OP2: An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures *

G.R. Mudalige, M.B. Giles
Oxford eResearch Centre,
University of Oxford, U.K.
gihan.mudalige@oerc.ox.ac.uk,
mike.giles@maths.ox.ac.uk

I. Reguly
Pázmány Péter Catholic
University, Hungary
reguly.istvan@itk.ppke.hu

C. Bertolli, P.H.J Kelly
Dept. of Computing,
Imperial College London, U.K.
{c.bertolli, p.kelly}@imperial.ac.uk

ABSTRACT

OP2 is an "active" 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 different parallel implementations for execution on different back-end hardware platforms. In this paper we present the design of the current OP2 library, and investigate its capabilities in achieving performance portability, near-optimal performance, and scaling on modern multi-core and many-core processor based systems. A key feature of this work is OP2's recent extension facilitating the development and execution of applications on a distributed memory cluster of GPUs.

We discuss the main design issues in parallelizing unstructured mesh based applications on heterogeneous platforms. These include handling data dependencies in accessing indirectly referenced data, the impact of unstructured mesh data layouts (array of structs vs. struct of arrays) and design considerations in generating code for execution on a cluster of GPUs. A representative CFD application written using the OP2 framework is utilized to provide a contrasting benchmarking and performance analysis study on a range of multi-core/many-core systems. These include multi-core CPUs from Intel (Westmere and Sandy Bridge) and AMD (Magny-Cours), GPUs from NVIDIA (GTX560Ti, Tesla C2070), a distributed memory CPU cluster (Cray XE6) and a distributed memory GPU cluster (Tesla C2050 GPUs with InfiniBand). OP2's design choices are explored with quantitative insights into their contributions to performance. We demonstrate that an application written once at a high-level using the OP2 API can be 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.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]; C.1.2 [**Multiple Data Stream Architectures**]

*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 and the Algorithms and Software for Emerging Architectures (ASEArch) EP/J010553/1 project

Keywords

OP2, Active Library, Domain Specific Language, Unstructured mesh, GPU

1. INTRODUCTION

With the advent of novel processor architectures such as general purpose GPUs and heterogeneous many-core processors (e.g. Intel MIC [33], AMD APUs [1]), the latest "many-core" programming extensions and technologies are needed to take advantage of the full potential of emerging parallel high performance systems. Even traditional CPUs have increasingly larger vector units (e.g. AVX) and CPU clusters are advancing towards capabilities to scale up to a billion threads. In turn, the increasing number of processor cores demands more data to be exchanged between main-memory and processor, within a complicated memory hierarchy, making data-movement costly and bandwidth an increasingly important bottleneck. Such developments demand radical new approaches to programming applications in order to maintain scalability. This problem is compounded by the rapidly changing hardware architectures landscape with additional limitations due to chip energy consumption and resilience. Application developers would like to benefit from the performance gains promised by these new systems, but are very worried about the software development costs involved and the need to constantly maintain an expert level of knowledge in the details of new technologies and architectures in order to obtain the best performance from their codes. It is therefore clear that a level of abstraction is highly desirable so that domain application developers/scientists can reach an increased level of productivity and performance without having to learn the intricate details of new architectures.

Such an abstraction enables application developers to focus on solving problems at a higher level and not worry about architecture specific optimizations. This splits the problem space into (1) a higher application level where scientists and engineers concentrate on solving domain specific problems and write code that remains unchanged for different underlying hardware and (2) a lower implementation level, that focuses on how a computation can be executed most efficiently on a given platform by carefully analyzing the computation, data access/communication and synchronization patterns. The correct abstraction will pave the way for easy maintenance of a higher-level application source by domain application developers but allow optimization and

978-1-4673-2633-9/12/$31.00 ©2012 IEEE

parallel programming experts to apply radically aggressive and platform specific optimizations when implementing the required solution on various hardware platforms. The objective will be to provide near optimal performance without burdening the domain application developers. Furthermore, once a correct abstraction is established it will 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 where a single application code written using the OP2 API can be transformed in to different parallel implementations which can then be linked against the appropriate parallel library (e.g. OpenMP, CUDA, MPI, OpenCL, AVX, etc.) enabling execution on different back-end hardware platforms. A key feature of this approach is that at the user application level the API statements appear similar to normal function calls simplifying the application development process. 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.

OP2 currently supports code generation and execution on a number of different platforms: (1) single-threaded on a CPU, (2) multi-threaded using OpenMP for execution on a single SMP node consisting of multi-core CPUs (including large shared-memory nodes), (3) parallelized using CUDA for execution on a single NVIDIA GPU, (4) parallelized on a cluster of CPUs using MPI and (5) parallelized on a cluster of NVIDIA GPUs using MPI and CUDA. Additionally, back-ends targeting OpenCL, AVX multi-cores and a cluster of multi-threaded CPUs (using MPI + OpenMP) are currently nearing completion. In our previous work [24, 25, 16] we presented OP2's API and its back-end design facilitating the code generation for and execution of unstructured mesh applications on single-node systems. In this paper we (1) extend the OP2 design to include multi-GPU platforms and (2) explore OP2's capabilities in achieving near optimal performance, performance portability and scaling on various current multi-core and many-core processor based systems. We begin in Section 2 with a brief overview of the OP2 API and a review of its main design strategies for handling (1) data dependencies on single/shared-memory nodes and distributed memory systems and (2) unstructured mesh data layouts (array of structs vs. struct of arrays). We then present the design of OP2's multi-GPU back-end which facilitates the execution of an OP2 application on a cluster of GPUs. Next in Section 3 we investigate the performance of an industrial representative CFD code written using OP2 on a range of current flagship multi-core and many core systems. Benchmarked systems consist of multi-core CPUs from AMD (Magny-Cours), Intel (Westmere and Sandy Bridge) and GPUs from NVIDIA (GTX560Ti, Tesla C2070), a distributed memory cluster (Cray XE6) and a distributed memory GPU cluster (based on Tesla C2050 GPUs interconnected by DDR InfiniBand). OP2's design choices are explored with quantitative insights into their contributions to performance.

We believe that results from a performance analysis study of a standard CFD benchmark, the Airfoil, on GPU clusters is an important step forward with respect to our previous work [24, 25, 16]. With the inclusion of the multi-GPU back-end, the range of target back-ends 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. We use highly-optimized code generated through OP2 for both CPU and GPU 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 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.

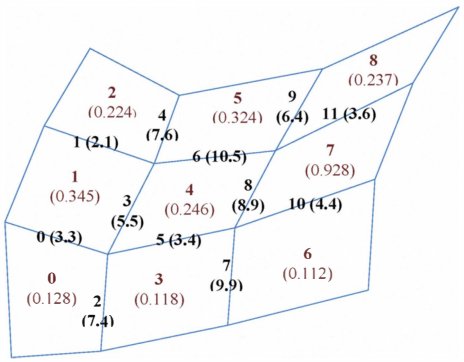


Figure 1: An example mesh with edge and quadrilateral cell indices (data values in parenthesis)

2. OP2

2.1 The OP2 API

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

Unstructured meshes, unlike structured meshes, use 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 [17, 19]) involves breaking down 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 which define how elements of one set connect with the elements of another set.

In our previous work [24, 25] 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 [16]. 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 of each map entry (e.g. 2; as **pecell** maps each edge to 2 cells). Once the sets and connectivity 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 its X,Y,Z 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.

```
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 elemental 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). **decells** 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 through the OP2 compiler and will produce a modified main program and back-end specific code. These are then compiled using a conventional compiler (e.g. gcc, icc, nvcc) and linked against platform specific OP2 back-end libraries to generate the final executable. In the OP2 project we currently have two prototype compilers, one written in MATLAB which only parses OP2 calls and a second source-to-source translator built using the ROSE compiler framework [10] which is capable of full source code analysis. Preliminary details of the ROSE source-to-source translator can be found in [16]. The slightly verbose API was needed as a result of the initial MATLAB prototype parser but also facilitate 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 restrictions on the actual algorithms, it just separates the components of a code [24, 25]. 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 [18], 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 [28] 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 multiple GPUs [31].

Currently, OP2 supports 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, a cluster of CPUs using MPI and a cluster of GPUs using MPI and CUDA. In the next sections we present in detail the design of OP2 for parallelizing unstructured mesh applications on these contrasting back-end platforms. The OP2 auto-generated code contains all of the best hand-tuning optimizations to our knowledge (except for using SoA data layout for solely directly accessed data as detailed in Section, 2.4) and we are not aware of any ways of obtaining additional performance at the time of writing. The performance data on bandwidth in Section 3.1 show that there is very little scope for additional speedup.

2.2 The OP2 Parallelization Strategy

OP2 uses hierarchical parallelism with two principal levels: (1) distributed memory and (2) single-node/shared-memory. Using exactly the same approach as OPlus [17, 19], the distributed memory level uses standard graph partitioning techniques in which the domain is partitioned among the compute nodes of a cluster, and import/export halos are constructed for MPI 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 [8] and PT-Scotch [12] to obtain high quality partitions.

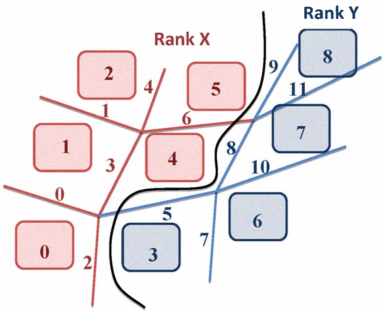
The single-node design is motivated by several key factors. Firstly a single node may have different kinds of parallelism depending on the target hardware: on multi-core CPUs shared memory multi-threading is available with the possibility of each thread using vectorization to exploit the capabilities 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 3-6GB 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 parallelised; 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 [21], 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. We investigate the performance impact of choosing the correct mini-partition and thread-block size quantitatively in Section 3.

2.3 Data Dependencies

One key design issue in parallelizing unstructured mesh computations is managing data dependencies encountered when



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

incrementing indirectly referenced arrays [24, 25]. 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 [17, 19] 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 2 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 forms 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 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

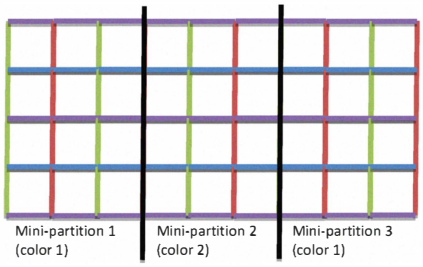


Figure 3: OP2 coloring of a mesh - each edge of the same color can be evaluated in parallel

execute block they need to reference cells 3, 7 and 8. Thus cells 3, 7 and 8 forms 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.

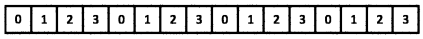
Within a single GPU, the size of the mini-partitions is very small, and so the proportion of redundant computation would be unacceptably large if we used the owner-compute approach. Instead we use the approach previously described in [24, 25], in which we adopt the "coloring" idea used in vector computing [32]. 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 to this 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 3. When incrementing the cells, the thread block first computes the increments for each edge, and then loops over the diferent edge colors applying the increments by color, with thread synchronisation between each color. This results in a slight loss of performance during the incrementing process due to warp divergence, but the cost is minimal.

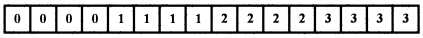
A similar technique is used for multi-core processors. The only difference is that now each mini-partition is executed by a single OpenMP thread. The mini-partitions are colored to stop multiple mini-partitions attempting to update the same data in the main memory simultaneously. This technique is simpler 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.

2.4 Data Layout in Memory

Another key design issue in generating efficient code for different processor architectures is the layout in which data should be organized when there are multiple components for each element. For example, a set element such as a cell or an edge can have more than one data variable; if there are 4 values per cell should these 4 components be stored contiguously for each cell (a layout which is referred to as



(a) Array-of-Structs (AoS)



(b) Struct-of-Arrays (SoA)

Figure 4: AoS vs SoA data layouts

an array-of-structs, AoS) or should all of the first components be stored contiguously, then all of the second components, and so on (a layout which is referred to as a struct-of-arrays, SoA)? Figure 4 illustrates the two options. The array-of-structs (AoS) approach views the 4 components as a contiguous item, and holds an array of these. The struct-of-arrays (SoA) approach has a separate array for each one of the components.

In [25] we qualitatively discussed OP2's design of data layouts. In this section we present a more detailed evaluation of the pros and cons of the AoS and SoA data layouts. Our assessment enables us to quantitatively estimate the benefits of the data layouts for a concrete application in Section 3.1. The SoA layout was natural in the past for vector supercomputers which streamed data to vector processors, but the AoS layout is natural for conventional cache-based CPU architectures for two reasons. Firstly, if there is a very large number of elements then in the SoA approach each component for a particular element will be on a different virtual page, and if there are a lot of components this leads to poor performance[1]. The second is due to the fact that an entire cache line must be transferred from the memory to the CPU even if only one word is actually used. This is a particular problem for unstructured grids with indirect addressing; even with renumbering of the elements to try to ensure that neighboring elements in the grid have similar indices, it is often the case that only a small fraction of the cache line is used (vector supercomputers circumvented this problem by adding gather/scatter hardware to the memory sub-system). This problem is worse for SoA compared to AoS, because each cache line in the SoA layout contains many more elements than in the AoS layout. In an extreme case, with the AoS layout each cache line may contain all of the components for just one element, ensuring perfect cache efficiency, assuming that all of the components are required for the computation being performed. For these reasons, the OP2 back-ends for x86 based CPUs use the AoS data layout for both direct and indirectly accessed data.

Until recently, NVIDIA GPUs did not have caches and most applications have used structured grids. Therefore, most researchers have preferred the SoA data layout which leads to natural memory transfer "coalescence" giving high throughput. However, the current NVIDIA GPUs based on the Fermi architecture have L1/L2 caches with a cache line size of 128 bytes, twice as large as used by Intel's Westmere and the Sandy Bridge CPUs [15]. This leads to significant problems with cache efficiency, especially since there is only 48kB of local shared memory and so not many elements are worked on at the same time by a single thread-block. For example, in the benchmark application used in Section 3 (Airfoil), in one of the indirect loops (`res_calc`) there are four floating-point values to be computed on. Within the Fermi architecture with a 128 bytes cache line, this corresponds to 32 floating-point values in single precision. When data is accessed indirectly, the SoA layout can lead to a worst-case scenario in which only 1/32 of the cache line is used. But with the AoS layout the worst case is only 1/8. Hence, in extreme cases with almost random addressing, the AoS layout could be 4 times more efficient than the SoA layout. The savings could be even larger for applications with more data per set element. Consequently, the AoS layout is used in OP2 for indirectly accessed data. Because indirect datasets are staged in shared memory and their transfer from global memory takes place before executing the user's kernel, cache efficiency is maximized.

For directly accessed data, on the other hand, if AoS is used, then if there are N components in the data array then a warp of 32 threads uses 32N pieces of data. This is N number of cache lines when working with floats. Given the very constrained cache size per thread, not all cache lines using AoS can fit into the cache, resulting in a high ratio of misses. For large N (e.g. for LMA matrices [29]) this will be significant. But if SoA is used instead in this case, then the same cache line will be reused N times. Furthermore, within direct loops where all data arrays are directly accessed, SoA gives perfectly coalesced access as the array allocation is cache aligned. Currently OP2 uses AoS for all arrays on GPUs but we plan to change this to SoA for arrays only accessed directly.

Currently for indirectly accessed data, OP2 does on-the-fly data transposition. The following code segment demonstrates this for an AoS with four elements similar to Figure 4:

```
float arg_l[4];                \\ register array
__shared__ float arg_s[4*32]; \\ shared memory

for (int m=0; m<4; m++)
    arg_s[tid+m*32] = arg_d[tid+m*32];

for (int m=0; m<4; m++)
    arg_l[m] = arg_s[m+tid*4];
```

The first loop does a coalesced transfer from the global memory array `arg_d` into the shared memory array `arg_s` for each `tid` thread. By using a separate shared memory "scratch-pad" for each warp, we can generalize this without needing thread synchronization. However, this method may utilize too much shared memory and could increase register pressure for future applications; hence the plan to implement SoA layout for solely directly accessed data.

2.5 Multi-GPU Systems

The latest addition to OP2's multi-platform capabilities is the ability to execute an OP2 application on a cluster of GPUs. The design of OP2 for such a back-end involved 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.

The OP2 multi-GPU 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 node. 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,

[1]On an IBM RS/6000 workstation in the 1990's, one of the authors experienced a factor 10 drop in performance due to the limited size of the Translation Look-aside Buffer which holds a cache of the virtual memory address tables.

Algorithm

1. **for** each op_dat requiring a halo exchange {
2. **execute** CUDA kernel to gather export halo data
3. copy export halo data from GPU to host
4. **start** non-blocking MPI communication }
5. **execute** CUDA kernel with op_plan for *core* elements
6. **wait for all** MPI communications to complete
7. **for** each op_dat requiring a halo exchange
8. { copy import halo data from host to GPU }
9. **execute** CUDA kernel with op_plan for non-*core* elements

Figure 5: Multi-GPU non-blocking communication

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 creates two separate plans, one for executing only the *core* elements and the second for all other elements (including redundant computation). 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 `op_plan` for the *core* elements will be computed while non-blocking communications are in-flight. Each `op_plan` consists of a mini-partitioning and coloring strategy optimized for their respective loop and number of elements. The halos are transfered 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 [7] technology for transferring data directly between GPUs. This will be implemented in future work for the OP2 MPI+CUDA back-end.

3. PERFORMANCE

In this section, we present quantitative results exploring the performance portability and scaling of the OP2 design. An industrial representative CFD code, Airfoil, written using OP2's C/C++ API is used in this performance analysis and benchmarking. Airfoil is a non-linear 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 [22, 23] CFD code at Rolls Royce plc.). The Airfoil code consists of five parallel loops: `save_soln`, `adt_calc`, `res_calc`, `bres_calc` and `update`. Out of these, `save_soln` and `update` are direct loops while the other three are indirect loops. We use two mesh sizes (1.5M and 26M edges) in this analysis. 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. All results presented are from execution in double-precision floating-point arithmetic.

Table 1: Single node CPU system specifications

Node System	Cores /node (Clock/core)	Mem. /node	Compiler [flags]
2×Intel Xeon X5650 (Westmere)	12 [24 SMT] (2.67GHz)	24 GB	ICC 11.1 [-O2 -xSSE4.2]
Intel Core i7-2600K (Sandy Bridge)	4 [8 SMT] (3.4GHz)	8 GB	ICC 12.0.4 [-O2 -xAVX]
2×AMD Opteron 6128 (MagnyCours)	16 (2.0GHz)	12 GB	ICC 12.0.4 [-O2 -xSSE2]

Table 2: Single node GPU system specifications

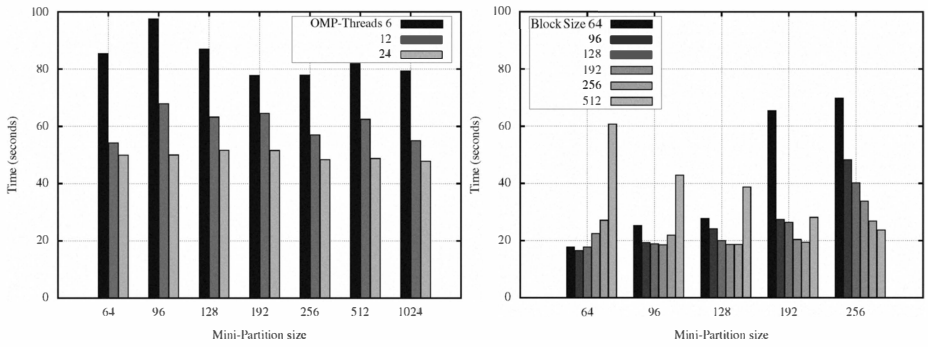
GPU	Cores	Clk	Glob.Mem (ECC)	Driver ver. (Comp.Cap.)
GeForce GTX560Ti	384	1.6 GHz	1.0 GB (off)	4.0 (2.1)
Tesla C2070	448	1.15 GHz	6.0 GB (off)	4.0 (2.0)

3.1 Single Node Systems

Table 1 and 2 detail the key hardware and software specifications of the CPU and GPU nodes used in our benchmarking study. The Intel Xeon E5650 (based on the Westmere micro-architecture) node consist of two Intel Xeon E5462 hex-core (total of 12 cores) processors operating at 2.67GHz and 24GB of main memory. The Intel Core i7-2600K processor node (based on Intel's Sandy Bridge micro-architecture), consists of a single 3.4GHz quad-core processor and 8GB of main memory. Both have simultaneous multi-threading (SMT) enabled for the execution of 24 and 8 SMT threads respectively. The AMD processor node (based on the Magny-Cours architecture) consist of two 8-core (total of 16 cores) 2.0GHz processors and 12GB of main memory. On each processor we set compiler flags to utilize the latest SSE/AVX instruction sets. For brevity and to avoid confusion for the rest of this paper, we refer to these processor nodes as Westmere, Sandy Bridge and Magny-Cours. The GPU systems consist of a consumer grade GPU (GTX560Ti) and the high performance computing NVIDIA C2070 GPU. Both are based on NVIDIA's Fermi architecture.

Recall that the thread-blocks and mini-partitions are key features in the OP2 design for efficiently distributing work and operating in parallel over the mesh elements. The first set of results explores the performance trends due to selecting different mini-partition and thread block sizes on these single node systems. The thread-block and mini-partition sizes for the overall application can be set at run-time using command-line arguments or a different value for each individual loop could be set at compile time. Figure 6 illustrates the typical performance trends we observe when the overall application level thread-block and mini-partition sizes are varied on the Westmere and C2070 nodes. These results follow the performance trends that was observed previously in [24, 25] for older multi-core CPUs (Intel Penryn, Intel Nehalem) and GPUs (GTX260, C2050).

Qualitatively all three CPUs and GPUs showed performance behaviors similar to the trends seen in Figure 6 (a) and (b) respectively, when the mini-partition and thread-block sizes are varied. On the CPU nodes, only mini-partitions are used, where each mini-partition of the same color is solved in parallel by an OpenMP thread. Increasing the number of OpenMP threads appear to give diminishing returns and different mini-partition sizes give only a minor variation in performance. In contrast, on the GPUs there is significant variation in performance due to using different



(a) Westmere (b) Tesla C2070

Figure 6: Runtime of Airfoil (1.5M edges, 1000 iterations) on the Westmere and C2070 node on a range of mini-partition and thread-block size configurations

Table 3: Airfoil single node results: 1.5M edges, 1000 iterations

System	Num. OMP	Time (sec)	res_calc GFlops /sec	res_calc GB/sec (useful)	res_calc GB/sec (cache)
Westmere	24	37.85	16.83	15.10	15.39
Sandy Bridge	8	62.80	9.39	13.71	13.90
Magny-Cours	16	46.30	12.70	11.40	11.61
GTX560Ti	-	19.63	23.01	24.17	40.30
C2070	-	13.20	35.50	34.38	46.51

mini-partition and thread block size configurations.

In general, on the GPUs we see that using a thread-block size equal to the mini-partition size gets close to the best performance achievable for each mini-partition size. However in some cases having a number of spare threads may be useful for carrying out more memory loads simultaneously with computation, but only when the GPU occupancy limits [14] are exceeded. Thus for example if having a larger thread-block size than the mini-partition size utilizes more registers than the maximum available number of registers per SM, then we see a performance degradation due to register spillage into global memory. Given a thread-block size equal to the mini-partition size, reducing the mini-partition size decreases the amount of shared memory used and thus the GPU is able to execute multiple thread-blocks at the same time on each SM. This is advantageous as now one thread-block can be loading data into shared memory while another block is doing the computation. On the other hand, smaller mini-partitions result in less data re-use as the ratio of boundary/interior nodes and cells increases. Smaller mini-partitions also decrease cache efficiency. We believe that these conflicting trends account for the run times we see in the above figure. However, there is no straightforward way of knowing the best parameters for a new application from the outset. Thus, on GPUs, incorrectly "guessing" these values can lead to significantly poorer performance.

Table 3 presents the best run-times gained on each single-node system. The optimum mini-partition size, thread block size and OpenMP number of threads (Num. OMP) were obtained using a recently developed auto-tuning framework [5]. In this case we auto-tuned both mini-partition and thread-block sizes for each of the five parallel loops to obtain the best run-times for each system. The final two columns presents the achieved floating-point rate (in DP) and effective bandwidth between main-memory or global-memory on the CPUs and GPUs respectively for the `res_calc` loop in Airfoil. As mentioned before, this loop is the most compute intensive

Table 4: Ratio of data transfer rates (SoA/AoS) on the Tesla C2070 : 1.5M edges, 1000 iterations

Loop	Mini-partition size			
	64	128	256	512
`adt_calc`	1.04	1.02	1.01	1.01
`res_calc`	1.99	1.65	1.39	1.22
`bres_calc`	2.49	2.48	2.48	2.47

loop in Airfoil and also the most time consuming one. When solving the 1.5M edge mesh, it performs about 30×10^{10} floating-point operations during the total runtime (i.e. 1000 iterations) of the application. 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 Airfoil, only a fraction of the advertised peak floating-point rates are achieved by any CPU or GPU. For example, only about 16 GFlops/sec out of a peak of about 127 GFlops/s (10.64 GFlops/s per core $\times$ 12 cores) on the Westmere is achieved. Similarly only about 30 GFlops/s out of a peak of 515 GFlops/s [13] is achieved on the C2070. On the other hand, the achieved bandwidth on the Westmere is close to half of its peak (32GB/s) which indicates that on CPUs the problem is much more constrained by bandwidth. 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 due to its low computational intensity. The trends on achieved computational intensity and bandwidth utilization remain very similar to the observed results from previous work [25] with about 25%-30% improvement over the previous CPUs (Intel Nehalem) and GPUs (C2050).

Next, we attempt to quantify and compare the amount of data transferred with GPU global memory due to the two different data layouts discussed in Section 2.4 for indirect data sets. We compute the amount of data transfered during each indirect loop (`adt_calc`, `res_calc` and `bres_calc`) as follows. Consider the case when a loop over elements (each containing a number of variables) of an indirectly accessed set is performed; for example the loop over cells (each with 4 flow variables) in `res_calc`. Then if the AoS data layout is used, we can compute the total number of bytes transferred from global memory to shared memory by counting the number of times a new cache line is loaded (assuming

Table 5: Cluster systems specifications

System	HECToR (Cray XE6)	SKYNET (GPU Cluster)
Node Architecture	2×12-core AMD Opteron 2.1GHz (Magny-Cours)	2×Tesla C2050 + 2×Intel Xeon E5440 2.83GHz
Memory/Node	32GB	2.6 GB/GPU (ECC on)
Interconnect	Cray Gemini	DDR InfiniBand
O/S	CLE 3.1.29	CentOS 5.6, Rocks 5.1
Compilers	PGI CC 11.3 Cray MPI	ICC 12.0.0 OpenMPI 1.4.3
Compiler flags	-Minline=levels:10 -Mipa=fast	-O2 -xSSE4.1 -arch=sm_20 -use_fast_math

that the first element of each array is cache-aligned). For each new cache line loaded, the amount of data transferred is incremented by the cache line size if the access is only a read operation. If the access is a write operation then we increment the amount of data transferred by two cache line sizes to account for the write back. The number of cache lines loaded will need to be increased if the variables making up an element takes more storage space than a single cache line size. Thus for example an element with 28 flow variables (each a double precision floating point value, i.e each of 8 bytes) will require 224 bytes of memory space in total. This is larger than the 128 byte cache line size on the NVIDIA Fermi architecture.

Alternatively consider the SoA data layout. Now, given a set with N elements each with v variables, then the distance between the first variable and the second variable (and so on) of each element will be $N\times$ `sizeof(double)` bytes. This number of bytes is significantly larger than a cache line of a GPU (or CPU), due to the size of N. Thus loading each element will mean that v number of new cache lines may need to be transferred from global memory to access all the variables for that element. In addition to the data values transferred, we also include in our calculation the bytes transferred due to loading mapping tables that are used to perform the indirect accessing of data. The ratio of data transfer rates (SoA/AoS) calculated in this method are given in Table 4. The results indicate that for indirect loops the AoS data layout is always better and for a number of cases reduces the data transfer between global memory and the GPU by over 50%.

3.2 Distributed Memory Systems

The single node results show that there are considerable performance gains to be made on GPU platforms. However, execution on large distributed memory clusters is needed for production-grade applications, due to higher computational and memory requirements. In this section we present OP2's performance on distributed memory platforms. We report run-time results and scaling behavior of Airfoil on two cluster systems: a traditional CPU cluster and a GPU cluster. Table 5 notes the key details of these systems. The first system, HECToR [6], is a large-scale proprietary Cray XE6 system which we use to investigate the scalability of the MPI implementation. The second system, SKYNET is a small C2050/InfiniBand cluster that we use to benchmark OP2's latest MPI+CUDA back-end. For this application we observed that PT-Scotch gave marginally better performing partitions (i.e smaller halo sizes and fewer MPI neighbors per process) and as such in all results presented we used PT-Scotch to partition the mesh.

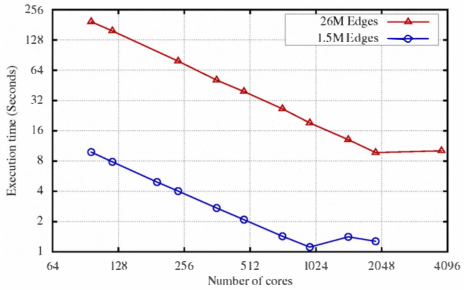


Figure 7: Airfoil strong scaling on HECToR (1.5M and 26M edges)

Table 6: CPU cluster vs. GPU cluster

System	Nodes	1.5M (sec)	26M (sec)
HECToR	5 (120 cores)	7.86	157.65
	10 (240 cores)	4.02	78.73
	20 (480 cores)	2.09	39.31
	40 (960 cores)	1.12	19.15
	60 (1440 cores)	1.41	13.07
	80 (1920 cores)	1.28	9.72
SKYNET	1/2 (1 × C2050)	22.08	-
	1 (2 × C2050)	12.22	186.83
	2 (4 × C2050)	7.44	93.57
	4 (8 × C2050)	5.25	53.06
	8 (16 × C2050)	4.28	27.39

Figure 7 reports the strong-scaled run-times of the application solving a mesh with 1.5M and 26M edges respectively on HECToR up to 3840 cores. The run-times given here are averaged from 5 runs for each processor core count. The standard deviation in run times was significantly less than 10% and thus we limited the number of times that each test was repeated to save time on the system. The figure shows excellent scalability for both problem sizes until it 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. For instance, the increase in runtime at 1440 cores for the 1.5M edges is due to an unusually large halo region created by the partitioner for that number of MPI processes. The best run-time for 1.5M edges is 1.12 seconds at 960 processor cores. As expected the 26M edge mesh continues to scale up further, giving a best runtime of about 9.72 at 1920 cores. We also observed up to 30% performance gains due to the use of non-blocking communications overlapped with computation during halo exchanges.

Comparing performance on HECToR to that of the GPU cluster SKYNET (See Table 6) reveals that for the 1.5M edge mesh both systems gives approximately similar performance at scale. For example five HECToR nodes (i.e. 120 Opteron cores) gives an equivalent runtime to 4×C2050 GPUs on SKYNET. However, for the larger 26M edge mesh, the performance gains are considerably higher on the GPU cluster. For instance four GPU nodes gives about 1.5 times the performance on five HECToR nodes. Our observation was that the GPU/CPU speedups are larger if we compare run-times from small machine sizes on these systems. However due to the limited memory resources on the GPU, larger mesh sizes may not fit on smaller systems (e.g. one C2050 GPU could not fit the the 26M edge mesh in global memory). The scalability on the GPU cluster is poorer than on HECToR for the 1.5M edge mesh. We believe that this is

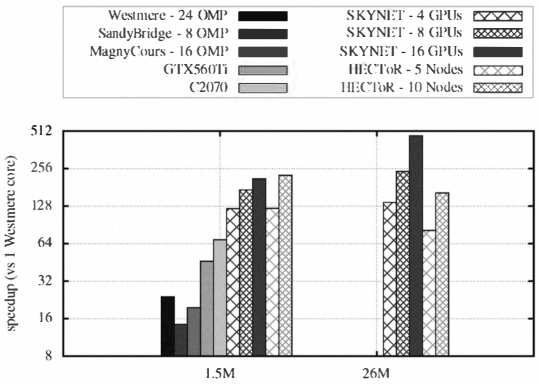


Figure 8: Airfoil speedups summary

due to the increasingly smaller amount of work assigned to each GPU at increasing scale. However, the 26M edge mesh scales well up to 16 GPUs, where the runtime is almost halved each time we double the number of GPUs.

We believe that a single node consisting of multiple GPUs will perform/scale in a similar (or even better) manner to SKYNET. The reason being that on a singe node with multiple GPUs the MPI messages will be transfered over PCIe while on a distributed memory GPU cluster (e.g. SKYNET) there is a much slower InfiniBand interconnect between inter-node GPUs. We will explore this further in future work.

A summary of the speedups gained on both single node and distributed memory systems is presented in Figure 8. The speedups are calculated compared to a single Westmere core. On the smaller mesh, the high-end C2070 GPU gives close to about 3× speedup over the 12-core Westmere node (running 24 OpenMP threads). It is surprising to see that notable performance gains can be achieved even on a single consumer-grade GTX560Ti for this application, especially as its double-precision performance is much poorer than the C2070. On the larger mesh, the GPU cluster gives higher performance gains than on the traditional cluster system.

4. 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 [9], Sierra [34] libraries as well as others such as Deal.II [2], Dune [3] and FEATFLOW [4]. There are also conventional libraries 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 for emerging multi-core/many-core technologies) for the solution of mesh based applications without the intervention of the application programmer.

OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [26] 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 [20] 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 compiler for UFL is being developed at Imperial College London to translate the FEniCS declarations down to code that uses the OP2 API. Thus, the performance results in this paper will directly relate to performance of code written using FEniCS in the future.

While OP2 uses an "active" library approach utilizing code transformation, Liszt [20] from Stanford University implements a domain specific language (embedded in Scala [11]) 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 (GPU, multi-core CPU, and MPI based cluster) executing a number of applications written using Liszt have been presented in [20]. The Navier-Stokes application in [20] is most comparable to the Airfoil application and shows similar speedups to those gained with OP2 in our work. Application performance on heterogeneous clusters such as on clusters of GPUs is not considered in [20] and is noted as future work.

5. CONCLUSION

In this paper, we presented the OP2 abstraction framework for the solution of unstructured mesh-based applications. A key contribution detailed in this work is OP2's recent extensions facilitating the development and execution of applications on a distributed memory cluster of GPUs.

We discussed OP2's key design strategies in parallelizing unstructured mesh based applications on a range of contrasting back-end platforms. These consisted of handling data dependencies in accessing indirectly referenced data, the impact of unstructured mesh data layouts (AoS vs. SoA) and design considerations in generating code for execution on a cluster of GPUs. OP2 currently supports generating code for execution on a single-threaded CPU, multi-threaded SMP/CMP node consisting of multi-core CPUs, a single NVIDIA GPU, a traditional CPU cluster and a GPU cluster. We benchmarked and analyzed the performance of an industrial representative CFD application written using the OP2 API on a range of modern flagship platforms to investigate OP2's performance portability and scaling.

Performance results show that for GPU platforms varying the thread-block and mini-partition size gives significant performance differences compared to CPU platforms. We also observed that indirectly accessed data should be formatted in the Array-of-Structs (AoS) layout for processors utilizing a cache. However, due to the limited cache size on NVIDIA GPUs, solely directly referenced arrays should be organized in the Struct-of-Arrays (SoA) format for higher performance.

The achieved floating-point performance on both GPU and CPU single node systems were only a small fraction of

the peak rates advertised by vendors. Bandwidth appears to become a bottleneck where, on some cases, over half of the peak bandwidth is utilized. We expect bandwidth to be a significant restriction for future processors where increased number of cores demands more data to be exchanged between main-memory and processor.

On distributed memory platforms, OP2's MPI back-end showed excellent scaling until the mesh was too small to be partitioned further. The performance gains on a GPU cluster were more significant on larger unstructured meshes compared to a traditional CPU cluster. Performance is affected considerably by the amount of parallelism available per partition to be exploited by each GPU at scale. Thus a balance must be achieved to not overload the resources of individual GPUs but at the same time have enough computation that can be parallelized within a node to gain good performance.

As future work we note that the OP2 multi-GPU back-end requires further benchmarking and analysis, particularly on a larger GPU cluster with better QDR InfiniBand networking. It will also need modifications to utilize NVIDIA's new GPUDirect technology. We are also aiming to complete development of other back-ends for OP2 including OpenCL and Intel AVX. Once completed, these will enable us to investigate performance on several other novel hardware platforms including heterogeneous processors such as the AMD fusion CPUs and the upcoming Intel MIC processor.

We believe that the future of numerical simulation software development is in the specification of algorithms translated to low-level code by a framework such as OP2. Such an approach will, we believe, offer revolutionary potential in delivering performance portability and increased developer productivity. This we predict will be an essential paradigm shift for utilizing the ever-increasing complexity of novel hardware/software technologies.

6. REFERENCES

[1] AMD Fusion APUs. http://fusion.amd.com/.

[2] Deal.II: A Finite Element Differential Equations Analysis Library. http://www.dealii.org/.

[3] DUNE - Distributed and Unified Numerics Environment. http://www.dune-project.org/.

[4] FEATFLOW - High Performance Finite Elements. http://www.featflow.de/en/index.html.

[5] Flamingo: A Flexible, Automatic Method for Independence-Guided Optimisation. http://mistymountain.co.uk/flamingo/.

[6] HECToR - hardware. http://www.hector.ac.uk/service/hardware/.

[7] NVIDIA GPUDirect. http://developer.nvidia.com/gpudirect.

[8] ParMETIS. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[9] PETSc. http://www.mcs.anl.gov/petsc/petsc-as/.

[10] The ROSE Compiler. http://www.rosecompiler.org/.

[11] The SCALA Programming Language. http://www.scala-lang.org/.

[12] Scotch and PT-Scotch. http://www.labri.fr/perso/pelegrin/scotch/.

[13] TESLA C2050/C2070 GPU Computing Processor. http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf.

[14] CUDA C Programming Guide 4.0, May 2011. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[15] Intel Xeon X5650 processor, 2011. http://www.cpu-world.com/sspec/SL/SLBV3.html.

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

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

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

[19] Crumpton, P. I., and Giles, M. B. 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.

[20] DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., and Hanrahan., P. Liszt: A domain specific language for building portable mesh-based PDE solvers. In *Proceedings of Supercomputing* ((to appear) 2011).

[21] Frigo, M., and Johnson, S. The Design and Implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (feb. 2005), 216 –231.

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

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

[24] Giles, M. B., Mudalige, G. R., Sharif, Z., Markall, G., and Kelly, P. H. J. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal In Press*, In Press (2011).

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

[26] Howes, L. W., Lokhmotov, A., Donaldson, A. F., and Kelly, P. H. J. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers* (Berlin, Heidelberg, 2009), HiPEAC '09, Springer-Verlag, pp. 168–182.

[27] JÄGERSKÜPPER, J., AND SIMMENDINGER, C. A novel shared-memory thread-pool implementation for hybrid parallel CFD solvers. In *Euro-Par 2011 Parallel Processing*, E. Jeannot, R. Namyst, and J. Roman, Eds., vol. 6853 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 182–193.

[28] KAHAN, W. Pracniques: Further Remarks on Reducing Truncation Errors. *Commun. ACM 8* (January 1965), 40.

[29] MARKALL, G. R., HAM, D. A., AND KELLY, P. H. J. Towards Generating Optimised Finite Element Solvers for GPUs from High-Level Specifications. *Procedia CS 1*, 1 (2010), 1815–1823.

[30] ØLGAARD, K. B., LOGG, A., AND WELLS, G. N. Automated code generation for discontinuous Galerkin methods. *CoRR abs/1104.0628* (2011).

[31] PENNYCOOK, S., HAMMOND, S., MUDALIGE, G., WRIGHT, S., AND JARVIS, S. On the Acceleration of Wavefront Applications using Distributed Many-Core Architectures. *The Computer Journal* (2011).

[32] POOLE, E. L., AND ORTEGA, J. M. Multicolor ICCG Methods for Vector Computers. *SIAM J. Numer. Anal. 24*, 6 (1987), pp. 1394–1418.

[33] SKAUGEN, K. Petascale to Exascale: Extending Intel's HPC Commitment, June 2011. ISC 2010 keynote. `http://download.intel.com/pressroom/archive/reference/ISC_2010_Skaugen_keynote.pdf`.

[34] STEWART, J. R., AND EDWARDS, H. C. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elem. Anal. Des. 40* (July 2004), 1599–1617.