Acceleration of a Full-scale Industrial CFD Application with OP2

István Z. Reguly, Oxford e-Research Centre, University of Oxford
Gihan R. Mudalige, Oxford e-Research Centre, University of Oxford
Carlo Bertolli, Department of Computing, Imperial College London
Michael B. Giles, Oxford e-Research Centre, University of Oxford
Adam Betts, Department of Computing, Imperial College London
Paul H.J. Kelly, Department of Computing, Imperial College London
David Radford, Rolls Royce plc.

Hydra is a full-scale industrial CFD application used for the design of turbomachinery at Rolls Royce plc. It consists of over 300 parallel loops with a code base exceeding 50K lines and is capable of performing complex simulations over highly detailed unstructured mesh geometries. Unlike simpler structured-mesh applications, which feature high speed-ups when accelerated by modern processor architectures, such as multi-core and many-core processor systems, 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. OP2 targets the domain of unstructured mesh problems and follows the design of an active library using source-to-source translation and compilation to generate multiple parallel implementations from a single high-level application source for execution on a range of back-end hardware platforms. We chart the conversion of Hydra from its original hand-tuned production version to one that utilizes OP2, and map out the key difficulties encountered in the process. To our knowledge this research presents the first application of such a high-level framework to a full scale production code. Specifically we show (1) how different parallel implementations can be achieved with an active library framework, even for a highly complicated industrial application such as Hydra, and (2) 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 production code could be achieved, but it can be significantly improved on conventional processor systems. Additionally, we achieve further acceleration by exploiting many-core parallelism, particularly on GPU 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 near-optimal performance without the intervention of the application programmer.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Design Studies

General Terms: Design, Performance

Additional Key Words and Phrases: Unstructured Mesh Applications, Domain Specific Language, Active Library, OP2, OpenMP, GPU, CUDA, CFD

Author's addresses: 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. 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

DOI:http://dx.doi.org/10.1145/0000000.0000000

ACM Reference Format:
István Z. Reguly, Gihan R. Mudalige, Carlo Bertolli, Michael B. Giles, Adam Betts, Paul H. J. Kelly, and David Radford, 2013. Acceleration of a Full-scale Industrial CFD Application with OP2. *ACM Trans. Parallel. Comput.* xx, xx, Article xx (October 2013), 35 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

High Performance Computing (HPC) is currently experiencing a period of enormous change. For many years, increased performance was achieved through higher clock frequencies giving an almost free boost to the speed and throughput of applications without the need to re-write software for each new generation of processors. However, 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 [Bohr 2007], 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 core 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. For example, traditional mainstream processors from Intel and AMD are continuing to be designed with more and more homogeneous cores. Each of these cores, usually operating at a high clock frequency, in turn consists of increasingly large vector units (e.g. Intel's AVX extensions) that simultaneously carry out operations on larger blocks of data. At the same time, there is an emergence of co-processors designed to act as "accelerators" for augmenting the performance of certain types of computations in combination with traditional processors. Examples of such co-processors range from discrete (e.g. NVIDIA GPUs [NVIDIA GPUs 2013], Intel MIC [Skaugen 2011]) or integrated (e.g. AMD APUs [AMD 2013]) SIMD type co-processors to more specialized DSP type compute engines [TI 2013] or FPGA based accelerators [Convey 2013; Lindtjorn et al. 2011]. The SIMD type accelerators currently consist of about 16-64 functional units, each in turn consisting of a number of processor cores operating at a lower clock frequency that together can perform a larger number of operations in parallel. Other HPC systems designs include the utilization of custom-built large networks of relatively small but energy-efficient CPUs as in the IBM Blue Gene [Haring et al. 2012] systems. In the future, we may also see developments further motivated by energy-efficient designs from companies such as ARM [Rajovic et al. 2011] who have achieved significant energy efficiency for mobile and embedded applications, and are now targeting HPC which increasingly shares similar energy goals.

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 optimisations 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* [Czarnecki et al. 2000; Veldhuizen and Gannon 1998] and *Domain Specific Languages* (DSLs) have emerged as a pathway offering 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. At the same time, appropriate code generation and compiler support will be provided to generate platform specific code, from the higher-level source, targeting different hardware with tailored optimisations. Within such a setting a separate lower implementation level is created to provide opportunities for parallel programming experts to apply radically aggressive and platform specific optimizations when implementing the required solution on various hardware platforms. The correct abstraction will pave the way for easy maintenance of a higher-level application source with near optimal performance for various platforms and make it possible to easily integrate support for any future novel hardware.

Much research [Howes et al. 2009; DeVito et al. 2011; Muranushi 2012; Orchard et al. 2010; Brandvik and Pullan 2010; Lee et al. 2011] has been carried out on such high-level abstraction methods targeting the development of scientific simulation software. However, there has been no conclusive evidence, so far, for the applicability of active libraries or DSLs in developing full scale industrial production applications, particularly demonstrating the viability of the high-level abstraction approach and its benefits in terms of performance and developer productivity. 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. This paper presents research addressing this open question by detailing our recent experiences in the development and acceleration of such an industrial application with the OP2 ([Giles et al. 2012; Giles et al. 2013]) active library framework.

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. With the initial development carried out over 15 years ago, Hydra has been continuously evolving ever since. It is written in Fortran 77, and it is parallelized to utilize a cluster of single threaded CPUs using message passing. 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; the abstraction is simply achieved with an API and its implementation is through a classical software library called OPlus (Oxford Parallel Library for Unstructured Solvers) [Burgess et al. 1994]. OPlus implements the API calls targeting a cluster of single threaded CPUs. OP2, the second iteration of OPlus, was designed to retain the same domain specific abstraction but develops an active library framework with code generation to exploit parallelism on modern heterogeneous multi-core and many-core architectures.

With OP2, a single application code written using its 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. At the same time, 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 near-optimal performance including high computational efficiency and minimized memory traffic. In previous works, we have presented

OP2's design and development [Giles et al. 2012; Giles et al. 2013] and its performance on heterogeneous systems [Mudalige et al. 2012]. These works investigated the performance through a standard unstructured mesh finite volume computational fluid dynamics (CFD) benchmark, called "Airfoil", written in C using the OP2 API and parallelized on a range of multi-core and many-core platforms; our results showed considerable performance gains could be achieved 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 optimisation strategies that allowed us to gain near-optimal performance on modern parallel systems. A key goal of this research is to investigate whether high-level frameworks such as OP2 could be used to develop large-scale industrial applications and at the same time achieve performance on par with a hand-tuned implementation. Specifically, we make the following contributions:

(1) *Deployment*: We present the conversion of Hydra to utilize OP2, mapping out the key difficulties encountered in the conversion of the Hydra application (designed and developed over 15 years ago) to OP2. Our work demonstrates the clear advantages in developing future-proof and performant applications through a *high-level* abstractions approach. Hydra, to our knowledge, is the first industrial application to successfully demonstrate the viability of such high-level frameworks.
(2) *Optimisations*: We present key optimisations 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 optimisations are radically different across the range of platforms under study, but through OP2, we are able to easily apply each new optimisation demonstrating portability and increased developer productivity.
(3) *Performance*: A range of performance metrics are collected to explore the performance of Hydra with OP2, including runtime, scalability and achieved bandwidth. The performance is compared to that of the original version of Hydra, contrasting the key optimisations that lead to performance differences. Benchmarked systems 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 optimisations 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 total performance.

We use highly-optimized code generated through OP2 for all system back-ends, using the same application code, allowing for direct performance comparison. Our work demonstrates how Hydra, through OP2, is developed to match the performance of the original and then further outperform it with platform specific optimisations 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 near-optimal 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 present the transformations made to Hydra enabling it to utilize OP2; Section 4 details performance and optimisations charting the effort to reach near optimal performance for Hydra. Section 5 presents preliminary results from a hybrid CPU-GPU execution scheme. Section 6 briefly compares related work and Section 7 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 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 three-dimensional meshes. Usually 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 which 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 [P. Moinier and Giles 2002; M.C. Duta and Campobasso 2002; Giles et al. 2003; Lapworth 2008]. 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 [Burgess et al. 1994], 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 any mesh or graph problem solution 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, 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

```
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

dependencies. As an example, consider the code in Figure 1 of a loop in Hydra over a set of triangular cells as illustrated in Figure 2:

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.

As it can be seen from the above loop, 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 [Skaugen 2011] or GPUs, relying only on coarse-grained message-passing is not going to 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 the above loop 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 optimisations tailored to OpenMP are needed. Thus, further substantial *implementation-specific* modifications will 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.

3. OP2

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 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 provides the opportunity to provide the necessary implementation specific optimisations. 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

The first step in getting Hydra to exploit multi-core and many-core parallelism through OP2 is to convert it to use the OP2 API. Continuing from the previous OPlus example, consider the unstructured mesh illustrated in Figure 2. The mesh consists of three sets - nodes (vertices), edges and triangular cells. There are `nn = 14` nodes and `nc = 17` cells. The OP2 API allows to declare these sets and connectivity between the sets together with any data held on the sets (see Figure 3).

In `op_decl_map` each element belonging to the set `nodes` is mapped to three different elements in the set `cells`. The `op_map` declaration defines this mapping where `ncell` has a dimension of 3 and thus the 1D array index 1,2 and 3 maps to nodes 1,3 and 10, index 4,5 and 6 maps to nodes 1,2 and 3 and so on. When declaring a mapping we first pass the source set (e.g. `cells`) then the destination set (e.g. `nodes`). Then we pass the dimension (or arity) of each map entry (e.g. 3; as `ncell` maps each cell to 3 nodes). Note that those literal numbers filling up the mapping array are purely for description purposes only as OP2 supports input from disk (e.g. using HDF5 files). Once the sets are defined, data can be associated with the sets through `op_decl_dat` statements. Note that here a single double precision value per set element is declared. A vector of a number of values per set element could also be declared (e.g. a vector with three doubles per node to store coordinates). In the case of Hydra, there are `op_dats` with over 6 values per set element.

Any loop over the sets in the mesh is expressed through the `op_par_loop`, 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. Thus, the same loop can be expressed as detailed in Figure 4 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 ac-

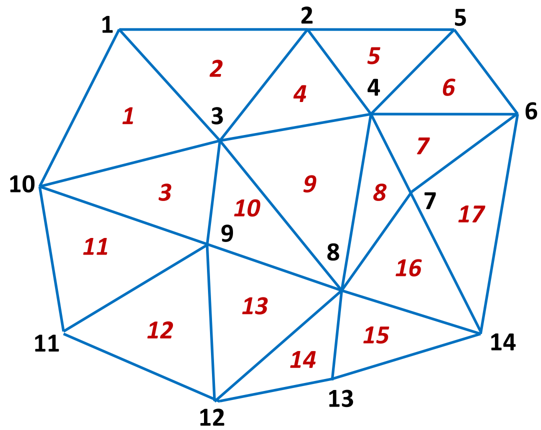


Fig. 2: An example unstructured mesh

```
integer(4),dimension(:),allocatable:: c_to_n
real(8),dimension(:),allocatable:: ca_data
real(8),dimension(:),allocatable:: na_data

allocate ( c_to_n ( 3 * nc ) )
allocate ( ca_data ( nc ) )
allocate ( na_data ( nn ) )

c_to_n = /1,3,10, 1,2,3, 3,9,10, 2,3,4, .../
ca_data = /.../
na_data = /.../

call op_decl_set(nn,nodes,'nodes')
call op_decl_set(ne,edges,'edges')
call op_decl_set(nc,cells,'cells')

call op_decl_map
&    (cells,nodes,3,c_to_n,ncell,'pcell')

call op_decl_dat
&    (cells,1,'r8',ca_data,areac,'c_area')
call op_decl_dat
&    (nodes,1,'r8',na_data,arean,'n_area')
```

Fig. 3: Declaring sets, maps and dats using the OP2 API

cessing 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.

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

```
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. 4: A Hydra loop written using the OP2 API

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.

3.2. Code Generation and Parallel Build

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 5.

In this illustration, the application consists of an `op_par_loop` that calls 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 [Burgess and Giles 1997], partitioning (under MPI) and conversion to an array-of-structs data layout (for GPUs [Giles et al. 2013]). These transformations, and OP2's ability to seamlessly apply them internally is key to achieving a number of performance optimisations.

Once the application developer is satisfied with the validity of the results produced by the sequential application, parallel code can be generated. The build process to obtain a parallel executable is detailed in Figure 6. 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

```
! 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. 5: Modules structure for a sequential build with OP2

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

OP2 currently supports 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 [Mudalige et al. 2013] 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 [Giles et al. 2012; Giles et al. 2013; Mudalige et al. 2012].

For each parallelization, the generated modified main program and back-end specific code follow the general module and procedure structure given in Figure 7. A modified application program is 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 7 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

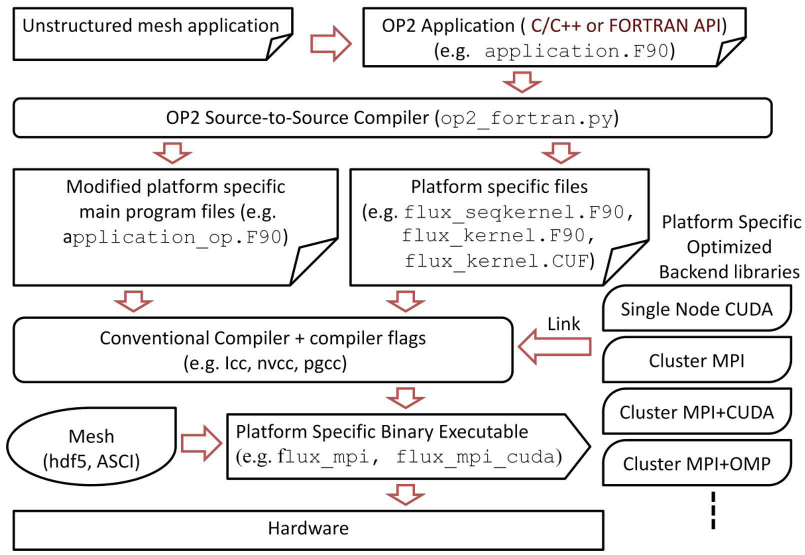


Fig. 6: OP2 build hierarchy

```
!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. 7: Modules structure for a parallel build with OP2

mapping arrays held internally by OP2, to Fortran pointers in order to pass them to

Table I: 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 flags	-O2 -xAVX -Mcuda=5.0,cc35	-O3 -h fp3 -h ipa5	-O2 -xAVX -Mcuda=5.0,cc35

the user kernel. This allows OP2 to use the user kernels without modification by the code generator.

One final step that was required to get the Fortran API working with the OP2 backend 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 then 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 gave Hydra any performance improvements compared to the original OPlus version, (2) whether further performance gains are achievable with modern multi-core/many-core hardware (3) what optimisations can be applied to improve performance for different parallel platforms and (4) how the OP2 framework facilitate the deployment of such optimisations. In the next section we analyze Hydra's performance with OP2 and present work assessing these issues.

4. PERFORMANCE AND OPTIMISATIONS

We begin by initially benchmarking the runtime of Hydra with OP2 on a single node. Exploring performance on a single node is not only motivated by the need to understand intra-node performance but also due to the fact that Hydra is regularly run on single node systems by CFD engineers for preliminary design purposes. Key specifications of the single node system are detailed in column 1 of Table I. The system is a two socket Intel Xeon E5-2640 system with 64GB of main memory. The processors are based on Intel's latest Sandy Bridge architecture. The compiler flags that give the best runtimes are listed. This system, named Ruby, also consists of two NVIDIA Tesla K20c GPUs, each with 5 GB of graphics memory. We use CUDA 5.0 in this study. Table I also details the large cluster systems used later in the benchmarking study. These will be used to explore the distributed memory scaling performance of Hydra.

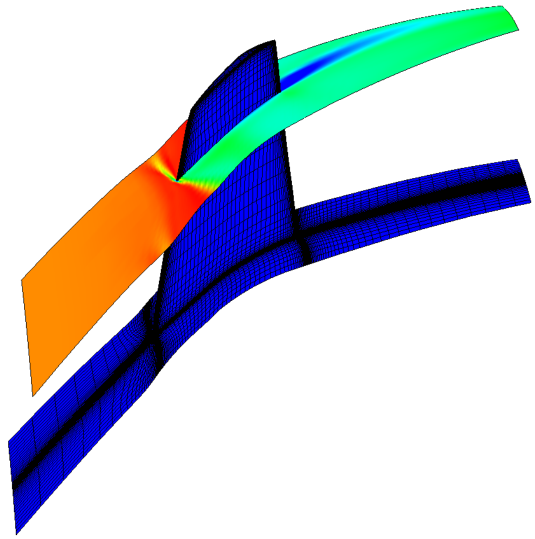
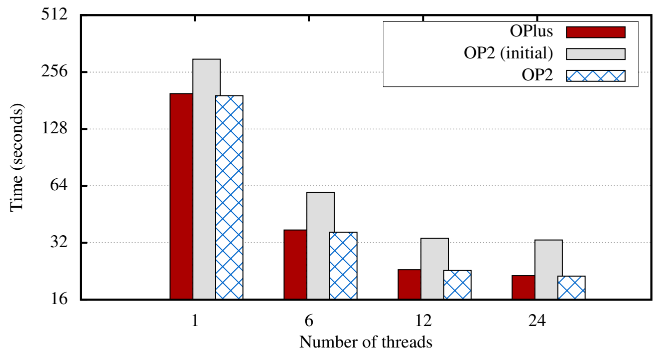


Fig. 8: Mach contours for NASA Rotor 37.

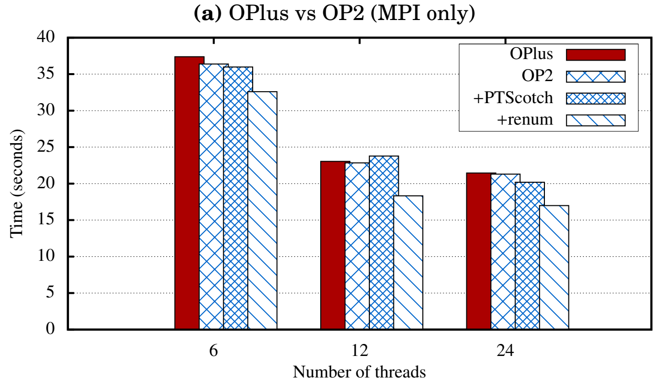
As mentioned previously, Hydra consists of several components [Lapworth 2008] and in this paper we report on the non-linear solver configured to compute in double precision floating point arithmetic. Hydra can also be used with multi-grid simulations, but for simplicity of the performance analysis and reporting we utilize experiments with a single grid (mesh) level. The configuration and input meshes of Hydra in these experiments model a standard application in CFD, called NASA Rotor37 [Denton 1997]. It is a transonic axial compressor rotor widely used for validation in CFD. Figure 8 shows a representation of the Mach contours for this application on a single blade passage. The mesh used for the single node performance benchmarking consists of 2.5 million edges.

Figure 9a 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. This is a like-for-like comparison where the same mesh is used by both versions. The partitioning routine used in both cases is a recursive coordinate bisection (RCB) mesh partitioning [Simon 1991] 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 the OP2 version (noted as OP2 initial) is about 50% slower than the hand coded OPlus version. The generated code from OP2 appears to be either missing a performance optimisation inherent in the original Hydra code and/or the OP2 generated code and build structure is introducing new bottlenecks. By simply considering the runtime on a single thread we see that even without MPI communications the OP2 (initial) version performs with the same slowdown. Thus it was apparent 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. As a solution, the code generator was modified to generate code that simply places the user kernel and the encompassing loop over set elements into the same source file, adding compiler pragmas to ensure the inlining of the user



(a) OPlus vs OP2 (MPI only)



(b) OPlus vs OP2 (with PTSotch and renumbering)

Fig. 9: Single node performance on Ruby (NASA Rotor 37, 2.5M edges, 20 iterations)

kernel. The resulting code indeed improved performance over the OP2 (initial) version on average by about 25%.

To analyze the performance further, we ran Hydra through the Intel VTune [VTune 2013] profiler. The aim was to investigate 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_loops` 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 in Figure 7. Further investigation revealed that the cause is a low level Fortran specific implementation issue [ISO/IES 2012] related to how arrays are represented internally in Fortran. The pointer `arg1Ptr` in Figure 7, 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 as illustrated in Figure 10 was generated. The new structure introduces a wrapper subroutine `flux_wrap` that is initially called by `flux_kernel` by passing the `arg1Ptr` pointer. At this point the complex indexing calculation is carried out, just once. Then `flux_wrap` works with an

```
module FLUX_GENSEQ
subroutine flux_user_kernel(x, ...)
real(8) x(3)
...
end subroutine

subroutine flux_wrap(arg1data,...,nelems)
real(8), arg1data(3,*)
do i = 0,nelems-1,1
  call flux_user_kernel(x(1,i+1),
  & ...)
enddo
end subroutine

subroutine flux_kernel(arg1, ...)
real(8), dimension(:), pointer :: arg1Ptr
...
call c_f_pointer(arg1%CPtr, arg1Ptr, ...)
...
call flux_wrap(arg1Ptr,...)
...
end subroutine
end module FLUX_GENSEQ
```

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

Fig. 10: Optimized modules structure for OP2

"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 9a. The OP2 based Hydra application now matches the performance of the original version. 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.

A number of new features implemented in OP2 allow further improvements for the MPI parallelization. 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 a trivial block partitioning,

Table II: 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 (NASA Rotor 37, 0.8M nodes, 2.5M edges, 20 iterations).

Loop	Time (sec)	% runtime	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

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 [ParMETIS 2013] or PTScotch [PT-Scotch 2013]. For the purposes of comparison with the original OPlus code, as shown in Figure 9a, we have also implemented the recursive coordinate bisection algorithm in OP2 allowing it to generate the exact same partitioning as the original Hydra application.

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 [Burgess and Giles 1997]: 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 [PT-Scotch 2013].

The results in Figure 9b 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 II when all the above optimisations 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 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. Further benchmarks exploring different aspects of performance are presented in Section 4.2 on large-scale systems.

4.1. Multi-core and Many-core Performance

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 illustrated in Figure 10. 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 [Giles et al. 2012; Mudalige et al. 2012]. 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.

4.1.1. OpenMP Performance. Figure 11 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 [Mudalige et al. 2013]. The causes for MPI outperforming OpenMP on SMPs/CMPs have been widely discussed in literature. These mainly consist of the non-uniform memory access issues (NUMA [Lameter 2013; Jr. and Ellis 1991]) discussed in further detail below, and thread creation and tear down overheads [Lindberg 2009]. In our case, an additional cause may also be the reduced parallelism due to coloring for blocks that avoid data races. The hybrid MPI+OpenMP parallelization provided better performance but was still about 10% slower than the pure MPI version; again the overhead of shared-memory multi-threading techniques may be causing performance bottlenecks. To explore the causes further, in Figure 12 and Table III we present the variation in the runtime and the number of block colors for different block sizes for a number of combinations of MPI processes and OpenMP threads. The number of colors and blocks per color are automatically output by OP2 for each `op_par_loop` and we have presented here a range of numbers taken from the most time consuming loops in Hydra.

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. In other words, 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 12. The average number of colors (nc) and the average number of blocks (nb) in Table III supports this reasoning where, overall we see a reduction in runtime as the block size is increased but only until there are enough blocks per color (nb/nc) to achieve good load balancing in parallel. We also experimented by using OpenMP's static and dynamic load balancing functionality but did not obtain any significant benefits as all the blocks executed (except for the very last one) have the same size.

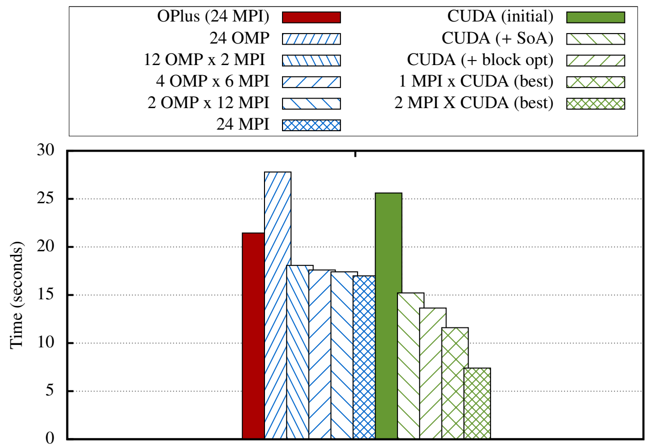


Fig. 11: OP2 Hydra Multi-core/Many-core performance on Ruby (NASA Rotor 37, 2.5M edges, 20 iterations) with PTScotch partitioning

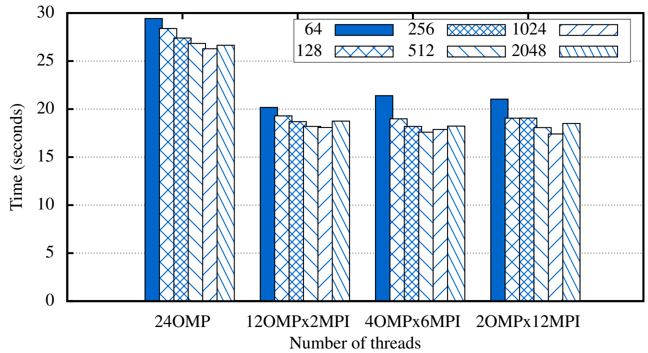


Fig. 12: Hydra OpenMP and MPI+OpenMP performance for different block sizes on Ruby (NASA Rotor 37, 2.5M edges, 20 iterations) with PTScotch partitioning

The overall poor performance of the pure OpenMP runtimes (24OMP) for any block size, compared to the other MPI+OpenMP runs, appears to be due to Non-Uniform Memory Accesses (NUMA) issues. On multiple-socket machines system memory is connected to different sockets, and so when a thread on one socket has to access memory connected to another socket there is a significant overhead [Lameter 2013; Jr. and Ellis 1991]. As OP2 does a colored execution, and the colors are dynamically determined at runtime and may vary between different loops, statically determining which thread processes which blocks is not possible, thus allocating memory close to the thread/core that will execute a block is not feasible. Thus, the pure OpenMP version gets affected, preventing further performance gains from increased block size. When executing in an MPI+OpenMP hybrid setting, processes and threads are pinned to specific sockets, thereby circumventing this issue. Thus it is important to use a sensible MPI and OpenMP process and thread combination for the given node/socket architecture.

Finally we conclude that the above effects, particularly the lost parallelism with colored execution, combined with thread creation and tear down overheads [Lindberg 2009] may account for the slightly worse performance of the hybrid MPI+OpenMP approach on a single node compared to the MPI only runtime.

Table IV 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

Block size	OMP × MPI 24 nb (K)	nc	12×2 nb (K)	nc	4×6 nb (K)	nc	2×12 nb (K)	nc
64	39	14	20	16	6.8	16	3	16
128	20	15	10	17	3.4	17	1.7	15
256	10	14	5	14	1.7	15	0.8	14
512	5	12	2.5	14	0.8	14	0.4	12
1024	2.5	10	1	11	0.4	11	0.2	11
2048	1	9	0.6	11	0.2	11	0.1	10

Table III: Hydra single node performance, Number of blocks (nb) and number of colors (nc) (K = 1000): 2.5M edges, 20 iterations

Table IV: Hydra single node performance, 6 MPI x 4 OMP with PTScotch: 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

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 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 advertised peak bandwidth of the Sandy Bridge processors (2×42.6 GB/s [SandyBridge 2013]). 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 [Giles et al. 2012; Mudalige et al. 2012] for the Airfoil benchmark application.

4.1.2. GPU Performance. 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 [Giles et al. 2012; Giles et al. 2013] 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 back-end and to connect C pointers to Fortran pointers. The same plan construction that was described in previous papers [Giles et al. 2013] is retained. Previously, indirect data from GPU global memory were loaded into each SMs shared memory space forming a local mini-partition. Each mini-partition was assigned to an SM (Streaming multi-processor) and executed in parallel. The SM executes the mini-partition by utilizing a number of threads (called a thread block). 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 [Giles et al. 2013] 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 11. 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 bandwidth utilization. Further investigation showed that a high amount of cache contention is caused by the default data layout of `op_dats`. This layout, called Array-of-Structs (AoS), was found to be the best layout for indirectly accessed data in our previous work [Giles et al. 2013] on Fermi GPUs. However without staging in shared memory it was damaging performance in Hydra as many of Hydra's `op_dats` have a large number of components (dimensions) for each set element (typically related to the number of PDEs ≥ 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 optimisation resulted in an increase of about 40% to the single GPU performance as shown in Figure 11.

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 consideration 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-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

Table V: Hydra single GPU performance: NASA Rotor 37, 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

to synchronization overheads, cache locality, etc. that are difficult to predict. We used auto-tuning techniques described in a previous work [Giles et al. 2013] to find the best thread block size for different parallel loops, and stored them in a look-up table. This optimisation can be carried out once for different hardware, but performance is unlikely to vary significantly when solving different problems. Together with the SoA optimisation, the use of tuned block sizes and limited register counts provided a further 10% performance improvement.

In the case of several Hydra loops that are bandwidth-intensive and do little computation, the thread coloring technique (that avoids write conflicts) introduces a significant branching overhead, due to adjacent edges (mapped to adjacent threads) having different colors, and execution over colors getting serialized. The negative effect on performance can be mitigated by sorting edges in blocks by color, ensuring that threads in a warp have the same color (or at least fewer colors), which in turn reduces branching within the warps. However, this solution degrades the access patterns to directly accessed data in memory, as adjacent threads are no longer processing adjacent set elements. Thus, we only apply it to the Hydra loops that are limited by branching overhead. 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 11. A single K20 GPU achieves about 1.8$\times$ speedup over the original OPlus version of Hydra on Ruby and about 1.5$\times$ 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$\times$ speedup over the best OP2 runtime with CPUs (2 CPUs, 24 MPI processes) and about 2.89$\times$ speedup over the best OPlus runtime with CPUs (2 CPUs, 24 MPI processes). Table V notes the achieved memory bandwidth utilization on a K20 GPU. A majority of the most time consuming loops achieve 20 - 50 % of the advertised peak bandwidth of 208 GB/s on the GPU [K20 2012]. 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

Table VI: Halo sizes : Av = average number of MPI neighbors per process, Tot. = average number total elements per process, %H = average % of halo elements per process

(a) Strong Scaling

nodes	MPI procs.	PTScotch Av	PTScotch Tot	PTScotch %H	RCB Av	RCB Tot	RCB %H
1	32	7	111907	5.12	9	117723	9.81
2	64	8	56922	6.74	10	61208	13.27
4	128	9	29175	9.02	11	32138	17.41
8	256	10	14997	11.52	11	17074	22.28
16	512	10	7765	14.55	12	9211	27.97
32	1024	10	4061	18.32	12	4949	32.98
64	2048	10	2130	22.16	13	2656	37.58
128	4096	10	1134	26.98	13	1425	41.89

(b) Weak Scaling

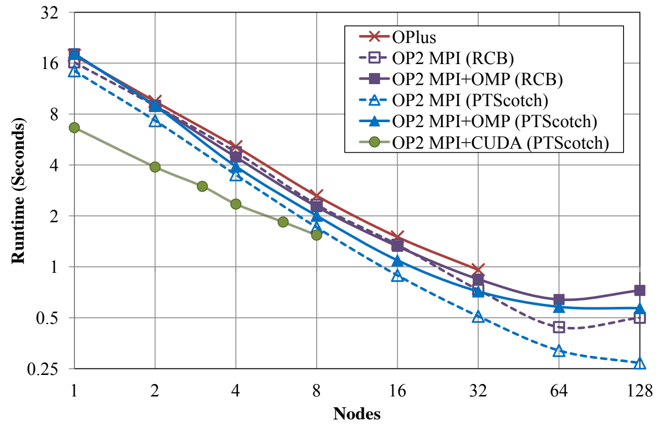
nodes	MPI procs.	PTScotch Av	PTScotch Tot	PTScotch %H	RCB Av	RCB Tot	RCB %H
1	32	8	70084	6.00	8	73486	10.35
2	64	9	73527	6.69	9	78934	11.07
4	128	10	79009	6.09	10	73794	12.26
8	256	12	73936	7.33	11	78396	12.98
16	512	13	78671	6.94	12	75224	13.76

to a near optimal GPU back-end for Hydra that is significantly faster than the CPU back-end, with very little change to the original source code.

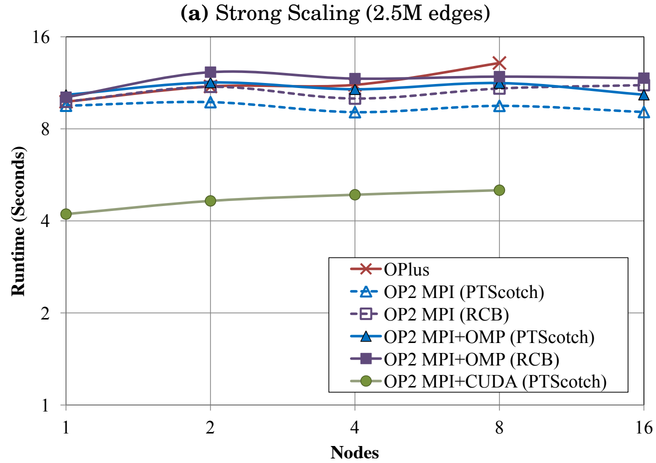
4.2. Distributed Memory Scaling Performance

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. In this section we explore the performance on such systems. Table I lists the key specifications of the two cluster systems we use in our benchmarking. The first system, HECToR [HECToR 2013], 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 [JADE 2013] is a small NVIDIA GPU (Tesla K20) cluster that we use to benchmark the MPI+CUDA execution.

4.2.1. Strong Scaling. Figure 13(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



(a) Strong Scaling (2.5M edges)



(b) Weak Scaling (0.5M edges per node)

Fig. 13: Scaling performance on HECToR (MPI, MPI+OpenMP) and Jade (MPI+CUDA) on the NASA Rotor 37 mesh (20 iterations)

four MPI processes per HECToR node, each MPI process consisting of eight OpenMP threads. This is due to the NUMA architecture of the Interlagos processors [Interlagos 2011] which combines two cores to a "module" and packages 4 modules with a dedicated path to part of the DRAM. Here we were able to leverage the job scheduler to exactly place and bind one MPI process per memory region (or die) reducing inter-die communications. Other combinations of processes and threads were also explored, but the above provided the best performance.

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 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. Evidence for this explanation can be gained by comparing the average number of halo elements and the average number of neighbors per MPI process reported by OP2 after

Table VII: Hydra strong scaling performance on HECToR, Number of blocks (nb) and number of colors (nc) for MPI+OpenMP and time spent in communications (comm) and computations (comp) for the hybrid and the pure MPI implementation: 2.5M edges, 20 iterations

Num of nodes	MPI+OMP nb	MPI+OMP nc	MPI+OMP comm (sec.)	MPI+OMP comp (sec.)	MPI comm (sec.)	MPI comp (sec.)
1	9980	17	1.33	16.6	1.2	13.2
2	4950	16	1.04	7.8	0.83	6.5
4	2520	17	0.57	3.3	0.36	3.14
8	1260	15	0.52	1.42	0.23	1.48
16	630	14	0.26	0.81	0.21	0.68
32	325	13	0.28	0.4	0.13	0.38
64	165	10	0.32	0.21	0.12	0.2
128	86	12	0.52	0.11	0.15	0.12

partitioning with PTScotch and RCB (see Table VI). We noted these results from runs on HECToR, but the halo sizes and neighbors are only a function of the number of MPI processes (where one MPI process is assigned to one partition) as the selected partitioner gives the same quality partitions for a given mesh for the same number of MPI processes on any cluster.

Column 4 and 7 of Table VI(a) details the average total number of nodes and edges per MPI process when partitioned with PTScotch and RCB respectively. Columns 5 and 8 (%H) indicate the average proportion of halo nodes and edges out of the total number of elements per MPI process while Columns 3 and 6 (Av) indicate the average number of communication neighbors per MPI process. With PTScotch, the proportion of the halo elements out of the average total number of elements held per MPI process ranged from about 5% (at 32 MPI processes) to about 27% (at 4096 MPI processes). The average number of MPI neighbors per MPI process ranged from 7 to 10. The halo sizes with RCB were relatively large, starting at about 10% at 32 MPI processes to about 40% at 4096 processes. Additionally the average number of neighbors per MPI process was also greater with RCB. These causes point to better scaling with PTScotch which agrees with the results in Figure 13(a).

The above reasoning, however, goes contrary to the relative performance difference we see between OP2's MPI only and MPI+OpenMP parallelizations. We expected MPI+OpenMP to perform better at larger machine scales as observed in previous performance studies using the Airfoil CFD benchmark[Mudalige et al. 2013]. The reason was that larger partition sizes per MPI process gained with MPI+OpenMP in turn resulting in smaller proportionate halo sizes. But for Hydra, adding OpenMP multi-threading has caused a reduction in performance, where the gains from better halo sizes at increasing scale have not manifested into an overall performance improvement. Thus, it appears that the performance bottlenecks discussed in Section 4.1.1 for the single node system are prevalent even at higher machine scales. To investigate whether this is indeed the case, Table VII presents the number of colors and blocks for the MPI+OpenMP runs at increasing scale on HECToR.

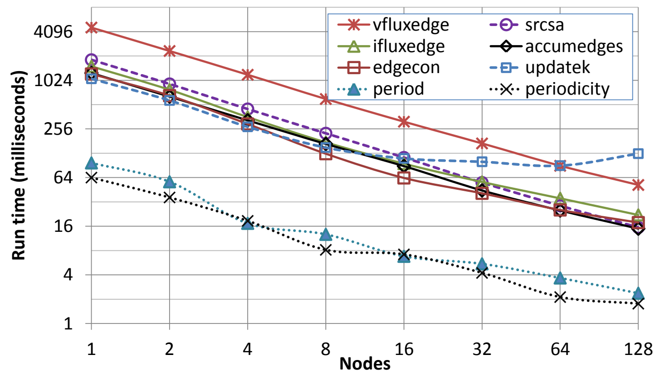
The size of a block (i.e. a mini-partition) was tuned from 64 to 1024 for each run, but at higher scale (from upwards of 16 nodes) the best runtimes were obtained by a block size of 64. As can be seen, the number of blocks (nb) reduces by two orders of magnitude when scaling from 1 node up to 128 nodes. However within this scale the number of colors remains between 10 to 20. These numbers provide evidence similar to the ones we observed on the Ruby single node system in Section 4.1.1 where a reduced number of blocks per color results in poor load balancing. The time spent computing and communicating during each run at increasing scale shows that although the computation time reduces faster for the MPI+OpenMP version, its communication times increase significantly, compared to the pure MPI implementation. Profiled runs of MPI+OpenMP indicate that the increase in communications time is in fact due to time spent in `MPI_Waitall` statements where due to poor load balancing, MPI processes get limited by their slowest OpenMP thread.

Getting back to Figure 13(a), 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 [Mudalige et al. 2012], 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. We further explore these issues and how they affect performance of different types of loops in Hydra later in Section 4.2.3.

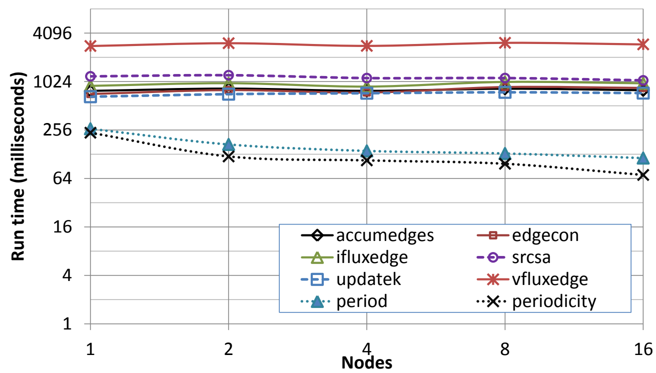
4.2.2. Weak Scaling. Weak scaling of a problem investigates the performance of the application at both increasing problem and machine size. For Hydra, we generated a series of NASA rotor 37 meshes such that a near-constant mesh size per node (0.5M vertices) is maintained at increasing machine scale. The results are detailed in Figure 13(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, while the RCB partitioning slows down 3-7% every time the number of processes and problem size is doubled. One reason for this is the near constant halo sizes resulting from PTScotch, but with RCB giving 7-10% larger halos. The second reason is the increasing cost of MPI communications at larger scale, especially for global reductions. 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. Adjusting the experiment to compare one HECToR node to one GPU (instead of a full JADE node with 2 GPUs) still shows a 10-15% performance advantage for the GPU.

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. 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



(a) Strong Scaling (2.5M edges)



(b) Weak Scaling (0.5M edges per node)

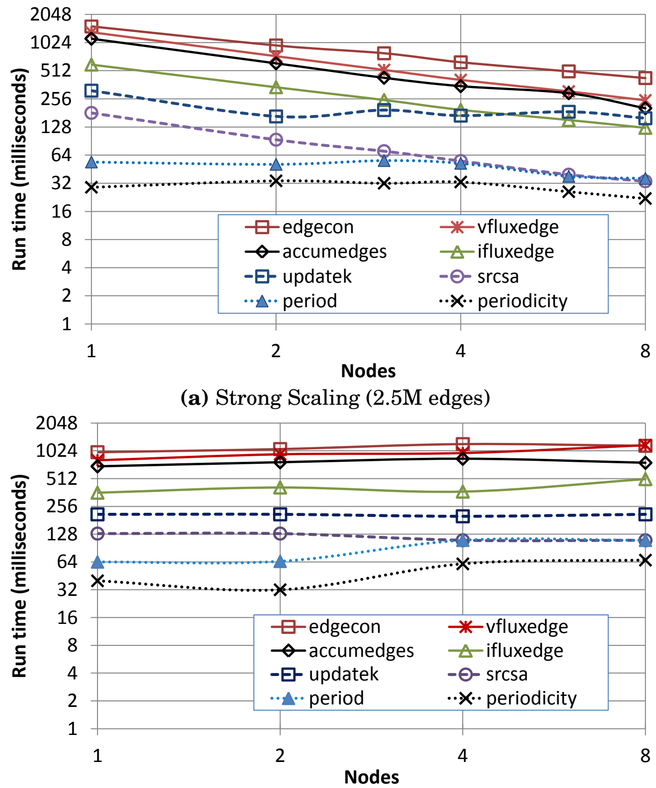
Fig. 14: Scaling performance per loop runtime breakdowns on HECToR (NASA Rotor 37, 20 iterations)

are much less scalable for small problem sizes and are best utilized in weak-scaling executions.

4.2.3. Performance Breakdown at Scale. In this section we delve further into the performance of Hydra in order to identify limiting factors. The aim is to break down the scaling performance to gain insights into how the most significant loops in the application scale on each of the two cluster systems.

Figure 14a shows the timing breakdowns for a number of key loops when, for the MPI only version (partitioned with PTScotch). Note how the loops `vfluxedge`, `edgecon` and `srcsa` at small scale account for most of total runtime. However as they are loops over either interior vertices or edges, that do not include any global reductions, they have near-optimal scaling. The loop `updatek` contains global reductions, and thus at scale, it is bound by the latency of communications. At 128 nodes (4096 cores) it becomes the single most expensive loop. Loops over boundary sets, such as `period` and `periodicity` scale relatively worse than loops over interior sets, since fewer partitions carry out operations over elements in those sets.

Per-loop breakdowns when strong scaling on the Jade GPU cluster are shown in Figure 15a. Observe how the performance of different loops is much more spread out compared to those on the CPU cluster scaling (as shown in Figure 14a). Also note how boundary loops such as `period` and `periodicity` are not so much faster than loops over interior sets, which is again due to GPU utilization. While the loop with reductions (`updatek`) was showing good scaling on the CPU up to about 512 cores, per-



(a) Strong Scaling (2.5M edges)

(b) Weak Scaling (0.5M edges per node)

Fig. 15: Scaling performance per loop runtime breakdowns on JADE (NASA Rotor 37, 20 iterations)

formance stagnates beyond 4 GPUs which is a result of the near-static overhead of on-device reductions and the transferring of data to the host, all of which are primarily latency-limited. Most other loops, such as `vfluxedge`, `ifluxedge`, `accumedges` and `srcsa`, scale with 65-80% performance gain when the number of GPUs are doubled, however `edgecon` only shows a 48-60% increase due to the loop being dominated by indirect updates of memory and an increasingly poor coloring at scale.

Figure 14b shows timing breakdowns for the same loops when weak scaling on HECToR, with very little increase in time for loops over interior sets and a slight reduction in time for boundary loops, as a result of the boundary (surface) of the problem becoming smaller relative to the interior. Similar results can be observed in Figure 15b when weak scaling on the GPU cluster. Here, some of the bigger loops gets relatively slower, due to the load imbalance between different GPUs. In this case some partitions need more colors than others for execution, which in turn slows down execution. Boundary loops such as `period` and `periodicity` become slower as more partitions share elements of the boundary set forcing halo exchanges that are limited by latency.

5. 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 [Agulleiro et al. 2012], 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 [Yang et al. 2013; Bilel et al. 2012; GROMACS 2013]. In 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 IV and Table V. 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 16 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. We have carried out an autotuning run for the value of the static load balance, yielding the data points in Figure 16, registering the execution times of different loops on the CPU and the GPU.

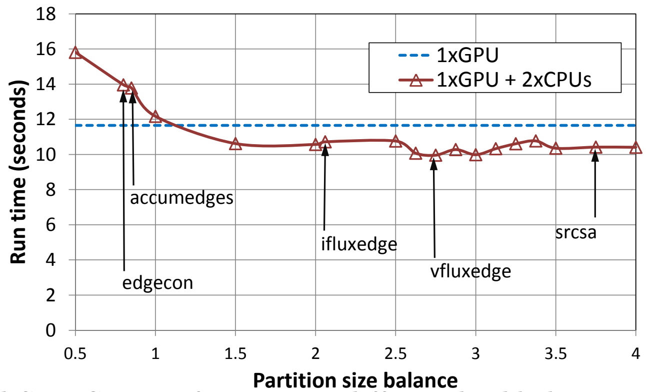


Fig. 16: Hybrid CPU-GPU performance at different load balancing values, marking perfect balance for different loops on Ruby (NASA Rotor 37, 2.5M edges, 20 iterations) with ParMetis partitioning

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%.

6. RELATED WORK

There are several well established conventional libraries supporting unstructured mesh based application development on traditional distributed memory architectures. These include the popular PETSc [PETSc 2013], Sierra [Stewart and Edwards 2004] libraries as well as others such as Deal.II [Deal.II 2013], Dune [DUNE 2013] and FEATFLOW [FEATFLOW 2013]. Recent research in this area include unstructured mesh frameworks that allows extreme scaling (up to 300K cores) on distributed memory CPU clusters [Zhou et al. 2012] while libraries such as PETSc have also implemented hand tuned back-ends targeting solvers on distributed memory clusters of GPUs. Other related work include RPI's FMDB [FMDB 2013] and LibMesh [Kirk et al. 2006]. A major goal of the LibMesh library is to provide support for adaptive mesh refinement (AMR) computations. There are also conventional applications such as computational fluid dynamics (CFD) solvers TAU [Jägersküpper and Simmendinger 2011], OpenFOAM [OpenCFD 2013] or Fluent [ANSYS 2011] that support GPU acceleration of parts of the simulation, however these take the approach of having separate hand-written implementations for different hardware. In contrast to these libraries, OP2's objective is to support multiple back-ends (particularly aimed at emerging multi-core/many-core processor systems) for the solution of mesh-based applications without the intervention of the application programmer.

OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [Howes et al. 2009] 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. The decoupled Access/Execute specification in turn creates the opportunity to apply powerful optimizations targeting the underlying hardware. A number of research projects have implemented similar or related programming frameworks. Liszt [DeVito et al. 2011] and FEniCS [Ølgaard et al. 2011] specifically target mesh based computations.

The FEniCS [Ølgaard et al. 2011] project defines a high-level language, UFL, for the specification of finite element algorithms. The FEniCS abstraction allows the user to express the problem in terms of differential equations, leaving the details of the implementation to a lower-library. Currently, a runtime code generation, compilation and execution framework that is based on OP2, called PyOP2 [Rathgeber et al. 2012], is being developed at Imperial College London to enable UFL declarations to use similar back-end code generation strategies to OP2. Thus, the performance results in this paper will be directly applicable to the performance of code generated by FEniCS in the future.

While OP2 uses an active library approach utilizing code transformation, Delite [Lee et al. 2011] and Liszt [DeVito et al. 2011] from Stanford University take the external approach to domain specific languages that require advanced compiler technology, but offer more freedom in implementing the language. Liszt targets unstructured grid applications; the aim, as with OP2, is to exploit information about the structure of data and the nature of the algorithms in the code and to apply aggressive and platform specific optimizations. Performance results from a range of systems (a single GPU, a multi-core CPU, and an MPI based cluster) executing a number of applications written using Liszt have been presented in [DeVito et al. 2011]. The Navier-Stokes application in [DeVito et al. 2011] is most comparable to OP2's Airfoil benchmark [Giles et al. 2011] and shows similar speed-ups to those gained with OP2 in our work. Application performance on heterogeneous clusters such as on clusters of GPUs is not considered in [DeVito et al. 2011] and is noted as future work.

There is a large body of work that develops higher-level frameworks for explicit stencil based applications (structured mesh applications) including Paraiso [Muranushi 2012], Ypnos [Orchard et al. 2010], SBLOCK [Brandvik and Pullan 2010], SDSL [Henretty et al. 2013], the HIPAcc framework [Membarth et al. 2012], Pochoir [Tang et al. 2011], Patus [Christen et al. 2011], Mint [Unat et al. 2011], Kranc [Husa et al. 2006], and others [Cohen et al. 2013; Holewinski et al. 2012]. In a structured mesh, the mesh is regular and the connectivity is implicit, where computations are performed based on a stencil that define a regular access pattern. Based on the stencil, the required computation (or a kernel) is used to compute a new element value from the current element value and neighboring elements. Ypnos [Orchard et al. 2010] is a functional, declarative domain specific language, embedded in Haskell and extends it for parallel structured grid programming. The language introduces a number of domain specific abstract structures, such as *grids* (representing the discrete space over which computations are carried out), *grid patterns* (stencils) etc. in to Haskell, allowing different back-end implementations, such as C with MPI or CUDA. Similarly, Paraiso [Muranushi 2012] is a domain-specific language embedded in Haskell, for the automated tuning of explicit solvers of partial differential equations (PDEs) on GPUs, and multicore CPUs. It uses algebraic concepts such as tensors, hydrodynamic properties, interpolation methods and other building blocks in describing the PDE solving algorithms. In contrast, most other projects, such as SBLOCK [Brandvik and Pullan 2010], SDSL [Henretty et al. 2013], HIPAcc [Membarth et al. 2012] and Pochoir [Tang et al. 2011] express computations as kernels applied to elements of a set and use compiler technologies and automatic source code generation to accelerate execution.

We are not aware of research into the applicability of the above DSLs to large-scale or industrial applications.

Recently, directive based approaches have emerged as a much publicized solution to programming heterogeneous systems. Most notable of these, OpenACC [OpenACC 2013] attempts to follow the accessibility and success of OpenMP in programming parallel systems but specifically targets accelerator based systems. Our experience suggests a range of extensive language and compiler-technology improvements would be needed to enable applications with significant non-affine array accesses (such as unstructured mesh applications with indirect array accesses) to provide the best performance in a directive-based programming framework. The alternative, in which a significant effort is made by application developers to explicitly optimize their code, is not a viable solution for an industrial code, due to its size and complexity. The abstractions provided by the OP2 library (particularly the `op_par_loop` construct) can be seen as encompassing not only the "parallel regions" of a directive based program that can be accelerated, but also an explicit description of how each data item within the parallel region is accessed, modified and synchronized with respect to the unstructured mesh data and computation it represents. With such an explicit *access-descriptor*, OP2 allows for optimization and parallel programming experts to choose significantly more radical implementations for very specific hardware in order to gain near-optimal performance.

7. 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 optimisations 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 2 to 3 $\times$ speedups over the latest Intel Sandy Bridge x86-64 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.

Some of the key optimisations that affect all backend 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 develop a deeper understanding of execution efficiency by investigating performance characteristics of the major computational loops in Hydra. We show how near-optimal performance is achieved in most loops and identify bottlenecks due to MPI collectives or inefficiencies in the execution scheme that are going to be a subject of future research.

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 [OP2 2011; OP2-GIT 2013] 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 acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work.

The authors would like to thank PGI and Brent Leback for their proactive support in improving their CUDA Fortran compiler and Maxim Milakov of NVIDIA for his help and advice on GPU optimizations.

We are thankful to Leigh Lapworth, Paolo Adami and Yoon Ho at Rolls-Royce plc. for providing access to the Hydra CFD application, Gr aham Markall, Fabio Luporini, David Ham and Florian Rathgeber, at Imperial College London, Lawrence Mitchell at theUniversity of Edinburgh for their contributions to the OP2 project and Endre László, András Oláh and Zoltán Nagy at PPKE Hungary.

REFERENCES

J.I. Agulleiro, F. Vázquez, E.M. Garzón, and J.J. Fernández. 2012. Hybrid computing: CPU+GPU co-processing and its application to tomographic reconstruction. *Ultramicroscopy* 115, 0 (2012), 109 – 114. DOI:http://dx.doi.org/10.1016/j.ultramic.2012.02.003

AMD 2013. AMD Fusion APUs. (2013). http://fusion.amd.com/.

ANSYS. 2011. *ANSYS FLUENT Theory Guide*. http://www.springerlink.com/content/u13552/#section=800779& page=21& locus=97

B.R. Bilel, N. Navid, and M.S.M. Bouksiaa. 2012. Hybrid CPU-GPU Distributed Framework for Large Scale Mobile Networks Simulation. In *Distributed Simulation and Real Time Applications (DS-RT), 2012 IEEE/ACM 16th International Symposium on*. 44–53.

M. Bohr. 2007. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE* 12, 1 (2007), 11–13.

T. Brandvik and G. Pullan. 2010. 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 (CIT '10)*. IEEE Computer Society, Washington, DC, USA, 1181–1188.

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

D. A. Burgess and M. B. Giles. 1997. Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Adv. Eng. Softw.* 28 (April 1997), 189–201. Issue 3.

M. Christen, O. Schenk, and H. Burkhart. 2011. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*. 676–687. DOI:http://dx.doi.org/10.1109/IPDPS.2011.70

A. Cohen, T. Grosser, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. 2013. Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles to Reconcile Parallelism and Locality, avoiding Divergence and Load Imbalance. In *GPGPU 6 - Sixth Workshop on General Purpose Processing Using GPUs*. Houston, États-Unis.

Convey 2013. The Convey HC-1 Computer. (2013). http://www.conveycomputer.com/Resources/ConveyArchitectureWhiteP.pdf.

K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. 2000. Generative Programming and Active Libraries. In *Selected Papers from the International Seminar on Generic Programming*. Springer-Verlag, London, UK, 25–39.

Deal.II 2013. Deal.II: A Finite Element Differential Equations Analysis Library. (2013). http://www.dealii.org/.

J. D. Denton. 1997. Lessons from rotor 37. *Journal of Thermal Science* 6, 1 (1997), 1–13.

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. 2011. 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 (SC '11)*. ACM, New York, NY, USA, Article 9, 12 pages.

DUNE 2013. DUNE - Distributed and Unified Numerics Environment. (2013). http://www.dune-project.org/.

FEATFLOW 2013. FEATFLOW - High Performance Finite Elements. (2013). http://www.featflow.de/en/index.html.

FMDB 2013. Flexible Distributed Mesh DataBase. (2013). http://www.scorec.rpi.edu/FMDB/.

M. B. Giles, M. C. Duta, J. D. Muller, and N. A. Pierce. 2003. Algorithm Developments for Discrete Adjoint Methods. *AIAA Journal* 42, 2 (2003), 198–205.

M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. 2011. Performance analysis of the OP2 framework on many-core architectures. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (March 2011), 9–15.

M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. J. Kelly. 2012. Performance Analysis and Optimization of the OP2 Framework on Many-Core Architectures. *Comput. J.* 55, 2 (2012), 168–180.

M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. 2013. Designing OP2 for GPU Architectures. *J. Parallel and Distrib. Comput.* 73 (November 2013), 1451–1460. Issue 11.

GROMACS 2013. Hybrid-CPU/GPU execution mode with GROMACS. (2013). http://www.scalalife.eu/content/hybrid-cpu-gpu-execution-mode-gromacs.

R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. 2012. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE* 32, 2 (2012), 48–60.

HECToR 2013. HECToR - Hardware. (2013). http://www.hector.ac.uk/service/hardware/.

T. Henretty, R. Veras, F. Franchetti, L Pouchet, J. Ramanujam, and P. Sadayappan. 2013. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS '13)*. 13–24. DOI:http://dx.doi.org/10.1145/2464996.2467268

J. Holewinski, L. Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing (ICS '12)*. 311–320. DOI:http://dx.doi.org/10.1145/2304576.2304619

L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. J. Kelly. 2009. Deriving Efficient Data Movement from Decoupled Access/Execute Specifications. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC '09)*. Springer-Verlag, Berlin, Heidelberg, 168–182.

S. Husa, I. Hinder, and C. Lechner. 2006. Kranc: a Mathematica package to generate numerical codes for tensorial evolution equations. *Computer Physics Communications* 174, 12 (2006), 983 – 1004. DOI:http://dx.doi.org/10.1016/j.cpc.2006.02.002

Interlagos Nov 2011. How to make best use of the AMD Interlagos processor. (Nov 2011). www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf.

ISO/IES. 2012. *Further Interoperability of Fortran with C*. ISO SO/IEC TS 29113:2012. International Organization for Standardization and International Electrotechnical Commission, Geneva, Switzerland.

JADE 2013. Oxford Supercomputing Center. (2013). http://www.osc.ox.ac.uk/content/services.

J. Jägersküpper and C. Simmendinger. 2011. A Novel Shared-Memory Thread-Pool Implementation for Hybrid Parallel CFD Solvers. In *Euro-Par 2011 Parallel Processing*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Lecture Notes in Computer Science, Vol. 6853. Springer Berlin / Heidelberg, 182–193.

R. P. LaRowe Jr. and C. S. Ellis. 1991. Page placement policies for {NUMA} multiprocessors. *J. Parallel and Distrib. Comput.* 11, 2 (1991), 112 – 129. DOI:http://dx.doi.org/10.1016/0743-7315(91)90117-R

K20 2012. Tesla Kepler GPU Accelerators. (2012). http://www.nvidia.co.uk/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf.

B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. 2006. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers* 22, 3–4 (2006), 237–254. http://dx.doi.org/10.1007/s00366-006-0049-3.

C. Lameter. 2013. An overview of non-uniform memory access. *Commun. ACM* 56, 9 (Sept. 2013), 59–54.

L. Lapworth. Sept, 2008. 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.

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

P. Lindberg. 2009. *Basic OpenMP Threading Overhead*. Technical Report. Intel. http://software.intel.com/en-us/articles/basic-openmp-threading-overhead.

O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and Haohuan Fu. 2011. Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications. *Micro, IEEE* 31, 2 (March - April 2011), 41 –49.

M.B. Giles M.C. Duta and M.S. Campobasso. 2002. The harmonic adjoint approach to unsteady turbomachinery design. *International Journal for Numerical Methods in Fluids* 40 (2002), 323–332. Issue 3-4.

R. Membarth, F. Hannig, J. Teich, and H. Kostler. 2012. Towards Domain-Specific Computing for Stencil Codes in HPC. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. 1133–1138. DOI:http://dx.doi.org/10.1109/SC.Companion.2012.136

G.R. Mudalige, M.B. Giles, J. Thiyagalingam, I. Reguly, C. Bertolli, P.H.J. Kelly, and A.E. Trefethen. in press 2013. Design and Initial Performance of a High-level Unstructured Mesh Framework on Heterogeneous Parallel Systems. *Journal of Parallel Computing* (in press 2013).

G. R. Mudalige, I. Reguly, M. B. Giles, C. Bertolli, and P. H. J. Kelly. 2012. OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures.. In *Proceedings of Innovative Parallel Computing (InPar '12)*. IEEE, San Jose, CA. US.

T. Muranushi. 2012. Paraiso : An Automated Tuning Framework for Explicit Solvers of Partial Differential Equations. *Computational Science & Discovery* 5, 1 (2012), 015003.

NVIDIA GPUs 2013. What is GPU Computing. (2013). http://www.nvidia.com/object/what-is-gpu-computing.html.

K. B. Ølgaard, A. Logg, and G. N. Wells. 2011. Automated code generation for discontinuous Galerkin methods. *CoRR* abs/1104.0628 (2011).

OP2 2011. OP2 for Many-Core Platforms. (2011). http://www.oerc.ox.ac.uk/research/op2.

OP2-GIT 2013. OP2 GitHub Repository. (2013). https://github.com/OP2/OP2-Common.

OpenACC 2013. OpenACC: Directives For Accelerators. (2013). http://www.openacc-standard.org/.

OpenCFD. 2013. *OpenFOAM - The Open Source CFD Toolbox - User's Guide* (2.2 ed.). OpenCFD Ltd., United Kingdom.

D. A. Orchard, M. Bolingbroke, and A. Mycroft. 2010. Ypnos: Declarative, Parallel Structured Grid Programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming (DAMP '10)*. ACM, New York, NY, USA, 15–24.

J.-D. Muller P. Moinier and M.B. Giles. 2002. Edge-based multigrid and preconditioning for hybrid grids. *AIAA Journal* 40 (2002), 1954–1960. Issue 10.

ParMETIS 2013. ParMETIS. (2013). http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

PETSc 2013. PETSc. (2013). http://www.mcs.anl.gov/petsc/petsc-as/.

PT-Scotch 2013. Scotch and PT-Scotch. (2013). http://www.labri.fr/perso/pelegrin/scotch/.

N. Rajovic, N. Puzovic, L. Vilanova, C. Villavieja, and A. Ramirez. 2011. The low-power architecture approach towards exascale computing. In *Proceedings of the second workshop on Scalable algorithms for large-scale systems (ScalA '11)*. ACM, New York, NY, USA, 1–2. DOI:http://dx.doi.org/10.1145/2133173.2133175

F. Rathgeber, G. R. Markall, L. Mitchell, M. Loriant, D. A. Ham, C. Bertolli, and P. H. J. Kelly. 2012. PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes. In *High Performance Computing, Networking, Storage and Analysis, 2012 SC Companion (SCC)*. 1116–1123.

SandyBridge 2013. Intel Xeon Processor E5-2640. (2013). http://ark.intel.com/products/64591/Intel-Xeon-Processor-E5-2640-15M-Cache-2_50-GHz-7_20-GTs-Intel-QPI.

H. D. Simon. 1991. *Partitioning of Unstructured Problems for Parallel Processing*. Technical Report RNR-91-008. National Aeronautics and Space Administration, Ames Research Center Moffett Field, California 94035-1000. www.nas.nasa.gov/assets/pdf/techreports/1991/rnr-91-008.pdf

K. Skaugen. June 2011. Petascale to Exascale: Extending Intel's HPC Commitment. (June 2011). ISC 2010 keynote. http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf.

J. R. Stewart and H. C. Edwards. 2004. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elem. Anal. Des.* 40 (July 2004), 1599–1617. Issue 12.

Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C. Luk, and C. E. Leiserson. 2011. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures (SPAA '11)*. 117–128. DOI:http://dx.doi.org/10.1145/1989493.1989508

TI 2013. Texas Instruments Multi-core TMS320C66x Processor. (2013). http://www.ti.com/c66multicore.

D. Unat, X. Cai, and S. B. Baden. 2011. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the international conference on Supercomputing (ICS '11)*. 214–224. DOI:http://dx.doi.org/10.1145/1995896.1995932

T. L. Veldhuizen and D. Gannon. 1998. Active Libraries: Rethinking the roles of compilers and libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO98)*. SIAM Press.

VTune 2013. Intel VTune Product Page. (2013). http://software.intel.com/en-us/intel-vtune-amplifier-xe.

C. Yang, W. Xue, H. Fu, L. Gan, L. Li, Y. Xu, Y. Lu, J. Sun, G. Yang, and W. Zheng. 2013. 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 (PPoPP '13)*. ACM, New York, NY, USA, 1–12. DOI:http://dx.doi.org/10.1145/2442516.2442518

M. Zhou, O. Sahni, T. Xie, M. S. Shephard, and K. E. Jansen. 2012. Unstructured Mesh Partition Improvement for Implicit Finite Element at Extreme Scale. *J. Supercomput.* 59, 3 (March 2012), 1218–1228.