Design and Initial Performance of a High-level Unstructured Mesh Framework on Heterogeneous Parallel Systems

G.R. Mudalige[a,*], M.B. Giles[a], J. Thiyagalingam[a], I. Reguly[a], C. Bertolli[b], P.H.J. Kelly[c], A.E. Trefethen[a]

[a] *Oxford e-Research Centre, University of Oxford, UK*
[b] *IBM TJ Watson Research Centre, New York, USA*
[c] *Dept. of Computing, Imperial College London, UK*

Abstract

OP2 is a high-level domain specific library framework for the solution of unstructured mesh-based applications. It utilizes source-to-source translation and compilation so that a single application code written using the OP2 API can be transformed into multiple parallel implementations for execution on a range of back-end hardware platforms. In this paper we present the design and performance of OP2's recent developments facilitating code generation and execution on distributed memory heterogeneous systems. OP2 targets the solution of numerical problems based on static unstructured meshes. We discuss the main design issues in parallelizing this class of applications. These include handling data dependencies in accessing indirectly referenced data and design considerations in generating code for execution on a cluster of multi-threaded CPUs and GPUs. Two representative CFD applications, written using the OP2 framework, are utilized to provide a contrasting benchmarking and performance analysis study on a number of heterogeneous systems including a large scale Cray XE6 system and a large GPU cluster. A range of performance metrics are benchmarked including runtime, scalability, achieved compute and bandwidth performance, runtime bottlenecks and systems energy consumption. We demonstrate that an application written once at a high-level using the OP2 API is easily portable across a wide range of contrasting platforms and is capable of achieving near-optimal performance without the intervention of the domain application programmer.

Keywords: OP2, Domain Specific Language, Active Library, Unstructured mesh, GPU, Heterogeneous systems, Energy consumption of parallel systems

1. Introduction

Heterogeneous parallel computing systems are becoming an increasingly common architecture adopted by many High Performance Computing (HPC) systems designers and vendors. Based on the current generation of CMOS micro-processor technology, it is seen as a key path towards reaching the exaFlop levels of performance, within a tractable energy envelope. In a heterogeneous system, traditional processors will be augmented by an attached co-processor (usually called an accelerator) which gives higher performance for solving certain types of computations. Current examples of heterogeneous combinations range from traditional x86 processors accelerated by discrete (e.g. NVIDIA GPUs [1], Intel MIC [2]) or integrated (e.g. AMD APUs [3]) SIMD type co-processors to more specialized DSP type compute engines [4] or FPGA based accelerators [5, 6]. To gain good performance from such systems, in addition to the complexity in programming the accelerator devices, the applications developer needs to manage concurrency between the different processors/co-processors and manage data locality within a complicated memory hierarchy. Maintaining high performance becomes even more complicated on clusters of heterogeneous nodes, which are essential for production applications requiring significantly more compute resources and capabilities.

Recent efforts in utilizing heterogeneous parallel systems in HPC [7, 8, 9] have mostly discussed hand-porting CPU based code (often single threaded sequential CPU code) to accelerator based systems. However this process is time-consuming, error-prone and takes a significant amount of software development effort by programmers who need to be experts in the new heterogeneous architectures. Emerging programming languages and extensions such as OpenACC [10] attempt to alleviate some of the burden in programming these systems, but the programmer still has to "instruct" the compiler where and how to identify and exploit parallel regions in the code for the accelerator. This may require significant effort, particularly for large production HPC applications. Furthermore, such directive based approaches are yet to provide evidence of achieving good performance for some classes of applications in comparison to hand-ported implementations that utilize specific many-core programming languages such as CUDA or OpenCL.

As the parallel processor landscape changes rapidly,

*Corresponding author, Address: Oxford e-Research Centre, University of Oxford, 7 Keble Road, Oxford. OX1 3QG, U.K., Tel: +44 (0)1865 610787, Fax: +44 (0)1865 610612

Email addresses: gihan.mudalige@oerc.ox.ac.uk (G.R. Mudalige), mike.giles@maths.ox.ac.uk (M.B. Giles), jeyarajan.thiyagalingam@oerc.ox.ac.uk (J. Thiyagalingam), istvan.reguly@oerc.ox.ac.uk (I. Reguly), cbertol@us.ibm.com (C. Bertolli), p.kelly@imperial.ac.uk (P.H.J. Kelly), anne.trefethen@oerc.ox.ac.uk (A.E. Trefethen)

there is a need to constantly maintain an expert level of knowledge in the intricate details of new technologies and heterogeneous architectures in order to obtain the best performance from applications. Application developers would like to benefit from the performance gains promised by these new systems, but are concerned about the associated costs of software development. One solution is to utilize domain-specific knowledge about applications in developing a suitable abstraction to increase productivity and maintain performance. This allows for scientists and engineers to develop code based on a higher level, writing applications that remain unchanged for different underlying hardware. At the same time, a lower implementation level provides opportunity for parallel programming experts to apply radically aggressive and platform specific optimizations when implementing the required solution on various hardware. 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.

OP2 aims to provide such an abstraction layer for the solution of unstructured mesh-based applications. OP2 uses an active library approach [11, 12] where a single application code written using the OP2 API can be transformed 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 from OP2 and the 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 papers we presented OP2's API and its back-end design facilitating the development and execution of unstructured mesh applications on single-node CPU and GPU systems [13, 14, 15, 16, 17] and performance on large CPU clusters [18]. These back-end designs enable the OP2 framework to generate code targeting: (1) single-threaded CPUs, (2) multi-threaded CPUs/SMPs, (3) single NVIDIA GPUs and (4) distributed memory clusters of single threaded CPUs. In this paper we present the next stage of the OP2 framework: the design facilitating the code generation and execution on multiple heterogeneous parallel systems, particularly incorporating distributed memory and many-core parallelism, and its performance. The contributions of this paper are twofold:

1. Design: We present the design of OP2's heterogeneous back-end which allows an application written using OP2 to be executed on a cluster of GPUs (using MPI and NVIDIA CUDA) and a cluster of multi-threaded CPUs (using MPI and OpenMP). This work specifically focuses on the combination of distributed memory and intra-node or many-core/multi-core parallelism. Key design issues such as (1) handling race-conditions due to data dependencies in indirectly accessed data, and (2) optimizing communications for the target heterogeneous/hybrid parallelization strategies are explored.
2. Performance: A range of performance metrics are benchmarked to explore the performance of OP2's heterogeneous back-end design, including runtime, scalability, achieved compute and bandwidth performance and system energy consumption. Two industrial-representative unstructured mesh application benchmarks (Airfoil and Aero) written using OP2 are used for this study. Benchmarked systems include three large-scale distributed memory systems (a Cray XE6, an Intel Westmere/InfiniBand cluster and a distributed memory Tesla M2090 GPU cluster interconnected by QDR InfiniBand). The OP2 design choices are explored with quantitative insights into their contributions to performance on these systems. We also isolate performance bottlenecks of the application by breaking down the runtime and analyzing the factors constraining total performance. The energy performance measurements and analysis investigate whether the speedups (if any) gained through OP2 on the heterogeneous platforms are achieved within a proportionate energy envelope.

We believe that the design of OP2 for heterogeneous clusters and its performance based on standard benchmarks, forms an important step forward with respect to our previous work [13, 14, 15, 16, 18, 17]. With the inclusion of the heterogeneous back-ends, the range of target platforms supported by OP2 gives us a unique opportunity to carry out an extensive study into the comparative performance of modern systems. As such, this paper details the most comprehensive platform comparison we have carried out to date with OP2 including a systems energy performance analysis, a metric that is becoming increasingly important for comparing modern processor systems. We use highly-optimized code generated through OP2 for all back-ends, using the same application code, allowing for a direct performance comparison.

Our results demonstrate 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 domain application programmer.

2. 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 [19], Sierra [20] libraries as well as others such as Deal.II [21], Dune [22] and FEATFLOW [23]. Recent research in this area include unstructured mesh frameworks that allows extreme scaling (up to 300K cores) on distributed memory CPU

clusters [24] while libraries such has PETSc have also implemented hand tuned back-ends targeting solvers on distributed memory clusters of GPUs. Other related work include RPI's FMDB [25] and LibMesh [26]. The research at RPI is especially relevant to the OP2 concepts of sets of mesh entities and mappings between sets. A major goal of the LibMesh library is to provide support for adaptive mesh refinement (AMR) computations. There are also conventional applications such as the computational fluid dynamics (CFD) solver TAU [27] which attempts to extend its capabilities to heterogeneous hardware for accelerating applications. 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. In contrast to LibMesh, OP2 only supports the solution of static mesh based problems.

OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [28] 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 [29] and FEniCS [30] specifically target mesh based computations.

The FEniCS [30] 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. Although well established finite element methods could be supported by such a declarative abstraction, it lacks the flexibility offered by frameworks such as OP2 for developing new applications/algorithms. Currently, a runtime code generation, compilation and execution framework that is based on Python, called PyOP2 [31], is being developed at Imperial College London to enable FEniCS declarations to use the OP2 back-ends. Thus, the performance results in this paper will be directly applicable to the performance of code written using FEniCS in the future.

While OP2 uses an active library approach utilizing code transformation, Liszt [29] from Stanford University implements a domain specific language (embedded in Scala [32]) for the solution of unstructured mesh based partial differential equations (PDEs). A Liszt application is translated to an intermediate representation which is then compiled by the Liszt compiler to generate native code for multiple platforms. 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 [29]. The Navier-Stokes application in [29] is most comparable to the Airfoil application 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 [29] 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 [33], Ypnos [34] and SBLOCK [35]. 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 [34] 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 [33] is a domain-specific language embedded in Haskell, for the automated tuning of explicit solvers of partial differential equations (PDEs) on GPUs, and multi-core CPUs. It uses algebraic concepts such as tensors, hydrodynamic properties, interpolation methods and other building blocks in describing the PDE solving algorithms. In contrast SBLOCK [35] uses extensive automatic source code generation very much similar to the approach taken by OP2, and expresses computations as kernels applied to elements of a set.

Recently, directive based approaches have emerged as a much publicized solution to programming heterogeneous systems. Most notable of these, OpenACC [10] attempts to follow the accessibility and success of OpenMP in programming parallel systems but specifically targets accelerator based systems. However, our experience has been that for applications with significant non-affine array access (such as unstructured mesh applications with indirect array accesses) directive based approaches fail to provide the best performance unless significant effort is made to explicitly optimize their code by application developers. 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.

This paper documents a number of significant developments in the design of OP2's heterogeneous back-ends and

their performance extending our previous work in [36]: (1) A major contribution is the development of OP2's MPI+OpenMP back-end design and performance which augments the MPI only and MPI+CUDA implementations. This new back-end provides key insights into the performance limiting factors of modern multi-core clusters, particularly demonstrating the issues encountered on NUMA type architectures of multi-core nodes. Additionally the extra scalability achieved by using MPI with OpenMP, for this class of applications, at increasingly higher scale, demonstrates extremely competitive performance rivaling GPU clusters. (2) A new finite element benchmark (Aero) is developed with OP2 and its performance is benchmarked and analyzed in addition to the standard Airfoil CFD application. Both Airfoil and Aero are utilized to elucidate how the OP2 API can be used to develop simulation codes to realize the required unstructured mesh methods. Aero has a higher bandwidth utilization in contrast to Airfoil which is much more compute intensive. Results from Aero enable to confirm the insights gained from Airfoil, significantly increasing the confidence of our performance analysis and conclusions. (3) In contrast to our previous work, this paper provides runtime performance and scalability on new platforms - a large-scale CrayXE6 system with compute nodes consisting of AMD's Interlagos processors based on the Bulldozer architecture [37], an Intel Westmere cluster interconnected by QDR InfiniBand and a large distributed memory GPU cluster with up to 150 GPUs. Both strong- and weak-scaling is benchmarked and analyzed on all the cluster systems demonstrating the scalability of the codes and the factors that need to be taken into account to maintain performance. (4) We also present the breakdown of runtimes of each parallel loop for both applications at scale on all three clusters. This allows us to comprehensively identify the performance bottlenecks of the applications as well as OP2 at the inter-node level. Additionally for the GPU cluster we present the achieved average bandwidth figures on GPUs at increasing scale. The analysis further demonstrates the performance limiting factors on the GPU cluster. (5) Finally, we measure and estimate the energy performance of OP2's heterogeneous back-ends for Airfoil and Aero. This is a novel performance metric that is becoming increasingly important in HPC performance studies. In conjunction with the runtime performance results it provides valuable insights into the total cost of ownership and running cost of the clusters given a workload consisting of unstructured mesh applications.

3. OP2

Unstructured grids/meshes have been, and continue to be, used for a wide range of computational science and engineering applications. They have been applied in the solution of partial differential equations (PDEs) in computational fluid dynamics (CFD), structural mechanics, computational electro-magnetics (CEM) and general finite element methods. In three dimensions, millions of elements

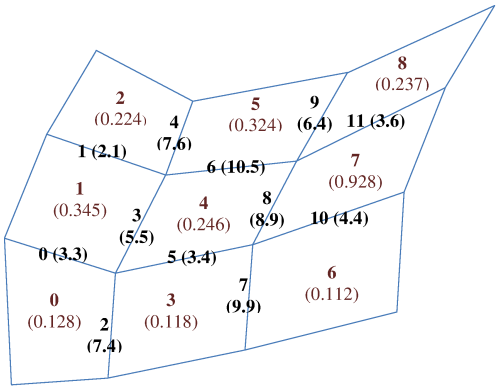


Figure 1: An example mesh with edge and quadrilateral cell indices (data values in parenthesis)

are often required for the desired solution accuracy, leading to significant computational costs.

Unstructured meshes, unlike structured meshes, use explicit connectivity information to specify the mesh topology. The OP2 approach to the solution of unstructured mesh problems (based on ideas developed in its predecessor OPlus [38, 39]) involves separating the algorithm 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 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) and mappings between sets defining how elements of one set connect with the elements of another set.

In our previous work [13, 14] we have presented in detail the design of the OP2 API. For completeness, here we give an overview of the API using the simple quadrilateral mesh illustrated in Figure 1. The OP2 API supports program development in C/C++ and Fortran. We use the C/C++ API in this paper; an illustration of the OP2 API based on Fortran is detailed in [15, 17]. The mesh in Figure 1 can be defined by three sets: edges, cells (quadrilaterals) and boundary edges. There are 12 edges, 9 cells and 12 boundary edges which can be defined using the OP2 API as follows:

```
int nedges = 12; int ncells = 9; int nbedges = 12;
op_set edges = op_decl_set(nedges, "edges");
op_set cells = op_decl_set(ncells, "cells");
op_set bedges = op_decl_set(nbedges, "bedges");
```

The connectivity is declared through the mappings between the sets. Considering only the interior edges in this example, the integer array `edge_to_cell` gives the connectivity between cells and interior edges.

```
int edge_to_cell[24] = {0,1, 1,2, 0,3, 1,4, 2,5,
              3,4, 4,5, 3,6, 4,7, 5,8, 6,7, 7,8};
op_map pecell = op_decl_map(edges, cells, 2,
                edge_to_cell, "edge_to_cell_map");
```

Each element belonging to the set `edges` is mapped to two different elements in the set `cells`. The `op_map` declaration defines this mapping where `pecell` has a dimension of 2 and thus its index 0 and 1 maps to cells 0 and 1, index 2 and 3 maps to cells 1 and 2 and so on. When declaring a mapping we first pass the source set (e.g. `edges`) then the destination set (e.g. `cells`). Then we pass the dimension (or arity) of each map entry (e.g. 2; as `pecell` maps each edge to 2 cells). Note that this literal numbers filling up the mapping array is purely for description purposes only, as in OP2 these will be read from disk (e.g. from HDF5 files). Once the sets are defined, data can be associated with the sets; the following are some data arrays that contain double precision data associated with the cells and the edges respectively. 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 cell to store coordinates).

```
double cell_data[9] = {0.128, 0.345, 0.224, 0.118,
                       0.246, 0.324, 0.112, 0.928,
                       0.237};
double edge_data[12] = {3.3, 2.1, 7.4, 5.5, 7.6,
                        3.4, 10.5, 9.9, 8.9, 6.4,
                        4.4, 3.6};
op_dat dcells   = op_decl_dat(cells, 1, "double",
                         cell_data, "data_on_cells");
op_dat dedges   = op_decl_set(edges, 1, "double",
                         edge_data, "data_on_edges");
```

All the numerically intensive computations in the unstructured mesh application 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. If the loop involves indirection through a mapping, OP2 denotes it as an *indirect* loop; if not, it is called a *direct* loop. The OP2 API provides a parallel loop declaration syntax which allows the user to declare the computation over sets in these loops. Consider the following sequential loop, operating over each interior edge in the mesh illustrated in Figure 1. Each of the cells updates its data value using the data values held on the edge connected to that cell and the corresponding neighboring cell.

```
void res_seq_loop(int nedges, int *edge_to_cell,
              double *edge_data, double *cell_data) {
  for (int i = 0; i<nedges; i++){
      cell_data[edge_to_cell[2*i]]   += edge_data[i];
      cell_data[edge_to_cell[2*i+1]] += edge_data[i];
  }
}
```

An application developer declares this loop using the OP2 API as follows, together with the "elemental" kernel function. This elemental function is called a "user kernel" in OP2 to indicate that it represents a computation specified by the user (i.e. the domain scientist) to apply to each element.

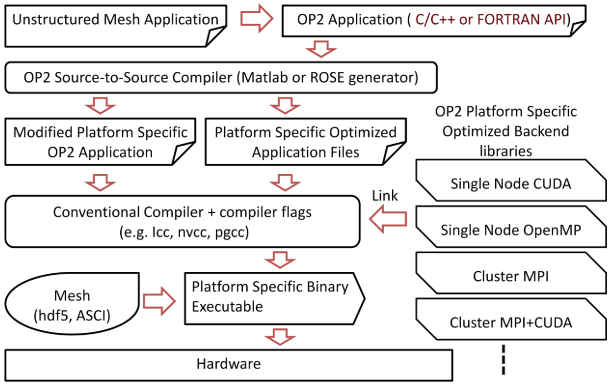


Figure 2: OP2 build hierarchy

```
void res(double* edge, double* cell0, double* cell1){
  *cell0 += *edge;
  *cell1 += *edge;
}
op_par_loop(res,"residual_calculation", edges,
   op_arg(dedges, -1, OP_ID,  1, "double", OP_READ),
   op_arg(dcells,  0, pecell, 1, "double", OP_INC),
   op_arg(dcells,  1, pecell, 1, "double", OP_INC));
```

The user kernel function takes 3 arguments in this case and the parallel loop declaration requires the access method of each to be declared (OP_INC, OP_READ, etc). OP_ID indicates that the data in `dedges` is to be accessed without any indirection (i.e. directly). `dcells` on the other hand is accessed through the `pecell` mapping using the given index (0 and 1). The dimension (or cardinality) of the data (in this example 1, for all data) is also declared.

The OP2 compiler handles the architecture specific code generation and parallelization. An application written using the OP2 API will be parsed by the OP2 compiler which will produce a modified main program and back-end specific code (see Figure 2). These are then compiled using a conventional compiler (e.g. gcc, icc, nvcc) and linked against platform specific OP2 back-end libraries to generate the final executable. The mesh data to be solved is input at runtime. In the OP2 project we currently have three prototype compilers, two of them (one written in MATLAB and the other in Python) only parse OP2 calls providing a simple source-to-source code generation prototype. A third compiler built using the ROSE compiler framework [40] is capable of full source code analysis. Preliminary details of the ROSE source-to-source translator can be found in [15, 17]. The slightly verbose API was needed as a result of the MATLAB prototype parser and the python parser, but also facilitates consistency checks to identify user errors during application development.

One could argue that most if not all of the details that an `op_par_loop` specifies could be inferred automatically just by parsing a conventional loop (e.g. `res_seq_loop`

in the above example) without the need for a specialized API such as used in OP2. However, in industrial-strength applications it is typically hard or impossible to perform full program analysis due to an over-complex control flow and the impossibility of full pointer analysis. The syntax presented in this paper permits instead the definition of "generic" routines which can be applied to different datasets, giving the compiler the opportunity to achieve code analysis and synthesis in a simple and straightforward way.

OP2's general decomposition of unstructured mesh algorithms imposes no limitation on the actual algorithms, it just separates the components of a code [13, 14]. However, OP2 makes an important restriction that the order in which elements are processed must not affect the final result, to within the limits of finite precision floating-point arithmetic. This constraint allows OP2 to choose its own order to obtain maximum parallelism, which on platforms such as GPUs is crucial to gain good performance. We consider that this is a reasonable limitation of OP2 considering that all high performance implementations for unstructured grids do extensive renumbering for MPI partitioning and also cache optimization [41], and accept the loss of bit-wise reproducibility. However, if this is a concern for some users, then one option is to move to approximate quad-precision [42] using two double-precision variables for the local summations so that it becomes very much less likely to get even a single bit difference when truncating back to double precision. This technique requires four floating point operations instead of one, but in most applications this is unlikely to increase the overall operation count by more than 5-10%, so in practice the main concern is probably the additional memory requirements. The same approach could also be used to greatly reduce the variation in the results from global summations.

Another restriction in OP2 is that the sets and mappings between sets must be static and the operands in the set operations cannot be referenced through a double level of mapping indirection (i.e. a mapping to another set which in turn uses another mapping to access data associated with a third set). The straightforward programming interface combined with efficient parallel execution makes it an attractive prospect for the many algorithms which fall within the scope of OP2. For example the API could be used for explicit relaxation methods such as Jacobi iteration; pseudo-time-stepping methods; multi-grid methods which use explicit smoothers; Krylov subspace methods with explicit preconditioning; semi-implicit methods where the implicit solve is performed within a set member, for example performing block Jacobi where the block is across a number of PDE's at each vertex of a mesh. However, algorithms based on order dependent relaxation methods, such as Gauss-Seidel or ILU (incomplete LU decomposition), lie beyond the current capabilities of the framework. The OP2 API could be extended to handle such sweep operations, but the loss in the degree of parallelism available means that it seems unlikely one would obtain good parallel performance on heterogeneous systems such as clusters of GPUs [8].

Previously, OP2 supported generating parallel code for execution on a single-threaded CPU, a single SMP system based on multi-core CPUs using OpenMP, a single NVIDIA GPU using CUDA and a cluster of CPUs using MPI. The research presented in the next section adds to these back-ends by facilitating code generation and execution on heterogeneous systems by encompassing distributed memory and multi-core/many-core parallelization: (1) cluster of GPUs (based on MPI and CUDA) and (2) cluster of multi-threaded CPUs (based on MPI and OpenMP). While the first type is currently the most common form of heterogeneous system installed [43], the second is a traditional extension to homogeneous clusters with SMP or CMP nodes allowing us to utilize both distributed memory and thread-level parallelism. The MPI+OpenMP combination is rapidly becoming an unavoidable combination, due to the major micro-processor developers' design direction of adding more and more cores to a single silicon chip. It would be unfeasible to have one MPI process per core while the number of cores per node reaches hundreds or thousands. Furthermore, we believe that the MPI+OpenMP combination will be important for programming heterogeneous systems such as the Intel Xeon Phi (based on their MIC architecture [2]) in the future. As such, this back-end library will be extended later with MIC processor instructions (including AVX) utilizing the larger SIMD width (512-bit) on the processor. Thus the MPI+OpenMP implementation is a key design step in supporting emerging heterogeneous platforms and also will provide good baseline performance for comparison with more explicit implementations such as ones based on OpenCL.

4. Heterogeneous Systems

For heterogeneous systems at least two layers of parallelization needs to be utilized simultaneously (1) distributed memory (process level parallelism) and (2) single-node/shared-memory (thread level parallelism). To provide context to the OP2 design, in the next sections, Section 4.1 and 4.2, we briefly review OP2's parallelization strategy and design from [14, 18] for single node and distributed memory systems. Then in Section 4.3 we discuss the design for heterogeneous cluster systems.

4.1. OP2 Parallelization Strategy

The OP2 distributed memory level is based on MPI and uses standard graph partitioning techniques (similar to ideas developed previously in OPlus [38, 39]) in which the domain is partitioned among the compute nodes of a cluster, and import/export halos are constructed for message-passing. A key issue impacting performance with the above design is the size of the halos which directly determines the size of messages passed when a parallel loop is executed. Our assumption is that the proportion

of halo data becomes very small as the partition size becomes large. This depends on the quality of partitions held by each MPI process. OP2 utilizes two well established parallel mesh partitioning libraries, ParMETISs [44] and PT-Scotch [45] to obtain high quality partitions.

The single-node design is motivated by several key factors. Firstly a single node may have different kinds of parallelism depending on the target hardware: on multi-core CPUs shared memory multi-threading is available with the possibility of each thread using vectorization to exploit the capabilities of SSE/AVX vector units. On GPUs, multiple thread blocks are available with each block having multiple threads. Secondly, memory bandwidth is a major limitation on both existing and emerging processors. In the case of CPUs this is the bandwidth between main-memory and the CPU cores, while on the GPUs this is the bandwidth between the main graphics (global) memory and the GPU cores. Thus the OP2 design is motivated to reduce the data movement between memory and cores.

In the specific case of GPUs, the size of the distributed-memory partition assigned to each GPU is constrained to be small enough to fit entirely within the GPU's global memory. This means that the only data exchange between the GPU and the host CPU is for the halo exchange with other GPUs. Within the GPU, for each parallel loop with indirect referencing, the partition is sub-divided into a number of mini-partitions; these are sized so that the required indirectly-accessed data will fit within the 48kB of shared memory in the SM. Each thread block first loads the indirectly-accessed data into the shared memory, and then performs the desired computations. Using shared memory instead of L1 cache makes maximum use of the available local memory; caches hold the entire cache line even when only part of it may be needed. This approach requires some tedious programming to use locally renumbered indices for each mini-partition, but this is all handled automatically by OP2's run-time routines. Since the global memory is typically 3 to 6 GB in size, and each mini-partition has at most 48kB of data, the total number of mini-partitions is very large, usually more than 10,000. This leads naturally to very good load-balancing across the GPU. Within the mini-partition, each thread in the thread block works on one or more elements of the set over which the operation is parallelized; the only potential difficulty here concerns the data dependencies addressed in the next section.

In standard MPI computations, because of the cost of re-partitioning, the partitioning is usually done just once and the same partitioning is then used for each stage of the computation. In contrast, for single node CPU and GPU executions, between each stage of the computation the data resides in the main memory (on a CPU node) or the global memory (on a GPU), and so the mini-partitioning and thread block size for each parallel loop calculation can be considered independently of the requirements of the other parallel loops. Based on ideas from FFTW [46], OP2 constructs for each parallel loop an execution "plan" (`op_plan`) which is a customized mini-partition and thread-block size execution template. Thus on a GPU the execution of a given loop makes optimum use of the local shared memory on each multiprocessor considering in detail the memory requirements of the loop computation. OP2 allows the user to set the mini-partition and thread block size both at compile and run-time for each loop, allowing for exploring the best values for these parameters for a given application. In [16] we explored the performance gains from auto-tuning the mini-partition and thread block size for the Airfoil application on single node (CPU and GPU) systems.

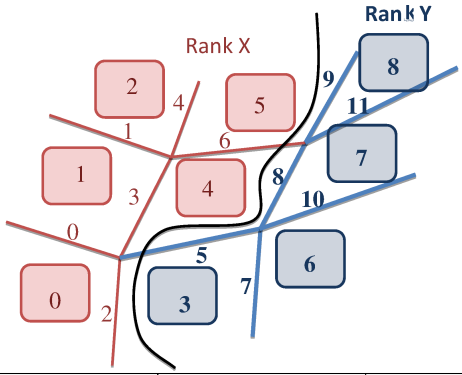
4.2. Data Dependencies

One key design issue in parallelizing unstructured mesh computations is managing data dependencies encountered when incrementing indirectly referenced arrays [13, 14]. For example, in a mesh with cells and edges, with a loop over edges updating cells (as in the `op_par_loop` example above) a potential problem arises when multiple edges update the same cell.

At the higher distributed-memory level, we follow the OPlus approach [38, 39] in using an "owner compute" model in which the partition which "owns" a cell is responsible for performing the edge computations which will update it. If the computations for a particular edge will update cells in different partitions, then each of those partitions will need to carry out the edge computation. This redundant computation is the cost of this approach. However, we assume that the distributed-memory partitions are very large such that the proportion of redundant computation becomes very small.

The current implementation is based on MPI. OP2 partitions the data so that the partition within each MPI process owns some of the set elements e.g. some of the cells and edges. These partitions only perform the calculations required to update their own elements. However, it is possible that one partition may need to access data which belongs to another partition; in that case a copy of the required data is provided by the other partition. This follows the standard "halo" exchange mechanism used in distributed memory message passing parallel implementations. Figure 3 illustrates OP2's distributed memory partitioning strategy for a mesh with edges and cells and a mapping between two cells to one edge.

The *core* elements do not require any halo data and thus can be computed without any MPI communications. This allows for overlapping of computation with communications using non-blocking MPI primitives for higher performance. The elements in the import and export halos are further separated into two groups depending on whether redundant computations will be performed on them. For example edges 5, 8 and 9 on rank Y form part of the import halo on rank X and a loop over edges will require these edges to be executed by rank X, in order for values held on cells 4 and 5 on rank X to be correctly updated/calculated. The import non-exec elements are, on the other hand, a



	Set	*core*	*export halo*		*import halo*	
			exec	non-exec	exec	non-exec
X	edges	0,1,3,4,6	2	-	5,8,9	-
X	cells	0,1,2,4,5	-	0,4,5	-	3,7,8
Y	edges	7,10,11	5,8,9	-	2	-
Y	cells	3,6,7,8	-	3,7,8	-	0,4,5

Figure 3: OP2 partitioning over two MPI ranks and resulting halos on each rank

read-only halo that will not be executed, but is referenced by other elements during their execution. Thus, for example when edges 5, 8 and 9 are to be executed on rank X as part of the import execute block they need to reference cells 3, 7 and 8. Thus cells 3, 7 and 8 form the import non-exec halo on X and correspondingly the export non-exec halo on Y. The export non-exec elements are a subset of the core elements.

Within a single GPU, the size of the mini-partitions is very small, and so the proportion of redundant computation would be unacceptably large if we used the owner-compute approach. Instead we use the approach previously described in [13, 14], in which we adopt the "coloring" idea used in vector computing [47]. The mini-partitions are colored so that no two mini-partitions of the same color will update the same cell. This allows for parallel execution for each color using a separate CUDA kernel, with implicit synchronization between different colors. Key to the success of this approach is the fact that the number of mini-partitions is very large and so even if 10-20 colors are required there are still enough mini-partitions of each color to ensure good load-balancing.

There is also the potential for threads within a single thread block, working on a single mini-partition, to be in conflict when trying to update the same cell. One solution is to use atomic operations, but the necessary hardware support (especially for doubles) is not present on all the hardware platforms we are interested in. Instead, we again use the coloring approach, assigning colors to individual edges so that no two edges of the same color update the same cell, as illustrated in Figure 4. When incrementing the cells, the thread block first computes the increments for each edge, and then loops over the different edge colors applying the increments by color, with thread synchronization between each color. This results in a slight loss of performance during the incrementing process due to warp divergence on NVIDIA GPUs, but the cost is minimal.

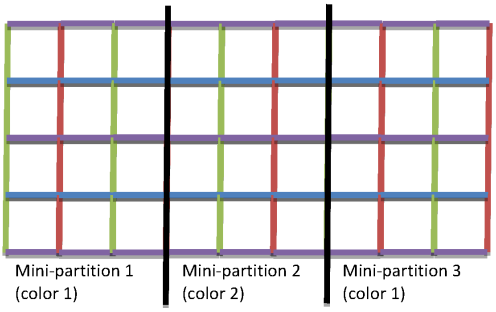


Figure 4: OP2 coloring of a mesh - each edge of the same color can be evaluated in parallel

A similar technique is used for multi-core processors. The only difference is that now each mini-partition is executed by a single OpenMP thread. The mini-partitions are colored to stop multiple mini-partitions attempting to update the same data in the main memory simultaneously. This technique is simpler than the GPU version as there is no need for global-local renumbering (for GPU global memory to shared memory transfer) and no need for low level thread coloring. Using the mini-partitioning strategy on CPUs (and by selecting the correct mini-partition size) good cache utilization can also be achieved.

4.3. Heterogeneous Clusters

Once the parallelization strategy for distributed memory and thread-level is established, the design for heterogeneous platforms involves two primary considerations; (1) combining the owner compute strategy across nodes and coloring strategy within a node and (2) implementing overlapping of computation with communication within the "plan" construction phase of OP2. For distributed memory clusters of GPUs, the OP2 design assumes that one MPI process will have access to only one GPU. Thus MPI will be used across nodes (where each node is interconnected by a communication network such as InfiniBand) and CUDA within each GPU. For clusters with each node consisting of multiple GPUs, OP2 assigns one MPI process per GPU. This simplifies the execution on heterogeneous cluster systems by allowing separate processes (and not threads) to manage any multiple GPUs on a single node. At runtime, on each node, each MPI process will select any available GPU device. Code generation with such a strategy reuses the single node code generation with only a few minor modifications as there is no extra level of thread management/partitioning within a node for multiple GPUs.

Recall that the MPI back-end achieves overlapping of computation with communication by separating the set-elements into two groups where the *core* elements can be computed over without accessing any halo data. To achieve the same objective on a cluster of GPUs, for each `op_par_loop` that does halo exchanges, OP2 assigns mini-partitions such that each will consist only of either *core* elements or non-*core* element (including import execute halo elements). This allows us to assign coloring to mini-partitions such that one set of colors are exclusively for

```
1: for all op_dats requiring a halo exchange do
2:   execute CUDA kernel to gather export halo data
3:   export halo: GPU → Host
4:   start non-blocking MPI communication
5: end for
6: for each color i = 0 to num_colors do
7:   if i == num_core_colors then
8:     wait for all MPI communications to complete
9:     for each op_dat requiring a halo exchange do
10:      import halo: Host → GPU
11:    end for
12:  end if
13:  execute CUDA kernel for
     mini-partitions with color i
14: end for
```

Figure 5: Multi-GPU non-blocking communication

mini-partitions containing only *core* element's while a different set will be assigned for the others. As such the pseudo-code for executing an `op_par_loop` on a single GPU within a GPU cluster is detailed in Figure 5.

The *core* elements will be computed while non-blocking communications are in-flight. The coloring of mini-partitions is ordered such that the mini-partitions with the non-*core* elements will be computed after all the *core* elements are computed. This allows for an MPI `wait_all` to be placed before non-`core` colors are reached. Each `op_plan` consists of a mini-partitioning and coloring strategy optimized for their respective loop and number of elements. The halos are transferred via MPI by first copying it to the host over the PCIe bus. As such the current implementation does not utilize NVIDIA's new GPUDirect [48] technology for transferring data directly between GPUs. An implementation of the MPI+CUDA back-end with GPUDirect is currently being tested within the OP2 project. Its performance will be reported in future work.

The multi-threaded CPU cluster implementation is based on MPI and OpenMP and follows a similar design to the GPU cluster design except that there is no data transfer to and from a discretely attached accelerator; all the data resides in CPU main memory. However, this may lead to contention for accessing main memory if the MPI process and thread allocation is not handled appropriately at runtime.

Currently, for simplicity the OP2 design does not utilize both the host (CPU) and the accelerator (GPU) simultaneously for the problem solution. However, OP2 does not limit the development of such a "fully-hybrid" implementation, and it is currently being investigated in the project. One possibility is to assign an MPI process that performs computations on the host CPU and another MPI process that "manages" the computations on the GPU attached to the host. The managing MPI process will utilize MPI and CUDA in exactly the same way described above, while the MPI process computing on the host will either use the single threaded implementation (MPI only) or multi-threaded (MPI and OpenMP) implementation.

The key issue in this case is in assigning and managing the load on the different processors depending on their relative speeds for solving a given mesh computation. The best load distribution across the GPUs and CPUs will also vary for each loop. If the load balancing is not carefully carried out then there may be significant performance degradations due to the faster GPU waiting for the CPU to compute its block of computations for each loop. This wait can compound for each call of the loops and eventually lead to a slower execution time than the current MPI+CUDA back-end. Our future work will report on the performance of this "fully-hybrid" implementation.

5. Applications

In this paper we benchmark the performance of two OP2 applications: (1) Airfoil, a standard finite volume CFD benchmark code and (2) Aero, a finite element 2D nonlinear steady potential flow simulation.

5.1. Airfoil

Airfoil [49] is a nonlinear 2D inviscid airfoil code that uses an unstructured grid. It is a finite volume application that solves the 2D Euler equations using a scalar numerical dissipation. The algorithm iterates towards the steady state solution, in each iteration using a control volume approach - for example the rate at which the mass changes within a control volume is equal to the net flux of mass into the control volume across the four faces around the cell. This is representative of the 3D viscous flow calculations OP2 aims to eventually support for production-grade CFD applications (such as the Hydra [50, 51] CFD code at Rolls-Royce plc.).

The application implements a predictor/corrector time-marching loop, by (1) looping over the cells of the mesh and computing the time step for each cell, (2) computing the flux over internal edges (3) computing the flux over boundary edges (4) updating the solution and (5) saving the old solution before repeating. These main stages of the application is solved in Airfoil within five parallel loops: `adt_calc`, `res_calc`, `bres_calc`, `update` and `save_soln`. Out of these, `save_soln` and `update` are direct loops while the other three are indirect loops.

To demonstrate how an application actually uses OP2's unstructured mesh API and declarations to perform parallel operations, we investigate the `res_calc` loop. `res_calc` computes the flux across two cells that share an edge. Given the direction of the flow, flux is added to one cell and is subtracted from the other. This operation is carried out for each internal edge (while `bres_calc` handles the boundary edge cases). The computation that needs to be carried out on each edge is defined in the user kernel as given below:

```
void res_calc(double *x1, double *x2, double *q1,
              double *q2, double *adt1, double *adt2,
              double *res1, double *res2) {
  double dx,dy,mu, ri, p1,vol1, p2,vol2, f;
```

```
  dx = x1[0]-x2[0];
  dy = x1[1]-x2[1];

  ri   = 1.0f/q1[0];
  p1   = gm1*(q1[3]-0.5f*ri*
              (q1[1]*q1[1]+q1[2]*q1[2]));
  vol1 =  ri*(q1[1]*dy-q1[2]*dx);

  ri   = 1.0f/q2[0];
  p2   = gm1*(q2[3]-0.5f*ri*
              (q2[1]*q2[1]+q2[2]*q2[2]));
  vol2 =  ri*(q2[1]*dy-q2[2]*dx);
  mu = 0.5f*((*adt1)+(*adt2))*eps;

  f = 0.5f*(vol1*q1[0]+
            vol2*q2[0])+mu*(q1[0]-q2[0]);
  res1[0] += f;
  res2[0] -= f;

  f = 0.5f*(vol1*q1[1]+p1*dy+
            vol2*q2[1]+p2*dy)+mu*(q1[1]-q2[1]);
  res1[1] += f;
  res2[1] -= f;

  f = 0.5f*(vol1*q1[2]-p1*dx+
            vol2*q2[2]-p2*dx)+mu*(q1[2]-q2[2]);
  res1[2] += f;
  res2[2] -= f;

  f = 0.5f*(vol1*(q1[3]+p1)+
            vol2*(q2[3]+p2))+mu*(q1[3]-q2[3]);
  res1[3] += f;
  res2[3] -= f;
}
```

Data pointed to by pointers `x1` and `x2` hold the xy coordinates of the nodes an either end of an edge, while `q1` and `q2` hold the solution in each of the cells sharing an edge. `adt1` and `adt2` hold the time step (that was previously calculated in `adt_calc`) for each of the cells. Finally `res1` and `res2` act as temporary stores to hold the flux movement in/out of the four faces of each cell. Additionally, `gm1, eps` are global constants. Looking through the code, we see that, `x1, x2, q1, q2, adt1` and `adt2` are all read-only variables. `res1` and `res2` are the only variables that are written to, in this case they are incremented by the computed flux.

Using these data access modes, we can declare the `res_calc` computation over all `edges` with the OP2 API, to form the `op_par_loop` declaration below:

```
op_par_loop(res_calc,"res_calc",edges,
   op_arg_dat(p_x,   0,pedge, 2,"double",OP_READ),
   op_arg_dat(p_x,   1,pedge, 2,"double",OP_READ),
   op_arg_dat(p_q,   0,pecell,4,"double",OP_READ),
   op_arg_dat(p_q,   1,pecell,4,"double",OP_READ),
   op_arg_dat(p_adt, 0,pecell,1,"double",OP_READ),
   op_arg_dat(p_adt, 1,pecell,1,"double",OP_READ),
   op_arg_dat(p_res, 0,pecell,4,"double",OP_INC ),
   op_arg_dat(p_res, 1,pecell,4,"double",OP_INC ));
```

The first three arguments in the loop, as described previously, indicate the user kernel name, the loop name and the set over which the loop is performed. Each of the next arguments declare the `op_dat` which holds the data, the mapping from the edges that needs to be used to indirectly reference the data and the way in which data is accessed (read only or increment in this case). The mapping and index resolve the address of the actual parameter. Now, given the above OP2 declaration of the `res_calc` computation, OP2 has all the information required to generate parallel code, that implements the solution targeting different back-ends. For example, for execution on GPUs, the code generator produces a host stub function as outlined in Figure 5 and a CUDA kernel function that in turn calls the user kernel above for its elemental computation (see `res_calc_kernel.cu` in [52]).

5.2. Aero

Aero is a 2D non-linear steady potential flow simulation of air moving around an airfoil developed based on standard finite element methods [53]. It uses a quadrilateral grid similar to that used by the Airfoil application but uses a Newton iteration to solve the non-linear equations defined by a finite element approximation. Each Newton iteration requires the solution of a linear system of equations. The assembly algorithm is based on quadrilateral elements and uses transformations from the reference square to calculate the derivatives of the first-order basis functions. Dirichlet-type boundary conditions are applied on the far-field, and the symmetric sparse linear system is solved with the standard conjugate-gradient (CG) algorithm. The flow variables are then updated, and the algorithm proceed to the next Newton iteration. The Aero code consists of several parallel loops: `res_calc` is an indirect kernel that implements the finite element assembly using a matrix-free method where the local stiffness matrices for each element are stored separately [54]. Kernels `dirichlet` and `init_cg` are direct loops over state variables and temporary arrays that serve to prepare for the conjugate-gradient solution. The CG iteration consists of several kernels, the most important of which is the sparse matrix-vector multiplication. The indirect kernel `spMV` takes the stiffness data stored in the matrix-free format and performs small, dense matrix-vector multiply operations, then gathers the values to the result vector. Boundary conditions are enforced and other vector-vector operations are carried out by direct loops inside the CG iteration. After at most 200 CG iterations or a hundred-fold decrease in the residual the state variables are updated and the *rms* residual after the current Newton iteration is calculated.

With the `init_cg` loop in Aero, we can further demonstrate how an application actually use OP2's unstructured mesh framework to perform parallel operations. In this case `init_cg` is a direct loop with a global reduction operation. Its aim is to simply initialize a number of data values held on each node of the mesh, while at the same

Table 1: Cluster systems specifications

System	HECToR (Cray XE6)	EMERALD (NVIDIA GPU Cluster)	IRIDIS (Intel Westmere CPU cluster)
Node Architecture	2×16-core AMD Opteron 6276 (Interlagos) 2.3GHz	3×Tesla M2090 + 2×Intel Xeon X5650 2.67GHz	2×6-core Intel Xeon E5645 (Westmere) 2.4GHz
Memory/Node	32GB	6GB/GPU (ECC on)	22GB
Interconnect	Cray Gemini	QDR InfiniBand	QDR InfiniBand
O/S	CLE 4.0	Linux 2.6.32 with CUDA 5.0	RedHat Linux Enterprise 5.3
Compilers	Cray CC (8.1.4) Cray MPI	ICC 12.1.0 Platform MPI 08.01.01	ICC 12.0.4 OpenMPI 1.4.5
Compiler flags	-O3 -h fp3 -h ipa5	-O2 -xSSE4.1 -arch=sm_20 -use_fast_math	-O2 -xSSE4.2

time computing a global sum of the residual, `resm`.

```
void init_cg(double *r, double *c,
             double *u, double *v, double *p){
    *c += (*r)*(*r);
    *p = *r; *u = 0; *v = 0;
}
```

Variables pointed to by `p, u` and `v` are initialized, i.e. written to, while `c` is read and written (in this case incremented). We can declare the `init_cg` computation over all `nodes` with the OP2 API, to form the `op_par_loop` declaration below:

```
op_par_loop(init_cg, "init_cg", nodes,
   op_arg_dat(p_resm,-1,OP_ID, 1,"double",OP_READ),
   op_arg_gbl(&c1,    1, "double", OP_INC),
   op_arg_dat(p_U,   -1,OP_ID, 1,"double",OP_WRITE),
   op_arg_dat(p_V,   -1,OP_ID, 1,"double",OP_WRITE),
   op_arg_dat(p_P,   -1,OP_ID, 1,"double",OP_WRITE));
```

As all the variables reside in data held on the nodes, no indirect access through mapping tables are required. `op_arg_gbl` specifies that `c1` is a global variable to be incremented. This translates to a parallel reduction operation. For example, for execution on GPUs, OP2's code generator will generate a CUDA reduction (see `init_cg_kernel.cu` in [52]). Similarly for execution under MPI, the reduction will trigger an `MPI_Allreduce` to compute the global sum across MPI ranks. As the resulting value from global reductions are commonly used in the control flow of the application (e.g. checking for convergence and terminating the iterative method), OP2 uses an `MPI_Allreduce` (as opposed to an MPI_Reduce) to enable each MPI process to end up with the result of the reduction enabling them to take the same control decision.

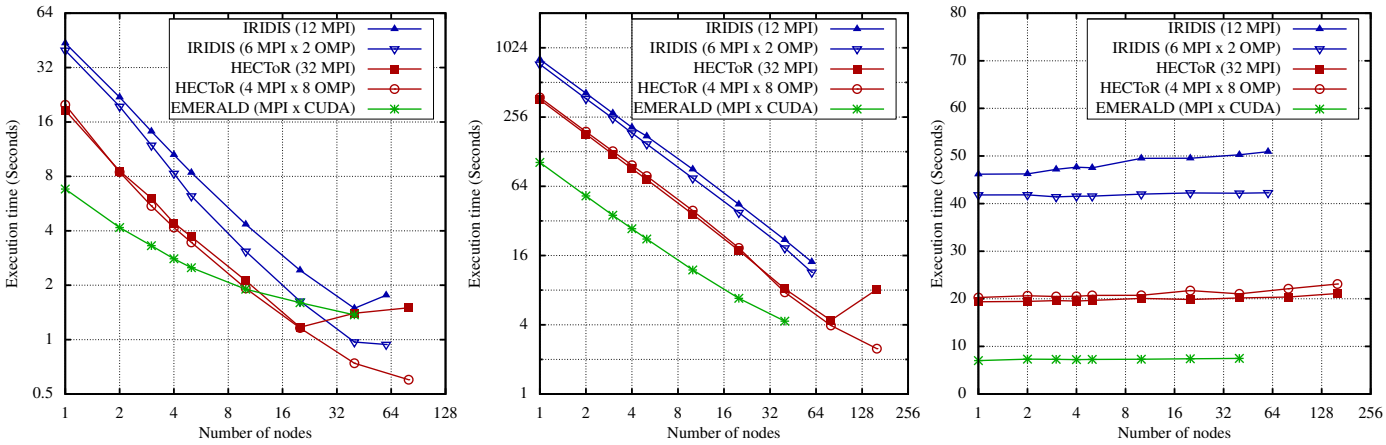
6. Performance

In this section, we present quantitative results exploring the performance portability and scaling of OP2's heterogeneous back-end design. Airfoil and Aero are executed on three distributed memory heterogeneous systems (Table 1) and their runtime performance, scaling as well as energy performance are benchmarked and analyzed. All results presented are from execution in double-precision floating-point arithmetic. For Airfoil, we use two mesh sizes (1.5M and 26M edges) and measure the end-to-end runtime of the main time marching loop. When solving the 1.5M edge mesh, the most compute intensive loop, `res_calc`, is called 2000 times during the total execution of the application and performs about 100 floating-point operations per mesh edge. For Aero we present performance results for the solution of a mesh with 720K and 12M cells. Additionally, for both applications we generated a sequence of meshes for the weak-scaling runs, such that an approximately fixed number of elements are allocated per machine node at increasing scale.

6.1. Runtime and Scaling Performance

We begin by exploring the runtime and scaling performance of Airfoil and Aero on three distributed memory systems. Table 1 notes the main details of these systems. The first system, HECToR [55], is a large-scale proprietary Cray XE6 system which we use to investigate the scalability of the MPI implementation as well as the MPI+OpenMP runtimes. The second system, EMERALD [56] is a large NVIDIA GPU (Tesla M2090) cluster that we use to benchmark OP2's latest MPI+CUDA backend. The final system, IRIDIS [57] is an Intel Westmere processor based cluster, which we use to present comparative performance for Intel processor based systems – the most common processor architecture used in modern HPC systems.

Figure 6 reports the run-times of Airfoil at scale, solving a mesh with 1.5M and 26M edges in a strong-scale setting and a 1.5M edges per node mesh in a weak-scaling setting respectively. The x-axis represents the number of nodes on each system tested, where a HECToR node consists of two Interlagos processors, an IRIDIS node consists of two Westmere processors and an EMERALD node consists of three Tesla M2090 GPUs. The run-times (on the y-axis) are averaged from 5 runs for each node count. The standard deviation in run times was significantly less than 10%. Airfoil scales well on the CPU clusters and on HECToR in particular we benchmarked its performance on up to 5120 cores (160 nodes). MPI+OpenMP results were obtained by assigning four MPI processes per HECToR node, each



(a) 1.5M Edges (strong-scaling) (b) 26M Edges (strong-scaling) (c) 1.5M Edges/node (weak-scaling)

Figure 6: Runtime of Airfoil (1000 iterations) on HECToR, EMERALD and IRIDIS

Table 2: Airfoil - halo sizes at scale (HECToR) : Av = average number of MPI neighbors per process, Tot. = average number total elements per process, %H = average % of halo elements per process

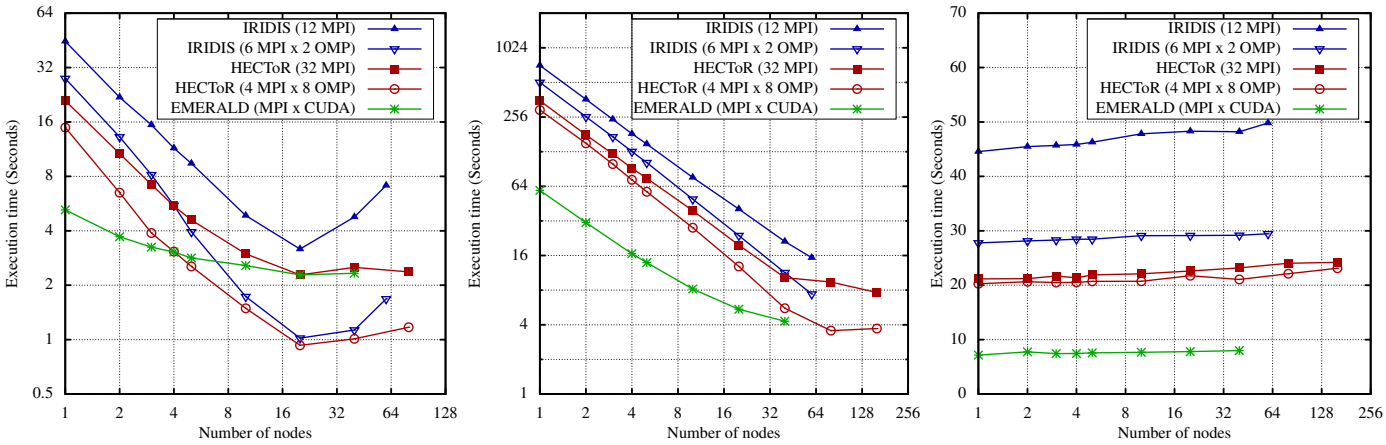
nodes	MPI	1.5M			26M		
	procs.	Av	Tot	%H	Av	Tot	%H
1	32	4	68K	1.55	4	1M	0.36
2	64	5	34K	2.41	5	636K	0.56
5	160	5	14K	3.77	5	255K	0.90
10	320	5	7124	5.32	5	128K	1.30
20	640	5	3640	7.36	5	64K	1.86
40	1280	8	2773	39.24	5	32K	2.61
80	2560	8	1509	44.20	5	16K	3.65
160	5120				9	11K	32.61

MPI process consisting of eight OpenMP threads. This is due to the NUMA architecture of the Interlagos processors [58] 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 "module" reducing inter-module communications. Other combinations of processes and threads were also explored, but the above provided the best performance. We also observed up to 30% performance gains due to the use of non-blocking communications overlapped with computation during halo exchanges. On IRIDIS, we used TAPS [59] to bind OpenMP threads to cores. Our experience was that the runtime was significantly worse (and variable) on this system if such a thread binding method was not utilized.

On HECToR, the application parallelized only using MPI performs as well as the version that uses the hybrid MPI+OpenMP parallelization on up to 20 nodes for the 1.5M mesh and up to 80 nodes for the 26M mesh when strong-scaling. However, further increases in scale only improves performance under the MPI+OpenMP hybrid version, where the pure MPI version collapses due to over partitioning the mesh, leading to an increase in redundant computation at the halo regions (compared to the `core` elements per partition) and an increase in communication time spent during halo exchanges. Evidence for this explanation can be gained by observing the average number of halo elements and the average number of neighbors per MPI process reported by OP2 after partitioning with ParMETIS or PT-Scotch (see Table 2). These results were taken 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 one partition) as the partitioner gives the same quality partitions given the same mesh for the same number of MPI processes on any cluster. In these runs we have used PT-Scotch for partitioning as it provided marginally better partitions than ParMETIS for our benchmark applications.

For Airfoil, ignoring the small boundary set `bedges`, the only sets that have either a halo exchange and/or computation over the halo block are cells and edges. As such Table 2 details the average total number of cells and edges per MPI process (Tot) in column 4 and 7 for the two strong-scaled meshes. Columns 5 and 8 (%H) indicate the average proportion of halo cells and edges out of the total number of elements per MPI process. Columns 3 and 6 (Av) indicate the average number of communication neighbors per MPI process. For the strong-scaled 1.5M edges mesh, the proportion of the halo elements out of the average total number of elements held per MPI process ranged from 1.5% (at 32 MPI processes) to about 44% (at 2560 MPI processes). For the 26M edge mesh the proportion of the halo out of the average total number of elements held per MPI process ranged from 0.36% (at 32 MPI processes) to about 3% (at 2560 MPI processes). The average number of MPI neighbors per MPI process ranged from 4 to 9 for both problem sizes.

The Intel Westmere cluster, IRIDIS, performs roughly two times slower than HECToR but displays similar scaling behavior. Comparing the performance on HECToR to that on the GPU cluster EMERALD, reveals that for the 1.5M edge mesh the CPU system gives better scalability than the GPU cluster. We believe that this is due to the smaller amount of work assigned to each GPU at increasing scale. However, the 26M edge mesh scales well to over 40 GPU nodes, where the runtime is almost halved each time we double the number of nodes. At 40 nodes (120



(a) 720K Cells (strong-scaling) (b) 12M Cells (strong-scaling) (c) 720K Cells/node (weak-scaling)

Figure 7: Runtime of Aero (20 Newton iterations) on HECToR, EMERALD and IRIDIS

Table 3: Aero - halo sizes at scale (HECToR) : Av = average number of MPI neighbors per process, Tot. = average number total elements per process, %H = average % of halo elements per process

nodes	MPI	720K			12M		
	procs.	Av	Tot	%H	Av	Tot	%H
1	32	7	45K	2.08	7	746K	0.53
2	64	7	23K	2.92	8	374K	0.81
5	160	9	9466	4.84	10	150K	1.34
10	320	10	4855	7.23	11	75K	1.91
20	640	10	2696	16.47	11	38K	2.70
40	1280	12	1809	37.81	12	19K	4.08
80	2560	13	1046	46.27	12	11K	19.50
160	5120				14	7559	38.60

GPUs) EMERALD gives approximately the same performance as HECToR with 80 nodes (2560 cores). We explore the performance of each loop later in section 6.3 with the aim to isolate performance bottlenecks that are constraining total performance and investigate for example which phases of the computations constrain the GPU results to not show speedups for the smaller problem size.

The weak-scaling results in Figure 6(c) demonstrate less than 2% increase in runtime each time the number of nodes is doubled, pointing to excellent scalability of the OP2 distributed memory heterogeneous back-ends. In this case, we maintained a constant mesh size (1.5M edges) per node at increasing machine scale. Thus the largest mesh benchmarked across 160 nodes (5120 cores) on HECToR consisted of about 0.2 Billion edges in total. Further scaling could not be attempted due to the total size of the systems and the availability of benchmarking time on them. The proportion of the number of halo elements out of the total number of elements held per MPI process, remains approximately constant at around 1.5–1.8% for all MPI only runs, 0.3–0.6% for all MPI+OpenMP runs and 0.27–0.5% for all MPI+CUDA runs. The average number of MPI neighbors per MPI process ranged from 2 to 5.

Similarly, Figure 7(a) and (b) reports the strong-scaling run-times of Aero, for a mesh with 720K and 12M cells respectively while Figure 7(c) reports on the weak-scaling performance with a mesh of 720K cells per node. Considering the strong-scaling results, both CPU clusters scale better than the GPU cluster for the 720K mesh. But on the CPU clusters the difference between the MPI only executions and MPI+OpenMP is much larger than it was for Airfoil. The MPI+OpenMP combination on HECToR gives the best performance at about 20 nodes. Again we see similar divergence of performance between the MPI only and MPI+OpenMP hybrid executables at large machine scale. In contrast the GPU cluster, EMERALD, provides only minimal performance gains at increasing node counts. However, when the problem size is increased to 12M cells, EMERALD gives good scalability. Again, we speculate that the poor scaling on the small mesh is due to: (1) not having enough parallelism to be exploited within a partition assigned to each GPU and (2) each partition not providing sufficient computation to hide kernel launch and communication latencies (PCIe and InfiniBand latency).

The weak-scale runs in Figure 7(c) again show excellent weak-scaling. In this case, we maintained a constant mesh size (720K cells) per node at increasing machine scale, with the largest mesh size benchmarked across 160 nodes (5120 cores) on HECToR consisting of about 115 Million cells in total. There is less than 5% increase in runtime for each doubling of the problem. Table 3 details the halos and MPI neighbor details for Aero. In this case the halo consisted of only cells and nodes and the total elements presented here is the average number of total cells and nodes per MPI process. Using PT-Scotch for partitioning for the strong-scaled 720K cells mesh, the proportion of the halo elements out of the average total number of elements held per MPI process ranged from about 2% (at 32 MPI processes) to about 46% (at 2560 MPI processes). For the 12M cells mesh these proportions ranged from 0.5% (at 32 MPI processes) to about 19% (at 2560 MPI processes). The average number of MPI neighbors per MPI process ranged from 7 to 14 for both problem sizes. Similar to Airfoil the weak-scale runs for Aero in Figure 7(c) demonstrates almost constant runtime at increasing scale pointing again to excellent scalability. The mesh size per node is 720K cells. The proportion of the number of halo elements

Table 4: Airfoil single node results for the `res_calc` loop: 1.5M edges

Node system	MPI. ×OMP	Time (sec)	GFlops /sec	GB/sec (useful)	GB/sec (cache)
2×Westmere	1×12	28.25	10.61	15.51	15.72
	6×2	19.25	15.58	22.18	22.69
2×Interlagos	1×32	44.66	6.71	9.82	10.09
	4×8	10.58	28.35	41.36	42.24
1 × M2090	CUDA	10.96	27.37	40.19	42.32

Table 5: Aero single node results for the `spMV` loop: 720K cells

Node system	MPI. ×OMP	Time (sec)	GFlops /sec	GB/sec (useful)	GB/sec (cache)
2×Westmere	1×12	38.50	n/a	15.57	15.67
	6×2	20.87		27.57	28.00
2×Interlagos	1×32	62.90	n/a	9.53	9.66
	4×8	11.18		53.87	54.96
1 × M2090	CUDA	5.86	n/a	102.72	108.22

out of the total number of elements held per MPI process, ranges from 2–3% for all MPI only runs, 0.5–1% for all MPI+OpenMP runs and 0.4–0.7% for all MPI+CUDA runs. The average number of MPI neighbors per MPI process ranged from 7 to 12.

The above scaling results for both Airfoil and Aero, give us considerable confidence in OP2's ability to give good performance at large machine sizes. We see that the primary factor affecting performance is the quality of the partitions: minimizing halo sizes and MPI communication neighbors. The above results illustrate that, in conjunction to utilizing state-of-the-art partitioners such as PT-Scotch, the halo sizes resulting from OP2's owner compute design for distributed memory parallelization provide excellent scalability for these two applications. Additionally overlapping computations with communications are also exploited within OP2 for higher performance. Our results further point to the extra scalability and higher performance that can be gained with hybrid implementations such as MPI with OpenMP. Particularly when weak-scaling, MPI+OpenMP gives considerably better performance than the MPI only parallelization. We see that GPU clusters are much less scalable for small problem sizes and are best utilized in weak-scaling executions. Specific results from Aero point to it being less scalable than Airfoil. We explore the reasons for this in section 6.3.

The MPI+OpenMP is an important heterogeneous back-end for modern multi-core systems, as the ever increasing number of cores makes it the only feasible solution to gain good performance. However, this is only valid if we are able to develop a good implementation of OpenMP on a single node. In the next section we investigate whether OP2 has achieved such an implementation.

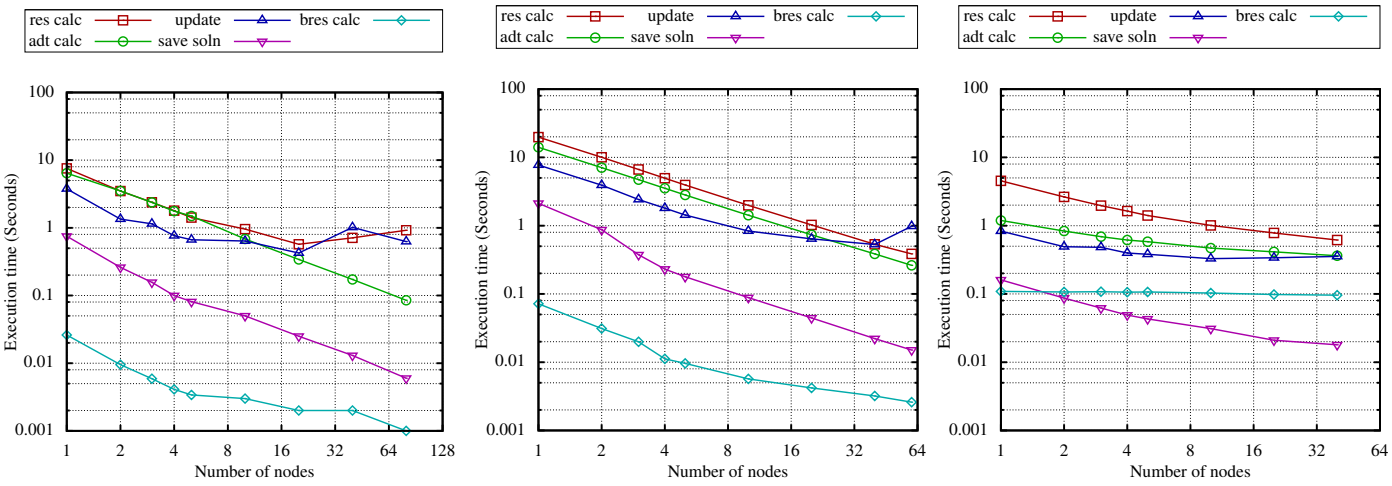
6.2. Single Node Performance

Evidence for whether OP2 is producing back-end specific code with near optimal performance can be ascertained by investigating the achieved floating-point operation rates and bandwidths on a single GPU or CPU node in comparison to the advertised peak rates on these GPUs or CPUs. In our previous work [13, 14] we have presented similar quantitative measures on a number of previous generation single node systems. In Table 4 and 5 we present these figures for both Airfoil and Aero on each of the CPU and GPUs of the cluster systems in Table 1. The table presents the run-times gained on each single cluster node executing the 1.5M edge mesh for Airfoil and 720K cell mesh for Aero.

Table 4 details the performance of the `res_calc` loop, the most compute intensive loop and also the most time consuming loop in Airfoil. On the 1.5M edge mesh, it performs about 30×10^{10} floating-point operations during the total runtime (i.e. 1000 iterations, `res_calc` called 2 times per iteration) of the application. The final two columns present the achieved double precision floating-point rate and effective bandwidth between main-memory or global-memory on the CPUs and GPUs respectively. The bandwidth figure was computed by counting the total amount of useful data bytes transferred from/to global memory during the execution of a parallel loop and dividing it by the runtime of the loop. The bandwidth is higher if we account for the size of the whole cache line loaded from main-memory/global-memory (see column 6).

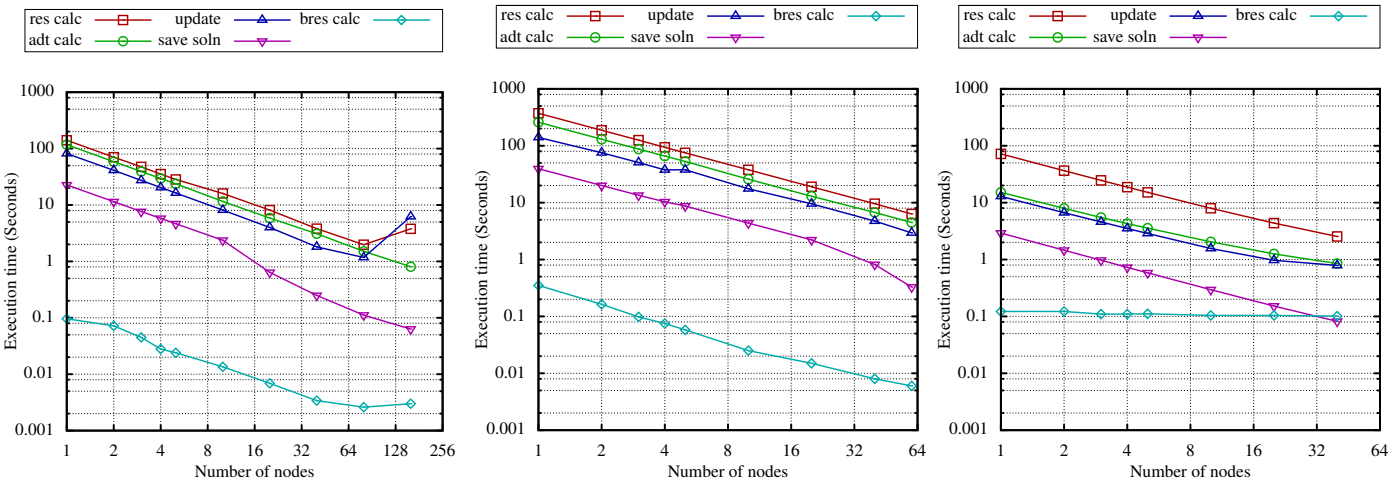
We see that, for `res_calc`, only a fraction of the advertised peak floating-point rates are achieved by any CPU or GPU. For example, only about 15 GFlops/sec out of a peak of about 115 GFlops/s (2.4 GHz × 4 Flops per core × 12 cores) is achieved on the Westmere node. On the Interlagos node significantly better performance was achieved when using the MPI+OpenMP hybrid combination as opposed to using only OpenMP. This, as discussed before, is due to the NUMA architecture of the Interlagos processors that organizes 8 cores within a processor with a dedicated path to part of the DRAM. Again with the MPI+OpenMP back-end we were able to leverage the job scheduler to exactly place one MPI process per "module" reducing inter-module communications. In this case about 28 GFlops/s out of a peak of 294 GFlops/s [58] was achieved on the Interlagos node. Similarly due to the Westmere node consisting of two sockets, the MPI+OpenMP back-end gave considerably better performance, particularly when thread allocation and affinity was managed with TAPS [59]. On the M2090 GPU the achieved floating-point operation rate is about 27 GFlops/s out of an advertised peak of about 665 GFlops/s [60].

On the other hand, the achieved bandwidth on the Westmere node and Interlagos node is closer to half of their peak (2×35 GB/s [61] and 85 GB/s [62] respectively) which indicates that on CPUs the problem is much more constrained by bandwidth than floating-point performance. Our experiments also showed that for direct loops such as `save_sol`, the memory bandwidth utilized on the GPUs gets closer to 70% of the peak bandwidth (177GB/s [60] with ECC off) due to its low computational intensity. The trends on achieved computational intensity and bandwidth utilization remain very similar to the observed results from



(a) HECToR (MPI) (b) IRIDIS (MPI) (c) EMERALD (MPI+CUDA)

Figure 8: Runtime breakdown of Airfoil 1.5M edges (strong-scaling) problem (1000 iterations) on HECToR, EMERALD and IRIDIS



(a) HECToR (MPI) (b) IRIDIS (MPI) (c) EMERALD (MPI+CUDA)

Figure 9: Runtime breakdown of Airfoil 26M edges (strong-scaling) problem (1000 iterations) on HECToR, EMERALD and IRIDIS

previous work [14] with about 25%-30% improvement over the previous CPUs (Intel Nehalem) and GPUs (C2050).

Table 5 presents the performance of spMV, the most time consuming loop in Aero, which in contrast to the `res_calc` loop in Airfoil, performs only about 32 floating-point operations per cell. However, it performs about 20 double-precision floating-point data read/writes from/to main memory (or GPU global memory) per cell. This is about 460 GB of data transfers between processor cores and memory in total for the 720K mesh. Thus the loop can be characterized as a bandwidth intensive loop. The achieved bandwidths on each node confirms this observation, with over 60% of the peak bandwidths utilized on the Interlagos CPU node and the M2090 GPU.

The OP2 auto-generated code contains all of the best hand-tuning optimizations to our knowledge and we are not aware of any ways of obtaining additional performance at the time of writing. The performance data on bandwidths in Tables 4 and 5 show that there is only little scope for additional speedup. One avenue we are currently investigating for obtaining further performance is to implement sparse/overlapped tiling [63, 64]. The idea is to employ communications avoiding algorithms such that we reduce the bottleneck due to memory bandwidth. If in the future we discover further optimisations to any of the OP2 back-ends, as well as different back-end implementations that are necessary to obtain the best performance for newer CPU/GPU architectures, applying such modifications to all codes that make use of OP2 could be easily achieved by updating the code generator to produce the required optimized code. In fact each of our previous works [13, 14, 15, 16, 17, 18] has done exactly this when various new optimisations were realized at each stage of the OP2 project. This we believe is one of the key strengths of a higher level framework, such as OP2 that significantly increases the longevity of the application code base. Additionally, as detailed in [16], we prefer to implement such optimisations through the code generator and back-end libraries by parameterizing or modularizing the new feature such that the best parameters (and in turn modules

Table 6: Airfoil kernel achieved bandwidth (GB/s) on EMERALD at increasing (strong-scale) : 1.5M edges

	Achieved bandwidth (GB/s)				
Nodes	save_soln	adt_calc	res_calc	bres_calc	update
1	96.45	59.50	36.95	5.97	85.06
2	89.76	40.06	32.81	3.24	66.44
3	81.09	35.91	30.18	2.18	45.61
4	76.68	31.91	27.29	2.12	41.82
5	69.66	29.12	25.78	1.85	34.75
10	48.79	20.21	18.54	1.03	19.72
20	35.84	12.73	12.39	0.86	9.33
40	21.17	7.85	7.04	0.66	4.33

Table 7: Airfoil kernel achieved bandwidth (GB/s) on EMERALD at increasing (strong-scale) : 26M edges

	Achieved bandwidth (GB/s)				
Nodes	save_soln	adt_calc	res_calc	bres_calc	update
1	99.41	56.40	43.67	22.17	93.21
2	99.33	55.35	43.07	13.30	92.03
3	99.25	55.00	42.67	8.12	91.42
4	99.14	54.18	42.25	5.83	88.06
5	99.05	53.21	41.80	5.56	83.70
10	97.50	58.19	40.49	3.06	54.16
20	94.98	46.27	37.10	2.45	65.61
40	87.76	39.58	32.50	1.61	43.12

of code) can be selected either at compile time or runtime with the use of methods such as auto-tuning. Examples include the CUDA block and partition size optimisation and the choice between SoA and AoS data layout on different architectures [16].

6.3. Performance Breakdown at Scale

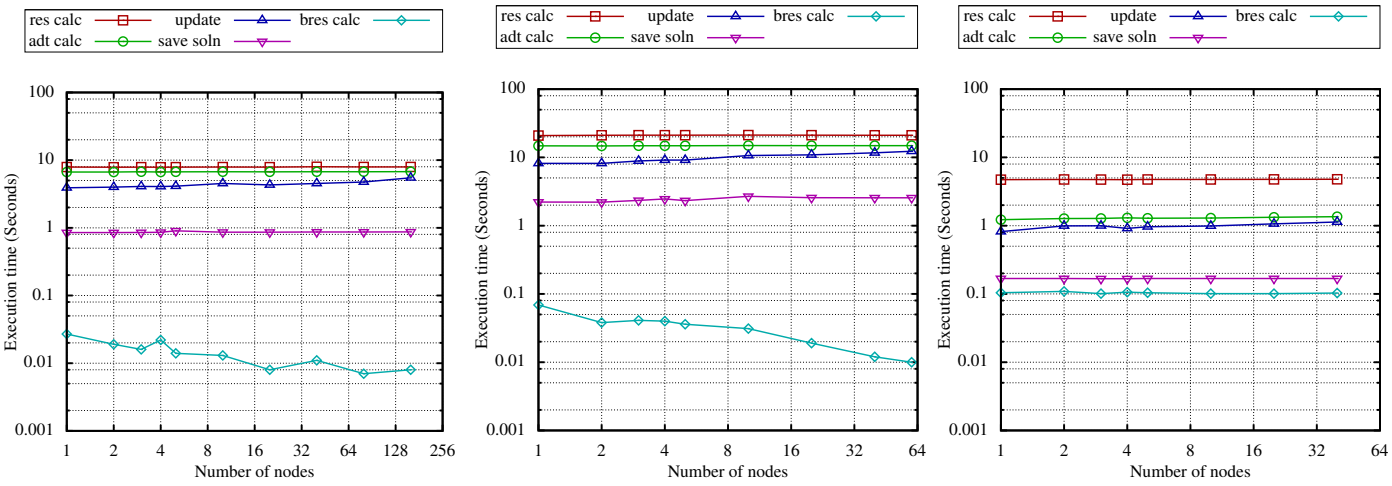
In this section we delve further into the performance of the two applications in order to identify performance limiting factors. The aim is to break-down the scaling performance to gain insights into how each loop in an application scale on each of the cluster systems. Figures 8 and 9 present the average times taken by each `op_par_loop` in Airfoil when strong-scaling with 1.5M and 26M edge meshes respectively. For clarity, the figures only include break-down time for the MPI only back-end but the MPI+OpenMP back-ends showed similar behavior, albeit with better scaling. Each loop time includes any MPI (and PCIe) communication times. The loops limiting scalability on the CPU clusters appear to be `res_calc` and `update`. Recall that the `res_calc` loop performs the only point-to-point MPI communications. Additionally it also performs a block of redundant computations over the halo edges. Thus it is this loop that mainly gets affected by over partitioning at large number of MPI processes. As `res_calc` is also the most compute intensive loop, the effect on overall performance is much more significant. `bres_calc` also perform redundant computation over halos but have less of an impact. `adt_calc` has several calls to the `sqrt()` function which may be slowing its performance on the HECToR nodes compared to IRIDIS nodes. The performance effect of these calls are much more visible on the Interlagos processors due to two Interlagos cores sharing one floating-point unit. The `update` loop has an `MPI_Allreduce` operation to calculate the rms value of the solution vector for each iteration. This operation, as expected, becomes significant at large machine scale.

In contrast, `bres_calc` does not scale at all for both problem sizes on the GPU cluster. The reason for this behavior is the amount of work that gets assigned to each GPU at increasing scale. `bres_calc` in particular appears to be limited by the CUDA kernel launch latency as it loops over the much smaller set of boundary edges (`bedges`). Here the launch latency dominates the time spent in the loop. Tables 6 and 7 report the achieved bandwidths (including cache line loading) within each CUDA kernel, on average for a GPU, for each `op_par_loop`. Note here that a single EMERALD node consists of 3 GPUs. As GPUs are throughput devices, these results show that the best scaling is only achieved when the GPUs have enough work to complete. For example it can be seen that for the large problem size the achieved bandwidth for all the loops except `bres_calc` remain high (over 30GB/s). This reaffirms our insights into `bres_calc`'s performance.

Figure 10 presents the runtime breakdown of each parallel loop for the weak-scaling executions. All loops except `bres_calc` behave as expected: `save_soln, adt_calc` and `res_calc`, show near constant execution time at increasing scale while the runtime taken by `update` steadily increase due to the MPI collective operation. However, the runtime taken by `bres_calc` in fact reduces on the CPU clusters as we execute on more and more processes. We speculate that even though the mesh increases with the number of processes, the number of boundary edges `bedges` per MPI process over which `bres_calc` operates over, gets reduced as they only appear at the boundary of the mesh. On the other hand on the GPU cluster, the loops over `bedges` are limited, as before, by the CUDA kernel launch latency.

The runtime breakdowns for Aero are given in Figures 11, 12 and 13. Aero has ten parallel loops, but as `dirichlet` is called twice we accumulate its runtime as one loop. `init_cg, dotPV and dotR` are direct loops that perform straightforward vector dot products and also consists of global reductions that trigger `MPI_Allreduce` operations. Thus on both HECToR and IRIDIS they scale poorly, particularly for the small problem size. Scaling on the GPU cluster for these loops is even worse as the GPU cannot achieve a recognizable speedup for loops with only a small amount of computations. `update` also does a global reduction resulting in poor scaling. The point-to-point halo exchanges occur only in `res_calc` and `spMV`. `dirichlet` loops over boundary nodes (`bnodes`) resulting in very small number of elements per partition. When weak-scaling, `dirichlet` runtime reduces on the CPU clusters similar to `bres_calc` in Airfoil. But on the GPU cluster it gets limited by CUDA kernel launch latency and remains approximately constant.

These profiles reconfirm the insights gained from Airfoil. Aero has a much larger number of loops that do only a few



(a) HECToR (MPI) (b) IRIDIS (MPI) (c) EMERALD (MPI+CUDA)

Figure 10: Runtime breakdown of Airfoil 1.5M edges per node (weak-scaling) problem (1000 iterations) on HECToR, EMERALD and IRIDIS

Table 8: Standby power of nodes

Node system	Standby Power ($\overline{P}_q$)
2× Intel Westmere (2.67 GHz) + 2× NVIDIA C2070	320 W
2× AMD Interlagos	280 W
2× Intel Westmere (2.4 GHz)	116 W

computations compared to Airfoil. This is the main reason for Aero's relatively poor scaling compared to Airfoil. Additionally having several loops with global reductions also limit scalability.

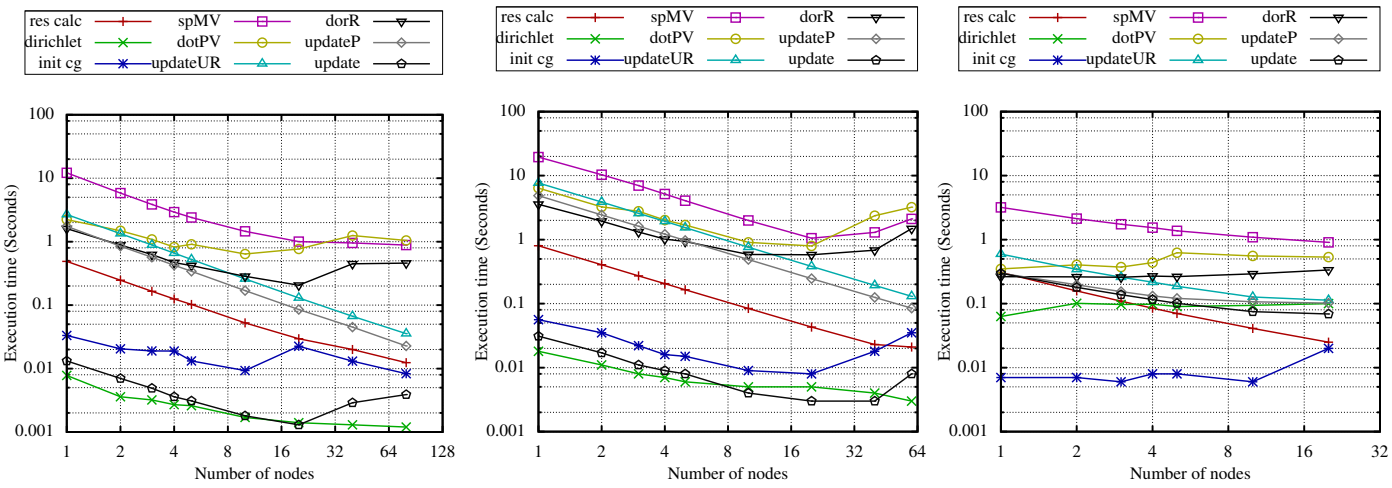
6.4. Energy Performance

In Figures 6 and 7 we have presented the performance of each system comparing their runtime performance on a node to node basis. By plotting the runtime against the number of nodes we have assumed that, for example, the performance on one HECToR node (2×Interlagos processors) is reasonable to be compared against the performance of one EMERALD node (3×C2090 GPUs). However, on modern multi-core/many-core systems, such a comparison may not provide a complete picture of achieved performance. As the number of transistors (and in turn the number of cores) on a single silicon chip is no longer the limiting factor of micro-processor design, the above performance curves could well be "slid" horizontally by an arbitrary increase/decrease in the number of cores per CPU/GPU node. Thus we believe that the Airfoil and Aero application performance could be re-evaluated with an alternative metric to gain further insights into the comparative performance of OP2's heterogeneous back-ends. The energy consumption of the systems may well provide the required normalizing factor, allowing us to observe the achieved performance of a system within a given energy envelope. In this section we attempt to explore OP2's performance through this alternative metric.

Similar to the presentation of single node achieved floating-point rate and bandwidth profiles in Tables 4 and 5, our objective is to illustrate the energy profiles of a single node on each of HECToR, EMERALD and IRIDIS. In order to measure the energy consumption of our applications, we used a recently developed power/energy analysis framework, EMPPACK [65]. The EMPPACK framework relies on sensors placed in-between the power source and the system(s) under test for extracting energy consumption. These sensors are capable of capturing the power line conditions, such as voltage and current, at regular intervals and can be triggered as needed. The framework enables applications to be instrumented with power profiling statements (similar to time measuring statements, e.g. `gettimeofday()`) to profile parts of code that are of interest. In our case, we profiled the main time marching loops of the Airfoil and Aero applications. When profiling, the framework captures the instantaneous power readings throughout the execution of an application. The energy consumption of the concerned block of code is then estimated by integrating the instantaneous power readings over the period of execution using stand-by power as the baseline. Thus, the overall energy consumption E on a node with Q power feeds is:

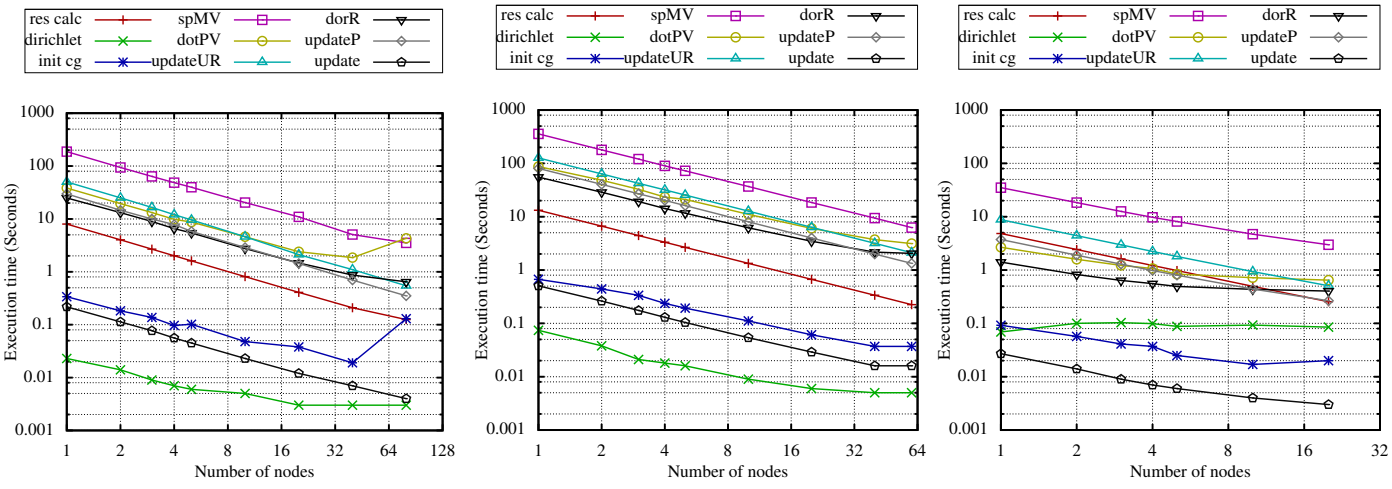
$$E = \sum_{q=1}^{q=Q} \Big(\int_T p_q(t)dt - \overline{P}_q.T \Big) \quad (1)$$

where $p_q(t)$ represents the instantaneous power values of power feed q, T is the execution time and $\overline{P}_q$ is the standby power of the feed q. To establish an acceptable statistical significance, we repeated the profiling a number of times before settling on their mean value. Since the usage of EMPPACK demands physical access to compute nodes (for instrumenting with the power sensors), we were not able to use nodes in the above clusters, but had to source separate nodes with near-equivalent specifications. The runtime of both applications on these nodes are approximately equivalent to the runtime on a single node on the respective clusters. This gives us considerable confidence in the accuracy of our energy performance profiles as well



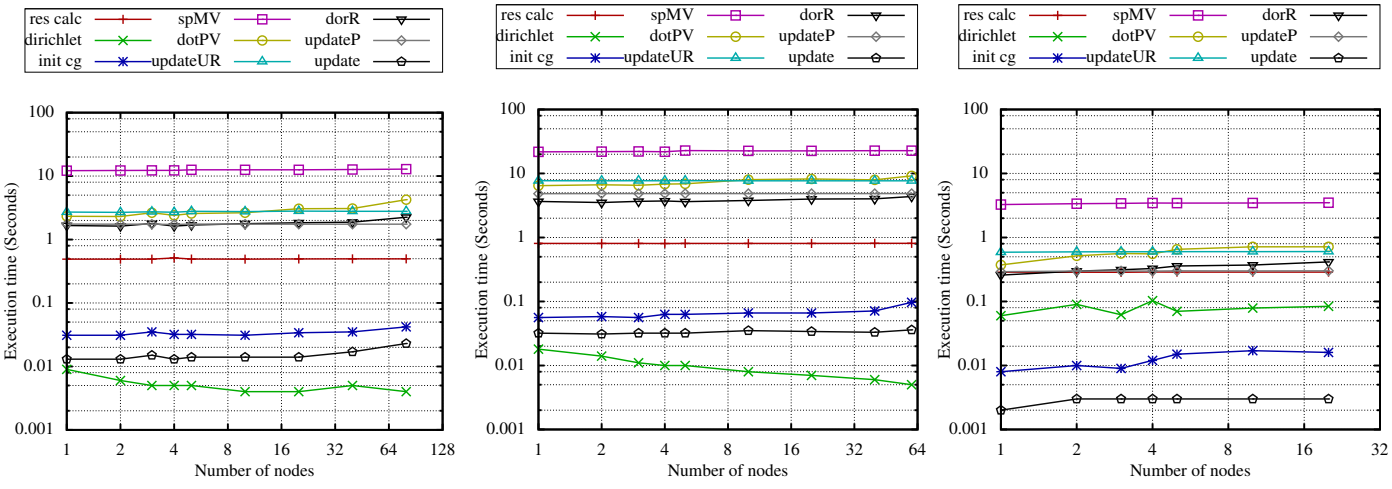
(a) HECToR (MPI) (b) IRIDIS (MPI) (c) EMERALD (MPI+CUDA)

Figure 11: Runtime breakdown of Aero 720K cells (strong-scaling) problem (20 Newton iterations) on HECToR, EMERALD and IRIDIS



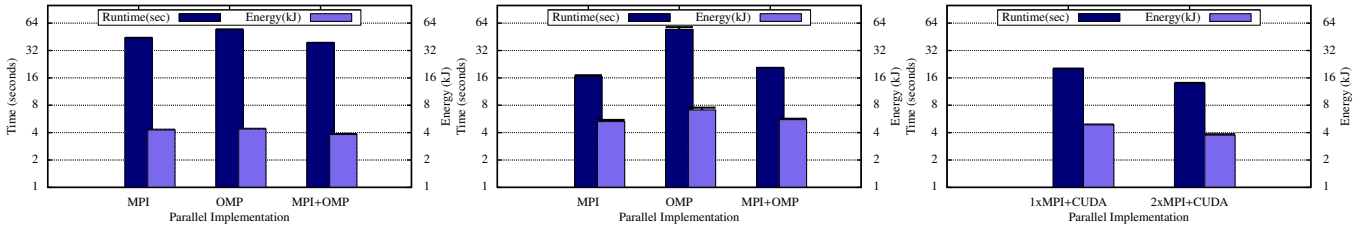
(a) HECToR (MPI) (b) IRIDIS (MPI) (c) EMERALD (MPI+CUDA)

Figure 12: Runtime breakdown of Aero 12M edges (strong-scaling) problem (1000 iterations) on HECToR, EMERALD and IRIDIS



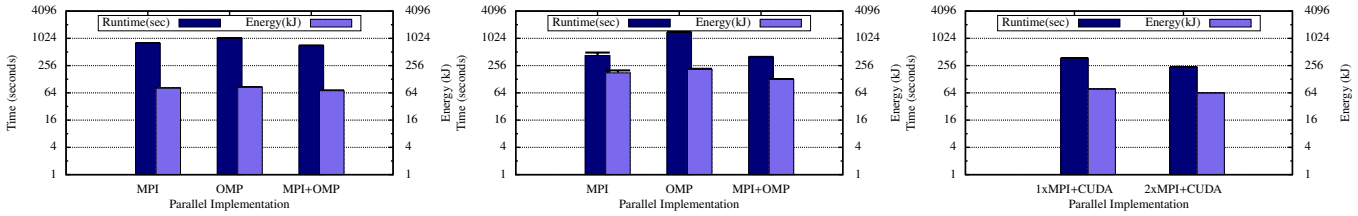
(a) HECToR (MPI) (b) IRIDIS (MPI) (c) EMERALD (MPI+CUDA)

Figure 13: Runtime breakdown of Aero 720K cells per node (weak-scaling) problem (1000 iterations) on HECToR, EMERALD and IRIDIS



(a) 2× Intel Westmere (b) 2× AMD Interlagos (c) 2× NVIDIA C2070

Figure 14: Runtime Energy Consumption of Airfoil - 1.5M Edges (1000 iterations)



(a) 2× Intel Westmere (b) 2× AMD Interlagos (c) 2× NVIDIA C2070

Figure 15: Runtime Energy Consumption of Airfoil - 26M Edges (1000 iterations)

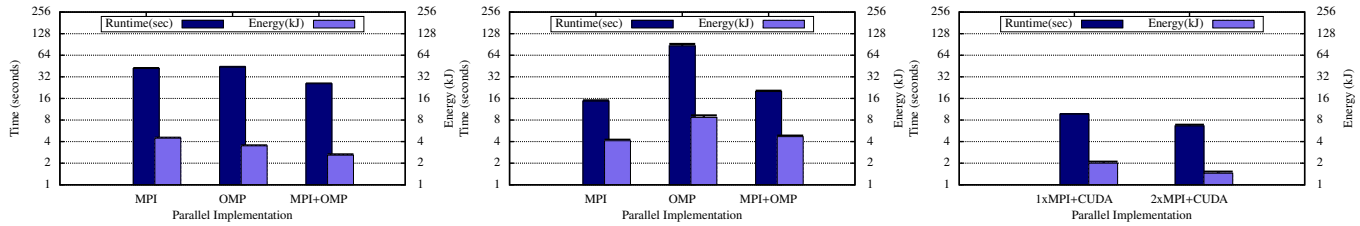
as their utility to compare different node systems. Figures 14, 15, 16 and 17 presents the energy consumption benchmark results. The figures also indicate the variance of the results using error bars. However these are considerably smaller than the mean energy consumption and not very visible at these graph scales. The standby power ($\overline{P}_q$) used in calculating the energy consumption of each node is listed in Table 8.

A key observation from these results for both applications is that the energy consumption approximately correlates to the runtime on the selected platform. That is, the longer the runtime the larger the energy consumption. Thus for example the short runtimes on the GPUs gives the lowest energy consumption figures. By comparing different CPU-based implementations, in majority of the cases, the hybrid version (MPI+OpenMP) offers better energy consumption than other implementations. This is true for both the Intel- and AMD-based platforms and for different problem sizes. The higher clock frequency of an Interlagos core (compared to a Westmere core) is one of the reasons why the Interlagos node is more energy consuming than the Westmere node. For the Airfoil benchmark, by comparing the best CPU-based version against GPU-based implementations, we see that MPI+CUDA variants have better energy footprint while offering nearly the same or better runtime performance. This is true irrespective of the problem size. For the Aero application, the OP2 heterogeneous back-end on two GPUs is about 3 to 4 times less energy consuming than a node with two Interlagos processors or 3 to 2 times less energy consuming than a node with two Westmere processors.

A key insight from these results is that the execution times of the applications on the heterogeneous platforms does not come at a disproportionate drain on energy. Thus for example, the approximately 4× to 2× speedups gained by the MPI+CUDA runs on the NVIDIA C2070 node for Aero (see Figure 16) is operating within the same energy envelope (and in this case consuming much less energy) as that of the energy consumed by the Westmere and Interlagos nodes.

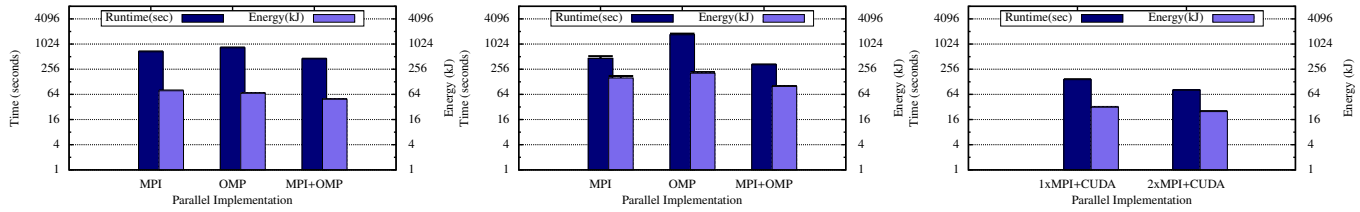
Our final set of results attempts to gain insights into the scaling of the cluster systems combining their runtime and energy consumption. In this case we use the TDP (Thermal Design Power) of the processors as advertised by the designers to compute the energy consumption of the cluster nodes. As exact energy consumption of the cluster scaling could not be directly measured, we believe that this method provides an approximate, yet useful method to compare the cluster systems. Figure 18 plots the energy consumption computed using the TDP of the nodes and execution time of Airfoil and Aero. For a HECToR node we use the TDP of two Interlagos 6276 processors (2×115W) [66] and for a IRIDIS node we use the TDP of two Westmere (E5645) processors (2×80W) [61]. For an EMERALD node we use the combined TDP of three C2090 GPUs (3×238W) [67] and one Westmere (X5650) processor (95W) [68]. We avoid using the full TDP of two Westmere processors assuming that these processors will only be used in MPI communications and both processors will not be working at its full capacity.

For both the applications, the CPU clusters (HECToR and IRIDIS) provide decreasing runtimes, at increasing scale, while having almost constant energy consumption. For example the execution using 1 HECToR node can be estimated to consume the same amount of energy as the execution on 10 nodes. The former takes a longer runtime, while the latter takes a shorter runtime. But the power×runtime product remains approximately constant. However this only happens when the application has good scaling behavior. Thus a reasonable decision can be made based on energy consumption, inferring when it is beneficial to commit additional nodes to a large-scale run. Such insights will be particularly valuable in the future when we are able to directly measure the energy cost of a cluster



(a) 2× Intel Westmere (b) 2× AMD Interlagos (c) 2× NVIDIA C2070

Figure 16: Runtime Energy Consumption of Aero - 720K Cells (20 Newton iterations)



(a) 2× Intel Westmere (b) 2× AMD Interlagos (c) 2× NVIDIA C2070

Figure 17: Runtime Energy Consumption of Aero - 12M Cells (20 Newton iterations)

during application execution. HECToR provides the best energy scaling for these two applications, but at large machine size. On the GPU cluster, EMERALD, both energy and runtime performance do not scale well. However, by using a small number of EMERALD nodes, we can obtain moderately fast completion times for both applications, consuming a relatively low amount of energy.

7. Conclusions and Future Work

In this paper, we presented the design of OP2's capabilities for developing and solving static unstructured mesh based applications on heterogeneous processor platforms. The research detailed key design issues in parallelizing unstructured mesh applications on heterogeneous back-ends, such as handling data dependencies in accessing indirectly referenced data and design considerations in generating code for execution on a cluster of GPUs and multi-threaded CPUs encompassing distributed memory and many- and multi-core parallelism. As part of this work, the runtime, scaling and energy performance of two application benchmarks developed using OP2 was benchmarked and analyzed on a range of heterogeneous systems.

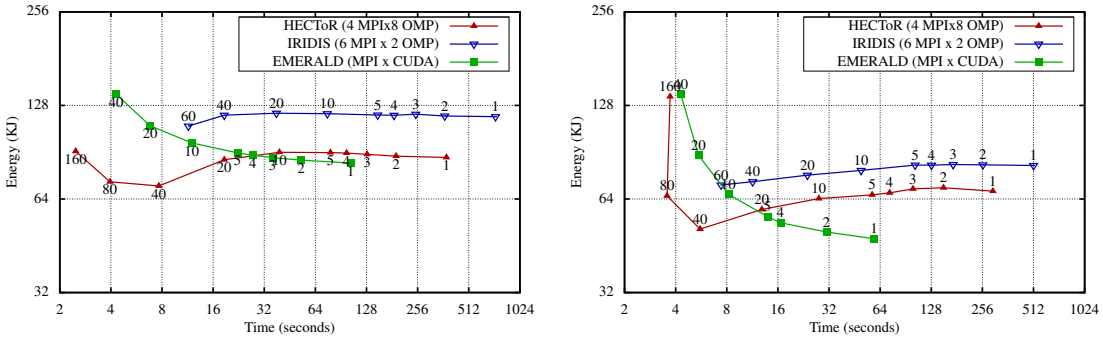
We carried out both strong- and weak-scaling benchmarks and reported the message passing performance metrics, such as proportion of halo sizes and neighbors per MPI process. When strong-scaling, in contrast to OP2's distributed memory single threaded MPI back-end, the heterogeneous multi-threaded cluster back-end (MPI+OpenMP) demonstrated better scalability at large machine sizes. The key factor in this case is the size of the halos and MPI communication neighbors. With the MPI+CUDA parallel back-end, the performance gains on a GPU cluster were more significant on larger unstructured meshes. However on the GPU cluster, performance is affected considerably by the amount of parallelism available per partition to be exploited by each GPU at scale. Thus when strong-scaling, at large machine scale the MPI+OpenMP back-end proves to give the best performance. In comparison, both applications on all three clusters showed excellent weak-scaling, pointing to near optimal performance of the distributed memory implementation. The GPU cluster, for these applications gave the best weak-scaling performance.

A number of key performance bottlenecks were identified when we analyzed the runtime break-down of the applications. These included (1) loops limited by CUDA kernel launch latency, (2) performance degradation of loops due to over partitioning at large scale and (3) performance limitations due to MPI collective operations.

The energy consumption of any back-end approximately correlates to the runtime on that platform. In general we see that MPI+CUDA variants have better energy footprint while offering nearly the same or better runtime performance. We see that the speedups gained through OP2 on the heterogeneous platforms do not incur a disproportionate drain on energy. However, further investigations into the energy performance at scale are needed to ascertain the best performing cluster system for these applications.

With the inclusion of the heterogeneous platform capabilities, in addition to single node CPU/GPU and homogeneous CPU clusters, OP2 now supports code generation for execution on a cluster of multi-threaded CPUs using MPI and OpenMP and a cluster of GPUs using MPI and CUDA. The GPU cluster back-end supports application development for discrete accelerator based heterogeneous systems. The MPI+OpenMP back-end will in the future be extended to support Intel MIC architecture based heterogeneous systems.

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



(a) Airfoil 26M Edges (strong-scaling)

(b) Aero 12M Cells (strong-scaling)

Figure 18: Energy (with TDP) vs Time - Airfoil (1000 iterations) and Aero (20 Newton iterations) on HECToR, EMERALD and IRIDIS (number of nodes used at each point indicated as a label on the plot)

paradigm shift for utilizing the ever-increasing complexity of novel hardware/software technologies.

The full OP2 source and the Airfoil and Aero application codes are available as open source software [69, 52] and the developers would welcome new participants in the OP2 project.

Acknowledgements

This research is funded by the UK Technology Strategy Board and Rolls-Royce plc. through the Siloet project, the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on "Multi-layered Abstractions for PDEs", the "Algorithms, Software for Emerging Architectures" (ASEArch) EP/J010553/1 project and the Future of Computing E9RCTDO Project funded by the Oxford Martin Foundation.

The work presented here made use of the IRIDIS and EMERALD High Performance Computing facility provided via the EPSRC funded Centre for Innovation (EP/K000144/1 and EP/K000136/1). The Centre is owned and operated by the e-Infrastructure South Consortium formed by the universities of Bristol, Oxford, Southampton and UCL in partnership with STFC Rutherford Appleton Laboratory.

We are thankful to Leigh Lapworth, Yoon Ho and David Radford at Rolls-Royce, Graham Markall, David Ham and Florian Rathgeber, at Imperial College London, Lawrence Mitchell at the University of Edinburgh and Endre László at PPKE Hungary, for their contributions to the OP2 project. We are also grateful to Andrew Richards of the Oxford Supercomputing Centre (OSC) and Ewan MacMahon at the Dept. of Physics at the University of Oxford for their support in providing access to a number of benchmarking systems used in this paper.

References

[1] What is GPU Computing, `http://www.nvidia.com/object/what-is-gpu-computing.html` (2013).

[2] T. Elgar, Intel Many Integrated Core (MIC) architecture, presentation at 2nd UK GPU Computing Conference, Cambridge, UK. `http://www.many-core.group.cam.ac.uk/ukgpucc2/talks/Elgar.pdf` (Dec, 2010).

[3] AMD Fusion APUs, `http://fusion.amd.com/`.

[4] Texas Instruments Multi-core TMS320C66x processor, `http://www.ti.com/c66multicore`.

[5] The convey hc-1 computer, `http://www.conveycomputer.com/Resources/ConveyArchitectureWhiteP.pdf`.

[6] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, H. Fu, Beyond traditional microprocessors for geoscience high-performance computing applications, Micro, IEEE 31 (2) (2011) 41 –49.

[7] F. Petrini, G. Fossum, J. Fernández, A. L. Varbanescu, M. Kistler, M. Perrone, Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine, in: IPDPS, 2007, pp. 1–10.

[8] S. J. Pennycook, S. D. Hammond, G. R. Mudalige, S. A. Wright, S. A. Jarvis, On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures, The Computer Journal 55 (2) (2012) 138–153.

[9] V. G. Asouti, X. S. Trompoukis, I. C. Kampolis, K. C. Giannakoglou, Unsteady CFD Computations Using Vertex-centered Finite Volumes for Unstructured Grids on Graphics Processing Units, International Journal for Numerical Methods in Fluids 67 (2) (2011) 232–246.

[10] OpenACC: Directives For Accelerators, `http://www.openacc-standard.org/`.

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

[12] T. L. Veldhuizen, D. Gannon, 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, 1998.

[13] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, P. H. J. Kelly, Performance Analysis of the OP2 Framework on Many-core Architectures, SIGMETRICS Perform. Eval. Rev. 38 (4) (2011) 9–15.

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

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

[16] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, I. Reguly, Designing OP2 for GPU Architectures, Journal of Parallel and Distributed Computing (in press, 2012).

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

[18] G. R. Mudalige, M. B. Giles, C. Bertolli, P. H. J. Kelly., Predictive modeling and analysis of OP2 on distributed memory

gpu clusters, SIGMETRICS Perform. Eval. Rev. 40 (2) (2012) 61–67.
[19] PETSc, http://www.mcs.anl.gov/petsc/petsc-as/.
[20] J. R. Stewart, H. C. Edwards, A Framework Approach for Developing Parallel Adaptive Multiphysics Applications, Finite Elem. Anal. Des. 40 (2004) 1599–1617.
[21] Deal.II: A Finite Element Differential Equations Analysis Library, http://www.dealii.org/.
[22] DUNE - Distributed and Unified Numerics Environment, http://www.dune-project.org/.
[23] FEATFLOW - High Performance Finite Elements, http://www.featflow.de/en/index.html.
[24] M. Zhou, O. Sahni, T. Xie, M. S. Shephard, K. E. Jansen, Unstructured Mesh Partition Improvement for Implicit Finite Element at Extreme Scale, J. Supercomput. 59 (3) (2012) 1218–1228.
[25] Flexible Distributed Mesh Database, http://www.scorec.rpi.edu/FMDB/ (2013).
[26] B. S. Kirk, J. W. Peterson, R. H. Stogner, G. F. Carey, 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.
[27] J. Jägersküpper, C. Simmendinger, A Novel Shared-Memory Thread-Pool Implementation for Hybrid Parallel CFD Solvers, in: E. Jeannot, R. Namyst, J. Roman (Eds.), Euro-Par 2011 Parallel Processing, Vol. 6853 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 182–193.
[28] L. W. Howes, A. Lokhmotov, A. F. Donaldson, P. H. J. Kelly, Deriving Efficient Data Movement from Decoupled Access/Execute Specifications, in: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 168–182.
[29] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Liszt: a Domain Specific Language for Building Portable Mesh-based PDE Solvers, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 9:1–9:12.
[30] K. B. Ølgaard, A. Logg, G. N. Wells, Automated Code Generation for Discontinuous Galerkin Methods, CoRR abs/1104.0628.
[31] F. Rathgeber, G. R. Markall, L. Mitchell, M. Loriant, D. A. Ham, C. Bertolli, P. H. J. Kelly, PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes, in: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:, 2012, pp. 1116–1123.
[32] The SCALA Programming Language, http://www.scala-lang.org/.
[33] T. Muranushi, Paraiso : An Automated Tuning Framework for Explicit Solvers of Partial Differential Equations, Computational Science & Discovery 5 (1) (2012) 015003.
[34] D. A. Orchard, M. Bolingbroke, A. Mycroft, 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, 2010, pp. 15–24.
[35] T. Brandvik, G. Pullan, SBLOCK: A Framework for Efficient Stencil-Based PDE Solvers on Multi-core Platforms, in: Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology, CIT '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1181–1188.
[36] G. R. Mudalige, I. Reguly, M. B. Giles, C. Bertolli, P. H. J. Kelly., 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., 2012.
[37] How to make best use of the AMD interlagos processor, www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf.
[38] D. A. Burgess, P. I. Crumpton, M. B. Giles, A Parallel Framework for Unstructured Grid Solvers, in: S. Wagner, E. Hirschel, J. Periaux, R. Piva (Eds.), Computational Fluid Dynamics'94:Proceedings of the Second European Computational Fluid Dynamics Conference, John Wiley and Sons, 1994, pp. 391–396.
[39] P. I. Crumpton, M. B. Giles, Multigrid Aircraft Computations Using the OPlus Parallel Library, Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers - 339–346, A. Ecer, J. Periaux, N. Satofuka, and S. Taylor, (eds.), North-Holland, 1996.
[40] The ROSE Compiler, http://www.rosecompiler.org/.
[41] D. A. Burgess, M. B. Giles, Renumbering Unstructured Grids to Improve the Performance of Codes on Hierarchical Memory Machines, Adv. Eng. Softw. 28 (1997) 189–201.
[42] W. Kahan, Pracniques: Further Remarks on Reducing Truncation Errors, Commun. ACM 8 (1965) 40.
[43] Top500 Systems, http://www.top500.org/list/ (Nov 2011).
[44] ParMETIS, http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.
[45] Scotch and PT-Scotch, http://www.labri.fr/perso/pelegrin/scotch/.
[46] M. Frigo, S. Johnson, The Design and Implementation of FFTW3, Proceedings of the IEEE 93 (2) (2005) 216 –231.
[47] E. L. Poole, J. M. Ortega, Multicolor ICCG Methods for Vector Computers, SIAM J. Numer. Anal. 24 (6) (1987) pp. 1394–1418.
[48] NVIDIA GPUDirect, http://developer.nvidia.com/gpudirect.
[49] M. B. Giles, D. Ghate, M. C. Duta., Using Automatic Differentiation for Adjoint CFD Code Development, Computational Fluid Dynamics Journal 16 (4) (2008) 434–443.
[50] M. Giles, Hydra, http://people.maths.ox.ac.uk/gilesm/hydra.html.
[51] M. B. Giles, M. C. Duta, J. D. Muller, N. A. Pierce, Algorithm Developments for Discrete Adjoint Methods, AIAA Journal 42 (2) (2003) 198–205.
[52] OP2 github repository, https://github.com/OP2/OP2-Common (2013).
[53] O. C. Zienkiewicz, R. L. Taylor, J. Z. Zhu, The Finite Element Method: Its Basis and Fundamentals, 6th Edition, Elsevier, New York, 2005.
[54] G. R. Markall, D. A. Ham, P. H. J. Kelly, Towards Generating Optimised Finite Element Solvers for GPUs from High-level Specifications, Procedia CS 1 (1) (2010) 1815–1823.
[55] HECToR - hardware, http://www.hector.ac.uk/service/hardware/.
[56] EMERALD gpu cluster, http://www.einfrastructuresouth.ac.uk/cfi/emerald (2013).
[57] IRIDIS gpu cluster, http://www.einfrastructuresouth.ac.uk/cfi/iridis (2013).
[58] How to make best use of the AMD Interlagos processor, www.hector.ac.uk/cse/reports/interlagos_whitepaper.pdf (Nov 2011).
[59] A. Solernou, J. Thiyagalingam, M. C. Duta, A. E. Trefethen., The Effect of Topology-Aware Process and Thread Placement on Performance and Energy, Lect. Notes Comput. Sci. (2013) 7905: 357–371.
[60] TESLA M2090 Product brief, http://www.nvidia.com/docs/IO/105880/DS_Tesla-M2090_LR.pdf.
[61] Intel xeon processor x5645, http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.
[62] A. Turner, X. Guo, L. Axner, Best Practice Guide CrayXE V3.1, www.prace-project.eu/IMG/pdf/Best-Practice-Guide-Cray-XE.pdf (2012).
[63] M. Giles, G. Mudalige, C. Bertolli, P. Kelly, E. Laszlo, I. Reguly, An Analytical Study of Loop Tiling for a Large-Scale Unstructured Mesh Application, in: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:, 2012, pp. 477–482.
[64] C. D. Krieger, M. M. Strout, C. Olschanowsky, A. Stone, S. Guzik, X. Gao, C. Bertolli, P. H. Kelly, G. Mudalige, B. V.

Straalen, S. Williams, Loop Chaining: A Programming Abstraction for Balancing Locality and Parallelism, in: (to appear) Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), 2013.

[65] Empack - Energy Efficient and Energy Aware Computing, http://www.oerc.ox.ac.uk/research/energy-efficient-computing.

[66] Bulldozer for Servers: Testing AMD's Interlagos Opteron 6200 Series, http://www.anandtech.com/show/5058/amds-opteron-interlagos-6200.

[67] Tesla C2050 and C2070 GPU Computing Processor, http://www.nvidia.in/object/product_tesla_C2050_C2070_in.html.

[68] Intel xeon processor x5650, http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI.

[69] OP2 for Many-Core Platforms, http://www.oerc.ox.ac.uk/research/op2 (2013).