.

As the scientific community start to pick up this new model, add their own components and use it in their research, there is still further work to be done from a software point of view. Based upon the results in this paper, it will be interesting to further investigate some of the techniques which have given performance improvements. A bespoke preconditioner, which exploits the problem's known mathematical structure, can be developed which boosts performance of the solver compared to the generic ILU preconditioner. If greater accuracy is required then a mixed-precision solver can be developed, which exploits single precision to achieve performance with a restart in double precision to achieve the desired accuracy. An additional benefit of such a solver is that performance tuning it being done dynamically by the model, rather than relying upon the user. The component based architecture also lends itself to providing support for the model on other platforms, for instance, by developing a number of GPU based components to take advantage of these machines.

# 6. REFERENCES


[1] S.J. Abel and B.J. Shipway. A comparison of cloud-resolving model simulations of trade wind cumulus with aircraft observations taken during rico. *Quarterly Journal of the Royal Meteorological Society*, 133(624), 2007.

[2] A.R. Brown, M.E.B Gray, and M.K. MacVean. Large-eddy simulation on a parallel computer. *Turbulence and diffusion*, (240), 1997.

[3] A.H Chen. Fortran unit test framework (fruit). `http://sourceforge.net/projects/fortranxunit/`, 2015. [Online; accessed 30-April-2015].

[4] A.M. Fridlind, A.S. Ackerman, J.P. Chaboureau, J. Fan, W.W. Grabowski, A.A. Hill, T.R. Jones, M.M. Khaiyer, G. Liu, P. Minnis, et al. A comparison of twp-ice observational data with cloud-resolving model results. *Journal of Geophysical Research: Atmospheres (1984–2012)*, 117(D5), 2012.

[5] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft. *IEEE Conference on Acoustics, Speech, and SignalProcessing*, 3:1381–1384, 1998.

[6] D. Heesch. Doxygen. `http://www.stack.nl/~dimitri/doxygen/`, 2015. [Online; accessed 30-April-2015].

[7] A.A. Hill, P.R. Field, K. Furtado, A. Korolev, and B.J. Shipway. Mixed-phase clouds in a turbulent environment. part 1. large-eddy simulation experiments. *Quarterly Journal of the Royal Meteorological Society*, 140(680), 2014.

[8] S.A. Klein, R.B. McCoy, H. Morrison, A.S. Ackerman, A. Avramov, G. Boer, M. Chen, J.N.S. Cole, A.D. Del Genio, M. Falk, et al. Intercomparison of model simulations of mixed-phase clouds observed during the arm mixed-phase arctic cloud experiment. i: Single-layer cloud. *Quarterly Journal of the Royal Meteorological Society*, 135(641):979–1002, 2009.

[9] A.P. Lock. The parametrization of entrainment in cloudy boundary layers. *Quarterly Journal of the Royal Meteorological Society*, 124(552), 1998.

[10] A.P. Lock, A.R. Brown, M.R. Bush, G.M. Martin, and R.N.B. Smith. A new boundary layer mixing scheme. part i. scheme description and single-column model tests. part ii. tests in climate and mesoscale models. *Monthly Weather Review*, 128(9), 2000.

[11] Y. Meurdesoif. Xios: An efficient and highly configurable parallel output library for climate modeling. In *The Second Workshop on Coupling Technologies for Earth System Models*, 2013.

[12] M. Ovchinnikov, A.S. Ackerman, A. Avramov, A. Cheng, J. Fan, A.M. Fridlind, S. Ghan, J. Harrington, C. Hoose, A. Korolev, et al. Intercomparison of large-eddy simulations of arctic mixed-phase clouds: Importance of ice size distribution assumptions. *Journal of Advances in Modeling Earth Systems*, 6(1):223–248, 2014.

[13] J.C. Petch. Sensitivity studies of developing convection in a cloud-resolving model. *Quarterly Journal of the Royal Meteorological Society*, 132(615), 2006.

[14] J.C. Petch and M.E.B. Gray. Sensitivity studies using a cloud-resolving model simulation of the tropical west pacific. *Quarterly Journal of the Royal Meteorological Society*, 127(557), 2001.

[15] M.C. Vanzanten, B. Stevens, L. Nuijens, A.P. Siebesma, A.S. Ackerman, F. Burnet, A. Cheng, F. Couvreux, H. Jiang, M. Khairoutdinov, et al. Controls on precipitation and cloudiness in simulations of trade-wind cumulus as observed during rico. *Journal of Advances in Modeling Earth Systems*, 3(2), 2011.