Modernising an Industrial CFD Application

István Z. Reguly
Faculty of Information Technology and Bionics
Pázmány Péter Catholic University
Budapest, Hungary
reguly.istvan@itk.ppke.hu

Gihan R. Mudalige
Department of Comuter Science
University of Warwick
Coventry, UK
g.mudalige@warwick.ac.uk

***Abstract*—Rolls-Royce Hydra is an industrial Computational Fluid Dynamics (CFD) code used for the design of turbomachinery. In this paper we describe the modernisation effort that takes the current MPI-only version and migrates it to use the OP2 Domain Specific Language (DSL), which enables OpenMP as well as CUDA parallelisations on top of MPI. We discuss the issue of the the original codebase being under continuous development, and having to keep upstreaming the modernised version using a set of conversion scripts that help to automate the process. The result is a single, easy to maintain, source code that can target modern CPUs and GPUs. We evaluate the performance and scalability of the modernised version on a set of the latest Intel CPUs and NVIDIA GPUs. Results demonstrate matching performance to the original production code using the same parallelization model (MPI) on CPUs, but further speedups of up to $2.8\times$ on modern many-core architectures.**

***Index Terms*—Code modernisation, CFD, CPU, GPU, CUDA, Fortran, DSL**

I. Introduction

Current and future high performance computing architectures increasingly require moving away from a pure MPI parallelisation towards using one of a variety of shared-memory parallelisation techniques. Unfortunately large legacy codebases are notoriously difficult to modernise for a variety of reasons, including (but not limited to):

1) A large fraction of the codebase needs to be refactored to achieve meaningful acceleration,
2) external dependencies that cannot be modernised, and require certain aspects of the code to remain the same (typically some data structures),
3) concurrent development of the original and modernised codebase.

In this paper we describe our modernisation of the Rolls-Royce Hydra CFD code, which is an unstructured mesh Reynolds Averaged Navier-Stokes (RANS) solver that has been in development for over 20 years. It is written in Fortran 77 using the OPlus MPI parallelisation library [1].

Hydra is a multi-component piece of software, which itself fits into a larger workflow designed to accurately simulate various aspects of turbomachinery design, such as steady and unsteady flows around rows of turbine blades that may be rotating relative to each other. It uses a 5-step Runge Kutta method for time-marching and multigrid solvers with block-Jacobi preconditioning [2]–[5]. Typical production simulations use 3D unstructured meshes with tens or hundreds of millions of edges, and run for several days on conventional CPU clusters. While there is a growing need to speed up Hydra for production uses by adopting newer compute architectures, there is a further push for this due to the ASiMoV project [6], funded by the UK EPSRC, that aims to carry out coupled full-engine simulations, which will run into the billions of edges, and require a high-performing, dynamic code base.

While Hydra had been converted to use OP2 previously [7], [8], it was based on a fork of the original codebase and could not be upstreamed, and therefore was only used for demonstrating the capabilities of OP2. In this paper, we describe a new effort to modernise Hydra, while also make it relatively simple to upstream, and allow it to be adopted into production use. We discuss the automated and semi-automated steps that convert the OPlus-based code to use the OP2 API, and how OP2 enables the utilisation of shared-memory parallelisation techniques. Finally, we demonstrate performance improvements on modern CPUs and GPUs.

II. Related Work

The modernisation of existing applications has been a long-standing challenge in the HPC community, most commonly faced by national research laboratories and larger companies. Due to the longevity and the large user base of scientific codes, there is considerable inertia, and many aspects of software development need to be considered, not just the performance of the code.

Work on MSSG (Multi- scale Simulator for the Geo-environment) [9] discusses a common challenge – while porting the application to OpenACC, simply including pragmas may not yield to performant code due to the unsuitability of underlying data structures and algorithms. Their paper discusses a number of code transformations through the Xevolver [10], [11] tool that yield structures more amenable to acceleration. The authors in [12] aim to enable optimizations in existing code by creating a Metaprogramming Framework that enables developers to write custom directives and transformations, that are then translated by the Omni [13] compiler. Of course, there are many who apply (often low-level) optimisations by hand [14], [15].

With new computational architectures, such as GPUs, memory capacity is also an issue – work by [16] discusses reducing the memory footprint of large scale applications through

FUNDING

```
do while(hyd_par_loop(ncells, istart, iend))
  call hyd_access_r8('r',areac,1,ncells,
  &    null,0,0,1,1)
  call hyd_access_r8('u',arean,1,nnodes,
  &    ncell,1,1,1,3)
  do ic = istart, iend
    i1 = ncell(1,ic)
    i2 = ncell(2,ic)
    i3 = ncell(3,ic)
    arean(i1) = arean(i1) + areac(ic)/3.0
    arean(i2) = arean(i2) + areac(ic)/3.0
    arean(i3) = arean(i3) + areac(ic)/3.0
  end do
end while
```

Fig. 1. A Hydra loop written using the OPlus API

```
subroutine distr(areac,arean1,arean2,arean3)
real(8), intent(in) :: areac
real(8), intent(inout) :: arean1,
   & arean2, arean3
arean1 = arean1 + areac/3.0
arean2 = arean2 + areac/3.0
arean3 = arean3 + areac/3.0
end subroutine

op_par_loop(cells, distr,
& op_arg_dat(areac,-1,OP_ID,1,'r8',OP_READ),
& op_arg_dat(arean,1,ncell,1,'r8',OP_INC),
& op_arg_dat(arean,2,ncell,1,'r8',OP_INC),
& op_arg_dat(arean,3,ncell,1,'r8',OP_INC))
```

Fig. 2. A Hydra loop written using the OP2 API

transformations, and demonstrate a reduction of an order of magnitude.

What is clear, is that automatic modernisation and optimisation methods work better when more assumptions can be made about the code – the more its computations are laid out according to a set of fixed patterns. The use of domain specific libraries helps tremendously; indeed for our own work we could assume that virtually all accesses to data, and operations on data, happen in well-understood (and parsable) code constructs.

III. The OPlus and OP2 libraries

OPlus [1], [17] is a classical software library written in Fortran 77 that facilitates computations on unstructured meshes; it handles distributed memory domain decomposition and halo exchanges. It provides a simple abstraction: a mesh is described using sets (such as vertices and edges), fixed arity mappings between sets (such as edges to nodes, which always has 2 connections per set element), and data defined on sets (such as coordinates – x,y,z values associated with each element). Computations are then described as a loop over the elements of a set, accessing data either directly on the set or via at most one level of indirection. The key assumption is that the order in which elements are executed may not change the result, at least within machine precision. Figure 1 describes a loop over cells, reading a dataset $areac$ defined on cells, and updating a dataset $arean$ defined on nodes. The library sets the $istart$ and $iend$ variables appropriately to make sure each process iterates over elements, while it also manages MPI communications.

The OP2 library [18], [19] is a successor to OPlus – it keeps the fundamental abstraction, but introduces data structures, datatypes, and new APIs that allow it to deliver shared-memory parallel implementations on top of MPI. Sets, mappings, and datasets have their own opaque datatype in OP2 – as opposed to OPlus which just used Fortran arrays – which allows OP2 to fully manage multiple memory spaces and data transfers, and at the same time, only allows access to data through API calls.

Figure 2 shows the same loop as before using the OP2 API – the key difference is that the loop body has to be outlined, and that indexing into data arrays is not done by the programmer. This achieves a *separation of concerns*; the user only describes what should be computed, and does not specify how (in terms of data movement and parallelism), allowing OP2 to then automatically perform this step as it is best suited for a given target architecture.

A distinct difference between OPlus and OP2 is how "global" variables are handled. There are two further cases; variables in global scope, which are always read-only (otherwise it would violate the order-less assumption), and iteration-invariant variables that are explicitly passed to the outlined subroutine, and are either read-only, or used for an associative reduction. In OPlus none of these are explicitly marked or passed through an API function, except for reductions, for which there is a separate reduction call *after* a computational loop. In OPlus-Hydra, global scope constants are defined using common blocks, other iteration-invariant variables are locally declared in encompassing subroutines. With OP2, global scope constants need to be declared using the *op_decl_const* API, which in a multiple memory space environment can make sure the values are consistent. Iteration-invariant variables have to be passed explicitly in the *op_par_loop* call using *op_arg_gbl*, with the appropriate access descriptor (read/increment/min/-max). OP2 again uses this information to move data and perform parallel reductions when needed.

While from a programming perspective, OP2 looks like a conventional software library (and can be used so), it also uses source-to-source translation to generate parallel code for various target architectures and compilers. Figure 3 illustrates the development and compilation workflow; an application written once with OP2's C/C++ or Fortran API is then run through a Python-based translator that generates specialised implementations for each *op_par_loop*, and a modified application file that calls these implementations. These files are then compiled with conventional compilers and linked against platform-specific OP2 libraries that manage distributed memory parallelism and data movement between separate memory spaces. While OP2 has built-in support for parallel

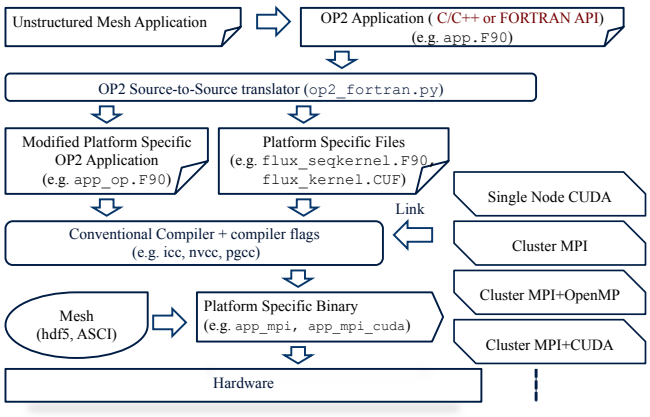

Fig. 3. OP2 build hierarchy

file I/O using HDF5 [20], Hydra uses its own (similarly HDF5-based) I/O library that does this.

IV. The Conversion Process

Considering the large codebase (100k+ LoC) and the need for a quick migration path from OPlus to OP2 to allow upstreaming we have developed a conversion process that automates as many steps as possible.

As a first step, we need to make sure the bodies of all OPlus *hyd_par_loop*s are outlined into separate subroutines, as shown in Figure 4 – this is already the case for most loops as they are implemented. For the rest of loops outlining was done by hand, because writing a robust refactoring algorithm to perform outlining was not feasible – it is likely that a full compiler infrastructure would have been required, which was not an option for the scope of this project. These changes were already submitted to the main development branch and some already adopted.

There were also a small number of cases where loop-carried dependencies were present that prohibit parallelisation, such as common block variables read *and* written, or Fortran data/save values updated. The former were rewritten to use local variables, the latter were moved out of parallel loops, these changes were also submitted to the main development branch.

The second step is parsing the code and finding API calls to the OPlus library, then converting them to their OP2 equivalents. Thanks to F77 restrictions and strict coding conventions this is just a matter of text processing using regular expressions;

- Finding *hyd_par_loop*s, and recording its arguments (index ranges)
- Finding the following *hyd_access* calls and recording its arguments (read/write/update, number of values per element, and mapping used for indirection)
- Locating the *do* loop with the recorded index ranges, and noting the loop counter variable
- Recording indirect indexes (such as $i1, i2, i3$ here)
- Finding call to outlined subroutine, and parsing the actual arguments.

```
subroutine distr(areac,arean1,arean2,arean3)
real(8), intent(in) :: areac
real(8), intent(inout) :: arean1,
   & arean2, arean3
arean1 = arean1 + areac/3.0
arean2 = arean2 + areac/3.0
arean3 = arean3 + areac/3.0
end subroutine

do while(hyd_par_loop(ncells, istart, iend))
  call hyd_access_r8('r',areac,1,ncells,
  &     null,0,0,1,1)
  call hyd_access_r8('u',arean,1,nnodes,
  &     ncell,1,1,1,3)
  do ic = istart, iend
    i1 = ncell(1,ic)
    i2 = ncell(2,ic)
    i3 = ncell(3,ic)
    call distr(areac(ic),arean(i1),arean(i2),
  &         arean(i3))
  end do
end while
```

Fig. 4. A Hydra loop with an outlined body

  - Arguments indexed with the loop counter are accessed directly, they need a matching *hyd_access* call from above
  - Arguments indexed with indirect indices are accessed indirectly, they need a matching *hyd_access* call from above
  - Any other arguments are should be invariant of the iteration, and will be passed as *op_arg_gbl* in OP2.

Using this information, the entire *do* loop using *hyd_par_loop* is replaced with a call to *op_par_loop*. The existing code passes Fortran arrays through the call tree, and refactoring that to pass OP2's opaque datatypes would be exceedingly difficult to do automatically, therefore we introduced new Fortran 90 interfaces that can take data pointers, and then match them up with *op_dat*s internally. OP2 internally makes a copy of the original data and deallocates the original pointer, so Hydra is passing pointers to invalid memory – but these pointers are never dereferenced in OP2. Hydra itself also does not dereference these outside of computational loops.

Third, we need to modify how global scope constants are declared and set. The OPlus version of Hydra uses common blocks, but CUDA Fortran does not support these. Instead, we place all globals into an F90 module, which is then used. To convert common blocks to be used in OP2, we have three fully automated steps:

1) Parse include files with common blocks in them, and generate the F90 module with matching variables.
2) Replace all occurrences of these files being included with *use* statements.
3) Scan all files for expressions that modify these variables, and insert *op_decl_const* API calls – which now may be called redundantly but that does not cause correctness issues, only minor performance issues.

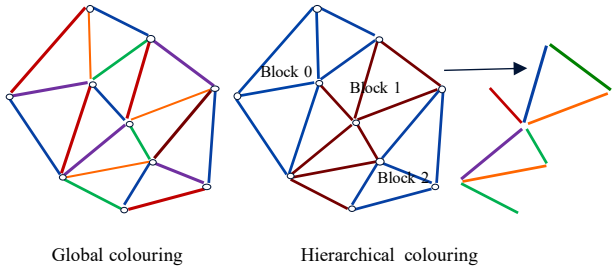

Fig. 5. OP2 colouring schemes

Finally, we need to place the appropriate set, mapping, and dataset declaration calls in the initialization phase of the code. Considering the presence of equivalent calls to OPlus, and that all such calls are in a single source file, this is a relatively simple process. Because the sets, mappings, and datasets concepts in OP2 and OPlus are fundamentally the same, when running sequentially (single process, no MPI), the two libraries are interoperable (since they both use the same data layout for mappings and datasets). This made incremental development and debugging possible, which was very helpful during the development process as issues had to be located and fixed.

With these steps, the conversion of Hydra from the OPlus API to the OP2 API is complete. The code can be compiled and linked against OP2's libraries and run using a generic implementation of *op_par_loop* to check for correctness.

V. OP2 PARALLELISATIONS

Once the conversion to the OP2 API was complete, we had a single source code that was seemingly sequential – the parallelism is implicit in OP2's abstraction. This allows OP2 to automatically manage data movement and parallelisation, which we discuss briefly in this section.

OP2 has two largely independent ways of parallelising execution; distributed memory parallelism which is implemented in the backend libraries, and shared memory parallelism, which is done using a combination of source-to-source translation and backend libraries. Distributed memory parallelism uses standard techniques; graph partitioning, creating and exchanging halo regions. OP2 uses the owner-execute strategy, where all calculations required for computing the new dataset values on a set element owned by the given process are done by that process, which along the boundaries involves redundant computations (executing a set element, owned by an adjacent process, which indirectly updates an owned set element). The details are documented in [21], [22]. Compared to OPlus, OP2 adds support for further graph partitioners (PT-Scotch [23]).

Support for shared memory parallelism is new in OP2 – OPlus had no such functionality, and simple parallelisation using compiler directives was not an option because of race conditions in unstructured meshes where multiple iterations may update the same value indirectly (e.g. edges updating vertices). OP2 handles race conditions by way of using execution plans – a runtime algorithm that calculates a safe order of execution of the iteration set, and the generated source code that executes this plan. There are several such execution strategies:

1) Hierarchical colouring (Figure 5): The mesh is broken up into blocks, the blocks themselves are coloured based on potential data races (this is used to assign blocks to OpenMP threads or CUDA thread blocks). Blocks of the same colour are executed in parallel. Elements within the block are then also coloured (this is used to assign elements to threads in a CUDA thread block and to synchronize between them).
2) Global colouring (Figure 5): All the elements in the mesh are coloured based on potential data races. Elements of the same colour are executed in parallel.
3) Atomics: when the indirect operation is an increment (as is the case in most Hydra indirect updates), data races can be handled using atomic increment operations.

Results with hierarchical parallelism were reported for Hydra in [7], but results with the other two strategies have not been published before. Depending on the choice of parallel implementation (OpenMP, CUDA, OpenACC) and the target architecture, some of these execution schemes cannot be used (e.g. OpenMP and both levels of hierarchical parallelism) or perform very poorly (e.g. atomics on CPUs). Hierarchical parallelism has the advantage of good data locality within blocks, but a more complex execution scheme. Global colouring is very simple, but has almost no data reuse. Atomics are not as widely applicable, and their performance varies with the target architecture, though on the latest NVIDIA V100 GPUs we will show that these perform the best.

Another key optimisation on GPUs is changing the data layout from an Array-of-Structs to the Structure-of-Arrays layout. This is done for each dataset that has multiple values per set element (most datasets in Hydra have 6-7, some up to 25). The conversion is done when data is uploaded from the CPU, and it also requires modifying array accesses in the subroutines passed to *op_par_loop*s. The code generator parses the input source code to find the implementation of these subroutines (which is possible due to globally unique names and no overloading in F77), then parses the list of formal arguments, matches them with arguments to the *op_par_loop* call to identify which require layout conversion, then uses regular expressions to change array indexing to SoA. A similar process is used for the atomics execution scheme; the code generator locates $a = a+b$ expressions where a is an indirectly incremented argument, and swaps it for a call to $atomicAdd$. Thanks to the coding conventions and the limitations of Fortran 77, these simple transformations could be done with regular expressions reliably.

Another key challenge in code generation and the manipulation of the implementations of subroutines is that many call further subroutines, and so on - for each of these calls the code generator locates implementation of the called subroutine and copies it into the F90 module created for the given *op_par_loop*. Argument lists of subroutine calls are then parsed and AoS to SoA or atomic access transformations are

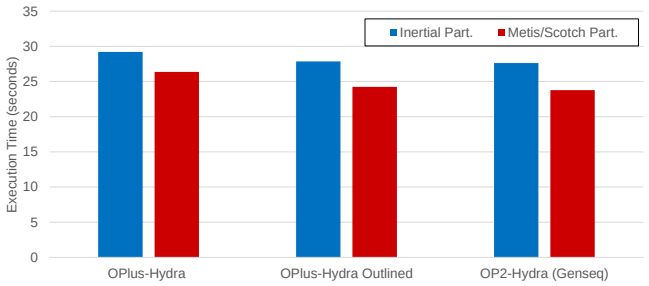

Fig. 6. Baseline performance of Hydra with OPlus and OP2 on a pair of Xeon Gold 6252 CPUs

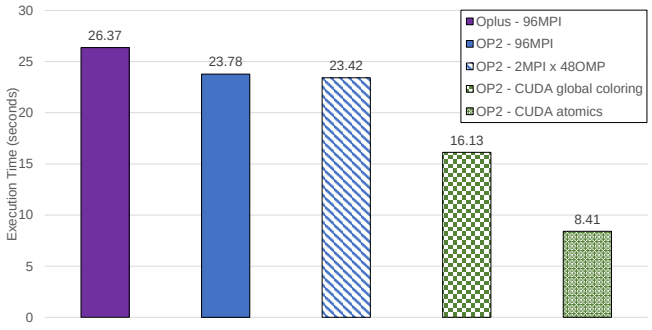

Fig. 7. Performance of Hydra with pure MPI, hybrid MPI+OpenMP, and CUDA on a pair of Xeon Gold 6252 CPUs and an NVIDIA V100 GPU

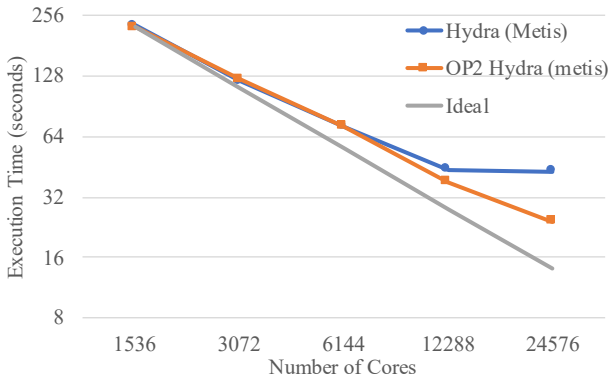

Fig. 8. Strong scaling a NASA Rotor 37 300M edge problem on ARCHER

propagated to the subroutines themselves. This does result in some code duplication and increased compilation times, but it is safe because the duplicates are now in F90 modules and not in a global namespace. Any updates to the underlying source code triggers a fresh source-to-source translation step which keeps these duplicates consistent.

VI. Performance

In this section we report on the performance of Hydra, running a NASA Rotor37 benchmark testcase. On a single node, we use a mesh with 2.1 millon edges and 4 multigrid levels, running for 20 time-marching iterations – which is much shorter than production runs, but is representative of relative runtimes.

First, Figure 6 shows the performance of the original unmodified code (jm51) that uses OPlus, then the version where all parallel loops are outlined (jm51_spd), and finally the OP2 version which only uses MPI distributed parallelism, the generated code is sequential on each process (genseq). We evaluate both the default partitioner (Inertial) as well as ParMetis for OPlus, and PT-Scotch for OP2 (which performed better than ParMetis). The benchmark was run on a dual-socket Intel(R) Xeon(R) Gold 6252 (Cascade Lake generation) system with 2x24 cores and Hyperthreading enabled (total of 96 MPI processes). The code was compiled with the Intel Compilers version 2018. It is clear that neither outlining, nor the conversion to OP2 degrades performance – rather they slightly improve it; OP2 is 6% faster with the Inertial partitioner, and 11% faster with PT-Scotch than OPlus with ParMetis.

Subsequently, we evaluate various shared-memory parallelisations that are now available; hybrid MPI+OpenMP on the same Xeon Gold server CPU and an NVIDIA V100 PCI-e GPU – the latter with the CUDA 10.2 and PGI 20.4 compilers. Figure 7 shows the results with shared memory parallelisations. The hybrid MPI+OpenMP currently only slightly outperforms pure MPI despite less time spent in MPI communications; this is because the relatively large number of colours required to colour the blocks (the first step of hierarchical colouring), which then creates a load imbalance between OpenMP threads. With a larger mesh size this is improved, but at this time we did not have such a mesh available. The CUDA implementation is similarly affected by underutilisation when using colouring, but less so when using atomics – the V100 GPU is 2.8× faster than the dual socket CPU.

We also evaluate the strong scalability on the UK National Supercomputer ARCHER [24], a Cray XC30 machine. An ARCHER node consists of two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series processors (24 cores in total), each core can support 2 hardware threads (Hyperthreads). The nodes are interconnected by a Cray Aries interconnect in a Dragonfly topology. The code was compiled using Intel Compilers, ifort version 17.0.0 20160721, and Intel MPI. Figure 8 shows scaling with a 300M edge mesh, comparing pure MPI versions of the OPlus and OP2 Hydra – we demonstrate improved scaling due to improvements to OP2's MPI communications: halo exchanges of multiple datasets and reductions of multiple variables in the same parallel loop are combined in OP2, whereas these are done separately in OPlus.

VII. Conclusions

In this paper, we have described our efforts to modernise an industrial CFD application that has been in development for over 20 years. After describing the fundamental layout of computations in the legacy and the modern version, we presented a number of steps – some that had to be done by hand, but most automated – that carry out the modernisation. A key requirement was to make it as easy to repeat as possible

so as to trivialise parallel development of the two versions and simplify the adoption into the larger workflow – initially for the academic ASiMoV project, and later for Rolls-Royce themselves. Since the development of the code conversion scripts, Rolls-Royce has released a major update to the OPlus Hydra codebase, which we were able to convert to OP2 in two days, which shows that our conversion process is indeed reasonably streamlined.

While the conversion scripts used are only directly applicable to the Hydra application, some of the methods apply more generally as well; numerous refactoring steps can be carried out using scripts and regular expressions, without having to use a full compiler stack to carry out the conversion.

We demonstrated that the modernised code can be run with shared memory parallel implementations as well, and thus can exploit hardware such as GPUs. The modernised OP2 version is also shown to be no worse in performance under identical testing scenarios, and up to 3× when running on GPUs.

ACKNOWLEDGMENT

This research is supported by Rolls-Royce plc and by the UK Engineering and Physical Sciences Research Council (EPSRC): (EP/S005072/1 - Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems - ASiMoV). Gihan Mudalige was supported by the Royal Society Industry Fellowship Scheme(INF/R1/1800 12). István Reguly was supported by National Research, Development and Innovation Fund of Hungary, project PD 124905, financed under the PD_17 funding scheme. This research has been carried out partly in the project Thematic Research Cooperation Establishing Innovative Informatic and Info-communication Solutions, which has been supported by the European Union and co-financed by the European Social Fund under grant number EFOP-3.6.2-16-2017-00013.

REFERENCES

[1] P. I. Crumpton and M. B. Giles, "Multigrid Aircraft Computations Using the OPlus Parallel Library," *Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers*, vol. -, pp. 339–346, 1995, A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, (eds.), North-Holland, 1996.

[2] J.-D. M. P. Moinier and M. Giles, "Edge-based multigrid and preconditioning for hybrid grids," *AIAA Journal*, vol. 40, pp. 1954–1960, 2002.

[3] M. C. M.C. Duta, M.B. Giles, "The harmonic adjoint approach to unsteady turbomachinery design," *International Journal for Numerical Methods in Fluids*, vol. 40, pp. 323–332, 2002.

[4] 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.

[5] L. Lapworth, "The Challenges for Aero-Engine CFD," Sept, 2008, invited Lecture, ICFD 25th Anniversary Meeting, Oxford, UK. www.icfd.rdg.ac.uk/ICFD25/Talks/LLapworth.pdf.

[6] "EPSRC Prosperity Grant - ASiMoV," 2018. [Online]. Available: https://app.dimensions.ai/details/grant/grant.7828967

[7] 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," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1265–1278, 2016.

[8] C. Bertolli, A. Betts, N. Loriant, G. R. Mudalige, D. Radford, M. B. Giles, and P. H. J. Kelly, "Compiler Optimizations for Industrial Unstructured Mesh CFD Applications on GPUs," in *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC'12)*, Sept 2012.

[9] K. Komatsu, R. Egawa, S. Hirasawa, H. Takizawa, K. Itakura, and H. Kobayashi, "Migration of an atmospheric simulation code to an openacc platform using the xevolver framework," in *2015 Third International Symposium on Computing and Networking (CANDAR)*, 2015, pp. 515–520.

[10] H. Takizawa, S. Hirasawa, Y. Hayashi, R. Egawa, and H. Kobayashi, "Xevolver: An xml-based code translation framework for supporting hpc application migration," in *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–11.

[11] R. Suda, H. Takizawa, and S. Hirasawa, "Xevtgen: Fortran code transformer generator for high performance scientific codes," *International Journal of Networking and Computing*, vol. 6, no. 2, pp. 263–289, 2016.

[12] H. Murai, M. Sato, M. Nakao, and J. Lee, "Metaprogramming framework for existing hpc languages based on the omni compiler infrastructure," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, 2018, pp. 250–256.

[13] A. RIKEN, "University of tsukuba. omni compiler project."

[14] I. Kanamori and H. Matsufuru, "Practical implementation of lattice qcd simulation on intel xeon phi knights landing," in *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, 2017, pp. 375–381.

[15] T. Boku, K. Ishikawa, Y. Kuramashi, and L. Meadows, "Mixed precision solver scalable to 16000 mpi processes for lattice quantum chromodynamics simulations on the oakforest-pacs system," in *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, 2017, pp. 362–368.

[16] R. Mathur, H. Matsuoka, O. Watanabe, A. Musa, R. Egawa, and H. Kobayashi, "A case study of memory optimization for migration of a plasmonics simulation application to sx-ace," in *2015 Third International Symposium on Computing and Networking (CANDAR)*, 2015, pp. 521–527.

[17] D. A. Burgess, P. I. Crumpton, and M. B. Giles, "A Parallel Framework for Unstructured Grid Solvers," in *Computational Fluid Dynamics'94:Proceedings of the Second European Computational Fluid Dynamics Conference*, S. Wagner, E. Hirschel, J. Periaux, and R. Piva, Eds. John Wiley and Sons, 1994, pp. 391–396.

[18] 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.

[19] C. Bertolli, A. Betts, G. R. Mudalige, M. B. Giles, and P. H. J. Kelly, "Design and Performance of the OP2 Library for Unstructured Mesh Applications," ser. Euro-Par 2011 Parallel Processing Workshops, Lecture Notes in Computer Science, 2011.

[20] The HDF Group. (2000-2010) Hierarchical data format version 5. [Online]. Available: http://www.hdfgroup.org/HDF5

[21] M. Giles, G. Mudalige, and I. Reguly. (2010-2020) OP2 MPI Developer Guide. [Online]. Available: https://op-dsl.github.io/docs/OP2/mpi-dev.pdf

[22] 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.

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

[24] "UK National Supercomputing Service - ARCHER," 2020. [Online]. Available: https://www.archer.ac.uk/about-archer/