2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing

The OPS Domain Specific Abstraction for Multi-Block Structured Grid Computations

István Z. Reguly, Gihan R. Mudalige, Michael B. Giles
Oxford e-Research Centre, University of Oxford,
7 Keble Road, Oxford, UK
Email: {istvan.reguly,gihan.mudalige}@oerc.ox.ac.uk,
mike.giles@maths.ox.ac.uk

Dan Curran, Simon McIntosh-Smith
Department of Computer Science, University of Bristol,
The Merchant Venturers Building, Bristol, UK
Email: dc8147@bristol.ac.uk, simonm@cs.bris.ac.uk

***Abstract*—Code maintainability, performance portability and future proofing are some of the key challenges in this era of rapid change in High Performance Computing. Domain Specific Languages and Active Libraries address these challenges by focusing on a single application domain and providing a high-level programming approach, and then subsequently using domain knowledge to deliver high performance on various hardware.**

In this paper, we introduce the OPS high-level abstraction and active library aimed at multi-block structured grid computations, and discuss some of its key design points; we demonstrate how OPS can be embedded in C/C++ and the API made to look like a traditional library, and how through a combination of simple text manipulation and back-end logic we can enable execution on a diverse range of hardware using different parallel programming approaches.

Relying on the access-execute description of the OPS abstraction, we introduce a number of automated execution techniques that enable distributed memory parallelization, optimization of communication patterns, checkpointing and cache-blocking. Using performance results from CloverLeaf from the Mantevo suite of benchmarks, we demonstrate the utility of OPS.

***Keywords*—Domain Specific Languages, Software Design, Structured Grid, High Performance Computing**

I. Introduction

Increasingly complex multicore CPU systems, as well as recently emerged massively parallel platforms, such as graphical processing units (GPUs) or the Xeon Phi, require more and more hardware-specific knowledge and low-level programming techniques for applications to be able to exploit their full performance potential. Even more importantly various hardware platforms often require very different programming approaches and low-level optimizations, which has prompted a lot of research into the area [1], [2], exploring how different algorithmic classes map to different hardware.

At the same time, most scientific codes are primarily developed by domain scientists, who are often self-educated in programming. While many codes are well designed with software sustainability in mind, in some cases the development strategy is "as soon as it starts working, it's done". In many cases these codes are used for 10-20 years, with new features constantly being added and the code being patched, as new research and methods are integrated - these applications are then heavily relied upon to deliver scientific results, thus the software engineering approach is critical to the extensibility and maintainability of the code. It is increasingly necessary to apply more and more involved optimisations to achieve high performance on modern hardware, but the current approach of porting an application to a specific platform is very expensive in terms of developer effort, and, perhaps more importantly, involves a high amount of risk, because it is not clear which platforms and programming approaches will "win" in the long term. Therefore, the "future proof" design of scientific applications has been receiving increasing attention.

Future proofing software for re-use and longevity is not a new concept in software engineering, there is a general push towards raising the level of abstraction for programming; to be able to design *any* code in a *productive* way and achieve high *performance*. Despite decades of research no language or programming environment exists that would deliver *generality, productivity* and *performance*. Below, we discuss some of the most prominent software design approaches aimed at addressing this challenge.

Classical numerical and software libraries give access to high performance implementations of a set of algorithms, such as MKL [3] or MAGMA [4] to Basic Linear Algebra Subprograms (BLAS), or to a wider set of algorithms, such as PETSc for the solution of PDEs [5]. By only providing a set of building blocks, this approach restricts the kinds of algorithms that can be constructed, furthermore it is very rigid in terms of data structures, because input data is generally expected to be laid out in a particular fashion. Some approaches support high-performance general-purpose computations, but they actually lower the level of abstraction in terms of expressing parallelism, thereby degrading productivity; languages or language extensions like OpenCL [6], CUDA [7] or Cilk Plus [8] expect a very specific, comparatively low-level, programming style. Other approaches aim to compile high-level languages like Python down to modern architectures, such as Copperhead [9].

One of the most promising directions for research is Domain Specific Languages [10] or Active Libraries [11] that exploit knowledge about a specific problem domain, such as structured grids, to provide a high-level abstraction that can be easily used by domain scientists, and at the same time address the aforementioned programming challenges. There is a wide range of DSLs being developed, targeting various problem domains, each using a slightly different programming approach. There are two main classes of DSLs [10], *standalone* ones define an entire new language, and *embedded* ones are embedded in the host language, utilizing its syntax

978-1-4799-7020-9/14 $31.00 © 2014 IEEE
DOI 10.1109/WOLFHPC.2014.7

and compilation tools. Most DSLs for scientific computations are embedded, to various degrees; some rely entirely on the features of the host language, such as templates and object oriented programming, and often appear to be classical software libraries, but most rely on compilation techniques to either directly generate machine code, or to generate source code (via source-to-source translation) that is then fed to a traditional compiler. Active Libraries look like traditional libraries, but use code generation to transform an application code written once to different parallel implementations. The extent to which DSLs modify the host language varies widely: some do not change anything (TIDA [12], OP2 [13]), some add new keywords (STELLA[14]), and most define various new language constructs: Delite [15], Halide [16], Pochoir [17], PATUS [18].

The use of DSLs as a development strategy was previously shown to have significant benefits both for developer productivity and gaining near-optimal performance [19], [20], [21]. However, currently most of these still remain as experimental research projects and have not yet been adopted by a wider HPC community. Partly the reason is a lack of DSLs or high-level frameworks that are actively used at creating production level applications.

In this paper, we further explore the utility of high-level abstraction frameworks, and study our approach to designing and implementing DSLs, and its inherent advantages and drawbacks. We introduce the OPS (Oxford Parallel library for Structured mesh solvers) Domain Specific Active Library targeting the development of parallel multi-block structured mesh applications. The domain of multi-block structured mesh applications can be viewed as an unstructured collection of structured-mesh blocks. It is distinct from the single-block structured mesh and unstructured mesh applications domains which are supported by a number of well established DSLs and active libraries [14], [16], [19], [20]. Thus, the challenges in developing a good abstraction to represent the description and declaration of multi-block problems and their efficient solution on modern massively parallel hardware platforms is unique. More specifically we make the following contributions:

1) We introduce the OPS abstraction and API for multi-block structured grid computations.
2) We present how, through this abstraction, automatic parallelization and data movement can be achieved in shared memory and distributed memory systems
3) We introduce a novel lazy execution scheme and discuss what runtime optimizations this enables.
4) We conclude by discussing the benefits and the drawbacks of our approach to designing a DSL, from the perspective of both domain scientists and computer scientists.

We argue that the OPS abstraction covers a wide range of structured mesh applications and at the same time the assumptions made and our approach to implementing this active library are strong enough that OPS can apply a wide range of optimisations and deliver near-optimal performance. The remainder of the paper presents evidence to support both claims; Section II introduces the OPS abstraction and gives an example of how to use its API. Section III describes how relying on the abstraction it is possible to utilize various parallel programming approaches to execute on different hardware,

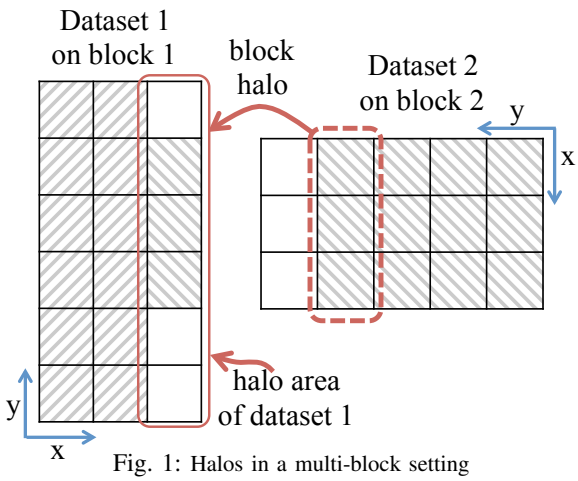


Fig. 1: Halos in a multi-block setting

introduces a lazy execution scheme used by OPS, and describes a number of optimizations that rely on the loop chaining abstraction [22]. Section IV gives an overview of the design choices of OPS discussing their benefits and drawbacks, and finally Section VI draws conclusion.

II. The OPS abstraction and API

The Oxford Parallel library for Structured grid computations (OPS) is a Domain Specific Active Library embedded in C/C++, targeting computations on multi-block structured meshes. The abstraction consists of four principal components:

1) Blocks: a collection of structured grid blocks. These have a dimensionality but no size.
2) Datasets: data defined on blocks, with explicit size.
3) Halos: description of the interface between datasets defined on different blocks.
4) Computations: description of an elemental operation applied to grid points, accessing datasets on a given block.

Given blocks, datasets and halos, an unstructured collection of structured meshes can be fully described. The principal assumption of the OPS abstraction is that the order in which elemental operations are applied to individual grid points during a computation may not change the results, within machine precision (OPS does not enforce bitwise reproducibility). This is the key assumption that enables OPS to parallelize execution using a variety of programming techniques.

From a programming perspective, OPS looks like a traditional software library, with a number of Application Programming Interface (API) calls that facilitate the definition of blocks, datasets and halos, as well as the definition of computations. From the user point of view, using OPS is like 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.

Take for example a simple multi-block scenario, shown in Figure 1, with two blocks, one dataset each that are of size 2×6 and 3×4 respectively, and with one layer of halo. The blocks are oriented differently with respect to each other, and a halo connection is defined between them, as shown in the

```
ops_block block1 = ops_decl_block(2,"block1");
ops_block block2 = ops_decl_block(2,"block2");
int halo_neg[] = {0,0}; int halo_pos[] = {1,0};
int size[] = {2,6}; int base[] = {0,0};
ops_dat dataset1 = ops_decl_dat(block1, 1, size,
                          base, halo_pos, halo_neg,
                          "double","dataset1");
size[0]=3;size[1]=4;halo_pos[0]=0;halo_pos[1]=1;
ops_dat dataset2 = ops_decl_dat(block2, 1, size,
                          base, halo_pos, halo_neg,
                          "double","dataset2");

int halo_iter[] = {3,1};
int base_from[] = {0,3}; int axes_from[] = {0,1};
int base_to[]   = {2,2}; int axes_to[]   = {1,0};
ops_halo halo0 = ops_decl_halo(dataset2,dataset1,
                          base_from, base_to,
                          axes_from, axes_to);
```

Fig. 2: OPS definition of blocks, datasets and halos as shown in Figure 1

figure. Here we briefly introduce the key components of the OPS API and give a sample code that defines the necessary data structures to describe the multi-block situation in Figure 1, using a sequence of OPS calls, as shown in Figure 2. Further details can be found in [23]:

`ops_block ops_decl_block(num_dims,..)` defines a structured block, which will serve to link datasets defined on the same block together.

`ops_dat ops_decl_dat(block,size[],..)` defines a dataset on a specific block with a given size; not all datasets have to have the same size (consider data on cells vs. faces, or a multigrid setup). Ownership of data is transferred to OPS, it may not be accessed directly, only via the `ops_dat` opaque handles.

`ops_halo ops_decl_halo(...)` defines a halo interface between two datasets with arbitrary range and orientation, as illustrated in Figure 1. Currently, this is restricted to a one-to-one matching between grid points.

`void ops_halo_transfer(halos)` triggers the halo exchange between the listed datasets.

`void ops_par_loop( void (*kernel)(...), block, ndim, range[], arg1,..., argN)` defines a parallel loop over a given block with a specific iteration range, applying the *user kernel* `kernel` (a function pointer) to every grid point in the iteration range, passing data pointers, described by the `ops_arg` arguments:

`ops_arg ops_arg_dat(dataset, stencil, type, accss)` gives access to a dataset, passing a pointer to the user kernel that may be dereferenced with the given stencil points, and read, written, read and written, or incremented according to the access specification.

`ops_arg ops_arg_gbl(data, size, type, access)` facilitates passing data to the user kernel that is not defined on any block, such as global variables, and enables global reductions.

```
int range[4] = {12,50,12,50};
for (int j = range[2]; j < range[3]; j++) {
  for (int i = range[0]; i < range[1]; i++) {
    a[j][i] = b[j][i] + b[j+1][i] + b[j][i+1];
  }
}
```

Fig. 3: A classical 2D stencil computation

```
//user kernel
void calc(double *a, const double *b) {
  a[OPS_ACC0(0,0)] = b[OPS_ACC1(0,0)] +
                     b[OPS_ACC1(0,1)] +
                     b[OPS_ACC1(1,0)];
}
...
int range[4] = {12,50,12,50};
ops_par_loop(calc, block, 2, range,
      ops_arg_dat(a,S2D_0,"double",OPS_WRITE),
      ops_arg_dat(b,S2D_1,"double",OPS_READ));
```

Fig. 4: A parallel loop defined using the OPS API

Take for example a classic nested loop performing a stencil operation as shown in Figure 3. The description of this operation using the OPS API is shown in Figure 4; it defines an iteration over the grid points specified by `range`, executing the user kernel `calc` on each, passing pointers to datasets `a` and `b`, `a` is written using a one-point stencil and `b` is read, using a three point stencil - these stencils are described by the data structures `S2D_0` and `S2D_1` respectively. The `OPS_ACC` macros are used to compute the index offsets required to access the different stencil points, these are set up by OPS automatically.

An application implemented once using the above API can be immediately compiled using a common C++ compiler (such as GNU g++ or Intel icpc), and tested for accuracy and correctness - this is facilitated by a header file that provides a single-threaded implementation of the parallel loops and the halo exchanges.

The high-level application code is built to rely entirely on the OPS API to carry out computations and to access data; after an initial setup phase where data is passed to OPS using either existing pointers or HDF5 files, OPS takes ownership of all data, and it may only be accessed via API calls. This enables OPS to make transformations to data structures that facilitate efficient parallel execution.

This abstraction and API can be viewed as an instantiation of the AEcute (Access-Execute descriptor) programming model [24] that separates the abstract definition of a computation from how it is executed and how it accesses data, this in turn gives OPS the opportunity to apply powerful optimisations and re-organize execution. The basic semantics and rules of execution are as follows:

1) For any given `ops_par_loop`, the order in which grid points are executed may be arbitrary.
2) Subsequent parallel loops over the same block respect data dependencies (e.g. one loop writes data that the other reads).
3) Subsequent parallel loops over different blocks do not have a fixed order of execution.
4) Only halo exchanges between blocks introduce a

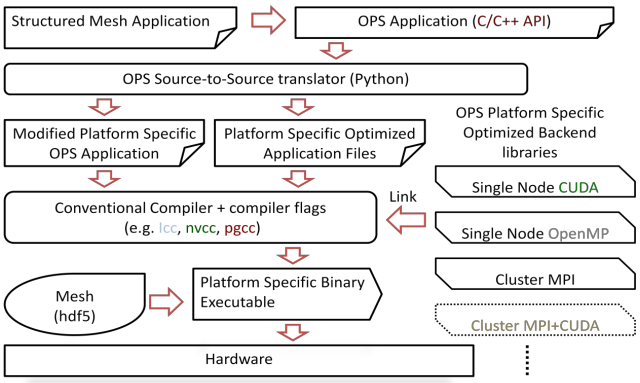


Fig. 5: OPS code generation and build process

data dependency and therefore a prescribed order of execution between blocks.

In this paper, we argue that a wide range of structured mesh applications can be implemented using this API, which we demonstrate with two industrially representative codes, CloverLeaf [25] and ROTORSIM [26]. Furthermore, we claim that the above API is expressive enough that no language extensions or custom compilers are required to enable execution on a variety of hardware platforms, utilizing a range of different parallel programming abstractions and environments. Finally, we demonstrate near-optimal performance matching or outperforming the hand-coded original, substantiating these claims.

III. What the abstraction lets us do

The principal idea of the OPS abstraction and API is the separation of the abstract definition of computations from their parallel implementation and execution. Because computations are described in such a descriptive manner, explicitly stating how and what data is accessed, we show that it is possible to dynamically organise parallelism and data movement, tailoring it to different target architectures. While input data structures are fixed, based on their description OPS can apply transformations to have them better suited for different hardware. OPS uses two fundamental techniques in combination to utilise different parallel programming environments and to organize execution and data movement; code generation and back-end logic. Code generation is used to produce the parallel implementation of individual parallel loops targeted at different hardware architectures and programming languages, and back-end logic is used to factor out common operations, and to organize execution and communication in a way that satisfies data dependencies and improves parallelism, locality or resilience. Figure 5 gives an overview of the OPS workflow; a structured mesh application is implemented using the OPS C/C++ API, handed to the code generator which creates the platform specific optimized implementations. This can then be compiled using conventional compilers, such as icc, gcc or nvcc, and linked against the platform specific back-end. The resulting executable may use the built-in capabilities of OPS to read HDF5 files in parallel, and is executed on the target architecture(s).

A. Code generation

Although an OPS application can be immediately compiled, the header file implementation of `ops_par_loop` is very general, and prohibits a number of optimizations that a compiler would carry out on hand-coded nested `for` loops with stencil computations, such as the one shown in Figure 3. However, a key point in designing the OPS API was to supply all the necessary information that could be used to reconstruct this formulation, thereby enabling the same, if not further, compiler optimizations.

```
#define OPS_ACC0(j,i) j*xdim0+i
#define OPS_ACC1(j,i) j*xdim1+i

//user kernel
void calc(double *a, const double *b) {...}

void ops_par_loop_calc(int ndim, int range,
                       ops_arg arg0, ops_arg arg1){
//set up pointers and strides
double *p_a0 = (double*)ops_base_ptr(range, arg0);
double *p_a1 = (double*)ops_base_ptr(range, arg1);
xim0 = arg0.dat->size[0]; xim1 = arg1.dat->size[0];
//do the computation
for(int j = 0; j < range[3]-range[2]; j++) {
  for(int i = 0; i < range[1]-range[0]; i++) {
    calc(&p_a0[j*xdim0+i],&p_a1[j*xdim1+i]);
  }
}
```

Fig. 6: A simple example of code generated by OPS

Figure 6 shows a simple example of code generated for sequential execution of the loop in Figure 4; observe that all the information required to generate this code is present in the body of the `ops_par_loop` call. This is a key property of the OPS API: it is sufficient to extract information from the `ops_par_loop` call, therefore we have chosen not to implement a full-fledged compiler system, rather to have a simple python script that looks for parallel loop calls in the source code and assembles data structures that contain all the information about the individual loops,

Using code generation, OPS targets optimized single threaded execution of loop nests as well as different shared-memory parallel execution techniques, such as OpenMP, OpenACC, CUDA and OpenCL. Because these programming approaches benefit greatly from compiler optimizations, the code generated is as specific to the given loop as possible, and it also includes a number of optimizations that were developed experimentally using one-off hand-coded implementations and then generalized for the code generator. Below, we list some of the main properties of code generated for different platforms:

1) Based on the iteration range, the size and halos of datasets, base pointers are calculated for each argument.
2) Based on the stencil stride, the iteration range and the properties of the dataset, pointer offsets are calculated that can be applied to the base pointer when iterating over the mesh.
3) Single-threaded execution: a nested loop is generated, with pointers being incremented at every level and passed to the user kernel. To achieve auto-vectorization, the innermost loop is sectioned (or strip-mined).
4) OpenMP: the outermost loop is split into equal partitions and picked up by the individual threads, the inner loops are implemented the same way as

for single-threaded execution. Partial reductions for each thread are set up (the only possible carried dependence).

5) OpenACC: a simple nested loop is generated with explicit pointer computations as opposed to pointer increments, and the appropriate pragmas are added.
6) CUDA and OpenCL: a grid is launched and a single thread (or work-item) assigned to each grid point, calculating the appropriate offsets and passing the pointers to the user kernel. All data movement is managed by the back-end.

As new programming models and hardware are released, it is not necessary to alter the high-level user code; rather it is sufficient to adapt the code generator to support the new language or to use optimization techniques specific to new generations of hardware; a good example would be the use of caching loads on NVIDIA Kepler generation GPUs for loading read-only data. Because the code generators only use simple text manipulation, it is almost trivial to add new optimizations and then deploy those optimizations on large-scale codes - something that may require going through the entire user code by hand in case of applications that do not use a DSL. OPS generates a separate executable for each combination of back-ends, it is up to the user to determine which one is appropriate for a given system.

B. Back-end logic

While code generation is necessary to enable low-level optimizations that affect each iteration of a nested loop, there are a number of techniques that are applied at a much higher level: the granularity of individual parallel loops. This is possible, because `ops_par_loop` calls represent operations over blocks in an atomic way, and provide a lot of information that can be used to reason about execution patterns. Assuming that the blocks have a reasonable size, the overhead of added logic before or after the actual computations is negligible, but it lets OPS apply radical changes to execution.

Functionality implemented in the back-end ranges from simple data management in heterogenous systems, through enabling distributed memory execution, to complex changes to execution patterns facilitated by lazy execution. Here, we present an informal, high-level discussion of these, illustrated with simple examples, but do not go into any implementation details, as those are often very convoluted to support arbitrary structured mesh computations. The the algorithms described in this section are all implemented and deployed in the trunk of the OPS repository [23].

1) Distributed memory parallelism: The most fundamental feature is distributed memory execution; OPS can automatically distribute a collection of structured blocks as well as individual blocks across MPI processes. When decomposing individual blocks, using the knowledge about iteration ranges and access patterns, it can satisfy data dependencies by means of *ghost cells* and automatic intra-block halo exchanges - note the difference from inter-block halo exchanges, which are triggered explicitly using an API call.

Given a number of datasets, OPS will partition a structured block among a number of MPI processes and allocate the ghost cells in a way that aims to minimize communication requirements and achieve good load balance. Communication

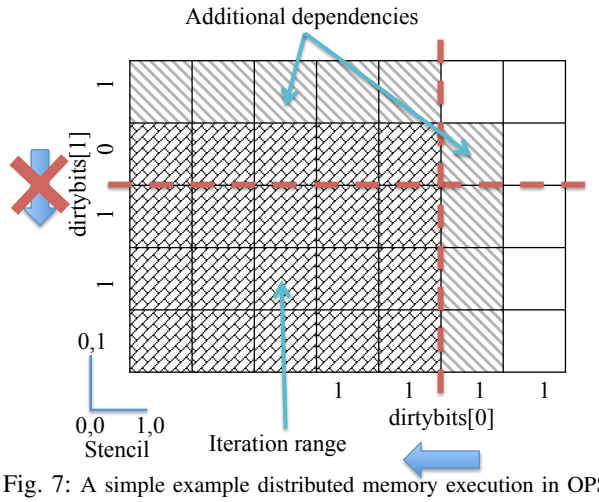


Fig. 7: A simple example distributed memory execution in OPS

is set up using MPI's Cartesian topology, optionally using MPI datatypes to send and receive non-contiguous data along the faces - albeit for some MPI distributions we have noticed poor performance compared to a hand-coded implementation of the same functionality. Our distributed memory execution strategy uses the traditional approach to satisfying data dependencies in structured mesh computations: on-demand messages exchanged before computations that need them, While in many existing codes exchanges need to be triggered explicitly, and with a user-specified depth (number of ghost cell layers, based on stencil size), OPS can reason about data dependencies based on information provided to the abstraction, and it can exchange the minimum amount of data required.

In OPS, each dataset has *dirty bit* fields in each dimension, direction and for different depths from the boundary; these keep track of what regions of the data were modified. At the beginning of each parallel loop, based on the access patterns of datasets, the iteration range, and the partitions held by different processes, each MPI rank can compute the number of halo layers that it needs to send or receive to satisfy data dependencies. After the computations, given the iteration range and a list of datasets modified, the dirty-bits are updated. Figure 7 illustrates various aspects of this logic; the execution range may not cover all MPI processes, but given a 3-point stencil, communication may still be necessary (indicated by the large arrows at the sides), but only if ghost cell data is outdated (dirty). Furthermore, in many cases, stencils are not symmetric, therefore bidirectional exchange may not be necessary. To reduce the number of MPI messages, halo data from different datasets in the same loop are aggregated and sent together.

2) Checkpointing and recovery: To ensure the resiliency of large-scale simulations, many codes implement some form of checkpointing; this consists of periodically saving the state space to disk, and in the event of a failure re-launching the application, fast-forwarding to the last checkpoint, restoring the state space and continuing as normal. Manual implementation of checkpointing is often very tedious, especially when the state space is large, as it is difficult to reason about what data needs to be saved and what doesn't. Due to the fact that OPS takes ownership of all data, and that data "leaving" the realm of OPS only happens through API calls (such as

reductions, the results of which might be used to alter control flow), it is possible for us to keep track of what, when and how data is modified, and therefore to reason about the state space. The fundamental observation behind our checkpointing strategy is that if a dataset is overwritten immediately after the checkpoint, then that dataset does not need to be saved. The question therefore becomes: when to create a checkpoint, and out of all datasets defined, which ones to save.

The execution of an application from an OPS point of view essentially comes down to a sequence of parallel loop calls, each of which read certain datasets and write others. However, any given loop usually only accesses a small subset of all datasets, therefore reasoning about the state space at any particular parallel loop, given the data it accesses, is not sufficient; this leads to the introduction of "checkpointing regions": the beginning of the region is the location of the checkpoint in the classical sense, but the actual process spans several subsequent parallel loops. The only modification in the user code is a call to the OPS API during initialization that specifies the checkpointing frequency. During execution, OPS will save the value of global reductions for each loop that includes one, and when a timer triggers checkpointing, it will automatically find the next entry into a "checkpointing region" and execute the algorithm below, saving data to a HDF5 file.

A high-level description of the algorithm is as follows:

1) If a dataset was never modified (as might be the case with e.g. mesh coordinates), then it is not saved at all.
2) The results of global reductions in loops are saved for every occurrence of the loop because data returned after a loop is out of the hands of OPS, and may be used for control decisions.
3) When checkpoint creation is triggered, then enter "checkpointing region" upon reaching the first parallel loop, and before executing that loop:
    a) Save datasets accessed that are not write-only.
    b) Drop datasets that are write-only in the loop from the checkpoint.
    c) Flag all other datasets that are not used by this loop, but do not save them yet.
4) When already in a "checkpointing region" (previous point), start executing subsequent loops to determine whether datasets that were not yet saved nor dropped (i.e. are flagged) would have to be saved:
    a) If a flagged dataset is encountered, save it if it's not write-only, otherwise drop it and remove the flag.
    b) If a flagged dataset is not encountered within a reasonable timeframe allocated for the "checkpointing region", then save it.

In the event of a failure, the application needs to be restarted, and if a checkpoint file is found then "restore mode" is enabled, during which calls to `ops_par_loop` do not carry out any computations, just set the value of reduction variables. Once the location of the last checkpoint is reached, the state space is restored from the HDF5 file, "restore mode" ends, and execution returns to normal.

One of the key challenges is deciding where exactly to enter the "checkpointing region" so that the state space that has to be saved is minimal; entering it at the first loop that has a write-only dataset may only be locally optimal. There are several possible algorithms that can help predict when it is optimal to enter checkpointing mode, based on the observation that most simulations have periodic execution patterns, however we have chosen to utilize our lazy execution scheme that gives OPS a view of a sequence of future parallel loops to choose from, details are described in Section III-C4.

3) Lazy execution: Following the same reasoning as in the previous section, it is easy to see that if all the data is owned by OPS and any user access to it can only happen through OPS, then an `ops_par_loop` without reductions does not have to be executed immediately before execution proceeds to the next instruction in the user code (i.e. synchronously). This makes it possible to delay execution of a sequence of parallel loops until some data has to be returned to the user, typically the result of a reduction.

Lazy or delayed execution is easily introduced in OPS; `ops_par_loop` calls create a data structure, storing all the arguments as well as a function pointer to the implementation of the actual computations, then this structure is stored in a queue before returning. The enqueueing process is implemented in the back-end; when a parallel loop with a reduction is encountered, execution is triggered; OPS loops over the sequence of kernels, and uses the function pointer stored in the data structure to invoke the actual computations.

C. Advanced optimizations

Knowing the details of a sequence of operations without having executed them yet enables OPS to carry out analysis and apply optimisations across a number of parallel loops and to reason about data dependencies and data movement at a much higher level; essentially relying on the *loop chaining abstraction* [22]. In subsequent sections we present a number of optimization opportunities that the loop chaining scheme enables OPS to apply - these have not yet been fully implemented in the OPS repository.

1) MPI messaging: The way of OPS keeping track of changes to datasets helps minimize the number and size of messages - but the method of on-demand messaging as described in Section III-B1 still results in a large number of messages, each of which suffers from the latency of MPI communications. By analysing a sequence of parallel loops and mapping out data dependencies between them, it is possible to aggregate messages across multiple parallel loops if not all data produced is consumed immediately in the following loop.

The sequence of parallel loops queued can be described with an ordered graph $G = (V, E)$, where vertices represent parallel loops and there exists a special vertex v_0 that represents the state space at the beginning of the queue. A directed edge $e_{i,j,k}$ exists between two vertices i and $j, i < j$ if i modifies dataset k that j reads in a way that requires a halo exchange (i.e. iteration ranges overlap and the stencil is non-trivial). Any dataset that was modified before the first loop in the queue is considered to be changed by loop 0, any dependency in subsequent loops gives an edge from v_0. Clearly, before loop j can be executed, all data dependencies represented by edges $e_{i,j,k}, i < j$ have to be satisfied by way

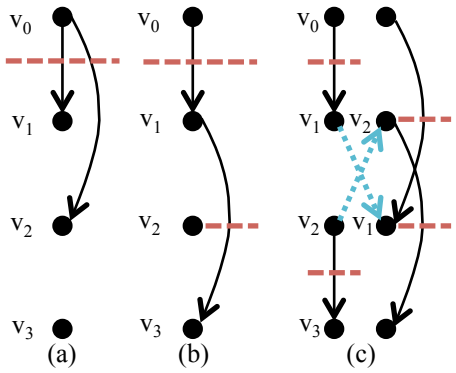


Fig. 8: MPI messaging optimization strategies

of a halo exchange - something we shall refer to as *cutting* edge $e_{i,j,k}$.

The most trivial optimization is that when a halo exchange is triggered, cut as many edges as possible, thereby aggregating messages together to reduce latency. *Strategy 1*: for a given loop j, if $\exists e_{i,j,k} i < j$, then cut all previously uncut edges $e_{m,n,k}, m < j, n \geq j$. This strategy makes it possible for subsequent loops to have their data dependencies satisfied before execution reaches them, thereby eliminating messaging latency that the classical on-demand exchanges introduce. Figure 8(a) shows a simple example where it is advantageous to aggregate two messages that would otherwise be sent one-by-one.

The second strategy, potentially used in conjunction with *Strategy 1* is to use computations, the execution of loops, to hide the latency of messaging; commonly referred to as latency hiding. However, unlike classical latency hiding strategies which require the separation of execution of a single parallel loop into interior and boundary parts (by way of peeling), *Strategy 2* hides the messaging latency that has to be completed before executing loop j with the execution of loop $i, i < j$. Thus, before initiating execution of loop i, if $\exists e_{n,j,k}, n < i, j > i$, then that edge may be cut, and asynchronous communications can be started, only to be finished before the execution of loop j. This may be a good strategy even if there exists a dependency between loops i and j, because smaller messages have lower latency. Figure 8(b) shows a simple example where communication may be overlapped with execution; v_3 may be executed concurrently with the halo exchange required for the execution of v_3.

Finally, if no direct or indirect dependency exists between two loops i and $j, i < j$, then the execution of the two loops may be interchanged; *Strategy 3*. This strategy is most useful in combination with the above two strategies, if for example there exists a dependency between loop i and $i+1$, but no dependency between $i-1$ and either i or $i+1$, then i may be executed first, and the communication latency between i and $i+1$ can be hidden by the execution of $i-1$. Figure 8(c) shows a simple example where by exchanging the execution of loops v_1 and v_2, latency hiding becomes possible.

The optimal combination of these optimization strategies for any given sequence of parallel loops is a subject of intense research, but out of scope of this paper.

2) Communication avoidance: While the above MPI messaging strategies help minimize the number of messages and hide their latency, the total amount of data that needs to be exchanged remains the same, and if a number of subsequent parallel loops have back-to-back data dependencies, then the incurred communication costs are still significant. Using the lazy execution technique, it is possible for OPS to automatically apply a communication avoiding execution scheme that uses redundant computations along partition boundaries - similar to [27], [28]. This technique relies on having a sequence of parallel loops, and then it works backwards, extending iteration ranges to satisfy data dependencies between parallel loops. An overview of the algorithm, given a sequence of loops $i = 1..N$ is as follows:

1) For loop N, record the dependency range (iteration range extended by stencil widths) for each dataset that is read (or incremented, or read-and-written).
2) For each loop $i = N-1...1$:
   a) Take the union of the iteration range and the recorded data dependency ranges of all datasets that are modified by loop i, and set it as the new iteration range of loop i.
   b) Remove recorded dependency ranges for datasets that are write-only in loop i.
   c) Record the dependency ranges of all datasets read by loop i, by extending the new iteration range by the stencil widths.
3) Before loop 1, exchange extended halos of all datasets with a recorded dependency range.
4) Execute loops $1...N$ using their extended iteration range. No communication is necessary.

The above algorithm uses redundant computations to resolve all data dependencies, therefore the results of the computations are going to be the same; the only issue that would arise is the result of reductions when computing on extended iteration ranges - however the lazy execution scheme's list of parallel loops may only have a reduction loop at the very end, and the iteration range of the last loop is not extended, therefore this is not an issue.

This optimization can be applied to any sequence of loops, but in practice it may not be advantageous to do so for a long sequence, as it may extend data dependency ranges and iteration ranges to the extent that the initial communication costs and the fraction of redundant computations to useful computations become too large. Therefore it is an important optimization challenge to find the optimal sequence of loops. In general this is a difficult problem, but in practice many codes are structured in a way that in some regions there are back-to-back dependencies between a number of loops. Often, these are preceded and followed by other loops where this is not the case; therefore restricting the application of this algorithm to these regions and combining it with the latency hiding strategy described in the previous section is a good heuristic.

3) Tiling: Most scientific computations are structured so that operations are carried out on the entire domain, followed by the next operation carried out again on the entire domain. This often means accessing the same data again and again, however, if the size of the datasets is larger than the size of the on-chip cache, then this data will be moved in from

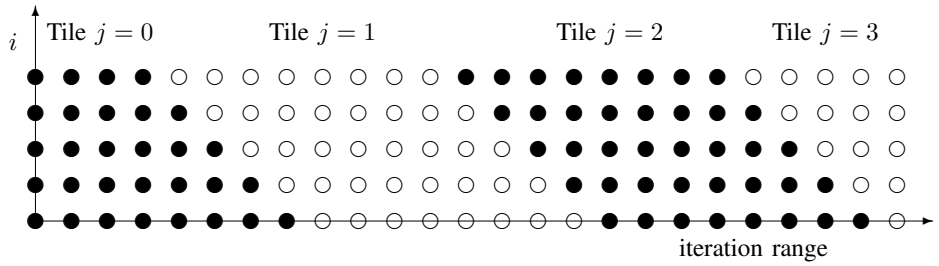


Fig. 9: An illustration of cache-blocking, or tiling, to reduce data movement. Tile 0 is executed first, then 1, 2 and 3

main memory repeatedly - an extremely expensive operation compared to accessing data in cache. The underlying idea of the communication avoiding algorithm naturally leads to the algorithm commonly referred to as tiling or cache blocking; partitioning the computational domain into smaller pieces that can fit in the cache. Tiling is the target of intense research [29], [17], although most publications only consider the case where the same stencil is applied repeatedly to the same data, which in practice is rarely the case. Furthermore, most of them use various compilation techniques which may struggle with a sequence of complex loops, especially if various parameters are unknown at compile time.

By using lazy execution, OPS has a lot of run-time information available, such as iteration ranges, datasets accessed and stencils, which makes it much easier to reason about data dependencies. Figure 9 illustrates the idea of tiling with a simple 1D example and a 3-point stencil; given the execution on a number of grid points (first row), data dependencies are then resolved for a subset of those grid points for the next iteration, etc. We have chosen skewed tiling for OPS as it does not require redundant computations or the redundant storage of data, the high-level overview of the algorithm for a sequence of loops $i = 1..N$ is as follows for a 1D problem:

1) Create K tiles of the full iteration range (number and size of tiles is an optimization parameter), and construct a data structure holding iteration range for loops $i = 1..N$, as well as data dependency range for any dataset modified during these loops
2) Initialize the data dependency range of each dataset in each tile by first computing the intersection of the tile's iteration range with the full iteration range of loops that modified the dataset, and then taking the union of these.
3) For tile $j = 1..K$, iterate over loops in reverse order $i = N...1$:
   a) Take the union of data dependency ranges of datasets that loop i modifies in tile (j), minus the intersection with the iteration range of loop i in tile $(j-1)$, and set it as the iteration range of loop i for tile (j)
   b) For datasets that are read by loop i in tile (j), set the new data dependency range as the iteration range, extended by the stencil used to access the datasets, but not beyond the original size of the dataset.
4) Execute tiles in-order, calling loops with the iteration range specific to the tile-loop combination, as computed above.

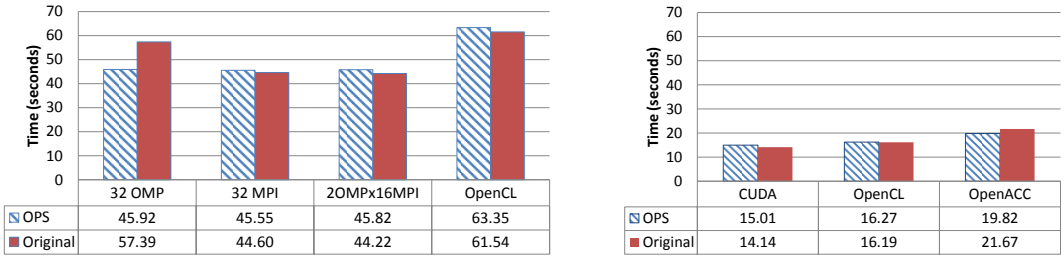
The above algorithm is described for 1D problems, but is trivially extendable for higher dimensions. For the simple example in Figure 9 the algorithm will construct tiles 1 through 3, and then execute them in the order 0-1-2-3. The optimization of the size and number of tiles depends on the number of loops tiled over, the number of datasets accessed and the size of the on-chip cache. The tile construction algorithm, while not particularly expensive, can take a while to construct especially for small tiles, therefore in OPS, we cache the tile execution plans and re-use them when the same sequence of loops is encountered.

4) Optimal checkpointing: As discussed in Section III-B2, it is easy to find a locally optimal checkpoint location, however in order to globally minimize the amount of data that needs to be saved, it is necessary to find a regularly occurring point during execution where entering checkpointing mode results in the least amount of data saved. By utilizing lazy execution, it is possible to reason about state space not only locally, but over a sequence of parallel loops. Therefore, the decision can be improved by calculating the amount of data that would be saved if checkpointing started at any of the loops in the sequence, and a database is built based on this information that records the loop at which checkpointing mode is entered, the subsequent list of loops and the amount of data to be saved. Later on, when it is time to do the actual checkpoint, one of the most frequently occurring checkpointing locations with the smallest size is chosen by matching the sequence of loops in the record and the lazy execution queue.

IV. Summary of the OPS approach

We have presented the OPS abstraction and API, which covers a variety of use cases, however it does have some restrictions. We already have a number of preliminary abstraction and API designs that could tackle multigrid situations where two datasets are accessed in the same loop with a different resolution, as well as sliding planes where the connectivity between datasets in different blocks may change over time (due to e.g. rotating geometries). These will naturally extend the existing API without changes to existing functionality, and we believe this will be possible for future extensions as well, such as non-matching dataset interfaces (where interpolation is necessary) or support for linear solvers.

Our choice to conform fully to the C standard (plus templates) in the API design is motivated by the fact that as we have shown it is not necessary to implement and maintain a compiler. Indeed, one of the main challenges in the adoption of Domain Specific Languages is the uncertainty about who is going to support and maintain them in the long-term. By only needing limited code parsing and text manipulation capabilities it is possible to keep the code generation part of OPS relatively simple; this is attractive to our academic



(a) Intel Xeon E5-2680 (dual socket), 32 HyperThreads

(b) NVIDIA K20 GPU

Fig. 10: CloverLeaf performance, comparing various original hand-coded and tuned implementations with the auto-generated implementations produced by OPS (lower is better)

and industrial partners who may not have the resources and expertise to maintain a complex compiler system. At the same time, we are aware that this somewhat restricts the range of possible optimizations we can apply; a good example is the use of different parallel programming techniques. While our current approach works well with SMP (Simultaneous Multiprocessing, e.g. OpenMP) and SIMT (Single Instruction Multiple Threads, e.g. CUDA) programming abstractions, it struggles with SIMD (Single Instruction Multiple Data, e.g. AVX vector intrinsics) - currently we have to rely on the compiler's ability to auto-vectorize the innermost loop; so far with generally positive results. We have explored the use of vector classes to wrap vector intrinsics before [30], [31], but currently there is no reliable way of modifying user code to use them (branching is particularly problematic). Furthermore, as we shown in the next section, our implementations closely match or outperform the hand-coded and tuned original.

While our approach does require a large-scale refactorization of existing codes, we believe that some form of restructuring is ultimately going to be necessary for future-proofing scientific codes anyway, and that doing this for OPS this is no more difficult than a one-off implementation for a particular parallel programming environment.

V. Performance Results

Here we briefly present performance results from one of the applications that we ported to use OPS; CloverLeaf, a single block 2D hydrodynamics code, which is part of the Mantevo suite [32] and is an unclassified code. Several publications discussed the performance of Cloverleaf and presented hand-coded implementations that use MPI, OpenMP, OpenMP v4, OpenACC, CUDA and OpenCL [33]; these serve as a performance baseline to compare our implementations against. One of the advantages of using OPS is immediately obvious, just by looking at the code; the original code's online repository [34] has 11 different versions of Cloverleaf, whereas OPS only has one [23] - code maintenance is therefore much easier.

We present the total runtime of the hydro loop of CloverLeaf for the 3840 $\times$ 3840 (`clover_bm16_short.in`) mesh input deck, to be consistent with the compiler flags recommended for gaining accurate results from the original CloverLeaf application, we enforce IEEE floating-point mathematics compliance on each compiler and benchmarks. Figure 10 shows performance on an CPU server, with a pair of high-end Intel Xeon E5-2680 processors (Intel Compiler 14.0 targeting AVX) as well as performance on an NVIDIA K20 GPU (CUDA 6.0); clearly, the use of a high-level approach such as OPS is not detrimental to performance - in itself result of great importance. Furthermore, in some cases it can even outperform the original hand-coded implementation, due to the better handling of factors such as NUMA [35] effects. There exists now a 3D version of CloverLeaf; its porting to OPS took three man-days with validation, followed by two man-days of extending the code generators and some back-end functionality to handle 3D cases. We speculate that it would take much longer to produce the various hand-coded parallel implementations, each of which would have to be then debugged and validated, but by using OPS, the different parallel implementations are now a single click away.

VI. Conclusions

We have presented an embedded Domain Specific Active Library and abstraction for computations on multi-block structured grids, named OPS. We have shown that given the right abstraction, an Application Programming Interface can be designed that looks like a regular software library and conforms to C/C++ standards, but at the same time it enables execution on a wide range of diverse hardware architectures, and the application of various low-level and high-level optimizations through code generation and back-end logic. We have shown that code generation for shared-memory parallelization techniques, such as OpenMP or CUDA, in combination with completely opaque distributed memory execution is capable of matching or outperforming hand-coded and tuned implementations, which gives us considerable confidence in our approach being capable of delivering high performance, code maintainability and future proofing to structured grid applications. Furthermore, by introducing a number of high-level optimizations that improve communications, locality and resilience, we have shown that these techniques, which given current hardware trends are likely to become increasingly important, can be easily applied through the use of DSLs.

Acknowledgments

The OPS project is funded by the UK Engineering and Physical Sciences Research Council projects EP/K038494/1, EP/K038486/1, EP/K038451/1 and EP/K038567/1 on "Future-proof massively-parallel execution of multi-block applications" and EP/J010553/1 "Software for Emerging Architectures" (ASEArch) project.

References

[1] W.-m. W. Hwu, *GPU Computing Gems Jade Edition*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[2] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[3] Intel, "Math kernel library," http://developer.intel.com/software/products/mkl/.

[4] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[5] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2014. [Online]. Available: http://www.mcs.anl.gov/petsc

[6] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[7] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[8] *Intel Cilk Plus Language Specification*, 2010. [Online]. Available: https://www.cilkplus.org/sites/default/files/open_specifications/cilk_plus_language_specification_0_9.pdf

[9] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an embedded data parallel language," EECS Department, University of California, Berkeley, Tech. Rep., Sep 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-124.html

[10] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[11] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen, "Generative programming and active libraries," in *Selected Papers from the International Seminar on Generic Programming*. London, UK, UK: Springer-Verlag, 2000, pp. 25–39. [Online]. Available: http://dl.acm.org/citation.cfm?id=647373.724187

[12] W. Z. J. B. Didem Unat, Cy Chan and J. Shalf, "Tiling as a durable abstraction for parallelism and data locality," in *Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, Nov. 2013.

[13] M. B. Giles, G. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly, "Performance analysis and optimization of the OP2 framework on many-core architectures," *The Computer Journal*, vol. 55, no. 2, pp. 168–180, 2012.

[14] T. Gysi, O. Fuhrer, C. Osuna, B. Cumming, and T. Schulthess, "Stella: A domain-specific embedded language for stencil codes on structured grids," in *EGU General Assembly Conference Abstracts*, vol. 16, 2014, p. 8464.

[15] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 4s, p. 134, 2014.

[16] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and 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*, ser. PLDI '13, 2013.

[17] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11, 2011. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508

[18] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 676–687.

[19] I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly, and D. Radford, "Acceleration of a Full-scale Industrial CFD Application with OP2," *submitted to ACM Transactions on Parallel Computing*, 2013, available at: http://arxiv.org/abs/1403.7209.

[20] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: a domain specific language for building portable mesh-based pde solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 9:1–9:12.

[21] O. Fuhrer, T. Gysi, and T. Schulthess, "Towards a domain specific library for stencil methods on grids," 2014, http://www.cscs.ch/fileadmin/SOS/SOS18_presentations/Oliver_Fuhrer.pdf.

[22] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. J. Kelly, G. R. Mudalige, B. van Straalen, and S. Williams, "Loop chaining: A programming abstraction for balancing locality and parallelism." in *IPDPS Workshops*. IEEE, 2013.

[23] "OPS github repository," 2013, https://github.com/gihanmudalige/OPS.

[24] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly, "Deriving efficient data movement from decoupled access/execute specifications," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC '09. Springer-Verlag, 2009, pp. 168–182.

[25] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the performance portability of structured grid codes on many-core computer architectures," in *Supercomputing*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8488.

[26] C. B. Allen, "An unsteady multiblock multigrid scheme for lifting forward flight rotor simulation," *International Journal for Numerical Methods in Fluids*, vol. 45, no. 9, pp. 973–984, 2004. [Online]. Available: http://dx.doi.org/10.1002/fld.711

[27] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 207–218. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259044

[28] L. Chen, Z.-Q. Zhang, and X.-B. Feng, "Redundant computation partition on distributed-memory systems," in *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, Oct 2002, pp. 252–260.

[29] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/106972.106981

[30] I. Z. Reguly, E. László, G. R. Mudalige, and M. B. Giles, "Vectorizing unstructured mesh computations for many-core architectures," in *Proceedings of Programming Models and Applications on Multicores and Manycores*, ser. PMAM'14, 2007.

[31] X. Huo, B. Ren, and G. Agrawal, "A programming system for xeon phis with runtime simd parallelization," in *Proceedings of the 28th ACM International Conference on Supercomputing*, ser. ICS '14, 2014. [Online]. Available: http://doi.acm.org/10.1145/2597652.2597682

[32] R. F. Barrett, M. A. Heroux, P. T. Lin, C. T. Vaughan, and A. B. Williams, "Poster: Mini-applications: Vehicles for co-design," in *Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion*, ser. SC '11 Companion. New York, NY, USA: ACM, 2011, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/2148600.2148602

[33] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis, "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 465–471.

[34] "CloverLeaf github repository," 2013, https://github.com/Warwick-PCAV.

[35] R. P. L. Jr. and C. S. Ellis, "Page placement policies for NUMA multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 11, no. 2, pp. 112 – 129, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/074373159190117R