CloverLeaf: Preparing Hydrodynamics Codes for Exascale

A. C. Mallinson*, D. A. Beckingsale*, W. P. Gaudin†, J. A. Herdman†, J. M. Levesque‡ and S. A. Jarvis*

*Performance Computing and Visualisation, Department of Computer Science, University of Warwick, UK. Email: {acm,dab}@dcs.warwick.ac.uk

†High Performance Computing, Atomic Weapons Establishment, Aldermaston, UK Email: {Andy.Herdman,Wayne.Gaudin}@awe.co.uk

‡CTO Office, Cray Inc, Knoxville, TN, USA. Email: levesque@cray.com

Abstract—**In this work we directly evaluate five candidate programming models for future exascale applications (MPI, MPI+OpenMP, MPI+OpenACC, MPI+CUDA and CAF) using a recently developed Lagrangian-Eulerian explicit hydrodynamics mini-application. The aim of this work is to better inform the exacsale planning at large HPC centres such as AWE. Such organisations invest significant resources maintaining and updating existing scientific codebases, many of which were not designed to run at the scale required to reach exascale levels of computation on future system architectures. We present our results and experiences of scaling these different approaches to high node counts on existing large-scale Cray systems (Titan and HECToR). We also examine the effect that improving the mapping between process layout and the underlying machine interconnect topology can have on performance and scalability, as well as highlighting several communication-focused optimisations.**

***Keywords*—MPI, Co-array Fortran, OpenMP, OpenACC, CUDA, Hydrodynamics, Exascale.**

I. Introduction

HPC systems continue to evolve through massive increases in hardware concurrency rather than CPU clock speeds. This presents a significant challenge to large HPC centres such as the Atomic Weapons Establishment (AWE), as in order to reach exascale levels of performance on future HPC platforms, many of their applications require significant scalability improvements. It is also recognised that a key feature of future exascale system architectures will be significantly increased intra-node parallelism, relative to current HPC system node architectures [1]. This increase in parallelism is likely to come from two sources: (i) increased CPU core counts; and (ii) the use of massively-parallel hardware accelerators, such as discrete general-purpose graphics processing units (GPGPUs).

It is argued that existing programming approaches, mainly based on the dominant MPI (Message Passing Interface) library, are starting to reach their scalability limits and it is also the case that developing and maintaining "flat" MPI applications is becoming increasingly problematic [2]. Additionally, MPI provides no mechanism for codes to make use of attached accelerator devices.

Unlike the MPI model, PGAS (Partitioned Global Address Space) based approaches such as CAF (Co-array Fortran) rely on a lightweight one sided communication model and a global memory address space. Together with hybrid programming approaches such as combining MPI with OpenMP or OpenACC, PGAS-based approaches represent promising areas of research for improving the performance and scalability of applications as well as programmer productivity. To date, insufficient work has been conducted to directly evaluate each of these approaches at scale on current high-end HPC platforms, particularly within the sphere of explicit Lagrangian-Eulerian hydrodynamics, which is a key focus of this work.

Large HPC sites such as AWE are required to make significant investments maintaining their existing scientific codebases. These organisations are also challenged with the task of deciding how best to develop their existing applications to take advantage of future HPC system architectures, in order to effectively harness future exascale-levels of computing power. Evaluating the strengths of each of these approaches using production codes is difficult, applications are often complex and consist of hundreds of thousands of lines of code. Together with limited developer and financial resources, this complexity means that it is not possible to explore every software development option for each HPC application. We report on how this programming model exploration, architecture evaluation and general decision making can be improved through the use of mini-applications. Mini-applications are small, self contained programs that embody essential performance characteristics of larger applications, and thus provide a viable way to conduct this analysis [3].

In this work we seek to directly evaluate five candidate programming models for future exascale applications (MPI, MPI+OpenMP, MPI+OpenACC, MPI+CUDA and CAF) and two candidate system architectures, using a recently developed Lagrangian-Eulerian explicit hydrodynamics mini-application (CloverLeaf). We evaluate each of these approaches in terms of their performance and scalability and present our results and experiences of scaling them to high node counts on existing large-scale Cray systems (Titan and HECToR). We also examine the effect that improving the mapping between process layout and the underlying machine interconnect topology can have on performance and scalability, as well as highlighting the effects of several other communication-focused optimisations.

Specifically, we make the following key contributions:

- We present a detailed description of CloverLeaf's implementation in each of the aforementioned programming models, as well as information on its hydrodynamics scheme;
- We present a direct performance comparison of these five approaches, at considerable scale, on two alternative Cray system architectures;
- Finally, we discuss in detail a number of key optimisations and their effects on the performance of CloverLeaf at scale.

The remainder of this paper is organised as follows: Section II discusses related work in this field. In Section III we present background information on the hydrodynamics scheme employed by CloverLeaf and the programming models which this work examines. Section IV describes the implementation of CloverLeaf in each of these programming models and also provides details of the optimisations which we have examined. The results of our study are presented and analysed in Section V, together with a description of our experimental setup. Finally, Section VI concludes the paper and outlines future work.

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. To our knowledge, these studies have generally focused on different scientific domains to the one we examine here, and on applications which implement different algorithms or exhibit different performance characteristics. Similarly, although CAF has only relatively recently been incorporated into the official Fortran standard, earlier versions of the technology have existed for some time. Consequently, a number of studies have already examined the technology; again, these have generally focused on different classes of applications, or different hardware platforms, to those employed here.

The results from these studies have also varied significantly, with some authors achieving significant speedups by employing the CAF and hybrid constructs and others presenting performance degradations. Substantially less work exists which directly evaluates the MPI, hybrid and CAF programming models when applied to the same application. We are also unaware of any work which has sought to directly compare the previous technologies to approaches based on OpenACC and CUDA for implementing the hybrid programming model on systems incorporating accelerator devices. Our work is motivated by the need to further examine each of these programming models, particularly when applied to Lagrangian-Eulerian explicit hydrodynamics applications.

In previous work we reported on our experiences of porting CloverLeaf to GPU-based architectures using CUDA and OpenACC [4], [5]. 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 for the S3D application [6].

One study that does directly evaluate the hybrid and CAF based programming models at considerable scale is from Preissl *et al.* [7]. They present work which demonstrates a CAF-based implementation significantly outperforming an equivalent MPI-based implementation on up to 131 thousand processor cores. However, their study examines a significantly different class of application, a 3D Particle-In-Cell code simulating a Gyrokinetic Tokamak system.

Additionally, Stone *et al.* were unable to improve the performance of the MPI application on which their work focused by using the CAF constructs, instead experiencing a significant performance degradation [8]. Their work, however, focused on the CGPOP mini-application, which represents the Parallel Ocean Program [9] from

Los Alamos National Laboratory.

Whilst the application examined by Lavallée *et al.* has similarities to CloverLeaf, and their work compares several hybrid approaches against a purely MPI based approach [10], they focus on a different hardware platform and do not examine either CAF- or OpenACC-based approaches.

Studies such as [11]–[13] report performance degradations when employing hybrid (MPI+OpenMP) based approaches, whilst others experience improvements [14]–[16]. 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 (coarse- and fine-grained) against the "flat" MPI approach [17]. 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 [18]. 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 [19]. Li *et al.* employ the hybrid approach in their work which examines how to achieve more power-efficient implementations of particular benchmarks [20]. Henty also provides a comparison between MPI and CAF using several micro-benchmarks [21]. Various approaches and optimisations for executing large-scale jobs on Cray platforms are also examined by Yun *et al.* in [22]. Minimising communication operations within applications has also been recognised as a key approach for improving the scalability and performance of scientific applications [23].

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.

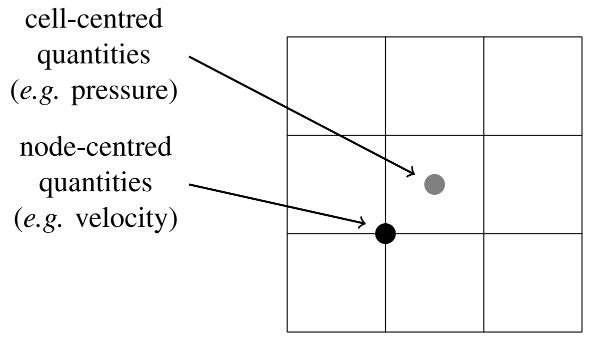

cell-centred quantities (*e.g.* pressure)
node-centred quantities (*e.g.* velocity)

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

A. Hydrodynamics Scheme

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 conservations 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.

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 node stores a velocity vector. An explicit finite-volume method is used to solve the equations with second-order accuracy. The system is hyperbolic meaning that the equations can be solved using explicit numerical methods, without the need to invert a matrix. 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 nodes move due to the fluid flow; (ii) an advection step restores the cells to their original positions by moving the nodes back, and calculating the amount of material that has passed through each cell. This is accomplished using two sweeps, one in the horizontal dimension and the other in the vertical.

The computational mesh is spatially decomposed and distributed across processes 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 process. Data exchanges occur mainly between immediate neighbouring processes (vertically and horizontally), within the decomposition. A global reduction operation is required by the algorithm during the calculation of the timestep value, which is calculated once per iteration.

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 [24].

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) CAF: Several CAF extensions have been incorporated into the Fortran 2008 standard. The additions aim to make parallelism a first class feature of the Fortran language [25].

CAF continues to follow the SPMD language paradigm with a program being split into a number of communicating processes known as images. Communications are one-sided, with each process able to use a global address space to access memory regions on other processes, without involving the remote processes in the communications. The "=" operator is overloaded for local assignments and also for remote loads and stores. Increasingly, off-image loads and stores are being viewed as yet another level of the memory hierarchy [26].

Two forms of synchronisation are available within CAF. The *"sync all"* construct provides a global synchronisation capability. Whilst the *"sync images"* construct provides functionality to locally synchronise a subset of images. Collective operators have not yet been standardised, although Cray have implemented their own versions of several commonly used collectives.

3) OpenMP: OpenMP is an Application Program Interface (API) and has become the de facto standard in shared memory programming [27]. The technology is supported by all the major compiler vendors and is based on the use of a set of pragmas that can be added to source code to express parallelism. The technology is based on a fork-join model of concurrency. 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 for the inclusion of additional directives to target accelerator based devices such as GPUs [28].

4) CUDA: NVIDIA's CUDA [29] 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 thus enables programmers to harness new hardware developments in NVIDIA's latest GPU devices, it does have application portability implications.

5) OpenACC: The OpenACC [30] 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) [31] and supported by an initial group of three compiler vendors [32]–[34]. 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 codes and easing programmability whilst also providing a portable, open standards-based solution.

```
!$OMP PARALLEL
!$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

IV. Implementation

The two main steps of CloverLeaf are implemented via fourteen individual kernels. In this instance, we use *kernel* to refer to a small 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 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 or OpenACC.

Not all the kernels used by CloverLeaf are as simple as the example in Figure 2. However, during the initial development of the code, we engineered the algorithm to ensure that all loop-level dependencies within the kernels were eliminated. Most of the dependencies were removed via small code rewrites: large loops were broken into smaller parts, extra temporary storage was employed where necessary, and branches inside loops were also removed.

Each CloverLeaf kernel can have multiple implementations; both C and Fortran kernels have previously been developed for the entire hydrodynamics scheme. The design of CloverLeaf also enables the desired kernel to be selected at runtime. This is managed by a driver routine (see Figure 3), which also handles data communication and I/O.

A. MPI, CAF & Hybrid Implementations

The MPI implementation is based on a block-structured decomposition in which each process is responsible for a rectangular region of the overall 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.

```
IF (use_fortran_kernels) THEN
    CALL ideal_gas_kernel

ELSEIF (use_c_kernels) THEN
    CALL ideal_gas_kernel_c

ELSEIF (use_cuda_kernel) THEN
    CALL ideal_gas_kernel_cuda
ENDIF
```

Fig. 3: Runtime selection for the `ideal_gas` kernel

Thirteen of CloverLeaf's kernels perform only computational operations; communication operations reside in the overall control code and a fourteenth kernel (*update_halo*). This 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 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, ensuring good computational load balancing. The depth of the halo exchanges also varies during the course of each iteration to further minimise data exchanges. Currently one MPI message is used per data field in exchanges involving multiple fields.

To reduce synchronisation, data is only exchanged when required by the subsequent phase of the algorithm. Consequently no *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. These are used for the calculation of the timestep value during each iteration and for periodic intermediary results. The MPI implementation therefore uses MPI constructs for both intra- and inter-node communication between processes.

The CAF versions of CloverLeaf largely mirror the implementation of the MPI version, except that the MPI communication constructs are replaced with CAF equivalents. The two-sided MPI communications are replaced with one-sided asynchronous CAF *"put"* operations, in which the CAF process responsible for the particular data items writes them into the appropriate memory regions of its neighbouring processes; no equivalent receive operations are therefore required. One existing version uses CAF constructs to exchange the communication buffers which were previously used by the MPI implementation. An alternative version, however, performs the data exchanges directly between the data fields (2D-arrays) stored within each process, using strided memory operations were necessary. In both versions the top-level Fortran type structure which contains all the data fields and communication buffers, is declared as a Co-array.

CAF synchronisation constructs are employed to prevent race conditions between the processes. To evaluate the two synchronisation constructs available within CAF, each version can be configured to use either the global *"sync all"* construct or the more local *"sync images"* construct between immediate neighbouring processes. The selection between these constructs is controlled by compile-time pre-processor directives.

The CAF versions examined here employ the proprietary Cray collective operations to implement the global reduction operations. Although we have also developed hybrid (CAF+MPI) alternatives which utilise MPI collectives, in order to make these versions portable to other CAF implementations. In this study, however, we only report on the performance of the purely CAF-based versions as we have not observed any noticeable performance differences between the CAF and MPI collectives on the Cray architecture.

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 suite 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.

B. OpenACC & CUDA Implementations

The OpenACC and CUDA versions of CloverLeaf are both based on the MPI version of the code, and use MPI for distributed parallelism. Although both versions execute on the 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 both the OpenACC and CUDA versions 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.

Integrating CloverLeaf's Fortran codebase directly with CUDA's C bindings is difficult. A global class was written to handle interoperability between the Fortran and CUDA codebases 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 of CloverLeaf's kernels. The implementation of each was split into two parts: (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 thus ensure that branching operations are always performed on the host instead of the attached GPU. This ensures that the device-side kernels avoid stalls and maintain a high level of performance. Intra-node reduction operations were implemented using the Thrust library in CUDA, and using the built-in reduction primitive in OpenACC.

C. Optimisations

This section outlines the techniques we examined to improve the communication behaviour of CloverLeaf, our focus was therefore on the *halo-exchange* routine. This is executed several times during each iteration of the application and performs halo-exchanges of processes' boundary cells with neighbouring processes. At each halo-exchange the boundary cells of multiple fields (2D-arrays each representing a particular physical property e.g. density) can be exchanged, at varying halo depths (1, 2…etc cells), depending on the requirements of the algorithm at that stage of its computation.

In the reference version of CloverLeaf the halo-exchange routine employs an approach which exchanges the boundary cells from the required fields, one field at a time. The algorithm first conducts an exchange between processes in the horizontal dimension, then after this is completed it repeats the exchange in the vertical dimension. After the vertical exchange is completed the algorithm repeats the entire process for the next field which requires boundary cells to be exchanged. Figure 4 documents this process in pseudo code.

We initially, were possible, used each of the techniques listed below in isolation to implement alternative versions of CloverLeaf, additionally we also combined several of these techniques to produce further versions of the application. The alternative versions which we implemented are presented in Section V where their performance is

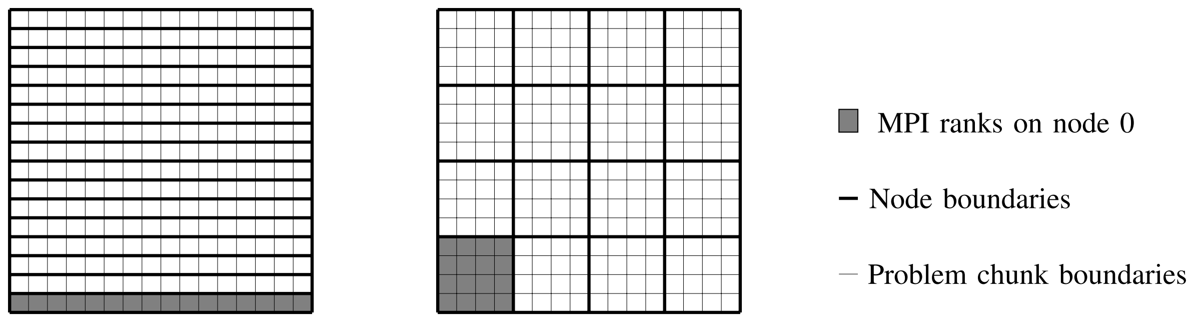
```
DO for all fields requiring boundary exchange

  IF (process has a left neighbour)
     Pack left MPI send buffer
     CALL MPI_ISEND/MPI_IRECV to the left
  ENDIF
  IF (process has a right neighbour)
     Pack right MPI send buffer
     CALL MPI_ISEND/MPI_IRECV to the right
  ENDIF

  CALL MPI_WAITALL
  Unpack left & right receive buffers

  IF (process has a bottom neighbour)
     Pack bottom MPI send buffer
     CALL MPI_ISEND/MPI_IRECV downwards
  ENDIF
  IF (process has a top neighbour)
     Pack top MPI send buffer
     CALL MPI_ISEND/MPI_IRECV upwards
  ENDIF

  CALL MPI_WAITALL
  Unpack bottom & top receive buffers
ENDDO
```

Fig. 4: CloverLeaf's *halo_exchange* routine

analysed using a standard benchmark problem from the CloverLeaf suite.

1) MPI rank reordering: By default on Cray systems *(Mpich_Rank_Reorder_Method=1)* cores within a job allocation are numbered consecutively within a node and this numbering continues on subsequent nodes, the number of a core corresponds to the MPI rank which will ultimately be executed on it. Within CloverLeaf chunks of the 2D-computational mesh are assigned to MPI ranks by traversing the mesh first in the *x*-dimension starting in the lower left corner of the mesh. Once one row of chunks has been completely assigned the allocation process restarts from the chunk on the left-hand side of the mesh which is one row up in the *y*-dimension from the previous row, and again proceeds along the *x*-dimension. The allocation process continues until all chunks of the mesh have been completely assigned.

This results in a chunk-to-node mapping with does not reflect the 2D nature of the overall problem and therefore is unable to take full advantage of the physical locality inherent in it. For jobs of above 16 processes, for example, communications in the *y*-dimension are all off-node and each process has a maximum of 2 neighbouring processes on its local node. A disproportional number of chunks are therefore co-located on nodes which are not physically close within the computational mesh. To improve this situation we implemented an alternative mapping strategy which better reflects the 2D communi-



(a) Original strategy

(b) Improved strategy

Fig. 5: MPI rank reordering strategy.

cation pattern inherent in the problem and attempts to increase *on-node* communications whilst reducing *off-node* communications. Our re-ordering approach assigns blocks of 4×4 chunks to each node, again starting in the lower left corner of the mesh.

We selected this blocking size as we generally use the nodes of Cray systems (which contain the AMD Interlagos processor) in a half-packed manner which results in 16 processes per node. Figure 5 shows chunk-to-node mappings for a 256 process allocation using the original and modified mapping strategies. This results in each 4×4 block (i.e. 1 node) having a maximum of 16 *off-node* neighbours compared to a maximum of 34 neighbours when the original mapping approach was employed.

2) One synchronisation per direction: The approach employed in the reference implementation of the *halo-exchange* routine results in two *MPI_Waitall* statements being executed for each field whose boundary cells need to be exchanged. This results in multiple synchronisations occurring between the communicating processes (two for each field) when boundary cells from multiple fields need to be exchanged during one invocation of the *halo-exchange* routine. These additional synchronisations are unnecessary as the boundary exchanges for each field are independent within each dimension (horizontal and vertical). It is therefore possible to restructure the *halo-exchange* routine to perform the horizontal boundary exchanges for all fields simultaneously, followed by only one synchronisation and then repeat this in the vertical dimension. This approach results in no more than two synchronisation operations per invocation of the *halo-exchange* routine whilst retaining the one MPI call/message per field approach.

3) Message Aggregation: The reference implementation of the *halo-exchange* routine utilises shared communication buffers, one for each direction within the 2D-space. This approach works as the boundary cells of only one field are exchanged simultaneously and therefore the MPI buffers can subsequently be reused by multiple fields. Buffer sharing is not possible when fields are exchanged simultaneously and each therefore requires its own MPI communication buffers one for each direction.

Message aggregation aims to reduce the number of MPI send/receive calls and also the number of MPI communication buffers by aggregating messages which would previously have used separate buffers into fewer but larger buffers. We applied this technique to the versions of CloverLeaf which send multiple messages simultaneously in each direction by aggregating all the messages into larger buffers (one for each communication direction). This has the effect of reducing the number of MPI send and receive calls to one per direction.

4) Pre-posting MPI receives: Previous studies have shown performance benefits from pre-posting MPI receive calls before the corresponding send calls [35]. In the reference *halo-exchange* implementation routine all MPI send calls are executed before their corresponding receive calls. We therefore applied this technique to CloverLeaf and created several versions which pre-post their MPI receive calls as early as possible. For most versions it was possible to completely remove the MPI receive calls from the *halo-exchange* routine and execute them before the computation kernel which immediately precedes the call to *halo-exchange*. Thus ensuring that a sufficient amount of computation occurs between each pre-posted MPI receive call and the execution of its corresponding send call.

5) Diagonal Communications: CloverLeaf's reference *halo-exchange* routine requires the horizontal communications between processes to be completed before the vertical communications in order to achieve an implicit diagonal communication between processes which are diagonal neighbours. It therefore enforces an explicit synchronisation between the horizontal and vertical communications. The requirement for this synchronisation could be removed if explicit diagonal communications were implemented between processes. This approach requires additional MPI buffers for the diagonal communications and additional MPI communication operations (sends and receives) to be initiated between processes. However, it removes the requirement for the explicit synchronisation between the horizontal and vertical communication phases and thus enables all communications to occur simultaneously in all directions, with only one synchronisation at the end of the *halo-exchange* routine.

6) CAF One-sided Gets: The existing CAF based implementations of CloverLeaf each utilise CAF one-sided *"put"* operations within the *halo-exchange* routine to communicate data items from the host process to its neighbouring processes. Other studies have shown that CAF one-sided *"get"* operations can actually deliver a performance improvement over their push-based equivalents. We therefore reimplemented the CAF-based buffer exchange version of CloverLeaf to utilise one-sided *"get"* operations whilst retaining the same overall approaches described in Section IV-A.

7) Sequential Memory plus MPI Datatypes: To evaluate the utility of: i) MPI's Derived Datatypes and ii) the technique of removing MPI communication buffers when data items are already stored contiguously; for improving the performance of the communication operations within CloverLeaf, we evaluated these optimisations within several versions of the mini-application. We removed all communication buffers which were used for communicating data items which were already stored contiguously in memory and instead employed MPI communication operations directly on the 2D-arrays within CloverLeaf. In situations in which data items were stored non-contiguously (i.e. with a stride) we utilised *MPI_Type_vector* Datatypes to communicate these values. These techniques enabled us to avoid having to explicitly allocate and pack MPI communication buffers within the code.

8) Actively checking for message arrivals: In the original implementation of CloverLeaf *MPI_WaitAll* operations are employed to provide synchronisation and thus prevent the computation from proceeding until all preceding communication operations are complete. In theory this may result in processes stalling whilst they wait for

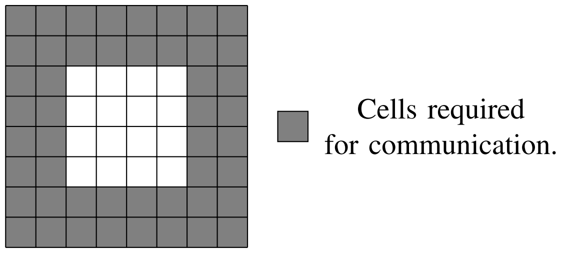


Fig. 6: Cell calculation order for comms/comp overlap

messages to be received or sent, when in fact they could be operating on (e.g. unpacking) messages which have already been received. This problem becomes more pronounced in subsequent versions of CloverLeaf which communicate large numbers of messages simultaneously. To address this issue we utilised *MPI_TestAny* operations to replace the previous *MPI_WaitAll* operations. This enables the mini-application to continuously check for the arrival of messages and upon receipt of a message immediately call the corresponding unpack routine to deal with the message. It was only possible to apply this optimisation to versions of the mini-application which do not utilise the "sequential memory plus MPI Datatypes" techniques identified in Section IV-C7.

9) Overlapping Communication and Computation: The reference implementation of CloverLeaf employs the Bulk Synchronous Parallel (BSP) model in which computation and communication phases are separated. In CloverLeaf the communication phase of the BSP model is implemented by the *halo-exchange* routine. Existing studies have shown that overlapping communication and computation can deliver performance and scalability improvements by enabling applications to utilise the hardware resources of HPC systems in a less "bursty" manner. The data-parallel nature of the computational kernels within CloverLeaf enable us to reorder the loop iterations within the kernels, which immediately precede communication events. This allows the outer halo of cells (which need to be communicated) to be computed first before the inner region of cells, which are only required on the host process. Figure 6 depicts this arrangement.

To create versions of CloverLeaf which overlap computation and communication operations we brought the MPI calls inside the computational kernels which immediately precede the particular communication events. Then once the outer halo of cells have been computed we employ non-blocking MPI calls to initiate the data transfers to the appropriate neighbouring processes. Whilst these transfers are taking place the computational kernel completes the remaining calculations to update the inner region of

	Titan	HECToR
Cray Model	XK7	XE6
Cabinets	200	30
Peak	20+ PF	800+ TF
Processor	AMD Opteron 6274	AMD Opteron 6276
Proc Clock Speed	2.2 GHz	2.3 GHz
GPU	NVIDIA K20x	N/A
Compute Nodes	18,688	2,816
CPUs/Node	1	2
GPUs/Node	1	0
Total CPUs	18,688	5,632
Total GPUs	18,688	0
CPU Memory/Node	32GB	32 GB
GPU Memory/Node	6GB	N/A
Interconnect	Gemini	Gemini
Compilers	Cray CCE v8.1.2	Cray CCE v8.1.2
MPI	Cray MPT v5.5.4	Cray MPT v5.6.1
CUDA	CUDA Toolkit v5.0.35	N/A

TABLE I: Summary of Titan & HECToR platforms

cells, with these computations being fully overlapped with communication operations. This approach relies on diagonal communication operations as described in Section IV-C5.

V. Results

To assess the current performance of the CloverLeaf mini-application and the success of our optimisations we conducted a series of experiments using two Cray platforms with different architectures, Titan and HECToR. The hardware and software configuration of these machines is detailed in Table I.

In our experiments CloverLeaf was configured to simulate the effects of a small, high-density region of ideal gas expanding into a larger, low-density region of the same gas, causing a shock-front to form. The configuration can be altered by increasing the number of cells used in the computational mesh. This increase in mesh resolution increases both the runtime and memory usage of the simulation. In this study we focused on two different problem configurations from the standard CloverLeaf benchmarking suite. We used the 3840^2-cell problem executed for 87 timesteps to examine the weak scaling performance of CloverLeaf on the two Cray machine architectures available to us, using several alternative programming models; the results of these experiments are detailed in Section V-A. Additionally we also used the 15360^2-cell problem executed for 2955 timesteps, strong-scaled to large processor counts, to analyse the success of each of our communication-focused optimisations; the results of these experiments are analysed in Section V-B.

Abbreviation	Description
RR	MPI Rank reordering
MA	Message Aggregation
MF	One synchronisation per direction
DC	Diagonal Communications
PP	Pre-posting MPI receives
O	Comms-Comp Overlap
TU	Actively checking for messages
SM&DT	Sequential Memory plus MPI Datatypes

TABLE II: Optimisation Abbreviations

A. *Weak-Scaling Experiments*

We evaluated the performance of CloverLeaf on both the "standard" CPU-based Cray XE6 architecture (HECToR) and the hybrid (CPU+GPU) Cray XK7 architecture of Titan. The aim of these experiments was to assess the suitability of each of these architectures for executing the hydrodynamics applications CloverLeaf represents. Additionally we sought to evaluate the performance of two candidate programming models for GPU-based architectures: MPI+OpenACC and MPI+CUDA.

We conducted experiments on a range of job sizes from 1 node up to the maximum job size on both machines (2048 and 16384 nodes respectively for HECToR and Titan). In all cases we compared the total application wall-time for the particular architecture and programming model combination, Figure 7 shows the results of these experiments.

The data in Figure 7 demonstrate that under a weak-scaling experimental scenario the scalability of CloverLeaf is already impressive. With the overall wall-time increasing by only 2.52s (4.2%) on HECToR between the 1 and 2048 node experiments and by only 4.99s (16.7%) and 4.12s (27.2%) for the MPI+OpenACC and MPI+CUDA versions respectively, as the application is

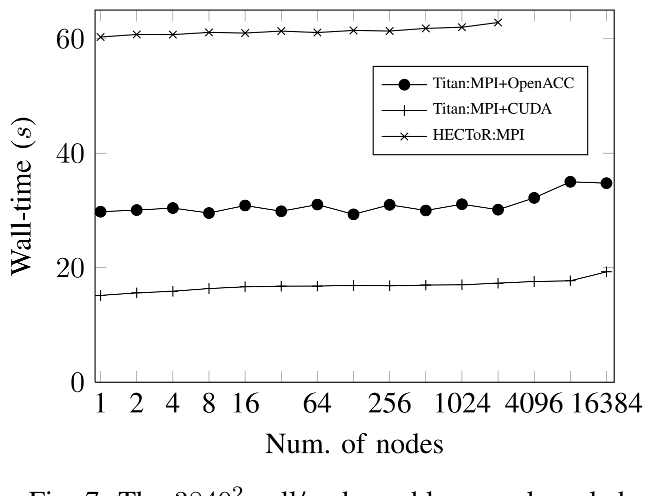


Fig. 7: The 3840^2 cell/node problem weak scaled

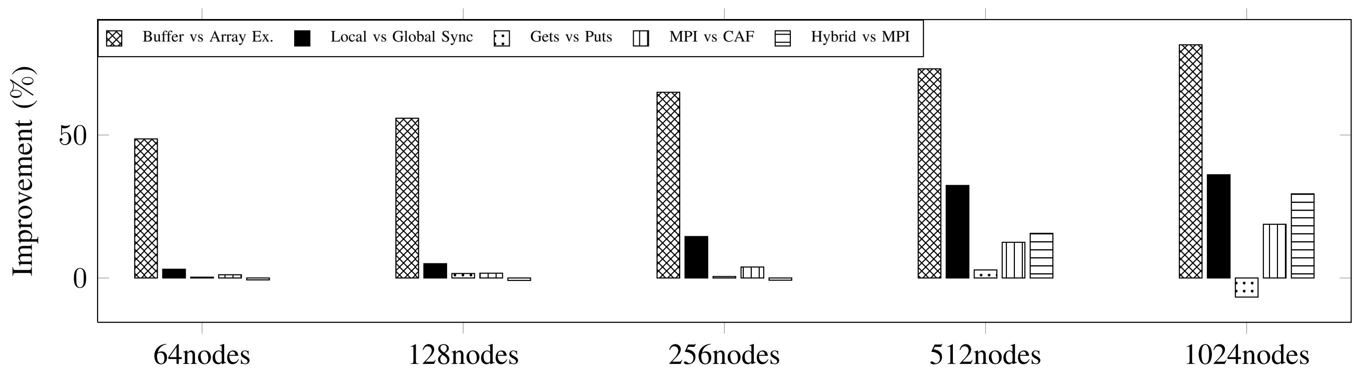


Fig. 8: Performance of CAF and Hybrid versions relative to the reference MPI implementation

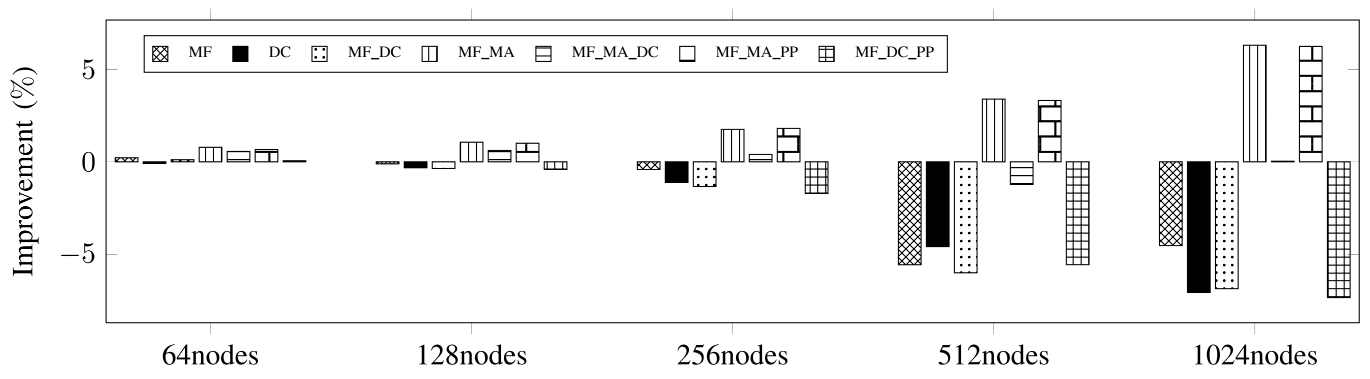


Fig. 9: Relative performance of the Message Aggregation, Multi-fields, Diagonal Comms & Pre-posting optimisations

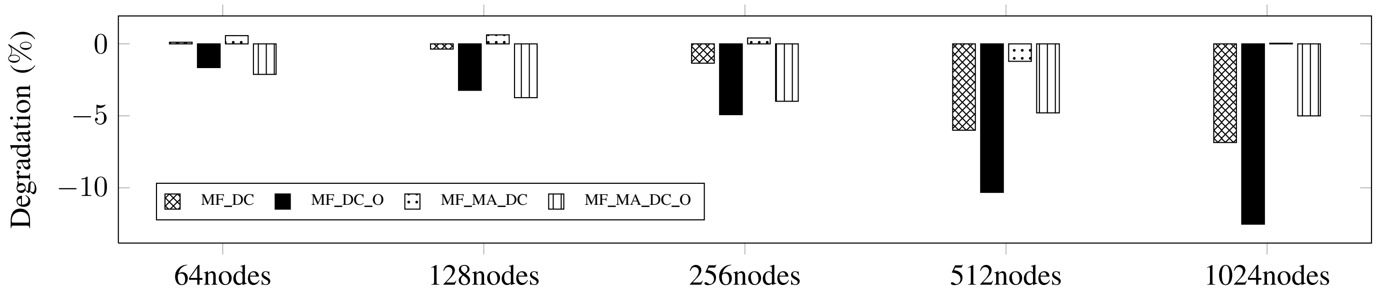


Fig. 10: Relative performance of the Computation-Communication Overlap optimisation

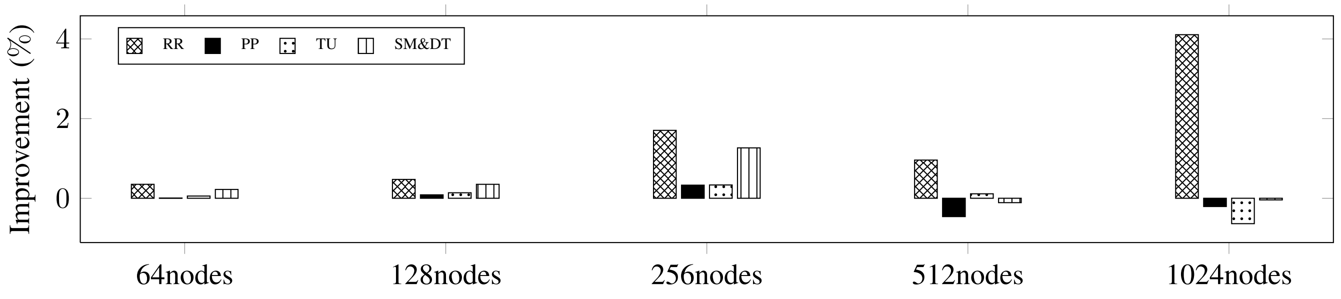


Fig. 11: Relative performance of the Rank Re-ordering, Pre-posting, Test-Unpack & Datatypes optimisations

scaled from 1 up to 16384 nodes on Titan.

Figure 7 also shows that the hybrid GPU-based XK7 architecture of Titan is able to outperform the CPU-based XE6 architecture of HECToR consistently by over $2\times$ on average under the OpenACC programming model and $3.7\times$ on average when the CUDA programming model is employed. This demonstrates the utility of the GPU-based architecture for accelerating this class of application especially as in the Titan experiments only the GPU devices are used for computational work. Additionally the Titan experiments show that the MPI+CUDA programming model outperforms the MPI+OpenACC model consistently by $1.8\times$ as the size of the experiments are increased from 1 up to 16384 nodes.

B. Strong-Scaling Experiments

To assess the performance of the various programming models used to implement CloverLeaf and the success of our communication-focused optimisations (discussed in Section IV-C) at scale. We conducted a series of strong-scaling experiments, on the HECToR platform, 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.

We again compared the overall wall-times of each version but present our results in terms of the relative performance (expressed as a %) of each particular version against a reference version, typically the original MPI implementation unless otherwise stated. Figures 8 to 11 contain the results of these experiments, for brevity we omit the results from jobs below 64 nodes. The mappings between the abbreviations used in these figures and the actual optimisations to which they correspond, are documented in Table II. Unfortunately, due to time constraints it has only been possible to conduct one run of each of these additional versions. However we feel that the longer running test problem employed in these experiments (2955 timesteps) limits the sensitivity of our results to system noise and other similar distorting effects. We leave the undertaking of these additional experiments to future work.

Figure 8 presents a comparison of the relative performance of the hybrid (MPI+OpenMP) version of CloverLeaf against the reference MPI implementation. This shows that overall the performance of both versions is broadly similarly up until 256 nodes, although the MPI version slightly outperforming the hybrid version by $<1\%$. Beyond 256 nodes, however, the hybrid version starts to significantly outperform the "flat" MPI version delivering performance improvements of 15.6% and 29.4% respectively at 512 and 1024 nodes. In our experiments the hybrid version was executed with 4 MPI tasks/node (1 per NUMA region) and 4 OpenMP threads per task.

We also compared the relative performance of the various CAF versions of CloverLeaf as well as the most performant CAF implementation, at each job size, against the reference MPI implementation (Figure 8). This analysis demonstrates that for the CAF versions the buffer exchange based strategy significantly outperforms its array-section based equivalent, with the performance disparity increasing as the scales of the experiments are increased, reaching 81% at 1024 nodes. Similarly the effect on performance of employing local synchronisation as opposed to global synchronisation also increases at scale, ranging from only 3% at 64 nodes to a 36% improvement at 1024 nodes. The effect of our CAF *"gets"*-based optimisation does initially deliver a modest performance improvement. However, at 1024 nodes the original *"puts"*-based implementation outperforms it by 6.7%. We speculate that this is due to the smaller messages exchanged at this job size and that *"gets"* are more suited to the larger message sizes exchanged at lower node counts. Overall no CAF implementation was able to improve on the performance of the reference MPI implementation, with the performance disparity increasing significantly as the scales of the experiments were increased, reaching 18% at 1024 nodes.

The results from our experiments with the communication-focused optimisations to the reference MPI implementation demonstrate the importance of selecting a process to network topology mapping which reflects the underlying communication pattern or physical geometry of an application. Figure 11 documents that the success of our MPI rank reordering optimisation increases as the scales of the experiments are increased, reaching a 4.1% relative improvement at 1024 nodes.

Of the other optimisations which we analysed the "message aggregation" technique proved to be the most successful, improving performance consistently by over 6% in the experiments which employed this technique in isolation (Figure 9). We speculate that this may also be a cause of a significant amount of the speedup which the hybrid version was able to achieve at large scale. This also exchanges fewer, larger, messages as a consequence of the coarser decomposition which it employs at the MPI level, however, we leave the confirmation of this hypothesis to future work.

Interestingly our analysis (Figure 9) shows that in both isolation and combination the "one synchronisation per direction" and "diagonal communications" techniques both had a detrimental affect on performance, consistently decreasing performance as the scales of the experiments were increased, reaching -4.5%, -7% and -6.9% respectively at 1024 nodes. Employing the "diagonal communication" technique in combination with the "message aggregation" technique also eliminated any performance gains, reducing performance to an almost identical level to the reference MPI implementation at 1024 nodes (Figure 9).

Additionally we conducted several experiments to analyse the success of our computation-communication overlapping technique. Figure 10 presents these results together with equivalent versions that do not employ this potential optimisation. Unfortunately rather than improving performance this technique consistently degraded performance. With the relative performance of both versions worsening as the scales of the experiments were increased, reaching -12% and -5% at 1024 nodes. At this job size both versions were approximately 5% slower than their equivalent version which did not employ the overlap technique. We speculate here that this performance degradation is due to the cache "unfriendly" nature of the memory access pattern which this techniques almost certainly causes, however we leave a more detailed analysis of this result to future work.

Furthermore Figures 9 and 11 demonstrate that in our experiments the "pre-posting" optimisation did not deliver a significant affect on overall performance as the versions which employed this technique performed almost identically when compared to equivalent versions which did not. Similarly the use of the "active checking for message arrivals" technique and the combined "sequential memory and MPI Datatypes" optimisation also had only minor affects on performance, slightly degrading relative performance on job sizes of $\geq$512 nodes, whilst delivering minor performance improvements on job sizes $<$512 nodes (Figure 11).

VI. Conclusions

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 thus achieve the required levels of performance. Similarly, 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.

The results presented here demonstrate the computational advantage which utilising GPU-based architectures such as Titan (XK7) can have over purely CPU-based alternatives such as HECToR (XE6). As the GPU-based architecture is able to consistently outperform the CPU-based architecture by as much as 3.7$\times$ in the weak-scaling experiments we conducted. Similarly, we have also shown an OpenACC based approach can deliver significant performance advantages on GPU-based architectures, consistently delivering 2$\times$ better performance when compared to the CPU-based equivalents with minimal additional programming effort compared to alternative approaches such as CUDA. However, a CUDA based approach is still required to achieve the greatest speedup from the GPU-based architecture, consistently delivering 1.8$\times$ the performance of the OpenACC based approach in our experiments.

Furthermore our strong scaling experiments demonstrate the utility of the hybrid (MPI+OpenMP) programming model. In our experiments the performance of both the hybrid and "flat" MPI implementations was broadly similar at scales $<$512 nodes. Beyond this point, however, the hybrid implementation delivers significant performance improvements reaching 29.4% at 1024 nodes.

Our experiments demonstrate the importance of selecting a process-to-node mapping which accurately matches the communication pattern inherent in the application, particularly at scale. Of the communication focused optimisations which we analysed as part of this work the "message aggregation" technique delivered the most significant performance improvement, with its effects again being more pronounced at large scale.

Surprisingly our "diagonal communications", "one synchronisation per direction" and "communication-computation overlapping" optimisations actually had a detrimental effect on performance. Whilst we did not observe any significant effects on performance from employing the "pre-posting", "actively checking for message arrivals" and "sequential memory plus MPI Datatypes" optimisations.

The CAF-based implementations were not able to match the performance of the reference MPI implementation. Additionally, particularly at scale, we observed significant performance differences between the various CAF implementations, which have important implications for how these constructs should be employed.

Overall, we feel that MPI is still the most likely candidate programming model for delivering inter-node parallelism going forward towards the exascale era. Although technologies such as CAF show promise they are not yet able—at least in our experiments—to match the performance of MPI. Hybrid programming approaches primarily based on OpenMP will also become increasingly important and can, as this work has shown, deliver significant performance advantages at scale. 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 to integrate some of our communications focused optimisations with the GPU targeted versions of the code (e.g. CUDA and OpenACC). With the aim of assessing whether, together with technologies such as NVIDIA's GPUDirect, these optimisations can deliver performance benefits on GPU based architectures such as the XK7.

We also plan to improve and generalise our MPI rank remapping work and build this into an external application which can be employed prior to the execution of the main application to optimise the rank-to-topology mapping within a given node allocation. Exploring alternative rank-to-topology mappings may also deliver 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. Furthermore, we also plan to investigate the Neighbourhood Collectives which have been standardised with MPI v3.0 and additional programming models such as SHMEM. As well as investigating alternative data structures to potentially alleviate the memory access pattern problems associated with the communication-computation overlap techniques employed here.

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 HECToR and Titan systems respectively. Additionally we would like to thank Alistair Hart of Cray. Also Simon McIntosh-Smith and Michael Boulton of the University of Bristol for developing the CUDA implementation of CloverLeaf and NVIDIA Corporation for their optimisations to it.

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] Dongarra, J. and Beckman, P. and Moore, T. and Aerts, P. and Aloisio, G. and Andre, J.C. and Barkai, D. and Berthou, J.Y. and Boku, T. and Braunschweig, B. and others, "The International Exascale Software Project roadmap," *International Journal of High Performance Computing Applications*, vol. 25, no. 1, 2011.

[2] Balaji, P. and Buntinas, D. and Goodell, D. and Gropp, W. and Kumar, S. and Lusk, E. and Thakur, R. and Träff, J., "MPI on a Million Processors," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009.

[3] 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.

[4] Herdman, J. and Gaudin, W. and McIntosh-Smith, S. and Boulton, M. and Beckingsale, D. and Mallinson,

A. and Jarvis, S., "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA," in *Proceedings of the 3rd International Workshop on Performance Modelling, Benchmarking and Simulation*, 2012.

[5] Mallinson, A. and Beckingsale, D. and Jarvis, S., "Towards Portable Performance for Explicit Hydrodynamics Codes," in *To be published in proceedings of the* 1^{st} *International Workshop on OpenCL*, 2013.

[6] Levesque, J. and Sankaran, R. and Grout, R., "Hybridizing S3D into an Exascale Application using OpenACC," *Proceedings of the 2012 ACM/IEEE conference on SuperComputing*, 2012.

[7] Preissi, R. and Wihmann, N. and Long, B. and Shalf, J. and Ethier, S. and Koniges, A., "Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms," *Super Computing*, 2011.

[8] Stone, A. and Dennis, J. and Strout, M., "Evaluating Coarray Fortran with the CGPOP Miniapp," *PGAS Conference*, 2011.

[9] Jones, P., "Parallel Ocean Program (POP) user guide," *Technical Report LACC 99-18, Los Alamos National Laboratory*, March 2003.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Henty, D., "Performance of Fortran Coarrays on the Cray XE6," *Cray User Group*, 2012.

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

[23] Demmel, J. and Hoemmen, M. and Mohiyuddin, M. and Yelick, K., "Avoiding Communication in Sparse Matrix Computations," *International Symposium on Parallel and Distributed Processing*, 2008.

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

[25] Numrich, R. and Reid, J., "Co-array Fortran for parallel programming," *ACM Sigplan Fortran Forum*, vol. 17, no. 2, pp. 1–31, August 1998.

[26] Barrett, R., "Co-Array Fortran Experiences with Finite Differencing Methods," *Cray User Group*, 2006.

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

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

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

[30] "The OpenACC Application Programming Interface version 1.0," http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2011.

[31] 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.

[32] "CAPS OpenACC Compiler The fastest way to many-core programming," http://www.caps-entreprise.com, November 2012.

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

[34] Lebacki, B. and Wolfe, M. and Miles, D., "The PGI Fortran and C99 OpenACC Compilers," in *Cray User Group*, 2012.

[35] Worley, P., "Importance of Pre-Posting Receives," in 2^{nd} *Annual Cray Technical Workshop*, 2008.