Experiences at scale with PGAS versions of a Hydrodynamics Application

A. C. Mallinson, S. A. Jarvis
Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
acm@dcs.warwick.ac.uk

W. P. Gaudin, J. A. Herdman
High Performance Computing
Atomic Weapons Establishment
Aldermaston, UK
{Wayne.Gaudin,Andy.Herdman}@awe.co.uk

ABSTRACT

In this work we directly evaluate two PGAS programming models, CAF and OpenSHMEM, as candidate technologies for improving the performance and scalability of scientific applications on future exascale HPC platforms. PGAS approaches are considered by many to represent a promising research direction with the potential to solve some of the existing problems preventing codebases from scaling to exascale levels of performance. 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 scales required to reach exascale levels of computational performance on future system architectures. We document our approach for implementing a recently developed Lagrangian-Eulerian explicit hydrodynamics mini-application in each of these PGAS languages. Furthermore, we also present our results and experiences from scaling these different approaches to high node counts on two state-of-the-art, large scale system architectures from Cray (XC30) and SGI (ICE-X), and compare their utility against an equivalent existing MPI implementation.

Categories and Subject Descriptors

[**Computing methodologies**]: Massively parallel and high-performance simulations; [**Software and its engineering**]: Software organization and properties—*Interoperability, Software performance*

General Terms

Languages, Performance, Measurement

Keywords

PGAS, Co-array Fortran, OpenSHMEM, MPI, HPC, Exascale, Hydrodynamics

1. INTRODUCTION

Due to power constraints it is recognised that emerging HPC system architectures continue to enhance overall computational capabilities through significant increases in hardware concurrency [14]. With CPU clock-speeds constant or even reducing, node-level computational capabilities are being improved through increased CPU core and thread counts. At the system-level, however, aggregate computational capabilities are being advanced through increased overall node counts. Effectively utilising this increased concurrency will therefore be vital if existing scientific applications are to harness the increased computational capabilities available in future supercomputer architectures.

It has also been recognised that the main memory capacity available per CPU core is likely to continue to decrease in future system architectures. Additionally for many applications memory bandwidth and latency are already the key resource constraints limiting performance. Potentially further necessitating the need to scale applications to higher node counts in order to utilise greater aggregate memory resources.

It is therefore highly likely that the ability to effectively scale applications across multi-petascale or exascale platforms will be essential if these classes of machine are to be utilised for improved science. Irrespective of the nodal hardware employed in a particular supercomputer architecture, there is a common requirement for improving the scalability of communication mechanisms within future systems. These trends present 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.

The established method of utilising current supercomputer architectures is based on an MPI-only approach, which utilises a two-sided model of communication for both intra- and inter-node communication. It is argued that this programming approach is starting to reach its scalability limits due to increasing nodal CPU core counts and the increased congestion caused by the number of MPI tasks involved in a large-scale distributed simulation [8].

Unlike *Message Passing*, PGAS (Partitioned Global Address Space) based approaches such as CAF (Co-array Fortran), OpenSHMEM or UPC (Unified Parallel C) rely on a lightweight one-sided communication model and a global memory address space. This model represents a promising area of research for improving the performance and scalability of applications as well as programmer productivity. Additionally they may also help to reduce the overall memory footprint of applications through e.g. the elimination of communication buffers, potentially leading to further performance advantages. The DARPA[1] recently funded the development of several PGAS-based languages through the HPCS[2] programme, including X10 [28], Chapel [11] and Fortress [6].

Historically, effectively utilising a PGAS-based approach often required the use of a proprietary interconnect technology, incorporating explicit hardware support, such as those commercialised in the past by Cray and Quadrics. Although the body of work which examines PGAS-based applications on these technologies is still relatively small, substantially less works exists which examines their performance on systems constructed from commodity-based technologies such as Infiniband. It is the view of the authors that this analysis may become increasingly important in the future given that Intel recently procured both the Cray Aries and QLogic Infiniband interconnect technologies and the potential for these technologies to converge within future Intel system-on-a-chip designs. Research is therefore needed to assess the relative merits of PGAS-based programming models and future hardware evolutions to ensure that the performance of scientific applications is optimised. To date, insufficient work has been conducted to directly evaluate both the CAF and OpenSHMEM models 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 are required to make significant investments maintaining their existing production scientific codebases. Production codes are usually legacy codes which are generally old, large and inflexible, with many man-years of development resources invested in them. This effort cannot be discarded and applications cannot be rewritten from scratch for the latest hardware. These organisations are therefore challenged with the task of deciding how best to develop their existing applications to effectively harness future HPC system architectures.

The optimal approach is unknown and likely to be application and machine architecture specific. Evaluating the strengths of each potential approach using existing production codes is problematic due to their size and complexity. Together with limited development and financial resources, this complexity means that it is not possible to explore every software development option for each HPC application. Decisions made now could also severely affect scientific productivity if the path chosen is not amenable to future hardware platforms. A rapid, low risk approach for investigating the solution space is therefore extremely desirable. 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 (mini-apps) are small, self contained programs that embody essential key algorithmic components and performance characteristics of larger more complex production codes [18]. Thus their use provides a viable way to develop and evaluate new methods and technologies.

Weak- and strong-scaling experimental scenarios are likely to be important on multi-petascale machines. The former are likely to scale well as the communication to computation ratio remains relatively 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 a shared, globally accessible 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.

In this work we utilise a simplified but still representative structured, explicit hydrodynamic mini-app known as CloverLeaf [18] to investigate the utility of these PGAS-based approach for this class of application. CloverLeaf forms part of the R&D Top 100 award winning Mantevo software suite [3]. We document our experiences porting the existing MPI codebase to CAF and OpenSHMEM and compare the performance and scalability of each approach, under a strong-scaling experimental scenario, on two state-of-the-art system architectures and vendor implementations.

Specifically, in this paper we make the following key contributions:

- We document the implementation of each of our OpenSHMEM and CAF versions of CloverLeaf; 10 and 8 distinct versions respectively. With the aim that this information will be useful to developers of future OpenSHMEM and CAF applications.

- We present a performance analysis of these versions, at considerable scale (up to 49,152 cores), to provide both a comparison of each programming model but also to assess how the communication constructs within each can best be incorporated into parallel applications. Based on these results we also make recommendations to improve the OpenSHMEM specification and potentially future CAF compiler and runtime systems.

- We provide a performance comparison of our PGAS versions against an equivalent MPI implementation in order to assess their utility against the dominant paradigm used in existing parallel scientific applications.

- Finally, we analyse the performance of our PGAS implementations on two systems incorporating different interconnect topologies and technologies. Namely the Cray Aries (Dragonfly) interconnect in the XC30 platform and the SGI 7D-hypercube Infiniband technology within the ICE-X platform.

The remainder of this paper is organised as follows: Section 2 discusses related work in this field. In section 3 we present background information on the hydrodynamics scheme employed in CloverLeaf and the programming models which this work examines. Section 4 describes the implementation of CloverLeaf in each of these programming models. The results of our study are presented and analysed in section 5, together with a description of our experimental setup. Finally, section 6 concludes the paper and outlines directions for potential future work.

[1]Defence Advanced Research Projects Agency
[2]High Productivity Computing Systems

2. RELATED WORK

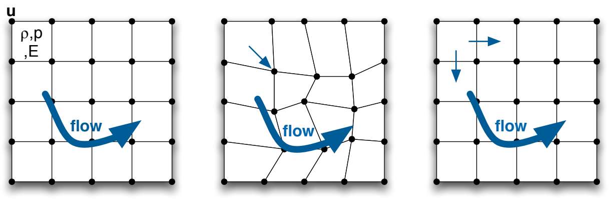
Although CAF has only relatively recently been incorporated into the official Fortran standard, earlier versions of the technology have existed for some time. Similarly several distinct SHMEM implementations have existed since the technology was originally developed by Cray in 1993 for its T3D supercomputers [5]. SHMEM has however only very recently been officially standardised under OpenSHMEM [4,12].

Consequently, a number of studies have already examined these technologies. 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. Additionally, relatively little work has been carried out to assess these technologies since their standardisation and on the hardware platforms we examine in this work. Overall, substantially less work exists which directly evaluates implementations of the MPI, OpenSHMEM and CAF programming models when applied to the same application. The results from these previous studies have also varied significantly, with some authors achieving significant speedups by employing PGAS-based constructs whilst others present performance degradations. 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 scaling CloverLeaf to large node counts on the Cray XE6 and XK7 [23]. Whilst this study did examine the CAF-based implementation of the application, as well as an OpenMP-based hybrid version, it focused on different hardware platforms to those we consider here; additionally it also did not include an examination of the OpenSHMEM-based implementation. Similarly, although we did examine the performance of CloverLeaf at scale, on the Cray XC30 (Aries interconnect) in [15], this work did not examine any of the PGAS-based versions of the codebase. Additionally we have also previously reported on our experiences of porting CloverLeaf to GPU-based architectures using OpenACC, OpenCL and CUDA [17,22].

Two studies which do directly evaluate the CAF and MPI programming models at considerable scale are from Preissl [26] and Mozdzynski [24]. Preissl *et al.* present work which demonstrates a CAF-based implementation of a Gyrokinetic Tokamak simulation code significantly outperforming an equivalent MPI-based implementation on up to 131,000 processor cores. Similarly Mozdzynski *et al.* document their work using CAF to improve the performance of the ECMWF IFS weather forecasting code, relative to the original MPI implementation, on over 50,000 cores. Both studies, however, examine significantly different classes of application, a 3D Particle-In-Cell code and a semi-Lagrangian weather forecast code, respectively.

Stone *et al.* were, however, unable to improve the performance of the MPI application on which their work focused by employing the CAF constructs, instead experiencing a significant performance degradation [29]. Their work, however, focused on the CGPOP mini-application, which represents the Parallel Ocean Program [19] from Los Alamos National Laboratory. Whilst the application examined by Lavallée *et al.* has similarities to CloverLeaf, their work compares several hybrid approaches against an MPI-only based approach [21], additionally they focus on a different hardware platform and do not examine either CAF- or OpenSHMEM-based approaches. Henty also provides a comparison between MPI and CAF using several microbenchmarks [16].



(a) Staggered grid (b) Lagrangian step (c) Advective remap

Figure 1: One CloverLeaf timestep

The OpenSHMEM programming model was shown to deliver some performance advantages relative to MPI by Bethune *et al.* in [10]. However, their work examined Jacobi's method for solving a system of linear equations and they utilised a previous generation of the Cray architecture (XE6) in their experiments. In [27], Reyes *et al.* discuss their experiences porting the GROMACS molecular dynamics application to OpenSHMEM. Their experiments, however, showed a small but significant performance degradation relative to the original MPI implementation, additionally they also utilised the Cray XE6 architecture.

Baker *et al.* looked at a hybrid approach using OpenACC within a SHMEM-based application [7]. They concentrate however on hybridising the application using OpenACC and their results are also focused on the XK7 architecture (Titan). Jose *et al.* studied the implementation of a high performance unified communication library that supports both the OpenSHMEM and MPI programming models on the Infiniband architecture [20]. Minimising communication operations within applications has also been recognised as a key approach for improving the scalability and performance of scientific applications [13].

3. BACKGROUND

CloverLeaf was developed with the purpose of assessing new technologies and programming models both at the inter- and intra-node system levels. It is part of the Mantevo mini-applications test suite [18], which was recognised as one of the 100 most technologically significant innovations in 2013, by R&D Magazine [2]. In this section we provide details of its hydrodynamics scheme and an overview of the programming models examined in this study.

3.1 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 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 1a) in which each cell centre stores 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 being advanced to the same point in time by the end of each computational step. One iteration, or timestep, of CloverLeaf proceeds as follows (see figure 1):

1. 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;

2. an advection step restores the cells to their original positions and calculates the amount of material which 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 [30]. 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 processes within the application, in a manner which attempts to minimise the communication surface area between processes. Whilst simultaneously attempting to assign a similar number of cells to each process to balance computational load. 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. Data exchanges occur multiple times during each timestep, between logically neighbouring processes within the decomposition, and with varying depths. A global reduction operation is required by the algorithm during the calculation of the minimum stable timestep, which is calculated once per iteration. The computational intensity per memory access in CloverLeaf is low which generally makes the code limited by memory bandwidth and latency speeds.

3.2 Programming Models

The following sections provide background information on each of the programming models examined in this study.

3.2.1 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 [1].

MPI programs are based on the SPMD (Single Process Multiple Data) paradigm in which each process (or rank) asynchronously executes a copy of the same program, but is able to follow different execution paths within the program. Each process makes calls directly into the MPI library in order to make use of the communication functions that it provides.

The technology is able to express both intra- and inter-node parallelism, with current implementations generally use optimised shared memory constructs for communication within a node. These *Message Passing* based communications are generally two-sided, meaning that explicit application-level management is required of both sides of the communication.

3.2.2 OpenSHMEM

The OpenSHMEM programming model was originally developed by Cray for their T3D systems. Although it is a relatively old model, it was only recently standardised in 2012 as part of the OpenSHMEM initiative [5]. Under the OpenSHMEM programming model, communications between processes are all one-sided and are referred to as *"puts"* (remote writes) and *"gets"* (remote reads). The technology is able to express both intra- and inter-node parallelism, with the latter generally requiring explicit RMA (Remote Memory Access) support from the underlying system layers. These constructs also purport to offer potentially lower latency and higher bandwidth than some alternatives.

OpenSHMEM is not explicitly part of the Fortran/C languages and is implemented as part of a library alongside these existing sequential languages. Processes within OpenSHMEM programs make calls into the library to utilise its communication and synchronisation functionality, in a similar manner to how MPI libraries are utilised. The programming model is much lower-level than other PGAS models such as CAF and enables developers to utilise functionality significantly closer to the actual underlying hardware primitives. It also makes considerably more functionality available to application developers.

The concept of a symmetric address space is intrinsic to the programming model. Each process makes areas of memory potentially accessible to the other processes within the overall application, through the global address space supported by the programming model. It is generally implementation-dependent how this functionality is realised, however it is often achieved using collective functions to allocate memory at the same relative address on each process.

Only a *global* process synchronisation primitive is provided natively. To implement *point-to-point* synchronisation it is necessary to utilise explicit *"flag"* variables, or potentially use OpenSHMEM's extensive locking routines, to control access to globally accessible memory locations. The concept of memory *"fences"*, which ensure the ordering of operations on remote processes' memory locations, are also intrinsic to the programming model. Collective operations are part of the standard, although currently no all-to-one operations are defined, just their all-to-all equivalents.

3.2.3 CAF

Several CAF extensions have been incorporated into the Fortran 2008 standard. These extensions were originally proposed in 1998 by Numrich and Reid as a means of adding PGAS (Partitioned Global Address Space) concepts into the main Fortran language, using only minimal additional syntax [25]. The additions aim to make parallelism a first class feature of the Fortran language.

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 the involvement of the remote processes. 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 [9]. In contrast to OpenSHMEM, CAF employs a predominantly compiler-based approach (no separate communications library), in which parallelism is explicitly part of the Fortran 2008 language. Consequently the Fortran compiler is potentially able to reorder the inter-image loads and stores with those local to a particular image.

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

4. 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. The kernels contain no subroutine calls and avoid using complex features such as Fortran derived types.

Twelve of CloverLeaf's kernels only perform computational operations. Communication operations reside in the overall control code and two additional kernels. One of these kernels 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 operation required for the calculation of the minimum timestep value.

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 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.

All of the MPI-, OpenSHMEM- and CAF-based versions of CloverLeaf implement the same block-structured decomposition of the overall problem domain. With each process responsible for a particular rectangular region of the overall computational mesh. As with the majority of block-structured, distributed, scientific applications, which solve systems of Partial Differential Equations, halo data is required to be exchanged between processes. To reduce synchronisation requirements, data is only exchanged when explicitly required by the subsequent phase of the algorithm, first in the horizontal and then in the vertical dimension. The depth of these halo exchanges also varies throughout the course of each iteration/timestep, depending on the numerical stencil required by each part of the algorithm. CloverLeaf v1.0, which is the basis for all versions examined in this work, employs a strategy in which one data-field is sent per communications operation, in exchanges involving multiple data-fields. It is in the implementation of these inter-process communication functions that the three broad implementation categories (MPI/SHMEM/CAF) of CloverLeaf involved in this study differ. The specific details of which are outlined in the following sections.

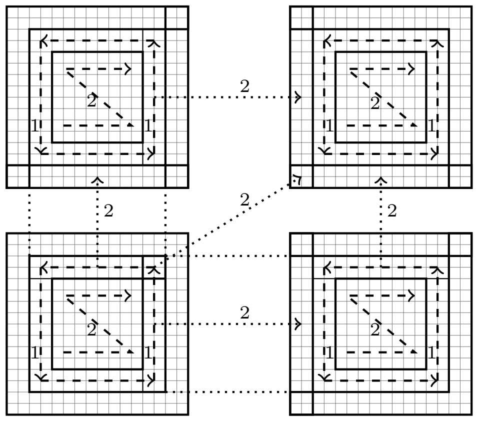


Figure 2: Cell calculation order for comms/comp overlap

To create versions of CloverLeaf which overlap communication and computation, the communication operations at a particular phase of the algorithm were moved inside the computational kernels which immediately preceded them. The data-parallel nature of the computational kernels within CloverLeaf enable us to reorder the loop iterations within these kernels. 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 2 depicts this arrangement. Once the outer halo of cells have been computed we employ non-blocking communication primitives to initiate the data transfers to the appropriate neighbouring processes. This approach relies on diagonal communication operations being implemented between diagonally adjacent processes in the computational mesh. Whilst these transfers are taking place the computational kernel completes the remaining calculations, with these computations being fully overlapped with the preceding communication operations.

4.1 MPI

The MPI-based versions of CloverLeaf perform their halo exchanges using the `MPI_ISend` and `MPI_IRecv` communication operations with their immediate neighbours. Communications are therefore two-sided, with `MPI_WaitAll` operations being employed to provide local synchronisation between the data exchange phases. Consequently no explicit global synchronisation operations (`MPI_Barrier` functions) are present in the hydrodynamics timestep.

To provide global reduction functionality between the MPI processes, the `MPI_AllReduce` and `MPI_Reduce` operations are employed. These are required for the calculation of the timestep value (dt) during each iteration and the production of periodic intermediary results, respectively. The MPI-based implementations therefore uses MPI communication constructs for both intra- and inter-node communications.

4.2 SHMEM

The OpenSHMEM-based versions of CloverLeaf employed in this study utilise one of two general communication strategies. These involve utilising the OpenSHMEM communication constructs to exchange data:

1. located within dedicated communication buffers. This data is generally aggregated from non-contiguous memory regions into one contiguous space. Before being written into the corresponding receive buffers on the neighbouring processes, using `shmem_put64` operations. Following synchronisation operations this data then has to be unpacked by the destination process.

2. directly between the original source and final destination memory addresses. To communicate data stored within multi-dimensional arrays `shmem_put64` operations are used to transmit contiguously stored data, with strided `shmem_iput64` operations being utilised to transmit data stored non-contiguously. In order to produce a functional implementation, we found it necessary to employ two separate calls to a `shmem_iput64` operation to transmit two columns of halo data items rather than one call to a `shmem_iput128` operation.

In section 5 versions which employ the first strategy contain the word *buffers* in their description, whereas versions which employ the second strategy are referred to as *arrays*. The two-dimensional data arrays and communication buffers are symmetrically allocated when necessary using the `shpalloc` operator. All other scalar variables and arrays which are required to be globally addressable are defined within Fortran `common` blocks to ensure they are globally accessible.

The only synchronisation primitive which OpenSHMEM provides natively is a global operation (`shmem_barrier_all`) which synchronises all of the processes involved. One of the OpenSHMEM versions involved in this study employs this synchronisation strategy, in section 5 this version has the word *global* in its description. All other versions employ a *point-to-point* synchronisation strategy in which a particular process only synchronises with its logically immediate neighbours. Integer *"flag"* variables are employed to achieve this, these are set on a remote process after the original communication operation completes. Either `shmem_fence` of `shmem_quiet` operations are utilised to ensure the ordering of these remote memory operations. Versions which employ `shmem_quiet` contain the word *quiet* within their description in section 5, all other versions employ the `shmem_fence` operation.

To prevent data access race conditions two methods of delaying process execution, until the associated *"flag"* variable is set, are examined. Versions either employ a call to `shmem_int4_wait_until` on the particular *"flag"* variable, these are referred to using *shmemwait* in their description in section 5. Alternative versions utilise an approach in which the *"flag"* variables are declared as `volatile` and processes perform "busy waits" on them until they are set remotely by the initiating process. Versions which employ this latter strategy have the word *volatilevars* in their descriptions in section 5.

The native OpenSHMEM collective operations `shmem_real8_sum_to_all` and `shmem_real8_min_to_all` were used to provide the global reduction facilities. The `shmem_sum_to_all` function was used despite the application only requiring a reduction to the master process. Two sets of symmetrically allocated `pSync` and `pWork` arrays are allocated for use with all the OpenSHMEM collective functions. These are initialised to the required default values using the Fortran `data` construct. The application alternates between each set of `pSync` and `pWork` arrays on successive calls to the OpenSHMEM collective operations.

4.3 CAF

The CAF implementations employed in this study all utilise one-sided asynchronous CAF *"put"* operations, in which the image responsible for the particular halo data items, remotely writes them into the appropriate memory regions of its neighbouring images. No equivalent receive operations are therefore required. Unless otherwise stated the top-level Fortran `type` data-structure (a *structure of arrays* based construct), which contains all data fields and communication buffers, is declared as a Co-array object. Additional versions examine the effect of moving the data fields and communication buffers outside of this derived-type data-structure and declaring them individually as Co-array objects. In section 5 of this paper, versions which employed this modified approach contain the word *FTL* in their description.

The CAF-based versions of CloverLeaf employed in this study utilise the same general communication strategies as the OpenSHMEM versions, which were outlined in section 4.2. Again versions which employ the communication buffer based strategy contain the word *buffers* in their description in section 5. Whereas versions which employ the direct memory access strategy contain the word *arrays*. In the versions which employ this latter strategy multi-dimensional Fortran array sections are specified in the *"put"* operations. These may require the CAF runtime to transmit data which is stored non-contiguously in memory, potentially using strided memory operations.

Synchronisation constructs are employed to prevent race conditions between the images. Each version can be configured to use either the global `sync all` construct or the local `sync images` construct between immediate neighbouring processes. The selection between these synchronisation primitives is controlled by compile-time pre-processor macros. In section 5, versions which employed the `sync all` synchronisation construct have the word *global* in their description; all other versions utilise `sync images`.

The CAF versions examined in this work employ the proprietary Cray collective operations to implement the global reduction operations. 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

	Archer	Spruce
Manufacturer	Cray	SGI
Model	XC30	ICE-X
Cabinets	16	16
Peak Perf	1.56PF	0.97PF
Processor	Intel Xeon E5-2697v2	Intel Xeon E5-2680v2
Proc Clock Freq	2.7GHz	2.8GHz
Cores / CPU	12	10
Compute Nodes	3,008	2,226
CPUs/Node	2	2
Total CPUs	6,016	4,452
Memory/Node	64GB	64GB
Memory Freq	1,833MHz	1,866 MT/s
Interconnect	Cray Aries	Mellanox IB-FDR
Topology	Dragonfly	7D-hypercube
Compilers	Cray CCE v8.2.6	Intel v14.0
MPI	Cray Mpich v6.3.1	SGI MPI v2.9
OpenSHMEM	Cray Shmem v6.3.1	SGI Shmem v2.9

Table 1: Experimental platform system specifications

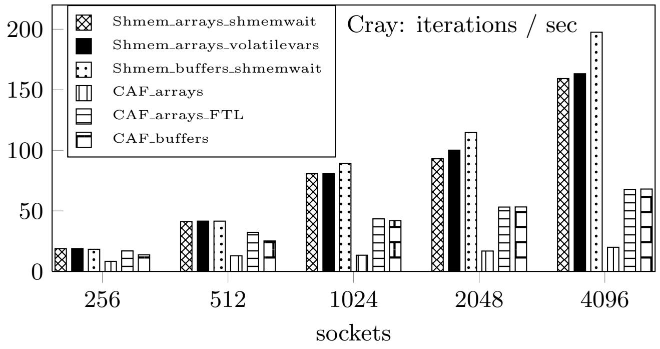
the purely CAF-based versions as we have not observed any noticeable performance differences between the CAF and MPI collectives on the Cray architecture.

5. RESULTS

To assess the performance of the CloverLeaf mini-application when expressed in each of the programming models examined here, we conducted a series of experiments using two distinct hardware platforms with significantly different architectures, a Cray XC30 (Archer) and an SGI ICE-X (Spruce). The hardware and software configuration of these machines is detailed in table 1. Both are based in the UK at the Edinburgh Parallel Computing Centre (EPCC) and AWE respectively.

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. Increasing the mesh resolution increases both the runtime and memory usage of the simulation. In this study we focused on a standard problem configuration from the CloverLeaf benchmarking suite. We used the $15{,}360^2$ cell problem executed for 2,955 timesteps and strong-scaled the simulation to large processor counts. The results of these experiments are analysed in sections 5.1 and 5.2. Unless otherwise noted, the results presented here are the averages of three repeated runs of the experiment to reduce the affects of system noise and jitter on our results.

For each job size examined we executed each version within the same node allocation to eliminate any performance effects due to different topology allocations from the batch system. We were able to conduct all of our experiments on the Spruce platform with the system in a fully dedicated mode, which should significantly reduce the affects of any system noise on our results. Unfortunately this was not possible on Archer, which we attribute to be the main cause of the performance disparities between the systems. We therefore do not present any direct performance comparisons between the two system architectures. Each version was configured to utilise the enhanced IEEE precision support for floating point mathematics, available under the particular compilation environment employed. On Archer all PGAS versions were also built and executed with support for 2MB huge



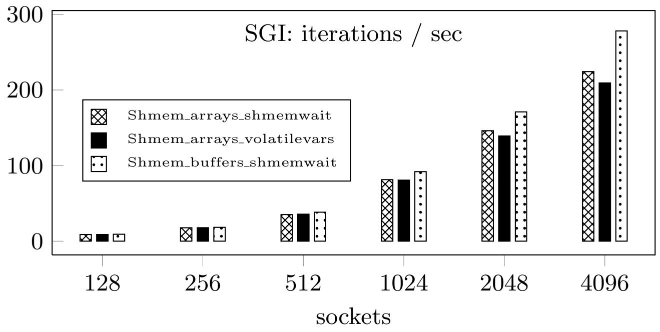


Figure 3: Array- and buffer-exchange based versions

memory pages enabled and 512MB of symmetric heap space available. Huge page support was not enabled for the standard MPI versions, as we have not previously observed any performance benefits for these versions. To again confirm this an additional execution of the standard MPI version, with this feature enabled, was completed as part of these experiments, however, for brevity these results are omitted.

Our interest in undertaking this work was to assess the potential of each of these programming models and techniques for improving (reducing) overall time to solution. In our experiments we therefore analysed their effectiveness by examining the runtime of the applications. For clarity the results presented here (figures 3 to 8) are expressed in terms of the number of sockets on which an experiment was conducted, and the application iterations / second which the particular version achieved (i.e. 2,955 / application wall-time).

5.1 First Strong-Scaling Results Analysis

The results from our experiments with the PGAS versions, which employ either the communications buffer or array-sections data exchange approaches, are shown in figure 3. These charts show the positive effect which employing communications buffers can have on both the Spruce and Archer platforms, particularly at high node counts. In our experiments on 4,096 sockets of Spruce the OpenSHMEM version, which employs communication buffers, achieved an average of 278.14 iterations/sec. An improvement of 1.2-1.3× over the equivalent array-section based approaches, which achieved 224.31 and 209.33 iterations/sec. The OpenSHMEM and CAF results from Archer also exhibit a similar pattern, at 4,096 sockets (49,152 cores) the communications buffer based OpenSHMEM version achieved 197.49 iterations/sec. Compared to the equivalent array-section based approaches which achieved only 159.23 and 163.24 respectively, an improvement of up to 1.24×. The CAF-based versions exhibit a significantly larger performance disparity,

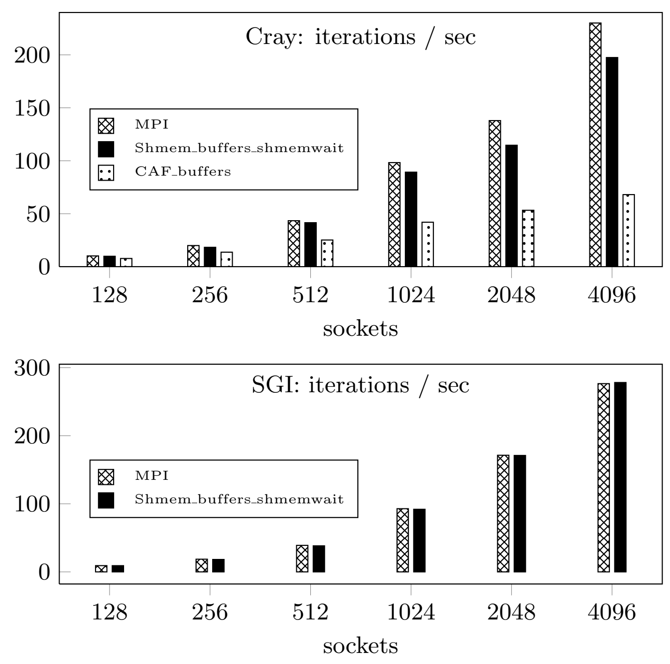


Figure 4: Equivalent MPI, OpenSHMEM and CAF versions

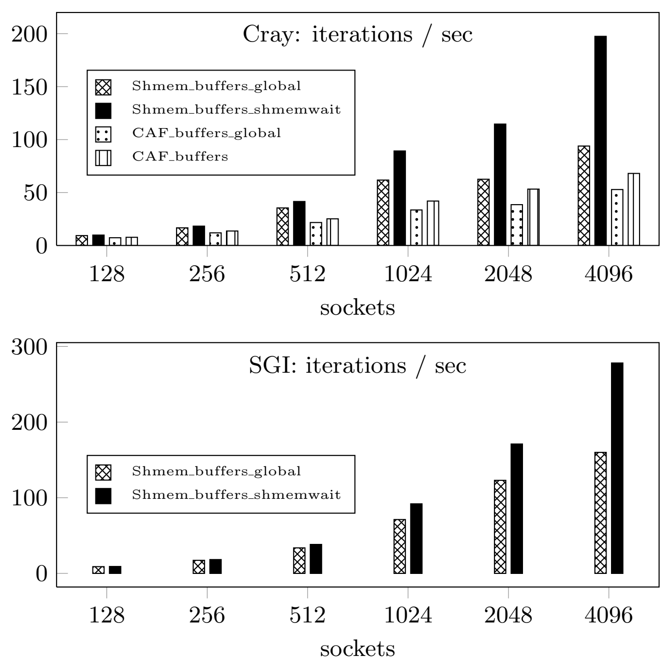


Figure 5: Local & global synchronisation versions

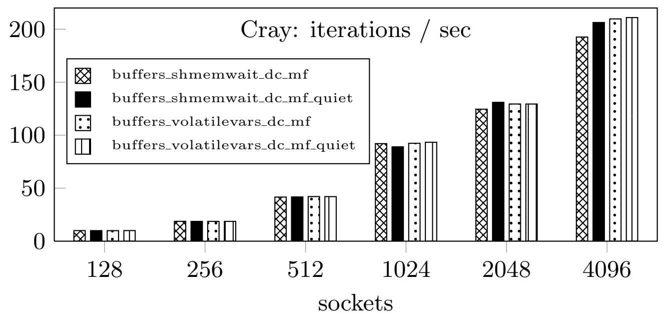
with the communication buffers approach achieving 3.4× the performance of the array-section based approach, 68.04 and 19.95 iterations/sec respectively. It should be noted that each application kernel, which contains only computation, is identical across these different versions (only the communication constructs differ), therefore given the same compiler environment is employed for all of the experiments on a particular system, performance due to e.g. automatic vectorisation should be constant throughout.

These results also show the performance improvement delivered by moving the data field definitions from within the original Fortran derived data type—defined as a Co-array—to be individual Co-array objects, each defined as top-level data structures. This optimisation (labeled *FTL*) improves the performance of the CAF array-section based approach by 3.39× (from 19.95 to 67.70 iterations/sec) at 4,096 sockets on Archer. It also enabled the array-section based approach to deliver equivalent performance to the communications buffer based approach in our experiments at 2,048 and 4,096 sockets, and to slightly exceed it in the 256 to 1,024 sockets cases.

Following a detailed inspection of the intermediate code representations produced by the Cray compiler, we believe that this is due to the compiler having to make conservative assumptions regarding the calculation of the remote addresses of the Co-array objects on other images. For each remote *"put"* within the *FTL* version, the compiler produces a single loop block containing one `__pgas_memput_nb` and one `__pgas_sync_nb` operation. In the original *array*-exchange version, however, the compiler generates three additional `__pgas_get_nb` and `__pgas_sync_nb` operations prior to the loop containing the *"put"* operation, a further set of these operations within this loop and an additional nested loop block containing a `__pgas_put_nbi` operation. Whilst it is not clear to us the precise function of each of these operations, as Cray does not publish this information. It would appear that due to the extra complexity of the original data-structure, e.g. the additional levels of indirection involved. The compiler is forced to insert *"get"* operations to the other images in order to retrieve the remote memory addresses to which an image should write the required data, despite these addresses remaining constant throughout the execution of the program. If this analysis is indeed correct, the creation of an additional compiler directive may prove to be useful here. This would enable developers to inform the compiler that the original data structure remains constant and therefore allow it to be less conservative during code generation.

Figure 4 shows the results from our experiments to assess the performance of our PGAS implementations against an equivalent MPI version. These charts document a significantly different performance trend on the two system architectures we examine here. The performance delivered on Spruce by both the OpenSHMEM and MPI implementations is virtually identical at all the scales examined (128 - 4,096 sockets), reaching 278.14 and 276.49 iterations/sec respectively on 4,096 sockets. On Archer, however, the performance of the two PGAS versions is not able to match that of the equivalent MPI implementation, with the performance disparity widening as the scale of the experiments is increased. Our OpenSHMEM implementation delivers the closest levels of performance to the MPI implementation and also significantly outperforms the CAF-based implementation. The results show 197.49 iterations/sec on 4,096 sockets compared to 230.08 iterations/sec achieved by the MPI implementation, an improvement of 1.17×. The CAF implementation, however, only delivers 68.04 iterations/sec on 4,096 sockets, a slowdown of 2.9× relative to the equivalent OpenSHMEM implementation.

To assess the affect of employing either the *global* or *point-to-point* synchronisation constructs on the performance of the PGAS versions, we analysed the results obtained on both experimental platforms from the OpenSHMEM and CAF ver-



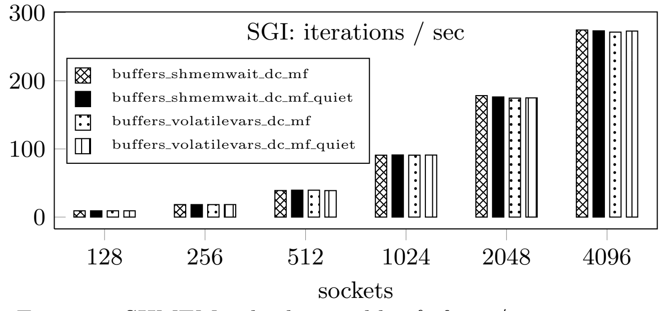


Figure 6: SHMEM volatile variables & fence/quiet versions

sions, which employed the communications buffer data exchange approach and either synchronisation construct. The OpenSHMEM versions examined here utilise the *shmem wait* approach to implement the *point-to-point* synchronisation. Figure 5 provides a comparison of the results obtained from each of these versions.

On both platforms it is clear that employing *point-to-point* synchronisation can deliver significant performance benefits, particularly as the scale of the experiments is increased. At 128 sockets there is relatively little difference between the performance of each version. On Spruce (1,280 cores) the OpenSHMEM implementation which employs *point-to-point* synchronisation achieved 9.13 iterations/sec, compared to 8.90 for the *global* synchronisation version. Whilst on Archer (1,536 cores) the *point-to-point* synchronisation versions of the OpenSHMEM and CAF implementations achieve 9.84 and 7.73 iterations/sec respectively, compared to the equivalent *global* synchronisation versions which achieve 9.34 and 7.31 iterations/sec. At 4,096 sockets (40,960 cores), however, the performance disparity between the two OpenSHMEM versions on Spruce increases to 278.13 and 159.97 iterations/sec, a difference of approximately 1.74×. The performance disparity between the OpenSHMEM versions is even greater on Archer, reaching 2.10× at 4,096 sockets (49,152 cores), with the *point-to-point* version delivering 197.49 iterations/sec and the *global* version 93.91. Interesting the CAF-based versions do not exhibit the same performance differences, with the *point-to-point* synchronisation version achieving only a 1.29× improvement (68.04 and 52.84 iterations/sec respectively). We speculate that this may be due to the performance of the CAF-based versions—which is significantly less than the OpenSHMEM-based versions—being limited by another factor and therefore the choice of synchronisation construct has a reduced, but still significant, affect on overall application performance.

The performance results obtained from several alternative versions of the OpenSHMEM implementation, on both the

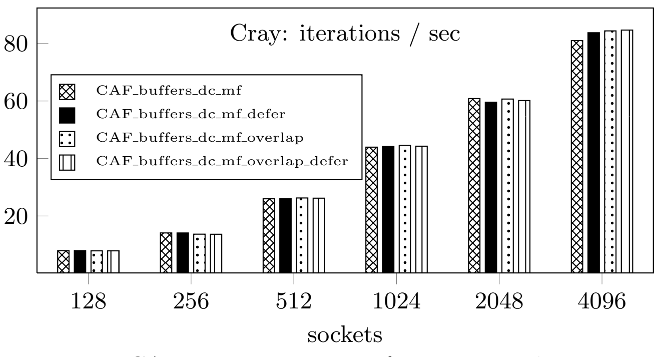


Figure 7: CAF `pgas defer_sync` & comms overlap versions

Cray and SGI platforms, are shown in figure 6. These charts compare versions which employ either the *shmem wait* or *volatile variables* synchronisation techniques and either the *quiet* or *fence* remote memory operation ordering constructs. All versions examined here employ diagonal communications (dc) between logical neighbouring processes, exchange multiple data fields simultaneously (mf) and employ a communications buffer-based approach to data exchange. It is evident from the charts that in our experiments the choice of each of these implementation approaches has no significant effect on overall performance. The results from both platforms show very little variation in the number of application iterations achieved per second as the scales of the experiments are increased. Although the Cray results do show some small variations at the higher node counts, we feel that this is likely due to the effects of system noise arising from the use of a non-dedicated system.

Figure 7 documents our experimental results on Archer from the CAF versions which employ our optimisations to overlap communications and computation together with the proprietary Cray `pgas defer_sync` directive. This purports to force the compiler to make all communication operations, in the next statement, non-blocking and to postpone the synchronisation of PGAS data until the next *fence* instruction is encountered, potentially beyond what the compiler can determine to be safe. The chart presents these results together with an equivalent CAF-based version which does not utilise any of these features. It shows that in our experiments the overall performance of CloverLeaf is not significantly affected (beneficially or detrimentally) by either of these potential optimisations, as the performance of all four versions is virtually identical in all cases.

5.2 Second Strong-Scaling Results Analysis

Following our results analysis in section 5.1, we conducted an additional set of experiments on Archer. Our aim was to examine the affect of: the proprietary Cray non-blocking SHMEM operations; employing 4MB huge-pages; and applying the *FTL* optimisation to the CAF buffer-exchange based version. We followed the same experimental methodology outlined in section 5 and our results can be found in figure 8. As these experiments were conducted at a different time (different system loads) and using different node allocations from the batch system, to our first set of experiments, the performance results between the two sets of experiments will differ, particularly at scale. We therefore only present performance comparisons within each set of experimental results rather than between them.

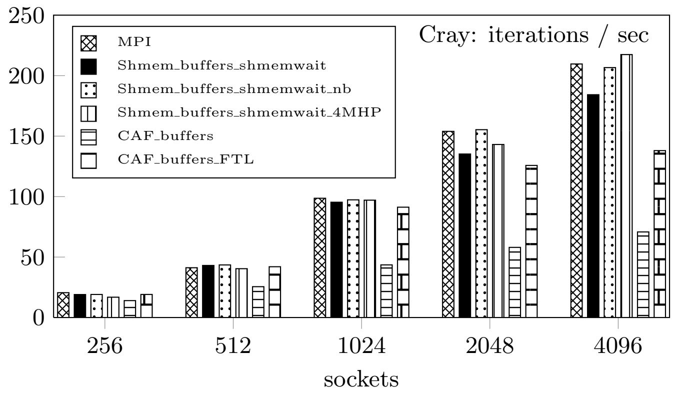


Figure 8: SHMEM non-blocking, huge-pages & CAF FTL

It is evident from figure 8 that the CAF buffer-exchange based version does indeed benefit significantly from the *FTL* optimisation. The modified version delivers substantially superior performance at all the node configurations examined, achieving 2.2× and 1.9× more iterations/sec during the 2,048 and 4,096 socket experiments, respectively. Although significantly improved, its performance still does not quiet match that of the OpenSHMEM-based equivalent particularly at large node counts. In the 4,096 socket experiment the OpenSHMEM buffer-exchange version achieved 184.23 iterations/sec compared to 138.01 for the CAF *FTL* version, an improvement of 1.33×. As in our previous set of experiments, the original OpenSHMEM version is not able to match the equivalent MPI implementation. It achieved 135.21 and 184.23 iterations/sec at 2,048 and 4,096 sockets respectively, compared to 153.92 and 209.65 for the MPI version.

In these experiments use of the proprietary Cray non-blocking operations and 4MB huge memory pages delivers some further performance benefits for the OpenSHMEM-based versions, particularly at high node counts. The performance of the non-blocking operations version is virtually identical to the original in the experiments ≤512 sockets. At 1,024 sockets and above, however, it starts to deliver significant performance advantages, achieving 206.66 iterations/sec at 4,096 sockets, compared to only 184.23 for the original version. In both the 2,048 and 4,096 socket experiments it also delivered broadly equivalent performance to the MPI implementation, achieving 155.31 and 206.66 iterations respectively, compared to 153.92 and 209.65 for the MPI version. The performance benefits from employing the larger 4MB huge memory pages are even more significant; in the 4,096 socket experiment this version achieved 217.42 iterations/sec. A 1.2× improvement over the original OpenSHMEM version and an improvement of 7.78 iterations/sec over the equivalent MPI implementation. Interestingly, however, its performance in the experiments below 1,024 sockets was slightly worse than the original OpenSHMEM version.

6. CONCLUSIONS AND FUTURE WORK

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. PGAS approaches such as OpenSHMEM or CAF, based on lighter-weight one-sided communication operations, represent a promising area of research to address this problem.

This work has evaluated the performance of equivalent OpenSHMEM, CAF and MPI based versions of CloverLeaf at considerable scale (up to 4,096 sockets/49,152 cores) on two significantly different, whilst still state-of-the-art, system architectures from two leading vendors. The results presented here demonstrate that the OpenSHMEM PGAS programming model can deliver portable performance across both the Cray and SGI system architectures. On the SGI ICE-X architecture it is able to match the performance of the MPI model, whilst delivering comparable—albeit surprisingly slightly slower—performance when compared to MPI on the Cray XC30 system architecture. Our experiments showed that the use of the proprietary Cray non-blocking operations enables the performance of the SHMEM-based versions to match and sometimes exceed that of their MPI equivalents, on the Cray architecture. We feel that the inclusion of these constructs together with “rooted” versions of some collective operations, in a future release of the OpenSHMEM standard, would be a useful improvement to the programming model. Additionally, we have also shown that the library-based PGAS model of OpenSHMEM can be significantly more performant than the equivalent language/ compiler based PGAS approaches such as CAF on the Cray XC30.

Our experiments demonstrated that applications based on either the OpenSHMEM or CAF PGAS paradigms can benefit, in terms of improved application performance, from the aggregation of data into communication buffers. In order to collectively communicate the required data items to the remote processes, rather than moving them directly using strided memory operations. The performance of CAF-based applications can also be sensitive to the selection of appropriate Co-array data structures within the application, as this can have implications for how these data-structures are accessed by remote memory operations. To potentially alleviate this problem we proposed the development of an additional compiler directive.

We also presented performance data documenting the performance improvements which can be obtained, for both OpenSHMEM- and CAF-based applications, by employing point-to-point synchronisation mechanisms rather than the alternative global synchronisation primitives. In our experiments our results showed that for OpenSHMEM-based versions of CloverLeaf, the choice of implementation mechanisms for the point-to-point synchronisation constructs (*shmem wait* or *volatile variables*), and the remote memory operation ordering constructs (*fence* and *quiet*), does not significantly affect the overall performance of the application. Similarly, the proprietary Cray CAF `pgas defer_sync` constructs and our optimisations to overlap communications and computation, do not significantly affect overall performance.

In future work, using our PGAS implementations of CloverLeaf, we plan to conduct additional experiments on the Archer XC30 platform including profiling the performance of each particular code variant (MPI, CAF and OpenSHMEM) to confirm the causes of the performance disparities documented in this work. We also plan to examine the one-sided communications constructs recently standardised within the MPI 3.0 specification. Additionally, running further experiments on both Spruce and Archer with different huge memory page and symmetric heap settings, would enable us to determine if any additional performance improvements can be achieved via these mechanisms. Similarly, for completeness, repeating the experiments with our CAF-based versions on Spruce, using the Intel CAF implementation, would be a potentially interesting research direction.

Analysing the overall memory consumption of our PGAS versions, when compared to the reference MPI implementation, maybe useful in determining whether these programming models deliver any advantages in terms of a reduction in memory consumption, e.g. due to the elimination of communication buffers. Hybridising all of our PGAS implementations of CloverLeaf with threading constructs such as OpenMP, to determine if the performance improvements which we have seen with the hybrid MPI-based versions can be replicated with the PGAS implementations, is another direction of research we plan to undertake. To determine if the performance trends we have seen in this work continue, we also plan to conduct larger-scale experiments on these and other platforms, including an Infiniband-based platform using the QLogic OpenSHMEM implementation. A longer-term goal of our research is to apply the PGAS programming models to applications which exhibit different communications characteristics (potentially irregular patterns) to the one we have studied here, to determine if they can deliver any improvements in performance for other classes of application.

7. 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 the Royal Society through their Industry Fellowship Scheme (IF090020/AM). We would also like to express our thanks to the UK Engineering and Physical Sciences Research Council for access to the Archer system.

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`.

8. REFERENCES

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

[2] Miniapps Pick Up the Pace. `http://www.rdmag.com/award-winners/2013/08/miniapps-pick-pace`, August 2013.

[3] Sandia wins three R&D 100 awards. `https://share.sandia.gov/news/resources/news_releases/`, July 2013.

[4] OpenSHMEM.org. `http://openshmem.org/`, July 2014.

[5] The OpenSHMEM PGAS Communications Library. `http://www.archer.ac.uk/community/techforum/notes/2014/05/OpenSHMEM-Techforum-May2014.pdf`, July 2014.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, L. Guy, J. Steele, and S. Tobin-Hochstadt. The Fortress Language Specification version 1.0. `http://homes.soic.indiana.edu/samth/fortress-spec.pdf`, March 2008.

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

[8] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. Träff. 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.

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

[10] I. Bethune, M. Bull, N. Dingle, and N. Higham. Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP. *Int. Journal of High Performance Computing Applications*, 28(1):97–111, Feb 2014.

[11] B. Chamberlain and Cray Inc. Chapel, 2013.

[12] B. Chapman, T. Curtis, P. Swaroop, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010.

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

[14] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J. Andre, D. Barkai, J. Berthou, T. Boku, and B. Braunschweig. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications*, 25(1), 2011.

[15] W. Gaudin, A. Mallinson, O. Perks, J. Herdman, D. Beckingsale, J. Levesque, and S. Jarvis. Optimising Hydrodynamics applications for the Cray XC30 with the application tool suite. In *The Cray User Group 2014, May 4-8, 2014, Lugano, Switzerland*, 2014.

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

[17] J. Herdman, W. Gaudin, D. Beckingsale, A. Mallinson, S. McIntosh-Smith, and M. Boulton. Accelerating hydrocodes with OpenACC, OpenCL and Cuda. In *High Performance Computing, Networking, Storage and Analysis (SCC) 2012 SC Companion*, 2012.

[18] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H. Edwards, A. Williams, M. Rajan, H. Keiter, E. Thornquist, and R. Numrich. Improving Performance via Mini-applications. Technical report, Sandia National Laboratories, 2009.

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

[20] J. Jose, K. Kandalla, L. Miao, and D. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. *41st*

International Conference on Parallel Processing (ICPP), 1(1):219–228, Sept 2012.

[21] P. Lavallée, C. Guillaume, P. Wautelet, D. Lecas, and J. Dupays. Porting and optimizing HYDRO to new platforms and programming paradigms - lessons learnt. `www.prace-ri.eu`, February 2013.

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

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

[24] G. Mozdzynski, M. Hamrud, N. Wedi, J. Doleschal, and H. Richardson. A PGAS Implementation by Co-design of the ECMWF Integrated Forecasting System (IFS). *High Performance Computing, Networking, Storage and Analysis(SCC), SC Companion*, 1(1):652–661, Nov 2012.

[25] R. Numrich and J. Reid. Co-array Fortran for parallel programming. *ACM Sigplan Fortran Forum*, 17(2):1–31, August 1998.

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

[27] R. Reyes, A. Turner, and B. Hess. Introducing SHMEM into the GROMACS molecular dynamics application: experience and results. In *Proceedings of PGAS 2013*, 2013.

[28] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification version 2.4. `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`, May 2014.

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

[30] B. van Leer. Towards the Ultimate Conservative Difference Scheme. V. A Second Order Sequel to Godunov's method. *Journal of Computational Physics*, 32(1), 1979.