Design and Development of Domain Specific Active Libraries with Proxy Applications

I.Z. Reguly[†*], G.R. Mudalige[†], M.B. Giles[†]
[†]Oxford e-Research Centre, University of Oxford, United Kingdom
[*]Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Hungary
Email: {istvan.reguly,gihan.mudalige}@oerc.ox.ac.uk, mike.giles@maths.ox.ac.uk

***Abstract*—Representative applications are versatile tools to evaluate new programming approaches, techniques and optimisations as a way to ensure continued high performance on future computing architectures. They make experimentation much easier before adopting changes/insights into the large scientific codes. In this paper we demonstrate the important role played by representative/proxy applications in designing and developing two high-level programming approaches: namely the OP2 and OPS domain specific (active) libraries. OP2 and OPS utilizes code generation techniques to produce automatic parallelisations from a high-level abstract problem declaration. The strategy delivers significant developer productivity to the domain scientist, while at the same time allowing computational experts to adopt the latest programming models and hardware-specific optimisations into the library and code generation tools to achieve near optimal performance. We show how representative applications have been a cornerstone in the development of OP2 and OPS and chart our experiences. In particular, we demonstrate how the range of hand-tuned optimized parallelisations of the CloverLeaf hydrodynamics mini-app allowed us to gain clear evidence that the OPS based code generated parallelisations were indeed as optimal as the hand-tuned versions. Additionally, with the use of a representative application from the CFD domain we demonstrate how the optimisations discovered and applied to proxy apps are indeed directly transferable to a large-scale industrial application at Rolls Royce plc. These results provide significant evidence into the utility of representative applications to improve productivity, enable performance portability and ultimately future-proof scientific applications.**

I. Introduction

While life-cycles of computer hardware are usually on the order of a few years, large-scale software, particularly numerical simulation programs, are often used for decades. They can easily grow to be several millions of lines of code in size, making them difficult to optimize and maintain. Compounding this issue is the evolution of computer architectures, as we head towards the exascale era. Current and emerging systems have features that require the utilisation of an ever-increasing level of parallelism, making it difficult to program for high performance. Fortran and C/C++, the most popular languages used in these numerical simulation programs, were designed with single-core scalar processors and a simple memory model in mind. However, now we are very much stretching their limits: these programming models aren't explicit enough about a number of factors important for parallelization, such as aliasing rules and independence of loop iterations, and these often prevent compilers from employing more aggressive optimizations. While the parallelism is semantically present, in many complex codes they cannot currently be exploited automatically. Thus, to ensure the continued high performance of these codes on emerging architectures, it is necessary to take a closer look at them from the viewpoint of modern hardware and programming models.

The ultimate aim is to "future-proof" large scientific code bases so that developments in hardware can be fully exploited for continued high performance. There is a large ongoing effort to evaluate modern hardware and software and to investigate how existing codes could be altered - in a minimally invasive way - to adopt them. Clearly, manually porting large codebases to use various different programming models and languages, and then to maintain these different versions is infeasible. As one solution the stakeholders that use these large codes have adopted a practice of developing small but representative applications, or "mini-apps", that can be used as a proxy to test and evaluate new technologies, approaches and optimisations.

The use of proxy applications are a natural strategy, one that has been growing in the last few decades since it became obvious that small benchmarks, such as LINPACK, do not give fair insights into a systems capabilities to handle an organisation's high performance computing (HPC) workload. As such it is common to make available a number of representative applications from an organisation's workload to vendors and evaluate systems based on their performance when procuring HPC systems [1]. More recently their utility in investigating application optimisations and code porting has become important. Due to their small size, representative applications are much more manageable than the actual production application and can be feasibly re-written in different programming languages and with optimizations are easy to experiment with. Recent successful proxy codes include the mini-apps in the Mantevo suite [2] and related codes published by the UK Mini-App Consortium [3] and the LLNL Proxy Apps [4]. These include a variety of computing patterns, ranging from dense and sparse linear algebra, n-body problems, Monte-Carlo simulations and many others. Related to these applications, there is a growing body of work showing how different optimizations can be applied to them [5], [6], [7].

In this paper we present a different point of view, one that is usually less well presented as a contribution of representative applications. Namely their use in developing new programming techniques for numerical simulation software development. We present the important role played by representative/proxy applications in designing and developing two high-level programming frameworks: the OP2 and OPS domain specific active libraries [8] at the University of Oxford. The underlying goal of OP2 [9], [10] and OPS [11], [12] is to allow automatic parallelisation of numerical simulation codes. To achieve this OP2 and OPS focus on an application domain so as to utilize domain specific knowledge of the application in defining an abstraction to specify the problem to be solved. OP2 targets the domain of unstructured-mesh applications while OPS focuses on multi-block structured mesh applications. Encapsulating the domain knowledge, an API is developed to give domain scientists a way to specify the problem to be solved. Its parallel implementation, or how it is to be solved is left to to a lower level. Parallel computing and optimisation experts can then adopt the latest programming models, hardware-specific optimisations to achieve near-optimal performance when implementing the parallel solution. To facilitate this multi-layered development, OP2 and OPS provide source-to-source translation tools to convert the application written in the high-level API to highly optimized, specific parallelisations (e.g. MPI, OpenMP, CUDA, OpenCL, etc.) which can then be linked with optimised library functions to obtain the final parallel executable.

The use of representative applications have been a cornerstone in the development of OP2 and OPS. The aim of this paper is to chart our experiences in this development over the past years. We present two cases through which we explore the design of OP2/OPS and the performance of parallel applications generated by them:

- The first addresses the important question of whether parallelisations generated through high-level frameworks such as OP2/OPS are indeed as optimal as the hand tuned versions. We use a hydrodynamics mini-app, CloverLeaf [13] and its many hand-tuned parallel versions available in the Sandia Mantevo suite to explore this open question, with performance results gained from a range of parallel platforms.
- The second explores whether the optimisations discovered and applied to proxy apps are directly transferable to large-scale industrial applications. We present the case where a representative CFD application called Airfoil, developed with OP2, acted as a forerunner enabling us to gain insights into the performance of a much larger industrial application to be re-engineered (i.e converted) to use OP2. With performance results from a range of systems we show how qualitative performance of Airfoil matched that of the much larger Hydra CFD application, one of the main production applications used at Rolls Royce plc. Our results give direct evidence of how qualitative performance from proxy apps translate to large-scale production applications which they represent.

The rest of the paper is organized as follows. In Section II we briefly describe the design of OPS and OP2 libraries to give context to the contributions of the paper. Section III describes the techniques used by OPS and OP2 to implement support for new architectures, programming models and optimisations, and how these were developed and tuned using proxy applications. Section V evaluates the domain specific programming approach by comparing performance against hand-tuned implementations of representative applications, and Section IV discussing how the results translate to a large-scale industrial code base. Section VI presents a number of optimisations and improvements that was applied to OP2/OPS through experimentation with proxy applications. Finally, Section VII describes related work and Section VIII draws conclusions.

II. Designing the Oxford Parallel Libraries with Proxy Applications

Oxford Parallel libraries for unstructured grid computations, OP2, and multi-block structured computations, OPS, both take a domain-specific active library approach. They provide a well-defined high-level abstraction in the form of a domain specific API that looks like a conventional software library. However a source-to-source translation or code-generation phase takes place where the API calls are parsed to automatically generate parallel implementations for various target architectures using different programming models. The generated code is compiled and linked with target-specific back-end implementations that enable run-time execution support, implementing features such as distributed memory MPI communications, or data movement in heterogeneous systems. For both libraries, the abstraction and API consists of two primary components; the definition of the computational grid and the description of parallel computations on the mesh.

The key challenge in designing these abstractions was to support a sufficiently wide range of applications and numerical methods, but at the same time be restrictive enough so that the library developers, based on domain-specific knowledge, can implement aggressive optimizations and execution transformations. Domain scientists need an easy to use programming model that is oblivious to issues of parallel execution and data movement - things that are crucial for performance but vary widely between target architectures. By decoupling the abstract definition of computations from their actual implementation, it is possible for a small group of computational experts to maintain the library and adopt new programming models and optimizations and easily deploy them to large-scale scientific codes through target specific back-ends and source-to-source translation/code generation.

A. The Abstraction

The OP2 library targets unstructured mesh computations its design [9] and performance [14], [10], [15] on various architectures was published in several papers. The abstraction defines the mesh as (1) a number of sets (such as vertices or edges), (2) mappings between these sets, and (3) data defined on the sets (such as coordinates or flow variables). This mesh is defined once, at the beginning of the application, and all data is handed over to the library. Computations are then expressed as a sequence of parallel loops, each over a given set, executing a “user function” on each set element, accessing data either directly on the iteration set or through at most one level of indirection. The API also captures how data is accessed; read, written, incremented or read-and-written. This loop definition follows the Access-Execute abstraction [16] and enables the library to handle all data movement and any race conflicts automatically. Applications are written from the perspective of a single-threaded implementation, parallelism and data movement is implicit.

In contrast, the OPS library targets multi-block structured mesh computations that often occur in complex CFD simulations. Its abstraction, API and performance was published in previous papers [11], [12]. The abstraction defines a collection of blocks, each with a number of dimensions (1D, 2D, 3D, etc.) but no particular size. Datasets are then defined on the blocks, each with its own size - this is to accommodate data on vertices, faces or cells, the number of which is often different, as well as multi-grid situations. Halos between datasets defined

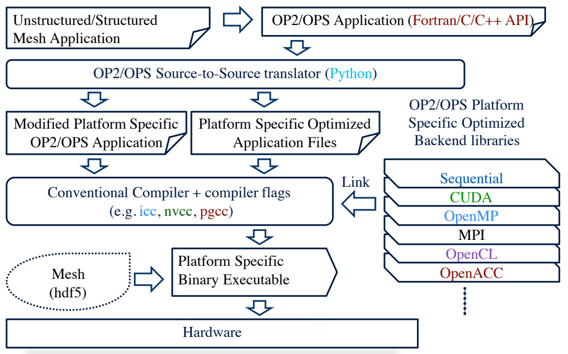

Unstructured/Structured Mesh Application
OP2/OPS Application (Fortran/C/C++ API)
OP2/OPS Source-to-Source translator (Python)
Modified Platform Specific OP2/OPS Application
Platform Specific Optimized Application Files
OP2/OPS Platform Specific Optimized Backend libraries
Conventional Compiler + compiler flags (e.g. icc, nvcc, pgcc)
Link
Sequential
CUDA
OpenMP
MPI
OpenCL
OpenACC
Mesh (hdf5)
Platform Specific Binary Executable
Hardware

Fig. 1: OP2/OPS code generation and build process

on different blocks are the explicitly defined by the user, including their extent and orientation relative to each other. Computations are then described in a similar way to OP2; as a sequence of parallel loops, each on a given block, executing a "user function" on each grid point and accessing data defined on the block through pre-defined stencils, stating the type of access. Inter-block halo exchanges are triggered explicitly by the user and serve as synchronization points between the execution of different blocks; in OPS there is implicit parallelism in the execution of different grid points in the same parallel loop, as well as between loops defined on different blocks. Again, computations are described by the user from the perspective of a single-threaded application, and all the parallelism and data movement is handled by the OPS library.

B. The Library Structure

The libraries are structured as shown in Figure 1; first the applications is implemented using either the C/C++ or Fortran API. This code is then parsed by a python source-to-source translator that modifies the user's high-level source code to call the specialized implementations instead of the generic one, and at the same time a number of platform-specific implementation files are generated, one for each parallel loop and target platform pair. These source files are then compiled using compilers for the target architecture and linked against the target-specific back-end libraries to create the executable. The application can then be run directly on the target platform, file I/O is either managed by the programmer, or OP2 and OPS have support for parallel I/O using HDF5.

Both OP2 and OPS support distributed memory execution with MPI; using the up-front definition of the mesh and the access-execute description [17] of computations, they automatically perform partitioning across processes and use standard halo exchanges, exchanging halo messages on-demand based on the type of access and the stencils. On top of MPI, both libraries support a number of shared memory parallel programming models; OPS generates code for multi/many-core execution with OpenMP, OpenACC, CUDA and OpenCL, and enables compiler auto-vectorization on CPUs. OP2 currently supports multi-core CPUs with OpenMP and GPUs with CUDA. Potential race conflicts that occur in a shared-memory environment are handled with two layers of coloring: an MPI partition is broken up into smaller blocks which are then colored based on potential data races, blocks of the same color can be safely executed by different OpenMP threads or CUDA thread blocks. In case of CUDA, a further level of coloring is necessary, within these blocks where individual set elements are colored, during execution intermediate results are stored in the registers of the threads and then stored to main memory in a colored fashion. This coloring is described in an execution plan that is created at run-time for any loop that has potential race conflicts.

C. Porting Applications

In order to make use of OP2 or OPS, an application has to be converted to use the domain-specific API; first all the sets, mappings and datasets have to be declared. The necessary file I/O can either be done by the user and the memory arrays handed to the library, or the library can read data from HDF5 files. These API calls return opaque handles that have to be used in the description of computations. Parallel loops are constructed from traditional loop nests by outlining the computational core (the "user function") and removing all iteration-specific indexing, and then writing the call to the OP2 or OPS API, describing the iteration set/block, specifying the user function and listing all the datasets to be accessed and the type of access. This code can be immediately compiled and run using generic implementations of the parallel loop, however this is only intended for validation, not for production. Code generation then creates high-performance implementations for various targets - the single-threaded sequential implementation is recommended for debugging purposes as well, since it is a human-readable code with a simple loop nest that computes pointers and passes them to the user function. Indeed all parallel code generated with the source-to-source translator is human-readable to enable debugging and profiling with third party tools. The development process is also aided by built-in mechanisms in these libraries that help check for consistency and correctness; for example OPS can automatically check whether the used stencils match the declared ones, and there are API calls to dump entire datasets to disk, even in a distributed memory environment.

In order to use the full range of features supported by OP2 or OPS, it is necessary to convert the entire application to use the library; once datasets are defined, these are often reallocated, reordered etc., internally by the back-end libraries. This means an all-or-nothing porting strategy is necessary. However, the CPU back-ends, without any MPI or advanced optimizations, do use the original data arrays, which makes it possible to gradually convert an application to OP2/OPS and verify correctness at every stage, or even to only partially convert the application and rely on an existing distributed memory implementation for example - this we hope lowers the barrier to using these libraries. Once the conversion is complete, the automatic MPI implementation of the library is enabled, along with GPU execution and many other advanced features, such as mesh renumbering, Array of Structures to Structure of Arrays conversion, automatic check-pointing and recovery and many others.

III. Representative Applications as Experimental Vehicles

Representative applications were essential for the design and development of these libraries for two main reasons: first, we had to appropriately scope the abstraction so that it would be possible to express complex problems and computations with it. This is perhaps the most important issue when designing a domain specific API, if it was too narrow then it would never be more than an academic experiment, and if it was too wide it may be too general purpose where domain specific knowledge could not be exploited. Second, in order to be

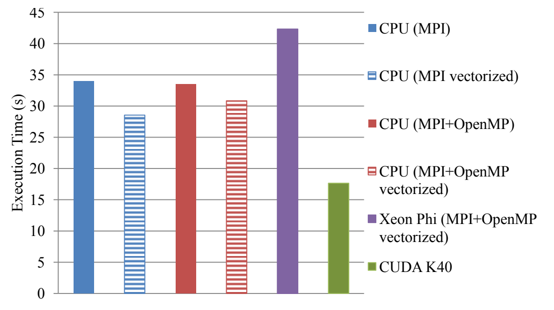


Fig. 2: Performance of Airfoil on single node systems (Xeon E5-2697v2 CPU, Xeon Phi and an NVIDIA K40 GPU)

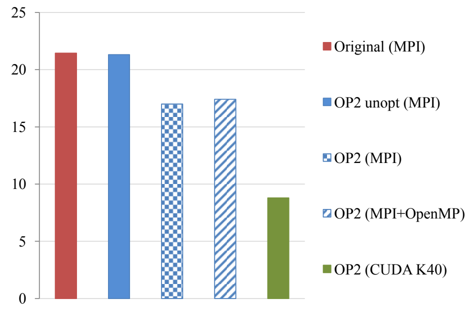


Fig. 3: Performance of Hydra on a single CPU node Xeon E5-2640

able to adopt new programming models and languages we need test cases that are well understood, where we can easily quantify performance in absolute terms and decide how close to the theoretically achievable a given optimization brings us. It is also much easier to test and debug new techniques because of the manageable size of these codes; indeed our development process includes first creating hand-coded implementations for a particular architecture based on the best practices of the target platform, and then we create a code generator to automatically produce the same code structure - the optimization can then be easily deployed to large applications through the code generator.

IV. Representative vs Industrial Applications

One of the key expectations from proxy applications is that they represent all aspects of large-scale applications that are relevant when making design and optimisation decisions. Our first set of experiments illustrate how the utility of a proxy application was crucial in addressing this issue for the OP2 library. The design, development and testing of OP2 was carried out with the use of a non-linear 2D inviscid mini CFD application called Airfoil [18], [9] that performs computations on an unstructured mesh. Airfoil was written directly using the OP2 API as a experimentation forerunner representative of the Rolls-Royce Hydra [19], [20] CFD code, a full-scale industrial application developed for the simulation of turbomachinery. Hydra consists of several components to simulate various aspects of the design including steady and unsteady flows that occur around adjacent rows of rotating and stationary blades in the engine, the operation of compressors, turbines and exhausts as well as the simulation of behaviour such as the ingestion of ground vortexes. The guiding equations that are solved are the Reynolds-Averaged Navier-Stokes (RANS) equations, which are second-order PDEs. By default, Hydra uses a 5-step Runge-Kutta method for time-marching, accelerated by multi-grid and

TABLE I: Time (seconds) and bandwidth (GB/s) breakdowns for the Airfoil benchmarks in double precision

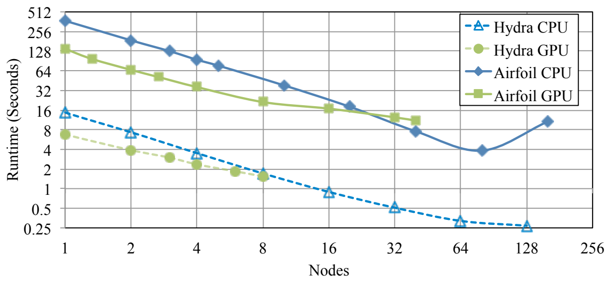
Kernel	E5-2697		Xeon Phi		NVIDIA K40	
	Time	BW	Time	BW	Time	BW
save_soln	2.9	62	2.17	84	0.81	213
adt_calc	5.6	57	6.86	47	2.63	115
res_calc	9.9	69	27.2	25	10.8	60
update	9.8	79	8.77	89	3.22	228

block-Jacobi preconditioning [19], [21]. The usual production meshes are in 3D and consist of tens of millions of edges, resulting in long execution times on modern CPU clusters. Hydra was originally designed and developed over 15 years ago at the University of Oxford, and has been in continuous development since, it has become one of the main production codes at Rolls Royce. Hydra is written in Fortran 77 and consists of over 300 computational loops, about 50 thousand lines of code. In this section we present our experiences on how the insights gained through Airfoil translated to re-engineering and converting Hydra to OP2 and its subsequent performance.

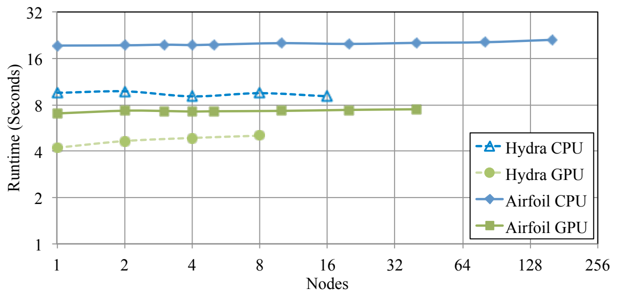
Figure 2 shows the overall performance of Airfoil on a number of single node processor architectures: a high-end Ivy Bridge Xeon server (E5-2697 v2), an Intel Xeon Phi 5110P and an NVIDIA K40 GPU, using various programming models. We see that these figures roughly mirror what we would expect from a bandwidth limited code on these hardware. To understand performance bottlenecks, we look into per-loop breakdowns and performance metrics, as shown in Table I. The two loops `update` and `save_soln` are straightforward direct loops with very little computations and a trivial memory access patterns, on most platform they achieve bandwidth near the optimum. The loop `adt_calc` reads indirectly but writes directly, and has expensive square root instructions, thus vectorization on CPU-like architectures is necessary. Finally `res_calc` does both reads and writes indirectly, making colored execution necessary. This in turn requires expensive gather and scatter memory access patterns. As such these operations appear to be the main bottleneck for this loop. The more reliant the architecture is on long vector units the worse the performance impact.

In comparison to Airfoil, the performance of Hydra on a similar Intel Xeon (E5-2640) system is given in Figure 3. Note again, that the *Original* and *OP2 unopt* versions' performance is nearly identical in this case with respect to runtime. This provides evidence that the high-level programming approach introduces no overhead, at least for pure MPI parallelisations compared to its hand-tuned implementations. We also see that further improvements introduced by OP2 enables for additional runtime improvements. Improvements include the use of state-of-the-art partitioners, such as PT-Scotch [22] or ParMetis [23], as well as automatic mesh reordering to improve locality. Figure 3 shows that even on a single node this leads to a 30% performance improvement.

In Figure 2, for Airfoil we see that the use of hybrid MPI+OpenMP does not improve performance on a single node over MPI (with vectorisation). The same observation can be made for Hydra. OP2 also supports targeting GPUs, and we were able to deploy optimizations such as the use of the texture cache or the automatic conversion from an Array of Structures data layout to Structure of Arrays through the code generator. Figure 3 shows that using a single K40 GPU, we significantly outperform the CPUs of the host system. As Hydra contains a

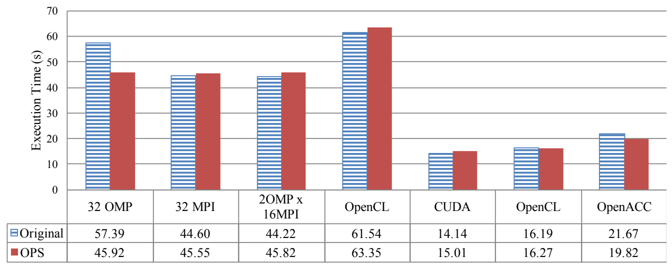


(a) Strong Scaling



(b) Weak Scaling

Fig. 4: OP2's distributed memory execution performance: CPU results from HECToR (CRAY XE6), Airfoil GPU results from Emerald (M2090 GPU cluster) and Hydra GPU results from Jade (K20m GPU cluster)



	32 OMP	32 MPI	2OMP x 16MPI	OpenCL	CUDA	OpenCL	OpenACC
▤ Original	57.39	44.60	44.22	61.54	14.14	16.19	21.67
🟥 OPS	45.92	45.55	45.82	63.35	15.01	16.27	19.82

Fig. 5: CloverLeaf's performance on CPUs and GPUs with different programming models

large number of indirect loops, moves many times more data per grid point than Airfoil does, and carries out more complex computations the GPU kernels achieve lower occupancy and have higher branch divergence leading to lower efficiency. As such the performance advantage is not as large as we experienced on Airfoil. However we see that the qualitative insights given by Airfoil are still valid for Hydra.

Figure 4 plots the strong scaling and weak scaling performance of Airfoil and Hydra on a classical CPU cluster (HECToR [24]) and a GPU cluster (EMERALD [25]). We again see that the qualitative trends at increasing scale observed with Airfoil are indeed similar to those from Hydra. In case of strong scaling, as the problem size per node decreases the performance trails off, much faster with GPUs than CPUs, due to the GPU architecture's sensitivity to the amount of workload. Weak scaling performance holds steady, and in case of Airfoil with less than 5% performance degradation when going from 32 CPU cores to 5120, or 1 GPU to 48. In conclusion these experiences demonstrate that techniques and insights gained from a representative application are indeed directly transferable to a large-scale industrial application such as Hydra.

For Airfoil, it was possible to generate code that explicitly used vector intrinsics or code that would get auto-vectorized by the compiler [15]; unfortunately the former method is too restrictive in OP2 (does not allow branching in user functions) and intrinsics are not available in Fortran. While code amenable for auto-vectorization along with the appropriate compiler pragmas could be generated for Hydra, most loops are much too complex for the compiler to vectorize, and the loops that do vectorize are trivial and do not improve overall performance.

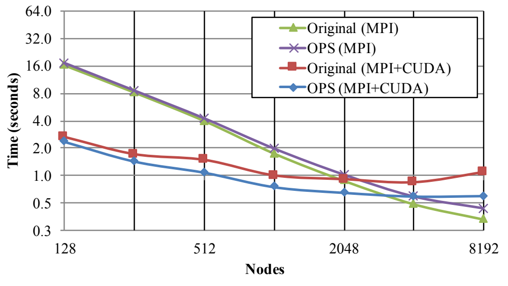
V. Performance Portability

One of the primary question we encountered during our research is on how the code produced by the high-level frameworks such as OP2/OPS perform compared to traditionally hand-ported/parallelised and optimized codes. This is a key aspect, naturally questioned by many stakeholders at HPC organisations who rightfully felt that adapting the high-level development strategy can only be approved if it also results in a code with near-optimal performance. In the previous section some evidence was presented to support the fact that the high-level programming approach in OP2 introduces no overhead, at least for the pure MPI parallelisation compared to its hand-tuned implementation. In this section we further explore this issue with results from a more extensive set of parallelisations and systems.

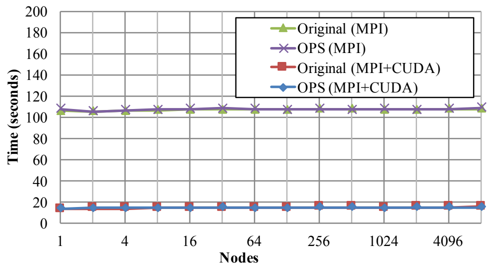
In support of this we utilize the many parallel implementations of the 2D CloverLeaf mini-app in evaluating the effectiveness of the code generated by our domain specific approach and OPS. CloverLeaf [3], [5], was developed to be representative of algorithms of interest in large-scale hydrodynamics codes. It involves the solution of the compressible Euler equations, which form a system of four partial differential equations. The equations are statements of the conservation of energy, density and momentum and are solved using a finite volume method on a structured staggered grid. The code is a self contained and made up of roughly about 7 thousand lines of Fortran. As such it has been ported to a large range of parallel implementations, tuned by experts and optimised for various modern architectures and programming models, including MPI, OpenMP, CUDA, OpenCL, OpenACC and other models. These versions can serve as a basis of comparison with the many parallel versions auto-generated through OPS.

Figure 5 compares the performance of code generated through OPS against the original implementations from [5]. We see that the results from OPS are on par with hand-coded and tuned implementations. On the CPU, OPS is at most 5% slower than the reference implementations, except for pure OpenMP execution, where OPS handles NUMA issues more efficiently and therefore runs 20% faster. On the GPU with CUDA, the reference implementation uses loop fusion in some places, thus it is not an exact port of the original, but even so OPS comes within 6%, and matches or outperforms the original with OpenCL and OpenACC.

The scaling performance can also be explored by comparing the code generate through OPS to that of the hand-tuned originals. Figure 6 illustrates the scaling performance on the ORNL Titan supercomputer [26] on up to 8K nodes. Again we see that the OPS versions closely matches the scaling performance of the hand-tuned versions pointing to the conclusion that the distributed memory execution strategy



(a) Strong Scaling



(b) Weak Scaling

Fig. 6: CloverLeaf's scaling using CPUs or GPUs on the Titan supercomputer

```
#if NOSOA
#define OP_ACC0(x) (x)
#elif SOA
#define OP_ACC0(x) ((x)*coord_stride)
#endif

__device__ void user_fun(double *coords,...) {
...
double x = coords[OP_ACC0(0)];
double y = coords[OP_ACC0(1)];
...
}

__global__ void wrapper(double *coords,...) {
int gbl_idx = ...;
#if STAGE_NOSOA
__shared__ double scratch[...];
scratch[2*threadIdx.x  ] = coords[2*gbl_idx+0];
scratch[2*threadIdx.x+1] = coords[2*gbl_idx+1];
user_fun(&scratch[2*threadIdx.x],...);
#elif NOSOA
user_fun(&coords[2*gbl_idx],...);
#elif SOA
user_fun(&coords[gbl_idx],...);
#end
}
```

Fig. 7: CUDA code generated with OP2 for different memory access strategies

implemented in OPS is indeed efficient. near-optimal strong scaling is achieved up to 4096 CPU nodes (65k cores), however, the GPU execution does not strong scale very well, due to the hardware's sensitivity to workload size. In case of weak scaling we see near-optimal efficiency on both platforms: only 1% reduction in performance when going from 16 cores to 65 thousand, and 6% when going from 1 GPU to 8 thousand.

These results by themselves are of huge significance; they show that the high-level programming approach is both easy to use and able to deliver high performance close to the target architecture's limits - on par with carefully hand-implemented versions. As mentioned before, hand-coded versions of CloverLeaf are all available in separate repositories, each over 7K lines of code, depending on the programming model used - clearly, any changes or additions to the "science" code is a major challenge, because it involves manual changes to each and every version. While for such small representative applications this is not a huge issue, maintaining separate versions of the large-scale codes would be infeasible In contrast to this, our domain specific programming models do not aim to be general purpose, and by separating the "science" parts of the code from the "performance" part, altering either is independent of the other: both productivity and performance is achievable.

VI. Developing Optimisations

It is far from obvious how different optimisations will impact performance. Proxy applications again proved invaluable by enabling the quick development, testing and evaluation of these optimizations. Important improvements applied to OP2 and OPS relate to the movement of data, particularly locality in data accesses and memory access patterns. Improving either of these is closely tied to the underlying data structures and the execution strategies implemented for a specific target architecture. Locality on CPUs can be improved using techniques such as cache blocking, or on GPUs with the use of the local scratch-pad (shared) memory. The closely related issue of memory access patterns is especially important on architectures with small caches. In case of vector computing, efficiently utilizing the available bandwidth requires aligned memory accesses and strict spatial locality (i.e. consecutive indices) across vector lanes. Figure 7 presents an example code segment for Airfoil, generated through the OP2 code generator for different memory access strategies with CUDA for NVIDA GPUs. A dataset `coords` stores x and y values for each vertex, before passing a pointer to a user-defined function. The preprocessor statements (`STAGE_NOSOA`, `NOSOA` and `SOA`) and the code contained within each of them gives the the options as to how data is accessed and whether we control data re-use or rely on the cache. A number of similar optimizations could be carried out that control concurrency or distributed memory messaging [9], [10].

Apart from the optimisations for improving runtime or scalability other improvements to an application could also be explored. An example is the check-pointing and recovery mechanisms implemented for OP2 and OPS. Minimizing the overhead of check-pointing by reducing the amount of data to be saved is crucial, and it is challenging to determine what data is necessary for recovery in large-scale applications. Representative applications once again help in developing and verifying an algorithm to determine the optimum location and set of data for check-pointing. Due to the access-execute description of computations and the fact that all data is handed over to our libraries, it is possible to reason about the state of all the datasets at any particular point during execution; when creating a checkpoint datasets that are immediately overwritten do not need to be saved. The user only needs to specify the frequency of checkpoints, the rest can be done automatically by the library; upon initiating a checkpoint the library decides which datasets to save and which ones to exclude from the checkpoint based on how they are accessed though a number of subsequent loops. Furthermore, the library can analyse recurring sequences of loops and find the most optimal location for check-pointing. Clearly, deciding how well these algorithms perform would be difficult on large-scale applications, but with the use of proxy-apps we can compare the results of the automated method with what we know is the

	bounds(1)	rms	x(2)	q(4)	q_old(4)	adt(1)	res(4)	units of data saved if entering checkpointing mode here
	vector	scalar	vector	vector	vector	vector	vector	
1 save_soln				🟩 R	🟨 W	🟦	🟦	8
2 adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
3 res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
4 bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
5 update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8
6 adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
7 res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
8 bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
9 update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8
10 save_soln				🟩 R	🟨 W	🟦	🟦	8
11 adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
12 res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
13 bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
14 update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8
15 adt_calc			▨ R	🟩 R	🟦	🟨 W	🟦	12
16 res_calc			▨ R	🟩 R	🟦	🟩 R	🟩 I	13
17 bres_calc	▨ R		▨ R	🟩 R	🟦	🟩 R	🟩 I	13
18 update		🟥 I		🟨 W	🟩 R	🟦	🟩 RW	8

🟥	▨	🟨	🟩	🟦
save every iteration	not saved ever	not saved	saved	unknown yet

Fig. 8: Deciding on what data to be checkpointed - Evaluation from Airfoil

optimum solution.

Figure 8 shows an example of how the checkpointing algorithm work when applied to the Airfoil benchmark application. It shows for each loop, which datasets are read (R), written (W), incremented (I) or read-and-written (RW) as well as the dimensionality, or number of components, for each dataset. It also indicates how many units of data would have to be saved if checkpointing mode was entered at the current loop. Colours are used to indicate when a dataset is saved (green), dropped (yellow) or flagged for further decision (blue). If checkpointing were to be triggered right before the execution of `adt_calc`, then checkpointing mode would be entered upon reaching `adt_calc`, saving `q` and dropping `adt` immediately, and then subsequently `res` would be saved when reaching `res_calc` and `q_old` when reaching `update`. Since `bounds` and `x` were never modified, they are not saved, and the values of `rms` are saved whenever `update` has executed. Clearly, entering checkpointing mode at kernel `adt_calc` is still better than entering it before either `res_calc` or `bres_calc`, but it does not give as much data savings as if checkpointing mode were to be entered before either `save_soln` or `update`. Therefore either the user can add a call to trigger checkpointing before these two loops, or OP2 can apply the "speculative" algorithm and recognise that there is likely a periodic execution because the sequence of kernels 1-9 repeats, thus it can wait with entering checkpointing mode until either `save_soln` or `update` are reached.

In the event of a failure, the application is simply restarted, but until reaching the point in execution where the checkpoint was made, the `op_par_loops` do not carry out any computations, only set the value of `op_arg_gbl` arguments, if any. This ensures that the same execution path is followed when fast-forwarding, once the checkpoint is reached, all state is restored from the saved data, and normal execution is resumed.

VII. Related Work

Representative applications have been used as a vehicle for testing new programming models and optimizations; LLNL published a number of Proxy Applications [4], and research has been carried out to investigate various programming models including low-level (MPI, OpenMP, CUDA) and high-level (Charm++, Liszt) approaches [27]. Sandia produced the Mantevo suite of mini-applications [2], and these have also been used to compare and analyse various programming approaches [5], [28], [7]. These applications have been tremendously helpful in allowing academics to experiment without having to access classified code, and they have been also used for procurements of new supercomputers.

VIII. Conclusions

In this paper, we have discussed our experiences with representative applications and domain specific programming models. We have shown that mini-apps are instrumental in designing and developing these domain specific abstractions because they help correctly scope the abstraction itself, and they make the testing and validation of new optimizations and target platforms much easier due to their manageable size.

Through OP2's Airfoil mini-application and OPS's CloverLeaf, we have demonstrated that a domain-specific high-level programming approach can deliver productivity to the application developer and at the same time high performance, both in terms of absolute performance metrics, and when compared to expert hand-coded implementations. This is achieved by separating the abstract description of computations form their actual parallel implementation, letting domain scientists and parallel programming experts work independently. The results underline the viability and the flexibility of such a high-level programming approach; from a single source code it is possible to automatically enable execution on a variety of target platforms, utilising different parallel programming models such as MPI, OpenMP, CUDA, OpenCL or OpenACC without user intervention and at the same time achieve performance within 6% of expert's hand-coded implementations. We have demonstrated performance portability across a single core of a CPU, a single GPU, up to 65 thousand CPU cores or 8 thousand GPUs.

We have also shown that a large-scale industrial application, Rolls-Royce Hydra can be ported to use OP2, and that our experiences with the development and performance evaluation of the Airfoil mini-application indeed translate to Hydra. This reinforces our belief that domain-specific abstractions are a viable option for future-proofing large scientific codes and that the design and development of these libraries using representative applications is a strategy to be followed in the future.

Acknowledgements

The OP2 project has been funded by the UK Technology Strategy Board and Rolls-Royce plc. through the Siloet project, the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on "Multi-layered Abstractions for PDEs".

The CloverLeaf development is funded by the UK AWE plc. under project "High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems". 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. This paper used the Archer UK National Supercomputing Service from time allocated through UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on "Multi-layered Abstractions for PDEs".

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.

Cloverleaf development is supported by the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme) and also the Royal Society through their Industry Fellowship Scheme (IF090020/AM).

The authors would like to acknowledge that the work presented here made use of the Emerald High Performance Computing facility made available by the Centre for Innovation. The Centre is formed by the universities of Oxford, Southampton, Bristol, and University College London in partnership with the STFC Rutherford-Appleton Laboratory

References

[1] S. D. Hammond, G. R. Mudalige, J. A. Smith, J. A. Davis, A. B. Mills, S. A. Jarvis, J. Holt, I. Miller, J. A. Herdman, and A. Vadgama, "Performance prediction and procurement in practice: assessing the suitability of commodity cluster components for wavefront codes," *IET Software*, vol. 3, no. 6, pp. 509–521, 2009.

[2] "The Mantevo Project," 2012, http://mantevo.org/.

[3] "UK mini-app consortium," 2014, http://uk-mac.github.io.

[4] "ASC Co-design Proxy App Strategy," 2012, https://codesign.llnl.gov/pdfs/proxyapps_20130106.pdf.

[5] A. Mallinson, D. Beckingsale, W. Gaudin, J. Herdman, and S. Jarvis, "Towards portable performance for explicit hydrodynamics codes," in *1st International Workshop on OpenCL (IWOCL 13)*, 2013. [Online]. Available: http://eprints.dcs.warwick.ac.uk/1756/

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

[7] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the performance portability of structured grid codes on many-core computer architectures," in *International Supercomputing 2014*, vol. 8488. Springer International Publishing, 2014, pp. 53–75.

[8] 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: Springer-Verlag, 2000, pp. 25–39.

[9] M. B. Giles, G. R. 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.

[10] G. Mudalige, M. Giles, J. Thiyagalingam, I. Reguly, C. Bertolli, P. Kelly, and A. Trefethen, "Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems," *Parallel Computing*, vol. 39, no. 11, pp. 669 – 692, 2013.

[11] G. R. Mudalige, I. Z. Reguly, M. B. Giles, A. C. Mallinson, W. P. Gaudin, and J. A. Herdman, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation. 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*. Springer, 2015, vol. 8966, no. 1, ch. Performance Analysis of a High-level Abstractions-based Hydrocode on Future Computing Systems, pp. 85–104.

[12] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The OPS Domain Specific Abstraction for Multi-block Structured Grid Computations," in *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 58–67. [Online]. Available: http://dx.doi.org/10.1109/WOLFHPC.2014.7

[13] "Cloverleaf 2d," 2014, http://uk-mac.github.io/CloverLeaf/.

[14] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly, "Designing OP2 for GPU Architectures," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 1451–1460, November 2013.

[15] 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. New York, NY, USA: ACM, 2007, pp. 39:39–39:50. [Online]. Available: http://doi.acm.org/10.1145/2560683.2560686

[16] 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. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 168–182.

[17] A. F. D. L. W. Howes, A. Lokhmotov 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. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 168–182.

[18] M. B. Giles, D. Ghate, and M. C. Duta., "Using Automatic Differentiation for Adjoint CFD Code Development," *Computational Fluid Dynamics Journal*, vol. 16, no. 4, pp. 434–443, 2008.

[19] M. B. Giles, M. C. Duta, J. D. Muller, and N. A. Pierce, "Algorithm Developments for Discrete Adjoint Methods," *AIAA Journal*, vol. 42, no. 2, pp. 198–205, 2003.

[20] M. Giles, "Hydra," 2013, http://people.maths.ox.ac.uk/gilesm/hydra.html.

[21] 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," *(under review) ACM Transactions on Parallel Computing*, 2013, available at http://arxiv-web3.library.cornell.edu/abs/1403.7209.

[22] "Scotch and PT-Scotch," 2013, http://www.labri.fr/perso/pelegrin/scotch/.

[23] "ParMETIS," 2013, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[24] "HECToR - hardware," http://www.hector.ac.uk/service/hardware/.

[25] "Emerald GPU cluster," 2012-2015, http://www.cfi.ses.ac.uk/cfi/emerald/.

[26] "Titan Cray XK7," 2014, https://www.olcf.ornl.gov/titan/.

[27] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still, "Exploring traditional and emerging parallel programming models using a proxy application," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 919–932. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.115

[28] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman, "Experiences at scale with pgas versions of a hydrodynamics application," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 9:1–9:11.