Optimising Hydrodynamics applications for the Cray XC30 with the application tool suite

W. P. Gaudin*, A. C. Mallinson†, O. Perks*, J. A. Herdman*, D. A. Beckingsale†, J. M. Levesque‡, M. Boulton§, S. McIntosh-Smith§ and S. A. Jarvis†
*High Performance Computing, Atomic Weapons Establishment, Aldermaston, UK
Email: {Wayne.Gaudin,Oliver.Perks,Andy.Herdman}@awe.co.uk
†Performance Computing and Visualisation, Department of Computer Science, University of Warwick, UK.
Email: {acm,dab}@dcs.warwick.ac.uk
‡CTO Office, Cray Inc, Knoxville, TN, USA. Email: levesque@cray.com
§Applied Computing, Dpt. of Computer Science, University of Bristol, UK. Email: sm@dcs.bristol.ac.uk

***Abstract*—Power constraints are forcing HPC systems to continue to increase hardware concurrency. Efficiently scaling applications on future machines will be essential for improved science and it is recognised that the "flat" MPI model will start to reach its scalability limits. The optimal approach is unknown, necessitating the use of mini-applications to rapidly evaluate new approaches. Reducing MPI task count through the use of shared memory programming models will likely be essential. We examine different strategies for improving the strong-scaling performance of explicit Hydrodynamics applications. Using the CloverLeaf mini-application across multiple generations of Cray platforms (XC30, XK6 and XK7), we show the utility of the hybrid approach and document our experiences with OpenMP, CUDA, OpenCL and OpenACC under both the PGI and CCE compilers. We also evaluate Cray Reveal as a tool for automatically hybridising HPC applications and Cray's MPI rank to network topology-mapping tools for improving application performance.**

***Keywords*-Exascale, HPC, Hydrodynamics, MPI, OpenMP, OpenACC, CUDA, OpenCL, Tools**

I. Introduction

Power constraints are forcing emerging HPC systems to continue to increase hardware concurrency in order to enhance floating point performance with decreased energy use. With CPU clock-speeds constant or even reducing, node level computational capabilities are improving through increasing core and thread counts. We must be able to efficiently utilise this increased concurrency if we are to harness the power of future systems. The ability to scale applications across multi-petascale platforms is also essential if we are to use this class of machine for improved science. Irrespective of the nodal hardware, there is a common need for scalable communication mechanisms within future systems.

The established method of utilising current platforms employs a "flat" MPI model for intra- and inter-node communications. This model takes no regard of the shared memory available at the node level within the application. This approach has served the scientific community well for 20 years, however, with increasing nodal core counts it is anticipated that it will reach its scalability limits due to congestion caused by the number of MPI tasks across a large distributed simulation. Additionally, on machines incorporating GPU accelerator technologies "flat" MPI is not a viable solution and precludes the use of them without transition to a hybrid model.

The optimal approach is, however, unknown and probably application and machine specific. Production codes are usually legacy codes which are generally old, large, unwieldy and inflexible, with many man years of development that cannot be discarded and rewritten from scratch for the latest hardware. Decisions made now could severely affect scientific productivity if the path chosen is not amenable to future hardware platforms. Research is needed to assess the relative merits of new programming models and hardware evolution in order to retain scientific productivity. Attempting this in a fully functional legacy code has been found to be time consuming and impractical, due to the number of potential solutions available. A rapid, low risk approach for investigating the solution space is therefore extremely desirable.

The approach we adopt is based on the use of a mini-application or mini-app. Mini-apps are small, self-contained codes, which emulate key algorithmic components of much larger and more complex production codes. Their use enables new methods and technologies to be rapidly developed and evaluated.

In this work we utilise a simplified but meaningful structured, explicit hydrodynamic mini-app called CloverLeaf [1] to investigate the optimal configurations for achieving portable performance across a range of Cray machines. We document our experiences hybridising the existing MPI code base with OpenMP, OpenACC, CUDA and OpenCL and compare the performance under both the PGI and CCE compilers for the OpenACC implementation.

OpenACC is a high level pragma based abstraction intended to provide support for multi-core technology for Fortran, C and C++ without resorting to vendor and hardware specific low level languages. Support is provided by a number of vendors and is defined through an open standard. However the standard is new and the compiler implementations are still maturing. This paper evaluates the maturity of two compilers by assessing two approaches to the OpenACC programming model, namely the `Parallel` and `Kernel` constructs. We then assess the performance and portability of the CloverLeaf mini-app across two Cray platforms. These are in turn compared against CUDA and OpenCL on each architecture to determine whether OpenACC provides a level of abstraction that is suitable for enabling existing large code bases to exploit emerging multi-core architectures, while maintaining scientific productivity.

Weak- and strong-scaling scenarios are important on multi-petascale machines. The former are likely to scale well as the ratio of communication to computation remain close to constant. While the latter is more challenging because the amount of computation per node reduces with increased scale and communications eventually dominate. Historically, the use of shared memory has not been vital for this class of application and treating each core/thread as a separate address space has been a valid strategy. If the expected explosion in thread counts materialises, reducing MPI task count using shared memory programming models is a strategy that needs research.

In this work we examine hybrid programming models for improving the strong-scaling performance of explicit hydrodynamics applications on the XC30 Cray platform. Using the "flat" MPI version as a baseline, we measure the relative performance against a hybrid version incorporating OpenMP constructs.

The task of developing, porting and optimising applications for future generations of HPC systems is becoming increasingly complicated as architectures evolve. The analysis of legacy applications in order to convert them to hybrid models is non-trivial. Even with an in-depth knowledge of the algorithm and target hardware, extracting the maximum concurrency is a difficult task. Improving the tool-suite available to developers will be essential if optimal performance is to be achieved productively. We therefore evaluate Cray's Reveal tool as a technology for improving this situation by automatically hybridising the "flat" MPI version of CloverLeaf and compare its performance to that of a hand-crafted MPI+OpenMP implementation. We also comment on developing an OpenMP code as the basis for an OpenACC one.

The increased scale and complexity of modern interconnects is also forcing us to question the placement of MPI ranks within the network for optimal performance. With the help of Cray's profiling and network topology mapping tools we explore the potential for optimisation based on application communication characteristics.

II. Related Work

A considerable body of work exists which has examined the advantages and disadvantages of the hybrid (MPI+OpenMP) programming model compared to the purely "flat" MPI model. A number of studies have already examined the technology but these have generally focused on different scientific domains, classes of applications, or different hardware platforms, to those we examine here.

The results from these studies have also varied significantly, with some authors achieving significant speed-ups by employing hybrid constructs and others performance degradations. Substantially less work exists which directly evaluates the MPI and hybrid programming models when applied to the same application. We are also unaware of any work which has directly compared approaches based on OpenACC, OpenCL and CUDA for implementing the hybrid programming model on systems incorporating accelerator devices. Our work is motivated by the need to examine each of these programming models when applied to Lagrangian-Eulerian explicit hydrodynamics applications. In previous work Mallinson reported on his experiences scaling the CloverLeaf mini-application large node counts on the Cray XE6 and XK7 architectures [2].

Additionally we have also previously reported on our experiences of porting CloverLeaf to a range of different architectures, both CPU and GPU based using CUDA, OpenACC and OpenCL, but only at small scale [3], [4]. Although we are not aware of any existing work which has examined using these technologies to scale this class of application to the levels we examine here, Levesque et al. examine using OpenACC at extreme scale with the S3D application [5].

The application examined by Lavallee et al. has similarities to CloverLeaf, and their work compares several hybrid approaches against a purely MPI based approach, however, they focus on a different hardware platform and do not examine the OpenACC-based approaches [6].

Studies such as [7]–[9] report performance degradations when employing hybrid (MPI+OpenMP) based approaches, whilst others experience improvements [10]–[12]. In particular, Környei presents details on the hybridisation of a combustion chamber simulation which employs similar methods to CloverLeaf. However, the application domain and the scales of the experiments are significantly different to those in our study.

Drosinos et al. also present a comparison of several hybrid parallelisation models (both coarse- and fine-grained) against the "flat" MPI approach [13]. Again, their work focuses on a different class of application, at significantly lower scales and on a different experimental platform to our work.

Nakajima compares the hybrid programming model to "flat" MPI for preconditioned iterative solver applications within the linear elasticity problem space [14]. Whilst the application domain; the scales of the experiments ($<$512 PEs) and platform choice (T2K HPC architecture) are again significantly different to ours, he does, as we do, explore several techniques for improving the performance of these applications.

Additionally, Adhianto et al. discuss their work on performance modelling hybrid MPI+OpenMP applications and its potential for optimising applications [15]. Li et al. employ the hybrid approach in their work which examines how to achieve more power-efficient implementations of particular benchmarks [16]. Various approaches and optimisations for executing large-scale jobs on Cray platforms are also examined by Yun et al. in [17]. Minimising communication operations within applications has also been recognised as a key approach for improving the scalability and performance of scientific applications [16].

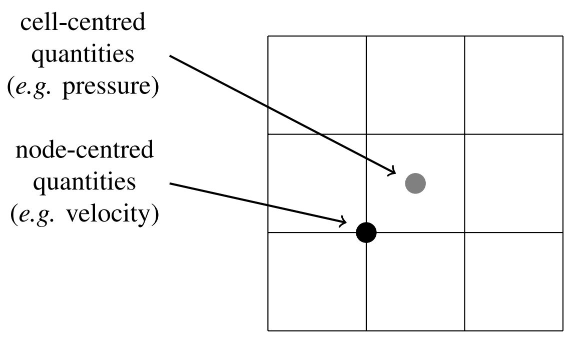

cell-centred quantities (*e.g.* pressure)
node-centred quantities (*e.g.* velocity)

Fig. 1: The *staggered grid* used by CloverLeaf

Baker et al. [18] looked at a hybrid approach using OpenACC within the BT-MZ benchmark application. They concentrate however on hybridising the application rather than on assessing the OpenACC implementations employed, their results are also focused on a single architecture: Titan.

There are a small number of comparative studies presenting direct comparisons of OpenACC against alternative programming models. Reyes et al. present a direct comparison between CUDA, PGIs Accelerator model and OpenACC using their own novel implementation of OpenACC: accULL [19]. Again, this focuses on a single type of accelerator, and a single instance of an architecture: an NVIDIA Tesla 2050.

A comparison of OpenCL against OpenACC can be found in [20] by Wienke et al. The paper compares OpenACC against PGI Accelerator and OpenCL for two real world applications, demonstrating OpenACC can achieve 80% of the performance of a best effort OpenCL for moderately complex kernels, dropping to 40% for more complex examples. The study only uses Crays CCE compiler and only the parallel construct. Also, it is limited to a single hardware architecture: an NVIDIA Tesla C2050 GPU.

III. Background

In this section we provide details on the hydrodynamics scheme employed in CloverLeaf, and an overview of the programming models examined in this study.

A. Hydrodynamics Scheme

CloverLeaf is part of the Mantevo test suite [1]. It has been written with the purpose of assessing new technologies and programming models both at the node and interconnect level.

CloverLeaf uses a Lagrangian-Eulerian scheme to solve Euler's equations of compressible fluid dynamics in two spatial dimensions. These are a system of three partial differential equations which are mathematical statements of the conservation of mass, energy and momentum. A fourth auxiliary equation of state is used to close the system; CloverLeaf uses the ideal gas equation of state to achieve this.

The equations are solved on a staggered grid (see Figure 1) in which each cell centre stores the three quantities: energy, density, and pressure; and each vertex stores a velocity vector. This is a hyperbolic system which allows the equations to be solved using explicit numerical methods, without the need to invert a matrix. An explicit finite-volume method is used to solve the equations with second-order accuracy in space and time. Currently only single material cells are simulated by CloverLeaf.

The solution is advanced forward in time repeatedly until the desired end time is reached. Unlike the computational grid, the solution in time is not staggered, with both the vertex and cell data remaining at the same time level by the end of each computational step. One iteration, or timestep, of CloverLeaf proceeds as follows: (i) a Lagrangian step advances the solution in time using a predictor-corrector scheme, with the cells becoming distorted as the vertices move due to the fluid flow; (ii) an advection step restores the cells to their original positions by moving the vertices back, and calculating the amount of material that has passed through each cell face. This is accomplished using two sweeps, one in the horizontal dimension and the other in the vertical using Van Leer advection [21]. The direction of the initial sweep in each step alternates in order to preserve second order accuracy.

The computational mesh is spatially decomposed into rectangular mesh chunks and distributed across MPI tasks within the application, in a manner which attempts to minimise the communication surface area between processes. Data that is required for the various computational steps and is non-local to a particular process is stored in outer layers of halo cells within each mesh chunk. Halo data exchanges occur between immediate neighbouring processes (vertically and horizontally), within the decomposition. A global reduction operation is required by the algorithm during the calculation of the minimum stable timestep, which is calculated once per iteration. The initial implementation was in Fortran90 and this was used as the basis for all other versions.

The computational intensity per memory access of CloverLeaf is low. This makes the code limited by memory bandwidth speeds, which is a very common property of scientific codes.

B. Programming Models

1) MPI: As cluster-based designs have become the predominant architecture for HPC systems, the Message Passing Interface (MPI) has become the standard for developing parallel applications for these platforms. Standardised by the MPI Forum, the interface is implemented as a parallel library alongside existing sequential programming languages [22].

The technology is able to express both intra- and inter-node parallelism. Current implementations generally use optimised shared memory constructs for communication within a node and explicit message passing for communication between nodes. Communications are generally two-sided meaning that all ranks involved in the communication need to collaborate in order to complete it.

2) OpenMP: OpenMP is an Application Program Interface (API) and has become the de facto standard in shared memory programming [23]. The technology is supported by all the major compiler vendors and is based on a fork-join model of concurrency, it consists of a set of pragmas that can be added to existing source code to express parallelism. An OpenMP-enabled compiler is able to use this additional information to parallelise sections of the code.

Programs produced from this technology require a shared memory-space to be addressable by all threads. Thus, this technology is aimed primarily at implementing intra-node parallelism. At present the technology only supports CPU-based devices although proposals exist in OpenMP 4.0 for the inclusion of additional directives to target accelerator based devices such as GPUs [24]. This has been implemented to varying levels in a number of compilers, but is not yet mature.

3) CUDA: NVIDIA's CUDA [25] is currently a well established technology for enabling applications to utilise NVIDIA GPU devices. CUDA employs an offload-based programming model in which control code, executing on a host CPU, launches parallel portions of an application (kernels) on an attached GPU device.

CUDA kernels are functions written in a subset of the C programming language, and are comprised of an array of lightweight threads. Subsets of threads can cooperate via shared memory which is local to a particular multiprocessor, however, there is no support for global synchronisation between threads. This explicit programming model requires applications to be restructured in order to make the most efficient use of the GPU architecture and thus take advantage of the massive parallelism inherent in them. Constructing applications in this manner also enables kernels to scale up or down to arbitrary sized GPU devices.

CUDA is currently a proprietary standard controlled by NVIDIA. Whilst this allows NVIDIA to enhance CUDA quickly and enables programmers to harness new hardware developments in NVIDIA's latest GPU devices, it does have application portability implications.

4) OpenCL: OpenCL [26] is an open standard that enables parallel programming of heterogeneous architectures. Managed by the Khronos group and implemented by over ten vendors—including AMD [27], Intel [28], IBM [29], and Nvidia [30]—OpenCL code can be run on many architectures without recompilation. Each compiler and runtime is, however, at a different stage of maturity, so performance currently varies between vendors.

The programming model used by OpenCL is similar to NVIDIA's CUDA model. Therefore, mapping OpenCL programs to GPU architectures is straightforward. The best way to map OpenCL programs to CPU architectures, however, is less clear.

The OpenCL programming model distinguishes between a *host* CPU and an attached accelerator *device* such as a GPU. The host CPU runs code written in C or C++ that makes function calls to the OpenCL library in order to control, communicate with, and initiate tasks on one or more attached devices, or on the CPU itself. The target device or CPU runs functions (*kernels*) written in a subset of C99, which can be compiled just-in-time, or loaded from a cached binary if one exists for the target platform. OpenCL uses the concepts of *devices*, *compute units*, *processing elements*, *work-groups*, and *work-items* to control how OpenCL kernels will be executed by hardware. The mapping of these concepts to hardware is controlled by the OpenCL runtime.

Generally, an OpenCL device will be an entire CPU socket or an attached accelerator. On a CPU architecture, both the compute units and processing elements will be mapped to the individual CPU cores. On a GPU this division can vary, but compute units will typically map to a core on the device, and processing elements will be mapped to the functional units of the cores.

Each kernel is executed in a Single Program Multiple Data (SPMD) manner across a one, two or three dimensional range of work-items, with collections of these work-items being grouped together into work-groups. Work-groups map onto a compute unit and the work-items that they contain are executed by the compute unit's associated processing elements. The work-groups which make up a particular kernel can be dispatched for execution on all available compute units in any order. On a CPU, the processing elements of the work-group will be scheduled across the cores using a loop. If vector code has been generated, the processing elements will be scheduled in SIMD, using the vector unit of each CPU core. On a GPU, the processing-elements run work-items in collections across the cores, where the collection size or width depends on the device vendor; NVIDIA devices run work-items in collections of 32 whereas AMD devices use collections of 64 work-items.

OpenCL is therefore able to easily express both task and data parallelism within applications. The OpenCL programming model provides no global synchronisation mechanism between work-groups, although it is possible to synchronise within a work-group. This enables OpenCL applications to scale up or down to fit different hardware configurations.

5) OpenACC: The OpenACC [31] Application Program Interface is a high-level programming model based on the use of pragmas. Driven by the Center for Application Acceleration Readiness (CAAR) team at Oak Ridge National Laboratory (ORNL) [32] and supported by an initial group of three compiler vendors. The aim of the technology is to enable developers to add directives into their source code to specify how portions of their applications should be parallelised and off-loaded onto attached accelerator devices, thus minimising the modifications required to existing code bases in Fortran, C and C++ and easing programmability whilst also providing a portable, open standards-based solution for multi-core technologies. It provides an approach to coding complicated technologies without the need to learn complex, vendor specific, languages, or understand the hardware at the deepest level. Portability and performance are the key features of this programming model, which are essential to productivity in a real scientific application.

Prior to the support of a common OpenACC Standard, Cray, PGI and CAPS each had their own bespoke set of accelerator directives from which their implementations of OpenACC were derived.

The PGI 10.4 release supported the PGI Accelerator model [33] for NVIDIA GPUs. This provided their own bespoke di-

```
!$OMP PARALLEL
!$OMP DO PRIVATE(v, pressurebyenergy , &
  pressurebyvolume , sound_speed_squared )
  DO k = y_min , y_max
    DO j = x_min , x_max

      p(j ,k) = (1.4 − 1.0)∗d(j ,k)∗e(j ,k)
      pe = (1.4 − 1.0)∗d(j ,k)
      pv = −d(j ,k)∗p(j ,k)
      v = 1.0/d(j ,k)
      ss2 = v∗v∗(p(j ,k)∗pe−pv)
      ss(j ,k)=SQRT(ss2)

    END DO
  END DO
!$OMP END DO
!$OMP END PARALLEL
```

Fig. 2: CloverLeaf's `ideal_gas` kernel

rectives for acceleration of regions of source code. In particular their `region` construct evolved into their implementation of the OpenACC `Kernel` construct. CUDA is generated from the OpenACC code which then uses NVCC to generate a GPU ready executable. There are plans to support alternative backends to allow the targeting of a wider range of hardware.

Cray originally proposed accelerator extensions to the OpenMP standard [34] to target GPGPUs, through their Cray Compilation Environment (CCE) compiler suite. These evolved into the `Parallel` construct in the OpenACC standard. Rather than creating CUDA source for the kernels, CCE translates them directly to NVIDIAs low-level Parallel Thread Execution (PTX) programming model [35]. CCE is currently only available on Cray architectures.

Initially, CAPS provided support for the OpenHMPP directive model [36], which served as their basis for the OpenACC standard. A major difference with CAPS is the necessity to use a host compiler. Code is directly translated into either CUDA or OpenCL [26] which in case of the latter, opens up a wide range of architectures which can be targeted.

All compilers now support both the `Kernel` and `Parallel` constructs. The main differences between these constructs relate to how they map the parallelism in the particular code region, which is being accelerated, to the underlying hardware. The `Parallel` construct is explicit, requiring the programmer to additionally highlight loops for parallelisation within the code region, it closely resembles several OpenMP constructs, while with the `Kernel` construct the parallelisation is carried out implicitly. Example code is shown in Figure 3 for the same code fragment using both constructs.

IV. IMPLEMENTATION

The computational intensive sections of CloverLeaf are implemented via fourteen individual kernels. In this instance, we use "kernel" to refer to a self contained function which carries out one specific aspect of the overall hydrodynamics algorithm. Each kernel iterates over the staggered grid, updating the appropriate quantities using the required stencil operation. Figure 2 shows the Fortran code for one of these

```
!$ACC DATA &
!$ACC PRESENT( density , energy , pressure , soundspeed )
!$ACC PARALLEL LOOP PRIVATE(v, pressurebyenergy ,
!$ACC        pressurebyvolume , sound_speed_squared )
             VECTOR_LENGTH(1024)
  DO k = y_min , y_max
    DO j = x_min , x_max

      p(j ,k) = (1.4 − 1.0)∗d(j ,k)∗e(j ,k)
      pe = (1.4 − 1.0)∗d(j ,k)
      pv = −d(j ,k)∗p(j ,k)
      v = 1.0/d(j ,k)
      ss2 = v∗v∗(p(j ,k)∗pe−pv)
      ss(j ,k)=SQRT(ss2)

    END DO
  END DO
!$ACC END PARALLEL LOOP
!$ACC END DATA
```

(a) Using the OpenACC Parallel constructs

```
!$ACC DATA &
!$ACC PRESENT( density , energy , pressure , soundspeed )
!$ACC KERNELS
!$ACC LOOP INDEPENDENT
  DO k=y_min , y_max
!$ACC LOOP INDEPENDENT PRIVATE(v, pressurebyenergy ,
!$ACC           pressurebyvolume , sound_speed_squared )
    DO j=x_min , x_max
      v=1.0_8/density(j ,k)
      pressure(j ,k)=(1.4_8−1.0_8)∗density(j ,k)∗energy(j ,k)
      pressurebyenergy=(1.4_8−1.0_8)∗density(j ,k)
      pressurebyvolume=−density(j ,k)∗pressure(j ,k)
      sound_speed_squared=v∗v∗(pressure(j ,k)
                              ∗pressurebyenergy−pressurebyvolume)
      soundspeed(j ,k)=SQRT(sound_speed_squared)
    ENDDO
  ENDDO
!$ACC END KERNELS
!$ACC END DATA
```

(b) Using the OpenACC Kernel constructs

Fig. 3: The `ideal_gas` kernel OpenACC implementations

kernels. The kernels contain no subroutine calls and avoid using complex features like Fortran's derived types, making them ideal candidates for evaluating alternative approaches such as OpenMP, CUDA, OpenCL or OpenACC. They have minimised dependencies, allowing them to be driven independently of the host code.

Not all the kernels used by CloverLeaf are as simple as the example in Figure 2. However, during the initial development of the code, the algorithm was engineered to ensure that all loop-level dependencies within the kernels were eliminated and data parallelism was maximised. Most of the dependencies were removed via small code rewrites: large loops were broken into smaller parts; extra temporary storage was employed where necessary; branches inside loops were replaced where possible; atomics and critical sections removed or replaced with reductions; memory access was optimised to remove all scatter operations and minimise memory stride for gather operations.

A. MPI

The MPI implementation is based on a block-structured decomposition in which each MPI task is responsible for a rectangular region of the computational mesh. These processes each maintain a halo of ghost cells around their particular region of the mesh, in which they store data which is non-local to the particular process. As in any block-structured, distributed MPI application, there is a requirement for halo data to be exchanged between MPI tasks.

The decomposition employed by CloverLeaf attempts to minimise communications by minimising the surface area between MPI processes, whilst also assigning the same number of cells to each process to balance computational load. The depth of the halo exchanges also varies during the course of each iteration, depending on the numerical stencil. CloverLeaf v1.0 sends one MPI message per data field in exchanges involving multiple fields. CloverLeaf v1.1 aggregates messages for multiple fields to reduce latency and synchronisation costs.

To reduce synchronisation, data is only exchanged when required by the subsequent phase of the algorithm. Consequently no explicit MPI Barrier functions exist in the hydrodynamics timestep. All halo exchange communications are performed by processes using MPI_ISend and MPI_IRecv operations with their immediate neighbours, first in the horizontal dimension and then in the vertical dimension. MPI_WaitAll operations are used to provide local synchronisation between these data exchange phases. To provide global reduction functionality between the MPI processes, the MPI_Reduce and MPI_AllReduce operations are employed. The MPI implementation therefore uses MPI constructs for both intra- and inter-node communication between processes.

Twelve of CloverLeaf's kernels perform computational operations only. Communication operations reside in the overall control code and two other kernels. One kernel is called repeatedly throughout each iteration of the application, and is responsible for exchanging the halo data associated with one (or more) data fields, as required by the hydrodynamics algorithm. The second carries out the global reduction required for the minimum timestep. A further reduction is carried out to report intermediate results, but this is not essential for the numerical algorithm.

B. OpenMP

The hybrid version of CloverLeaf combines both the MPI and OpenMP programming models. This is effectively an evolution of the MPI version of the code in which the intra-node parallelism is provided by OpenMP, and inter-node communication provided by MPI. The number of MPI processes per node and the number of OpenMP threads per MPI process can be varied to achieve this and to suit different machine architectures. This approach reduces the amount of halo-cell data stored per node as this is only required for communications between the top-level MPI processes, not the OpenMP threads.

To implement this version, OpenMP parallel constructs were added around the loop blocks within CloverLeaf's fourteen kernels to parallelise them over the available OpenMP threads. The data-parallel structure of the loop blocks within the CloverLeaf kernels is very amenable to this style of parallelism. Figure 2 shows how this was achieved for the ideal gas kernel. Private constructs were specified where necessary to create temporary variables that are unique to each thread. OpenMP reduction primitives were used to implement the intra-node reduction operations required by CloverLeaf. It was also essential to minimise fork and join overheads between parallel loop-blocks by containing them within one larger parallel region. To do this correctly, all race conditions had to be removed.

The process of producing data parallel kernels with optimised memory access patterns and minimal branching naturally lends itself to producing vector friendly code. All computational loops vectorise though this does not necessarily mean improved performance in an algorithm limited by memory bandwidth.

C. OpenACC

The OpenACC version of CloverLeaf was based on the hybrid MPI/OpenMP version of the code, and uses MPI for distributed parallelism. Although driver code executes on the host CPUs within each node, only the GPU devices are used for computational work. The host CPUs are employed to coordinate the computation, launching kernels onto the attached GPU device, and for controlling MPI communications between nodes. Data transfers between the host processors and the accelerators are kept to a minimum and the code and data are fully resident on the GPU device.

In order to convert each kernel to OpenACC, loop-level pragmas were added to specify how the loops should be executed on the GPU, and to describe their data dependencies. Fully residency was achieved by applying OpenACC data "copy" clauses at the start of the program, which results in a one-off initial data transfer to the device. The computational kernels exist at the lowest level within the application's call-tree and we employ the OpenACC "present" clause to indicate that all input data is already available on the device. Immediately before halo communications, data is transferred from the accelerator to the host using the OpenACC "update host" clause. Following the MPI communication the updated data is transferred back from the host to its local accelerator using the OpenACC "update device" clause. The explicit data packing (for sending) and unpacking (for receiving) of the communication buffers is carried out on the GPU for maximum performance.

The first OpenACC version of CloverLeaf was developed under CCE using the Cray compiler using the OpenACC `Parallel` constuct. As alternative OpenACC implementations became available, the initial code was ported to these. Once a port was successfully carried out, the new source was ported back to the other compilers in order to remove all compiler specific implementations and produce a single, fully portable OpenACC source. Separate `Parallel` and `Kernel` construct based versions were created to allow fair comparisons to be made between all compilers. In the case of single loops the two constructs are virtually interchangeable.

D. OpenCL

Integrating OpenCL with Fortran is not trivial as the C and C++ bindings described by the OpenCL standard are not easy to call directly from Fortran. In order to create the OpenCL implementation of CloverLeaf, we wrote a new OpenCL-specific version for each of the existing kernel functions.

The implementation of each kernel is split into two parts: (i) an OpenCL device-side kernel that performs the required mathematical operations and; (ii) a host-side C++ routine to set up the actual OpenCL kernel. The Fortran driver routine calls the C++ routine, which is responsible for transferring the required data, setting kernel arguments, and adding the device-side kernel to the OpenCL work-queue with the appropriate work-group size.

Since each kernel performs a well defined mathematical function, and the Fortran versions avoid the use of any complex language features, writing the OpenCL kernels is almost a direct translation. In order to produce comparable results to the Fortran kernel, all computation is performed in double precision.

Each C++ setup routine relies on a static class, `CloverCL`, which provides common functionality for all the different setup routines. We moved as much logic as possible from the actual kernel functions into this static class. This helped to ensure that particular pieces of logic (e.g. the kernel `setArg` commands) are only re-executed when absolutely necessary thus improving overall performance.

All Fortran intrinsic operations (such as `SIGN`, `MAX etc.`) were also replaced with the corresponding OpenCL built-in function to ensure optimal performance.

The majority of the control code in the original Fortran kernels was moved into the C++ setup routines. Figure 4a illustrates this for the `ideal_gas` kernel. This ensures that branching is also always performed on the host instead of on any attached device, enabling the device kernels to avoid stalls and maintain higher levels of performance. It was also necessary to implement our own reduction operations.

E. CUDA

The CUDA version of CloverLeaf was based on the hybrid MPI/OpenACC version of the code, and uses MPI for distributed parallelism.

Integrating CloverLeaf's Fortran code base directly with CUDA's C bindings is difficult. A global class was written to handle interoperability between the Fortran and CUDA code-bases and to coordinate the data transfers with, and computation on, the GPU devices. Full device residency is achieved by creating and initialising all data items on the device, and allowing these to reside on the GPU throughout the execution of the program. Data is only copied back to the host when required for MPI communications and in order to write out visualisation files.

In order to create the CUDA implementation, we wrote a new CUDA version of each CloverLeaf kernel. The implementation of these was split into two parts:

```
try {
    ideal_knl.setArg(0, x_min);
              .
              .
              .
    if (predict == 0) {
        ideal_knl.setArg(4,
              CloverCL::density1_buffer);
    } else {
        ideal_knl.setArg(4,
              CloverCL::density0_buffer);
    }
} catch(cl::Error err) {
    CloverCL::reportError(err, ...);
}

CloverCL::enqueueKernel(ideal_knl, x_min, x_max,
                        y_min,y_max);
```

(a) The `ideal_gas` kernel OpenCL C++ host code

```
for (int k = get_global_id(1); k <= y_max;
      k += get_global_size(1)) {
    for (int j = get_global_id(0); j <= x_max;
          j += get_global_size(0)) {
        double ss2,v,pe,pv;

        p[ARRAY2D(j,k,...)]= (1.4-1.0)
            *d[ARRAY2D(j,k,...)]
            *e[ARRAY2D(j,k,...)];

        pe=(1.4-1.0)*d[ARRAY2D(j,k,...)];
        pv=-d[ARRAY2D(j,k,...)] *p[ARRAY2D(j,k,...)];

        v = 1.0/d[ARRAY2D(j,k,...)];
        ss2=v*v*(p[ARRAY2D(j,k,...)]*pe-pv);

        ss[ARRAY2D(j,k,...)]=sqrt(ss2);
    }
}
```

(b) The `ideal_gas` kernel OpenCL device code

Fig. 4: The two components of the OpenCL version of the `ideal_gas` kernel

(i) a C-based routine which executes on the host CPU and sets up the actual CUDA kernel(s); and (ii) a CUDA kernel that performs the required mathematical operations on the GPUs. Each loop block within the original C kernels was converted to an individual CUDA kernel, which typical resulted in numerous device-side kernels being developed to implement one original host-side kernel. This approach enabled us to keep the vast majority of the control code within the host-side C based routines and ensure that branching operations are always performed on the host instead of the attached GPU. This also ensures that the device-side kernels avoid stalls and maintain a high level of performance. The intra-node reduction operations were implemented using the Thrust library.

V. Results

To assess the current performance of CloverLeaf and the effectiveness of the optimisation techniques examined as part of this work we conducted a series of experiments.

A. Experimental Configuration

The hardware used in these experiments is summarised in Table I.

MACHINE	CPU	GPU	NODES	INTERCONNECT
Chilean Pine	AMD Opteron	X2090	40	Gemini
Archer	Intel Sandybridge	None	3008	Aries
Swan	Intel Sandybridge	K20X	130/8	Aries
Titan	AMD Opteron	K20X	18688	Gemini

TABLE I: Summary of Cray platforms

Chilean Pine is a 40 node Cray XK6, each node consisting of one 16-core AMD Opteron 6272 CPU and one NVIDIA Fermi X2090 GPU, each with 512 "Cuda cores" clocked at 1.15 GHz.

Titan is an XK7 16-core AMD Opteron CPUs and NVIDIA Tesla K20 GPU. The Cray XK7 system contains 18,688 nodes, with each holding a 16-core AMD Opteron 6274 processor. This too has the Gemini interconnect.

Archer is an 3008 node XC30 compute nodes contain two 2.7 GHz, 12-core E5-2697 v2. The Cray Aries interconnect links all compute nodes in a Dragonfly topology. It has no attached accelerators.

Swan is primarily a Cray XC series system, however a subset is configured as an XK7 consisting of 8 nodes each with an 8 core Intel Xeon E5-2670 and an attached NVIDIA Kepler K20X, with 2688 732 MHz cores and 6 GB of memory.

B. Intra-node Programming Model Experiments

A small and large representative square-shock benchmark test case has been taken from CloverLeaf's input suite. These have 960^2 and a 3840^2 meshes respectively, simulating a shock propagating from high-density region of ideal gas expanding into a larger, low density region of ideal gas. Execution is for 2955 and 87 timesteps respectively, which for their respective sizes, give reliable compute time for benchmarking.

Although CloverLeaf is capable of multi accelerated node runs these experiments were limited to a single card.

We compare performance for OpenACC for `Parallel` and `Kernel` variants for the CCE and PGI compilers. We also compare the performance of these against alternative programming methods to achieve acceleration, OpenCL and CUDA.

1) OpenACC: Table II shows the total run time for the small test problem for Chilean Pine and Swan. The results for the two OpenACC compilers for the `Parallel` and `Kernel` construct are shown alongside the CUDA and OpenCL figures.

MACHINE	CCE P	CCE K	PGI P	PGI K	CUDA	OpenCL
Chilean Pine	67.67	86.58	90.89	100.33	58.07	59.95
Swan	46.07	44.36	61.89	47.04	34.84	36.06

TABLE II: Small Test Run Time

MACHINE	CCE P	CCE K	PGI P	PGI K	CUDA	OpenCL
Chilean Pine	31.69	32.23	35.12	39.17	24.05	26.59
Swan	18.22	19.54	21.46	18.95	13.71	14.97

TABLE III: Large Test Run Time

Table III shows the same information as Table II for the large test problem.

For the small test case on Chilean Pine, CCEs `Parallel` version is the fastest OpenACC implementation, followed by the CCE `Kernel` version. The PGI `Parallel` version is outperformed by its `Kernel` version, but both fall short of the CCE times, by approximately 25% for the two best cases. The hand coded OpenCL and CUDA both outperform all the OpenACC implementations. CUDA is about 14% faster than the best case OpenACC result.

For the small test case on Swan, CCEs `Kernel` version is now the fastest OpenACC implementation, and the difference between CCE and PGI best cases drops to only about 6%. The PGI `Parallel` is now outperformed by its `Kernel` version again, and the `Kernel` version has improved significantly from the Chilean Pine result. The hand coded OpenCL and CUDA still outperform all the OpenACC implementations, but CUDA now drops to 21% faster than the best case OpenACC result.

For the large test case on Chilean Pine the results are broadly the same but there are noticeable changes in relative performance compared to the small test. The CCE `Parallel` version is still the fastest OpenACC implementation, but now the CCE `Kernel` version and the PGI `Kernel` version are much closer in performance. Hand coded OpenCL and CUDA also outperform all the OpenACC implementations. CUDA is about 24% faster than the best case OpenACC result.

For the large test case on Swan, CCEs `Parallel` construct now outperforms its `Kernel` construct. The PGI `Kernel` version is now faster than the CCE `Kernel` version and only lags the best CCE result by about 4%. The PGI `Parallel` version also performs significantly better for the larger data set. Clearly, the various implementations are tuned to different vector lengths. Hand coded OpenCL and CUDA still outperform all the OpenACC implementations. CUDA is about 25% faster than the best case OpenACC result.

All the systems used NVIDA hardware so it is not unexpected that CUDA is the most performant option, but OpenCL only falls short of the CUDA result by about 3% and 9% for the small and large tests on Chilean Pine respectively, and by 3% and 8% on Swan, but its open status potentially allows a wider range of platforms to be targeted.

The NVIDIA hardware performed significantly better from the X2090 to the K20X, achieving a speed up of 2.4 and 1.75 for the two test problems respectively.

C. Multi-node Strong Scaling Experiments

To assess the performance, at scale, of the various programming models and the optimisations techniques examined we conducted a series of strong-scaling experiments, on the Cray platforms, using the 15360^2 cell problem from the CloverLeaf benchmarking suite. For each job size examined we executed all versions of CloverLeaf within the same node allocation to eliminate any performance effects due to different topology allocations from the batch system.

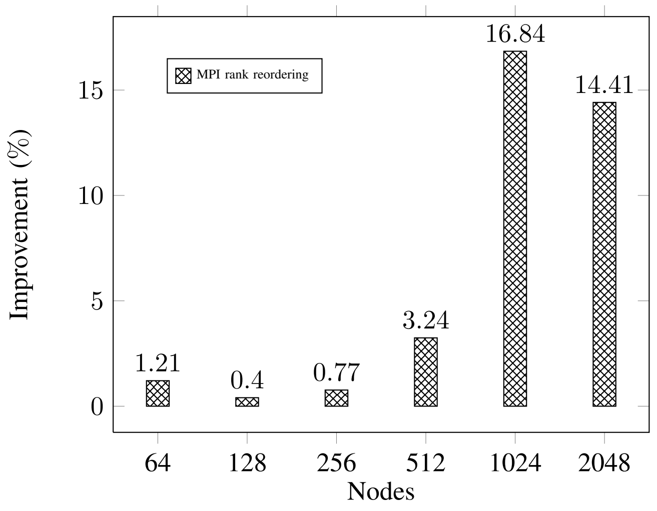


Fig. 6: Relative performance improvement of the reference MPI implementation due to rank reordering on Archer (XC30)

1) Hybrid (MPI+OpenMP) Programming Model: To assess the affect of employing a hybrid programming model compared to a "flat" MPI model we conducted a series of experiments on the Archer Cray XC30 platform. Figure 5 shows the relative performance of the hybrid model against the "flat" MPI result. In this chart a positive result represents an improvement in performance. This demonstrates that employing a hybrid programming model can deliver a significant improvement in performance over the "flat" MPI model, reaching as much as 20% at some scales. Our results seem to show the hybrid model initially performing better than the "flat" MPI model, before the relative performance falls off and is comparable with the "flat" MPI model at 512 nodes, before increasing again at the higher nodes levels (1024 and 2048 nodes). We speculate that these effects are initially due to memory savings enabled by the hybrid model and are due to message aggregation at the higher node counts. All configurations of our hybrid model which utilise at least 1 MPI process per Numa region deliver a performance improvement, however, the configurations which only use 1 MPI process per node and 24 OpenMP threads across the node, consistently perform worse compared to the "flat" MPI model. In our experiments this slowdown was as much as 60% in the worse case.

2) MPI rank reordering: When a parallel program executes, MPI tasks are assigned to compute cores within the overall system. Since compute nodes (which each contain 24 cores on Archer) may be located on different "islands" within the Dragonfly topology (Aries network), communication time between tasks will vary depending not only on node placement, but also the placement of each task within the allocated nodes. Intra-node communications should perform more efficiently compared to inter-node communications due to the increased bandwidth available via on-node shared memory compared to the data transfers over the interconnect.

One way to change MPI task placement on cores is to change the rank ordering, the order in which MPI tasks (or ranks) are assigned to cores. By default CloverLeaf takes no account of MPI task-to-node placement when assigning chunks of the decomposed mesh to MPI ranks. If MPI tasks are reordered so that neighbouring mesh blocks within the overall decomposition are co-located on the same node, then the communication traffic over the interconnect should be minimised and communication exchange times reduced.

On Cray platforms when a parallel program is executed the environment variable Mpich_Rank_Reorder_Method determines the order in which tasks are assigned to cores. This environment variable can be set to an integer ranging between 0 and 3: 0 represents a round robin allocation, 1 is SMP style and 2 indicates that folded-rank should be used, SMP style is the default option. Setting this variable to 3 allows a custom rank reordering to be employed which is read in from a file by the job on startup.

The custom rank reordering file can be generated either manually or automatically using Cray tools. Cray provides the `Grid_order` tool to manually explore alternative placement options. Additionally Cray's CrayPAT suite of tools can be used to automatically generate a suggested rank reordering placement. This can be achived by building the application with the `perftools` module loaded then using the `pat_build` tool to produce an instrumented binary. Next the instrumented binary can be executed using the default rank order method to produce a profile. Finally `pat_report` can be used to generate two recommended rank orders: d and u. Figure 7 shows this workflow diagrammatically.

To assess the affect on performance of the rank reordering optimisation on CloverLeaf we conducted a series of strong-scaling experiments using the Archer Cray XC30 platform. In these experiments we used the `Grid_order` tool to manually generate a custom rank reordering file. As the Cray XC platform (Archer) used in these experiments has 24 cores per node, we specified the local blocks of chunks assigned to each node to have dimensions of 4x6 chunks, as shown in figure 8.

We then executed the reference MPI implementation of CloverLeaf using both the default rank ordering and the new custom rank reordered placement file. Figure 6 shows the results of these experiments. This shows that as the scales of the experiments are increased the rank reordering optimisation has a greater affect on overall performance relative to the default ordering. In our experiments the performance improvement reached 16.8% and 14.4% at the 1024 and 2048 node experiments respectively. It is the view of the authors' that this represents a relatively straightforward mechanism with which to improve the performance of applications at scale, as it does not actually involve any changes to the code of the application.

3) Cray Reveal: Hybrid programming (MPI+OpenMP) has been recognised as a potential solution for improving the scalability and performance of parallel scientific applications. It has also been recognised that incorporating OpenMP directives into existing MPI applications is an extremely complex and time consuming task. To alleviate this problem Cray developed the Reveal tool to automatically hybridise applications.

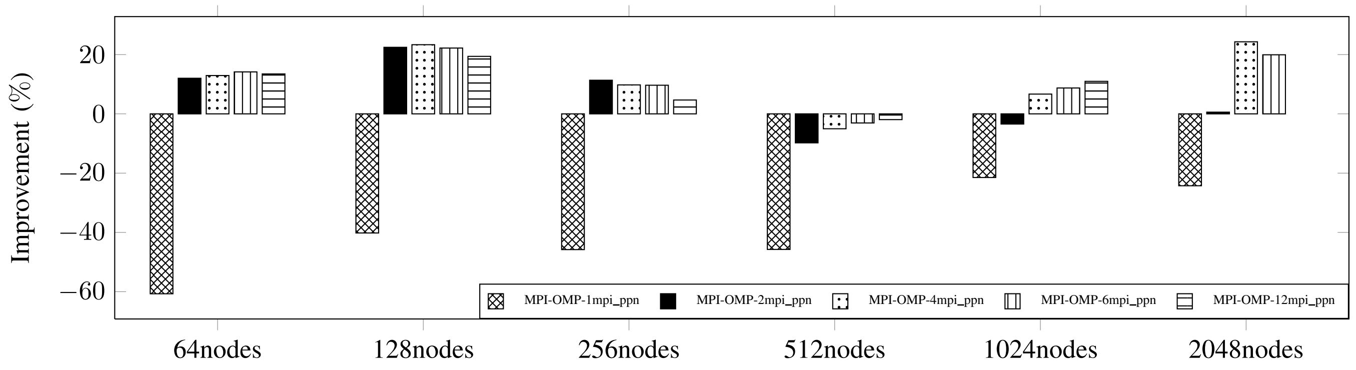
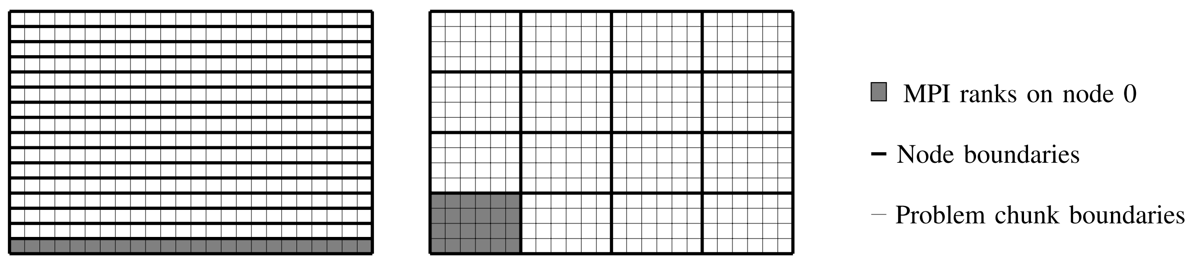


Fig. 5: Performance of the hand-coded hybrid version relative to the reference MPI implementation on Archer (XC30)



(a) Original strategy

(b) Improved strategy

Fig. 8: MPI rank reordering strategy.

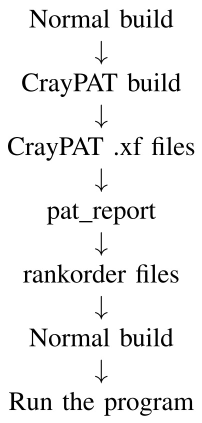


Fig. 7: Rank Reordering Process

Reveal is included in CrayPat v6+ as part of the Perftools tool suite and must be used with the Cray compiler suite. It allows compiler feedback information to be viewed along with performance information.

It helps to identify time consuming loop blocks, with compiler feedback on dependency and vectorisation, however the loop scope analysis was the main functionality used in this study. This provides variable scope and compiler directive suggestions for inserting OpenMP parallelism into a serial or pure MPI code. Reveal works through a GUI which gives scoping assistance and suggests directives that can be added. It is not intended as a full solution but as a guide to improve programmer productivity.

The tool successfully scoped all of the loop blocks within the “flat” MPI version of CloverLeaf with the exception of three variables which it requested user assistance for. After specifying scoping information for these additional variables the generated code was tested and verified to be correct. It recognised all reductions correctly and did not require any atomics or critical sections, which was the correct solution.

Overall Reveal is very easy to use and successfully hybridised the full codebase of CloverLeaf within a couple of hours. This could potentially be reduced significantly if the tool had a mechanism which allowed all the files of an application to be queued up and analysed automatically. Currently however each file needs to be analysed separately.

The main issue encountered, however, was that the tool is currently unable to assess whether multiple separate parallel loop blocks could be included within one larger parallel region. Consequently the generated code wraps each parallel loop block within its own parallel region which potentially increases thread synchronisations and overheads. The ability to detect race conditions would also be a useful addition to the tool as would the inclusion of support for OpenACC.

In these experiments we conducted a series of strong-scaling experiments to assess the performance of the hybrid version of CloverLeaf produced by Reveal from the reference “flat” MPI implementation of CloverLeaf. We again used the 15360^2 benchmark problem from the CloverLeaf suite and executed all of the application runs within the same node allocation from the batch system to eliminate the effect of different topology allocations.

We compared the performance of the hybrid version of CloverLeaf produced by Reveal with that of our hand-coded hybrid version on the Archer Cray XC30 platform. Figure 9 shows the results of these experiments.

Overall the hybrid version produced by Reveal was surprisingly performant, it was consistently within 2% of the performance of the hand-coded hybrid version. On occasions, however, this performance disparity slipped to 8%

VI. Conclusions and Future Work

A. OpenACC

While performance is important, for a new programming standard like OpenACC, the convergence of the standard across a range of compilers is an equally important factor for portability.

CCE provided the best performance on both test problems and both platforms, but for the large data set size on Swan all the run times were close and the PGI `Kernel` version ran quicker than the equivalent CCE version. This shows that value has been added by all providers and also that there is room for improvement in performance towards the low level representations. The CUDA and OpenCL versions achieve a higher level of performance across all four test configurations, but never more than 25% better.

It is eighteen months since the first OpenACC implementation of CloverLeaf was developed on the Cray compiler. At this time other compilers either refused to compile the code or crashed at run time or produced the wrong answer. This was partly due to the fact that the code was originally developed on the Cray compiler and the immaturity of the compilers in general. Another reason was the starting point of OpenMP at the loop level, which maps more readily to the `Parallel` construct. If another compiler had been chosen as a starting point then the same outcome would have occurred, it would work on that compiler and not the others.

We now have a single source for the `Parallel` and the `Kernel` construct versions that works without modification on both compilers and delivers similar run times. OpenACC has matured significantly in both its portability and performance, which is major step forward. To achieve this, the source tended towards the lowest common denominator which had some minor impact on performance. Once the initial optimisations that re-factored the essentially serial algorithm into a data parallel algorithm had been implemented, the actual computational kernels changed very little between versions. The fact that the re-factored code also performed better on a traditional CPU showed that the same optimisations are important on all platforms, though the impact of poorly optimised code is less noticeable than on many core architectures. It should be noted that the `Parallel` and `Kernel` constructs are not mutually exclusive, and can be mixed and matched depending on individual accelerator region performance, and optimal constructs could be used depending on specific compiler and target architecture configurations, as part of an optimisation strategy.

The easy translation from OpenMP to OpenACC and then into Offload and OpenMP 4.0 is a major boon for OpenACC and it is hoped that it will converge with OpenMP 4.0 in the coming years.

We have shown that mini-apps allow us to rapidly measure the reduction in performance due to abstraction. They also enable us to provide value in compiler and runtime development by allowing multiple versions to be evaluated in different programming models. Vendors have used these multiple versions to enhance their OpenACC implementations.

We have also shown that simple Fortran kernels, which as a whole, make up a valid scientific application, albeit it in mini-app form, can use OpenACC as a abstraction to the hardware and low level programming models, with an acceptable level of performance.

CloverLeaf contains an internal profiler which records total run times of individual kernels. In the future we will carry out a kernel to kernel comparison to see if the relative differences in performance between CCE, PGI, CUDA and OpenCL can be explained, based on kernel function. Currently we are only using one element of the compute node, the GPGPU. We also intend to investigate using all the compute on a heterogeneous node.

The compiler vendors are also planning to provide a wider range of back-ends including PTX, LLVM, SPIR, OpenCL and CUDA. This will allow us to target a wider range of hardware, not limited to GPGPUs and NVIDIA devices in the future.

B. Multi-node Strong Scaling Experiments

Our strong scaling experiments demonstrate the utility of the hybrid (MPI+OpenMP) programming model. In our experiments the performance of the hybrid (MPI+OpenMP) is consistently superior to that of the “flat” MPI implementation. With performance improvements due to hybridisation reaching as much as 20% at some scales. At 512 nodes however the performance of both versions is broadly similar. Below this we believe the performance improvements are due to memory savings, whilst above this we attribute the performance improvement to the superior message aggregation which the hybrid approach delivers.

Cray’s Reveal tool also provided a very robust and reliable method of automatically producing a hybrid (MPI+OpenMP) version of the code from the original “flat” MPI version. Although the code had already been re-factored to remove data dependencies and atomics, it successfully threaded all the kernels correctly. Its inability to check dependencies between parallel loops does mean that some unnecessary synchronisations are included, however we feel that the application programmer is probably best placed to assess potential race conditions. Its full definition of private and public variables around the OpenMP loop constructs is also a very useful feature. We feel that it should also be relatively straightforward to convert the hybrid code produced by Reveal to a full OpenACC based implementation. It would therefore also be extremely useful if Reveal was able to recognise when data transfers would be required between host and accelerator devices.

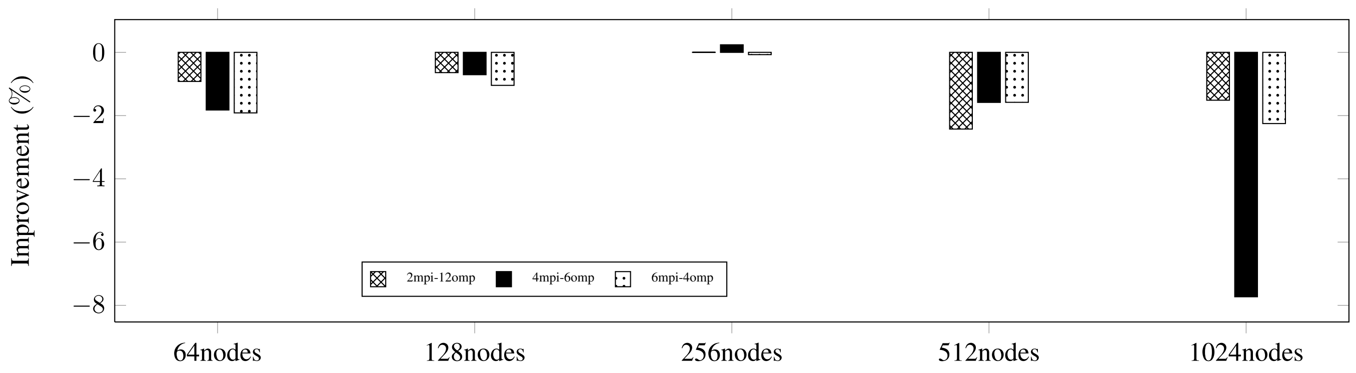


Fig. 9: Performance of the hybrid (MPI+OMP) versions produced by Reveal relative to the reference hybrid implementation on Archer (XC30)

As we approach the era of Exascale computing improving the scalability of applications will become increasingly important in enabling applications to effectively harness the parallelism available in future architectures and to achieve the required levels of performance. Our experiments also demonstrate the importance of selecting a process-to-node mapping which accurately matches the communication pattern inherent in the application, particularly at scale. In this work we have demonstrated that Cray's Grid_order tool can be effective is producing MPI rank reorder files to achieve this.

Additionally, developing hybrid applications which are able to effectively harness the computational power available in attached accelerator devices such as GPUs will also become increasingly important.

Overall, we feel that MPI is still the most likely candidate programming model for delivering inter-node parallelism going forward towards the Exascale era. Hybrid programming approaches primarily based on OpenMP will also become increasingly important and can, as this work has shown, deliver significant performance advantages at a range of different job scales. We also feel that utilising, in some form, the computational power available through the parallelism inherent in current accelerator devices will be crucial in reaching Exascale levels of performance. However a performant open standards based approach will be vital in order for large applications to be successfully ported to future architectures. In this regard OpenACC shows promise, however, we eagerly await the inclusion of accelerator directives into OpenMP implementations.

In future work, using CloverLeaf, we plan explore additional alternative rank-to-topology mappings which may deliver further performance benefits e.g. increasing the number of network hops between neighbouring processing may effectively increase the bandwidth available to each process by increasing the number of communication paths available to them. This may present an interesting trade-off against the increased communication latency in such an arrangement. To determine whether our hypotheses are correct, regarding the causes of the performance disparities presented here, we plan to conduct additional experiments to produce detailed profiles of the various implementations. We also plan to experiment with Reveal on a code that hasn't yet been optimised to be fully data parallel, as would be the case for most legacy and production codes.

Acknowledgments

This work is supported by the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Programme) and also the Royal Society through their Industry Fellowship Scheme (IF090020/AM).

We would like to express our thanks to the UK Engineering and Physical Sciences Research Council and to Oak Ridge National Laboratory in the USA for access to the Archer and Titan systems respectively. Additionally we would like to thank Alistair Hart of Cray, PGI and NVIDIA.

This paper would not have been possible without the considerable assistance form each of the compiler vendors: Cray, PGI. In particular John Levesque and Alistair Hart from Cray, Inc.; Doug Miles, Craig Toepfer and Michael Wolfe of The Portland Group (PGI).

CloverLeaf is available via GitHub (see http://warwick-pcav.github.com/CloverLeaf/), and is also released as part of the Mantevo project run by Sandia National Laboratories which can be accessed at http://www.mantevo.org.

References

[1] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep., 2009.

[2] A. Mallinson, D. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, S. Jarvis, "CloverLeaf: Preparing Hydrodynamics Codes for Exascale," in *The Cray User Group 2013, May 6-9, 2013, Napa Valley, California, USA (2013)*, 2013.

[3] A. Mallinson, D. Beckingsale, W. Gaudin, J. Herdman, S. Jarvis, "Towards Portable Performance for Explicit Hydrodynamics Codes," in *The International Workshop on OpenCL (IWOCL) 2013*, 2013.

[4] J. Herdman, W. Gaudin, D. Beckingsale, A. Mallinson, S. McIntosh-Smith, and M. Boulton, "Accelerating hydrocodes with openacc, opecl and cuda," in *High Performance Computing, Networking, Storage and Analysis (SCC) 2012 SC Companion*, 2012.

[5] J. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3d into an exascale application using OpenACC," in *SC12, November 10-16, 2012, Salt Lake City, Utah, USA (2012)*, 2012.

[6] Lavallée, P. and Guillaume, C. and Wautelet, P. and Lecas, D. and Dupays, J., "Porting and optimizing HYDRO to new platforms and programming paradigms - lessons learnt," *available at www.prace-ri.eu*, accessed February 2013.

[7] Henty, D., "Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling," *ACM/IEEE SuperComputing Proceedings*, 2000.

[8] Cappello, F. and Etiemble, D., "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks," *Proceedings of the 2000 ACM/IEEE conference on SuperComputing*, 2000.

[9] Mahinthakumar, G. and Saied, F., "A Hybrid MPI-OpenMP Implementation of an Implicit Finite-Element Code on Parallel Architectures," *International Journal of High Performance Computing Applications*, 2002.

[10] Wang, X. and Jandhyala, V., "Enhanced Hybrid MPI-OpenMP Parallel Electromagnetic Simulations Based on Low-Rank Compressions," *International Symposium on Electromagnetic Compatibility*, 2008.

[11] Sharma, R. and Kanungo, P., "Performance Evaluation of MPI and Hybrid MPI+OpenMP Programming Paradigms on Multi-Core Processors Cluster," *International Conference on Recent Trends in Information Systems*, 2001.

[12] Jones, M. and Yao, R., "Parallel Programming for OSEM Reconstruction with MPI, OpenMP, and Hybrid MPI-OpenMP," *Nuclear Science Symposium Conference Record*, 2004.

[13] Drosinos, N. and Koziris, N., "Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters," *International Parallel and Distributed Processing Symposium*, 2004.

[14] Nakajima, K., "Flat MPI vs. Hybrid: Evaluation of Parallel Programming Models for Preconditioned Iterative Solvers on "T2K Open Supercomputer"," *International Conference on Parallel Processing Workshops*, 2009.

[15] Adhianto, L. and Chapman, B., "Performance Modeling of Communication and Computation in Hybrid MPI and OpenMP Applications," *IEEE: International Conference on Parallel and Distributed Systems*, 2006.

[16] Li, D. and Supinski, B. and Schulz, M. and Cameron, K. and Nikolopoulos, D., "Hybrid MPI/OpenMP Power-Aware Computing," *International Symposium on Parallel and Distributed Processing*, 2010.

[17] He, Y. and Antypas, K., "Running Large Scale Jobs on a Cray XE6 System," in *Cray User Group*, 2012.

[18] M. Baker, S. Pophale, J.-C. Vasnier, H. Jin, and O. Hernandez, "Hybrid programming using OpenSHMEM and OpenACC," in *OpenSHMEM, Annapolis, Maryland. March 4-6, 2014*, 2014.

[19] R. Reyes, I. Lopez, J. Fumero, and F. de Sande, "Directive-based programming for gpus: A comparative study," *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[20] S. Wienke, P. Springer, C. Terboven, and D. Mey, "OpenACC first experiences with real-world applications," in *Euro-Par 2012, LNCS, Springer Berlin/Heidelberg(2012)*, 2012.

[21] van Leer, B, "Towards the Ultimate Conservative Difference Scheme, V. A Second Order Sequel to Godunov's Method," *J. Comput. Phys. 32 (1): 101136*, 1979.

[22] "Message Passing Interface Forum," http://www.mpi-forum.org, February 2013.

[23] "OpenMP Application Program Interface version 3.1," http://www.openmp.org/mp-documents/OpenMP3.1.pdf, July 2011.

[24] Stotzer, E. *et al* , "OpenMP Technical Report 1 on Directives for Attached Accelerators," The OpenMP Architecture Review Board, Tech. Rep., 2012.

[25] "CUDA API Reference Manual version 4.2," http://developer.download.nvidia.com, April 2012.

[26] "The OpenCL specification version 1.2," http://www.khronos.org/opencl/, March 2014.

[27] "Accelerated Parallel Processing (APP) SDK," http://developer.amd.com/tools/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/, November 2012.

[28] "Intel SDK for OpenCL Applications 2012," http://software.intel.com/en-us/vcsource/tools/opencl-sdk, November 2012.

[29] "OpenCL Lounge," https://www.ibm.com/developerworks/community/alphaworks/tech/opencl, November 2012.

[30] "OpenCL NVIDIA Developer Zone," https://developer.nvidia.com/opencl, November 2012.

[31] "The OpenACC application programming interface," http://www.openacc-standard.org, November 2011.

[32] Bland, A. and Wells, J. and Messer, O. and Hernandez, O. and Rogers, J., "Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory," in *Cray User Group*, 2012.

[33] "PGI Fortran & C accelerator compilers and programming model," http://www.pgroup.com/lit/pgiwhitepaperaccpre.pdf, March 2014.

[34] "OpenACC accelerator directives," http://www.training.prace-ri.eu/uploads/tx_pracetmo/OpenACC.pdf, November 2012.

[35] "Parallel thread execution isa application guide v3.2," ADDURL, July 2013.

[36] "HMPP: A hybrid multicore parallel programming platform," http://www.caps-entreprise.com/en/documentation/caps, .