Acceleration of a Full-scale Industrial CFD Application with OP2

ARTICLE *in* IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS · JULY 2015

Impact Factor: 2.17 · DOI: 10.1109/TPDS.2015.2453972

7 AUTHORS, INCLUDING:

I.Z. Reguly

Pázmány Péter Catholic University

15 PUBLICATIONS **41** CITATIONS

SEE PROFILE

Gihan R Mudalige

University of Oxford

37 PUBLICATIONS **186** CITATIONS

SEE PROFILE

Adam Betts

Imperial College London

27 PUBLICATIONS **107** CITATIONS

SEE PROFILE

Paul Kelly

Imperial College London

181 PUBLICATIONS **1,658** CITATIONS

SEE PROFILE

Acceleration of a Full-scale Industrial CFD Application with OP2

I.Z. Reguly, G.R. Mudalige, C. Bertolli, M.B. Giles, A. Betts, P.H.J. Kelly and D. Radford

Abstract—Hydra is a full-scale industrial CFD application used for the design of turbomachinery at Rolls Royce plc., capable of performing complex simulations over highly detailed unstructured mesh geometries. Hydra presents major challenges in data organization and movement that need to be overcome for continued high performance on emerging platforms. We present research in achieving this goal through the OP2 domain-specific high-level framework, demonstrating the viability of such a high-level programming approach. OP2 targets the domain of unstructured mesh problems and enables execution on a range of back-end hardware platforms. We chart the conversion of Hydra to OP2, and map out the key difficulties encountered in the process. Specifically we show how different parallel implementations can be achieved with an active library framework, even for a highly complicated industrial application and how different optimizations targeting contrasting parallel architectures can be applied to the whole application, seamlessly, reducing developer effort and increasing code longevity. Performance results demonstrate that not only the same runtime performance as that of the hand-tuned original code could be achieved, but it can be significantly improved on conventional processor systems, and many-core systems. Our results provide evidence of how high-level frameworks such as OP2 enable portability across a wide range of contrasting platforms and their significant utility in achieving high performance without the intervention of the application programmer.

Index Terms—Unstructured Mesh Applications, Domain Specific Language, Active Library, OP2, OpenMP, GPU, CUDA, CFD

1 Introduction

High Performance Computing (HPC) is currently experiencing a period of enormous change. Within the last decade, further increase of clock frequencies and in turn higher performance was severely curtailed due to the rapid increase in energy consumption. This phenomenon, commonly referred to as the end of Dennard's scaling, has become significant as we reach the physical limits of the current CMOS based microprocessor technologies. The only clear direction in gaining further performance improvements now appears to be through increased parallelism, where multiple processor units are utilized to increase throughput. As a result, modern and emerging microprocessors feature multiple homogeneous cores optionally augmented with many simpler processing cores on a single piece of silicon; ranging from a few cores to thousands, depending on the complexity of individual processing elements. Traditional CPUs now come with 4-18 cores and wide vector processing units (AVX), and there is an emergence of co-processors, such as NVIDIA GPUs or the Intel MIC with 16-64 functional units, each with long vector processing capabilities.

In light of these developments, an application developer faces a difficult problem. Optimizing an application for a target platform requires more and more low-level hardware-specific knowledge and the resulting code is increasingly difficult to maintain. Additionally, adapting to new hardware may require a major re-write, since optimizations and languages vary widely between different platforms. At the same time, there is considerable uncertainty about which platform to target: it is not clear which approach is likely to "win" in the long term. Application developers would like to benefit from the performance gains promised by these new systems, but are concerned about the software development costs involved. Furthermore, it can not be expected of domain scientists to gain platform-specific expertise in all the hardware they wish to use. This is especially the case with industrial applications that were developed many years ago, often incurring enormous costs not only for development and maintenance but also for validating and maintaining accurate scientific outputs from the application. Since these codes usually consist of tens or hundreds of thousands of lines of code, frequently porting them to new hardware is infeasible.

Recently, *Active libraries* [1] and *Domain Specific Languages* (DSLs) have emerged as a pathway pointing to a solution to these problems. The key idea is to allow scientists and engineers to develop applications by providing higher-level, abstract constructs that make use of domain specific knowledge to describe the problem to be solved. Active libraries look like conventional software libraries that rely on an API, but at the same time, appropriate code generation and compiler support is provided to generate platform specific code, from the higher-level source, targeting different hardware with tailored optimizations. Within such a setting a separate lower implementation level is created to provide opportunities for the library developers to apply radically aggressive and platform specific optimizations when implementing the required solution on various hardware platforms. At the same time, this strategy frees up the application developers to focus on the problem to be solved. The correct abstraction will pave the way for easy maintenance of a higher-level ap-

- *I.Z. Reguly, G.R. Mudalige and M.B. Giles are with the Oxford e-Research Centre at the University of Oxford, 7, Keble Road, Oxford OX1 3QG, UK. Email: {istvan.reguly, gihan.mudalige}@oerc.ox.ac.uk, mike.giles@maths.ox.ac.uk.*
- *I. Z. Reguly is also with PPCU ITK, Budapest, Hungary*
- *C. Bertolli (Current address) is with the IBM TJ Watson Research Centre, New York, USA. Email:cbertol@us.ibm.com.*
- *A. Betts and P.H.J. Kelly are with the Dept. of Computing, Imperial College London UK. Email:{a.betts,p.kelly}@imperial.ac.uk.*
- *D. Radford is with Rolls Royce plc. Derby UK. Email: David.Radford@rolls-royce.com*

Manuscript received XXX

plication source with near optimal performance for various platforms and make it possible to easily integrate support for any future novel hardware. In contrast, a DSL either defines its own language, or embeds into a host language, extending it, therefore it requires advanced compilers and development tools. Even though much research [2], [3], [4], [5], [6] has been carried out on the subject, there has been little conclusive evidence so far, demonstrating the viability of such an approach in developing full scale industrial applications. Indeed, it has been the lack of such an exemplar that has made these high level approaches confined to university research labs and not a mainstream HPC software development strategy.

In this paper we focus on the industrial application Hydra, used at Rolls Royce plc. for the simulation of turbomachinery components of aircraft engines. Hydra is a highly complex and configurable CFD application, capable of accommodating different simulations that can be applied to any mesh. Its initial development was carried out over 15 years ago, but continues to be actively maintained and optimized. Simulations implemented in Hydra are typically applied to large meshes, up to tens of millions of edges, with execution times ranging from a few minutes up to a few weeks. Hydra uses a design based on a domain specific abstraction for the solution of unstructured mesh problems; through a classical software library called OPlus (Oxford Parallel Library for Unstructured Solvers) [7] that targeted execution on a cluster of single-threaded CPUs by providing MPI halo exchange functionality.

OP2 ([8], [9], [10], [11]) is the successor of OPlus, and adopts an active-library approach; a single application code written using the OP2 API can be transformed (through source-to-source translation tools) into multiple parallel implementations which can then be linked against the appropriate parallel library (e.g. OpenMP, CUDA, MPI, OpenCL etc.) enabling execution on different back-end hardware platforms. It is due to this automatic code generation layer that we call OP2 (and similar frameworks) an active library as oppose to a classical software library. The generated code and the OP2 platform specific back-end libraries are highly optimized utilizing the best low-level features of a target architecture to make an OP2 application achieve high performance including high computational efficiency and minimized memory traffic. In previous works, we have presented OP2's design and development [8], [9] and its performance on heterogeneous systems [10], [12] on simpler benchmarks, and demonstrated considerable performance gains on a diverse set of hardware.

In this paper we chart the conversion of Hydra from its original version, based on OPlus, to one that utilizes OP2, and present key development and optimization strategies that allowed us to gain high performance on modern parallel systems. Specifically, we make the following contributions:

- *Deployment*: We present the conversion of Hydra to utilize OP2, mapping out the key difficulties encountered in the conversion of the application which, was designed and developed over 15 years ago, to OP2. Our work demonstrates the clear advantages in developing maintainable, future-proof and performant applications through a *high-level* abstraction approach.
- *Optimizations*: We present key optimizations to OP2, both existing and new ones, that incrementally allowed Hydra to gain near optimal performance on modern parallel systems, including conventional multi-core processors, many-core accelerators such as GPUs as well their heterogeneous combinations. The optimizations, while radically different across different platforms under study, were easily applied through OP2, demonstrating portability and increased developer productivity.
- *Performance*: The performance of Hydra is analyzed on a range of different hardware platforms, by studying metrics such as runtime, scalability and achieved bandwidth. The performance is compared to that of the original version of Hydra, contrasting the key optimizations that lead to performance differences. Hardware systems studied include a large-scale distributed memory Cray XE6 system, and a distributed memory Tesla K20 GPU cluster interconnected by QDR InfiniBand. The OP2 design choices and optimizations are explored with quantitative insights into their contributions to performance on these systems. Additionally, performance bottlenecks of the application are isolated by breaking down the runtime and analyzing the factors constraining overall performance.

Our work demonstrates how OP2 can be used to develop large-scale industrial applications, and that in the same environment the performance is on par with the original. Furthermore, the new code is capable of outperforming it with platform specific optimizations for modern multi-core and accelerator systems. Re-enforcing our previous findings, this research demonstrates that an application written once at a high-level using the OP2 framework is easily portable across a wide range of contrasting platforms, and is capable of achieving high performance without the intervention of the application programmer.

The rest of this paper is organized as follows: Section 2 introduces the Hydra CFD application; Section 3 discusses the OP2 abstraction, API and code generation process; Section 4 presents the transformations made to Hydra enabling it to utilize OP2. Section 5 details the iterative process that we went through to make sure that the OP2 version matches the performance of the original under identical circumstances. Section 6 presents a number of optimizations applied by the OP2 library that improve upon the performance, still using the same MPI parallelization approach. Section 7 shows how OP2 enables execution on multi-core and many-core platforms and presents a number of optimizations that help achieve high performance. Section 8 discusses OP2's strategy to scaling to large-scale systems and studies strong and weak scaling performance. Section 9 presents preliminary results from a hybrid CPU-GPU execution scheme. Section 10 briefly compares related work and Section 11 concludes the paper.

2 HYDRA

The aerodynamic performance of turbomachinery is a critical factor in engine efficiency of an aircraft, and hence is an important target of computer simulations. Historically, CFD simulations for turbomachinery design were based on structured meshes, often resulting in a difference between simulation results and actual experiments performed on engine prototypes. While the initial hypothesis for the cause

```
do while(op_par_loop(ncells, istart, iend))
  call op_access_r8('r',areac,1,ncells,null,0,0,1,1)
  call op_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

was the poor quality of the turbulence model, the use of unstructured meshes showed that the ability to model complex physical geometries with highly detailed mesh topologies is essential in achieving correct results. As a result, unstructured mesh based solutions are now used heavily to achieve accurate predictions from such simulations.

Significant computational resources are required for the simulation of these highly detailed (usually three-dimensional) meshes. The solution involves iterating over millions of elements (such as mesh edges and/or nodes) to reach the desired accuracy or resolution. Furthermore, unlike structured meshes, which utilize a regular stencil, unstructured mesh based solutions use the explicit connectivity between elements during computation. This leads to very irregular patterns of data access over the mesh, usually in the form of indirect array accesses. These data access patterns are particularly difficult to parallelize due to data dependencies resulting in race conditions.

Rolls Royce's Hydra CFD application is such a full-scale industrial application developed for the simulation of turbomachinery. It 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 behavior 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 multigrid and block-Jacobi preconditioning [13], [14]. 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's design is based on a domain specific abstraction for unstructured mesh based computations [7], where the solution algorithm is separated into four distinct parts: (1) sets, (2) data on sets, (3) connectivity (or mapping) between the sets and (4) operations over sets. This leads to an API through which mesh or graph problem solutions can be completely and abstractly defined. Depending on the application, a set can consist of nodes, edges, triangular faces, quadrilateral faces, or other elements. Associated with these sets are data (e.g. node coordinates, edge weights, velocities) and mappings between sets defining how elements of one set connect with the elements of another set. All the numerically intensive computations can be described as operations over sets. Within an application code, this corresponds to loops over a given set, accessing data through the mappings (i.e. one level of indirection), performing some calculations, then writing back (possibly through the mappings) to the data arrays.

The above API was implemented with the creation of a classical software library called OPlus [7], which provided a concrete distributed memory parallel implementation targeting clusters of single threaded CPUs. OPlus essentially carried out the distribution of the execution set of the mesh across MPI processes, and was responsible for all data movement while also taking care of the parallel execution without violating data dependencies. As an example, consider the code in Figure 1 of a loop in Hydra over a set of triangular cells.

The `op_par_loop` API call returns the loop bounds of the execution set, in this case triangular cells, while the calls to `op_access_r8` update the halos at the partition boundary by carrying out MPI communications. The actual numerical computation consists of distributing the area of each cell, `areac`, to the three nodes that make up the cell. This is achieved by looping over each cell and accessing the nodes making up each cell indirectly through the mapping `ncell` which points from the cell to its three nodes. The distributed memory parallel implementation of the above loop is carried out by partitioning and distributing the global execution set on to each MPI process. A single threaded CPU, assigned with one MPI process, will be sequentially iterating over its execution set, from *istart* to *iend* to complete the computation. However, data dependencies at the boundaries of the mesh partitions need to be handled to obtain the correct results. OPlus handles this by creating suitable halo elements such that contributions from neighboring MPI processes are received to update the halo elements. The implementation follows the standard MPI mesh partitioning and halo creation approach commonly found in distributed memory MPI parallelizations.

3 OP2 Library for Unstructured Grids

The ability to adapt to the rapidly changing hardware landscape motivated the development of OP2 [8], [9], a successor to OPlus. While the initial motivation was to enable Hydra to exploit multi-core and many-core parallelism, OP2 was designed from the outset to be a general high-level *active library* framework to express and parallelize unstructured mesh based numerical computations. OP2 retains the OPlus abstraction but provides a more complete high-level API (embedded in C/C++ and Fortran) to the application programmer, which for code development appears as an

```
! in file flux.F90
module FLUX
subroutine flux_user_kernel(x, ...)
real(8) x(3)
...
end subroutine
end module FLUX
```

```
! in file flux_app.F90
program flux_app
use OP2_Fortran_Reference
use OP2_CONSTANTS
use FLUX
...
call op_decl_set (nodes, ..)
call op_decl_map (..)
call op_decl_dat (x, ..)
...
call op_par_loop(nodes, flux_user_kernel,
&  op_arg_dat(x,-1,OP_ID,3,'r8',OP_READ),
&  ...)
...
end program flux_app
```

Fig. 3: Modules structure for a sequential build with OP2

API of a classical software library. However, OP2 uses a source-to-source translation layer to transform the application level source to different parallelizations targeting a range of parallel hardware. This stage gives the opportunity to provide the necessary implementation specific optimizations. The code generated for one of the platform-specific parallelizations can be compiled using standard C/C++/Fortran compilers to generate the platform specific binary executable.

3.1 The OP2 API

OP2 relies on the same abstraction as OPlus, describing the mesh as (1) a collection of sets, such as vertices or edges, (2) mappings between sets and (3) data on sets. Computations are then described as operations over a given set, accessing data either on the iteration set or through at most one level of indirection. A more detailed illustration of the OP2 API including the key API calls are given in Appendix A [15].

Any loop over a set in the mesh is expressed through the `op_par_loop`, an API call similar in purpose to the OPlus API call in Figure 1 for a loop over `cells`. However, OP2 enforces a separation of the per set element computation aiming to de-couple the declaration of the problem from the implementation, and introducing the restriction that the order of execution can not affect the end result, within machine precision. Thus, the same loop in Figure 1 can be expressed as detailed in Figure 2 using the OP2 API.

The `subroutine distr()` is called a *user kernel* in the OP2 vernacular. Simply put, the `op_par_loop` describes the loop over the set `cells`, detailing the per set element computation as an outlined "kernel" while making explicit indication as to how each argument to that kernel is accessed (OP_READ - read only, OP_INC - increment) and the mapping, `ncell` (cells to nodes) with the specific indices are used to indirectly accessing the data (`arean, areac`) held on each node. With this separation, the OP2 design, gives a significantly larger degree of freedom in implementing the loop with different parallelization strategies.

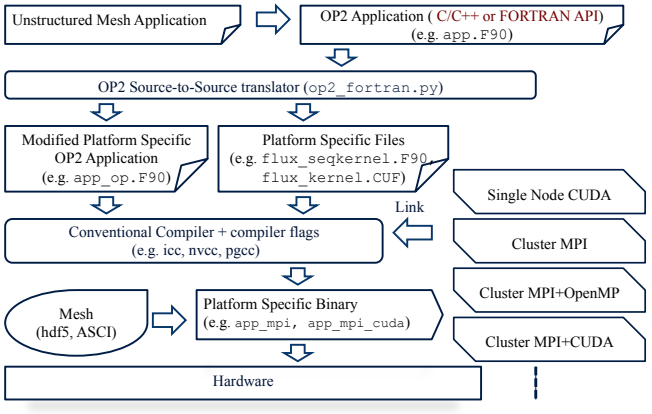


Fig. 4: OP2 build hierarchy

3.2 Development and Code Generation with OP2

An application written with the OP2 API in the above manner can be immediately debugged and tested for accuracy by including OP2's "sequential" header file (or its equivalent Fortran module if the application is written in Fortran). This, together with OP2's sequential back-end library, implements the API calls for a single threaded CPU and can be compiled and linked using conventional (platform specific) compilers (e.g. gcc, icc, ifort) and executed as a serial application. OP2's CPU back-end libraries are implemented in C. To support applications developed with the Fortran API, such as Hydra, the build process uses standard Fortran-to-C bindings, available since Fortran 2003. The Fortran application code passes a Fortran procedure pointer and arguments to the `op_par_loop`. The module structure for the sequential build is illustrated in Figure 3.

In this illustration, the application consists of an `op_par_loop` that calls a subroutine in a Fortran 90 module called `FLUX`. The module is in a separate file (`flux.F90`) and consists of the user kernel as a subroutine called `flux_user_kernel`. The Fortran application code passes the `flux_user_kernel` procedure pointer and arguments to the `op_par_loop`. The sequential implementation of the `op_par_loop` is provided in the OP2 back-end library in the `OP2_Fortran_Reference` module. The calls to `op_decl_set, op_decl_map` and `op_decl_dat` give OP2 full ownership of mappings and the data. OP2 holds them internally as C arrays and it is able to apply optimizing transformations in how the data is held in memory. Transformations include reordering mesh elements [16], partitioning (under MPI) and conversion to an array-of-structs data layout (for GPUs [9]). These transformations, and OP2's ability to seamlessly apply them internally is key to achieving a number of performance optimizations.

Once the application developer is satisfied with the validity of the results produced by the sequential application, parallel code can be automatically generated. The build process to obtain a parallel executable is detailed in Figure 4. In this case the API calls in the application are parsed by the OP2 source-to-source translator which will produce a modified main program and back-end specific code. These are then compiled using a conventional compiler (e.g. gcc, icc, nvcc) and linked against platform specific OP2 back-end libraries to generate the final executable. The mesh data to be solved is input at runtime. The source-to-source code

```
!in file flux_seqkernel.F90 or
!flux_kernel.F90 or flux_kernel.CUF
module FLUX_HOST
use FLUX
subroutine flux_host_kernel(arg1, ...)
real(8), dimension(:), pointer :: arg1Ptr
call c_f_pointer(arg1%CPtr, arg1Ptr, ...)
...
! platform specific parallelisation
do i = 0,nelems-1,1
  call flux_user_kernel(arg1Ptr(i*3+1,i*3+3),
  &  ...)
...
end subroutine
end module FLUX_HOST
```

```
!in file flux_app_op.F90
program flux_app_op
use OP2_FORTRAN_DECLARATIONS
use OP2_Fortran_RT_Support
use OP2_CONSTANTS
use FLUX_HOST
...
call flux_host_kernel(nodes,
  & op_arg_dat(x,-1,OP_ID,3,'r8',OP_READ),
  &  ...)
...
end program flux_app_op
```

Fig. 5: Modules structure for a parallel build with OP2

translator is written in Python and only needs to recognize OP2 API calls; it does not need to parse the rest of the code.

For each parallelization, the generated modified main program and back-end specific code follow the general module and procedure structure given in Figure 5. A modified application program is automatically generated in a new file called `flux_app_op.F90` which now uses the OP2 back-end specific libraries implemented in `OP2_FORTRAN_DECLARATIONS` and `OP2_Fortran_RT_Support` modules. In this case the `op_par_loop` API call is converted to a subroutine named `flux_host_kernel`, which is implemented in a module called `FLUX_HOST`. The parallel implementation of the `op_par_loop` in the `flux_host_kernel` subroutine differs for each parallelization (e.g MPI, OpenMP, CUDA etc.). As such the subroutine `flux_host_kernel` is placed in a separate file `flux_seqkernel.F90`, `flux_kernel.F90` or `flux_kernel.CUF` depending on whether we are generating a single threaded CPU cluster parallelization (with MPI), multi-threaded CPU cluster parallelization (with OpenMP and MPI) or cluster of GPUs parallelization (with CUDA and MPI). The `flux_host_kernel` subroutine in turn calls the user kernel `flux_user_kernel`. In the code illustration in Figure 5 we have shown the MPI only parallelization. The calls to `c_f_pointer` are necessary to bind the C pointers, which point to the C data and mapping arrays held internally by OP2, to Fortran pointers in order to pass them to the user kernel. This allows OP2 to use the user kernels without modification by the code generator. The combination of code generation and back-end functionality enables the users to maintain only a single high-level source code and rely on the OP2 library to automatically generate code for execution on different platforms. As future architectures and parallel programming abstractions appear, only the library developers will have to work on implementing support for them, the users will only have to re-generate code and re-compile.

OP2 currently supports automatic parallel code generation to be executed on (1) multi-threaded CPUs/SMPs using OpenMP, (2) single NVIDIA GPUs, (3) distributed memory clusters of single threaded CPUs using MPI, (4) cluster of multi-threaded CPUs using MPI and OpenMP and (5) cluster of GPUs using MPI and CUDA. Race conditions that occur during shared-memory execution on both CPUs (OpenMP) and GPUs (CUDA) are handled through multiple levels of coloring while for the distributed memory (MPI) parallelization, an owner-compute model [12] similar to that used in OPlus is implemented. More details on the various parallelization strategies and their performance implications can be found in previous papers [8], [9], [10], [12], and will be discussed in more detail in the subsequent sections, as we describe optimizations applied to them.

4 Deploying Hydra to OP2

The original Hydra, based on OPlus, is tailored for execution on distributed memory single threaded CPU clusters. However, as CPUs are increasingly designed with multiple processor cores and each with increasingly large vector units, simply assigning an MPI process per core may not be a good long-term strategy if the full capabilities of the processors are to be used. Moreover, based on experiences in attempting to exploit the parallelism in emerging SIMD-type architectures such the Xeon Phi or GPUs, relying only on coarse-grained message-passing may not be a viable or scalable strategy. Thus at least a thread-level, shared memory based parallelism is essential if Hydra is to continue to perform well on future systems. On the other hand further performance improvements appear to be obtainable with the use of accelerator based systems such as clusters of GPUs.

Directly porting the Hydra code to a multi-threaded implementation (e.g. with OpenMP or pthreads) is not straightforward. Consider parallelizing an OPlus loop, such as the one given in Figure 1, with OpenMP. Simply adding a `!omp pragma parallel` for the loop from `istart` to `iend` will not give correct results, due to the data races introduced by the indirect data accesses through the `ncell` mapping. If higher performance is required more optimizations tailored to OpenMP are needed. Thus, further substantial *implementation-specific* modifications would be required to achieve good thread-level parallelism. A much more significant re-write would be required if we were to get this loop running on a GPU, say using CUDA. Again the application code base would be changed with significant implementation specific code making it very hard to maintain. Porting to any future parallel architectures in this manner would yet again involve significant software development costs. Considering that the full Hydra source consists of over 300 loops written in Fortran 77, "hand-porting" in the above manner is not a viable strategy for each new type of parallel system. The design of the OP2 framework was motivated to address this issue.

The one-off conversion of all the Hydra parallel loops to OP2's API involved extracting "user-kernels" from each loop and then putting them in separate Fortran 90 modules.

TABLE 1: Benchmark systems specifications

System	Ruby (Development machine)	HECToR (Cray XE6)	Jade (NVIDIA GPU Cluster)
Node Architecture	2×Tesla K20c GPUs+ 2×6-core Intel Xeon E5-2640 2.50GHz	2×16-core AMD Opteron 6276 (Interlagos)2.3GHz	2×Tesla K20m GPUs+ Intel Xeon E5-1650 3.2GHz
Memory/Node	5GB/GPU + 64GB	32GB	5GB/GPU
Num of Nodes	1	128	8
Interconnect	shared memory	Cray Gemini	FDR InfiniBand
O/S	Red Hat Linux 6.3	CLE 3.1.29	Red Hat Linux 6.3
Compilers	PGI 13.3, ICC 13.0.1, OpenMPI 1.6.4, CUDA 5.0	Cray MPI 8.1.4	PGI 13.3, ICC 13.0.1, OpenMPI 1.6.4, CUDA 5.0
Compiler	-O2 -xAVX -Mcuda=5.0,cc35	-O3 -h fp3 -h ipa5	-O2 -xAVX -Mcuda=5.0,cc35

In this manner, the whole of Hydra was converted consistently to use only the OP2 API. The conversion process was relatively straightforward due to the similarities of the OPlus and OP2 APIs. Such a straightforward conversion may not have been possible if we were to convert a different unstructured mesh application to use OP2. However, we believe that such a development cost is imperative for most applications attempting to utilize the benefits of DSLs or Active Library frameworks, and it is not more costly than the one-off conversion of the code to an advanced parallel programming environment, such as CUDA. As we will show in this paper, the advantages of such frameworks far outweigh the costs, by significantly improving the maintainability of the application source, while making it possible to also gain near optimal performance and performance portability across a wide range of hardware.

A final step that was required to get the Fortran API working with OP2 was the handling of global constants. The original Hydra code with OPlus used common blocks to declare and hold global constants where their value is set during an initialization phase and then used throughout the code. With the move to use F90 in OP2, all constants were declared in a separate module `OP2_CONSTANTS`. For the GPU implementation, in order to separate constants held in the device (i.e. GPU) and the host CPU the `constant` keyword in the variable type qualifiers (alternatively `device` if the array is too large) and the `_OP2CONSTANT` suffix was added to the names of constant variable. In Hydra, all global variables that use common blocks are defined in a number of header files, thus we were able to implement a simple parser that extracts variable names and types, and generates the required constants module.

At this stage, the conversion of Hydra to utilize the OP2 framework was complete. The application was validated to check that the correct scientific results were obtained. The open question now was whether the time and effort spent in the conversion of Hydra to utilize an active library such as OP2 is justified. Specifically the key questions were: (1) whether the conversion to OP2 affected the performance of Hydra, compared to the original OPlus version, (2) what new capabilities can be enabled through OP2 that improve performance, (3) whether further performance gains are achievable with modern multi-core/many-core hardware, and what optimizations can be applied on different parallel platforms and (4) whether OP2's multi-level parallelization approach is scalable to thousands of cores and large problem sizes. The following sections are organized to answer each of these points, and at the same time show how the OP2 framework facilitates the deployment of such optimizations.

5 HYDRA: OPLUS VS. OP2

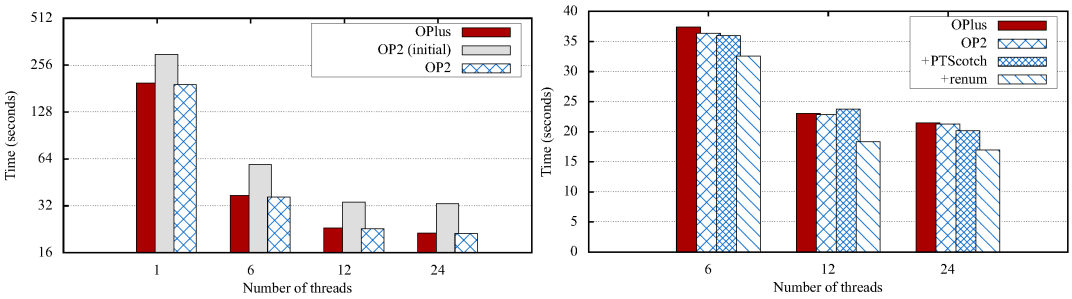
We begin by determining whether using a high-level approach such as OP2 is in any way detrimental to performance when compared to the hand-coded original; it is obviously critical to be able to perform a like-for-like comparison and demonstrate that the OP2 version can indeed match the performance of the original under identical circumstances. Therefore, we explore the single-thread and single node performance, and present the iterative design process that enabled us to match the performance of the original.

Our initial experimental system is a two socket Intel Xeon server named Ruby(see Table 1 column 1 for brief specifications). The configuration and input meshes of Hydra in these experiments model a standard application in CFD, called NASA Rotor37 [17]. It is a transonic axial compressor rotor widely used for validation in CFD. It is used for Rolls-Royce's HPC system procurements and validations, and as such is representative of other use cases of Hydra.

The mesh used for the single node performance benchmarking consists of 2.5 million edges. We use the non-linear solver configured to compute in double precision floating point arithmetic. Hydra can also be used with multi-grid simulations, but for simplicity of the performance analysis and reporting we utilize experiments with a single grid (mesh) level.

Figure 6a presents the performance of Hydra with both OPlus and OP2 on up to 12 cores (and 24 SMT threads) on the Ruby single node system using the message passing (MPI) parallelization. The partitioning routine used in both cases is a recursive coordinate bisection (RCB) mesh partitioning [18] where the 3D coordinates of the mesh are repeatedly split in the x, y and z directions respectively until the required number of partitions (where one partition is assigned to one MPI process) is achieved. The timings presented are for the end-to-end runtime of the main time-marching loop for 20 iterations. Usual production runs solving this mesh would take hundreds of iterations to converge.

We see that at first the OP2 version (noted as OP2 initial) was about 50% slower than the hand coded OPlus version. The generated code from OP2 appeared to be either missing a performance optimization inherent in the original code and/or the OP2 generated code and build structure introduced new bottlenecks. By simply considering the runtime on a single thread we see that even without MPI communications the OP2 (initial) version performed with the same slowdown. It was clear that some issue was affecting single threaded CPU performance. One plausible explanation was that the generated files and the separation of the user kernel is inhibiting function in-lining optimizations. By placing



(a) OPlus vs OP2 (MPI only) **(b)** OPlus vs OP2 (with PTSotch and renumbering)

Fig. 6: Single node performance on Ruby (2.5M edges, 20 iterations)

the user kernel and the code generated by OP2 setting up the pointers and looping over the execution set in the same compilation unit, performance improved over the OP2 (initial) version on average by about 25%.

To analyze the performance further, we ran Hydra through the Intel VTune profiler. The aim was to investigate any discrepancies in the performance of each subroutine that is called when executing an `op_par_loop`. As Hydra consists of more than 300 parallel loops, we focused on one of the most time consuming `op_par_loop`s called `edgecon`. The profiling revealed that a significant overhead (up to 40% of the total instruction count for the loop) occurs at the call to the user kernel; for example, the call to the `flux_user_kernel` subroutine provided by the user.

Further investigation revealed that the cause is a low level Fortran specific implementation issue [19] related to how arrays are represented internally in Fortran: the pointer results in an "assumed shape" array which is represented by an internal struct called a "dope vector" in Fortran. The dope vector contains the starting pointer of the array but also bounds and stride information. The extra information facilitates strided array sections and features such as bounds checking. However computing the memory address of an array element held in the struct causes a complex indexing calculation. Thus, directly using this array pointer and passing it as an argument to the `flux_user_kernel` subroutine causes the indexing calculation to occur for each and every iteration in the `do i = 0,nelems-1,1` loop.

The resolution is to force the complex index calculation to occur only once. After some experimentation, a modified module and subroutine structure was generated (see Appendix B [15]). It introduces a wrapper subroutine `flux_wrap` that is initially called by `flux_kernel` by passing the Fortran pointer. At this point the complex indexing calculation is carried out, just once. Then `flux_wrap` works with an "assumed shape array" and calls the user kernel `flux_user_kernel` with a simple offset that is cheap to calculate. To apply the modification to all parallel loops we simply modified the code generator to generate code with this new subroutine structure. None of the application level code with the OP2 API was affected. The performance of the resulting code is presented by the third bar in Figure 6a. At this point, the OP2 based Hydra application matches the performance of the original version, within 5% for each loop when executed by a single thread. These results by themselves provide enormous evidence of the utility of OP2, particularly (1) showing how an optimization can be applied to the whole industrial application seamlessly through code generation, reducing developer effort and (2) demonstrating that the same runtime performance as that of a hand-tuned code could be achieved through the high-level framework.

6 Systemic Optimizations

Having established that an implementation using a high-level approach can indeed match the performance of the original hand-coded version, we now focus on some of the improvements provided by the OP2 library that can be seamlessly integrated into execution. Firstly, the OP2 design allows the underlying mesh partitioner for distributing the mesh across MPI processes to be changed. During the initial input, OP2 distributes the sets, maps and data assuming trivial block partitioning, where consecutive blocks of set elements are assigned to consecutive partitions. As this trivial partitioning is not optimized for distributed memory performance, a further re-partitioning is carried out (in parallel) with the use of state-of-the-art unstructured mesh partitioners such as ParMetis [20] or PTScotch [21]. For the purposes of comparison with the original OPlus code, as shown in Figure 6a, we have also implemented the recursive coordinate bisection algorithm in OP2 allowing it to generate the exact same partitioning as the original OPlus.

Secondly, OP2 allows the ordering or numbering of mesh elements in an unstructured mesh to be optimized. The renumbering of the execution set and related sets that are accessed through indirections has an important effect on performance [16]: cache locality can be improved by making sure that data accessed by elements which are executed consecutively are close, so that data and cache lines are reused. OP2 implements a renumbering routine that can be called to convert the input data meshes based on the Gibbs-Poole-Stockmeyer algorithm in Scotch [21].

The results in Figure 6b show the effect of the above two features. The use of PTScotch resulted in about 8% improvement over the recursive coordinate bisection partitioning, on Ruby. However, renumbering resulted in about 17% gains for Hydra solving the NASA Rotor 37 mesh. Overall, partitioning with PTScotch gave marginally better performance than ParMetis (not shown here). In the results presented in the rest of the paper we make use of OP2's mesh renumbering capability, unless stated otherwise.

Breakdowns for some of the most time-consuming loops are shown in Table 2 when all the above optimizations are applied; observe how only 6 loops make up 75% of the total runtime. The table also shows the data requirements per set element; the number of double precision numbers read and written, either directly or indirectly, ignoring temporary

TABLE 2: Hydra single node performance with 24 MPI processes, showing data requirements per set element as number of double precision values read and written either directly or indirectly : 0.8M nodes, 2.5M edges, 20 iterations

Loop	Time	%	Direct R/W	Indirect R/W
accumedges	1.32	7.83	3/0	74/52
edgecon	1.08	6.56	3/0	68/48
ifluxedge	1.49	8.96	3/0	34/12
invjacs	0.17	1.03	27/27	0/0
srck	0.35	1.86	30/6	0/0
srcsa	1.32	7.60	34/6	0/0
updatek	0.98	5.83	53/6	0/0
vfluxedge	6.52	38.58	3/0	92/12
volap	0.43	2.58	29/24	0/0
wffluxedge	0.23	1.38	7/0	72/12
wvfluxwedge	0.21	1.24	4/0	45/6

memory requirements due to local variables. The results suggest that the memory system is under considerable pressure, especially when large amounts of data is accessed indirectly, but also shows that performance depends on the computations carried out within the user kernel. Since OP2 doesn't have a fully-fledged compiler, it is currently unable to generate SIMD vectorized code to further accelerate performance, however loops that do not indirectly increment data are candidates for compiler auto-vectorization, thus the appropriate compilers pragmas are generated in the code - currently without much effect on performance. A more complete investigation into achieving vectorization in OP2 have been presented in [10].

7 Multi-core and Many-core Execution

So far, we have only considered the MPI parallelization, which was already achieved with the original OPlus version of Hydra. However, one of the most important features of OP2 is that it enables execution on modern many-core hardware using the latest parallel programming approaches by automatically generating code from the high-level user application code. OP2 supports parallel code generation for execution on multi-threaded CPUs or SMPs using OpenMP and on NVIDIA GPUs using CUDA. The generated OpenMP code uses the same subroutine, module and file structure as the MPI-sequential code. At the time of writing, CUDA and NVIDIA's compiler, nvcc only supports code development in C/C++, so OP2 utilizes PGI's CUDA FORTRAN compiler to support code generated for the GPU via the Fortran API.

To exploit shared memory parallelization techniques, OP2 segments the execution set on a partition into blocks or mini-partitions and each mini-partition is assigned to an OpenMP thread (or a CUDA thread block) for execution in parallel [8], [12]. However, to avoid race conditions due to indirectly accessed data, the blocks are colored such that adjacent blocks are given different colors. When executing the computations per block, only blocks of the same color are executed in parallel; furthermore, on the GPU, a subsequent coloring of set elements within each block is necessary to avoid race conditions when one set element is assigned to one GPU thread. All these form an execution plan that is created for each loop when it is first encountered and then reused during subsequent executions.

7.1 OpenMP Execution

The use of a hybrid MPI+OpenMP programming approach on modern supercomputers is well established, and is often

TABLE 3: Hydra single node performance, 6 MPI x 4 OMP: 2.5M edges, 20 iterations

Loop	Time (sec)	GB/sec	% runtime
accumedges	1.46	28.57	7.99
edgecon	2.00	58.40	10.95
ifluxedge	1.88	48.88	10.32
invjacs	0.16	67.37	0.90
srck	0.47	81.72	2.57
srcsanode	1.34	31.62	7.38
updatek	0.94	68.67	5.14
vfluxedge	7.05	16.11	38.70
volapf	0.47	72.19	2.57
wffluxedge	0.26	25.24	1.41
wvfluxedge	0.26	16.81	1.41

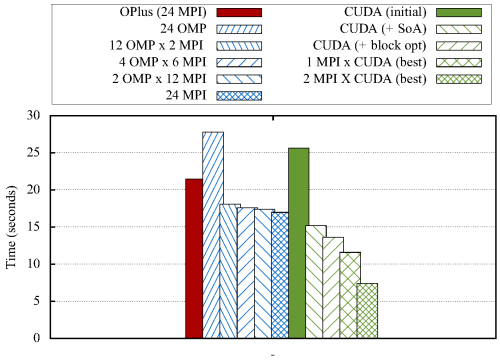


Fig. 7: OP2 Hydra Multi-/Many-core performance on Ruby (2.5M edges, 20 iterations)

utilized to reduce the overhead of MPI communications. Figure 7 presents the runtime performance of the OpenMP parallel back-end. The experiments varied from running a fully multi-threaded version of the application (OpenMP only), to a heterogeneous version using both MPI and OpenMP. We see that executing Hydra with 24 OpenMP threads (i.e. OpenMP only) resulted in significantly poorer performance than when using only the MPI parallelization, on this two socket CPU node. We have observed similar performance with the Airfoil CFD benchmark code [12] and shown it to be caused by the non-uniform memory access issues (NUMA [22], [23]). When executing in an MPI+OpenMP hybrid setting, processes and threads are pinned to specific sockets circumventing such problems.

With the shared memory parallelism execution scheme employed by OP2, we reason that the size of the block determines the amount of work that a given thread carries out uninterrupted. The bigger it is, the higher data reuse within the block with better cache and prefetch engine utilization. At the same time, some parallelism is lost due to the colored execution; only those blocks that have the same color can be executed at the same time by different threads, with an implicit synchronization step between colors. This makes the execution scheme prone to load imbalances, especially when the number of blocks with a given color is comparable to the number of threads that are available to execute them. The above two causes have given rise to the runtime behavior we see from each MPI and OpenMP combination in Figure 8. Evidence supporting these conclusions can be found by inspecting the number of blocks and number of colors used in each configuration (see Appendix C [15]) .

Finally we conclude that the above effects, particularly the lost parallelism with colored execution, combined with threading overheads [24], [25] may account for the slightly

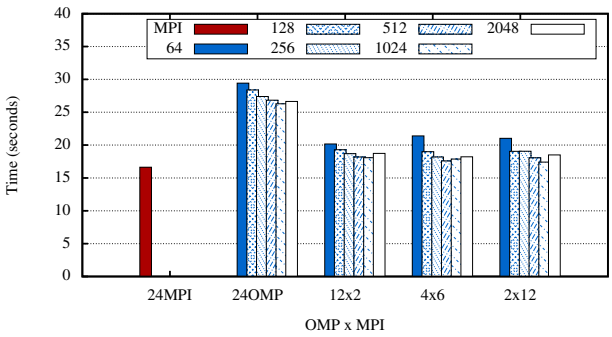


Fig. 8: Hydra OpenMP and MPI+OpenMP performance for different block sizes on Ruby (2.5M edges, 20 iterations)

worse performance of the hybrid MPI+OpenMP approach on a single node compared to the MPI only runtime. Indeed, many PDE codes written by threading experts [26] are found to run faster with flat MPI. This is partly to do with the overhead of software synchronization and parallel vs serial packing as compared to parallel access to NIC hardware contexts.

Table 3 details the achieved bandwidth for the most time consuming loops of Hydra with OpenMP (6 MPI $\times$ 4 OMP). These loops together take up about 90% of the total runtime. The achieved bandwidths are reported through code generated by OP2 for performance reporting. The bandwidth figure was computed by counting the useful amount of data bytes transferred from/to global memory during the execution of a parallel loop, accounting for data re-use within blocks, and dividing it by the runtime of the loop. The achieved bandwidth by a majority of the key loops on Ruby is over or close to 60% of the practical peak bandwidth of the Sandy Bridge processors (66.8 GB/s STREAM Triad, 85.2 GB/s theoretical peak). The best performing loops, such as `srck`, `updatek` and `volapf`, are direct loops that only access datasets defined on the set being iterated over, hence they achieve a very high fraction of the peak bandwidth, due to trivial access patterns to memory. Indirect loops use mapping tables to access memory, and they often require colored execution in order to avoid data races; both of these factors contribute to lower bandwidth achieved by loops such as `accumedges`, `ifluxedge` and `edgecon`. Additionally some loops are also compute and control-intensive, such as `vfluxedge`. Finally, loops over boundary sets, such as `wvfluxedge`, have highly irregular access patterns to datasets and generally much smaller execution sizes, leading to a lower utilization of resources. Our experiments also showed that the trends on achieved bandwidth utilization remain very similar to the observed results from previous work [8], [12] for the Airfoil benchmark application.

7.2 GPU Execution

The Ruby development machine contains two Tesla K20 GPUs, using which we investigate Hydra's performance on GPUs. Many aspects of the generated CUDA code had been optimized from our previous work [8], [9] targeting the older generation of NVIDIA GPUs based on their Fermi architecture. Thus, we re-evaluated the generated CUDA code targeting the Tesla K20 which are based on NVIDIA's latest Kepler architecture.

In a way similar to the code generation for the CPUs, we use Fortran to C bindings to call functions in the OP2 backend and to connect C pointers to Fortran pointers. The same plan construction that was described in previous papers [9] is retained. Previously, indirect data from GPU global memory were loaded into each thread block's shared memory space forming a local mini-partition. However, we observed that this staging of indirectly accessed data is not beneficial for Hydra contrary to the results we observed with the Airfoil benchmark [9] on previous-generation hardware. This is due to the fact that the large amounts of data moved by parallel loops in Hydra would require excessive amounts of shared memory, which in turn would severely degrade multiprocessor occupancy (the number of threads resident in a multiprocessor at the same time) and performance. Therefore, we eliminated the staging in shared memory by directly loading the data from global memory into SMs, and rely on the increased L2 and texture cache size to speed up memory accesses that have spatial and temporal locality. The performance achieved by the CUDA application generated by OP2 with such a parallel implementation is presented in Figure 7. However, this initial code on a single GPU ran about 45% slower than the best CPU performance on Ruby (2 CPUs, 24 MPI processes).

Running the generated CUDA code through the NVIDIA Visual Profiler revealed more opportunities for optimizations. It appeared that the switch to directly loading data from global memory (without staging in shared memory) has made most parallel loops in Hydra limited by poor memory access patterns. Further investigation showed that a high amount of cache contention is caused by the default data layout of `op_dat`s. This layout, called Array-of-Structs (AoS), was found to be the best layout for indirectly accessed data in our previous work [9] on Fermi GPUs. However without staging in shared memory it was damaging performance in Hydra as many of Hydra's `op_dat`s have a large number of components (dimensions) for each set element (typically related to the number of PDEs $\geq$ 6). Thus, adjacent threads are accessing memory that are far apart, resulting in high numbers of cache line loads (and evictions, since the cache size is limited). OP2 has the ability to effectively transpose these datasets and use a Struct-of-Arrays (SoA) layout, so when adjacent threads are accessing the same data components, they have a high probability of being accessed from the same cache line. This is facilitated by either the user annotating the code or by telling the OP2 framework to automatically use the SoA layout for datasets above a given dimension. The above optimization resulted in an increase of about 40% to the single GPU performance as shown in Figure 7.

To improve performance further, two other aspects of GPU performance were investigated. The goal was to allow as many threads to be active simultaneously so as to hide the latency of memory operations. The first option is to limit the number of registers used per thread. A GPU's SM can hold at most 2048 threads at the same time, but it has only a fixed number of registers available (65k 32-bit registers on Kepler K20 GPUs). Therefore kernels using excessive amounts of registers per thread decrease the number of threads resident on an SM, thereby reducing parallelism and performance. Limiting register count (through compiler flags) can be beneficial to occupancy and performance if the spilled registers can be contained in the L1 cache. Thus for the most time-

TABLE 4: Hydra GPU performance: 2.5M edges, 20 iterations

Loop	Time (sec)	GB/sec	% runtime
accumedges	2.07	13.87	19.30
edgecon	2.46	33.55	22.87
ifluxedge	1.03	71.90	9.59
invjacs	0.41	25.94	3.85
srck	0.34	108.09	3.20
srcsa	0.34	121.72	3.15
updatek	0.54	115.91	5.01
vfluxedge	2.32	53.82	21.62
volap	0.23	142.85	2.14
wffluxedge	0.11	31.55	1.00
wvfluxwedge	0.08	25.69	0.77

consuming loops in Hydra, we have manually adjusted register count limitations so as to improve occupancy.

The second consideration is to adjust the number of threads per block. Thread block size not only affects occupancy, but also has non-trivial effects on performance due to synchronization overheads, cache locality, etc. that are difficult to predict. The best thread block size for different parallel loops are stored them in a look-up table. This optimization can be carried out once for different hardware, but performance is unlikely to vary significantly when solving different problems. Together with the SoA optimization, the use of tuned block sizes and limited register counts provided a further 10% performance improvement.

The final best runtime on a single K20 GPU and on both the K20 GPUs on Ruby is presented in the final two bars of Figure 7. A single K20 GPU achieves about 1.8× speedup over the original OPlus version of Hydra on Ruby and about 1.5× over the MPI version of Hydra with OP2. However, considering the full capabilities of a node, the best performance on Ruby with GPUs (2 GPUs with MPI) is about 2.34× speedup over the best OP2 runtime with CPUs (2 CPUs, 24 MPI processes) and about 2.89× speedup over the best OPlus runtime with CPUs (2 CPUs, 24 MPI processes). Table 4 notes the achieved memory bandwidth utilization on a K20 GPU. A majority of the most time consuming loops achieve 20 - 50 % of the the practical peak performance (163 GB/s copy, out of 208 GB/s theoretical peak). Bandwidth utilization is particularly significant during the direct loops `srck`, `updatek` and `volapf`.

As it can be seen from the above effort, a range of low-level features had to be taken into account and a significant re-evaluation of the GPU optimizations had to be done to gain optimal performance even when going from one generation of GPUs by the same vendor/designer to the next. In our case the previous optimizations implemented for the Fermi GPUs had to be considerably modified to achieve good performance on the next generation Kepler GPUs. However, as the code was developed under the OP2 framework, radical changes to the parallel code could be easily implemented. In contrast, a directly hand-ported application would cause the application programmer significant difficulties to maintain performance for each new generation of GPUs, not to mention new processor architectures. This shows that the use of OP2 has indeed led to a near optimal GPU back-end that is significantly faster than the CPU back-end, with very little change to the original source code.

7.3 Fine-tuning execution parameters

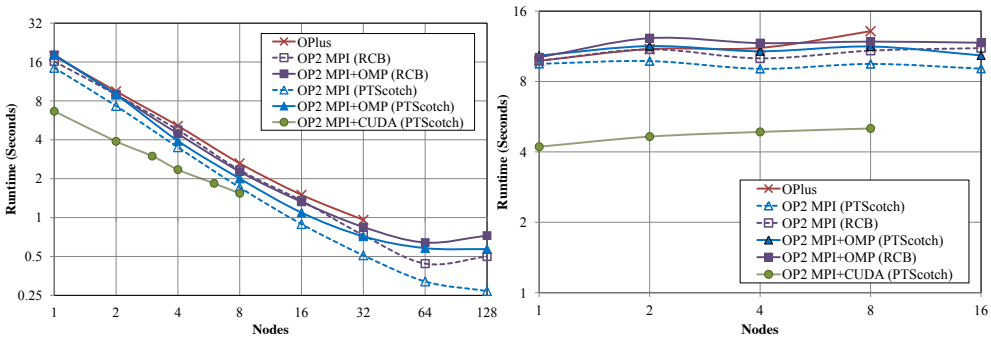
In the previous discussion of OpenMP and CUDA execution, a number of parameters came up that needed to be tuned to achieve the best performance. In case of OpenMP, the size of mini-partitions assigned to threads as well as the specific MPI-OpenMP process-thread combination affected performance. For GPU execution, there is an even wider range of parameters; mini-partition size, thread block size, and limitation of register usage for individual loops. In OP2, most of these parameters can be defined or over-ridden either at compilation time (register limitations) or at execution time (mini-partition size, block size, etc.). We use an auto-tuning tool called Flamingo [9] to explore the parameter space on a representative test case restricted in runtime in combination with OP2's built-in performance-reporting mechanisms, that give a timing breakdown for each computational loop, to select the best combination of parameters, some of the results are then fed back to an auxiliary program module that can be queried for the best combination of parameters for a loop with a given name.

8 Distributed Memory Scaling

The industrial problems simulated by Hydra require significantly larger computational resources than what is available today on single node systems. An example design simulation such as a multi-blade-row unsteady RANS (Reynolds Averaged Navier Stokes) computation where the unsteadiness originates from the rotor blades moving relative to the stators, would need to operate over a mesh with about 100 million nodes. Currently with OPlus, Hydra can take more than a week on a small CPU cluster to reach convergence for such a large-scale problem. Future turbomachinery design projects aim to carry out such simulations more frequently, such as on a weekly or on a daily basis, and as such the OP2 based Hydra code needs to scale well on clusters with thousands to tens of thousands of processor cores. Previous papers have explored OP2's strong and weak scaling on a benchmark problem [12], here we restrict our study and focus on the overall scaling performance we experience for Hydra. Table 1 lists the key specifications of the two cluster systems we use in our benchmarking. The first system, HECToR [27], is a large-scale proprietary Cray XE6 system which we use to investigate the scalability of the MPI and MPI+OpenMP parallelizations. The second system, JADE is a small local NVIDIA GPU (Tesla K20) cluster that we use to benchmark the MPI+CUDA execution. While during this study we could not attempt experiments at an even larger scale, other studies have demonstrated extreme scalability of similar unstructured mesh codes [28].

8.1 Strong Scaling

Figure 9(a) reports the run-times of Hydra at scale, solving the NASA Rotor 37 mesh with 2.5M edges in a strong-scaling setting. The x-axis represents the number of nodes on each system tested, where a HECToR node consists of two Interlagos processors, a JADE node consists of two Tesla K20 GPUs. The run-times (on the y-axis) are averaged from 5 runs for each node count. The standard deviation in run times was always less than 10%. MPI+OpenMP results were obtained by assigning four MPI processes per HECToR node, each MPI process consisting of eight OpenMP threads, in accordance with the NUMA architecture of the Interlagos processors [29]. On HECToR, we see that the overall scaling of Hydra with OP2 is significantly better than that with OPlus. OP2's MPI-only parallelization scales well up to 128



(a) Strong Scaling (2.5M edges) **(b)** Weak Scaling (0.5M edges per node)

Fig. 9: Scaling on HECToR (MPI, MPI+OpenMP) and Jade (MPI+CUDA) : 20 iterations

nodes (4096 cores). At 32 nodes (1024 cores) the MPI-only parallelization partitioned with PTScotch gives about 2x speedup over the runtime achieved with OPlus.

As with all message passing based parallelizations, one of the main problems that limits the scalability is the over-partitioning of the mesh at higher machine scale. This leads to an increase in redundant computation at the halo regions (compared to the non-halo elements per partition) and an increase in time spent during halo exchanges; the ratio of halo elements to interior elements on each partition reaches 27-40% (depending on the partitioner) at 4096 processes. It is important to note that at the largest scale, almost half of the total runtime was spent in global reductions. We expected MPI+OpenMP to perform better at larger machine scales as observed in previous performance studies using the Airfoil CFD benchmark [12], due to the smaller number of partitions and thus smaller ratio of halo elements; unfortunately the performance bottlenecks discussed in Section 7.1 for the single node system are worse at higher machine scales, on the HECToR system, as Figure 9(a) shows, pure MPI execution gives better scaling performance.

Comparing the performance on HECToR to that on the GPU cluster JADE, reveals that for the 2.5M edge mesh problem the CPU system gives better scalability than the GPU cluster. We believe that similar to the Airfoil code's GPU cluster performance [12], this comes down to GPU utilization issues: the level of parallelism during execution. Since the GPU is more sensitive to these effects than the CPU, where the former relies on increased throughput for speedups and the latter depends on reduced latency, the impact on performance is more significant due to reduced utilization at increasing scale. Along with the reduction in problem size per partition, the same fragmentation as we observed with the MPI+OpenMP implementation due to coloring is present. Colors with only a few blocks have very low GPU utilization, leading to a disproportionately large execution time. This is further complemented by the different number of colors on different partitions for the same loop, leading to faster execution on some partitions and then the idling at implicit or explicit synchronization points waiting for the slower ones to catch up.

8.2 Weak Scaling

Weak scaling of a problem investigates the performance of the application at both increasing problem and machine size. For Hydra, we generated a series of NASA rotor 37 meshes such that a near-constant mesh size per node (0.5M vertices) is maintained at increasing machine scale. The results are detailed in Figure 9(b). The largest mesh size benchmarked across 16 nodes (512 cores) on HECToR consists of about 8 million vertices and 25 million edges in total. Further scaling could not be attempted due to the unavailability of larger meshes at the time of writing.

With OPlus, there is about 8-13% increase in the runtime of Hydra each time the problem size is doubled. With OP2, the pure MPI version with PTScotch partitioning shows virtually no increase in runtime. Similar to strong scaling, the MPI-only parallelization performs about 10-15% better than the MPI+OpenMP version. The GPU cluster, JADE, gives the best runtimes for weak scaling, with a 4-8% loss of performance when doubling problem size and processes. It roughly maintains a 2× speedup over the CPU implementations at increasing scale.

The above scaling results give us considerable confidence in OP2's ability to give good performance at large machine sizes even for a complex industrial application such as Hydra. We see that the primary factor affecting performance is the quality of the partitions: minimizing halo sizes and MPI communication neighbors. For completeness Appendix D [15] provides further evidence and analysis of the distributed memory scaling performance of OP2 Hydra. These results illustrate that, in conjunction with utilizing state-of-the-art partitioners such as PTScotch, the halo sizes resulting from OP2's owner-compute design for distributed memory parallelization provide excellent scalability. We also see that GPU clusters are much less scalable for small problem sizes and are best utilized in weak-scaling executions.

9 Full Hybrid GPU-CPU Execution

Most related work published on many-core acceleration, and GPU acceleration in particular, focuses on migrating the entire code base to the GPU and then comparing performance with the CPU. However, modern GPU supercomputers, such as Titan at Oak Ridge NL, consist of roughly the same number of GPUs and CPU sockets, and often pricing is only calculated on a per-node basis. Thus, if an application only exploits the computational resources of the GPUs, then the CPUs are idling, even though they might have considerable computational power themselves; this is a waste of energy and resources. Several papers address this issue by employing different techniques, where the CPU and the GPU either have the same "rank", such as in the case of shared task-queues [30], or where the GPU computes on the bulk of the workload and the CPU handles the parts where the GPU would be underutilized, such as the boundary in domain decomposition systems [31], [32]. In

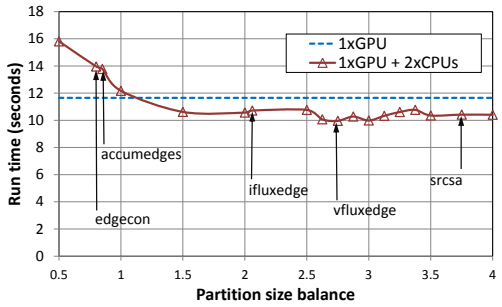


Fig. 10: Hybrid CPU-GPU performance at different load balancing values, marking perfect balance for different loops on Ruby (2.5M edges, 20 iterations) with ParMetis partitioning

this section we present preliminary results on OP2's support for what we call full-hybrid execution where both the CPUs and the GPUs on a node is used for mesh computations. Again, the possibility of seamlessly integrating such a fully-hybrid parallelization to Hydra is only possible due to the high-level abstraction approach implemented through OP2.

The natural approach to enable hybrid CPU-GPU execution in OP2 is to assign some processes to execute on the GPU and others to execute on the CPU. This hardware selection happens at runtime: on a node with N GPUs, the first N processes assigned to it pick up a GPU and the rest become CPU processes. To enable hybrid execution, the generated kernel files include code for execution with both MPI+CUDA and MPI+OpenMP, thus at runtime the different MPI processes assigned to different hardware can call the appropriate one.

The most important challenge with hybrid execution in general, not just for Hydra, is to appropriately load balance between different hardware so that both are utilized as much as possible. From our results in the previous sections, the importance of load balancing, even for executing computations the CPU only, was evident. Finding such a balance for simple applications where one computational phase (such as a single loop) dominates the runtime may not be difficult. One only needs to compare execution times on the CPU and the GPU separately and assign proportionally sized partitions to the two.

However, for an application such as Hydra consisting of several phases of computations, such a load balancing is not trivial: the performance difference between the CPU and the GPU varies widely for different loops, as shown in Table 3 and Table 4. For example the loop `vflux` is about 3 times faster on the GPU, but the loop `edgecon` is 25% faster on the CPU. Load balancing for each computational loop is infeasible as it would require repartitioning the mesh and transferring large amounts of data between different processes from one loop to another, losing any performance benefit it might offer. Thus in OP2 we have to come up with a static load balance upfront, which implies that some loops will be executed faster on the GPU than the CPU and vice versa, leading to the faster one waiting for the slower one to catch up whenever a halo exchange is necessary between them. This can severely restrict the potential performance gains expected from a fully hybrid execution.

We perform hybrid tests on the Ruby development machine, using a single GPU and both CPU sockets, each running OpenMP with 10 threads. Partitioning is carried out using the heterogeneous load balancing feature of ParMetis. Figure 10 shows performance while adjusting the static load balance between the work assigned to the CPU and the GPU. For example a partition size balance of 2.5 implies that the GPU executes a partition that is 5 times larger than a single CPU (i.e. 2.5 times larger than the combined size of the partitions assigned to both of the CPUs on Ruby). The first guess for the static load balance would be based on single GPU execution time and MPI+OpenMP execution time on Ruby, yielding a balance of about 1.5, giving a 9% performance improvement over single GPU runtime. An auto-tuning run for the value of the static load balance yields the data points in Figure 10, registering the execution times of different loops on the CPU and the GPU.

The figure marks load balance values where different loops are in perfect balance (i.e. the execution time on the GPU and the two CPUs are approximately the same); it is obvious that there is a big variation, but the trends show that it is best to shift the balance towards loops where the GPU has a significant speedup over the CPU, such as `ifluxedge`, `vfluxedge` or `srcsa`. This increases the performance gain from hybrid execution up to 15%.

10 Related Work

There is a large body of research on classical software libraries for unstructured mesh computations, such as PETSc [33]. Recently domain specific languages and high-level approaches have received a lot of attention; FEniCS [34], [35] for the solution of PDEs, Liszt [3] for unstructured meshes, and a large number of DSLs for structured meshes and stencil-based computations [4], [5], [36], which all take varying approaches to defining a language and providing programming tools and compilers to exploit domain-spcific knowledge and enable execution on different hardware. Recently the COSMO weather code from Switzerland was demonstrated to successfully use a similar domain-specific approach [37]. In contrast to these approaches, OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [2] programming model that separates the specification of a computational kernel with its parallel iteration space, from a declarative specification of how each iteration accesses its data.

In addition to Hydra, using OP2 we have so far developed a range of applications solving a wide variety of numerical algorithms. These include two test applications that implement a finite volume and a finite element algorithm [38], an unstructured mesh Lagrangian Hydrodynamics mini-app from the UK Mini-App Consortium called BookLeaf [38], [39], an academic shallow-water simulation code called VOLNA [40]. The OP2 API was sufficiently general to be applicable to such a range of applications without the need for any application specific modifications. For example, BookLeaf originally written in Fortran 90, once converted to use the OP2 API was code generated immediately to gain OpenMP and CUDA parallel code that included all of the optimizations that we have discussed in this paper for Hydra.

11 Conclusions

Rolls Royce's Hydra is a full-scale industrial CFD application currently in regular production use. Porting it to exploit multi-core and many-core parallelism presents a major challenge in data organization and movement requiring the

utility of a range of low-level platform specific features. The research presented in this paper illustrates how the OP2 high-level domain specific abstraction framework can be used to future-proof this key application for continued high performance on such emerging processor systems. We charted the conversion of Hydra from its original hand-tuned production version to one that utilizes OP2, and mapped out the key difficulties encountered in the process. Over the course of this process, OP2 enabled the application of increasingly complex optimizations to the whole code to achieve near optimal performance. The paper provides evidence of how OP2 significantly increases developer productivity in this task.

Performance results for the code generated with OP2 demonstrate that not only could the same runtime performance as that of the hand-tuned original production code be achieved, but it can be significantly improved on conventional processor systems as well as further accelerated by exploiting many-core parallelism. We see that OP2's MPI and MPI+OpenMP parallelizations achieve about 2 $\times$ speedup when strong-scaled and maintain 15-30% speedup when weak-scaled over the original implementation on a large distributed memory cluster system. Running on NVIDIA GPUs, OP2's CUDA implementation achieves about 1.5 to 3 $\times$ speedups over the Intel Sandy Bridge x86-64 server processors and maintains a similar performance advantage over the CPU cluster implementations when weak-scaling over a GPU cluster. We also demonstrate that executing Hydra on both the CPUs and GPUs in a fully-hybrid setting provides up to 15% speedup over the purely GPU execution and is primarily bound by load-balancing issues. However the cost and energy consumption of high-end GPUs is similar to that of a high-end server CPU.

Some of the key optimizations that affect all back-end implementations are the use of improved partitioning methods, mesh renumbering for improved cache locality and partial halo exchanges. Furthermore, for shared memory parallelism techniques, we have shown the importance of optimizing the mini-partition size so as to have a balance of parallelism and data locality, and we presented further techniques involving data layout transformations and the fine-tuning of resource usage to improve performance on the GPU.

We believe that the future of numerical simulation software development is in the specification of algorithms translated to low-level code by a framework such as OP2. Such an approach will, we believe, offer revolutionary potential in delivering performance portability and increased developer productivity. This, we predict, will be an essential paradigm shift for addressing the ever-increasing complexity of novel hardware/software technologies. The full OP2 source and example benchmark applications are available as open source software [11] and the developers would welcome new participants in the OP2 project.

ACKNOWLEDGMENTS

This research 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" and the "Algorithms, Software for Emerging Architectures" (ASEArch) EP/J010553/1 project. The authors would like to thank Brent Leback at PGI, Maxim Milakov at NVIDIA, Leigh Lapworth, Paolo Adami and Yoon Ho at Rolls-Royce plc., Graham Markall, Fabio Luporini, David Ham and Florian Rathgeber, at Imperial College London, Lawrence Mitchell at the University of Edinburgh, and Endre László at PPCU Hungary.

REFERENCES

[1] 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.

[2] 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.

[3] 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. New York, NY, USA: ACM, 2011, pp. 9:1–9:12.

[4] T. Muranushi, "Paraiso : An Automated Tuning Framework for Explicit Solvers of Partial Differential Equations," *Computational Science & Discovery*, vol. 5, no. 1, p. 015003, 2012.

[5] T. Brandvik and G. Pullan, "SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, ser. CIT '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1181–1188.

[6] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky, "Implementing Domain-Specific Languages For Heterogeneous Parallel Computing," *IEEE Micro*, vol. 31, pp. 42–52, 2011.

[7] 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.

[8] 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.

[9] 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.

[10] 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

[11] "OP2 for Many-Core Platforms," 2011, http://www.oerc.ox.ac.uk/research/op2.

[12] 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.

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

[14] 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.

[15] "Supplementary material: Acceleration of a full-scale industrial cfd application with op2," 2015, http://TPDS-Website.

[16] D. A. Burgess and M. B. Giles, "Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines," *Adv. Eng. Softw.*, vol. 28, pp. 189–201, April 1997.

[17] J. D. Denton, "Lessons from rotor 37," *Journal of Thermal Science*, vol. 6, no. 1, pp. 1–13, 1997.

[18] H. D. Simon, "Partitioning of Unstructured Problems for Parallel Processing," National Aeronautics and Space Administration, Ames Research Center Moffett Field, California 94035-1000, Tech. Rep. RNR-91-008, Feb 1991. [Online]. Available: www.nas.nasa.gov/assets/pdf/techreports/1991/rnr-91-008.pdf
[19] ISO/IES, "Further Interoperability of Fortran with C," International Organization for Standardization and International Electrotechnical Commission, Geneva, Switzerland, ISO SO/IEC TS 29113:2012, 2012.
[20] "ParMETIS," 2013, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.
[21] "Scotch and PT-Scotch," 2013, http://www.labri.fr/perso/pelegrin/scotch/.
[22] C. Lameter, "An overview of non-uniform memory access," *Commun. ACM*, vol. 56, no. 9, pp. 59–54, Sep. 2013.
[23] 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
[24] P. Lindberg, "Basic OpenMP threading overhead," Intel, Tech. Rep., 2009, http://software.intel.com/en-us/articles/basic-openmp-threading-overhead.
[25] J. Bull, F. Reid, and N. McDonnell, "A microbenchmark suite for openmp tasks," in *OpenMP in a Heterogeneous World*, ser. Lecture Notes in Computer Science, B. Chapman, F. Massaioli, M. Mller, and M. Rorro, Eds. Springer Berlin Heidelberg, 2012, vol. 7312, pp. 271–274. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30961-8_24
[26] S. Williams, "HPGMG-FV, fastforward2 proxy app presentation," Dec 2014, http://crd.lbl.gov/departments/computer-science/performance-and-algorithms-research/research/hpgmg/.
[27] "HECToR - hardware," 2013, http://www.hector.ac.uk/service/hardware/.
[28] M. Zhou, O. Sahni, T. Xie, M. Shephard, and K. Jansen, "Unstructured mesh partition improvement for implicit finite element at extreme scale," *The Journal of Supercomputing*, vol. 59, no. 3, pp. 1218–1228, 2012. [Online]. Available: http://dx.doi.org/10.1007/s11227-010-0521-0
[29] "How to make best use of the AMD Interlagos processor," Nov 2011, www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf.
[30] J. Agulleiro, F. Vázquez, E. Garzón, and J. Fernández, "Hybrid computing: Cpu+gpu co-processing and its application to tomographic reconstruction," *Ultramicroscopy*, vol. 115, no. 0, pp. 109 – 114, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304399112000344
[31] C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng, "A peta-scalable cpu-gpu algorithm for global atmospheric simulations," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '13. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2442516.2442518
[32] "Hybrid-CPU/GPU execution mode with GROMACS," 2013, http://www.scalalife.eu/content/hybrid-cpu-gpu-execution-mode-gromacs.
[33] "PETSc," 2013, http://www.mcs.anl.gov/petsc/petsc-as/.
[34] K. B. Ølgaard, A. Logg, and G. N. Wells, "Automated code generation for discontinuous Galerkin methods," *CoRR*, vol. abs/1104.0628, 2011.
[35] G. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. Ham, and P. Kelly, "Performance-portable finite element assembly using pyop2 and fenics," in *Supercomputing*, ser. Lecture Notes in Computer Science, J. Kunkel, T. Ludwig, and H. Meuer, Eds. Springer Berlin Heidelberg, 2013, vol. 7905, pp. 279–289. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38750-0_21
[36] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '11, 2011, pp. 117–128.
[37] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. C. Schulthess, "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models," *Supercomputing frontiers and innovations*, vol. 1, no. 1, 2014.
[38] "OP2 github repository," 2013, https://github.com/OP2/OP2-Common.
[39] "Uk mini-app consortium," 2014, http://uk-mac.github.io.
[40] D. Dutykh, R. Poncet, and F. Dias, "The {VOLNA} code for the numerical modeling of tsunami waves: Generation, propagation and inundation," *European Journal of Mechanics - B/Fluids*, vol. 30, no. 6, pp. 598 – 615, 2011, special Issue: Nearshore Hydrodynamics.

István Z. Reguly is a lecturer at PPCU ITK, Hungary and an academic visitor at the Oxford e-Research Centre, University of Oxford. He holds an MSc and a PhD in computer science from the PPCU, Hungary. His research interests include high performance scientific computing on many-core hardware and domain specific active libraries for structured and unstructured meshes.

Gihan R. Mudalige is a research associate at the Oxford e-Research Centre, University of Oxford. His research interests are in performance analysis/optimization of scientific applications on parallel, high-performance systems. Previously he has worked as a research fellow at the High Performance Systems Group at the University of Warwick and a research intern at the University of WisconsinMadison, US. Dr. Mudalige holds a PhD. in Computer Science from the University of Warwick and is a member of the ACM.

Carlo Bertolli is a Research Staff Member in the Advanced Compiler Technology group at the IBM T. J. Watson Research Center. His interests include High Performance architectures, compilers, and applications. Previously he was Research Associate at Imperial College London, UK. Dr. Bertolli holds a PhD in Computer Science at the University of Pisa, Italy.

Michael B. Giles is Professor of Scientific Computing in Oxford University's Mathematical Institute where he carries out research into the development and analysis of more efficient Monte Carlo methods for computational finance and engineering uncertainty quantification. He is also an Associate Director of the Oxford e-Research Centre, where directs the NVIDIA CUDA Centre of Excellence and leads associated research into the use of GPUs for a variety of applications.

Adam Betts Adam Betts is a post-doctoral researcher at Imperial College London. Previously he has worked as a software engineer in Rapita Systems Ltd and as a post-doctoral fellow at Malardalen University, Sweden. Since joining Imperial College, Dr. Betts became interested in compiler techniques for generating GPU code and, more recently, verification techniques For GPUs. Dr. Betts holds a PhD in Computer Science from the University of York.

Paul H. J. Kelly leads the Software Performance Optimization group at Imperial College London, and is Co-director of Imperials Centre for Computational Methods in Science and Engineering. His research spans single-address-space operating systems, scalable shared-memory architectures, compilers (bounds checking and pointer analysis), graph algorithms, and performance-portability across diverse parallel architectures.

David Radford David Radford is a Technologist working in CFD Methods at Rolls-Royce Plc in Derby, UK. He holds an Aeronautical Engineering Degree from Imperial College, London. He is part of the Hydra CFD solver team, specialising in adjoint and linear methods using Automatic Differentiation and with an interest in application on novel HPC systems.