Large-scale Performance of a DSL-based Multi-block Structured-mesh Application for Direct Numerical Simulation

G.R. Mudalige[a,*], I.Z. Reguly[b], S.P. Jammy[c], C.T. Jacobs[d], M.B. Giles[e], N.D. Sandham[d]

[a] *Department of Computer Science, University of Warwick, UK*
[b] *Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary*
[c] *Department of Mechanical Engineering, SRM University, Amaravti, India*
[d] *Aerodynamics and Flight Mechanics Group, Faculty of Engineering and the Environment, University of Southampton, Southampton, UK*
[e] *Mathematical Institute, University of Oxford, UK*

Abstract

SBLI (Shock-wave/Boundary-layer Interaction) is a large-scale Computational Fluid Dynamics(CFD) application, developed over 20 years at the University of Southampton and extensively used within the UK Turbulence Consortium. It is capable of performing Direct Numerical Simulations (DNS) or Large Eddy Simulation (LES) of shock-wave/boundary-layer interaction problems over highly detailed multi-block structured mesh geometries. SBLI presents major challenges in data organization and movement that need to be overcome for continued high performance on emerging massively parallel hardware platforms. In this paper we present research in achieving this goal through the OPS embedded domain-specific language. OPS targets the domain of multi-block structured mesh applications. It provides an API embedded in C/C++ and Fortran and makes use of automatic code generation and compilation to produce executables capable of running on a range of parallel hardware systems. The core functionality of SBLI is captured using a new framework called OpenSBLI which enables a developer to declare the partial differential equations using Einstein notation and then automatically carryout discretization and generation of OPS (C/C++) API code. OPS is then used to automatically generate a wide range of parallel implementations. Using this multi-layered abstractions approach we demonstrate how new opportunities for further optimizations can be gained, such as fine-tuning the computation intensity and reducing data movement and apply them automatically. Performance results demonstrate there is no performance loss due to the high-level development strategy with OPS and OpenSBLI, with performance matching or exceeding the hand-tuned original code on all CPU nodes tested. The data movement optimizations provide over 3× speedups on CPU nodes, while GPUs provide 5× speedups over the best performing CPU node. The OPS generated parallel code also demonstrates excellent scalability on nearly 100K cores on a Cray XC30 (ARCHER at EPCC) and on over 4K GPUs on a CrayXK7 (Titan at ORNL).

Keywords: DSLs, Multi-block Structured mesh applications, OPS, SBLI, Finite Difference Methods

1. Introduction

The exa-scale ambitions of the high performance computing (HPC) community, including major stakeholders in government, industry and academia, entail the development of parallel systems capable of performing an exa-FLOP (10^{18} floating-point operations per second) by the end of this decade [1, 2]. The underpinning expectation is that performance improvements of applications could be maintained at historical rates by exploiting the increasing levels of parallelism of emerging devices. However, a key barrier that has become more and more significant is the difficulty in programming these systems. The hardware architectures have become complex where a highly skilled parallel programming know-how is required to fully exploit the potential of these devices. The issue has further been compounded by a rapidly changing hardware design space with a wide range of parallel architectures such as traditional x86 type CPUs with multiple cores, many-core accelerators (GPUs, Intel Xeon Phi), processors with large vector units, FPGAs, DSP[3] type processors and heterogeneous processors. In most cases each architecture could be programmed using multiple parallel programming models while at the same time it is not at all clear which architectural approach is likely to "win" in the long-term. It is unsustainable for domain scientists to re-write their applications for each new type of architecture or parallel system, as their code-base is typically large, usually comprising several millions of LoC. Furthermore, these code-bases most usually have taken decades to develop and validate to produce the required scientific output. Thus, developing

*Corresponding author
Email address: g.mudalige@warwick.ac.uk (G.R. Mudalige)

applications to be "future-proof" have become crucial for continued scientific delivery. A very closely desired feature is performance portability [4], where an application can be efficiently executed on a wide range of HPC architectures without significant manual modifications.

An emerging solution to future-proof applications and achieve performance portability is to separate the concerns of declaring the problem to be solved from its parallel implementation. The idea is to define the problem at a higher abstract level using domain specific constructs and then utilize automated code generation techniques to produce optimized parallel implementations. The technique then essentially allows scientists to maintain the "science" source code without being tied to any platform specific implementation, acting as a base implementation of their model. At the same time, parallel computing experts then have the freedom to derive a parallel implementation using the best optimizations for a given parallel platform. Translating the specification to parallel implementations is carried out by automated code generation techniques such as source-to-source translation and compilation. While this concept of abstraction is not new in computer science, its application to real-world high-performance computing code development has only been successfully attempted recently. In particular, domain specific languages (DSLs) have utilized this development strategy to great effect with a number of successful frameworks targeting several application domains [5, 6, 7, 8, 9, 10, 11, 12].

The underlying goals of this paper are to apply such high-level abstractions and code development for performance portability to large-scale production-grade codes, providing evidence of their utility in future-proofing HPC applications for the exa-scale. Here, we focus on the multi-block structured mesh SBLI (Shock-wave/Boundary-layer Interaction) application developed at the University of Southampton [13]. SBLI has been used extensively for projects within the Aerodynamics and Flight Mechanics group at Southampton as well as with DLR (The German Aerospace Center) [14], European Space Agency, Fluid Gravity Engineering, DSTL, to name a few, and is also representative of a number of codes within the UK Turbulence Consortium [15] and CCP12 for Computational Engineering[16]. At the University of Southampton, it is currently being used for Direct Numerical Simulation (DNS) or Large Eddy Simulations (LES) of shock-induced separation bubbles, receptivity and transition to turbulence and transonic flows. Its initial development was carried out over 20 years ago, but continues to be actively maintained and optimized [17]. Simulations implemented in SBLI are typically direct numerical simulations [18] where the whole range of both spatial and temporal scales of turbulence is directly resolved by the computational mesh without utilizing any turbulence model. In this case the number of mesh points increases rapidly with the Reynolds number.

Recently SBLI was re-engineered [19] to use the OPS [20] embedded domain specific language (EDSL). OPS (Oxford Parallel library for Structured mesh solvers) is aimed at the development of parallel multi-block structured mesh applications and provides a domain-specific API embedded in C/C++/Fortran. It uses source-to-source translation to automatically parallelize applications written using this API. OPS is being used to parallelize a number of applications, including hydrodynamics [6], lattice Boltzmann codes [21] and CFD applications [22, 23]. Currently supported parallel platforms include distributed memory clusters (using MPI), multi-core CPUs including Intel's Xeon Phi many-core processors (using SIMD, OpenMP, MPI and OpenCL) and GPUs (using CUDA, OpenCL and OpenACC) including clusters of GPUs.

In this paper we present research that facilitated the re-engineering of SBLI to utilize OPS and charts the challenges and findings from converting a production-grade legacy application to ultimately execute on modern massively parallel many-core systems at peta-flop scales. Specifically we make the following contributions:

1. We present the process of re-engineering SBLI to utilize OPS. The core functionality of SBLI is implemented using a new Python based open source framework called OpenSBLI [19], which enables a developer to define the problem to be solved at a higher level using Einstein notation. Multiple levels of automatic code-generation are then used to automatically carryout discretization of the problem and translating the higher-level "mathematical" description to a range of platform specific parallelizations. Our work demonstrates the clear advantages in developing maintainable, future-proof and performant applications through this high-level abstractions development strategy. To our knowledge this is the first production-grade multi-block application of its kind to be developed with a high-level abstraction DSL.

2. The multiple levels of abstraction exposes opportunities that can be exploited to increase parallelism and reduce data movement for improved performance. We present two such optimizations specifically aimed at improving the computation-communication balance in the application: (1) re-computing values within compute kernels without storing the values to temporary variables in RAM and (2) reducing memory accesses by tiling[24]. We generate code for OpenSBLI that includes these optimizations and analyze its performance on a range of systems.

3. A representative application developed with OpenSBLI is benchmarked on single node systems. These include systems with multi-core CPUs (Intel Broadwell, Intel Skylake, IBM Power 8) and many-core processors (NVIDIA P100, and V100 GPUs) parallelized with a range of parallel programming models including OpenMP, MPI, and CUDA. We compare the performance with that of the original hand-coded SBLI code while reporting achieved compute

and bandwidth performance on each system. Results demonstrate the versatility of OPS to produce highly performance portable applications.

4. Finally, we present the performance of the OpenSBLI application (parallelized with OPS) comparing it to the performance of the original SBLI code on three large-scale systems, ARCHER at EPCC [25], Titan at ORNL [26] and Wilkes2 at the University of Cambridge [27] on nearly 100K CPU cores and over 4K GPUs. The OPS design choices and optimizations are explored with quantitative insights into their contributions with respect to performance on these systems.

Our work provides evidence into how DSLs and OPS in particular can be used to develop large-scale applications for peta-flop scale systems and demonstrates that in the same environment the performance is on par with the original. Furthermore, the new code is capable of outperforming the legacy applications with platform specific optimizations for modern multi-core and many-core/accelerator systems.

The rest of the paper is organized as follows: In section 2 we briefly introduce OPS including its API and code development strategy. Section 3 details the re-engineering of the SBLI application to utilize the OPS EDSL leading to the creation of the OpenSBLI framework. Section 4 presents performance from a representative multi-block application written using OpenSBLI and compares it to the legacy Fortran-based SBLI. Section 5 gives a brief overview of the state-of-the art with related work in this area. Section 6 concludes the paper.

2. The OPS Embedded DSL

OPS (Oxford Parallel library for Structured-mesh solvers) [20] targets the domain of multi-block structured mesh applications. These applications can be viewed as computations over an unstructured collection of structured mesh blocks (see Figure 1). Within a structured mesh block, implicit connectivity between neighboring mesh elements (such as vertices, cells) are utilized. This is in contrast to unstructured meshes that require explicit connectivity between neighboring mesh elements via mappings. As such, within a structured-mesh block, operations involve looping over a "rectangular" multi-dimensional set of mesh points using one or more "stencils" to access data. While each block will be a structured mesh, a key characteristic of multi-block is the connectivity between blocks via block-halos.

OPS separates the above problem into five abstract parts. This enables a multi-block structured mesh application to be declared at a higher-level by a domain scientist. These parts consists of: (1) structured mesh blocks, (2) data defined on blocks, (3) stencils defining how data is accessed, (4) operations over blocks and (5) communications between blocks. Declaring an application using

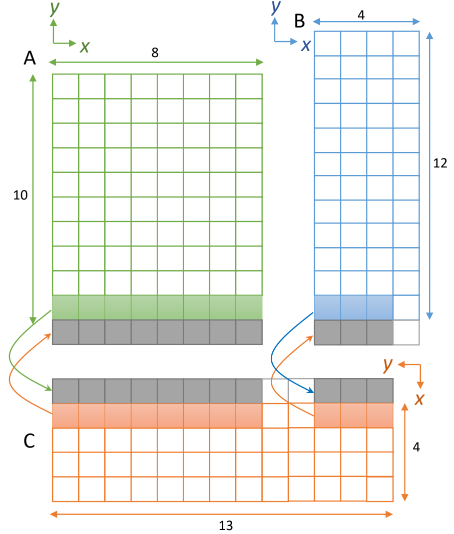


Figure 1: Example 2D multi-block mesh

these high-level domain specific constructs is facilitated by an API which appears to the developer as functions in a classical software library. OPS then uses source-to-source translation techniques to parse the API calls and generate different parallel implementations. These can then be linked against the appropriate parallel library enabling execution on different hardware platforms. The aim is to generate highly optimized platform specific code and link with equally efficient back-end libraries utilizing the best low-level features of a target architecture. The next section briefly illustrates the OPS API and code development with OPS.

2.1. OPS API

From the domain scientist's point of view, using OPS is akin to programming a traditional single-threaded sequential application, which makes development and testing intuitive. Data and computations are defined at a high level, making the resulting code easy to read and maintain.

A typical multi-block scenario is illustrated in Figure 1. Here, three 2D blocks exist in affinity to each other. The data set on the blocks share boundaries, where each boundary takes the form of a halo (called a block-halo) over which data is transfered[1]. Furthermore in this case one of the blocks has a different orientation with respect to the other blocks. This multi-block mesh can be defined using the OPS API[2] as follows. We first define the blocks with its dimensionality and a name (string):

```
ops_block A = ops_decl_block(2,"A");
ops_block B = ops_decl_block(2,"B");
ops_block C = ops_decl_block(2,"C");
```

[1] The block halo is not to be confused with the typical distributed memory / MPI halos.

[2] We discuss and use the C/C++ API throughout this paper, but an equivalent Fortran API also exists

Next, data sets on each block are declared. Assume for this example that block A contains two data sets `dat1` and `dat2` defined on it, while B and C each contain one data set (`dat3` and `dat4` respectively) defined on them. The two data sets defined on A each have a size of 10×8 and a block-halo of depth 1 at the start of their 2nd dimension (i.e. extending the data set in the negative y direction). Similarly the data set on block B has a size of 12×4 and a block-halo of depth 1 at the start of its 2nd dimension (i.e. extending the data set in the negative y direction). The data set on block C has a size of 13×4 and a block-halo of depth 1 at the start of its 1st dimension (i.e. extending the data set in the negative x direction).

```
4  int halo_neg1[] = {0,-1}; //negative block halo
5  int halo_pos1[] = {0,0}; //positive block halo
6  int size1[] = {10,8};
7  int base1[] = {0,0};
8  double* d1 = ... /* data in previously
9                      allocated memory */
10 double* d2 = ... /* data in previously
11                     allocated memory */
12 ops_dat dat1 = ops_decl_dat(A,1, size, base,
13   halo_pos1, halo_neg1, d1, "double", "dat1");
14 ops_dat dat2 = ops_decl_dat(A,1, size, base,
15   halo_pos1, halo_neg1, d2, "double", "dat2");
16
17 int halo_neg2[] = {0,-1};
18 int halo_pos2[] = {0,0};
19 int size2[] = {12,4};
20 int base2[] = {0,0};
21 double* d3 = ... /* data in previously
22                     allocated memory */
23 ops_dat dat3 = ops_decl_dat(B,1, size, base,
24   halo_pos2, halo_neg2, d3, "double", "dat3");
25
26 int halo_neg3[] = {0,0};
27 int halo_pos3[] = {-1,0};
28 int size3[] = {13,4};
29 int base3[] = {0,0};
30 double* d4 = ... /* data in previously
31                     allocated memory */
32 ops_dat dat4 = ops_decl_dat(C,1, size, base,
33   halo_pos3, halo_neg3, d4, "double", "dat4");
```

`ops_decl_dat(...)` declares a dataset on a specific block with a number of data values per data point (1 in this case for all three `ops_dats`) and a size, together with parameters declaring the base index of the data set (i.e. the start index of the actual data), the sizes of the block halos for the data, the initial data values and strings denoting the type of the data and name of the `ops_dat`. In this example arrays containing the relevant initial data (`d1`, `d2` and `d3` of type `double`) are used in declaring `ops_dats`. On the other hand, data can be read from HDF5 files directly using a slightly modified API call (`ops_decl_dat_hdf5`). Alternatively a null pointer can be passed as the data argument to `ops_decl_dat`, which will then be allocated as an empty data set internally by OPS so that it can be initialized later within the application. Once a field's data is declared via `ops_decl_dat` the ownership of the data is transferred from the user to OPS, where it is free to re-arrange the memory layout as is optimal for the final parallelization and execution hardware. Once ownership is handed to OPS, data may not be accessed directly, but can only be accessed via the `ops_dat` opaque handles.

The above separation of data in OPS is driven by the principal assumption that the order in which elemental operations are applied to individual mesh points during a computation may not change the results, to within machine precision (OPS does not enforce bitwise reproducibility). The order-independence enables OPS to parallelize execution using a variety of programming techniques. However it is not inconceivable to relax this restriction for implementing some order dependent algorithms within OPS, with the disadvantage being reduced flexibility for parallelizations. An example of such an order-dependent algorithm, currently being explored for implementation with OPS is the solution to a system of tridiagonal equations (see OPS repository on GitHub [28] feature/Tridiagonal_singlenode branch).

Once blocks and data on blocks are declared, the connectivity between blocks can be declared:

```
34 /* halo from C to A*/
35 int iter_CA[]    = {1, 8}; // # elems in each dim
36 // starting index of from-block
37 int base_from[] = {0, 5};
38 // 1 = x-dim, 2 = y-dim
39 int axes_from[] = {1, 2};
40 // starting index of to-block
41 int base_to[]   = {0,-1};
42 // -2 = negative y dim, 1 = x-dim
43 int axes_to[]   = {-2,1};
44
45 ops_halo halo_CA = ops_decl_halo(dat3, dat1,
46                    iter_CA, base_from, base_to,
47                    axes_from, axes_to);
48
49 /* halo from A to C*/
50 int iter_AC[]   = {8, 1};
51 int base_from[] = {0, 0};
52 int axes_from[] = {1,2};
53 int base_to[]   = {-1,5};
54 int axes_to[]   = {-2,1};
55 ops_halo halo_AC = ops_decl_halo(dat3, dat1,
56                     iterAC, base_from, base_to,
57                     axes_from, axes_to);
58
59 /*create a halo group*/
60 ops_halo grp[] = {halo_CA,halo_AC};
61 ops_halo_group G1 = ops_decl_halo_group(2,grp);
```

Currently, block-halo connectivity declaration is restricted

to a one-to-one matching between mesh points. In the above code listing, the connectivity between the blocks A and C (in Figure 1) are detailed. `iter_CA` gives the number of elements in the block halos, i.e. the number of elements that will be copied over from C to A in each dimension. `base_from` gives the starting index of the element on the source block that will be copied from and `base_to` gives the starting index of the element on the destination block that will be copied to. `axes_from` and `axes_to` defines the orientation in which the elements will be copied from and to. For example, in the above code listing, if the x-dimension (represented by 1) is copied to the y-dimension (represented by 2) and y-dimension copied to the x-dimension, we define `axes_from[] = 1, 2` and `axes_to[] = -2,1`. In the code listing the x-dimension is copied to the negative direction of the y-dimension, which is indicated by the - sign. Once the individual block halos have been defined, they can be combined into a group so that halo exchanges can be triggered for the whole group.

All the numerically intensive computations can be described as operations over the mesh points in each block. Within an application code, this corresponds to loops over a given block, accessing data through a stencil, performing some calculations, then writing back (again through the stencils) to the data arrays. Consider the following typical 2D stencil computation:

```
1 int range[4] = {0, 8, 0, 10}; //iteration range
2
3 for (int j = range[2]; j <range[3]; j++) {
4   for (int i = range[0];i < range[1]; i++) {
5     d2[j][i] = d1[j][i] +
6                d1[j+1][i] + d1[j][i+1] +
7                d1[j-1][i] + d1[j][i-1];
8   }
9 }
```

An application developer can declare this loop using the OPS API as follows (lines 78-80), together with the "elemental" kernel function (lines 69-73):

```
62 /* Stencil declarations */
63 int st0[]  = {0,0};
64 ops_stencil S0 = ops_decl_stencil(2,1,st0,"00");
65
66 int st1[]  = {0,0, 0,1, 1,0, -1,0, 0,-1};
67 ops_stencil S1 = ops_decl_stencil(2,5,st1,"5P");
68 /* User kernel */
69 void calc(double *a, const double *b) {
70   b[OPS_ACC1(0,0)] = a[OPS_ACC0(0,0)] +
71           a[OPS_ACC0(0,1)]  + a[OPS_ACC0(1,0)] +
72           a[OPS_ACC0(0,-1)] + a[OPS_ACC0(-1,0)];
73 }
74
75 /* OPS parallel loop */
76 int range[4] = {0,8,0,10};
77
78 ops_par_loop(calc, A, 2, range,
79     ops_arg_dat(dat2,S0,"double",OPS_WRITE),
80     ops_arg_dat(dat1,S1,"double",OPS_READ));
```

The elemental function is called a "user kernel" in OPS terminology to indicate that it represents a computation specified by the user (i.e. the domain scientist) to apply to each element (i.e. mesh point). By "outlining" the user kernel in this fashion, OPS allows the declaration of the problem to be separated from its parallel implementation. The `ops_par_loop` declares the structured block to be iterated over, its dimension, the iteration range and the `ops_dats` involved in the computation. `OPS_ACC0 and OPS_ACC1` are macros that will be resolved to the relevant array index to access the data stored in `dat2` and `dat1` respectively. The explicit declaration of the stencils (lines 63-67) will additionally allow for error checking of the user code. In this case the stencils consists of a single point referring to the current element and a 5-point stencil accessing the current element and its four nearest neighbors. Further, more complicated stencils can be declared giving the relative position from the current `(0,0)` element. The `OPS_READ` indicates that `dat1` will be read only. Similarly, `OPS_WRITE` indicates that `dat2` will only be accessed to write data to it. The actual parallel implementation of the loop is specific to the parallelization strategy involved. OPS is free to implement this with any optimizations necessary to obtain maximum performance.

The `ops_arg_dat(..)` indicates an argument to the parallel loop that refers to an `ops_dat`. A similar function `ops_arg_gbl()` enables users to indicate global reductions. The related API call of interest concerns the declaration of global constants (`ops_decl_const(..)`). Global constants require special handling across different hardware platforms such as GPUs. As such, OPS allows users to indicate such constants at the application level, so that its implementation is tailored to each platform to gain best performance.

The final API call of note is the explicit call to communicate between data declared over different blocks. `ops_halo_transfer(halos)` triggers the exchange of halos between the listed datasets.

```
83 /* halo transfer over halo group G1 */
84 ops_halo_transfer(G1);
```

As mentioned before, for a given `ops_par_loop`, the order in which mesh points are executed in a block may be arbitrary. However, subsequent parallel loops over the same block respect data dependencies. For example the order in which loops over a given block appear in the code will enforce an execution order that OPS will not violate. On the other hand, subsequent parallel loops over different blocks do not have a fixed order of execution and OPS is free to change this as appropriate. Only halo exchanges between blocks introduce a data dependency and therefore a prescribed order of execution between blocks. Further

details of the OPS API can be found in the user documentation [29], tutorial [30] and in [31, 6].

2.2. Developing Applications with OPS

While input data structures are fixed, based on their description, OPS can apply transformations to tailor them to different hardware. OPS uses two fundamental techniques in combination to utilize different parallel programming environments and to organize execution and data movement. The first technique is to simply factor out common operations/functions for a given parallelization and indeed operations common to multiple parallelizations, and implement them within a classical software library. For example, the distributed memory, MPI-based message passing operations can be viewed as near-identical when used for passing messages between any type of node, be it a CPU based node, a GPU node or indeed other many-core nodes or nodes with hybrid processors. Such functions can therefore be readily implemented in a classical library, well optimized for the kind of message passing operations specifically found in multi-block structured mesh applications. Similarly, parallel file I/O operations such as using HDF5 to read in mesh data can also be factored out to a classical library.

The second technique is code generation or more specifically source-to-source translation, where it is used to produce specific parallelizations for the `ops_par_loop` declarations. Thus for example, a parallel loop to be executed on a multi-core CPU needs to be a multi-threaded implementation and OPS needs to generate the code for a nested loop (based on the dimensionality of the loop) and also generate code to multi-thread the loop nest with OpenMP. The same loop, to be executed on a GPU will require the loop to be implemented for example using CUDA (i.e. OPS will need to generate CUDA code) including code to instruct the system to move data to and from the GPU.

Figure 2 gives an overview of the OPS work flow that a domain scientist will be using for developing a multi-block structured mesh application. The problem to be solved should be declared using the OPS API. During development, the application with API calls can be tested for accuracy by including a header file that provides a single-threaded CPU implementation of the parallel loops and the halo exchanges and compiling with a C++ compiler such as GNU's `g++` or Intel's `icpc`. No code generation is required. This sequential developer version allows for the application's scientific results to be inspected before code generation takes place. It also validates the OPS API calls and provides feedback on any errors, such as differences between declared stencils and the corresponding user kernels, differences between data types and/or mismatches between declared access types (`OPS_READ, OPS_WRITE, OPS_RW` etc.) and how data is actually accessed in a kernel. All such feedback is intended to reduce the complexity of programming and simplify debugging.

Once the application developer is satisfied with the validity of the results produced by the sequential application, parallel code can be generated. At this stage, the API calls in the application are parsed by the OPS source-to-source translator which will produce parallelization-specific code (i.e OpenMP, CUDA, OpenCL, etc.) together with function/procedure calls to the relevant classical library. Thus, for example, to generate a version of the code that can run on a cluster of NVIDIA GPUs, OPS will produce CUDA code interspersed with calls to functions that implement MPI halo exchanges. The generated code can then be compiled using a conventional compiler (e.g. gcc, icc, nvcc) and linked against the specific classical libraries to generate the final executable. As mentioned before, there is the option to read in the mesh data at runtime. The source-to-source code translator is written in Python and only needs to recognize OPS API calls; it does not need to parse the rest of the code. We have deliberately chosen to use Python and a simple source-to-source translation strategy to significantly simplify the complexity of the code generation tools and to ensure that the software technologies on which it is based have long-term support. The use of Python makes the code generator easily modifiable allowing for it to even be maintained and extended by third-parties without relying on expertise of the original OPS developers. Furthermore, the code generated through OPS is itself human readable which helps with maintenance and development of new optimizations.

3. OpenSBLI : Automated Derivation of Finite-Difference Computations

OPS abstracts the parallel implementation of a multi-block structured-mesh problem. However, the problem needed to be declared at a level where loops over blocks and communications/transfers with block-halos are explicitly specified. As such, the problem needs to be posed as iterations over mesh points, i.e. loops and each loop needs to be declared as an `ops_par_loop` statement, together with code to set up the blocks, halos and data on blocks using the OPS API. A further increase in abstraction, one which is utilized in this research when re-engineering SBLI allows a scientist to declare a problem as a set of partial differential equations written in Einstein notation and then automatically generate C/C++ code with OPS API statements that performs the finite difference approximation to obtain a solution. The resulting DSL-based modeling framework called OpenSBLI is the subject of this section.

3.1. Re-Engineering SBLI to OpenSBLI

Legacy versions of SBLI, initially developed at the University of Southampton over 20 years ago comprise static hand-written Fortran code, parallelized with MPI, that implements a fourth-order central differencing scheme and a low-storage, third or fourth-order Runge-Kutta time stepping routine. It is capable of solving the compressible Navier-Stokes equations coupled with various turbulence models (e.g. Large Eddy Simulation models), a TVD

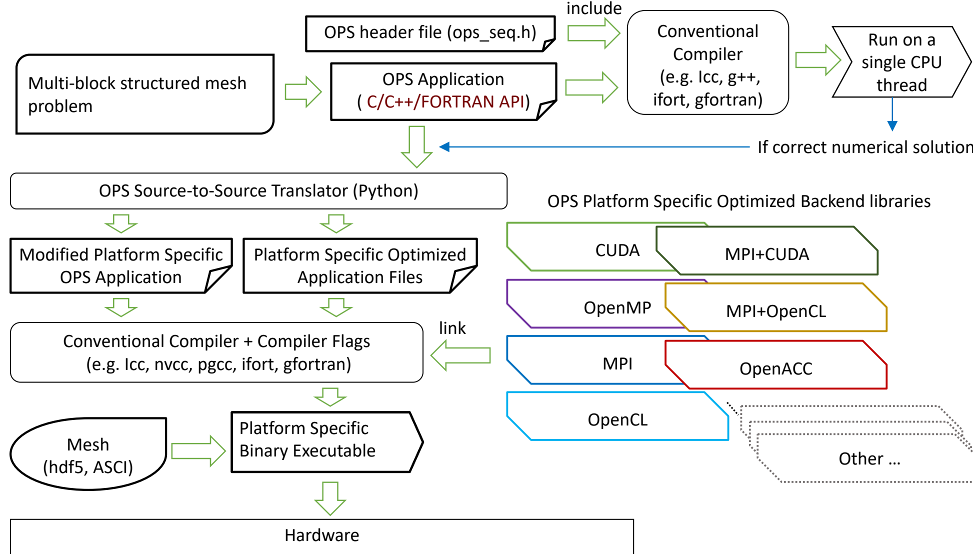


Figure 2: The workflow for developing an application with OPS

scheme to capture discontinuities in the flow and diagnostic routines. The code-base consists of over 64K LoC with multiple different versions of the application used for solving different numerical problems. The core of SBLI is the evaluation of central finite differences in the governing equations.

OpenSBLI, was developed from grounds-up as a replacement to SBLI, making use of code generation to produce the core functionality of the legacy code, but generating OPS API code which in turn enables us to target multiple hardware and parallel programming models [23]. As a result, new functionality is introduced where the finite difference problems that can be solved through OpenSBLI have become a superset of the legacy SBLI code.

To future-proof numerical methods based on finite difference formulations, OpenSBLI separates the numerical solution into two parts: (1) discretization of the governing equations, applying boundary conditions and advancing the solution in time, known as the numerical method (mathematical manipulation), and (2) a computer code that performs the numerical method. OpenSBLI follows the principle that these two can be decoupled as the former is purely mathematical in nature. Thus we can use symbolic mathematics by creating symbolic computations for the solution for the first part. The second part then involves converting the symbolic computations into a code based on the syntax of the desired programming language [23]. In our current work, this target language is C/C++ with the OPS API. A different DSL or a programming languages could also be targeted. Only the code-generation part of the framework should be rewritten. This capability gives rise to what we call "future-proofing numerical methods" where further decoupling the problem implementation from its declaration enables greater flexibility tailored to the context (e.g. numerical method and parallelization method). An additional outcome of this multi-layered strategy is that we are able to develop solutions for finite difference problems beyond the original solvers in the legacy SBLI code-base. As demonstrated in this section one can easily add new equations, change the order of the numerical scheme as well as control the algorithm with relative ease.

OpenSBLI is written in Python and uses the symbolic Python library (SymPy) building blocks. The equation expansion, numerical method discretization, boundary condition implementation and other mathematical manipulations are performed symbolically. To develop an application using OpenSBLI, the user describes the problem in a high-level Python script (which can be viewed as a problem configuration/setup file). This contains the partial differential equations to be solved along with the constituent relations, numerical schemes, boundary conditions and initial conditions. For example, consider the solution of the continuity equation in three dimensions with second order central difference scheme for spatial derivatives. The equations can be written in Einstein notation as:

$$\frac{\partial \rho}{\partial t} = -\frac{\partial \rho u_j}{\partial x_j} \tag{1}$$

Here, ρ is density, u_j is the velocity vector, x_j are the coordinates and t is time. The components of the coordinates and velocity vector in three dimensions are given by x_0, x_1, x_2 and u_0, u_1, u_2 respectively. The key details of the high-level Python code to solve this equation is given below (for the full version see [23]).

```
1 mass = "Eq(Der(rho,t), - Der(rhou_j,x_j))"
2 ...
3 # Define coordinate direction symbol
4 coordinate = "x"
5 ndim = 3
6 # Create a problem
7 problem = Problem(mass, [], ndim, [],
8           coordinate, metrics=[None], [])
```

```
9  # expand the equations
10 expanded =
11 problem.get_expanded(problem.equations)
12
13 grid = Grid(ndim)
14 # Spatial Scheme
15 spatial_scheme = Central(2)
16 ...
17 # Perform the spatial discretization
18 discretise_s = SpatialDiscretisation(expanded,
19                [], grid, spatial_scheme)
20 ...
21 # Generate the code.
22 code = OPSC(grid, discretise_s, ....)
```

The continuity equation is written as strings in OpenSBLI (line 1). This equation is parsed into symbolic form and expanded depending on the number of dimensions of the problem (lines 5 – 11). OpenSBLI also facilitates breaking up long equations into substitutions. Similarly, other equations to be solved along with any constituent relations are expanded. The numerical discretization of Equation 1, requires a numerical grid (that is instantiated in line 13, with the number of points and grid spacing represented symbolically until later replaced with the inputs to a specific simulation). The discretization of the spatial terms in the equations, it requires the numerical scheme to be used. In the current example we are using a second order central finite-difference scheme. This is set by instantiating the `Central` class, with desired order of accuracy as an argument. To change the order of accuracy, this argument needs to be changed. The rest of the script remains the same, showing the ease at which different codes can be generated. The spatial discretization is performed by calling the `SpatialDiscretisation` class with the the expanded equations, constituent relations, the numerical grid and the spatial scheme as inputs. This class collects the spatial derivatives in the continuity equation ($\partial \rho u_j / \partial x_j$) and applies the numerical method to create kernels. A kernel is a collection of equations that are to be evaluated over a specified range of the domain. A collection of kernels give the numerical solution. For example, the derivative $\partial \rho u0 / \partial x0$ results in the following equations added to the kernel:

$$wk0\left[0,0,0\right] = \frac{1}{2dx0}\left(\rho u0[1,0,0] - \rho u0[-1,0,0]\right) \quad (2)$$

The array indexing used in OpenSBLI is the relative indexing approach, similar to the stencil indexing of OPS. The creation of work arrays (e.g. $wk0$) or thread-/process-local variables can be controlled from a higher-level as we shall detail in Section 3.2.1. Once all the kernels are generated for the spatial derivatives, these are ordered based on their dependencies and the original derivatives in the given equations are substituted with their work arrays. The right hand side (RHS) of the equation is converted to:

$$RHS = -\left(wk0\left[0,0,0\right] + wk1\left[0,0,0\right] + wk2\left[0,0,0\right]\right) \quad (3)$$

Similarly, computational kernels are created for temporal discretization, boundary conditions, initialization, input/output, diagnostics etc. All the manipulations are symbolic and the framework, even at this stage of automatic generation, has no knowledge of the final programming language.

The final step is to convert all the symbolic kernels into a specific programming language. As described earlier we generate a compatible code using the OPS API. The `OPSC` class (line 22) of OpenSBLI takes the symbolic kernels and convert them into OPS C/C++ code. Each kernel is written as an `ops_par_loop` over the range of evaluation for that particular kernel. During this process, the type of access for `ops_dat` (read, write or read-write), number for `OPS_ACC` and stencil access are derived from the equations defined for that kernel. The stencil accesses are created as a set to avoid repetitions.

To reduce computationally expensive divisions, the rational constants, division by constants (like $dx0$) are evaluated to a constant variable at the start of the simulation. The full list of optimizations performed are given in [22]. For easier debugging the entire algorithm used to generate the code can be written to a LaTeX file.

3.2. Computation vs. Data Movement Transformations

When a problem is described at a high-level with OpenSBLI and OPS, explicitly stating how and what data is accessed, it exposes opportunities that can be exploited to increase parallelism and reduce data movement. This allows the generated code to be tailored to different target hardware architectures. In this section we present two such transformations that significantly alters the underlying parallel implementation to achieve higher performance through balancing computation intensity vs. data movement.

3.2.1. Re-Computation of Values

With OpenSBLI, one can easily change the underlying algorithm to increase or decrease the amount of work arrays and or computational intensity of the algorithm. In this paper we generate two versions of the code (1) a baseline (referred to as BL) code and (2) a code requiring minimal storage (store none, SN). The BL algorithm follows the conventional approach frequently used in large-scale finite difference codes such as the original SBLI code. This contains subroutines or `ops_par_loop`'s to evaluate the derivative of a function and store it to a temporary array. Due to ease of programming such an implementation, it is typically used when manually writing a finite-difference solver. If a function is a multiple of two different variables (example $rho * u0$) then first the function is evaluated in a loop to a temporary array and then the derivative is evaluated to another temporary array using the subroutines. The BL algorithm uses 41 work arrays (same as SBLI) and 66

derivatives for the convective and viscous terms. Out of these 18 functions are a combination of two different variables (these are evaluated to a temporary variable first). As such the algorithm has low computational intensity, but has significantly more data movement operations. In contrast to BL, the SN version does not store any derivatives to work arrays but the derivatives are stored to local variables and are used to evaluate the right hand side of the equations. If we consider the discretization of the continuity equation (Equation 1), the RHS is evaluated as:

$$var0 = \frac{1}{2dx0}\left(\rho u0[1,0,0] - \rho u0[-1,0,0]\right) \quad (4)$$

$$var1 = \frac{1}{2dx1}\left(\rho u1[0,1,0] - \rho u1[0,-1,0]\right) \quad (5)$$

$$var2 = \frac{1}{2dx2}\left(\rho u2[0,0,1] - \rho u2[0,0,-1]\right) \quad (6)$$

$$RHS = -(var0 + var1 + var2) \quad (7)$$

If a derivative has combination of two different variables or for mixed derivatives ($\partial u0/\partial(x0x1)$), the function or the inner derivative $\partial u0/\partial x0$ is recomputed across the stencil. This increases the computational intensity of the algorithm, while reducing memory accesses. Such an algorithm is generated by setting the `store_derivatives` attribute of the `grid` class to `False`. In addition to BL and SN, the ease at which such modifications can be performed is explained in detail by generating several other variations of the solver in [22].

3.2.2. Tiling

Most computational stages in OpenSBLI, and generally in structured-mesh CFD are bound by how fast the architecture can move data between the processor and main memory. As described above, one way of addressing this is via algorithmic optimizations, but it is also possible to improve data re-use by changing the scheduling of operations. For example, if two subsequent parallel loops access some of the same data, this data (due to its size) will not stay in cache between accesses. Therefore the second loop has to read it again from RAM. However, by only scheduling some of the iterations in the first loop, accessing just a subset of the entire dataset, and then, accounting for dependencies, scheduling the appropriate iterations in the second loop, we can exploit data re-use between these two loops.

This implements a type of cross-loop scheduling, which in this use case we call cache-blocking tiling. Programming such an optimization manually is tedious and virtually impossible for large-scale codes. Traditional compilers and Polyhedral compilers [32, 33, 34, 35, 36, 37, 38] have been targeting this problem for many years, yet they are not applicable to problems of this size. Therefore in OPS we have implemented a feature that determines the cross-loop dependencies and constructs execution schedules at runtime [24]. On top of achieving better cache locality, the algorithm also improves distributed-memory communications; by determining inter-process data dependencies for a number of computational steps ahead of time, it can group MPI messages together and exchange more data but less frequently. Additionally, it can avoid communicating halos for working arrays altogether, by redundantly computing around the process boundaries.

Tiling in OPS relies on delaying the execution of parallel loops, and queuing up a number of them to analyze together. Execution has to be triggered when some data has to be returned to user space. In OpenSBLI we can analyze a large number of loops together, because data is only returned to the user at the end of the simulation. After some experimentation we found that the optimum is to analyze and tile over all the loops in a single time step.

4. Performance

Having re-engineered SBLI to OpenSBLI, we validated the results from the new code to ascertain whether the correct scientific output is produced. Validation consisted of running OpenSBLI on known problems, comparing its results to the legacy SBLI application solving the same problem. The correctness of different algorithms on CPUs were previously reported in [22]. In the present work, the min, max difference between the algorithms for all the conservative variables were found to be less than 10^{-12} on each architecture, and the difference between the runs on CPU, and GPU are found to be less than 10^{-12} for the number of iterations and the optimization options considered here.

After establishing accuracy, the remaining open question is whether the time and effort spent in re-engineering SBLI to utilize OPS, producing OpenSBLI was justified in terms of performance. The key questions are (1) whether the utilization of OPS through OpenSBLI affected (improved/degraded) the performance compared to SBLI (2) what new capabilities can be enabled through OPS that improve performance, (3) whether further performance gains are achievable with modern multi-core/many-core hardware, (4) what quantitative and qualitative performance gains result in computation vs. data movement transformations on different parallel platforms and (5) whether OpenSBLI scales well, considering large-scale systems (with node architectures of pre-exascale and upcoming exascale designs) and problem sizes of interest in the target scientific domains.

We begin by determining whether using a high-level approach such as OpenSBLI and OPS is in any way detrimental to performance when compared to the hand-coded original; it is obviously important to be able to perform a like-for-like comparison and demonstrate that the OPS version can indeed match the performance of the original under identical circumstances. We first explore the single node performance in Section 4.1 followed by performance on a number of large-scale distributed memory systems in Section 4.2.

As mentioned before, a number of known problems were used for validating OpenSBLI [23, 39, 40]. One of them is the 3D Taylor-Green Vortex problem [41, 42, 43], which

Table 1: Single node Benchmark systems specifications

System	Kos	Scyrus	Saffron
Node Architecture	Intel Xeon E5-2660 v4 @ 2.0GHz (Broadwell) + 1 × NVIDIA Tesla V100 GPU + 3 × NVIDIA Tesla P100 GPUs	Intel Xeon Silver 4116 CPU @ 2.10GHz (Skylake)	POWER 8E 8247-42L @ 2.06 GHz
Procs × cores	2×14 (2 SMT/core)	2×12 (2 SMT/core)	2×10 (8 SMT/core)
Memory	132 GB + 16GB (P100) and 16GB (V100)	96 GB	268 GB
O/S	Debian 4.9.51	Debian 4.9.82	Ubuntu 16.04 LTS

Table 2: Compilers and Compiler Flags

Compiler	Version	Compiler Flags
Intel Compilers `icc, icpc` and `ifort`	18.0.2	Broadwell and Skylake : `-O3 -fno-alias -finline -inline-forceinline -fp-model strict -fp-model source -prec-div -prec-sqrt -xHost -parallel`
`nvcc`	CUDA 9.1.85	`-O3 -restrict --fmad false` P100 : `-gencode arch=compute_60,code=sm_60` V100 : `-gencode arch=compute_70,code=sm_70`
IBM XL C/C++ `xlc, xlc++` and `xlf`	V13.1.7 (Beta 2) for Linux	`-O5 -qmaxmem=-1 -qnoxlcompatmacros -qarch=pwr8 -qtune=pwr8 -qhot -qxflag=nrcptpo -qinline=level=10 -Wx,-nvvm-compile-options=-ftz=1 -Wx, -nvvm-compile-options=-prec-div=0 -Wx,-nvvm-compile-options=-prec-sqrt=0`

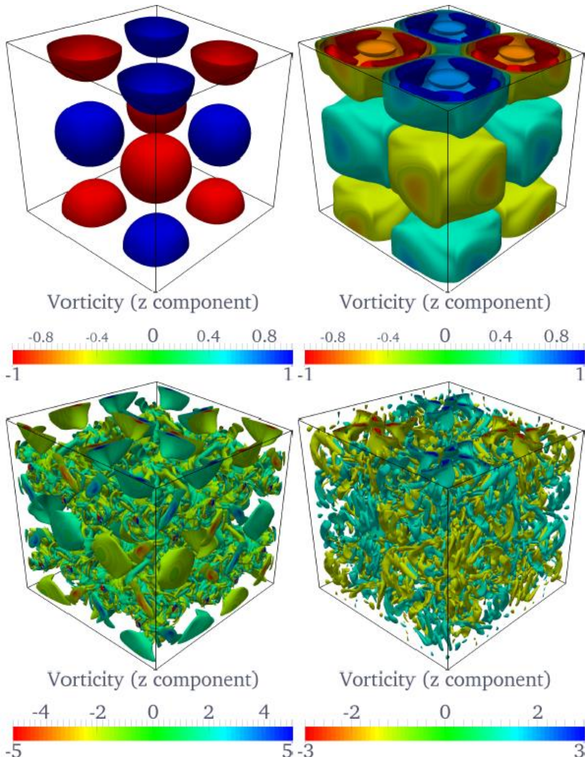


Figure 3: Non-dimensional vorticity iso-contours from the Taylor-Green vortex problem (256^3 mesh). Top left to bottom right: non-dimensional time t = 0, 2.5, 10, 20. [23]

we use in this section to benchmark and explore performance. This problem considers the evolution of a vortex system, starting from the initial stages of roll-up, stretching and vortex-vortex interaction, all the way to vortex break-down and transition to turbulence. No turbulence is artificially induced/generated, such that all the energy is dissipated by the small-scale structures with the compressible fluid eventually coming to rest. The numerical method must be capable of accurately representing each of these physical processes/stages [23]. Figure 3 presents a visualization of the z-component of vorticity in the Taylor-Green vortex test case with a 256^3 mesh points, at various non-dimensional times.

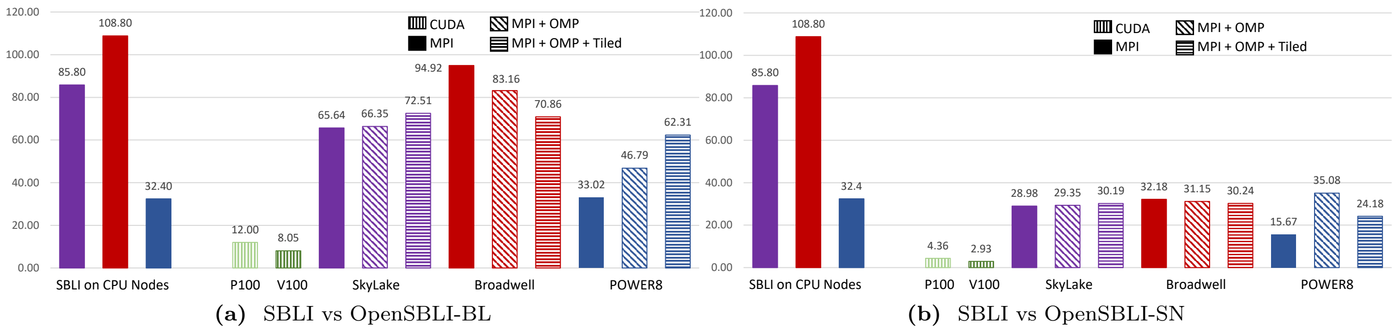
4.1. Single Node Performance

Table 1 provides brief specifications of the single node systems used in our initial performance analysis. The first system, Kos is a single node system consisting of two Intel Xeon Broadwell processors together with multiple NVIDIA GPUs, 1 × Tesla V100 and 3 × Tesla P100. Each Intel processor consists of 14 cores each configured with 2 hyper-threads (SMT). The second system, Scyrus consists of two Intel Skylake processors, each consisting of 12 cores. Again, each core is configured to have 2 hyper-threads. The final system, Saffron, is an IBM POWER8 processor system. It consists of two POWER8 processors, each with 10 cores. Each core is configured to run up to 8 hyper-threads. The compilers and flags used to build SBLI and OpenSBLI applications and OPS are detailed in Table 2. We believe these three single node systems, including the GPUs in them, adequately represent current multi-core and many-core node architectures used in large-scale machines. Performance on these systems provides initial insights into the key questions posed above.

For the remainder of the paper we focus on the performance of the main time-marching loop without considering time spent in file I/O. In production runs it is common to use file I/O for regular checkpointing. A previous paper [44] with OPS demonstrates how high-level abstractions improves the performance of checkpointing. Of particular note is one of the checkpointing modes in OPS that enables a separate thread of execution to save data to disk as a non-blocking call that executes asynchronously. As such there is minimal impact to performance with checkpointing. In the experiments for this paper we do not do any checkpointing as we only focus on time to solution.

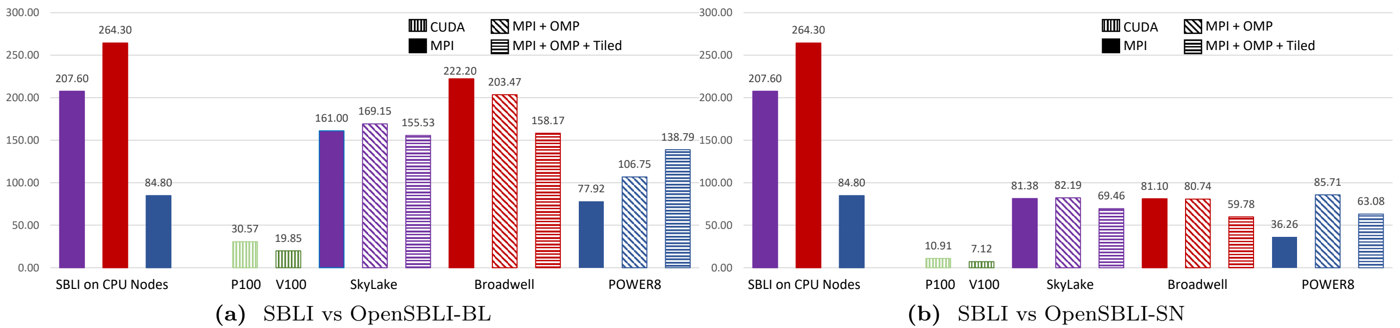
4.1.1. Runtime Performance

Figure 4 (a) and (b) presents the runtime (100 iterations) of the baseline Taylor-Green Vortex application, BL and



(a) SBLI vs OpenSBLI-BL **(b)** SBLI vs OpenSBLI-SN

Figure 4: Taylor-Green Vortex, Single node runtimes (seconds) - 192^3 Mesh



(a) SBLI vs OpenSBLI-BL **(b)** SBLI vs OpenSBLI-SN

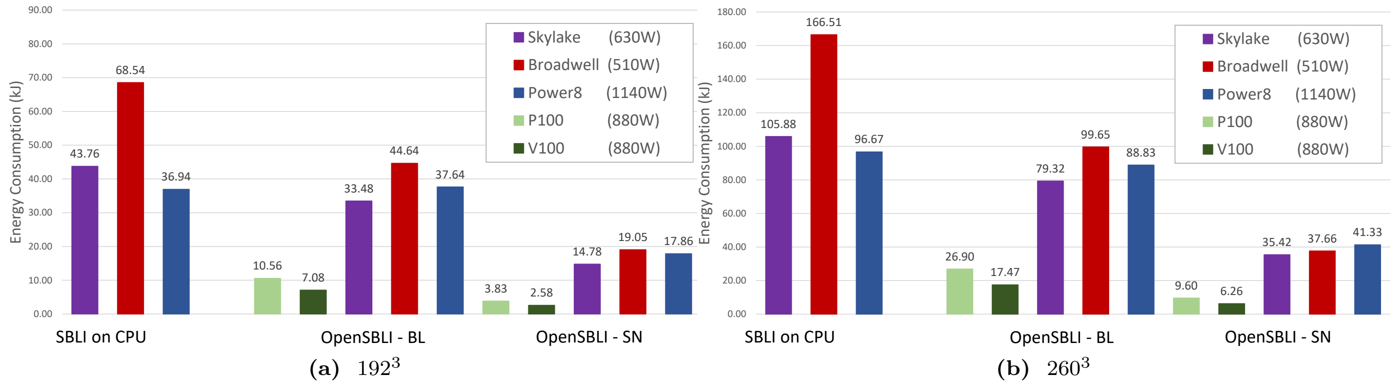
Figure 5: Taylor-Green Vortex, Single node runtimes (seconds) - 260^3 Mesh

the SN version of the application solving a mesh of size 192^3 on the single node benchmark systems. Recall that the SN (Store None) version of the code, has a higher computational intensity, where instead of evaluating the derivatives and storing them on to work arrays, the OpenSBLI code generation process directly replaces the derivatives by their respective finite difference formulas such that they are recomputed every time. The figures present all parallel versions generated through OPS that can execute on each of the benchmark systems in Table 1. For comparison, the first three bars of the graph indicate the runtime of the same problem (and problem size) solved by the original SBLI application on the Skylake, Broadwell and POWER8 nodes respectively. MPI distributed memory parallelism is the only parallel version supported by SBLI.

We see that the baseline Taylor-Green Vortex version, BL implemented with OpenSBLI matches the same application implemented using the legacy SBLI code for this problem size (see Figure 4(a)). The MPI parallel version of BL is actually 30% faster on the Skylake and 15% faster on the Broadwell systems than its SBLI counterpart. On the IBM POWER8, BL gives almost the same runtime as SBLI (less than 2% difference). This result gives us an initial indication that the high-level abstraction and code generation has not resulted in any degradation in performance compared to the original hand-tuned version. Other parallelizations on the Broadwell node give further performance improvements (4× MPI+12× OMP+Tiling giving about 30% speedups), while the MPI only version of BL gives the best performance on the Skylake and POWER8. Considering the performance of BL across systems, the best CPU performance is given by the POWER8 system (running 80× MPI procs) with nearly 2× speedup compared to the two-socket Skylake system (24× MPI) and over 2× speedup compared to the two-socket Broadwell system (4× MPI+14× OMP+Tiled). However, the best overall performance is achieved on the single GPUs using CUDA, with close to 4× speedup over the best CPU performance (POWER8, 80× MPI) on the V100 GPU. The V100's speedup over the original SBLI code is also about 4×.

Considering the SN version of the application (Figure 4(b)), we see close to 3× speedup over SBLI on Skylake (48× MPI), 3.3× speedup on Broadwell (56× MPI) and 2× speedup on POWER8 (80× MPI). The MPI-only SN version also provides over 2× speedup over the MPI-only BL version on the Skylake node and nearly 3× over the MPI only BL version on the Broadwell node. Similarly the SN MPI runtime given by the POWER8 is just over 2× faster than its BL MPI runtime. As such it is evident that optimizing for reducing memory accesses / communications have played a considerable role in improving the performance of the application. The performance of the SN version across systems show that now there are less performance differences between CPU systems. GPUs still give the best runtimes with over 5× speedup given by SN on the V100 GPU over the best performing CPU (POWER8, 80× MPI).

Results from benchmarking the same problem on a



(a) 192^3 **(b)** 260^3

Figure 6: Single node energy consumption (kJ)

larger problem size of 260^3 is given in Figure 5 (a) and (b). In this case, the problem size was chosen such that it maximizes the global memory used on both the P100 and V100 GPUs. Any larger mesh, we found could not be solved by a single GPU with OpenSBLI-BL (the SN version uses less memory). Again we see most of the performance trends from the smaller problem size occurring for the larger problem size. The performance differences between SBLI and the MPI versions of BL on Skylake and Broadwell are now 28% and 18% respectively. On the POWER8, BL performs with a 8% speedup over SBLI. The best BL version on the CPU systems was again on the POWER8 (80× MPI). We also see that there is again a notable benefit of tiling on the Broadwell system with about 30% improvement over MPI+OpenMP. Some minor improvements with tiling is also observable on Skylake. However, as before the best overall runtime is given by the V100 GPU. This is close to 4× speedup over BL and 4.2× over SBLI on the POWER8 system (80× MPI).

Considering the SN version on the 260^3 problem size in Figure 5(b), we see similar performance trends as before. The V100 GPU, is now giving over 5× speedup compared to the best runtime on a CPU system, in this case given by the POWER8 (80× MPI) node.

4.1.2. Power and Energy Consumption

While the runtime performance shows GPUs giving significant speedups over CPUs, it is always difficult to judge performance solely on time to solution. This is particularly true due to the fact that in order to run applications on a GPU, it currently needs to be hosted on a system with a CPU. All the GPU systems we present results for has this characteristic. A fairer cost metric could be the energy consumption of a simulation depending on the hardware. Such a measure could be obtained by estimating the power consumed by the single node system, multiplying it by the runtime of the simulation on that system.

We use the thermal design power (TDP) to estimate the power consumed by each processor. A single Intel Xeon Broadwell CPU has a TDP of 105 Watts[45]. The two socket system used in these experiments will therefore consume upto 210 Watts. We triple this value (630 Watts) to account for the power draw of the whole node, due to memory, disk, networking etc. A similar figure of 510 Watts is calculated for the dual socket Skylake system [46] and 1140 Watts for the dual socket POWER8 [47] system. Both P100 and V100 GPUs each has a TDP of 250 Watts [48, 49]. Given that the runtime figures only consider 1 GPU executing the application we assume that the power consumption will be equivalent to the TDP of a single GPU plus three times the TDP of the dual socket Intel Broadwell CPUs that hosts it. This gives a total of 880 Watts per GPU. These estimates appear to be acceptable compared to the single-node power consumption figures of the three cluster systems we use in Section 4.2.

Figure 6 details the energy consumed during the best runtimes on each architecture for SBLI and OpenSBLI's BL and SN versions. Energy consumption for both problem sizes give similar CPU vs GPU trends. While both the P100 and V100 has a larger power-draw than the Intel CPUs, due to the higher runtime speedups the GPUs exhibit superior energy efficiency per execution. For the best performing SN version, on the 192^3 problem, the P100 and V100 are over 3× to just over 7× more energy efficient than any of the CPU runs. A similar energy efficiency speedup can be observed for the 260^3 problem. The key insight, as we have previously observed is that for the above OpenSBLI application versions [50] and similar mesh based traditional HPC applications [5] the most energy efficient executions are almost always achieved by the systems giving the fastest runtime.

4.1.3. Computation and Bandwidth Performance

Investigating the achieved computation performance, in terms of achieved Flops/s and the achieved bandwidth on each system provides further insights into the behavior of the application, especially given the significant performance difference of the BL and SN versions, and given the memory access optimizations in SN. Table 3 details these achieved performance results for the most time consuming kernels in the BL and SN versions of the applications.

Seven kernels are selected for BL of which three $(001, 022, 037)$ are for the evaluation of multiple functions as described in Section 3.2.1, one each for the evaluation of

Table 3: Achieved Application Bandwidth (BW) GB/s

BL 192^3

Krn.	V100			P100			SklyLake			Broadwell			POWER8		
	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.
001	44.85	725.24	0.29	28.90	462.61	0.45	5.11	84.24	2.55	7.06	128.48	1.84	11.29	186.00	1.15
015	123.21	461.52	0.22	71.33	266.93	0.38	32.09	105.60	0.99	49.12	176.64	0.64	77.58	252.80	0.41
022	38.25	620.94	0.34	26.01	418.15	0.50	5.24	85.44	2.48	10.78	188.68	1.21	11.34	181.20	1.15
037	38.25	620.49	0.34	26.01	417.06	0.50	4.96	79.92	2.62	10.55	181.08	1.23	11.77	188.00	1.10
052	138.19	715.39	1.26	89.29	463.58	1.95	19.77	79.20	11.39	19.43	83.04	11.58	52.07	206.40	4.32
081	227.52	719.40	0.84	145.89	459.99	1.31	46.03	102.72	5.85	48.68	115.84	5.53	96.09	212.40	2.80
091	66.73	681.24	0.35	45.80	463.14	0.51	11.89	88.56	2.68	12.57	99.56	2.53	29.53	217.80	1.08

SN 192^3

	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.
00	479.48	504.03	0.33	247.23	264.16	0.64	35.07	93.60	1.89	35.62	93.52	1.87	86.61	232.00	0.77
01	1076.41	136.86	1.73	750.88	95.78	2.48	120.95	12.96	17.99	134.69	15.00	16.16	231.53	24.80	9.40
02	64.88	660.55	0.36	45.80	464.65	0.51	11.74	87.36	2.71	16.20	123.22	1.97	36.65	270.40	0.87
03	70.78	729.83	0.33	46.71	470.74	0.50	16.04	119.52	1.99	30.96	232.28	1.03	38.72	285.60	0.82

BL 260^3

	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.
001	47.49	757.31	0.68	29.10	469.05	1.11	7.20	131.04	4.46	6.64	113.78	4.84	11.02	181.60	2.91
015	134.62	496.59	0.50	78.27	286.67	0.86	46.83	168.48	1.65	50.23	169.96	1.54	72.73	244.80	1.06
022	37.12	596.21	0.87	24.28	391.11	1.33	8.85	156.64	3.63	14.85	246.48	2.16	12.53	205.60	2.56
037	37.12	597.47	0.87	24.47	392.67	1.32	10.01	174.40	3.21	13.72	227.62	2.34	11.47	188.00	2.80
052	138.14	715.81	3.13	89.70	464.66	4.82	23.86	101.52	23.42	28.11	117.18	19.89	52.30	209.60	10.69
081	229.27	720.49	2.07	146.03	459.23	3.25	44.62	105.76	14.98	79.72	184.56	8.39	94.60	211.20	7.07
091	65.91	670.37	0.88	46.03	466.95	1.26	14.51	114.96	5.45	11.96	91.80	6.61	26.24	195.20	3.01

SN 260^3

	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.	CP GF/s	BW GB/s	T sec.
00	510.28	531.69	0.77	274.76	287.97	1.43	43.21	112.50	3.77	34.36	89.34	4.74	84.93	225.60	1.92
01	1063.04	135.55	4.35	713.62	90.98	6.48	100.63	11.14	53.71	149.19	16.52	36.23	246.75	27.20	21.90
02	65.91	668.40	0.88	44.96	456.26	1.29	22.25	168.26	3.55	13.75	104.04	5.75	30.99	231.20	2.55
03	73.42	746.55	0.79	46.03	467.39	1.26	31.77	238.14	2.49	32.47	243.18	2.44	29.15	217.60	2.71

RHS of convective and viscous terms (052, 081), time advancement (091) and one for the evaluation of constituent relation (015). As there are only four kernels in the SN version all of these are selected. These represent evaluation of constituent relations (00), RHS of the equations (01), temporal advancement sub stage (02) and time advancement (03).

To measure the computation performance, Intel's Advisor 2018 (update 2) profiler was used on the Intel CPU systems. On the GPUs we used NVIDIA's nvprof profiler. Both profilers provided a relatively direct way to obtain the total number of floating-point operations executed in each kernel of interest for the parallelizations tested. On the IBM POWER 8 system, however, a suitable profiler was not available. As such we estimated the number of floating-point operations based on the Intel CPU figure.

The most time-consuming kernel in the BL version of the code (052) achieves approximately 20 GFlops/sec on both Broadwell and Skylake for the 192^3 problem. This is only about 4% and 3.2% of the DGEMM peak of the two processors on each system (623.19 GFlops/sec and 482.53 GFlops/sec respectively). For the two socket POWER8 the most time consuming kernel achieved about 52 GFlops/s which is about 10% of peak (501 GFlops [51]). Such a low computation performance achieved of the peak is more prominent on the P100 and V100 GPUs with only less than 3% achieved (out of 4.7TFlops/sec and 7TFlops/sec respectively) on either of the GPUs for the most time consuming kernel. Moving to the SN version, we see a considerable improvement of the achieved computation performance of the most time consuming kernel (01). On the Intel Broadwell and Skylake systems, 25% and 21% of the peak is achieved respectively, and on the POWER8 46% of the peak is reached. The achieved computation performance is also improved on the V100 and P100 GPUs, both at about 15% of the peak. A similar trend can be seen for the 260^3 problem size for both BL and SL. Considering all the kernels, we see that the SN version achieves over 3.5× the average achieved GFlops/s rate for BL (i.e. total floating-point operations / sum of time spent in the most time consuming kernels) on all the CPU systems. On both GPUs, SN gives over 6× the computation performance vs BL.

To compare with each system's peak achievable bandwidth performance we establish a simple roof-line model by benchmarking all the sockets of a node for the CPUs using the STREAM[52] benchmark. For the GPUs we use the BabelSTREAM [53] benchmark on a single GPU. Figure 7

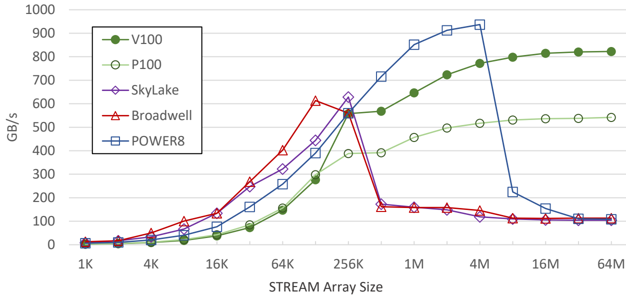


Figure 7: STREAM Bandwidth performance of node systems

plots the achieved BW for Triad, (1K to 64M number of elements in the benchmark array), repeated 100000 times. The roof-line model can be compared with the achieved bandwidth of the most time consuming kernels in the BL and SN versions of the application as detailed in Table 3. The kernels and their bandwidths here have been selected from the best performing parallel version of the applications (whose runtimes are indicated in Figures 4 and 5). Bandwidth figures shown for each kernel are rough estimates computed by the automated profiler in OPS, which uses the iteration range and the type of access (R/W) to data. As such, the calculation ignores data reuse within a single loop nest due to multi-point stencils which stay in cache.

Firstly, it is noted that for each problem size the achieved bandwidth of each kernel on each system is, in the majority of cases approximately similar. The CPU systems, Skylake, Broadwell and POWER8, has achieved a STREAM bandwidth to DDR4 of about 112 GB/s, 128GB/s and 140 GB/s respectively. Any higher bandwidth is bandwidth to the last level cache. It can be seen that, on the CPU systems, the most time-consuming kernel in the BL version (052)of the code utilizes over half of the DDR4 bandwidth. Other top kernels have an even higher utilization. It is, therefore clear that the performance limiting factor of BL is memory bandwidth. In contrast, for SN, on all CPU systems, the most time consuming kernel has less than 15% of DDR4 bandwidth utilization. On the GPUs, the kernels utilize over half of the BW, for both TGV and SN, for both problem sizes. As such we can conclude that SN is still bandwidth limited on the GPUs.

4.2. Large-scale Performance

Next, we explore the scaling performance of the application. Table 4 gives brief details of the three large-scale systems we used for benchmarking. The first system, ARCHER [25] based at the University of Edinburgh, is a Cray XC30 system consisting of a CPU only node architecture. ARCHER is based on the Intel Ivy Bridge processors, each node consisting of two processors, each with 12 cores. The other two systems are GPU clusters, Titan at ORNL [26] and Wilkes2 [27] at the University of Cambridge. Titan, a Cray XK7 system, consists of an older generation of NVIDIA GPUs, the K20x, one GPU per node while Wilkes2 consists of four NVIDIA P100 GPUs per node. On both the Cray systems we used the Cray compilers to compile the application and the OPS library. On Wilkes2, we used the Intel 17.0.4 compiler suite. We found that these compilers (and the flags used) gave us the best performance on the respective systems.

4.2.1. Runtime and Scaling Performance

Figure 8 reports the runtime (again for 100 iterations) of the Taylor-Green Vortex problem's SBLI, and OpenSBLI versions on ARCHER. The x-axis represents the number of nodes on each system, where an ARCHER node has 24 cores. The run-times (on the y-axis) are the minimum obtained from about 5 runs for each node count. The standard deviation in run times was significantly less than 10%. Figure 8(a) presents strong scaling, where a mesh size of 384^3 is solved at increasing machine size. Figure 8(b) presents weak-scaling, where a mesh size of 192^3 per node is solved. In this case the problem size increases with the machine size. The largest machine size we tested the application is on 4K nodes (98304 cores), which is $4/5^{ths}$ of the total machine size.

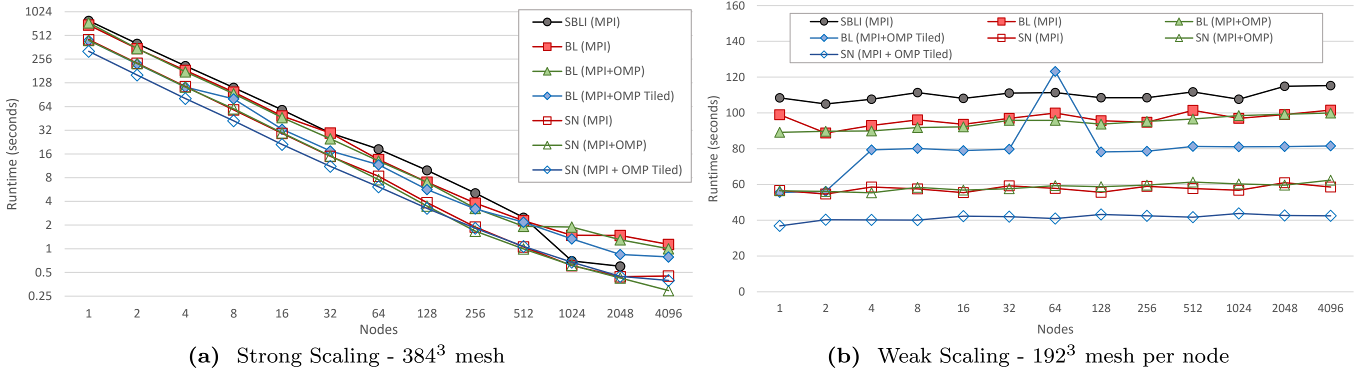
All versions of the application demonstrate good strong scaling (see Figure 8(a)) on ARCHER, with near linear scaling up to 512 nodes. SN's MPI+OMP with tiling version is approximately two times faster than SBLI up till this node count. However, it appears that SBLI gets a performance boost due to the smaller mesh size solved by a single node possibly fitting in cache, from 1024 nodes. At larger machine scale SN's non-tiled MPI+OMP paralleization gives better scalability. It appears that the overhead of tiling may outweigh the performance benefits of it at this problem size per node. It is on weak-scaling, we see the significant benefits of tiling. In the weak scaling graph, Figure 8(b), the best runtime at increasing scale is given by SN's MPI+OMP with tiling (nearly 50% faster over MPI+OMP without tiling). It is nearly 2.75× faster than SBLI, and 2× faster than the best BL runtime given by MPI+OMP with tiling. Additionally, there appears to be excellent weak scaling where the runtime remains nearly the same at larger and larger machine size on ARCHER.

Next, Figure 9 compares the best scaling runtime on ARCHER with that of the GPU clusters. The best runtime on ARCHER is given by SN, MPI+OMP for strong scaling and MPI+OMP Tiled for weak scaling. The parallel version on the GPU clusters use MPI+CUDA. For comparison we also indicate the runtime from SBLI on ARCHER. The x-axis represents the number of nodes on ARCHER and the number of GPUs in the other two systems. In other words the comparison is between 1 GPU and 1 ARCHER (2 CPUs, i.e. 24 cores) node. We believe that this provides a fairer comparison, given that there are 4 GPUs per Wilkes2 node. The single node power draw as computed in Section 4.2.2 gives further justification, where a Wilkes2 node has approximately four times the power draw of a Titan node. The largest machine size used is 4k nodes (i.e. 4K GPUs) on Titan. The equivalent number of AMD Opteron cores is 61440.

Strong scaling (Figure 9(a)) again uses a total problem

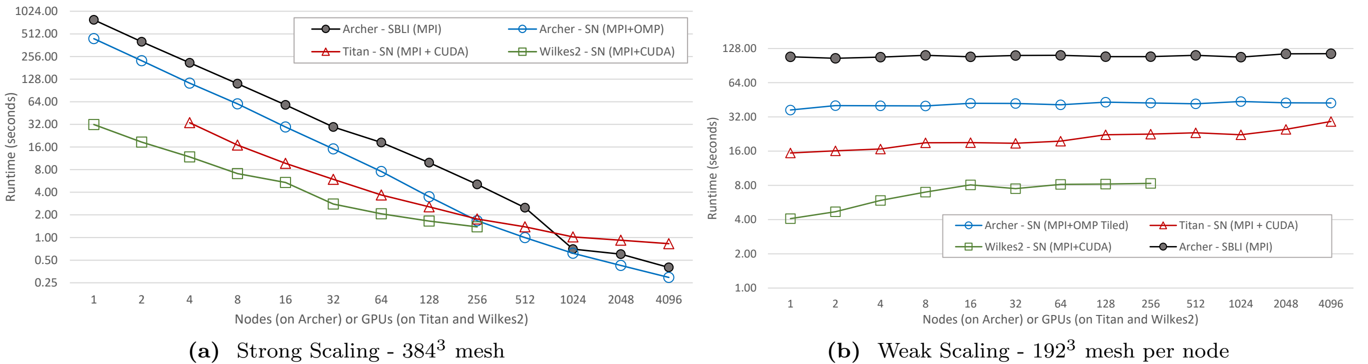
Table 4: Large-scale systems specifications

System	ARCHER (Cray XC30)	Titan (Cray XK7)	Wilkes2 (P100 Cluster)
Node Architecture	Intel Xeon E5-2697 v2 @ 2.7 GHz (Ivy Bridge)	AMD Opteron 6274 @ 2.2 GHz (Interlagos) + NVIDIA Tessla K20x GPU	Intel Xeon E5-2680 v4 @ 2.40GHz + 4 × NVIDIA Tessla P100 GPUs
Procs × cores per node	2 × 12	1 × 16 + 1 GPU	2 × 12 + 4 × GPUs
Memory/Node	64 GB	32 GB + 6 GB (K20x)	96GB + 16GB/P100 GPU
Total Nodes(cores), GPUs	4920 Nodes (118,080)	18688 Nodes, 18688 GPUs	90 Nodes, 360 GPUs
Interconnect	Cray Aries	Cray Gemini	Mellanox EDR Infiniband
OS	CLE 3.0.101	CLE 3.0.101	Scientific Linux release 7.5 (Nitrogen)
Compilers	PrgEnv-cray/5.2.82	PrgEnv-cray/5.2.82, CUDA 9.1	Intel Compilers 17.0.4, CUDA 9.0
Compiler flags	`-O3`	`-O3`	`-O3 -fno-alias -finline -inline-forceinline -xHost -parallel`



(a) Strong Scaling - 384^3 mesh **(b)** Weak Scaling - 192^3 mesh per node

Figure 8: Scaling on ARCHER



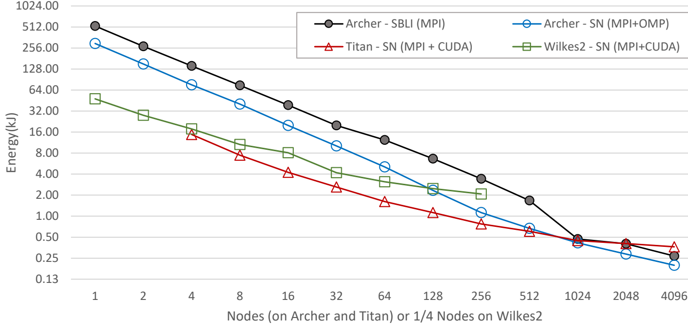
(a) Strong Scaling - 384^3 mesh **(b)** Weak Scaling - 192^3 mesh per node

Figure 9: Scaling on Titan and Wilkes2 vs ARCHER

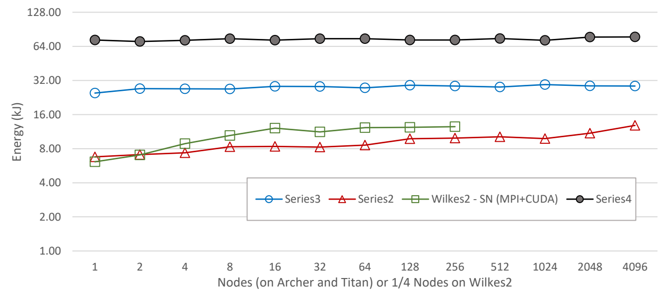
size of 384^3 solved at increasing machine size. At small machine sizes (1 to 64 GPUs or nodes), Titan gives about 4× to 2× the speedup than ARCHER, while Wilkes2 gives speedups ranging from 8× to 4×. However, ARCHER begins to narrow down this speedup from 128 nodes, overtaking the Titan runtime at 1024 nodes and later at 4K nodes giving a speedup of 2× over Titan. However, this strong-scaling behavior is completely expected, as each GPU, at larger scales are solving a smaller and smaller problem size. This results in (1) each GPU not having enough parallelism to be exploited within a partition assigned to it and (2) each partition not providing sufficient computation to hide kernel launch and communication latencies (PCIe and network latency). However, the weak-scaling results (Figure 9(b)) demonstrate the significant benefits of the GPU clusters where speedups ranging from 2.3× to 1.4× on Titan and over 4× on Wilkes2 compared to ARCHER can be seen at scale.

4.2.2. Power and Energy Consumption

Similar to the energy consumption estimates carried out in Section 4.1.2 we can estimate the energy consumed per execution on each of the large-scale clusters. The total power draw of a single node of ARCHER is 672 Watts. This was estimated from their Top500 submission [54] for the whole system (3306kW), divided by the total number of nodes (4920). A similar estimate for a Titan node is 439 Watts (8209kW/18688) [55]. A single node in the Wilkes2 system was estimated to have maximum Wattage of 1500 based on empirical measurements by the Wilkes2 cluster administrators, consisting of hardware described in [56]. These enables us to calculate the energy consumption due to the execution of the applications at scale, as detailed in Figure 10 for both strong and weak scaling. Note that these graphs plots the nodes for Titan and ARCHER on the x-axis while plotting quarter nodes for Wilkes2, given that a Wilkes2 node consumes approximately four times

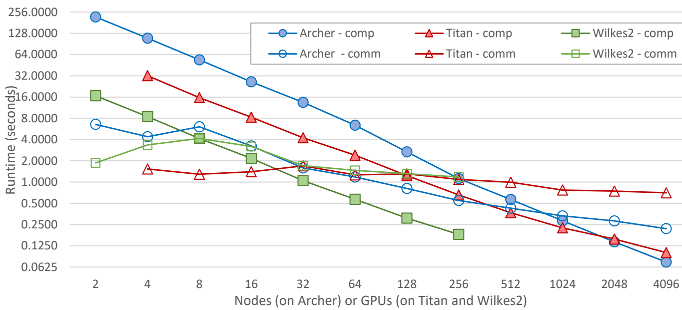


(a) Strong Scaling - 384^3 mesh

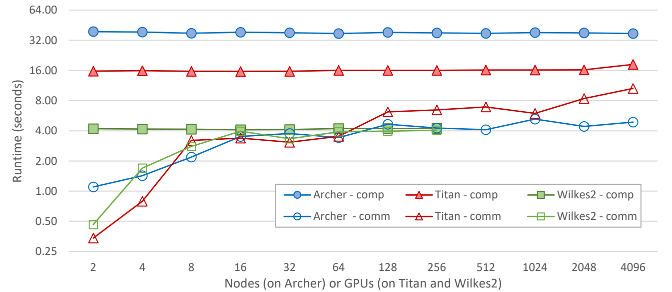


(b) Weak Scaling - 192^3 mesh per node

Figure 10: Energy consumption at scale on Titan, Wilkes2 and ARCHER



(a) Strong Scaling - 384^3 mesh



(b) Weak Scaling - 192^3 mesh per node

Figure 11: SN - Computation vs Communication times at scale on Titan, Wilkes2 and ARCHER

the power consumption of a Titan node. Again we see that best runtimes produce the most energy efficient executions. However an exception to this is Titan vs Wilkes2 where the faster Wilkes2 runs were not fast enough to overtake Titan's superior energy-performance.

4.2.3. Computation and Communications Performance

Breaking down the total runtime to their computation and communication time components provides further insights into the behavior of the application at scale. Figure 11 details the time spent in computing and time spent in communicating (including PCIe and MPI halo exchange time) during the best performing runs in Figure 9. The breakdown for strong scaling in Figure 11(a) shows how, as expected, the computation per node (or GPU) decrease at higher scale near-linearly. However, communication times does not reduce at the same rate. This behavior is more prominent on the GPU clusters, indicating that there is an overhead that does not reduce for communicating smaller halos at scale. As such we conclude that this overhead is due to message communication latency between GPUs across nodes and over PCIe. We see comparable behavior for weak scaling in Figure 11(b) where now the computation time, as expected remains largely constant. The communication time also remains constant for most of the runs, but tend to gradually increase at higher scales. This is more prominently observable on Titan. This evidence suggest that further optimizations to the MPI+CUDA code generated by OPS should be explored to improve performance.

5. Related Work

The use of domain specific languages, and similar high-level abstractions for future-proofing applications and to achieve performance portability has been steadily gaining wider acceptance in HPC. While the concept of abstraction is not new in Computer Science, its application to real-world high-performance computing code development has only been successfully attempted recently.

OP2 [57, 5], also an embedded DSL in C/C++ or Fortran, is one of the earliest high-level abstraction frameworks to demonstrate the use of this development strategy for production applications. OP2 targets the domain of unstructured mesh applications. In [58], OP2 was used to re-engineer a production CFD application from Rolls-Royce demonstrating higher scalability and speedups on large-scale multi-core many core systems. Related frameworks for unstructured mesh applications include FeniCS [9], Firedrake [8, 10] and PyFR [11]. All three of these provides a higher-level notation for declaring a particular class of problems, whereas OpenSBLI is more general in that it can be used for any equations to be solved using explicit finite differences. A similar high-level mathematical notation is provided by Devito [12], for the solution of finite difference problems, particularly arising in seismic inversion problems. Devito, is perhaps the most similar framework to OpenSBLI in terms of the use of symbolic Python. While the numerical methods adopted are similar, OpenSBLI has other advantages of Einstein expansion, more boundary conditions and changing the back-end programming language. Additionally Devito is currently only

able to utilize CPU systems (including Intel XeonPhis) for parallel execution.

There also exists several high-level DSL-type frameworks used in weather modeling applications such as STELLA [59] (and its successor GridTools) and PSyclone [60]. While the parallel motif targeted by these frameworks are the same as OPS (i.e. structured-mesh applications), they are specialized for weather simulations. However they differ from OpenSBLI in how the problem declaration is created using a high-level mathematical notation. They also do not provide the same range of target parallelizations as OPS does (e.g PSyclone currently uses only OpenACC for executing on GPUs) and their combinations with distributed memory execution.

Kokkos [7] and RAJA [61] relies on C++ templates to provide a thin portability layer for parallel loops. As such the abstraction is not as high as OPS and no code generation is carried out. Through template meta-programming, Kokkos and RAJA allow to link to different back-end implementations, specifically OpenMP, CUDA and their combinations with MPI. Given the abstraction at parallel loop level, they are able to handle a wider range of domains.

6. Conclusions

In this paper we focused on the re-engineering of the multi-block structured mesh SBLI application developed at the University of Southampton to utilize the OPS embedded domain specific language. The paper charts the steps and challenges in converting a production-grade legacy application to execute on modern massively parallel many-core systems.

The core functionality of the legacy SBLI application written in Fortran was implemented in a new framework called OpenSBLI. OpenSBLI allows finite-difference problems to be declared using Einstein notation with symbolic Python and then automatically carryout discretization and generation of OPS (C/C++) API code. OPS is then able to automatically generate a wide range of parallel implementations that can be executed on multi-core CPUs, many-core GPUs using a range of programming models such as SIMD, OpenMP, CUDA, OpenCL and OpenACC and their combinations with MPI.

We also demonstrated how the high-level abstractions development strategy opens up opportunities for further optimizations, particularly data movement or communication avoidance. OpenSBLI is able to automatically generate OPS loops that has different computation/communication intensities; we presented details of two versions generated via OpenSBLI that operates at the two ends of this spectrum - highly-bandwidth limited and highly computationally intensive. OPS, at a lower level, is also able to further optimize for data movement using cache-blocking tiling.

We then explored the performance of the code generated automatically with OpenSBLI to that of the code manually implemented in SBLI. A representative application, a Taylor-Green Vortex problem, was used for this comparison. Performance results indicate a number of important conclusions: (1) There is no evidence to indicate that the high-level abstractions development method in OpenSBLI causes any performance degradations compared to the hand-tuned SBLI application; in a like-for-like comparison OpenSBLI performed upto 30% better on the Intel CPU nodes tested, and matched the performance on the POWER8 node. (2) Data movement optimizations provide additional speedups, where reducing the communications intensity with OpenSBLI provide 2× - 3× speedup on all the CPU node systems and tiling providing additional speedups (up to 30%) on the Intel CPU systems, particularly when large problem sizes are solved. The balance of computation to communications change due to these optimizations, where we see over 3-fold increase on CPUs and 6-fold increase on GPUs in the achieved computational performance. The achieved bandwidth of the most time-consuming kernels of the OpenSBLI application, compared to the STREAM bandwidth of the executed systems, also confirms the significant change of bandwidth utilization with these optimizations. (3) Significant speedups on GPUs over equivalent CPU nodes were obtained. A single V100 GPU providing 5× speedups (using the best application version) over the best performing CPU node (the two-socket IBM POWER8 CPU node). These speedups, we observed, also helps the GPU executions be between 3×to 7×more energy efficient than any of the CPU systems. This holds, in spite of the GPUs (including their host CPU system) having a much larger Wattage than the Intel CPU systems. (4) Excellent strong- and weak-scaling was observed on ARCHER, a large-scale CPU cluster at nearly 100K cores. The OpenSBLI versions matched or outperformed the SBLI code. (5) Good scalability of the MPI+CUDA version generated through OpenSBLI, was found on GPU clusters, Titan and Wilkes2 on over 4K GPUs, but providing evidence to indicate further improvements to halo exchanges should be carried out to get near-ideal scaling on these systems. (6) Significant speedups in terms of strong-scaling were given by the GPU clusters, Titan and Wilkes2 at smaller machine scales compared to ARCHER, but diminishing returns were observed as the problem size per GPU gets reduced at higher scale. Significant speedups ranging from 2.3× to 1.4× were seen on Titan compared to ARCHER when weak-scaling on all machine scales tested. Over 4× speedups were seen on Wilkes when weak-scaling compared to ARCHER. However, we observed that the energy efficiency of Titan during the execution of these applications at scale were better than Wilkes2 due to the low Wattage of Titan nodes.

Future work aims to further optimize the distributed memory communications performance that led to less-than ideal scaling. Optimizations include overlapping computations with communications such that communication time could be hidden behind computation time with non-blocking MPI exchanges. Further improving performance through fully hybrid execution, where both the CPUs and

the GPUs of a node can be used to carry out useful computations on a mesh can be explored. Load balancing will be an issue in this case, where the partition of the mesh allocated to the CPU will need to be carefully selected such that it can execute on-par with the attached GPU so that no overall slow-down occurs. Implementation of the back-end library infrastructure for OPS to allow the solution of order dependent algorithms such as the solution to a system of tridiagonal equations or wave-front algorithms will also be carried out to extend the classes of applications that can be solved with OPS.

OPS, OpenSBLI and the Taylor-Green Vortex application used in this research are available as open source software in [20] and [19] respectively. The developers welcome new users and developers to these projects.

Acknowledgements

Funding for this research has come from the European Commission Horizon 2020 project entitled "ExaFLOW: Enabling Exascale Fluid Dynamics Simulations" (reference 671571) and the UK EPSRC project entitled "Future-proof massively-parallel execution of multi-block applications" (EP/K038567/1). Gihan Mudalige was supported by the Royal Society Industrial Fellowship Scheme (INF/R1/180012). István Reguly was supported by the János Bólyai Research Scholarship of the Hungarian Academy of Sciences. Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme.

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (http://www.csd3.cam.ac.uk/), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1).

An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

This research used the resources of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. http://dx.doi.org/10.5281/zenodo.22558

References

[1] W. Dally, Power and programmability: the challenges of exascale computing, http://www.orau.gov/archI2011/presentations/dallyb.pdf (2011).

[2] M. Snir, B. Gropp, P. Kogge, Exascale research: Preparing for the post-moore era, whitepaper, http://hdl.handle.net/2142/25469 (2011).

[3] Top500 News - China Will Deploy Exascale Prototype This Year, https://www.top500.org/news/china-will-deploy-exascale-prototype-this-year/ (2017).

[4] S. J. Pennycook, J. D. Sewall, V. W. Lee, A metric for performance portability, 2016, in: S. A. Jarvis, S. A. Wright, and S. D. Hammond. (eds) High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2016, Saltlake City, UT, USA, November 16, 2016.

[5] G. R. Mudalige, M. B. Giles, J. Thiyagalingam, I. Z. Reguly, C. Bertolli, P. Kelly, A. Trefethen, Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems, Parallel Computing 39 (11) (2013) 669–692.

[6] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, J. A. Herdman, Performance analysis of a high-level abstractions-based hydrocode on future computing systems, Springer International Publishing, Cham, 2015, pp. 85–104, in: S. A. Jarvis, S. A. Wright, and S. D. Hammond. (eds) High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers. doi:10.1007/978-3-319-17248-4_5.

[7] H. Carter Edwards, C. R. Trott, D. Sunderland, Kokkos, J. Parallel Distrib. Comput. 74 (12) (2014) 3202–3216. doi:10.1016/j.jpdc.2014.07.003.

[8] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G.-T. Bercea, G. R. Markall, P. H. J. Kelly, Firedrake: Automating the Finite Element Method by Composing Abstractions, ACM Transactions on Mathematical Software.
URL http://arxiv.org/abs/1501.01809

[9] K. B. Ølgaard, A. Logg, G. N. Wells, Automated Code Generation for Discontinuous Galerkin Methods, CoRR abs/1104.0628.

[10] C. T. Jacobs, M. D. Piggott, Firedrake-Fluids v0.1: numerical modelling of shallow water flows using an automated solution framework, Geoscientific Model Development 8 (3) (2015) 533–547. doi:10.5194/gmd-8-533-2015.

[11] P. Vincent, F. Witherden, B. Vermeire, J. S. Park, A. Iyer, Towards green aviation with python at petascale, in: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 1–11. doi:10.1109/SC.2016.1.

[12] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, G. Gorman, Devito: Towards a generic finite difference dsl using symbolic python, IEEE, 2016, pp. 67–75. doi:10.1109/PyHPC.2016.9.

[13] SBLI Computer Code, http://www.southampton.ac.uk/engineering/research/projects/sbli_computer_code.page.

[14] N. D. Sandham, E. Schlein, A. Wagner, S. Willems, J. Steelant, Transitional shock-wave/boundary-layer interactions in hypersonic flow, Journal of Fluid Mechanics 752 (2014) 349–382. doi:10.1017/jfm.2014.333.

[15] UK Turbulence Consortium, http://www.turbulence.ac.uk/.

[16] CCP12: High Performance Computing in Engineering, http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/J010448/1.

[17] Y. Yao, Z. Shang, J. Castagna, R. Johnstone, L. Jones, J. Redford, R. Sandberg, N. Sandham, V. Suponitsky, N. D. Tullio, Re-engineering a dns code for high-performance computation of turbulent flows, 47th AIAA Aerospace Sciences Meeting and Exhibit, US (2009).
URL https://eprints.soton.ac.uk/185807/

[18] G. N. Coleman, R. D. Sandberg, A Primer On Direct Numerical Simulation of Turbulence - Methods, Procedures and Guidelines, Tech. rep., University of Southampton, UK, school of Engineering Sciences Aerospace Engineering AFM Reports. AFM 09/01a (2010).

[19] OpenSBLI, https://opensbli.github.io/ (2016).

[20] OPS for Many-Core Platforms, http://www.oerc.ox.ac.uk/projects/ops (2014).

[21] HiLeMMS: High-Level Mesoscale Modelling System, https://www.epcc.ed.ac.uk/projects-portfolio/hilemms-high-level-mesoscale-modelling-system (2017).

[22] S. P. Jammy, C. T. Jacobs, N. D. Sandham, Performance evaluation of explicit finite difference algorithms with varying amounts of computational and memory intensity, Journal of Computational Science (2016) –doi:10.1016/j.jocs.2016.10.015.

[23] C. T. Jacobs, S. P. Jammy, N. D. Sandham, OpenSBLI: A Framework for the Automated Derivation and Parallel Execution of Finite Difference Solvers on a Range of Computer Architectures, Journal of Computational Science 18 (2017) 12 – 23. doi:10.1016/j.jocs.2016.11.001.

[24] I. Z. Reguly, G. R. Mudalige, M. B. Giles, Loop tiling in large-scale stencil codes at run-time with ops, IEEE Transactions on Parallel and Distributed Systems 29 (4) (2018) 873–886. doi:10.1109/TPDS.2017.2778161.

[25] Archer at EPCC, https://www.epcc.ed.ac.uk/facilities/archer (2017).

[26] Titan at ORNL, https://www.olcf.ornl.gov/titan/ (2017).

[27] Wilkes2, https://www.top500.org/system/179044 (2017).

[28] OPE Github Repository, https://github.com/gihanmudalige/OPS (2017).

[29] I. Z. Reguly, G. Mudalige, M. B. Giles, OPS C++ users manual, http://www.oerc.ox.ac.uk/sites/default/files/uploads/ProjectFiles/OPS/user.pdf (2017).

[30] I. Reguly, Tutorial: Migrating an application to use OPS, https://op-dsl.github.io/docs/OPS/tutorial.pdf (2018).

[31] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, S. McIntosh-Smith, The ops domain specific abstraction for multi-block structured grid computations, in: 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, 2014, pp. 58–67. doi:10.1109/WOLFHPC.2014.7.

[32] M. Classen, M. Griebl, Automatic code generation for distributed memory architectures in the polytope model, in: Proceedings 20th IEEE International Parallel Distributed Processing Symposium, 2006, pp. 1–7. doi:10.1109/IPDPS.2006.1639500.

[33] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, P. Sadayappan, Effective automatic parallelization of stencil computations, SIGPLAN Not. 42 (6) (2007) 235–244. doi:10.1145/1273442.1250761.
URL http://doi.acm.org/10.1145/1273442.1250761

[34] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, C. E. Leiserson, The pochoir stencil compiler, in: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '11, ACM, New York, NY, USA, 2011, pp. 117–128. doi:10.1145/1989493.1989508.

[35] R. T. Mullapudi, V. Vasista, U. Bondhugula, Polymage: Automatic optimization for image processing pipelines, SIGARCH Comput. Archit. News 43 (1) (2015) 429–443. doi:10.1145/2786763.2694364.

[36] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, ACM, New York, NY, USA, 2013, pp. 519–530. doi:10.1145/2491956.2462176.
URL http://doi.acm.org/10.1145/2491956.2462176

[37] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model, in: International Conference on Compiler Construction (ETAPS CC), 2008.

[38] T. Denniston, S. Kamil, S. Amarasinghe, Distributed halide, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '16, ACM, New York, NY, USA, 2016, pp. 5:1–5:12. doi:10.1145/2851141.2851157.
URL http://doi.acm.org/10.1145/2851141.2851157

[39] D. J. Lusher, S. P. Jammy, N. D. Sandham, Shock-wave/boundary-layer interactions in the automatic source-code generation framework opensbli, Computers and Fluidsdoi:10.1016/j.compfluid.2018.03.081.

[40] C. T. Jacobs, M. Zauner, N. D. Tullio, S. P. Jammy, D. J. Lusher, N. D. Sandham, An error indicator for finite difference methods using spectral techniques with application to aerofoil simulation, Computers and Fluids 168 (2018) 67 – 72. doi:10.1016/j.compfluid.2018.03.065.

[41] M. E. Brachet, D. I. Meiron, S. A. Orszag, B. G. Nickel, R. H. Morf, U. Frisch, Small-scale structure of the taylorgreen vortex, Journal of Fluid Mechanics 130 (1983) 411452. doi:10.1017/S0022112083001159.

[42] J. DeBonis, Solutions of the taylor-green vortex problem using high-resolution explicit finite difference methods, in: 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, Aerospace Sciences Meetings, 2013. doi:10.2514/6.2013-382.

[43] J. R. Bull, A. Jameson, Simulation of the compressible taylor green vortex using high-order flux reconstruction schemes, in: The 7th AIAA Theoretical Fluid Mechanics Conference, AIAA Aviation, 2014. doi:10.2514/6.2014-3210.

[44] I. Z. Reguly, G. R. Mudalige, M. B. Giles, S. Maheswaran, Improving resilience of scientific software through a domain-specific approach, Parallel and Distributed Computing (2019, accepted for publication).

[45] Intel Xeon Processor -E5-2660-v4 @ 2.0GHz (Broadwell), https://ark.intel.com/products/91772/Intel-Xeon-Processor-E5-2660-v4-35M-Cache-2-00-GHz- (Accessed Jan 2019).

[46] Intel Xeon Processor Silver 4116 @ 2.10GHz (SkyLake), https://ark.intel.com/products/120481/Intel-Xeon-Silver-4116-Processor-16-5M-Cache-2-10-GHz- (Accessed Jan 2018).

[47] Assessing IBM's POWER8, Part 1: A Low Level Look at Little Endian, https://www.anandtech.com/show/10435/assessing-ibms-power8-part-1/5 (Accesed Jan 2019).

[48] NVIDIA Tesla P100 GPU, https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf (Accessed Jan 2019).

[49] NVIDIA Tesla V100 GPU, https://images.nvidia.com/content/technologies/volta/pdf/tesla-volta-v100-datasheet-letter-fnl-web.pdf (Accessed Jan 2019).

[50] S. Jammy, C. T. Jacobs, D. J. Lusher, N. D. Sandham, Energy Consumption of Algorithms for Solving the Compressible Navier-Stokes Equations On CPU's, GPU's and KNL's, in: M. Taufer, B. Mohr, J. M. Kunkel (Eds.), 7th European Conference on Computational Fluid Dynamics (ECFD 7), Glasgow, UK, 2018.

[51] I. Z. Reguly, A.-K. Keita, R. Zurob, M. B. Giles, High performance computing on the ibm power8 platform, in: M. Taufer, B. Mohr, J. M. Kunkel (Eds.), High Performance Computing, Springer International Publishing, Cham, 2016, pp. 235–254.

[52] J. D. McCalpin, Memory bandwidth and machine balance in current high performance computers, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (1995) 19–25.

[53] T. Deakin, J. Price, M. Martineau, S. McIntosh-Smith, Gpu-stream v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models, in: ISC Workshops, 2016.

[54] ARCHER - Cray XC30: November 2018 List, https://www.top500.org/system/178188 (Accessed Jan 2019).

[55] Titan - Cray XK7: November 2018 List, https://www.top500.org/system/177975 (Accessed Jan 2019).

[56] Wilkes2, https://www.hpc.cam.ac.uk/systems/wilkes-2 (Accessed Jan 2019).

[57] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, P. Kelly, Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures, 2012

Innovative Parallel Computing, InPar 2012doi:10.1109/InPar.2012.6339594.
URL http://dx.doi.org/10.1109/InPar.2012.6339594

[58] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly, D. Radford, Acceleration of a full-scale industrial cfd application with op2, IEEE Transactions on Parallel and Distributed Systems 27 (5) (2016) 1265–1278. doi:10.1109/TPDS.2015.2453972.

[59] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, T. C. Schulthess, Stella: A domain-specific tool for structured grid methods in weather and climate models, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, ACM, New York, NY, USA, 2015, pp. 41:1–41:12. doi:10.1145/2807591.2807627.

[60] PSyclone Project - GitHub Repository, https://github.com/stfc/PSyclone (2018).

[61] R. D. Hornung, J. A. Keasler, The RAJA portability layer: Overview and status, Tech. rep., Lawrence Livermore National Lab. (LLNL) (9 2014). doi:10.2172/1169830.