Predictive Modeling and Analysis of OP2 on Distributed Memory GPU Clusters[*]

G.R. Mudalige, M.B. Giles
Oxford e-Research Centre, University of Oxford
mike.giles@maths.ox.ac.uk,
gihan.mudalige@oerc.ox.ac.uk

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

ABSTRACT

OP2 is an "active" library framework for the development and solution of unstructured mesh based applications. It aims to decouple the scientific specification of an application from its parallel implementation to achieve code longevity and near-optimal performance through re-targeting the back-end to different multi-core/many-core hardware. This paper presents a predictive performance analysis and benchmarking study of OP2 on heterogeneous cluster systems. We first present the design of a new OP2 back-end that enables the execution of applications on distributed memory clusters, and benchmark its performance during the solution of a 1.5M and 26M edge-based CFD application written using OP2. Benchmark systems include a large-scale CrayXE6 system and an Intel Westmere/InfiniBand cluster. We then apply performance modeling to predict the application's performance on an NVIDIA Tesla C2070 based GPU cluster, enabling us to compare OP2's performance capabilities on emerging distributed memory heterogeneous systems. Results illustrate the performance benefits that can be gained through many-core solutions both on single-node and heterogeneous configurations in comparison to traditional homogeneous cluster systems for this class of applications

Categories and Subject Descriptors

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

Keywords

OP2, Unstructured mesh, GPU, Performance modeling

1. INTRODUCTION

With heterogeneous HPC systems such as Tianhe-1A, Tsubame 2.0 and Nebulae gaining recognition as leading multi Peta-FLOP systems [6], there appears to be an increasing trend in using many-core processor architectures, in a hybrid combination with traditional CPUs. On the other hand, homogeneous systems such as the K-Computer and the IBM BlueGene range of systems, appear to be defending their dominant positions as the top performing systems in the world. As the many-core vs multi-core debate rages on, technologies that enable users to efficiently exploit these systems appear to be in an ever increasing state of flux, with a range of competing programming languages and architectural optimizations/-configurations. Application developers will need to constantly keep up an expert level of knowledge in the intricate details of these new technologies and architectures in order to obtain the best performance from their codes.

The demand of maintaining such a programming skills-set is distracting domain application developers from investing their full intellectual efforts in the scientific/engineering problems they are solving. It is clear that a level of abstraction is very much desirable such that computational scientists can increase their productivity by focusing on solving problems at a higher level, write code that remains unchanged for different underlying hardware and not worry about architecture specific optimizations. At the same time, a lower implementation level, maintained by HPC technology professionals and optimization experts, can focus on how a computation can be executed most efficiently on a given platform by carefully analyzing the computation and data access patterns. This paves the way for easily integrating support for any future novel hardware architecture and maintain near optimal performance.

OP2 aims to provide such an abstraction layer, by developing an "active" library framework for the solution of unstructured mesh applications. The "active" library approach uses program transformation tools, so that a single application code written using the OP2 API is transformed into the appropriate form that can be linked against a given parallel implementation enabling execution on different back-end hardware platforms. At the same time, OP2 attempts to maintain near-optimal performance by exploiting low-level optimizations and/or configurations on a target platform without the intervention of the domain application programmer.

OP2 currently enables application developers to write a single program (using the OP2 API, in either C/C++ or Fortran) which then can be transformed (using OP2 code transformation tools) into executables for three different platforms: (1) single-threaded on a CPU, (2) multi-threaded using OpenMP for execution on an SMP with multi-core CPUs, and (3) parallelized using CUDA for execution on a single NVIDIA GPU. Additionally, back-ends targeting OpenCL and AVX multi-cores are currently nearing completion. In our previous work [13, 10] we presented the design and implementation of OP2 for single node systems. As part of this work we analyzed and optimized the performance of an industry-representative unstructured mesh application (Airfoil [12]) from the CFD domain, written using the OP2 API. Performance was benchmarked on a range of flagship single node systems, consisting of NVIDIA GPUs and x86 based multi-core CPUs.

The next step of the OP2 development is to facilitate code generation and execution on heterogeneous systems such as clusters of single/multi-threaded CPUs or GPUs. In this paper we present an early performance evaluation of OP2's distributed memory capabilities for such systems. We present the design and performance of a distributed memory back-end based on MPI and utilize it together with performance modeling to present an "ahead of implementation" predictive performance analysis for a cluster of GPUs. Our objective is to gain quantitative and qualitative insights into the achievable performance on such systems and contrast it with performance from traditional homogeneous cluster solutions for

[*]This research is funded by the UK TSB and Rolls-Royce plc. through the Siloet project, and the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on Multi-layered Abstractions for PDEs.

unstructured mesh based applications. More specifically this paper makes the following contributions:

1. We present the design of the OP2 distributed memory layer which enables the execution of an OP2 application on a single threaded CPU cluster using MPI. The Airfoil application, previously explored on single multi-core CPU and GPU nodes, is transformed to utilize OP2's new MPI layer. Its performance is then benchmarked during the solution of an unstructured mesh consisting of about 1.5 million edges on a large CrayXE6 system. The performance of the application is presented including application run-times at scale and the impact of overlapping computations with communications.

2. The performance of Airfoil on the CPU cluster is compared to equivalent single GPU (NVIDIA GTX560Ti, Tesla C2070) and multi-threaded CPU (Intel Westmere, AMD MagnyCours) node performance. Results illustrate the performance benefits that can be gained from single node many-core systems in contrast to traditional homogeneous clusters for this class of applications.

3. Finally, a predictive performance model for Airfoil is developed and is used to explore the potential performance of the application on current/future peta-scale capable GPU clusters with projections for larger problem sizes (26 million edges). Our analysis gives insights into the performance of industrial unstructured mesh applications and the limiting factors of performance/scalability for such applications on GPU clusters.

2. OP2

Unstructured mesh based solutions have been and continue to be used over 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 involves breaking down the algorithm into four distinct parts: (1) sets, (2) data on sets, (3) connectivity (or mapping) between the sets and (4) mesh-wide computations on the data. 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. All the numerically intensive computations in the application are described as operations over sets. 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. This problem decomposition leads to an API with which a mesh or graph can be completely and abstractly defined. Figure 2 illustrates a simple quadrilateral mesh that we will use as an example to describe the OP2 API[1] in Figure 1.

This mesh can be defined by two sets (lines 1-3), nodes (vertices) and cells (quadrilaterals). The connectivity is declared through the mappings between the sets (lines 5-7). The integer array `cell_map` can be used to represent the four nodes that is referenced (or connected to) by each cell. Once the sets

[1]We use the C/C++ API throughout this paper. A similar Fortran API is also available.

```
1   int nnode = 16; int ncell = 9;
2   op_set nodes = op_decl_set(nnode, "set_nodes");
3   op_set cells = op_decl_set(ncell, "set_cells");
4
5   int cell_map[36] = { 0,1,5,4, 1,2,6,5, 2,3,7,6, ... };
6   op_map mcell     = op_decl_map(cells, nodes, 4,
7                       cell_map,"cell_to_node_map");
8
9   double cell_data[9] = {0.128, 0.345, 0.224, 0.118, ... };
10  double* cell_data_u = (double *)malloc(sizeof(double)*9);
11  double node_data[16] = {5.3, 6.8, 7.8, 5.4, ... };
12  op_dat dcells   = op_decl_dat(cells, 1, "double",
13                    cell_data, "data_on_cells");
14  op_dat dcells_u = op_decl_dat(cells, 1, "double",
15                    cell_data_u, "updated_data_on_cells");
16  op_dat dnodes   = op_decl_dat(nodes, 1, "double",
17                    node_data, "data_on_nodes");
18
19  void add(double* cell_u, double* cell,
20       double* n0, double* n1, double* n2, double* n3){
21    *cell_u = *cell + n0[0] + n1[0] + n2[0] + n3[0];
22  }
23  op_par_loop(add,"addkernel", cells,
24     op_arg(dcells_u,-1,OP_ID, 1, "double", OP_WRITE),
25     op_arg(dcells,  -1,OP_ID, 1, "double", OP_READ),
26     op_arg(dnodes,  0, mcell, 1, "double", OP_READ),
27     op_arg(dnodes,  1, mcell, 1, "double", OP_READ),
28     op_arg(dnodes,  2, mcell, 1, "double", OP_READ),
29     op_arg(dnodes,  3, mcell, 1, "double", OP_READ));
```

Figure 1: OP2 API example

and connectivity are defined, data can be associated with the sets; Lines 9-17 declare some data arrays that contain double precision data associated with the cells and the nodes respectively. Here a single double precision value per set element is declared. A vector of a number of values per set element could also be declared (e.g. a vector with three doubles per node to store the X,Y,Z coordinates).

Computations over elements of the mesh are defined as special OP2 loops (lines 19-29), called `op_par_loops`. The `op_par_loop` expresses a set-wide parallel computation, which in the proposed example visits all the cells of the mesh, and updates its data value by adding the data values held by the four nodes connected to it. The function given in `add` is the "elemental" operation to be performed per iteration. This elemental *kernel* function takes six arguments in this case. In this example, it is important to notice that OP2 requires the programmer to explicitly state as part of the `op_par_loop` call the access method for each argument (OP_WRITE, OP_READ, etc). To access the data of the four nodes connected to a given cell, indirection via the `mcell` mapping is used (e.g. data of the ith cell's 1st node is given by $node\_data\ [cell\_map[4{*}i{+}1]]$) with the given index (0, 1, 2, and 3). However OP_ID indicates that the data in `dcells` and `dcells_u` is to be accessed without any indirection. If the loop involves at least one argument accessed through a mapping, then we refer to it as an indirect loop; if not, it is called a direct loop. Complete details of the API can be found in [11].

Given a loop declaration of the above form, the architecture specific code generation and parallelization is handled by the OP2 framework. The OP2 strategy for building executables for different back-end hardware consists of firstly generating the architecture specific code by pre-processing the user code, which is written using the OP2 API, and then secondly linking the generated code with the appropriate parallel implementation back-end (e.g. OpenMP, CUDA, MPI, etc.).

2.1 Distributed Memory Parallelization

A key design issue in parallelizing unstructured mesh computations is managing data dependencies encountered when incrementing indirectly referenced arrays. For example, in a mesh with cells and nodes, with a loop over cells updating nodes, a potential problem arises when multiple cells update

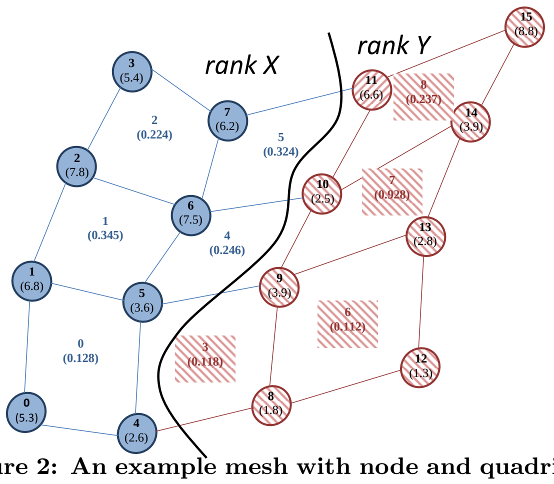

Figure 2: An example mesh with node and quadrilateral cell indices (data values in parenthesis)

the same node. A solution at a coarse grained level would be to partition the nodes such that the *owner* of the nodal data would carry out the computation. The drawback in this case is redundant computation when the nodes for a particular cell have different *owners*. At a finer-grained level, we could assign a "color" for the cells so that no two cells of the same color update the same node. This allows for parallel execution for each color followed by a synchronization. In our previous work [13, 10] we have presented the design and implementation of OP2 for single node systems which utilizes coloring to resolve dependency conflicts. In this section we present the design and implementation of the distributed memory backend layer based on MPI using an owner-compute model. The design builds on ideas developed previously in [7].

With an owner-compute parallelization, OP2 partitions the data so that the partition within each MPI process owns some of the set elements e.g. some of the nodes and cells. 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, where efficiency is based on the assumption that as partition size becomes larger, the proportion of halo data decreases in size.

For example consider the mesh illustrated in Figure 2. There are 16 nodes and 9 cells partitioned across two MPI processes (rank X and rank Y). Assume that the only mapping available is a cell to nodes mapping. Rank X holds nodes 0, 1, 2, 3, 4, 5, 6 and 7 and cells 0, 1, 2, 4, and 5. Rank Y holds nodes 8, 9, 10, 11, 12, 13, 14, and 15 and cells 3, 6, 7 and 8. A loop over the cells will possibly need data on nodes 9, 10 and 11 to be imported into rank X from rank Y. Additionally data on nodes 4 and 5 needs to be imported into rank Y from rank X. On the other hand a loop over cells will possibly need cells 4, 5 to be imported in to rank Y in order to compute correct nodal updates for nodes 9,10 and 11. Given the above scenario, each MPI process needs to construct a list of elements for each set that are to be imported from and exported to other "neighboring" MPI processes.

The OP2 design separates the set elements held within each MPI process according to the mappings that references other elements. On an MPI process, if all the elements referenced through all the mapping tables from a given set element is located at the same MPI process, we categorize the element to be a *core* element in this MPI process. If, however, at least one element referenced through any of the mapping tables from this set element is not in the *core* elements of the (local) MPI process, the set element is separated into the "export execute halo" (*eeh*). The *eeh* is a subset of the owned elements in an MPI process. A copy of the *eeh* will be exported to the relevant foreign MPI processes on which it will form part of the "import execute halo" (*ieh*). The *ieh* elements will be a redundant computation block which will need to be kept up-to-date in order to compute the correct contributions to any local element that it references. If an element located at an MPI process or an element belonging to the *ieh* on that process references (via some mapping) an element that is located on a foreign MPI process, then the element on the foreign MPI process needs to be imported. The imported element will fall in to the "import non-execute halo" if it is not already a part of the *ieh*. In turn this element will form part of the "export non-execute halo" (*enh*) on the foreign MPI process. The *enh* is a subset of *core*. For the mesh given in Figure 2, the import/export elements can be separated as given in Table 1. Developer documents further detailing the distributed memory parallelization design can be found in [11].

Table 1: Import/Export lists for the mesh in figure 2

X	*core*	*ieh*	*eeh*	*inh*	*enh*
Nodes	0,1,2,3,4,5,6,7	-	-	8,9,10,11	4,5,6,7
Cells	0,1,2	3	4,5	-	-
Y	***core***	***ieh***	***eeh***	***inh***	***enh***
Nodes	8,9,10,11,12,13,14,15	-	-	4,5,6,7	8,9,10,11
Cells	6,7,8	4,5	3	-	-

The above classification and ordering allows OP2 to clearly determine which elements of a set can be computed over without MPI communications, facilitating overlapping of computation with communications for higher performance. In an OP2 application executed in a distributed memory environment, prior to executing any loops involving computation (i.e. calls to `op_par_loop`), the above import/export halo elements list for each set is created and reordered into a contiguous array beginning with the *core* elements, followed by the *eeh*, *ieh* and *inh*.

A call to `op_par_loop` in an OP2 application executed under MPI will result in the loop being executed over the local elements of the set on each MPI process (i.e. *core* and *eeh*). Additionally, if the loop is an indirect loop then computation will be done over the *ieh* as well. The MPI execution follows the standard single program multiple data (SPMD) operation, where computations are done in steps which are interleaved with message passing steps. Computation over the *core* elements of a set does not require accessing any data on foreign processes. These computations can be overlapped with the communications required for exchanging the import/export halos between processes using non-blocking MPI communication primitives. For a given `op_par_loop`, halos are exchanged only for data that is accessed with OP_READ (read) or OP_RW (read/write). Additionally each iteration of the loop needs only to exchange the import/export halos if a previous iteration has marked the halos for this data as modified – by setting a "dirty" bit. The dirty bit is set at the end of an iteration for all data that has been accessed as OP_INC (increment), OP_WRITE or OP_RW.

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. In other words, 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 [3] and PT-Scotch [5] to obtain high quality partitions. In the current implementation, the user gets the option to select a routine from the above partitioner packages, indicating the "primary set" to be partitioned. OP2 creates the required adjacency lists and then calls the appro-

Table 2: Cluster systems specifications

System	HECToR (Cray XE6)	CX1 (Dell Cluster)
Node Architecture	2×12-core AMD Opteron 2.1GHz (Magny Cours)	2×6-core Xeon X5650 2.67GHz (Westmere)
Cores/Node	24	12 (24 SMT)
Memory/Node	32GB	24GB
Interconnect	Gemini interconnect	Dual QDR InfiniBand
O/S	CLE 3.1.29	RHEL 5.6
Compilers	PGI CC 11.3 Cray MPI	ICC 11.1 Intel MPI 3.1
Compiler flags	-Minline=levels:10 -Mipa=fast	-O2 -xSSE4.2

priate partitioner routine. Once a partitioning of this primary set is achieved, OP2 infers the MPI process to which all other secondary set elements should belong using the available mappings and migrates data and mappings of all set elements to the new MPI processes. Finally it renumbers the mappings with the new element indices. Halo creation occurs only after these steps have been completed by all the MPI processes.

3. PERFORMANCE

Our first set of experiments are directed at benchmarking and analyzing the performance of the new OP2 MPI layer. The example application, Airfoil, used in our analysis is a non-linear 2D inviscid airfoil code developed using the OP2 API. The performance of Airfoil was previously explored on single multi-core CPU and GPU nodes [13, 10]. In our current analysis, we use the same mesh consisting of over 720K nodes, 720K cells and about 1.5 million edges to investigate the performance on distributed memory cluster systems. In section 4 we analyze the performance of a larger mesh (26 million edges) using performance modeling techniques. The code consists of five parallel loops: `save_soln`, `adt_calc`, `res_calc`, `bres_calc` and `update`. `save_soln` and `update` are direct loops while the other three are indirect loops. The application has four sets declared (`nodes, edges, bedges`, and `cells`) and five mapping tables that define the connectivity between these sets. Additionally, there are four data arrays defined on `cells` (`p_q, p_qold, p_adt` and `p_res`), a data array consisting of the xy coordinates defined on the `nodes` (`p_x`) and a data array with the boundary data defined on the `bedges` (`p_bound`).

Table 2 notes the key specification of the cluster systems that were used for benchmarking. The first system, HECToR [2], is a large-scale proprietary Cray XE6 system which we use to investigate the scalability of the MPI implementation. The second system, CX1, is a small commodity Westmere/InfiniBand cluster that we use to compare performance with in the next section. All the results presented are taken during the execution of the application in double-precision floating-point arithmetic.

We begin our analysis by investigating the performance of Airfoil, at scale. Figure 3 presents the strong-scaled run-times of the application on HECToR for up to 1920 cores. The run-times given here are averaged from 5 runs for each processor core count. The standard deviation in run times were significantly less than 10% and thus we limited the number of times that each test was repeated to save time on the system. Halo creation and mesh partitioning in total takes less than 10 seconds at 1920 cores and is not included in the runtime. In Airfoil, the only loop that exchanges halos during an iteration is `res_calc`. The block partitioning method splits the sets, data on sets and mappings equally on to P number of processor cores. However, as expected the best run-times were given by the kway partitioned executions with PTScotchKway giving marginally better performance. The best run-time overall is 1.12 seconds at 960 processor cores. The application scales well on up to 960 cores and then becomes limited by

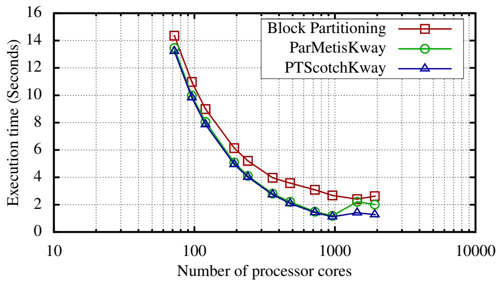

Figure 3: Airfoil on HECToR (1000 iterations)

the amount of data per partition (i.e. per MPI process), the amount of redundant computation (due to the *ieh*) and the communication time spent during halo exchanges. A larger mesh, as we will show in the next section, will continue to scale up to more processors. We also observed up to 30% performance gains (with k-way partitioning) due to the use of non-blocking communications overlapped with computation during halo exchanges.

Table 3: Airfoil run-times comparison

Node System	Cores /node (Clk/core)	Mem /node	Run-time (seconds)
2×Intel Xeon X5650 (Westmere)	12 (24 SMT) (2.67GHz)	24 GB	37.89 (24 OMP)
2×AMD Opteron 6128 (Magny Cours)	16 (2.0GHz)	12 GB	46.30 (16 OMP)
GeForce GTX560Ti	384 (1.6GHz)	1.0 GB	19.63
Tesla C2070	448 (1.15GHz)	6.0 GB	13.20
HECToR	72 cores		13.22
	480 cores		2.09
	960 cores		1.12
CX1	36 cores		20.66
	60 cores		12.29
	120 cores		6.07

Our final set of benchmark results in this section compares the best MPI performance to that of the OP2 single node performance. Table 3 presents best run-times from four single node systems, HECToR and CX1. The GPUs execute NVIDIA CUDA code generated by OP2, while the Westmere and Opteron multi-core CPUs utilize OpenMP generated by OP2. Both GPUs were running the latest NVIDIA driver (version 4.0) with a compute capability of 2.1 with ECC switched off. The best run-times for the single node systems were gained using parameter configurations (mini-partition size and thread-block size [13, 10]) discovered through an auto-tuning framework [11]. The best run-time with OpenMP (37.89 seconds) on the Westmere processor node (compiled with icc -O2 -xSSE4.2) was obtained by executing 24 OpenMP threads on 12 symmetric multi-threading (SMT) enabled cores.

On the Opteron processor node, the best run-time (46.30 seconds) was gained with 16 OpenMP threads (compiled with icc -O2 -ipo -xSSE2 -funroll-loops). The GPU results give a best runtime of 19.63 seconds and 13.20 seconds on the GTX560Ti and the Tesla C2070 respectively. These are speed-ups of 1.93× and 2.8× respectively, compared to the Intel Westmere processor node with 24 OpenMP threads on 12 cores. In contrast we see that the performance of Airfoil on HECToR with 72 cores (3 nodes) and CX1 with 60 cores (5 nodes) is approximately equivalent to the performance of one C2070 GPU. It is surprising to see that some performance gains can be achieved even on a single consumer-grade GTX560Ti (equivalent to approximately 36 Westmere cores) for this application.

4. PREDICTIVE ANALYSIS

The significant speedups gained from utilizing a single GPU, led us to explore whether these are maintainable on a cluster of GPUs across a distributed memory environment. During a distributed memory execution, for message passing between compute nodes via the network, halos need to be copied to

$$
\begin{aligned}
T_{iter} &= T_{ss} + 2(T_{ac} + T_{rc} + T_{brc} + T_u) && (1)\\
T_{ss} &= w_{g,ss} \times n_{cells} && (2)\\
T_{ac} &= w_{g,ac} \times n_{cells} && (3)\\
T_{rc} &= max(w_{g,rc} \times n_{core,edges}, T_{comm,rc}) + \\
&\quad w_{g,rc} \times (n_{ieh,edges} + n_{eeh,edges}) && (4)\\
T_{brc} &= w_{g,brc} \times (n_{bedges} + n_{ieh,bedges}) && (5)\\
T_u &= w_{g,u} \times n_{cells} + T_{reduce} && (6)\\
T_{comm,rc} &= (n_{ieh,cells} + n_{inh,cells}) \times 8B \times \\
&\quad (esize_{p\_q} + esize_{p\_adt}) + 2LN_{avg,cells} + \\
&\quad L_{on\_chip} \times CN_{avg,cells} && (7)
\end{aligned}
$$

Figure 4: Performance model for CPU cluster

the GPU global memory from the node's main memory (and vice versa) over the PCIe bus. A key concern for many hybrid CPU-GPU application developers is to identify whether there is any performance degradation due to the InfiniBand bandwidth available per GPU and the need to frequently copy messages over the relatively slow PCIe bus. The transfer over the PCIe will remain even with MPI implementations adopting NVIDIA's new GPUDirect [1] technology for communicating directly between GPUs without the involvement of the host CPU. As advanced support for extending OP2's MPI back-end layer implementation to GPU clusters and to understand the performance implications of such a parallelization, in this final section we utilize performance modeling to investigate an "ahead of implementation" performance analysis of the Airfoil code for executing on a cluster of GPUs.

We first develop an analytic performance model for the Airfoil application running on both HECToR and CX1, using techniques similar to those published in [15, 8]. The time taken by an iteration of the Airfoil application can be modeled as the critical path time taken by the five parallel loops. Out of these loops, `save_soln` is called once, while the others are called twice per iteration. Thus the total time for an iteration can be given by (1), where T_* are the time taken by each of the parallel loops per call. As only `res_calc` performs any halo exchanges, all other loops only involves computation over a given set (and possibly the *ieh*). In addition to this, `update` includes a global operation which results in a single MPI_Reduce per loop call. Given the time to compute over a single set element ($w_{g,*}$), we can express the times taken by each loop, as (2),(3),(4),(5) and (6). The terms n_* represents the number of elements that computation will be performed over. `save_soln`, `adt_calc` and `update` loops over `cells` while `res_calc` loops over `edges` and `bres_calc` loops over `bedges`. The *max* term in (4) accounts for the overlapping of computation and communications. If blocking communications are used then the two terms should be summed. As per our halo and set element classification in Section 2.1 any element that belongs to the *eeh* on an MPI process cannot be computed over until the halo exchanges are complete. The $w_{g,*}$ costs were measured on a processor core and are assumed to remain constant as the global mesh is solved at increasing machine scale. Our experience so far has been that on both HECToR and CX1 these computation per element times remain fairly constant at increasing scale.

Given the average number of MPI neighbors ($N_{avg,cells}$) that a process will communicate with during the `res_calc` loop and the size of the halos, the communication time can be modeled as (7). B is the time taken by the communication network to transfer a byte of data from one processor to another. Thus $1/B$ gives the bandwidth of the network. L is the latency associated with communicating a message with a neighbor. To account for the critical path time during message passing, we use the off-node message communication times. We double the latency term as there are two data arrays being exchanged. The $esize_*$ gives the size of an element

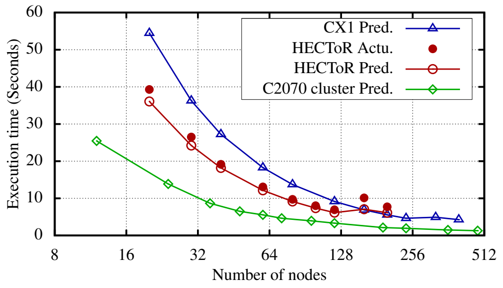

Figure 5: Airfoil - 26M edge mesh

(i.e. number of double precision values per set element) for each data array. The 8 multiplier accounts for the size of a double precision floating-point value on the system. C is the number of cores that share a NIC (12 cores share a NIC in HECToR [2] and CX1). We assume that some serialization of MPI messages is caused at the NIC during message passing [8, 15] and approximate it as the latency for communicating a message within a node (L_{on_chip}) multiplied by the average number of MPI messages sent simultaneously. The values for B, L and L_{on_chip} were found by benchmarking the end-to-end message transfer time between two nodes (and two cores) for a range of message sizes. The time for a reduce operation T_{reduce} was approximately modeled as a tree gather operation [15].

To extend the above homogeneous multi-core CPU cluster model to that of a GPU cluster model requires us to consider the additional costs involved during MPI operation over GPUs. Such techniques have been previously used for predicting GPU cluster performance with high accuracy [16]. For this paper we develop the GPU cluster model for Airfoil assuming a cluster of NVIDIA C2070 GPUs that is interconnected by an InfiniBand network with similar performance to that of CX1. Computation times for each loop were benchmarked on a single C2070 GPU for various mesh sizes. This gives us approximate times for the GPU to execute a given number of set elements belonging to its local partition. The communication time for `res_calc` in (7) was augmented with PCIe bandwidths and latencies (measured using the NVIDIA CUDA SDK's bandwidthTest benchmark, and a custom latency benchmark) to copy halo data to and from the GPU. Our measurements indicated a host to device PCIe bandwidth and latency of approximately 3700 MB/sec and $9\mu S$ respectively. The device to host bandwidth and latency was 3130 MB/sec and $11\mu S$. Assuming that each C2070 has exclusive access to a NIC we remove the serialization costs terms from (7). The current model does not taken into consideration the possible performance gains with NVIDIA's new GPUDirect [1] technology. However, the model could be easily modified to account for this case by updating the times for the halo copies between GPUs.

To investigate the the application's scalability on the GPU cluster, we benchmark and project performance for solving a larger 26 million edge mesh. Figure 5 projects the performance (run-time vs. nodes) of Airfoil solving this mesh on both CX1 and the hypothetical C2070 GPU cluster. Note that HECToR has 24 cores/node, CX1 has 12 cores/node and C2070 cluster - 1 GPU/node. Actual run times from HECToR are also provided to demonstrate the model accuracy. For most runs model accuracy exceeds 90% but is more sensitive to the system communication performance at large scale. However the model accurately predicts the number of cores that gives the optimum runtime and the qualitative trend in scaling on HECToR, allowing us to establish the limits of scalability. The model predicts, for example, a cluster with 36 C2070 GPUs to give equivalent performance to that of over 1920 HECToR cores (80 nodes) or a Westmere/InfiniBand cluster with 1440 cores (120 nodes). Thus, we see a C2070 cluster to give the same performance that is equivalent to performance given by traditional homogeneous clusters that are more than three times its size. However this should be considered in the context of the amount of available memory on a GPU to hold and execute the required partition size.

On HECToR and CX1 we see that the increase in redundant computations due to *ieh* at large scales degrades performance. The runtime at 160 HECToR nodes and 320 CX1 nodes was particularly affected by a large *ieh*. However, due to one C2070 GPU handling one MPI process on the GPU cluster the model predicts a much smoother performance curve. The C2070 cluster scales up to approximately 128 nodes after which the performance plateaus. The model indicates that the PCIe overheads and MPI/InfiniBand overheads that are not hidden due to using non-blocking communications account for less than 10% of the total run-time up to about 480 GPUs. We expect that the communication overheads to further decrease with the use of the faster PCIe3 bus and the upcoming GPUDirect implementation over InfiniBand fabrics. Thus we predict that effects on performance and scalability due to such overheads will be further diminished for these applications. The limitations to scalability will be more prominent due to the amount of parallelism available to be exploited at a finer-grained thread level within a GPU node in a large-scale execution.

5. RELATED WORK

There are many established conventional libraries (e.g. PETSc [4], Sierra [17]) supporting unstructured mesh based application development on traditional clusters. In contrast, OP2's objective is to support multiple back-ends, particularly based on emerging multi-core/many-core technologies, without the intervention of the application programmer. OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [14] 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. A number of research projects have implemented similar or related programming frameworks. The most comparable of these projects is LISZT [9] from Stanford University. While OP2 uses an "active" library approach utilizing code transformation, LISZT implements a domain specific language (embedded in the Scala language) 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 [9]. The Navier-Stokes application in [9] is most comparable to the Airfoil application and show 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 [9] and is noted as future work.

6. CONCLUSION

The Airfoil application benchmarked in this paper show roughly up to 3× speedup on current flagship GPUs compared to two equivalent multi-core x86 CPUs. Our experiments show that such speedups can be achieved on both single node CPUs utilizing thread-level parallelism (OpenMP) as well as traditional homogeneous distributed memory clusters.

On a heterogeneous cluster system, we expect such applications to exhibit similar performance gains given that the individual GPU nodes do not exhaust their resources solving a given workload. On the other hand, our performance modeling predicts that the limiting factor in scalability is not primarily the PCIe or MPI/InfiniBand overheads, particularly when non-blocking operations are used to hide communication costs. Scalability is affected more by the amount of parallelism available per partition to be exploited by each GPU.

The results from the predictive modeling study in this paper provide us with important quantitative and qualitative insights into the achievable performance on heterogeneous cluster systems. The model itself is a representative building block for future performance projections for applications developed using OP2. The full OP2 source and the Airfoil test case code are available as open source software [11] and the developers welcome new participants in the OP2 project.

7. REFERENCES

[1] GPUDirect. `http://developer.nvidia.com/gpudirect`.

[2] HECToR. `http://www.hector.ac.uk/service/hardware/`.

[3] ParMETIS. `http://www.cs.umn.edu/~metis`.

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

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

[6] Top500 Systems, June 2011. `http://www.top500.org/list/`.

[7] Burgess, D. A., Crumpton, P. I., and Giles, M. B. A Parallel Framework for Unstructured Grid Solvers. In *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.

[8] Davis, J., Mudalige, G., Hammond, S., Herdman, J., Miller, I., and Jarvis, S. Predictive analysis of a hydrodynamics application on large-scale cmp clusters. In *International Supercomputing Conference (ISC11)*, vol. 26 of *Lecture Notes in Computer Science (R&D)*. Springer, Hamburg, Germany, June 2011, pp. 175–185.

[9] 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* (2011).

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

[11] Giles, M. B. OP2 for Many-Core Platforms, May 2011. `http://people.maths.ox.ac.uk/gilesm/op2/`.

[12] Giles, M. B., Ghate, D., and Duta, M. C. Using automatic differentiation for adjoint CFD code development. *Computational Fluid Dynamics Journal 16*, 4 (2008), 434–443.

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

[14] Howes, L. W., Lokhmotov, A., Donaldson, A. F., and Kelly, P. H. J. Deriving efficient data movement from decoupled access/execute specifications. In *High Performance Embedded Architectures and Compilers*, A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, and T. Ungerer, Eds., vol. 5409 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 2009, pp. 168–182.

[15] Mudalige, G., Vernon, M., and Jarvis, S. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (April, 2008), IEEE Computer Society.

[16] Pennycook, S. J., Hammond, S. D., Jarvis, S. A., and Mudalige, G. R. Performance analysis of a hybrid MPI/CUDA implementation of the NAS-LU benchmark. *SIGMETRICS Performance Evaluation Review 38*, 4 (2011), 23–29.

[17] 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.