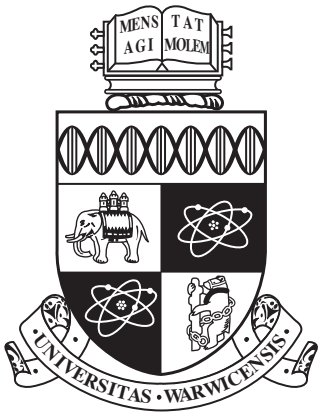


Evaluating Technologies and Techniques for Transitioning Hydrodynamics Applications to Future Generations of Supercomputers

by

Andrew Colin Mallinson

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

May 2016

Abstract

Current supercomputer development trends present severe challenges for scientific codebases. Moore's law continues to hold, however, power constraints have brought an end to Dennard scaling, forcing significant increases in overall concurrency. The performance imbalance between the processor and memory sub-systems is also increasing and architectures are becoming significantly more complex. Scientific computing centres need to harness more computational resources in order to facilitate new scientific insights and maintaining their codebases requires significant investments. Centres therefore have to decide how best to develop their applications to take advantage of future architectures. To prevent vendor *"lock-in"* and maximise investments, achieving portable-performance across multiple architectures is also a significant concern.

Efficiently scaling applications will be essential for achieving improvements in science and the MPI (Message Passing Interface) only model is reaching its scalability limits. Hybrid approaches which utilise shared memory programming models are a promising approach for improving scalability. Additionally PGAS (Partitioned Global Address Space) models have the potential to address productivity and scalability concerns. Furthermore, OpenCL has been developed with the aim of enabling applications to achieve portable-performance across a range of heterogeneous architectures.

This research examines approaches for achieving greater levels of performance for hydrodynamics applications on future supercomputer architectures. The development of a Lagrangian-Eulerian hydrodynamics application is presented together with its utility for conducting such research. Strategies for improving application performance, including PGAS- and hybrid-based approaches are evaluated at large node-counts on several state-of-the-art architectures. Techniques to maximise the performance and scalability of OpenMP-based hy-

brid implementations are presented together with an assessment of how these constructs should be combined with existing approaches. OpenCL is evaluated as an additional technology for implementing a hybrid programming model and improving performance-portability. To enhance productivity several tools for automatically hybridising applications and improving process-to-topology mappings are evaluated.

Power constraints are starting to limit supercomputer deployments, potentially necessitating the use of more energy efficient technologies. Advanced processor architectures are therefore evaluated as future candidate technologies, together with several application optimisations which will likely be necessary. An FPGA-based solution is examined, including an analysis of how effectively it can be utilised via a high-level programming model, as an alternative to the specialist approaches which currently limit the applicability of this technology.

Acknowledgements

The completion of this thesis, and the research work contained within it, was made possible by the support of a number of people. Their academic advice and personal support throughout my time at Warwick, has helped to maintain my research focus.

I would like to thank AWE plc for funding this work. Specifically Andy Herdman and Wayne Gaudin for providing guidance throughout; you have both been a pleasure to work for.

My beloved partner Ruth, to whom I am eternally grateful, deserves huge credit for all her love and support during my Ph.D. without which I would not have been able to complete this work. Additionally, for their continued and invaluable support throughout, I would also like to thank my Mum, Dad, Sister and Brother-in-law.

Within my research group at Warwick I am very grateful to Dr. Oliver Perks, David Beckingsale, Robert Bird, Dr. John Pennycook and James Davis for their advice and constructive critiques of my ideas. Finally, I would also like to thank Prof. Stephen Jarvis for accepting me onto the Ph.D. programme and for supervising my research.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree.

The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- The collection of the execution times of the OpenCL version of CloverLeaf on the Teller platform at SNL, was performed by Andy Herdman of AWE plc.

Parts of this thesis have been previously published by the author in the following:

1. A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, J. M. Levesque and S. A. Jarvis. CloverLeaf: Preparing Hydrodynamics Codes for Exascale, In *Proceedings of the Cray User Group 2013* (CUG), Napa Valley, USA, May 2013 [132].

2. A. C. Mallinson, D. A. Beckingsale, W. P. Gaudin, J. A. Herdman and S. A. Jarvis. Towards Portable Performance for Explicit Hydrodynamics Codes, In *Proceedings of the International Workshop on OpenCL 2013* (IWOCL), Atlanta, USA, May 2013 [131].

3. W. P. Gaudin, A. C. Mallinson, O. Perks, J. A. Herdman, D. A. Beckingsale, J. M. Levesque, M. Boulton, S. McIntosh-Smith and S. A. Jarvis. Optimising Hydrodynamics applications for the Cray XC30 with the application tool suite, In *Proceedings of the Cray User Group 2014* (CUG), Lugano, Switzerland, May 2014 [69]. Awarded best research paper.

4. A. C. Mallinson, W. P. Gaudin, J. A. Herdman and S. A. Jarvis. Experiences at scale with PGAS versions of a Hydrodynamics Application, In *Proceedings of the 8^{th} International Conference on Partitioned Global Address Space Programming Models* (PGAS2014), Eugene, Oregon, USA, Oct 2014 [133].

5. J. A. Herdman, W. P. Gaudin, D. A. Beckingsale, A. C. Mallinson, M. Boulton, S. McIntosh-Smith and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA, In *Proceedings of the 3rd International Workshop on Performance Modelling, Benchmarking and Simulation*, (PMBS12), Salt Lake City, Utah, USA, Nov 2012 [104].

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- UK Atomic Weapons Establishment, under grants:
  "The Production of Predictive Models for Future Computing Requirements" (CDK0660)
  "AWE Technical Outreach Programme" (CDK0724)
  "AWE CASE studentship" (ref. 30197965)

Abbreviations

ADRES	Architecture for Dynamically Reconfigurable Embedded Systems
AMD	Advanced Micro Devices
API	Application Programming Interface
APU	Accelerated Processing Unit
ASIC	Application Specific Integrated Circuit
AVX	Advanced Vector Extensions
AWE	Atomic Weapons Establishment, UK
BG/Q	Blue Gene Q
BSP	Bulk Synchronous Parallel
CAAR	Center for Application Acceleration Readiness
CAF	Co-array Fortran
CCE	Cray Compilation Environment
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DARPA	Defence Advanced Research Projects Agency
DDR	Double Data Rate
DMA	Direct Memory Access
DOD	Department of Defence
DOE	Department of Energy
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
EDA	Electronic Design Automation
ESL	Electric System Level
FLOP/s	Floating-Point Operations per Second
FPGA	Field Programmable Gate Array

(GP)GPU	(General Purpose) Graphics Processing Unit
HDD	Hard Disk Drive
HDL	Hardware Description Language
HMC	Hybrid Memory Cube
HMPP	Hybrid Multicore Parallel Programming
HPC	High-Performance Computing
IB	InfiniBand
IBM	International Business Machines Corporation
IEEE	Institute of Electrical Engineers
IFE	Inertial Fusion Energy
LANL	Los Alamos National Laboratory
LLC	Last Level Cache
LLNL	Lawrence Livermore National Laboratory
LLVM	Low Level Virtual Machine
LUT	Look Up Table
MPI	Message Passing Interface
MTTI	Mean Time to Interruption
MW	Megawatt
NIC	Network Interface Card
NIF	National Ignition Facility, USA
NOC	Network on a chip
NUMA	Non-Uniform Memory Access
ORNL	Oak Ridge National Laboratory
OS	Operating System
PCIe	Peripheral Component Interconnect Express
PGAS	Partitioned Global Address Space
PGI	Portland Group Incorporated
PRAM	Parallel Random Access Machine
PTX	Parallel Thread Execution
RAM	Random Access Memory

RDMA	Remote Direct Memory Access
RTL	Register Transfer Level
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multi-Processor
SNL	Sandia National Laboratories
SOC	System on a Chip
SPH	Smoothed Particle Hydrodynamics
SPMD	Single Program Multiple Data
SSE	Streaming SIMD Extensions
TDP	Total Power Draw
TLB	Translation Lookaside Buffer
UPC	Unified Parallel C
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Definitions

Collective

Refers to a communication event, within parallel application programming, in which more than two end-points are involved. Communication of this type can potentially involve multiple source or destination end-points (or both).

Computational Kernel

A collection of application program code, such as multiple loop-block structures, which has been logically co-located within the same program function or subroutine, and collectively performs a particular well-defined task or operation.

Compute Bound

Is a term used to refer to one or a series of operations whose overall runtime is dominated by the length of time required to process the particular instructions and associated data-values within the computational device.

Exascale

Is a term which refers to high performance computing systems which are capable of executing a thousand Petaflops or a quintillion (10^{18}) floating point operations per second.

Elemental Function

Denotes a function which operates on a scalar argument or single array element but can also be applied in parallel to a series of, potentially multi-dimensional, array elements.

Energy to Solution

Refers to the total energy (Joules) consumed by an application during the course of its execution on a particular processing architecture.

Global Address Space

In parallel programming this relates to the ability of any thread of execution to directly access any memory location, which as been designated as being globally

accessible, within the overall parallel application.

Halo Cells

The design of parallel applications often involves the decomposition of the overall problem domain across multiple processors, such that each process is responsible for a distinct subset of the domain. The operations performed by each processor, however, often require data-values from parts of the problem domain which are managed by other processes within the overall computation. This frequently occurs on the boundary between the contiguous domains managed by different processes. To minimise the accesses to remote memory locations on other processes, boundary data cells from logically adjacent processes are often replicated in a layer of cells around the domain managed by each process. This additional layer of cells is referred to as a *"halo"* region and can be of varying depths depending on the requirements of the algorithm currently being executed.

Kernel Driver

A program which is able to unit-test a particular *computational kernel* routine in terms of both its overall performance and correctness.

Memory Bound

Is a term used to refer to one or a series of operations whose overall runtime is dominated by the length of time required to load (or store) the particular instructions and associated data-values from the memory sub-system rather than to actually process them within the computational device.

Network On a Chip (NOC)

Is a term used to refer to the inclusion of a dedicated interconnection network between the processing components of a System On a Chip design i.e. within the same integrated circuit.

Non-Uniform Memory Access (NUMA)

Refers to a particular design of multi-processing system in which the time to access individual memory locations varies depending on the proximity of the particular memory locations to the accessing processor. A "NUMA" region is

used to refer to a collection of memory locations which all have the same access time relative to a particular processor.

Parallel Speedup

Is calculated by the time recorded for the execution of the application in serial divided by the execution time of the application when run in parallel.

$$S(parallel) = \frac{T(serial)}{T(parallel)} \tag{1}$$

Petascale

Is a term which refers to high performance computing systems which are capable of executing one quadrillion (10^{15}) floating point operations per second.

Point-to-Point

Refers to a communication event, within parallel application programming, between a distinct pair of end-points i.e. with a well-defined source and a destination.

Portable Performance

Refers to the goal of achieving optimal or acceptable levels of performance across multiple different types of system architectures from a single source code representative of an application, that is without including optimisations or modifications for specific architectures.

Remote Direct Memory Access (RDMA)

Is a form of communication in which the initiating CPU sends information regarding the message transfer (length, remote memory address etc) to its local NIC, which then manages the actual data transfer across the network [22]. Communication is one-sided and consequently the remote CPU is not involved in the data transmission, the network hardware at the destination handles all of the processing involved in the receipt of the data and committing it to memory.

Strong Scaling

Solving a fixed problem size by utilising an increasing amount of computational resources.

System On a Chip (SOC)

Is a term used to refer to an integrated circuit that incorporates all of the necessary components required for a computational device within a single chip substrate.

Wall-clock

A measure of application performance (the actual length of execution time) recorded by an observer external to the application. This is different to CPU or user time which relates to the total amount of time processor devices actually spend executing applications.

Weak Scaling

In these studies the overall simulated problem size is increased proportionally in line with the computational resources employed in the computation.

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

The use of scientific computing / HPC has grown significantly over the last decades and increasingly organisations and national governments are recognising that it is crucial to their competitiveness and future prosperity [153, 11]. The field promises to improve scientific insight and reduce product development cycles by enabling more experiments (higher throughput) to be conducted in significantly reduced time frames and overall operating budgets, whilst reducing the need for more expensive physical tests. Additionally it enables experiments to be conducted, potentially at higher fidelities and which couple multiple different physics packages, that were previously not possible due to their sheer size, complexity or cost [153, 206]. Increasingly HPC is also being utilised to simulate particular problems which are impossible or extremely impractical to test physically due to either the regulatory environment or safety concerns. This has led to simulation being widely recognised as the third pillar of scientific discovery alongside theory and experimentation [64, 167].

Several scientific "*grand challenge*" problems have been identified that will require systems capable of delivering exascale levels of computational performance in order to effectively simulate them and produce the required advances in science [128, 206]. These include the solution of vastly more accurate predictive models to improve scientific understanding within, for example, the fields of: climate/weather forecasting; efficient low-carbon transportation; nuclear and renewable energy; the certification of nuclear stockpiles; materials science; national security; and the advancement of certain biology/medical applications such as effectively simulating the human brain [153, 128, 206].

Historically these systems were exclusively the preserve of large multi-national organisations and government laboratories, primarily due to the costs associated with procuring and operating them. The increasing commoditisation of the technologies used to construct HPC systems has, however, facilitated significant reductions in their overall cost and enabled smaller commercial organisations and universities to gain access to them [11]. This has simultaneously enabled substantially larger, more computationally capable and power-consuming systems to be constructed for organisations at the forefront of the field.

Despite the growing requirements for the use of HPC / scientific computing technologies the field faces numerous significant challenges as organisations continue to push towards the construction of systems capable of delivering exascale levels of computational performance [11, 128, 206]. The improvements in pro-

cessor clock speeds, seen over the last decades, have proved to be unsustainable due to their power and cooling requirements [153, 11]. System designers have therefore been forced to significantly increase the amount of parallelism available at all system levels, in order to continue to improve computational performance capabilities. Overall system power consumption continues to become a major concern to large HPC sites as systems become larger [153, 11, 206]. Due to these increased scales, system MTTI (Mean Time to Interruption) is reducing to levels below the time required to perform a check-point and restart operation, resulting in overall system resiliency becoming increasingly problematic. Research into fault resilient programming models for applications is therefore becoming increasingly necessary [153, 11, 128]. At the processing chip/device level transistor feature sizes continue to decrease in order to reduce energy requirements and increase the computational capabilities of the associated devices. Similarly advanced architectures such as GPGPUs, which exhibit even larger degrees of parallelism, are increasingly being considered to further improve performance. As the floating-point computational capabilities of processing devices improve in terms of both execution time and power consumption, actually performing these operations is becoming relatively inexpensive, whilst the cost of moving data is becoming extremely expensive [153, 11, 128, 206, 115]. Consequently memory bandwidth/latency and inter-node communication speeds are increasingly limiting application performance and accounting for the most significant proportion of overall power consumption [11].

The rapid technological change, currently being experienced by supercomputer architectures, represents a significant challenge to HPC application code teams. Approaches based on the concept of "*co-design*" have been proposed to address these challenges [153, 11, 206]. The growth in on-chip parallelism is forcing algorithms/applications to move away from their existing coarse-grained BSP (Bulk Synchronous Parallel) based models of concurrency, towards a more fine-grained model of parallelism and to rely more on strong scaling [153, 11, 128]. Whilst weak-scaling simulation configurations will still be important on exascale systems, it is highly likely that in order to reduce simulation time-to-solution to currently required levels, the ability to effectively strong-scale applications across future multi-petascale or exascale platforms will be essential if these classes of machine are to be fully 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 [10, 79, 11, 128]. Scientific application code bases are also increasingly large and extremely complex; consequently porting them to advanced novel architectures, in a manner which delivers portable performance across different platforms, is becoming increasingly problematic [153,

11, 115]. Effectively utilising the increased concurrency available will also be vital if existing scientific applications are to harness the increased computational capabilities present within future supercomputer architectures. Additionally, simply maintaining them productively given current limited financial and development resources also presents challenges and requires significant investments.

Given these trends and the pressing need to improve the performance of key scientific codes on existing and future system architectures this thesis focuses on evaluating the utility of particular newly proposed technologies for the advancement of explicit hydrodynamics applications. In particular it strives to evaluate both software and hardware technologies and techniques that will enable this class of applications to achieve greater overall performance and scalability. Achieving these aims will facilitate improvements in the science which it is possible to accomplish by improving overall scientific throughput (time-to-solution) as well as current simulation resolutions.

1.1 Motivations and Problem Statement

The scientific need to develop more advanced, potentially exascale-class, computational facilities is well documented, see Section 1 for more details. Actually achieving the successful construction of future multi-petascale or exascale capable supercomputer systems and developing scientific simulation and modelling applications which are able to effectively take advantage of their capabilities, however, currently presents a number of significant challenges [11].

These include but are not limited to, addressing the overall power efficiency of existing supercomputers to enable future larger and more computationally powerful systems to be constructed [11]. Employing today's technology to construct a system capable of delivering an exaflop of computation would require more than 1GW of power [153]. The DOE (Department of Energy), in the USA, has set the HPC industry the challenge of delivering an exascale capable solution within an overall power budget of 20MW, necessitating an improvement of $>150\times$ in power efficiency over current technology [153]. At the same time some observers do not believe that the 20MW target is achievable [114]. A practical limit of approximately 100MW exists, however, as the largest data-centres currently in existence only have access to this amount of power [48]. Regardless of the exact power budget figure, achieving a solution which lies within this range will still require a huge improvement in computational power efficiency over current technological solutions [153, 79, 10].

Actually developing and maintaining scientific applications and their underlying software components, to enable them to effectively utilise future supercomputing architectures will also become increasingly challenging. The creation of

new programming paradigms designed to support more fine-grained parallelism and deeper memory hierarchies may, therefore potentially be required [153, 11]. Additionally, it is recognised that achieving the necessary computational power efficiencies will require future systems to use significantly different processor architectures to current generations of systems [153, 11]. Supercomputer architectures are thus at present experiencing a transitional period. Potential future candidate technologies include the use of accelerator devices such as GPGPUs, many-core CPU devices with lower clock frequencies such as the Intel Xeon Phi or the use of lower-power technologies from the mobile and embedded computing sectors, such as ARM processors or FPGAs (Field Programmable Gate Arrays) [183, 142]. Regardless of which approaches prevail achieving optimal performance for existing applications and software stacks on these advanced architectures will be extremely problematic [153, 11]. Additionally, enabling applications to deliver portable performance across a range of future architectures, which is a requirement of large HPC sites to avoid vendor "*lock in*", also presents significant challenges [11].

Furthermore scaling applications and systems to the levels of concurrency which will be required to achieve exascale-levels of computational performance also represents a significant challenge [153, 79, 10, 11]. It has been argued that existing software approaches, mainly based on the MPI-only model of computation, are already starting to reach the limits of their scalability, due to the number of MPI ranks competing for shared interconnect and memory resources, necessitating additional research into alternative programming models and techniques [18, 11]. Additionally, on machines incorporating accelerator technologies, MPI-only is not a viable solution and precludes their use [11]. Hybrid programming models, which are able to make use of accelerators and the shared memory capabilities available within nodes, represent a promising area of research for improving performance by reducing the overall number of MPI ranks involved in the computation. They may also enable applications to be better adapted to future system architectures which are likely to exhibit significant reductions in the memory capacity, memory bandwidth and network bandwidth resources available per processing core [11].

It has also been recognised that if certain classes of application were able to increase the levels of asynchronicity inherent within them, by fully exploiting their potential to overlap communication and computation, then it would be possible to utilise significantly lower performance interconnects for these applications, without negatively impacting performance [175, 11]. Additionally the increased complexity of modern interconnects is forcing us to examine topology-aware communication mechanisms and the placement of application processes within the network in order to achieve optimal performance [4, 35, 11].

Unlike the MPI model which utilises a two-sided model of communication, PGAS (Partitioned Global Address Space) based approaches such as CAF (Co-array Fortran) or OpenSHMEM rely on a lightweight one-sided communication model and a global memory address space [40, 148]. This model represents another promising area of research for improving the performance and scalability of applications as well as programmer productivity [11]. It may also potentially deliver further performance advantages by facilitating a reduction in the overall memory footprint of applications through, for example, the elimination of communication buffers. 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 [200]. Although the body of work which examines PGAS-based applications on these technologies is still relatively small, substantially less research exists which examines their performance on systems constructed from commodity-based technologies such as Infiniband. It is likely that this analysis will 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 SOC (System On a Chip) designs [96, 95]. 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 [11].

The task of developing, porting and optimising applications for future generations of HPC systems is becoming increasingly complicated as architectures evolve [153, 11]. Developing and maintaining MPI-only applications is also becoming increasingly problematic due to their complexity and the analysis of legacy applications in order to convert them to hybrid models is non-trivial [11]. Even with an in-depth knowledge of the algorithm and target hardware, extracting the maximum concurrency is a difficult, time-consuming task. Improving the tool-suite available to developers which assists with this task will be essential if optimal performance is to be achieved productively [153, 11].

1.2 Domain

This thesis is exclusively concerned with improving the performance of hydrodynamics applications and the identification of the most appropriate processing solutions to facilitate their execution on future supercomputer system architectures. The research undertaken is therefore focused on the fields of scientific and high performance computing and is concerned with the performance, in terms of overall time-to-solution, of a suite of applications of interest to the sponsor of this work. Additionally it also focuses on the computational resources

(e.g. memory capacity/bandwidth and power/energy consumption) consumed by these applications whilst executing on particular architectures of interest.

Many of the research topics which are examined in this thesis have significantly wider applicability to other application domains within the scientific computing field. The applications utilised within these domains exhibit similar performance characteristics to the hydrodynamics applications examined within this work, and researchers are also pursuing similar directions for improving the current state-of-the-art, e.g. utilising PGAS and hybrid programming models. Additionally other communities, such as the mobile and embedded computing sectors may also potentially benefit from this research, as these fields already extensively utilise several of the technologies examined in this research, e.g. FPGAs. Similarly the research methodology employed in this work has much broader applicability than to just scientific computing applications and technologies. This thesis is, however, deliberately constrained to the advancement of explicit hydrodynamics applications within the scientific computing field in order to adequately explore the applicability of the examined techniques, optimisations and technologies to this domain of interest.

1.3 Research Questions and Hypothesis

The trends and challenges, outlined in Section 1.1 motivate the author's research and the work presented in this thesis specifically examines the following research questions within the domain documented in Section 1.2:

1. Is it possible to improve the scalability of hydrodynamics applications, and thereby their performance, by enabling these applications to execute more efficiently on larger scale supercomputer resources, through the utilisation of alternative design and implementation approaches. These include utilising optimisation techniques such as overlapping the execution of communication and computation constructs; evaluating alternative communication strategies which are not based on the BSP-model; improving the mapping between application processes and the underlying machine interconnect topology; and employing a distributed approach for the management of computational mesh meta-data.

2. Does the use of a hybrid programming model, based on either OpenMP and OpenCL, enable the performance and scalability of this class of scientific applications to be significantly improved, and if so to determine how these models should be combined with existing approaches to achieve optimal performance.

3. Can the utilisation of PGAS-based programming models deliver any performance and programmer productivity benefits for these hydrodynamics applications, and if so to establish how this class of scientific applications should be developed in order to maximise any potential benefits from the use of this technology.

4. Determine which prospective supercomputer architectures currently represent the most performant and also energy efficient processing solution for the execution of hydrodynamics applications. In particularly whether x86 CPUs, IBM BG/Q CPUs, AMD APUs (Accelerated Processing Unit), GPU-based accelerators, or the Intel Xeon Phi many-core accelerator, are currently the most optimal choice for these applications.

5. Is it possible to utilise the OpenCL programming model to improve the performance portability of hydrodynamics applications across a range of prospective supercomputer architectures, including platforms based on CPU, GPU, APU or many-core accelerator technologies.

6. Finally, to determine whether FPGAs currently represent a viable processing technology which could be utilised within future supercomputer systems in order to improve the overall energy consumption of these applications, thus potentially enabling the construction of larger, more computationally capable systems within a fixed power budget.

The primary research hypothesis of this work is that:
The performance of computational hydrodynamics simulations can be improved through the use and implementation of the aforementioned technologies and optimisation techniques on current generations of supercomputer platforms.

The overall objective of this research is therefore to improve the performance of key hydrodynamics simulation applications through the examination of these research questions and the testing of this hypothesis. Thereby potentially facilitating advances in the scientific knowledge which it is currently possible to generate through their use, either by delivering improvements in overall scientific throughput by reducing the time-to-solution of existing simulations, or by enabling larger more sophisticated simulations to be conducted which are not currently feasible.

1.4 Research Methodology

The research documented in this thesis was undertaken using the following research methodology to address the problems and challenges listed in Section 1.1

within the domain outlined in Section 1.2.

To enable the research objective of this thesis to be completed in a reasonable time efficient manner an approach based on the use of a mini-application (or mini-app) was employed. Mini-apps are small, self contained programs that embody essential performance characteristics of larger applications, and thus provide a viable way to conduct more rapid experimentation [84]. This work utilises and further develops a simplified but still representative structured, explicit hydrodynamic mini-app known as CloverLeaf (Section 1.6) [132]. Attempting this work using fully functional legacy production codes has in the past been found to be time consuming and impractical, due to the number of potential solutions available and the time required to port the codebases to the new technologies [84, 11]. A more rapid, lower risk approach for investigating the solution space is therefore extremely desirable. The use of a mini-app enables this rapid development and exploration of new technologies, architectures and techniques, in a manner which is still representative of the main production codebases which CloverLeaf represents.

Evaluating the utility of each of the different programming models and techniques involved in this research required the development of numerous new versions of CloverLeaf. Each new version examined one particular technique or programming model enhancement, which ensured that changes in results can be accurately attributed to particular modifications within the codebase. During development the functionality and correctness of these additional versions was regularly and frequently validated against the original version of the codebase to ensure that bit-wise identical results, or results to within an acceptable error tolerance, were produced at each stage. These validation tests were frequently executed at small experimental scales (e.g. <64 nodes), however, during each subsequent large-scale experiment the original CloverLeaf codebase was also executed alongside the modified versions, enabling the results produced by all additional versions to be validated at each stage of this work.

To examine the success of each candidate code optimisation technique, programming model or technology, quantitative assessment methods using results obtained from experiments on actual existing supercomputer hardware systems were employed at each stage, rather than relying on the use of simulation environments. Due to the scales of some of the experiments involved in this research, system noise, caused by OS (Operating System) jitter and other concurrently executing jobs, contending for globally shared system resources on several of the key architectures under consideration, became a factor in the analysis of the obtained experimental results. Specifically, it was therefore possible for the jobs of other users, which were simultaneously executing on the experimental platforms, to perturb these experimental results. To mitigate the effects of

this system noise these experiments utilised, whenever possible, experimental platforms in a fully dedicated mode. This ensured that only experiments related to this research had access to the globally shared resources within a particular supercomputing system, thus minimising any system noise caused by other simultaneously executing applications. Additionally each experiment was also repeated several (typically three) times and the results averaged to produce a final value, before any analysis was conducted, thus further limiting the effects of any system noise on the obtained results and conclusions. To mitigate the influence of different network topologies and node allocations from the batch schedulers managing the various supercomputer platforms examined in this work, experiments at a particular scale were aggregated and executed within the same allocations.

The range of experimental architectures and platforms involved in this research were also selected to provide an extensive range of candidate technologies, at both the node and system levels, which could potentially be utilised to construct future generations of systems. Similarly experiments were selected to enable conclusions to be drawn regarding the performance of a particular technology or technique at a range of experimental scales. This included experiments which examined performance on 1 node through to the largest job sizes which it was practical to obtain on a particular platform, up to 8,192 nodes (131,072 cores) in certain cases. These large scale experiments were essential in enabling the utility of particular approaches to be accurately assessed as potential candidates for enabling future applications to achieve exascale-levels of computational performance.

The PowerInsight [119] technology was selected in order to conduct experiments to accurately assess the power consumption/efficiency of the individual technology components involved in this research. This has been developed and appropriately validated to accurately monitor, at a sufficiently high sampling frequency (maximum of 1,000MHz), the power drawn by all of the power rails supplying each particular component. This includes the power drawn over the PCIe bus connections which particular component cards use to interface with the main circuit ("mother") board on the nodes of supercomputer systems. It is possible for components to draw up to 75W over these PCIe connections, which is potentially a significant proportion of their overall power consumption [121]. Additionally, PowerInsight also enables the actual power supply lines into the other node components to be accurately monitored, including CPU and memory devices, HDD devices and the PCIe cards. Use of this technology enables out-of-band power consumption traces to be generated for applications executing on a particular technology without perturbing their actual execution, which would potentially further effect overall power consumption. It also enables power mon-

itoring research to be conducted without relying on the power/energy consumption counters available within some processing devices, which are potentially inaccurate. Additionally, devices which do not contain these built-in monitoring subsystems can also be measured consistently. Whilst this technology is able to accurately measure the power consumption of individual components at the node level it is not able to produce accurate power consumption measurements for large-scale experiments on actual supercomputer platforms. The power monitoring capabilities available natively on the IBM BG/Q [201] and the Cray XC30 [135] platforms were therefore employed in order to conduct this aspect of this research.

1.5 Thesis Contributions

Specifically, to address the challenges and motivations discussed in Section 1.1 and answer the research questions documented in Section 1.3, this thesis makes the following key contributions:

Mini-app Development and Utilisation
It reports on how the CloverLeaf mini-app, which is documented in detail in Section 1.6, was further developed and utilised as part of this work in order to conduct the necessary research into potential application optimisations, candidate programming models and prospective supercomputer architecture choices. Additionally, it also documents how the general planning and decision making relating to the future development of scientific applications can be improved through the use of mini-apps. This research contributed significantly to CloverLeaf being accepted as part of the Mantevo mini-applications suite from Sandia National Labs [84], which was recognised as one of the top 100 most technologically significant innovations in 2013 by R&D Magazine [171, 184]. It was also the UK's only contribution to the initiative and is currently being actively utilised by a large number of HPC centres, vendors and researchers across the world.

Evaluation of PGAS Programming Models
Utilising PGAS-based programming models is recognised as a potential approach for improving the performance and scalability of applications and enabling them to achieve exascale-levels of computational performance. A further contribution of this thesis is to examine whether two such PGAS programming models (OpenSHMEM and CAF) can deliver any performance or scalability improvements for this class of application. The implementation of CloverLeaf in both PGAS programming models is documented together with experiences

gained during the conversion from the original MPI-based implementation to these models. This included the development of 10 distinct OpenSHMEM- and 8 distinct CAF-based versions, each of which examine alternative implementation approaches.

A performance analysis is presented to provide both a comparison of each programming model and to assess how the communication constructs within each can best be incorporated into existing parallel applications. This examines the performance of these versions, at considerable scale (up to 49,152 cores) under a strong-scaling experimental scenario, on two state-of-the-art system architectures and vendor implementations (SGI and Cray). To assess the utility of these PGAS implementations against the dominant programming paradigm used in existing parallel scientific applications a performance comparison against an equivalent MPI-based implementation of CloverLeaf is presented. This information will be useful to developers of future OpenSHMEM and CAF applications. Similarly, based on these results, recommendations to improve both the OpenSHMEM specification and potentially future CAF compiler and runtime systems are also identified.

Examination of Hybrid Programming Models

The incorporation of hybrid programming model constructs, based on both OpenMP and OpenCL, into this class of application is examined together with a quantitative assessment of whether these models can deliver benefits in terms of improved application performance or scalability. A detailed description of CloverLeaf's hydrodynamics algorithm, and its implementation in both OpenMP and OpenCL is presented, together with a description of how both models integrate with the existing MPI-based Fortran code. Comparisons of the performance of the MPI+OpenMP and MPI+OpenCL versions of CloverLeaf are presented, relative to the original MPI-only version, at considerable scale on a number of system architectures including, two alternative Cray system architectures, an SGI ICE-X platform and an IBM BG/Q. A smaller-scale (1 node) analysis is also conducted across a broader range of potential candidate HPC architectures. For both programming models a number of optimisations to improve performance and portability are documented and their effects analysed.

The ability of the OpenCL programming model to deliver portable application performance from a single code base across a broad range of future candidate supercomputer architectures is assessed. Additionally the viability of both approaches for expressing large scientific codebases and achieving acceptable levels of programmer productivity is also analysed.

To potentially improve programmer productivity tools to automatically hybridise MPI-only codebases using OpenMP constructs are being developed.

The utility of the Reveal software tool from Cray is therefore also evaluated as a technology for achieving this, by automatically hybridising the MPI-only version of CloverLeaf and comparing its performance to that of a hand-optimised MPI+OpenMP implementation.

Development of Application Optimisations

The effect of several candidate optimisation techniques on the performance and scalability of the MPI-only versions of this class of scientific application, are also examined and quantitatively assessed at considerable scale on three candidate system architectures: IBM BG/Q, Cray XC30 and SGI ICE-X. These optimisations include the examination of the effect of: utilising an implementation based on the use of distributed mesh meta-data information; overlapping communications and computational operations; several recently standardised MPI v3.0 constructs; as well as several message aggregation and early data transmission communication strategies. Additionally, the effect of optimising the placement of MPI ranks within the supercomputer interconnect fabric is explored together with the effectiveness of employing software tools from Cray in achieving this rank remapping.

Supercomputer Architecture Analysis

This thesis examines a range of technologies which are currently available for the construction of supercomputer platforms and provides an evaluation of the suitability of several intra- and inter-node processing architectures for the execution of explicit hydrodynamics applications. This enables the solution space of candidate technologies, which will likely be available for the construction of future exascale capable supercomputer systems, to be explored in order to assess their potential utility for delivering the performance improvements required for the scientific applications which are the focus of this research. Performance results from the execution of CloverLeaf are presented and analysed under a range of programming models on discrete GPGPU solutions from Nvidia and AMD, Intel Xeon Phi coprocessors, AMD APU based systems as well as CPU-based solutions from Intel and AMD. A performance comparison of the OpenCL version of CloverLeaf, against optimised native versions (OpenMP and CUDA), is also included as well as the effect of various optimisation techniques.

Additionally, the performance and behaviour of numerous versions of the application (MPI-only, MPI+OpenMP, MPI+CUDA, MPI+OpenCL, PGAS-based) are also assessed at scale on several existing large-scale system architectures incorporating different interconnect topologies and technologies. These include a Cray XC30 (Aries Dragonfly), a Cray XK7 (Gemini 3D-torus), an IBM BG/Q (5D-torus) and an SGI ICE-X (IB 7D-hypercube) platform.

As well as assessing candidate technologies in terms of overall performance (time-to-solution), this thesis also examines the power consumption of several of these technologies and presents an analysis of the energy consumed in achieving a solution on a range of different technologies. This analysis is conducted at both small- (1 node) and large-scale (>2,048 nodes) using a variety of power-measurement solutions.

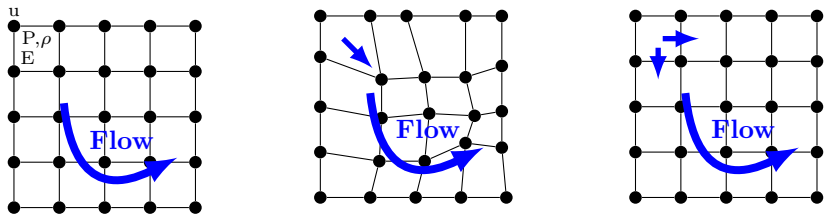
Furthermore the viability of FPGAs devices from Altera, as candidate technologies to employ in future system architectures, is also examined. This includes an examination of how to optimally express particular explicit hydrodynamics computational kernels in order to maximise performance on these FPGA devices using the OpenCL compiler and runtime systems developed by Altera. A quantitative assessment is also conducted of whether this technology is able to deliver significant reductions in the energy required to achieve a solution, whilst delivering acceptable levels of performance, relative to existing state-of-the-art processing solutions, which are currently commonly utilised for this class of application.

1.6 CloverLeaf

Mini-apps are small, self-contained codes, which emulate key algorithmic components of much larger and more complex production codes. One of the main contributions of this research was the significant enhancements made to the development of the CloverLeaf mini-application, which was extensively used as a research tool through this work. CloverLeaf was originally developed with the explicit purpose of assessing new technologies and programming models both at the inter- and intra-node system levels. This section provides details of the implementation of the mini-app. Further information on the specific hydrodynamics scheme simulated within the application can be found in Section 2.3.

1.6.1 Implementation

CloverLeaf employs a Lagrangian-Eulerian scheme to solve Euler's equations of compressible fluid dynamics [87, 42], using the ideal-gas equation of state, in two spatial dimensions. The equations are solved on a *staggered grid* (see Figure 1.1a) 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 discretise the Euler equations and facilitate their solution 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 within CloverLeaf.



(a) The staggered-grid with material flow

(b) Vertices move during the Lagrangian step

(c) Material motion calculated in the advective remap

Figure 1.1: Staggered grid employed in 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):

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 then 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 [199]. 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. The implementation also simultaneously attempts to assign a similar number of cells to each process in order to balance computational load. As with the majority of block-structured, distributed, scientific applications which solve systems of partial differential equations, data that is required for the various computational steps that is non-local to a particular process is stored in outer layers of *halo* cells within each mesh chunk. To keep these *halo* cells updated data exchanges, between logically neighbouring processes within the decomposition, occur multiple times during each timestep with varying depths. 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. A global reduction operation is required by the algorithm during the calculation of the minimum stable timestep, which is calculated once per iteration.

The codebase of CloverLeaf is predominantly Fortran based and its computational intensive sections are implemented via fourteen individual *kernels*. In this instance, *kernel* refers to a self-contained function which carries out one specific step 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 the use of complex features such as Fortran derived types. Twelve of CloverLeaf's kernels only perform computational operations, with communication operations residing within the overall control code and two other 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. A further reduction is carried out to report intermediate results, but this is not essential to the numerical algorithm.

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 by refactoring large loops into smaller parts, adding extra temporary storage where necessary; replacing branches inside loops where possible; replacing atomic operations and critical sections with reduction operations; memory accesses were also optimised to remove all scatter operations and minimise memory stride for gather operations. The computational intensity per memory access in CloverLeaf is low which typically makes the code limited by memory bandwidth and latency speeds.

In the experiments documented in this thesis (Chapters 3 to 7) 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, which causes a shock-front to form. The configuration can be altered by varying the number of cells employed in the computational mesh; increasing mesh resolution generally increases both the runtime and memory usage of the simulation.

1.7 Thesis Structure

The remainder of this thesis is organised as follows:

Chapter 2 provides background information and a future trends analysis relating to several of the research areas examined in this thesis, including the hardware platforms utilised, the software technologies examined and the explicit hydrodynamics applications on which this work has focused.

The research work which examined optimisations to improve the perfor-

mance of the OpenMP-based versions of the CloverLeaf codebase is presented in Chapter 3, and includes an analysis of their performance at high thread counts on the Intel Xeon Phi architecture. This contributed towards answering research question 2. The work presented in this chapter also extends research documented in publications 1 and 3, as listed in the Declarations section of this thesis.

The research conducted to improve the performance of the CloverLeaf mini-app at extreme scale and thus answer research question 1.7 is documented in Chapter 4. The work presented in this chapter is based on research previously published in papers 1 and 3.

The implementation of the PGAS programming model based versions of CloverLeaf are documented in Chapter 5 together with an analysis of their performance against equivalent MPI-based versions, at significant scale on two candidate system architectures (Section 5.4). This chapter examines research question 3 and the work extends the research previously published in paper 4.

Chapter 6 examines the use of the OpenCL programming model and assesses its utility for delivering portable application performance for explicit hydrodynamics applications on a range of current processing architectures. It addresses research question 5 and extends the work previously documented in publications 2 and 5.

The suitability of utilising FPGAs as candidate processing solutions for explicit hydrodynamics applications within future architectures (research question 6) is examined in Chapter 7.

Chapter 8 presents the conclusions which this thesis has facilitated, together with its key contributions (Section 8.1) and limitations (Section 8.3). It also outlines some potential directions for future research (Section 8.4). Research question 2 is examined by Chapters 3 to 7 but the overall conclusion derived through this work is documented in this chapter.

Finally, Appendix A documents in detail the experimental architectures and platforms utilised throughout this research.

1.8 Project Availability

This research work was conducted as part of the overall CloverLeaf mini-app development project. In keeping with the ethos of the project all of the codebases developed as part of this specific research can be found within the main CloverLeaf Github development repository at `https://github.com/Warwick-PCAV/CloverLeaf`. Each major version of the codebase which was developed as part of this work, e.g. all of the CAF-based versions, are made available within separate sub-repositories. Minor versions which e.g. examine a specific optimisation or

technique within these broader categories, are then generally made available as separate branches within these sub-repositories. It is hoped that making this work as open and accessible as possible will foster greater collaborations within the scientific research community, enable others to learn and benefit from the derived conclusions and general approach, as well as to also ultimately improve upon it.

CHAPTER 2
Background Information

This chapter presents background information on the hardware (Section 2.1) and software (Section 2.2) technologies employed and examined in this research, as well as information on the hydrodynamics applications and algorithms studied (Section 2.3). Historical information is provided together with existing issues and current, as well as likely future, development trends.

2.1 Hardware Background and Trends

This section provides background information on the three major hardware subsystems, within a HPC platform, which the research documented in this thesis interacts most closely with. These include interconnect technologies (Section 2.1.2) as well as the processor (Section 2.1.3) and memory (Section 2.1.4) subsystems. It also discusses what many consider to be the single most significant challenge currently facing the construction of exascale systems, their power consumption (Section 2.1.1).

2.1.1 Power Consumption

It has been widely recognised that power consumption will be the primary constraint governing the design of HPC systems in the future [11, 206, 128]. Several existing large-scale systems are currently consuming of the order of 10MW of power [114], with ORNL's Titan and Riken's K computer consuming ∼8MW and ∼12.6MW respectively, whilst Tianhe-2 in China consumes ∼17.8MW [194]. Employing today's technology to construct a system capable of delivering an exaflop of computation per second would require more than 1GW of power [153]. To address this issue the DOE in the USA has set the HPC industry the challenge of delivering an exascale capable solution within an overall power budget of 20MW, necessitating an improvement of >150× in power efficiency over current technologies and equating to approximately $20 million in electricity costs annually [153, 129, 206]. Whilst some observers do not believe that the 20MW target is achievable [114], a practical limit of approximately 100MW would appear to exist, as the largest data-centres currently in existence only have access to this amount of power [48]. Regardless of the exact power budget figure, achieving a solution which lies within this range will still require a huge improvement in computational power efficiency and require considerable

research and development, but would potentially deliver considerable financial savings if achieved [153, 79, 10].

2.1.2 HPC Interconnect Technology

The interconnect technology has always been a key component of HPC systems and this trend will only continue as the cost of communication (moving data) starts to dominate performance in future system architectures [11, 196]. Historically HPC systems employed proprietary interconnect technologies from vendors such as Quadrics [161], Cray [9, 31, 65], IBM[41], Fujitsu [3] and Myricom [70]. These technologies generally incorporated proprietary ASICs (Application Specific Integrated Circuits) on dedicated NICs (Network Interface Cards) and delivered improved performance in terms of reduced latencies and higher bandwidth over commodity solutions by offloading some of the communication processing to the NICs. They often provided support in hardware for operations commonly required by scientific applications, such as collective, atomic and one-sided Remote Direct Memory Access (RDMA) communication operations, which were not generally available in alternative commodity solutions. Additionally, they also supported topologies which closely mirrored the communication patterns of scientific applications, or enabled systems to be scaled to larger processor counts, such as 3 [9, 31], 5 [41] and 6 [3] dimensional tori; "fat" trees [161, 70]; and dragonflies [65]. The Quadrics network, for example, offloaded the processing of MPI (Message Passing Interface) communications onto the NIC processor via the Elan Tports interface enabling the host processor to undertake additional tasks during communication operations. The QsNetII solution was capable of autonomously completing MPI message matching operations, although the performance of the raw RDMA Elan interfaces was shown to be faster [23].

Driven primarily by reductions in costs from higher volumes, the HPC industry has more recently been moving away from proprietary interconnect technologies and towards more open standards-based, commodity technologies primarily based on Infiniband [94]. The use of Infiniband in systems ranked in the Top500 has risen from <1% in 2004 to >40% at present [194]. Although originally a storage interconnect design targeted at data-centre solutions it now incorporates many of the hardware facilities required by scientific applications, such as native support for RDMA operations and the offloading of communication operations to dedicated NICs. Infiniband has also been shown to enable some of the overheads of the two-sided communication model to be avoided [105, 176]. These include the requirement for the remote processes to be involved in the communications, handshake synchronisations, queue maintenance, message

tag matching and flow control.

Cray and IBM have been able to sustain their interconnect product lines, although IBM recently announced that it would be discontinuing production of its Blue Gene series machines [183] and Cray recently sold its interconnect business to Intel [96]. By contrast, Fujitsu recently announced that it would be continuing development of its Tofu interconnect [67].

To reduce power consumption and improve performance there has been a growing trend (although no products have thus far reached general availability) for chip manufacturers to develop SOC designs which incorporate the network interface logic previously located on the dedicated NICs. Intel recently purchased the Cray and Qlogic interconnect technologies [96, 95], whilst IBM announced plans through the OpenPOWER initiative [89] to incorporate Mellanox Infiniband technologies onto its Power processor architecture and Fujitsu outlined plans to incorporate its Tofu2 interconnect into its next generation of processors [67].

Future Trends

CPU processing capabilities, memory access latencies and hard disk seek times have gradually improved over time, however, inter-node message latencies across communications interconnects and their associated software overheads have not [22]. Additionally, in future systems the relative cost of data movement will be considerably higher than for floating point operations, as the energy required for the former is not improving at the same rate as for the latter, necessitating the creation of more power efficient interconnect designs [11, 206, 196]. The incorporation of interconnect technologies within future SOC designs should facilitate improvements, although additional technologies such as silicon photonics may still be required. Nevertheless within future systems the interconnect fabric is likely to be increasingly viewed as yet another level of the overall memory hierarchy.

It has been recognised that in order to achieve the required levels of application performance the levels of support for asynchronous data transmission and the movement of non-contiguous data will need to be improved within future interconnect designs [11, 128]. Additionally, improving support for the transfer of small data packets will also be increasingly important as the levels of overall parallelism increase and strong-scaling simulations become more prevalent [11]. There is also a trend towards more constrained, scalable topologies, such as multi-dimensional tori or dragonflies, to enable the construction of larger, more parallel systems [11]. The larger numbers of processing elements being incorporated within future SOC designs will necessitate the inclusion of interconnects

within these chips (networks-on-a-chip). These systems will also be increasingly limited by the communication infrastructure both within and between these nodes, and it is therefore likely that communication will be the main performance bottleneck at exascale levels of computational performance [11, 128].

2.1.3 Processor Subsystem Technology

After it was originally proposed in 1974 Dennard scaling held for over 30 years and started to breakdown in approximately 2005 [52]. It states that the power density of transistors remains constant as their size is scaled down and therefore the total chip power consumed per unit area remained the same from one manufacturing process generation to the next [134]. Post 2005, energy constraints in particular the significant increase in leakage current caused by the reductions in transistor feature sizes, have brought an end to Dennard scaling [11, 196]. Consequently, as the dynamic power consumption of a processor is proportional to its operating frequency, it is no longer possible to realise significant increases in overall CPU clock speeds.

With Moore's Law continuing to hold and the number of transistors per unit area doubling approximately every 18 months, manufacturers are increasingly being forced to incorporate more parallelism into their chip designs. Over recent years this has manifested itself most noticeably as increases in the number of cores (or explicit parallel processing elements) on a chip, which have been doubling approximately every 18-24 months [11, 206]. Most manufacturers now only offer multi-core designs for their processor offerings, such as the offerings from IBM [74] and Fujitsu [136] for their HPC platforms. Additionally there is a continuing trend to incorporate wider vector processing elements into CPU designs, necessitating applications to be able to use SIMD operations in order to achieve optimal performance [100].

Due to their greater power efficiency accelerator and co-processor solutions, such as GPGPUs from Nvidia [152] and AMD [13] as well as the Intel Xeon Phi [101], are also becoming increasingly utilised within the HPC community. It has also been argued that their use will be crucial in order for exascale systems to be realised [11, 128]. This will also require significant increases in fine-grained parallelism and the use of lightweight threading or task models [11, 206, 128]. GPUs support this style of parallelism particularly well, although they have been recognised as being considerably harder to program than alternative approaches such as the Xeon Phi [50].

At present, these devices are generally employed as separate discrete processing elements alongside traditional CPU devices, usually connected to the main system board over a PCIe link. As part of their Fusion APU processor

line, however, AMD have combined a CPU and GPU onto the same silicon chip [14]. Other manufacturers have also announced plans to produce similar hybrid devices, including project Denver [72] from Nvidia and IBM who are planning to incorporate Nvidia GPU devices onto Power-based processors as part of the OpenPOWER initiative [89]. Intel has also announced that future versions of its Xeon Phi processor will be "self-hosting" and will therefore not require a traditional CPU alongside them [21].

The power consumption of existing processors is also forcing manufacturers and researchers to consider low-power technologies from the embedded and mobile computing sectors, which have evolved to be more power efficient due to the additional power constraints within these environments [195]. These technologies, such as processors based on technologies from ARM [15], are starting to be considered for HPC systems and generally have higher sales volumes enabling their costs to be kept low [142]. ARM are also developing their designs to incorporate 64-bit processors in order to potentially gain additional business from new sectors such as HPC [195].

FPGAs incorporate large collections of generic logic and memory blocks connected via a reconfigurable interconnect fabric. By changing the routing configurations of this interconnect they enable customised processor designs to be created which are specifically tailored to implement applications using dedicated logic. This approach potentially delivers significant performance advantages whilst consuming substantially less power. The technology is found throughout the embedded computing sector meaning that the chips are produced in high volumes, which significantly lowers their overall costs.

Historically it has only been possible to "program" FPGAs via low-level approaches such as VHDL [91] and Verilog [90], which require extremely specialist knowledge and takes considerable development resources. More recently compilers have been developed to translate high-level languages, such as OpenCL, to these low-level languages, which potentially enables a broader range of scientific applications to be targeted at these devices.

FPGA manufacturers have also seen the potential to grow their business into new sectors such as HPC. Altera has recently announced that their latest Generation 10 products can now natively support IEEE 754 compliant single-precision floating point arithmetic, using dedicated hardware circuitry in each DSP (Digital Signal Processing) block within the FPGA fabric [8]. Similarly existing processor manufacturers have also realised the potential of FPGA based solutions with Intel recently announcing that it plans to incorporate an FPGA into future versions of its Xeon products [99] and IBM partnering with Altera through the OpenPOWER initiative [89].

Future Trends

Future processor designs are expected to continue the trend of increasing the number of processing elements which they contain as well as incorporating wider vector units [11, 206, 196]. This will necessitate the exposure of even greater levels of parallelism within applications in order to achieve optimal performance on future system architectures. It has also been recognised that the intra-node parallelism, delivered by the processor and memory sub-systems, will need to increase by 3 orders of magnitude if exascale systems are to be successfully realised, compared to only 1 order of magnitude for inter-node parallelism [206, 11, 196, 115, 128]. Consequently the execution of over 1 billion simultaneous instruction streams will likely be required within future systems in order to achieve exascale levels of computational performance [11, 206, 128].

Additionally, processor chip designs are likely to become increasingly heterogeneous, potentially incorporating sophisticated interconnects between the processing elements as well as the functionality historically performed by dedicated NICs. Due to energy constraints it is also becoming increasingly impractical for chip designs to provide uniform memory access bandwidth and latencies between processor elements, necessitating architectures to increase the number of NUMA domains and become more non-uniform [11, 196]. The use of so-called *"dark silicon"*, in which specialised components are incorporated into processor designs and only powered-up when required to save energy, is also a potential possibility [191].

Employing a design methodology based on the principles of "co-design" to improve the integration between all of the various hardware and software elements is also likely to be crucial in realising effective exascale systems [206, 11, 115, 128].

2.1.4 Memory Subsystem Technology

The density of DRAM and processor off-chip bandwidth are not currently increasing at the same rate as processor logic densities and this imbalance between computation and memory access speeds is forecast to continue to grow [11, 206, 128]. Consequently it is increasingly likely that future systems will incorporate significantly reduced memory capacities as well as access bandwidth and latencies per processor element. The performance of the memory sub-system is therefore likely to increasingly limit the performance of scientific applications on future platforms. These trends will necessitate the development of deeper memory hierarchies, which may potentially require the use of explicit memory space management constructs within applications, such as software managed caches [11, 206, 196]. The inclusion of transactional memory mecha-

nisms and additional atomic memory operations are also likely to be required in future systems [11].

Additionally it is likely that utilising technologies such as the Hybrid Memory Cube [141] from Micron will be required in order to improve memory system capacity and performance. This technology also offers the potential to conduct processing closer to the memory subsystem to further improve performance through reductions in data-motion. Fujitsu recently announced plans to support the technology in their forthcoming processor designs [67] and existing implementations which utilise FPGAs already exist [5]. The incorporation of faster and larger memories onto the actual processor die, through the potential utilisation of 3-dimensional stacking technologies, is another direction of potential development which should further reduce data access speeds and the energy consumed by moving data [58].

Again the use of a "co-design" methodology to holistically design the software, processor and memory sub-systems is likely to be crucial if these technologies are to be utilised optimally within future exascale system architectures [11, 206, 115, 128].

2.2 Software Background and Trends

Background information on each of the programming models examined as part of this research is presented in this section, together with information on existing and likely future trends in their development. The intra-node programming models are examined initially followed by those which can be utilised to implement inter-node parallelism.

2.2.1 OpenMP

OpenMP is an Application Program Interface (API) and has become the de facto standard in shared memory programming [156]. The technology is supported by all the major compiler vendors and is based on a *fork-join* model of concurrency. It consists of a set of pragmas that can be added to existing source code to express parallelism. An OpenMP-enabled compiler is able to use this additional information to parallelise these annotated sections of code.

The model is primarily focused at implementing intra-node parallelism, with OpenMP programs requiring a shared memory-space to be addressable by all threads. At present the technology only supports CPU-based devices although proposals exist in OpenMP version 4.0 for the inclusion of additional directives to target accelerator based devices such as GPUs [187]. This has been implemented to varying levels in a number of compilers.

2.2.2 OpenCL

OpenCL is an open standard that enables parallel programming of heterogeneous architectures. Managed by the Khronos group and implemented by over ten vendors—including AMD [12], Intel [98], IBM [88], and Nvidia [150]—OpenCL code can be run on many architectures without recompilation. The programming model is similar to CUDA, developed by Nvidia.

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

Generally, an OpenCL *device* maps to an entire CPU socket or an attached accelerator. Additionally, on CPU architectures it is normal for both *compute units* and *processing elements* to be mapped to the individual CPU cores. On GPUs, however, this division can vary, with *compute units* typically being mapped to a core on the device, and processing elements to functional units within these cores.

Kernels are executed in a SPMD manner across a one, two or three dimensional range of *work-items*, with collections of *work-items* being grouped together into *work-groups*. *Work-groups* map directly onto a *compute unit* and the *work-items* which they contain are executed by the *compute unit's* associated *processing elements*. The *work-groups* which make up a particular *kernel* can be dispatched for execution on any available *compute units* in any order. On a CPU, the *work-items* within a *work-group* are generally scheduled for execution within one core, although this is not a strict requirement. If vector code has been generated, the *work-items* will be scheduled using SIMD instructions to utilise the vector unit within the particular CPU core. On a GPU, *work-groups* are generally assigned to individual cores and their *work-items* executed in collections across the *processing-elements* within the core. The collection size or width depends on the specific device vendor; Nvidia devices utilise collections of 32 *work-items* whereas AMD devices use collections of 64.

The programming model provides no global synchronisation mechanism between *work-groups*, although it is possible to synchronise within a *work-group*,

which enables OpenCL applications to scale up or down to fit different hardware configurations. It also includes a sophisticated queuing mechanism, which is able to express complex dependencies between kernels and manage multiple target devices. OpenCL is therefore able to easily express both task and data parallelism within applications.

2.2.3 CUDA

Nvidia's CUDA [149] 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*) onto 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, each of which is assigned a unique global-id. *Threads* are grouped into *thread-blocks* that each execute on a single GPU multi-processor contained within an Nvidia GPU, although several *thread-blocks* can reside concurrently on each multi-processor. *Kernels* are thus executed as a grid of *thread-blocks* which collectively contain all the aforementioned *threads*. *Threads* within a *thread-block* can cooperate and synchronise via shared memory which is local to a particular multiprocessor, however, there is no support for global synchronisation between *threads* in different *thread-blocks*. This explicit programming model requires applications to be restructured in order to make the most efficient use of the GPU architecture and thus take advantage of the massive parallelism inherent in them. Constructing applications in this manner also enables *kernels* to scale up or down to arbitrary sized GPU devices.

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

2.2.4 OpenACC

OpenACC [155] is a high-level, pragma based programming model intended to provide support for many-core technologies from within standard Fortran, C and C++. Driven by the CAAR team at ORNL [27] and supported by an initial group of three compiler vendors, although one vendor (CAPS) has since ceased trading. The technology enables 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. This approach minimises the modifications required to existing codebases and eases programmability, whilst

also providing a portable, open standards-based solution for many-core technologies. The technology potentially provides a solution for targeting applications at complicated hardware technologies without the requirement for developers to learn complex, sometimes vendor specific, languages or to understand intricate hardware details. The standard is still, however, relatively new and implementations are still maturing.

Prior to a common OpenACC standard being agreed, Cray, PGI and CAPS had each developed their own proprietary accelerator directives, which formed the basis of their OpenACC implementations. PGI developed their `region` construct, within their original Accelerator model [162] for Nvidia GPUs, into their implementation of the OpenACC `Kernel` construct. Whilst Cray originally proposed accelerator extensions to the OpenMP standard [44] to target GPGPUs through their CCE compiler, they developed their proposal into the `Parallel` construct within the OpenACC standard. CAPS originally developed support for accelerator devices through their OpenHMPP directive model [39], although their OpenACC compiler still required the utilisation of a third-party host compiler.

Each implementation now supports both the `Kernel` and `Parallel` OpenACC constructs. The main differences between these constructs relate to how they map the parallelism, present in the particular code region which is being accelerated, to the underlying hardware. The `Parallel` construct is explicit, requiring the programmer to highlight loops for parallelisation within the code region; it closely resembles several OpenMP constructs, such as the OpenMP `parallel do` pragma. The `Kernel` construct, however, enables code to be parallelised and accelerated implicitly.

The three implementations also utilise a range of different *"back-end"* code representations in order to actual execute OpenACC applications on target hardware devices. The CAPS compiler translated code directly to either CUDA or OpenCL, whilst PGI originally only supported CUDA they have since also released support for OpenCL. The generated CUDA code can, therefore, only be utilised to target applications at Nvidia GPU devices through the NVCC compiler, however, the use of OpenCL enables a larger range of devices to be supported. Cray CCE, however, only generates low-level Nvidia PTX [151] instructions from the OpenACC directives, which consequently constrains their implementation to Cray architectures with attached Nvidia GPU devices.

2.2.5 VHDL and Verilog

VHDL [91] and Verilog [90] are both low-level HDLs (Hardware Definition Languages) originally developed by the DOD and Cadence Design Systems,

respectively. They are now both standardised by the IEEE and used heavily within the EDA (Electronic Design Automation) community to describe logic circuits in a textual format, predominantly at the register transfer level. Digital systems can therefore be designed and verified using these languages, although they can also be employed for mixed-signal and analogue system designs.

Although loosely based on procedural programming languages, Ada and C respectively, their models differ significantly from traditional procedural programming languages, as they contain mechanisms to describe electrical signal propagation times and strengths, rather than just logical functionality. Both languages employ a data-flow model of computation and enable parallel/concurrent systems of circuits to be described.

Circuit designs described in either language can be tested using logic simulators. Synthesis tools can then subsequently be employed to generate actual hardware circuit representations which can then be used to create ASICs or to program FPGA devices. Developing solutions using either of these low-level approaches requires high levels of expertise and experience and is often extremely time/resource consuming and error prone. This usually precludes their use within the scientific application development community. Although these technologies can potentially enable extremely performant solutions to be developed.

2.2.6 BSP Programming Model

The BSP programming model was originally proposed by Valiant [197] as an abstraction model for the design of parallel applications. The model bridges the divide between software and hardware by abstracting some of the details of the underlying parallel computing devices. It improves on other models such as the PRAM (Parallel Random Access Machine) by enabling communication and synchronisation costs to be accounted for.

The model is comprised of a collection of processing resources which have access to their own dedicated local memories and an interconnect fabric to facilitate pair-wise communication and synchronisation between all or a subset of the processing elements. An overall computation is formed from a series of global *"supersteps"* in which processing elements may each concurrently perform computation on their local memory resources. Communications, which can either be one- or two-sided operations, can occur between processes during each *superstep*, these do not need to be ordered and may also be overlapped with computation. A barrier operation exists at the end of each *superstep* which causes all processes to be synchronised before they proceed to the next *superstep*. All computation and communication from the preceding *superstep* is therefore

completed before the next one commences. The model thus maps well onto the architectures of most modern HPC systems and has become the de facto approach for developing parallel applications for them.

2.2.7 MPI Programming Model

As cluster-based designs have become the predominant architecture for HPC systems, the MPI programming model has become the de facto 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 [144]. MPI programs are based on the MPMD (Multiple Program Multiple Data) paradigm in which each process (or rank) asynchronously executes a separate (but potentially identical) program, with each rank therefore able to independently follow different execution paths within their associated programs. Each process makes calls directly into the MPI library in order to make use of the communication and synchronisation functions that it provides; both *point-to-point* and *collective* communication operations are provided by the library.

The technology is thus 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. Although support for one-sided communication has been available since version 2.0 of the standard, these constructs are not currently widely used and have been enhanced significantly in version 3.0. MPI version 3.0 also introduced several new collective operations such as *neighbourhood* and *non-blocking* collectives, which although not widely supported yet, claim to offer performance and productivity benefits in particular circumstances.

2.2.8 PGAS Programming Model

PGAS-based programming models aim to provide the ease of shared memory approaches such as OpenMP (Section 2.2.1) whilst also providing the performance and scalability of message passing based approaches such as MPI (Section 2.2.7). To implement shared memory constructs they utilise a global address space and a one-sided communication model to potentially enable processes to access any memory location. This global address space is, however, logically partitioned with each segment assigned to a particular processing element within the overall application. The model is thus able to express memory access locality and maps well to the architecture of current generations of HPC platforms, which

facilitates improved performance and scalability, potentially equivalent to or greater than that of the message passing model. It has also been recognised that the per-message overheads of models such as MPI may not be reducing sufficiently for MPI to be practicable on exascale system architectures, potentially necessitating the use of PGAS-based approaches [11].

Numerous PGAS languages and programming models are currently in existence including but not limited to: UPC, Global Arrays, X10 and Chapel; each of which is targeted at a different user-base and is subtly different in their particular implementation of the general PGAS approach. This thesis examines the applicability of two additional PGAS implementations, CAF and OpenSHMEM, to explicit hydrodynamics applications and provides background information on each of these models in the following sections.

The CAF Programming Model

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. These extensions were originally proposed in 1998 by Numrich and Reid as a means of adding PGAS concepts into the main Fortran language, using only minimal additional syntax [148].

CAF continues to follow the SPMD (Single Process Multiple Data) language paradigm with a program being split into a number of communicating processes known as *images*. The number of images is defined at runtime and is static throughout the execution of the program; no language facility exists yet for dynamic *image* creation. Communications are all 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 [19]. In contrast to OpenSHMEM, CAF employs a predominantly compiler/language 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.

The CAF language also enforces a local view of computation, requiring programmers to explicitly manage data locality and communication. Objects are declared to be co-arrays using an additional syntax operator *"[ ]"*. Any object, both arrays and scalars, can be declared as a co-array and when declared as such a copy of this object must exist, and be of the same size, on each image within the overall CAF program. The square brackets essentially assign an

additional dimension (potentially multiple dimensions) to a particular object, enabling the object to be uniquely referenced by other images. Images can use the *"( )"* notation to access the elements of a local array but must use a combination of both notations *"( )[ ]"* in order to access the elements of remote co-array objects, whether they reside within the local or a remote node.

Two forms of synchronisation are available within the language, the `sync all` construct provides a global synchronisation capability, whilst the `sync images` construct provides functionality to synchronise particular subsets of images. Collective operators have not yet been standardised, although Cray have implemented their own versions of several commonly used operations. Additionally no support exists for image *"teams"* or communicators within the current Fortran 2008 standard.

The OpenSHMEM Programming Model

The SHMEM programming model was originally developed by Cray for their T3D systems [81]. Although the technology has existed for some time, it was only recently standardised in 2012 as part of the OpenSHMEM initiative [40, 157]. 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 RDMA support from the underlying system layers. These constructs also purport to offer potentially lower latency and higher bandwidth than alternative approaches.

OpenSHMEM is not explicitly part of the Fortran and C language standards 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 operates at a 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 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 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.

2.2.9 Hybrid Programming Models

Hybrid, potentially multi-resolution, programming approaches have been recognised as promising areas of research for enabling applications to achieve the scalability required for exascale levels of computation on future platforms [11, 206, 128]. They typically utilise models such as OpenMP or OpenCL (Sections 2.2.1 and 2.2.2) to express intra-node parallelisation, together with MPI or the PGAS approaches (Sections 2.2.7 and 2.2.8) for inter-node communication.

A purported advantage of these approaches is that they potentially facilitate reductions in overall memory usage, which will be crucial given the trend towards reduced memory capacities and access bandwidths in future system architectures (Section 2.1). The use of these programming models can achieve these reductions by enabling data structures to be shared between different threads of execution within the individual systems nodes, which would otherwise be duplicated within each MPI/PGAS process. The number of MPI/PGAS processes can also be substantially reduced through the utilisation of these models, which potentially facilitates improvements in scalability by reducing the overall amount of memory required by the inter-node communication runtime systems. Additionally inter-node communication messages can also be aggregated into fewer larger messages, potentially improving performance in particular situations, and reducing message injection rate requirements.

This is an extremely active area of research and it has been recognised that it will be necessary to improve the integration of the inter- and intra-node runtime systems in order to achieve exascale levels of computation [11, 128]. Additionally, it has also been shown that the optimal ratio of OpenMP to MPI can change depending on specific application characteristics, the problem size being simulated and the scale of the particular experiment, necessitating further research [11]. Similarly how to optimally combine the constructs of both models within applications is also a subject of much debate. It has been shown, for example, that for some applications, performance can be improved by incorporating calls to the MPI routines within OpenMP threaded code regions rather than within serial code regions [11].

2.2.10 Current & Future Trends

The trend towards many-core devices (Section 2.1.3) and the potential incorporation of accelerators into future exascale systems will necessitate the creation of new programming abstractions, including new threading models with improved thread control semantics for thread placement, launching and synchronisation as well as more scalable runtime systems [11, 206]. Improving support for more fine-grained, potentially nested, parallelism within programming models will also likely become increasingly important [11, 206, 128]. It has also been recognised that the exclusive use of existing relatively heavy-weight threads will not be able to meet exascale requirements, necessitating the development of more light-weight models supporting task parallelism [11, 206, 128]. This, together with the fact that it is argued that the scalability of OpenMP implementations needs to be significantly improved in order to facilitate the creation of exascale systems, indicates that the exploration of programming models similar to OpenCL may be worthwhile [11]. The ability to coordinate dynamic task teams is also likely to be required on future system architectures and future NOC processor designs will likely necessitate the inclusion of topology awareness within applications at the node level [11, 128].

Data movement has been forecast to be extremely expensive relative to the cost of floating-point operations in future supercomputer system designs (Section 2.1.2). This may potentially necessitate the creation of programming models which are able to capture the cost of data movement and can better express data locality, in order to reduce the amount of data actually transferred [11, 206, 196, 115, 128]. The creation of intelligent runtime systems to handle data movement are also likely to be required, together with increasing the levels of asynchronicity within applications [11, 128].

Due to the increased levels of parallelism, the consequences of load imbalances are also likely to be considerably more significant at exascale. This may potentially require new programming models to be considered as alternatives to SPMD, which may be too restrictive. Strong-scaling is also likely to become increasingly important in inter-process parallelism, potentially further necessitating a move towards more fine-grained parallel models. Developing topology-aware communication mechanisms and optimising the mapping of application processes within the overall interconnect fabric will therefore be increasingly required. Additionally, undertaking research to improve the underlying scalability of algorithms and software (including both one- and two-sided models) is also likely to become increasingly important. Similarly programming models will need to be able to scale from one node up to the full machine size of an exascale-class system, and it is recognised that both unified and hybrid

programming models are still candidates for achieving this [11].

The portability and productivity of a programming model, across both machine architectures and applications domains, as well as its ability to deliver portable performance have been recognised as crucial requirements. Furthermore it is also the case that this will become increasingly difficult to achieve on future machine architectures. It is therefore likely that improving the hierarchical interoperability between languages and programming models will be required, together with an increased use of auto-tuning solutions to improve the performance portability of applications. Similarly approaches that enable expert, performance orientated programmers, as well as domain scientists (non-expert programmers) to simultaneously collaborate on the development of software at different levels of abstraction, are likely to be necessary [11, 128]. Maintaining a clear separation of concerns between the development of system components, which has been shown to boost productivity, may also be required [11, 128].

It has been forecast that the resilience or reliability of future supercomputing systems will likely become increasingly problematic as the scale, and the levels of inherent parallelism within them, increase. Applications are unlikely to be able to rely exclusively on hardware-based error detection and correction approaches and may therefore need to incorporate explicit mechanisms within the software [11, 115]. Additionally it is also forecast that the check-point restart resiliency approach will not scale to exascale capable systems, necessitating applications to be designed to tolerate hardware failures [11, 128].

2.3 Hydrodynamics Mathematical Foundations & Applications

This section presents background information on the system of hydrodynamics equations (Section 2.3.1) which the CloverLeaf mini-application (Section 1.6) solves. Motivational factors for improving the state-of-the-art within this area of science are also documented within Section 2.3.2.

2.3.1 Euler's Equations of Compressible Fluid Dynamics

Euler's equations of compressible flow [87, 42] are a system of three partial differential equations and are mathematical statements of the conservation of mass, momentum and energy, Equations 2.1 to 2.3 within Figure 2.1 present these statements respectively. These are expressed in conservation form although the numerical method employed in CloverLeaf (Section 1.6) does not conserve kinetic energy and therefore also the total energy within the system. This is

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.1}$$

$$\frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\mathbf{u} \otimes (\rho \mathbf{u})) + \nabla p = 0 \tag{2.2}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot (\mathbf{u}(E + p)) = 0 \tag{2.3}$$

$$pV = nRT \tag{2.4}$$

in which:

- ρ denotes the mass density
- $\mathbf{u}$ denotes the velocity vector
- E is the total energy per unit volume
- p represent pressure
- $\otimes$ is a tensor product
- 0 is the zero vector
- V represents volume
- n denotes the amount of the gas in moles
- R is the universal gas constant
- T represents temperature

Figure 2.1: The Euler equations of compressible flow

a natural consequence of the use of a staggered grid (Figure 1.1a), in which velocities are modeled at the nodes and kinetic energy is modelled separately to internal energy [24]. Consequently it is only possible to conserve momentum (mv) and not kinetic energy (mv^2). Internal energy refers to the temperature of the material within each cell, whereas kinetic energy captures the energy due to the motion of the material. The greater the internal energy of a cell, the harder it is to compress.

The right hand sides of Equations 2.1 to 2.3 each sum to 0, this captures the fact that each particular physical quantity (e.g. mass) is being conserved and therefore that overall the particular physical property is neither being created or destroyed. Equation 2.1 states that the rate of change of density is equal to the divergence of the product of density and velocity. The flow of density throughout the system therefore has to balance out and sum to zero overall. Equation 2.2 states that the rate of change of momentum is equal to the divergence of momentum plus the acceleration term (∇p). Specifically, that the momentum of a cell depends on its existing momentum and the force ($F = ma$) which is being exerted on it due to the pressure gradient. Finally, Equation 2.3 captures the conservation of energy principle and states that the rate of change of energy is equal to the divergence of energy plus pressure, and therefore that the overall energy of a cell depends on the work being done to it.

A fourth auxiliary equation of state, such as the ideal-gas equation of state (Equation 2.4), is employed to close the system of equations and enables the derivation of a unique solution. The ideal-gas equation of state captures the relationship between the constituent variables in Equations 2.1 to 2.3. It thus enables the exact physical condition of matter to be modelled, due to the particular set of properties currently being simulated. For example, it enables the pressure of each cell to be calculated based on properties such as the internal energy of each cell. Émile Clapeyron first proposed the ideal-gas law in 1834 as a combination of Boyle's law, Charles' law and Avogadro's law [208]. Currently within CloverLeaf the system is solved for three unknown variables: energy, density and momentum.

The Euler equations are capable of modelling, convecting and creating vorticity and consequently they are often employed to simulate vortical flows caused by either shocks or artificial mechanisms such as fixed stagnation points [168]. Additionally, they also represent an intermediate point in the hierarchy of equations which lead to the Navier Stokes equations.

The equations are generally solved using explicit numerical methods due to the fact that stable hydrodynamics simulation time-steps scale proportionally to $1/(\textit{overall mesh size})$, which makes explicit time-stepping computationally tractable. Explicit methods also generally produce second order accurate solutions in both time and space, in contrast to implicit methods which are generally only first order accurate. Additionally, the use of explicit methods enables the equations to be solved without the requirement to globally invert a matrix within the simulating application, thus avoiding a computationally expensive operation. Implicit methods also do not model physical discontinuities–such as shock waves or density jumps–very accurately, and can lead to the smearing of these distinct feature and oscillations.

2.3.2 Motivations for Improving the State-of-the-art

Lagrangian-Eulerian simulation methods have established themselves as one of the most dominant approaches for solving the hydrodynamic equations for compressible flow [173]. To achieve accurate numerical solutions, a converged mesh resolution is required. Lagrangian-based approaches can achieve accurate solutions to problems involving multiple materials and moving boundaries, as the mesh is able to move naturally in unison with the motion of the material [127]. A purely Lagrangian-based approach can be problematic due to vorticity or strong shearing forces within the simulation, causing the computational mesh to distort and potentially become tangled [127, 173]. This necessitates the incorporation of Eulerian-based approaches to reset or relax the mesh in order to achieve more

accurate solutions. Additionally, for complex flows that generate interacting shock waves, the mesh resolution required around shock fronts can be very small when compared to the size of the entire domain.

It is widely recognised that achieving accurate solutions to some of the most significant challenges in Lagrangian-Eulerian explicit hydrodynamics simulations, across a wide range of scientific domains, require computational resources that are not currently available [97, 127, 77]. In particular the simulation of the high-energy hydrodynamic physics processes which scientists rely upon to understand, for example, the properties of supernovas or space weather, and the inertial fusion energy (IFE) gain from projects such as the National Ignition Facility (NIF) in the USA, require such scales of computational facilities. Similarly it has also been recognised that improving current hydrodynamic simulation capabilities could enable significant advancements in medicine, potentially facilitating the delivery of "real-time" simulations during surgery [117]. To reach the required resolutions/fidelities and reductions in time-to-solution, huge numbers of floating-point operations and very large amounts of memory are therefore required. Calculations at this scale require extreme levels of processing resources, which will only become available with exascale supercomputers.

Exascale capable supercomputing systems will therefore be needed to reach the required levels of simulation accuracy, and current methods (specifically algorithms and codes) need to be re-evaluated and significantly improved, if researchers are to have access to applications which can effectively utilise the computational capabilities of future computational platforms. Designing and preparing codes, which can achieve such calculations across a large domain requires significant additional community research. To date insufficient work has been undertaken to examine how explicit hydrodynamics applications can be optimised to achieve exascale levels of performance and, into the supporting programming models and technologies which can best facilitate this transition. CloverLeaf (Section 1.6) is representative of a wide-class of explicit Lagrangian-Eulerian hydrodynamics applications, including those used to model high-energy physics processes.

2.4 Summary

This chapter has documented details of the previous, current and likely future development trends of the hardware components used to construct HPC systems, as well as the primary design constraints which are currently influencing their development. The implications of these trends for the future development of scientific applications, and the software technologies used to construct them, are also discussed.

Additional background information was presented on several state-of-the-art intra-node programming models, which are examined within this research. Some of these are already well-established within the HPC community, whilst others are relatively new and aim to deliver some of the advanced features (such as fine-grained parallelism and improved application portability) which will be necessitated by current hardware development trends, in order for applications to optimally utilise future hardware platforms. The de facto inter-node programming model used within HPC application development, MPI, is also discussed together with several issues which may constrain its scalability on future system architectures. Furthermore, information on several proposed alternative solutions (hybrid-programming and PGAS models), which purport to resolve some of these issues, were also presented. Specifically, the CAF and OpenSHMEM PGAS models, which are evaluated within this research, were documented.

Finally, the system of hydrodynamics equations solved by the scientific applications, which are the focus of this work, were described together with several motivating factors for improving the capabilities of these applications for simulating complex phenomena on future platforms.

CHAPTER 3
Intra-Node Performance Optimisations

This chapter documents the work undertaken as part of this research to develop techniques for improving the intra-node performance of the CloverLeaf mini-application, and to thereby also improve the larger production explicit hydrodynamics applications which it represents. The work focuses exclusively on the OpenMP-based version of the codebase and examines several candidate optimisations, ultimately the end goal was to develop an optimal OpenMP-based version of the codebase. In particular several key objectives included developing optimisations to improve the performance of the codebase on the Intel Xeon Phi architecture and in situations in which OpenMP parallelism is employed across multiple processor sockets and NUMA (Non-Uniform Memory Access) domains within individual compute nodes. The chapter initially discusses some related and motivating work within this arena (Section 3.1) and then, in Section 3.2, documents the current implementation of the mini-app in the OpenMP programming model as well as each candidate optimisation examined. The performance of these potential optimisations on two current state-of-the-art processor architectures is analysed in Section 3.3. Finally, Section 3.4 concludes the chapter.

3.1 Related Work

Optimising OpenMP-based applications has been studied extensively for a number of years and improvements to enhance data locality and NUMA region affinity [188, 53] as well as iteration partitioning and scheduling strategies [154] have been proposed. The performance of nested-parallelism within OpenMP was studied in [54, 190] using a range of applications and micro-benchmarks. The scalability of barrier and synchronisation algorithms for OpenMP has also been examined and various approaches for improving the available synchronisation constructs have been proposed [146]. These include approaches based on Phasers [181, 180] and *point-to-point* synchronisation [36]. Developing an OpenMP implementation for a SOC incorporating a large number of processor cores was also studied extensively in [49].

Additionally in [126] Liu *et al.* examine an approach based on the privatisation of array sub-sections as a mechanism for converting OpenMP programs to an SPMD style of computation. To facilitate the implementation of similar optimisations Hernandez *et al.* have also developed a tool to analyse the memory

access patterns of OpenMP programs [83]. Several studies have also evaluated the performance of the OpenMP programming model at high thread counts on the Intel Xeon Phi co-processor [170, 37, 43]. The importance of appropriately vectorising applications on the Xeon Phi co-processor is emphasised in [16, 193], together with several techniques for improving the levels of vectorisation within existing applications.

3.2 OpenMP-based Optimisations Examined

This section documents the reference implementation of the OpenMP-based version of CloverLeaf as well as the techniques examined (Sections 3.2.1 to 3.2.12) to improve the single-node performance of the codebase.

The reference implementation is an evolution of the serial version of the codebase in which OpenMP constructs are utilised to provide intra-node parallelism. OpenMP `parallel` regions are employed within each of the 14 computational kernels i.e. at the lowest level within the call-graph of the application. To minimise the *fork/join* overheads inherent in the OpenMP programming model one `parallel` region is employed per kernel; each region therefore potentially encompasses several loop-blocks. To enable individual loop-blocks within the computational kernels to be parallelised over the available threads, additional OpenMP `do` constructs are employed, generally around the outer-loops of each loop-block. OpenMP `private` constructs are specified where necessary to create temporary variables that are unique to each thread, whilst `reduction` primitives are employed to implement intra-node reduction operations at two locations.

During this research certain optimisations were applied only to particular kernels—these are clearly identified in the following sections—whilst others were implemented throughout the entire codebase. Each technique was initially utilised in isolation to implement alternative versions of the codebase, however, several of these techniques were subsequently combined to produce further versions of the application. Additionally, Section 3.2.13 also describes research undertaken with the Cray Reveal tool in order to automatically generate an OpenMP based version of the codebase. Section 3.3 analyses the effect of each of these alternative approaches on the performance of the mini-app.

3.2.1 First-touch Memory Placement

Modern multi-processor systems generally exhibit non-uniform memory access times between the local memory sub-system of a processor and those located on different processors within the same node. When executing threaded programs across multiple sockets it is therefore important to ensure that threads primarily

```
!$OMP PARALLEL
    array=0.0

!$OMP END PARALLEL
```

(a) Reference

```
!$OMP PARALLEL
  !$OMP DO
  DO k=y_min,y_max
    DO j=x_min,x_max
      array(j,k)=0.0
    ENDDO
  ENDDO
  !$OMP END DO
!$OMP END PARALLEL
```

(b) First-touch

Figure 3.1: The modified "first-touch" memory initialisation approach

access memory resources located in the memory sub-system of their local processor and therefore minimise inter-socket memory accesses. Memory locations allocated by an application are also only mapped into actual physical memory once they are first accessed or "*touched*". Once accessed these allocations will be mapped into the memory sub-system physically local to the processor on which the particular accessing thread is executing.

The reference implementation originally employed an approach which initialised each entire 2D-array using Fortran 90 array assignment syntax within an OpenMP `parallel` region (Figure 3.1a). This created a data-race condition in which each OpenMP thread attempted to initialise all array elements. This was not detrimental to performance when threads were contained within one processor socket as regardless of the thread execution orderings all memory locations were mapped to the same physical memory sub-system of the local processor. The initialisation code was also located outside of the main timing loop of the application. In situations in which OpenMP parallelism is utilised across multiple sockets, however, this approach resulted in significant memory affinity problems.

To address this situation a modified approach (Figure 3.1b) was implemented which ensured that each thread only initialised the memory locations for which it was directly responsible, thus ensuring that these memory locations were physically mapped as close as possible to the particular thread. In this modified implementation the additional double-loop block and the OpenMP `do` parallelisation construct ensures that individual threads will only access particular sets of rows from the 2D-array (named "array" in Figure 3.1) and that these sets will be contiguous but non-overlapping between different threads. Versions which employ this modified "NUMA-aware" approach contain the acronym *ft* (First Touch) within their descriptions in Section 3.3.

3.2.2 *Array-of-arrays* Data Structure

Memory allocations are, however, physically mapped into the memory sub-system of a node at the granularity of individual memory pages. This occurs when a memory location allocated within the particular memory page is first accessed, the page is then mapped into the memory sub-system which is directly connected to the processor on which the accessing thread is executing. It is therefore possible, and often the case, for the contiguous sets of memory locations accessed by different threads to reside within the same memory page. This does not usually represent a problem when OpenMP is only utilised within individual sockets, as no matter which thread *"touches"* this memory first, the page containing the memory locations for all of the threads will always be mapped into the same memory sub-system, which is local to all of the executing threads. When OpenMP is utilised across multiple sockets, however, this can result in the creation of race conditions between threads located on different sockets, potentially allowing the particular memory locations to be mapped onto any of the sockets within the node, depending on thread execution orderings. This is particularly problematic when huge-pages are utilised to reduce pressure on the TLB (Translation Lookaside Buffer) sub-system and can result in large amounts of memory being mapped onto the wrong sockets in a sub-optimal manner. This causes threads to incur additional overheads by having to access memory locations across the inter-socket bus network.

To ensure that the memory locations managed by different threads are always allocated on different memory pages, the data structures within the application were modified from standard 2D-arrays into an *"array-of-arrays"* configuration. In this approach each 2D-array, which contains information on a particular physical property (e.g. density), is split into one "top-level" array which contains multiple sub-arrays, one for each row of the original 2D-array. The sub-arrays are each allocated and initialised separately by the thread which is responsible for managing those particular memory locations, ensuring that each is located on a separate memory page. Thus each sub-array will be mapped within the local memory sub-system of the processor on which its managing thread resides, regardless of any differences in thread execution orderings. This optimisation is therefore mainly targeted at improving application performance in situations when OpenMP is employed across multiple sockets within each system node (e.g. one MPI process per node). Within Section 3.3 versions which employed this optimisation contain the acronym *AoA* (Array of Arrays) within their descriptions.

3.2.3 Data Alignment & Cache Line Padding

To potentially increase the efficiency of load and store memory operations as well as to assist compilers with the implementation of optimisation techniques such as automatic vectorisation, additional versions of the codebase were created which incorporated specific directives to align all data arrays on appropriate byte boundaries. This was achieved under the Intel software tool-chain using a combination of compiler options (e.g. `-align arraynbyte`) and source code directives (`!dir$ attributes align:64 :: array`) to inform the compiler to align the particular data arrays. As compilers cannot generally assume that arbitrary data passed into subroutine calls is appropriately aligned, additional source code directives (e.g. `!dir$ attributes align:64 :: array` and `!dir$ vector aligned`) were also employed, at the required locations throughout the codebase.

To eliminate any *"false sharing"* of cache lines between OpenMP threads and further improve data alignment additional versions were created which inserted redundant memory locations into the array allocations. These *"padded"* the rows of the arrays such that each starts on an appropriately aligned cache line boundary. This ensures, therefore, that no cache lines are shared for writing, between two different threads. Versions which employed this optimisation contain the word *Cpadd* within their descriptions in Section 3.3, whereas versions which utilise the previous data alignment optimisations are denoted using the word *Align*.

3.2.4 High-level OpenMP Parallel Region

When a process encounters an OpenMP `parallel` region a number of threads are created, or *"forked"*. An implicit global synchronisation operation also exists at the end of each `parallel` region, at which point threads are *"joined"* back into the main process thread and control continues serially. The reference implementation employs a design strategy in which one `parallel` region is utilised per computational kernel. Consequently the invocation of each kernel routine forces the OpenMP runtime to initially *"fork"* control to the required number of threads at the start and *"join"* these threads back into the main process at the end of its execution. This potentially incurs significant additional overheads particular for large thread counts.

To potentially alleviate these threading overheads an additional version was developed which raised the OpenMP `parallel` regions from each *bottom-level* kernel and combined them within the main *top-level* application routine. In this modified approach the start of the one remaining `parallel` region is only encountered once during the execution of the application. The threads created

within this region are thus maintained throughout the entire execution of the application and are only *"joined"* back into the main process at the end of this *"top-level"* routine, i.e. when the application is terminated. This also required the inclusion of additional OpenMP directives such as `!$omp master` and `!$omp barrier` at critical points throughout the program, and the creation of additional `thread-private` variables, in order to prevent race-conditions and ensure program correctness. Versions which incorporated this candidate optimisation are denoted by the acronym *hlpr* (High Level Parallel Region) within their descriptions in Section 3.3.

3.2.5 Duplicating Constant Data per NUMA-region

CloverLeaf utilises several 1D-arrays to store particular properties relating to the simulated mesh cells. Once initialised the values stored within these arrays remain constant throughout the execution of the application. When the reference OpenMP implementation is utilised across multiple sockets these arrays are generally stored such that half of the elements in each array are located within each NUMA-region. All of the values within these arrays are required by each application thread; consequently this results in significant numbers of memory accesses across the inter-socket communication bus to the remote NUMA-region.

As the contents of these arrays remain constant throughout the execution of the application it is possible to create copies of each array which reside exclusively within a particular NUMA-region and for each application thread to be configured to only access its local copy of a particular array. To examine the effect of eliminating these remote memory references on the performance of the application, additional versions were developed which incorporate this optimisation (denoted by the word *dupConst* within Section 3.3). In these versions the array copies are created during the initialisation phases of the application, pointers—which are declared as *private* to each application thread—are then initialised to reference the local copy of a particular array within each thread. Threads thus proceed to access these arrays through the appropriate local pointer. The implementation of the *High-level OpenMP Parallel Region* optimisation described in Section 3.2.4 is required in order for the contents of these pointer variables to persist throughout the execution of the application.

3.2.6 Explicit Loop Schedules

To automatically parallelise the loop iterations across the available threads, the reference implementation utilises OpenMP `do` directives, generally on the outer loop of the double loop-blocks within each kernel. In this approach the OpenMP runtime system calculates how to actually partition the iterations

of each loop-block when the corresponding `do` directive is encountered, based on the total number of iterations and whether particular `schedule` clauses are specified. An additional optimisation was therefore implemented, which utilised explicit iteration allocations between threads for each loop-block, to remove any overheads incurred due to the OpenMP runtime loop partitioning and scheduling. These schedules are pre-calculated during application initialisation, depending on the particular problem size being simulated, and are stored within pairs of dedicated arrays which each contain one entry for each OpenMP thread. One array contains the starting iteration number of each thread for a particular loop-block, whilst the second array stores the final iteration number. Upon encountering a particular loop-block each thread uses its identification number to access its unique location within these arrays and to obtain the iteration range which it should process (Figure 3.2). The OpenMP `do` directives can thus be completely removed from the loop-blocks. Versions which employ this candidate optimisation contain the initials *ELS* (Explicit Loop Schedules) within their descriptions in Section 3.3.

3.2.7 Inter-thread Synchronisation Elimination

Reducing synchronisation within applications is recognised as a potential optimisation to increase the scalability of OpenMP applications. OpenMP `do` constructs contain an implicit global synchronisation at the end of each construct, which is often not required by the application. In situations in which no dependencies exist between threads, `nowait` directives were added to the `do` constructs to remove the implicit synchronisation operations. Versions which employ this technique contain the word *nowait* within their descriptions in Section 3.3.

3.2.8 Reducing Inter-thread Synchronisation

In situations in which dependencies do exist between threads it is frequently the case that these are only present between immediately adjacent pairs of threads, e.g. due to stencil operations in the y-dimension of the mesh. Consequently global barrier operations, which synchronise all of the threads, are often not required, potentially computationally expensive and do not allow for the execution of different code regions to be overlapped between different threads. To examine whether alternative approaches, which reduce overall synchronisation requirements, could deliver any performance benefits an approach, similar to the pseudo code in Figure 3.2 was implemented. This utilises explicit *point-to-point* synchronisation operations between threads, and was implemented for the *Cell-Advection* kernel within CloverLeaf.

```
!$OMP PARALLEL
        tid = omp_get_thread_num()

        k=loopblock1_ystart(tid)
        DO j=x_min,x_max
           ... loopblock1 code ...
        ENDDO

        !$OMP FLUSH(data arrays)
        Update loopblock2_locks(tid-1)
        !$OMP FLUSH(loopblock2_locks)

        DO k=loopblock1_ystart(tid)+1,loopblock1_yend(tid)
           DO j=x_min,x_max
              ... loopblock1 code ...
           ENDDO
        ENDDO

        DO k=loopblock2_ystart(tid),loopblock2_yend(tid)-1
           DO j=x_min,x_max
              ... loopblock2 code ...
           ENDDO
        ENDDO

        !$OMP FLUSH(loopblock2_locks)
        Busy_wait on loopblock2_locks(tid)
        Reset loopblock2_locks(tid)
        !$OMP FLUSH(data arrays)

        k=loopblock2_yend(tid)
        DO j=x_min,x_max
           ... loopblock2 code ...
        ENDDO

!$OMP END PARALLEL
```

Figure 3.2: OpenMP *point-to-point* synchronisation approach

To achieve this an array of *lock* variables was created for each loop-block which has a potential dependency on a loop iteration executed by another thread. These arrays are appropriately aligned and include sufficient memory *padding* to ensure that each *lock* resides on a completely separate cache line to avoid access conflicts and excessive cache-coherency traffic. Threads set these *lock* variables to indicate to their neighbouring threads that they have completed a particular operation and written their results to memory. Consumer threads are configured to continue execution until they require data produced by another thread, at which point they utilise *"busy-wait"* operations on the appropriate *lock*. OpenMP `flush` directives are employed to ensure data and *locks* are appropriately written back to, and read from, memory rather than cache. This approach also requires the utilisation of the explicit loop schedules described in Section 3.2.6.

Frequently it is the case that these inter-thread dependencies only exist between the first iteration of a loop-block in one thread and the last iteration of a subsequent loop-block in another thread. Figure 3.2 depicts such a situation. In cases such as these it is possible to separate the first iteration of loop-block

1 from the main body of the loop and to update the appropriate lock variable immediately following its execution, to communicate that the dependency is satisfied. As only the last iteration of the second loop-block contains the inter-thread dependency, this can also be separated from the main loop body, with a *"busy-wait"* operation being employed between them to ensure that the dependency is satisfied before a particular thread executes this iteration. Assuming computational load is well-balanced between threads, and the runtime interleaves thread executions fairly, this arrangement should ensure that threads do not have to synchronise ("*busy-wait*"), as by the time the last iteration of the second loop is reached, its dependencies should generally have already have been satisfied. This approach contributes to increasingly asynchronicity levels within applications as well as overlapping thread executions. Versions which incorporate these *point-to-point* synchronisation approaches are denoted using the word *p2psync* within Section 3.3.

Additionally, to completely eliminate any inter-thread synchronisation constructs the use of an explicit recalculation approach was also examined. In this approach when a dependency exists between two threads the code was re-factored to enable the "*consuming*" thread to actually calculate the required values rather than relying on the original "*producing*" thread. This typically involved a thread temporarily recalculating values for a row which is either immediately above or below it in the overall mesh. These would originally have been produced by the immediately adjacent threads within the overall decomposition. The threads store the recalculated values within temporary variables, with the original thread producing the final values which are ultimately stored within main memory. Versions which incorporate the recalculation approach to reduce inter-thread synchronisation are denoted using the word *recalc* in Section 3.3.

3.2.9 *Thread-private* Temporary Variables

Several computationally intensive kernels in the reference implementation utilise global 2D-arrays to store temporary intermediary values required throughout the execution of a particular kernel. Through the use of OpenMP *thread-private* temporary variables it was possible to reduce, and in some cases eliminate, the use of these temporary arrays. This ultimately has the effect of reducing the overall number of global memory operations within these kernels and also the overall memory storage requirements of the application. A detailed analysis of the codebase enabled this optimisation to be applied to the *Cell-Advection*, *Momentum-Advection*, *PdV*, *Accelerate* and *Calc-DT* kernels, which each perform a particular phase of the overall hydrodynamic simulation implemented

```
!$OMP DO PRIVATE( ... , temp_array) REDUCTION(MIN : dt_min_val)
  DO k=y_min,y_max
    !DIR$ SIMD VECTORLENGTH(CALCDTVECTORLENGTH)
    DO j=x_min,x_max

      .
      .
      temp_array(MOD(j-1,CALCDTVECTORLENGTH)) = MIN(...)
      dt_min_val = MIN(dt_min_val,MINVAL(temp_array))

    ENDDO
  ENDDO
!$OMP END DO
```

Figure 3.3: The "vectorising" version of the Calc-DT kernel

within CloverLeaf. Versions which utilised this optimisation are denoted using the description *privateVars* within Section 3.3. Implementing this optimisation frequently required the merging of loop-blocks within each kernel, when this was required for a particular version it is denoted using the word *merge* in the descriptions within Section 3.3.

3.2.10 Loop Vectorisation

An analysis of the vectorisation reports produced by the compiler for the reference implementation identified that the second loop-block within the Calc-DT kernel, which contained a reduction operation, could not be successfully vectorised. To enable this kernel to be fully vectorised by the compiler a subsequent version was developed (see Figure 3.3) in which the reduction loop was merged into the main loop-block of the kernel and the 2D-temporary array replaced with a smaller array of the same size as the vector length of the particular architecture. A `!dir$ simd vectorlength(calcdtvectorlength)` directive was then applied to the inner loop of the kernel in order to ensure that the compiler generated vectorised operations of a particular width, equal to the value of `calcdtvectorlength`. This was passed into the kernel via the compiler's pre-processor facility. The array was populated using the inner loop index to ensure that adjacent iterations store their values in different but contiguous array locations. This array variable was also declared as `private` to ensure that each thread maintained its own copy in which to accumulate temporary values. On the system containing the Xeon Phi processor architecture a later version of the Intel compiler was available which was able to successfully vectorise this kernel after only the loop merger and temporary 2D-array elimination optimisations. The versions denoted by *Kernel_Opts* within Section 3.3 incorporate these optimisations.

This analysis also identified that the *Field-Summary* kernel was also not being successfully vectorised by the compiler. The vectorisation reports produced

by the compiler identified that this was due to a perceived iteration dependency within a double loop-block nested within the main kernel loop-block. Manually unrolling this nested inner loop-block enabled the compiler to successfully vectorise the kernel. The versions denoted by the description *Kernel_Opts* within Section 3.3 also incorporate this optimisation.

3.2.11 *Accelerate* Kernel Optimisations

The reference implementation of the *Accelerate* kernel was implemented as a series of five consecutive loop-blocks each of which was parallelised using OpenMP `do` constructs. To potentially improve the performance of this kernel several candidate optimisations were examined, including applying OpenMP `nowait` directives to each loop-block to eliminate the synchronisation operations between them, versions which utilised this optimisation are denoted using the description *nowait* within Section 3.3. A subsequent version (denoted by the word *merge*) also examined the effect of manually merging these loop blocks together into one larger loop-block. Building on this an additional version (denoted by the word *privateVars*) examined a further optimisation which employed several temporary *thread-private* variables to eliminate the use of a global 2D-array, which was used to store temporary values within the original kernel. This also reduced the number of global memory operations required to update the persistent global 2D-arrays. The versions denoted by *Kernel_Opts* within Section 3.3 also incorporate these optimisations.

3.2.12 *Update-Halo* Kernel Optimisations

The *Update-Halo* kernel performs boundary reflections around the edges of the mesh region assigned to each process. In the reference version this kernel is implemented via collections of four double-nested loop blocks (one for each mesh edge). This arrangement is also repeated for each data-field whose values need to be reflected. As the x-dimension of each mesh is stored within contiguous memory locations it is generally more efficient for each thread to access memory sequentially along the rows of each array. This necessitates that the inner j-loops traverse each row and for the OpenMP parallelism to be applied to the outer k-loops which iterate over individual rows. For the *halo*-updates to the top and bottom mesh edges this arrangement would involve the OpenMP parallelism being applied to a loop with a very short trip-count and therefore only generating a small number of threads, which would be inefficient. Consequently the reference implementation is constructed using inner k-loops and outer j-loops for the top and bottom *halo*-exchanges, which results in a sub-optimal memory access pattern. To potentially improve on this arrangement an additional version

(referred to as *UHinter*) was therefore created. In this modified version the original loops for the top and bottom mesh exchanges were interchanged, and an OpenMP `collapse(2)` directive was applied on the now outer k-loops. This causes the two loops to be coalesced into one larger iteration space and for this modified loop-structure to then be parallelised across application threads.

When OpenMP thread teams are utilised across multiple sockets this also results in the top and bottom mesh edges each being stored exclusively within the local memory locations of different processors, assuming two sockets per node. Performing the memory copy operations on each of these edges sequentially, as the reference implementation does, thus results in half of the threads accessing memory locations on the remote socket. A further potential optimisation was therefore examined which enabled this kernel to operate on both mesh edges simultaneously. The memory locations of each edge are therefore processed by only a subset of the threads which have an affinity to the processor that is strictly local to the memory locations of the particular edge. This was implemented using two levels of *nested*-parallelism and OpenMP v4.0 thread placement constructs. The first level of parallelism specifies that two threads should be created (`num_threads(2)`); and that by using the `proc_bind(spread)` directive each should be located on different processor sockets. Two lower-level OpenMP `parallel` regions were then subsequently employed, one per edge, each of which was contained within a separate OpenMP `sections` construct. A further `num_threads(X) proc_bind(close)` directive was utilised on these lower-level `parallel` regions to ensure that the required number of threads is created with an affinity to only the particular local processor. Versions which employed this candidate optimisation are denoted by the word *UHnested* within their description in Section 3.3.

The reference kernel implementation also performs a global synchronisation operation after each loop-block which operates on a particular edge of the mesh. To potentially reduce the number of synchronisation operations required, the kernel was restructured into two distinct phases. The first phase performed the necessary memory operations on both the top and bottom mesh edges, whilst the second operated exclusively on only the left and right edges. OpenMP `nowait` directives were applied to each loop-block within the kernel to remove the implicit synchronisation operation which occurs by default at the end of each OpenMP `do` construct. One OpenMP `barrier` construct was then employed between the two phases to provide the minimum synchronisation required by the hydrodynamics algorithm and ensure correct execution orderings. Versions which employed this optimisation are referred to using the word *UHnowait* within the descriptions in Section 3.3.

3.2.13 Automatic Application Hybridisation

It has been recognised that incorporating OpenMP directives into existing applications can be an extremely complex and time-consuming task. To alleviate this problem Cray developed the Reveal tool to automatically hybridise applications. Reveal provides functionality to perform an automated scope analysis of particular loop-blocks and to insert suggested OpenMP directives, for variable scoping and loop partitioning, into the codebase. As part of this research the tool was utilised to automatically hybridise the serial version of the CloverLeaf codebase, in order to produce a new hybrid version, denoted with the description *reveal* within Section 3.3. The tool successfully scoped all of the loop-blocks with the exception of three variables, for which it requested user assistance and correctly recognised all of the required reduction constructs. After additional scoping information was specified the generated code was verified to be correct and its performance is analysed within Section 3.3. The data-parallel nature of the CloverLeaf kernels does make it significantly easier for Reveal to generate the required scoping information also the tool is not yet able to automatically generate code which incorporates multiple loop-blocks within the same OpenMP `parallel` region.

3.3 Results Analysis

The aim of this research was to explore techniques for improving the time-to-solution achievable using the OpenMP-based version of CloverLeaf, therefore the results presented here are expressed in terms of execution wall-time. This analysis was conducted in two parts, firstly the utility of certain candidate optimisations within individual application kernels was assessed. This utilised the *kernel driver* functionality contained within the CloverLeaf software suite; Section 3.3.1 presents the results of this analysis. Secondly, the effectiveness of the successful optimisations on the full application codebase was then examined, together with several additional candidate optimisations; Section 3.3.2 contains the results of this analysis.

To examine the effectiveness of these optimisations at improving the performance of the codebase, when OpenMP parallelisation constructs are utilised across NUMA-domains, the dual-socket nodes of the Archer platform (which are based on Intel Xeon processors) were utilised. Additionally, in order to assess their utility on a current state-of-the-art high core count processor device, a series of experiments were also conducted using the Intel Xeon Phi co-processor within the Tuck system. This enabled the determination of whether certain optimisation techniques will be required within future applications, as

the construction of HPC platforms progresses towards processing devices which integrate larger numbers of CPU cores, which is a current trend within the HPC/Scientific computing field. Section A.1 contains more detailed information on the architectures of both of these experimental platforms.

The $3{,}840^2$ cell problem, from the standard CloverLeaf benchmarking suite, was examined in these experiments. Additionally to reduce the effects of system noise, unless otherwise noted, the results presented here are averages from 3 separate executions of each experiment. Version 14.0 of the Intel compiler suite was utilised throughout this work and all of the experiments on the Xeon Phi co-processor were conducted with the platform in *"native"* mode. On the CPU-based nodes of Archer each experiment utilised 24 OpenMP threads and the `KMP_AFFINITY` environment variable was set to explicitly bind each thread to a specific processor core. On the Xeon Phi platform, however, each experiment was conducted using 120 OpenMP threads with two consecutive threads executed on successive processor cores (`KMP_AFFINITY=granularity=fine, balanced; KMP_PLACE_THREADS=60c,2t`). Previous experiments have shown this to be the most performant configuration for this architecture. The IEEE floating-point mathematics options were also enabled in all experiments on Archer, whilst on the Xeon Phi these options were disabled.

3.3.1 Individual Kernel Optimisation Analysis

The following sections and Figures 3.4 to 3.10 each examine the effect on performance of applying particular optimisation techniques to individual application kernels.

Cell-Advection Kernel Optimisations

The effect of applying a series of optimisations to the *Cell-Advection* kernel was explored in a number of experiments. This involved examining the utility of the *NoWait* construct (Section 3.2.7), the *explicit loop schedules* (Section 3.2.6) and the *point-to-point* synchronisation mechanisms (Section 3.2.8), as well as the *variable privatisation* techniques (Section 3.2.9). In these experiments the kernel was executed for 1,000 and 500 iterations on the CPU and Xeon Phi architectures respectively, Figure 3.4 presents the results of these experiments.

Although the results contain some similar trends on both architectures they also exhibit some important differences. Implementing the *variable privatisation* optimisation to eliminate four 2D temporary arrays and the associated global memory operations delivers significant performance advantages on both architectures. The results show that this improves performance by as much as 27.2% on the CPU architecture and by 7.8% on the Xeon Phi platform.

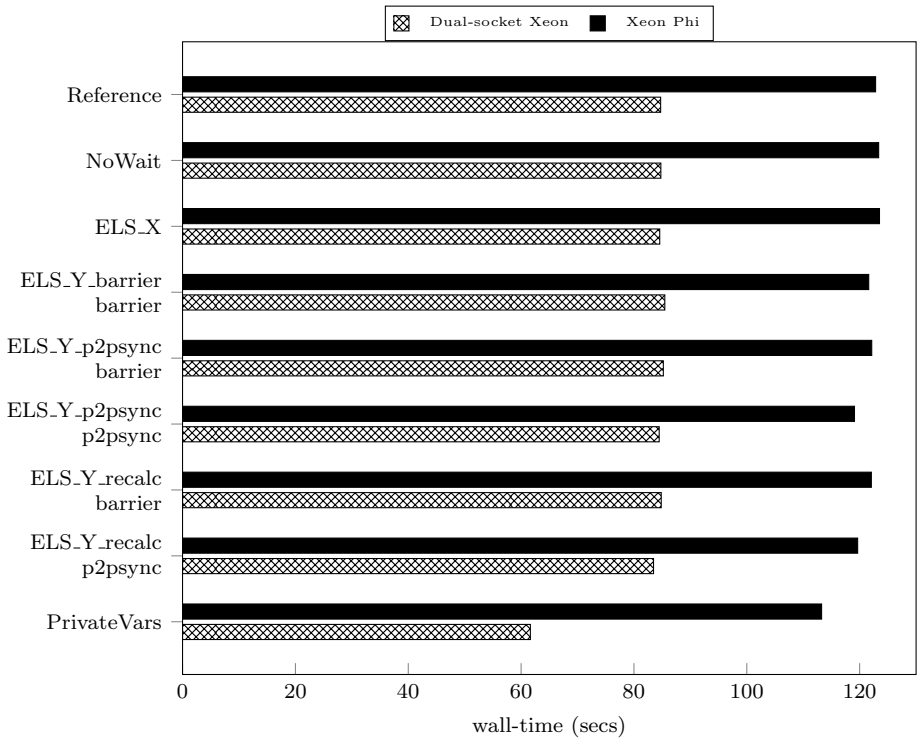


Figure 3.4: Optimisations to the *Cell-Advection* kernel

Applying the *NoWait* optimisation to the x-direction loop-blocks within the kernel and the *explicit loop schedules* optimisation to both the x- and y-direction loops (*ELS_X* and *ELS_Y_barrier_barrier*), however, does not have a significant effect on the overall performance of the kernel on either architecture.

Employing the *point-to-point* synchronisation optimisations (Section 3.2.8), affects performance differently on both processor architectures. The advection phase of the kernel in the y-direction contains three loop-blocks and this technique was utilised to reduce the synchronisation operations between successive pairs of these loop-blocks. The naming conventions used in Figure 3.4 indicates which technique was used between each particular pair of loop-blocks. For example, the *ELS_Y_recalc_p2psync* experiment employs the *recalculation* technique between the first two loop-blocks and the *point-to-point* synchronisation technique between the second pair of loop-blocks.

On the CPU architecture these candidate optimisation techniques do not deliver any significant performance benefits as the results show that execution time is virtually identical, allowing for system noise, to that of the reference implementation. The results from the experiments on the Xeon Phi architecture, however, show that employing either the *p2psync* or *recalc* techniques between the first two loop-blocks and the *p2psync* technique between the second pair

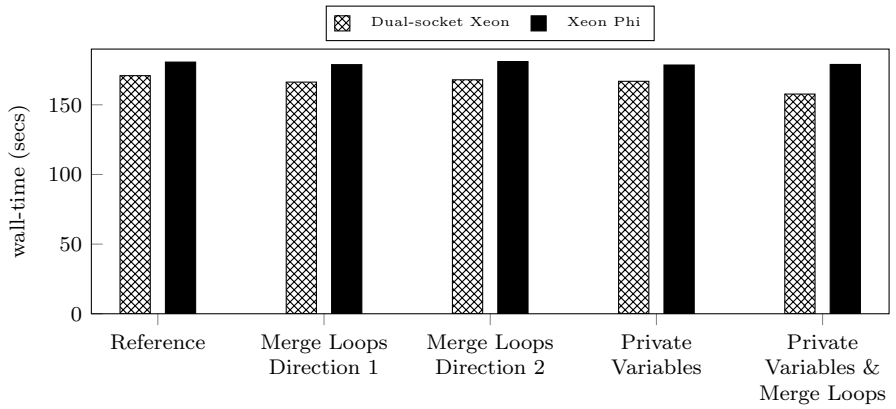


Figure 3.5: Optimisations to the *Momentum-Advection* kernel

of loop-blocks does delivery some performance advantages on this platform. In these experiments the reference implementation required 122.8s on average to complete the required iterations with a standard deviation (σ) of 0.49s. The optimised version which utilised the *p2psync* synchronisation technique between both loop-blocks (*ELS_Y_p2psync_p2psync*), however, improved performance by 3.7s (3.0%) on average, and a σ value of 0.3s was recorded. Additionally, the *ELS_Y_recalc_p2psync* version increased performance by 3.2s (2.6%) on average, with a σ of 0.13s. The performance of the versions which employed a global OpenMP barrier operation between at least one pair of loop-blocks was practically identical to that of the reference implementation.

Although these performance improvements are relatively small the fact that they only occur at the large thread counts utilised on the Xeon Phi co-processor indicate that these techniques may become increasingly important as the architecture of future processor devices forces application developers to significantly increase the levels of *"threading"* within their software designs. It should also be noted that the kernels of this application have already been heavily optimised and therefore achieving any performance improvements is both extremely challenging and worthwhile. Additionally, even small percentage improvements in performance can result in considerable financial cost savings when applications are executed at considerable scale.

Momentum-Advection Kernel Optimisations

As part of this research the effect on performance of merging particular loop-blocks within the *Momentum-Advection* kernel was examined in a series of experiments. Additionally, the use of the techniques described in Section 3.2.9, for reducing the use of global array data structures to store intermediary results within the kernel was also examined. Figure 3.5 presents the results from these experiments.

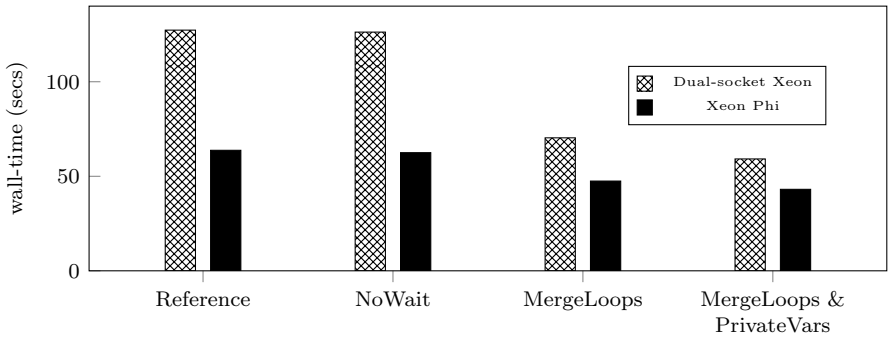


Figure 3.6: Optimisations to the *Accelerate* kernel

The results show that on the CPU architecture merging several of the loop-blocks within both the x- and y-directions of the kernel delivers a 2.7% and a 1.7% improvement in performance respectively. Additionally applying the *Private Variables* optimisation to eliminate one global 2D temporary array, and the associated global memory accesses, delivers a further 2.4% performance improvement. Combining these optimisations improves the overall performance of the kernel by 7.74% relative to the reference implementation. On the Xeon Phi architecture, however, these improvements are less successful. The implementation of the *Private Variables* technique and the optimisation to merge the x-direction loops only deliver a $\sim$1% improvement in performance. The merging of the loops in the y-direction, however, actually has a slightly detrimental effect on performance of -0.2%. Collectively these optimisations only improved the performance of the kernel on the Xeon Phi architecture by <1%.

Accelerate Kernel Optimisations

The effect of applying the *Loop-merger* and *Private Variables* optimisations to the reference implementation of the *Acceleration* kernel, as well as employing OpenMP `NoWait` directives to remove synchronisation operations, are shown in Figure 3.6. In these experiments the kernel was executed for 4,000 iterations on the CPU architecture and for 2,000 iterations on the Xeon Phi processor. The results indicate that on both the CPU and Xeon Phi processor architectures employing the `NoWait` directives delivers negligible performance benefits for this kernel. Manually merging the loop-blocks within the kernel, however, delivers significant performance improvements, with these reaching 1.8× and 1.34× on the CPU and Xeon Phi architectures respectively. Additionally, combining this technique together with the optimisation to convert global temporary arrays to `Thread-private` variables delivers further performance benefits of 1.2× and 1.1× respectively.

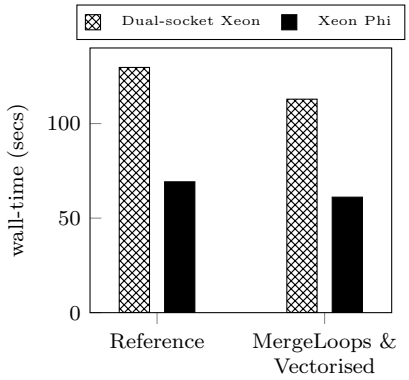


Figure 3.7: Optimisations to the *Calc-DT* kernel

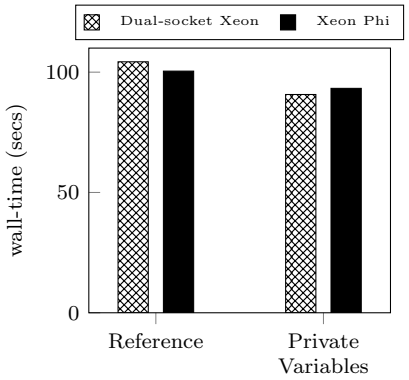


Figure 3.8: Optimisations to the *PdV* kernel

Calc-DT Kernel Optimisations

Applying the *Loop-merging* and *Vectorisation* optimisations described in Section 3.2.10 to the *Calc-DT* kernel improves performance on both processor architectures examined here. Figure 3.7 presents the results from these experiments and shows that these optimisations reduced the runtime of the kernel by 12.9% on the CPU architecture and by 11.7% on the Xeon Phi. In these experiments the kernel was executed for 10,000 and 2,000 iterations on the CPU and Xeon Phi processor architectures respectively.

PdV Kernel Optimisations

Similarly applying the optimisation described in Section 3.2.9 to convert the global arrays, utilised within the *PdV* kernel to store temporary values, to `Thread-private` temporary variables also delivers similar performance improvements. Figure 3.8 presents the results of this analysis and shows that this optimisation improves the performance of the *PdV* kernel by 13.0% on the CPU architecture and by 7.1% on the Xeon Phi. In these experiments the kernel was executed for 5,000 iterations on the CPU architecture, whilst on the Xeon Phi it was executed for 3,000 iterations.

Update-Halo Kernel Optimisations

The performance of the optimisations described in Section 3.2.12 to the *Update-Halo* kernel was also examined in a series of experiments, the results of which are presented in Figure 3.9. In these experiments the kernel was executed for 500,000 iterations on the CPU architecture and for 50,000 on the Xeon Phi processor. The results show that employing the OpenMP v4.0 process

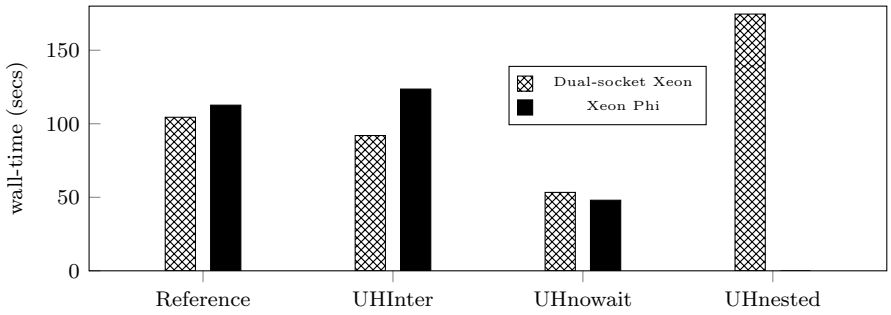


Figure 3.9: Optimisations to the *Update-Halo* kernel

placement constructs together with *Nested*-parallelism to restructure the computation across the NUMA-domains within the node, actually has a detrimental effect on performance. This caused a 1.67× slowdown, relative to the reference implementation, on the CPU architecture.

Manually reordering certain loops within the kernel and employing the `Collapse` OpenMP directive to improve the memory access patterns of the kernel delivers a performance improvement of 1.13× on the CPU architecture, however, on the Xeon Phi co-processor it causes a performance slowdown of 1.1×. An analysis of the vectorisation reports produced by the compiler indicates that this is likely due to the compiler generating more optimal vector code for the reference implementation, as on this architecture it is able to automatically permute the loops within this original implementation.

Restructuring the kernel into two distinct phases to reduce the number of global synchronisation operations from a worst case of 60 down to 2, however, delivers significant performance improvements on both platforms. On the CPU architecture the results show that this optimisation delivers a 1.96× improvement in kernel performance compared to the reference implementation, whilst on the Xeon Phi it achieves a 2.35× speedup.

Field-Summary Kernel Optimisations

Figure 3.10 presents the results of the experiments which examined the effect of applying the optimisations described in Section 3.2.10 to the *Field-Summary* kernel. In these experiments the *Field-Summary* kernel was executed for 10,000 and 2,000 iterations on the CPU and Xeon Phi architectures respectively. The results show that by enabling this kernel to be successfully vectorised delivered a 1.14× improvement in the performance of this kernel on the CPU architecture. On the Xeon Phi architecture, however, the performance improvement was significantly greater, reaching 4.77× relative to the performance of the reference version. This demonstrates the importance of fully vectorising loop-blocks on the Xeon Phi co-processor.

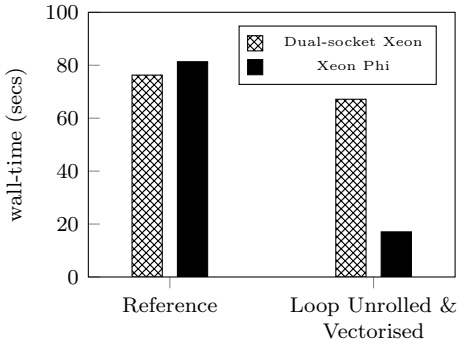


Figure 3.10: Optimisations to the *Field-Summary* kernel

3.3.2 Application Performance Analysis

Following the performance analysis conducted using the individual application kernels a series of experiments was subsequently undertaken using the full CloverLeaf codebase. These examined the effectiveness of a series of optimisations which targeted the entire codebase as well as the effect of incorporating the most successful individual kernel optimisations into the full application. In these experiments the application was configured to simulate the $3{,}840^2$ cell problem for 87 timesteps, which is a standard configuration from the CloverLeaf benchmarking suite. Figures 3.11 and 3.12 present the results from these experiments on both the dual-socket CPU and Xeon Phi processor architectures respectively. Each of the following sections analyses the utility of a specific optimisation technique.

First-touch Memory Placement

The results show that when the reference OpenMP implementation is executed across multiple CPU sockets it experiences a significant degradation in performance due to sub-optimal data placement across the different NUMA-domains. Applying the *first-touch* memory placement optimisation (Section 3.2.1) improves performance, relative to the reference implementation, by 14.8% on the dual-CPU architecture. On the Xeon Phi co-processor, however, this optimisation does not deliver any performance benefits and the runtime of this version is practically identical to that of the reference implementation. All subsequent versions examined in these experiments therefore include this *first-touch* memory placement optimisation.

Array-of-arrays Data Structure

As memory is allocated at the granularity of individual memory pages it is possible for the locations directly managed by a particular thread to be located

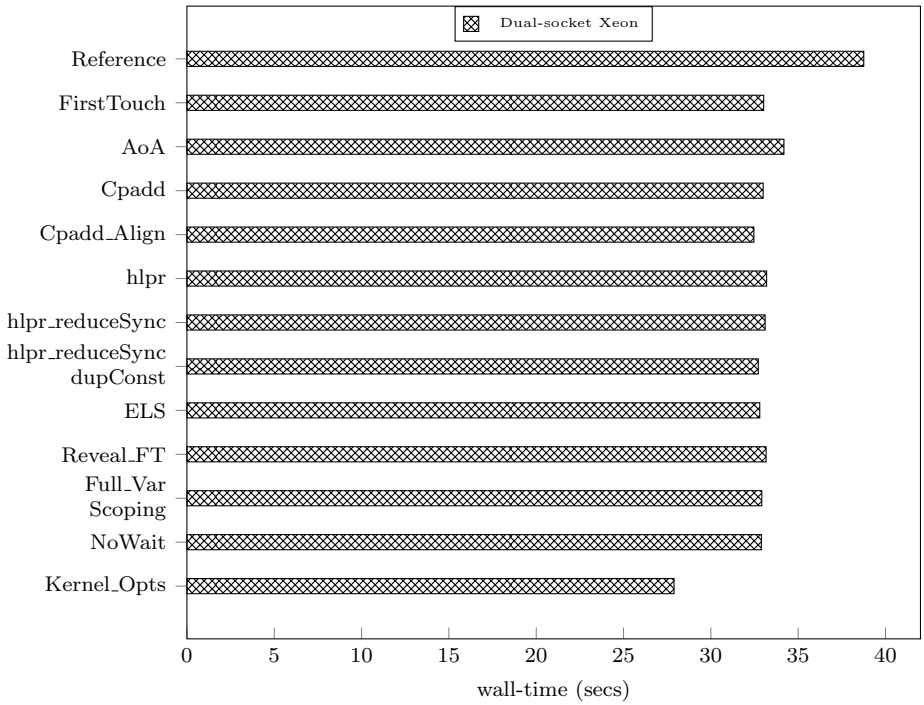


Figure 3.11: Application optimisations on the dual-socket CPU architecture

within the remote NUMA-domain of the node. This is due to these locations being assigned to a memory page which is accessed first by a thread located on the other CPU socket. An experiment was therefore conducted using a code variant which incorporated the *Array-of-arrays* modification (Section 3.2.2) to the codebase to potentially alleviate this problem. The results, however, show that constructing the codebase to utilise this data structure actually leads to a reduction in performance of 3.5% on the CPU architecture. Any performance benefits resulting from the more optimal data placement were negated by the reductions in performance from accessing the array data through this modified structure e.g. due to the additional levels of indirection involved. On the Xeon Phi architecture the performance of this implementation was substantially worse, and caused a slowdown in performance of 3.3× relative to the reference implementation.

Cache Line Padding & Memory Access Alignment

The results also show that introducing cache line *"padding"* into the application, in order to ensure that threads to do not share the same cache lines and experience "false sharing", had no significant effect on overall performance on the CPU architecture. On the Xeon Phi co-processor, however, this modification actually

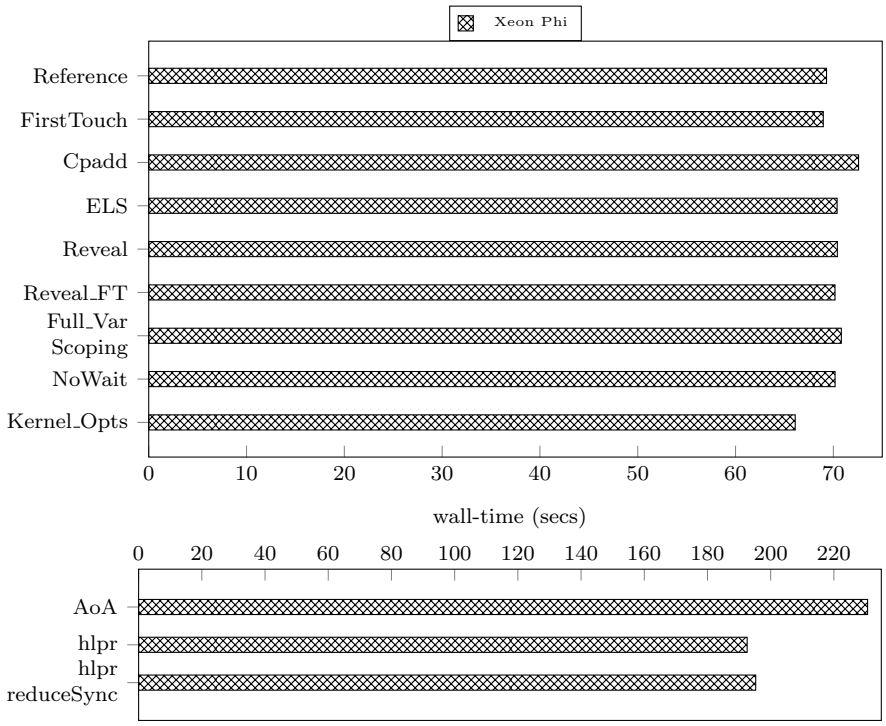


Figure 3.12: Application optimisations on the Xeon Phi co-processor

slightly reduced performance by 4.7%. Combining this optimisation with the Intel proprietary directives to align data placement and memory accesses did, however, deliver some small but measurable performance improvements of 1.7% on the CPU architecture.

High-level OpenMP Parallel Regions

The results from the experiments with the versions of the codebase which employed a *High-level OpenMP Parallel Region* (Section 3.2.4) show that on average this optimisation technique was not able to deliver any performance advantages for this codebase when it is used to simulate this particular problem size. On the CPU architecture the performance of the version containing the initial *High-level Parallel Region* optimisation was slower than the *First-touch* implementation by 0.17 seconds. Whilst the average runtime of the version with reduced synchronisation was fractionally faster, it was still marginally slower than the original reference version.

Interestingly, applying the technique of duplicating the 1D data arrays—which remain constant throughout the execution of the application—within each NUMA-domain (Section 3.2.5), actually fractionally improved average performance by 0.4 seconds (1.2%). This indicates that small improvements in code

performance can be obtained for applications by minimising remote NUMA-domain memory accesses.

Surprisingly, the performance of the *High-level Parallel Region*-based implementations on the Xeon Phi co-processors was significantly worse than the reference version, delivering as much as a 2.8× degradation in overall performance.

Explicit Loop Schedules

The effect on performance of employing *Explicit Loop Schedules* (Section 3.2.6) throughout the entire application, instead of relying on the OpenMP runtime system to partition loop-blocks, was also examined in these experiments. The results indicate that in these experiments this optimisation only delivered a fractional overall improvement in performance of 0.22 seconds on the CPU architecture, whilst on the Xeon Phi co-processor a marginal reduction in performance of 1.07 seconds was recorded.

Automatic Application Hybridisation

To assess the effectiveness of the Reveal tool at automatically incorporating the OpenMP parallelisation constructs into the application a series of additional experiments was conducted. Initially the implementation produced by Reveal performed poorly on the CPU architecture delivering a ∼1.98× reduction in performance relative to the *First-touch* version (this result is omitted from Figure 3.11 for brevity). A subsequent performance analysis of the codebase, however, identified that this was due to similar data placement problems to those experienced with the original reference implementation. Consequently a further version was developed which incorporated the *First-touch* data placement optimisations discussed in Section 3.2.1. This significantly improved the performance of this implementation on the CPU architecture to be within 0.14 seconds of the manually developed version.

On the Xeon Phi co-processor the initial version did not experience the same NUMA-related memory access problems and the performance of both versions was practically identical (within 0.24 seconds of each other). Similarly, in these experiments the performance of the versions produced by Reveal was fractionally slower than the reference implementation, although their performance was within 2% of this implementation. These performance discrepancies are likely attributable to the fact that Reveal generates code with one OpenMP `parallel` region per loop-block, whereas the reference implementation minimises the number of these regions by incorporating multiple loop-blocks within each kernel into these constructs.

Full OpenMP Scoping Information

A further version of the codebase was developed in which full OpenMP scoping information was specified for each OpenMP directive. The `default(none)` directive was added to each OpenMP construct and scoping information (e.g. the `shared` or `private` qualifiers) defined for each additional variable or data structure accessed within a particular `parallel` region. On the CPU architecture this implementation delivered almost identical performance to the reference implementation, with only a 0.1 second improvement in performance being recorded on average relative to the reference implementation. On the Xeon Phi co-processor, however, this optimisation actually resulted in a slight reduction in performance of 1.5 seconds relative to the reference implementation.

Synchronisation Elimination

The effect of applying the optimisation technique described in Section 3.2.7 to remove, where possible, the global OpenMP barrier operations throughout the codebase was also examined in these experiments. The results show that on the CPU architecture this candidate optimisation only fractionally improved performance, reducing the runtime of the application by only 0.13 seconds relative to the initial *First-touch* version. On the Xeon Phi architecture, however, a fractional performance degradation of 0.87 seconds was recorded.

Individual Kernel Optimisations

The optimisations to the individual application kernels developed as part of this research and analysed in Section 3.3.1 were subsequently incorporated into the full application codebase to produce a further optimised version. With the exception of the *point-to-point* synchronisation optimisations to the *Cell-Advection* kernel, all of the optimisations examined in Section 3.3.1 were incorporated into this version (labelled *Kernel_Opts* within Figures 3.11 and 3.12). The experimental results show that on the CPU architecture the use of these optimisations improved the overall performance of the full application codebase by 15.5% relative to the initial *First-touch* implementation. These optimisations also improved the performance of the application on the Xeon Phi co-processor by 4.6% compared to the reference implementation. As these optimisations deliver a consistent performance improvement on both processor architectures these changes will be utilised within future versions of the codebase.

3.4 Summary

This chapter has documented the findings from the research which was undertaken to improve the intra-node performance of the OpenMP-based versions of CloverLeaf. It presents a detailed description of the current OpenMP-based implementation of the mini-application together with each potential modification which has been examined. This includes optimisations focused on individual kernels as well as those which apply to the entire codebase. The performance of each of these alternative approaches is examined on a range of current state-of-the-art processor technologies, specifically a dual-socket Intel Xeon based platform and an Intel Xeon Phi co-processor.

The experimental results show that the performance of the various alternative approaches can vary significantly on the two architectures examined in this work. On the CPU-based architecture, due to its multiple NUMA-domains, optimising the placement of data within the application using *"first-touch"* initialisation techniques delivered a 14.8% improvement in performance. This is a significant performance improvement for an already highly optimised codebase and enabled the performance achievable when OpenMP threading constructs are utilised across multiple NUMA-domains, to match the performance recorded with the MPI-only model. Manually merging loop-blocks and improving the levels of vectorisation delivered significant additional performance improvements for several key application kernels. Reducing global memory operations and overall memory consumption by converting temporary 2D data-arrays to *"thread-private"* variables also proved to be a key approach for improving application performance. When these optimisations were subsequently applied to the full application codebase they resulted in an overall performance improvement of 15.5% on the CPU architecture and 4.6% on the Xeon Phi co-processor.

Employing *point-to-point* thread synchronisation and *data re-calculation* techniques to reduce and avoid synchronisation operations within key application kernels delivered some small performance benefits (∼3%) at the high thread counts examined on the Xeon Phi co-processor. On the CPU-based architecture, however, the performance of the versions which incorporated these techniques was almost identical to that of the reference implementation. This indicates that the use of these techniques may potentially become increasingly required in order to achieve optimal performance for applications which utilise large numbers of threads on future processor architectures. Existing research has demonstrated that the overheads associated with globally synchronising all application threads increases with the number of threads involved in the particular synchronisation operations [146].

Additionally, this research demonstrated that utilising an *array-of-arrays* data structure in order to optimise memory-layout across the different NUMA-regions is not able to improve overall application performance on the Intel Xeon E5-2620 CPU architecture. Furthermore in these experiments this modification resulted in a substantial performance degradation of 3.3× on the Xeon Phi co-processor.

Surprisingly, converting the application to utilise an OpenMP SPMD construction using the *High-level Parallel Region* optimisation, in order to reduce thread synchronisation and *fork/join* overheads, also resulted in a significant reduction (2.8×) in performance on the Xeon Phi architecture. On the CPU architecture, whilst the performance of this version was able to match that of the reference implementation, it required the use of additional techniques, such as the duplication of constant data within both NUMA regions, in order to deliver any performance benefits.

The results also indicate that the use of the *Explicit Loop Schedules* optimisation did not deliver any significant performance benefits on the CPU architecture and resulted in a fractional slowdown in performance on the Xeon Phi. Similarly introducing *"padding"* into the data-arrays to reduce *false sharing* resulted in no significant performance benefits on the CPU-based platform and a small slowdown in performance on the Xeon Phi co-processor. The incorporation of memory alignment constructs also only appears to fractionally improve the performance of this codebase on the CPU architecture.

This research also demonstrated that with minimal manual intervention the Cray Reveal tool is capable of automatically generating parallel code based on OpenMP directives, the performance of which is able to closely match that of manually developed code. It should be noted that the data-parallel nature of the CloverLeaf kernels does make it significantly easier for Reveal to generate the required code and that the tool is not yet able to automatically incorporate multiple loop-blocks within the same OpenMP `parallel` region. Nevertheless, the use of this and similar tools, should help to improve the overall productivity of the developers of parallel applications which incorporate OpenMP parallelisation constructs.

Through this research it was possible to improve the overall performance of the application, relative to the initial reference implementation, by 28.0% and 4.6% on the CPU and Xeon Phi processor architectures, respectively. It should also be noted that this codebase has already been highly optimised by both academic and industrial partners and therefore achieving any further optimisations is both challenging and worthwhile. Even small percentage optimisations are important in contributing to achieving one of the goals of this research, i.e. developing a fully optimal version of the codebase, and can result in considerable

financial cost savings when applications are executed at extreme scale.

Although the techniques examined in this work were developed exclusively within the CloverLeaf mini-app, the optimisations are also generally applicable to a significantly wider range of scientific applications which exhibit similar performance characteristics. In particular these include applications which utilise regular collections of loop-blocks to process data which is stored predominantly in a structured manner within n-dimensional arrays.

CHAPTER 4
Achieving Efficient Application Execution at Extreme Scale

This chapter documents the research which was undertaken, at high processor counts, to develop and evaluate techniques for improving the performance and scalability of the CloverLeaf mini-application, and therefore to also improve the performance of the explicit hydrodynamics applications for which CloverLeaf functions as a proxy application. The work focuses primarily on the MPI-based version of the codebase and examines several candidate optimisations including hybridising the code using OpenMP. The chapter initially discusses some related and motivating work within this arena (Section 4.1). Section 4.2 documents the current implementation of the mini-app in the MPI-only programming model, together with a description of each candidate optimisation examined for this particular variant of the codebase. The implementation of the hybrid (MPI+OpenMP) version of the mini-app, is then presented in Section 4.3, together with the candidate optimisations techniques which were examined for this particular implementation of the codebase. The performance of these potential optimisations on a range of architectures is subsequently analysed in Section 4.4, together with an assessment of their effect on overall energy consumption. Finally, Section 4.5 concludes the chapter.

4.1 Related Work

Minimising communication operations within applications has been recognised as a key approach for improving the scalability and performance of scientific applications [123]. Yun *et al.* examined various approaches and optimisations for improving the performance of large-scale jobs on Cray platforms [78]. The aggregation of small messages, when possible, has previously been identified as the ideal communication strategy for scientific applications [22]. In [20], however, Barrett *et al.* present work which examines alternatives to the message aggregation strategies generally employed within BSP programming model based applications. Their work, which examines an application similar to CloverLeaf, is motivated by current development trends in HPC interconnect technologies for existing, and future exascale, system designs. They show that their alternative approach, which communicates data as soon as possible after it is modified, delivers a considerable improvement in application performance at scale on several current system architectures, compared to the original BSP-based approach.

It is also recognised that increasing the levels of asynchronicity within ap-

plications, through the overlapping of computation and communication operations, can deliver performance advantages. Several techniques for achieving the overlap of these operations are examined in [175], together with a quantitative analysis of their benefits for a range of applications. Jiang *et al.* show that employing an RDMA based approach can improve the overlap of communication and computation operations [105]. Similarly Bell discusses the benefits of overlapping communication operations with computation and further communication operations through message pipelining, on a range of network architectures [22]. Overlapping communications with computation at a finer granularity has also been shown to deliver performance benefits by interspersing more of the communication events with computation, whilst also decreasing message size and increasing the injection rate [23]. The effectiveness of both the one- and two-sided communication models at overlapping communication and computation operations has also been analysed with the former, when expressed using UPC, performing favourably compared to the latter when implemented with MPI [23, 147]. Additionally, Potluri *et al.* examined using MPI one- and two-sided operations to overlap communication and computation and were able to achieve a speedup of 10-12% in application performance [165].

New communication constructs have also been developed within version 3.0 of the MPI standard to potentially improve the performance and scalability of applications [144]. Hoefler *et al.* were able to achieve a significant performance improvement of up to 40% over existing approaches using their own implementations of several MPI 3.0 neighborhood collective communication operations [85]. Similarly, Gerstenberger *et al.* document their work developing an MPI 3.0 compliant implementation (FOMPI), which utilises scalable buffer-less protocols, and achieves equivalent or superior performance to UPC and CAF [71].

A considerable body of work also exists which has examined the advantages and disadvantages of the hybrid (MPI+OpenMP) programming model compared to other multi-level paradigms or the MPI-only model [110, 73]. These studies have generally focused on different scientific domains; classes of applications; and different hardware platforms, to those examined is this research. Results have also varied significantly, with some authors achieving significant speed-ups by employing hybrid constructs [203, 179, 106], whilst others experience performance degradations [82, 38, 130] .

In particular, Környei presents details on the hybridisation of a combustion chamber simulation which employs similar methods to CloverLeaf. The application domain and the scales of the experiments are, however, significantly different to those examined here. Drosinos *et al.* also present a comparison of several hybrid parallelisation models (both coarse- and fine-grained) against the MPI-only approach [59]. Again, their work focuses on a different class

of application, at significantly lower scales and on a different experimental platform to this research. Nakajima compares the hybrid and MPI-only programming models for preconditioned iterative solver applications within the linear elasticity problem space [145]. In this research the application domain, the scales of the experiments (<512 PEs) and the choice of platform (T2K HPC architecture) are again significantly different to those examined here. Although the application examined by Lavallée *et al.* has similarities to CloverLeaf and they compare several hybrid approaches against an MPI-only based approach, their work focuses on a significantly different hardware platform [120]. Additionally, Adhianto *et al.* discuss their work on performance modelling hybrid MPI+OpenMP applications and demonstrate its potential for facilitating the optimisation of scientific applications [2].

The energy consumption of supercomputer platforms is increasingly becoming a major concern to large HPC sites [153, 11, 129]. Consequently there is currently significant interest in the fine grained monitoring and analysis of the power consumption of scientific applications. Both Cray and IBM have recently incorporated such facilities into their latest supercomputer solutions [135, 201]. Hart and Wallace document their experiences utilising these technologies to successfully analyse the power consumption of applications on the Cray XC30 and IBM Blue Gene/Q respectively [76, 202]. Additionally, Li *et al.* examine employing a hybrid programming approach to achieve more power-efficient implementations of particular benchmarks [123].

4.2 MPI-only Based Versions

This section documents the implementation of the reference MPI-only based version of CloverLeaf and the optimisations applied to it as part of this research (Section 4.2.1). Details on how the codebase was instrumented to enable its power consumption to be analysed are also presented (Section 4.2.2).

The MPI-based implementations of CloverLeaf employ a block-structured decomposition (see Section 1.6.1) in which each MPI task is responsible for one rectangular region of the computational mesh. The *halo-exchange* routine performs the required *halo* cell communications, during which multiple fields (2D-arrays each representing a particular physical property e.g. density) can be exchanged with varying depths of cells (1, 2...etc), depending on the requirements of the algorithm at that stage of its computation. Processes perform these *halo* exchanges using the `MPI_ISend` and `MPI_IRecv` communication operations with their logically immediate neighbours, first in the horizontal dimension and then in the vertical dimension. Communications are therefore two-sided, with `MPI_WaitAll` operations being employed to provide local synchronisation

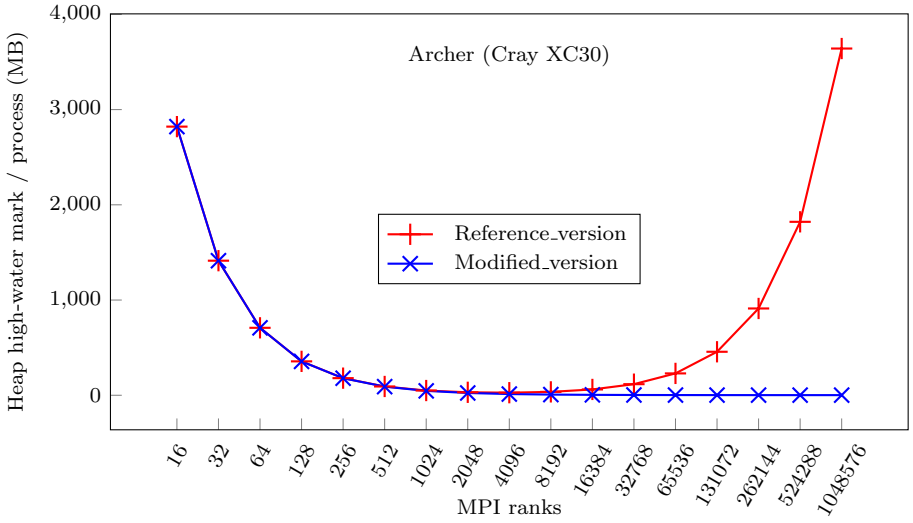


Figure 4.1: CloverLeaf heap memory consumption per process

between the data exchange phases. Consequently no explicit global synchronisation operations (`MPI_Barrier` functions) are present in the hydrodynamics timestep. In the reference version of CloverLeaf the *halo-exchange* routine employs an approach in which the *halo*-cell data from individual fields is exchanged separately.

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

4.2.1 Optimisations Examined

The techniques examined as part of this research to improve the scalability and performance of the MPI-only version of CloverLeaf at high node counts are presented here. These techniques were initially employed in isolation to implement alternative versions of the codebase; several were then subsequently combined to produce further versions of the application. Section 4.4 analyses the effect of each of these potential optimisation techniques on the performance of the mini-app.

Distributed Meta-data

A memory consumption analysis of the mini-app was conducted following initial strong-scaling experiments with the codebase, using the $15{,}360^2$ cell problem from the standard CloverLeaf benchmarking suite, on the Archer and Mira platforms (see Section A.1). The CrayPat [46] performance analysis tool available on the Archer platform was employed to conduct this analysis. This was utilised to examine the *"high-water"* mark of the total memory consumed from the heap memory region by each MPI rank of the mini-application as process counts were increased. Figure 4.1 presents the results from this analysis.

Given that this is a strong-scaling experimental configuration the memory consumption per MPI process should decrease as the scale of the experiments is increased. The results for the reference implementation in Figure 4.1, do initially show this trend. Beyond approximately 4,096 MPI ranks, however, the memory consumption per process starts to grow significantly.

A subsequent investigation to identify the source of this additional memory consumption determined that this was due to the scaling characteristics of the data structures within the codebase. These were originally designed to enable the computational mesh to be over-decomposed, with multiple mesh regions or chunks being assigned to each MPI rank. In the original reference implementation, however, a one-to-one mapping between mesh regions and ranks was specified. This was implemented using a strategy which required each rank to maintain meta-data information on each mesh region within the overall decomposition, regardless of whether a particular rank was required to actually manage these regions or not. Each MPI process was therefore required to create an array—called *chunks* within the source code—of $\mathcal{O}$(the number of mesh regions within the overall decomposition).

This array stores the meta-data relating to the individual mesh regions, with each location able to store an additional derived type data structure (called *field_type*), if the process is required to actually manage that particular region of the computational mesh. The additional *field_type* derived type contains the actual data fields (2D-arrays) which model the particular physical quantities contained within the computational mesh. The size of the *field_type* components decrease as the scale of the experiments is increased; however, the size of the "top-level" *chunks* array increases linearly with the number of MPI ranks involved in the particular simulation. At relatively small scales, $\leq$4,096 ranks, this does not have a significant affect on the overall memory consumption of each process. Beyond this point, however, the additional meta-data storage locations start to consume significant amounts of memory. Under this configuration if 1,048,576 MPI ranks were employed, the memory *"high-water"* mark of the

heap region on each rank would reach ∼3.5GB and the overall simulation would require ∼3.5PB of main memory. It should also be noted that the ∼3.5GB total is a per-process memory consumption figure and therefore overall node-level memory consumption would be proportion to the number of MPI ranks employed per-node, i.e. significantly higher.

To improve this implementation an additional version of the codebase, which employed a distributed meta-data strategy, was implemented as part of this research. This required each MPI rank to only maintain meta-data for the number of computational mesh regions which it was actually required to directly manage and simulate. Consequently, the size of the *chunks* array on each MPI rank became $\mathcal{O}$(the number of mesh regions which each process is required to manage). The effect of this optimisation on the total memory *"high-water"* mark of the heap region within each MPI rank can be seen in Figure 4.1 (*modified version*). Section 4.4.1 also contains an assessment of the effect of this optimisation on the actual performance of the mini-app. All subsequent results presented within Section 4.4 are, however, from versions of the mini-app which incorporate this optimisation technique.

Communicating Multiple Fields Simultaneously

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. Consequently, multiple synchronisations occur between communicating processes (two per field exchange) when boundary cells from multiple fields need to be exchanged during one invocation of the routine. These additional synchronisations are unnecessary as the boundary exchanges for each field are independent operations within each dimension (horizontal and vertical). It is therefore possible to restructure the *halo-exchange* routine to perform the horizontal *halo* 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 operation/message per field approach. Versions which employed this optimisation are denoted by the abbreviation *MF* (Multiple Fields) within their descriptions in Section 4.4.

Pre-posting MPI Receives

Previous studies have shown performance benefits from *pre-posting* MPI receive calls before the corresponding send calls [209]. In the reference *halo-exchange* implementation routine all MPI send calls are executed before their correspond-

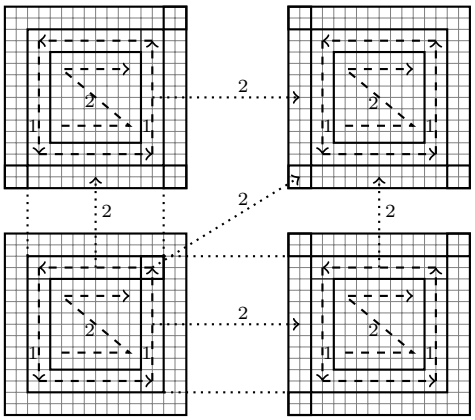


Figure 4.2: Cell calculation order for communication-computation overlap

ing receive calls. Additional versions of CloverLeaf were therefore created which *pre-post* their MPI receive calls as early as practicable within the codebase. 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 particular call to the *halo-exchange* routine. This ensures that a sufficient amount of computation occurs between each *pre-posted* MPI receive operation and the execution of its corresponding send operation.

Diagonal Communications

The reference implementation of the *halo-exchange* routine also requires the horizontal communication phase to be completed before the vertical communications in order to achieve an implicit communication between logically diagonal neighbouring processes. The synchronisation requirement between the phases can, however, be removed by employing an explicit communication between logically diagonal processes. This approach requires additional communication buffers and MPI communication operations to be initiated, but enables all communications in all directions to occur simultaneously, with only one synchronisation required at the end of the *halo-exchange* routine. Versions which employed this communication strategy are denoted by the letters *DC* within their descriptions in Section 4.4.

Overlapping Communications and Computation

The reference implementation is based on the BSP model (Section 2.2.6) with separate computation and communication phases. Additional versions which attempt to overlap the communication and computation phases were also developed as part of this research. This was achieved by moving the communication

operations at particular phases of the algorithm inside the computational kernels which immediately precede them. The loop iterations within these kernels were also reordered in order to compute the outer *halo* cells, which need to be communicated, before the inner region of cells (Figure 4.2). In these modified implementations once the outer *halo*-cells have been computed non-blocking communication primitives are then employed to initiate the data transfers. This approach also relies on the implementation of the diagonal communication operations (Section 4.2.1). Each computational kernel then completes the remaining calculations, with these computations being potentially fully overlapped with the preceding communication operations. Versions which employ this technique contain the word *Overlap* within their descriptions in Section 4.4. Some MPI implementations also provide dedicated *"progress"* threads to potentially aid this process, versions which utilised these additional facilities are denoted by the acronym *PT* within Section 4.4.

MPI v3.0 Construct Evaluation

The MPI v3.0 standard defines a set of new collective operations which initiate communications between immediate neighbouring processes within a virtual process topology. Such process topologies, created via the `MPI_Cart_Create` or `MPI_Graph_Create` routines, have existed for sometime within the standard. The new *neighbourhood collectives*, however, enable the communications between immediate neighbours (one hop within the virtual topology) to be completed with only one MPI operation and purport to enable the MPI compiler and runtime system to be able to implement additional optimisations.

The `MPI_Neighbor_AllToAllV` collective operator was selected to implement this optimisation as it enables communications of differing sizes to occur directly between all the neighbouring processes within the topology. This operation replaces all of the MPI *point-to-point* and *synchronisation* operations within the *halo-exchange* routine. As the *neighbourhood collectives* require all communications to occur simultaneously this necessitated the use of direct communications between logically diagonal neighbouring processes (Section 4.2.1). It was therefore also necessary to utilise the *graph* virtual process topology to create communication links between each process and all of its immediate neighbours, up to a maximum of eight edges per process, as the *cartesian* virtual topology does not support this level of connectivity. *MPI_Info* objects were also employed to provide additional information on the required memory access and communication patterns. Versions which employed these *neighbourhood collective* operations are denoted by the word *nColl* within their descriptions in Section 4.4.

The non-blocking reduction operation (`MPI_IReduce`) was also utilised within subsequent versions to implement the reduction operations required to produce the intermediary results printouts. Use of this non-blocking collective adds more asynchronicity into the application and enables it to potentially continue to make forward progress whilst these reduction operations are being completed, with this additional work being overlapped with the communication operations. The computational operations which make use of the intermediate result data values were therefore relocated, such that they occur after subsequent phases of the application, whilst ensuring program correctness. A `MPI_WaitAll` operation was also employed immediately prior to their execution to ensure that the non-blocking reductions complete successfully before the dependent computational operations are executed. It was not possible to utilise this approach for the calculation of the timestep value as this is required immediately after the existing reduction operation. Versions which employed the non-blocking `MPI_IReduce` operation are denoted using the acronym *NBR* within their descriptions in Section 4.4.

Message Aggregation

The reference implementation of the *halo-exchange* routine utilises shared communication buffers, one for each communication direction. These MPI buffers can be reused for multiple fields as the *halo* cells of only one field are exchanged at once. Buffer sharing is not possible when fields are exchanged simultaneously and each field therefore requires its own communication buffers, one for each direction. Message aggregation reduces the number of communication buffers, as well as the number of MPI *send* and *receive* calls required to one per direction, by combining messages into fewer but larger buffers. This technique was applied to produce additional versions of CloverLeaf, which send multiple messages simultaneously in each direction, by first aggregating all of the smaller messages into larger communication buffers. Versions which employed this technique are denoted by the letters *MA* (Message Aggregation) within their descriptions in Section 4.4.

Eager Transmission of Data

Additional versions were also subsequently developed to determine whether the implementation of a communication strategy, which attempts to transmit data to neighbouring processes as soon as it is updated, can deliver performance advantages for the applications which CloverLeaf represents. A similar strategy was employed by Barrett *et al.* in [20] and achieved significant performance advantages by enabling applications to transition away from the BSP model

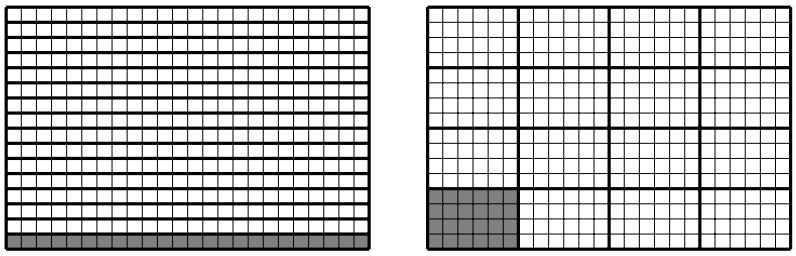


Figure 4.3: MPI rank reordering strategy

and to utilise advanced features within modern interconnect designs. Versions which employed this advanced communication strategy are denoted using the abbreviation *EDT* (Eager Data Transmission) within their descriptions in Section 4.4.

The implementation of this technique requires communication and computation operations to be overlapped in versions which previously did not implement this approach. Communication operations were therefore again relocated to the computational kernel which immediately preceded their current location. The kernels were however restructured such that the calculations on certain fields were completed earlier than others, whilst still maintaining program semantics, enabling their data items to be transmitted sooner. The required asynchronous communication operations were therefore interspersed throughout these kernels to facilitate the earlier data transmissions.

A slightly modified strategy was adopted in order to apply this candidate optimisation to existing versions which already attempt to overlap computation and communication. As part of this approach kernels were restructured such that only the calculations of the *halo*-cells of the particular data fields which actually need to be communicated, were completed before fields which did not need to be transmitted. Additional asynchronous communication operations were also inserted immediately after the point in the program code where each set of *halo*-cells becomes ready for transmission. This generally enabled the communication operations to occur earlier in the computational kernel and provided more opportunities for overlapping these operations.

MPI Rank Reordering

The reference CloverLeaf implementation assigns chunks of the two dimensional computational mesh to MPI ranks sequentially, by traversing the decomposition first in the x-dimension starting in the lower left corner. Once one row of

chunks has been completely assigned the allocation process restarts from the chunk on the left-hand side of the decomposition which is one row higher than the previous row in the y-dimension, and again proceeds sequentially along the x-dimension. The allocation process continues until all chunks of the mesh have been completely assigned.

This potentially results in a chunk-to-node mapping which does not reflect the two dimensional nature of the overall problem and therefore is unable to take full advantage of the physical locality inherent in it. Figure 4.3a depicts a typical default mapping of a 384 rank job on current system architectures with 24 processor cores per node, although this is system-dependent. In this arrangement communications in the y-dimension are all inter-node and each process only has a maximum of 2 neighbouring processes located within its local node. A disproportionate number of chunks, which are not physically close within the computational mesh, are therefore co-located within cluster nodes. This potentially results in a situation where local memory communication resources, which are usually substantially faster than inter-node communication resources, are not effectively utilised.

It is possible to use MPI rank reordering facilities to change the placement of MPI ranks within a given node allocation. Figure 4.3b depicts an alternative mapping strategy for the same 384 rank job. This better reflects the two dimensional communication pattern inherent within the application, by attempting to increase intra-node communications whilst also reducing inter-node communications. Versions which employ this *"blocked"* rank reordering strategy are referred to using the acronym *RR* within their descriptions in Section 4.4.

MPI Reduction Consolidation

To periodically produce intermediate results the reference implementation employs a series of five separate, but consecutive, global MPI reduction operations. These calculate the sum of five individual data fields (arrays) within the application. To improve the efficiency of this operation, these reduction operations were consolidated into one operation which operates on a vector of five values. Versions which employed this candidate optimisation technique contain the abbreviation *RedCon* in their descriptions within Section 4.4.

4.2.2 Power Consumption Instrumentation

Power monitoring facilities are not available on all available system architectures, however, both the Cray XC30 and IBM BG/Q platforms provide this functionality [135, 201]. On the BG/Q, IBM provides a dedicated API which applications can use to query the underlying power monitoring infrastructure.

Cray, however, make this information available via dedicated files within the `/sys/cray` file-system. These are continuously refreshed to reflect the accumulated energy consumption of the application on the particular node and can be read directly by an application.

As part of this research the MPI-only versions of CloverLeaf were instrumented to enable the power/energy consumption of the application to be measured at specific points during its execution, on both the XC30 and BG/Q architectures. The main hydrodynamics iteration loop, which is also timed to produce the runtime of the application, was instrumented at its start and end positions to enable the energy consumption of only the main computational sections of interest to be measured. The results from this energy consumption analysis can be found in Figure 4.16.

4.3 Hybrid (MPI+OpenMP) Based Versions

This section documents the reference implementation of the MPI+OpenMP version of CloverLeaf as well as the potential optimisation techniques (Section 4.3.1), which have been examined as part of this research. This version is an evolution of the MPI-only codebase in which OpenMP is utilised to provide the majority of intra-node parallelism, whilst MPI still provides the inter-node and potentially some intra-node communications. The ratio of OpenMP threads to MPI processes can be varied to suit different platform architectures and problem classes. This approach reduces the memory consumed per node by the *halo*-cells as these are only required for communication operations between *"top-level"* MPI processes. Additional data structure can also be shared across OpenMP threads rather than duplicated between MPI processes, further reducing memory consumption.

The reference version of this implementation employs OpenMP `parallel` regions within each of the 14 computational kernels. To minimise the *fork* and *join* overheads of the OpenMP programming model one `parallel` region is employed around all of the loop-blocks within a particular kernel. To enable the individual loop-blocks within the computational kernels to be parallelised over the available threads, additional OpenMP `do` constructs are utilised, generally around the outer-loops within the kernel. OpenMP `private` constructs are specified where necessary to create temporary variables that are unique to each thread, additionally `reduction` primitives are also utilised to implement intra-node reduction operations.

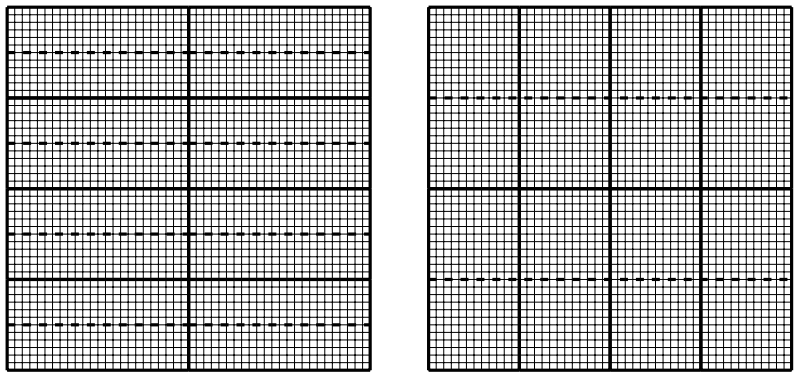


Figure 4.4: Vertical decomposition optimisation

4.3.1 Optimisations Examined

The techniques examined as part of this research to improve the single-node performance of the OpenMP version are presented in Sections 3.2.1 to 3.2.13. These techniques were initially used in isolation to implement alternative versions of the codebase, however, several were also subsequently combined to produce a more optimal version of the application. The motivations for conducting this work included ascertaining whether a hybrid implementation could be developed to deliver performance advantages over the MPI-only version and also to determine the optimal ratio of OpenMP threads to MPI processes for particular problem classes. A further key objective was to examine whether it is possible to improve the OpenMP implementation such that utilising these constructs across entire system nodes is a viable solution. The following sections also describe additional optimisation techniques which were examined as potential approaches for further improving the performance of the codebase. Section 4.4.2 analyses the effect of each of these candidate optimisation techniques on the performance of the mini-app.

Vertical Rectangular Decomposition

In order to reduce inter-process communication volumes the reference implementation attempts to decompose the overall problem such that mesh *"chunks"* which are as square as possible, are assigned to the individual MPI processes, whilst also distributing the computational load as equally as possible. Each process subsequently utilises OpenMP parallelism to further decompose its assigned mesh region, with each thread being assigned a contiguous number of rows (Figure 4.4a). For particular problem sizes and MPI process counts it is

not possible to assign perfectly square mesh *"chunks"*, necessitating the use of rectangular regions. These rectangular regions are, by default, assigned such that their longer side is orientated in the x-dimension of the mesh. The OpenMP parallelisation constructs are, however, applied to the individual mesh regions in the y-dimension, such that each thread accesses a contiguous block of memory. Due to the larger surface area between adjacent rows of the mesh within a particular node boundary, this arrangement requires greater levels of inter-thread communication (Figure 4.4a) and potentially causes additional communication traffic across the inter-socket interconnect, when OpenMP parallelisation is utilised across multiple processor sockets, which incurs additional overheads. To reduce the levels of inter-thread communication in these scenarios an additional version was developed which decomposes the mesh such that the rectangular *"chunks"* are orientated in the y-dimension (Figure 4.4b). Versions which incorporated this candidate optimisation are referred to using the word *Vdecomp* within their descriptions in Section 4.4.

MPI Construct Integration

The reference implementation employs the `mpi_thread_single` approach in which MPI communication constructs are only utilised within serial sections of the application, despite the actual (un)packing of communication buffers being parallelised using OpenMP `parallel` constructs. To evaluate the effectiveness of alternative approaches an additional version was created which utilises an OpenMP `parallel` region directly around the MPI functions within the codebase. A `sections` directive was utilised to enable each MPI function to be executed in parallel on a separate thread and the MPI runtime was also initialised using the `mpi_thread_multiple` option. The version which utilised this approach is denoted as *ThreadMultiple* within Section 4.4.

Alternative Communication Buffer (Un)Packing Approaches

The communication buffer (un)packing routines are similar in structure to those employed in the *update-halo* kernel (Section 3.2.12). The optimisations applied to this kernel are therefore also broadly applicable to the functionality required for the communication buffers. A modified version was therefore developed in which the (un)packing routines which operate on the communication buffers for the top and bottom mesh edges were restructured such that the OpenMP `do` directives were relocated to the outer k-loop which has a significantly shorter trip-count. A `collapse(2)` directive was also specified to ensure appropriate levels of parallelism are generated, with a potentially improved memory access pattern. Versions which employ this modified approach are referred to as

BufferCollapse within Section 4.4.

A further version was also created to examine alternative approaches for potentially improving the efficiency of the communication buffer (un)packing, in cases in which OpenMP parallelism is utilised across multiple processor sockets. In this version the code was restructured to allow the top and bottom buffers to be (un)packed simultaneously using half of the available threads to operate on each buffer. An identical approach to that described in Section 3.2.12, which utilises *nested*-parallelism and OpenMP v4.0 thread placement directives, was therefore again employed. To ensure that the top and bottom communication buffers were each exclusively located within the correct memory sub-systems, the buffers were initialised (*"first-touched"*) by threads with the correct processor affinity. The version which employed this approach is denoted by the description *IntelOMPNested* within Section 4.4.

4.4 Results Analysis

To assess the performance, at scale, of the MPI-only and hybrid (MPI+OpenMP) programming models and the various optimisation techniques examined as part of this research, a series of experiments were conducted. Sections 4.4.1 and 4.4.2 document the results of these experiments for both codebases. In these, performance was assessed using the $15{,}360^2$ cell problem, executed for 2,955 timesteps, from the standard CloverLeaf benchmarking suite. This was strong-scaled to high node counts on a range of state-of-the-art system architectures, specifically the Archer, Mira and Vulcan platforms (Section A.1).

During a particular experiment on each platform, all versions of the mini-app were executed within the same node allocation to eliminate any performance effects due to different topology allocations from the batch scheduling systems. Additionally, to reduce the effects of system noise, unless otherwise noted, the results presented here are averages from three separate executions of each individual experiment. For clarity, the performance results are also expressed in terms of the *"speedup"* which each version achieved relative to the reference implementation. In these charts values greater than 1 represent a performance improvement, whilst values below 1 indicate a degradation in performance.

In the experiments on Archer, version 8.3.3 of the Cray CCE compiler and version 7.0.3 of the Cray MPICH communications library were utilised. To provide baseline performance results several experiments were also conducted using older versions of these technologies, specifically version 8.2.1 of Cray CCE and version 6.1.1 of Cray MPICH. Additionally, no huge memory pages were utilised apart from the experiments which explicitly examined the performance effects of this particular technology; in these 4MB huge memory pages were

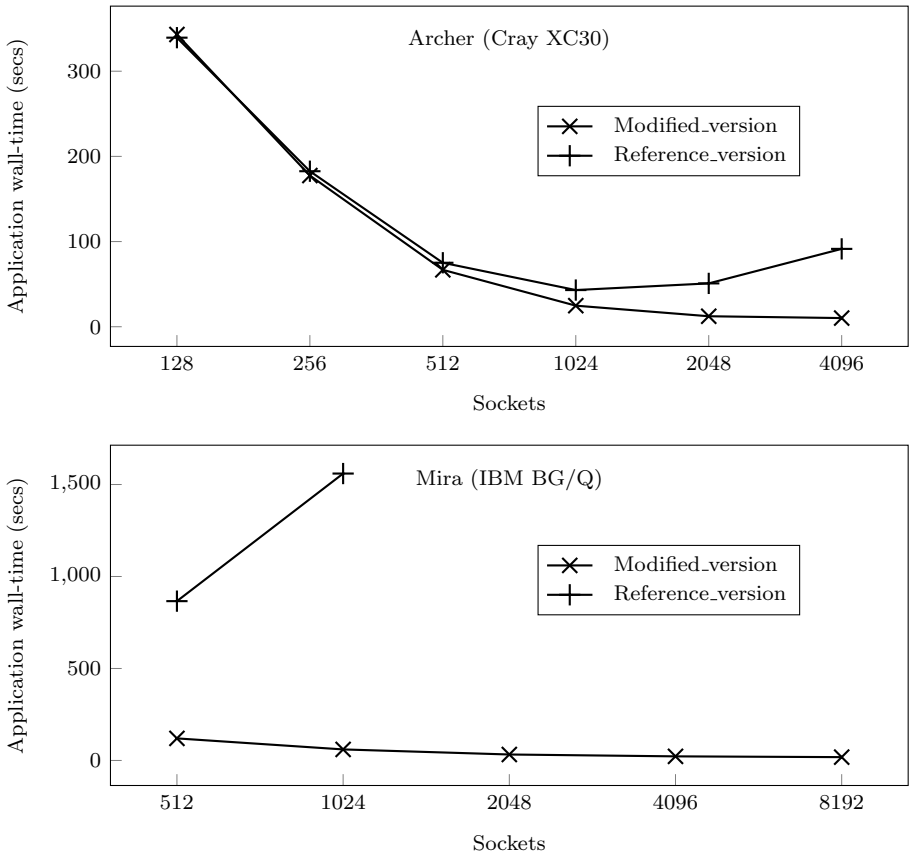


Figure 4.5: Distributed meta-data optimisation performance improvement

utilised. In the experiments on both Mira and Vulcan version 14.1 of the IBM XL Fortran compiler and version 12.1 of the IBM XL C compiler were employed, together with IBM's MPI communication library for the BG/Q, which is based on MPICH2 version 1.4.

4.4.1 MPI-only Results Analysis

The following sections analyse the performance of the MPI-only versions of the codebase and the candidate optimisations which have been applied to it. Additionally the energy-efficiency of two of the experimental platforms is also examined.

Distributed Meta-data

Figure 4.5 presents the performance improvement obtained through the application of the distributed meta-data optimisation (Section 4.2.1) to the original implementation, on both the Archer and Mira platforms.

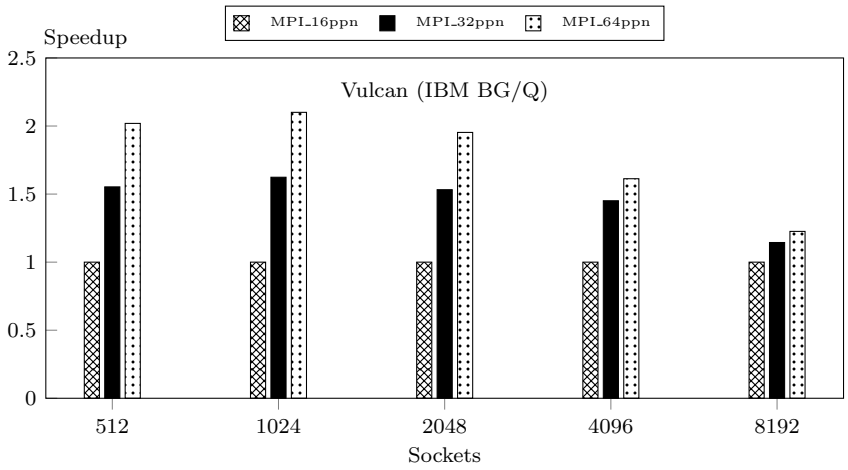


Figure 4.6: MPI processes / node configuration options on Vulcan

On Archer the scaling of both versions is initially (<512 sockets) broadly equivalent, however, beyond this point the distributed meta-data approach delivers significant performance advantages. The performance of the original reference implementation *"turns-over"* at approximately 1,024 sockets whilst the modified implementation continues to scale up to 4,096 processor sockets. During the 2,048 and 4,096 processor socket experiments this optimisation resulted in a 4.1× and 8.99× improvement in performance respectively, relative to the original implementation.

On the BG/Q architecture of Mira, however, the performance disparity between the two versions is even more severe. Scaling the original application from 512 to 1,024 sockets actually causes application execution time to increase by ∼1.8× and to completely fail beyond 1,024 sockets due to the more limited memory resources available per node on the BG/Q architecture. On this platform utilising the distributed meta-data optimisation improved the performance of the application by 7.2× and 26.0× in the 512 and 1,024 socket experiments respectively and enabled the application to be scaled successfully from 512 up to 8,192 processor sockets.

All the experiments documented in subsequent sections of this chapter utilise versions of both the MPI-only and hybrid (MPI+OpenMP) codebases which employ this distributed meta-data optimisation. Henceforth, this version is therefore referred to as the new *"reference"* implementation.

Utilisation of Hardware Threads and Huge Memory Pages

To determine the optimal approach with which to execute the MPI-only version of the codebase on the BG/Q architecture a series of experiments were conducted

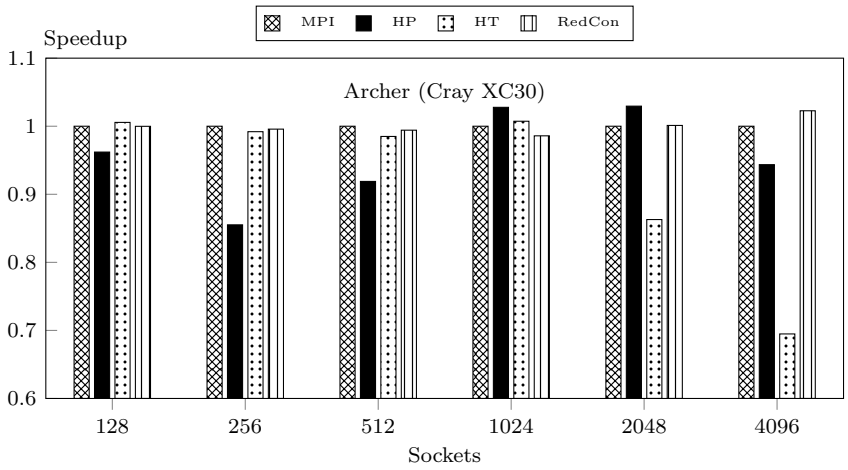


Figure 4.7: Huge-pages, hyper-threads and consolidated reduction

on Vulcan to examine the use of varying numbers of hardware threads. Experiments were therefore conducted using 1, 2 and 4 hardware threads per core, which equates to 16, 32 and 64 MPI processes per node respectively. Figure 4.6 presents the results of these experiments. The results show that the use of the additional hardware threads is indeed beneficial for this codebase at all the experimental scales examined. Their use provides a greater performance benefit in the smaller scale experiments, i.e. when each node/process has a larger allocation of the overall computational mesh and the performance of the codebase is limited more by computational resources. During the 64 socket experiment utilising 2 hardware threads per core improved application performance by $\sim$1.6$\times$ whilst utilising all 4 hardware threads improved performance by $\sim$2.0$\times$. In the 8,192 socket experiment these performance improvements reduced to $\sim$1.1$\times$ and $\sim$1.2$\times$ respectively. Based on these results all subsequent experiments with the MPI-only codebase on the BG/Q architecture were configured to utilise all 4 hardware threads (64 MPI processes per node).

A series of experiments was subsequently undertaken on Archer to examine the use of the additional hardware threads (Intel Hyper-threads) available on the Cray XC30 architecture. Figure 4.7 presents the results of these experiments, in which the abbreviation *HT* is used to denote versions which employed this technology. These results demonstrate a significantly different trend to those obtained from the Vulcan platform, specifically that the use of the additional hardware threads does not affect application performance in the smaller scale experiments ($\leq$1,024 sockets). In the larger scale experiments (>1,024 sockets), however, the use of this technology caused a significant degradation in application performance, resulting in a 1.15$\times$ and 1.44$\times$ slowdown in the 2,048 and 4,096 socket experiments respectively.

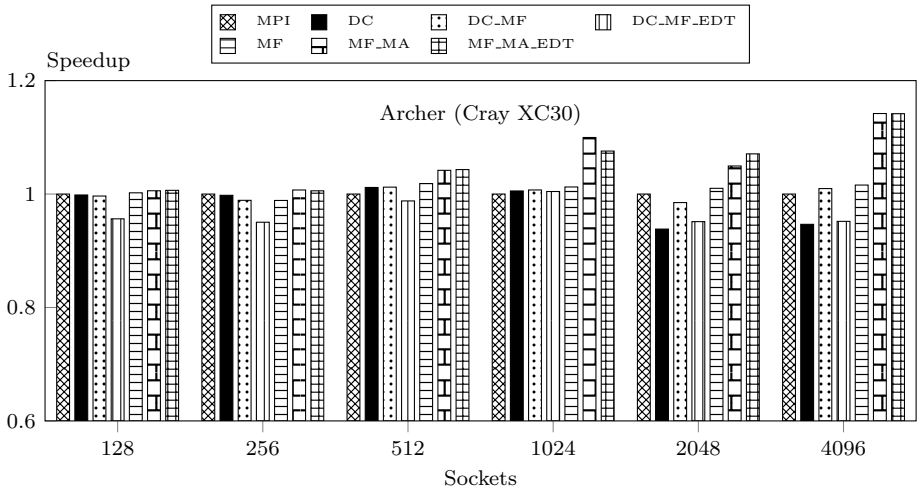


Figure 4.8: Message aggregation and early transmition optimisations

Additionally, the use of huge memory pages was also examined in a series of experiments on the Archer platform. The results (Figure 4.7) from these experiments (denoted using the abbreviation *HP*) do not demonstrate a discernible performance trend. In the majority of the experiments, however, employing this technology resulted in significant performance degradations of up to ∼1.2× and only relatively minor performance improvements (<3%) in the 1,024 and 2,048 socket experiments.

Message Aggregation

The results from the experiments which examined the effect of the MPI message aggregation optimisation are shown in Figures 4.8 and 4.9 for the Archer and Vulcan platforms respectively. The charts document the speedup achieved by each version relative to the reference MPI-only implementation which is shown with a speedup of 1 for all experimental scales examined. The results show that the use of this technique facilitated significant performance improvements for the application as the scales of the experiments were increased on Archer, reaching 1.14× and 1.1× at 4,096 and 1,024 sockets respectively. At the smaller scales examined the performance of these versions matched or slightly exceeded (<1%) that of the reference MPI-only implementation.

This trend is repeated on Vulcan with the results showing a consistent increase in the speedup achieved through the use of this technique as the scale of the experiments is increased. In the 512 processor socket experiments this optimisation delivered on average a 1.07× improvement in performance, which increased to 1.3× on average in the 8,192 socket experiments.

This demonstrates that reducing overall message transmission overheads, by

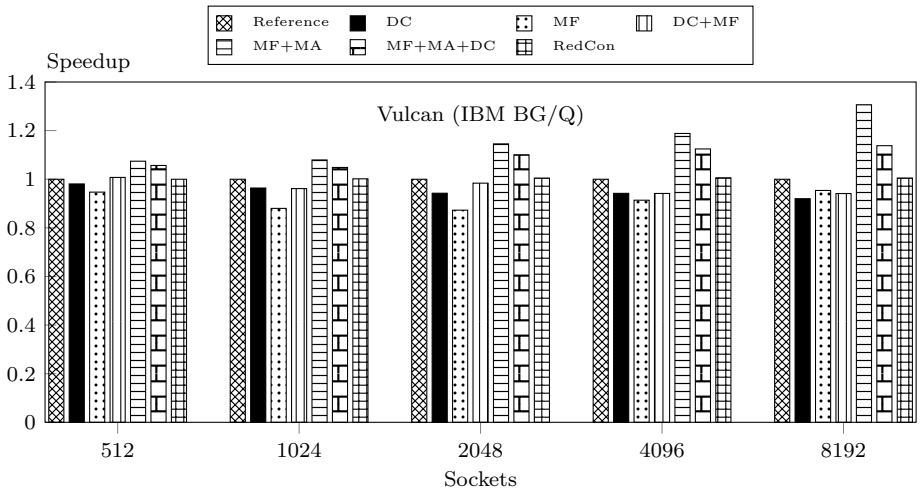


Figure 4.9: Performance of MPI-only Optimisations on Vulcan

aggregated data into fewer larger messages, can deliver significant improvements in performance for this class of applications.

Diagonal Communications & Communicating Multiple Fields Simultaneously

Figures 4.8 and 4.9 also show the effect on application performance of the *"Diagonal Communications"* and *"Communicating Multiple Fields Simultaneously"* optimisations on the Archer and Vulcan platforms respectively. The results show that at the higher socket counts examined on Archer the use of diagonal communications has a detrimental effect on application performance, reaching a 5.6% and 6.6% performance degradation in the 4,096 and 2,048 socket experiments respectively. In the smaller scale experiments (<1,024 sockets), however, the performance of this version matches that of the reference MPI-only implementation.

The results recorded on Vulcan show that this optimisation had a detrimental effect on overall application performance at all of the experimental scales examined, with the effect increasing as the scales of the experiments were increased. At 8,192 processor sockets the slowdown in application performance reached 8.7% relative to the reference MPI-only implementation. It is also evident that combining this optimisation with the version which employs the *"Message Aggregation"* strategy also significantly reduces performance at scale. This version now only achieved a 1.14× speedup over the reference MPI implementation at 8,192 processor sockets, compared to a 1.3× speedup achieved by the version which only utilised the *"Message Aggregation"* optimisation. This indicates that for this class of application the overheads incurred by sending the additional

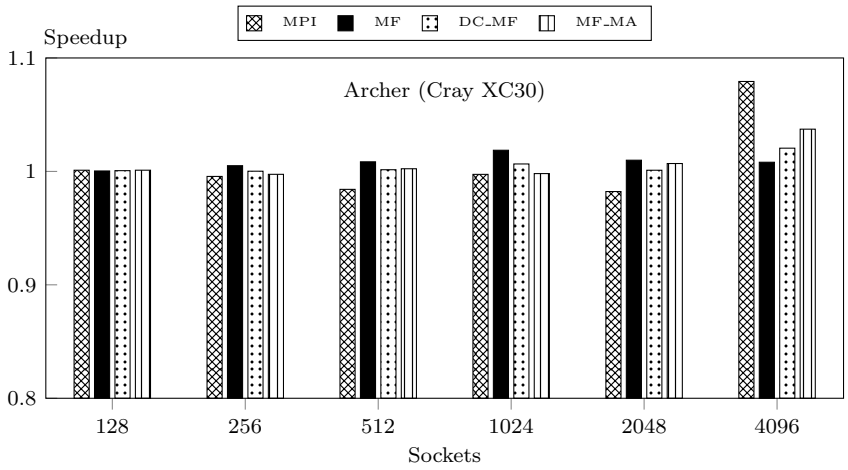


Figure 4.10: Pre-posting MPI receives on Archer

very small diagonal communication messages outweigh the savings made by reducing synchronisation operations between the communication phases of the application.

Additionally, the results also demonstrate that the *"Communicating Multiple Fields Simultaneously"* optimisation also generally has a detrimental effect on application performance. On Vulcan the performance of this version is consistently worse than the reference MPI implementation at all of the experimental scales examined, with the performance degradation reaching 6.2% and 6.3% in the 4,096 and 8,192 socket experiments respectively. The results from the Archer platform do not, however, exhibit this trend with the performance of the version which incorporates this optimisation matching that of the reference MPI-only implementation at all of the experimental scales examined. This indicates that on the BG/Q architecture it is more efficient to spread the network message injections out over multiple communication phases, and to employ additional synchronisation operations between these phases, rather than attempting to inject all of the messages into the network simultaneously. Furthermore, that the Aries NIC present in the Cray XC30 has greater capabilities at injecting messages into the communication interconnect than the NIC available within the BG/Q architecture.

Pre-posting MPI Receives

A series of experiments was also conducted to examine the effect on performance of *pre-posting* MPI receive operations, Figure 4.10 presents the results of these experiments. Due to time and supercomputer allocation limitations these experiments were only undertaken on the Archer experimental platform and not

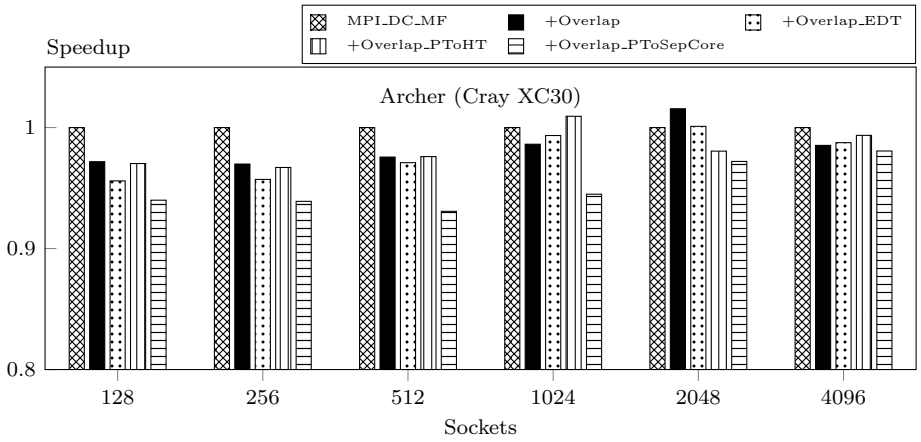
on either the Vulcan or Mira BG/Q platforms. This chart presents the results in terms of the speedup obtained by applying the *pre-posting* optimisation to a particular version of the MPI-only codebase relative to an identical version without the *pre-posting* optimisation applied to it.

The results show that for the four code variants examined in this research the *pre-posting* optimisation has a minimal effect on application performance in all of the experiments conducted up to the 2,048 socket experiment. In the 4,096 socket experiment, however, the results show some significant improvements in performance for all 4 code versions. These improvements reached 7.3% for the version which applied the pre-posting optimisation to the reference MPI-only implementation.

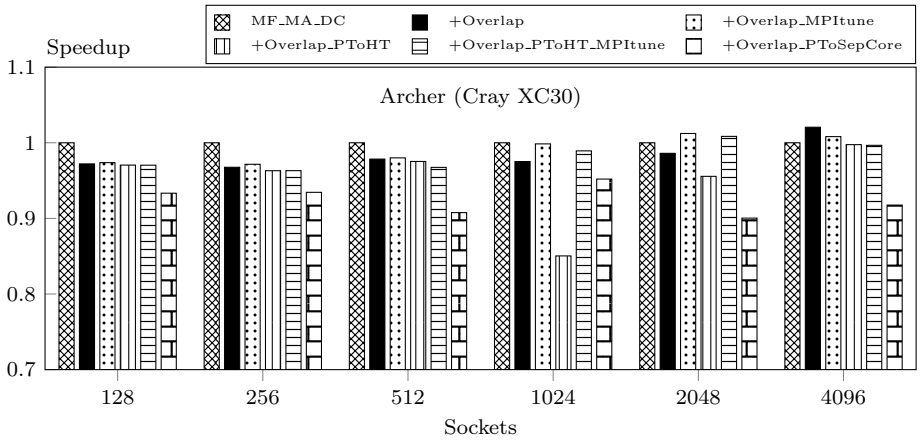
Overlapping Communications & Computation

To assess the utility of the optimisation technique which attempts to overlap communication and computational operations a series of experiments was conducted on both the Archer and Vulcan experimental platforms. Figures 4.11 and 4.12 present the results from these experiments on the Archer and Vulcan experimental platforms respectively. Results obtained on the Archer platform by applying this optimisation to MPI-only versions of the codebase which do not aggregate communication messages, are presented in Figure 4.11a, whilst results obtained through the use of this optimisation with versions which do aggregate MPI messages are presented in Figure 4.11b.

The results documented in Figure 4.11 show that on Archer the use of this optimisation generally results in a small but consistent degradation in application performance relative to equivalent versions which do not incorporate this optimisation. In all of the experiments below 512 processor sockets the performance of the code versions which attempt to overlap communication operations with computation are worse than that of the equivalent non-overlapping version. The experiments at these scales have a larger computational mesh size per MPI process and will thus be more affected by the sub-optimal memory access patterns resulting from this optimisation. This is due to the fact that proportionally less of the computational mesh will fit within the processor caches compared with the larger scale experiments. At the larger experimental scales (>1,024 sockets) the trends in the results are less clear, with some of the versions which incorporate the overlapping technique matching and fractionally, but not significantly, exceeding the performance of the reference implementation. Generally, however, the performance of the versions which incorporate the overlapping optimisation are worse than that of the reference implementation. This is due to the additional message transmission overheads which these versions incur and



(a) Non-aggregated Message Version



(b) Aggregated Message Version

Figure 4.11: Performance of computation/communication overlap on Archer

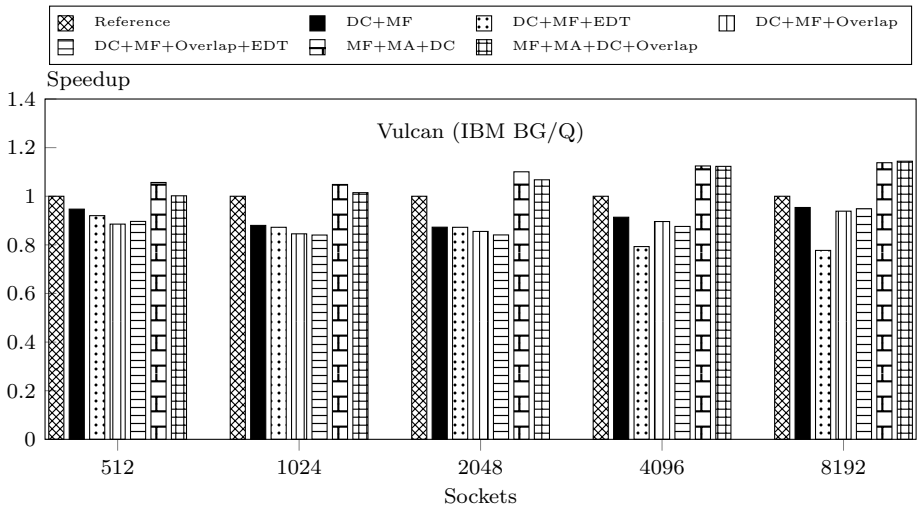


Figure 4.12: Early-sending & communication overlap optimisations on Vulcan

also the reduction in performance caused by the sub-optimal memory access pattern which they require.

Additionally, these results also indicate that the use of the explicit *"progress threads"* supported by the Cray MPI communication library does not deliver any significant performance benefits or facilitate greater overlap between the communication and computation operations. The performance delivered by the code versions which utilised this technology (denoted by the abbreviation PT) is broadly the same as the equivalent versions which did not. The results also show that explicitly dedicating a separate CPU processor core to execute a progress thread (*PToSepCore*), at the expense of using this processing resource for the main application workload, delivers significantly worse overall performance than utilising a CPU hyper-thread to execute the progress thread ($PToHT$). Similarly the results show that increasing the number of internal communication buffers within the Cray MPI communication layer and the threshold below which messages will be sent using the *"eager"* communication protocol (version denoted by the abbreviation *MPItune*) also does not significantly affect overall application performance either positively or detrimentally.

The results obtained from the experiments on the Vulcan platform show a similar trend in performance. At the smaller experimental scales ($\leq$2,048 sockets) examined as part of this research the versions which incorporate the overlapping optimisation perform fractionally, but consistently, worse than the equivalent versions which do not incorporate the optimisation. In the experiments beyond 2,048 sockets; however, this performance disparity disappears and the performance of these versions matches, but does not exceed, that of the equivalent versions which do not incorporate the optimisation.

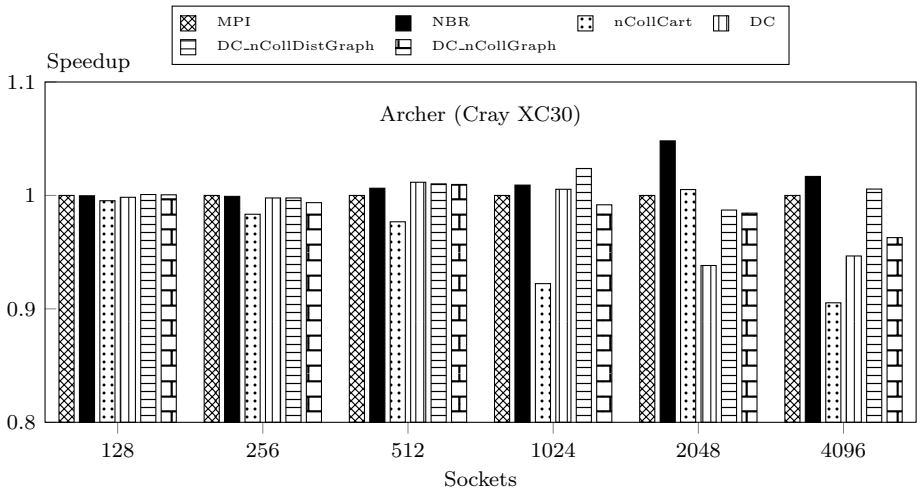


Figure 4.13: Performance of MPI v3.0 constructs on Archer

Eager Data Transmission

As part of this research a series of experiments was also conducted to examine the effect on performance of the *"Eager Data Transmission"* optimisation described in Section 4.2.1. The results from experiments on the Archer platform with versions which incorporate this optimisation technique are presented in Figures 4.8 and 4.11, whilst Figure 4.12 documents results obtained by employing this optimisation on the Vulcan platform. It is clear that on both platforms the use of this optimisation consistently delivers a performance degradation relative to equivalent versions which do not incorporate it. Figure 4.8 shows that on Archer the use of this candidate optimisation can result in a performance degradation of up to 6% in overall application performance.

On Vulcan, however, the performance obtained by applying this optimisation to a code variant which already incorporates the *"diagonal communications"* and *"communicating multiple fields simultaneously"* optimisations, results in virtually identical performance being delivered in all of the experiments $\leq$2,048 sockets. In the larger scale experiments ($\geq$4,096 sockets), however, the use of this optimisation results in a significant degradation in performance, reaching a 22.7% increase in application runtime in the 8,192 socket experiment. Additionally the results show that applying this optimisation to a code variant which employs the *"overlapping communications"* technique, also does not significantly affect overall performance, either beneficially or detrimentally, at any of the experimental scales examined as part of this research.

MPI v3.0 Constructs

To examine whether the use of the MPI v3.0 communication constructs described in Section 4.2.1 could deliver any performance benefits for this class of application a series of experiments was conducted on the Archer platform. Figure 4.13 presents the results from these experiments. The IBM MPI communication library available on the BG/Q does not yet support these constructs which prevents similar experiments from being undertaken on this architecture. The results show that the use of the non-blocking MPI reduction operation does not have a significant effect on application performance in the experiments $\leq$1,024 sockets, as the run-times are on average virtually identical to those of the reference MPI-only implementation. The use of this construct in the larger scale experiments can, however, deliver some modest improvements in application performance, in these experiments run-times were reduced by 4.6% and 1.6% respectively in the 2,048 and 4,096 socket cases.

The performance of the version which employed the cartesian neighbourhood collective operations (labelled *"nCollCart"* in Figure 4.13) was virtually identical to the reference implementation at the smaller experimental scales examined. As the scale of the experiments was increased, however, the performance of this version was generally not able to match that of the reference implementation; in the 4,096 socket experiment its performance was 10.5% slower. The performance of the versions which utilised the graph-based neighbourhood collective operations (*"DC_nCollDistGraph"* and *"DC_nCollGraph"*) was also generally superior to that of the versions which employed the equivalent cartesian operations. The code variant which utilised the distributed graph communication construct (*"DC_nCollDistGraph"*) was the most performant, compared to the equivalent version which incorporated the fully connected graph communication constructs (*"DC_nCollGraph"*), and was able to match the performance of the reference implementation at all the experimental scales examined. At no point in these experiments, however, did the use of any of the MPI v3.0 neighbourhood collective operations deliver any significant improvements in overall application performance relative to the reference implementation. This indicates that although these constructs deliver programmer productivity benefits through reductions in the number of MPI library calls required to complete a particular operation, reducing the number of these calls does not deliver any performance improvements for this class of applications. The Cray MPI runtime system, present on the Archer platform, is also not yet able to utilise the additional information provided by these new constructs (e.g. the communication topology of the application) in order to improve overall application performance.

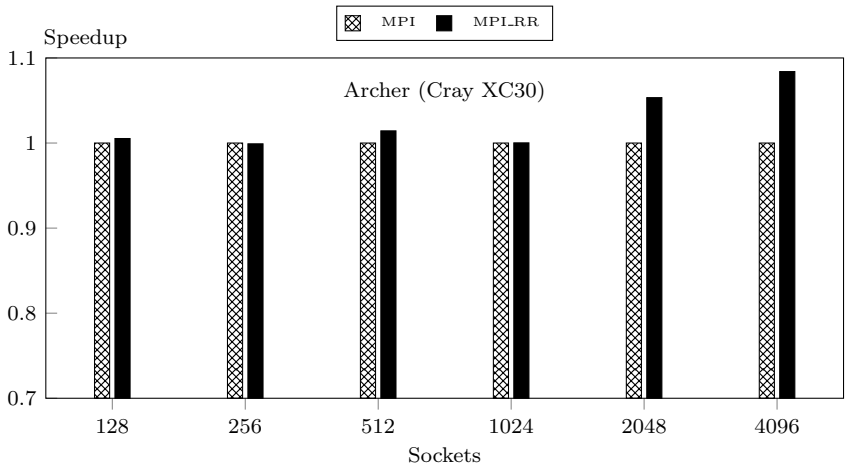


Figure 4.14: MPI rank reordering on Archer

MPI Rank Reordering

To assess the effect on performance of the rank reordering optimisation (Section 4.2.1) a series of experiments were conducted using the Archer Cray XC30 platform. On Cray platforms the environment variable `Mpich_Rank_Reorder_Method` determines the order in which MPI ranks are assigned to cores. Within an allocation the number assigned to a particular core corresponds to the MPI rank which will ultimately be executed on it. By default (`Mpich_Rank_Reorder_Method=1`) cores are numbered consecutively within a node with this numbering continuing on subsequent nodes. Custom mappings can be specified using a rank reorder file (`Mpich_Rank_Reorder_Method=3`) and these can be generated either manually or automatically using Cray tools.

In these experiments the *Grid_order* tool was employed to manually generate a custom rank mapping file. As Archer has 24 cores per node, the blocks assigned to each node were specified to have dimensions of 6×4 chunks (Figure 4.3b). The reference MPI implementation was then executed using both the default and customised rank placement settings. Figure 4.14 presents the results of these experiments and shows that in these experiments this optimisation improved overall application performance by 5.1% and 7.7% in the 2,048 and 4,096 socket experiments respectively.

This demonstrates that modifying the layout of application processes within a particular supercomputer node allocation to better reflect the communication pattern of an application can deliver significant improvements in performance. As the scales of the experiments are increased the rank reordering optimisation also delivers a greater improvement in overall performance relative to the default ordering. These performance improvements are realised through applications

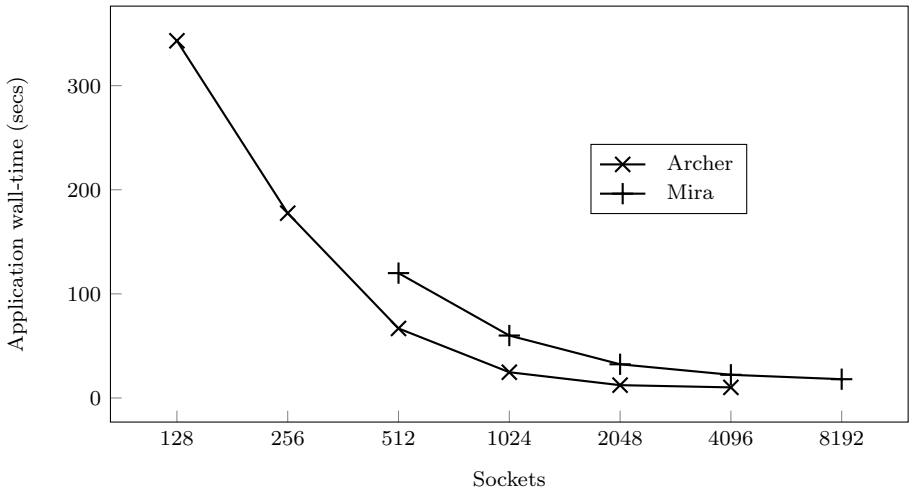


Figure 4.15: Performance due to the distributed meta-data optimisation

being able to better utilise the shared memory resources, available within the nodes of particular supercomputer platforms, for inter-process communication rather than having to exclusively rely upon slower inter-node message transmissions to move data across larger distances.

Reduction Consolidation

Figures 4.7 and 4.9 present the results from the experiments conducted on Archer and Vulcan respectively to examine the performance of the *"Consolidated Reduction"* optimisation (Section 4.2.1). The results show an almost identical trend on both system architectures, that is that the incorporation of this optimisation into the application does not significantly affect performance either beneficially or detrimentally. The performance of the code variant which includes this optimisation is identical to that of the reference implementation even as the scales of the experiments are increased to 4,096 and 8,192 sockets on Archer and Vulcan respectively.

Architecture Comparison

Figure 4.15 shows the performance results obtained from the experiments with the MPI-only codebase on both the Archer (Cray XC30) and Mira (IBM BG/Q) experimental platforms. They demonstrate that approximately 2-4× more processor sockets are required for the runtime performance of the application on the BG/Q architecture to match that of the Cray XC30 architecture.

Using the application power consumption instrumentation facilities available on both the Archer and Mira platforms (Section 4.2.2) a series of experiments

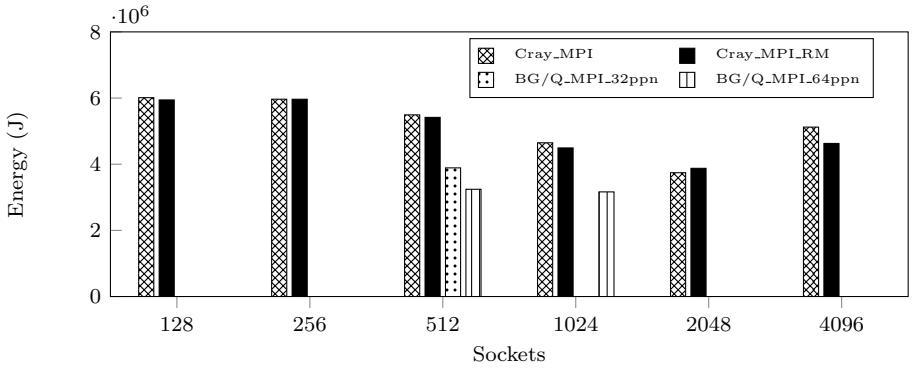


Figure 4.16: Energy to solution analysis on Archer(XC30) and Mira(BG/Q)

were undertaken to examine the energy consumed by the nodes of each platform in achieving equivalent numerical solutions. The results of this analysis are shown in Figure 4.16. On the Archer platform average figures from three separate runs of each experiment are presented, however, due to time and machine allocation limitations it was only possible to obtain one run for each of the results shown for the Mira platform.

The results from Archer show that the energy-to-solution profile decreasing consistently as the application is scaled from 128 to 2,048 processor sockets. Beyond this point, however, this profile "turns-over" and the energy required to achieve a solution on 4,096 processor sockets is actually significantly greater (1.3×) than that required to achieve the same solution on 2,048 sockets. This occurs despite the fact that the actual runtime performance of the application continues to decrease between the 2,048 and 4,096 socket experiments. This reduction in runtime is, however, lessened by the fact that the communication operations within the application are becoming increasingly dominant and limiting its scalability, and it is therefore not sufficiently large enough to offset the approximate doubling of power consumption which occurs between the 2,048 and 4,096 socket experiments.

The results also show that the MPI rank reordering optimisation delivers approximately a 1.1× reduction in energy consumption in the 4,096 sockets experiment by reducing the actual runtime of the application and thus its overall energy consumption. The energy consumed by this version was, however, practically identical to the reference implementation in all the other experimental scales examined.

Additionally, the results from the Mira platform demonstrate that—for the data-points which it was possible to collect as part of this research—the BG/Q architecture is able to deliver significant advantages over the Cray XC30 architecture in terms of the energy required to achieve equivalent solutions for this

application. These energy-to-solution advantages reached as high as 1.7× in these experiments.

4.4.2 Hybrid (MPI+OpenMP) Results Analysis

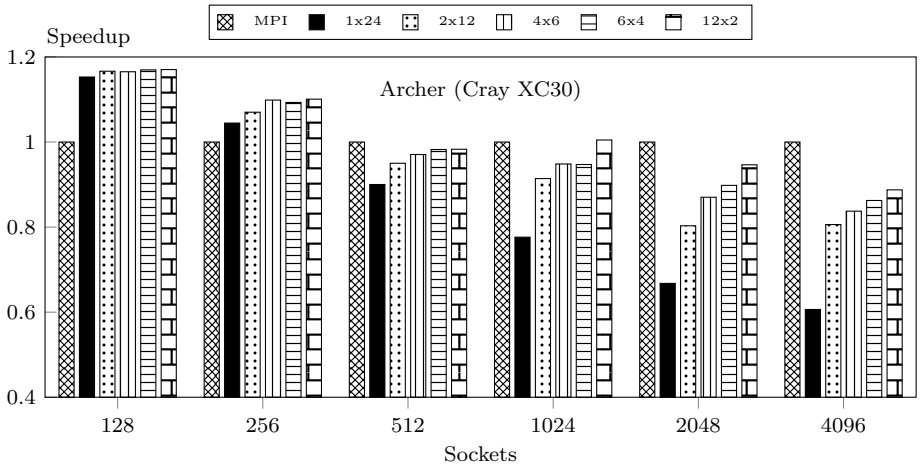
Building on the research documented in Chapter 3 a series of experiments was conducted to assess whether the MPI+OpenMP hybrid programming model can deliver any performance advantages for this class of applications, specifically at large-scale. Additionally, these experiments also examined the utility of the candidate optimisation techniques outlined in Section 4.3. The performance of this codebase and each optimisation technique are examined in the following sections.

MPI-only and MPI+OpenMP Comparison

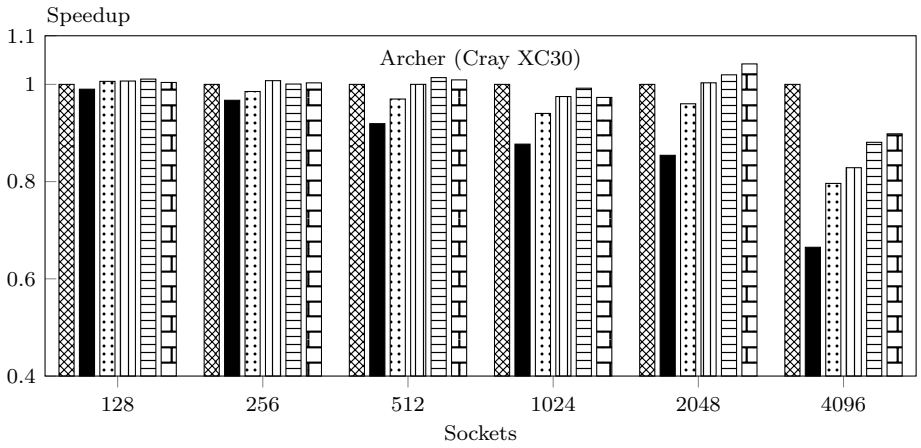
To determine whether the hybrid (MPI+OMP) version of the codebase can deliver any performance advantages compared to the reference MPI-only version a series of experiments was conducted on both the Archer and Vulcan platforms. On both architectures these experiments examined the performance of the hybrid version when executed using a range of different ratios between the number of MPI processes and OpenMP threads employed per node. Figures 4.17 and 4.18 present the results of these experiments; additionally on the Archer platform separate experiments were also conducted using a range of different Cray MPI and compiler versions.

Figure 4.17a shows the results from the experiments on Archer using the older version of the Cray MPI library and compiler infrastructure; please refer to Section 4.4 for more details on the specific versions employed. The results show that in the smaller scale experiments (128 and 256 sockets) employing the hybrid programming model can deliver significant performance advantages over the MPI-only approach. In these experiments this performance advantage was as much as 1.2× in the 128 socket experiment but declined to 1.1× in the 256 socket experiment. This decline in performance relative to the MPI-only version of the codebase continued as the scales of the experiments were increased resulting in the MPI-only approach delivering superior performance in all of the experiments ≥512 sockets. The results show that the performance of the hybrid versions was inversely proportional to the number of OpenMP threads utilised in the experiments, with the performance of the 12MPIx2OpenMP configuration being consistently superior and the 1MPIx24OpenMP ratio the least performant. In the 4,096 socket experiment the performance of these versions was 1.1× and 1.6× worse than the MPI-only version of the codebase.

This performance trend is, however, not matched in the results obtained



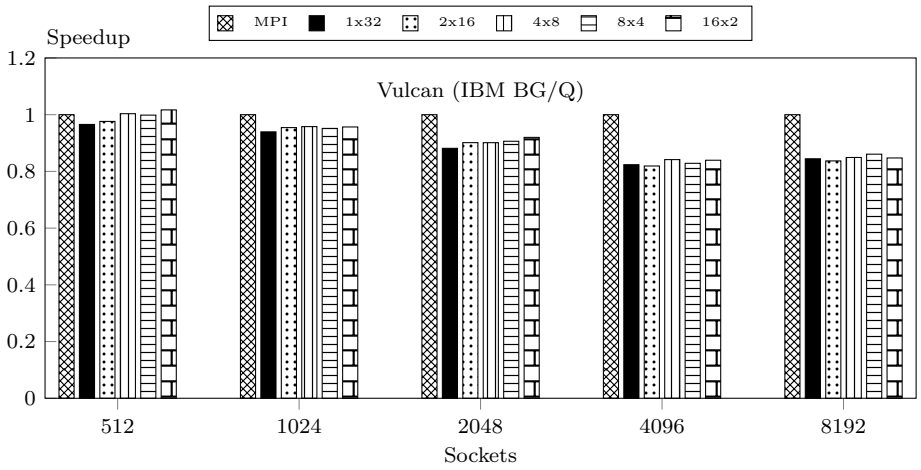
(a) Cray MPI v6.1.1



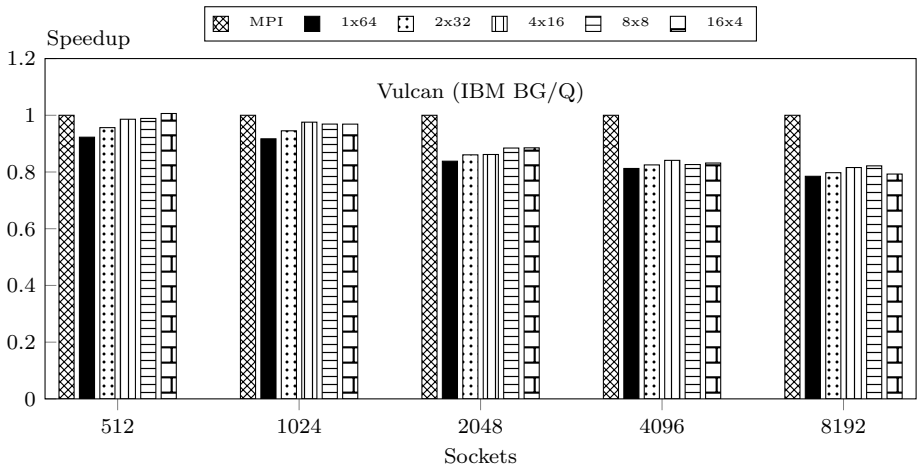
(b) Cray MPI v7.0.3

Figure 4.17: Hybrid (MPI+OMP) performance on Archer

from the experiments on Archer with the more recent versions of the Cray MPI library and compiler software (Figure 4.17b). These results indicate that improvements in the Cray MPI library now enable the performance of the MPI-only codebase to match that of the hybrid variants in the small scale experiments and to continue to exceed the performance of the hybrid versions in the 4,096 socket experiment by as much as 1.5×. For the hybrid versions the ratio of 12MPIx2OpenMP is again the most performant in the larger scale experiments (2,048 and 4,096 sockets); however, the 6MPIx4OpenMP and 4MPIx6OpenMP ratios now deliver slightly superior performance in the smaller scale experiments (<2,048 sockets). The hybrid version which employs 24 OpenMP threads across the 2 processor sockets within each node is again the least performant configuration; however the performance of this version is able to match the other hybrid



(a) Vulcan 32ppn



(b) Vulcan 64ppn

Figure 4.18: Performance of the MPI+OMP implementation on Vulcan

configurations in the smaller scale experiments. Although certain experiments do show some of the hybrid versions delivering superior performance compared to the MPI-only implementation, these performance improvements are generally <4%.

The results obtained from the equivalent experiments on the Vulcan platform (Figure 4.18) show a similar performance trend to that observed on Archer with the more recent version the Cray MPI library. In these experiments the performance of the hybrid code variants is again able to match that of the MPI-only version in the small scale experiments on 512 sockets. As the scale of the experiments is increased; however, the relative performance of the hybrid version decreases, and in the 8,192 socket experiment this implementation is 1.16-1.2× slower than the reference MPI-only implementation. This performance trend is

observable in the experiments with both 32 (Figure 4.18a) and 64 (Figure 4.18b) processes per node.

This demonstrates that with less efficient MPI implementations hybridising codebases with OpenMP can deliver significant performance advantages when application performance is dominated by computational operations, as it is in the smaller scale experiments examined here. This is due to the hybrid approach facilitating the more efficient use of the shared memory resources within the nodes of the supercomputer. Furthermore, it is possible to improve the efficiency of an MPI implementation such that the application performance, which is achievable with the MPI-only model, is able to match that of a hybrid approach. The results also show that due to the additional threading overheads (e.g. OpenMP fork/join and synchronisation overheads etc.) which are a consequence of the hybrid approach, the MPI-only approach is significantly more performant at high node counts for this class of application. In these particular experiments the size of the computational mesh assigned to each node is significantly smaller than in the low node count experiments and consequently the performance of the application is increasingly dominated by communication operations. Additionally as the memory footprint required per node is considerably smaller, the benefits due to the use of the threading constructs, which result from the more efficient utilisation of the shared memory resources, are substantially reduced and do not offset the additional overheads caused by the use of a hybrid approach. The results also demonstrate that the overheads due to the use of the OpenMP constructs increase with the number of threads utilised per MPI rank.

Message Aggregation

To examine whether the optimisation of aggregating MPI messages can also provide a performance benefit for the MPI+OpenMP versions of the codebase a series of further experiments was conducted on both the Archer and Vulcan platforms using a variant of the hybrid codebase which incorporated this optimisation. Figures 4.19 and 4.20 present the results of these experiments. In these charts the speedup due to the *"Message Aggregation"* optimisation is calculated relative to the performance of the reference implementation, when executed using the same MPI to OpenMP ratio.

The results show that on Archer this optimisation also delivers significant performance benefits for the hybrid version of the codebase, with the performance improvements growing as the scale of the experiments is increased. In the 128 socket experiment the performance of the reference implementation matches that of the version which incorporates this optimisation. With the

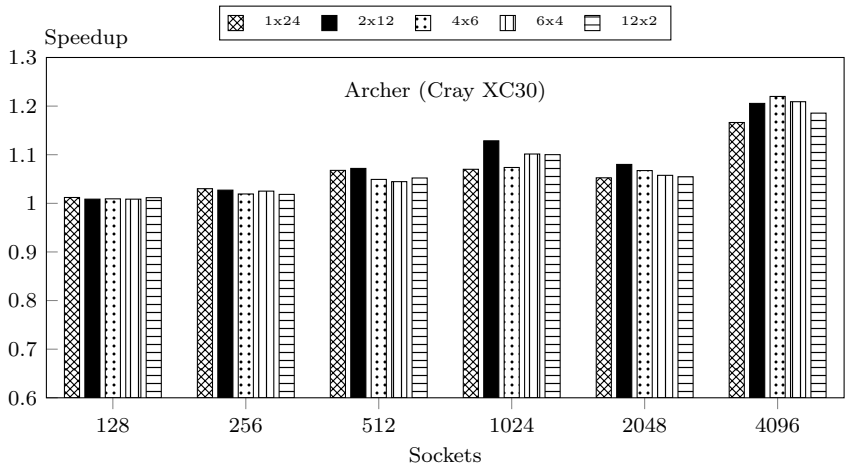


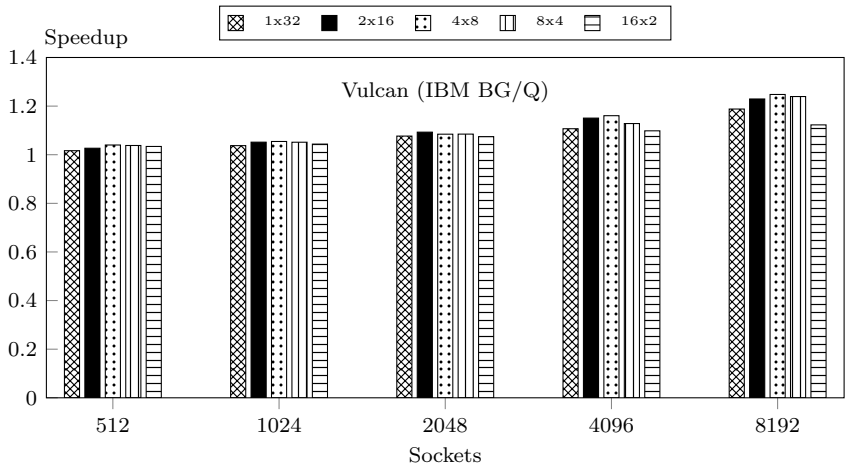
Figure 4.19: Message aggregation for the MPI+OMP version on Archer

exception of the 2,048 socket experiment, however, the results show a consistent increase in the speedup achieved due to message aggregation as the scale of the experiments is increased. In the 4,096 socket experiment the speedup due to this optimisation is as high as 1.22× the performance of the original implementation.

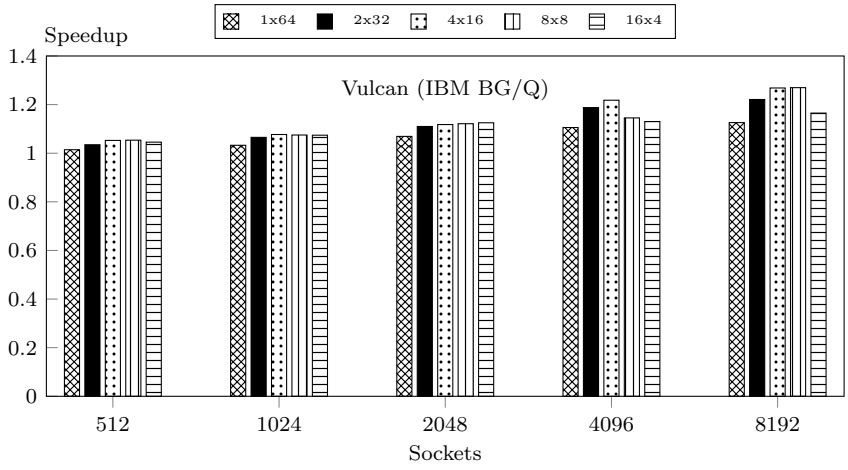
A similar trend can also be observed on the Vulcan platform for the 32 and 64 processes per node experiments (Figures 4.20a and 4.20b). The results again show the performance of the reference hybrid implementation matching that of the version which incorporates the message aggregation optimisation in the smaller scale experiments (512 processor sockets). As the scale of the experiments are increased the performance speedup due to this optimisation again increases, reaching up to a ∼1.27× improvement in the 8,192 socket experiment.

Individual Kernel OpenMP Optimisations

A series of experiments was undertaken to examine whether the individual kernel optimisations, identified in Chapter 3, can deliver any performance benefits when CloverLeaf is executed at significant scale on the Archer platform. Figure 4.21 presents the results of these experiments. In these charts the results labeled *"KernelOpts"* refer to the particular version which incorporates these optimisations. The results show that for the experiments which utilised the 1MPIx24OMP and 4MPIx6OMP configurations, employing these optimisations can deliver significant performance improvements in the smaller scale experiments, relative to the reference MPI-only and hybrid implementations. In the 128 socket experiment these optimisations achieved a ∼1.10× and ∼1.12× improvement in performance relative to the reference MPI-only implementa-



(a) Vulcan 32ppn



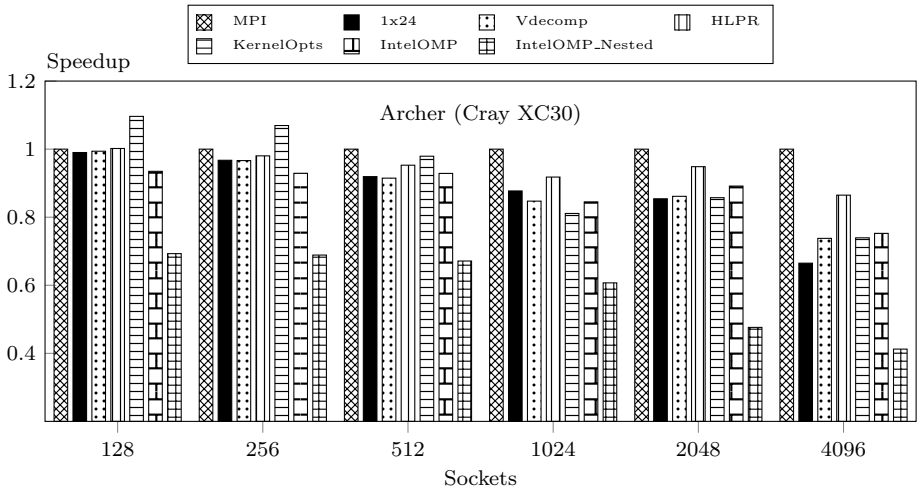
(b) Vulcan 64ppn

Figure 4.20: Message aggregation for the MPI+OMP version on Vulcan

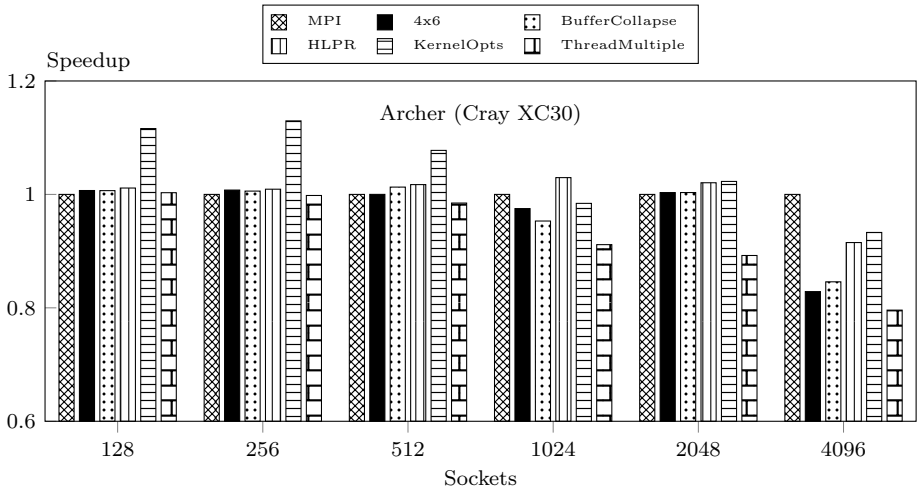
tion for the 1MPIx24OMP and 4MPIx6OMP configurations respectively. As the scale of the experiments is increased, however, the results show that this optimisation becomes less effective with relative application performance falling back to approximately match that of the reference hybrid implementation. This indicates that these optimisations are more effective when the amount of computational work, which each thread has to perform, is greater relative to the levels of communication operations, which is the case in the smaller scale experiments.

High-level Parallel Region

To determine whether the *"High-level Parallel Region"* optimisation discussed in Section 3.2.4, could deliver any performance benefits as the execution scales

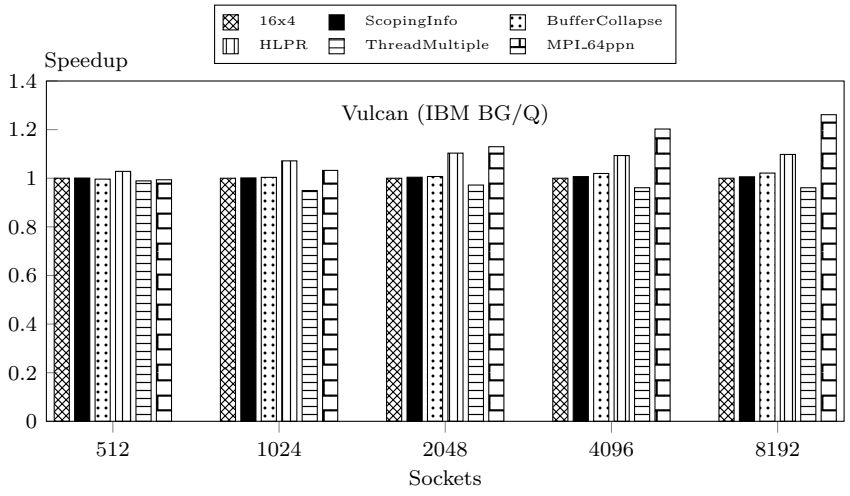


(a) 1 MPI process x 24 OMP threads / node

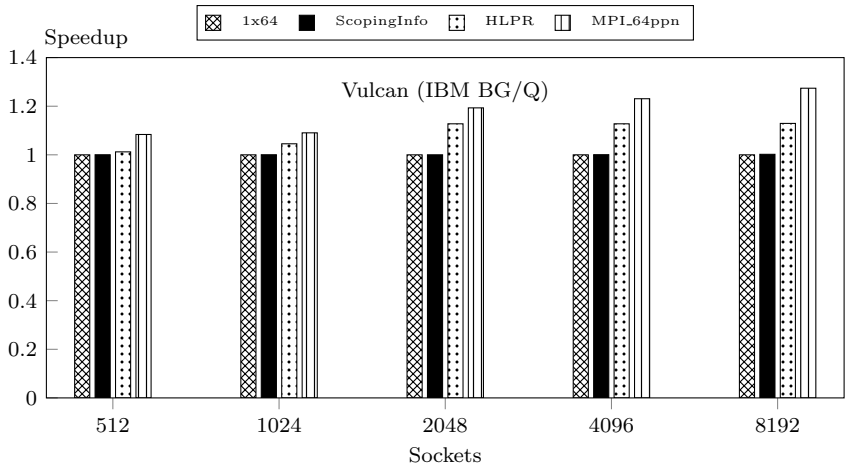


(b) 4 MPI processes x 6 OMP threads / node

Figure 4.21: Optimisations to the hybrid versions on Archer



(a) Vulcan 4MPIx16OMP configuration



(b) Vulcan 1MPIx64OMP configuration

Figure 4.22: Optimisations to the hybrid version on Vulcan

of the application are increased, a series of experiments was conducted on both Archer and Vulcan. The version labelled *"HLPR"* within Figures 4.21 and 4.22 shows the effect of this optimisation on the performance of CloverLeaf.

The results from the Archer platform show that the performance of this version is approximately equivalent to the reference hybrid version in the smaller scale experiments on 128 and 256 processor sockets. As the scale of the experiments is increased, however, the version incorporating this optimisation starts to consistently outperform the reference hybrid implementation for both the 1MPIx24OMP and 4MPIx6OMP experimental configurations. In the 4,096 socket experiments this optimisation delivered respective performance improvements of ∼1.3× and ∼1.1× relative to the reference hybrid implementation. In

several experiments with the 4MPIx6OMP configuration employing this optimisation also enabled the hybrid implementation to outperform the reference MPI-only implementation, although only by ∼2.9%.

A similar performance trend is also exhibited in the results obtained from employing this optimisation on the Vulcan platform. These results (Figure 4.22) show that the version which incorporates this optimisation consistently delivers superior performance compared to the reference hybrid implementation, and that the performance disparity grows significantly as the scale of the experiments is increased. In the 512 socket experiment this optimisation improved the performance of the hybrid codebase by ∼2.7% and ∼1.2% for the 16MPIx4OMP and 1MPIx64OMP configurations respectively. The improvement in performance, however, increases to ∼9% and ∼11% for these configurations in the 2,048 to 8,192 socket experiments respectively. Similarly employing this optimisation also enabled the hybrid implementation (16MPIx4OMP configuration) to out perform the MPI-only implementation by 3.4% and 3.6%, in the 512 and 1,024 socket experiments, respectively.

The fact that this optimisation delivers significantly more performance benefits in the larger scale experiments is likely due to the OpenMP synchronisation overheads representing proportionally more of the overall computational workload at these scales. As this optimisation contributes to reducing the levels of synchronisation within the hybrid codebase, it is therefore more effective in the experiments on the higher processor counts, as during these the size of the mesh processed by each thread is considerably reduced relative to the smaller scale experiments.

Vertical Rectangular Decomposition

To analyse the performance of the *"Vertical Rectangular Decomposition"* candidate optimisation (Section 4.3.1) a series of experiments were performed on the Archer platform using the 1MPI x 24OpenMP threads configuration. The version labelled *"ChangeDecomp"* in Figure 4.21a presents the results of these experiments. The results show that the performance of this version is virtually identical to that of the reference hybrid implementation in the 128 to 2,048 sockets experiments. The result from the 4,096 socket experiment, however, demonstrates that this version achieved a ∼9.9% performance improvement on average over the reference hybrid implementation. This indicates that this optimisation may deliver some performance advantages when OpenMP parallelism is utilised across multiple sockets and the amount of computational work per node is sufficiently small, such that minimising the data transfers across the inter-socket buses becomes important in achieving optimal performance.

MPI-OpenMP Integration Options Exploration

The experimental results obtained with the version of the hybrid implementation which employs the MPI-OpenMP integration optimisations described in Section 4.3.1 are shown in Figure 4.21 and 4.22a for the Archer and Vulcan platforms respectively. In both figures the version labelled *"ThreadMultiple"* presents the results obtained with this version. The results from the experiments on the Archer platform show that the performance of the version which incorporates this modification matches that of the reference hybrid implementation in the smaller scale experiments (128 and 256 sockets). In the larger scale experiments, however, as the performance of the application becomes increasingly dominated by the communication operations, this version performs consistently worse than the reference hybrid implementation. The results obtained from Vulcan demonstrate that on this platform the use of this construct also results in a performance degradation; however the reduction in performance is not as great as was observed on the Cray XC30 architecture.

Overall as this optimisation relates to how the communication operations are utilised within the application, this result indicates that the approach of initiating multiple MPI communication operations in parallel and in close temporal proximity, using OpenMP constructs, is not as performant as the original method utilised within in the reference version. This is due to additional mutual exclusion/locking overheads which are required within the MPI library in order to coordinate access to the underlying communication resources for each OpenMP thread. Additionally, as a significantly smaller reduction in performance is observed on the IBM BG/Q due to the utilisation of this approach, compared to the Cray XC30, this indicates that the implementation of the multi-threaded constructs within the MPI library is also more efficient on the BG/Q.

Alternative Communication Buffer Packing Approaches

The performance of the modified hybrid version which utilises OpenMP Nested Parallelism and the OpenMP v4.0 thread placement constructs (Section 4.3.1), with the aim of improving the performance of the communication buffer packing operations was examined in a series of experiments on the Archer platform. As the Cray OpenMP runtime system does not yet support the OpenMP v4.0 thread placement constructs, the Intel compiler and OpenMP runtime systems (version 14.0.4) were utilised for these experiments. Figure 4.21a presents these results together with those from an experiment with the reference hybrid implementation compiled using the Intel tool-chain in order to provide a baseline against which to compare the performance of the modified approach. The results

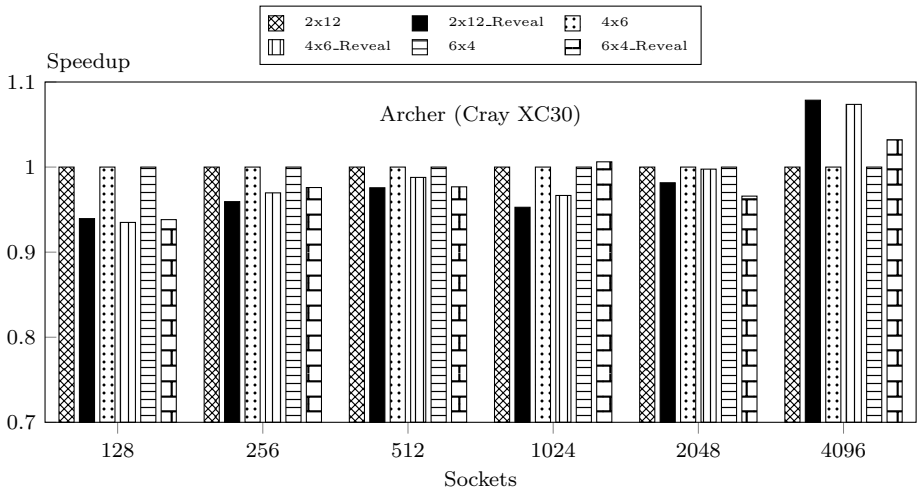


Figure 4.23: Hybrid version produced by Reveal on Archer

show that in all of the application scales examined the performance of the modified version is significantly worse than that of the reference hybrid implementation. In the 4,096 socket experiment the use of this modified approach results in a slowdown in overall application performance of $\sim$1.8$\times$ relative to the reference hybrid implementation. This demonstrates that the use of nested parallelism currently results in too much additional overhead for this modified approach to be viable for this class of application.

Additionally, Figures 4.21b and 4.22a also present results, from Archer and Vulcan respectively, of experiments with the version of the hybrid codebase which incorporates the modified (using loop inter-change and the `collapse` directive) communication buffer packing functionality described in Section 4.3.1. This version is labelled as *"CommsBufferCollapse"* within these charts. The results from both Archer and Vulcan show that the use of this modification does not significantly affect the overall performance of the codebase as in both cases the performance of the modified version is equivalent to that of the reference hybrid implementation in all of the experiments conducted as part of this research.

Automatic Hybridisation

To assess the performance at scale of the hybrid codebase produced automatically by the Cray Reveal tool (Section 3.2.13) a series of experiments was conducted on both the Archer and Vulcan platforms. Figures 4.23 and 4.24 present the performance results obtained through the use of this codebase on Archer and Vulcan respectively, and compare it against the reference hybrid implementation. The results from the Archer platform show that in the smaller

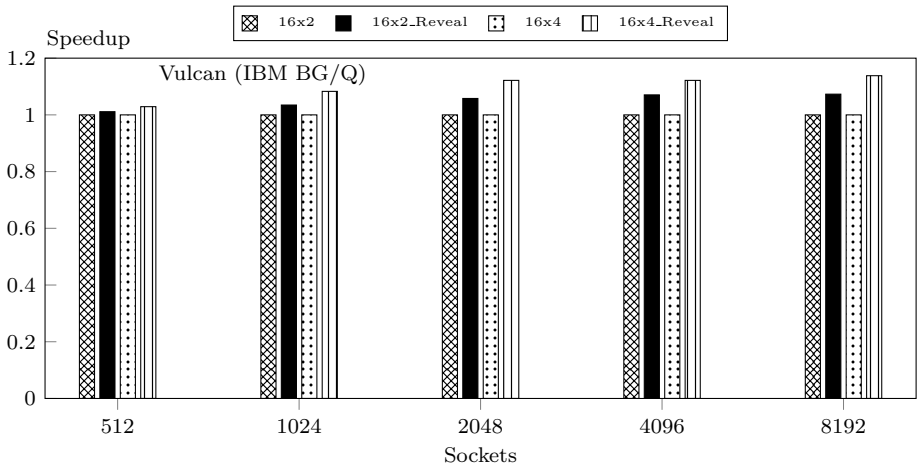


Figure 4.24: Hybrid version produced by Reveal on Vulcan

scale experiments on 128 sockets the performance of the version produced by Reveal is within ∼7.0% of the performance of the reference hybrid implementation. As the scale of the experiments is increased, however, this performance disparity reduces and in the largest experiment conducted (4,096 sockets) the version produced by Reveal actually significantly outperforms the reference hybrid implementation by as much as 7.3%, in each of the three configurations examined (2MPIx12OMP, 4MPIx6OMP and 6MPIx4OMP).

The results produced on Vulcan, however, demonstrate a significantly different performance trend. On this platform the performance of the hybrid version produced by Reveal is able to match that of the reference hybrid implementation in the smaller scale (512 socket) experiment. As the scales of the experiments are increased, the hybrid version produced by Reveal starts to deliver superior performance compared to the reference hybrid implementation for both the examined configurations (16MPIx2OMP and 16MPIx4OMP). In the largest experiment conducted on the BG/Q architecture (8,192 processor sockets) the hybrid version produced by Reveal is ∼12.1% faster than the reference hybrid implementation for the 16MPIx4OMP configuration.

These results indicate that the structure of the hybrid implementation produced by Reveal (i.e. one nested loop block per `parallel` region) may deliver some performance advantages over the structure implemented within the reference hybrid version in situations in which the size of the computational mesh processed by each thread is significantly reduced. This is the case in the experiments on the BG/Q architecture due to the larger numbers of threads involved in the overall computation and in the larger scale experiments on the Cray XC30.

The Reveal tool also employs the `default(done)` OpenMP directive on

each parallelised loop block and also specifies additional scoping information for each variable within this block, whilst the reference hybrid implementation only specifies the minimal amount of variable scoping information. To eliminate this as a factor causing the observed performance differences on the BG/Q architecture, the additional variable scoping information was manually added to the reference hybrid implementation, including the `default(none)` directive. Using this modified version an additional experiment was conducted on the Vulcan platform, the results of which are shown in Figure 4.22b. These demonstrate that the performance of this modified version (labelled "*ScopingInfo*") is identical to that of the original reference hybrid implementation at all of the experimental scales examined. This indicates, therefore, that the inclusion of the additional variable scoping information does not provide any performance benefits for the hybrid version produced by Reveal.

4.5 Summary

This chapter documented the research which was undertaken to improve the performance of the CloverLeaf mini-application at extreme scale (up to 131,072 processor cores) on several current state-of-the-art supercomputer architectures, and thereby to also improve the performance and scalability of the main applications which it represents. Several pieces of related work are identified and analysed first, and information is then provided on the actual implementations of the MPI-only and hybrid (MPI+OpenMP) versions of the CloverLeaf codebase. Additionally, each of the candidate optimisations, developed as part of this research, are also extensively documented.

A detailed analysis of the performance results, which were recorded during the experiments with these codebases, is presented in the results analysis section of this chapter. This analysis showed that selecting application data structures which are able to scale to large process counts without consuming significantly more memory resources is crucial in enabling applications to execute efficiently at scale. This research identified that, for CloverLeaf specifically, adopting a distributed approach for mesh meta-data management enabled the performance of the application to be significantly improved at scale and for significant memory savings to be achieved compared to the original implementation.

Of the candidate optimisations examined for both the MPI-only and hybrid codebases, the strategy of aggregating communication data into larger message sizes and delaying their transmission until all the data items are ready, was the most optimal approach for CloverLeaf. This approach achieved significant performance improvements over the reference implementation on both supercomputer architectures examined. The strategy of communicating data

as soon as it is ready for transmission, which was found to be beneficial by other researchers examining similar types of applications, actually resulted in significant reductions in performance when it was applied to CloverLeaf.

Fully utilising the available hardware threads on the BG/Q architecture was found to be beneficial for both the MPI-only and hybrid codebases, particularly in the smaller experimental scales examined. During these experiments the performance of the application is predominantly dominated by computational, rather than communication, operations. In contrast the use of the Intel Hyper-threads, on the Cray XC30 architecture, did not however affect performance in the smaller scale experiments and their use resulted in a substantial reduction in overall performance when the application was executed at scale. Similarly the use of huge-memory pages on the XC30 generally resulted in degradations in overall performance.

Utilising small message communications directly between logical diagonally neighbouring processes in order to reduce synchronisation operations within the application proved to be an inferior approach on both system architectures, compared to the approach employed in the reference implementation. In this version an implicit diagonal communication is achieved by exchanging data first in x-dimension of the mesh and then, following a synchronisation operation, in the y-dimension. The experimental results also show that when the performance of CloverLeaf is dominated by the time required for inter-process communication operations (e.g. when the application is executed at scale on the Cray XC30 platform), the pre-posting of MPI receive operations can deliver significant performance improvements.

The results presented here also show that the techniques which were developed as part of this research to overlap communication and computation operations within CloverLeaf, actually have a detrimental effect on the overall performance of the application on both supercomputer architectures examined. Additionally, the use of dedicated *Progress Threads*, which are available on the Cray architecture to potentially improve the overlap of computation and communication operations, do not significantly improve application performance, at least in these experiments. Executing these *Progress Threads* on additional hyper-threads also appears to be the most efficient approach compared with utilising a completely separate, dedicated compute core within each node.

On Archer employing the non-blocking reduction MPI v3.0 operations within CloverLeaf appears to provide some modest performance improvements for the application. The use of the neighbourhood collective operations, however, did not deliver any performance benefits in any of the experiments conducted. Similarly the candidate optimisation to consolidate the number of reduction operations within the application also did not provide any additional perfor-

mance benefits.

Reordering MPI ranks to improve the utilisation of shared memory communication resources and reduce the number of inter-node communication operations was shown to improve the performance of CloverLeaf. During the large scale experiments on the Cray XC30 platform, these performance improvements increased linearly with the size of the experiments. This approach represents a relatively straightforward mechanism with which to improve the performance of applications at scale, as it does not involve any changes to the source code of the application, and a suite of tools is available to rapidly generate the MPI rank mapping files.

Using these results to directly compare the two supercomputer architectures examined in this research shows that it is necessary to employ approximately 2-4× more processor sockets on the BG/Q architecture in order to achieve comparable performance to the Cray XC30 architecture. The experimental results, however, show that the BG/Q architecture can deliver superior performance, in terms of the energy required to achieve an equivalent solution. Additionally, the energy-to-solution profile of CloverLeaf on the Cray XC30 demonstrates an optimal job size with which to execute the application in order to minimise overall energy consumption.

The hybrid version of CloverLeaf initially delivered performance improvements at the smaller experimental scales examined on the Cray XC30 platform. The release of a later version of the Cray MPI communication layer, however, subsequently improved the performance of the MPI-only codebase to approximately match that of the hybrid versions. Additionally, on the BG/Q architecture and in the larger scale experiments on the Cray XC30, the MPI-only approach was always the most performant. The experimental results also show that the optimisations documented in Chapter 3 can deliver significant performance improvements for the hybrid versions of CloverLeaf when the application is executed across multiple nodes and performance is dominated by computation, rather than communication, operations.

The optimisation of combining OpenMP `parallel` regions higher up in the call-chain of the application was shown to consistently deliver significant performance improvements on both experimental platforms, particularly as the scales of the experiments were increased and OpenMP synchronisation overheads become a larger proportion of the overall runtime of the application. Additionally, changing the decomposition strategy within the hybrid version, to orientate the rectangular array sections in the vertical dimension, and thereby minimise inter-socket communication, was also shown to deliver some performance benefits at scale when OpenMP threading constructs were being utilised across multiple processor sockets on the Cray XC30.

Utilising the `ThreadMultiple` construct to enable MPI operations to be initiated in parallel by multiple OpenMP threads resulted in a significant reduction in performance on both architectural platforms. Similarly, employing OpenMP v4.0 thread placement constructs together with *Nested Parallelism* for the communication buffer packing operations also resulted in a substantial performance penalty in the experiments in which OpenMP threads were employed across multiple processor sockets.

This research also demonstrated that the Reveal tool from Cray can provide a viable solution for rapidly and automatically hybridising codebases. Furthermore, the performance of the automatically generated codebase is generally within ∼7% of the manually written version on the Cray architecture. On the BG/Q platform and in specific configurations on the Cray XC30, however, the automatically generated codebase is able to deliver superior performance compared to the manually developed versions.

CHAPTER 5
Evaluating the Utility of PGAS-based Approaches

This chapter documents work undertaken to assess whether PGAS-based programming models can deliver any performance advantages, particularly at large scale, for explicit Lagrangian-Eulerian hydrodynamics codes. Section 5.1 assesses existing work relating to this research area. The PGAS-based implementations of CloverLeaf which were developed as part of this work, are then documented in Sections 5.2 and 5.3. The results of several experiments, undertaken to assess the utility of these models against the de facto MPI approach, are presented in Section 5.4. Finally, Section 5.5 concludes the chapter.

5.1 Related Work

In the one-sided RDMA-based communication models utilised by PGAS languages, the communication initiator generally provides all relevant information regarding the operation. This alleviates the destination processor of any involvement, which has been recognised as an important factor in reducing communication latency [23]. It has also been argued that these models can potentially deliver additional benefits over standard message passing solutions, including, eliminating message matching and synchronisation overheads, improving energy consumption through reductions in data-motion, relaxing message ordering guarantees and reducing memory consumption by removing communications buffers [71, 23]. Minimising communication operations within applications has been recognised as a key approach for improving the scalability and performance of scientific applications [51].

Background information on the OpenSHMEM and CAF programming models can be found in Section 2.2.8. Additionally, although CAF has only relatively recently been incorporated into the official Fortran standard, earlier versions of the technology have existed for some time. Researchers are also actively seeking to further improve the existing standard with proposed changes to the programming model and communication constructs [140]. Similarly, although several distinct SHMEM implementations have existed since the model was originally developed by Cray in 1993 for the T3D supercomputer architecture [81], the technology has only been officially standardised very recently as part of the OpenSHMEM initiative [157, 40].

Consequently, a number of studies have already examined these technologies. These have generally focused, however, on different scientific domains to the

one examined in this research, and on applications which implement alternative 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 examined in this work. Overall, substantially less work exists which directly evaluates the MPI, OpenSHMEM and CAF programming models when applied to the same application. The results from previous studies have also varied significantly, with some authors achieving significant speedups by employing PGAS-based constructs, whilst others present performance degradations.

Several studies which do directly evaluate particular PGAS and MPI programming models at considerable scale are from Preissl [166], Mozdzynski [143] and Shan [178]. Preissl *et al.* present work which demonstrates a CAF-based implementation of a Gyrokinetic Tokamak simulation code delivering significantly improved performance compared to 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. Whilst Shan demonstrates CAF and UPC versions of the IMPACT-T and MILC applications significantly outperforming equivalent MPI versions. Additionally, a UPC version of the NAS FT benchmark has also been shown to significantly outperform an equivalent MPI implementation [23, 147].

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 they experienced a significant performance degradation [186]. Their work examined on the CGPOP mini-application, which represents the Parallel Ocean Program [107] 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 [120]; 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 micro-benchmarks [80].

Using "*lower-level*" one-sided communication APIs has been shown to deliver performance improvements for parallel applications which send large numbers of small messages [22]. OpenSHMEM delivered some performance advantages relative to MPI for Bethune *et al.*, however, they examined the Jacobi method for solving a system of linear equations and utilised a previous generation of the Cray architecture (XE6) in their experiments [26]. In [172] Reyes *et al.* discuss their experiences porting the GROMACS molecular dynamics application to OpenSHMEM. Their experiments show consistent performance degradations (up to $\sim$12% in particular experiments) relative to the original MPI implemen-

tation, additionally they also utilised the Cray XE6 architecture.

Baker *et al.* examined a hybrid approach using OpenACC within a SHMEM-based application; however, they concentrated primarily on hybridising the application and their results were collected on the Cray XK7 architecture (Titan) [17]. A comparison of the use of one-sided MPI, UPC and SHMEM communication constructs within a distributed hash table application on the Cray XE6 architecture is provided by Maynard [138]. In [109] Jose *et al.* also studied the implementation of a high performance unified communication library that supports both the OpenSHMEM and MPI programming models on the Infiniband architecture.

5.2 SHMEM Implementation

The OpenSHMEM-based versions of CloverLeaf created as part of this research utilise one of two general communication strategies. These involve employing the OpenSHMEM communication constructs to exchange data:

1. Between 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 contiguously within multi-dimensional arrays `shmem_put64` operations are used, whilst strided `shmem_iput64` operations are utilised to transmit data which is stored non-contiguously. On the platforms examined in this research it is necessary to employ two separate calls to the `shmem_iput64` operation in order to transmit two columns of *halo* data rather than one call to the `shmem_iput128` operation.

In Section 5.4 versions which employ the first strategy are denoted by the word *buffers* in their descriptions, whereas versions which employ the second are referred to as *arrays*. Additional versions were also created as part of this research which utilise the proprietary Cray non-blocking SHMEM *"put"* operations; within Section 5.4 these are referred to using the suffix *nb* (non-blocking). 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 appropriately accessible.

The only synchronisation primitive which OpenSHMEM provides natively is a global operation (`shmem_barrier_all`) which synchronises all of the processes involved. Versions developed as part of this research which employ this synchronisation strategy are denoted by the word *global* within their descriptions in Section 5.4. All other versions employ a *point-to-point* synchronisation strategy in which processes only synchronise with their immediate neighbours. Integer *"flag"* variables, which are set on a remote process after the original communication operation completes, are employed to achieve this. To ensure the correct ordering of remote memory operations either `shmem_fence` or `shmem_quiet` operations are utilised. Versions which employ `shmem_quiet` contain the word *quiet* within their descriptions in Section 5.4; 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. Several versions employ a call to `shmem_int4_wait_until` using the particular *"flag"* variable, these are referred to using *shmemwait* within their description in Section 5.4. Alternative versions utilise an approach in which the *"flag"* variables are explicitly declared as `volatile` and processes perform "*busy waits*" until their values are set remotely by the initiating process. Versions which employ this latter strategy are denoted by the word *volatilevars* within their descriptions in Section 5.4.

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

5.3 CAF Implementation

The CAF-based implementations of CloverLeaf created as part of this research all utilise one-sided asynchronous CAF *"put"* operations. The image responsible for particular *halo* data, remotely writes this 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 within CloverLeaf (a *structure of arrays* based construct), which contains all data-fields and communication buffers, is declared as a *co-array* object. Additional versions were, however, created to examine the effect of moving the

data-fields and communication buffers contained within this *derived-type* outside of this data structure and declaring them as individual *co-array* objects. Within Section 5.4 of this chapter, versions which employed this modified approach are denoted by the acronym *FTL* within their descriptions.

All of the versions employed in this study utilise the same general communication strategies as the OpenSHMEM implementations, which were outlined in Section 5.2. Again code variants which employ the communication buffer based strategy contain the word *buffers* within their descriptions in Section 5.4, whereas versions which employ the direct memory access strategy are denoted by 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 systems 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 *point-to-point* `sync images` construct between immediate neighbouring processes. The selection between these synchronisation primitives is controlled by compile-time pre-processor directives. Versions employing both the direct memory access data exchange strategy (referred to as *arrays*) and the `sync images` synchronisation construct require the inclusion of an additional synchronisation operation between logical diagonally neighbouring images. In Section 5.4, versions which employ the global synchronisation construct contain the word *"global"* within their descriptions; all other versions utilise the alternative *point-to-point* synchronisation construct.

Versions which explicitly attempt to overlap communication and computation operations, using the PGAS constructs together with the approach outlined in Section 4.2.1, were also developed as part of this research. Within Section 5.4 these implementations are denoted by the word *overlap* within their descriptions. Additional versions which utilise the proprietary Cray `pgas defer_sync` directive were also developed; these can be identified by the word *defer* within their descriptions in Section 5.4. This directive purports to ensure that the synchronisation of PGAS operations is delayed until as late as possible, typically the next fence instruction [45].

The CAF versions examined as part of this research each employ the proprietary Cray *collective* operations to implement the required global reduction operations. Alternative hybrid versions, which utilise MPI collective operations, were also developed in order to ensure the portability of these codebases to additional CAF runtime implementations. This thesis, however, only reports on the performance of the purely CAF-based versions in order to provide a direct comparison between the CAF and MPI programming models; additionally only

the Cray architecture is examined during experiments involving the CAF-based versions of CloverLeaf.

5.4 Results Analysis

To assess whether the OpenSHMEM and CAF programming models can improve (reduce) the overall time-to-solution of explicit hydrodynamics applications, a series of experiments were undertaken. The performance of the PGAS-based versions of CloverLeaf, were examined on two distinct hardware platforms with significantly different architectures, a Cray XC30 (Archer) and an SGI ICE-X (Spruce). These machine architectures were selected for these experiments as they each contain state-of-the-art technology and also both provide native support for PGAS programming models within the vendor supplied system software. The hardware and system software configuration of these machines is detailed in Appendix A.1. Additionally version 8.2.2 of the Cray CCE compiler and version 6.3.0 of the Cray Mpich2 and Shmem communication libraries were utilised in these experiments. The $15{,}360^2$ cell problem, which is a standard configuration from the CloverLeaf benchmarking suite, was simulated in these experiments and was executed for 2,955 timesteps (see Section 1.6.1 for more details). This was strong-scaled to large processor counts on both architectures, in order to stress the inter-process communication infrastructure provided by each programming model.

These experiments were conducted in two phases, with the second set of experiments conducted specifically to further explore particular observations which were made during the results analysis of the first set of experiments. The results produced from both sets of experiments are presented in Sections 5.4.1 and 5.4.2 and examine the effect of employing each programming model on the runtime of the application. For clarity the results presented here (Figures 5.1 to 5.6) are expressed in terms of the number of nodes on which an experiment was conducted, and the rate of application iterations / second which the particular version achieved (i.e. 2,955 iterations / application wall-time). In order to reduce the effects of system noise and jitter, unless otherwise noted the presented results are averages of three repeated executions of each experiment.

To eliminate any performance effects due to different topological allocations from the batch system, each version was executed within the same node allocation, for each specific job size which was examined. The experiments which utilised the Spruce platform were also conducted with the system in a fully dedicated mode, which should significantly reduce the effects of any system noise on the recorded results. Unfortunately this was not possible on Archer, and therefore no direct comparisons are provided in this thesis between the

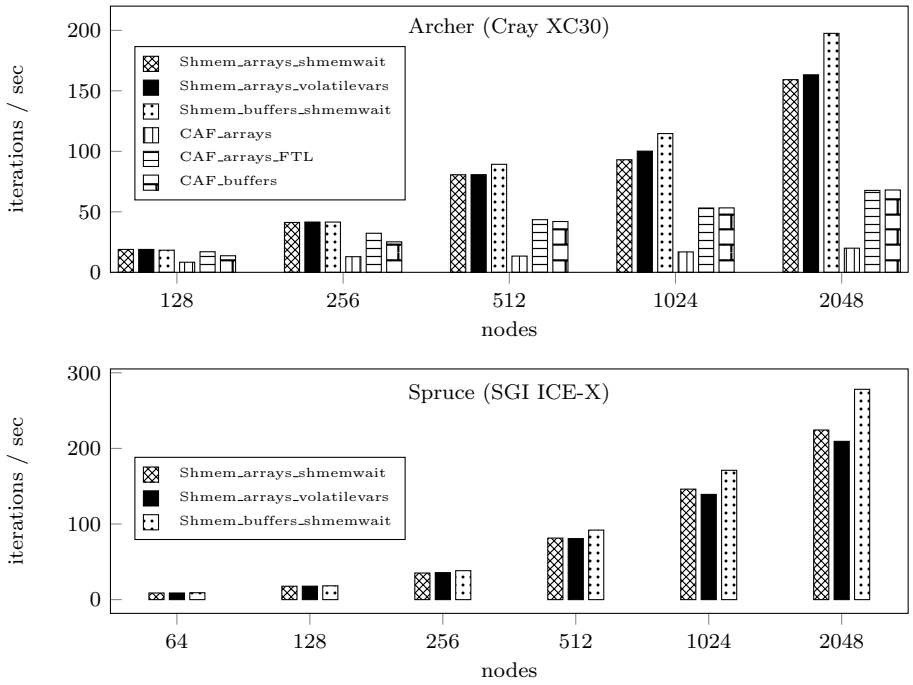


Figure 5.1: PGAS implementations: Array- and buffer-exchange versions

performance of the two system architectures. In these experiments each version was also configured to utilise enhanced IEEE precision support for floating point mathematics operations, available under the particular compilation environment employed on each platform. On Archer all PGAS versions were also built and executed with support for 2MB huge memory pages enabled and 512MB of symmetric heap space available. Huge page support was not enabled for the standard MPI versions, as previous work did not observe these features delivering any performance benefits for these implementations [132].

5.4.1 First Strong-scaling Experiment Results Analysis

This section analyses the performance results obtained during the first phase of the PGAS experiments.

Communications Buffer & Array-sections Approaches

The results from the experiments with the PGAS versions, which employ either the communications buffer or array-sections data exchange approaches, are shown in Figure 5.1. These charts show the positive effect which employing communications buffers can have, particularly at high node counts, on both the Spruce and Archer platforms. In the 2,048 node experiments on the Spruce plat-

form 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 2,048 nodes (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, with the communication buffers approach achieving 3.4× the performance of the array-section based approach, the results show that these achieved 68.04 and 19.95 iterations/sec respectively in these experiments.

This demonstrates that the performance of applications, implemented within either the OpenSHMEM or CAF PGAS models, can be significantly improved through the aggregation of communication data into larger transmission buffers, rather than moving data directly from its original memory locations using considerably larger volumes of smaller messages and potentially strided memory operations.

Co-array Object Selection Options

These results also show (Figure 5.1) the performance improvement delivered by moving the data field definitions from within the original Fortran *derived* data type, which was originally defined as a *co-array*, to be individual top-level data structures, each separately defined as *co-array* objects. 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 2,048 nodes on Archer. It also enabled the array-section based approach to deliver equivalent performance to the communications-buffer based approach in the 1,024 and 2,048 node experiments, and to slightly exceed it in the 64 to 512 nodes cases.

To ascertain the cause of this performance disparity a detailed inspection of the intermediate code representations, produced by the Cray compiler, was conducted. This indicated that this disparity is due to the compiler having to make conservative assumptions regarding the calculation of the remote addresses of the *co-array* objects on the remote images. For each remote *"put"* operation within the *FTL* version of the code, 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, with a further set of these operations within the actual loop

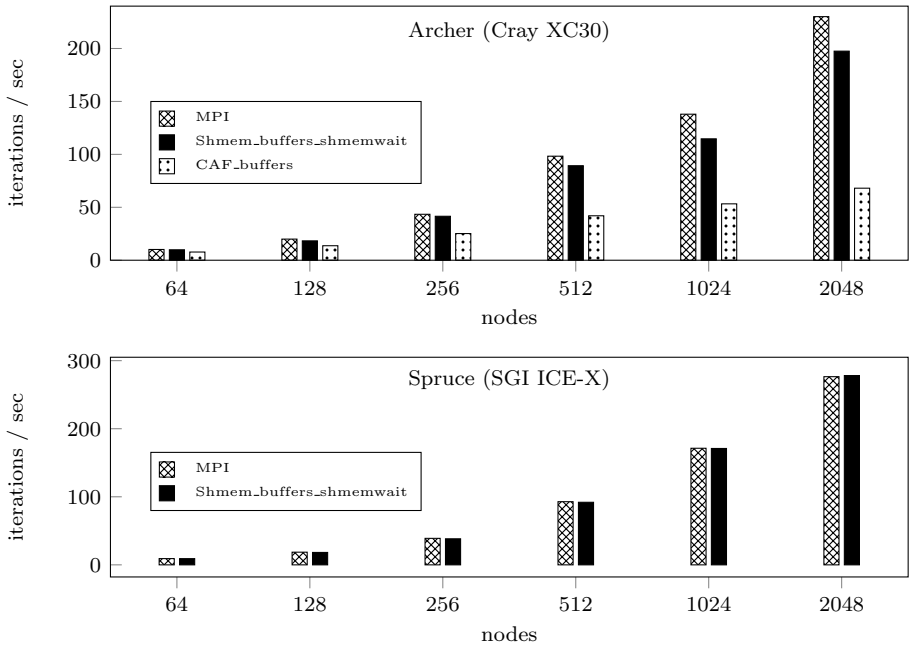


Figure 5.2: Equivalent MPI, OpenSHMEM and CAF performance

and an additional nested loop block containing a `__pgas_put_nbi` operation.

Unfortunately the precise functionality of each of these operations is not clear, as Cray does not publish this information. This analysis, however, appears to indicate that the compiler is forced to insert additional *"get"* operations due to the extra complexity (i.e. the additional levels of indirection involved) of the original data structures. These additional operations are required to retrieve the memory addresses from the remote images, to which a particular image should write the required data to, despite these addresses remaining constant throughout the execution of the program. The creation of an additional compiler directive may therefore prove to be useful here, as it would enable developers to inform the compiler that the original data structure remains constant, and therefore allow it to make less conservative decisions during code generation.

PGAS and MPI Performance Comparison

Figure 5.2 presents the results from the experiments conducted to assess the performance of the PGAS implementations relative to equivalent MPI-based versions. These charts document a significantly different performance trend on the two system architectures examined here. The performance recorded on Spruce from both the OpenSHMEM and MPI implementations is virtually identical at all the node counts examined (64 to 2,048 nodes), reaching 278.14 and 276.49 iterations/sec respectively on 2,048 nodes. On Archer, however, the

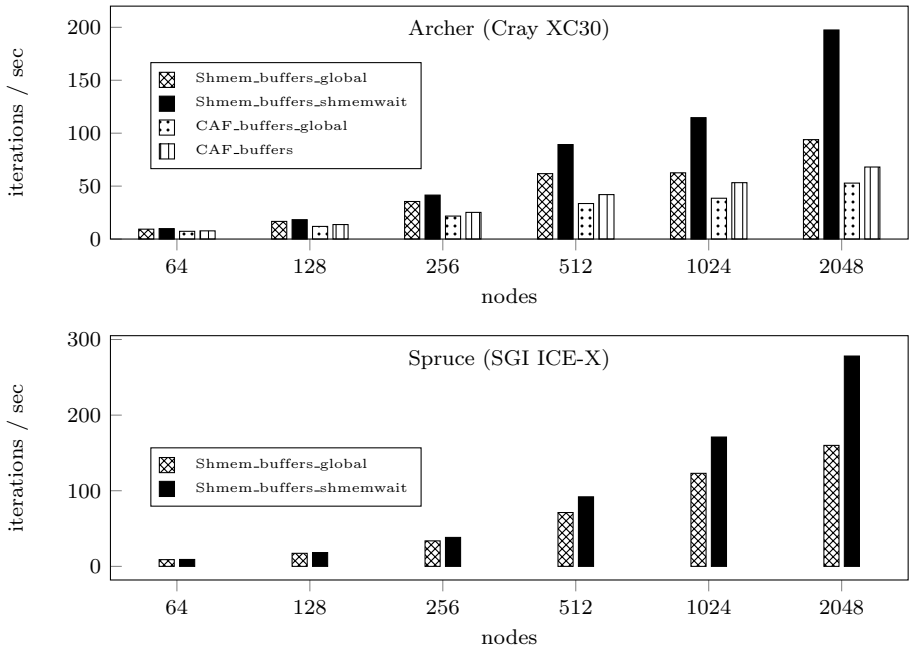


Figure 5.3: Local & global synchronisation approaches

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. The OpenSHMEM implementation delivers the closest levels of performance to the MPI implementation and also significantly outperforms the CAF-based implementation. The results show it achieving 197.49 iterations/sec on 2,048 nodes compared to 230.08 iterations/sec for the MPI implementation, an improvement of 1.17×. The CAF implementation, however, only delivers 68.04 iterations/sec on 2,048 nodes a slowdown of 2.9× relative to the equivalent OpenSHMEM implementation.

Synchronisation Approaches

To assess the effect of employing either the *global* or *point-to-point* synchronisation constructs on the performance of the PGAS versions, the results obtained from experiments on both platforms involving versions which employed the communications buffer data exchange approach together with either synchronisation construct, were analysed. The OpenSHMEM versions examined here utilised the *shmemwait* approach to implement the *point-to-point* synchronisation operations. Figure 5.3 provides a performance comparison of the results obtained from the experiments with 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

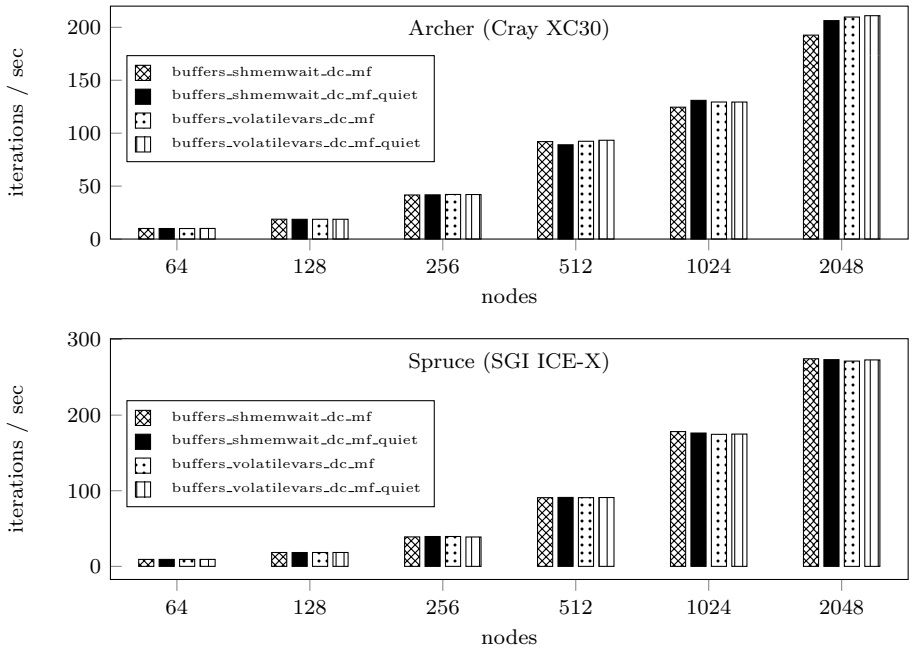


Figure 5.4: SHMEM volatile variables & fence/quiet optimisations

experiments is increased. At 64 nodes there is relatively little difference between the performance of each version. On Spruce (1,280 cores) both OpenSHMEM implementations achieve 9.13 and 8.90 iterations/sec respectively, 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 each achieve 9.34 and 7.31 iterations/sec respectively. At 2,048 nodes (40,960 cores) on Spruce the performance disparity between the two OpenSHMEM versions increases to 278.13 and 159.97 iterations/sec respectively, a difference of approximately 1.74×. On Archer, however, the performance disparity between the OpenSHMEM versions is even greater reaching 2.10× in the 2,048 node (49,152 cores) experiments, 197.49 and 93.91 iterations/sec were recorded respectively.

Interestingly 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). This indicates that the performance of the CAF-based versions—which is significantly less than the OpenSHMEM-based versions—is limited by another factor and therefore the choice of synchronisation construct has a reduced, but still significant, effect on overall application performance.

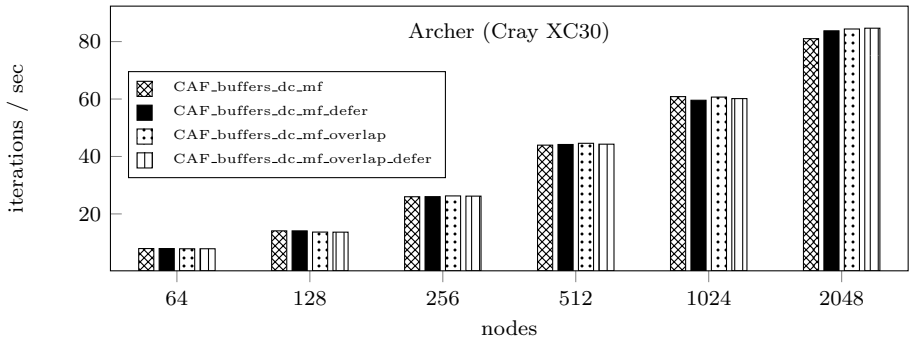


Figure 5.5: CAF `pgas defer_sync` construct & communication overlap

Remote Memory Operation Ordering Constructs

The performance results obtained from several alternative versions of the OpenSHMEM implementation, on both the Cray and SGI platforms, are presented in Figure 5.4. These charts compare versions which employ either the *shmemwait* or *volatile variables* synchronisation techniques and either the *quiet* or *fence* remote memory operation ordering constructs. All versions examined in this chart employ a communications buffer-based approach to data exchange, as well as implementing diagonal communications (Section 4.2.1) between logical neighbouring processes (denoted by the acronym *dc* within their descriptions). They also exchange multiple data fields simultaneously (Section 4.2.1), which is indicated by the acronym *mf* within their descriptions. It is evident from the charts that in these 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 in the higher node count experiments, this is likely to be due to the effects of system noise arising from the use of a non-dedicated system.

Figure 5.5 documents the results obtained on the Archer platform from experiments with the CAF versions which employ the proprietary Cray `pgas defer_sync` directives and the optimisations to enable communication operations to be overlapped with computation. The chart presents these results together with an equivalent CAF-based version which does not utilise any of these constructs. This shows that in these experiments the overall performance of CloverLeaf is not significantly affected (beneficially or detrimentally) by either of these potential optimisations techniques, as the performance of all four versions is virtually identical in each of the examined cases.

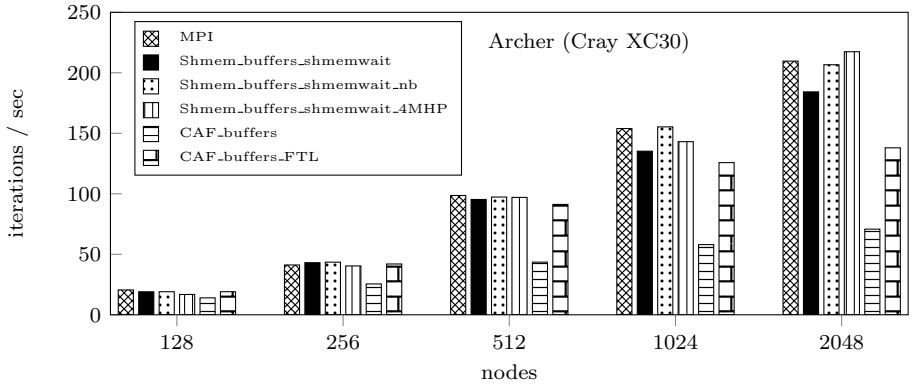


Figure 5.6: SHMEM non-blocking, huge-pages & CAF FTL

5.4.2 Second Strong-scaling Experiment Results Analysis

Following the results analysis documented in Section 5.4.1, an additional set of experiments was conducted on Archer. The aim of these additional experiments was to examine the effect of: the proprietary Cray non-blocking SHMEM operations; employing 4MB huge-pages; and applying the *FTL* optimisation (Section 5.3) to the CAF buffer-exchange based version. The same experimental methodology, previously outlined in Section 5.4, was followed and the results of the experiments are presented in Figure 5.6. As these experiments were conducted at a different time (different system loads) and using different node allocations from the batch system, compared to the first set of experiments, the performance results between the two sets of experiments will differ, particularly at scale. This thesis therefore only presents performance comparisons within each set of experimental results rather than between them.

FTL Optimisation Technique

It is evident from Figure 5.6 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 1,024 and 2,048 node experiments, respectively. Although significantly improved, its performance still does not quite match that of the equivalent OpenSHMEM-based version particularly at large node counts. In the 2,048 node experiment the OpenSHMEM buffer-exchange version achieved 184.23 iterations/sec compared to 138.01 for the CAF-based *FTL* version, an improvement of 1.33×. As in the initial set of experiments, the original OpenSHMEM version is not able to match the performance of the equivalent MPI implementation. It achieved 135.21 and 184.23 iterations/sec in the 1,024 and 2,048 node experiments respectively, compared to 153.92 and 209.65 for the MPI version.

Non-blocking SHMEM Operations

The use of the proprietary Cray non-blocking operations, however, delivers some further performance benefits for the OpenSHMEM-based versions, particularly at high node counts. The performance of the version which utilises these non-blocking operations is virtually identical to that of the original in the experiments $\leq$256 nodes. At 512 nodes and above, however, it starts to deliver significant performance advantages, achieving 206.66 iterations/sec in the 2,048 node experiment, compared to only 184.23 for the original version. In both the 1,024 and 2,048 node experiments it also delivered broadly equivalent performance to the MPI implementation, achieving 155.31 and 206.66 iterations/sec respectively, compared to 153.92 and 209.65 for the MPI version.

This demonstrates that the use of the proprietary non-blocking operations can deliver some significant performance improvements for this class of applications, by reducing the overheads associated with inter-process message communication and enabling sequences of messages to be more rapidly injected into the network. The OpenSHMEM standard would therefore benefit from the standardisation of these constructs within future versions of the specification.

Utilisation of Huge Memory Pages

The performance benefits observed from employing the larger 4MB huge memory pages are even more significant. In the 2,048 node experiment the version which utilised these larger page sizes 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 was fractionally worse than the original OpenSHMEM version in all of the experiments below 1,024 nodes.

5.5 Summary

The research presented within this chapter examined the PGAS based programming models of OpenSHMEM and CAF as potential candidate technologies for delivering performance advantages, on current and future system architectures, for the explicit hydrodynamics applications which CloverLeaf represents. Related work in the field was documented together with the implementation of multiple CAF- and OpenSHMEM-based versions which were developed as part of this work. The performance of each programming model is evaluated and compared to an equivalent MPI-based implementation, at considerable scale (up to 2,048 nodes/49,152 cores) on two significantly different, whilst still state-of-the-art, system architectures from two leading vendors.

The recorded results 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 compared to MPI on the Cray XC30 system architecture. Use of the proprietary Cray non-blocking operations, however, enabled the performance of the SHMEM-based versions to match and sometimes exceed that of their MPI equivalents. Additionally, the library-based PGAS model of OpenSHMEM can be significantly more performant than equivalent language/compiler-based PGAS approaches such as CAF on the Cray XC30.

Applications based on either PGAS paradigm can also benefit, in terms of improved application performance, from the aggregation of data into communication buffers. This enables the required data to be collectively communicated to the remote processes, rather than moving it via strided memory operations. The performance of CAF-based applications was also shown to 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.

This research also demonstrated that performance improvements can be achieved, for both OpenSHMEM- and CAF-based applications, by employing *point-to-point* synchronisation mechanisms rather than *global* synchronisation primitives. Furthermore, the selection of implementation mechanisms for the *point-to-point* synchronisation operations (*shmemwait* or *volatile variables*), and the choice of the remote memory operation ordering constructs (*fence* and *quiet*), was shown to not significantly affect the overall performance of this class of application. Similarly, the use of the proprietary Cray CAF `pgas defer_sync` constructs and the optimisations to overlap communications and computation also do not significantly affect overall application performance.

CHAPTER 6
Portable Performance Through OpenCL

This chapter documents the work undertaken to assess the utility of OpenCL for delivering portable performance for hydrodynamics applications. In particular it examines the ability of OpenCL to express intra-node parallelism and implement a hybrid programming model which enables multiple novel processing architectures (e.g. GPGPUs) to be utilised for this class of application. Related work within this research arena is first discussed within Section 6.1. Following this the actual OpenCL implementation of CloverLeaf, produced as part of this research, is documented in Section 6.2 together with several optimisations which have been implemented within the codebase (Section 6.2.2). Results from both small (single processor) and large scale experiments are then analysed in Section 6.3. Finally, Section 6.4 summaries the findings of this research and concludes the chapter.

6.1 Related Work

Insufficient work has, to date, been undertaken to examine whether OpenCL is a viable alternative programming model for delivering intra-node parallelism on HPC system architectures, particularly for Lagrangian-Eulerian explicit hydrodynamics applications. This includes examining whether OpenCL runtime systems are now able to automatically optimise a single source-code for different platforms in order to achieve portable performance for these hydrocodes, or whether device specific optimisations are still required.

A considerable body of work has, however, examined porting smoothed particle hydrodynamics (SPH) applications to GPU-based systems [68, 47, 164, 174]. These applications generally employ mesh-less, particle based numerical methods and are therefore significantly different to the hydrodynamics scheme simulated within CloverLeaf. Existing studies have also predominantly focused on utilising CUDA and have not sought to examine OpenCL as an alternative technology for delivering portable performance.

Bergen *et al.* developed an OpenCL version of a finite-volume hydrodynamics application which is similar to CloverLeaf [25]. They do not, however, present any performance results or compare the development, performance or portability of the application to alternative approaches or across architectures. The GAMER library also provides similar functionality, however, it is implemented entirely in CUDA and therefore does not allow OpenCL to be evaluated as an

alternative approach [182]. Additionally, Brook *et al.* present their experiences porting two computational fluid dynamics (CFD) applications to an accelerator [32]. Whilst their Euler-based solver has similar properties to CloverLeaf, they focus exclusively on the Intel Xeon Phi architecture and employ only the OpenMP programming model.

Existing work has examined using OpenCL to deliver portable performance within other scientific domains. Pennycook presents details of the development of OpenCL implementations of the NAS LU benchmark [159] and a molecular dynamics application [160], which achieve portable performance across a range of current architectures. Similarly, Brown *et al.* describe work and performance results, for both OpenCL and CUDA, within the molecular dynamics domain which enables computational work to be dynamically distributed across both CPU and GPU architectures [34]. Du [60] and Weber [205] also provide direct analyses of OpenCL's ability to deliver portable performance for applications targeting accelerator devices; however, both focus on different scientific domains. Additionally, Komatsu [116] and Fang [66] provide a detailed examination of the performance of CUDA and OpenCL, as well as the performance portability of both programming models. Van der Sanden also evaluates the performance portability of several image processing applications expressed in OpenCL [198].

The majority of existing work also focuses on accelerator devices; consequently there is considerable uncertainty regarding how to optimise OpenCL codebases for CPU devices. Several techniques for improving performance on CPU architectures are, however, presented in [112]. Lan *et al.* also document several techniques for improving the performance of GPU-focused OpenCL *kernels* on CPUs [118]. Additionally, Seo *et al.* examine how optimised versions of the NAS parallel benchmarks should be expressed in OpenCL for both CPU and GPU architectures [177].

OpenACC [155] has recently emerged as a new, *directive*-based, programming model for porting applications to accelerator devices. Consequently insufficient work has thus far been conducted to assess the utility of OpenCL-based approaches relative to this model, however, Wienke *et al.* do provide one direct comparison [207]. Although little work exists which has examined using OpenCL to scale this class of application to the levels examined in this research, Levesque *et al.* used OpenACC at extreme scale within the S3D application [122].

Existing studies have examined utilising OpenCL together with MPI to deliver portable performance [163, 189]; however, these studies have generally focused on applications from different scientific domains. Additionally, Kim *et al.* propose a novel framework which enables OpenCL applications to be executed in a distributed manner [113].

Auto-tuning has also been recognised as a key technology for enabling scientific applications to be rapidly ported to, and achieve optimal performance on, new computational platforms. In [56] Dolbeau *et al.* examined using OpenCL as a target software layer for an OpenACC compiler, as well as employing an auto-tuning strategy to achieve optimal performance on a range of processor technologies. Rahman *et al.* developed an auto-tuning framework with the ability to optimise for both performance and energy efficiency, it supports a broad range of code optimisation techniques, and they demonstrate it on several commonly used scientific *kernels* [169]. The tuning of thread counts and loop tiling parameters was also shown to deliver significant performance improvements by improving cache utilisation by Jordan *et al.* [108]. Similarly, Kamil *et al.* examined applying an auto-tuning strategy to a range of stencil-based codes to achieve portable performance across several different processor architectures [111]. Additionally, Zhang *et al.* examined auto-tuning stencil computations on GPU architectures, although their work focused on the iterative Jacobi method and the CUDA programming model [210].

6.2 OpenCL Implementation

To create the OpenCL implementation of CloverLeaf, new OpenCL-specific versions of each of the existing *kernel* functions were developed. The implementation of each of these was separated into two distinct parts:

(i). OpenCL *device*-side *kernels* which perform the required operations

(ii). *Host*-side C++ based routines used to setup and control the OpenCL runtime environment

The existing Fortran driver routines were reconfigured to execute the C++ routines. These utilise the OpenCL C++ bindings to transfer any required data to the target computational devices, set *kernel* arguments and add the *device*-side *kernels* to the *work-queues* with the appropriate `NDRange` dimensions.

Since each *kernel* performs a well defined mathematical function, and the Fortran versions avoid the use of complex language features, it was possible to almost directly translate each *kernel* into an equivalent OpenCL specification. Fortran intrinsic operations (such as `SIGN or MAX`) were all replaced with the corresponding OpenCL built-in function to ensure optimal performance. To finalise the OpenCL *kernels*, however, several additional changes were required to produce the initial implementation (Figure 6.1b). The loops over the staggered grid were re-factored such that the actual loop constructs were completely removed from the individual *kernels*. Instead the application was configured

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

} catch(cl::Error err) { CloverCL::reportError(err, ...); }

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

(a) The OpenCL C++ host side code for the `Ideal_gas` kernel.

```
__kernel void ideal_gas_ocl_kernel(const int x_min, const int x_max,
                                   const int y_min, const int y_max,
                                   __global double *d, __global double *e,
                                   __global double *p, __global double *ss)
{
    double ss2,v,pe,pv;

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

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

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

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

(b) The OpenCL device code for the `Ideal_gas` kernel.

Figure 6.1: Components of the OpenCL version of the `Ideal_gas` kernel

to launch these *kernels* with the required index space. As a consequence of employing the OpenCL launch mechanisms in this manner, only one *work-item* is launched for each mesh point. Each *work-item* therefore only processes one mesh cell which ensures that *buffer* objects are not accessed beyond their bounds. In order to produce comparable results to the Fortran *kernels*, all computation is also performed in double precision.

The C++ setup routines each rely on a static class, `CloverCL`, which provides common functionality. To reduce redundant computation all initialisation logic was removed from the actual *kernel* functions and placed within this static class. This helped to ensure that particular operations (e.g. the *kernel* `setArg` commands) were only re-executed when absolutely necessary thus improving overall performance. The static class also contains other methods that provide an additional layer of abstraction around common OpenCL routines.

The required OpenCL *buffers* and *kernels* are created, stored and managed from within this class, which allows *buffers* to be shared between different *kernels*. This *buffer* sharing was particularly important in maximising performance across different architectures. It also enabled the implementation to achieve full

device *residency* on architectures constructed from accelerator based devices (e.g. GPGPUs) which are generally attached via a PCIe bus to the main system nodes. Achieving full device *residency* and thus minimising data movement across the relatively slow PCIe bus was crucial in achieving high performance on many current architectures.

The use of OpenCL `wait` operations was also minimised in the initial implementation via the use of a single *in-order* work-queue and global event objects, which were also stored within the static class (`CloverCL`). This enables a dependency chain to be established between the *kernel* invocations within each timestep of the algorithm. The overall approach thus proceeds such that *kernels* are continually added to the *work-queue* in the order in which they are required to be executed, with the *in-order* properties providing the necessary synchronisation between the various invocations.

The majority of the control code within the original Fortran *kernels* was also moved into the C++ setup routines (Figure 6.1a). This ensures that branching is always performed on the *host* instead of on any associated *devices*, enabling the *device*-side *kernels* to avoid stalls and thus maintain higher levels of performance.

To enable the implementation to be utilised across the nodes of a distributed memory cluster the initial OpenCL implementation was combined with MPI communication constructs. The former was employed to deliver the intra-node parallelism required by the application and the latter for all inter-node parallelism. In the initial integration between the OpenCL and MPI constructs within CloverLeaf the OpenCL built-in function `clEnqueueReadBufferRect`, was utilised to read back only the minimum amount of required data from the *device*-side OpenCL buffers, directly into the *host*-side MPI communication buffers. The original data ordering within the MPI communication buffers was also altered to better integrate with the `clEnqueueReadBufferRect` function. This eliminated the requirement to explicitly manage the communication buffers on the target *device* using separate OpenCL *kernels* and potentially makes use of optimised OpenCL built-in functions. Similarly, the OpenCL `clEnqueueWriteBufferRect` function was also employed for transferring data back to the OpenCL *device*-side buffers following an MPI communication operation.

6.2.1 Reduction Operators

Reduction operations are required by the algorithm in two locations, for the calculation of the minimum timestep and the generation of intermediate results. Since the timestep value is calculated frequently, it is crucial that a high perfor-

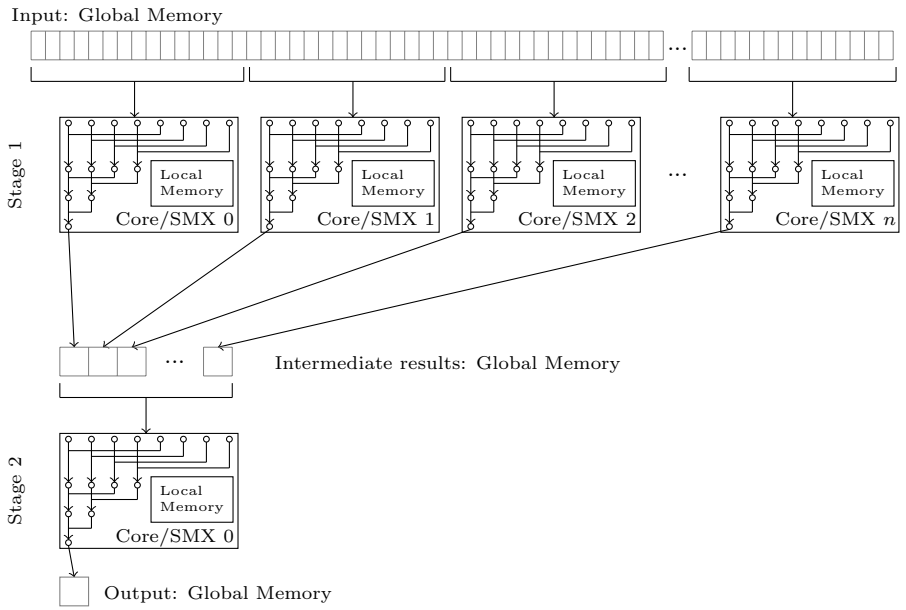


Figure 6.2: OpenCL Reduction Implmentation for GPUs

mance reduction implementation is utilised. As a general optimised reduction operator written in OpenCL is not, at present, readily available an optimised reduction function was developed as part of this research.

Due to the architectural differences between CPU and GPU devices, separate OpenCL reduction functions were developed, and specifically optimised, for each particular architecture. These were implemented as separate OpenCL *kernels* and their operation differs significantly from their Fortran and C based equivalents, which either use nested loops to iterate over the entire source array, or OpenMP reduction primitives. Whilst the performance of these *kernels* is not portable across architectures it makes sense to specialise them, as reduction operations are fundamental to scientific applications and the *kernels* can be reused within other applications. Ultimately, reduction operations should be provided by a library, and therefore specialising these *kernels* should not affect the portability of the actual application code.

GPU Reduction Kernel

The reduction *kernel* that targets GPU devices (Figure 6.2) is based on work presented by Harris, although his method is generalised as part of this research to handle arbitrary sized arrays [75]. A multi-level tree-based approach is employed in which *kernel* launches are used as synchronisation points between different levels of the tree. The tree continues until the input to a particular

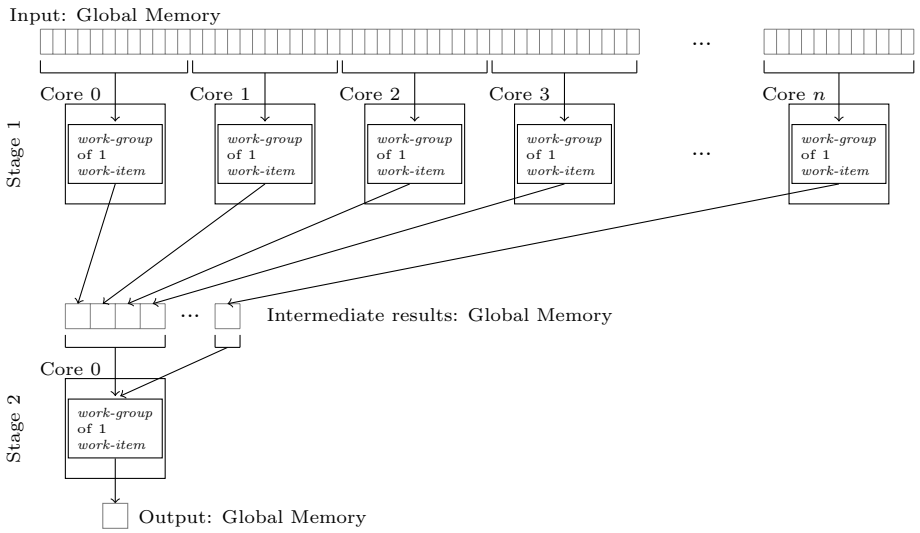


Figure 6.3: OpenCL Reduction Implmentation for CPUs

level is small enough to fit within one *work-group* on the current target *device*. In the final level a single *work-group* is launched on one *compute unit* of the associated *device*, which subsequently calculates the final result. At each stage *work-items* initially read two values from *global* memory, apply the binary reduction operator to them and store the result within *local* memory. To enable memory operations to be coalesced and to ensure efficient bandwidth utilisation, these *global* memory operations are aligned to the preferred vector width of the *device*.

A tree-based reduction is then initiated on the partial results stored within the *local* memories. In this phase the number of active threads is halved in each successive iteration, until all of the partial results have been reduced to a single value. To ensure efficient bandwidth utilisation, the *local* memory references are also arranged to avoid memory bank conflicts. The derived single value is then written by one thread back to *global* memory for the next level of the reduction tree to operate on.

To reduce the number of levels within the tree (and thus the number of *kernel* launches) the number of *work-items* launched within each particular *work-group* is maximised. Thus, for each *work-group*, the number of input values read from *global* memory into the *local* memories is also maximised, relative to the single value written back to *global* memory. The implementation ensures that the number of *work-items* launched for the reduction *kernels* is always a power of 2, and an exact multiple of the preferred vector width of the *device*. This generalises to handle arbitrary sized arrays by limiting, if required,

the number of data values read from *global* memory by the final initiated *work-group*. Instead, *work-items* beyond this limit insert dummy values into their corresponding local memory locations, ensuring that the tree-based part of the reduction is always balanced.

CPU Reduction Kernel

The reduction *kernel* that targets CPU devices (Figure 6.3) operates in a similar manner using a two-level hierarchical approach, in which *kernel* launches are used to provide synchronisation between the levels. In the first level, the input array is partitioned such that it is distributed as evenly as possible across all of the available CPU cores. If required, the last *work-group* is again limited to handle uneven distributions of arbitrary sized arrays. Only one *work-item* is launched for each core of the associated CPU and all *work-groups* contain only one *work-item*. Each *work-item* then sequentially reduces the data values within the portion of the input array which is assigned to it, and stores the resultant value back into *global* memory. The number of partial results output from this phase is therefore equal to the number of cores available on the CPU device.

In the second stage of the reduction only one *work-item* is launched on one core of the associated CPU. This *work-item* operates on the array of partial results produced from the previous stage, reducing them sequentially, before outputting the final result. No *local* memory constructs are employed at any stage of this implementation, as these are generally mapped to the same memory address space as *global* memory objects on CPU architectures, and their use would therefore potentially result in additional memory operations for no performance benefit.

6.2.2 Optimisations

Additional optimisations were subsequently applied to the initial implementation in order to assess their effectiveness at improving performance on a range of candidate processing devices, as well as their overall performance portability. The following sub-sections each document a particular candidate optimisation technique which was evaluated as part of this research.

`NDRange` Padding

An additional version of the application was developed to examine the effect on performance of employing different `NDRange` configurations. The requirement in the initial implementation for an exact `NDRange` to be specified for each *kernel* was relaxed and additional `if`-tests were added at the start of each *kernel* (Figure 6.4). These `if`-tests prevent grid points from being recalculated,

```
int k = get_global_id(1);
int j = get_global_id(0);

if ( (j>=2) && (j<=x_max) && (k>=2) && (k<=y_max) ) {
    double ss2,v,pe,pv;

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

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

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

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

Figure 6.4: The new device code for the `Ideal_gas` kernel.

or *buffers* from being accessed beyond their bounds, when *kernels* are launched with additional *work-items*. This approach enabled the use of the `NDRange` offset mechanism, required by the original *buffer* indexing scheme, to be removed.

An additional method, `enqueueKernel`, was also added to the static class to provide a wrapper around the `enqueueNDRangeKernel` function used to actually launch the *kernels*. Passing all calls which add a *kernel* to the *work-queue* through this function enabled the number of *work-items* launched for each *kernel* to be centrally controlled. As part of this optimisation this function was configured to ensure that *kernels* were launched with an `NDRange` which was always a multiple of the preferred *work-group* size of the current *device*[1]. This was accomplished by rounding the `NDRange` up in the x-dimension, whilst keeping the y-dimension constant.

Pre-processing Constant Values

To reduce data movement and redundant computation the OpenCL pre-processor was subsequently employed to replace all constant values within the *device*-side *kernels* prior to their compilation. This optimisation removed the need to explicitly pass these values into the *kernels* at run-time via the `setArg` mechanism. Additionally, use of the pre-processor also enabled the *buffer* index arithmetic calculations within the *kernels* to be further minimised.

Array Notation

The initial implementation of the codebase utilised pre-processor macros (of the form $[y * array\_width + x]$) to perform the array index calculations within each *kernel*. This approach potentially prevents the OpenCL compiler from

[1] `CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE`

implementing certain optimisations and may result in additional integer arithmetic. To eliminate these calculations and potentially improve the performance of the codebase an additional version was developed. This utilised explicit array notation (`[][]`) to index each array access within the *kernels*. Versions which employed this techniques are referred to by the description *ArrayNotation* within Section 6.3. A subsequent version, which removed the explicit cast operations required by this implementation, was also developed (denoted by *ArrayNotation_noCast* in Section 6.3).

Out-of-order Execution Command Queue

The *in-order* command queue employed in the initial implementation appropriately captures the dependency chain and synchronisation requirements of the vast majority of *kernel* invocations within the application. This approach, however, places unnecessary synchronisation constraints on the invoked OpenCL *kernels* at two locations within each CloverLeaf timestep. In particular during the *Field Summary* function when multiple reduction operations are required in parallel, and as part of the *Update-Halo* operation when multiple *kernels* are launched in parallel to modify different data buffers.

An additional *out-of-order* command queue was therefore employed to operate alongside the original *in-order* queue. *Kernels* which can execute in parallel were enqueued into the *out-of-order* command queue in batches separated by `enqueueBarrier` or `enqueueWaitForEvents` operations. These provide the required synchronisation constructs between these batches of *kernel* invocations. A global event object was also used to delay the execution of the first parallel batch of *kernels* in this queue until the immediate preceding *kernel* has finished executing within the *in-order* queue. On particular platforms, however, it was more performant to employ `event-wait` operations between the *kernel* batches rather than explicitly enqueuing `barrier` operations. On these platforms it is likely that the enqueuing of `barrier` operations does not cause the preceding batch of *kernels* to be executed on the actual target *devices*, however, the confirmation of this hypothesis is left to future research.

Specifying Explicit *Work-group* Sizes

The reference implementation also relied on the underlying OpenCL runtime system to select the most appropriate local *work-group* size for each *kernel* invocation. That is, a `null` value was passed to the appropriate argument when each *kernel* was enqueued, instead of an `NDRange`. An additional version was therefore developed which explicitly specified a local *work-group* size in order to examine the effect of this optimisation on application performance.

Merging Kernels

To reduce the overheads associated with frequently launching *kernels* additional versions were developed which merged particular *kernels* in order to increase the amount of computation performed per launch. Separate versions applied this potential optimisation at different locations within the overall algorithm. Specifically one version examined merging the light-weight, predominantly memory copy dominated, *Update-Halo kernels*, as well as several more computationally intense *kernels* within the *Advection* routines. Additionally a subsequent version also examined merging the first stage of the *reduction* operations into the immediately preceding *kernels* in order to potentially take advantage of *local* memory resources and minimise data motion to *global* memory. In Section 6.3 versions which employed these optimisation techniques are denoted by the acronyms *MK* (Merge Kernels) and *MR* (Merge Reductions), respectively.

`Restrict` & `Const` Keywords

Using knowledge of the algorithm and the implementation it was possible to determine that the pointers used to access the *buffer* objects, within the *kernel* implementations, each only reference a unique *buffer*. To communicate to the compiler that pointer-aliasing is not therefore employed, and thus enable it to potentially implement further optimisations, the `restrict` keyword was added to the *buffer* definitions within each *device*-side *kernel*. It was necessary to employ particular compiler options in order to enable this optimisation with certain OpenCL implementations. Additionally the `const` keyword was also applied to each of the *buffer* object declarations whose contents are not modified during the execution of a particular *kernel*.

OpenCL `clFlush` & `clFinish` Operations

To potentially improve the speed with which *kernels* are dispatched by the *host* and executed on target *devices*, use of the `clFlush` operation was examined within an additional version (labelled *Remove clFinish Calls and clFlush* within Section 6.3). This command was utilised directly after each *kernel* function or *barrier* operation was enqueued into the particular *work-queues*. Additionally, this version also minimised the use of `clFinish` synchronisation operations within the codebase. These had previously been included within the reference implementation in order to force particular *kernel* invocations to be dispatched.

Processing Multiple Grid-points per OpenCL *Work-item*

The performance of the reference implementation was particularly poor on CPU-based architectures. To examine whether this was due to excessive thread scheduling overheads caused by this version utilising one OpenCL *work-item* to process each grid-point, an additional implementation was developed (denoted by the description *J-loops* within Section 6.3). This version reduced the number of *work-groups*, created during each *kernel* launch, to a value closer to the overall number of CPU cores available on current system architectures. It also utilised OpenCL in a manner similar in approach to how OpenMP applications are generally constructed. An additional loop was employed within each *kernel* to enable the computations previously carried out by multiple *work-items* to be merged into a single *work-item*. The application was also further modified to reduce the index space used to launch each *kernel* to a one-dimensional `NDRange`, with one *work-item* now being initiated to process each row of the overall two-dimensional grid. This ensures that each *work-item* only accesses contiguous memory locations.

Overlapping Data Movement with Computation

An additional version was developed in order to examine whether performance improvements could be gained through the overlapping of data movement operations with subsequent computational *kernel* executions. Data movement operations can be particularly time consuming on architectures in which computational *devices* are connected to the main host system via relatively slow PCIe-bus connections. These operations occur at two locations within the CloverLeaf application, immediately following the calculation of the time-step value and the generation of intermediary results. The time-step value is required by the *kernel* which immediately follows its calculation; however, the transfer of the intermediary result values from the compute devices can be overlapped with subsequent *kernel* executions.

Data movement operations were therefore modified to be fully non-blocking operations and to also record their completion status within OpenCL `event` objects. The synchronisation operation which previously followed these operations was also removed and the application restructured, such that the functionality which requires the intermediate result values was executed as late as possible within the overall application sequence. A synchronisation operation, which depends on the previous `event` objects, was also inserted immediately prior to this functionality. To ensure that the required data transfers are successfully completed before execution proceeds. This modified arrangement ensures that the time spent waiting for the data transfers to complete is minimised, as

they are now considerably more likely to have been completed by the time the synchronisation operation is executed.

Auto-tuning OpenCL *Work-group* Parameters

To examine how the performance of the OpenCL application is affected on various architectures by the selection of different parameters, including the *work-group* (block) size of each *kernel*, an additional version was developed. The existing version which employed the optimisation to explicitly fix the value of the local *work-group* size of each *kernel* (Section 6.2.2) was further modified to enable a different *work-group* size to be specified independently for each *kernel* invocation. This modified version was also integrated into the Flamingo [185] auto-tuning framework, which enabled larger ranges of application configuration parameters to be evaluated more rapidly across a range of different architectures.

To examine whether the approach of specifying different values for each potential configuration parameter could deliver additional performance this version was subsequently utilised within the auto-tuning framework to determine the optimal local *work-group* size for each *kernel* on a range of different architectures. This approach was also applied to the reduction *kernels* to evaluate the optimal configuration sizes for each stage of the reduction tree. Versions which employed this technique are denoted by the description *auto-tuning* within Section 6.3.

Explicitly (Un)Packing MPI Communication Buffers

To evaluate the performance of the `clEnqueue[Write|Read]BufferRect` functions within a particular OpenCL runtime system and therefore to determine how performant the original MPI communication buffer (un)packing routines were, a subsequent version of the codebase was created. Explicit routines were developed within this version to pack and unpack the MPI communication buffers on the target computational *devices*. This functionality was implemented within additional *kernels*, each of which was specifically dedicated to operate on a particular face of the two-dimensional mesh. Additional *device*-side *buffer* objects were also created in order to contiguously store the data which was required to be communicated. The contents of these buffers was also transferred to/from the *host*-side MPI communication buffers using `enqueue[Write|Read]Buffer` functions. In the final version of the codebase which was developed as part of this research, the selection between both approaches is controlled by a compile time pre-processor directive. The performance of this modified approach is examined within Section 6.3.3.

6.3 Results Analysis

The OpenCL standard guarantees the functional portability of applications across architectures, however, there is no guarantee regarding the portability of performance. A series of experiments was therefore conducted to assess the *"performance portability"*—whether the same codebase can be performant on many devices—which it is possible to achieve by utilising OpenCL as a technology to implement a hybrid programming model for hydrodynamics applications.

In order to fully evaluate the performance and portability of the codebase, a wide range of hardware architectures from several major vendors was examined. These included CPUs from both AMD and Intel; GPUs from both AMD and Nvidia; an APU from AMD; and the Xeon Phi coprocessor architecture from Intel. Initially single-node experiments were conducted to assess the success of each of the candidate optimisations described in Section 6.2.2, the results of this analysis are presented in Section 6.3.1. The performance of the most effective versions were then subsequently analysed across a range of single-node systems in Section 6.3.2, and at considerable scale in Section 6.3.3. The *Tuck*, *Teller*, *Chilean Pine* and *Shannon* platforms were employed in the single-node experiments, whilst the multi-node experiments utilised the *Titan* supercomputer platform. The hardware and software setup used in the experiments on these platforms, including the options used to compile the OpenCL *kernels*, is detailed in Section A.1

To provide a baseline against which to compare the performance of the OpenCL-based implementations, the experiments also examined the performance of alternative versions of CloverLeaf, which were optimised for the particular platform architecture in their *native* programming models. For the CPU-based devices this involved comparing the implementations against an optimised OpenMP-based version and against an optimised CUDA-based implementation for the Nvidia GPU devices. Section A.1 contains information on the specific OpenMP and CUDA runtime systems employed on each architecture. No such comparison was, however, performed for the AMD GPU devices, as OpenCL is the *native* programming model on these platforms.

In order to assess the performance of OpenCL under different processing conditions (e.g. during both high and low memory usage scenarios) several different problem configurations, from the standard CloverLeaf benchmarking suite, were utilised. Except where noted the 960^2 and the $3{,}840^2$ cell problems, which were executed for 2,955 and 87 time-steps respectively, were employed in the single-node experiments to assess the utility of each candidate optimisation, as well as the performance portability of OpenCL across a range of hardware devices. In the multi-node experiments, however, the $15{,}360^2$ cell problem,

Version	3840^2(s)	960^2(s)
Initial version	16.803	42.646
ArrayNotation	16.669	41.938
ArrayNotation (NoCasts)	16.674	41.929
Pre-processing constants (PC)	16.610	41.788
J-loops	25.254	142.334
Merging Kernels (MK)	16.755	42.446
Remove clFinish Calls and clFlush	16.809	42.652
Out-of-Order Queue (OoOQ)	16.815	42.837
Overlapping Reads	16.809	42.638
Const & Restrict Keywords (RES)	16.403	43.284
Padding Kernel NDRanges (PADD)	17.901	42.954
ArrayNotation + J-loops	25.127	141.593
PADD + RES	18.645	44.425
PADD + Fix local workgroup (FLWG)	16.291	38.897
PADD + FLWG + MK	16.257	38.634
PADD + FLWG + MK + Merge Reductions into Kernels (MR)	16.184	38.550
PADD + FLWG + MK + MR + PC	15.938	37.867
PADD + FLWG + MK + MR + PC + RES	15.263	36.380
PADD + FLWG + MK + MR + PC + RES + OoOQ	15.284	36.578
PADD + FLWG + MK + MR + PC + RES + Autotuning	14.951	35.880

Table 6.1: OpenCL optimisations on the Nvidia K20X

executed for 2,955 timesteps, was examined in a strong-scaling experimental configuration. Additionally the $3{,}840^2$ cells per node problem, executed for 87 timesteps, was also examined in a weak-scaling experimental configuration. During each experiment CloverLeaf was configured as described in Section 1.6.1. All performance results presented show the total application wall-clock time in seconds and are averages from three separate executions of each particular experiment. Except where noted, all hardware platforms are paired with the OpenCL SDK and runtime systems from their particular manufacturer.

6.3.1 Optimisations Analysis

As part of this research experiments were conducted to examine the utility of each candidate OpenCL optimisation technique (documented in Section 6.2.2) on the: Nvidia K20X, Intel Xeon Phi 7120P co-processor, Intel Xeon E5-2620 and AMD Opteron 6272 architectures. Tables 6.1 to 6.4 present the results obtained from these experiments on each processor architecture respectively. Each result is an average from three repeated executions of the particular experiment. The utility of each candidate optimisation is analysed in further detail in the subsequent sections.

Array Notation

The results show that employing array notation to index each array access within the OpenCL *kernels* delivers a modest performance improvement of ∼1% on the K20X architecture for both problem classes examined. On the Xeon

Version	3840^2(s)	960^2(s)
Initial version	60.656	171.186
ArrayNotation	60.966	168.021
ArrayNotation (NoCasts)	61.031	174.047
Pre-processing constants (PC)	60.702	166.684
J-loops	79.046	231.920
Merging Kernels (MK)	60.798	170.294
Remove clFinish Calls and clFlush	60.985	167.931
Const & Restrict Keywords (RES)	60.427	175.225
Padding Kernel NDRanges (PADD)	60.789	175.053
PADD + Fix local workgroup (FLWG)	76.639	214.714
PADD + FLWG + MK	79.749	220.829
PADD + FLWG + MK + Merge Reductions into Kernels (MR)	78.440	211.248
PADD + FLWG + MK + MR + PC	80.815	216.834
PADD + FLWG + MK + MR + PC + RES	81.244	214.115
PADD + FLWG + MK + MR + PC + RES + Autotuning	68.042	178.374
PADD + FLWG + PC + RES + Autotuning	67.350	182.413
PADD + FLWG + Autotuning	67.568	190.207

Table 6.2: OpenCL optimisations on the Intel Xeon E3-2620

Phi platform, however, this candidate optimisation resulted in performance degradations of 16.2% and 10.5% for the $3{,}840^2$ and 960^2 cell problems respectively. Additionally, in the experiments on the Xeon E5-2620 CPU architecture this optimisation resulted in a performance improvement of $\sim$1.8% for the 960^2 cell problem size whilst it did not significantly affect performance in the experiments with the $3{,}840^2$ cell problem size. In the experiments with the $3{,}840^2$ problem class on the AMD Opteron platform this optimisation also resulted in a performance improvement of $<$1%.

Furthermore removing the cast operations required by the initial array notation implementation resulted in no significant change in application performance on the K20X, Xeon Phi and Opteron architectures. On the Xeon CPU architecture, however, removing these operations resulted in a performance degradation of 3.6% for the 960^2 cell problem.

Processing Multiple Grid-points per *Work-item*

Employing the candidate optimisation technique of reconfiguring the OpenCL *kernels* such that each is launched with only a one-dimensional `NDRange` and the associated *work-items* each process multiple grid-points, results in significant reductions in performance for both problem sizes on the K20X, Xeon Phi and Xeon platforms. On the K20X this optimisation resulted in 3.3× and 1.5× reductions in performance for the 960^2 and $3{,}840^2$ cell problem classes respectively. Additionally, on the Xeon and Xeon Phi architectures it resulted in performance slowdowns of 3.8× and 1.3× for the $3{,}840^2$ problem size and slowdowns of 3.2× and 1.4× for the 960^2 problem classes respectively. The AMD Opteron architecture was, however, the only platform on which this optimisation

Version	3840^2(s)	960^2(s)
Initial version	64.869	231.550
ArrayNotation	75.362	255.813
ArrayNotation (NoCasts)	74.824	255.368
Pre-processing constants (PC)	63.511	224.473
J-loops	248.979	734.973
Merging Kernels (MK)	67.800	233.071
Remove clFinish Calls and clFlush	64.515	231.211
Const & Restrict Keywords (RES)	63.168	228.392
Padding Kernel NDRanges (PADD)	64.006	232.070
PADD + Fix local workgroup (FLWG)	66.072	258.359
PADD + FLWG + MK	70.023	258.339
PADD + FLWG + MK + Merge Reductions into Kernels (MR)	75.114	267.104
PADD + FLWG + MK + MR + PC	69.819	256.265
PADD + FLWG + MK + MR + PC + RES	68.161	257.695
PADD + FLWG + MK + MR + PC + RES + Autotuning	62.466	235.730
PADD + FLWG + PC + RES + Autotuning	58.805	228.818
PADD + FLWG + Autotuning	62.439	238.724

Table 6.3: OpenCL optimisations on the Intel Xeon Phi 7120P

delivered a performance improvement. In the experiments on this architecture the application of this optimisation technique resulted in a 1.15× performance improvement for the $3{,}840^2$ cell problem class.

Preprocessing Constants

Utilising the OpenCL preprocessor to pass constant values into the *kernels* during compilation rather than at runtime also consistently delivered improvements in performance on all of the architectures examined in this research. On the K20X GPU architecture employing this optimisation resulted in performance improvements of 2.0% and 1.15%, relative to the reference implementation, for the 960^2 and $3{,}840^2$ cell problem classes respectively. Similarly, on the Xeon Phi it also delivered performance improvements of 3.1% and 2.1% for the 960^2 and $3{,}840^2$ cell problems respectively. In the experiments on the Xeon architecture, however, utilising this optimisation resulted in no significant change in performance during the experiments with the $3{,}840^2$ cell problem size. Whilst it delivered a 2.6% performance improvement for the 960^2 cell problem class. During the experiment on the Opteron, however, this optimisation was less effective delivering a <1% improvement in application performance.

Out-of-order Command Queue

Employing an *out-of-order* command queue where possible within the OpenCL implementation of CloverLeaf achieved variable levels of success across the architectures examined in this research. The results show that on the K20X GPU architecture the use of this approach delivered factional reductions in application performance of <1% for both problem classes examined (960^2 and

Version	3840^2(s)	960^2(s)
Initial version	206.831	17.737
ArrayNotation	205.037	-
ArrayNotation (NoCasts)	205.138	-
Pre-processing constants (PC)	205.743	-
J-loops	179.778	17.500
Merging Kernels (MK)	204.883	-
Remove clFinish Calls and clFlush	206.670	-
Out-of-Order Queue (OoOQ)	188.024	16.472
Const & Restrict Keywords (RES)	207.384	-
PADD + Fix local workgroup (FLWG)	226.339	-
PADD + FLWG + MK	319.267	-
PADD + FLWG + MK + Merge Reductions into Kernels (MR)	222.875	-
PADD + FLWG + MK + MR + PC	219.593	-
PADD + FLWG + MK + MR + PC + RES	218.721	-
PADD + FLWG + MK + MR + PC + RES + OoOQ	199.487	-

Table 6.4: OpenCL optimisations on the AMD Opteron 6272

$3{,}840^2$ cells). On the Opteron architecture, however, the use of this technique resulted in performance improvements of 9.1% and 7.1% for the $3{,}840^2$ and 960^2 cell problem classes respectively. In all of the experiments on both the Xeon and Xeon Phi platforms the application binary produced by the incorporation of this optimisation into the CloverLeaf codebase consistently delivered incorrect simulation results, for both problem classes examined in this research. This suggests that there maybe an underlying problem with the implementation of this functionality within the Intel OpenCL runtime system, as the identical codebase produced the correct results on the equivalent Nvidia and AMD OpenCL runtime systems.

Removing `clFinish` & Utilising `clFlush`

The experimental results show that eliminating the `clFinish` operations within the reference implementation of the OpenCL version of CloverLeaf and utilising `clFlush` operations immediately after every *kernel* `enqueue` operation does not significantly affect application performance, for both the problem classes examined on the K20X, Xeon Phi and Opteron architectures. On the Xeon architecture, however, employing this technique resulted in a 1.9% improvement in application performance for the 960^2 cell problem class and a fractional reduction in performance of $<1\%$ in the experiments with the $3{,}840^2$ cell problem class.

Merging Kernels

The candidate optimisation of reducing the number of *Update-halo* and *Advection kernels* through mergers delivered fractional but consistent improvements in application performance on the K20X architecture of 0.46% and 0.28% for the

960^2 and $3{,}840^2$ cell problem classes respectively. On the Xeon Phi, however, this approach resulted in performance slowdowns of 4.5% and 0.66% for the $3{,}840^2$ and 960^2 cell problems respectively. During the experiments on the Xeon CPU architecture employing this optimisation resulted in a 0.5% improvement in performance for the 960^2 cell problem class and delivered a 0.23% slowdown for the $3{,}840^2$ cell problem. In the experiments on the Opteron architecture with the $3{,}840^2$ cell problem, however, it improved application performance by 0.94%.

The effect of merging the first stage of the reduction operations into the preceding *kernels* also varied across the architectures. On the K20X platform this optimisation delivered fractional performance improvements of 0.22% and 0.45% for the 960^2 and $3{,}840^2$ cell problem classes respectively. Similarly on the Xeon architecture employing this technique also delivered improvements in application performance of 4.34% and 1.64% for the 960^2 and $3{,}840^2$ cell problem classes respectively. This trend was reversed on the Xeon Phi, however, as the optimisation resulted in degradations in application performance of 7.3% and 3.4% for the $3{,}840^2$ and 960^2 cell problem classes respectively.

Overlapping Data Movement with Computation

The technique developed to overlap computational operations with the movement of data between the OpenCL *host* and compute *devices* also did not significantly affect application performance, either detrimentally or beneficially, in the experiments conducted with both problem classes on the K20X architecture. On the Xeon and Xeon Phi architectures, however, the implementation of this technique resulted in the production of incorrect simulation answers for both problem classes, indicating that a problem potentially exists within the Intel OpenCL runtime system, as an identical codebase produced the correct results on all other processing technologies examined in this research.

Padding `NDRange` and Fixing Local *Work-group* Sizes

The experimental results show that on the K20X architecture padding the `NDRange` used to launch each *kernel*, such that it is a multiple of the preferred vector width of the target device, actually initially results in performance degradations. Reductions of 6.5% and 0.7% were recorded, relative to the reference implementation, for the $3{,}840^2$ and 960^2 cell problem classes respectively. Combining this technique with the optimisation to specify a fixed local *work-group* size for each *kernel* launch, however, improves performance on the K20X architecture by 8.8% and 3.0%, for the 960^2 and $3{,}840^2$ cell problem classes respectively.

On both the Xeon and Xeon Phi architectures, however, these candidate optimisation techniques generally result in significant degradations in application performance. The experiments on the Xeon architecture indicate that padding the *kernel* `NDRange` results in modest reductions in performance of 2.3% and 0.2% for the 960^2 and $3{,}840^2$ cell problem classes respectively. Specifying a fixed local *work-group* size for each *kernel* launch, however, resulted in significant further performance reductions of 25.4% and 26.4% for the 960^2 and $3{,}840^2$ cell problem classes respectively. A similar trend can be observed in the results obtained from the experiments on the Xeon Phi architecture. On this platform the technique of padding the *kernel* `NDRange` results in a fractional 0.2% performance reduction for the 960^2 cell problem class and a performance improvement of 2.6% in the experiments with the $3{,}840^2$ cell problem class. Applying the candidate optimisation of specifying fixed local *work-group* sizes, however, again reduces performance by 11.6% and 1.9% for the 960^2 and $3{,}840^2$ cell problem classes respectively.

Similarly, on the Opteron architecture a 9.43% reduction in performance was recorded for the modified version which combined the `NDRange` padding and fixed local *work-group* size optimisations.

The selection of the local *work-group* block size employed in these experiments may well be more suited to the K20X GPU architecture than to the Opteron, Xeon and Xeon Phi architectures. The extent to which this is the case, particularly for the Xeon Phi architecture, will be explored in subsequent sections of this chapter.

Utilising the `Restrict` & `Const` Keywords

On the K20X architecture the experimental results indicate that employing the `restrict` and `const` keywords on the appropriate OpenCL *kernel* parameters generally delivered a performance degradation when this technique was employed in isolation. Applying these modifications to the reference implementation and to the version which employed the `NDRange` padding optimisation resulted in performance degradations of 1.5% and 3.4% respectively for the 960^2 cell problem class. For the $3{,}840^2$ cell problem class, applying these constructs to the `NDRange` padding version resulted in a 4.2% performance reduction, however, when applied to the reference implementation performance was improved by 2.4%.

The results obtained during the experiments on the Xeon Phi architecture show that employing these constructions generally delivers performance improvements. For the $3{,}840^2$ cell problem case applying these modifications to the reference version resulted in a 2.6% improvement in performance. Whilst when

incorporated into the version which also includes the `NDRange` padding, fixed local *work-group* sizes, *kernel* mergers and pre-processing constants optimisations, this optimisation delivered a further 2.4% improvement in performance. In the experiments with the 960^2 cell problem class applying these constructs to the reference implementation resulted in a 1.4% performance improvement; however, applying the technique to the version which incorporated the aforementioned list of optimisations, resulted in a fractional performance degradation of 0.8%.

A similar trend was also observed in the experiments with the $3{,}840^2$ cell problem class on the Xeon architecture. The application of these modifications resulted in a 0.4% performance improvement for the reference implementation but a performance degradation of 0.5% when they were applied to the version which incorporated the previously mentioned list of additional optimisations. The results from the experiments with the 960^2 cell problem class on the Xeon architecture, however, demonstrated the opposite trend. In these experiments a 2.3% reduction in performance was observed as a result of applying these constructs to the reference implementation. When these modifications were subsequently applied to the version which incorporated the aforementioned list of optimisations, however, a performance improvement of 2.3% was recorded.

Additionally, during the experiments with the $3{,}840^2$ cell problem class on the Opteron architecture employing these constructs did not significantly alter the overall performance of the application.

Combining Optimisations

The previous experiments generally examined the utility of each optimisation technique in isolation. The particular optimisation techniques which the previous results analysis indicates delivers potential performance benefits were subsequently combined, in the next stage of this work, to produce further alternative versions of the codebase. The results from the experiments with these additional versions are also presented within the lower sections of Tables 6.1 to 6.4.

On the Nvidia K20X GPU architecture the results (Table 6.1) show that combining the `NDRange` padding and fixed local *work-group* size optimisations delivered a 8.8% and 3.0% improvement in performance relative to the reference implementation for the 960^2 and $3{,}840^2$ problem classes respectively. Supplementing this version with the *kernel* merger optimisations further improved performance and increased the achieved speedup to 9.6% and 3.7%, relative to the reference implementation, for the 960^2 and $3{,}840^2$ cell problem classes respectively. Additionally, incorporating the pre-processing of constant values optimisation also delivered further performance benefits increasing the achieved

speedup, relative to the reference implementation, by up to 11.2% for the 960^2 cell problem class and by up to 5.1% for the $3{,}840^2$ cell problem class. Finally, utilising the `restrict` and `const` keywords optimisation, together with the `NDRange` padding, fixed local *work-group*, *kernel* merger and pre-processing constant values modifications resulted in further performance improvements. This optimisation generally resulted in performance degradations, however, when the technique was applied in isolation to the reference implementation. In the experiments with the 960^2 cell problem class the use of this technique increased the achieved performance speedup, relative to the reference implementation, to 14.7% and to 9.2% for the $3{,}840^2$ cell problem class.

The results recorded during similar experiments on the Xeon Phi architecture (Table 6.3) indicate that combining these optimisation techniques was ultimately less successful on this architecture. Relative to the reference implementation the version which incorporated the `NDRange` padding and fixed local *work-group* size optimisations resulted in performance degradations of 11.5% and 1.9% for the 960^2 and $3{,}840^2$ cell problem sizes respectively. The application of the *kernel* merger optimisations to this version resulted in additional performance degradations, with the cumulative performance reduction increasing to 15.4% and 15.8%, relative to the reference implementation, for the 960^2 and $3{,}840^2$ cell problem classes respectively. In addition to these optimisations, however, applying the pre-processing constant values optimisation resulted in a performance improvement for both the 960^2 and $3{,}840^2$ cell problem classes. This reduced the performance degradation relative to the reference implementation to 10.7% and 7.6% respectively. The inclusion of the `restrict` and `const` keywords also resulted in further performance benefits for the $3{,}840^2$ cell problem class on this architecture, reducing the performance degradation relative to the reference implementation to 5.1%. Although for the 960^2 cell problem class the use of this optimisation resulted in a fractional reduction in performance, increasing the performance degradation relative to the reference implementation from 10.7% to 11.6%.

On the Intel Xeon CPU the performance results (Table 6.2) obtained during the experiments with the versions of the codebase which incorporate the combined optimisations show similar trends to those observed on the Xeon Phi architecture. Initially combining the `NDRange` padding and fixed local *work-group* size optimisations led to a 25.4% and a 26.4% performance reduction, relative to the reference implementation, for the 960^2 and $3{,}840^2$ cell problem classes, respectively. Adding the *kernel* merger technique to these optimisations further reduced performance for the $3{,}840^2$ cell problem class relative to the reference implementation, increasing the degradation to 29.3%. Although for the 960^2 problem class the inclusion of this optimisation fractionally improved application

performance, reducing the degradation relative to the reference implementation to 23.4%. Incorporating the pre-processing constant values optimisation into these experiments, however, resulted in further reductions in performance for both problem classes on this platform. Due to this optimisation the performance degradation, relative to the reference implementation, was increased to 33.2% and 26.7% for the $3{,}840^2$ and 960^2 cell problem classes respectively. The additional inclusion of the `restrict` and `const` keywords did not significantly affect the performance of the codebase. Relative to the reference implementation, this optimisation marginally increased the performance slowdown to 33.9% for the $3{,}840^2$ cell problem class but fractionally improved performance for the 960^2 cell simulations, decreasing the overall performance degradation to 25.1%.

Results were also recorded from the execution of the same set of experiments on the AMD Opteron processor architecture (Table 6.4) although only for the $3{,}840^2$ cell problem class. These indicate that the combination of the `NDRange` padding and the fixed local *work-group* size optimisations again result in significant performance reductions, in this instance a 9.43% degradation was recorded relative to the reference implementation. Incorporating the *kernel* merger optimisation initially resulted in a further large reduction in performance, increasing the performance disparity relative to the reference implementation to 54.4%. Subsequently employing the optimisation to merge the first stage of the reduction operations into the preceding *kernels*, however, delivered significant performance benefits and decreased the performance degradation from 54.4% down to 7.8%. The addition of the pre-processing constant values optimisation resulted in further performance benefits and reduced the performance degradation to 6.2% relative to the reference implementation. Similarly including the `restrict` and `const` keywords optimisation also delivered important performance benefits and further decreased the performance disparity to 5.7%. The inclusion of the *out-of-order* command queue optimisation, however, delivered further significant performance improvements and enabled the modified codebase to out-perform the reference implementation by 3.6%.

Auto-tuning Analysis

The results from the auto-tuning experiments conducted as part of this research are presented in Table 6.5. This table shows the dimensions of the local *work-group* block-sizes for each application *kernel*, which produced the most optimal overall application performance on both the Nvidia K20X GPU and Intel Xeon Phi 7120P platforms. The results demonstrate significant variations in the optimal local *work-group* block-sizes for each individual application *kernel* across both architectures as well as between the different *kernels* on a particular

Kernel	Nvidia GPU	Intel Xeon Phi
Ideal Gas	32×4	128×1
Viscosity	32×4	256×2
Accelerate	256×2	128×4
Flux Calc	128×1	128×8
Reset Field	512×2	8×4
Revert Field	128×1	1024×1
PDV	128×1	256×4
Advec Cell Xdir Kernel1	32×4	8×16
Advec Cell Xdir Kernel2	128×4	32×8
Advec Cell Xdir Kernel3	256×1	64×16
Advec Cell Xdir Kernel4	32×2	128×1
Advec Cell Ydir Kernel1	128×2	32×1
Advec Cell Ydir Kernel2	128×2	16×16
Advec Cell Ydir Kernel3	32×4	64×8
Advec Cell Ydir Kernel4	256×4	512×2
Advec Mom Volume	64×2	128×1
Advec Mom Xdir Node	256×2	128×1
Advec Mom Xdir MassPre	64×2	64×1
Advec Mom Xdir Flux	32×4	512×1
Advec Mom Xdir Velocity	128×1	256×4
Advec Mom Ydir Node	256×2	512×2
Advec Mom Ydir MassPre	64×2	256×1
Advec Mom Ydir Flux	32×4	128×2
Advec Mom Ydir Velocity	256×1	128×4
Calc DT	32×4	32×1
Field Summary	32×4	32×1
Reductions	512×1	128×1
Comms Buffer Packing	64×1	-
Update-halo	16×2	8×2

Table 6.5: Optimal work-group sizes for each OpenCL CloverLeaf kernel

architecture. In general the results indicate that block-sizes which are wider (generally >128 *work-items*) in the x-dimension are required to produce optimal performance on the Xeon Phi, however, this is not always the case on the K20X architecture.

The optimal local *work-group* block-sizes, presented in Table 6.5, were subsequently applied to the main application codebase to produce several additional versions. The fourth section of Tables 6.1 to 6.3 present the results obtained from these experiments on the K20X, Xeon and Xeon Phi architectures respectively.

These results show that on the K20X GPU platform (Table 6.1) the use of these optimal *kernel* specific local *work-group* block-sizes resulted in further performance improvements of 2.9% and 1.4% for the $3{,}840^2$ and 960^2 cell problem classes respectively.

On the Xeon Phi architecture applying the optimal local *work-group* block-sizes to the version which incorporates the `NDRange` padding, *kernel* merger, pre-processing constant values and the `restrict` keyword optimisations resulted in further performance improvements of 9.6% and 8.5% for the $3{,}840^2$ and 960^2 cell problem sizes respectively. For the $3{,}840^2$ cell problem size this enabled this version to out-perform the reference implementation by 3.7% and further reduced the performance disparity to the reference implementation for the 960^2 cell problem class to 1.8%.

Additionally utilising the optimal local *work-group* block-sizes derived for the Xeon Phi in the experiments on the Xeon CPU platform, also delivered further performance improvements on this architecture. Applying the auto-tuned local *work-group* block-sizes to the version which incorporated the `NDRange` padding, *kernel* merger, pre-processing constant values and `restrict` keyword optimisations facilitated performance improvements of 16.4% and 17.8% for the $3{,}840^2$ and 960^2 cell problem sizes respectively. This optimisation enabled the performance disparity to the reference implementation to be further reduced to 4.2% for the 960^2 cell problem class and to 12.2% for the $3{,}840^2$ cell problem class.

Optimisations Analysis Summary

Overall this research enabled the performance of the OpenCL-based version of CloverLeaf on the Nvidia K20X architecture, to be improved by 15.8% and 11.0%, relative to the reference implementation, for the 960^2 and $3{,}840^2$ cell problem sizes respectively. On this architecture the most performant version, for both problem classes, utilised the following optimisations: `NDRange` padding, fixed local *work-group* sizes, *kernel* merger, pre-processing constant values, `restrict` & `const` keywords and the auto-tuning of local *work-group* block-sizes. The use of array notation, *out-of-order* command queues, overlapping computation with data movement and processing multiple grid-point per *work-item* techniques did not deliver any performance benefits.

On the Xeon Phi platform the most performant version for the $3{,}840^2$ cell problem class utilised the `NDRange` padding, fixed local *work-group* sizes, pre-processing constant values, `restrict` & `const` keywords and the auto-tuned block-size optimisations. This version delivered a performance improvement, relative to the reference implementation, of 9.3% for the $3{,}840^2$ cell problem class but only 1.2% for the 960^2 cell problem class. On this architecture the use of the `NDRange` padding and the fixed local *work-group* block-size optimisations result in performance degradations when they are used in isolation. They are, however, required in order to employ the auto-tuning optimisation which can deliver significant performance benefits. The most performant version for the 960^2 cell problem class was actually the reference implemented with only the pre-processing constant values optimisation applied to it, this achieved a 3.0% performance improvement compared to the reference implementation. On this architecture the array notation, processing multiple grid-point per *work-item*, *kernel* merger, *out-of-order* command queue and overlapping computation with data-movement optimisations were ineffective and often resulted in significant reductions in overall performance.

The experimental results obtained on the Xeon CPU architecture demonstrate that for the $3{,}840^2$ cell problem class the original reference implementation is overall the most performant version. Although the use of the auto-tuned local *work-group* block-sizes derived on the Xeon Phi architecture also delivers significant performance benefits on this architecture. This optimisation requires the use of the `NDRange` padding and the fixed local *work-group* size optimisations, the use of which results in significant performance degradations and the net-result is an overall reduction in performance. An identical performance trend is also demonstrated in the results obtained from the experiments with the 960^2 cell problem class. In these experiments the *array notation* optimisation delivers some performance benefits and results in an overall performance improvement of 1.8% relative to the reference version. However for this problem class the most performant version is again the reference implementation with only the pre-processing constant values optimisation applied to it. On this architecture implementing the optimisations to: process multiple grid-points per *work-item*, merge *kernels*, utilise an *out-of-order* command queue and overlap data movement with computation, were ineffective and resulted in significant performance reductions.

On the Opteron CPU architecture the most performant version employed the optimisation of processing multiple grid-point per *work-item*. In the experiments with the $3{,}840^2$ cell problem class this optimisation achieved a performance improvement of 13.1% relative to the reference implementation. Utilising an *out-of-order* command queue also delivered significant performance benefits on this architecture and improved performance by 9.1% when compared to the reference implementation. The candidate optimisations of utilising array notation, pre-processing constant values, merging *kernels*, utilising the `restrict` & `const` keywords and minimising `clFinish` operations were largely ineffective and their use resulted in negligible changes in overall application performance.

6.3.2 Single-node Performance Analysis

Following the analysis documented in Section 6.3.1 the most performant OpenCL-based version of CloverLeaf on each particular architecture was subsequently used to conduct an inter-architecture performance comparison on single node instances of each processor type. This enabled the performance of the OpenCL programming model to be objectively assessed across multiple different architectures and also relative to the native programming models for those particular platforms. In these experiments optimised OpenMP and CUDA versions of the application were utilised as the native programming models on the CPU and Nvidia GPU architectures respectively.

Device	OpenCL (s)	Native (s)	Speedup (%)
Tesla K20X	14.95	13.77	-7.89
Xeon E3-2620 × 2	60.66	52.67	-13.17
Xeon Phi 7120P(2tperC)	58.80	57.03	-3.10
Xeon Phi 7120P(3tperC)	58.80	58.79	-0.01
Xeon Phi 7120P(4tperC)	58.80	66.45	11.51
Opteron 6272	179.78	233.97	30.14

Table 6.6: Runtime of the OpenCL implementation for the $3,840^2$ problem

These experiments examined the performance (total application wall-time) of the codebase on the Nvidia Tesla K20X, Intel Xeon E3-2620, Intel Xeon Phi 7120P, AMD Opteron 6272, AMD A10-5800K and AMD HD-7660D architectures. The *Shannon, Tuck, Chilean Pine and Teller* platforms were utilised to archive this architectural coverage (see Section A.1 for more details). The 960^2 and $3,840^2$ cell problems from the standard CloverLeaf benchmarking suite were again utilised and executed for 2,955 and 87 timesteps respectively. Tables 6.6 and 6.7 present the results obtained from the experiments with the $3,840^2$ and 960^2 cell problem classes respectively. The approximate memory usage of the 960^2 cell problem is 500MB, which means that it is able to fit within the available memory on all of the devices employed in this study. The $3,840^2$ problem class, however, consumes approximately 5GB of main memory capacity, preventing it from being examined on the AMD A10-5800K and AMD HD-7660D architectures.

The native programming model experiments on the Xeon Phi 7120P platform utilised OpenMP in the *"offloading"* mode configuration and examined the effect on performance of varying the total number of threads as well as the number of threads employed per processing core. The results obtained from the Opteron 6272 architecture were derived from experiments which employed 8 OpenMP threads, i.e. they utilised one thread per floating-point unit within the CPU. Similarly, the experiments on the Xeon E3-2620 architecture utilised OpenMP across both processor sockets and employed one thread per processor core (i.e. the Intel Hyper-Threads within the CPU were not utilised).

The results show that for the $3,840^2$ cell problem class, the performance of the OpenCL implementation on the Nvidia K20X architecture is not able to match that of the optimised CUDA version, delivering a 7.89% slowdown in relative performance. In the experiments with the 960^2 cell problem class, however, the OpenCL version actually delivered a performance improvement of 1.64% over the native CUDA implementation. This performance discrepancy is likely due to the fact that the local *work-group* size auto-tuning optimisations were not implemented within the native CUDA version. Collectively, however, both results demonstrate that the OpenCL programming model is able to pro-

Device	OpenCL (s)	Native (s)	Speedup (%)
Tesla K20X	35.88	36.48	1.64
Xeon E3-2620 × 2	166.68	132.77	-20.34
Xeon Phi 7120P(2TperC)	224.47	664.63	66.22
Opteron 6272	16.47	13.76	-16.42
Trinity A10-5800K	947.08	627.06	-51.03
Trinity HD-7660D	678.26	-	-

Table 6.7: Runtime of the OpenCL implementation for the 960^2 problem

vide broadly equivalent performance to CUDA on processing architectures of this type.

On the Intel Xeon E3-2620 dual CPU architecture the performance of the OpenCL implementation is 13.17% and 20.34% slower than that of the optimised OpenMP version for the $3,840^2$ and 960^2 cell problem classes respectively. In the experiments on the AMD Opteron 6272 CPU architecture, however, the OpenCL implementation was able to deliver superior performance to the OpenMP programming model for the $3,840^2$ cell problem class, achieving a speedup of 30.14%. Although for the 960^2 cell problem class the performance of the OpenCL implementation is approximately 16.42% slower than that of the native OpenMP implementation.

The experimental results from the Xeon Phi 7120P platform show significant variations when different numbers of OpenMP threads are utilised per processing core. In the experiments with the $3,840^2$ cell problem class, utilising two threads per processor core was the most performant configuration, delivering performance improvements of 14.17% and 2.99% relative to the four and three threads per core configurations respectively. On this platform the OpenCL implementation was able to broadly match the performance of the OpenMP version for this problem class. Its performance was only 3.10% slower than that of the OpenMP version in the two threads per core experiment and the performance of both versions was almost identical (within 0.01%) in the three threads per core case. Relative to the OpenMP version (four threads per core), however, the OpenCL implementation delivered a performance improvement of 11.51%. It is not clear how many hardware threads the OpenCL implementation actually utilises, however, these results demonstrate that significant performance benefits could potentially be obtained by restricting their use. In the experiments with the 960^2 cell problem class, however, the OpenCL implementation delivered a significant performance advantage of 66.22% (2.96×) relative to the OpenMP version. This result together with the observation that performance is generally worse on the Xeon Phi, relative to the K20X architecture, for the smaller 960^2 cell problem class (6.3×) compared to the larger $3,840^2$ cell problem size (3.9×), indicates that the Xeon Phi is less effective at processing problem configurations

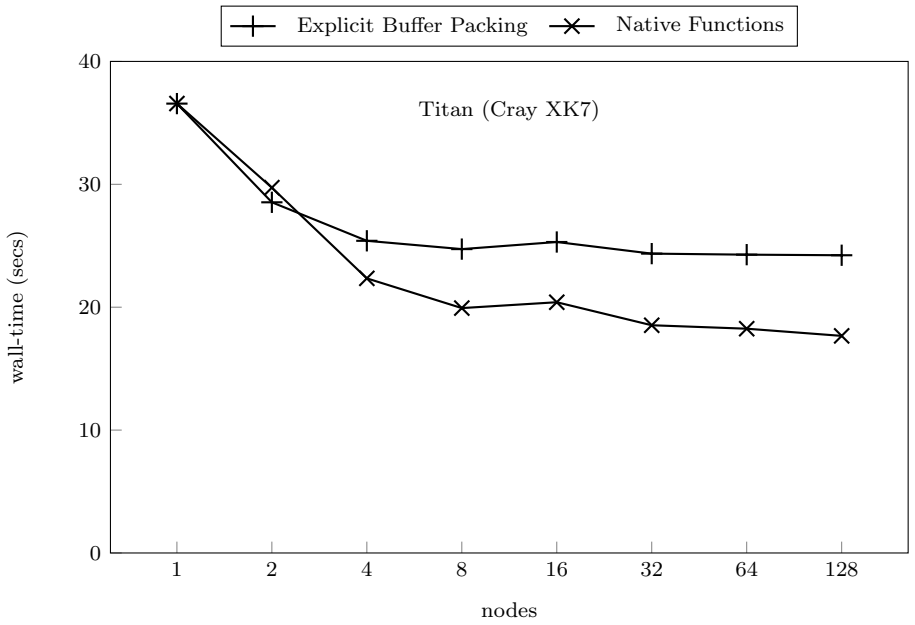


Figure 6.5: Buffer packing strong scaling performance (960^2 cell problem)

with smaller mesh sizes.

The OpenCL implementation was the only version able to execute on the HD-7660D part of the AMD Trinity APU. Although the performance of the 960^2 cell problem class on this architecture was 1.4× better than on the CPU component on the Trinity APU, it was still 18.9× slower than the Nvidia K20X architecture.

Overall the Nvidia K20X GPU platform proved to be the most performant architecture for this class of application. In the experiments with the $3,840^2$ cell problem class and the OpenCL implementation of CloverLeaf, the K20X outperformed the Xeon Phi by 3.93×, the dual socket Xeon E3-2620 platform by 4.1×, and the single socket Opteron 6272 by 12.0×.

6.3.3 Multi-node Performance Analysis

Further research was subsequently conducted to assess the performance of the OpenCL programming model at extreme-scale. This examined the performance of the MPI+OpenCL implementation of CloverLeaf relative to equivalent MPI+CUDA and MPI+OpenACC implementations on Titan, and relative to an equivalent MPI-only version on the Archer and HECToR platforms. The experiments examined the performance characteristics of the various programming models in a strong-scaling experimental configuration, using the $15,360^2$ cell problem, and also the weak-scaling performance using the $3,840^2$ cells/n-

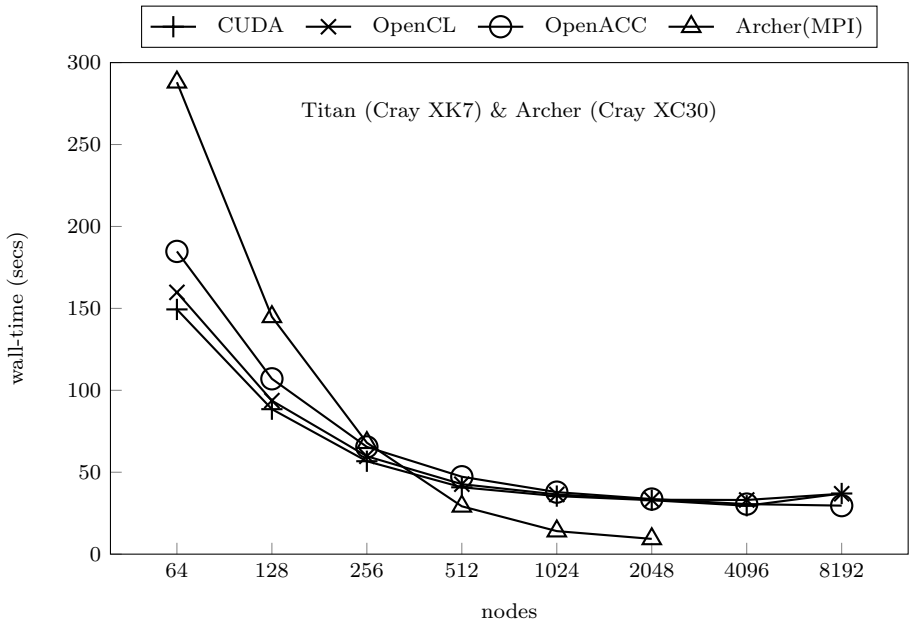


Figure 6.6: Strong-scaling performance ($15{,}360^2$ cell problem)

ode problem. These experiments were executed for 2,955 and 87 timesteps respectively. Additionally two alternative communication buffer management approaches were examined using the 960^2 cell problem which was executed for 2,955 timesteps in a strong-scaling experimental configuration. Each experimental configuration represents a standard simulation available within the CloverLeaf benchmarking suite. All experimental results presented in this section are also averages from three separate executions of each particular experiment.

Alternative Communications Buffer Management Approaches

Figure 6.5 presents the results obtained from the experiments which examined the alternative communications buffer management approaches described in Section 6.2.2. The results show that initially, in the one and two node experiments, the performance of both versions is virtually identical. Beyond this point, however, the performance of the version which utilises the native OpenCL built-in functions is significantly superior to that of the version which employs the explicit buffer management *kernel* routines. In the four node experiment the version which utilises the native functions is approximately 1.14× quicker. This performance disparity widens as the scale of the experiments is increased and the performance of the application becomes increasingly dominated by the speed of communication operations (smaller problem size per GPU), reaching ∼1.37× in the 128 node case.

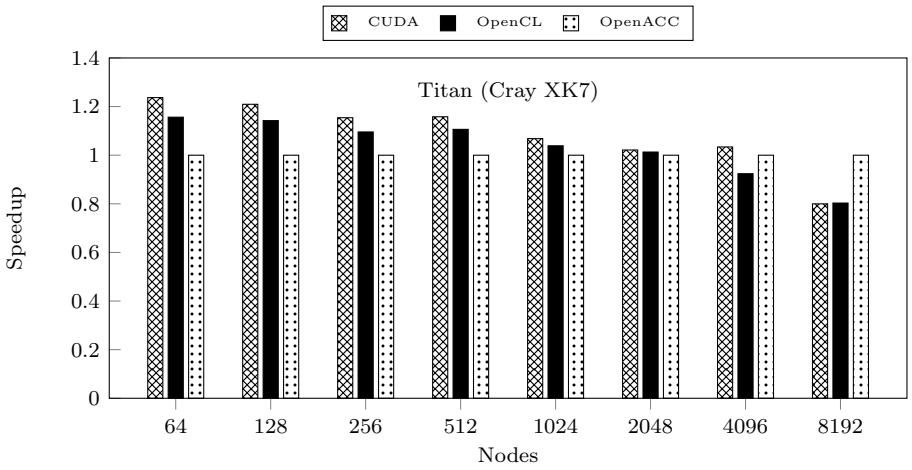


Figure 6.7: Speedup, relative to OpenACC, of CUDA and OpenCL

Strong-scaling Results Analysis

Figure 6.6 presents the absolute performance results (application wall-time) obtained during the strong-scaling experiments. These experiments employed the larger $15,360^2$ cell problem size (executed for 2,955 timesteps) and examined the performance of the MPI+CUDA, MPI+OpenCL and MPI+OpenACC (using the OpenACC `Kernel` constructs) versions of the codebase on the Titan platform as well as the MPI-only version on the Archer platform. It is evident from this chart that the performance of the MPI+OpenCL and MPI+CUDA versions of the codebase is broadly equivalent throughout all of the experimental scales examined. The MPI+OpenACC version is initially $\sim$1.23$\times$ slower than the CUDA- and OpenCL-based versions but matches their performance as the experiments are scaled to larger node counts.

The results obtained from these strong-scaling experiments on Titan are also presented in Figure 6.7, in terms of the application speedup achieved relative to the performance of the OpenACC-based version. Analysing the results in this manner identifies an additional performance trend which was not evident in Figure 6.6 due to the scales of the chart. In this format the results show that initially the most performant configuration is MPI+CUDA closely followed by the MPI+OpenCL version, in the 64 node experiment these respectively deliver 1.23$\times$ and 1.16$\times$ superior performance relative to the MPI+OpenACC based approach. As the scale of the experiments is increased, however, the relative performance of the MPI+CUDA and MPI+OpenCL codebases decreases such that they are approximately equal to that of the MPI+OpenACC codebase in the 1,024 to 4,096 node experiments. Additionally, in the 8,192 node experiment the OpenACC-based approach outperformed both the CUDA- and OpenCL-based

approaches by ∼1.3× on average. This performance disparity is likely to be due to the explicit block-sizes employed within the OpenCL and CUDA versions being significantly sub-optimal for the smaller mesh-sizes per node which occur in the experiments at this scale. The block-sizes employed in these versions were previously derived during experiments with larger mesh-sizes per node at smaller node counts. The OpenACC-based version does not explicitly specify block-sizes and therefore the runtime system is able to select a configuration during application execution, which it estimates will be most appropriate for the size of mesh currently being simulated per GPU in the particular experiment.

It is also evident from these results that the performance advantages demonstrated by the MPI+CUDA configuration over the OpenCL-based approach, in the initial smaller node count experiments, decrease significantly as the scales of the experiments are increased. In the larger scale experiments, in which application performance is less computationally bound, the performance of both the CUDA- and OpenCL-based codebases is virtually identical.

The results presented in Figure 6.6 also facilitate a performance comparison between the CPU-only and GPU-based architectures of the Archer and Titan platforms. In the 64 node experiments the MPI-only version executing on the Archer platform is ∼1.93× slower than the MPI+CUDA implementation executing on the Titan platform. This demonstrates the performance advantages which utilising the Nvidia K20X GPU architecture can have over the Intel Xeon CPU processors. This performance disparity was also achieved despite 2× more CPUs (2 per node) being employed, compared to the experiments on the GPU-based architecture which only contains 1 GPU per node. As the scales of the experiments are increased, however, and the performance of the application in this configuration becomes increasingly communication bound, this trend changes significantly. Between the 256 and 512 node experiments the performance of the MPI-only codebase executing on the CPU-based architecture starts to deliver significant performance advantages, relative to the GPU-based approach. This performance advantage increases in the higher node count experiments and reaches ∼3.5× in the 2,048 node experiment. This is due primarily to the relatively slow performance of the PCIe bus which connects the GPU devices to the host nodes. As the scales of the experiments are increased, the amount of computation performed per node decreases, and the performance of the application becomes increasingly communication bound. In this scenario the time taken to move data across the PCIe buses therefore starts to dominate the overall performance of the application. The CPU-based architecture does not exhibit this problem and therefore is able to deliver superior scalability for applications in this experimental configuration.

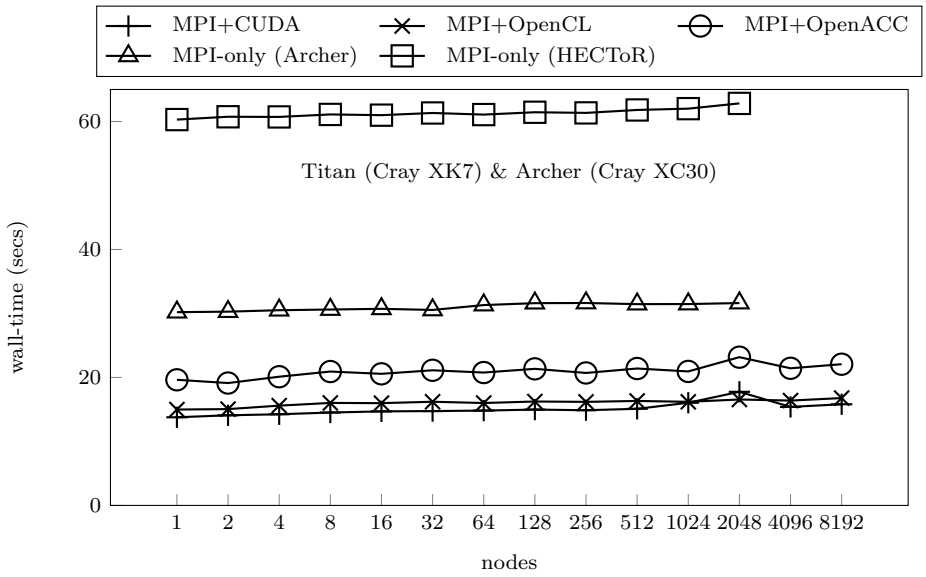


Figure 6.8: Weak-scaling performance ($3{,}840^2$ cell/node problem)

Weak-scaling Results Analysis

To assess the performance of the MPI+OpenCL programming model in a weak-scaling experimental configuration a series of experiments was conducted on the Titan, Archer and HECToR supercomputers. These employed the $3,840^2$ cell problem from the standard CloverLeaf benchmarking suite, which was scaled from 1 to 8,192 nodes in a manner such that each node processed a local mesh-size of $3,840^2$ cells. Figure 6.8 presents the results of these experiments.

The results obtained from the GPU-based architecture of Titan show that the performance of the MPI+OpenCL programming model is generally within 6-8% of the native MPI+CUDA version of the codebase. Furthermore these results demonstrate that the performance of the OpenACC version which utilises a directive-based programming model is consistently $\sim$1.4$\times$ slower than the versions which utilise the more explicit GPU-programming approaches of OpenCL and CUDA. The chart also demonstrates the significant performance advantages which the GPU-based architecture of Titan can deliver, for hydrodynamics applications such as CloverLeaf, compared to the CPU-only architectures of Archer and HECToR.

These results demonstrate that relative to the MPI-only programming model on the Archer and HECToR platforms, utilising the MPI+CUDA or the MPI+ OpenCL programming models on the GPU-based architecture of Titan, delivers performance advantages of $\sim$2$\times$ and $\sim$4$\times$ respectively. Furthermore, the newer Intel Xeon E5-2697 CPU devices and the Aries interconnect within the Archer platform (Cray XC30 architecture) provides a $\sim$2$\times$ performance advantage

for the MPI-only version of CloverLeaf, compared to the older AMD Opteron 6276 CPU devices and the Gemini interconnect technology available within the HECToR platform (Cray XE6 architecture).

Additionally as the scale of the experiments is increased the performance disparities between the various versions are consistently maintained. This is due to the fact that large mesh-sizes are simulated per computational node, during each of the experiments conducted in this weak-scaling configuration. Consequently the performance of the application remains computationally bound and thus the GPU-based architecture and the explicit programming approaches of CUDA and OpenCL are able to maintain their performance advantages throughout the series of experiments.

6.4 Summary

This research has demonstrated that the use of OpenCL enables an application to be expressed in a single codebase in such a manner that it is possible to execute it on a wide range of current state-of-the-art processor architectures. It is currently not possible to achieve this with the other programming models examined in this research, the CUDA implementation is effectively confined to the Nvidia GPU devices and the OpenMP implementation to the Intel and AMD CPU devices. Additionally, the use of OpenCL enabled the application to be executed on the GPU component of the AMD Trinity-APU devices which it would otherwise not have been possible to utilise. Use of the OpenCL technology can therefore significantly improve the portability of application codebases across diverse processor architectures from multiple vendors.

The results produced as part of this research also show that it is possible for the performance of OpenCL applications to match and sometimes exceed that of their equivalent native implementations. In the experiments conducted on the Nvidia architecture the performance of the OpenCL codebase is always within 6-8% of that of the equivalent CUDA implementation. In particular experimental scenarios, however, the OpenCL implementation delivered superior performance by as much as 1.6%. On the Intel CPU-based architectures the performance of the OpenCL implementation was significantly worse than that of the native OpenMP implementation, with the performance discrepancy reaching as high as a $\sim$20% slowdown. The results obtained during the experiments on the AMD CPU processor (Opteron 6272), however, show that in particular experimental scenarios the OpenCL implementation can outperform the native OpenMP implementation by as much as 30%, whilst in other configurations the native implementation can be as much as 16.4% more performant. Similarly, in the experiments with the larger $3{,}840^2$ cell problem size on the Xeon Phi

co-processor architecture the performance of the OpenCL codebase is generally within 3% of that of the native implementation (in *"offload"* mode) and can be as much as 11.5% more performant in particular configurations (e.g. when 4 threads per core are utilised). Additionally in the experiments with the smaller 960^2 cell problem, the OpenCL implementation produced as part of this research was ∼66% more performant than the native OpenMP version, when this was utilised in the *"offloading"* configuration.

Achieving these performance levels, however, generally required device specific optimisations to be implemented and therefore performance cannot necessarily be regarded as being portable across multiple architectures. The results presented in Section 6.3.1 show that on the Nvidia K20X architecture the most effective optimisation techniques were the `NDRange` padding, fixed local *work-group* sizes, merging *kernels*, pre-processing constant values, utilising the `restrict` & `const` keywords and the auto-tuning of local *work-group* block-sizes. The use of array notation, *out-of-order* command queues, overlapping computation with data movement and processing multiple grid-point per *work-item* techniques did not deliver any performance benefits. On the Xeon Phi platform, however, the most successful optimisation techniques included the `NDRange` padding, fixed local *work-group* sizes, pre-processing constant values, applying the `restrict` & `const` keywords as well as the auto-tuning of the local *work-group* block-sizes. The use of array notation, processing multiple grid-point per *work-item*, merging *kernels*, *out-of-order* command queues and overlapping computation with data-movement optimisations were ineffective and often resulted in significant reductions in overall performance on this architecture. All of the candidate optimisation techniques examined in this research negatively impacted the performance of the OpenCL codebase on the Intel Xeon CPU architecture. On the AMD Opteron architecture, however, the optimisation of processing multiple grid-points per *work-item* and the use of an *out-of-order* command queue delivered significant performance benefits, whilst the other candidate optimisation techniques were almost completely ineffective.

This work also demonstrated that the selection of appropriate local *work-group* block-sizes is crucial in order for the performance of an OpenCL application to be maximised. The use of auto-tuning techniques to determine the optimal configuration was examined as part of this research and shown to be an extremely effective approach. The optimal block-sizes identified through the use of these techniques showed significant variations both between the various processor architectures and also across the individual OpenCL *kernels* within the application. This complexity further supports the use of auto-tuning as an effective technique for the identification of such optimal configurations.

The multi-node experiments identified that in a weak-scaling experimental

configuration on the Nvidia GPU based architectures, the OpenCL programming model consistently maintains a 6-8% performance deficit relative to an equivalent native CUDA implementation, in all of the experiments between the 1 and 8,192 node cases. In this configuration the experiments showed that utilising an explicit low-level programming model such as OpenCL can deliver performance improvements of up to 1.4× relative to the higher-level directives based approach of OpenACC. Additionally for this class of application, targeting the Nvidia K20X GPU-based architecture of Titan via a programming model such as OpenCL can also deliver performance improvements of up to 2× relative to current state-of-the-art CPU-only based architectures such as Archer.

This research also showed that in a strong-scaling experimental configuration the use of the OpenCL built-in functions to transfer data to/from the Nvidia K20X GPU devices on the Titan platform can deliver significant performance advantages compared to approaches which employ explicit buffer management *kernel* functions. Additionally, although the OpenCL programming model can deliver performance advantages over the directive based approach of OpenACC when the mesh size per node is relatively large, the performance of the OpenCL implementation can be inferior to that of OpenACC at large scale. This is likely to be due to the fact that the OpenACC implementation employed in these experiments is able to select more appropriate *kernel* block-sizes at execution time. The experiments conducted here also demonstrated that in a strong-scaling scenario the performance of the OpenCL based implementation can match that of the native CUDA implementation as the scales of the experiments are increased and the computational workload per node reduces.

Overall the improved portability which OpenCL offers for, in some cases, only relatively small performance penalties may be an extremely attractive trade-off for HPC sites as they attempt to cope with ever increasing workloads and a myriad of complex programming models and architectures. At least in these experiments, solely utilising OpenCL cannot, however, guarantee the delivery of full portable performance for scientific applications.

CHAPTER 7
Evaluating FPGAs as Low Power Processing Solutions

This chapter commences, in Section 7.1, with a discussion of the factors which motivate this research together with an overview of existing related work. The research undertaken to develop implementations of certain CloverLeaf kernels which can effectively execute on an Altera FPGA device is then discussed in Section 7.2. Detailed descriptions of the candidate optimisations which have been examined as part of this research are also presented in Section 7.2.1. Section 7.3 documents the actual experiments conducted to evaluate an FPGA device as a potential processing solution for hydrodynamics applications, as well as the suitability of OpenCL as a programming model to enable scientific applications to be targeted at these devices. In Sections 7.3.2 and 7.3.3 time- and energy-to-solution analyses are also presented which examine the performance of the technology against several alternative state-of-the-art processing solutions. Finally, Section 7.4 summarises the findings of this research and concludes the chapter.

7.1 Related Work

FPGA technology has existed for several decades, although the applicability of the technology for scientific workloads has been limited due to the complexities associated with the low-level HDLs required to utilise them [124]. Recent advances are, however, enabling applications to be targeted at these devices through high-level programming models. Historically the attainable floating point arithmetic performance of FPGA devices has also been limited, although this is considerably improved on existing state-of-the-art devices. Altera recently announced that their latest *Generation 10* solutions will be able to natively support IEEE 754 compliant single-precision floating point arithmetic using dedicated hardware circuitry within each FPGA DSP block [8]. FPGAs also possess significant internal memory bandwidth resources and their memory access sub-system can be defined for specific applications [124]. The fact that the majority of the hydrodynamics applications of interest to this research are memory bound, means that FPGAs are potentially a well-suited processing solution for these applications. Additionally their capabilities have increased such that it is now possible to create complex SOC designs within one FPGA device [158]. Intel also recently announced that it plans to incorporate an FPGA into future versions of its Xeon products [99].

Several researchers have also documented significant successes through the application of these technologies. Lindtjorn *et al.* present work using an FPGA-based approach to accelerate a reservoir simulation application within the oil and gas sector, which employs a finite volume method [125]. They created data-flow representations of their key algorithms of interest and targeted a Xilinx-based system using the Maxeler compiler tool-suite [137]. Additionally, they also present novel techniques, made possible by the use of an FPGA-based architecture, to make better use of the available memory bandwidth resources. Ultimately they achieve significant speedups in performance (20-70×) and power efficiency (7.5-28×) over equivalent CPU- and GPU-based solutions.

Bose *et al.* present work which examines using a coarse-grained reconfigurable processor, based on an ADRES design [30] from Samsung, to implement a 3D physics engine used within the computer gaming industry [29]. They highlight the efficiencies of using a reconfigurable processor over a standard ARM-based microprocessor design and present results showing a significant performance advantage, although they do indicate that they use a fixed point arithmetic implementation. Brossard *et al.* also describe their research to develop a high level language and compilation system which is able to target scientific applications at FPGA-based platforms, however, their work concentrates on a genomics based application [33].

A methodology for utilising OpenCL as a programming model for FPGA devices is presented by Economakos [61]. This is based on the use of the CatapultC ESL technology and documents several low-level design issues, as well as a significant performance improvement over a GPU-based processing solution. Lin *et al.* also present research which examines the construction of an OpenCL compiler and runtime system, based on LLVM, for FPGA devices [124]. They achieve equivalent performance compared to a GPU device but with a significant improvement (5×) in power consumption, although the application they examine is significantly different to hydrodynamics applications, which are the focus of this research. Similarly, Owaida *et al.* developed a compilation framework which, like the Altera tool-chain, is able to translate potentially unmodified OpenCL to Verilog, in order to target FPGA devices [158]. Their work identifies a series of optimisations and assesses the success of their techniques on a range of benchmarks, some of which are predominantly floating point arithmetic based. They do not, however, compare the performance of the FPGA devices to alternative processing solutions.

These developments, together with the need to find more power efficient processing solutions for hydrodynamics applications, motivate the research documented here. In particular this work aims to determine whether FPGAs can deliver sufficient levels of *floating-point* computational performance, whilst

adequately driving the memory subsystem, and consuming substantially less energy compared to more established processing solutions. It also seeks to determine whether OpenCL represents an appropriate level of abstraction for expressing algorithms, and how efficiently the model enables them to be targeted at FPGA devices. Additionally, it also examines how kernels should be expressed in order for the Altera tool-suite to optimally target them at this architecture. Information on the Altera OpenCL tool-suite can be found in the Altera Programming [7] and Optimisation Guides [6].

7.2 FPGA Targeted OpenCL Implementations

The CloverLeaf kernels, documented in Chapter 6, were primarily developed to target GPU-based devices and consequently maximise the number of OpenCL *work-items* launched within each `NDRange` index space. Each kernel employs a strategy which utilises one OpenCL *work-item* to process each mesh grid-point. These implementations are referred to as the *reference* versions within this research. To determine the optimal approach for implementing computational kernels to target Altera FPGA devices a series of candidate optimisations, which are documented in Section 7.2.1, were applied to the codebase and their utility subsequently analysed.

7.2.1 Optimisations Examined

The following sub-sections each document in detail a specific optimisation technique which has been examined as part of this research.

Resource Driven Compiler Optimisations

The ability of the Altera OpenCL compiler to perform automatic resource driven optimisations of the individual kernels was examined as part of this research. Through this process the compiler attempts to improve performance (e.g. the number of *work-items* executed / second) by iteratively varying a number of compilation parameters or *design-points*. These include the selection of optimal vectorisation widths, the number of instantiated *compute-units* and the level of resource sharing within a kernel. This optimisation is enabled by specifying the `-O3` option to the compiler, which by default automatically tunes kernel implementations to use a maximum of 85% of the logic area available on the target FPGA. Employing this technique facilitates an assessment of the ability of the Altera compiler to automatically optimise existing kernel functions and provides a useful baseline against which to assess the utility of additional, more

complex, optimisation techniques. Versions which employed this approach are denoted using the abbreviation *O3* within their descriptions in Section 7.3.

Work-group Size Optimisations

To determine whether kernel performance on FPGA devices is affected, by different OpenCL *work-group* settings, in a similar manner to the results presented in Section 6.3. This research examined a number of *work-group* size related optimisations by applying the `reqd_work_group_size` and `max_work_group_size` kernel attributes to individual computational kernels. These attributes purport to enable the compiler to perform more aggressive optimisations and enable it to generate hardware configurations which exactly match the required number of *work-items* per *work-group*. This potentially leads to resource savings and more efficient computational pipeline implementations [6]. In Section 7.3 versions which employed both the `reqd_work_group_size` and `max_work_group_size` attributes are denoted by the description $wgA{\times}B$, in which characters A and B represent the dimensions of the specified *work-group*.

Kernel Vectorisation

Vectorising individual kernel routines potentially enables higher throughput to be achieved through the creation of pipelines which process multiple mesh points simultaneously in a SIMD fashion. The Altera OpenCL compiler supports both automatic and manual kernel vectorisation techniques.

Automatic vectorisation is performed exclusively by the compiler and can translate scalar operations within each kernel to SIMD operations. The vectorisation factor for these operations is specified via the `__attribute__((num_simd_work_items(X)))` directive. This technique also requires a specific *work-group* size to be specified for the kernel and for this value to be an exact multiple of the value of X. It does not, therefore, require any changes to the code within the actual bodies of the *kernel* functions and enables multiple *work-items* to be executed simultaneously in a SIMD manner, with each instantiated vector-lane processing one *work-item*.

Manually vectorising kernels, however, requires the explicit use of OpenCL *vector datatypes* and the modification of the `NDRange` index space used to launch a particular kernel. Under this approach each OpenCL *work-item* processes additional mesh grid-points, depending on the width of the *vector datatype* employed (e.g. `float4`, `double8` etc), with these again being executed simultaneously in a SIMD manner. Incorporating this optimisation technique within the kernels examined in this work also required the width of the data-arrays processed by each kernel to be increased, such that their width was padded to

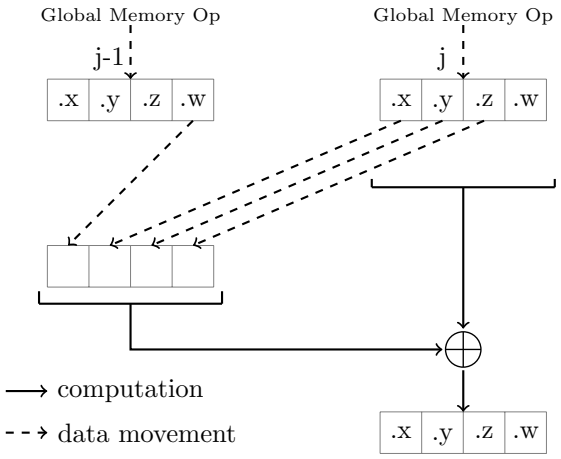


Figure 7.1: Vector shift operation implemented within the FPGA

be an exact multiple of the vector-width of the particular datatype being utilised. It was also necessary to include *work-item* dependent branches to ensure that *halo*-cells at the edge of each data-array were processed correctly. Generally, this required additional `if`-tests to be employed to ensure that particular *work-items* only update the required vector elements (e.g. `variable.x`). Within Section 7.3 versions which utilised this optimisation technique are denoted by expressions of the form *datatypeX*. Here *X* refers to the particular vector-width employed and *datatype* refers to the particular datatype used in the execution e.g. `double`.

Kernel vectorisation also facilitates the coalescing of global memory (DDR) operations in order to further improve application performance. Memory accesses will be coalesced, under an automatic vectorisation approach, if the Altera compiler is able to identify a sequential memory access pattern within the kernel. The explicit use of vector datatypes, however, guarantees that memory accesses will be statically coalesced by the compiler.

Due to its regular memory access pattern and no stencil operations, manual vectorisation was implemented for the *Ideal-gas* kernel by increasing the size of each global memory operation proportionally with the width of the vectorisation. Implementing this candidate optimisation technique within the *Accelerate* kernel was, however, significantly more challenging due to the 4-point stencil operation required within the kernel.

Two general approaches for implementing this candidate optimisation technique within the *Accelerate* kernel were examined as part of this work. For an arbitrary OpenCL *work-item* co-ordinate `[k][j]`, both approaches perform additional global memory accesses, from each of the required source data arrays, at the following relative co-ordinates: `[k][j-1]`, `[k-1][j-1]` and `[k-1][j]`.

The first approach, which is labeled *indivLanes* within Section 7.3, is based

on the use of OpenCL *vector datatype* subscripts. These were employed within the kernel, to enable vector elements of particular variables to be individually accessed. Each calculation within the original kernel implementation was then modified such that it is expressed as an *Elemental Function*. These functions were then duplicated within the kernel source code, once for each element of the new vector datatypes. The executions of these *Elemental Functions* can potentially occur in parallel as each updates a unique individual vector subscript e.g. `.x`, `.y` etc.

Rather than accessing individual vector elements and duplicating calculation logic within the source code of a particular kernel, an alternative approach was also implemented. This applied the original kernel logic once, in a vectorised manner, to the OpenCL vector datatypes within the new *Accelerate* kernel implementations. The new approach required *shift* operations to be employed, in order to align the required data-items in additional vector variables/registers, and for the calculations to operate collectively on the entire vector datatypes. Figure 7.1 demonstrates how this arrangement was implemented within the CloverLeaf *Accelerate* kernel. Implementations which employed this approach are denoted by the description *vectorOps* within Section 7.3.

Multiple Compute Units

Utilising multiple compute units on the FPGA device, in order to potentially improve the overall throughput of kernels, was also examined as part of this research. Individual compute units each contain a unique complete computational pipeline and are thus able to execute different OpenCL *work-groups* from the current kernel, these are dispatched in parallel by the hardware scheduler. The number of compute units generated for a particular kernel must be manually configured using a directive of the form `__attribute__((num_compute_units(X)))`, where X specifies the particular number of compute units. Each instantiated compute unit will occupy additional area on the FPGA device and increase global memory bandwidth contention. This potentially requires increased logic/bandwidth utilisation to be traded against overall kernel throughput. Versions which employed this candidate optimisation technique contain the word *XcUnits* within their descriptions in Section 7.3, here X refers to the number of compute units actually instantiated.

Minimising Global Memory Operations

It is recognised that reducing the number of global memory operations can significantly increase available bandwidth resources and improve kernel performance [6]. This optimisation was implemented for the kernels examined as part

of this research by identifying occurrences within each kernel which duplicate references to global memory. These accesses were subsequently re-factored such that the required data was pre-loaded from global memory into on-chip memory resources within the FPGA, as early as possible during the kernel execution pipeline. In this research kernels were restructured to pre-load data into `private` OpenCL memory objects and to perform their computations directly on these objects, with data only being written out to global memory when absolutely necessary. OpenCL `private` memory objects are generally mapped to FPGA registers, which are a plentiful hardware resource, by the compiler thus maximising performance. This optimisation enabled data to be reused for multiple calculations within the kernel pipeline, without global memory having to be re-accessed, before final results were eventually written back to global memory. Implementations which incorporated this optimisation are referred to using the description *minMemOpts* within Section 7.3.

To examine the effect on performance due to the location of global memory operations within the *Accelerate* kernel, additional implementations were developed. In these modified versions global memory operations were relocated to occur as early as possible within the kernel, in order to ensure that the latency of any data movement operations was minimised and to maximise the opportunities for overlapping these operations with computation. Versions which incorporated this candidate optimisation technique are denoted by the description *EarlyMemOps* within Section 7.3.

Larger Calculations through Reductions in Temporary Variables

The reference implementation of the *Accelerate* kernel also utilises several temporary variables to hold intermediary result values during its execution. To potentially improve the performance of the kernel an additional version was developed which eliminated these temporary variables. This modified version also consolidates the calculations within the kernel into two large expressions, each of which updates a particular final output value produced by the kernel. Implementations which incorporated this candidate optimisation technique are denoted by the description *RemoveTemps* within Section 7.3.

Modifying All Elements of Vector Datatypes

Due to the required *halo*-regions, the CloverLeaf data arrays often do not match the exact vectorisation width employed within a kernel. Kernels which utilise vector datatypes therefore, often required the inclusion of additional branching operations in order to ensure that only the required target array elements are updated correctly. This frequently occurs when only particular elements of a

vector datatype need to written back to global memory, which also prevents the coalescing of memory operations.

To eliminate these branching operations and ensure that only full vector datatypes are written to/read from global memory this research examined a further potential candidate optimisation technique. This involved modifying the kernels to accumulate results in temporary vector variables/registers, which could then be written back to global memory using a single coalesced memory operation. Often this required individual vector element updates to be employed and an additional global memory operation to allow particular memory locations to be updated with their original contents. The use of the temporary "*accumulation*" registers enabled branching operations to be minimised and for calculations to generally operate on entire vector datatypes within FPGA device memory. These individual vector-element operations are therefore confined to only update the actual "*accumulation*" registers, and not global memory locations. Implementations which employ this candidate optimisation technique are denoted by the description *AllVector* within Section 7.3.

Partitioning Global Memory

The global memory resources, available on the FPGA-based system examined in this research, are by default accessed in a burst-interleaved manner. In this configuration global memory references are interleaved across the available memory banks, which leads to memory capacity usage being efficiently balanced across the available banks. Configuring these memory banks into separate, contiguous, non-interleaved memory regions can, however, potentially improve access load balance and therefore performance.

This candidate optimisation technique was implemented by allocating each OpenCL *buffer* object, accessed by a particular kernel, to a specific memory bank using the proprietary Altera memory object creation flags (e.g. `cl_mem_bank_1_altera`). As the majority of these *buffer* objects are effectively of equal size, load balance was ensured by allocating equal numbers to each memory bank. *Buffers* were also distributed such that the read and write operations performed by each kernel to global memory, were distributed as evenly as possible across the available banks. Implementations which employed this technique are denoted using the description *memPart* within Section 7.3.

Floating-point Mathematics Optimisation Options

To potentially improve the efficiency with which *floating-point* mathematics operations are implemented on the FPGA devices, this research examined the effect of several *floating-point* based optimisations available with the Altera

compiler. These included allowing the compiler to create more *balanced*-trees of *floating-point* operations. This achieves efficiencies by shortening the overall length of the computational pipeline, whilst potentially also reducing calculation accuracy, as this optimisation is not compliant with IEEE standard 754-2008. Implementations which employed this candidate optimisation include the description *fprelax* within Section 7.3.

Additionally, postponing rounding operations until the end of the *floating-point* calculations through the generation of *fused* operations, was also examined. This optimisation again potentially violates IEEE standard 754-2008 and enables hardware resources to be re-purposed away from rounding operations. It does, however, enable additional precision bits to be carried forward through the *floating-point* calculation, potentially leading to more accurate results. Versions which employed this candidate optimisation techniques are denoted using the acronym *fpc* within Section 7.3.

Minimising *Floating-point* Operations

As the *floating-point* capabilities of existing FPGA devices are still limited relative to established processing technologies, a series of optimisations to limit the number of *floating-point* calculations within individual kernels were also examined. This involved re-expressing the algorithm such that the number of *floating-point* calculations was minimised, in some instances temporary `private` variables were employed to remove the need to re-calculate particular intermediary values. Thus providing the compiler with the greatest possible opportunity to generate a computational pipeline which minimised the number of *floating-point* operations. In Section 7.3 implementations which incorporated this optimisation are referred to using the word *redFlops* within their descriptions.

Additionally, implementing the vectorisation optimisation techniques described previously, potentially results in redundant computation occurring within the *Accelerate* kernel, during the execution of the final iteration of its inner loop. This occurs whenever the width of the problem domain being simulated is not evenly divisible by the length of the vectorisation employed within the kernel. To potentially alleviate this problem additional `if-then` statements were inserted into the kernel to remove the redundant computations. This significantly reduces the number of floating point calculations performed during the execution of the final iteration of the inner loop within the kernel, at the expense of inserting additional branching operations during each iteration. It was, however, only possible to implement this candidate optimisation technique for kernels based on the *indivLanes* style of vectorisation. Within Section 7.3 versions which incorporated this technique are denoted by the description *min-*

FinalComp.

To remove the requirement for the additional branching operations, a further version was developed which completely peeled the final iteration from the main kernel loop structure and explicitly inserted the additional logic operations immediately after it. This enabled the branching constructs to be removed from the main kernel loop and for the redundant computational logic to be removed from the peeled iteration. Versions which incorporated this candidate optimisation technique are denoted by the description *peelFinalIt* within Section 7.3.

Removing Kernel Bounds Checks

The reference implementation of the OpenCL kernels employed an approach which performed an array bounds check (using an `if-then` construct) at the start of each kernel. This verified whether each *work-item* was required to process a particular mesh grid-point and ensured that individual data-arrays were not accessed beyond their bounds. This enables kernels to be launched with `NDRange` index spaces which are specific multiples of the preferred vector width of the current device, and potentially greater than the dimensions of the individual data-arrays. This approach facilitates performance improvements on certain processing architectures, however, reducing *work-item* dependent branching is recognised as a method for improving performance on FPGA devices [6].

To examine the most performant method for constructing and launching kernels on Altera FPGA devices, additional implementations were therefore developed. These reduced or completely eliminated the *work-item* checks at the beginning of each kernel. This also required the *host* application to be modified to ensure that particular kernels were only launched with the exact `NDRange` dimensions required for their correct execution. In Section 7.3 versions which employed this candidate optimisation technique are denoted by the description *redBoundChecks*.

OpenCL `Local` Memory Based Cache

The stencil operations within the CloverLeaf *Accelerate* kernel necessitate that an arbitrary *work-item* requires access during its execution to data values which are offset, relative to the currently index coordinates (`[k][j]`), by 1 array element in both the x and y dimensions. Specifically, each *work-item* is required to read data from an adjacent memory location (`[k][j-1]`) and two contiguous memory locations which are offset by a large, but constant stride, relative to the current *(x,y)* coordinates of the *work-item* (`[k-1][j-1]` and `[k-1][j]`). In the reference implementation this potentially causes additional global memory

operations to be generated for each *work-item* index initiated.

To reduce these additional global memory operations and thus potentially improve kernel execution performance, OpenCL `Local` memory objects were employed. These were utilised to function as caches for data items read from global main memory by previously executed *work-items* and therefore facilitate the re-use of data values within subsequently executed *work-items*. OpenCL `local` memory objects are generally mapped to *on-chip* memory blocks within the FPGA fabric by the Altera compiler. Versions which utilised this candidate optimisation technique also required the implementation of explicit OpenCL `local` *work-group* sizes. Within Section 7.3 versions which incorporated this candidate optimisation technique are identified by the description *LocalMemCache*.

Array Notation

Additionally, employing array syntax (of the form `[k][j]`) to access the elements of the two dimensional array data structures within the kernels, instead of pointer arithmetic ($[k{*}x\_width{+}j]$), was also examined. Implementations which employed this candidate optimisation technique are denoted by the description *arrayNotation* within Section 7.3.

Single *Work-item* Execution

Implementations of the CloverLeaf *Accelerate* kernel, based on the "*single work-item*" OpenCL paradigm, were also developed. This candidate optimisation technique purportedly facilitates greater optimisation opportunities for the OpenCL compiler. The approach allows the entire execution flow of a particular kernel to be better analysed, which enables more efficient computational pipelines to be generated. Without this technique the compiler would generally only be able to analyse the execution of a singular "*elemental*" function, which is the case when `NDRange`-based kernels are utilised.

To implement kernels based on the "*single work-item*" paradigm, modified versions were developed which incorporated a nested double loop structure. This enabled each function invocation to execute the entire iteration space required by a particular kernel, which would previously have been specified using an OpenCL `NDRange`. The inner and outer loop iterated counts were therefore configured to be equal to the previously specified `NDRange` x- and y-dimensions, respectively. Additionally calls to the OpenCL runtime, within the kernel, to determine *work-item* index values were removed and the *host* application was also modified to enqueue each kernel invocation as an OpenCL *task* rather than as an `NDRange` *kernel*. Implementations which employed this candidate optimisation technique

are referred to by the description *SingleWI* within Section 7.3.

The Altera OpenCL compiler also produces a detailed optimisation report for kernels expressed in the "*single work-item*" paradigm. Through the examination of this report it was possible to identify that, although pipelined execution had been inferred for the *Accelerate* kernel, the execution of several code regions was being serialised due to the inclusion of an `if-then-else` construct. This clause was required to ensure that only data values within the particular problem boundaries were updated within global memory, in situations in which the problem size was not evenly divisible by the vectorisation width employed within the kernel. To eliminate this serialisation the kernel was re-structured such that full OpenCL vector datatypes were always written out to global memory. The `if-then-else` construct was also eliminated and replaced by two `if-then only` clauses. These enable the data values, which are to be written out to global memory, to be replaced when required by the original contents of the particular memory locations, which have previously been read from global memory. This ensures that only the required memory locations are updated with new data and enables the compiler to infer pipelined execution for the kernel without any serialised computational stages. Implementations which employed this candidate optimisation technique are denoted by the description *PipelineOpts* within Section 7.3.

Due to the requirements of the OpenCL standard, the execution ordering of the individual *work-items* within an `NDRange`-based kernel cannot be guaranteed or determined at compilation time. This necessitates that `NDRange`-based kernels must be implemented such that the execution of an individual *work-item* does not depend on the prior execution of other *work-items*. Consequently, individual *work-items* must therefore contain all of the global memory references which they require in order to complete their execution. This potentially results in the generation of additional global memory operations, that would otherwise not be required if a collective execution ordering could be guaranteed for the *work-items* within an `NDRange`, and data values could be reused between their executions.

Implementing stencil-based computations such as the CloverLeaf *Accelerate* kernel using the "*single work-item*" paradigm enables an ordering to be expressed between the execution of different loop iterations within the double nested loop structure. The execution of these loop iterations replace the actual individual `NDRange` *work-items*. This facilitates the implementation of several additional optimisations, including the reuse of data values across loop iterations and a reduction in the overall number of global memory operations performed per loop iteration.

The inner loop within the *"single work-item" Accelerate* kernel processes

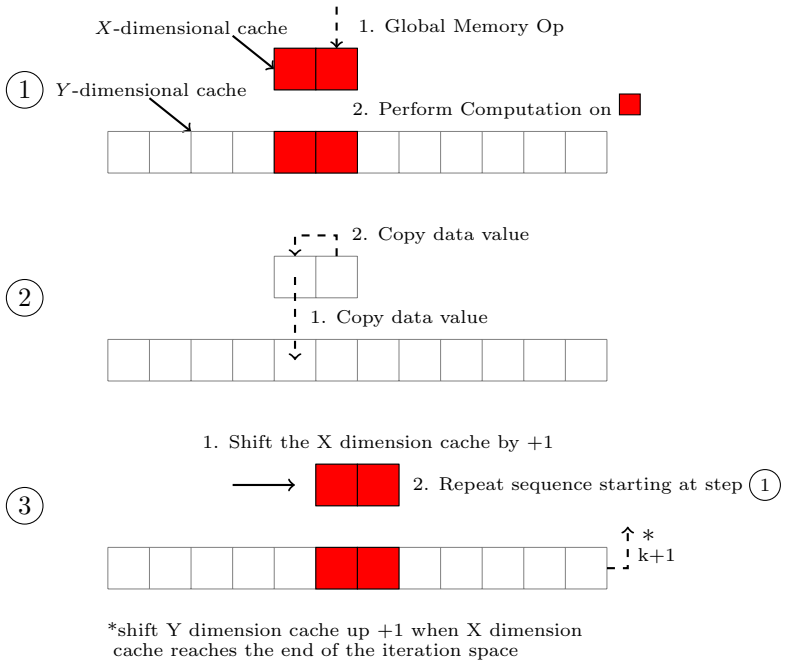


Figure 7.2: Data caching across loop iterations on the FPGA

the two dimensional data arrays sequentially in the x-dimension. An arbitrary iteration of this inner loop depends on several data values which were first accessed by the loop iteration immediately prior to it. The kernel was therefore modified to cache these values within temporary buffers, which facilitates their reuse within subsequent iterations, rather than reloading them again from global memory as would be the case with an `NDRange`-based implementation. These buffers were implemented using `private` OpenCL data objects to ensure that they were instantiated using the *on-chip* memory resources within the FPGA. The kernel was also modified to rotate the contents of these buffers at the end of each iteration, such that the most recent values read from global memory replace the previously cached values, ready for the next loop iteration to commence. Versions which implemented this optimisation technique across the x-dimension are denoted by the description *xDimBufferCache*, within Section 7.3.

A similar technique was also implemented for the y-dimensional memory accesses within the *Accelerate* kernel. This required data values to be cached and reused across different iterations of the outer k-loop and therefore necessitated the creation of several larger caches. These were implemented to be equal in size to the width of the problem domain being simulated and enabled data values to be cached across an entire execution of the inner j-loop. The caches were initially primed by reading in entire rows from the data arrays stored

within global memory prior to the execution of the main double loop-nest within the kernel. During kernel execution, as data values are removed from the smaller x-dimensional caches employed within the inner j-loop of the kernel, they are written to their corresponding location (in the x-dimension) within the equivalent, larger y-dimensional caches. This arrangement facilitated the reuse of these data values within the next iteration of the outer k-loop. Versions which implemented this optimisation technique across the y-dimension are denoted by the description *yDimBufferCache* within Section 7.3.

The combination of both the x- and y-dimensional data caching optimisation techniques is shown diagrammatically in Figure 7.2.

The implementation of this data reuse technique across the y-dimension of the kernel resulted in the Altera OpenCL compiler creating additional serialisation dependencies within the computational pipeline it generates. Overall, however, the implementation of these optimisations enabled the number of global memory load operations performed by the *Accelerate* kernel during each loop iteration to be reduced by 2.75×, from 22 to 8.

7.3 Results Analysis

To assess the computational performance and energy efficiency which an FPGA-based processing solution can deliver for hydrodynamics scientific applications a series of experiments were conducted as part of this research. Due to the time required to synthesise the hardware implementations of the kernels for the FPGA device an approach which utilised the *kernel-driver* routines, from the standard CloverLeaf software distribution package, was adopted. This enabled the performance of individual kernels to be examined in isolation and in a more time efficient manner. The *Ideal-gas* and *Accelerate* kernels were selected for these experiments as collectively they embody the key computational characteristics exhibited by the overall algorithm and also a much wider class of scientific applications. Specifically these kernels include no stencil and fixed stencil operations, respectively.

The *Tuck* platform (see Section A.1) was utilised in order to assess the performance of these kernels on an Altera Stratix V (D5) FPGA device and on a range of other state-of-the-art hardware accelerator/co-processor architectures. This system also contains the PowerInsight [119] monitoring technology which enabled the power consumption of each processing solution to be measured over time and for the corresponding energy-to-solution figures to be derived.

The experimental setup of the system software is further documented within Figure A.5. The experiments with the *Ideal-gas* kernel on the Stratix FPGA architecture utilised version 13.1 of the Altera OpenCL SDK, whilst version

14.1 was employed during all of the experiments with the *Accelerate* kernel. The GNU compiler (v4.4.6) was utilised for all experiments involving the Altera FPGA and Nvidia GPU devices, in order to compile the code which executed on the "*host*" CPU devices. Version 6.0 of the CUDA toolkit was also employed for all of the experiments involving the Nvidia GPU device. In the experiments on the Intel Xeon and Xeon Phi architectures, however, the Intel tool-suite was utilised. In particular version 15.0 of the Intel compiler was employed for all of the host-based software which executed on the Xeon CPUs, whilst version 2013 of the Intel OpenCL SDK was utilised for the OpenCL kernel code which targeted the Xeon Phi.

The experiments documented here examined the $3{,}840^2$ cell problem from the standard CloverLeaf benchmarking suite. During these the *Ideal-gas* and *Accelerate* kernels, were executed for 1,000 and 2,000 iterations, respectively. Section 7.3.1 analyses the performance of each kernel on the Altera Stratix V FPGA device and includes an analysis of the utility of the candidate optimisation techniques examined as part of this work. Additionally, time- and energy-to-solution analyses are presented in Sections 7.3.2 and 7.3.3 respectively, these examine the most optimal kernel implementations across a range of state-of-the-art processing architectures.

The results documented here were recorded from single executions of each particular experiment on the Altera FPGA platform. This approach was selected due to time constraints and an observation from an initially conducted set of experiments, which indicated that system noise levels on the FPGA architecture are negligible. Within this chapter the results presented from the experiments on the Nvidia GPU and Intel Xeon Phi architectures, however, are averages from three separate executions of each particular experiment.

7.3.1 Optimisations Analysis

The following sub-sections each examine the impact on performance due to the utilisation of specific candidate optimisation techniques, documented in Section 7.2.1, within the CloverLeaf *Ideal-gas* and *Accelerate* kernels.

Ideal-gas Kernel

The results from the experiments which examined the performance of the *Ideal-gas* kernel are presented in Figure 7.3. They show that the Altera Stratix V FPGA was able to execute 1,000 iterations of the reference implementation of the kernel in $64.62s$. Enabling the resource driven optimisations available with the Altera OpenCL compiler (*O3*), however, actually generated an implementation which performed fractionally worse than the original version. The experimental

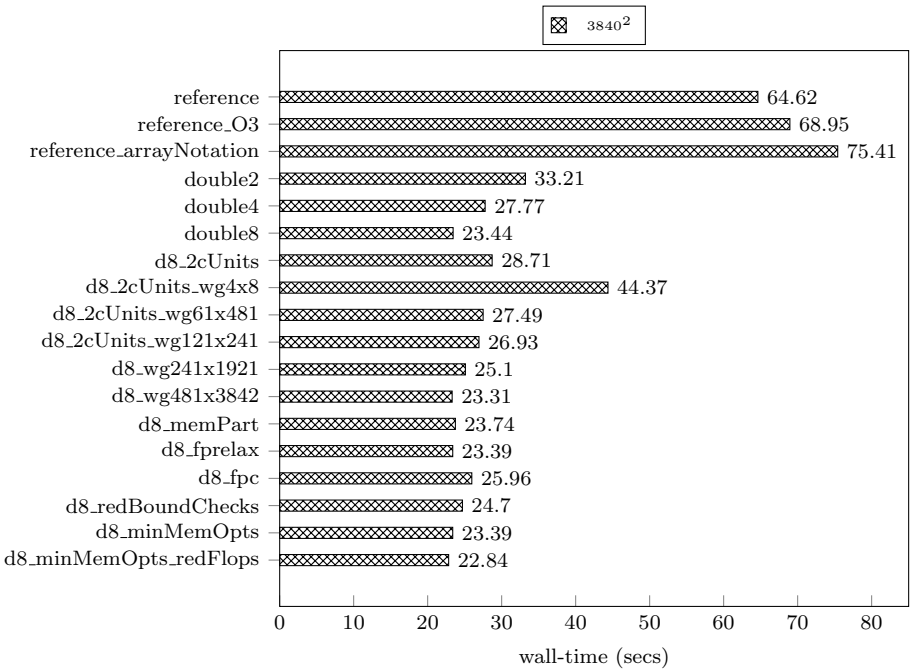


Figure 7.3: Optimisations to the *Ideal-gas* kernel on the Altera FPGA

results show a decrease in performance of 4.3s (6.69%) due to the use of this facility. Similarly, utilising array notation for the two dimensional array accesses within the kernel (*arrayNotation*) also resulted in a significant performance degradation of 10.79s (16.7%).

Applying the explicit vectorisation optimisation techniques (described in Section 7.2) to the reference implementation, however, delivered significant performance improvements. This research identified that it was possible to increase the vectorisation width, employed within the kernel, up to 8 `double` precision data elements. Each successive increase improving the overall performance of the kernel, although the results show that the performance improvements diminished as the higher vectorisation widths were implemented. In these experiments increasing the vectorisation width to 2 `double` precision elements reduced the overall execution time by 31.41s, a 1.95× increase in performance. The performance improvements due to the implementation of 4 and 8 element vectorisation widths, however, decreased to 1.20× and 1.18×, respectively. As the 8 element wide vectorised version (`double8`) was the most performant, this implementation was utilised in all subsequent experiments with this kernel. This is denoted by the description *d8* within Figure 7.3.

Implementing multiple computational units within the FPGA, each with an 8 element wide vectorised pipeline, to potentially improve the *work-item* throughput of the kernel ultimately proved to be unsuccessful. The experimental results

show that the implementation of this technique, using 2 computational units, increased overall execution time by 5.27s, a 1.22× decrease in performance.

To examine whether explicitly specifying `local` *work-group* sizes could improve the performance of the version of the kernel which employed multiple computational units, a series of additional experiments was conducted using a variety of configurations. The results, denoted by expressions of the form *wgA*×*B* in Figure 7.3, indicate that for this kernel it was not possible to improve the performance of the multi-computational unit version through the specification of explicit `local` *work-group* sizes. Each *work-group* size examined resulted in a performance degradation, and in some experiments these increases in execution time were substantial. Implementing a *work-group* size of 4×8 elements, for example, resulted in a significant performance degradation of 1.89× (20.9s), whilst the performance slowdowns due to the specification of the 61×481 and 121×241 *work-groups* were 1.17× and 1.15×, respectively.

A similar trend was also observed in the experiments which applied the `local` *work-group* candidate optimisation directly to the single computational unit implementations. In these experiments specifying a 241×1921 element `local` *work-group* increased the recorded execution time by ∼7.1%. The performance of the version which employed the 481×3842 element *work-group* was, however, virtually identical to that of the previously unmodified version, fractionally improving performance by <0.5%. These results indicate, therefore, that overall the *Ideal-gas* kernel does not benefit, in terms of performance on the Altera FPGA device, from the specification of a `local` *work-group* size. Additionally, employing *work-groups* which have large x-dimensions is generally the most optimal configuration. The results show that in these experiments the utilisation of small *work-group* sizes caused significant degradations in overall performance.

Partitioning global memory resources into separate, contiguous, non-interleaved memory regions was also unsuccessful in improving the performance of the *Ideal-gas* kernel. The implementation of this candidate optimisation technique (*memPart*) resulted in a fractional performance degradation of ∼1.3%.

Additionally, attempting to improve the efficiency of the *floating-point* mathematics operations generated by the Altera compiler, using the `fprelaxed=true` and the `fpc=true` compiler options, also did not improve the execution time of the kernel. The performance of the version which employed the `fprelaxed` option was practically identical (<0.2%) to that of the unmodified version, whilst the use of the `fpc` option degraded execution time by ∼10.8% (labelled *fprelax* and *fpc* respectively in Figure 7.3). Surprisingly, removing the array bounds checks within the kernel (*redBoundChecks*) also increased the overall execution time of the experiment by ∼5.4%.

The implementation of the candidate optimisation to minimise the number

of global memory operations within the *Ideal-gas* kernel, also had a negligible effect on overall performance, improving execution time by <0.2%. Reducing the number of floating point instructions within the kernel (*redFlops*), however, delivered a fractional improvement in performance, reducing the overall execution time in these experiments by ∼2.6%.

Overall, this research enabled the performance of the *Ideal-gas* kernel on the Altera Stratix V FPGA device, to be improved by ∼2.83×, relative to the original reference implementation. In these experiments the implementation of the explicit vectorisation techniques contributed most significantly to this improvement in performance.

Accelerate Kernel

The performance results obtained during the experiments with the *Accelerate* kernel are presented in Figure 7.4, whilst Table 7.1 documents additional profiling data which was collected on selected variants during this analysis. Figure 7.4 shows the execution time (in seconds) of the reference implementation of the kernel as the first entry in the chart. Subsequent entries document the effect on performance due to the incorporation of the candidate optimisation techniques (Section 7.2.1) examined as part of this research. Entries which commence with a + sign denote an optimisation technique which successfully improved the performance of the kernel and was taken forward within subsequent experiments. Thus an arbitrary version within the chart implicitly contains all of the optimisation techniques listed above it which commence with a + sign.

The results show that for the *Accelerate* kernel, unlike with the *Ideal-gas* kernel, the removal of the bounds checks within the kernel and the specification of exact `NDRange` dimensions actually fractionally improves performance by ∼1%. Applying the optimisation techniques to reduce the number of global memory operations (*MinMemOpts*) also improved the overall performance of the experiment by 10.6s (∼4.2%). Refactoring the *Accelerate* kernel, however, to reduce the number of temporary variables employed within the kernel, through the use of larger calculation sequences (*RemoveTemps*), actually fractionally reduced the performance of the kernel by 3.2*s* (∼1.4%) overall. This implies that structuring the computational kernels in this manner actually impedes the Altera OpenCL compiler in the generation of efficient computational pipelines for FPGA devices.

Explicitly vectorising the *Accelerate* kernel using the OpenCL vector datatypes also yielded significantly different results compared to those observed when these optimisation techniques were applied to the *Ideal-gas* kernel. Here specifying a vectorisation width of two `double` elements and using the *individual vector*

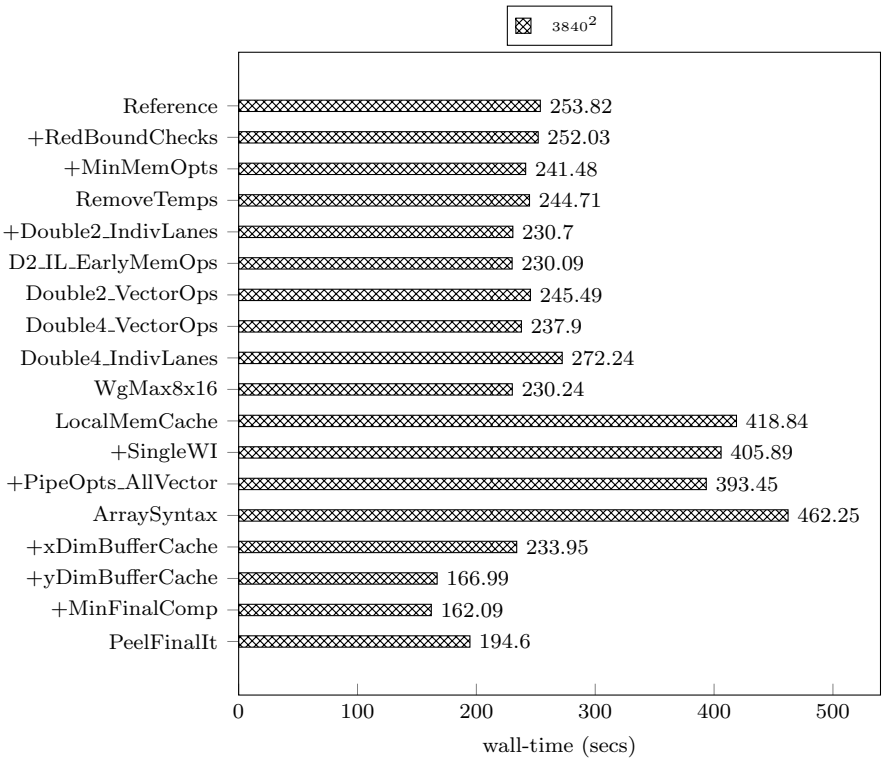


Figure 7.4: Optimisations to the *Accelerate* kernel on the Altera FPGA

lanes style of vectorisation (described in Section 7.2) within the kernel, reduced overall execution time by 10.8*s* (an improvement in performance of ∼4.5%). Switching the style of vectorisation, however, such that shift and full vector operations were utilised within the kernel (labeled *vectorOps* in Figure 7.4) actually fractionally degraded performance by 4.0*s* (∼1.7%).

This trend was reversed when the vectorisation width was increased to four `double` elements. In the experiments which examined the effect of increasing the vectorisation width, the version which utilised full vector operations (*VectorOps*) delivered a fractional performance increase of 3.6*s* (∼1.5%), whilst the application of the *individual vector lanes* technique (*IndivLanes*) resulted in a significant degradation in performance of 30.8*s* (∼12.7%). It was not possible to explicitly vectorise the *Accelerate* kernel using widths of >4 elements (`double8` or `double16` datatypes), as the Altera compiler generated implementations which required more hardware resources than were available on the Stratix V D5 FPGA.

Varying the location of the global memory operations within the kernel also resulted in no significant change in overall kernel execution time. The performance of the version which incorporated these optimisations (*earlyMem-*

Ops) was virtually identical to that of the unmodified implementation. Similarly, as with the *Ideal-gas* kernel, explicitly specifying a `local` *work-group* size (*wg8×16*) also resulted in no significant reduction in overall execution time, with performance remaining practically identical to the previously unmodified version.

Utilising OpenCL `local` memory constructs to create an "*on die*" cache within the FPGA fabric, in order to increase the reuse of data values and reduce the number of global memory operations, actually resulted in a substantial reduction in performance. In these experiments the performance of the version which incorporated this modification (*LocalMemCache*) was 188.6*s* slower than the equivalent unmodified implementation, a reduction in performance of ∼1.82×.

Solely refactoring the *Accelerate* kernel into the "*single work-item*" paradigm, whilst leaving its overall structure (e.g. the number of global memory operations) largely unmodified, similarly resulted in a significant decrease in overall performance. The version which incorporated this modification increased the overall runtime of the experiment by 175.2*s*, a slowdown in performance of ∼1.76×.

Restructuring the kernel in this manner, however, facilitated the implementation of several subsequent candidate optimisation techniques. In particular, implementing the optimisations labelled *PipelineOpts* and *allVector* within Section 7.2, reduced the levels of serialisation in the computational pipeline generated by the compiler, and improved the performance of the "*single work-item*" version by 12.4*s* (3.0%).

Reducing the number of global memory operations, by facilitating the reuse of data values between inner loop iterations, also significantly improved the performance of the "*single work-item*" based version of the kernel. The application of the *xDimBufferCache* optimisation reduced overall execution time by ∼1.68× (159.5s), such that overall execution time was now approximately equal to that of the equivalent `NDRange`-based implementation. Additional performance improvements were also observed, due to the application of the *yDimBufferCache* optimisation technique, which further reduced the number of global memory operations. This enabled data values to be reused between iterations of the outer loop in the y-dimension of the mesh, and delivered an additional 1.4× (66.9s) improvement in performance, reducing the overall runtime of the experiment to 166.99*s*.

Finally reducing the number of floating point operations within the final loop iteration, through the application of the *peelFinalIT* optimisation technique, also delivered a further reduction of 4.9*s* (2.9%) in execution time.

As with the *Ideal-gas* kernel, converting a "*single work-item*" based variant of the *Accelerate* kernel to use array syntax for the two dimensional array accesses

Version	Overall Kernel BW (MB/s)	Mem Op Av BW (MB/s)	Mem Op Av Stalls (%)	Comp Op Number of Stalls	Clock (MHz)	Av Write Burst	Av Read Burst
Reference	16,150	1,000.33	17.68	0	236.9	5	6
+D2_indivLanes	17,161	1,146.38	32.60	0	247.8	5	5
D2_VectorOpts	16,126	1,079.42	33.41	0	252.6	5	4
LocalMem	10,137	612.75	30.90	0	210.5	2	1
+SingleWI	13,388	548.00	53.15	0	195.0	3	2
ArrayNot	12,032	447.7	92.30	0	167.9	2	1
+X&YdimCache minFinalComp	13,870	1,370.38	40.46	0	184.7	6	5

Table 7.1: *Accelerate* kernel profiling statistcs on the Altera FPGA

also significantly reduced performance. The results from these experiments show that the application of this technique increased overall execution time by 68.80s, a slowdown of ∼1.17×. Furthermore, completely separating the final iteration from the main inner loop, to reduce the number of branching operations within the kernel, also substantially reduced overall performance by ∼1.2× (32.5s).

Overall, through the application of these modifications this research improved the overall performance of the *Accelerate* kernel, relative to the `NDRange`-based reference implementation, by ∼1.56×. The optimisations which delivered the most significant contributions to these performance improvements were the explicit vectorisation, *"single work-item"* and caching data values between loop iterations modifications.

***Accelerate* Kernel Profiling Statistics Analysis** Profiling statistics collected on the performance of several versions of the *Accelerate* kernel are shown in Table 7.1. These indicate that the performance of the *Accelerate* kernel is limited primarily by the memory subsystem available on the Altera Stratix V D5 FPGA utilised as part of this research, and not by the floating point computational performance available on the device. The number of pipeline stalls caused by compute-only operations within the kernel was 0 for each variant examined. Indicating that in each case only memory operations caused the pipeline to stall, and thus limited performance.

The results presented in Table 7.1 also show the effect of the optimisations examined for the *Accelerate* kernel. The average percentage of pipeline stalls, which an individual memory operation is responsible for, increases substantially by 1.84× as explicit vectorisation is implemented and by a further 1.24× due to the *dimBufferCache* and *minFinalComp* optimisations. This indicates that the performance of the optimised kernel variants is increasingly limited by fewer global memory operations.

Additionally, the average bandwidth consumed by an individual memory operation also increases significantly as the optimisations are implemented. Rising by 14.6% due to the incorporation of explicit vectorisation and by a further 19.5% when the *dimBufferCache* and *minFinalComp* optimisations are implemented. This suggests that the optimisations developed as part of this research facilitate the more efficient use of global memory resources and that kernel performance is improved through each individual memory operation being able to access a greater proportion of the available bandwidth resources. The overall global memory bandwidth consumed by the kernel also increased by 6.3% due to the explicit vectorisation (*D2_indivLanes)* optimisation. Interestingly, however, this decreased by 19.2% for the version which incorporated the *dimBufferCache* and *minFinalComp* optimisations, potentially indicating that global memory access latency may also be starting to limit kernel performance.

The profiling statistics also show that the operating clock frequency of the version which incorporated the *dimBufferCache* and *minFinalComp* optimisations is significantly lower than that of the *reference* and explicitly vectorised versions. This further indicates that the performance of the *Accelerate* kernel is memory bound, as the overall execution time of the former is ∼1.57× quicker.

Table 7.1 also shows that the introduction of the "*single work-item*" optimisation initially resulted in a substantial reduction of 21.99%, in the overall bandwidth utilised by the kernel and caused the average bandwidth consumed per memory operation to fall by 52.20%. The average read and write burst statistics were also reduced substantially from 5 down to 3 and 2 operations, respectively. Additionally, the utilisation of this candidate optimisation technique reduced the overall operating clock speed of the implementation by 22.80%. These reductions mirror the decrease in overall performance which was observed due to the incorporation of this candidate optimisation (Figure 7.4), further demonstrating why the introduction of this modification in isolation is not able to improve kernel performance.

Profiling statistics for the two alternative explicit vectorisation approaches, indicate that the *Individual Lanes* method of vectorisation facilitates the utilisation of >1.01 GB/s more overall bandwidth than the alternative *VectorOpts* implementation. The average number of read/write burst operations and the average bandwidth consumed per memory operation (>66 MB/s) metrics were also higher for this explicit vectorisation approach.

The overall clock speed achieved by the *Individual Lanes* method of vectorisation was, however, 5.6 MHz lower compared to the *VectorOpts* version. This indicates that as the *Individual Lanes* approach of explicit vectorisation delivered significantly superior overall performance, the memory focused metrics of overall kernel bandwidth, individual memory operation bandwidth and average

read/write bursts, are better indicators of the overall performance of the *Accelerate* kernel, compared to the more computationally focused statistics of e.g. clock speed. The fact that the operating clock speed of the implementation increased due to the incorporation of the *VectorOpts* explicit vectorisation approach, but overall performance decreased, further supports the assertion that the performance of the *Accelerate* kernel is limited by the performance/utilisation of the memory subsystem available on the FPGA architecture.

The reductions in overall performance observed in Figure 7.4 due to the adoption of *array notation* for the two dimensional data array accesses and the utilisation of OpenCL `local` memory constructs, are also reflected in the results presented in Table 7.1. The profiling statistics show that for the *Accelerate* kernel the use of *array notation* caused a reduction of ∼1.96 GB/s in the overall bandwidth achieved, a fall of 43.85 MB/s in the average memory bandwidth consumed by an individual memory operation, and a decrease of 28 MHz in the operating clock speed of the implementation.

The introduction of OpenCL `local` memory constructs to implement data caches in order to potentially reduce the number of global memory operations also resulted in similar reductions. Overall kernel bandwidth decreased by ∼6.86 GB/s (40%) due to the implementation of this modification, whilst the average bandwidth consumed per memory operation also decreased by 533.63 MB/s (46.55%). The average read and write burst operation statistics were also reduced significantly from 5 down to 2 and 1, respectively. Additionally, the operating clock speed decreased by 37.3 MHz due to the introduction of the `local` memory constructs.

7.3.2 Time to Solution Analysis

To assess the performance of the Altera Stratix V D5 device against a range of alternative state-of-the-art processing solutions a series of further experiments were conducted. These examined the performance of the *Ideal-gas* and *Accelerate* kernels on both the Nvidia K20 GPU and the Intel Xeon Phi 7120P architectures. Additionally, the performance of the Intel Xeon E5-2620 CPU architecture was also examined in a series of experiments with the *Accelerate* kernel. Each experiment utilised the kernel implementation which delivered the most optimal performance for the particular architecture. On the Intel Xeon CPU architecture a functional equivalent OpenMP-based version of the *Accelerate* kernel was utilised, whilst OpenCL-based implementations were employed in the experiments on all the other architectures. Figures 7.5 and 7.6 present the results of these experiments for the *Ideal-gas* and *Accelerate* kernels.

The results from both sets of experiments demonstrate an extremely similar

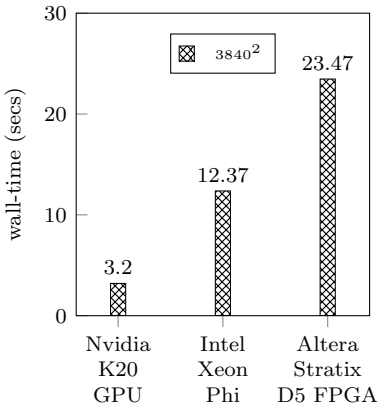


Figure 7.5: *Ideal-gas* kernel time-to-solution analysis

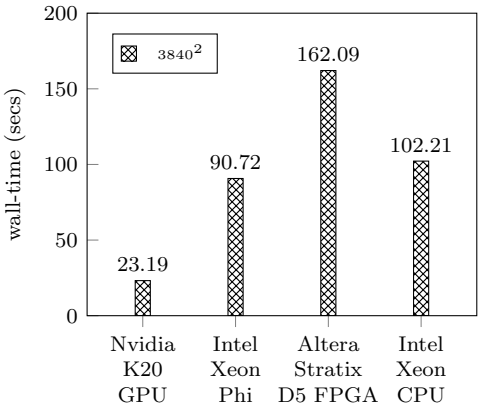
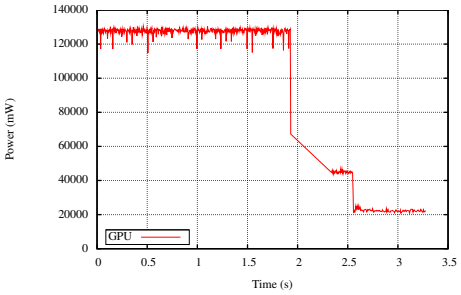


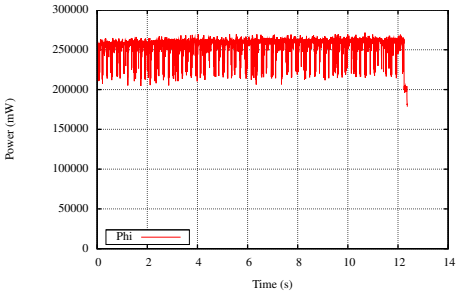
Figure 7.6: *Accelerate* kernel time-to-solution analysis

trend. In each case the performance of the Altera FPGA device is not able to match that of the Nvidia GPU and Intel Xeon Phi architectures. In the experiments which examined the *Ideal-gas* kernel the results show that the GPU and Xeon Phi delivered performance improvements, relative to the Altera Stratix D5 FPGA, of ∼7.3× and ∼1.9×, respectively. Additionally, for the *Accelerate* kernel the GPU and Xeon Phi architectures outperformed the Stratix D5 FPGA by ∼6.9× and ∼1.8×, respectively.

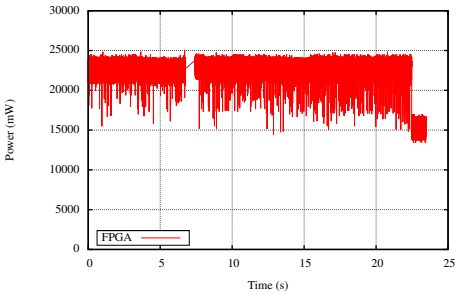
The theoretical global memory bandwidth available on the Nvidia K20 and Intel Xeon Phi 7120P architectures has been identified to be 208 and 352 GB/s, respectively [204]. The performance profiling tools available with the Altera OpenCL SDK also enable the maximum global memory bandwidth, which it is possible to achieve on the Stratix D5 based platform, to be determined. The profiling analysis conducted in Section 7.3.1, indicates that the maximum global memory bandwidth which it is possible to achieve on this platform, is 25.6 GB/s. This does not compare favourably with the global memory bandwidth resources available on both the K20 GPU and Xeon Phi devices. The Altera FPGA based platform examined in this research may therefore potentially have ∼8.1× and ∼13.8× less global memory bandwidth resources available to it than the Nvidia GPU and Intel Xeon Phi devices, respectively. This places the Altera Stratix V at a considerable computational disadvantage for executing this class of hydrodynamics applications, as previous profiling analyses have indicated that the performance of the *Ideal-gas* and *Accelerate* kernels are limited by the global memory subsystem access resources available on current platforms.



(a) Nvidia K20 GPU



(b) Intel Xeon Phi 7120P



(c) Altera Stratix V FPGA

Figure 7.7: Power consumption: *Ideal-gas* kernel

7.3.3 Energy to Solution Analysis

A series of further experiments were conducted to assess whether the Altera Stratix V D5 FPGA architecture could deliver any advantages in terms of the energy required to produce a solution. These involved re-executing the experiments described previously in Section 7.3.2. During these additional experiments, however, instead of only measuring the execution time of the application, the PowerInsight functionality (available within the *Tuck* platform) was utilised to measure the power (W) consumed by each particular processing solution throughout the execution of the application. This enabled the total energy consumed by a particular device, during the course of executing the application, to be derived by calculating the area under the power consumption trace recorded for the particular technology.

The results obtained for the *Ideal-gas* kernel during these experiments are presented in Figures 7.7 and 7.8. Figure 7.7 presents the power consumption traces which were collected during each experiment on the respective processing architectures, whilst Figure 7.8 documents the energy consumed by a particular processing solution in completing the overall computation. These charts show a considerably different trend to those observed previously in Section 7.3.2, during the time-to-solution analysis for the *Ideal-gas* kernel (Figure 7.5).

The power consumption traces presented in Figure 7.7 show that the Altera

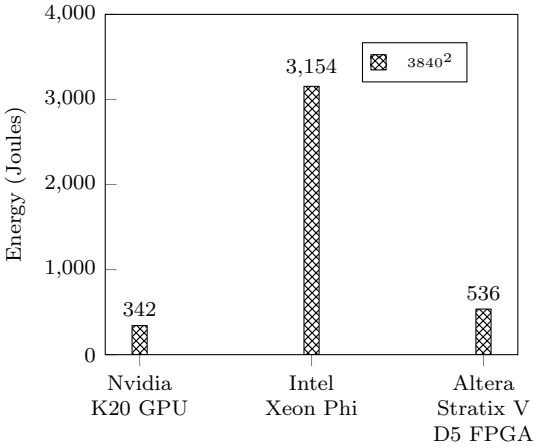


Figure 7.8: *Ideal-gas* kernel energy-to-solution analysis

Stratix V FPGA draws considerably less power during the execution of the experiment than both the Nvidia GPU and Intel Xeon Phi technologies. In the experiments with the *Ideal-gas* kernel the average power consumed by the Altera Stratix V was ∼22.9 W, approximately 4.5× less than the Nvidia GPU and approximately 11.1× less than the Intel Xeon Phi, which consumed on average ∼104.1 W and ∼254.9 W, respectively.

Figure 7.8 indicates, however, that overall the Nvidia K20 GPU delivers the most energy efficient performance, followed by the Altera Stratix V FPGA, and that the Intel Xeon Phi is the least efficient. In these experiments the Nvidia GPU required 342 J to complete the overall computation. This represents a ∼9.2× improvement over the Intel Xeon Phi, which required 3,154 J to perform the same computation. Compared to the Altera Stratix V, however, which required 536 J to fully execute the application, this only represents a ∼1.6× improvement. Additionally, this analysis indicates that in these experiments, the computations performed with the *Ideal-gas* kernel by the Altera Stratix V FPGA were ∼5.9× more energy efficient than those delivered by the Intel Xeon Phi, despite the Xeon Phi being able to execute the overall computation ∼1.9× quicker than the FPGA.

Overall, these results indicate that although the Altera Stratix V draws significantly less power than the Nvidia GPU, the GPU is able to deliver the greatest energy efficiency through its ability to execute the experiment significantly quicker than the FPGA, thus consuming less energy overall during the course of the computation.

7.4 Summary

The research documented within this chapter examined the utility of an Altera FPGA as a potential alternative processing solution for executing hydrodynamics applications on future generations of supercomputer. Related work within the field was presented which both motivates and positions the research undertaken here.

The results from this research demonstrate that OpenCL is a viable high-level language for enabling hydrodynamics applications to be successfully executed on Altera FPGA devices. The accepted approaches for targeting computational kernels expressed in OpenCL at GPU-based architectures do not, however, deliver optimal performance on Altera FPGAs. It is therefore necessary for kernels to be reimplemented, using alternative approaches available within the OpenCL standard, in order to maximise their performance on these devices. In particular, employing techniques to implement explicit vectorisation and expressing kernels as "*single work-item*" tasks which contain all of the loop constructs required for their execution, were necessary in order to optimise performance on the Altera FPGA device examined in this research.

Minimising global memory and floating point operations were also demonstrated to be key to improving overall kernel performance on Altera FPGA devices. This research identified, however, that the *Accelerate* kernel is mainly limited by the performance of the memory subsystem currently available on the FPGA architecture examined as part of this work. The performance of this subsystem was shown to be particularly problematic, relative to the performance achievable on the equivalent subsystems available within several alternative state-of-the-art processing solutions. This severely limits the effectiveness of the Altera FPGA as a candidate processing solution for executing this class of application, compared to alternative approaches such as Nvidia GPU and Intel Xeon Phi devices.

Additionally, although the FPGA-based processing solution was not able to match the performance of the equivalent Nvidia GPU or Intel Xeon Phi based solutions for executing the computational kernels examined in this research. These experiments did demonstrate that whilst executing the *Ideal-gas* kernel the Altera FPGA examined here draws significantly less power, than both the Nvidia GPU and Intel Xeon Phi technologies. It was also able to deliver superior energy efficiency compared to the Intel Xeon Phi architecture. The Nvidia GPU device, however, proved to be the most energy efficient processing solution, due to the fact that in these experiments it was able execute the simulations considerably faster than the Altera FPGA, despite drawing more power.

Hypothetically, if in the future FPGA devices could be combined with ex-

isting technologies which increase the memory bandwidth resources available to them, whilst also reducing memory access latency, then the technology may represent a more viable, more energy efficient processing solution for executing this class of hydrodynamics applications.

CHAPTER 8
Conclusion

The overall aim of this research was to enable greater levels of performance to be achieved for hydrodynamics scientific applications on potential future supercomputer architectures. This helps to facilitate advances in science within this domain through the simulation of larger, more detailed problems, as well as also improving the overall time-to-solution of key simulations.

As we approach the era of exascale computing, improving the scalability of applications will become increasingly important in enabling codebases to effectively harness the substantially increased levels of parallelism available in future architectures and thus achieve the required levels of performance. Applications based on the MPI-only paradigm are already starting to reach their scalability limits due to memory constraints and the shear number of ranks involved in the overall computation. Additionally, the level of on-node parallelism is likely to increase substantially in the approach to the construction of exascale capable systems. Accelerator devices have also been forecast to play an increasing role in scientific computing, which would significantly increase the levels of heterogeneity present within the nodes of supercomputer platforms. Furthermore, the amount of energy which will likely be required to power future supercomputing platforms also potentially threatens to limit their construction, which therefore necessitates the investigation of more energy efficient computing technologies.

To achieve these aims this research has examined the utility of several approaches and techniques for improving the scalability of existing hydrodynamics codebases and transitioning applications to future generations of supercomputing platforms. In particular PGAS technologies such as OpenSHMEM and CAF, based on lighter-weight one-sided communication operations, together with hybrid approaches based on OpenMP and OpenCL have been investigated. Additionally, the suitability of several prospective technologies have also been assessed as potential candidate solutions for improving computational performance and energy efficiency levels on future architectures.

This chapter concludes the research which was undertaken to achieve these aims and is documented in this thesis. Section 8.1 presents the key contributions which this work has made to the HPC and scientific computing fields. The main beneficiaries of the research are identified in Section 8.2 and several limitations of the work are outlined in Section 8.3. Finally, potential directions for future work to extend this research are outlined in Section 8.4.

8.1 Contributions

This research has delivered the following overall contributions within the domains of scientific and high performance computing:

8.1.1 Mini-app Development and Utilisation

The development of the CloverLeaf mini-app (Section 1.6) was significantly extended as part of this work to include PGAS-based implementations of the codebase as well as several alternative versions which utilise a hybrid programming paradigm. This enabled CloverLeaf to be used as a research vehicle in order to conduct the necessary work to examine potential application optimisations, alternative programming models and prospective supercomputer architecture choices. This research also demonstrated how the required planning and decision making relating to the future development of scientific applications can be improved through the use of mini-apps. Additionally, it also contributed significantly towards CloverLeaf being accepted as part of the Mantevo mini-applications suite from Sandia National Labs [84], which was recognised as one of the top 100 most technologically significant innovations in 2013 by R&D Magazine [171, 184]. It was also the UK's only contribution to the initiative and is currently being actively utilised by a large number of HPC centres, vendors and researchers across the world.

8.1.2 Evaluation of PGAS Programming Models

This work evaluated the utility of several PGAS programming models, in particular OpenSHMEM and CAF, as candidate technologies for improving the performance and scalability of hydrodynamics applications on current and future supercomputer architectures. The performance of these models was examined relative to an equivalent MPI-based implementation at considerable scale (up to 2,048 nodes/49,152 cores) on two significantly different, whilst still state-of-the-art, system architectures. The results demonstrate that for this class of scientific application, the OpenSHMEM PGAS programming model can deliver portable performance across both system architectures (SGI ICE-X and Cray XC30) and that it is able to match the performance of the MPI model, although this can require the utilisation of proprietary *non-blocking* operations. Overall, however, no significant performance improvements were observed from employing any of the PGAS constructs in preference to those utilised in the reference MPI implementation. The experimental results also show that the library-based PGAS model of OpenSHMEM can be significantly more performant than the equivalent language/compiler-based PGAS approaches employed in CAF.

Employing the PGAS communication constructs also did not deliver any significant improvements in the ability of the underlying system to overlap communication and computation operations. Additionally, although somewhat contrary to the PGAS *philosophy*/approach, this research demonstrated that applications based on either the OpenSHMEM or CAF paradigms can benefit from the aggregation of data into larger communication buffers, rather than moving data items directly using significantly smaller, potentially strided, memory operations. Furthermore, the performance of CAF-based applications can be extremely 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.

8.1.3 Examination of Hybrid Programming Models

The utilisation of hybrid programming model constructs, based on either OpenMP or OpenCL, were also examined and a quantitative assessment provided regarding whether the use of these models can deliver any performance and scalability benefits for this class of hydrodynamics application.

The development of the OpenMP version of CloverLeaf was progressed towards a fully optimal implementation of the codebase, thus enabling this model to be evaluated as a candidate technology for implementing a hybrid-programming approach within future scientific applications. In particular, this research developed and implemented several optimisations to the codebase, which improved overall performance by 28.0% and 4.6% on the Intel Xeon and Xeon Phi architectures, respectively. The experimental results also show that the performance of the individual optimisation techniques, developed as part of this research, can vary significantly across the two architectures.

To further reduce and avoid the cost of synchronisation operations within the codebase several *point-to-point* thread synchronisation and data *re-calculation* techniques were developed and implemented. Experimental results indicate that the use of these techniques may become increasingly necessary in order to achieve optimal application performance on future processor architectures, which are likely to include significantly more hardware threads than present day designs. Additionally, converting the application to use an OpenMP SPMD approach in order to reduce OpenMP thread synchronisation and *fork/join* overheads resulted in significant performance improvements in experiments with small problem sizes per node.

Recognising that converting MPI-only codebases to incorporate OpenMP threading constructs can be challenging and time consuming this research also evaluated the utility of the Reveal tool from Cray as a mechanism for improving

this process. It was demonstrated that for CloverLeaf, Reveal is able to automatically generate parallel code based on OpenMP directives, with minimal manual intervention. The experimental results show that whilst this code is functionally correct, its performance is also able to closely match that of manually developed and optimised code.

Furthermore, this research demonstrated that utilising OpenCL enables hydrodynamics applications to be executed across a wide range of current state-of-the-art processor architectures using a single codebase, which is currently not possible with other programming models. Additionally, it was also shown that OpenCL can be effectively combined with MPI to successfully implement a hybrid programming model for applications such as CloverLeaf. Overall this improved portability may be an extremely attractive proposition for some HPC sites as they attempt to cope with ever increasing workloads and a myriad of complex programming models and architectures.

The results show that it is also possible for the performance of OpenCL applications to match and sometimes exceed that of their equivalent native implementations, and to deliver performance improvements of up to 1.4× relative to the higher-level directive based approaches such as OpenACC. Achieving these performance levels, however, generally required the implementation of device specific optimisations and therefore this performance cannot necessarily be regarded as being portable across multiple architectures. Additionally, the performance of the OpenCL implementation was also shown to be considerably worse than that of equivalent native implementations in several experimental scenarios.

This work also identified particular optimisation techniques which result in performance improvements and degradations on each of the specific architectures examined in this research. In particular, the selection of appropriate OpenCL local *work-group* block-sizes was shown to be crucial in order for the performance of an OpenCL application to be maximised. An auto-tuning technique was demonstrated to be an extremely effective approach for determining this optimal configuration. The optimal block-sizes identified through this technique showed significant variations between the different processor architectures examined and also across the different OpenCL kernels within the CloverLeaf application. This complexity further supports the use of auto-tuning as an effective technique for the identification of optimal configuration parameters on future system architectures.

8.1.4 Development of Application Optimisations

This research also developed and implemented techniques to improve the performance of the CloverLeaf mini-application at significant scale on several state-of-the-art supercomputer platforms (IBM BG/Q, Cray XC30 and SGI ICE-X). This analysis showed that the selection of data structures which are able to scale to large process counts without consuming significantly more memory resources is crucial in enabling applications to execute efficiently at extreme-scale. Specifically for CloverLeaf, adopting a distributed meta-data based approach enabled the performance of the application to be improved significantly at scale and to achieve considerable memory savings compared to the original implementation. The reordering of MPI ranks, to improve the utilisation of shared memory communication resources and reduce the number of inter-node communication operations was also shown to significantly improve the performance of this class of application. Additionally, a number of candidate optimisation techniques for potentially improving the performance of the MPI-based communication phases of the application at significant scale were developed and examined. These included developing approaches which enabled communication operations to be overlapped with computation, examining the utility of several recently standardised MPI v3.0 constructs, as well as several message aggregation and early data transmission communication strategies.

Overall, this research identified that two-sided message passing via the MPI library is still the most likely technology for providing the inter-node communication constructs required by this class of hydrodynamics applications on future generations of supercomputers. In addition to MPI, OpenMP is the most likely candidate technology for delivering intra-node parallelism; however, whilst performance improvements through the use of this technology are possible, they are not universally observed across the architectures involved in this study. In particular this research showed that the MPI+OpenMP version of CloverLeaf initially delivered performance improvements in the smaller scale experiments examined on the Cray XC30. The release of a later version of the Cray MPI communication library, however, improved the performance of the MPI-only codebase to approximately match that of the hybrid versions. On the BG/Q architecture, however, and in the larger scale experiments on the Cray XC30 when the performance of the application is dominated by communication operations, the MPI-only approach was always the most performant. Additionally, the experimental results demonstrate that the developed optimisations deliver significant performance improvements for the hybrid versions of the codebase when the application is executed across multiple nodes, and performance is dominated by computation rather than communication operations.

8.1.5 Supercomputer Architecture Analysis

The derived experimental results also enabled the utility of several leading state-of-the-art supercomputing architectures and prospective intra- and inter-node processing solutions to be assessed for the execution of hydrodynamics applications. GPU-based accelerator devices were shown to be able to deliver considerable performance improvements of up to 2×, relative to state-of-the-art CPU-only based equivalent solutions, for the hydrodynamics applications on which this research focused. The results also demonstrate, however, that in terms of the energy required to achieve an equivalent solution, the BG/Q architecture can deliver significantly superior performance, relative to the Intel CPU-based alternatives. This would indicate that future supercomputing platforms would benefit from the incorporation of some of the design features and approaches implemented within this architecture. Additionally, the energy-to-solution profile recorded on the Cray XC30 demonstrates an optimal job size with which to execute CloverLeaf in order to minimise overall energy consumption.

This research also examined the utility of current FPGA-based accelerator devices as novel, lower power processing solutions for hydrodynamics applications. To improve the applicability of the technology for the execution of these applications several necessary hardware modifications were identified. Additionally, software-based optimisation techniques were also developed for improving the performance of key computational kernels on these devices. The results show that, it is possible to utilise OpenCL to enable hydrodynamics applications to be successfully targeted at Altera FPGA devices. Although it is necessary to structure kernels considerably differently compared to how they should be implemented for alternative accelerator solutions such as Nvidia GPUs. This further calls into question the *performance portability* which it is possible to achieve, through the utilisation of OpenCL, for scientific applications.

The performance of the Altera FPGAs examined in this research was also shown, for this class of application, to be significantly limited by the memory bandwidth resources currently available on existing devices. Relative to the Nvidia GPU and Intel Xeon Phi devices, the FPGA accelerator was up to ∼7× less performant. Additionally, an energy-to-solution analysis identified that the GPU architecture also delivered the most energy efficient performance. Despite the Altera FPGA device consuming considerably less power than both the Nvidia GPU and Intel Xeon Phi devices, overall it consumed more energy than the GPU due to its longer application execution times. The FPGA did, however, deliver considerable energy consumption reductions relative to the Xeon Phi co-processor solution.

8.2 Beneficiaries

The research findings derived as part of this work should directly benefit a wide range of scientific and high performance computing users, researchers and professionals. In particular, architects, code-custodians and team managers, who are responsible for establishing the higher-level development strategies of current hydrodynamics codebases will be able to use the results from this work to better inform their application development plans and priorities. Additionally, scientific application developers will also benefit significantly, through the utilisation of the optimisation techniques identified in this work, to better inform their implementation and maintenance choices. Overall this should contribute towards achieving considerable improvements in productivity by enabling application teams to focus on the most beneficial development directions for their applications of interest.

Researchers and HPC centres considering utilising an approach based on the use of *"mini-applications"* to improve their strategic development decisions, should also benefit. They would be able to cite this work as a successful case study which demonstrates the use of a *"mini-application"* as a research vehicle for the rapid exploration of prospective development and architectural options. It would otherwise not have been possible to evaluate as many different approaches by using a full production codebase.

HPC system procurement managers will also be able to utilise the results and conclusions to better inform their machine purchasing decisions, particularly during the procurements of platforms which may potentially incorporate some of the advanced accelerator architectures examined in this work. Additionally, technology manufacturers will benefit from these research findings as they seek to improve their product offerings, based on some of the deficiencies identified in this work, in order to secure more business from HPC centres.

8.3 Limitations

A significant amount of work has been undertaken as part of this project and whilst the initial research questions (Section 1.3) have been fully addressed, some limitations relating to the applicability of these research findings do exist and are documented in the following sections:

8.3.1 Application Characteristics

The CloverLeaf *"mini-application"* utilised throughout this work is representative of the production hydrodynamics applications which are the ultimate target

of this research. This therefore enables valid conclusions to be derived regarding the likely performance impact of the research findings on these codebases. Additionally, these applications also exhibit particular performance characteristics which are commonly found throughout a large number of applications employed across a number of different scientific domains, such as weather forecasting, reservoir simulation within the oil and gas industry, image processing and astronomy. Specifically, in these applications the majority of the computational functions are based on stencil operations and the inter-process communication patterns predominantly involve the exchange of boundary *halo*-cells, using relatively large message sizes, between logically neighbouring processes.

Due to the regularity of the operations involved, these structured communication patterns naturally lend themselves towards a two-sided model of communication. The research findings, particularly those relating to the use of the one-sided PGAS communication constructs, may therefore be less applicable to applications which exhibit more irregular communication patterns, and transmit smaller message sizes.

Furthermore, due to the nature of the hydrodynamic system which it simulates, CloverLeaf also utilises stencil-based computational operations on a staggered, but ultimately structured, mesh/grid. Consequently, this potentially limits the applicability of these research findings, particular those which relate to the performance of the different accelerator devices, to applications which employ similar computational operations and numerical methods. As applications which exhibit significantly different computational performance characteristics, such as those which utilise Arbitrary Lagrangian Eulerian methods, fully unstructured meshes or adaptive mesh refinement, may behave significantly differently on these processor architectures.

8.3.2 The Utility of FPGA Architectures

Despite extensive research being conducted into the potential utilisation of an FPGA device as a alternative, lower power processing solution, the derived findings from this section of the project do have certain limitations. In particular, as only one FPGA device from a single manufacturer was examined, the performance capabilities of this device may not be fully representative of all the FPGA-based processing solutions currently available within the marketplace. Consequently the conclusions relating to the performance of the key application kernels on this technology may, therefore, not be applicable to other alternative FPGA-based processing solutions. These additional architectures may, for example, possess greater memory bandwidth capabilities or a different balance of DSP and logic resources.

Utilising OpenCL as a high-level language from which to synthesise an FPGA targeted implementation of a kernel, is also still a very new technological approach. Consequently it is likely that as the technology matures the performance of the FPGA implementations which it is able to produce will also improve. Additionally, as only one high-level synthesis tool was examined, it may be possible for alternative models to deliver higher levels of application performance. It was also only possible, due to time constraints, for the performance of two CloverLeaf kernels to be examined on the FPGA technology, although their performance characteristics are representative of a large class of stencil-based scientific applications.

Despite these limitations, however, the conclusions derived as a result of this research do still make a very useful contribution towards establishing a more complete understanding of the applicability of FPGA technologies as potential lower power processing solutions for scientific applications.

8.4 Future Work

This thesis presents the research undertaken to address a number of key problems facing the hydrodynamics scientific simulation community. It also includes the evaluation of a well defined set of technologies, techniques and approaches which are of significant interest to the sponsor of this work. Despite this, however, there are a number of potential research directions which it was not possible to explore due to time and resource constraints. Examining these would help to address the limitations documented in Section 8.3 and may deliver significant further benefits to the hydrodynamics, and wider scientific, simulation communities.

8.4.1 Extending the PGAS Language Evaluation

The work undertaken to evaluate the utility of the PGAS programming languages examined thus far in this research has shown that it is possible for certain PGAS models (OpenSHMEM) to match the performance of MPI-based message passing approaches. The use of these models for the hydrodynamics simulation problems represented by the CloverLeaf *"mini-application"* does not, however, deliver significantly improved application performance beyond the levels currently achievable with the standard MPI-based approaches.

To determine whether this is universally the case for all PGAS programming models this work should be extended to examine the one-sided communications constructs recently standardised within version 3 of the MPI specification. Several new language additions to the CAF PGAS model have also recently been

approved and should be evaluated to determine whether the use of these features can deliver any performance benefits for this class of application. Similarly, for completeness, repeating the experiments with the PGAS implementations of CloverLeaf, using the Intel CAF and QLogic OpenSHMEM runtime systems, would be a potentially interesting research direction to determine whether an alternative implementation could deliver improved performance. Future architectures may also offer improved support for one-sided communication operations and therefore better support the PGAS programming models.

Evaluating the utility of the PGAS languages as potential future intra-node programming models for hybridising applications would help to determine whether the global address space and one-sided communication facilities of these models could deliver any performance benefits for applications when employed in this manner. The PGAS implementations of CloverLeaf should therefore be hybridised with threading constructs such as OpenMP, to facilitate the analysis of their potential as intra-node programming models. This would also determine if the performance improvements, which have been observed with the hybrid MPI-based versions, can be replicated with the PGAS implementations.

Utilising an auto-tuning framework to examine different symmetric heap and huge memory page settings, would enable further evidence to be collected on how to optimally execute the PGAS-based implementations of CloverLeaf. Additionally, analysing the overall memory consumption of the PGAS versions, when compared to the reference MPI implementation, may also be useful in determining whether these programming models deliver any advantages in terms of reductions in overall memory consumption.

Applying the PGAS programming models to scientific applications which exhibit different communications characteristics, to the one which has been studied in this research, would also be an extremely useful extension to this work. In particular applying these constructs to hydrodynamics applications which make use of Adaptive Mesh Refinement methods and exhibit irregular patterns of communication, would help to determine if these models can deliver any performance improvements for these additional classes of applications. Furthermore, utilising some of the more recently proposed PGAS concepts such as Active Messages [63] may also deliver performance benefits.

8.4.2 Intra-node Programming Models

OpenMP is currently the most likely candidate technology to be utilised for implementing a hybrid programming model for hydrodynamics applications. Alternative models, based on intra-node programming languages such as Kokkos Array [62], TBB [103], Cilk [102], Raja [86], the C++ threading model [28] and

OpenCL version 2.0, however, should also be examined. Concepts, such as Endpoints [55], have also been proposed to improve the intra-node programming functionality available within languages such as MPI and OpenSHMEM. Similarly, developing a version of CloverLeaf which utilises OpenMP in a similar manner to how MPI is employed within the codebase may deliver further performance improvements. Implementing these alternative versions would enable the relative merits of each approach to be objectively assessed, the achievable performance measured, and ultimately a more complete understanding to be reached regarding the optimal choice of an intra-node programming model for hydrodynamics applications. Additionally, extending the OpenMP threading model to incorporate some of the concepts found within PGAS languages, such as local barriers and *point-to-point* synchronisation operations, could also contribute to improving the suitability of this language as a future intra-node programming model for scientific applications.

To further assess the suitability of OpenCL as a technology for implementing the hybrid programming model, additional optimisations should be implemented within this version of CloverLeaf. These include investigating the effect of utilising explicit vector types and operations, particularly on CPU-based architectures. Additionally, further work should examine how best to execute OpenCL codes across multi-CPU nodes which contain numerous NUMA regions and also investigate whether device fissioning can deliver any performance advantages on these platforms. Implementing more advanced hybrid models in which the CPU does not merely act as a host, but shares some of the computational work with the attached accelerator devices, should also be evaluated using OpenCL and the recently proposed accelerator extensions to OpenMP. This may prove particularly effective on integrated CPU-GPU devices, on which it may also be beneficial to evaluate the utility of the *"zero-copy"* OpenCL constructs. Employing the ArrayFire software library from Accelereyes [1], together with approaches which utilise the improved atomic operations within the Kepler architecture from Nvidia, may also enhance the performance of the developed OpenCL reduction operations.

8.4.3 Energy Efficient Processing Technologies

To provide a more complete understanding of the suitability of FPGA-based technology for executing hydrodynamics applications, FPGA targeted implementations of the remaining CloverLeaf kernels should be developed. To further examine the floating-point computational capabilities of these devices single and fixed precision versions of the kernels should be developed and their performance compared to the existing double precision versions. The version of the

Accelerate kernel which incorporates the *"single work-item"* and *inter-iteration data caching* optimisations may also benefit from additional increases in the vectorisation width. To examine the optimality of the FPGA implementations which the Altera OpenCL compiler currently produces, their performance should be compared against additional versions produced using alternative high-level synthesis tools such as SystemC [92], ImpulseC [93] and the Maxeler compiler [137]. Similarly, VHDL [91] or Verilog [90] versions should also be developed to determine whether utilising a high-level synthesis approach based on OpenCL results in significant degradations in performance relative to these *native* FPGA programming models.

The performance of additional hardware platforms should also be evaluated in order to provide a more complete understanding of the suitability of all the FPGA-based processing solutions currently available. This includes examining devices which provide greater memory bandwidth resources such as those incorporating the Hybrid Memory Cube technology from Micron [141, 5]. Additionally, the forthcoming Arria and Stratix 10 FPGA products from Altera include native support for floating-point operations, and should also be evaluated to determine whether these technologies can deliver significant performance improvements [8]. Finally, alternative low-power processing solutions such as DSP-based processors from e.g. Texas Instruments [192], or ARM [15] based processor designs should also be examined.

Porting the Stream [139] and DGEMM [57] micro-benchmarks to the FPGA architecture would also enable the maximum sustainable performance of the memory and compute subsystems, which are available on these architectures, to be determined rather than relying on theoretical peak measurements. Furthermore, extending this research to incorporate applications from different scientific domains would provide useful information on the applicability of FPGA-based technologies to the wider scientific community.

Bibliography

[1] Accelereyes. ArrayFire. `http://www.accelereyes.com`, January 2013.

[2] L. Adhianto and B. Chapman. Performance modeling of communication and computation in hybrid MPI and OpenMP applications. In *Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on*, volume 2, 2006.

[3] Y. Ajima, T. Inoue, S. Hiramoto, Y. Takagi, and T. Shimizu. The Tofu Interconnect. *Micro, IEEE*, 32(1):21–31, Jan 2012.

[4] C. Albing. Evaluating Node Ordering for Improved Compactness. In *The Cray User Group 2013, May 6-9, 2013, Napa Valley, California, USA (2013)*, 2013.

[5] Altera. Addressing Next-Generation Memory Requirements Using Altera FPGAs and HMC Technology, August 2014. `http://www.altera.com/literature/wp/wp-01214-fpgas-hmc-technology.pdf`.

[6] Altera. Altera SDK for OpenCL Optimisation Guide. `http://www.altera.co.uk/literature/hb/opencl-sdk/aocl_optimization_guide.pdf`, September 2014.

[7] Altera. Altera SDK for OpenCL Programming Guide. `http://www.altera.co.uk/literature/hb/opencl-sdk/aocl_programming_guide.pdf`, September 2014.

[8] Altera. The Industry's First Floating-Point FPGA. `http://www.altera.co.uk/literature/po/bg-floating-point-fpga.pdf`, August 2014.

[9] R. Alverson, D. Roweth, and L. Kaplan. The Gemini System Interconnect. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 83–87, Aug 2010.

[10] K. Alvin, B. Barrett, et al. On the path to exascale. *International Journal of Distributed Systems and Technologies*, 1:1–22, April 2010.

[11] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, and K. Yelick. Exascale Programming Challenges. Technical report, Department of Energy, Office of Science, 2011. `http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/ProgrammingChallengesWorkshopReport.pdf`.

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

[13] AMD. AMD FirePro Workstation Graphics, August 2014. http://www.amd.com/en-us/products/graphics/workstation.

[14] AMD. AMD Fusion Family of APUs: Enabling a Superior, Immersive, PC Experience, August 2014. http://www.amd.com/Documents.

[15] ARM. Processors, March 2016. http://www.arm.com/products/processors/index.php.

[16] M. Bader, A. Breuer, W. Holzl, and S. Rettenberger. Vectorization of an augmented Riemann solver for the shallow water equations. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pages 193–201, July 2014.

[17] Baker, Matthew and Pophale, Swaroop and Vasnier, Jean-Charles and Jin, Haoqiang and Hernandez, Oscar. Hybrid Programming Using OpenSHMEM and OpenACC. In *Proceedings of the First Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools - Volume 8356*, OpenSHMEM 2014, pages 74–89, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[18] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. 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*, pages 20–30, Berlin, Heidelberg, 2009. Springer-Verlag.

[19] R. Barrett. Co-Array Fortran Experiences with Finite Differencing Methods. In *Cray User Group*, 2006. https://cug.org/5-publications/proceedings_attendee_lists/2006CD/S06_Proceedings/pages/Authors/Barrett-14B/Barrett-14B_Paper.pdf.

[20] R. Barrett, C. Vaughan, S. Hammond, and D. Roweth. Reducing the Bulk in the Bulk Synchronous Parallel Model. *Parallel Processing Letters*, 23(04):1340010, December 2013.

[21] J. Bashor. NERSC, Cray, Intel to collaborate on next-generation supercomputer, April 2014. https://www.nersc.gov/news-publications/nersc-news/nersc-center-news/2014/nersc-cray-intel-announce-next-generation-supercomputer/.

[22] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, April 2003.

[23] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[24] D. J. Benson. Computational Methods in Lagrangian and Eulerian Hydrocodes. *Comput. Methods Appl. Mech. Eng.*, 99(2-3):235–394, Sept. 1992.

[25] B. Bergen, M. Daniels, and P. Weber. A Hybrid Programming Model for Compressible Gas Dynamics Using OpenCL. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 397–404, Sept 2010.

[26] I. Bethune, J. M. Bull, N. J. Dingle, and N. J. Higham. Performance Analysis of Asynchronous Jacobi's Method Implemented in MPI, SHMEM and OpenMP. *Int. J. High Perform. Comput. Appl.*, 28(1):97–111, February 2014.

[27] A. Bland, J. Wells, O. Messer, O. Hernandez, and J. Rogers. Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory. In *Cray User Group*, 2012. `https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap138-file2.pdf`.

[28] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[29] M. Bose and V. Rajagopala. Physics Engine on Reconfigurable Processor–Low Power Optimized Solution empowering Next-Generation Graphics on Embedded Platforms. In *Computer Games (CGAMES), 2012 17th International Conference on*, pages 138–142, July 2012.

[30] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural Exploration of the ADRES Coarse-grained Reconfigurable Array. In *Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'07, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag.

[31] R. Brightwell, K. Pedretti, K. Underwood, and T. Hudson. SeaStar Interconnect: Balanced Bandwidth for Scalable Performance. *Micro, IEEE*, 26(3):41–57, May 2006.

[32] R. Brook, B. Hadri, V. Betro, R. Hulguin, and R. Braby. Early Application Experiences with the Intel MIC Architecture in a Cray CX1. In *Cray User Group*, 2012.

[33] E. Brossard, D. Richmond, J. Green, C. Ebeling, L. Ruzzo, C. Olson, and S. Hauck. A Model for Programming Data-Intensive Applications on FPGAs: A Genomics Case Study. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pages 84–93, July 2012.

[34] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing Molecular Dynamics on Hybrid High Performance Computers–Short Range Forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[35] R. Budiardja, L. Crosby, and H. You. Effect of Rank Placement on Cray XC30 Communication Cost. In *The Cray User Group 2013, May 6-9, 2013, Napa Valley, California, USA (2013)*, 2013.

[36] J. M. Bull and C. Ball. Point-to-point synchronisation on shared memory architectures. In *In 5th European Workshop on OpenMP (EWOMP03)*, Sept 2003. `http://www1.rz.rwth-aachen.de/computing/events/2003/ewomp03/omptalks/Monday/Session2/T10p.pdf`.

[37] C. Calvin, F. Ye, and S. Petiton. The Exploration of Pervasive and Fine-Grained Parallel Model Applied on Intel Xeon Phi Coprocessor. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pages 166–173, Oct 2013.

[38] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 12–12, Nov 2000.

[39] CAPS. HMPP: A hybrid multicore parallel programming platform. `http://www.dolbeau.name/dolbeau/publications/caps-hmpp-gpgpu-Boston-Workshop-Oct-2007.pdf`, July 2014.

[40] 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.

[41] D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker. The IBM Blue Gene/Q Interconnection Fabric. *Micro, IEEE*, 32(1):32–43, Jan 2012.

[42] D. Christodoulou. The Euler equations of compressible fluid flow. *Bulletin of the American Mathematical Society*, 44(4):581–602, 2007.

[43] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *The Many-core Applications Research Community Symposium,*, pages 38–44, November 2012. `http://www.lfbs.rwth-aachen.de/marc2012/07_Cramer.pdf`.

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

[45] Cray. Cray Fortran Reference Manual. Technical Report S-3901-83, Cray, August 2014. `http://docs.cray.com/books/S-3901-83/S-3901-83.pdf`.

[46] Cray. Using Cray Performance Measurement and Analysis Tools. Technical Report S237662, Cray, 2014. `http://docs.cray.com/books/S-2376-62/S-2376-62.pdf`.

[47] J. R. d. S. Junior, E. W. Clua, A. Montenegro, and P. A. Pagliosa. Fluid Simulation with Two-Way Interaction Rigid Body Using a Heterogeneous GPU and CPU Environment. In *Games and Digital Entertainment (SBGAMES), 2010 Brazilian Symposium on*, pages 156–164, Nov 2010.

[48] Dallas South News. The 5 Biggest Data Centers in the World, 2013. `http://www.dallassouthnews.org/2013/03/22/the-5-biggest-data-centers-in-the-world/`.

[49] J. del Cuvillo, W. Zhu, and G. R. Gao. Landing OpenMP on Cyclops-64: An Efficient Mapping of OpenMP to a Many-Core System-on-a-Chip. `http://www.capsl.udel.edu/pub/doc/papers/cf06.pdf`, 2006.

[50] C. Demerjian. What will Intel Xeon Phi do the GPGPU market? `http://semiaccurate.com/2012/11/13/what-will-intel-xeon-phi-do-to-the-gpgpu-market/`, November 2012.

[51] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Parallel and Distributed*

Processing, 2008. IPDPS 2008. IEEE International Symposium on, pages 1–12, April 2008.

[52] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc. Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions. *Proceedings of the IEEE*, 87(4):668–678, April 1999.

[53] A. Di Biagio, E. Speziale, and G. Agosta. Exploiting Thread-data Affinity in OpenMP with Data Access Patterns. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 230–241, Berlin, Heidelberg, 2011. Springer-Verlag.

[54] V. V. Dimakopoulos, P. E. Hadjidoukas, and G. C. Philos. A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 1–12, Berlin, Heidelberg, 2008. Springer-Verlag.

[55] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling communication concurrency through flexible MPI endpoints. *International Journal of High Performance Computing Applications*, 28(4):390–405, 2014.

[56] R. Dolbeau, F. Bodin, and G. de Verdiere. One OpenCL to rule them all? In *IEEE the 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*, pages 1–6, Sept 2013.

[57] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Mar. 1990.

[58] R. Dreslinski, D. Fick, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, D. Sylvester, D. Blaauw, and T. Mudge. Centip3De: A 64-Core, 3D Stacked Near-Threshold System. *Micro, IEEE*, 33(2):8–16, March 2013.

[59] N. Drosinos and N. Koziris. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004.

[60] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Parallel Comput.*, 38(8):391–407, Aug 2012.

[61] G. Economakos. ESL as a Gateway from OpenCL to FPGAs: Basic Ideas and Methodology Evaluation. In *Informatics (PCI), 2012 16th Panhellenic Conference on*, pages 80–85, Oct 2012.

[62] H. C. Edwards and D. Sunderland. Kokkos Array Performance-portable Manycore Programming Model. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '12, pages 1–10, New York, NY, USA, 2012. ACM.

[63] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation, 1992.

[64] EPSRC. International Review of Research Using HPC in the UK. Technical Report ISBN 1-904425-54-2, EPSRC, December 2005. `http://www.epsrc.ac.uk/newsevents/pubs/`.

[65] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray Cascade: A scalable HPC system based on a Dragonfly network. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9, Nov 2012.

[66] J. Fang, A. L. Varbanescu, and H. Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, Sept 2011.

[67] Fujitsu. Next-Generation PRIMEHPC, August 2014. `http://www.fujitsu.com/global/Images/next-generaton-primehpc_tcm100-1050349.pdf`.

[68] X. Gao, Z. Wang, H. Wan, and X. Long. Accelerate Smoothed Particle Hydrodynamics using GPU. In *Information Computing and Telecommunications (YC-ICT), 2010 IEEE Youth Conference on*, pages 399–402, Nov 2010.

[69] 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.

[70] P. Geoffray. Myrinet express (MX): Is your interconnect smart? In *High Performance Computing and Grid in Asia Pacific Region, 2004. Proceedings. Seventh International Conference on*, July 2004.

[71] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 53:1–53:12, New York, NY, USA, 2013. ACM.

[72] M. Hachman. Nvidia reveals PC-like performance for 'Denver' Tegra K1, August 2014. `http://www.pcworld.com/article/2463900/nvidia-reveals-pc-like-performance-for-denver-tegra-k1.html`.

[73] G. Hager, G. Jost, and R. Rabenseifner. Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes. In *The Cray User Group 2007, May 7-10, 2007, Seattle, Washington, USA (2007)*, 2007.

[74] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim. The IBM Blue Gene/Q Compute Chip. *Micro, IEEE*, 32(2):48–60, March 2012.

[75] M. Harris. Optimizing Parallel Reduction in CUDA. `http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf`, accessed Jan 2013.

[76] A. Hart, H. Richardson, J. Doleschal, T. Ilsche, M. Bielert, and M. Kappel. User-level Power Monitoring and Application Performance on Cray XC30 Supercomputers. In *The Cray User Group 2014, May 4-8, 2014, Lugano, Switzerland*, 2014.

[77] A. Hauer. The Continuing NNSA Commitment to State of the Art Modelling and Computation, March 2011. `https://asc.llnl.gov/content/assets/docs/exascale-meisner.pdf`.

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

[79] S. Hemmert et al. Exascale Hardware Architecture Working Group report. Technical Report LLNL-TR-474891, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, March 2011. `http://science.energy.gov/~/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pd://asc.llnl.gov/exascale/exascale-hwaWG.pdf`.

[80] D. Henty. Performance of Fortran Coarrays on the Cray XE6. In *Cray User Group*, 2012. `https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap181.pdf`.

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

[82] D. S. Henty. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 10–10, Nov 2000.

[83] O. Hernandez, C. Liao, and B. Chapman. A Tool to Display Array Access Patterns in OpenMP Programs. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 3732 of *Lecture Notes in Computer Science*, pages 490–498. Springer Berlin Heidelberg, 2006.

[84] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thornquist, and R. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, September 2009. `http://prod.sandia.gov/techlib/access-control.cgi/2009/095574.pdf`.

[85] T. Hoefler and T. Schneider. Optimization Principles for Collective Neighborhood Communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 98:1–98:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[86] R. D. Hornung and J. A. Keasler. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, Lawrence Livermore National Laboratory, August 2015.

[87] J. K. Hunter. An Introduction to the Incompressible Euler Equations. *Notes, Univ. of California, Davis*, 1, 2006.

[88] IBM. OpenCL Lounge. `https://www.ibm.com/developerworks/community/alphaworks/tech/opencl`, November 2012.

[89] IBM. OpenPOWER, August 2014. `http://openpowerfoundation.org/`.

[90] IEEE. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–560, 2006.

[91] IEEE. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009.

[92] IEEE. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012.

[93] Impulse Accelerated Technologies. Impulse Accelerated Technologies. `http://www.impulseaccelerated.com`, August 2015.

[94] Infiniband Trade Association. Infiniband Architecture Specification. `http://www.infinibandta.org`, 2014.

[95] Intel. Intel Augments Networking Portfolio with Best-in-Class High-Performance Computing Fabric Technology. `http://www.intel.com/content/www/us/en/high-performance-computing/infiniband-products.html`.

[96] Intel. Intel Acquires Industry-Leading, High-Performance Computing Interconnect Technology and Expertise. `http://newsroom.intel.com/community/intel_newsroom/blog/2012/04/24/intel-acquires-industry-leading-high-performance-computing-\\interconnect-technology-and-expertise`, April 2012.

[97] Intel. Intel European Exascale Labs Report, Sept 2012. `http://www.exascale-computing.eu/wp-content/uploads/2013/09/Intel-European-Exascale-Labs-Annual-Report-2012.pdf`.

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

[99] Intel. Disrupting the Data Center to Create the Digital Services Economy. `https://communities.intel.com/community/itpeernetwork/datastack/`, August 2014.

[100] Intel. Intel Architecture Instruction Set Extensions Programming Reference. Technical Report 319433-020, Intel, July 2014. `https://software.intel.com/sites/default/files/managed/c6/a9/319433-020.pdf`.

[101] Intel. Intel Xeon Phi Product Family, August 2014. `http://www.intel.co.uk/content/www/uk/en/processors/xeon/xeon-phi-detail.html`.

[102] Intel. Cilk Reference Manual. `https://software.intel.com/en-us/node/522579`, August 2015.

[103] Intel. Threading Building Blocks Reference Manual. `https://software.intel.com/en-us/node/506130`, August 2015.

[104] J. A. Herdman and W. P. Gaudin and S. McIntosh-Smith and M. Boulton and D. A. Beckingsale and A. C. Mallinson and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpenCL and CUDA. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 465–471, Nov 2012.

[105] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, and R. Thakur. High performance MPI-2 one-sided communication over InfiniBand. In *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 531–538, April 2004.

[106] M. D. Jones and R. Yao. Parallel programming for OSEM reconstruction with MPI, OpenMP, and hybrid MPI-OpenMP. In *Nuclear Science Symposium Conference Record, 2004 IEEE*, volume 5, pages 3036–3042, Oct 2004.

[107] P. Jones. Parallel Ocean Program (POP) User Guide. Technical Report LACC 99-18, Los Alamos National Laboratory, March 2003. `http://www.cesm.ucar.edu/models/ccsm4.0/pop/doc/users/`.

[108] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A Multi-objective Auto-tuning Framework for Parallel Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[109] J. Jose, K. Kandalla, M. Luo, and D. K. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 219–228, Sept 2012.

[110] G. Jost, J. Labarta, and J. Gimenez. *Shared Memory Parallel Programming with OpenMP: 5th International Workshop on Open MP Applications and Tools, WOMPAT 2004, Houston, TX, USA, May 17-18, 2004, Revised Selected Papers*, chapter What Multilevel Parallel Programs Do When You Are Not Watching: A Performance Analysis Case Study Comparing MPI/OpenMP, MLP, and Nested OpenMP, pages 29–40. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[111] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams. An Auto-Tuning Framework for Parallel Multicore Stencil Computations. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[112] R. Karrenberg and S. Hack. Improving Performance of OpenCL on CPUs. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pages 1–20, Berlin, Heidelberg, 2012. Springer-Verlag.

[113] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

[114] P. Kogge. Architectural Challenges at the Exasacle Frontier, Sept 2008. Invited talk, STF'08 - Simulating the Future; Using One Million Cores and Beyond.

[115] P. Kogge, K. Bergman, S. Borkar, D. Campbell, et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, DARPA IPTO, September 2008.

[116] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating performance and portability of OpenCL programs. In *The fifth international workshop on automatic performance tuning*, volume 66, 2010.

[117] Kurzweil Accelerating Intelligence. Designing the exascale copmuters of the future, July 2014. `http://www.kurzweilai.net/designing-the-exascale-computers-of-the-future`.

[118] Q. Lan, C. Xun, M. Wen, H. Su, L. Liu, and C. Zhang. Improving Performance of GPU Specific OpenCL Program on CPUs. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, pages 356–360, Dec 2012.

[119] J. Laros, P. Pokorny, and D. DeBonis. PowerInsight - A commodity power measurement capability. In *Green Computing Conference (IGCC), 2013 International*, pages 1–6, June 2013.

[120] 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.

[121] C. Lee. Board Design Guidelines for PCI Express Architecture, 2004. `http://kavi.pcisig.com/developers/main/training_materials/get_document?doc_id=c48e4d9b1409c7f697669d476995348cf1cd1830`.

[122] J. M. Levesque, R. Sankaran, and R. Grout. Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-petaflops and Beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 15:1–15:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[123] D. Li, B. R. de Supinski, M. Schulz, K. Cameron, and D. S. Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[124] M. Lin, I. Lebedev, and J. Wawrzynek. OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 458–463, Aug 2010.

[125] O. Lindtjorn, R. Clapp, O. Pell, O. Mencer, M. Flynn, and H. Fu. Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications. *IEEE Micro*, 31(2):41–49, March-April 2011.

[126] Z. Liu, B. Chapman, Y. Wen, L. Huang, T.-H. Weng, and O. Hernandez. Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03, pages 26–41, Berlin, Heidelberg, 2003. Springer-Verlag.

[127] LLNL. Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254, LLNL, July 2011. `https://codesign.llnl.gov/pdfs/spec-7.pdf`.

[128] R. Lucas, J. Ang, K. Bergman, S. Borkar, et al. Top Ten Exascale Research Challenges. Technical report, Department of Energy, Office of Science, ASCAC Subcommittee, 2014. `http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf`.

[129] T. Ludwig and M. Dolz. Total Cost of Ownership in High Performance Computing, May 2014. `http://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2014/tco-14-anna-lena_pdf.pdf`.

[130] G. Mahinthakumar and F. Saied. A Hybrid MPI-OpenMP Implementation of an Implicit Finite-Element Code on Parallel Architectures.

International Journal of High Performance Computing Applications, 16(4):371–393, 2002.

[131] 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.

[132] 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.

[133] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman. Experiences at Scale with PGAS Versions of a Hydrodynamics Application. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 9:1–9:11, New York, NY, USA, 2014. ACM.

[134] C. Martin. Post-Dennard Scaling and the final Years of Moore's Law. *Technical Report, Hochschule Augsburg University of Applied Sciences*, Sept 2014.

[135] S. Martin and M. Kappel. Cray XC30 Power Monitoring and Management. In *The Cray User Group 2014, May 4-8, 2014, Lugano, Switzerland*, 2014.

[136] T. Maruyama, T. Yoshida, R. Kan, I. Yamazaki, S. Yamamura, N. Takahashi, M. Hondou, and H. Okano. Sparc64 VIIIfx: A New-Generation Octocore Processor for Petascale Computing. *Micro, IEEE*, 30(2):30–40, March 2010.

[137] Maxeler. Maxeler Technologies. `https://www.maxeler.com/technology/`, September 2014.

[138] C. Maynard. Comparing One-sided Communication With MPI, UPC and SHMEM. In *The Cray User Group 2012, May 6-9, 2013, Napa Valley, California, USA (2012)*, 2012.

[139] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[140] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin. A New Vision for Coarray Fortran. In *Proceedings of the Third Conference on*

Partitioned Global Address Space Programing Models, PGAS '09, pages 5:1–5:9, New York, NY, USA, 2009. ACM.

[141] Micron. Hybrid Memory Cube, Aug 2014. `http://www.micron.com/products/hybrid-memory-cube`.

[142] Monte-Blanc Project. Mont-Blanc. `http://www.montblanc-project.eu/`, Aug 2014.

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

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

[145] K. Nakajima. Flat MPI vs. Hybrid: Evaluation of Parallel Programming Models for Preconditioned Iterative Solvers on "T2K Open Supercomputer". In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 73–80, Sept 2009.

[146] R. Nanjegowda, O. Hernandez, B. Chapman, and H. Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 42–52. Springer Berlin Heidelberg, 2009.

[147] R. Nishtala, P. Hargrove, D. Bonachea, and K. Yelick. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[148] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[149] NVIDIA. CUDA API Reference Manual version 4.2. `http://developer.download.nvidia.com`, April 2012.

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

[151] NVIDIA. Parallel Thread Execution ISA v4.0. `http://docs.nvidia.com/cuda/parallel-thread-execution`, July 2014.

[152] NVIDIA. Tesla GPU Accelerators for Servers, August 2014. http://www.nvidia.co.uk/object/tesla-server-gpus-uk.html.

[153] Office of Science, Department of Energy, USA. The Opportunities and Challenges of Exascale Computing. Technical report, Department of Energy: Office of Science, Sept 2010. http://science.energy.gov/~/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf.

[154] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins. OpenMP Task Scheduling Strategies for Multicore NUMA Systems. *Int. J. High Perform. Comput. Appl.*, 26(2):110–124, May 2012.

[155] OpenACC-standard.org. The OpenACC Application Programming Interface. http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf, June 2013.

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

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

[158] M. Owaida, N. Bellas, C. Antonopoulos, K. Daloukas, and C. Antoniadis. Massively parallel programming models used as hardware description languages: The OpenCL case. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pages 326–333, Nov 2011.

[159] S. J. Pennycook, S. D. Hammond, S. A. Wright, J. A. Herdman, I. Miller, and S. A. Jarvis. An Investigation of the Performance Portability of OpenCL. *J. Parallel Distrib. Comput.*, 73(11):1439–1450, Nov 2013.

[160] S. J. Pennycook and S. A. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 386–395, Washington, DC, USA, 2012. IEEE Computer Society.

[161] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network (QsNet): high-performance clustering technology. In *Hot Interconnects 9, 2001.*, pages 125–130, 2001.

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

[163] J. Phillips, J. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–9, Nov 2008.

[164] J. M. Pier, I. Figueroa, and J. Huegel. CUDA-enabled Particle-Based 3D Fluid Haptic Simulation. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA), 2011 IEEE*, pages 391–396, Nov 2011.

[165] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. W. Schulz, W. L. Barth, A. Majumdar, and D. K. Panda. Quantifying Performance Benefits of Overlap Using MPI-2 in a Seismic Modeling Application. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 17–25, New York, NY, USA, 2010. ACM.

[166] R. Preissl, N. Wichmann, B. Long, J. Shalf, S. Ethier, and A. Koniges. Multithreaded Global Address Space Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 78:1–78:11, New York, NY, USA, 2011. ACM.

[167] President's Information Technology Advisory Committee. Computational Science: Ensuring America's Competitiveness. Technical report, The Networking and Information Technology Research and Development Program, April 2005. `http://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf`.

[168] T. Pulliam. The Euler Equations. `http://people.nas.nasa.gov/~pulliam/Classes/New_notes/euler_notes.pdf`, November 1994.

[169] S. F. Rahman, J. Guo, and Q. Yi. Automated Empirical Tuning of Scientific Codes for Performance and Power Consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 107–116, New York, NY, USA, 2011. ACM.

[170] A. Ramachandran, J. Vienne, R. Van Der Wijngaart, L. Koesterke, and I. Sharapov. Performance Evaluation of NAS Parallel Benchmarks on Intel Xeon Phi. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 736–743, Oct 2013.

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

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

[173] B. Rider. A Very Brief History of Hydrodynamic Codes. Technical report, Sandia, June 2007. `https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/Rider_CSRI_June27_2007.pdf`.

[174] E. Rustico, G. Bilotta, G. Gallo, A. Herault, and C. D. Negro. Smoothed Particle Hydrodynamics Simulations on Multi-GPU Systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 384–391, Feb 2012.

[175] J. Sancho, K. Barker, D. Kerbyson, and K. Davis. Quantifying the Potential Benefit of Overlapping Communication and Computation in Large-Scale Scientific Applications. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 17–17, Nov 2006.

[176] G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp, and D. Panda. Natively Supporting True One-Sided Communication in MPI on Multi-core Systems with InfiniBand. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on*, pages 380–387, May 2009.

[177] S. Seo, G. Jo, and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 137–148, Nov 2011.

[178] H. Shan, B. Austin, N. J. Wright, E. Strohmaier, J. Shalf, and K. Yelick. Accelerating Applications at Scale Using One-Sided Communication. In *Conference on Partitioned Global Address Space Programming Models (PGAS'12)*, 2012.

[179] R. Sharma and P. Kanungo. Performance evaluation of MPI and hybrid MPI+OpenMP programming paradigms on multi-core processors cluster. In *Recent Trends in Information Systems (ReTIS), 2011 International Conference on*, pages 137–140, Dec 2011.

[180] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: A Unified Deadlock-free Construct for Collective and Point-to-point Synchronization. In *Proceedings of the 22Nd Annual International Conference*

on Supercomputing, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[181] J. Shirako, K. Sharma, and V. Sarkar. Unifying Barrier and Point-to-point Synchronization in OpenMP with Phasers. In *Proceedings of the 7th International Conference on OpenMP in the Petascale Era*, IWOMP'11, pages 122–137, Berlin, Heidelberg, 2011. Springer-Verlag.

[182] H. Shukla, H.-Y. Schive, T.-P. Woo, and T. Chiueh. Multi-science Applications with Single Codebase - GAMER - for Massively Parallel Architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 37:1–37:11, New York, NY, USA, 2011. ACM.

[183] H. Simon. Why we need Exascale and why we won't get there by 2020, August 2013. `http://cacs.usc.edu/education/cs653/Simon-Exascale-LBL13.pdf`.

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

[185] B. Spencer. A General Auto-Tuning Framework for Software Performance Optimisation. `http://mistymountain.co.uk/flamingo/report/autotuning-2011-05-30.pdf`, 2011.

[186] A. Stone, J. Dennis, and M. Strout. Evaluating Coarray Fortran with the CGPOP Miniapp. Technical report, Colorado State University, 2011. `http://dx.doi.org/10.1007/978-3-642-03770-2_9`.

[187] E. Stotzer et al. OpenMP Technical Report 1 on Directives for Attached Accelerators. Technical Report TR1_167, The OpenMP Architecture Review Board, November 2012. `http://www.openmp.org/mp-documents/TR1_167.pdf`.

[188] C. Su, D. Li, D. Nikolopoulos, M. Grove, K. W. Cameron, and B. R. de Supinski. Critical Path-based Thread Placement for NUMA Systems. In *Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, PMBS '11, pages 19–20, New York, NY, USA, 2011. ACM.

[189] T. P. Stefanski and N. Chavannes and N. Kuster. Hybrid OpenCL-MPI parallelization of the FDTD method. In *Electromagnetics in Advanced Applications (ICEAA), 2011 International Conference on*, pages 1201–1204, Sept 2011.

[190] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '00, pages 100–112, London, UK, UK, 2000. Springer-Verlag.

[191] M. Taylor. A Landscape of the New Dark Silicon Design Regime. *Micro, IEEE*, 33(5):8–19, Sept 2013.

[192] Texas Instruments. Texas Instruments. `http://www.ti.com/lsds/ti/processors/dsp/overview.page?DCMP=DSP&HQS=dsp`, August 2015.

[193] X. Tian, H. Saito, S. Preis, E. Garcia, S. Kozhukhov, M. Masten, A. Cherkasov, and N. Panchenko. Practical SIMD Vectorization Techniques for Intel Xeon Phi Coprocessors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1149–1158, May 2013.

[194] Top500.org. Top500 list. `http://www.top500.org`, June 2014.

[195] T. Trader. NVIDIA Boasts 'Compelling HPC Solution', August 2014. `http://www.hpcwire.com/2014/08/20/nvidia-calls-cuda-6-5-compelling-hpc-solution/`.

[196] D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. Dubey, et al. Programming Abstractions for Data Locality. Technical Report Padal14, Swiss National Supercomputing Center (CSCS), April 2014.

[197] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[198] J. van der Sanden. Evaluating the Performance and Portability of OpenCL. Master's thesis, Electronic Systems Group, Faculty of Electrical Engineering, Eindhoven University of Technology, 2011. `http://alexandria.tue.nl/extra1/afstversl/wsk-i/sanden2011.pdf`.

[199] B. van Leer. Towards the Ultimate Conservative Difference Scheme. *J. Comput. Phys.*, 135(2):229–248, August 1997.

[200] A. Vishnu, M. ten Bruggencate, and R. Olson. Evaluating the Potential of Cray Gemini Interconnect for PGAS Communication Runtime Systems. In *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, pages 70–77, Aug 2011.

[201] S. Wallace, V. Vishwanath, S. Coghlan, Z. Lan, and M. E. Papka. Measuring Power Consumption on IBM Blue Gene/Q. In *IPDPS Workshops'13*, pages 853–859, 2013.

[202] S. Wallace, V. Vishwanath, S. Coghlan, J. Tramm, Z. Lan, and M. Papka. Application power profiling on IBM Blue Gene/Q. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8, Sept 2013.

[203] X. Wang and V. Jandhyala. Enhanced hybrid MPI-OpenMP parallel electromagnetic simulations based on low-rank compressions. In *Electromagnetic Compatibility, 2008. EMC 2008. IEEE International Symposium on*, pages 1–5, Aug 2008.

[204] Y. Wang, Q. Qin, S. Wee, and J. Lin. Performance Portability Evaluation for OpenACC on Intel Knights Corner and Nvidia Kepler. Technical report, Shanghai Jiao Tong University, 2013. `http://ccoe.sjtu.edu.cn/blog/wp-content/uploads/2013/09/`.

[205] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing Hardware Accelerators in Scientific Applications: A Case Study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, Jan 2011.

[206] A. White. Exascale Challenges: Applications, Technologies, and Co-Design. Technical Report LA-UR 11-02200, Los Alamos National Laboratories, 2011. `https://asc.llnl.gov/content/assets/docs/exascale-white.pdf`.

[207] S. Wienke, P. Springer, C. Terboven, and D. an Mey. OpenACC: First Experiences with Real-world Applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

[208] Wikipedia. Ideal gas law. `https://en.wikipedia.org/wiki/Ideal_gas_law`, April 2016.

[209] P. Worley. Importance of Pre-Posting Receives. In *2nd Annual Cray Technical Workshop*, 2008. `http://www.csm.ornl.gov/~worley/talks/Worley_CrayTech08.pdf`.

[210] Y. Zhang and F. Mueller. Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters. *Parallel and Distributed Systems, IEEE Transactions on*, 24(3):417–427, March 2013.

Appendices

APPENDIX A

Experimental Platforms/Architectures

This chapter documents the specifications and configuration of the computational platforms utilised in this research. Section A.1 describes the large-scale, *"production"* supercomputer architectures employed, whilst section A.2 presents the configuration details of the smaller-scale *"test-bed"* platforms. It is likely that these machines may have been upgraded or otherwise changed since the submission of this thesis, however, the results presented within this research are based on the machines as specified in sections A.1 and A.2. Finally section A.3 concludes the chapter.

A.1 *Production* Supercomputer Platforms

Tables A.1 and A.2 document the specifications of the supercomputer platforms utilised in this research, whilst sections A.1.2 to A.1.5 present additional information on each of them. These production platforms are used for conducting scientific research at the particular hosting institutions as well as for supporting the research of other remotely-located collaborators.

A.1.1 HECToR

The *HECToR* platform, which was previously located at the Edinburgh Parallel Computing Centre (EPCC), functioned as the UK's national high-end computing resource between 2007 and 2014. Phases 1 and 2a of the platform were based on the XT4 technology from Cray, whilst in phases 2b and 3 the system was upgraded to the Cray XE6 technology.

	Archer	Spruce	HECToR
Manufacturer	Cray	SGI	Cray
Model	XC30	ICE-X	XE6
Location	EPCC	AWE	EPCC
Cabinets	16	16	30
Peak Perf	1.56PF	0.97PF	800+ TF
Processor	Intel Xeon E5-2697v2	Intel Xeon E5-2680v2	AMD Opteron 6276
Proc Clock Freq	2.7GHz	2.8GHz	2.3 GHz
Cores / CPU	12	10	16
Compute Nodes	3008	2226	2,816
CPUs/Node	2	2	2
Total CPUs	6016	4452	5,632
Memory/Node	64GB	64GB	32 GB
Memory Freq	1833MHz	1866 MT/s	-
Interconnect	Cray Aries	Mellanox IB-FDR	Cray Gemini
Topology	Dragonfly	7D-hypercube	3D-torus
Compilers	Cray CCE v8.2.6	Intel v14.0	Cray CCE v8.1.2
MPI	Cray Mpich v6.3.1	SGI MPI v2.9	Cray MPT v5.6.1
OpenSHMEM	Cray Shmem v6.3.1	SGI Shmem v2.9	N/A

Table A.1: UK-based experimental platform system specifications

A.1.2 Archer

The *Archer* platform, which is currently located at EPCC, is the latest deployment of the UK's National High Performance Computing Facility and is available primarily to support the academic research community, although it is also available for some industrial use. The system is based on the Cray XC30 architecture and is estimated to provide nearly 4× the scientific throughput of its predecessor, *HECToR*.

A.1.3 Spruce

The *Spruce* supercomputer complex was commissioned in 2014 to conduct scientific research in support of the UK's national nuclear deterrent. It is located at AWE plc and is comprised of two separate, albeit connected, systems known as *Spruce-A* and *Spruce-B*. Table A.1 shows the relevant system specifications for the larger *Spruce-A* portion of the machine. Both sub-systems are constructed from SGI ICE-X infrastructure and each contain a 7D-hypercube interconnect based on FDR Infiniband technology from Mellanox. The compute nodes within the system each contain two processors which are individually water cooled.

	Titan	Mira	Vulcan
Manufacturer	Cray	IBM	IBM
Model	XK7	BG/Q	BG/Q
Location	ORNL	ANL	LLNL
Cabinets	200	48	24
Peak Perf	20+ PF	10 PF	5 PF
Processor	AMD Opteron 6274	IBM PowerPC	IBM PowerPC
Proc Clock Freq	2.2 GHz	1.6 GHz (A2 core)	1.6 GHz (A2 core)
Cores / CPU	16	16	16
Compute Nodes	18,688	49,152	24,576
CPUs/Node	1	1	1
Accelerator/Node	1	0	0
Accelerator Type	Nvidia GPU	N/A	N/A
Accelerator Model	K20x	N/A	N/A
Accelerator Freq		N/A	N/A
Total CPUs	18,688	49,152	24,576
Total Accelerators	18,688	0	0
CPU Memory/Node	32 GB	16GB	16GB
CPU Memory Freq		1.333 GHz	1.333GHz
Acc. Mem/Node	6GB	N/A	N/A
Interconnect	Gemini	BG/Q	BG/Q
Topology	3D-torus	5D-torus	5D-torus
Compilers	Cray CCE	IBM XL	IBM XL
MPI	Cray Mpich2	IBM MPI	IBM MPI
OpenSHMEM	Cray Shmem	N/A	N/A

Table A.2: Specifications of platforms located at ORNL, ANL & LLNL

A.1.4 Mira

Mira is an IBM Blue Gene/Q machine located at the Leadership Computing Facility at Argonne National Laboratory in the USA. Its peak-performance is over 10-petaflops making it one of the most computationally powerful machines in the world. Additionally, it is also constructed from 48 racks, which makes it physically one of the largest machines in the world. It is intended for conducting open science research which is only possible through access to large-scale computational resources.

A.1.5 Titan

Titan is located at ORNL in the USA and was one of the first major, large-scale supercomputer deployments to utilise a hybrid (CPU+GPU) architecture. The use of both processor solutions enables higher levels of computational performance to be attained as well as for space and power constraints to be overcome. The system is $\sim$10$\times$ more powerful than its predecessor *Jaguar*, whilst occupying the same space and drawing approximately the same power. Thus the machine exhibits architectural trends and performance characteristics,

which are likely to be present in future generations of supercomputers, as the HPC industry moves towards the construction of exascale capable platforms. It previously held the number 1 position on the Top500 list and has a theoretical peak of more than 20 petaflops. It is, again, primarily intended for open science research across a broad range of scientific disciplines.

A.1.6 Vulcan

Vulcan is a 24 rack, BG/Q system from IBM and is located at Lawrence Livermore National Laboratory in the USA. It is available to support industrial collaborations and was procured to provide an unclassified supercomputing resource to compliment its larger *"sister"* system Sequoia, which is employed for conducting classified work.

A.2 *Test-bed* Platforms

In addition to the supercomputer architectures described in section A.1, this research also utilised several smaller-scale, more experimental and novel architectures. These were often constructed primarily for Computer Science focused research and allow specific application and hardware experiments to be conducted. Sections A.2.1 to A.2.3 document the specifications of these machines.

A.2.1 Teller, Compton & Shannon

As part of the *co-design* development approach adopted by SNL to prepare application- and system-level software for the disruptive architecture changes which are likely to occur in the build-up to the creation of exascale systems, several advanced architecture test-bed platforms have been constructed. These are generally novel prototypes which facilitate experimentation with non-production applications on a range of future candidate architectures. The test-beds are primarily used for exploring alternative node-level architectures, although the

	Teller	Compton	Shanon
Manufacturer	Penguin	Penguin	Cray
Location	Sandia	Sandia	Sandia
Proc Manufacturer	AMD	Intel	Intel
Processor	Trinity A10-5800K	Xeon E5-2670	Xeon E5-2670
Proc Clock Freq	3.8 GHz	2.6 GHz	2.6 GHz
Proc TDP (W)	100	115	115
Cores / CPU	4	8	8
Compute Nodes	104	42	32
CPUs/Node	1	2	2
CPU Memory/Node	16 GB	64 GB	128 GB
CPU Memory Freq	1.8 GHz	1.6 GHz	1.6 GHz
Total CPUs	104	84	64
Accelerator/Node	1	1	2
Accelerator Type	AMD GPU	Intel Xeon Phi	Nvidia GPU
Accelerator Model	HD-7660D	3100	K20x
Accelerator Freq	800 MHz	1.1GHz	732 MHz
Total Accelerators	104	42	64
Acc. Mem/Node	16 GB	6 GB	12 GB
Interconnect	Qlogic QDR IB	Mellanox QDR IB	Mellanox QDR IB
Topology	Tree	Tree	Tree
Host Compilers	GNU v4.8.1	Intel v15.0	GNU v4.8.1, CCE v8.3.0
OpenMP Libraries	GNU	Intel	GNU, Cray
OpenCL SDK	AMD APP v2.8.1	Intel v14.1.0	Nvidia v5.0.0
MPI	OpenMPI v1.8.2	Intel v4.1.3	OpenMPI v1.8.4, Cray v2.0.0
OpenSHMEM	N/A	N/A	N/A

Table A.3: Specifications of the experimental platforms located at SNL

machines themselves are of sufficient size to also allow their inter-node communication characteristics to be examined.

Specifically the *Teller*, *Compton* and *Shannon* test-beds were utilised within this research, Table A.3 documents the specifications of these machines. *Teller* provides access to the novel APU processing devices, developed by AMD, which combine both CPU and GPU architectures within one silicon die. This enables hybrid programming model experiments to be conducted on an architecture without a PCIe bus located between the CPU and GPU components. *Compton* incorporates advanced pre-production Intel Xeon Phi co-processors which are likely to be an important future architectural processing solution. Additionally, the nodes within *Shannon* contain two advanced Nvidia Kepler GPU devices, enabling programming models to be developed and experimented with, which allow applications to target multiple GPU devices.

	Chilean Pine
Manufacturer	Cray
Model	XK6
Location	AWE
Cabinets	1
Peak Perf	-
Processor	AMD Opteron 6272
Proc Clock Freq	2.1
Processor Peak Perf	147 GFlop/s (Double Precision)
Cores / CPU	16
Processor TDP	115 W
Compute Nodes	40
CPUs/Node	1
Accelerator/Node	1
Accelerator Type	Nvidia GPU
Accelerator Model	X2090 ("Fermi")
Accelerator Freq	1.15 GHz
Total CPUs	40
Total Accelerators	40
CPU Memory/Node	32GB
CPU Memory Freq	1.6 GHz
Mem. Bandwidth	36.5 GB/s
Acc. Mem/Node	6GB
Interconnect	Cray "Gemini"
Topology	3D-torus
Host Compilers	Cray 4.1.40, GNU 4.7.2
GNU Host Flags	`-O3 -march=native -funroll-loops`
Cray Host Flags	`-em -ra -h`
OpenMP Libraries	Cray, Intel
OpenCL/Cuda SDK	Nvidia Cuda Toolkit 5.0, AMD OpenCL SDK 2.7
OpenCL Flags	`-cl-mad-enable -cl-fast-relaxed-math`
Cuda Flags	`-gencode arch=compute_30, code=sm_35`
MPI	Cray MPI (Mpich2) v5.6.2.2
OpenSHMEM	N/A

Table A.4: *Chilean Pine* platform system specifications

A.2.2 Chilean Pine

Chilean Pine is a small-scale test-bed platform intended for application experimentation with hybrid (CPU+GPU) architectures and high-bandwidth, low-latency interconnect technologies. The system is located at AWE plc in the UK and its specifications can be found in Table A.4. Within this research it is employed primarily for experiments which target the AMD Opteron processors. Additionally, the platform also has an OpenCL runtime system installed on both its CPU and GPU components, enabling experiments which target the entire processing resources of the nodes to be conducted using this programming model.

	Tuck		
Manufacturer	Penguin		
Location	Warwick		
Cabinets	1		
Peak Perf	-		
Processor	Intel Xeon E5-2620		
Proc Clock Freq	2.00 GHz		
Cores / CPU	6		
Compute Nodes	1		
CPUs/Node	2		
CPU Memory/Node	64 GB		
CPU Memory Freq	2.0 GHz		
Accelerator/Node	3		
Accelerator	#1	#2	#3
Type	Nvidia GPU	Intel Xeon Phi	Altera FPGA
Model	K20c	7120P	Stratix V GS D5
Clock Freq (GHz)	0.7	1.238	0.6
Memory (GB)	5.1	16	4
Cores	13	61	N/A
Host Compilers	Intel v15.0, GNU v4.4.6		
OpenMP Libraries	Intel v15.0, GNU v4.4.6		
MPI	Intel v5.0.0		
Intel Host Flags	`-O3 -ipo -no-prec-div -restrict -fno-alias -prec-div -fp-model strict -fp-model source -prec-sqrt`		
OpenCL SDK	Intel 2012 & 2013, Altera v13.1, v14.1		
Nvidia SDK	Cuda Toolkit 5.0, 6.0		
GPU OpenCL Flags	`-cl-mad-enable -cl-fast-relaxed-math`		
Cuda Flags	`-gencode arch=compute_30, code=sm_35`		

Table A.5: System specifications of the *Tuck* experimental platform

A.2.3 Tuck

The *Tuck* platform is a small, 1-node experimental test-bed located at the University of Warwick (see table A.5). It is intended for experimental Computer Science research and is therefore not utilised for production work. The platform enables novel hardware and software configurations, which are not available on the existing larger-scale platforms, to be rapidly trialled and experimented with including for example FPGA-based processing solutions. *Tuck* also contains the PowerInsight [119] power monitoring technology, which provides a mechanism for conducting high-frequency power consumption analyses at the level of individual system components.

A.3 Summary

This chapter has described in detail the hardware and software configuration of each of the experimental platforms utilised throughout this research. Additionally it also provides high-level information on the purpose, location and ownership of each of these resources.