THE UNIVERSITY OF
WARWICK

University of Warwick institutional repository: http://go.warwick.ac.uk/wrap

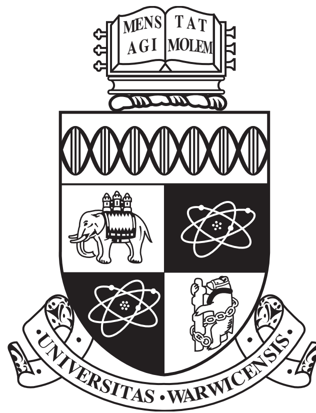
A Thesis Submitted for the Degree of PhD at the University of Warwick

http://go.warwick.ac.uk/wrap/72739

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it. Our policy information is available from the repository home page.



Towards Scalable Adaptive Mesh Refinement on Future Parallel Architectures

by

David Alexander Beckingsale

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

February 2015

Abstract

In the march towards exascale, supercomputer architectures are undergoing a significant change. Limited by power consumption and heat dissipation, future supercomputers are likely to be built around a lower-power many-core model. This shift in supercomputer design will require sweeping code changes in order to take advantage of the highly-parallel architectures. Evolving or rewriting legacy applications to perform well on these machines is a significant challenge.

Mini-applications, small computer programs that represent the performance characteristics of some larger application, can be used to investigate new programming models and improve the performance of the legacy application by proxy. These applications, being both easy to modify and representative, are essential for establishing a path to move legacy applications into the exascale era.

The focus of the work presented in this thesis is the design, development and employment of a new mini-application, CleverLeaf, for shock hydrodynamics with block-structured adaptive mesh refinement (AMR). We report on the development of CleverLeaf, and show how the fresh start provided by a mini-application can be used to develop an application that is flexible, accurate, and easy to employ in the investigation of exascale architectures.

We also detail the development of the first reported resident parallel block-structured AMR library for Graphics Processing Units (GPUs). Extending the SAMRAI library using the CUDA programming model, we develop datatypes that store data only in GPU memory, as well the necessary operators for moving and interpolating data on an adaptive mesh. We show that executing AMR simulations on a GPU is up to 4.8× faster than a CPU, and demonstrate scalability on over 4,000 nodes using a combination of CUDA and MPI.

Finally, we show how mini-applications can be employed to improve the performance of production applications on existing parallel architectures by selecting the optimal application configuration. Using CleverLeaf, we identify the most appropriate configurations on three contemporary supercomputer architectures. Selecting the best parameters for our application can reduce runtime by up to 82% and reduce memory usage by up to 32%.

Acknowledgements

I am indebted to many people for the advice, support and friendship they have provided over the past few years. It is my great pleasure to acknowledge them, and their contribution to shaping the contents of this thesis, here.

First and foremost, I would like to thank my supervisor, Prof. Stephen Jarvis. From supporting a foray into research as an undergraduate working on my final-year project, through to the very end of my time at Warwick, you have set an example for, and taken a keen interest in, my professional development. Thank-you for the hard work, enthusiasm, and willingness to engage with all my research.

In is my pleasure to acknowledge my colleagues, past and present, in the Performance Computing and Visualisation Group: Richard Bunt, Dr. Adam Chester, Peter Coetzee, James Davis, Dr. Henry Franks, Dr. Simon Hammond, Dr. Matthew Leeke, Andrew Mallinson, Dr. John Pennycook, Dr. Oliver Perks and Dr. Steven Wright. I reserve special thanks for Robert Bird, there from the start and still there at the end of all things.

In addition to the members of the Performance Computing and Visualisation group, I have also enjoyed the support of many individuals within the Department of Computer Science, including Dr. Roger Packwood, Paul

Williamson, Richard Cunningham, Dr. Christine Leigh, Catherine Pillet and the secretarial team. Thanks for your tireless, often unseen, work that holds the department together.

Thanks also to the staff at Lawrence Livermore National Laboratory, for their valuable support throughout my secondment and beyond. In particular, I'd like to thank Dr. Rich Hornung, Dr. Jeff Keasler, and Dr. Mike Wickett. Rich and Jeff for your continued involvement in my research, including countless discussions and lots of good advice. Mike, thanks for all the discussions (work-related and otherwise) in the beautiful surroundings we often found ourselves climbing. I'd also like to thank Dr. Todd Gamblin, whose supervision, guidance, and friendship has been instrumental in the writing of this thesis and the surrounding research.

Much of the work presented in this thesis has been both influenced, and guided by technical work at AWE. I would like to thank the members of AWE's Applied Computer Science team for involving us in discussions, external collaborations, and work streams: in particular Ash Vadgama, Dr. Iain Miller, Dr. Paddy Gillies, Dr. Satheesh Mahesharawan, and Andy Bennett. In particular, I'd like to thank Andy Herdman and Wayne Gaudin, whose guidance and encouragement (both professionally and beyond) has been an essential component of the work presented here.

Finally, I'd like to thank my family: Mum and Garry, Dad and Jo, Em and Harry. Thank you for understanding, letting me make the most of the past few years, and encouraging me both to work hard and to relax!

Declarations

This thesis is presented in accordance with the regulations 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 described in this thesis has been undertaken by the author except where otherwise stated.

The work presented (including data generation and analysis) was carried out by the author except in the following cases:

- Execution times for CloverLeaf on *HECToR* and *ARCHER* (Chapter 4) were collected by Andrew Mallinson (University of Warwick).
- Execution times for CloverLeaf on *Titan* (Chapter 4) were collected by John Levesque (Cray).
- CloverLeaf and TeaLeaf were developed as open-source projects in collaboration with: Wayne Gaudin (AWE), Andy Herdman (AWE), Andy Mallinson (University of Warwick), and Michael Boulton (University of Bristol). An up to date list of all contributors can be found on the Web at: `uk-mac.github.io`

Where possible, source code developed by the author is made available online:

- CloverLeaf: `http://uk-mac.github.io/CloverLeaf/`
- CleverLeaf: `http://uk-mac.github.io/CleverLeaf/`
- TeaLeaf: `http://uk-mac.github.io/TeaLeaf/`

Much of the work presented in this thesis has been published in peer reviewed conferences, workshops and journals. Publication highlights include the following:

- [18] D. A. Beckingsale, W. P. Gaudin, J. A. Herdman, and S. A. Jarvis, "Resident Parallel Block Structured Adaptive Mesh Refinement on Graphics Processing Units", in: Proceedings of the 44th International Conference on Parallel Processing, September 2015.
- [72] J. A. Herdman, W. P. Gaudin, O. F. J. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving portability and performance through OpenACC," in: Proceedings of the 1st Workshop on Accelerator Programming using Directives, November 2014.
- [105] 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, Napa, CA, May 2013.
- [104] 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 1st International Workshop on OpenCL, Atlanta, GA, May 2013.
- [71] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating Hydrocodes with OpenACC, OpeCL and CUDA," in: Proceedings of the 24th IEEE / ACM International Conference on Supercomputing, November 2012, pp. 465–471.

Other work which has been published in connection with the research presented in this thesis or conducted during the period of registration includes:

- [17] D. A. Beckingsale, O. Perks, W. P. Gaudin, J. A. Herdman, and S. A. Jarvis, "Optimisation of Patch Distribution Strategies for AMR Applications," Lecture Notes in Computer Science, vol. 7587, pp. 210–223, 2013.
- [129] O. Perks, D. A. Beckingsale, A. S. Dawes, J. A. Herdman, C. Mazauric, and S. A. Jarvis, "Analysing the influence of InfiniBand choice on OpenMPI memory consumption," in: Proceedings of 5th the International Conference on High Performance Computing and Simulation, 2013, pp. 186–193.
- [25] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis, "Performance Modelling of Magnetohydrodynamics Codes," Lecture Notes in Computer Science, vol. 7587, pp. 197–209, 2013.
- [133] O. F. J. Perks, D. A. Beckingsale, S. D. Hammond, I. Miller, J. A. Herdman, A. Vadgama, A. H. Bhalerao, L. He, and S. A. Jarvis, "Towards Automated Memory Model Generation Via Event Tracing," The Computer Journal, vol. 56(2), pp. 156–174, Feb. 2013.
- [132] O. F. J. Perks, R. F. Bird, D. A. Beckingsale, and S. A. Jarvis, "Exploiting spatiotemporal locality for fast call stack traversal," in: Proceedings of the 2nd Workshop on High-Performance Infrastructure for Scalable Tools, 2012, pp. 1–10.

Additional work currently submitted to journals for review or pending corrections includes the following:

- D. A. Beckingsale, A. Bhatele, J. Thiagarajan, S. A. Jarvis, T. Gamblin, "Optimising Parallel Loop Execution in Scientific Applications with Supervised Learning," submitted to *Supercomputing*, April 2015.

Further investigations and extensions of the studies presented in this thesis are described in internal reports and research articles for the United Kingdom Atomic Weapons Establishment (AWE). These are not contained or referenced within this thesis but demonstrate the applicability and extensibility of the techniques to other contexts.

Sponsorship and Grants

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

The University of Warwick Postgraduate Research Scholarship (2011–2014).

The United Kingdom Atomic Weapons Establishment under grants CDK0660 (*The Production of Predictive Models for Future Computing Requirements*), CDK0724 (*AWE Technical Outreach Programme*) and KTP006740 (*TSB Knowledge Transfer Partnership*).

The Royal Society Industry Fellowship Scheme under grant IF090020/AM.

The research presented in this work has made extensive use of the resources at Lawrence Livermore National Laboratory, which is supported U.S. Department of Energy under Contract DE-AC52-07NA27344.

Access to the Titan supercomputer was provided by the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

The ARCHER and HECtoR UK National Supercomputing Services (http://www.archer.ac.uk and http://www.hector.ac.uk) managed by EPSRC on behalf of Research Councils UK.

Finally, this material is based upon work supported by the National Science Foundation under Grant Number 1137097 and by the University of Tennessee through the Beacon Project. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the University of Tennessee.

Abbreviations

AMR Adaptive Mesh Refinement

APU Accelerated Processing Unit

AWE United Kingdom Atomic Weapons Establishment

CEA Commissariat á l'Énergie Atomique

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

EFLOPs Exa (10^{18}) Floating Point Operations per Second

FLOPs Floating Point Operations per Second

FPGA Field-Programmable Gate Array

GFLOPs Giga (10^{9}) Floating Point Operations per Second

GPGPU General-Purpose Graphics Processing Unit

GPU Graphics Processing Unit

HPC High-Performance Computing

HPCG High-Performance Conjugate Gradient

HPL High-Performance Linpack

LANL Los Alamos National Laboratory

LINPACK Linear Algebra Package

LLNL Lawrence Livermore National Laboratory

LOC Lines of Code

MFLOPs Mega (10^6) Floating Point Operations per Second

MHD Magnetohydrodynamics

MPI Message Passing Interface

ORNL Oak Ridge National Laboratory

PFLOPs Peta (10^{15}) Floating Point Operations per Second

PCI Peripheral Component Interconnect

SNL Sandia National Laboratories

TFLOPs Tera (10^{12}) Floating Point Operations per Second

Contents

Abstract ii

Acknowledgements iv

Declarations vi

Sponsorship and Grants x

Abbreviations xii

List of Figures xix

List of Tables xxiii

List of Algorithms xxv

1 Introduction 1

1.1 Motivations 4

1.2 Contributions 7

1.3 Thesis Overview 9

2 Parallel Programming and High-Performance Computing 11

2.1 Parallel Computers 12

2.1.1 Flynn's Taxonomy of Parallel Processors 14

2.1.2 Abstract Models of Parallel Computation 16

2.2 Laws of Parallel Programming 18

2.2.1 Amdahl's Law 18

2.2.2 Gustafson's Law 19

2.3 Parallel Computing Hardware 20

2.3.1 Mainframes 20

2.3.2 Vector Machines 21

2.3.3 Distributed-Memory Machines 21

2.3.4 Many-Core Machines and the Future of Supercomputers 22

2.4 Parallel Programming Models and Languages 24

2.4.1 Message-Passing 24

2.4.2 Multi-Threading 26

2.4.3 Vectorisation 27

2.5 Performance Analysis and Engineering 29

2.5.1 Benchmarks 30

2.5.2 Mini-Applications 33

2.6 Summary 35

3 Computational Physics and Adaptive Mesh Refinement 36

3.1 Computational Hydrodynamics 38

3.1.1 Euler's Equations 39

3.1.2 Lagrangian-Eulerian Scheme 41

3.1.3 Test Problems 44

3.2 Adaptive Mesh Refinement 47

3.2.1 Berger's Integration Algorithm 50

3.2.2 Alternative Adaptive Mesh Refinement Approaches . . 51

3.2.3 Adaptive Mesh Refinement Libraries 53

3.3 Summary 56

4 Performance Engineering with Mini-Applications **58**

4.1 Production Applications and Future Architectures 59

4.2 Mini-Applications 60

4.3 Mantevo: Mini-Applications in Practice 65

4.4 Mini-Application Development and Deployment 67

4.4.1 CloverLeaf 68

4.4.2 TeaLeaf 75

4.4.3 CleverLeaf 76

4.5 Summary 77

5 CleverLeaf: An Adaptive Mesh Refinement Mini-Application **79**

5.1 Related Work 80

5.2 Design and Development 82

5.2.1 SAMRAI 83

5.2.2 CleverLeaf 88

5.3 Validation and Verification 95

5.3.1 Validation 95

5.3.2 Verification 102

5.4 Summary 110

6 Scalable AMR on Graphics Processing Units **113**

6.1 Related Work 115

6.2 Design and Development 117

6.2.1 Programming Models 117

6.2.2 Extending SAMRAI with CudaPatchData 119

6.2.3 Adding CudaPatchData to CleverLeaf 124

6.3 Validation and Verification 128

6.3.1 Validation 129

6.3.2 Verification 129

6.4 Performance 133
6.5 Summary 136

7 Improving AMR Parameter Selection on Contemporary Compute Platforms 138

7.1 Application Parameters 139
7.2 Related Work 141
7.3 Experimental Setup 143
7.4 Performance Analysis 146
7.4.1 Scalability 147
7.4.2 Parameter Evaluation 151
7.5 Parameter Selection for Production Applications 159
7.5.1 Optimal Parameter Configuration 159
7.5.2 Increasing Job Throughput 160
7.5.3 Reducing Data Use 163
7.6 Summary 165

8 Conclusions and Future Work 167

8.1 Research Impact Highlights 168
8.2 Discussion and Implications 169
8.3 Limitations 170
8.4 Future Work 172
8.4.1 Investigating Simulation Accuracy 173
8.4.2 Application to Production Codes 174
8.4.3 Extension of GPU Implementation 175
8.4.4 Additional Parallel Architectures 176

Bibliography 179

A Experimental Results 201

A.1 Performance Engineering with Mini-Applications 201
A.2 Scalable AMR on Graphics Processing Units 203

A.3 Improving AMR Parameter Selection on Contemporary Compute Platforms 204

List of Figures

1.1 Traditional, current, and future supercomputer node architectures. 3

2.1 Flynn's Taxonomy of Computer Architectures. 15

2.2 The three parts of the Bulk-Synchronous Parallel model superstep. 17

2.3 Gustafson's Law for fixed- and scale-size problems. 20

2.4 Mapping between hardware, software, and the levels of parallelism in a supercomputer. 25

2.5 The fork-join model used for thread level parallelism in OpenMP. 26

2.6 Vector instructions generated for a simple loop. 28

2.7 The representativeness and simplicity of the three classes of benchmark. 30

3.1 An example mesh used to discretise the spatial domain in a scientific simulation. 37

3.2 The difference between the Lagrangian and Eulerian hydrodynamics schemes 40

3.3 Cell variable positions in the Lagrangian-Eulerian scheme. . . . 42

3.4 Initial conditions for Sod's shock tube problem. 45

3.5 Initial conditions for the Woodward-Colella interacting blastwaves problem. 46
3.6 Initial conditions for Sedov's blastwave problem. 47
3.7 Example adaptive mesh and the corresponding grid hierarchy. . 49
3.8 Mesh configurations for block-structured, cell-based, and tile-based AMR. 52

4.1 Kernel-driver design used in CloverLeaf. 69
4.2 CloverLeaf weak-scaling on HECToR and Titan. 73
4.3 MPI rank re-ordering strategy used to improve performance by reducing off-node communication. 74

5.1 The object-oriented design used in CleverLeaf. 84
5.2 Relationship between the `Patch`, `PatchData` and `CellData` classes. 85
5.3 AMR meshes and density at the initial and final times of the Sod problem. 96
5.4 Plot of density, velocity, and energy at $t = 0.2$ for the CPU-based solution of Sod's shock tube problem. 97
5.5 AMR meshes and density at the initial and final times of the Woodward-Colella problem. 98
5.6 Plot of density, velocity, and energy at $t = 0.038$ for the CPU-based solution of Woodward-Colella interacting blastwaves problem. 99
5.7 AMR meshes and density at the initial and final times of the Sedov problem. 100
5.8 Plot of density at $t = 1$ for the CPU-based solution of Sedov's blast wave problem. 100
5.9 Correlation between performance counter and efficiency loss calculated by Veritas. 105
5.10 Performance characteristics of Shamrock covered by CleverLeaf. 107
5.11 Performance characteristics not well covered by CleverLeaf. . . 109

5.12 Mean correlation with efficiency loss grouped by hardware resource. . . . 112

6.1 Accelerator-based heterogeneous architectures. . . . 118
6.2 The SAMRAI PatchData interface. . . . 120
6.3 The SAMRAI CudaPatchData datatypes. . . . 122
6.4 Data-parallel buffer packing for MPI operations. . . . 123
6.5 Host and device code for data-parallel node-centred linear refine. 125
6.6 Original Fortran linear refine code for node-centred data. . . . 126
6.7 Flexible CPU and GPU implementation in CleverLeaf. . . . 127
6.8 Data-parallel volume-weighted coarsen operator. . . . 128
6.9 Code listing for the data-parallel volume-weighted coarsen kernel. 129
6.10 Code listings for the original Fortran and data-parallel CUDA equation of state kernels. . . . 130
6.11 Code listings for the original and data-parallel linear refine kernels. 132
6.12 Wall-clock time for increasing problem size on a single GPU vs. dual-socket CPUs. . . . 134
6.13 Wall-clock time for strong-scaling runs on up to 8 GPUs. . . . 135
6.14 Initial and final density and patch configuration for the triple-point shock interaction problem. . . . 135
6.15 Weak-scaling performance analysis on Titan. . . . 136

7.1 AMR provides increased accuracy by increasing the resolution around areas of interest. . . . 144
7.2 Initial conditions and coarse patch configuration for the replication-scaled Sod problem. . . . 146
7.3 AMR (- - - ·) and uniform (——) replication scaling on Cab, Vulcan, and ARCHER. . . . 147
7.4 Message size histogram comparison for a 16-core run. . . . 150
7.5 Communication pattern comparison for a 16-core run. . . . 150
7.6 Regridding time and optimal frequency for each platform. . . . 153

7.7 Impact of maxmimum level number on runtime. 155
7.8 Impact of refinement ratio on runtime. 157
7.9 Predicted improvement in timesteps/month with optimal parameter selection, compared to the default configuration. 161
7.10 Relative data usage and runtime compared to the default configuration. 164

8.1 Performance of CleverLeaf on an Intel Xeon Phi and an Intel CPU.178

List of Tables

4.1 The solutions mini-applications provide to the problem of porting production codes. . . . 61
4.2 The 10 mini-apps included in the Mantevo Suite version 2. . . . 65

5.1 Description for the PAPI counters used during the analysis of Shamrock and CleverLeaf. . . . 104
5.2 Groups assigned to PAPI counters by VERITAS. . . . 111

6.1 IPA and Titan: hardware and software configurations. . . . 133

7.1 Parameters exposed by CleverLeaf. . . . 140
7.2 Cab, Vulcan, and ARCHER: hardware and software configurations. . . . 146
7.3 Optimal parameter values and achieved improvements over the default configuration, derived from our CleverLeaf mini-app study. 159

8.1 Error norms and runtimes for various simulation configurations. 174

A.1 Runtime for CloverLeaf kernels using three GPU-based programming models. . . . 202

A.2 Weak-scaling runtimes for CloverLeaf on Titan and HECToR. . 202
A.3 Performance improvements (% decrease in runtime) for rank-reordering on HECToR. 202
A.4 CPU vs. GPU runtime for increasing problem size. 203
A.5 Strong scaling results for the CPU- and GPU-based versions of CleverLeaf. 203
A.6 Grind times for replication scaled runs on Titan. 203
A.7 Runtime breakdown on ARCHER. 205
A.8 Runtime breakdown for varying regridding frequency. 206
A.9 Runtime breakdown for varying maximum level number. 207
A.10 Runtime breakdown for varying refinement ratio. 208
A.11 Relative impact of parameters on runtime and memory when compared to the default parameter configuraiton. 209

List of Algorithms

3.1 Lagrangian-Eulerian hydrodynamics scheme. 44

3.2 AMR integration algorithm. 51

CHAPTER 1

Introduction

Computational simulation is an essential tool in modern scientific research, allowing the validation of theories that may be too dangerous, impractical or expensive to validate via experiment. The results of these simulations are incredibly valuable, and scientific simulations push scientific discovery in fields as diverse as astrophysics, engineering and medicine. This impact drives the desire for results to be produced as fast as possible. Additionally, many scientific domains rely on time-sensitive results; if a simulation of tomorrow's weather takes two days to run, it won't be particularly useful! To deliver results quickly, researchers turn to supercomputers; these machines are typically at least one order of magnitude faster than desktop computers. Researchers in the field of High-Performance Computing (HPC) are interested in maximising the performance and efficiency of these machines through hardware development, algorithmic research, and the optimisation of existing codes.

Since their inception, the performance of supercomputers has grown almost exponentially. These performance improvements mean that results are

delivered faster, and gives supercomputers a new capability to solve problems that were previously intractable due to runtime or memory constraints. The key metric for assessing supercomputer performance is Floating Point Operations per Second (FLOPs), the number of a specific type of operation that a computer is able to perform per-second. Floating-point operations are mathematical operations involving decimal numbers, so FLOPs measures arithmetic throughput, and is relevant in determining the performance of a scientific application. In terms of FLOPs, the fastest supercomputer today is over 200,000 times faster than the fastest computer in 1994 [156]. Modern supercomputers are typically constructed from a huge number of commodity components connected using a high-speed interconnect. Scientific applications execute using a collection of parallel tasks that communicate in order to share data; each task processes a small part of the simulation data.

Although clusters of commodity processors and high-speed interconnects make successful supercomputers, in 2008 Los Alamos National Laboratory (LANL) bought the Roadrunner machine from IBM. Not only was this machine the first to achieve a sustained performance of over one Peta (10^{15}) Floating Point Operations per Second (PFLOPs), it was the first accelerated computing platform, with each node containing one conventional AMD processor and an IBM-designed Cell accelerator. The use of accelerators was motivated in part by the increasing cost of running conventional architectures; accelerators are designed to deliver more FLOPs per Watt than a standard processor. At the time, the Roadrunner architecture was revolutionary, but now accelerated supercomputers appear everywhere from University campuses to the largest supercomputer centres in the world. In June 2014, half of the ten fastest supercomputers in the world relied on a range of accelerators to achieve over 30 Tera (10^{12}) Floating Point Operations per Second (TFLOPs) of computational performance [156]. One feature common to all accelerators is the large number of low power cores that they provide. This era of many-core computing is set to continue as we approach the next milestone in supercomputing: a sustained perfor-

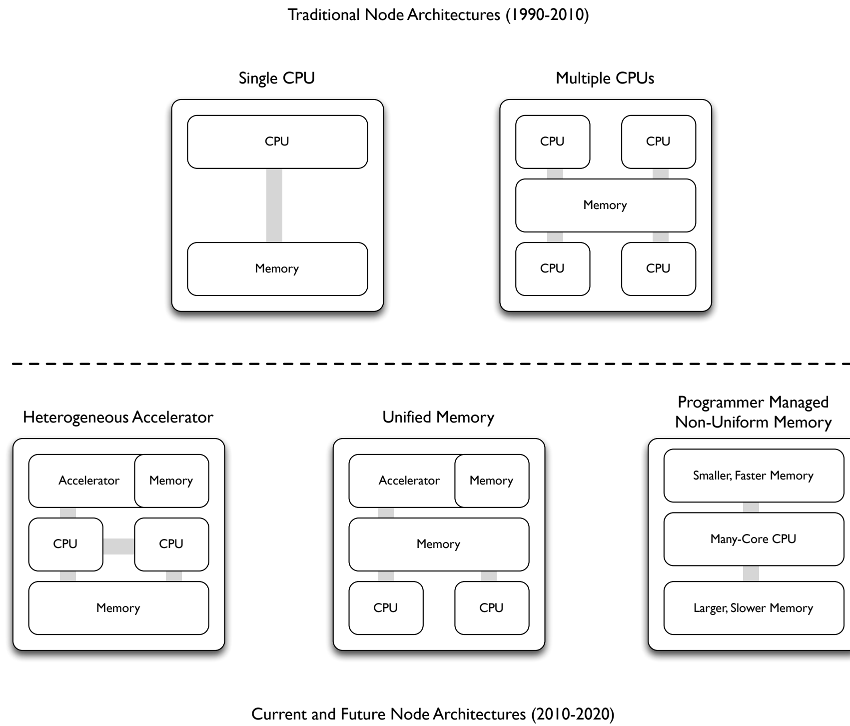


Figure 1.1: Traditional, current, and future supercomputer node architectures.

mance of over one Exa (10^{18}) Floating Point Operations per Second (EFLOPs). Figure 1.1 highlights the changes in supercomputer node architecture, from the single- and multi-core cluster architectures of the 90s, to the more complex current and future architectures. A key feature of many accelerated architectures is the separation of memory into distinct regions, and managing the transfer of data between the different memory spaces is an important consideration.

The applications used for scientific delivery are often at least 15 years old, with some applications still being used over 20 years after they were first developed [11, 53, 137]. To solve new, pivotal problems, computational scientists and code developers will expend most effort adding new scientific features to an application without necessarily considering code portability and maintenance. This focus on scientific features has created a huge suite of *legacy* codes; those that are written in older programming languages and without modern software engineering principles [138, 149]. It is essential that these codes are

updated and ported to the new many-core architectures predicted to be at the core of HPC for the next 10 years. However, due to their monolithic nature, ensuring these applications can run on next generation supercomputers is a considerable challenge. Mini-applications, small computer programs that represent the legacy application, can be employed to help ease this porting effort.

The research presented in this thesis represents over three years of investigations conducted with the Atomic Weapons Establishment (AWE), one of the largest supercomputing sites in the UK. These investigations focus on evaluating, understanding, and improving the performance of a range of applications on existing and future computing architectures. As we approach the exascale era, applications must adapt in order to continue to deliver timely and useful scientific results. This work details the development of a mini-application designed to investigate some of these evolutionary and revolutionary approaches to preparing applications for future supercomputers. The investigations are validated on some of the largest supercomputers available in the world today including those in the United Kingdom, and at Lawrence Livermore National Laboratory (LLNL) and Oak Ridge National Laboratory (ORNL) in the United States of America. The principle domain of the mini-application developed in this work is shock hydrodynamics, a type of simulation at the core of many applications used at AWE and other large supercomputing sites such as LANL and LLNL.

1.1 Motivations

In the 50 years since the introduction of Seymour Cray's CDC 1604 in 1964, supercomputers have followed a steady trend of increasing application performance. These improvements were provided by Moore's law, the prediction by Gordon Moore that the transistor density of a chip would double every 18 months [112]. With each new generation of hardware, these transistors were used to improve the performance of the chip, facilitating increased clock

speeds and additional low-level parallelism increasing the speed at which applications would run. This period of significant performance improvement every generation was known as the “free lunch”, and application developers could rely on each new supercomputer allowing them to solve larger problems faster [152, 153]. However, the reliability of these performance improvements meant that research into application performance and efficient algorithms was unnecessary to increase scientific delivery, and application developers focused on scientific features.

In 2001, issues concerning heat dissipation and power consumption meant that clock speeds began to stall and hardware manufacturers instead focused on increasing the number of processor cores on a single chip [113]. The free lunch was over, and in some cases clock speeds even decreased in order to accommodate the extra hardware required to manage the increased number of processor cores. These new multi-core chips widened the gap between the theoretical and achieved performance, with typical applications performing at only a tenth of any supercomputer’s peak FLOPs [88]. At the system, rather than the processor scale, concerns about power consumption have led hardware manufacturers to propose vastly different architectures to reach exascale levels of performance. With one Megawatt of power costing around one million US dollars a year, reducing the power consumption of supercomputers has a huge impact on the total cost of ownership.

The significant shift in supercomputer design will require sweeping code changes and means that an application that is both easy to modify yet representative is essential for discovering a path to move existing legacy applications into the exascale era. Performance engineering is a branch of computer science that encompasses the tools and techniques used to measure, analyse, and optimise the performance of applications running on anything from an embedded chip to the largest supercomputer. A recent branch of performance engineering research aims to use mini-applications, small computer programs that represent some larger application, to improve the performance of the larger application

by proxy. Tools and techniques are applied to the mini-application to identify optimisation opportunities. Once these optimisations are implemented and verified in the mini-application, they can be transferred to the larger application with the hope of seeing similar improvements in performance. In this role, mini-applications can be seen to supplant earlier benchmarking methodologies where either small, micro-benchmarks or large application benchmarks were used. Micro-benchmarks lack the representativeness of a larger application, but if the application is too large the work required to measure, analyse, and optimise its performance becomes significant. To date, all three classes of benchmarks have been used to successfully optimise code performance in both academic and industrial settings.

The subject of this thesis is the use of mini-applications to improve application performance and evaluate future hardware. We focus on optimising and porting a code with Adaptive Mesh Refinement (AMR), where the accuracy of the simulation is dynamically increased where it is most necessary. For example, when simulating the earth during a meteor impact, the location of the impact is the most important feature. Focusing computational resources on this region of the simulated domain can save both time and memory, but adds complexity to the application due to the management of the dynamic workload. It is the dynamic nature of AMR simulations that make them particularly challenging to port to accelerator-based architectures. Performing these investigations with a mini-application removes unnecessary code complexity stemming from the features required in a production code, and allows us to focus on challenges specific to adaptive execution on a highly-parallel architecture.

The context of this investigation into application performance and future hardware is CleverLeaf, a hydrodynamics mini-application with an AMR capability. The class of applications represented by CleverLeaf form a significant proportion of the workload at AWE, one of the largest supercomputer sites in the UK. Similar codes also occupy large portions of total execution time at LANL and LLNL. Our investigation is driven by the desire to reduce appli-

cation runtime through the tuning of software parameters, appropriate hardware choices, algorithm optimisation, and machine configurations. When the improvements we identify using a mini-application can be applied to real applications, the reductions in time to solution provide a measurable reduction in the cost associated with day-to-day operations at AWE. The research presented in this thesis is thus of interest to computer scientists, domain experts, and management.

1.2 Contributions

This thesis makes the following specific contributions to knowledge in the field of performance engineering:

- We present an investigation into the role of mini-applications in preparing existing production applications for future high-performance computing architectures. This study describes the development and use of two mini-applications at the University of Warwick and AWE. One of these applications, CloverLeaf, has been released as part of the Mantevo suite [73, 106]. We extended our study with a discussion of how a centralised repository of applications, such as Mantevo, provides valuable infrastructure and guidance for a large collection of mini-applications. We illustrate the use of mini-applications in exascale application preparation with a number of examples. **This study provides application developers with an overview of available mini-applications and a range of ways to use them to steer application development.**
- **We develop the first reported shock hydrodynamics mini-application with adaptive mesh refinement: CleverLeaf.** Applications using AMR can be characterised by their complex control logic, and developing a mini-application with this same control flow allows us to easily investigate possible strategies for running AMR applications on future archi-

tectures. We show that the simulation is physically accurate, as well as accurate in terms of how it captures the key performance characteristics of another AMR benchmark application. CleverLeaf has also been released under an open-source licence as part of the Mantevo suite.

- **We develop the first reported resident GPU-based block-structured adaptive mesh refinement library.** Specifically, we develop a set of extensions for the SAMRAI library, and use them to extend CleverLeaf so that it can execute with all data stored exclusively on Graphics Processing Unit (GPU)s. These changes involve the development of classes to manage data storage in GPU memory and novel algorithms that implement the necessary adaptive mesh refinement operations in a data-parallel fashion. The AMR-specific code we have developed to run on GPUs forms part of a larger library, and as such can be applied to any other block-structured AMR code that uses a Cartesian geometry.

- We present an investigation into applying mini-applications to influence and predict future application performance. **Using CleverLeaf, we expose and experiment with a range of application parameters and identify the most appropriate for three contemporary supercomputer architectures.** These values can be used in production codes, and where this opportunity is not available, we are able to inform code developers about the most effective code development path for their application. This study shows the way that mini-applications can be used to influence current production applications. The optimal parameter values we identify can be applied immediately for improvements in both application runtime and memory consumption.

1.3 Thesis Overview

This chapter addresses the fundamental motivation for the research presented in this thesis and describes the key contributions of this research. The remainder of this thesis is structured as follows:

Chapter 2 presents the terms, theory, hardware and software essential to the fields of high performance computing and performance engineering. The challenges addressed by this research largely arise from the complex combination of programming models and hardware described in this chapter. This presentation includes a thorough review of the future hardware and programming models at the core of this thesis.

Chapter 3 describes the use of computers to simulate physical systems. We focus on computational shock hydrodynamics, the simulation of fluids under intense pressure and temperature conditions. We provide an overview of Berger's adaptive mesh refinement technique, which is used to reduce the resources required to reach an accurate solution. This includes a thorough review of related literature, and a detailed description of the mathematical techniques used to solve Euler's equations on an adaptive mesh.

Chapter 4 provides a comprehensive introduction to mini-applications, and their use in exploring possible solutions for exascale code development. Mini-applications can be used to explore new hardware, programming languages, and programming models. They can also be used to explore application performance on current supercomputers. Our research applies mini-applications in both these contexts, and this chapter presents additional examples of the successful use of mini-applications.

Chapter 5 presents the development of the first hydrodynamics mini-application with an AMR capability. This case-study outlines the physics represented by

the application and the integration scheme used to advance the simulation using AMR. We describe the implementation of this scheme in CleverLeaf, and present results that verify both the accuracy and representativeness of our mini-application.

Chapter 6 extends CleverLeaf to execute on contemporary GPU architectures. This requires modification of the data-structures and algorithms at the core of both the AMR routines and the hydrodynamics scheme, and highlights the flexibility and utility of mini-applications in investigating new architectures and programming models. We evaluate the accuracy and performance of our GPU-based application on over 4,000 nodes of the largest GPU-accelerated supercomputer in the world.

Chapter 7 demonstrates how mini-applications can be used to improve the performance of applications on existing architectures through the selection of optimal runtime parameters for a given architecture. With a series of benchmarking experiments we identify hardware-appropriate values for core application parameters. Using these parameters, we identify the increase in job throughput and decrease in data usage that can be obtained through careful parameter selection.

Chapter 8 concludes the thesis with a discussion of the implications of the research presented to application developers, facilities staff, and organisation management. We also identify and address the limitations of our research and discuss possible directions for future work.

CHAPTER 2

Parallel Programming and High-Performance Computing

Since the 1960s, supercomputers have been used to solve some of the largest computational problems in the world. At least an order of magnitude faster than a desktop computer, these machines are used to tackle problems including cryptography, scientific simulations, and data analysis. Thanks to the doubling of transistor density every 18 months predicted by Intel's Gordon Moore in 1965, the performance of applications was able to grow without explicit programmer intervention, through the increase in parallelism of the chip [112]. The extra transistors were used to add multiple instructions units, increase instruction level parallelism, and add caches to reduce delays due to fetching data; all features which improved application performance. At the start of the millennium, hardware limitations meant that the ever increasing transistor counts couldn't be used to increase the performance of a single processor core much further. To put the transistors to good use, engineers began creating *multi-core* CPUs, where each Central Processing Unit (CPU) contained multiple processing cores that could run simultaneously. Parallel computers are those in which

multiple parts of a program are executed simultaneously, and in the case of multi-core CPUs, each core will execute one part of the program. Programming these multiple cores is difficult without the support of some underlying programming model.

Parallel computing is inherent in all modern supercomputer design, and future architectures are likely to require the application to expose more parallelism. Many different types of parallelism are supported at a hardware level, including: instruction-level parallelism, where multiple instructions are executed every clock cycle; data-level parallelism, where each instruction can operate on a different of data item; and task-level parallelism, where many threads, processes, or programs can execute independently. The levels of parallelism can be viewed as a hierarchy that must be exploited at all stages in order to achieve maximum performance.

Parallel programming is a complex combination of theory, hardware and software. In this chapter we present a detailed description of the hierarchy of parallelism exhibited by supercomputers, along with the laws that govern them. We also provide an overview of past supercomputer architectures and present trends exhibited by current platforms that provide a context for the research presented in this thesis. We describe the programming models that support each level of parallelism, with four of these models used in the applications discussed throughout this thesis. Finally, we consider performance analysis and engineering, detailing the use of benchmark applications in measuring and improving computer performance.

2.1 Parallel Computers

Parallel computers are those that can execute multiple parts of a program simultaneously. This parallel execution can come in three forms: (i) instruction-level, (ii) data-level, and (iii) task-level. Each form moves successively further from low-level detail towards a more global view of the entire parallel machine. Par-

allelism at every level is supported by a number of technologies: programming languages, programming models, and compilers; that help the application programmer use this parallelism.

Instruction-level parallelism is the most technical of the three forms, and involves multiple instruction units and multiple independent instructions being executed simultaneously. Parallelism at the instruction level is enabled in the processor, and modern processors can use a number of techniques to extract parallelism at the instruction level. Pipelining partially overlaps the execution of multiple instructions, separating the different phases so that one instruction is always in some part of the fetch-decode-execute cycle. Branch prediction and speculative execution are two more advanced techniques that involve beginning to compute instructions before the required data is available to determine whether the instruction is necessary. For example, one part of an if-then-else statement may be executed before the condition has been fully evaluated. If the correct branch is executed then this technique avoids the stall that would be introduced whilst waiting for the condition to be evaluated.

Whilst instruction level parallelism is responsible for much of the performance provided by modern processors, the parallel programming models considered most frequently in this thesis are task- and data-level parallelism. We define data-level parallelism at a node level, where multiple data-items are manipulated simultaneously by a single processor, provides a higher level of parallelism that is amenable to programmer control. First available in early vector-processing machines, vectorisation allows multiple data items to be manipulated in parallel, provided they are all manipulated with the same instructions.

We define task-level parallelism as communication between a number of co-ordinating tasks that are executing simultaneously. This communication can be explicit, where the programmer must manually tell the tasks to exchange data, or implicit, where the system uses knowledge of dependencies within the program to exchange data without prompting.

Task-level parallelism can also exist within a typical supercomputer node, where each executing thread forms a parallel task. Whilst concurrent tasks executing within a node can communicate using shared memory, tasks executing on different nodes must communicate by sending data across the network.

2.1.1 Flynn's Taxonomy of Parallel Processors

To classify the possible types of parallel computer, Flynn proposed a taxonomy based on the method of instruction and data dispatch of the processor [51]. He described four classes of processor, each having either single or multiple data and instruction units (see Figure 2.1). Single instruction, single data processors (SISD) are the most simple and were common in early computers.

Vector processors (like the Cray–1) or processors with vector instruction sets (for example Intel's Advanced Vector Extensions) apply the same instruction to multiple pieces of data simultaneously and are considered examples of the single instruction, multiple data class of processor. This gives rise to the initialism SIMD, which is commonly used to describe vector-parallel computing. The third category: multiple-instruction, single-data processors (MISD) are considered redundant due to the complexity of manipulating single data values in parallel. Examples of MISD processors exist in fault-tolerant computing where the same calculation is performed in parallel and the result combined to reduce the chances of an incorrect outcome.

The final category: multiple-instruction, multiple-data (MIMD); is an abstract representation of a distributed computing architecture such as a modern cluster. Each processor has its own local memory and the capability to execute instructions independently of any other processor. Another term for describing this kind of execution class is single-program multiple-data (SPMD) [40], and it is a ubiquitous description for contemporary distributed scientific applications, where multiple instances of the same program are launched with each processing some portion of the application domain. Extending this notion are those scientific applications that use a multiple-program multiple-data

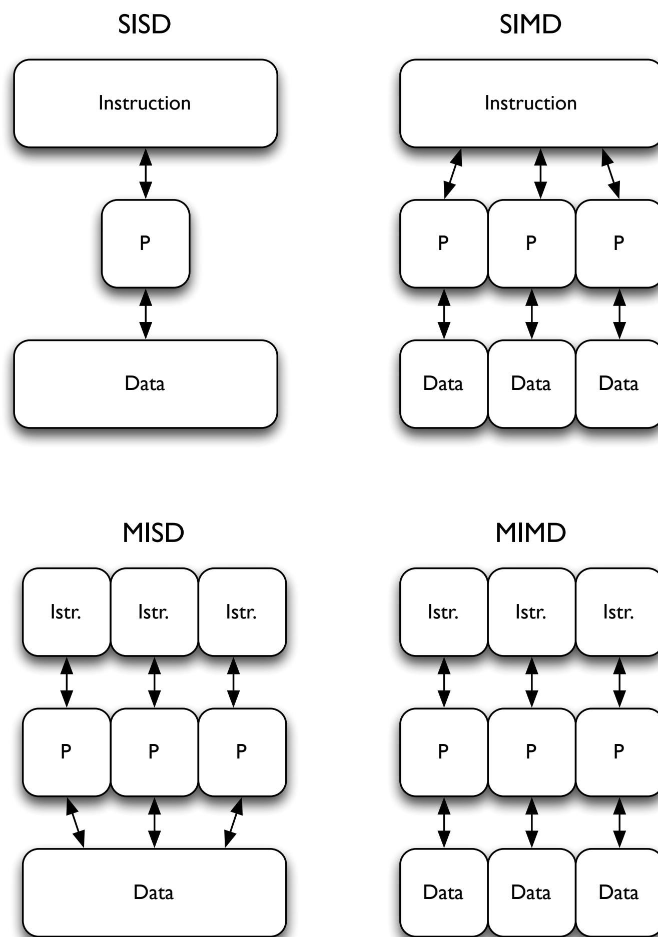


Figure 2.1: Flynn's Taxonomy of Computer Architectures.

(MPMD) design. These heterogenous applications use multiple programs to compute different parts of the simulation, often combining results from the different programs to calculate some final result [157].

2.1.2 Abstract Models of Parallel Computation

Having a general abstract model of parallel computation allows us to discuss different parallel machines without worrying about their specific hardware configurations. This provides an architecture independent description of the machine which can be used to analyse or model application behaviour.

One of the first models of parallel computation was the Parallel Random Access Machine (PRAM) model [52]. The PRAM model extends the concept of a conventional random-access memory machine to include a set of processing units. Each processing unit has its own local memory, and is connected to other processing units via a shared global memory. This simple model overlooks many hardware features of common architectures such as non-uniform memory access times, and contention on shared resources. Additionally, the reality of providing a large, shared global memory is prohibitively expensive at the scale required in contemporary computer systems.

The Bulk Synchronous Parallel (BSP) model is a successor to PRAM, and is more complex and inherently more representative of typical parallel programs [158]. As a model, BSP represents computation as a series of global super-steps executed by p processors. The pattern of each super-step is highly structured, and consists of: parallel computation, taking w operations; communication of h units of data between processes, which consumes $h \times g$ time on a network of bandwidth g^{-1}; and finally a global synchronisation barrier with a constant time l. In practice, barrier-type communications scale with the number of processes at $O(\log p)$. Figure 2.2 shows the general pattern for a BSP program.

The BSP model can be used to predict the total execution time of an al-

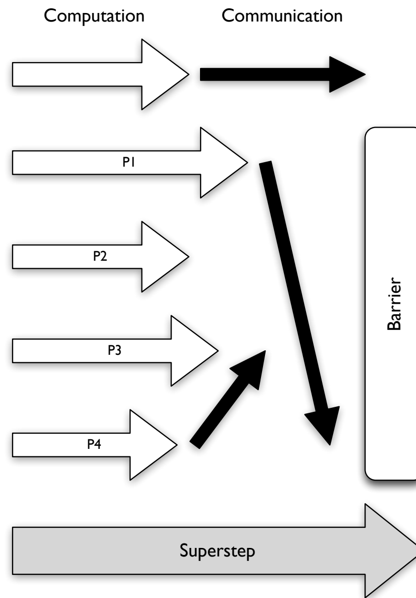


Figure 2.2: The three parts of the Bulk-Synchronous Parallel model superstep.

gorithm with S super-steps using the formula:

$$W + H \times g + S \times l \tag{2.1}$$

where

$$W = \Sigma_{s=1}^{S} w_s \tag{2.2}$$

and

$$H = \Sigma_{s=1}^{S} h_s \tag{2.3}$$

Whilst the utility of the BSP model for predicting execution time has been succeeded by later, more accurate, approaches such as LogP [39] and LogGP [4], as a general model for describing the behaviour of a typical parallel application, BSP is useful. Most of the applications discussed in this thesis follow this model. Additionally, software packages exist that provide primitives for developing BSP-based applications. The BSPLib package has been under active development for over 10 years [74]. A recent package, MulticoreBSP, aims to

bring the BSP model to modern multi-core processors, showing the BSP model can still be used more than 24 years after it was initially proposed [166].

2.2 Laws of Parallel Programming

The study of parallel programming often lies at the intersection between the practical and the experimental. Each of the programming models and supporting technologies described here is governed by a set of laws that succinctly and completely describe the limitations of parallel programming.

2.2.1 Amdahl's Law

Amdahl's law, published in 1967, provides a series of equations that govern the maximum speedup that can be achieved by a parallel program when compared to its serial runtime [6]. Speedup is defined as the ratio between the serial and parallel runtimes of a program. For N processors:

$$\text{Speedup}(N) = \frac{t_s}{t_p} \tag{2.4}$$

where t_s is the execution time of the program on a single processor and t_p is the execution time of the program on N processors. Another measure derived from the speedup is the parallel efficiency:

$$\text{Efficiency}(N) = \frac{\text{Speedup}(N)}{N} \tag{2.5}$$

A parallel program can be divided into two parts, one part in which the work is parallel, and one in which the work is sequential. It is because of this that Amdahl's law can state the maximum speedup of a parallel program will be determined by the runtime of the sequential part of the program:

$$\text{Speedup}(N) \leq \{\frac{1}{f_s + \frac{f_p}{N}}\} \tag{2.6}$$

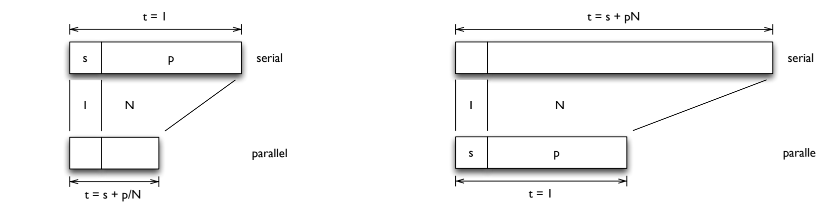
where N is the number of processors, and $f_s + f_p = 1$ are the sequential and parallel fractions of the program. As N tends to infinity, it is clear that the speedup will remain limited by the sequential fraction of the program.

2.2.2 Gustafson's Law

The speedups bounded by Amdahl's law can often be exceeded as users increase the amount of work in proportion to the number of processors. Gustafson recognised this, and in his 1988 paper *Reevaluating Amdahl's Law* he describes several cases where a speedup larger than Amdahl's proposed maximum was observed [66]. Executing an application in this fashion, known as weak-scaling, means that each processor has a fixed local problem size, rather than sharing part of a fixed global problem. As more processors are used, larger and larger problems can be solved. Gustafson argued that this mode of execution is typical for many users and programs, where the size of problem is scaled to use all available computing resources. This reasoning leads to the *scaled speedup* equation:

$$\text{Scaled speedup}(N) = \frac{s + p \times N}{s + p} \tag{2.7}$$

Gustafson observed that the serial runtime of the program s will contain start-up routines and serial bottlenecks which can make it a fixed cost that does not grow with problem size. When considering scaled speedup, if the input size is increased such that the parallel work p is larger than s then we will have good parallel scalability. For a serial processor to execute this work it would take time equal to $s + p \times N$. Figure 2.3 shows the difference between the fixed and scaled speedup. Scaled speedup provides a theoretical basis for analysing weak-scaled codes, which allow scientists to increase the size of the problems they solve by adding more hardware. The ever increasing number of cores in modern supercomputers are perfectly suited for solving huge problems using this kind of configuration.



(a) Fixed-size (strong-scaling) speedup.

(b) Scaled-size (weak-scaling) speedup.

Figure 2.3: Gustafson's Law for fixed- and scale-size problems. Scale-size problems often exhibit performance that exceeds the constraints of Amdahl's law.

2.3 Parallel Computing Hardware

Parallel computing hardware can be organised into three eras prior to today: (i) mainframe systems, (ii) parallel vector machines, and (iii) the distributed-memory machines. Each era defines a common hardware model, and with this hardware model comes one or more programming models that allow programmers and applications to make efficient use of the available hardware. The distinct hardware and programming models make transition between each era difficult, and the shift from the distributed-memory hardware model to the more complex many-core model of the exascale era will contain a range of challenges from both a hardware and software perspective.

2.3.1 Mainframes

The first era of supercomputers was that of the mainframe. Mainframes are general purpose computers designed for scientific computing. Examples of these include early IBM systems such as the IBM 701, and the first Seymour Cray-designed system, the CDC 1604. The CDC 6600 is considered the first real supercomputer, and provided full separation between IO and computation. Mainframes were serial computers, and often programmed using intimate knowledge of the target system and programming languages that would seem arcane to modern computer scientists.

2.3.2 Vector Machines

Vector computers are in the SIMD model of Flynn's taxonomy, and execute the same instruction on multiple data items simultaneously, giving them improved performance when compared to mainframe machines. The first vector machine developed by Seymour Cray after he left CDC, the Cray 1, is one of the most famous vector machines and one of the most successful supercomputers. Programming in the vector era often meant writing custom software to take full advantage of the vector processors, but some compilers were able to perform sophisticated code transformations to automatically vectorise code where appropriate. The main problem with vector machines was executing serial portions of code. Since the speed of a vector processor was only realised when processing multiple items simultaneously, serial portions of the program (where only one data item could be processed at a time) suffered from significant slowdowns. This serial bottleneck is neatly encapsulated by Amdahl's Law, and occurs at both the processor and system level on modern multi-core architectures.

2.3.3 Distributed-Memory Machines

The 1990s saw the rise of cluster computing, using cache-based reduced instruction set processors. Clusters are high-performance computers comprised of commodity components and fast interconnects, and allowed a much wider range of users to have access to the power of High-Performance Computing (HPC) platforms. Task-parallel programming models such as Message Passing Interface (MPI) evolved to allow programmers to use these machines effectively [111].

The development of these systems was driven by the fact that desktop computer components were becoming a commodity, and the size of shared-memory systems such as vector computers and mainframes was reaching a cost scalability limit. Fast ethernet interconnects and the free Linux operating

system helped allow anyone to build a supercomputer, known as a Beowulf cluster. The distributed computing era has been the dominant hardware model since the 90s. This has led to the development of robust programming models, languages and all their accompanying infrastructure.

2.3.4 Many-Core Machines and the Future of Supercomputers

Performance improvements in the first three eras of supercomputing were driven by the relentless increases predicted by Moore [112]. This period of significant performance improvement every generation was known as the "free lunch", and application developers could rely on each new supercomputer allowing them to solve larger problems faster [152, 153]. However, the reliability of these performance increases meant that research into application performance and efficient algorithms was sacrificed. This focus on scientific discovery rather than application maintenance means that codes have stood still as machines have advanced, making the switch from the distributed-memory era to future many-core architectures harder than previous leaps.

In 2001, heat dissipation and power consumption issues meant that CPU clock speeds began to stall and hardware manufacturers instead focused on increasing the number of processor cores on a single chip [113]. The free lunch was over, and in some cases clock speeds have decreased in order to accommodate the extra hardware required to manage an increased number of processor cores. These new multi-core chips widened the gap between theoretical and achieved performance, with typical applications performing at only a tenth of any supercomputers peak Floating Point Operations per Second (FLOPs) [88]. At the system, rather than the processor scale, concerns about power consumption have led hardware manufacturers to propose vastly different architectures to provide the necessary performance. Many-core architectures are the natural extension of the architectural trends introduced by multi-core processors, and

consist of processors with even more cores, running at even lower frequencies.

It is now common to see an accelerator attached to a typical multi-core processor. These devices are specialised for fast floating point performance and have their own memory space. In 2005, Fung et al. realised that the video-focused hardware of a consumer Graphics Processing Unit (GPU) could be harnessed for scientific applications through the use of complex shader programming languages [54]. In 2006, NVIDIA released their CUDA programming model and provided an easy-to-use method for programming GPUs. These new general-purpose GPUs (GPGPUs) offered huge improvements in computational performance over a traditional chip provided the application suited the new architecture.

Accelerated Processing Units (APUs) from AMD consist of a CPU and a Graphics Processing Unit (GPU) on a single chip, meaning that different hardware resources can be used to perform the computations for which they are most suitable. In 2012, Intel released the Xeon Phi, a co-processor with around 60 processor cores. These cores have more in common with a typical multi-core CPU than GPUs and can be programmed without any special programming language, but as with other accelerators aim to provide more FLOPs using many slower cores. The most customised form of accelerator is a Field-Programmable Gate Array (FPGA). These devices are produced by a variety of manufacturers, and provide high-performance at an extremely low power budget since the hardware is configured by the programmer to implement the required algorithm.

Machines such as IBM's Blue Gene series are customised for supercomputing. With a typical system containing many thousands of cores, each of which has four hardware threads, a parallel job running on a Blue Gene/Q may use up to one million concurrent threads. The nodes are connected with a custom torus network designed to minimise the time parallel tasks will spend communicating.

At the system level, current supercomputers exhibit a trend towards a

mixture of custom and commodity parts. Cray's XC30 supercomputer exemplifies this middle ground between a commodity cluster and a bespoke machine such a Blue Gene. The XC30 uses commodity processing components, such as Intel processors and NVIDIA GPUs, but houses them in custom packaging, and connects each node using a custom interconnect for increased performance.

2.4 Parallel Programming Models and Languages

Writing applications for parallel computers is a complex discipline, since the details of the communicating processes will be left to the application developer. The software infrastructure surrounding high-performance computing and parallel programming has grown considerable since early mainframes, and the incremental development used for scientific applications means that the programming languages used have not changed for many years.

The most common languages for writing scientific applications remain C, C++ and Fortran. The mathematical slant of Fortran means it is popular with domain scientists, but the object-oriented nature of C++ often allows computer scientists to write more flexible and maintainable code. Newer languages and programming models have been developed to support the more complex parallelism present in contemporary parallel architectures.

The mapping between the three levels of parallelism to the corresponding layers of the hardware/software hierarchy is shown in Figure 2.4. In this section we focus on the software side of the hierarchy and discuss the programming models and languages that can be used at each level.

2.4.1 Message-Passing

The highest level of the parallel programming hierarchy, message-passing, allows processes to communicate through the sending and receiving of messages. First used during the era of distributed memory machines, message-passing provides a way for processes to share data when they no longer have access

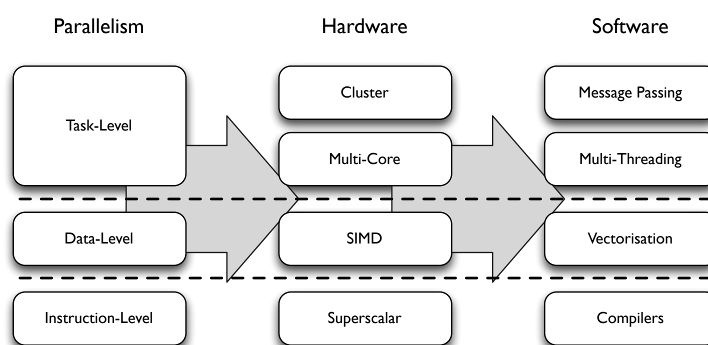


Figure 2.4: Mapping between hardware, software, and the levels of parallelism in a supercomputer.

to the same shared memory. This paradigm is typically used through a software library, with the canonical standard being the Message Passing Interface (MPI) standard. MPI provides an explicit style of data transfer, where processes must actively call send and receive functions in order to communicate. Other approaches to message-passing rely on a more implicit style of data transfer. One such approach, languages implementing the Partitioned Global Address Space (PGAS) model, transfers data behind the scenes based on knowledge inferred from the application code [136]. One example of a PGAS language is co-array Fortran, where the implicit data transfers are supported by extensions to the Fortran language [33]. Accessing memory on another node is given special syntax, and the compiler and runtime can use this knowledge to transfer data without the involvement of the programmer.

The ubiquitous nature of the message-passing model and its familiarity to scientific application developers means that HPC programs are almost always written using this paradigm, and it is often used to handle both inter- and intra-node parallelism. This means that the same code could be used from the start of the distributed-memory era (where single-core processors were connected) right through to the end of that era and the present day (where nodes typically contain one or more multi-core processors).

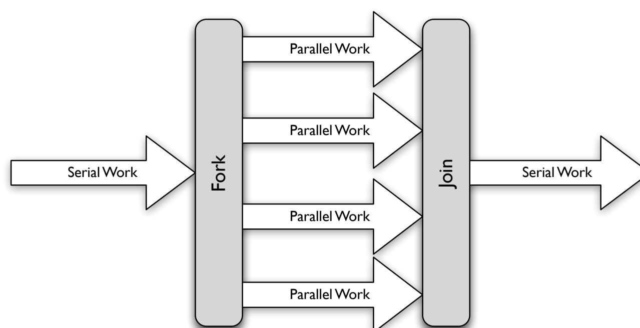


Figure 2.5: The fork-join model used for thread level parallelism in OpenMP.

2.4.2 Multi-Threading

The second level of the parallel programming hierarchy, the multi-threading level, is how multiple tasks execute concurrently within a supercomputer node. As seen in Figure 2.4, this level of the model maps to the cores of a processor and task-based programming models. Task-based parallelism is most often implemented using the fork-join model (see Figure 2.5). Serial sections of work are executed by one master thread. When a region of parallel work is encountered, multiple threads are created and each will be responsible for some portion of the parallel work. At the end of the parallel region there will be a synchronisation before the master thread continues to execute the next serial part of the code.

The most widely used task-based programming model in HPC is OpenMP [121]. The OpenMP standard defines a number of compiler directives that can be inserted into code and used as instructions to the compiler about which parts of the code should be executed in parallel. OpenMP offers two main forms of multi-threading: loop-based, and task-based. The loop-based style implements the fork-join model with a specific emphasis on executing the body of a loop in parallel. This kind of parallelism is often found in HPC applications, where arrays representing a physical domain are processed one element at a time in an iterative manner. The task-based model allows a programmer to create mul-

tiple task threads that process data concurrently.

Whilst OpenMP is the most popular threading model for HPC codes, other programming models and libraries exist that implement the multi-threading model. Technologies such as Intel's Cilk Plus, and Threading Building Blocks provide primitives that allow a programmer to write code that will be executed using multiple threads [82, 83]. At a lower level, the `pthreads` library is the standard way of writing threaded POSIX software [114]. Whilst not as frequently used in scientific computing, using the `pthreads` library allows the programmer to write much more explicit multi-threaded code. This avoids the thread-management overhead that is often present in the higher-level programming models.

As well as being used to program typical multi-core processors, the multi-threading approach can be used to program massively parallel accelerators such as GPUs. The OpenACC Application Program Interface is similar to OpenMP and allows the application programmer to mark code for parallel execution on a GPU through the use of compiler directives [120]. OpenCL and CUDA are programming models that rely on device-specific code being launched through calls to a programming library on the host CPU [119, 155]. Both these models use the single instruction, multiple thread (SIMT) approach to parallelism, and when a device kernel is launched, each thread runs one instance of the kernel and will operate on a separate data item.

2.4.3 Vectorisation

The lowest level of the parallel-programming hierarchy usually exposed to users, vectorisation, allows multiple data items to be processed in parallel. This level of parallelism can be accessed in four ways: using an auto-vectorising compiler, using directives to provide hints to the compiler, using intrinsic language functions to write high-level vector code, or writing code in assembly language and using explicit vector instructions.

Using an auto-vectorising compiler is the easiest method of accessing

```
for(int i = 0; i < 16; i++) {
  y[i] = a*x[i] + y[i];
}
```

(a) A simple loop amenable to vectorisation.

```
## y = ax + y
mulpd -272(%rbp,%rcx,8), %xmm0
addpd -144(%rbp,%rcx,8), %xmm0
movapd  %xmm0, -144(%rbp,%rcx,8)
```

(b) The vector instructions generated for the loop body.

Figure 2.6: Vector instructions generated for a simple loop. Each vector operation (with suffix pd) operates on two double-precision operands.

this kind of parallelism, however, it relies wholly on the compiler being able to recognise code that can be vectorised. A typical target for auto-vectorisation is a for loop. Figure 2.6 shows how a simple for loop can be converted into vector instructions by the compiler. Each instruction operates on two double-precision operands, and hence halves the number of iterations required in the loop. If it is to be vectorised, a loop must have known bounds and independent iterations. Since compilers are expected to produce correct code, if they cannot verify that these conditions are true then they will abandon any attempt to vectorise the loop. This can be frustrating for programmers, who may know that a loop can be safely vectorised but have no way to communicate this to the compiler. In this case, compiler directives can be used to advise the compiler that code is safe to vectorise.

Language intrinsics alleviate this problem by allowing programmers to use functions that will compile to one or more vector instructions in the assembly language of the target architecture. Intel provides a number of intrinsic functions for its Advanced Vector Extensions (AVX) instruction set. These functions can be mixed in with C or C++ code and provide a way to more explicitly control vectorisation without resorting to writing code in assembly language. Writing code in assembly language allows the programmer to be absolutely explicit about which instructions are executed, but there is no access to any of the

modern development tools that programmers take for granted. As such, this approach may be appropriate for small and performance critical portions of an application, but is difficult to apply across a whole application.

2.5 Performance Analysis and Engineering

Performance analysis is an important aspect of high-performance computing. Programming supercomputers is hard, and applications rarely obtain a fraction of maximum machine performance when first written. Additionally, the rapid development of HPC hardware means that the foundations on which the applications are running are constantly changing. An application that performs well on current hardware may suffer from any number of issues when moved to a new system, perhaps with a new processor or interconnect design. Performance analysis involves measuring the application performance through a number of metrics. The simplest measure is total runtime, but more granular measures such as number of MPI messages sent or bytes written to disk can also be useful. Performance engineering uses the knowledge gained during performance analysis to improve an application, most often by reducing runtime. Reasoning about, and improving, the performance of applications is a complex discipline with a multitude of approaches. Despite the simplicity of abstract models such as BSP and LogP, constructing a model that accurately represents a given application is an involved process that requires an understanding of both the application being modelled and the modelling technique being used. It is in part due to this difficulty that other forms of performance analysis are being used on the path to exascale. The formal models often fail to capture subtleties of complex modern supercomputer architectures, and the more direct approach of performance analysis via measurement provides actual performance data. In the context of this thesis, we consider performance analysis by means of benchmarking; using a representative program of some sort to measure, and then reason about, system and application performance.

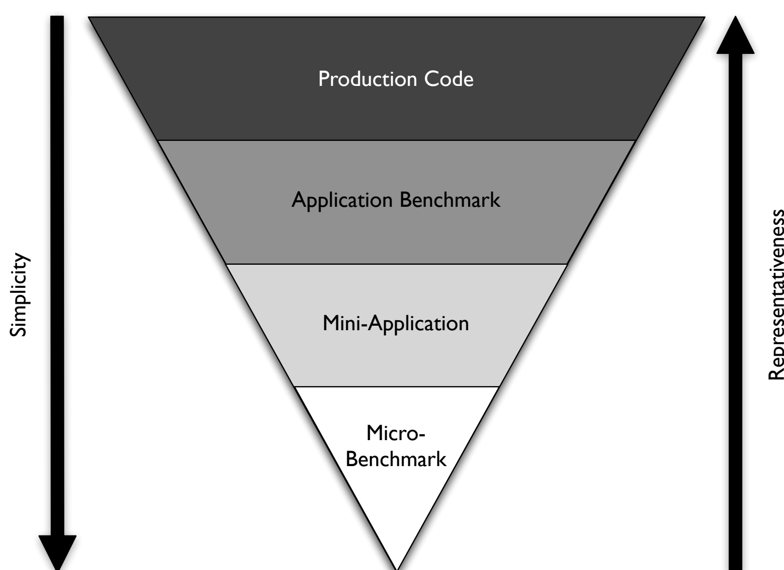


Figure 2.7: The representativeness and simplicity of the three classes of benchmark.

2.5.1 Benchmarks

Benchmarking is the act of measuring some aspect of a system. A benchmark is a small program designed to stress one particular part of a supercomputer. The output of a benchmark is a set of metrics, typically including runtime, that can be used to infer things about the performance of the hardware being measured. In high-performance computing, more complicated application benchmarks are often used. Production supercomputing applications are large, complex, and often commercially sensitive. A benchmark provides a way to measure aspects of a prospective system without needing to release sensitive source code. Despite being smaller than production applications, the degree to which a benchmark represents a production application often means it will still be large. Figure 2.7 shows the range of benchmark categories and their corresponding simplicity and representativeness.

Micro-Benchmarks

A micro-benchmark is designed to time specific machine components. The most famous benchmark is LINPACK [115], and more specifically, the portable High-Performance Linpack (HPL) implementation [48, 134]. A simple linear algebra program first used in the 1970s, the LINPACK benchmark has become responsible for determining which machine is classified as the fastest in the world. Performing dense matrix-matrix operations is a good test of floating point performance, and the output of the LINPACK benchmark is the number of floating point operations performed per second on the system being benchmarked. Other low-level numerical benchmarks include Livermore Loops, a collection of small loops designed to characterise the kind of operations performed by key applications at Lawrence Livermore National Laboratory (LLNL) [109], and the HPC Challenge benchmark suite [101], a set of seven tests to measure numerical, memory, and network performance.

Another example of a micro-benchmark, SkaMPI, is used to examine the performance of the machine interconnect using the MPI library [140]. The benchmark can be configured to perform multiple repeated communications, with each transfer being timed. The output of the benchmark is a set of timings, typically grouped by message size. This data allows an engineer to examine network performance in the machine being benchmarked. Alternative benchmarks for measuring network performance include MPPTest [63] and the Intel MPI Benchmarking Utility [84].

Other machine components such as memory can also be benchmarked. The STREAM benchmark measures memory bandwidth by executing a variety of read and write operations [107]. These operations are arguably more important that the famous FLOPs metric, since many modern codes are memory- rather than compute-bound, i.e. they are limited by how fast they can fetch data rather than how fast the data can be processed.

The problem with benchmarks like SkaMPI and LINPACK is that they very rarely mimic the behaviour found in real applications. Rather than in-

tensive sections using a small subset of the architectural capability, production applications contain a unique mix of memory access, computation, and communication. Whilst benchmarks that are cut-down (either in terms of code base or problem size) versions of production codes can be useful, the complexity inherent in a production application can make it difficult to reason about its behaviour and attribute different performance aspects to architectural capability. In the early 90s, the correlation between HPL and real application performance was good, but the regular memory access patterns exhibited by the dense linear algebra kernels no longer correlate with the more irregular pattern seen in production applications.

The recently proposed High-Performance Conjugate Gradient (HPCG) benchmark aims to provide a more realistic mix of computation, memory access, and communication that resembles production applications [47]. Unlike the regular memory access pattern of HPL, HPCG contains a mix of more complex communication and computation patterns that reward investment in collective communication operations and the performance of the local memory system. Both these characteristics have been shown to affect the performance of real applications. The legacy of HPL, and its use as tool for ranking the fastest supercomputers in the world, looks set to maintain some influence on the HPC design space, but HPCG now provides an alternative ranking system that should more accurately reflect FLOPs achieved by real applications.

Application Benchmarks

The benchmarks discussed so far are small in both number of lines of code, and the scope of which parts of a machine they are designed to stress. Typical scientific codes are large, and problems of interest can have runtimes of months. To make performance analysis via benchmarking feasible, application benchmarks must be smaller, and problem runtimes must be in the range of hours, rather than days or weeks. Application benchmarks aim to replicate the behaviour of a large parallel code but with fewer lines of code and less complex-

ity. Examples of application benchmarks include the NAS Parallel Benchmark Suite [12, 13], the ASC benchmarks [96], and codes such as Shamrock and Hydra from the United Kingdom Atomic Weapons Establishment [41, 70].

Application benchmarks fall short of being small enough to enable the rapid investigations necessary for future programming models and architectures. Hence in this thesis we consider mini-applications, a class of benchmark that is carefully designed to allow easy experimentation with programming languages, programming models and machine architectures.

2.5.2 Mini-Applications

Mini-applications are small, self-contained programs that embody the key performance characteristics of some key application [73]. They are a potential solution to the challenges raised in understanding and improving the movement of production codes to future platforms. Mini-apps rely on two key observations made by Heroux et al. in [73]. First, that most production applications spend the majority of their execution time in a small percentage of the total lines of code, and second, that much of the remaining code will be composed of sections that while mathematically different, will exhibit similar performance characteristics.

Using these two observations, it becomes easier to write a small application that accurately represents the most expensive portion of the application, and combines the rest of the performance-similar sections. Mini-apps are key tools for investigating new programming languages [87, 95] and testing new machines and architectures [15]. They can also be used as drivers for co-design, where software developers and hardware designers work together as supercomputers approach the exascale milestone [16, 49].

One of the first mini-apps, miniMD, is representative of the LAMMPS production code [73]. LAMMPS is a production application for simulating molecular dynamics that contains over 100,000 lines of code [137]. Rather than trying to provide all the complex functionality of LAMMPS, miniMD instead

focuses on a single algorithm that contains the core computational behaviour of the LAMMPS code. Focusing on a few loops that contain the majority of the computational core of the application means that the 3,000 lines of code that comprise miniMD can be easily used as a proxy for LAMMPS in performance engineering investigations.

Apart from the complexity of each level of the benchmark hierarchy, one major difference is the amount of coverage provided by each category. Production applications will contain a mixture of behaviours that stress the components of a supercomputer in different ways. For example, some portions of the application may stress the memory hierarchy with a series of random access requests, while other portions may be computationally intensive and perform calculations repeatedly with the same few data items. Different architectural characteristics can also influence the performance of different application classes. For example, a BSP-type program that proceeds by way of blocks of computational work with barrier-like communications at regular intervals will be sensitive to network latency. An application that can overlap communications and computation will be less sensitive to network performance, since the program can perform useful work while waiting for data to be transmitted across the network.

These specific behaviours can be covered by micro-benchmarks. The LINPACK benchmark introduced previously would correspond well to the computationally intensive application region described above. However, whilst a micro-benchmark may capture one specific behaviour well, it is the combination of these behaviours and their fractions that gives each production application a unique computational profile. Larger application benchmarks will cover multiple behaviours but the cost of this is increased complexity. With mini-applications, the goal is to develop a minimal set of these behaviours, so that not only the behaviours themselves are accurately captured, but also the relationship between them and the specific interplay between this combination of behaviours and a given hardware platform.

2.6 Summary

In this chapter we have considered the history of supercomputing and presented current trends in hardware and software. Whilst the impact of these trends on future hardware are yet to be verified, it is clear that we are approaching an age of complex new system design that will rely on new programming languages and models. We present a description of the most pertinent models and languages, and in particular we show how these models map to different layers of hardware. The laws in Section 2.2 provide a theoretical basis for evaluating application performance on supercomputers. Combined with benchmarks, programs designed to stress one particular aspect of a computer system, we can measure and reason about the performance of a given application class on a certain architecture.

Typical MPI-based applications have mapped well to the distributed multi-core architectures of the cluster era, but as clock speeds fall and parallelism increases, applications must adapt to take advantage of all the processing power offered by new, complex architectures. Shifting between the eras of supercomputing has been, and will remain, challenging due to the changing programming models, languages, and hardware, coupled with the complexity of production codes. Performance engineering, and specifically, performance engineering via mini-applications can offer a possible solution to exploring future hardware and software, and providing a path to move production codes into the many-core era of HPC.

CHAPTER 3

Computational Physics and Adaptive Mesh Refinement

Tackling the world's most challenging scientific problems has been the responsibility of supercomputers since their inception. The complex mathematical equations that describe our physical environment are almost impossible to solve using a pen and paper. Instead we must turn to the thousands of numerical operations per-second that can be performed by computer. To solve these equations computationally, they must be discretised. The equations are turned into approximations and solved at a number of points in time and space (see Figure 3.1). The collection of cells and nodes formed by the spatial discretisation is known as the mesh. Increasing the resolution, the number of points used to represent some region (for example, the number of points used to represent one square millimetre of space), provides a more accurate solution but requires more resources. It is this drive towards increased resolution that requires faster and faster supercomputers.

Solving equations at an increased resolution is more expensive both in terms of computational time and memory used. Adaptive Mesh Refinement

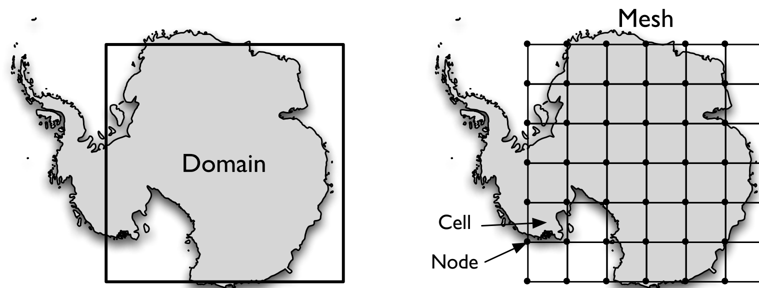


Figure 3.1: An example mesh used to discretise the spatial domain in a scientific simulation.

(AMR) is a computational technique where the resolution of the simulation is only increased in areas where it is most necessary. For example, when simulating a tsunami travelling across the ocean the location of the wave is the most important feature in the solution. What's happening in the rest of the ocean is much less interesting, and the impact of the wave is either negligible or easily approximated. An adaptive simulation would only simulate the area containing the wave at a high resolution, saving both time and memory.

In this chapter, we present an overview of the mathematical equations solved by the applications discussed in this thesis. Our applications are all from the domain of hydrodynamics, the study of liquids and gases in motion. There are a range of equations that can be used to represent hydrodynamic systems, and we present a detailed description of the Euler's equations; those that are used in the applications of interest to us. These equations can be solved in a number of ways, and we present two possible hydrodynamic schemes that can be used to solve these equations. We also motivate the use of AMR with a number of example problems, before providing a detailed description of how partial differential equations (used to represent the hydrodynamics systems of interest) can be solved on an adaptive mesh.

3.1 Computational Hydrodynamics

Hydrodynamics is the study of liquids and gases in motion. Computational hydrodynamics involves simulating this motion using computers. This broad field is applied to great effect in a wide range of scientific domains, including astrophysics, climate modelling, and explosion simulations. The motion of the liquid being studied can be described using a number of formulae, ranging from Schrödinger's equations that describe the quantum mechanical interaction of particles, to partial differential equations that describe the fluid at a statistical level.

Each formula describes the motion of the fluid at a different level of details. Schrödinger's equations are the most closely connected with simulating the physical world since they model the interaction of particles. Other formulae that describe the motion of fluids include Newton's law, the Liouville equation, the Boltzmann equation, and the Navier-Stokes equations. The formula can be solved in a variety of ways. Schrödinger's equation, the Boltzmann equation and the Navier-Stokes equation can be solved directly, Newton's law is typically solved using molecular dynamics, and the Liouville equation using Monte Carlo methods.

The formulae chosen and the solution method typically depend on the scale of interest and the density of the fluid. Oran and Boris present a study of the validity of each of these representations [123]. For fluids with low density, Direct Simulation Monte Carlo is used, but with higher density fluids and larger length scales, the Navier-Stokes equations provide a better description of the system. The mean-free path—the average distance a particle can travel between collisions—of the system of interest can be used to determine the most appropriate equation.

The primary domain of this thesis is shock hydrodynamics, where gases move under extreme pressures and temperatures. Defined as hydrodynamic flow driven by strong shocks, these kinds of flows occur at hypersonic speeds

with materials moving faster than sound. Pressure waves cannot escape the hypersonic material flow, and build up, forming shock waves and creating discontinuities in all fields quantities (pressure, energy and density).

These hypersonic flows typically start in one of three ways: (i) from an initial pressure discontinuity, (ii) from a large jump in velocity (such as a fast body impacting a stationery body), or (iii) from a rapid increase in internal energy. Examples of the first case can be seen in the test problems described later in this chapter. The flows we consider can best be described using Euler's equations (a form of the more general Navier-Stokes equations), described in detail in the next section.

3.1.1 Euler's Equations

Euler's equations are a set of three partial differential equations, corresponding to the Navier-Stokes equations with no vorticity (rotational forces), no heat conduction, and no viscous stress [8]. The equations describe the conservation of mass, energy and momentum in a system:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{3.1}$$

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u} \otimes \rho \mathbf{u}) + \nabla p = \mathbf{0} \tag{3.2}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot (\mathbf{u}(E + p)) = 0 \tag{3.3}$$

Here, ρ is the material density, p is the pressure, E is the total internal energy, and $\mathbf{u}$ is the velocity. This system of equations is closed with a fourth equation, the equation of state, that relates the pressure and energy of the material. The simplest state equation is the ideal gas equation:

$$p = (\gamma - 1)\rho e \tag{3.4}$$

where γ is the ratio of specific heats (the ratio of the heat capacity at constant pressure to the heat capacity at constant volume), and $e = E/\rho$ is the specific

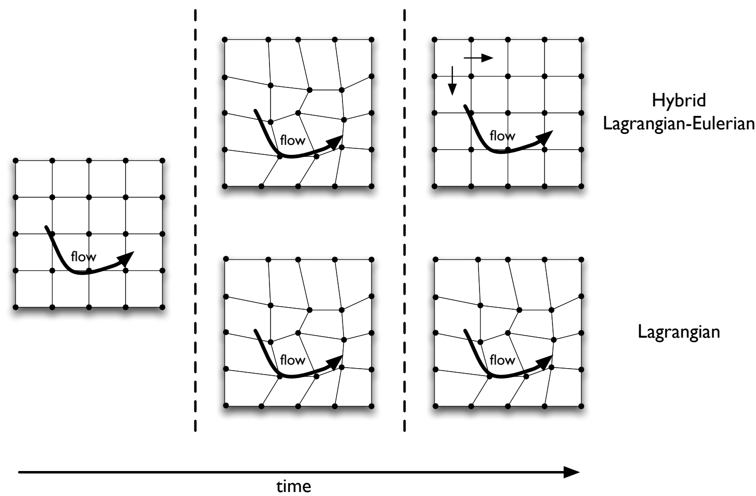


Figure 3.2: The difference between the Lagrangian and Eulerian hydrodynamics schemes. Note the lack of the remap to the original grid in the Lagrangian scheme.

internal energy (energy per unit mass).

Choosing a suitable set of equations for representing the motion of the fluid is only the first step in solving a computational hydrodynamics problem. Euler's equations can be solved using one of many hydrodynamics schemes. The two most common approaches are the Lagrangian and Eulerian schemes. We briefly describe both, and discuss why a hybrid Lagrangian-Eulerian method is used in CleverLeaf.

The Lagrangian and Eulerian schemes differ most in their frame of reference. In a Lagrangian method, one can think of the simulation mesh as being embedded in the fluid flow. The mesh nodes move with the fluid, providing an accurate representation of some features (such as interfaces between multiple materials) as mesh nodes gather near areas of interest. In an Eulerian method, the simulation mesh is fixed, and rather than the mesh nodes moving with the fluid, the fluid flows through the cells. Figure 3.2 shows the difference between these two schemes pictorially.

The main benefit of using an Eulerian hydrodynamics scheme, particularly in shock hydrodynamics, is that the simulation is numerically robust. In

the Lagrangian case, since the mesh nodes move with the fluid flow the mesh can become tangled, with mesh cells turning inside out. This is a numerical impossibility that will cause the simulation to fail but can be avoided with a fixed Eulerian mesh. However, the key weakness of an Eulerian scheme is that the mesh, and hence the simulation resolution, is fixed. A Lagrangian scheme will naturally refine around features in the solution due to the movement of the nodes. By applying AMR to the Eulerian method we can partially mitigate this weakness.

At the implementation level, an Eulerian method can use a much simpler mesh, and data layout is known implicitly. The structured mesh can easily be represented by an n-dimensional array, providing fast, regular data access patterns. The computational stencil required to update each point is fixed allowing for further optimisations to be applied. A Lagrangian application lacks this mesh structure, and often requires array indirection when accessing mesh data. Nodes, edges, and cells may be stored as lists with connectivity between elements being defined explicitly. Our CleverLeaf mini-application is designed to be representative of Eulerian shock hydrodynamics codes using a hybrid Lagrangian-Eulerian scheme, and hence we use this method.

3.1.2 Lagrangian-Eulerian Scheme

We solve the Euler equations on a staggered-grid using a predictor-corrector method that is second-order accurate in both space and time. This Lagrangian-Eulerian method is good for simulations involving strong shocks, and is used in many production hydrodynamics codes including CTH and BBC [108, 151]. Despite the Lagrangian step used in the first half of the algorithm, this method is considered Eulerian since mesh nodes remain stationary. In this section, we present a description of the computational grid, and an overview of the numerical algorithm. Details specific to adding AMR to this scheme are reserved for Chapter 5.

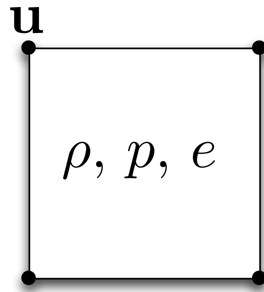


Figure 3.3: Cell variable positions in the Lagrangian-Eulerian scheme. The velocity vectors (**u**) are defined at the cell vertices, and the scalar fields: pressure (p), specific internal energy (e), and density (ρ) are defined at the cell centers.

Grid Description

The equations are solved on a staggered grid, where each computational cell has the form shown in Figure 3.3. The velocity vector, **u** is defined at the cell vertices. Each scalar field: pressure p, energy e, and density ρ; is defined at the cell centres.

Timestep Control

The timestep control uses the Courant-Friedrich-Lewy (CFL) condition; the maximum sound speed is an upper bound for Δt, and the timestep is thus limited to the time it would take for the the fastest sound wave to cross a cell [37]. Since Δt is calculated using the current data a multiplicative safety factor is used to ensure that the timestep remains within the stable limit for the whole timestep. Two additional tests are applied to the timestep: the first to ensure that no two vertices can cross as the mesh deforms; and the second to ensure that a cell cannot deform to the point that its volume will become negative. Intuitively, the chosen timestep ensures that a cell cannot turn "inside-out" due to the motion of the vertices during the Lagrangian step.

Lagrangian Step

The Lagrangian step is performed using a second order predictor-corrector scheme, where the solution is advanced by half the Δt to provide an estimate of the mid-step energy and density. These values are used to evaluate the equation of state to produce a half-step pressure (called the predictor). The gradient of these pressures is then used to calculate the acceleration and a half-step velocity on each node (the acceleration step). The start of step grid is now advanced by this half-step velocity for the full timestep; this is the corrector. The advantage of the predictor-corrector method is that all mesh variables finish at the same time level, unlike a leapfrog method, where cell and node centred variables are always half a step apart [26].

Advective Remap

Remapping the Lagrangian solution back to the Eulerian grid is accomplished with two one-dimensional sweeps (in the x and y dimensions), where the transport of mass, internal energy, and momentum across cell boundaries is calculated. Splitting the advection into separate x and y sweeps allows Van Leer's monotonic 1D method to be used [159]. Because the grid is not actually deformed, edge-centred velocity values are used to approximate the volume of material swept through a face, and these volumes are used to transport mass and energy back into the cell.

The momentum advection is used to remap nodal velocities from their Lagrangian position back to the Eulerian mesh. This takes place on the staggered grid, since node-centred velocity values are updated. Momentum is advected and then converted back to velocity to ensure conservation of momentum. This advection scheme conserves mass, internal energy and momentum but kinetic energy is not conserved due to the staggering of mass and velocity in a cell.

Algorithm 3.1 Lagrangian-Eulerian hydrodynamics scheme.

while $t < t_{end}$ **do**
  Evaluate equation of state
  Calculate stable Δt

  Lagrangian step
  Predictor
  Evaluate equation of state at $t = n + \frac{1}{2}$
  Acceleration
  Corrector
  Calculate fluxes

  Advection step
  Cell-centred advection
  Momentum advection in x
  Momentum advection in y
  Cell-centred advection
  Momentum advection in x
  Momentum advection in y
end while

3.1.3 Test Problems

To verify the accuracy of a computational physics application, test problems with analytic solutions are often used [9, 29, 53, 91, 165]. A problem having an analytic solution means that when the initial conditions are substituted into the equations of interest, Euler's equations in this case, the resulting system of equations can be solved exactly and does not need to be approximated. Given some exact solution, we can solve the same problem in our application and check that the results are the same as the analytic solution, or at least converge towards it as the resolution of the simulation increases.

In this thesis, we use three test problems: Sod's shock tube problem [147], Woodward and Colella's interacting blastwaves problem [165], and the Sedov blastwave problem [144]. All three problems are single-material problems containing multiple regions of ideal gas with $\gamma = 1.4$. Both Sod's shock tube problem and the Taylor-Sedov blastwave problem have analytical (exact) solutions. The Woodward-Colella interacting blastwaves problem can be solved to mesh-convergence, meaning that once a certain resolution is reached the numerical

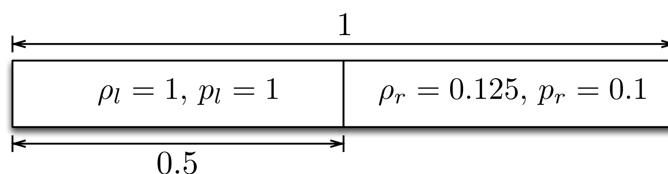


Figure 3.4: Initial conditions for Sod's shock tube problem.

answer will no longer change. A problem of this type is useful for evaluating how quickly the results produced by an application converge towards the correct answer as the resolution of the simulation is increased.

Sod's Shock Tube

The shock tube problem described by Sod provides a good test of a code's ability to capture contact discontinuities, shocks, and the rarefaction profile [147]. Consisting of two regions of fluid of different initial densities and pressures, the fluids are initially at rest, with the initial conditions being specified as follows (see also Figure 3.4):

$$\rho_l = 1 \tag{3.5}$$

$$\rho_r = 0.125 \tag{3.6}$$

$$p_l = 1 \tag{3.7}$$

$$p_r = 0.1 \tag{3.8}$$

The interface is at the point $x = 0.5$ and γ is 1.4 throughout the problem. At time $t > 0$ the two regions begin to interact, with a shock wave forming and travelling towards the right-hand boundary of the domain.

Woodward-Colella Interacting Blastwaves

Despite not having an analytic solution, Woodward and Colella's interacting blastwaves problem can be easily solved to convergence [165]. The interacting blastwaves problem consists of a two reflecting walls separated by distance unity. The density $\rho = 1.0$ throughout the problem, and three regions of ideal

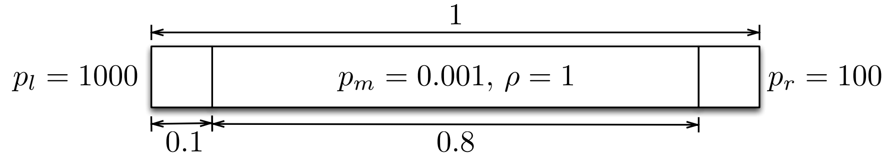


Figure 3.5: Initial conditions for the Woodward-Colella interacting blastwaves problem.

gas with different initial pressures are used to create the strong shocks (see Figure 3.5). The initial pressures in the left, middle and right regions of the domain are:

$$p_l = 1000 \tag{3.9}$$

$$p_m = 0.001 \tag{3.10}$$

$$p_r = 100 \tag{3.11}$$

The left region makes up the leftmost tenth of the volume; the right region, the rightmost tenth. The ratio of specific heats $\gamma = 1.4$.

Sedov Blastwave

The Sedov blastwave problem involves the self-similar evolution of a cylindrical blastwave from an initial point source of energy [144]. For an Eulerian hydrodynamics application, the particular challenge in simulating the Sedov problem comes from the circular shape of the shockwave, something which an Eulerian mesh can only approximate. We simulate a quarter of the problem, with the initial energy deposited in a single cell at the origin. The specific internal energy of the cell is set such that the total energy deposited is 0.25. Using a resolution of $\Delta x = 0.01$ means that the specific internal energy e of the cell is set to 2500. All other cells are given energy $e = 0$. Density ρ is unity throughout the domain. These initial conditions are shown diagrammatically in Figure 3.6. The shock wave should reach radius $r = 1.0$ at time $t = 1.0$, and hence we run our simulation to this physical time.

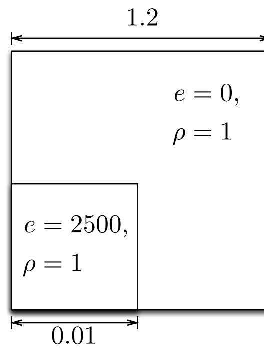


Figure 3.6: Initial conditions for Sedov's blastwave problem.

3.2 Adaptive Mesh Refinement

Adaptive mesh refinement (AMR) is a computational technique used to increase the accuracy of a simulation in the areas of a problem where it is most effective. Applying more computational resources to select parts of the domain means both application runtime and memory usage can be reduced. Developed by Berger et al. [21, 23], block-structured AMR has been successfully applied to domains including cosmology, astrophysics, and shock hydrodynamics [28, 53, 139]. For simplicity, throughout this thesis we use the terms adaptive mesh refinement and AMR to refer to the block-structured method unless explicitly stated.

Quirk illustrates the benefits of AMR with the example of a detonation rate stick [139]. The area of interest (the detonation front) is only 0.02 mm wide, but the whole stick is 100 mm long and 100 mm in diameter. If the entire domain was simulated at the resolution necessary to resolve the detonation front, over one billion mesh cells would be required. Applying AMR to this problem reduces the total number of cells to around 100 thousand, allowing huge savings in simulation time and memory use. Other domains in which these disparate scales are observed include astrophysics (stars in galaxies), military applications (small projectiles impacting much larger structures), and laser fusion experiments. Using AMR can improve runtime and decrease memory usage by reducing the amount of cells necessary to solve a problem with the required

accuracy. In all the domains mentioned, interesting scientific problems can be intractable without AMR because of the scale of the calculations required. Improving the performance of AMR will allow more of these important problems to be solved without increasing resource usage.

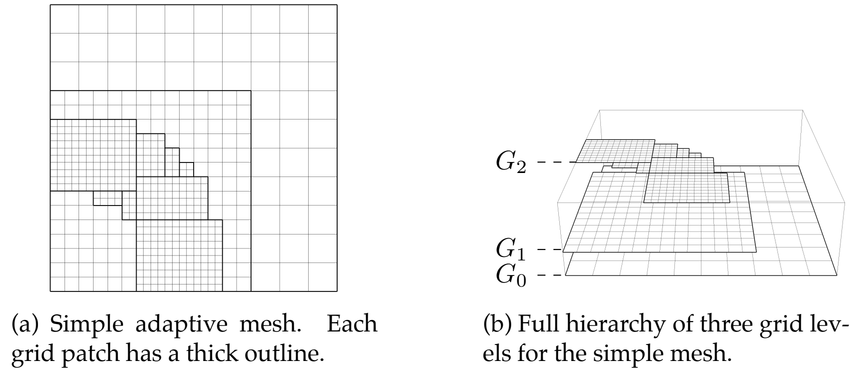
The method proposed by Berger provides a flexible way of solving a system of equations on a sequence of nested, logically rectangular grids, along with a procedure for managing and updating these grids as areas of interest move through the problem space. As in Berger et al. [21, 23], we provide a formal notation for these grids. The coarsest grid is the base grid, specified at the start of the computation and denoted G_0. It may be composed of several possibly overlapping patches. This base grid remains fixed throughout the simulation. Each component patch is denoted $G_{0,j}$, and thus G_0 is the union of its components $G_{0,j}$:

$$G_0 = \cup_j G_{0,j} \tag{3.12}$$

During the simulation, refined sub-grids of patches will be created in response to features in the solution. Sub-grids are not placed *in* the coarse grid, but on top of it. Each sub-grid is defined independently and has its own solution vector, and can be advanced almost independently of all other grids. These independent grids provide a natural method of domain decomposition allowing for easy parallelisation of the algorithm.

Fine sub-grids can contain finer sub-grids within their boundaries. Sub-grids are recursively generated to provide the necessary level of refinement, creating a hierarchy of grid levels. The coarse grid G_0 is at level 0 in the hierarchy. Sub-grids of G_0 are part of G_1 and are described as level 1 refinements. Refined grids within G_1 are at level 2. A nested sequence of sub-grids may be created to cover a portion of the domain. Figure 3.7 shows an example hierarchy containing three grid levels.

The mesh spacing, or resolution, h_l for each grid level l is normally specified in advance, where each h_l is an integer multiple of h_{l-1}. The relationship



(a) Simple adaptive mesh. Each grid patch has a thick outline.

(b) Full hierarchy of three grid levels for the simple mesh.

Figure 3.7: Example adaptive mesh and the corresponding grid hierarchy.

between the mesh spacing at each level is typically specified as the refinement ratio:

$$r_l = \frac{h_{l-1}}{h_l} \tag{3.13}$$

Grids at different levels of the hierarchy must be properly nested. A fine grid must start and end at the corner of a cell in the next coarser grid, and there must be at least one level $l-1$ cell separating a grid cell at level l from a cell at level $l-2$ in any direction unless the cell is at the physical boundary of the domain. The proper nesting requirement does not require a fine grid to be contained in only one coarse grid, so one fine grid may be nested in two or more coarser grids.

Refinement of the grid is arranged around a hierarchy of levels. Figure 3.7b shows an example adaptive hierarchy containing three levels. Each level is a collection of cells at the same resolution i.e. each cell has the same physical size. Levels are often described by their relation to the current level being considered. Levels with a higher resolution are described as *finer*, and levels with a lower resolution are described as *coarser*. The coarsest level of refinement, *level 0*, will always cover the whole problem domain. The resolutions of the levels are related by the refinement ratio, which defines the factor of increase in resolution between two levels. For a given refinement ratio r and level n, there will be r^n cells in each dimension for every single cell on level 0.

In Figure 4.3b the refinement ratio is $\{2, 2\}$, hence on level 2 there are 4 cells in each dimension for every level 0 cell. A pair of refinement ratios is used to describe the possibility for different ratios in each of the problem dimensions.

3.2.1 Berger's Integration Algorithm

First, an initial hierarchy must be created. To create the patch hierarchy, cells of interest on the coarsest level are tagged. The tagged cells are then clustered into a large rectangular region, which may also contain untagged cells. This large region is repeatedly split along its largest dimension until the percentage of tagged cells (the efficiency) reaches a certain threshold. This collection of patches will then form the new finer level. This procedure is repeated until the maximum number of levels is reached.

Each patch can be advanced through the integration step independently of any other, and the timestep provides an implicit synchronisation point for the levels. The equations are solved in exactly the same way on every patch as they would be solved in a simulation without AMR, regardless of the refinement level and physical position of the patch within the problem domain.

To parallelise the algorithm, the fact that each patch can be advanced independently is used, and patches are shared between MPI processes. Each patch will require some *halo* data which exists in additional cells around the patch edge. The halo data provides boundary conditions for the system of partial differential equations. Halo data for each patch can be filled in one of three ways: (i) with the physical boundary conditions, (ii) with the data from a neighbouring patch on the same level, or (iii) with the data from a neighbouring patch on the next coarsest level. When data is transferred between levels it must be interpolated to correctly fill the increased number of smaller cells on the fine level. At the end of the integration step, the more accurate fine solution is transferred to the coarser levels in the hierarchy.

Areas of interest in problems are rarely static, so the hierarchy must evolve throughout the simulation to ensure that the interesting areas of the

problem remain covered with the finest resolution cells. To adapt the grid, a regridding procedure is used. Regridding begins by tagging cells for refinement, and clustering these tagged cells into a large, rectangular region. This region is then split along its largest dimension until the required efficiency threshold of tagged cells is reached. Unlike the initialisation of the hierarchy, the regridding procedure is applied from the finest level down to the second coarsest. This ensures that the new fine level is always fully nested within the next coarser level. Once the new patches have been created, the solution can be transferred over from the old hierarchy. In cases where a given cell continues to exist at the same resolution, data can be copied directly from the previous hierarchy. In areas newly covered by the fine mesh, data is interpolated from the next coarser level. The simulation continues in this fashion, advancing and regridding until the desired end time is reached. Algorithm 3.2 summarises the AMR integration algorithm.

Algorithm 3.2 AMR integration algorithm.

while $t < t_{end}$ **do**
  Calculate stable Δt
  for $l = 0$ **to** l_{max} **do**
    Set boundary conditions for l
    Integrate l
  end for
  for $l = l_{max}$ **to** 1 **do**
    Coarsen l to $l - 1$
  end for
  if Regrid **then**
    for $l = l_{max} - 1$ **to** 0 **do**
      Regrid l
    end for
  end if
end while

3.2.2 Alternative Adaptive Mesh Refinement Approaches

The block-structured AMR technique presented by Berger is only one of a number of approaches for managing the solution of partial differential equations of an adaptive mesh. Cell-based AMR takes a much more fine-grained approach

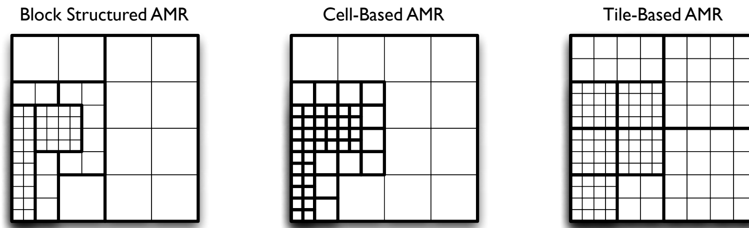


Figure 3.8: Mesh configurations for block-structured, cell-based, and tile-based AMR.

to adapting the mesh, with individual cells being split into finer cells. This means that meta-data can be managed using a tree, which provides much more explicit connectivity between a cell and its neighbours and children. Whilst this reduces some of the meta-data management complexity, it does mean that data will no longer be stored contiguously in memory. Additionally, a cell-based AMR approach will require a modified integration algorithm to deal with disjoint nodes at level boundaries. Applications using cell-based AMR include RAGE and CLAMR [61, 116].

The other approach for supporting AMR is a combination of the block-structured and cell-based approaches. By using *tiles*, which can either be seen as fixed-size patches or regular clusters of refined cells, the meta-data associated with each tile can be known explicitly, since all possible tiles are known in advance. Tiled AMR can be stored using a tree structure, as in PARAMESH [103], or it can be used as a constraint on a more general patch-based library, with the additional assumptions that are valid being used to increase application performance. Applications using the tiled approach include the Arches code, based on the Uintah framework [81, 100], and FLASH [53].

Figure 3.8 highlights the different mesh configurations that might be generated in each of the three schemes. The units of work (patches, cells, or tiles) are outlined. While a cell-based scheme allows the fewest total number of cells, the patch-based and tile-based schemes provide more regular patterns of computation that can be more efficient.

As well as more complex numerics, AMR adds a number of compu-

tational challenges to an application. Additional housekeeping code, associated with managing an adaptive mesh must be added, and can become a large proportion of simulation runtime. The nature of an AMR simulation revolves around the management of dynamic computation and data structures that must be mapped to hardware. On current architectures, fast serial cores mean that this dynamic behaviour does not cause problems. However, on future architectures like graphics processing units, managing this data (and possibly transferring it between memory spaces) could add complexity to the code and add overhead to the simulation.

Each AMR approach outlined above has different implications for how it will map to future hardware. The patch-based approach we use in this thesis is amenable to GPUs, with patches forming large, contiguous chunks of work. The tile-based approach has these same benefits, with the additional advantage of all units of work being a regular size. These regular sized units of work can be easier to load balance. The cell-based AMR approach is the most dynamic, and possibly the most difficult to map to future hardware, since the computational stencil used on each cell is entirely dependent on its neighbours. Investigating these issues and determining how to map a dynamic application to complex future architectures is an essential research area as High-Performance Computing (HPC) approaches the exascale era.

3.2.3 Adaptive Mesh Refinement Libraries

Supporting adaptive mesh refinement in an application requires managing a large amount of dynamic, complex, and distributed data. Abstraction in the software is often used to provide flexible implementations of key components such as a patch. This means that AMR functionality is well suited to being encapsulated in a purpose-built library. Many existing libraries provide the concepts necessary to construct a structured AMR application, including Chombo [35] from Lawrence Berkeley National Laboratory, SAMRAI [164] from Lawrence Livermore National Laboratory, and PARAMESH [103] from NASA Goddard

and Drexel University.

Extending an application to support AMR adds additional housekeeping code, associated with managing an adaptive mesh. This metadata and bookkeeping overhead can become a large proportion of simulation runtime. Reducing the cost of storing and managing the metadata required to advance a simulation on an adaptive mesh is an active area of research, and there are a number of possible techniques used to help ensure that the required algorithms and data structures are scalable [65, 99, 100]. For example, the metadata about the location of patches in the hierarchy can be distributed, saving space but adding complexity [64]. Using a library means that users and application developers are no longer responsible for writing this complex code, and can instead focus on scientific and domain-specific concerns.

The components found in the AMR hierarchy provide a convenient program design template, allowing concepts to be encapsulated inside classes, such that most of the program code can act on these objects with some level of abstraction. Development of AMR libraries focuses on both scalable implementations of the core AMR algorithms: regridding, synchronisation, and boundary conditions; as well as on the usability of the package from a user perspective. Here we present an overview of current AMR libraries.

The AMR software released by Berger contains all the numerical routines needed to implement AMR for hyperbolic conservation laws in two- or three-dimensions [19]. However, this Fortran package is not designed to provide an object-oriented view of AMR concepts, and as such, can make it more difficult to integrate with existing applications, or to ensure that new applications are written in such a way as to be extensible and easy to maintain.

The AMRCLAW library is a general framework for simulating wave propagation algorithms on an adaptive mesh, and used for modelling tsunamis and other ocean phenomena as part of the GEOCLAW software [20, 22, 32]. The AMR implementation follows Berger's original formulation but with some specific modifications for the simulation of wave propagation.

The BoxLib library from the Center for Computational Science and Engineering and Lawrence Berkeley National Laboratory (LBNL) was the first of these more component-based libraries [38]. BoxLib provides abstractions for a programmer to use when creating an application: a global index space, rectangular regions of the index space, data defined on the regions of the index space. Each of these abstractions is a class, so also provides a set of operations that provide an intuitive way to work with the data. Based in part on the work of the BoxLib developers, Chombo is another C++ AMR framework from LBNL. Designed around supporting parallel AMR calculations at a range of scales, the Chombo framework has been used for both domain-specific scientific simulations, as well as research into scalable AMR techniques [34, 160]. The Berkeley libraries have provided a well designed set of abstractions for AMR that are used by many other libraries.

The AMROC package is a generic block-structured AMR package written in C++ [44, 45]. AMROC's design focuses on the definition of routines that will advance the simulation on a single patch. These routines are primarily the numerical integrator, the physical boundary settings, and the initial conditions. The AMROC package adds an additional constraint by not distributing refined patches to a processor other than that which owns the coarse region of the domain. This constraint means that most of the AMR algorithm can be performed locally, avoiding the complex communications required when patches are distributed. However, the problem with this approach is that it may lead to load imbalance. Nevertheless, the AMROC package has been used to perform detonation simulations in parallel on up to 48 processors [43].

The PARAMESH package, from Drexel university and the NASA Goddard Space Flight Center, contains a set of Fortran subroutines designed to allow a developer to extend an existing serial application with AMR capability [103]. Rather than allowing arbitrary patches, PARAMESH uses a fixed subgrid approach, where grids can be toggled on and off, dependent on whether or not they contain an area of interest. Each sub grid is identical in logical struc-

ture to its parent, so a grid containing 6×4 cells would be refined into four 6×4 sub-grids, where the spatial resolution is twice as fine.

The Structured Adaptive Mesh Refinement Application Infrastructure (SAMRAI) package from Lawrence Livermore National Laboratory is a collection of AMR abstractions, with design roots in the work of the Berkeley developers [78]. Like Chombo, SAMRAI is used both as a framework for algorithmic research in AMR and for large scientific simulations. The US Department of Energy's Exascale program, and the close working relationship that the University of Warwick has with members of Lawrence Livermore National Laboratory meant we selected the SAMRAI package when designing CleverLeaf. The SAMRAI library is described fully in Chapter 5.

Adaptive mesh refinement can also be supported using dynamic runtime systems, such as Charm++ and Uintah [100, 125]. Charm++ organises computation around the concept of migratable objects, which are created by dividing the problem space up into chunks of work. The objects are assigned to processors, and relationships created between the objects allow for communication of boundary conditions. In an AMR application, the concept of migratable objects maps neatly to patches. Uintah uses a task-graph approach to describe computational tasks and data communication. A scheduler can then assign tasks to processors based dynamically, allowing for a high-degree of parallelism. As with Charm++, using tasks to represent patches allows the development of AMR applications within Uintah. Other dynamic runtime systems include Overture and GrACE [68].

3.3 Summary

In this chapter we have described the partial differential equations used to represent the motions of fluids, and shown how they are solved computationally. Understanding hydrodynamics is essential in a range of industrial and research contexts such as astrophysics, defence, and the oil and gas industries. The dis-

cretisation of the equations influences the accuracy of the solution, but a high-resolution calculation requires more computational resources. AMR is a technique used to only increase the resolution of a computational simulation in areas where it will be most effective. Combined with the observation that most scientific domains exhibit this locality, where important problem features such as shock waves are confined to a small portion of the domain, we are able to reduce the number of resources required while maintaining solution accuracy.

We discuss Euler's equations, one form of equations describing fluid motion, and present an outline of the solution scheme used in the two hydrodynamics mini-applications described in this thesis. Supporting AMR in an application requires managing a large amount of dynamic, complex, and distributed data and software-level abstraction can provide flexible implementations of key components such as a patch. We consider examples from the literature of available AMR libraries, and discuss the AMR library we chose to use for the work presented in this thesis.

Whilst AMR has the potential to deliver results faster, it requires complex communication and management to ensure that the simulation is advanced correctly. This complexity can harm application performance, and understanding and improving this performance on current and future architectures is an issue at the core of this thesis.

CHAPTER 4

Performance Engineering with Mini-Applications

The United States of America's Department of Energy has maintained that exascale computing power (10^{18} Floating Point Operations per Second) will be ready to use in the next decade [5, 89]. Whilst vendor technology roadmaps and research and development strategies are tightly protected by non-disclosure agreements, it is accepted that the first exascale systems will look dramatically different to traditional supercomputer architectures [89]. Some predictions for an exascale architecture feature accelerator-type devices with slower processing cores and vastly increased opportunities for executing many instructions in parallel. Multi-level memory hierarchies may be used, offering complex tiers of memory performance on-node. The network infrastructure will be fast and may be connecting millions of cores across the full system. With this huge core count comes a reduction in the mean time between failures: even if the expected failure rate of a single processor core is low, when there are over one million cores in the system, the failure rate of the machine could become a significant issue [146].

One example of a machine similar in style to the predicted exascale architecture is Sequoia, an IBM Blue Gene/Q at Lawrence Livermore National Laboratory. However, it still lacks some of the more novel features such as low-powered accelerator-type processing cores and a complex memory hierarchy. Investigating these kinds of systems now is essential in preparing for the complexities future supercomputers will introduce.

In this chapter we discuss the problems faced when porting production codes to future architectures, with a particular focus on the exascale machines predicted to arrive in the next five years. We introduce mini-applications—small, self-contained programs that represent the performance characteristics of a production application—as a possible solution to this problem. The value of mini-applications is fully realised when they are carefully applied to consider specific questions; this requires guidance and some high-level organisation. We describe the Mantevo project, a collection of mini-applications from a wide-range of scientific domains, one provider of this required direction. Two of the mini-applications in the Mantevo suite were developed at the University of Warwick and the Atomic Weapons Establishment (AWE) as part of this thesis, and we use these as examples of how mini-applications can be used to solve the problems of moving production codes to future architectures.

4.1 Production Applications and Future Architectures

Production applications are full-featured software packages, used on a daily basis in research and industry to further scientific discovery. These applications are large, both in terms of the number of lines of source code they contain and the size of the scientific output they produce. The next few hardware generations of supercomputers predicted to arrive in the exascale era will look dramatically different to today's clusters, with much higher levels of on-node parallelism, slower (but more power-efficient) processing cores, complex multi-

level memory hierarchies, and fast network fabrics. Porting legacy applications to these architectures will not simply be a case of recompiling existing source code. Instead, extensive modification and optimisation of the application code will be required. This modification may require different programming libraries or languages (as evidenced by CUDA, OpenACC, or OpenMP) or it may involve modifications at an algorithmic level which fundamentally change how a given application produces its scientific output. This porting problem affects old applications most seriously, as newer programming models can provide more flexibility in allowing the same application code to be used on multiple architectures.

Regardless of the scale of the changes that will be required, any changes to a production application are difficult to implement. The primary programming difficulty is the number of lines of source code, but this kind of obstacle is often far smaller than the extensive validation and verification required when a production application is changed. Organisations rely on these applications; any changes must not adversely affect their scientific output.

Since high-performance computing applications are focused on scientific discovery, application developers will prioritise adding scientific features to an application, often at the expense of the maintainability and performance of the code. These applications have a large number of lines of source code and will often rely on other software libraries, creating complex dependencies between various applications. The size of production applications means that they take a long time to compile, and produce large executable files. We need an effective and appropriate path to move these codes towards exascale.

4.2 Mini-Applications

In Chapter 2 we saw the wide range of benchmarks that already exist and could potentially be used to move production codes towards exascale. The pitfall to using a benchmark for making difficult decisions regarding the development

Area	**Problem:** Production code	**Solution:** Mini-Application
External interactions	Commercially sensitive	Open-source
	>100,000 LOC	<10,000 LOC
New languages and programming	Library dependencies	Standalone code
models. Simulations. Compiler tuning.	Large executable	Small executable
Node and network scalability studies.	Long runtimes	Short runtimes
	Large input files	Self-contained test problems

Table 4.1: The solutions mini-applications provide to the problem of porting production codes.

of a production application is that benchmarks are too small to capture the key aspects of the performance of a particular application. The range of interacting computational behaviours that a production code contains means a single benchmark will not be representative of the application as a whole. Benchmarks are often less representative of the actual numerical algorithms found in an application, even if their performance characteristics are representative. By containing so few lines of code, there is little chance for a benchmark to contain anywhere near the complexity of an application that is dictated by the science it performs.

Consider the LINPACK benchmark [115]: whilst a stalwart of the High-Performance Computing (HPC) benchmark roster, it bears little resemblance to real applications. LINPACK, which solves a dense linear system of equations, runs at up to 90% of a supercomputer's peak theoretical Floating Point Operations per Second (FLOPs). In practice, a production application may achieve as little as 1% of the peak theoretical FLOPs available, and studies report between 5% and 10% of peak [88]. This is not necessarily a result of poor programming, as many codes must contain complex control code or lots of memory access; both of which reduce the number of floating point operations that are performed in a given time period.

Mini-applications provide the functionality of a benchmark with more resemblance to real applications. A mini-application is a small application that models a key performance aspect of a *parent* production application. Like a low-level benchmark, a mini-application is small and simple. Much of the

scientifically-relevant code found in a production application can be removed, and what is left are small computational kernels that are equivalent in loop structure, data access patterns, and communication requirements. Whilst the easiest observed metric from a mini-application is still the total wall-clock time, a mini-application can also provide a breakdown of runtime performance, enabling an engineer to attribute performance to different machine components.

Mini-applications are small, but only small enough to capture the key performance characteristics of an application. This means that they remain representative, but with far fewer lines of code. LAMMPS is a production application for simulating molecular dynamics that contains over 100,000 lines of code [137]. LAMMPS has a mini-application, miniMD, that contains less than 3,000 lines of code [73]. Being small means that investigating programming and portability issues on future architectures becomes viable. Most mini-applications can be rewritten in a few weeks. Performing the same study with a production application would take years, and would then require years of verification and validation.

To the extent that they mimic the unique mix of computational behaviours found in production codes, mini-applications are representative. Rather than being intensive benchmarks that only stress a small subset of hardware components, mini-applications aim to emulate the mix of memory access, computation, and communication found in a key production code. However, not all behaviours will be captured by the mini-application, instead, the most expensive or most frequently exercised portions of the parent application will be captured. Mini-applications bridge the gap between micro-benchmarks and production applications and make it easier to reason about the behaviour of a production code and attribute aspects of its performance to different hardware components.

Being small, mini-applications are perfect for investigating new programming languages and models. Karlin et al. used the LULESH mini-application to investigate four emerging programming models [87]. Whilst the original

LULESH application was written in C++, it was ported to all four programming models with each new version taking approximately eight weeks to complete [86]. Two of the programming models are C++-based frameworks, but one is a new parallel programming language and the final framework is developed in Scala (a Java-based programming language not often used in HPC). Being able to investigate a wide range of languages is a huge benefit when considering how future supercomputers will be programmed.

By their nature, production applications are often commercially sensitive, and often fall under other restrictions such as export-control, inflexible licences, or other regulations. Mini-apps provide an easy way for the developers of these applications to interact with researchers in academia. They can also be shared with the vendors of various components in the supercomputer toolchain. For example, in [77], the authors identify several potential code optimisations that were not performed by a range of compilers. The LCALS suite which they use to perform this analysis is available under an open-source licence, so compiler vendors can be provided with a reproducer that they can use to fix this problem. Mini-applications can also be shared with hardware vendors at an early stage, and it is then conceivable that a machine could be designed with a particular application in mind. This approach borrows much from the co-design ideas most often seen in the embedded systems community [69].

The representativeness of mini-applications offers a major advantage in actually improving production applications. By being both similar in code design and performance characteristics, any lessons learned by using the mini-application will be much easier to apply to the production application. This provides a much more pragmatic advantage for mini-apps, since most organisations are understandably concerned with the fiscal return from this research. Applying optimisations learned from mini-applications back to production applications can yield significant performance gains with little effort. Simple optimisations opportunities can be hidden by the complexity of the production

application's source code. These opportunities become easy to identify using the mini-application, and since it will share key code structure with the production application, it becomes easier to re-apply these optimisations where they will make a real difference.

Mini-applications are not the only approach for moving codes to exascale systems. One approach that has worked in the past is to wait until the new machine has arrived and then gradually begin the porting effort for existing codes. One of the first accelerated supercomputers was Los Alamos National Laboratory (LANL)'s Roadrunner, an IBM machine that used a combination of AMD Opteron Processors and IBM's Cell Broadband Engine accelerator chips. Porting applications to Roadrunner could be done in one of three ways: (i) using only the AMD processors and ignoring the Cell processors, (ii) offloading key application hotspots to the Cell for acceleration, or (iii) porting the entire application to the Cell processors and only using the AMD Opteron processors for communication, I/O and visualisation. The third approach was used to port the Sweep3D benchmark, achieving a performance improvement of up to $4\times$ [14, 135].

Whilst in the case of Roadrunner, waiting for the new machine to begin porting worked, it relies on the new architecture being somewhat similar to the older to be applied successfully. The fact that applications could continue to run unmodified on Roadrunner's AMD Opteron processors meant that there was no delay in using the machine, even if its full performance was not initially realised. With architectures becoming more complex, increasing clock speeds and extra nodes no longer provide the same performance improvements they once did. With the dramatic changes in both node and system architecture required to deliver exascale performance in an acceptable power budget, it is unlikely that a slow, evolutionary approach will be the optimal one.

Another way to approach the exascale problem is to write new codes. This may be the ideal solution, as each new application can be written to take best advantage of the new exascale system that it is likely to be running on.

Application	Domain
miniFE	Implicit unstructured finite element.
HPCCG	Sparse iterative linear solvers.
Epetra Kernels Benchmark	Matrix-vector kernels.
miniGhost	Explicit structured partial differential equations.
miniSMAC2D	Implicit structured partial differential equations.
miniMD	Molecular dynamics.
CoMD	Molecular dynamics.
MiniXyce	Circuit simulation.
CloverLeaf	Explicit structured hydrodynamics.
CleverLeaf	Explicit structured hydrodynamics with AMR.

Table 4.2: The 10 mini-apps included in the Mantevo Suite version 2.

However, the time it takes to develop, then validate and verify a new scientific application means this approach will require significant investment of both time and resources. If mistakes are made during the design and development stages they may be impossible to correct, and even in the best case are likely to be costly.

Mini-applications can act as a compromise between these two extremes. When a mini-application is developed to represent a key production application, then any incremental improvements made to the mini-application should be easier to transfer back into the parent application. Additionally, mini-applications can be treated as brand new codes, and used to explore new programming languages, models, and libraries, all while performing the same fundamental science as the parent application.

4.3 Mantevo: Mini-Applications in Practice

Sandia National Laboratories's Mantevo Suite [106] is a collection of mini-apps that target a wide range of scientific domains (see Table 4.2). An early report published in 2009 pioneered the concept of mini-apps [73], and since then they have been applied to study: performance at both the node- and network-level, programming languages and programming models, program simulation, and compiler tuning. This research has attracted recognition from the wider scientific community and won an R&D 100 award in July 2013 [2].

The domains covered by the Mantevo Suite include explicit and implicit solution of partial differential equations, molecular dynamics (MD), hydrodynamics, and circuit simulations. The use of multiple applications in each domain is important because different production codes may solve the same fundamental problem with different numerical methods. Since a mini-application must be representative of a production application to be useful, it is most effective to have one mini-application represent one production application.

Two of the applications from the award-winning Mantevo suite have been developed, either in whole or in part, at the University of Warwick and AWE as part of this PhD thesis. These applications: CloverLeaf and CleverLeaf; are described in detail in the next section. We also describe TeaLeaf, a third mini-application developed during the course of this thesis. In the remainder of this section we present a brief overview of the successful use of three other Mantevo mini-applications to further the progress towards production codes on exascale architectures.

One of the first apps in the Mantevo suite, miniMD has been extensively studied and applied in a wide range of experiments. Two papers by Pennycook et al. used miniMD to investigate performance improvements specific to molecular dynamics in two programming models. The first uses Intel's Advanced Vector Extensions to target the wide vector units of Intel CPUs, achieving a five-fold performance improvement over the original version of the application [128]. The second uses the OpenCL framework to develop performance portable kernels; the same code can be run on a range of devices including CPUs, GPUs and other accelerators [126]. Using a single source code they achieve a performance within 50% of an optimised native version specialised for the particular device.

Approximating an unstructured finite-element application, miniFE has been used to investigate the performance of this algorithm on accelerator devices including an Intel Xeon Phi and an AMD Trinity Accelerated Processing Unit (APU) (which combines an Central Processing Unit (CPU) and a Graphics

Processing Unit (GPU) on a single chip). In [16], the authors describe using miniFE to perform a detailed investigation into the matrix assembly phase of the algorithm on an NVIDIA GPU. This phase of the algorithm contains many floating point operations, but is actually bound by memory bandwidth. By altering the CUDA kernel the matrix assembly phase could be tuned to become bound by the number of floating point operations that can be performed. These optimisations were also applied to the CPU version of the code, showing how mini-applications can continually evolve to reflect the state of the art implementation of a particular algorithm.

Continuing the investigation of new programming models, the HPCCG mini-application is used to asses a hybrid programming model with CPU and GPU nodes [73]. The model uses conventional MPI to exchange data between distributed nodes, and then work on a node is divided into resource management and computation. Units of work are dispatched to stateless computational functions, a flexible approach that enables portability between future architectures.

4.4 Mini-Application Development and Deployment

CloverLeaf, TeaLeaf, and CleverLeaf are three mini-applications that have been developed in conjunction with this PhD thesis. These applications attempt to represent typical codes used to solve explicit hydrodynamics and heat diffusion, both of which are areas of interest to AWE. The applications have been used primarily to study issues surrounding performance and programmability of contemporary and future architectures, both at the node- and network-level.

CloverLeaf is a structured-grid hydrodynamics application that solves Euler's equations in two dimensions. These equations can be used to represent anything from the implosions critical for research into nuclear fusion, movement of gases in an engine, water in the ocean, meteor impacts, or the fiery

gases found at the heart of every star in the universe. CloverLeaf provides a lightweight proxy for identifying how we can run these kinds of simulations more efficiently on current and future machines.

TeaLeaf solves the heat conduction equation on an Eulerian grid. These equations are solved as a linear system, and TeaLeaf is designed to test the wide range of linear-solver libraries available today. The transfer of energy via conduction is essential in many engineering domains, but beyond any specific domain, the use of linear-solver libraries to solve linear systems representing partial differential equations can be found in many scientific simulations.

CleverLeaf is a structured-grid hydrodynamics mini-application that extends CloverLeaf with Adaptive Mesh Refinement (AMR) using the SAMRAI toolkit from Lawrence Livermore National Laboratory [98]. The primary goal of CleverLeaf is to evaluate the application of AMR to the Lagrangian-Eulerian hydrodynamics scheme used by CloverLeaf, and as a proxy to investigate the performance of AMR for structured Lagrangian-Eulerian hydrodynamics at scale.

4.4.1 CloverLeaf

CloverLeaf is written in Fortran 90, and contains approximately four thousand lines of code. The research carried out with CloverLeaf has focused on addressing two key components of future supercomputer architectures: on-node programming and performance, and network scalability.

Further detail on the hydrodynamics scheme employed by CloverLeaf is reserved for Chapter 3. Here we present a brief description of the design of CloverLeaf to provide context to the following discussion on scalability and portability. CloverLeaf consists of a small main loop that advances the simulation in time, calling the kernel-driver pairs to perform the necessary steps of the Eulerian hydrodynamics scheme. Visualisation and summary routines are also included as useful debugging tools when investigating new optimisations and programming languages.

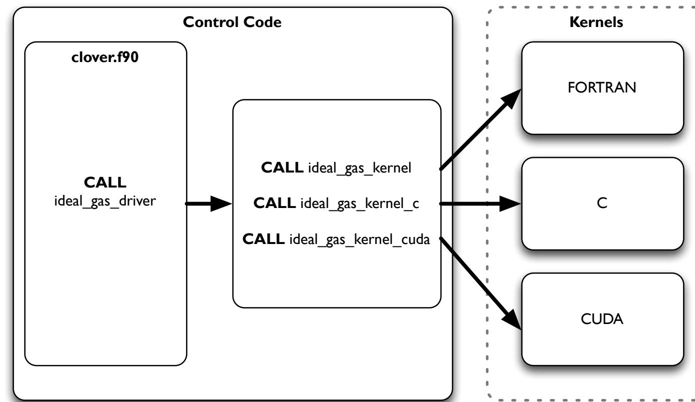


Figure 4.1: Kernel-driver design used in CloverLeaf. Different kernel implementations can be selected at runtime by the driver code.

The kernel-driver idiom (see Figure 4.1) is at the heart of CloverLeaf and all the mini-applications we have developed. The driver code is called by the main loop of the application and has access to mesh information and simulation data. This data is stored in derived types that allow for a meaningful grouping of data. To isolate the kernel from the rest of the application code, each routine is written in a functional style: all the necessary data is passed as arguments to the kernel routine, which is responsible for updating the necessary values. By ensuring that each kernel has a well defined input and output, we can modify the underlying implementation without worrying about impacting the rest of the simulation code. Kernels can be selected at runtime and different implementations used within a single execution of the application.

The simple kernel-driver design has allowed rapid implementation of over 10 different versions of CloverLeaf, including: Fortran 90, C, OpenMP, OpenACC, OpenCL and CUDA. The OpenCL version was developed during the period of thesis registration, and contributions have also been made to the Fortran and OpenMP versions of the application. When a new implementation is written each underlying kernel call will be modified. In the case of OpenCL and CUDA, these modifications may be extensive, containing additional data management and control code necessary for interacting with the attached ac-

celerator device. However, by isolating all this to a kernel function, the control routine need not worry about how each kernel operation is performed, as long as the correct memory locations contain the correct values after the kernel call.

Performance and Productivity

To study on-node programming and performance we use the CloverLeaf implementations that employ programming models targeting attached accelerator devices. This work, presented in [71], considers each implementation from two different angles: performance, and productivity. We first measure the runtime of a typical simulation using each programming model. We also consider the productivity of each programming model, by measuring the additional code required to port CloverLeaf to any given model. Whilst measuring the amount of code added does not capture all aspects of developing a new version of an application, it does highlight the dramatic difference in work for each of the three programming models we investigated.

The three programming models we investigated were: OpenACC, CUDA and OpenCL. The OpenACC programming model provides an ease of programmability through the use of directives and portability by allowing a single source code to be run on both CPUs and accelerators [120]. The disadvantage of OpenACC is that the programmer must surrender some control to the compiler, and may find themselves with less flexibility in terms of how they implement particular code regions.

The OpenCL programming model is more explicit, and uses library calls to launch device-specific code [155]. The flexibility of OpenCL comes from the powerful library of host functions that allow a programmer to determine, at runtime, the kinds of OpenCL devices available to the program. The kernels can then be compiled when the program is run, allowing them to be optimised for some particular piece of hardware. The disadvantages of OpenCL are the same as with any community-driven effort where members have conflicting concerns. Each member of the OpenCL group will want to ensure that the

standard enables their hardware or implementation to perform the best, but oftentimes this may be at the expense of useful functionality that could readily be provided by other vendors.

CUDA uses the same explicit design as OpenCL, with device-specific code being launched using library functions [119]. While less flexible than OpenCL, the advantage of CUDA is that equivalent programs normally perform much better on the same hardware, although this hardware can only be NVIDIA GPUs. The disadvantage of CUDA is the vendor lock-in that it enforces. Whilst CUDA compilers for x86 are available from other vendors, the fundamental target of CUDA remains NVIDIA GPUs [118]. Despite this, due to its status as the most well-established GPU programming model, CUDA is often the first choice when writing a GPU-based application.

Each of these three GPU-based implementations of CloverLeaf took only a few months to develop. The OpenACC implementation used directives to mark code for execution on the device and to handle data movement. We ensure that all data is allocated and stored in the GPU memory at all times, and is only transferred to the CPU memory when required for MPI data exchange or for simulation output.

The OpenCL model distinguishes between a *host* CPU and an attached accelerator *device* such as a GPU. The host CPU runs code written in C or C++ that makes function calls to the OpenCL library in order to control, communicate with, and initiate tasks on one or more attached devices, or on the CPU itself. The target device or CPU runs functions written in a subset of C, which can be compiled at runtime, or loaded from a cached binary if one exists for the target platform.

The OpenCL implementation extends CloverLeaf's kernel driver architecture with a management class written in C++. This class controls interaction between the Fortran code and the new OpenCL kernels. In particular, it allows the kernel objects and OpenCL buffers to be accessed from the kernel implementations as necessary. At runtime, the Fortran driver routine calls the

OpenCL kernels function and begins executing host-side code. This code sets up the corresponding OpenCL device kernel and schedules it for execution on the device. As with the OpenACC version of the application, all data is allocated and stored on the device. Data is only copied back to the CPU memory for MPI communications and simulation output.

The CUDA implementation of CloverLeaf is almost identical in design to the OpenCL implementation, and uses a global class to coordinate data transfer and computation on the GPU, with helper functions to handle interoperability between the CUDA and Fortran code.

These implementations enable in-depth analysis of the performance of each programming model for the target domain: explicit hydrodynamics. Running a small test problem, the overall wall-clock times for the OpenACC, OpenCL and CUDA versions were 2.057s, 2.558s, and 2.780s respectively. The OpenACC implementation was therefore 1.24× faster overall than the OpenCL version and 1.35× faster than CUDA version[1]. These results have been superseded by later work [72, 105], and hence the focus here should be on the performance of an initial GPU port, rather than a highly-tuned final implementation.

In terms of programmer productivity, OpenACC proved superior to both OpenCL and CUDA, requiring the addition of only 184 OpenACC pragmas as opposed to thousands of additional lines of code. CloverLeaf provided an essential platform to investigate these issues of both performance and programmability. Using these results, it becomes clear that in terms of development effort, the directive-based approach of OpenACC requires far less code modification. In a mini-application this isn't such a problem, but as we have discussed, in a production application, this would be significant.

Scalability

Mini-applications also prove useful for investigating application performance at scale. When run in parallel, the advantages relating to the size of mini-

[1]Full results of this study are presented in Appendix A.

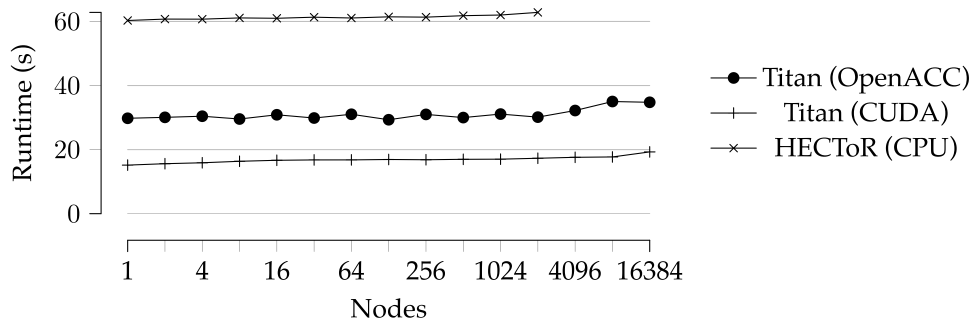


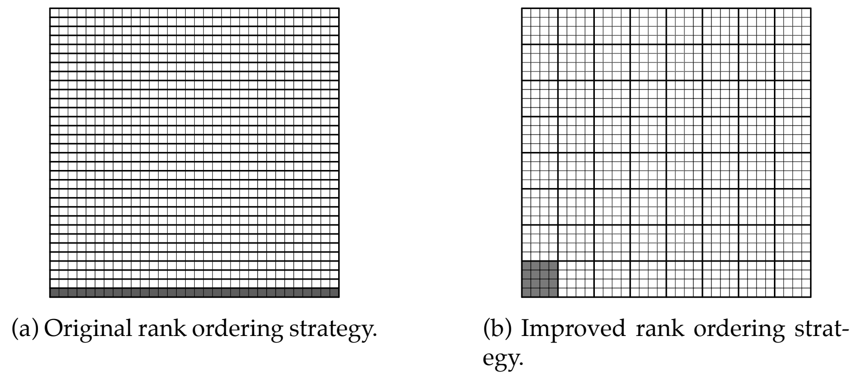
Figure 4.2: CloverLeaf weak-scaling on HECToR and Titan.

applications become emphasised. The short runtimes mean many experiments can be carried out, and the applications can be easily modified to try and improve scalability. Using CloverLeaf, we evaluated three possible programming model combinations for running on multiple nodes: MPI, MPI with OpenACC, and MPI with CUDA. Using the supercomputers Titan and HECToR, two large Cray systems at Oak Ridge National Laboratory and Edinburgh Parallel Computing Centre, respectively, CloverLeaf was run on over 50,000 CPU cores.

The two systems provide two unique configurations: HECToR is a typical supercomputer with a CPU-based architecture, while Titan is a hybrid system where each node contains one CPU and one GPU. The experiments allow us to investigate which of these two architectures is most appropriate for large-scale explicit hydrodynamics runs[2]. On both architectures, CloverLeaf is highly scalable with the overall runtime increasing by only 4.2% on HECToR and by up to 27.2% on Titan as the problem is run at the largest scale. The GPU-based architecture of Titan is considerably faster than that of HECToR. CloverLeaf is over twice as fast at all node counts when running on Titan.

In addition to evaluating programming models, we also evaluated application configuration. Typically, each process in a job is mapped onto consecutive processor cores within a node. CloverLeaf distributes work in two dimensions, mapping naturally onto the two-dimensional Eulerian mesh on

[2]Full results of these experiments are presented in Appendix A, and we present only the pertinent details here.



(a) Original rank ordering strategy.

(b) Improved rank ordering strategy.

Figure 4.3: MPI rank re-ordering strategy used to improve performance by reducing off-node communication. Thick lines are node boundaries; shaded cells highlight the processes on a single node.

which the problem is solved. However, regions of the domain are assigned consecutively to processes, and this mapping does not reflect the two-dimensional communication pattern where each rank communicates with its four nearest neighbours. When more than one node is used, half of all the communications will need to travel across the network. By redistributing the processes intelligently and matching the communication pattern in CloverLeaf, we can reduce these off-node communications.

This reconfiguration provides a performance improvement of 4.1% on over one thousand nodes, and highlights the range of experiments that can be performed with a mini-application. Optimisations such as this revised mapping configuration can be easily applied to production applications on current architectures, providing immediate performance improvements.

By evaluating both programming models and performance optimisations at this scale we are able to identify possible approaches for running production codes on future architectures. As with the programming model study, we have seen that all approaches are scalable (largely due to the nature of the explicit hydrodynamics scheme, described in detail in Chapter 3), but that performing the computation on the highly-parallel GPUs halves the runtime of the experiment. Whilst experiments to evaluate the scalability of the existing programming model could have been performed with a production application,

and the new mapping configuration could have been tested, there would be no easy way to evaluate any alternative programming models. Additionally, the complex physics solved in a production application greatly increases the run-time of even a simple test problem. With supercomputer time charged by the core-hour, long running experiments using thousands of cores use a significant amount of any budgeted time.

4.4.2 TeaLeaf

TeaLeaf solves the heat conduction equation on an Eulerian grid, and is designed to test the wide range of linear-solver libraries available today. It also provides a built-in iterative Jacobi scheme, that can be run on any architecture. This scheme has been ported to GPUs using OpenACC. In this section, we present the first details of the TeaLeaf mini-application, including a brief design overview and some plans for possible performance investigations.

To remove any library dependencies, one algorithm to solve the heat conduction equation is provided. This is a Jacobi scheme that solves the heat conduction equation using a second-order central-difference in space, and a forward-difference in time. The scheme is implemented using a matrix-free method. The simple loops required by this scheme can easily be ported to GPUs using OpenACC directives. Each loop was annotated such that OpenACC would generate the required device code in order to run the loops on GPU. This extension also integrates with the CloverLeaf OpenACC implementation. Using `data` directives, we can inform the compiler that the required arrays will already be in the device memory and avoid any expensive copy operations.

In addition to the Jacobi method, TeaLeaf can also use one of three different linear solver libraries: HYPRE, PETSc, or Trilinos. Each of these libraries provides a software components that can be used to solve a linear system, in parallel, on modern supercomputer architectures. In the case of HYPRE and PETSc, these components understand the notion of a structured grid, and hence

can easily be used by adapting the existing Jacobi version of the application. The Trilinos library provides parallel vector and matrix objects, and a large collection of solver classes that operate on these vectors and matrices. This means that we must construct the correct vector representation for our system, and then retrieve values after the system has been solved. Each library is implemented in a separate software module and can be selected at runtime.

Whilst library dependencies are one of the problems with production codes, the TeaLeaf mini-application provides a flexible framework for testing linear-solver libraries. The system of equations results in a structured tri-diagonal matrix that is easy to construct and pass to any library for solution. The heat-conduction kernels are isolated from the rest of the source code; while this separation would be unlikely to exist in a production application, it is essential in allowing new solver libraries to be added and tested rapidly.

4.4.3 CleverLeaf

Whilst the hydrodynamics scheme implemented by CleverLeaf is identical to that of CloverLeaf (and we have two mini-applications that can solve the same problem in a similar fashion), the added complexity of AMR is adequate justification for a new mini-application. AMR adds an additional level of algorithmic overhead to the application since not only must the simulation be advanced, but the patch hierarchy must be managed, maintained and updated as well.

This management overhead introduces a complex interplay between the proportion of runtime spent performing meaningful scientific work, and that spent maintaining the hierarchy and related metadata necessary for an AMR simulation. Typically, minimising time spent doing computational work will reduce the overall time to solution but increase the management overhead. The opposite is true if more computation is performed. The key to the effective use of AMR is finding the sweet spot where both simulation and management time are minimised. These issues can be investigated using production AMR codes, but to be able to fully explore the parameter space that controls AMR perfor-

mance, a small mini-application is both easier to understand and to modify.

The complexity of AMR is often abstracted into a library. These libraries offer the same basic features and abstractions, so choosing the best library for a given application is difficult. A small application like CleverLeaf provides the perfect vehicle for exploring the range of AMR libraries available, since the small application footprint can be ported to each library model in much the same way as the CloverLeaf mini-application was ported to various programming models.

The CleverLeaf mini-application provides the context for the rest of this thesis, and is discussed and applied in the remainder of this text. In Chapter 5 we describe in detail the design and implementation of CleverLeaf, providing insight into the mini-application development process. In Chapter 6, we use CleverLeaf to investigate how block structured AMR applications might be written for future architectures, and in Chapter 7 we show how, using CleverLeaf, we are able to identify key performance-impacting AMR parameters that can improve application performance.

4.5 Summary

Whilst mini-applications are by no means the only approach for moving production codes to future architectures, they are one of the most attractive. By providing a lightweight representation of a production application, they offer developers a chance to analyse the performance of the application on current systems, and make projections about how the application might perform on future systems. Additionally, the nature of mini-applications makes them easy to port to new programming languages or programming models in a matter of weeks. This is essential for evaluating the possible approaches for programming complex exascale machines.

In this chapter we presented a description of the problems production applications face when moving to future supercomputer architectures. Ensur-

ing that mini-applications meet some high-level objective is essential in ensuring their unique strengths are used. The Mantevo project provides some of this organisation for a collection of 10 mini-apps. These applications have already been used to improve the performance of production codes on existing architectures, and through instruction-level simulation are being used to evaluate the performance of mini-applications on models of proposed future architectures.

We present the three mini-applications, CloverLeaf, TeaLeaf, and CleverLeaf; developed as part of this thesis, and show how CloverLeaf provides examples of mini-applications solving the problems that production applications face. Using CloverLeaf to investigate programming models that target accelerators, we show that OpenACC is both high-performance and low-effort, providing a performance improvement of $1.24\times$ over the next fastest GPU-based code, with the addition of only 184 pragmas. Additionally, this code comes in the form of compiler directives, meaning that the source modifications are invisible when compiled in an environment where accelerators aren't present. We also consider the problems production applications will face as they move towards the scales required by exascale-class architectures. Running on up to 16,384 nodes of Titan (over 100 thousand MPI ranks) we show that the explicit hydrodynamics scheme represented by CloverLeaf is scalable on both CPUs and GPUs. However, with a simple modification of the machine configuration we are able to improve performance by an additional 4%. This kind of optimisation can be applied independently of the application being run, meaning it can be used to benefit production applications immediately.

CHAPTER 5

CleverLeaf: An Adaptive Mesh Refinement Mini-Application

Block-structured Adaptive Mesh Refinement (AMR) is a technique used to increase simulation accuracy and reduce resource usage by dynamically increasing mesh resolution in the areas in which it will be most effective. Developed by Berger et al. [21, 23], AMR has been successfully applied to domains including cosmology, astrophysics, and shock hydrodynamics [28, 53, 139]. These areas all contain disparate physical scales that make using a fine mesh resolution throughout the problem domain unattractive.

In this chapter we introduce CleverLeaf, a mini-application that solves Euler's equations using a second-order accurate Lagrangian-Eulerian method on an adaptive mesh. Motivated by the desire to investigate factors that will affect the performance of AMR codes as the high-performance computing discipline approaches exascale, CleverLeaf uses the uniform hydrodynamics scheme found in CloverLeaf, and extends it using the SAMRAI library to support adaptive meshes. We describe the algorithms used by CleverLeaf, and document

how they are implemented in a small, easy to modify mini-application. Using three test problems, we verify the accuracy of CleverLeaf. Finally, we use the Veritas analysis framework to show that CleverLeaf can be considered representative of production AMR applications.

5.1 Related Work

CleverLeaf is the first block-structured AMR mini-application using a Lagrangian-Eulerian hydrodynamics scheme. Whilst numerous production codes use Berger's AMR approach and a Lagrangian-Eulerian hydrodynamics scheme to solve Euler's equations and simulate a vast range of physical phenomena [42, 75, 151], we are not aware of any other mini-applications using this approach. In this section, we present a brief summary of related code development efforts, and highlight the novelty of our work.

Whilst no other block-structured AMR mini-applications yet exist, we can break related work into two main areas. The first area is the collection of production-quality applications that use block-structured AMR to solve shock hydrodynamics problems. The second area encompasses other mini-applications, either those using some other form of AMR, or those that solve the same hydrodynamics equations as CleverLeaf.

Due to the opportunity for large savings in both application runtime and memory usage, AMR is an essential technique in many scientific domains. The kind of shock hydrodynamics that CleverLeaf simulates often occurs within the field of astrophysics, additionally, the dramatic changes in length scales that are required to simulate regions of the universe mean that this is the perfect domain in which to employ AMR. As such, many astrophysics codes solve similar problems to CleverLeaf.

FLASH is a large community-driven code from the University of Chicago that uses the Chombo and PARAMESH packages to provide AMR, and contains a range of physics packages including a hydrodynamics capability [53].

Enzo is another block-structured AMR code for modelling astrophysical fluid flows [29]. The AMR implementation in Enzo is built-in to the code, and not provided by an external library. RAMSES, from Commissariat á l'Énergie Atomique (CEA) in France is another astrophysics code that uses its own AMR implementation [154], relying on a tree data structure to represent the patch hierarchy, where each patch is a fixed-size group of cells, much like the tile-based approach seen in PARAMESH. The NIRVANA code again uses a custom AMR implementation with fixed-size blocks of four cells in each coordinate direction [167]. The final astrophysics code we will consider is GAMER [143]. Developed at National Taiwan University, GAMER uses a tile-based AMR approach and also supports the offloading of computation to a Graphics Processing Unit (GPU). All these codes solve Euler's equations, augmented with the necessary equations representing the magnetic fields found in the universe; collectively, these equations are known as the Magnetohydrodynamics (MHD) equations.

Applications developed to study high-energy and density physics also contain similar algorithms to CleverLeaf. These codes include the ALE-AMR project from Lawrence Livermore National Laboratory (LLNL), and the Shamrock benchmark from Atomic Weapons Establishment (AWE). The ALE-AMR code uses a Lagrangian hydrodynamic scheme and an adaptive mesh. Like CleverLeaf, it uses the SAMRAI library [9, 90]. Shamrock is an Eulerian hydrodynamics application with AMR, similar to CleverLeaf. Developed at AWE, the Shamrock benchmark has been previously used to investigate AMR application performance on a range of CPU-based architectures [70].

The CLAMR application is the first code from the second area of related work we will consider. Developed at Los Alamos National Laboratory (LANL), CLAMR is a cell-based AMR application that solves the shallow-water equations. The key difference between CLAMR and CleverLeaf is the AMR formulation that is used. In CLAMR, the structure of the mesh is stored as an oct-tree, with each cell splitting into four children when it is refined. The cells become

nodes in the tree, with edges representing the parent-child relationships [116].

MiniAMR is a 3D stencil calculation mini-application with AMR developed at Sandia National Laboratories (SNL) and released as part of the Mantevo project [106]. The mesh refinement is driven by simple objects moving through the mesh instead of a physics calculation, but can be used to mimic the behaviour of real AMR codes. The mesh refinement phase is capable of several levels of refinement where blocks are refined or coarsened as needed and the blocks are load balanced between MPI ranks.

Other hydrodynamics mini-applications exist, such as PENNANT from LANL and LULESH from LLNL, although neither application supports AMR [1, 50]. Both these applications are fully Lagrangian approach to solve Euler's equations so lack an advection step. Whilst these applications exhibit similar ideas to CleverLeaf and the other mini-applications mentioned in this section, without AMR they fail to capture the specific hydrodynamics algorithm, data layout, and code-specific details of the production codes we aim to represent.

To the best of our knowledge, CLAMR and miniAMR are the only two other AMR mini-applications that exist, and neither contains the same hydrodynamics algorithms and range of features of CleverLeaf. Additionally, neither code is designed to allow the use of a production-quality AMR library package, an important part of CleverLeaf when considering how production codes may be improved and moved to future High-Performance Computing (HPC) architectures.

5.2 Design and Development

In this section, we describe how algorithms to solve Euler's equations on an adaptive mesh are implemented in our mini-application. CleverLeaf uses a number of SAMRAI objects along with 23 custom classes that provide the necessary organisation to advance the solution with the Lagrangian-Eulerian scheme (see Chapter 3). To advance the local simulation state on each patch CleverLeaf

uses Fortran kernels taken directly from the CloverLeaf mini-application.

The CleverLeaf code architecture is typical of other mini-applications. The fresh start provided when writing a new application means that current software best practices can be used, resulting in a simulation code that is modular, maintainable, and easier to modify. We take cues from the approach put forth in the SAMRAI library and use a range of object-oriented design patterns. These software development approaches are almost universally used in enterprise software (when it is written in an object-oriented language), and by applying these design patterns in a scientific code we are able to leverage the benefits these well documented patterns provide.

Design patterns originated in the field of architecture, providing a set of flexible solutions to common architectural problems [3]. In computer science, object-oriented design patterns provide solutions to common software problems. Rather than recommending construction techniques, an object-oriented design pattern provides instructions on how software objects can be created and composed to address the problem with the most flexibility and maintainability.

The main pattern used by CleverLeaf is the *Strategy* design pattern [161]. This pattern encapsulates functionality behind a common interface, allowing a different implementation of the interface to be selected at runtime. This reduces coupling between simulation components, and in Chapter 6 we will see how this pattern allows CleverLeaf to be extended to run on thousands of GPUs with minimal impact to the code. Figure 5.1 shows how the SAMRAI and CleverLeaf objects are combined.

5.2.1 SAMRAI

The SAMRAI library is a toolbox of classes for the development of block-structured AMR applications. Developed at the Centre for Applied Scientific Computing at Lawrence Livermore National Laboratory, SAMRAI has been used successfully in modelling application domains including debris generation at laser facilities [90], cardiac simulation [62], and helicopter wake dynamics [145]. Con-

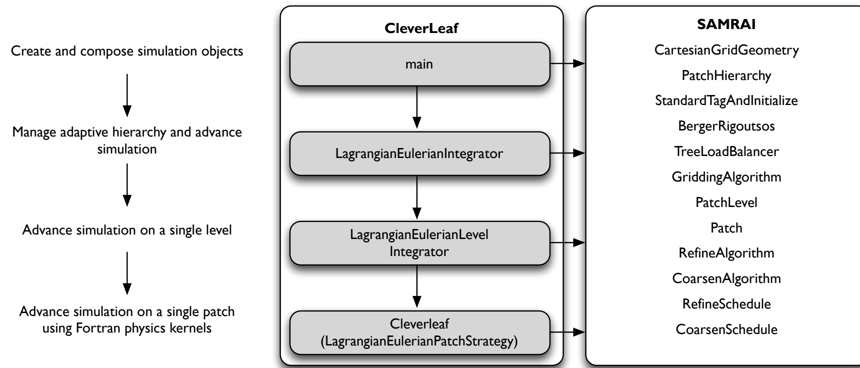


Figure 5.1: The object-oriented design used in CleverLeaf.

cepts fundamental to AMR are encapsulated in a collection of C++ objects, grouped by functionality into a range of packages. In this section we describe the three most important packages used by CleverLeaf. The design of the SAMRAI library is documented in detail elsewhere [10, 78, 79].

Patch Hierachy

The most important concept that an AMR application must represent is that of a patch. The patch is the fundamental unit of work in an AMR simulation, and having a flexible and extensible representation of a patch makes it much easier to write simulation code. Auxiliary structures that support the patches are levels and a hierarchy. In SAMRAI, these concepts are grouped in the `hier` package.

The `Patch` class is a container for data defined over a box representing some region of the problem domain. This class is responsible for allocating and deallocating data as the simulation evolves. A `PatchLevel` holds an array of these objects and is used to represent a single level of the hierarchy. Patches will be distributed between processors, so when iterating over the patches in a patch level, only the local patches will be processed. The final class, the `PatchHierarchy`, maintains an array of patch levels and is responsible for creating and removing levels as required while the application runs.

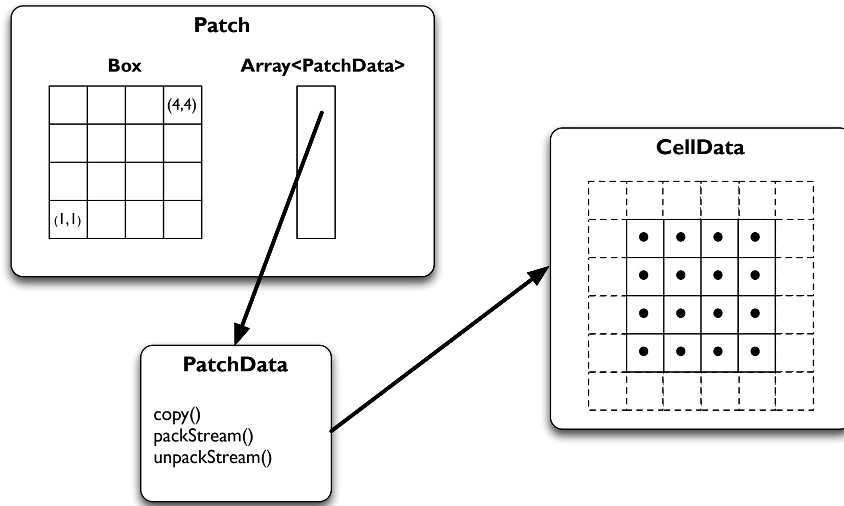


Figure 5.2: Relationship between the `Patch`, `PatchData` and `CellData` classes.

Data on the patch hierarchy is described by `PatchData` classes in the `pdat` package. This class provides the interface that SAMRAI uses to interact with user-defined data during AMR operations. Providing an interface (rather than forcing the user to use pre-defined data types) means that SAMRAI can be used to augment existing applications. This interface has been used to develop applications that use their own data structures in conjunction with SAMRAI's mesh management and communication routines [9]. In Chapter 6 we show how this interface can be used to develop a GPU-based AMR library.

A `PatchData` object represents data on a region of the problem domain defined by the associated patch. It also implements an interface necessary for all communication routines. To decouple the data representation for the communication methods described below, two routines must be provided: `packStream` and `unpackStream`. Both methods are passed a contiguous buffer of memory and a region of space described by a box. The methods must then either pack or unpack the data between the buffer and the region of the `PatchData` object corresponding to the provided box. As the simulation evolves, these objects will be created and destroyed. The `Patch` object stores an array of pointers to the necessary `PatchData` classes.

As an example, we describe the `CellData` class detailed in Figure 5.2. This class provides a representation for cell-centred data on a patch. The `Box` that the data is allocated over is extended to include the ghost region. The data (of arbitrary dimension) is stored as a one-dimensional array in memory, with the leading dimension of the data being stored contiguously. This column-major data layout matches that of Fortran, so when Fortran numerical kernels are used the data can be treated as a multi-dimensional array without any data re-ordering. The class also implements the `PatchData` methods, providing routines that will pack and unpack a region of cell-centred data.

Due to the transient nature of patches, and the data that exists on them, SAMRAI uses the `Variable` interface to represent a named quantity, such as pressure, that can be allocated on the adaptive mesh. While a `PatchData` object only persists for as long as the patch exists, the corresponding variable is typically static and exists throughout the simulation. A variable provides a way to interact with the same field anywhere on the adaptive mesh hierarchy, regardless of the current patch configuration.

Communication

Communication of boundary values between patches is handled by objects from the `xfer` package. When a patch touches a physical boundary, the problem boundary conditions can be applied, regardless of the level on which the patch resides. When a patch has internal boundaries, they must be filled with data transferred from another patch. Transferring data between processors to correctly fill the patch boundaries requires knowledge of how each patch is spatially related to others in the problem. SAMRAI does not maintain this metadata globally, and uses two concepts to improve communication performance: algorithms and schedules [164]. These concepts split each communication into a persistent *what* phase, and a transient *where* phase.

A communication algorithm encapsulates the *what* of a boundary update. Variables are registered with communication algorithms depending on

when they need to be exchanged. The algorithms typically persist throughout an application run, and specify exactly which simulation variable data needs to exchanged in any given communication.

A communication schedule encapsulates the *where* of the boundary update. Since where the data must be sent will change as patches in the hierarchy move, communication schedules must be recreated each time the hierarchy changes. The metadata containing patch locations is distributed, so this step can be more costly than the alternative algorithms used when global metadata is stored by each process. However, by ensuring that this step is only performed when required, the performance impact is minimised.

When transferring data between levels, the source data must be interpolated to correctly fill a different number of cells on the target patch. Data being transferred from a coarse to a fine level will be *refined*, and data transferred from a fine to a coarse level will be *coarsened*. In SAMRAI, this interpolation is described by two interfaces: `CoarsenOperator` and `RefineOperator`; that provide the necessary methods for coarsening or refining data. SAMRAI provides a number of common coarsen and refine operators, and custom operators can be created by implementing the appropriate operator interface. Each variable and communication algorithm can select the correct operators to use, allowing different fields to be communicated and interpolated using a different formula.

Algorithms

SAMRAI does provide a number of algorithm classes that manage the solution of a set of partial differential equations on an adaptive mesh. These include the `TimeRefinementIntegrator` class, that manages hierarchy construction, advancement and synchronisation as described in Berger's algorithm. This class uses the *strategy* design pattern [161], and expects another class to provide the necessary routines for advancing a single level of the patch hierarchy. The `HyperbolicLevelIntegrator` class provides the routines needed to integrate a system of hyperbolic conservation laws using Berger's algorithm. None of

the provided algorithm classes are suitable for the solution of Euler's equations using the Lagrangian-Eulerian method found in CloverLeaf, hence, we wrote our own classes to coordinate the SAMRAI objects and advance the simulation.

5.2.2 CleverLeaf

The hydrodynamics scheme implemented in CleverLeaf can be broken into two parts, the Lagrangian step and the advective remap. The equations are solved on a staggered grid. Each vertex stores a velocity vector, and each cell stores a pressure, energy, and density value as a scalar. Intermediate flux variables are stored on cell edges. The two parts of the algorithm proceed as follows: first, a Lagrangian step is used to advance the simulation variables in time. This distorts the Eulerian grid, since in the Lagrangian frame of reference, the vertices move with the fluid flow. Second, an advective step is used to remap the vertices to their original positions, restoring the original grid. In order to do this, material is moved through the cell edges based on the direction of the flow.

Each part can be further separated into a number of simple kernels that implement the required numerical methods to correctly advance the simulation. Developing, maintaining, and testing these kernels requires expertise from both domain scientists and computer scientists. Integrating the functionality of each kernel into a monolithic simulation code that will be maintainable and extensible is difficult. The alternative, using a flexible code architecture, can alleviate some of these difficulties, and is particularly effective when developing a mini-application.

Lagrangian-Eulerian Scheme

Solving Euler's equations using a Lagrangian-Eulerian scheme on an adaptive mesh requires far more management in order to ensure the accuracy of the results and the correct progression of the simulation.

As with the uniform simulation, each variable is first set using its initial

conditions on the coarse grid. The cells in level 0 are then evaluated and tagged in order to create the first refined grid. After the new refined grid has been created, the initial conditions are applied to the newly created patches. Applying the initial conditions to each level ensures that the problem is accurately initialised. For example, complex shapes can be more accurately represented at higher levels of refinement and if the initial conditions were interpolated from the coarse level, much of this detail would be lost. New levels are created and initialised until the maximum number of levels is reached.

Whilst Berger uses an error estimation based on local truncation error to flag cells [23], we take the approach of Quirk, and instead use a heuristic function to determine which cells to flag [139]. Cells are flagged using three different criteria, which capture the strong shock phenomena of interest. These are: (i) the density gradient ($\nabla\rho$), (ii) the energy gradient (∇E), and (iii) the artificial viscosity (q). The second-derivative of the density and energy gradients is evaluated for each cell, and if the gradient is above the user-specified threshold the cell is flagged for refinement. Testing the curvature of the density and energy fields will also capture material discontinuities, convergent flow, and divergent flow. The artificial viscosity is a term added to the equations to account for the discontinuity in the solution at a shock. A shock is a discontinuity that would introduce errors into the numerical solution, and when a shock is detected the artificial viscosity (q) is used to smear the shock over a few cells and remove the discontinuity. By checking the absolute value of the viscosity, shocks can be identified and the cell flagged for refinement.

Once the adaptive hierarchy has been initialised, the simulation can be advanced through time. As with the uniform simulation, the first step in the simulation is calculating the stable timestep. Since the cells on each level are of a different size, we calculate a minimum timestep for each level, before calculating the global minimum timestep. Whilst Berger's original AMR algorithm allows for the possibility of advancing finer levels through multiple smaller timesteps before advancing the coarse level through a single larger step, we

do not use such a technique here. Temporal refinement adds additional complexity when interpolating coarse values to finer levels that is unnecessary in a mini-application. Additionally, the time spent in computing the coarse levels of the hierarchy is dwarfed by the time spent computing the finest level, hence taking fewer coarse timesteps has minimal impact on simulation runtime.

Once the stable timestep has been calculated, we advance each kernel on all levels of the simulation, ensuring that the data in every level is advanced to the same simulated time. As in the uniform case, the kernel calls are interleaved with boundary cell updates.

The standard AMR algorithm proposed by Berger et al. uses a single boundary update phase [21, 23]. The Lagrangian-Eulerian scheme used by CleverLeaf requires multiple boundary updates. Frequent, small halo exchanges occur throughout the timestep; we advance each kernel and update each boundary for each level in turn to ensure simulation accuracy. Halo data for each patch can be filled in one of three ways: (i) with the physical boundary conditions, (ii) with the data from a neighbouring patch on the same level, or (iii) with the data from a neighbouring patch on the next coarsest level. When data is transferred between levels it must be interpolated, and all quantities are interpolated using conservative operators, based on the gradient of the variable being transferred.

Once the simulation has been advanced through a timestep, the fine solution is synchronised onto the coarser levels of the hierarchy. Only the field variables: velocity, density, and energy; need to be synchronised. Velocity values are copied (injected) directly onto the node beneath. If a node exists on a fine level with no corresponding coarse node, then the value is ignored. The density values are interpolated using a volume-weighted coarsen, and the energy values are interpolated using a mass-weighted coarsen. Using these two operators ensure that the simulation is conservative.

At the end of each timestep the regridding procedure can be applied. This need not happen every step, and the frequency is controlled by the user.

The regridding step occurs in a similar fashion to the initialisation of the adaptive hierarchy, but is applied from the second finest to the coarsest level. Starting with the second finest level, the cells on the level are tagged, and the tagged cells are then divided into a set of patches forming the new finer level. Data is copied into this new level from the old solution, or interpolated from the coarse level if no fine solution previously existed. During regridding, a buffer is added around tagged cells. One crucial requirement when applying AMR to problems with strong shocks is that a strong shockwave is not allowed to travel over the boundary between two levels [21]. By adding a buffer equal to the number of steps until the next regridding operation, we can ensure that a shockwave will not cross a level boundary. After the regridding operation the simulation steps are repeated until the desired end time is reached.

Application Design

In designing CleverLeaf, we have taken cues from the object-oriented architecture of SAMRAI and the kernel-driver paradigm of CloverLeaf to develop a flexible framework for solving Euler's equations on an adaptive mesh using a Lagrangian-Eulerian method. In Chapter 4 we have already identified features essential for mini-applications that this design must provide. We avoid commercially licensed software and commercially sensitive code components to ensure that CleverLeaf can be released under an open-source licence. The total number of lines of code should be low, ideally less that 10 thousand. Building on a well-designed software architecture can help by eliminating repetition in the code base and neatly encapsulating common features.

Ideally, a mini-application would be standalone, without any library dependencies. However, the complex nature of managing an AMR simulation means that rather than go through the error-prone process of reimplementing an AMR framework, we rely on the SAMRAI library. The library is mature, well-supported, and used in a number of production applications. Hence, use of the SAMRAI library by CleverLeaf is a way to ensure rapid development

and minimise coding errors.

The final two key aspects of a mini-application are less related to code development, and instead focus more on runtime. To allow us to perform the necessary experiments with new programming languages, programming models, and architectures, it's desirable to have self-contained test problems with short runtimes. We implement a simple problem generator that allows us to initialise arbitrary regions of gases at the start of the simulation. Whilst this generator lacks the features required in a production application, it nevertheless provides the necessary flexibility for setting up the range of test and benchmark problems used throughout this thesis.

A flexible code architecture can harm application performance. Whilst the fastest possible implementation of an algorithm might be implemented in a single large file, this monolithic code would be difficult to understand and modify. Conversely, over-engineering software to the point of hiding even the most simple task behind too many layers of abstraction can add significant runtime overhead. A mini-application is designed to investigate application performance (which suggests an optimal implementation would be best), yet it will also be used for investigating new programming models and languages. Performing these investigations will require code modifications, hence a flexible code base is desirable. Production applications, as well as being large, often lean towards more monolithic designs; this is one of the reasons they are unsuitable for investigating new programming languages and models.

Berger's algorithm describes the integrator used to advance the simulation on a patch as a black box. The physics performed by the integrator is irrelevant, as long as it advances the simulation to the correct time. We follow this pattern and decouple the hydrodynamics kernels from the rest of the AMR algorithm entirely. Not only does this make it easy to swap in new kernels written in a different programming language, it also means we can add entirely new physics capability without changing the majority of the control code in CleverLeaf. Conversely, isolating the physics code means that the AMR library could

be replaced without reimplementing the hydrodynamics kernels. However, the tight integration between the application control code and the SAMRAI library means replacing the SAMRAI objects with a different library would be more difficult than replacing the hydrodynamics kernels.

The patch hierarchy provides a convenient set of objects that separate simulation concerns. At the highest level, a single object is responsible for advancing the entire hierarchy. This involves performing each step of the AMR control algorithm, but delegating the concrete implementation of the integration steps to other objects. This integrator object will initialise the hierarchy, advance the simulation, and perform regridding and co-ordinate the collection of runtime statistics.

The level integrator object is used by the integrator to advance the simulation on a single level of the patch hierarchy. The required hydrodynamics driver routines are called for each patch in the level, with each hydrodynamics step being interleaved with the necessary boundary updates. The driver for each hydrodynamics kernel is provided by another object. This is where we isolate the numerical integration as described in Berger's original formulation. Each driver routine uses the appropriate Fortran kernel (taken from CloverLeaf) to advance the simulation.

This design provides three levels of re-use: the integrator, level integrator, and patch integrator. Other codes using a Lagrangian-Eulerian algorithm will be able to use the integrator class, and any algorithm using the same halo exchange points as CleverLeaf will be able to use the level integrator without modification. Finally, the kernel-driver model used by the patch integrator means we can re-use hydrodynamics kernels from other codes. This will be a key factor when using the CleverLeaf mini-application to improve production codes, since it provides a point for the kernels of the parent code to be inserted and investigated in the context of a SAMRAI-based AMR application.

Solving Euler's equations using the Lagrangian-Eulerian scheme on an adaptive grid requires some modified components that are not provided by the

SAMRAI library. These three components are the volume-weighted coarsening operator, the mass-weighted coarsen operator, and the tagging routines. SAMRAI provides a number of common coarsen and refine operators for interpolating data between different mesh resolutions, and for most variables, CleverLeaf uses the built-in operators to conservatively coarsen and refine data. The custom operators we have developed are required for coarsening energy and density.

The `CoarsenOperator` interface defines the methods that a C++ class must implement in order to be used for coarsening an arbitrary variable. The most important method, `coarsen`, is passed a coarse and fine patch, an area to be filled, and the ratio between the two patches. The method must then fill the coarse patch with a coarsened version of the data on the fine patch.

Two classes are created that implement the `CoarsenOperator` interface. One to perform the volume-weighted coarsen, and one to perform the mass-weighted coarsen. In each class, the `coarsen` method fills the coarse patch using the correct formula. These classes are also used to establish an ordering between the two operators. The mass-weighted coarsen requires the coarse cell density data so the volume-weighted coarsen must be applied first. This is ensured by assigning a priority to each class.

The three tagging criteria described in the previous section are implemented in the `Cleverleaf` class. The method is passed a patch and must fill the array of tags to indicate whether or not the cell has been tagged for refinement. As with the hydrodynamics kernels, the SAMRAI `Patch` data is retrieved and passed to a separate Fortran kernel for each tagging routine. Each kernel modifies the same array of tags, and the final tag array is used by SAMRAI during regridding to determine where new patches are created.

The parameters used for AMR components can have a large affect on application runtime. CleverLeaf readily exposes a range of AMR parameters for conducting experiments into their impact on performance. We describe these parameters in detail in Chapter 7.

CleverLeaf contains 5,777 lines of code (LOC), including 1,837 LOC of Fortran kernels that are shared with CloverLeaf. CleverLeaf also relies on SAMRAI, which is approximately 200 thousand LOC. The CloverLeaf mini-application (which solves Euler's equations without the adaptive grid) has 4,405 LOC. When considering lines of code solely attributed to the application, CleverLeaf and CloverLeaf are a similar size. However, CleverLeaf does rely on thousands of lines of library code. We can classify CleverLeaf as both a mini-application, since it contains a representation of a key algorithm, and also a mini-driver, since it drives a performance impacting library.

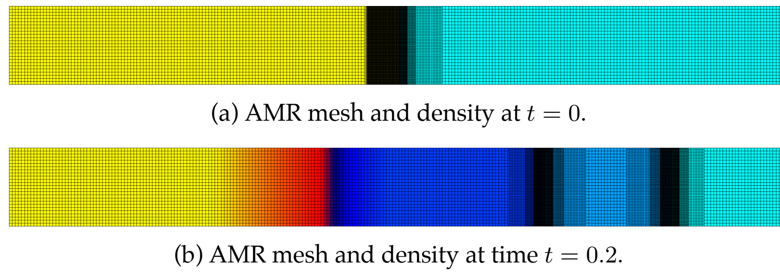
5.3 Validation and Verification

To validate CleverLeaf, we ensure that the hydrodynamics scheme has been implemented correctly by comparing our numerical solution to analytic or mesh-converged solutions for three test problems. To verify CleverLeaf, we ensure that its computational behaviour matches that of a benchmark application we wish to approximate. Whilst the primary goal of CleverLeaf is to explore new libraries and programming languages for explicit hydrodynamics with AMR, it is important that our mini-application is representative of some larger application. This provides the necessary context for integrating the results of our investigations into future production codes.

5.3.1 Validation

The hydrodynamics scheme and associated AMR interpolation operators in CleverLeaf have been validated using the three test problems described in Chapter 3: Sod's shock tube problem, the Woodward-Colella interacting blastwaves problem, and the Sedov blastwave problem.

For all problems we use four levels of refinement, and a ratio of two in the dimension of interest. As in [23], the regridding procedure is applied every four timesteps in all problems. These parameters allow us to test both the



(a) AMR mesh and density at $t = 0$.

(b) AMR mesh and density at time $t = 0.2$.

Figure 5.3: AMR meshes and density at the initial and final times of the Sod problem.

numerical accuracy of the simulation as well as the suitability of the heuristic refinement criteria contained in CleverLeaf.

Sod's Shock Tube

CleverLeaf was run with 250 zones in the x dimension at the coarsest level, corresponding to a Δx of 0.004, with a fixed timestep $\Delta t = 0.004$. The area of interest in the Sod problem, the shock wave, only travels in the x-dimension hence we use a refinement ratio of {2,1} (the resolution remains constant in the y-dimension). The Δx at the finest level is thus 0.0000625. Figure 5.3 shows the density of the domain at times $t = 0$ and $t = 0.2$, and Figure 5.4 shows the profiles of density, velocity and internal energy. The solution contains a small error at the contact discontinuity (the discontinuity between physical domains) and in the rarefaction (the expanding material behind the shock), however across the rest of the domain the solution is almost exact, and no oscillations are present. These small errors at the discontinuity are to be expected, due to the numerical noise associated with modelling a discontinuity using a second-order method [59, 60]. The loss of energy visible in the third part of Figure 5.4 is due to the lack of conservation of kinetic energy during the advection step of CleverLeaf's hydrodynamic algorithm. The results of this lack of conservation are most visible when running the Sedov test problem, and hence are discussed later in this section.

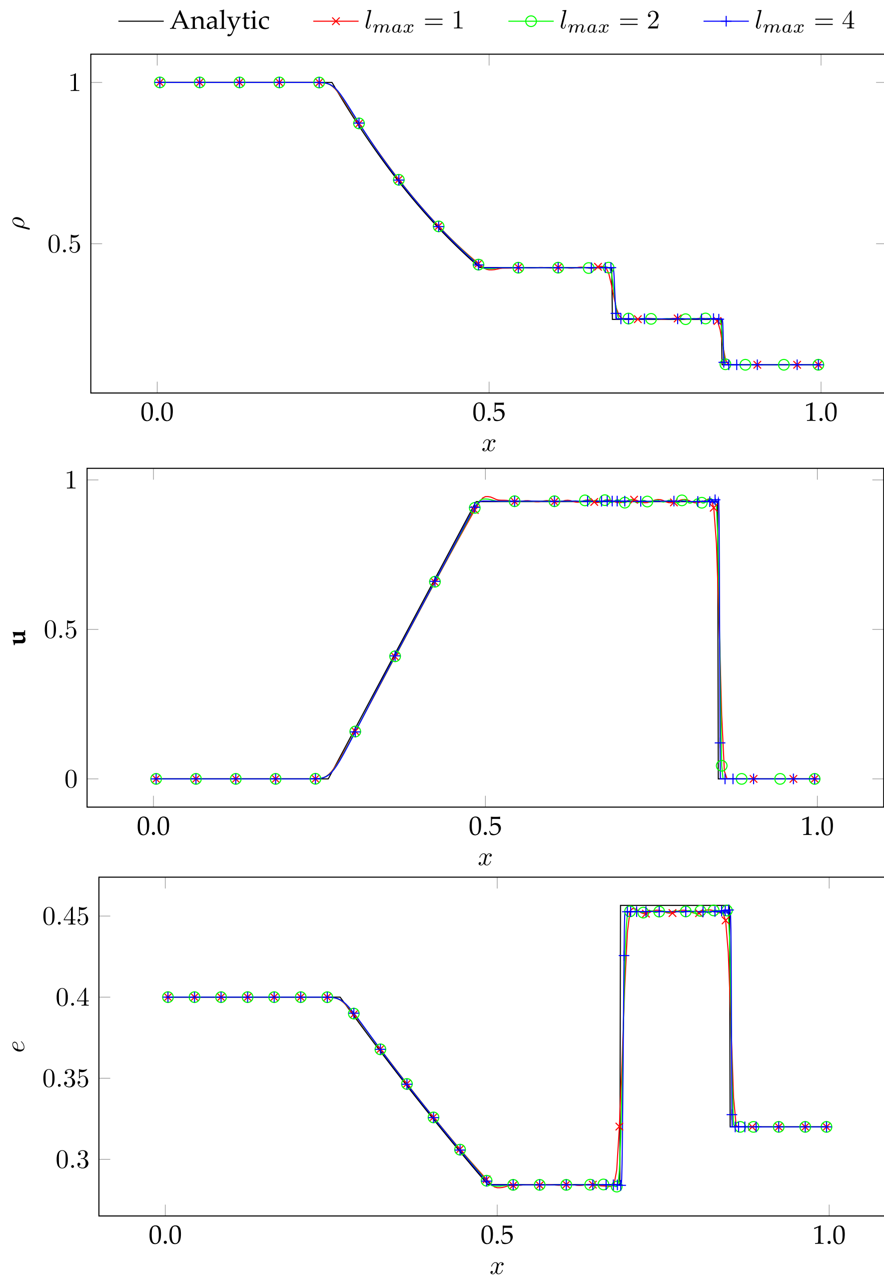
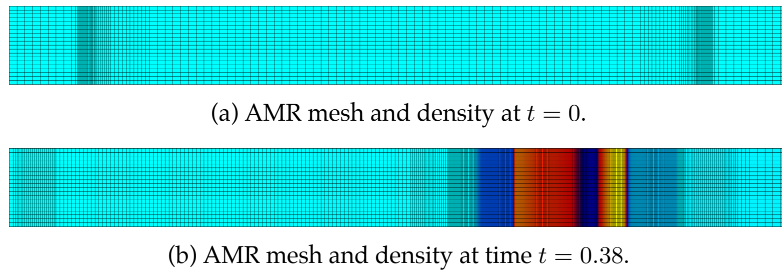


Figure 5.4: Plot of density, velocity, and energy at $t = 0.2$ for the CPU-based solution of Sod's shock tube problem.



(a) AMR mesh and density at $t = 0$.

(b) AMR mesh and density at time $t = 0.38$.

Figure 5.5: AMR meshes and density at the initial and final times of the Woodward-Colella problem.

Woodward-Colella Interacting Blastwaves

We ran the Woodward-Colella problem with a coarse resolution of 100 cells and up to 5 levels of refinement. We also ran the problem with a resolution of 32,000 cells in order to obtain a highly accurate solution with which to calculate fractional errors for each of the resolutions. The blastwaves only travel in the x-dimension, hence we use a refinement ratio of $\{2, 1\}$. Figure 5.6 contains plots of the density, energy, and velocity of the system at time $t = 0.038$ for both the converged, and the one-, two-, and four-level solutions. As the maximum number of levels is increased, the AMR solution converges towards the reference values for all three field quantities.

Sedov Blastwave

The Sedov problem ran with 100 cells in each dimension. Since the shock moves radially outward from the origin, we refine in both the x- and y-dimensions using a refinement ratio of {2,2}. The nature of the Sedov problem means that at time $t = 1$ the radius of the blastwave should be at radius $r = 1$. Figure 5.7 shows the initial and final density and mesh, and from the results in Figure 5.8 we can see that the shockwave does not reach the correct location in the simulation domain. The results produced by CleverLeaf in this case are incorrect. The erroneous blast wave position is caused by the lack of conservation of kinetic energy. Since the advection step of the Lagrangian-Eulerian scheme used in CleverLeaf only conserves mass, internal energy and momen-

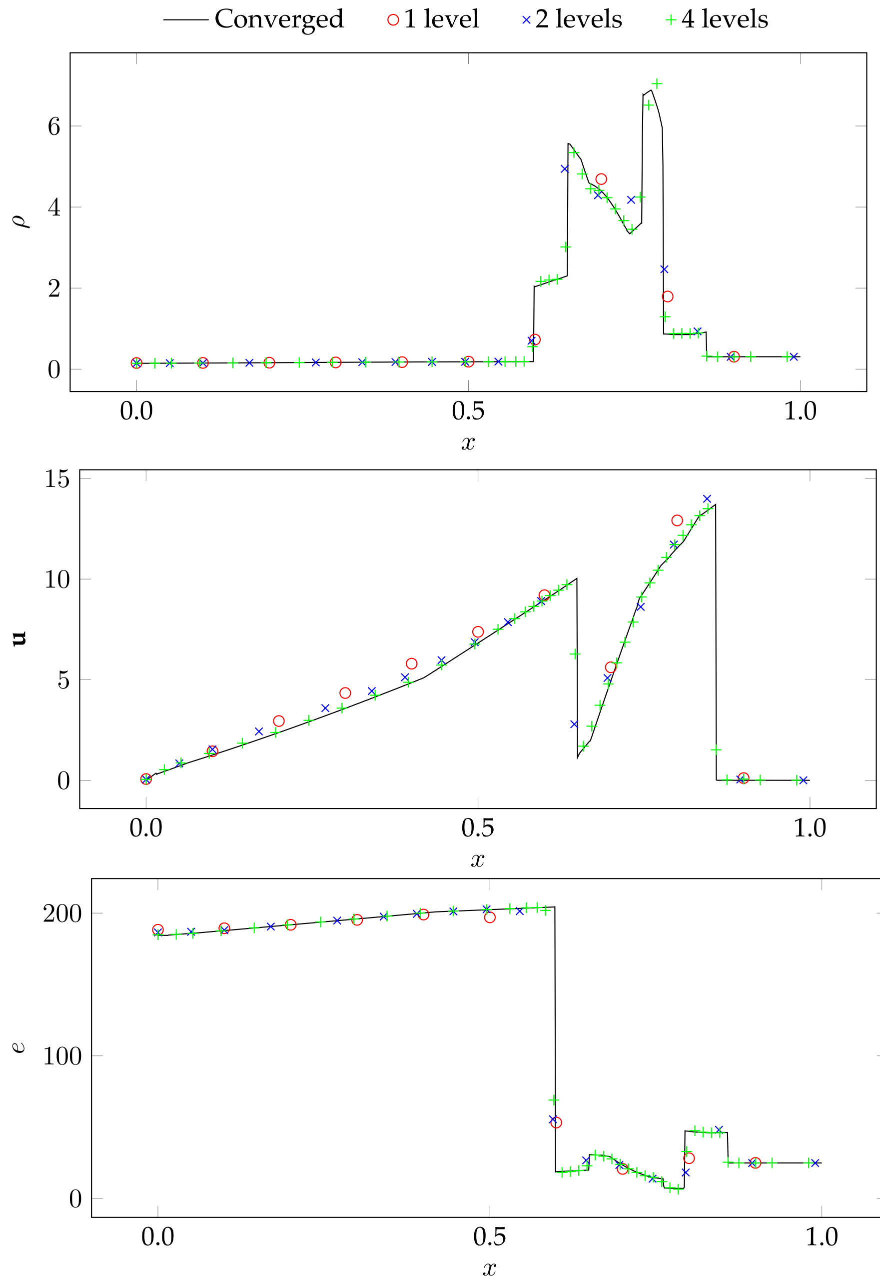
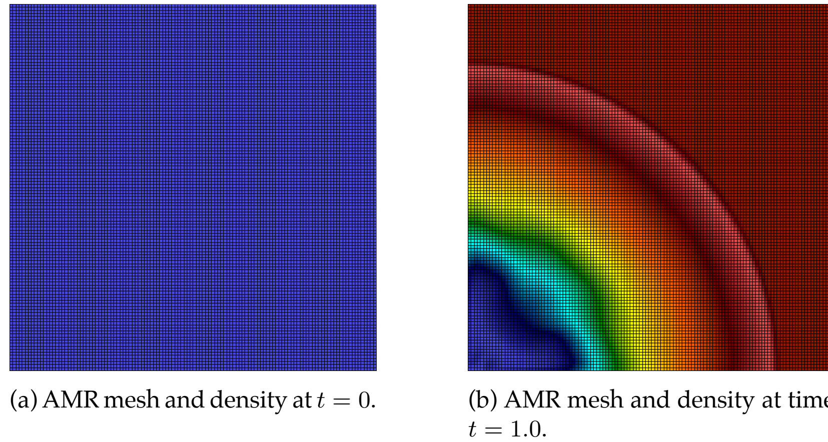


Figure 5.6: Plot of density, velocity, and energy at $t = 0.038$ for the CPU-based solution of Woodward-Colella interacting blastwaves problem.



(a) AMR mesh and density at $t = 0$.

(b) AMR mesh and density at time $t = 1.0$.

Figure 5.7: AMR meshes and density at the initial and final times of the Sedov problem.

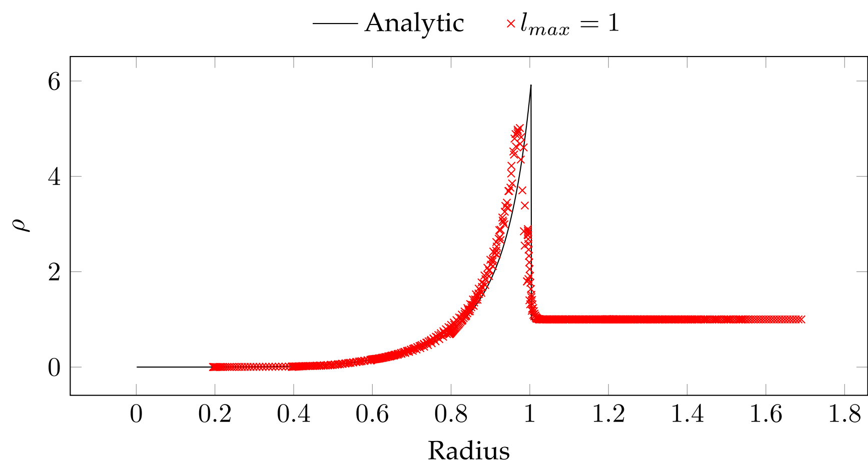


Figure 5.8: Plot of density at $t = 1$ for the CPU-based solution of Sedov's blast wave problem.

tum (but not kinetic energy). Production codes typically contain a fixup step where kinetic energy that has been lost during advection is added back to the field variables [31, 42, 151].

This step is not computationally intensive, but involves complex coding that has been deliberately avoided in both CloverLeaf and CleverLeaf. Although the lack of the fixup means that the results of the simulations can be physically incorrect, the main computational portions of the code: the Lagrangian step and the advective remap; are representative of a larger class of hydrodynamics codes.

Other aspects of a production-ready hydrodynamics code that have been avoided in our mini-applications include support for simulations with multiple discrete materials and more complex equations of state. Different equations of state will have varying performance so introduce an imbalance in the computational work required for each cell, depending on the equation of state being used.

The impact of the addition of multi-material support is entirely dependent on which portion of the code one is considering. Much of the hydrodynamics can remain unchanged, navigating but not updating the multi-material data-structures. If the data structures are well designed, then computational performance may be unaffected. The cell-centred advection step will be affected the most. During this step, the multi-material data structure must be updated as materials move between cells. The number of possible scenarios for cell state (pure cell becomes mixed, a mixed cell because pure, a mixed cell gains/loses components) increases the complexity of the code by adding different control branches. This additional logic will likely harm performance, particularly on emerging parallel architectures like GPUs. Performance of the cell-centred advection kernel is linked to the number of materials in a cell, so will introduce an imbalance in computational work. Finally, during the exchange of boundary cells, message sizes and pack/unpack times will vary depending on the proportion of mixed cells, and their properties, in the boundary region.

The results of this test problem highlight an interesting and important weakness in mini-application development. The aspects of a parent code that remain unimplemented in a mini-application can have an impact on the accuracy of the results produced and possibly on the computational behaviour of the application. Just because a mini-application has been developed by a programmer familiar with the parent code does not mean it will be inherently representative. In the following section, we show that despite the lack of a kinetic energy fixup, multi-material support, or a complex equation of state function, CleverLeaf remains representative of a larger hydrodynamics code with AMR.

5.3.2 Verification

Ensuring that a mini-application and its parent application exhibit the same computational characteristics is hard. Most often mini-application developers rely on the fact that they also developed the parent application to provide some amount of similarity. Thus the design of mini-applications is ad-hoc and driven by implicit assumptions made by the application developer about how they think the parent application influences the system. It is often not clear that the mini-application actually captures the characteristics that are important for the performance of its parent, especially since it often performs a subset of operations. Whilst a mini-application might *look* the same as its parent, and even solve an identical problem, there is no guarantee that the reduction in complexity that is inevitable in a mini-application will have the same demands on hardware resources.

The VERITAS framework uses comparative performance analysis in order to validate mini-applications and ensure that some guarantee can be made about the hardware characteristics that it represents [97]. VERITAS focuses on single node, multi-core performance characteristics to determine which hardware events are responsible for the change in parallel efficiency of an application, and then deduces which of these events are covered by the mini-application. These observations enable developers to understand which characteristics they

can expect to test with a particular mini-application on a particular architecture. An understanding of which performance-critical characteristics are not covered can then be used to drive future mini-application development.

We present our study in the context of Shamrock, a 2-dimensional, Lagrangian hydrodynamics application using AMR, developed at the UK Atomic Weapons Establishment (AWE). Shamrock is an industry-strength benchmark supporting a range of architectures [70]. It is a key benchmark in evaluating future HPC technologies at AWE, yet at almost 100 thousand Lines of Code (LOC), investigating future programming models is difficult and the benchmark is less flexible than a mini-application. As such, a representative mini-application like CleverLeaf will enable such investigations to be conducted rapidly, and the results applied effectively based on the knowledge that the performance of both CleverLeaf and Shamrock are affected by the same hardware characteristics. For this verification, we focus on single node performance, rather than cross-node communication characteristics since CleverLeaf uses a different communication library. Whilst this may alter the representativeness of highly parallel runs, it is important to note that a key driver for the development of CleverLeaf is to provide a framework in which to evaluate the SAMRAI library.

Methodology

At a high level, Veritas collects a range of PAPI counters [27] that measure performance metrics from both the mini-application and some parent application and analyses the data to identify resource utilisation bottlenecks common between these two applications. Specifically, Veritas performs the following three tasks: (i) collects performance metrics, (ii) correlates each performance metric to performance loss, and (iii) validates the mini-application by identifying metrics that are strongly correlated with performance loss for the parent application, and then comparing correlations for these metrics between the parent and the mini-application.

Collection of performance metrics uses a lightweight library that is linked

Counter	Description
PAPI_L2_ICM	L2 instruction cache misses
PAPI_L3_ICA	L3 instruction cache accesses
PAPI_L3_ICR	L3 instruction cache reads
L2_LINES_IN:ANY	Counts the number of lines allocated in the L2 cache
PAPI_L2_DCM	L2 data cache misses
PAPI_L3_DCA	L3 data cache accesses
PAPI_L2_TCM	L2 total cache misses
PAPI_L3_DCR	L3 data cache reads
PAPI_L2_STM	L2 store misses
PAPI_L3_DCW	L3 data cache writes
PAPI_L3_TCW	L3 total cache writes
PAPI_L3_TCM	L3 total cache misses
BR_MISP_RETIRED:ALL_BRANCHES	Counts all mispredicted retired calls
L2_LINES_OUT:DEMAND_CLEAN	L2 clean line evicted by a demand
perf::NODE-PREFETCHES	Node prefetch accesses
perf::DTLB-LOAD-MISSES	Data TLB load misses
perf::DTLB-STORES	Data TLB store accesses
PAPI_L1_ICM	L1 instruction cache misses
PAPI_DP_OPS	Double precision floating point operations executed
perf::LLC-STORES	LLC store accesses
PARTIAL_RAT_STALLS:FLAGS_MERGE_UOP	Number of flags-merge micro-operations in flight in each cycle
PAPI_TLB_DM	Data TLB misses
PAPI_STL_ICY	Cycles with no instruction issue
PAPI_L1_TCM	L1 total cache misses
L1D_PEND_MISS:OCCURRENCES	Occurrences of L1D_PEND_MISS going active
perf::LLC-STORE-MISSES	LLC store misses
PAPI_L1_LDM	L1 load misses
PAPI_L2_DCR	L2 data cache reads
BR_MISP_EXEC:NONTAKEN_COND	All macro conditional non-taken branch instructions
PAPI_FP_INS	Floating point instructions executed
PAPI_FP_OPS	Floating point operations executed
PAPI_BR_TKN	Conditional branch instructions taken
PAPI_L1_DCM	L1 data cache misses
PAPI_L2_DCA	L2 data cache accesses
PAPI_BR_PRC	Conditional branch instructions predicted
PAPI_BR_UCN	Unconditional branch instructions executed
PAPI_BR_CN	Conditional branch instructions executed
PAPI_SR_INS	Store instructions executed
PAPI_BR_INS	Total branch instructions executed
PAPI_BR_NTK	Conditional branch instructions not taken
PAPI_BR_MSP	Conditional branch instructions mispredicted
RS_EVENTS:EMPTY_CYCLES	Cycles the RS is empty for this thread
PAPI_TOT_INS	Total instructions executed
PAPI_LD_INS	Load instructions executed
PAPI_TLB_IM	Instruction TLB misses
PAPI_L1_STM	L1 store misses
PAPI_L2_DCW	L2 data cache writes
PAPI_TOT_CYC	Total cycles
L2_L1D_WB_RQSTS:HIT_E	Non-rejected writebacks from L1 to L2 cache lines
OTHER_ASSISTS:ITLB_MISS_RETIRED	Counts the number of retired instructions that missed the instruction TLB when the instruction was fetched
RESOURCE_STALLS2:ALL_FL_EMPTY	Cycles stalled due to free list empty
PAPI_L2_DCH	L2 data cache hit
RESOURCE_STALLS:ANY	Cycles stalled due to any resource related reason

Table 5.1: Description for the PAPI counters used during the analysis of Shamrock and CleverLeaf.

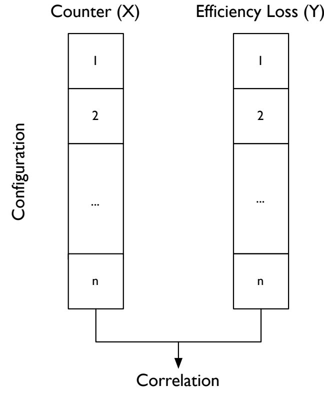


Figure 5.9: Correlation between performance counter and efficiency loss calculated by VERITAS.

into the application. Values for each available PAPI counter (detailed in Table 5.1) are collected in regions deemed by the application developers to be analagous between the parent and mini-application. In the case of CleverLeaf and Shamrock, these are the hydrodynamics kernels, as well as the AMR-specific routines such as flagging areas of interest. Each counter value is stored in an HDF5 file along with information about the parallel configuration used and the region the counter was collected in.

Application performance typically changes depending on the number of parallel tasks (OpenMP threads or MPI ranks) used during a given run. A given number of parallel tasks is called a configuration, and VERITAS calculates the correlation between each collected counter and the parallel efficiency loss of a given configuration using, ρ, Spearman's rank correlation coefficient [148]. This provides a nonparametric measure of the dependence between the two variables. The two sets of variables X and Y are assigned ranks x_i and y_i (for each i in X and Y) based on the ordering of their numeric values. Then:

$$\rho = 1 - \frac{6\Sigma d_i^2}{n(n^2 - 1)} \tag{5.1}$$

where $d_i = x_i - y_i$ is the difference in rank between the two variables and

n is the number of configurations. Analysis is performed by inspection, with the per-counter correlations being plotted for both applications. In the case of VERITAS, each X is a set of values for a single performance counter, and Y is the set of efficiency losses for each configuration, where the efficiency loss of each configuration c is defined as:

$$(1 - \text{Efficiency}(c)) \times 100\% \tag{5.2}$$

where Efficiency is defined as in Chapter 2. Figure 5.9 shows this calculation pictorially. The aim is to identify performance counters that are strongly correlated with a change in performance. If the correlation for a given performance counter event is negative, then there is a negative correlation and as the number of parallel workers increases, the occurrence of this event decreases while the efficiency loss increases, or vice versa. For events such as cache hits or number of instructions per cycle, a negative correlation indicates inefficiency in resource utilisation. A positive correlation indicates that, as the number of workers increases, the occurrence of this event increases along with the efficiency loss, or vice versa. For events such as number of cache misses, a positive correlation indicates contention for resources due to increase in the number of workers.

Experimental Setup

To collect performance metrics for analysis, the following experimental setup is used. Both applications use the Sod problem from earlier in this chapter with 16 different parallel configurations; from 1 to 16 MPI ranks, with each rank bound to a single CPU core. We collect values for 45 available PAPI counters for each configuration and then use the VERITAS framework to calculate the correlation between each performance counter and the loss in parallel efficiency of each configuration. VERITAS then compares the events that are strongly correlated with efficiency loss between the mini-application and the benchmark.

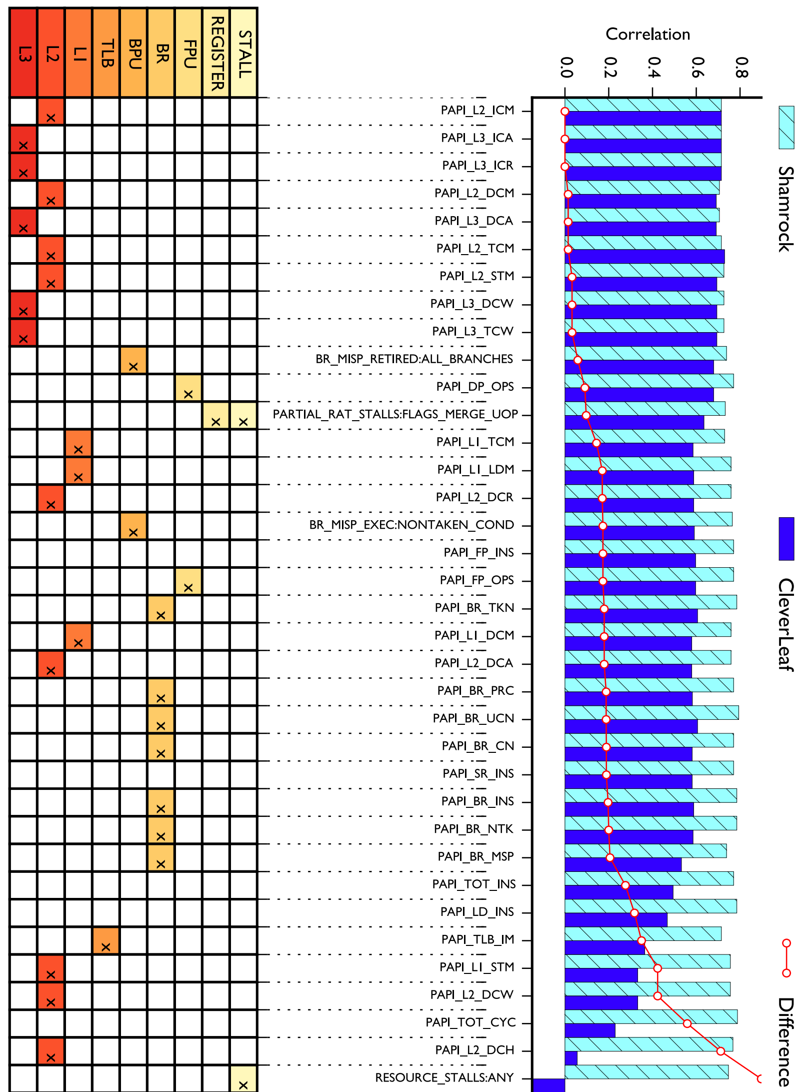


Figure 5.10: Performance characteristics of Shamrock covered by CleverLeaf.

Analysis

Figure 5.10 contains all events that have a correlation with efficiency loss (for Shamrock) of greater than 0.7. Looking at these events, we see that for the majority of events strongly correlated with efficiency loss, the difference between Shamrock and CleverLeaf is small, and hence we say that these performance events are well covered by CleverLeaf. Each hardware counter can be linked to a particular hardware resource to allow a more intuitive understanding of the results. The link between counters and hardware resources is determined by inspection, and the hardware resource attributed to each counter is shown at the bottom of Figure 5.10. This classification of counters allows us to state that most of the L2 and L3 behaviour of AMR and hydrodynamics algorithms of interest to AWE that are present in Shamrock are covered by CleverLeaf.

The results in Figure 5.11 focus on events that have a strong correlation with efficiency loss (greater than 0.7) in Shamrock, yet have a weak correlation in CleverLeaf. These events are ones that we consider poorly covered. The first counter, `PAPI_TOT_CYC`, simply measures the total number of Central Processing Unit (CPU) cycles performed by each application. Since Shamrock is more complicated than CleverLeaf and will perform more cycles, we can expect these values to differ significantly without loss of representativeness. The other memory-related counters that show a large difference in correlation are related to stalls. We predict that the data layout and array sizes in Shamrock cause a different frequency and count of such stalls when compared to CleverLeaf, thus having a different impact on parallel efficiency.

These results allow us to state that the majority of performance characteristics, particularly those related to the cache behaviour, of Shamrock are covered by CleverLeaf. Overall, the scaling behaviour of the performance events that pertain to shared resources (L2, L3) with respect to their corresponding performance loss match between these two applications. Additionally, the performance characteristics that we deem to be poorly covered may not have a critical impact on application performance.

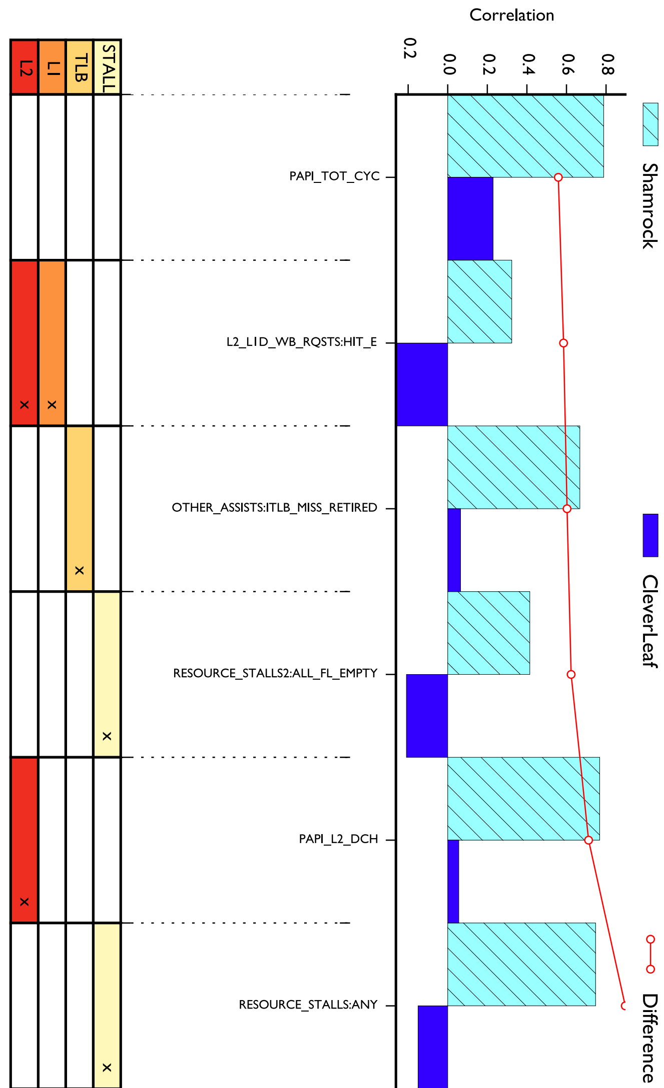


Figure 5.11: Performance characteristics not well covered by CleverLeaf.

By grouping the performance counters by the hardware resources that they are measuring and focusing on the mean correlation with efficiency loss, we can gain a more intuitive understanding of the hardware coverage of CleverLeaf. Each counter is grouped based on the hardware resources it is determined to be related to. This grouping is performed by manual inspection, and implemented within the Veritas framework (see Table 5.2). Figure 5.12 contains plots mean correlation with efficiency loss for each group. We can see that the correlation between cache behaviour and efficiency loss is similar between the two applications. Two of the hardware resource groups not well covered are the CPU events, which we have already discussed as will be different, and performance counters attributed to the HIT group. This group contains events relating to attempted memory accesses that result in some kind of cache hit. The difference in these counts will be affected by the data stored and the memory layout. Since this layout and data size differ between CleverLeaf and Shamrock, these results reinforce the notion that these differences are caused by a change in data layout, not in computational performance.

5.4 Summary

The complex nature of simulations using AMR makes them an ideal candidate for representation via mini-applications. With the adaptive mesh refinement algorithms encapsulated in library code, we can perform performance studies easily and investigate new programming languages and models

In this chapter we have described CleverLeaf, the first open-source block-structured adaptive mesh refinement mini-application. In particular we describe the algorithms and simulation code that allow us to use CleverLeaf to study adaptive mesh refinement at large scale and on emerging parallel architectures. We have validate the mathematical accuracy of CleverLeaf using three test problems, thus ensuring that the physics performed by CleverLeaf is correct and a true reflection of the hydrodynamics algorithms used in production

Counter	Hardware Resource Groups
L2_STORE_LOCK_RQSTS:HIT_E	L2, INS
HW_PRE_REQ:L1D_MISS	L1, MISS, PREFETCH
perf::CACHE-MISSES	L1, L2, L3, LLC
L1D_BLOCKS:BANK_CONFLICT	L1, STALL
perf::LLC-PREFETCHES	L3, LLC, PREFETCH
perf::PERF_COUNT_HW_CACHE_MISSES	L1, L2, L3, LLC, MISS
RS_EVENTS:EMPTY_CYCLES	RS, STALL
L2_LINES_IN:ANY	L1, L2, PREFETCH
LD_BLOCKS_PARTIAL:ADDRESS_ALIAS	LSU, STALL
perf::LLC-LOADS	L3, LLC
LD_BLOCKS:DATA_UNKNOWN	LSU, STALL
PERF_COUNT_HW_CACHE_LL:READ	L3, LLC
LAST_LEVEL_CACHE_MISSES:e=0	L3, LLC, MISS
perf::LLC-LOAD-MISSES	L3, LLC, MISS
RESOURCE_STALLS:ANY	STALL
PAPI_TOT_CYC	CPU
perf::TASK-CLOCK	CPU
PAPI_DP_OPS	FPU
PAPI_FP_OPS	FPU
perf::PERF_COUNT_HW_CACHE_REFERENCES	L1, L2, L3, LLC
perf::CACHE-REFERENCES	L1, L2, L3, LLC
PARTIAL_RAT_STALLS:FLAGS_MERGE_UOP	REGISTER, STALL
INT_MISC:RAT_STALL_CYCLES	REGISTER, INS, STALL
perf::LLC-STORES	L3, LLC
PAPI_STL_ICY	INS, STALL
LAST_LEVEL_CACHE_REFERENCES:e=0	L3, LLC
LLC_MISSES:e=0	L3, LLC, MISS
LLC_REFERENCES:e=0	L3, LLC
IDQ:EMPTY	INS
BR_MISP_RETIRED:ALL_BRANCHES	BPU
BR_MISP_EXEC:NONTAKEN_COND	BPU
BACLEARS:ANY	BPU
LSU_COMMIT_STCX	L2
LSU_LARX_FINISHED	L2
L2_STORE_LINE	MEM
L2_STORE_LINE_SLICE	MEM
L2_TRANS:L1D_WB	L1
PAPI_L1_STM	L2
PERF_COUNT_HW_CACHE_MISSES	L1, L2, L3, LLC, MISS
DATA_WRITE_MISS	L1, L2, L3, LLC, MISS
DATA_READ_MISS	L1, L2, L3, LLC, MISS
data_write	L1, MEM
DATA_READ	L1, MEM
DATA_READ_OR_WRITE	MEM
DATA_PAGE_WALK	TLB
DATA_CACHE_LINES_WRITTEN_BACK	L1, L2, LLC, DATA
LONG_DATA_PAGE_WALK	TLB, DATA
LONG_CODE_PAGE_WALK	TLB, INS
PERF_COUNT_HW_CACHE_L1D:READ	L1, DATA
PERF::CACHE-MISSES	L1, L2, LLC, MISS
PAPI_L1_DCM	L1, DATA
perf::PERF_COUNT_HW_CACHE_BPU	BPU
SNP_HITM_BUNIT	BPU
PIPELINE_FLUSHES	BPU, PIPELINE
L2_DATA_WRITE_MISS_MEM_FILL	MEM, MISS

Table 5.2: Groups assigned to PAPI counters by VERITAS.

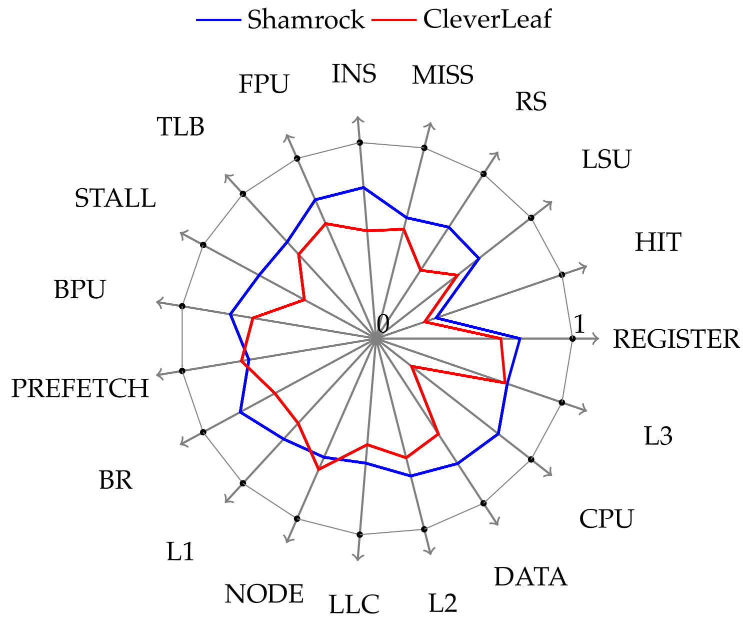


Figure 5.12: Mean correlation with efficiency loss grouped by hardware resource.

codes. The small errors observed fall well within expected tolerances and allow us to assert that the adaptive mesh refinement algorithms have been correctly applied to advance the simulation. Finally, we have verified that CleverLeaf is representative of some of the key behaviours in Shamrock, a production-quality benchmark application.

CHAPTER 6

Scalable AMR on Graphics Processing Units

Massively parallel accelerator architectures like an attached Graphics Processing Unit (GPU) often provide order of magnitude improvements in application performance [7, 124, 162]. With tremendous memory bandwidth and the ability to operate on hundreds of data items in parallel, these architectures provide the perfect platform for many High-Performance Computing (HPC) applications providing they are ported correctly. These many-core architectures are the natural extension of the architectural trends introduced by multi-core processors, and consist of processors with even more cores running at lower frequencies. It is now common to see an accelerator attached to a multi-core processor.

An algorithm with structured data and little dynamic change throughout its execution might be ported to an accelerator with little effort, provided it exhibits sufficient parallelism. As an example, the OpenACC version of CloverLeaf discussed in Chapter 4 took little over 12 weeks to port to GPUs. More complex algorithms such as the parallel sweep along a hyperplane described by Lamport (and used in many scientific applications) are much harder to modify,

even though they operate on structured data [92, 127]. In an Adaptive Mesh Refinement (AMR) application like CleverLeaf, the algorithm is sufficiently parallel, but the hierarchy of patches adapts throughout the simulation as the areas of interest move. Despite the potential for large savings in computation time and memory usage with maintained accuracy, AMR requires dedicating a portion of application runtime to managing the mesh hierarchy; this requires complex data management and communication.

Most AMR applications run exclusively on the Central Processing Unit (CPU), and those that do use GPUs often copy the necessary data between GPU and CPU memory at the beginning and end of every GPU-based routine [110, 142, 162]. In this chapter, we present the first *resident* implementation of block-structured AMR on GPUs. Building on the SAMRAI framework [98], we create classes that manage the life cycle of AMR patches where data is stored exclusively on the GPU. All routines that manage the patch hierarchy continue to be handled by SAMRAI on the CPU, but all AMR-specific routines that operate on patch data, such as the coarsening and refining of data between adjacent levels in the hierarchy, execute on the GPU.

Using the object-oriented interface of SAMRAI we develop a set of routines and data structures that allow patch-based data to reside on and be manipulated by the GPU. We use these extensions to write a GPU-based version of CleverLeaf that performs over 4$\times$ faster than the CPU-based implementation on a single node, and has been scaled to over 4 thousand GPUs using a combination of MPI and CUDA. In this chapter, we describe the design and implementation of our GPU-based AMR library extensions, including the classes used to manage patch data, the routines used for transferring data between GPUs on different nodes, and the data parallel operators developed to coarsen and refine mesh data. We also validate the accuracy of our implementation on three test problems, and present performance studies using up to 4,096 NVIDIA K20x GPUs.

6.1 Related Work

Many computational physics codes have been ported to GPUs since the release of CUDA in 2007 [7, 23, 58, 76, 102, 141], and although Berger's adaptive mesh refinement algorithm was presented in 1984, there is little work where AMR codes have been ported to GPUs. One explanation for the lack of GPU-based AMR codes is the large amount of data management required when updating the adaptive hierarchy, and the fact that the naïve method for porting codes to GPUs involves repeatedly copying simulation data to and from the GPU across the slow Peripheral Component Interconnect (PCI) bus.

An early paper by Wang et al. describes an implementation of a compressible flow solver with AMR on GPUs [162]. At the beginning and end of the Runge-Kutta kernel used to advance the solution, the required data must be copied between the CPU and the GPU. This basic implementation achieves an order-of-magnitude speedup over a single CPU core, although with today's supercomputer nodes having at least 16 processor cores, this number is not sufficient for the application to perform faster at the node level.

In [30] the authors briefly describe an AMR algorithm based on a forest-of-octrees for seismic wave propagation on GPUs. The implementation doesn't appear to be fully resident. Although the text lacks sufficient details about the GPU-based implementation, the results presented include timings for transferring the mesh and initial data to the GPU from the CPU memory. Nevertheless, the parallel performance of the code is scalable on up to 256 GPUs.

Schive et al. introduce GAMER, an astrophysical simulation code with both AMR and GPU support [143]. Both the Eulerian hydrodynamics and self-gravity phases of the application are solved on the GPU, but the necessary data is stored in the CPU memory, and must be transferred to the GPU memory before the computational kernel is launched. The data transfer is performed concurrently with other computation, so the impact is minimised, and the authors note that data transfer time typically only takes 30% of the application

runtime.

The Uintah framework from the University of Utah is an AMR framework that supports GPUs [81, 110]. The focus in Uintah is on heterogeneous platforms, and as with GAMER, solution data must be copied between the CPU and GPU memory as required by the numerical kernels. These data transfers are overlapped with other work, but nevertheless, this is not a fully resident framework.

Shamrock, an Eulerian hydrodynamics benchmark with AMR developed at the Atomic Weapons Establishment (AWE), supports execution on GPUs via OpenCL [70]. Only a small fraction of the necessary methods are ported to the GPU, and data is again copied between the CPU and GPU memory at the beginning and end of the four routines that are ported. As is to be expected, the performance of the GPU-based routines was better than the CPU-based equivalents when data transfer time was excluded. Using a simple performance model and the application of Amdahl's law, the authors predict an order of magnitude speed up if 95% of the application is ported to the GPU and data transfer only occurs at the start and end of the simulation.

The CLAMR application, developed at Los Alamos National Laboratory (LANL), is a cell-based AMR code that solves the shallow-water equations [116]. Implemented in OpenCL, the code is fully resident. Initial conditions are set on the CPU and then copied to the GPU memory at that start of the simulation, but data is not copied back to the CPU during the simulation timestep. The cell-based scheme is different to the block-structured approach described by Berger and used in our work.

The most promising application is presented in [142], which describes a resident implementation of patch-based AMR application for solving the shallow-water equations. The authors take a similar approach to our library and ensure all computationally expensive parts of the AMR library are executed on the GPU, and they demonstrate performance improvements of up to $3.4\times$ compared to a uniform GPU-based implementation of the same algorithm. Despite

the similarities to our work, the application domain is different, and there is not the focus on large-scale parallel performance analysis.

To the best of our knowledge we have developed the only fully resident GPU-based shock hydrodynamics code with AMR. Furthermore, by developing the necessary code as part of the SAMRAI library, we provide a collection of components that can be re-used in other block-structured AMR applications.

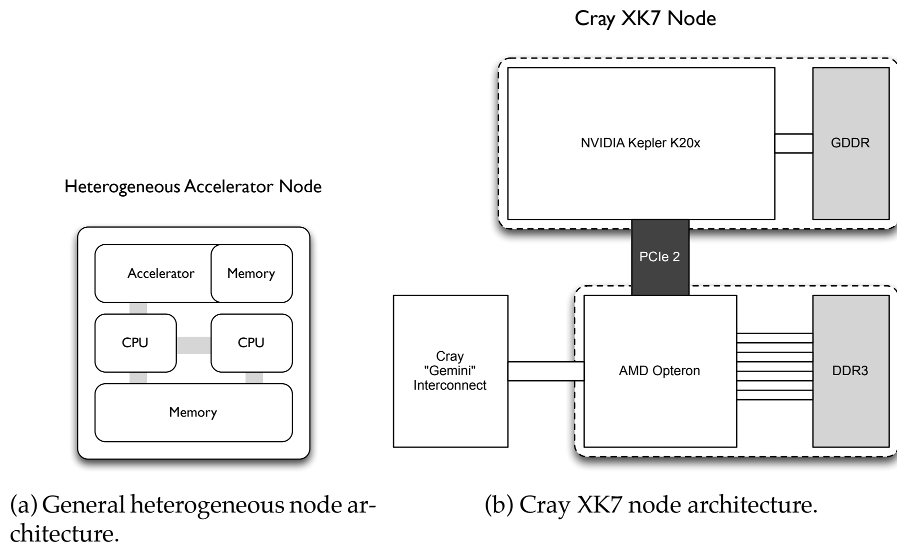
6.2 Design and Development

In this section we describe our GPU-based extensions to the SAMRAI library that allow AMR simulations to execute using accelerator-based node architectures. To test the library in the context of a real application, we extend the CleverLeaf mini-application, and perform a performance analysis on over four thousand GPUs. Development of our library, and the extensions to CleverLeaf, are made easier by adherence to the design patterns present in SAMRAI. We highlight the essential object-oriented abstractions that allow our GPU-based library to be fully compatible with existing SAMRAI code.

6.2.1 Programming Models

The design of the GPU-based extensions to SAMRAI are constrained by the design of heterogenous accelerator nodes. It is now common to see an accelerator attached to at typical multi-core processor. These devices are specialised for fast floating point performance and have their own memory. Currently, the CPU and GPU communicate by transferring data across the PCI bus. This link between the two memory spaces is much slower than access to main memory, so a key design point is avoiding unnecessary transfer of data over this interface.

Figure 6.1a shows the general design of a heterogeneous node, and Figure 6.1b is a simplified schematic of a node from Titan, a large GPU-based supercomputer at Oak Ridge National Laboratory. In both cases, the GPU and



(a) General heterogeneous node architecture.

(b) Cray XK7 node architecture.

Figure 6.1: Accelerator-based heterogeneous architectures.

CPU have distinct memory spaces. Additionally, in Titan's Cray XK7 architecture, the network controller is connected to the CPU. When data must be transferred from the GPU across the network, it must first be copied to the CPU memory.

Programming for GPUs has typically required the use of a programming model such as CUDA or OpenCL [119, 155]. More recent developments in directive-based approaches like OpenACC and OpenMP 4 provide another way to execute code on an attached accelerator [120, 122]. For this work, we use NVIDIA's CUDA programming model, as it is the most mature and feature-rich model for programming NVIDIA hardware. GPU functions are written as *kernels* which are executed simultaneously in a single-instruction-multiple-data (SIMD) fashion on the device.

A CUDA-capable GPU is a collection of stream multiprocessors (SMs), consisting of a number of stream processors (SPs) that share an instruction cache. The CUDA programming model revolves around the concept of threads, blocks, and grids that execute on these hardware units. A thread executes on a single SP, and blocks are groups of threads that are mapped to SMs and will

execute concurrently. A grid is a collection of thread blocks, typically dependent on the size of the data being manipulated. The grid can be either one- or two-dimensional, and defines the total index space for the threads. These grids are used to map threads onto portions of the application domain. When a device kernel is launched, each thread runs one instance of the kernel. The co-ordinates of a thread can be accessed inside the kernel, allowing each thread to determine which elements of global data to process.

OpenCL uses a similar programming model to CUDA, with GPU functions being written as kernels that will be executed in parallel on a given device. The use of CUDA in our work is an implementation detail, and the techniques we apply would map equally well to OpenCL. The OpenACC and OpenMP programming models rely on source code annotation to mark regions of code for execution on the GPU. These annotations are designed to be flexible and portable between different architectures, and hence tend to discourage explicit control of important parameters such as the number of threads launched. They also hide the low level control required to explicitly manage memory, a feature that is essential in our library.

6.2.2 Extending SAMRAI with CudaPatchData

The SAMRAI library uses object-oriented design patterns to allow for easy interaction with user-supplied code [78]. Each of the basic structural units of the AMR hierarchy: patches, patch levels, and the patch hierarchy itself; are provided as fundamental software constructs by SAMRAI. The `Patch` class is a container for all the data living in a particular mesh region, and provides a way to access and manipulate this data. All the data on a patch are handled using `PatchData` objects, each of which represents some simulation quantity on the mesh. The `PatchData` interface uses the Strategy design pattern [161], and defines a set of operations that a class must provide in order to be interoperable with SAMRAI's data management and communication routines. We use this interface to develop a library capable of storing patch-based data in GPU

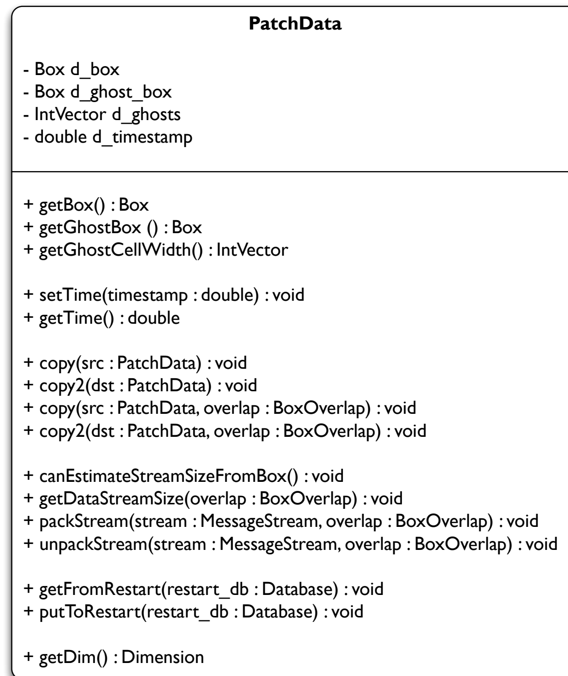


Figure 6.2: The SAMRAI PatchData interface.

memory whilst still using SAMRAI for mesh management, communication and visualisation.

SAMRAI's PatchData Interface

The `PatchData` interface is used to define the operations a class needs to support to allow SAMRAI to gather data from the patch in order to transfer it to other patches in the hierarchy. Figure 6.2 documents the full `PatchData` interface. During a simulation, `Patch` objects store an array of pointers to the necessary `PatchData` objects. The functions that the `PatchData` routines perform include copying data from one `PatchData` object to another, packing the data corresponding to a given region of the patch into a message buffer, and unpacking data from a message buffer into a given patch region. These methods are the key points of the `PatchData` interface that we implement.

The ability of this interface to allow applications to use their own datatypes

whilst still accessing the full capability of SAMRAI to manage a simulation on an adaptive mesh has been used to its full potential in a number of applications. For example, in [163], Wijesinghe et al. present a hybrid hydrodynamics code that uses a combination of continuum methods and Direct Simulation Monte Carlo to perform simulations at a range of scales. This implementation relies on a custom datatype, but the AMR capability of the SAMRAI library.

By allowing an application to fully control data management, SAMRAI is easy to use in an existing application. In the case of our GPU-based extensions, the abstraction provided by the `PatchData` interface let us store simulation data in the GPU memory at all times and only copy data across the PCI bus when necessary.

CudaPatchData Library

The library we have developed contains two packages, modelled after those in the main SAMRAI distribution. The first, `pdat`, contains three different `PatchData` implementations for managing data in GPU memory. The second, `geom`, provides a collection of coarsen and refine routines that are essential when copying data between patches at different levels of the hierarchy. We describe both packages in the remainder of this section.

The three different `PatchData` implementations we have developed are collectively called `CudaPatchData`, and the three implementations exist since they are specialised for the three data centrings required for the hydrodynamics scheme used in CleverLeaf. The common data store for each class is the `CudaArrayData` object. This class is responsible for allocating a contiguous array of data in GPU memory, corresponding to a given box size. This class also contains data-parallel routines to copy data, pack a region of the array into a buffer, and unpack a buffer into a region of the array. Each data-centring passes a slightly different `Box` to the `CudaArrayData` object it owns, allowing the necessary data to be stored. The three centrings required for CleverLeaf are: cell-centred, node-centred, and side-centred. Figure 6.3 shows the design and data

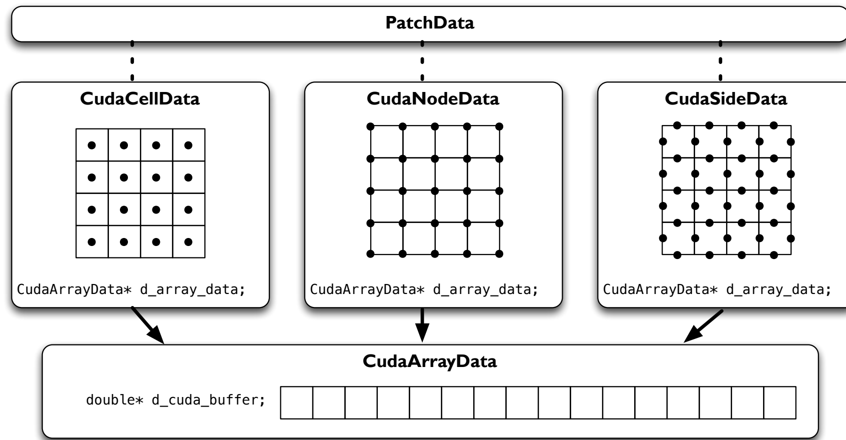


Figure 6.3: The SAMRAI CudaPatchData datatypes.

stored by each class.

During an AMR simulation, boundary conditions can be filled in one of three ways: (i) using the physical boundary conditions; (ii) with data from a neighbouring patch on the same level; or (iii) with data from a patch on the next coarser level. Filling the boundary cells with the physical boundary conditions is handled by the application, and requires no additional features to be added to our library. When data must be transferred between two patches at the same level of refinement, a copy routine is used. If the two patches involved in the copy operation are located on different nodes the required data must be transferred using MPI. Supporting MPI is essential for any modern scientific code, and by including the necessary routines in our library we can run on multiple GPUs.

The data-parallel copy and packing operators are designed as follows. Each operation will receive a `Box` as one of its parameters. This box contains the region of the patch that needs to be operated on (whether data is being copied, packed, or unpacked). In all cases, the size of this box controls the number of CUDA threads that will be launched. Each thread will then be responsible for copying, packing, or unpacking one array element.

In the case of the pack and unpack methods, we provide CUDA kernels

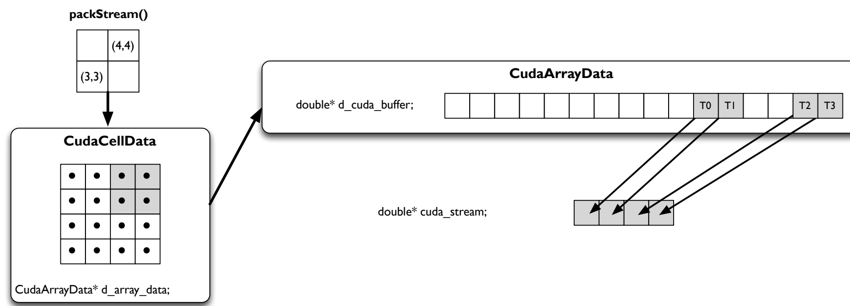


Figure 6.4: Data-parallel buffer packing for MPI operations.

to pack data from the required region into a contiguous buffer in GPU memory. This buffer is then copied to the host memory and passed to SAMRAI, which handles the MPI communications. To unpack received data, the buffer is copied into the GPU memory and then unpacked in parallel using another CUDA kernel. Once the data has been transferred, a new `PatchData` object is created locally and the copy operators described previously can be used to fill the boundary cells on the receiving processor. We launch one CUDA thread per element to be packed into the buffer, ensuring the maximum amount of parallelism is exposed. As an example, Figure 6.4 shows how the overlapping region is copied into the contiguous buffer in parallel.

The operators described by the `PatchData` interface are sufficient for transferring data between objects at the same level of refinement. However, to transfer data between objects at different refinement levels, we must use a refinement operator (if data is moving to a higher refinement level) or a coarsen operator (if data is moving to a lower refinement level). In SAMRAI, these operations are handled by two interfaces: `CoarsenOperator` and `RefineOperator`, that provide the necessary methods for coarsening or refining data. To allow the `CudaPatchData` classes to be used in an AMR simulation, we must provide operators to coarsen and refine data between different levels of the hierarchy.

We provide four data-parallel operators for coarsening and refining data. As with the copy, pack, and unpack routines, each method executes using multiple CUDA threads in a highly parallel fashion. These are, to the best of our

knowledge, the first data-parallel implementations for each of these operators. As an illustrative example, we consider linear interpolation for node-centred data. The code listing for our data-parallel algorithm is shown in Figure 6.5. In a typical implementation, data dependencies exist between temporary variables in different loop iterations and the algorithm is not immediately amenable to the data-parallel programming model of a GPU. Through substitutions and operation re-ordering, we remove these dependencies and develop an algorithm that *is* fully data-parallel (see Figure 6.6). When this kernel is launched to refine some region of data, one CUDA thread can be used per fine node, offering massive parallelism. We also provide conservative linear refine operators for the cell- and side-centred data, as well as a node-centred injection operator.

Following the same general design as SAMRAI, we have partitioned our library into two packages. The first, `pdat`, contains all the data types described so far, the second `geom` contains the operators used to coarsen and refine data on a Cartesian grid. Together, the `pdat` and `geom` packages provide all the necessary components for a block-structured AMR simulation to be solved on a GPU. All that the user code must provide is a black-box integrator that can advance the simulation on a single patch.

6.2.3 Adding CudaPatchData to CleverLeaf

CleverLeaf uses a single class to control the integration of the numerical solution over patches. This class functions as a black box, and the remaining routines written to advance the simulation on the mesh hierarchy can remain unchanged even when a different patch integrator is used. The similarity in the interfaces between the CPU-based `PatchData` classes and their GPU-based counterparts meant that we were able to easily modify the existing code to support GPU-based execution.

To develop the GPU-based version of CleverLeaf, we created a new patch integrator class that contains the code specific to advancing the solution on a single patch on a GPU. All references to the CPU-based `PatchData` objects pro-

```
const int nblocks = (fine_box_size + BLOCK_SIZE - 1)/BLOCK_SIZE;

const int2 ratio = make_int2(ratio_vector[0], ratio_vector[1]);

cudaEvent_t fine_kernel;
cudaStreamSynchronize(coarse_stream);

cartcudanodelinrefine2d_kernel<<<nblocks, BLOCK_SIZE, 0,
  fine_stream>>>(
    coarse_data,
    coarse_data_offset,
    coarse_data_width,
    fine_data,
    fine_data_offset,
    fine_data_width,
    fine_box_width,
    fine_box_height,
    ratio);

cudaEventCreate(&fine_kernel);
cudaEventRecord(fine_kernel, fine_stream);
cudaStreamWaitEvent(coarse_stream, fine_kernel, 0);
```

(a) Host C++ code for launching the data-parallel linear refine kernel.

```
if (column < fine_box_width && row < fine_box_height) {

  const double realrat0 = 1.0/static_cast<double>(ratio.x);
  const double realrat1 = 1.0/static_cast<double>(ratio.y);

  const int ic0 = floor(column/static_cast<double>(ratio.x));
  const int ic1 = floor(row/static_cast<double>(ratio.y));

  const int ir0 = column - (ic0*ratio.x);
  const int ir1 = row - (ic1*ratio.y);

  const double x = ir0*realrat0;
  const double y = ir1*realrat1;

  const int coarse_index = (coarse_data_offset+ic0) + (ic1*
    coarse_data_width);

  arrayf[fine_index] = (arrayc[coarse_index]*(1.0-x) + arrayc[
    coarse_index+1]*x)*(1.0-y)
   + (arrayc[coarse_index+coarse_data_width]*(1.0-x) + arrayc[
    coarse_index+1+coarse_data_width]*x)*y;
}
```

(b) Data-parallel CUDA linear refine kernel for node-centred data.

Figure 6.5: Host and device code for data-parallel node-centred linear refine.

```
do ic1=ifirstc1,ilastc1
  if1=ic1*ratio(1)
  do ir1=0,ratio(1)
    ie1=if1+ir1
    if ((ie1.ge.filo1).and.(ie1.le.(fihi1+1))) then
       do ic0=ifirstc0,ilastc0
         if0=ic0*ratio(0)
         do ir0=0,ratio(0)
           ie0=if0+ir0
           if ((ie0.ge.filo0).and.(ie0.le.(fihi0+1))) then
             x = dble(ir0)*realrat0
             y = dble(ir1)*realrat1
             arrayf(ie0,ie1)= (arrayc(ic0,ic1)*(one-x) +
               & arrayc(ic0+1,ic1)*x)*(one-y) +
               & (arrayc(ic0,ic1+1)*(one-x) +
               & arrayc(ic0+1,ic1+1)*x)*y
           endif
         end do
       end do
    endif
  end do
end do
```

Figure 6.6: Original Fortran linear refine code for node-centred data.

vided by SAMRAI were replaced with references to the GPU-based `CudaPatchData` objects we have developed. As in Chapter 5, we advance the simulation by passing a pointer to the data from these objects to CUDA kernels functions (rather than the existing numerical methods written in Fortran). Figure Figure 6.7 shows how the two patch integrator classes are driven by the top level algorithm.

Control of data communication and mesh management continues to provided by the `LagrangianEulerianIntegrator` and `LagrangianEulerianLevelIntegrator` classes through SAMRAI's objects. Observing and implementing the `PatchData` interface meant that no additional changes were needed to allow CleverLeaf to run on GPUs.

Within the `Cudaleaf` class we require three routines (in addition to the hydrodynamics kernels) to allow data-parallel execution on GPU hardware. These routines are used to flag cells for refinement and to coarsen data between two levels in two specific ways: mass-weighted, and volume-weighted.

Evaluating the tagging heuristic at each mesh cell is trivially parallel. Since the heuristic does not update any mesh data, and since each point can

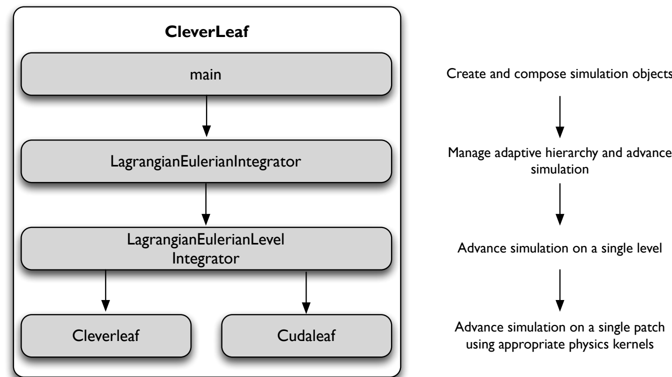


Figure 6.7: Flexible CPU and GPU implementation in CleverLeaf.

be calculated independently of any other, developing a data-parallel tagging routine just meant launching enough CUDA threads to evaluate each point in the patch. However, once cells had been flagged for refinement, we had to transfer them to the host memory to allow SAMRAI to construct the updated mesh hierarchy.

To transfer the data, we compress the array of tags (stored as integers) to an array of bits, where a 1 represents a flag, and 0 represents no flag. This compression minimises the amount of data that must be transferred, and is particularly important when a patch is large. Additionally, we store a tagged variable for each patch. If no cells in a patch are flagged for refinement then we don't copy data, since creating the appropriate data (an array filled with 0s) in host memory is trivial.

Volume- and mass-weighted coarsen operations are essential in hydrodynamics simulations using AMR because they ensure that the quantities being simulated are conserved. To the best of our knowledge, we present the first implementation of these data-parallel operators. Each coarsen operator follows the same general pattern, with one CUDA thread being launched for every coarse value that needs to be filled. This thread then reads the relevant fine values and performs the necessary mathematical operations to calculate the coarse value. Figure 6.8 shows this operation for the volume-weighted coarsen

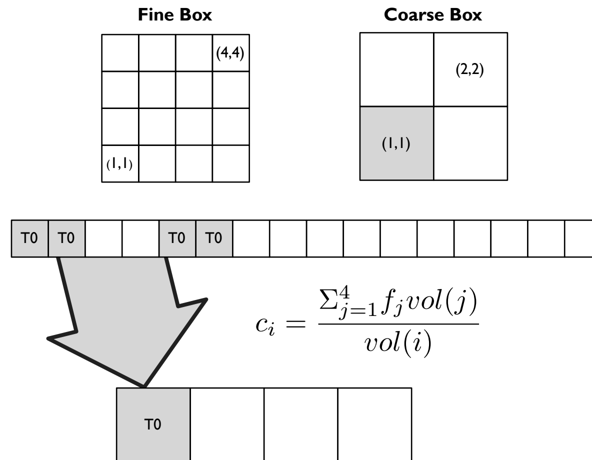


Figure 6.8: Data-parallel volume-weighted coarsen operator.

schematically, and the algorithm we use is presented in Figure 6.9.

Combining the `CudaPatchData` classes and the routines described in this section allows CleverLeaf to simulate Euler's equations natively in GPU memory. Simulation data is stored in global memory at all times, and the relevant regions of data are copied to the host memory in three situations: regridding, boundary updates, and synchronisation. In the following sections we study the accuracy and performance of our GPU-based AMR mini-application.

6.3 Validation and Verification

In this section we present a detailed study of the accuracy and performance of the GPU-based implementation of CleverLeaf. Despite the main role of mini-applications being to investigate issues surrounding application performance, we also present an accuracy study to reassure the reader that our GPU-based library functions correctly and is ready to be used in any block-structured AMR application.

```
const int global_index = blockDim.x * blockIdx.x + threadIdx.x;

const int coarse_row = global_index / box_width;
const int coarse_column = global_index % box_width;

const int fine_row = coarse_row*row_ratio;
const int fine_column = coarse_column*column_ratio;

const int coarse_index = (coarse_data_offset + coarse_column)
  + (coarse_row * coarse_data_width);
const int fine_index = (fine_data_offset + fine_column)
  + (fine_row * fine_data_width);

if (coarse_row < box_height && coarse_column < box_width) {

  double spv = 0.0;

  for (int j = 0; j < row_ratio; j++) {
    for (int i = 0; i < column_ratio; i++) {
      const int current_fine_index = fine_index + i + (j *
  fine_data_width);
      spv += fine_data[current_fine_index]*Vf;
    }
  }

  coarse_data[coarse_index] = spv/Vc;
}
```

Figure 6.9: Code listing for the data-parallel volume-weighted coarsen kernel.

6.3.1 Validation

To verify the accuracy of the results produced by the GPU-based version of CleverLeaf, we perform a comparison between the results produced by the CPU version and those produced by the GPU version. Throughout the development of the GPU-based version of CleverLeaf, we compared each array to the corresponding array from a calculation run on the CPU. By ensuring that these numbers were the same (within machine accuracy) we can be confident that the codes produced the same results without needing to run additional validation tests such as the Sod or Woodward-Colella problems.

6.3.2 Verification

The validation approach used in Chapter 5 to compare CleverLeaf and Shamrock is no longer applicable since we are running on a different architecture. Correlating the computational behaviour of the CUDA version of CleverLeaf

```
DO k = y_min, y_max
  DO j = x_min, x_max
    v = 1.0 / density(j,k)
    pressure(j,k) = (1.4 - 1.0) * density(j,k) * energy(j,k)
    pressurebyenergy = (1.4 - 1.0) * density(j,k)
    pressurebyvolume = -density(j,k) * pressure(j,k)
    sound_speed_squared = v * v * (pressure(j,k) *
    pressurebyenergy - pressurebyvolume)
    soundspeed(j,k) = SQRT(sound_speed_squared)
  ENDDO
ENDDO
```

(a) Fortran equation of state kernel.

```
if (row < box_height && column < box_width) {
  v = 1.0 / density[i];
  pressure[i] = (1.4-1.0) * density[i] * energy[i];
  pressure_by_energy = (1.4-1.0) * density[i];
  pressure_by_volume = -1.0 * density[i] * pressure[i];
  sound_speed_squared = v * v * (pressure[i] *
    pressure_by_energy - pressure_by_volume);
  sound_speed[i] = sqrt(sound_speed_squared);
}
```

(b) CUDA equation of state kernel.

Figure 6.10: Code listings for the original Fortran and data-parallel CUDA equation of state kernels.

and the CPU-based Shamrock benchmark would not be a useful result. Using two different architectures means that even if we ignore the fundamental performance gap, the performance counters present on each architecture will not match. This prevents us from using the Veritas framework to perform any analysis. Whilst the question of correlating performance across architectures remains open, we address the issue of verifying the GPU-based version of CleverLeaf using three observations.

First, having already verified the CPU-based version of CleverLeaf with the Shamrock benchmark, we are confident that the fundamental algorithms expressed in CleverLeaf capture the key computational behaviours of Shamrock. The GPU-based version of CleverLeaf contains these same algorithms, expressed identically in most cases. Hence, we are confident that the performance characteristics of CleverLeaf executing on a GPU would match the characteristics of Shamrock if such a comparison could be performed.

Second, when an algorithm is modified to perform better on the highly-parallel architecture of the GPU, we ensure that the output of the algorithm doesn't change. This is evident in the validation results, where we observe the same solutions for both of the two test problems. The changes made to make an algorithm data-parallel and more suitable for the GPU would also need to be made in Shamrock. Hence, if the algorithmic changes were to alter the performance characteristics of the application, both applications should change in the same way and CleverLeaf would remain representative of the Shamrock benchmark.

Figure 6.10 contains the original and data-parallel versions of the equation of state kernel from CleverLeaf. The only modifications are the slight syntax changes (for CUDA) and the use of an if statement, rather than a for loop. This is because one instance of the kernel will be launched for each cell, corresponding to one kernel launch per loop iteration. As a second example, consider the code in Figure 6.11. This is the algorithm used to interpolate node-centred coarse data to the next finer level. In the original code, unnecessary dependencies exist between iterations of the outer loops. By collapsing the loops we remove these dependencies, but still perform identical operations, ensuring the output of the algorithm is the same.

Finally, it is important to revisit the goals of mini-applications: they are key tools for investigating new programming languages [87, 95] and testing new machines and architectures [15]. Ensuring the representativeness of mini-applications whilst also rapidly prototyping versions of the application in new programming languages (and new architectures) is a balancing act. Given the uncertainty surrounding future parallel architectures, we are confident that development of novel versions is more important that being hampered by producing an application with identical computational behaviour in all cases.

```
realrat0=one/dble(ratio(0))
realrat1=one/dble(ratio(1))

do ic1=ifirstc1,ilastc1
  if1=ic1*ratio(1)
  do ir1=0,ratio(1)
    ie1=if1+ir1
    if ((ie1.ge.filo1).and.(ie1.le.(fihi1+1))) then
       do ic0=ifirstc0,ilastc0
         if0=ic0*ratio(0)
         do ir0=0,ratio(0)
           ie0=if0+ir0
           if ((ie0.ge.filo0).and.(ie0.le.(fihi0+1))) then
             x = dble(ir0)*realrat0
             y = dble(ir1)*realrat1
             arrayf(ie0,ie1)= (arrayc(ic0,ic1)*(one-x) +
               & arrayc(ic0+1,ic1)*x)*(one-y) +
               & (arrayc(ic0,ic1+1)*(one-x) +
               & arrayc(ic0+1,ic1+1)*x)*y
           endif
         end do
       end do
    endif
  end do
end do
```

(a) Origan Fortran linear refine kernel.

```
if (column < fine_box_width && row < fine_box_height) {

  const double realrat0 = 1.0/static_cast<double>(ratio.x);
  const double realrat1 = 1.0/static_cast<double>(ratio.y);

  const int ic0 = floor(column/static_cast<double>(ratio.x));
  const int ic1 = floor(row/static_cast<double>(ratio.y));

  const int ir0 = column - (ic0*ratio.x);
  const int ir1 = row - (ic1*ratio.y);

  const double x = ir0*realrat0;
  const double y = ir1*realrat1;

  const int coarse_index = (coarse_data_offset+ic0) + (ic1*
    coarse_data_width);

  arrayf[fine_index] = (arrayc[coarse_index]*(1.0-x) + arrayc[
    coarse_index+1]*x)*(1.0-y)
    + (arrayc[coarse_index+coarse_data_width]*(1.0-x) + arrayc[
    coarse_index+1+coarse_data_width]*x)*y;
}
```

(b) Data-parallel CUDA linear refine kernel.

Figure 6.11: Code listings for the original and data-parallel linear refine kernels.

	IPA	Titan
Processor	Intel Xeon E5-2670	AMD Opteron 6274
Clock	2.6 GHz	2.2 GHz
Accelerator	NVIDIA Tesla K20x	NVIDIA Tesla K20x
PCI gen	2	3
Nodes	8	18,688
CPUs/node	2×8 cores	1×16 cores
GPUs/node	2	1
CPU RAM/node	128 Gb	32 Gb
GPU RAM/node	6 Gb	6 Gb
Interconnect	Mellanox FDR Infiniband	Cray Gemini
Compiler	Intel 13.1.163	Intel 13.1.3.192
MPI	MVAPICH 1.9	Cray MPT
CUDA Version	5.5	5.5

Table 6.1: IPA and Titan: hardware and software configurations.

6.4 Performance

To assess the performance and scalability of our implementation we performed a series of experiments using two different architectures: the IPA testbed machine at Lawrence Livermore National Laboratory and the Titan supercomputer at Oak Ridge National Laboratory. The hardware and software configuration of each platform is detailed in Table 6.1. The experiments use a range of problem sizes and node counts, and are designed to test both single-GPU performance and parallel scalability. Full results for each study are presented in Appendix A.

Our first study compares a single NVIDIA Kepler K20x GPU to one node (16 cores) of dual-socket Intel Xeon E5–2670 CPU. We use the Sod problem described in Chapter 3 and run 1 thousand timesteps at a range of coarse resolutions from 3 thousand to over 6 million zones, using 3 levels of refinement and a refinement ratio of 2. Figure 6.12 contains the results of this experiment. At small problem sizes the GPU and CPU performance are similar, however, at large problem sizes, we see a performance improvement of over $2.6\times$. This performance improvement at larger problem sizes is typical of the throughput-oriented GPU architecture [56].

The second performance experiment investigates the scalability of our code as the number of GPUs is increased from 2 to 16 (1 to 8 nodes). We also

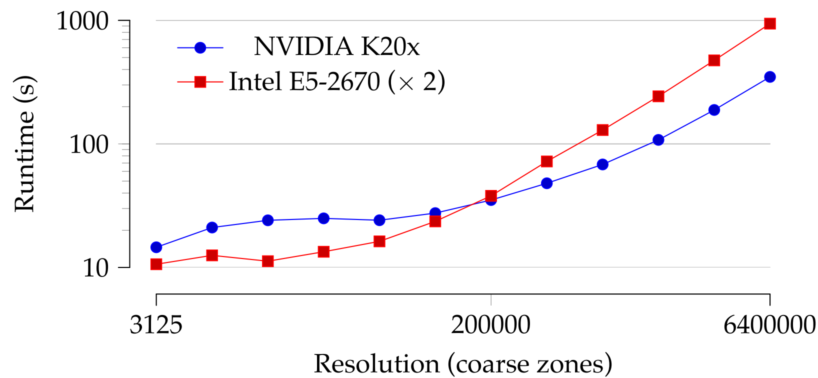


Figure 6.12: Wall-clock time for increasing problem size on a single GPU vs. dual-socket CPUs.

include equivalent results for the CPU-based code. The experiment is a strong-scaling study, where the problem size remains constant as the number of GPUs (or nodes) is increased. We use the 6.4 million zone Sod problem and run for 1 thousand timesteps. The results of this experiment are detailed in Figure 6.13, and for all node counts, the performance of the GPU-based code is better than the GPU-based code. For a single node, with two GPUs compared against two CPUs (16 cores), the GPUs are 4.87× faster. At eight nodes (16 GPUs vs. 128 cores) the GPU-based code is still 1.92× faster. We attribute this reduction in performance to the data transfer required during the boundary exchanges and the regridding phase beginning to dominate the simulation runtime; a consequence of running our experiment as a strong-scaling study that echoes the predictions of Amdahl's law [6].

Our third experiment investigates the performance of our code at large scale, running on over four thousand GPUs on the Titan system at Oak Ridge National Laboratory. This experiment is a weak-scaling study, where the problem size is increased as the number of GPUs is increased. In theory, this means that each GPU will have a constant amount of work, and any costs associated with using an increasing number of nodes will be highlighted. We use a modified version of the triple point shock interaction problem presented in [55]. A rectangular domain is split into three regions, and as the simulation progresses

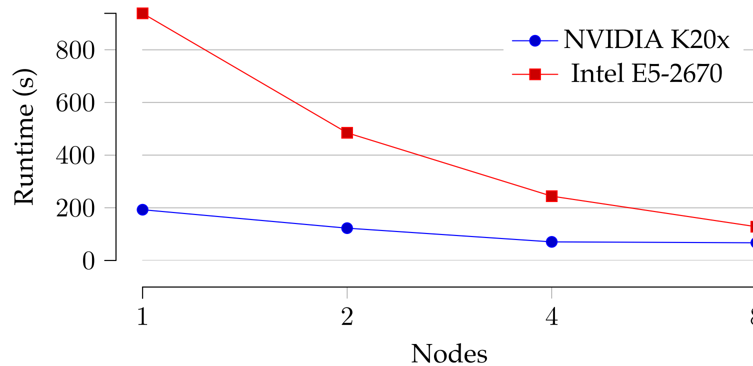
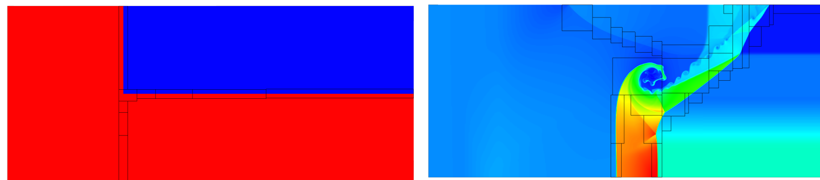


Figure 6.13: Wall-clock time for strong-scaling runs on up to 8 GPUs.



(a) ρ and finest level patches at $t = 0$. (b) ρ and finest level patches at $t = 2.5$.

Figure 6.14: Initial and final density and patch configuration for the triple-point shock interaction problem.

from its initial state a strong shock travels from left to right. This shock generates a large amount of vorticity and creates a complex area of interest, creating a large number of patches that are shown as black lines in Figure 6.14.

We run at seven different node counts, from 1 to 4,096; we use effective (finest-level) resolutions from 2 million to over 8 billion cells with 3 levels of refinement and a refinement ratio of 2. Weak scaling an AMR problem can be difficult since keeping the computational work per-GPU the same is hard. In this experiment we increase only the coarse resolution and always run to the same physical end time regardless of the number of timesteps required. Figure 6.15 presents our results, normalised as average grind times per-cell for each node count. Each component of simulation runtime gradually increases as more nodes are added, however, we are able to run the problem on over four thousand nodes. It is also interesting to note that the majority of the simulation runtime is spent in the communications of the application (including MPI trans-

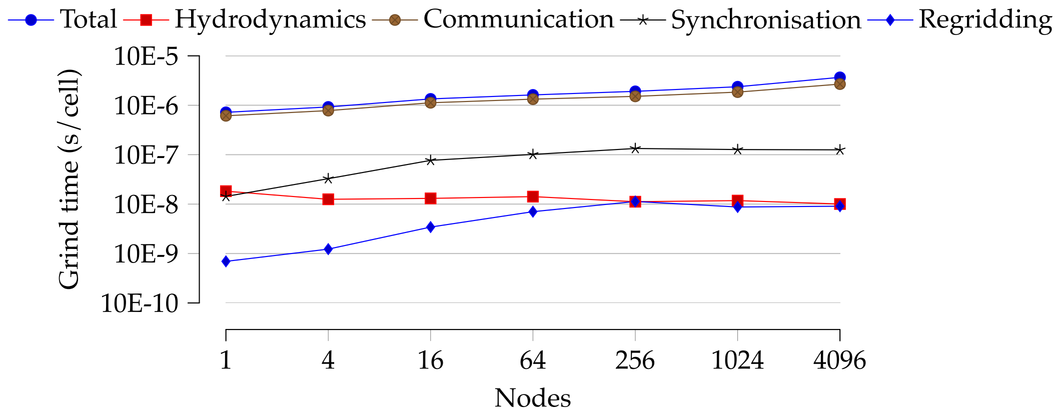


Figure 6.15: Weak-scaling performance analysis on Titan.

fer, local data transfer, and transfer between host and device memory). The AMR-specific runtime components, regridding and synchronisation, comprise only a fraction of the overall runtime. Specifically, at 4,096 nodes 44% of the runtime is spent advancing the simulation, although the majority of this time is spent transferring data between patches both locally and globally. Calculating the timestep, which contains the only global reduction operation consumes 6% of the runtime. Synchronising fine data to the coarser levels takes an average of 3% of the runtime. In contrast, on a single node 59% of the runtime is spent advancing the simulation, with only 1% of time spent synchronising levels, and less than 1% calculating the global timestep.

6.5 Summary

In this chapter, we describe the GPU-based implementation of CleverLeaf, the first open-source block-structured adaptive mesh refinement mini-application that will run residently on GPUs. In particular, we describe the algorithms, data structures and simulation code we have developed that allow us to fully utilise the highly parallel architecture of a GPU. The common data structures and algorithms are contained within a reusable software module, and can be incorporated into any code using SAMRAI with a Cartesian geometry. The

data structures we have developed store simulation data in the GPU memory at all times, and the necessary coarsen and refine algorithms have been written that allow simulation data to be coarsened and refined in parallel between two patches in GPU memory.

We validate the mathematical accuracy of the GPU-based version of CleverLeaf by comparing the results to the CPU-based version of the code. The results are identical to machine precision and allow us to assert that the adaptive mesh refinement algorithms have been correctly applied to advance the simulation. Whilst we cannot directly compare CleverLeaf and Shamrock for GPU architectures, we present a reasoned argument as to the continued representativeness of our mini-application.

Finally, we present a performance analysis of CleverLeaf. The application is up to 4.87× faster than the CPU-based code, and we have demonstrated scalable performance on up to 4,096 GPUs on the Titan system at Oak Ridge National Laboratory.

CHAPTER 7

Improving AMR Parameter Selection on Contemporary Compute Platforms

The dynamic nature of Adaptive Mesh Refinement (AMR) can be controlled and constrained through a number of parameters provided at simulation runtime. Selecting appropriate parameter values for the mesh creation and integration algorithms is critical in determining the runtime of applications using AMR. The ratio by which the mesh is refined, the total number of levels of refinement, and the frequency with which the refined area is moved in order to cover the areas of interest in the problem can all be parameterised and controlled. Changing these parameters can affect both simulation runtime and solution accuracy, and can also have a large impact on resource usage.

As an example of this, consider the frequency with which the refined area is moved. Moving the refined area ensures that areas of interest in the problem are accurately tracked. However, this step requires a large amount of data movement and communication, making it a costly procedure. By moving the refined area less often the communication and data movement costs are

reduced. However, this requires a larger area to be refined each step, increasing the computational cost. These tradeoffs are typically only understood intuitively, and parameter settings often use default values, even across multiple generations of different supercomputer architectures.

In this chapter we use CleverLeaf to investigate the impact of three different parameters: regridding frequency, maximum level number, and refinement ratio; on simulation runtime. Identifying the most effective application settings and the most appropriate parameter assignment for a given supercomputer architecture provides a means of improving the performance and productivity of production AMR codes. Additionally, we can use this research to develop a pathway for the software development of future codes.

We conduct a range of experiments on three of the AMR configuration parameters exposed by CleverLeaf, and analyse the results to determine which parameter values offer the best performance on each of the three architectures: an InfiniBand cluster, an IBM Blue Gene/Q, and a Cray XC30. Using data collected from CleverLeaf we then infer the impact of optimal parameter selection on production applications. In particular, we show how job throughput can be improved by up to 82 % and application memory usage and data generation can be reduced by up to 32 % through the use of appropriate parameter settings.

7.1 Application Parameters

The parameters that control AMR problems have a large degree of freedom, and a small change to the configuration of an application can have a large impact on the runtime of the simulation. To fully understand the impact of each parameter in the context of both a particular problem and machine architecture, we must perform a large number of runs to search some portion of the parameter space. Performing these experiments with a production AMR application can be both difficult and expensive. The complex physics takes a large amount of runtime, and sometimes the desired parameter might not be modifiable with-

Hierarchy Parameters	Description
Minimum patch size	Constraint on minimum patch size.
Maximum patch size	Constraint on maximum patch size.
Number of levels	Maximum number of levels in AMR hierarchy.
Refinement ratio	Increase in resolution between levels.
Patch efficiency	Desired ratio of flagged to unflagged cells in a patch.
Combine efficiency	Desired ratio of flagged to unflagged cells if joining two existing patches.
Regridding frequency	Number of steps between each regridding operation.

Physics Parameters	Description
Physics weighting	Factor by which to increase the physics cost using redundant work.
Viscosity threshold	Used to tag cells based on the artificial viscosity value.
Energy threshold	Used to tag cells based on the energy gradient.
Density threshold	Used to tag cells based on the density gradient.

Table 7.1: Parameters exposed by CleverLeaf.

out extensive modifications to the application. By developing CleverLeaf as a lightweight proxy of the physics and management code found in a production AMR application, we have a platform for experiments that is easy to understand and modify.

Through a combination of configurable parameters in SAMRAI objects and CleverLeaf, we have a comprehensive set of runtime options that can be modified. These modifications can be observed and the runtime of the program measured, providing useful feedback in the path to improving production AMR codes.

CleverLeaf exposes two different types of parameters, which we call the hierarchy parameters and the physics parameters. The hierarchy parameters include the number of levels and the refinement ratio used between each level. The physics parameters include the values used during the heuristic tagging function, as well as the cost of the physics kernels. Making the physics cost of the application tuneable means that it can be used to more accurately represent a target production application, and more accurately mimic the balance between the time spent advancing the simulation, and the time spent managing components of the AMR algorithm. The full list of parameters is presented

in Table 7.1.

The physics parameters exposed by CleverLeaf are all specific to its hydrodynamics scheme. The physics weighting is a multiplier that will increase the time spent performing computational work. This alters the ratio of communication to computation, so that we can tune the balance to more accurate represent a production workload. The three threshold parameters are used to control which cells are tagged for refinement during execution. The values will affect the accuracy of the simulation, since they define the areas that will be simulated at a higher resolution. The number of cells that are refined will also affect the runtime of the simulation, since as the cell count increases so does the amount of work.

The hierarchy parameters exposed by CleverLeaf belong to the SAMRAI library. The minimum and maximum patch size constraints provide a hard upper and lower bound on the size of every patch, even if satisfying these constraints means breaking others. The number of levels, refinement ratio, and the regridding frequency all have a more complex interaction with simulation accuracy and performance. Altering the number of levels or the refinement ratio will change both simulation runtime and accuracy. Changing the regridding frequency should not affect simulation accuracy, but will affect simulation runtime due to the changing cell count. In Section 7.4.2 we investigate the impact of three of these parameters on application performance.

7.2 Related Work

The parameters exposed by AMR provide users and developers with many options for improving application performance. Colella et al. identify two general areas of work for improving AMR performance: first, algorithmic improvements that aim to reduce the complexity of particular parts of the AMR management routines; and second, configuration improvements that aim to reduce runtime through parameter selection [34]. The improvements we present in

this chapter belong to the second area.

Gunney et al. present a parallel clustering algorithm that is up to 40× faster than a simple parallelisation of the standard Berger-Rigoutsos clustering algorithm [24, 65]. Allowing each possible patch to be evaluated as a parallel task, and assigning each task an owner to manage the evaluation of the patch, means that the clustering work can be shared between processors. At small scale, the difference between clustering algorithms is negligible, but the simple parallel Berger-Rigoutsos algorithm will always be limited by the serialisation caused by having one root processor evaluate every patch.

Other clustering approaches also demonstrate improved scalability (over Berger-Rigoutsos), but do so at the cost of additional constraints on patch sizes and locations. Luitjens and Berzins present five parallel regridding algorithms, two of which constrain the generated patches [99]. Their approach uses a set of predefined patches (tiles) that can be enabled if they contain a tagged cell. Since all possible patches are predefined, they are more likely to contain untagged cells. Their local Berger-Rigoutsos approach uses an instance of the Berger-Rigoutsos algorithm on each patch. This means that new fine patches cannot span the boundary of two coarser patches, and hence increase the total number of patches. By avoiding global communications, both these algorithms have improved scalability. This work shows the holistic approach required to tune AMR performance, since improving the regridding algorithm creates less favourable patch configurations.

Configuration improvements can typically be made at run-time, and include user-supplied patch and level constraints, refinement ratios, and load balancing parameters. Load balancing determines which processor a patch gets assigned to, and load balancers typically focus on sharing the computational load generated by each patch. However, considering the communication cost of patch placement can improve application scalability. In previous work, we have used performance modelling techniques to determine, at run-time, the cost of placing a patch on a particular processor [17]. Considering communica-

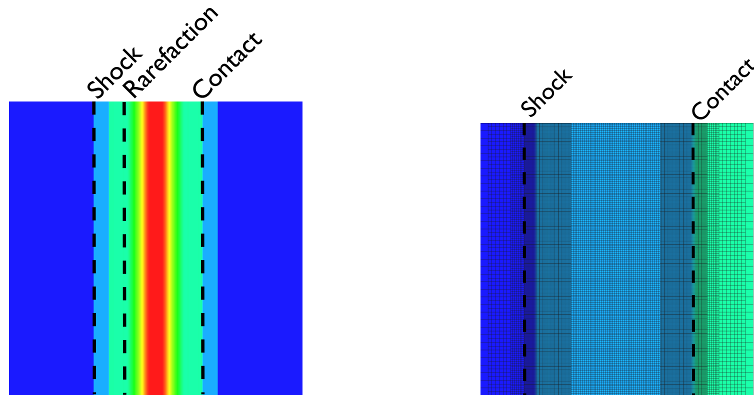
tion cost during load balancing provided runtime improvements of up to 29% compared to a standard, bin-packing approach.

Lan et al. use a dynamic load balancing technique with a two-phase approach, first transferring patches from overloaded to underloaded processors, and then splitting patches to further fine-tune the load balancing [93]. With this technique they are able to reduce application runtime by up to 47%. Again, we see that modifications to one part of the AMR configuration impact other areas. Patches must be split in order to balance the load; this means that more, smaller, patches must be used, which increases communication overheads.

At present, there is a lack of literature that investigates the fundamental parameters that control AMR performance. When AMR is used, it is typically applied as an enabling technique for a given numerical simulation; any parameter studies focus on physical parameters such as initial conditions or refinement criterion, rather than the basic parameters that control the AMR algorithm [36, 80, 117]. Additionally, such studies are not concerned with application performance. In [80], Hummels and Bryan do vary the maximum level number to reach a desired mesh resolution, however, no performance results are presented. The three parameters we have investigated have a large impact on performance, and for two of the parameters, the optimal values were considerably different to the established default configuration.

7.3 Experimental Setup

Throughout our experiments we use a simple two-dimensional problem that is a modified version of Sod's shock tube problem [147]. A vertical strip of ideal gas with density $\rho = 1$ and pressure $p = 1$ is centred in an ambient region of ideal gas with $\rho = 0.125$ and $p = 0.1$. At time $t > 0$ the strip of high-density gas expands into the ambient region, forming the shock wave seen in Figure 7.1a. The propagation of the shock wave provides three important solution features that the AMR tagging routines will capture: the contact discontinuity, the shock



(a) Solution features captured by AMR flagging routines: shock front, contact discontinuity, and the rarefaction.

(b) Increased mesh resolution around the shock front and contact discontinuity.

Figure 7.1: AMR provides increased accuracy by increasing the resolution around areas of interest.

front, and the rarefaction. The simulation is advanced to time $t = 2.5$.

The execution time of an AMR-enabled simulation depends on both problem size and a number of configuration parameters including the number of AMR levels, the minimum and maximum patch sizes, the refinement criteria, and the frequency of regridding. In order to reduce the impact of all these factors, we use a carefully constructed problem for each core count to approximate a weak-scaling configuration.

When running an AMR simulation, we ensure that the resolution of the finest level is equal to the resolution of the uniform calculation, where a uniform calculation is one without mesh refinement. In order to weak-scale a uniform problem, the global number of cells is increased. However, with an AMR problem, increasing the global number of cells (at the coarsest level) will result in a smaller portion of the problem being refined and the amount of computational work decreasing as more processors are added.

To approximate weak-scaling behaviour with an AMR problem, we use replication scaling. Replication scaling creates a larger problem by taking a configuration for a fixed number of processors and making identical copies of

the problem for each additional set of processors. This approach was first used by Van Straalen et al. in their work on scalability optimisations for the Chombo framework [160]. Figure 7.2 shows the basic problem configuration used for: (a) a 16-core run on one node, and (b) a 256-core run on 16 nodes. The difference in coarse patch configuration is due to the tree-based load balancing algorithm provided by SAMRAI.

We use a uniform resolution of 800^2 cells per-processor, and four levels of refinement with a refinement ratio of two. This means that the resolution of level 0 is 100^2 per processor. It is important to note here that not all of the 10 thousand coarse cells on each processor will be refined, so the total number of cells assigned to each processor when using AMR will be less than the uniform case. Other factors that will affect the total number of cells in the problem include patch size constraints, flagging criteria, and load balancing parameters. To flag cells for refinement, we used the following criteria:

$$\nabla\rho > 0.001 \tag{7.1}$$

$$\nabla E > 0.001 \tag{7.2}$$

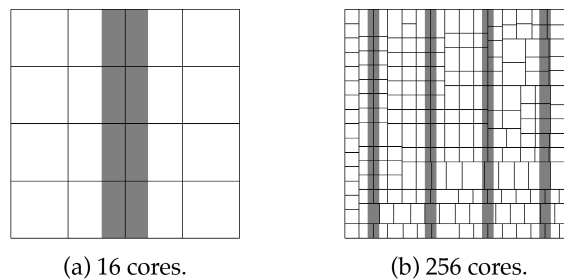
$$q > 0.1 \tag{7.3}$$

Cells in which at least one of the criterion is true are flagged for refinement. The minimum patch size constraint was set to 50^2, and the default load balancing parameters were used. As in [23], regridding is performed every four steps.

Our experiments were conducted on three machines, selected due to their differing characteristics: Cab, Vulcan, and ARCHER. Lawrence Livermore National Laboratory's Cab supercomputer is composed of 1,296 dual socket nodes based on 8-core Sandy Bridge processors. Nodes are connected via QDR InfiniBand in a fat-tree topology. Vulcan is an IBM Blue Gene/Q also installed at Lawrence Livermore National Laboratory. Vulcan has 24,576 nodes each containing a 16-core PowerPC processor. The nodes are connected using a five-dimensional torus. ARCHER is a Cray XC30 supercomputer containing 3,008

	Cab	Vulcan	ARCHER
Processor	Intel Xeon E-5 2670	PowerPC A2	Intel Xeon E-5 2697
Clock	2.6 GHz	1.6 GHz	2.7 GHz
Nodes	1,296	24,576	3,008
CPUs/node	2×8 (16) cores	16 cores	2×12 (24) cores
RAM/node	32 GB	16 GB	64 GB
Interconnect	InfiniBand Fat-Tree	5D Proprietary Torus	Cray Aries
Compiler	Intel 13.1.1.163	IBM XL 12.1	Intel 13.1.3.192
MPI	MVAPICH2 1.7	IBM MPI	Cray MPT

Table 7.2: Cab, Vulcan, and ARCHER: hardware and software configurations.



(a) 16 cores. (b) 256 cores.

Figure 7.2: Initial conditions and coarse patch configuration for the replication-scaled Sod problem.

compute nodes, managed by the Engineering and Physical Sciences Research Council at the University of Edinburgh. Each node contains two 12-core Ivy Bridge processors. The ARCHER nodes are connected via Cray's Aries interconnect in a dragonfly topology. Further details on hardware and software configuration of the three machines can be found in Table 7.2. When running our experiments one MPI task is allocated to each processor core for all architectures. On Vulcan, each MPI task also uses four OpenMP threads in order to take full advantage of the four floating point units in each PowerPC core.

7.4 Performance Analysis

To investigate the impact of AMR parameters on application performance we conduct two sets of experiments. First, we conduct a performance analysis of CleverLeaf using a default set of parameters. We compare these performance results to those obtained running an identical simulation in CloverLeaf.

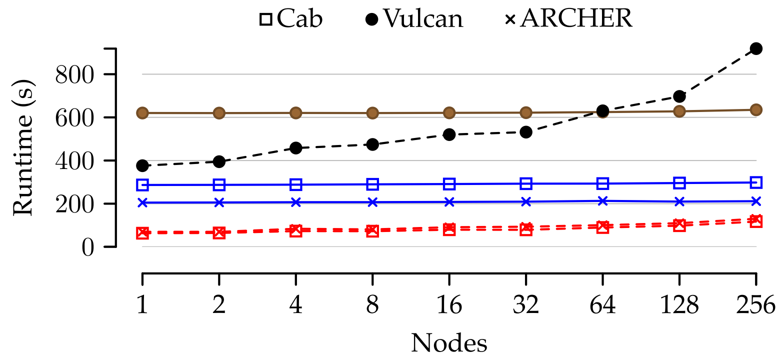


Figure 7.3: AMR (- - - ·) and uniform (——) replication scaling on Cab, Vulcan, and ARCHER.

These results highlight the differences in performance that can be expected when switching from a uniform to an adaptive application. The second set of experiments varies the value of three parameters in turn: regridding frequency, number of levels, and refinement ratio. Analysing the results of these experiments allows us to compare the impact of each parameter on simulation runtime, as well as identify the appropriate parameter values for the three experimental platforms we use.

7.4.1 Scalability

In this section we compare the initial scaling performance of CleverLeaf to CloverLeaf (the uniform mini-application described in Chapter 4) on our three test platforms. We ran the replication-scaled problem on a range of core counts—from 16 to 6,144—on each of the three architectures.

Figure 7.3 shows that, in general, across all three architectures the performance of CleverLeaf (AMR) is better than that of CloverLeaf (uniform). In particular, kernel runtime is reduced by up to a factor of eight[1]. This is due to the capability of AMR to reduce the total number of cells in the problem, whilst still maintaining the desired mesh resolution in areas of interest.

In the uniform calculation, the total number of cells is 10 million per-core.

[1]Full results are presented in Tables A.7a, A.7b and A.7c.

In the AMR calculation, the initial total number of cells is 720,000 per-core. Due to the mesh evolving throughout the problem, this cell count grows to a maximum of 1,350,400 at step 503. The average total cell count (for one node) is 1,158,730, so each processor is responsible for an average of 72,792 cells. The difference in cell count, assuming the problem is well balanced, highlights the strength of AMR in reducing computational work; the total number of cells in the adaptive mesh is an order of magnitude lower than that of the uniform mesh.

The reduced number of cells in the AMR problem also provides a reduction in memory usage, with the memory high watermark (measured using WMTrace [131]) reduced by 20 % from 123 MB to 99 MB when running on 16 cores. Whilst the 90 % decrease in cell count would suggest that more memory should be saved, CleverLeaf has additional overhead such as the C++ objects used to manage the adaptive hierarchy. Each patch requires its own set of ghost cells to store boundary values; this also increases memory usage.

For three core counts (1,024, 2,048, and 4,096) on the Blue Gene/Q, CleverLeaf performs worse than CloverLeaf. Despite good initial performance, where the AMR calculation is almost as fast as the uniform calculation on a much faster Intel-based cluster, the weak scaling performance is not maintained. On 1,024 processor cores the runtime of the two programs is almost identical, and on 2,048 and 4,096 cores, the performance of CleverLeaf is up to $1.4\times$ slower than CloverLeaf.

The trend common to all three architectures is the significant reduction in computation time: 91 % on Cab, 87 % on Vulcan, and 87 % on ARCHER. With the replication-scaled problem, the kernel time remains similar regardless of core count. As is typical, the cost of boundary communication increases with the number of cores. In CleverLeaf, the cost of boundary communications also contains any busy waiting caused by a load imbalance between the processors.

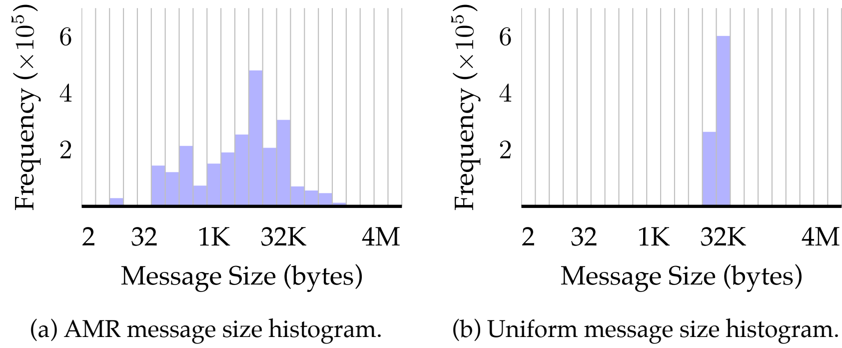
Across all architectures, regridding is a significant cost as the scale of the application is increased. At 4,096 processor cores, regridding takes 41 % and

73 % of the total runtime on Cab and Vulcan respectively. This large regridding cost is the cause of the poor scalability exhibited when running CleverLeaf on the Blue Gene/Q architecture. At 6,144 processor cores, regridding takes 51 % of the total runtime on ARCHER.

The cost of the synchronisation step of the AMR algorithm, where the more accurate fine solution is mapped to the coarse levels is insignificant on all architectures, even as the scale of the application runs is increased. Even though this synchronisation procedure is performed every timestep, it takes as little as 1.2 % of the total simulation runtime.

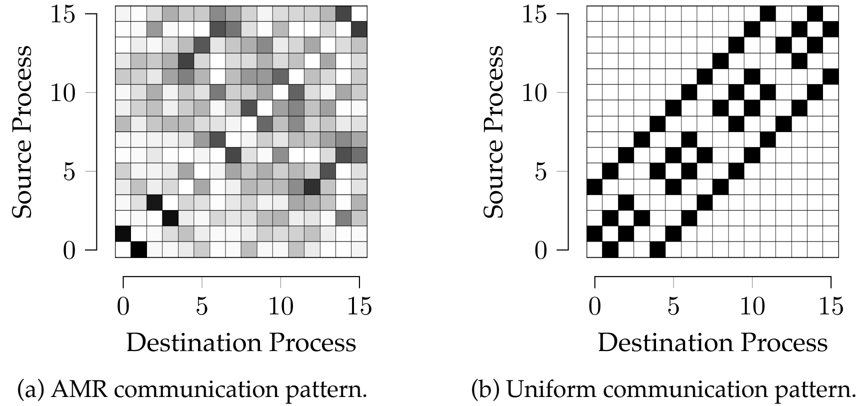
The other costs captured in Tables A.7a, A.7b and A.7c represent the time taken for application initialisation, program logic, and—in CleverLeaf—object management. CloverLeaf uses only scalar values and two-dimensional arrays, minimising the time spent outside of actual simulation code. Each processor is assigned one chunk of the simulation mesh which is created once and exists throughout the simulation. In contrast, CleverLeaf uses `Patch` objects that are created and destroyed throughout the simulation, resulting in more management time.

Whilst the time spent updating boundaries in CleverLeaf is small, it is in most cases an order of magnitude more than the time spent updating boundaries in CloverLeaf. This is a tradeoff that must be made when using AMR, since the additional data movement and coordination required will create more communication. Figures 7.4 and 7.5 show the message size histogram, and communication pattern for the 16-core run of both CleverLeaf and CloverLeaf. These results confirm the complicated communication requirements of AMR, which we believe cause the increased communication overhead that we observe. CleverLeaf sends over two million messages, with sizes ranging from 4 B to 8 MB. CloverLeaf exhibits a much more regular communication pattern, both in terms of message size and sender-receiver pairs. Over 800 thousand messages are sent, though only two sizes are used: 8 kB and 16 kB. Despite using the same hydrodynamics scheme and solving the same problem, the dif-



(a) AMR message size histogram. (b) Uniform message size histogram.

Figure 7.4: Message size histogram comparison for a 16-core run.



(a) AMR communication pattern. (b) Uniform communication pattern.

Figure 7.5: Communication pattern comparison for a 16-core run.

ference in communication pattern highlights how two applications can appear and perform differently at the machine level.

The set of experiments using our mini-applications in this section highlight the benefits of AMR, chiefly the reduced cell count. This improves runtime by reducing the computational workload and memory requirement of the simulation. The results presented here also show the cost of the extra management work that an AMR application requires: regridding, synchronisation, and the increased cost of boundary communications. This management cost is strongly linked to parameter values, so it can be controlled and often reduced. For example, reducing the regridding frequency means that the regridding operations will be performed less often and will make up a lower proportion of the total runtime. However, a decreased regridding frequency requires larger patches,

meaning an increase in boundary communication, synchronisation, and computation time. Finding the best set of parameters is key for achieving optimal application performance; using CleverLeaf, it is this question that we address next.

7.4.2 Parameter Evaluation

In this section we use CleverLeaf to investigate the impact of three parameters on application runtime on three contemporary compute platforms. The parameters: regridding frequency, number of levels, and refinement ratio, all affect runtime by changing the complexity of management work required to advance the AMR algorithm. As in Section 7.4.1, we use the replication-scaled Sod problem, with a finest resolution of 800^2 cells per-processor. Since changing the regridding frequency and refinement ratio can effect the results of the simulation, we impose some additional constraints: the timestep is fixed and the simulation is run for 926 steps to finish at time $t = 2.5$. In each set of experiments, we vary only one parameter. The others are assigned the default values used in the previous section.

Regridding Frequency

As we observed in Section 7.4, the regridding operation—when applied every four timesteps—represents a large portion of the simulation runtime. On Vulcan the cost of regridding at 4,096 processor cores was 73 % of the total runtime. By decreasing the frequency at which the regridding procedure is applied, this cost can be reduced. However, a reduction in regridding frequency requires a larger portion of the problem domain to be refined, increasing the cost of other areas of the application.

The hydrodynamics applications that CleverLeaf represents are used to model the movement of gases under extreme pressures and temperatures. In these conditions, shock waves form, and must be accurately modelled by the hydrodynamics scheme used. In mathematical terms, the shockwave repre-

sents a discontinuity in the continuous function describing the value of a quantity at various points in the mesh. Since a discontinuity in the discretised equations would have an infinite gradient, the shock must be smeared, such that it has a very large, but not infinite gradient. The AMR-specific impact of this condition is the constraint that a strong shock wave must not be allowed to pass through a level boundary. When this happens, errors are introduced into the numerical solution [21].

The rate at which the shock moves through the problem domain is controlled by the timestep and the CFL condition and hence a shock cannot pass through more than one cell in a timestep [37]. To stop a shock passing through a level boundary we must add an extra layer of cells around any cells flagged for refinement corresponding to the length of time until the next regridding operation. For example, in the previous section, each group of flagged cells had an additional four-cell layer added, since the regridding operation was applied every four steps.

We hypothesise that the extra border cells might increase simulation runtime in three ways. The additional cells increase the amount of the domain that is refined, thus increasing time spent in the computational kernels of the application. Refining a larger proportion of the domain also means that a larger amount of halo data will need to be communicated, increasing the time spent updating boundaries. Finally, a larger proportion of the domain being refined means that more data must be synchronised between levels every timestep.

When adjusting the regridding frequency, finding a balance between the reduced regridding cost and the increased computation and communication cost is essential in finding an optimal runtime configuration. We ran a range of experiments, varying both regridding frequency and processor core count. As in the previous section, the processor core counts range from 16 to 6,144. The regridding frequencies used ranged from 1 to 20. Both the refinement ratio and number of levels remain unchanged.

Figure 7.6 highlights the affect of regridding frequency on runtime. The

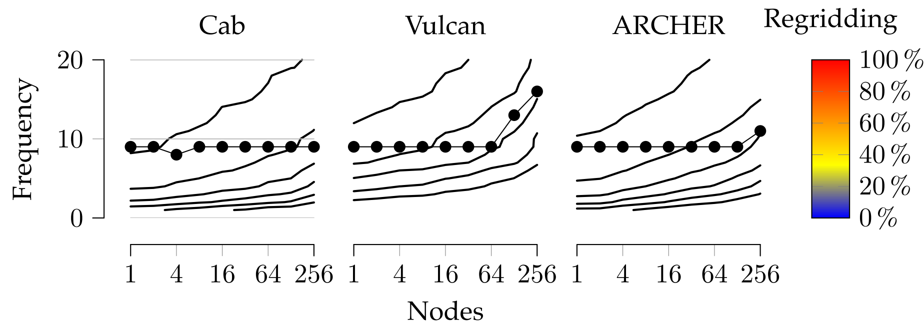


Figure 7.6: Regridding time and optimal frequency for each platform.

colour and contours of the plot show the percentage of time spent in regridding for a given frequency and core count. The points show the frequency that provided the minimum runtime at each core count.

We will examine the results first in terms of the percentage time spent regridding. On Cab, the time spent in regridding for the fastest runs is between 9 % and 24 %. On Vulcan, the regridding time increases to between 24 % and 37 % for the fastest runs. On ARCHER, the regridding time is between 11 % and 26 %. At increased core counts on Vulcan and ARCHER, the optimal regridding frequency begins to decrease. The link between percentage time spent regridding and the optimal regridding frequency suggests that frequency can be modified at higher core counts to provide better runtime.

Despite a regridding frequency of every nine steps being optimal for all core counts less than 1,024 on Vulcan and 3,072 on ARCHER, a decrease in frequency is necessary on higher core counts to maintain a reasonable percentage runtime spent regridding. Vulcan spends 38.2 % of the application runtime in regridding at 1,024 cores at a frequency of every nine steps. At 4,096 cores the optimal frequency decreases to every 16 steps, but the portion of runtime spent regridding remains similar at 37.9 %. ARCHER spends 26.47 % of the application runtime in regridding at 3,072 cores at a frequency of every nine steps. At 6,144 cores the optimal frequency decrease to every 11 steps, but the percentage of runtime spent regridding remains similar: 29.12 %. As is to be expected, there is a clear link between percentage of runtime spent regridding and the

optimal regridding frequency; CleverLeaf allows us to investigate this trade off with ease on our three contrasting architectures.

The regridding phase of the AMR algorithm requires a large amount of communication, so the cost of regridding increases proportionally as more cores are used. On Cab, the increase is from just over 18 % to over 43 % on 4,096 cores when using a regridding frequency of four. A lower regridding frequency thus improves application scalability. Regridding frequency can affect the time spent in the numerical kernels, since the larger patches required by the lower regridding frequencies mean more cells must be calculated. The time to fill boundaries also increases as the larger patches mean that more data must be transferred between processors in order to correctly fill any boundary cells. Finally, the synchronisation cost also increases as regridding frequency decreases, again as a result of an increased amount of data that must be transferred. All these runtime increases seem to directly relate to the amount of the problem domain that is refined. If this can be minimised, then runtime can be improved.

Number of Levels

The impact on runtime caused by the maximum number of levels used cannot be observed from the results already collected in Section 7.4.1. However, we suppose that the main cost of using additional levels will be the increased complexity in the structure of the patch hierarchy. This will increase the cost of regridding, as well as increasing the time spent performing boundary updates and synchronising the solution to the coarser levels.

Reducing the number of levels will remove the complexity introduced with a high maximum level count. However, with fewer levels, the resolution of the coarsest level will need to be increased to ensure that the finest level remains at an appropriately accurate resolution, this will increase the time spent in computational kernels.

To observe the impact of maximum level number on simulation runtime

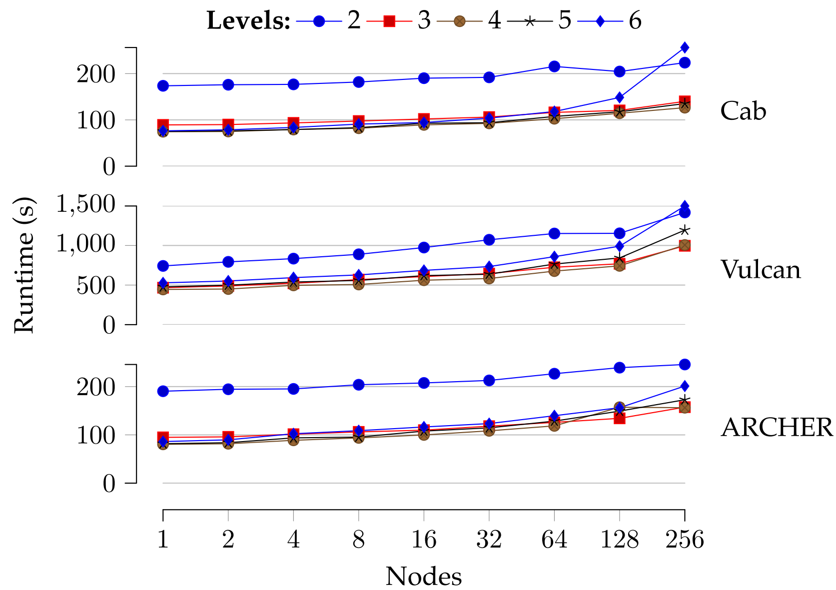


Figure 7.7: Impact of maxmimum level number on runtime.

we ran a range of experiments using our mini-application in which we varied both processor core count and maximum level number. As in previous experiments, the core counts range from 16 to 6,144. The maximum level ranges from two to six. The default values of four and two are used for the regridding frequency and refinement ratio. In order to maintain a finest level resolution of 800^2, we must vary the resolution of the coarsest level. The coarse resolutions used in each case are: 400^2, 200^2, 10^2, 50^2, and 25^2.

Looking at the runtimes presented in Figure 7.7 we can see that the level count does affect runtime in the way we predicted. Runtimes on all core counts begin to increase when more than five levels of AMR are used. On Cab and ARCHER, the increase in runtime due to the number of levels is only evident when six levels of refinement are used. On all three architectures, the best runtimes are recorded using either three or four levels of refinement.

Tables A.9a, A.9b and A.9c contain detailed timings for the 256-node experiments on Cab, Vulcan and ARCHER. We can see the impact of an increase in level count on application runtime. Considering the trends in this data, we can see that using either too few or too many levels can harm application per-

formance. With two levels of refinement, the large increase in coarse patch resolution means that there is too much computational work. Using three levels of refinement halves the amount of time spent doing computational work, but also reduces the boundary communication time. Adding levels adds complexity, and hence the synchronisation and regridding times also increase. The most significant increase in regridding time is 144 % on Cab.

Increasing the number of levels used decreases the amount time spent in the physics kernels. When using more and more levels, the low resolution of the coarse levels means that a smaller portion of the domain can be refined, and less cells need processing.

Refinement Ratio

The refinement ratio works in concert with the number of levels. Increasing the refinement ratio means that the resolution of the problem can increase more rapidly, reaching the desired resolution with fewer levels. Alternatively, the same number of levels can be used, and the resolution of the finest level can be increased. If the number of levels is kept constant, then the resolution of the coarse level can be reduced, and the total number of cells in the problem will decrease.

Provided that the number of levels and the finest level resolution are kept constant, we hypothesise that an increased refinement ratio could affect simulation runtime in two key ways. Firstly, the increased refinement ratio will increase the time taken to refine and coarsen data between levels. This will show up as an increase in the boundary update time and the synchronisation time. Secondly, the reduction in total cell count provided by an increased refinement ratio should cause a significant decrease in runtime.

To observe the impact of the refinement ratio on simulation runtime we ran a range of experiments in which we varied the number of nodes and the refinement ratio. As in all previous experiments the number of nodes ranges from 1 to 256, and the refinement ratio ranges from two to six. As in our default

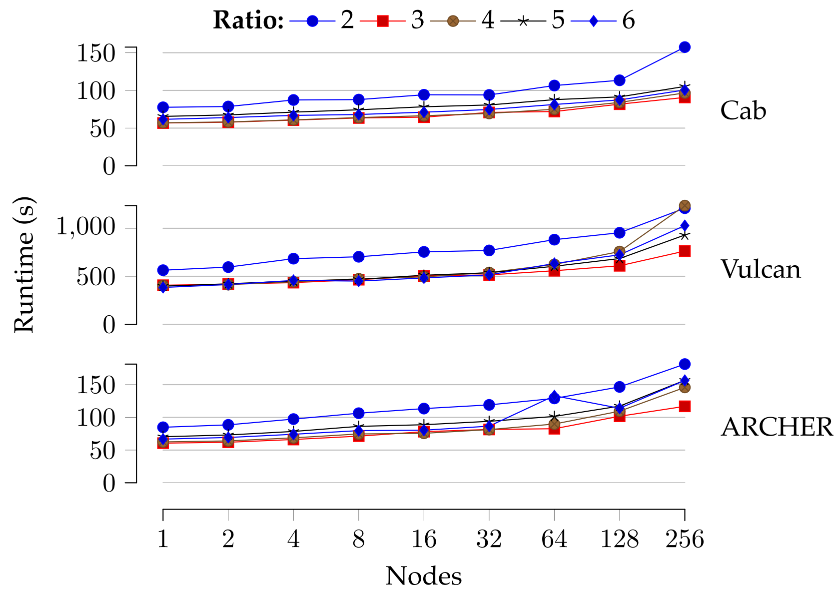


Figure 7.8: Impact of refinement ratio on runtime.

configuration, the regridding frequency used is four, and the maximum level number is four. By only changing one parameter we are able to better isolate the costs of the varied refinement ratios. We maintain a finest level resolution of approximately 800^2 by varying the coarse resolution. For example, when using a refinement ratio of five, the coarse resolution per-core will be 6^2, and the finest level resolution 750^2.

The runtimes presented in Figure 7.8 show the impact of refinement ratio on simulation runtime for all three platforms. As predicted, the decreased cell count provides a decrease in simulation runtime. In particular, on all three platforms a ratio of 3 provides the best performance. Increasing the refinement ratio from 2 to 3 but keeping the other parameters constant provides an improvement in runtime of up to $1.73\times$.

The decrease in total cell count is most easily observed by considering the time spent in the physics kernels. On Cab, ARCHER, and Vulcan the lowest kernel times are found at refinement ratios of 3 and 4, suggesting these ratios offer the lowest total cell counts. The reduction in kernel time when moving from a refinement ratio of 2 to 3 is up to 30 %. The time to exchange boundaries

on these two architectures begins to increase at refinement ratios of 5 and 6, suggesting that needing to interpolate more data from the coarse level is increasing runtime.

Despite our hypothesis, refinement ratio only has a small impact on synchronisation time, and additionally, synchronisation is only a small portion of simulation runtime. On Vulcan, the increase in synchronisation time is only noticeable with a refinement ratio of 6; this is not seen on Cab and ARCHER. We hypothesise that the faster CPUs in Cab and ARCHER, and the small cost of synchronising data (typically less than 1 % of runtime) mean that any increase in synchronisation cost caused by refinement ratio is dwarfed by other factors. On Vulcan, the increase in synchronisation time from a ratio of 3 to 6 is 35 %, however, this still represents only 2.5 % of the total simulation runtime. Given the low impact of synchronisation cost on runtime, the biggest impact caused by changing the refinement ratio is reducing the total number of cells in the problem, where reductions in kernel time of up to 30 % contribute to overall runtime improvements of up to 1.73×.

The experiments in Sections 7.4.2 to 7.4.2 show that parameter values have a significant impact on simulation. Often this impact is an improvement, however, bad parameter value selection can be incredibly harmful to simulation runtime. With a better value selected, all three parameters are able to improve the runtime of our test problem. Compared to the default configuration, the optimal regridding frequency provides a runtime reduction of up to 45 %, the optimal number of levels provides a runtime reduction of up to 1 %, and the optimal refinement ratio provides a reduction of up to 31 %. Conversely, with inappropriate values selected, we see increases in runtime of up to 186 %. Enforcing a better set of default values for these parameters and ensuring they are exposed for modification at runtime is essential for achieving optimal application performance; CleverLeaf provides a new tool for performing this analysis.

	Platform		
Parameter	Cab	Vulcan	ARCHER
Regrid	9	16	11
Levels	4	3	4
Ratio	3	3	3

(a) Optimal parameter values.

	Platform (% improvement)		
Parameter	Cab	Vulcan	ARCHER
Regrid	22.67	45.27	27.59
Levels	0	0.96	0
Ratio	31.24	24.21	25.58

(b) Achieved improvement.

Table 7.3: Optimal parameter values and achieved improvements over the default configuration, derived from our CleverLeaf mini-app study.

7.5 Parameter Selection for Production Applications

In this section we present an analysis of the most effective parameter values for each of our three experimental platforms, along with showing how improved parameter selection via mini-applications can be used to influence future application design and use. We perform a detailed analysis of the most effective values for the three parameters we examined in Section 7.4.2, highlighting the link between value selection and architecture characteristics. To show how improved parameter value selection can influence real-world applications we consider two important constraints in production use: simulation timesteps completed and memory used. Using the more effective values we have identified allows up to 33 more production-sized jobs to be completed per-month, a reduction of up to 32 % in memory usage, and a 45 % improvement in application runtime.

7.5.1 Optimal Parameter Configuration

Table 7.3a presents the optimal value for each parameter and each architecture, based on the results obtained in Section 7.4.2. The values chosen provide the best runtime at the largest scale on each architecture. The most significant change from the default configuration is the decreased regridding frequency. On all three platforms, the optimal regridding frequency is over double that of the default configuration. The best regridding frequency is strongly tied to

the number of nodes being used, and as seen in Section 7.4.2, the regridding frequency typically needs to decrease as the scale of the runs increases. The largest change necessary to find the optimal frequency is on Vulcan, where the slow cores of the Blue Gene/Q architecture mean that the algorithms in the regridding step have a big impact on performance.

The default configuration of four levels of refinement remains optimal at 256 nodes on both Cab and ARCHER. On Vulcan, the optimal number of levels to use is three; we attribute this to the reduced cost of regridding when the adaptive hierarchy only contains three levels. For all three architectures, a refinement ratio of three provides the best runtime. Using a higher refinement ratio reduces the total number of cells in the problem, reducing the time spent performing numerical calculations and thus reducing runtime.

Table 7.3b shows the improvement in runtime that each value provides over the default configuration of four levels, a refinement ratio of two, and a regridding frequency of every four steps. Selecting a better regridding frequency can improve runtime up to 45 %, and on Vulcan, using three levels of refinement improves runtime by 1 %. Using the optimal refinement ratio can provide runtime improvements of up to 31 %.

Based on our experiments, we conclude that regridding frequency and refinement ratio are the two most important parameters for ensuring scalable AMR performance. Our recommendation for future application development would be to select a larger default regridding frequency and to ensure that this parameter can easily be varied at run time. Refinement ratio can have a large impact on simulation accuracy as well as runtime, hence, our advice is to ensure that in all code development refinement ratio is left as a variable rather than a constant, allowing for easy extension of the code as desired by users.

7.5.2 Increasing Job Throughput

Mini-application runs typically last for minutes; the longest experiment in Section 7.4 takes just 15! In production, runs can last for days or weeks as thou-

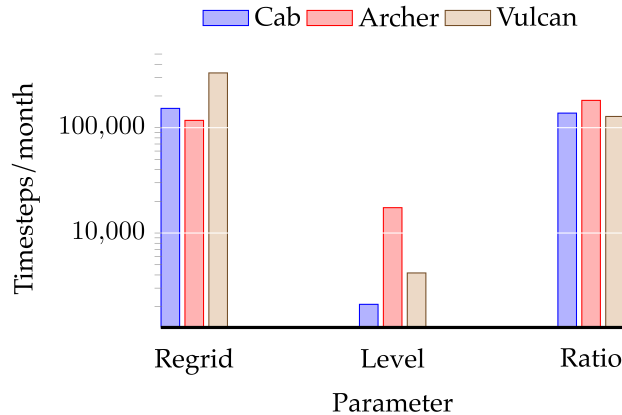


Figure 7.9: Predicted improvement in timesteps / month with optimal parameter selection, compared to the default configuration.

sands of timesteps are simulated on hundreds of processor cores. To frame the results from CleverLeaf in a way that is applicable to real-world application usage, we consider the number of timesteps that can be simulated in one month of wall clock time. We then examine the influence of the improved parameter values on this derived metric, showing the impact that a small percentage in performance improvement provides large real-world gains in terms of job throughput.

A real world problem might take 18 hours to complete 10 thousand simulation timesteps [130]. Scaling the results obtained in the previous sections in line with this more realistic runtime, we let the default configuration perform at a rate of 10,000 timesteps per 18 hours on each architecture (or 400,000 timesteps per month). Figure 7.9 presents the improvements in timesteps per month that each of the three parameter values provides compared to the default configuration. By keeping the baseline timesteps per month for each architecture the same we can easily identify which of the parameter changes is the most effective on each architecture.

The most significant improvement in throughput is on Vulcan when the regridding frequency is optimised: an 82 % improvement from 400,000 to 730,974 timesteps per month. This improvement in runtime would allow an

additional 33 simulations to be completed per month. On all architectures an improved regridding frequency provides a large improvement in throughput. On Cab, 38 % more timesteps per month are completed, and on ARCHER 29 % more timesteps can be completed. This corresponds to an additional 15 and 11 jobs per month respectively.

In our experiments we found the optimal number of levels on Cab and ARCHER to be four, the same as the default configuration, hence there is no improvement in throughput. When running on Vulcan, the optimal number of levels was three. This offers a small improvement in timesteps per month, meaning an additional 5,000 timesteps can be completed every 30 days. Over the course of a year, this would allow an additional six simulations to be completed.

Improving the refinement ratio provides the most significant gain in throughput on ARCHER. Over 45 % more timesteps can be simulated per month, corresponding to an additional 18 jobs per month. On Cab and Vulcan, the optimal refinement ratio provides a 34 % and 31 % increase in job throughput respectively, allowing an additional 13 and 12 jobs to be completed per month.

The most effective configuration change is improving the regridding frequency on Vulcan, allowing over 330,000 more timesteps to be simulated per month. With this 82 % improvement in timesteps per month, we could complete 33 more simulations. On Cab and ARCHER the gains are more modest, but using the most effective configuration changes on these architectures still allows us to complete 15 and 18 more jobs per month respectively. In the context of a mini-application like CleverLeaf, the impact of configuration changes may seem somewhat abstract. However, using the results in this section we have shown that with small changes to parameter values we can increase real-world throughput of a typical production problem by up to 82 %.

7.5.3 Reducing Data Use

Current trends in supercomputer design point towards decreasing amounts of memory per core; the design of the IBM Blue Gene/Q, with only 1 GB memory per core exemplifies this. Typical clusters like Cab and ARCHER have up to 4 GB per core. AMR can provide significant memory savings by reducing the number of cells required to perform any calculation. This allows simulations to be run using fewer hardware resources than a uniform application. The reduction in data also affects the amount of information generated during a simulation. Leadership-class simulations place significant demands on the I/O system of architectures as they will be writing thousands of gigabytes of data in the form of visualisation files and other data output [94]. Using AMR can reduce the amount of data generated at the same rate it reduces the memory consumed by the application at runtime without affecting solution accuracy.

The key factor that determines the memory used and data generated by an application is the number of cells in the simulation. The AMR parameters with the greatest affect on cell count are the maximum level number and the refinement ratio. However, the values that give the maximum reduction in cell count may not reduce the runtime of the simulation; this is especially true in the case of number of levels used. On the Blue Gene/Q, using six levels of refinement to achieve the same finest level resolution as the default configuration caused a 1.5× increase in runtime compared to the default configuration.

We extend the range of experiments that we have run using CleverLeaf to allow us to consider the tradeoff in runtime that must be made in order to achieve an upper-bound on memory usage. As a mini-application, the amount of memory used by CleverLeaf is smaller than a production application, and hence we consider number of cells rather than MB of memory consumed.

Running our replication-scaled problem on 256 nodes using the default configuration results in a total cell count of over 296 million, and runtimes of approximately 130 s, 800 s, and 150 s respectively on Cab, Vulcan, and ARCHER. Figure 7.10 shows the tradeoffs that can be made between simulation runtime

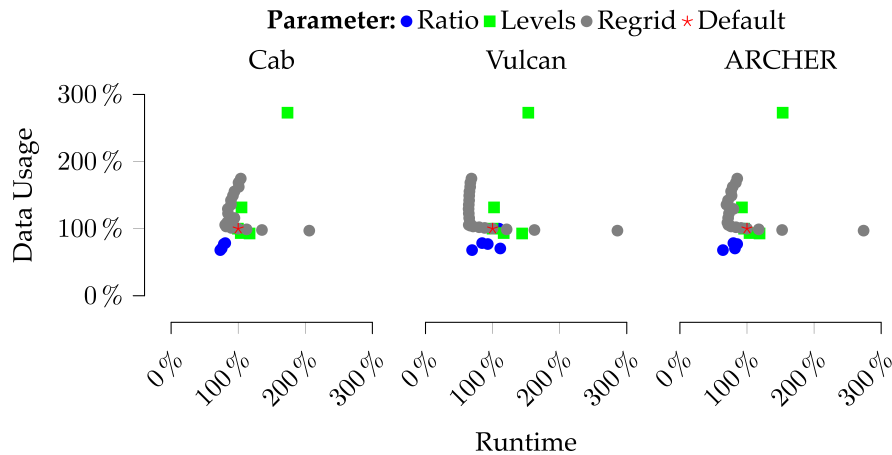


Figure 7.10: Relative data usage and runtime compared to the default configuration.

and total cell count by varying the three parameters on each architecture.

On all three architectures, varying the refinement ratio provides the largest savings in both runtime and total cell count. Using a refinement ratio of three (which was found to be optimal in terms of runtime in Section 7.4.2) provides a reduction in cell count of 32 % compared to the default configuration. This also corresponds to a reduction in runtime of 26 %, 30 % and 35 % on Cab, Vulcan and ARCHER respectively. Even though refinement ratio provides the best improvements in runtime and memory usage, it is important to note that of the three parameters we have examined, it is the most likely to affect simulation accuracy. It is thus necessary to use domain expertise to determine when the tradeoff between accuracy and memory and runtime saving ought to be made.

Varying the regridding frequency shows an interesting pattern on all three architectures. As we saw in Section 7.4.2, choosing a better regridding frequency will reduce the application runtime. However, when the regridding operation is applied less often, more cells are needed to ensure areas of interest remain inside refined areas of the mesh. When regridding every timestep, the lowest total number of cells is reached; a reduction of 3 % from the default parameter set. This decrease in total cell count is accompanied by an increase of over 100 % in simulation runtime on all three architectures. The maximum

decrease in runtime that can be achieved by varying only the regridding frequency is 30 %, but in every case the total cell count increases. This leads us to suggest that regridding frequency is the best parameter to change in order to reduce runtime without affecting simulation accuracy, particularly at scale.

Varying the number of levels used offers minimal benefit in terms of either data usage or simulation runtime. In fact, using only two levels increases cell count by 172 % and increases simulation runtime by up to 53 %. Using either five or six levels of refinement does offer a small reduction in total cell count (6.5 % and 7.2 %) but causes an increase in runtime of up to 44 %. These results suggest that varying only the number of levels used cannot provide the improvements we are looking for. However, because of the tightly coupled relationship between refinement ratio and number of levels necessary to reach a given resolution, varying both these parameters may be of some use.

When looking to reduce both simulation runtime and total cell count, changing the refinement ratio is the most effective parameter to investigate, providing reductions of up to 32 % and 35 % in cell count and runtime. These gains have the potential to affect simulation accuracy, so must be varied with caution. Selecting a more optimal regridding frequency can reduce simulation runtime, but will increase total cell count; the best regridding frequency in terms of runtime increases cell count by 5 %. Altering the number of levels used offers minimal improvements in terms of either cell count or runtime.

7.6 Summary

In this chapter, we used CleverLeaf to perform a detailed analysis of the performance of the default AMR parameter configuration on three architectures: an InfiniBand cluster, a Cray XC30, and an IBM Blue Gene/Q. The detailed analysis presented also includes a comparison of the runtime, memory usage and communication pattern of CleverLeaf with a uniform hydrodynamics application. The AMR application achieves a performance improvement of up to 75 %

in runtime, and a 20 % reduction in memory consumption.

The impact of three of the parameters exposed by CleverLeaf was evaluated on all three experimental platforms. These experiments identify the optimal regridding frequency, maximum level number, and refinement ratio to use on each architecture. The experimental results demonstrate that identifying the optimal value for each parameter and architecture can improve application performance by up to 45 % over the default AMR parameter configuration.

We applied the results of the parameter configuration experiments to examine the impact of optimal parameter on production application scenarios. Selecting an optimal parameter configuration can improve throughput of a typical production workload by up to 82 %, corresponding to an additional 33 jobs completed per month. The impact of parameter configuration on memory consumption was also quantified, and we have shown savings of up to 32 % in total problem cell count. This reduction in cell count not only affects runtime memory consumption, but also the overall data needs of the application, including any visualisation and restart data that may be written to disk during a run.

Performing the experiments in this chapter with a production AMR application would be both difficult and expensive. As noted, the time taken to run a production simulation can be in the order of days. The experiments in this chapter were enabled by the use of a mini-application—CleverLeaf—and show how mini-applications could be used to improve the performance of applications on current architectures.

CHAPTER 8

Conclusions and Future Work

The work presented in this thesis details a methodology for the development and use of mini-applications in improving application performance and investigating future architectures. This process requires the development of a representative mini-application, relying on both the advice of domain experts as well as a subjective evaluation of the correlation between the mini-application and some parent application. Once the mini-application has been developed, it can then be applied in two ways: (i) to investigate application and architecture configuration through experimentation, and (ii) to investigate new parallel architectures and programming models through code porting. We apply this methodology to the domain of shock hydrodynamics with block-structured Adaptive Mesh Refinement (AMR) and successfully demonstrate improvements in two areas: application parameters selection, and execution on Graphics Processing Units (GPUs). This development path can be applied by any High-Performance Computing (HPC) site as they prepare their current suite of applications for future supercomputer architectures.

In this chapter, we present the impact of this research and discuss the implications of results of this thesis to developers and users of scientific and engineering applications, as well as facilities staff and higher-level management. We describe the applicability of our techniques to additional mini-applications, production codes, and future architectures. We also discuss the key limitations of our work.

8.1 Research Impact Highlights

The work conducted during this thesis has impacted both AWE and the wider scientific community. In particular, the following key outcomes highlight the specific impact of each chapter.

- The “leaf” mini-applications discussed in Chapter 4 were the **first UK contribution** to the award-winning Mantevo suite.
- CleverLeaf is the **first reported shock hydrodynamics mini-application with AMR**. It was used by AWE to study the feasibility of using the SAMRAI library, and now **forms the basis for a new AMR-enabled application being developed at AWE**.
- The GPU-based extensions to the SAMRAI library have been shown to scale to **over 4,000 GPUs** on the Titan system, and are now being **used by the SAMRAI developers at Lawrence Livermore National Laboratory** to investigate ways in which they might enhance the library and begin moving their own applications to GPUs.
- The improvements in performance highlighted in Chapter 7 were the result of varying some parameters that are unavailable in Shamrock, AWE’s benchmark AMR code. As such, the parameters that have been identified are **driving code development**, enabling these improvements to be realised in larger applications.

8.2 Discussion and Implications

Chapter 4 introduces the concept of mini-applications and highlights how they can be used to prepare key parent applications for future high-performance computing architectures. We present results that show how different programming models allow performance improvements of over $2\times$ by allowing the application to execute on accelerator architectures. We also present results where the performance improvements are achieved through a more code-independent means: improving the layout of the parallel processes in a CloverLeaf run (by taking application communication patterns into account) can result in a performance increase of over four percent. The two sets of results exemplify two ways in which mini-applications can be applied to improve the performance of key applications. The large performance improvement and minimal investment, particularly in the first experiment, shows how essential it is to ensure that the application is run in an optimal configuration. Taken together, the results from Chapters 5, 6 and 7 show how a representative mini-application can be developed and then used to influence both current code performance and future code development.

In order for the results derived using mini-applications to be applicable to a key code, the mini-application must be carefully designed to be representative of the parent application. In Chapter 5 we develop CleverLeaf, the first block-structured AMR mini-application for shock hydrodynamics codes. The validation and verification results imply that developing a small, representative mini-application is possible.

The first way that mini-applications can be used to improve the performance of a parent application is through identifying optimal machine configurations or application parameters, and it is this path we use in Chapter 7 to study the optimal choice of a set of AMR parameter values on three contemporary supercomputer architectures. Using the correct parameter values for a given architecture offers performance improvements of up to 45%. Applying

these improvements to a typical production workload shows that job throughput (per month) could be increased by up to 82%.

Exascale systems are likely to rely on accelerator devices, or in the absence of these, hierarchical memory systems with varied access times. The second way we can use a mini-application to improve parent application performance is by investigating future architectures. In Chapter 6, we present results from a version of CleverLeaf developed to run on NVIDIA GPUs. With a reduction in runtime of over four times when compared to the CPU-based code, and a clear demonstration of scalability when running on over 4,000 nodes of Titan, these results imply that accelerator architectures must be seriously considered for future code development.

When considering other aspects of large HPC applications, software development can be as important as performance engineering. The development details of CleverLeaf presented in Chapter 5 and the Graphics Processing Unit (GPU)-based version of CleverLeaf presented in Chapter 6 highlight the importance of using sound design principles, such as object-oriented design patterns, when developing scientific software. With future supercomputer architectures likely to change significantly in the next ten years, abstracting hardware-specific aspects of an application into a library will be key for ensuring maintainable software development.

8.3 Limitations

The primary limitation of this thesis is the focus or our techniques on a single mini-application: CleverLeaf. However, the development of the first block-structured AMR hydrodynamics application is a significant contribution of this work, and the purpose of later chapters is to demonstrate how a mini-application can be used in practice. Although a single application may at first appear to be a strong constraint, the use of single *mini-application* is less so since the application is designed from first principles to be representative of a larger application

(or even a class of applications). As demonstrated in Chapter 5, CleverLeaf is in fact representative of a larger benchmark application, Shamrock. Optimisations investigated through the hydrodynamics kernels of CloverLeaf (the uniform mini-application) that have been applied to Shamrock improved the performance of the momentum advection kernel up to 5× [57]. We use this performance result and the correlation between CleverLeaf and Shamrock to infer that the optimisations and techniques used to improve the performance of CleverLeaf would also work in the context of a larger application. Although focused, this thesis acts as a case study for using mini-applications to improve the performance of larger codes.

A second potential limitation is our use of the CUDA programming model in Chapter 6 for developing a scalable, GPU-based AMR application. The CUDA programming language and runtime environment are proprietary technologies owned by NVIDIA. As such, the CUDA programming model can only be used to write applications that will run on NVIDIA's GPUs. This focus on NVIDIA hardware means we only examine one possible future architecture in this thesis. However, despite not being officially supported, some research projects enhance the portability of the CUDA programming model by allowing CUDA code to run on other hardware.

The MCUDA translation framework uses source code translation and a custom runtime system to allow CUDA kernels to execute on multi-core CPUs [150]. The Swan project is a source-to-source translator that will take an existing CUDA application and translate it to an equivalent OpenCL program [67]. The Ocelot project is a dynamic compilation framework that provides a way to execute NVIDIA's PTX virtual instructions on a range of platforms; currently, NVIDIA GPUs, AMD GPUs and x86 CPUs are supported [46]. The PGI CUDA C/C++ for x86 compiler (CUDA-x86) processes CUDA C as a native parallel programming language for multi-core x86 processors [118]. CUDA-x86 will inline device kernel functions, translate CUDA C's *chevron* syntax to parallel loops, that execute using multiple cores and vector instructions. Results presented by PGI

show that the performance of CUDA-x86 are comparable with other programming models such as OpenMP.

Each of these projects provides a simple way to execute CUDA code on a range of architectures, expanding the generality of our techniques. However, due to the object-oriented design of SAMRAI, the CUDA-specific code we have developed is isolated behind a select few methods. Writing equivalent versions of our classes in other programming models is feasible, and would have a reduced development cost since the high-level design of the classes and the thread-parallel nature of the algorithms would be similar.

A final limitation is our focus on time-to-solution as the primary metric for evaluating application performance. There are a range of other metrics that can be used to evaluate application performance, both in terms of application performance but also application accuracy. In Chapter 7, we briefly considered additional hardware metrics such as memory consumption and network traffic, but these are not our primary means of measuring application performance. From an application accuracy perspective, the complex nature of AMR means that changing application parameters that affect simulation runtime can also impact simulation accuracy. Quantifying the change in accuracy due to changes in parameter values presents a huge search space. Later in this chapter we present some initial work where the link between accuracy and performance is investigated.

8.4 Future Work

The work presented in this thesis can be applied and extended in a number of ways. Specifically, we discuss the utility of mini-applications in investigating simulation accuracy in addition to performance; we consider the applicability of our work to production codes; and we discuss the possible ways the work could be extended to future parallel architectures.

8.4.1 Investigating Simulation Accuracy

Each of the three configuration improvements presented in Chapter 7 focuses on improving application performance by reducing runtime or memory consumption. However, changing mesh parameters can have an impact on solution accuracy, something that is important in the scientific domains in which AMR is typically employed. By developing techniques with which we can measure the affect of changing parameter values on simulation accuracy, we can begin to optimise parameter selection along two dimensions: accuracy and performance.

To investigate these issues we propose the use of a test problem with an analytical solution, such as Sod's shock tube problem [147]. Calculating some error metric for each parameter value, we can measure the correlation between accuracy and performance for each parameter.

Table 8.1 shows the affect that changing the refinement ratio, regridding frequency, and number of levels has on simulation accuracy and runtime for the Sod problem. Average error is calculated using three different metrics: L^1-norm, L^2-norm, and L^∞-norm:

$$L^1 = \frac{\Sigma_{i=1}^{N} Y_i - f(x_i)}{N} \tag{8.1}$$

$$L^2 = \frac{\sqrt{\Sigma_{i=1}^{N} (Y_i - f(x_i))^2}}{N} \tag{8.2}$$

$$L^\infty = max_{i=1}^{N} |Y_i - f(x_i)| \tag{8.3}$$

where Y_i and $f(x_i)$ are the analytic and numerical solutions at the point i, and N is the number of grid points. Normalising using N allows us to compare the different resolutions of the AMR configurations.

The intuitive interpretation that accuracy and performance can be decoupled for certain parameters suggested in Chapter 7 is reinforced. Optimising runtime via a reduced regridding frequency does not have a negative impact on accuracy. Increasing the refinement ratio decreases error but rapidly

Parameter	Runtime	L^1-error	L^2-error	L^∞-error
		Regridding Frequency		
1	1.863	-3.102×10^{-3}	8.295×10^{-4}	1.570×10^{-1}
2	1.012	-3.084×10^{-3}	8.230×10^{-4}	1.571×10^{-1}
4	0.651	-3.081×10^{-3}	8.135×10^{-4}	1.570×10^{-1}
8	0.522	-3.084×10^{-3}	8.138×10^{-4}	1.577×10^{-1}
		Refinement Ratio		
2	0.662	-3.081×10^{-3}	8.135×10^{-4}	1.570×10^{-1}
3	2.227	-3.272×10^{-3}	6.625×10^{-4}	1.608×10^{-1}
4	5.654	-1.808×10^{-3}	2.986×10^{-4}	$\mathbf{1.609 \times 10^{-1}}$
5	15.282	$\mathbf{-9.461 \times 10^{-4}}$	$\mathbf{1.273 \times 10^{-4}}$	1.608×10^{-1}
		Maximum Level		
1	**0.187**	-3.815×10^{-3}	1.364×10^{-3}	7.193×10^{-2}
2	0.195	-3.072×10^{-3}	8.413×10^{-4}	1.003×10^{-1}
3	0.652	-3.081×10^{-3}	8.135×10^{-4}	1.570×10^{-1}
4	2.242	-3.174×10^{-3}	6.790×10^{-4}	1.607×10^{-1}

Table 8.1: Error norms and runtimes for various simulation configurations.

increases runtime due to the large increase in cell count. Adjusting the number of levels of refinement has the largest impact on runtime, but just adding one level of refinement provides a large decrease in L^2 error. Using a mini-application we can extend the process from Chapter 7 and begin to investigate multi-dimensional optimisation of both simulation accuracy and performance.

8.4.2 Application to Production Codes

The use of a mini-application is underpinned by a desire to learn more about how production applications might be used, improved or even re-written for future supercomputer architectures. As such, it is critical that the techniques presented in this thesis can be applied to full-scale production applications. The nature of this work, involving large, complex, and commercially sensitive production applications is beyond the scope of this thesis.

In Chapter 5 we showed a correspondence between the performance of CleverLeaf and Shamrock. This verification illustrates that CleverLeaf accurately represents the computational characteristics seen in Shamrock, and thus the improvements we investigate and implement using CleverLeaf in Chap-

ters 6 and 7 are likely to result in similar improvements for a larger application such as Shamrock. Optimisations investigated through the hydrodynamics kernels of CloverLeaf (which has the same hydrodynamics scheme as CleverLeaf, but no AMR) that have been applied to Shamrock improved the performance of the momentum advection kernel up to 5× [57]. Whilst this only resulted in a small improvement in the overall runtime of the application, incorporating more optimised kernels in Shamrock should provide similar performance improvements.

Due to the commercially sensitive nature of the codes, the improvements proposed in this thesis would need to be implemented through collaboration with the Atomic Weapons Establishment (AWE). A number of open-source AMR codes exist however which could provide a more accessible context for applying our techniques to production codes. Specifically, the FLASH application from the University of Chicago is a large astrophysics application with AMR that has some similarity to the class of codes represented by CleverLeaf [53]. FLASH uses the PARAMESH and Chombo libraries for AMR functionality; both of these libraries encapsulate common AMR concepts, so it is feasible we could extend it using an approach similar to that described in Chapter 5. Another open-source production AMR application, IBAMR, uses the SAMRAI library [62] so could benefit from our research more directly. However, this application simulates fluid-structure interactions using an immersed boundary method; a domain distinctly different from the shock hydrodynamics codes that CleverLeaf is designed to represent.

8.4.3 Extension of GPU Implementation

The GPU-based extensions to SAMRAI developed in Chapter 6 are fully interoperable with the `PatchData` interface that allows SAMRAI to manage user-defined datatypes. As such, these extensions are ready to be used in additional codes. Whilst CleverLeaf, the first fully resident GPU-based AMR hydrodynamics mini-application, is a good case study in using accelerators for AMR,

other applications will have unique performance characteristics that may expose areas in our design in need of optimisation.

Whilst porting an application can be time-consuming, we have identified three small benchmark applications that could be used to further investigate the performance of our GPU-based library extensions. In addition to a fully-featured software library for developing AMR applications, the SAMRAI source code also contains three example applications. These examples are small, yet fully functional, containing representative physics and production-quality features like visualisation output and flexible initial condition support. Each application solves a different physical simulation: convection-diffusion, Euler's equations, and linear advection. An interesting test would be the Euler application, since whilst CleverLeaf also solves these equations, it uses a different hydrodynamics scheme and a different overall code design. The Euler application included in the SAMRAI distribution uses Riemann solvers, rather than the explicit Lagrangian-Eulerian approach we use.

Each of these applications is available under an open-source licence, easing the porting process. The recent interest in GPU-based supercomputing means that interaction with the SAMRAI development team should be mutually beneficial and would result in a comprehensive suite of applications to verify our resident approach to GPU-based AMR.

8.4.4 Additional Parallel Architectures

GPUs have been a key simulation architecture for over three years. At the time of writing (June 2014) the Titan supercomputer, using over 18 thousand GPUs, is the second fastest computer in the world. The fastest computer, Tianhe–2 at the National Super Computer Center in Guangzhou, China, uses 48 thousand Intel Xeon Phi co-processors to deliver over 50 Tera (10^{12}) Floating Point Operations per Second (TFLOPs) peak theoretical performance. The Xeon Phi is a highly-parallel architecture based around a large number of slower (compared to a traditional Central Processing Unit (CPU)) cores.

This many-core architecture is in some ways similar to a GPU: since the cores are slower, more parallelism must be exposed to harness the full power of the device. However, a key advantage over GPUs is that a special programming model is not required. Rather than device kernels written in a superset of C or C++, Intel's Xeon Phi co-processors use regular C, C++ or Fortran code that is compiled specifically for the device. Each of the cores in the Xeon Phi has four hardware threads, and to achieve optimal performance most of these hardware threads must be used when executing an application. This means that many application threads must be used, either by launching multiple MPI processes or using a programming model that enables threading.

One advantage of the Xeon Phi architecture is that existing code can be re-compiled without modification to run on the new hardware. If this path is taken however, there is no guarantee that the achieved performance will be anywhere near optimal [85]. Any Xeon Phi-specific version of CleverLeaf would need to be carefully tuned to make the best use of the slower processing cores and the multiple hardware threads.

To obtain a baseline measurement of the performance of CleverLeaf on a Xeon Phi, we recompiled the standard CPU-based code and accompanying libraries. Using the Sod problem from Chapters 5 and 6, in Figure 8.1 we present a comparison of the performance of the Intel Xeon Phi and an Intel CPU for a range of problem sizes. On the Xeon Phi we use the following configurations of MPI tasks and OpenMP threads: 1×120, 2×60, 4×30, 8×15 and 30×4. Using a total of 120 threads on the Xeon Phi means we are using two threads per core, where each core supports up to four hardware threads. On the CPU we use two MPI tasks with eight OpenMP threads (2×8). This configuration corresponds to one MPI task for each NUMA region, and one OpenMP thread for each physical core.

In most cases the performance of the CPU-based code, when recompiled for Xeon Phi, does not match the performance obtained from an Intel CPU running at 2.6 GHz. However, for two different threading configurations at the

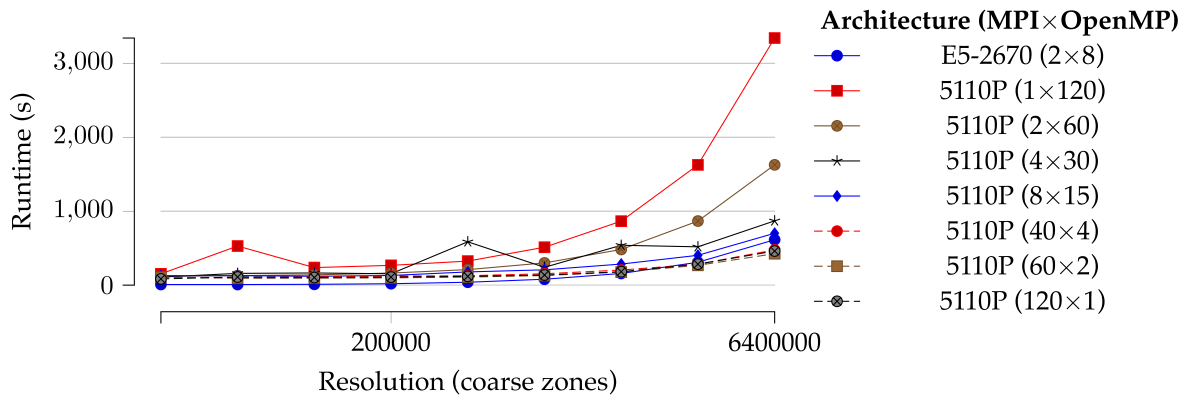


Figure 8.1: Performance of CleverLeaf on an Intel Xeon Phi and an Intel CPU.

two largest problem sizes, the performance of the Xeon Phi is better than the CPU. When running with 60 MPI tasks and 2 OpenMP threads per task, the Xeon Phi is over 30% faster for the largest problem. The fact the code could be run unmodified on this new architecture is an important point to consider when evaluating a future architecture for legacy codes.

Bibliography

[1] Hydrodynamics Challenge Problem . Technical Report LLNL-TR-490254, Lawrence Livermore National Laboratory, Aug. 2011. (Cited on page 82.)

[2] Advantage Business Media. 2013 R&D 100 Award Winners. URL `http://www.rdmag.com/award-winners/2013/07/2013-r-d-100-award-winners`. (Cited on page 65.)

[3] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. (Cited on page 83.)

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1), July 1997. (Cited on page 17.)

[5] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Daly, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Kogge, J. Levesque, D. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. ExaScale Software Study: Software Challenges in Extreme Scale Systems. Techni-

cal report, Oct. 2009. URL http://crd.lbl.gov/assets/pubs_presos/CDS/ATG/ECSS-report-101909.pdf. (Cited on page 58.)

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, Apr. 1967. (Cited on pages 18 and 134.)

[7] J. A. Anderson, E. Jankowski, T. L. Grubb, M. Engel, and S. C. Glotzer. Journal of Computational Physics. *Journal of Computational Physics*, 254 (C):27–38, Dec. 2013. (Cited on pages 113 and 115.)

[8] J. D. Anderson. *Computational Fluid Dynamics*. Springer, 1995. (Cited on page 39.)

[9] R. W. Anderson, N. S. Elliott, and R. B. Pember. An arbitrary Lagrangian–Eulerian method with adaptive mesh refinement for the solution of the Euler equations. *Journal of Computational Physics*, 199(2): 598–617, Sept. 2004. (Cited on pages 44, 81, and 85.)

[10] R. W. Anderson, W. J. Arrighi, N. S. Elliott, B. T. N. Gunney, and R. D. Hornung. SAMRAI Concepts and Software Design. Technical Report LLNL-SM-617092-DRAFT, Lawrence Livermore National Laboratory, May 2013. (Cited on page 84.)

[11] T. D. Arber, A. W. Longbottom, C. L. Gerrard, and A. M. Milne. A Staggered Grid, Lagrangian–Eulerian Remap Code for 3-D MHD Simulations. *Journal of Computational Physics*, 171(1):151–181, July 2001. (Cited on page 3.)

[12] D. Bailey, D. Browning, R. Carter, S. Fineberg, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Mar. 1994. (Cited on page 33.)

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991. (Cited on page 33.)

[14] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In *Proceedings of the 21st ACM/IEEE International Conference on Supercomputing*, Nov. 2008. (Cited on page 64.)

[15] D. W. Barnette, R. F. Barrett, S. D. Hammond, J. Jayaraj, and J. H. Laros III. Using Miniapplications in a Mantevo Framework for Optimizing Sandia's SPARC CFD Code on Multi-Core, Many-Core, and GPU-Accelerated Compute Platforms. In *Proceedings of the 51st AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, Jan. 2013. (Cited on pages 33 and 131.)

[16] R. F. Barrett, X. S. Hu, S. S. Dosanjh, S. Parker, M. A. Heroux, and J. Shalf. Toward Codesign in High Performance Computing Systems. In *Proceedings of the 31st International Conference of Computer-Aided Design*, pages 443–449, Nov. 2012. (Cited on pages 33 and 67.)

[17] D. A. Beckingsale, O. Perks, W. P. Gaudin, J. A. Herdman, and S. A. Jarvis. Optimisation of Patch Distribution Strategies for AMR Applications. *Lecture Notes in Computer Science*, 7587:210–223, 2013. (Cited on pages viii and 142.)

[18] D. A. Beckingsale, W. P. Gaudin, R. D. Hornung, B. N. Gunney, T. Gamblin, J. A. Herdman, and S. A. Jarvis. Parallel Block Structured Adaptive Mesh Refinement on Graphics Processing Units. In *Proceedings of the 44th International Conference on Parallel Processing*, Sept. 2015. (Cited on page vii.)

[19] M. Berger. Adaptive Mesh Refinement Software for Hyperbolic Conservation Laws. URL `http://cs.nyu.edu/berger/amrsoftware.html`. (Cited on page 54.)

[20] M. Berger, D. L. George, R. J. Leveque, and K. T. Mandli. The GeoClaw software for depth-averaged flows with adaptive refinement. *Advances in Water Resources*, 34(9):1195–1206, Sept. 2011. (Cited on page 54.)

[21] M. J. Berger and P. Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, May 1989. (Cited on pages 47, 48, 79, 90, 91, and 152.)

[22] M. J. Berger and R. J. LeVeque. Adaptive Mesh Refinement Using Wave-Propagation Algorithms for Hyperbolic Systems. *SIAM Journal on Numerical Analysis*, 35(6):2298–2316, Dec. 1998. (Cited on page 54.)

[23] M. J. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics*, 53(3):484–512, Mar. 1984. (Cited on pages 47, 48, 79, 89, 90, 95, 115, and 145.)

[24] M. J. Berger and I. Rigoutsos. An Algorithm for Point Clustering and Grid Generation. *IEEE Transactions on Systems, Man, and Cybernetics*, 21 (5):1278–1286, Oct. 1991. (Cited on page 142.)

[25] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. Performance Modelling of Magnetohydrodynamics Codes. *Lecture Notes in Computer Science*, 7587:197–209, Feb. 2013. (Cited on page viii.)

[26] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. CRC Press, 2014. (Cited on page 43.)

[27] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High Performance Computing Applications*, 14 (3):189–204, Aug. 2000. (Cited on page 103.)

[28] G. Bryan. Fluids in the Universe: Adaptive Mesh Refinement in Cosmology. *Computing in Science & Engineering*, 1(2):46–53, Apr. 1999. (Cited on pages 47 and 79.)

[29] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, and Y. Li. Enzo: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal Supplement Series*, 211, Mar. 2014. (Cited on pages 44 and 81.)

[30] C. Burstedde, O. Ghattas, M. Gurnis, T. Isaac, G. Stadler, T. Warburton, and L. C. Wilcox. Extreme-Scale AMR. In *Proceedings of the 22nd IEEE/ACM International Conference on Supercomputing*, Nov. 2010. (Cited on page 115.)

[31] D. E. Burton. Exact conservation of energy and momentum in staggered-grid hydrodynamics with arbitrary connectivity. *Lecture Notes in Physics*, 395:7–19, Jan. 1991. (Cited on page 101.)

[32] Clawpack Development Team. AMRClaw. URL `http://www.clawpack.org/amrclaw.html`. (Cited on page 54.)

[33] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, page 36, June 2005. (Cited on page 25.)

[34] P. Colella, J. Bell, N. Keen, T. Ligocki, M. Lijewski, and B. v. Straalen. Performance and Scaling of Locally-Structured Grid Methods for Partial Differential Equations. *Journal of Physics: Conference Series*, 78:1–13, Aug. 2007. (Cited on pages 55 and 141.)

[35] P. Colella, D. Graves, N. D. Keen, T. Ligocki, D. Martin, P. W. McCorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. Van Straalen. Chombo Software Package for AMR Applications - Design Document. Technical Report LBNL-6616E, Lawrence Berkeley National Laboratory, Apr. 2009. (Cited on page 53.)

[36] S. L. Cornford, D. F. Martin, D. T. Graves, D. F. Ranken, A. M. Le Brocq, R. M. Gladstone, A. J. Payne, E. G. Ng, and W. H. Lipscomb. Adaptive mesh, finite volume modeling of marine ice sheets. *Journal of Computational Physics*, 232:419–437, Nov. 2013. (Cited on page 143.)

[37] R. Courant, K. Friedrichs, and H. Lewy. On the Partial Difference Equations of Mathematical Physics. *IBM Journal of Research and Development*, 11(2):215–234, Mar. 1967. (Cited on pages 42 and 152.)

[38] W. Y. Crutchfield and M. L. Welcome. Object-Oriented Implementation of Adaptive Mesh Refinement Algorithms. *Scientific Programming*, 2(4): 145–156, Dec. 1993. (Cited on page 55.)

[39] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993. (Cited on page 17.)

[40] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, Apr. 1988. (Cited on page 14.)

[41] J. A. Davis, G. R. Mudalige, S. D. Hammond, J. A. Herdman, I. Miller, and S. A. Jarvis. Predictive Analysis of a Hydrodynamics Application on Large-Scale CMP Clusters. *Computer Science - Research and Development*, 26(3-4):175–185, June 2011. (Cited on page 33.)

[42] R. DeBar. Fundamentals of the KRAKEN code. Technical Report UCID-17366, Lawrence Livermore Laboratory, Mar. 1974. (Cited on pages 80 and 101.)

[43] R. Deiterding. Detonation Simulation with the AMROC Framework. Technical report, California Institute of Technology, 2003. URL `http://csdrm.caltech.edu/publications/cit-asci-tr/cit-asci-tr317.pdf`. (Cited on page 55.)

[44] R. Deiterding. A generic framework for blockstructured Adaptive Mesh Refinement in Object-oriented C++, June 2004. URL `http://amroc.sourceforge.net/index.htm`. (Cited on page 55.)

[45] R. Deiterding. Construction and Application of an AMR Algorithm for Distributed Memory Computers. *Lecture Notes in Computer Science*, 41: 361–372, 2005. (Cited on page 55.)

[46] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 353–364, Sept. 2010. (Cited on page 171.)

[47] J. Dongarra and M. A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report SAND2013-4744, Sandia National Laboratories, June 2013. (Cited on page 32.)

[48] J. Dongarra, P. Luszczek, and A. Petitet. The Linpack Benchmark: Past, Present, and Future. *Concurrency and Computation: Practice & Experience*, 15(9):803–820, Aug. 2003. (Cited on page 31.)

[49] S. S. Dosanjh, R. F. Barrett, D. W. Doerfler, S. D. Hammond, K. S. Hemmert, M. A. Heroux, P. T. Lin, K. T. Pedretti, A. F. Rodrigues, T. G. Trucano, and J. P. Lutjiens. Exascale design space exploration and co-design. *Future Generation Computer Systems*, 30:46–58, Jan. 2014. (Cited on page 33.)

[50] C. R. Ferenbaugh. The PENNANT Mini-App. Technical Report LA-CC-12-021, Los Alamos National Laboratory, Feb. 2014. (Cited on page 82.)

[51] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sept. 1972. (Cited on page 14.)

[52] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, May 1978. (Cited on page 16.)

[53] B. Fryxell, K. Olson, P. Ricker, F. Timmes, M. Zingale, D. Lamb, P. MacNeice, R. Rosner, J. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, Nov. 2000. (Cited on pages 3, 44, 47, 52, 79, 80, and 175.)

[54] J. Fung, S. Mann, and C. Aimone. OpenVIDIA: Parallel GPU Computer Vision. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, 2005. (Cited on page 23.)

[55] S. Galera, P.-H. Maire, and J. Breil. A two-dimensional unstructured cell-centered multi-material ALE scheme using VOF interface reconstruction. *Journal of Computational Physics*, 229(16):5755–5787, Aug. 2010. (Cited on page 134.)

[56] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–9, Nov. 2010. (Cited on page 133.)

[57] W. P. Gaudin. personal communication, Aug. 2014. (Cited on pages 171 and 175.)

[58] C. Gheller, P. Wang, F. Vazza, and R. Teyssier. Numerical cosmology on the GPU with Enzo and Ramses. *arXiv:astro-ph/1412.934*, Dec. 2014. (Cited on page 115.)

[59] J. W. Gibbs. Fourier's Series. *Nature*, 59(1522):200, Dec. 1898. (Cited on page 96.)

[60] J. W. Gibbs. Fourier's Series. *Nature*, 59(1539):606, Apr. 1899. (Cited on page 96.)

[61] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, D. Ranta, and R. Stefan. The RAGE radiation-hydrodynamic code. *arXiv:comp-ph/0804.1394*, Apr. 2008. (Cited on page 52.)

[62] B. E. Griffith, R. D. Hornung, D. M. McQueen, and C. S. Peskin. An adaptive, formally second order accurate version of the immersed boundary method. *Journal of Computational Physics*, 223(1):10–49, Apr. 2007. (Cited on pages 83 and 175.)

[63] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. *Lecture Notes in Computer Science*, (1697):11–18, 1999. (Cited on page 31.)

[64] B. N. Gunney. Scalable Mesh Management for Patch-based AMR. Technical Report LLNL-PROC-610672, Lawrence Livermore National Laboratory, 2013. (Cited on page 54.)

[65] B. T. N. Gunney, A. M. Wissink, and D. A. Hysom. Parallel Clustering Algorithms for Structured AMR. *Journal of Parallel and Distributed Computing*, 66(11):1419–1430, Nov. 2006. (Cited on pages 54 and 142.)

[66] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988. (Cited on page 19.)

[67] M. J. Harvey and G. De Fabritiis. Swan: A tool for porting CUDA programs to OpenCL. *Computer Physics Communications*, 182(4):1093–1099, Apr. 2011. (Cited on page 171.)

[68] W. Henshaw. Overture: An Object-Oriented Framework for Overlapping Grid Applications. In *Proceedings of the 32nd AIAA Fluid Dynamics Conference and Exhibit*, June 2012. (Cited on page 56.)

[69] T. A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design. *Computer*, 40(10):32–40, 2007. (Cited on page 63.)

[70] J. A. Herdman, W. P. Gaudin, D. Turland, and S. D. Hammond. Benchmarking and Modelling of POWER7, Westmere, BG/P, and GPUs: An Industry Case Study. *SIGMETRICS Performance Evaluation Review*, 38(4): 16–22, Mar. 2011. (Cited on pages 33, 81, 103, and 116.)

[71] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Accelerating Hydrocodes with OpenACC, OpeCL and CUDA. In *Proceedings of the 24th IEEE/ACM International Conference on Supercomputing*, pages 465–471, Nov. 2012. (Cited on pages vii and 70.)

[72] J. A. Herdman, W. P. Gaudin, O. Perks, O. F. J. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. Achieving portability and performance through OpenACC. In *Proceedings of the 1st Workshop on Accelerator Programming using Directives*, 2014. (Cited on pages vii and 72.)

[73] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, Sept. 2009. (Cited on pages 7, 33, 62, 65, and 67.)

[74] J. Hill, B. McColl, D. C. Stefanescu, and M. W. Goudreau. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, Dec. 1998. (Cited on page 17.)

[75] C. W. Hirt, A. A. Amsden, and J. L. Cook. An arbitrary Lagrangian-

Eulerian computing method for all flow speeds. *Journal of Computational Physics*, 14(3):227–253, Mar. 1974. (Cited on page 80.)

[76] J. K. Holmen and D. L. Foster. Accelerating Single Iteration Performance of CUDA-Based 3D Reaction–Diffusion Simulations. *International Journal of Parallel Programming*, May 2013. (Cited on page 115.)

[77] R. D. Hornung. A Case for Improved C++ Compiler Support to Enable Performance Portability in Large Physics Simulation Codes. Technical Report LLNL-TR-635681, Lawrence Livermore National Laboratory, Apr. 2013. (Cited on page 63.)

[78] R. D. Hornung and S. R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice & Experience*, 14(5):347–368, Apr. 2002. (Cited on pages 56, 84, and 119.)

[79] R. D. Hornung, A. M. Wissink, and S. R. Kohn. Managing complex data and geometry in parallel structured AMR applications. *Engineering with Computers*, 22(3):181–195, Dec. 2006. (Cited on page 84.)

[80] C. B. Hummels and G. L. Bryan. Adaptive Mesh Refinement Simulations of Galaxy Formation: Exploring Numerical and Physical Parameters. *The Astrophysical Journal*, 749(2):1–17, Apr. 2012. (Cited on page 143.)

[81] A. Humphrey, Q. Meng, M. Berzins, and T. Harman. Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment*, July 2012. (Cited on pages 52 and 116.)

[82] Intel Corporation. Home | Threading Building Blocks, . URL `https://www.threadingbuildingblocks.org/`. (Cited on page 27.)

[83] Intel Corporation. Cilk Home Page | CilkPlus, . URL `http://www.cilkplus.org/`. (Cited on page 27.)

[84] Intel Corporation. Intel MPI Benchmarks 4.0, . URL https://software.intel.com/en-us/articles/intel-mpi-benchmarks. (Cited on page 31.)

[85] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming*. Morgan Kaufmann, Feb. 2013. (Cited on page 177.)

[86] I. Karlin. personal communication, July 2014. (Cited on page 63.)

[87] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *Proceedings of the 27th International Symposium on Parallel & Distributed Processing*, pages 919–932, May 2013. (Cited on pages 33, 62, and 131.)

[88] D. Kerbyson, A. Hoisie, and H. J. Wasserman. A comparison between the Earth Simulator and AlphaServer systems using predictive application performance models. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 10, 2003. (Cited on pages 5, 22, and 61.)

[89] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzen, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, Oct. 2008. URL http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf. (Cited on page 58.)

[90] A. E. Koniges, N. D. Masters, A. C. Fisher, R. W. Anderson, D. C. Eder, T. B. Kaiser, D. S. Bailey, B. Gunney, P. Wang, B. Brown, K. Fisher, F. Hansen, B. R. Maddox, D. J. Benson, M. Meyers, and A. Geille. ALE-AMR: A new 3D multi-physics code for modeling laser/target effects.

Journal of Physics: Conference Series, 244(3):1–4, Sept. 2010. (Cited on pages 81 and 83.)

[91] I. M. Kulikov, I. G. Chernykh, A. V. Snytnikov, B. M. Glinskiy, and A. V. Tutukov. AstroPhi: A code for complex simulation of dynamics of astrophysical objects using hybrid supercomputers. *Computer Physics Communications*, 186:71–80, Jan. 2015. (Cited on page 44.)

[92] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, Feb. 1974. (Cited on page 114.)

[93] Z. Lan, V. Taylor, and G. Bryan. Dynamic Load Balancing for Structured Adaptive Mesh Refinement Applications. In *Proceedings of the 30th International Conference on Parallel Processing*, pages 571–579, Sept. 2001. (Cited on page 143.)

[94] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the 21st ACM/IEEE International Conference on Supercomputing*, Nov. 2009. (Cited on page 163.)

[95] P. F. Lavallée, G. C. de Verdière, P. Wautelet, D. Lecas, and J. Dupays. Porting and optimizing HYDRO to new platforms and programming paradigms – lessons learnt. Technical report, PRACE, Dec. 2012. (Cited on pages 33 and 131.)

[96] Lawrence Livermore National Laboratory. ASC Sequoia Benchmark Codes. URL `http://asc.llnl.gov/sequoia/benchmarks/`. (Cited on page 33.)

[97] *Proxy-Application Validation Using Hardware Performance Counters*, Aug. 2014. Lawrence Livermore National Laboratory. Poster presentation. (Cited on page 102.)

[98] Lawrence Livermore National Laboratory. SAMRAI Overview, May 2014. URL http://computation.llnl.gov/casc/SAMRAI/. (Cited on pages 68 and 114.)

[99] J. Luitjens and M. Berzins. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurrency and Computation: Practice & Experience*, 23(13):1522–1537, Sept. 2011. (Cited on pages 54 and 142.)

[100] J. Luitjens, B. Worthen, M. Berzins, and T. C. Henderson. Scalable Parallel AMR for the Uintah Multiphysics Code. In *Petascale Computing: Algorithms and Applications*, pages 67–82. Dec. 2007. (Cited on pages 52, 54, and 56.)

[101] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. D. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, Mar. 2005. URL http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/hpcc-challenge-benchmark05.pdf. (Cited on page 31.)

[102] Z. H. Ma, H. Wang, and S. H. Pu. GPU computing of compressible flow problems by a meshless method with space-filling curves. *Journal of Computational Physics*, 263(C):113–135, Apr. 2014. (Cited on page 115.)

[103] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, Apr. 2000. (Cited on pages 52, 53, and 55.)

[104] 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 1st International Workshop on OpenCL*, May 2013. (Cited on page vii.)

[105] 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*, May 2013. (Cited on pages vii and 72.)

[106] Mantevo.org. Mantevo - Home. URL `http://mantevo.org`. (Cited on pages 7, 65, and 82.)

[107] 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. (Cited on page 31.)

[108] J. M. McGlaun, S. L. Thompson, and M. G. Elrick. CTH: a three-dimensional shock wave physics code. *International Journal of Impact Engineering*, 10(1-4):351–360, 1990. (Cited on page 41.)

[109] F. H. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Dec. 1986. (Cited on page 31.)

[110] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers. In *Proceedings of the 24th ACM/IEEE International Conference on Supercomputing*, Nov. 2013. (Cited on pages 114 and 116.)

[111] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, May 1994. URL `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. (Cited on page 21.)

[112] G. E. Moore. Cramming More Components Onto Integrated Circuits. In *Proceedings of the IEEE*, pages 82–85, Jan. 1998. (Cited on pages 4, 11, and 22.)

[113] T. Mudge. Power: A First-Class Architectural Design Constraint. *Computer*, 34(4):52–58, Apr. 2001. (Cited on pages 5 and 22.)

[114] G. J. Narlikar and G. E. Blelloch. Pthreads for Dynamic and Irregular Parallelism. In *Proceedings of the 24th ACM/IEEE International Conference on Supercomputing*, Nov. 1998. (Cited on page 27.)

[115] Netlib. LINPACK. URL `http://www.netlib.org/linpack/`. (Cited on pages 31 and 61.)

[116] D. Nicholaeff, N. Davis, D. Trujillo, and R. W. Robey. Cell-Based Adaptive Mesh Refinement Implemented with General Purpose Graphics Processing Units. Technical Report LA-UR-11-07127, Los Alamos National Laboratory, Mar. 2012. (Cited on pages 52, 82, and 116.)

[117] M. Niklaus, W. Schmidt, and J. C. Niemeyer. Two-dimensional adaptive mesh refinement simulations of colliding flows. *Astronomy & Astrophysics*, 506(2):1065–1070, Nov. 2009. (Cited on page 143.)

[118] NVIDIA Corporation. PGI CUDA-X86, . URL `https://www.pgroup.com/resources/cuda-x86.htm`. (Cited on pages 71 and 171.)

[119] NVIDIA Corporation. Parallel Programming and Computing Platform | CUDA | NVIDIA, . URL `http://www.nvidia.com/object/cuda_home_new.html`. (Cited on pages 27, 71, and 118.)

[120] OpenACC.org. OpenACC Home | openacc.org. URL `http://www.openacc.org`. (Cited on pages 27, 70, and 118.)

[121] OpenMP Architecture Review Board. OpenMP.org. URL `http://openmp.org`. (Cited on page 26.)

[122] OpenMP Architecture Review Board. OpenMP Application Program Interface. Technical report, July 2013. URL `http://www.openmp.org/mp-documents/spec30.pdf`. (Cited on page 118.)

[123] E. S. Oran and J. P. Boris. *Numerical Simulation of Reactive Flow*. Cambridge University Press, 2nd edition, Nov. 2000. (Cited on page 38.)

[124] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. (Cited on page 113.)

[125] Parallel Programming Laboratory. *The Charm++ Parallel Programming System Manual*. University of Illinois at Urbana-Champaign. URL `http://charm.cs.illinois.edu/manuals/html/charm++/manual.html`. (Cited on page 56.)

[126] S. J. Pennycook and S. A. Jarvis. Developing Performance-Portable Molecular Dynamics Kernels in OpenCL. In *Proceedings of the 26th ACM/IEEE International Conference on Supercomputing*, pages 386–395, Dec. 2012. (Cited on page 66.)

[127] S. J. Pennycook, G. R. Mudalige, S. D. Hammond, and S. A. Jarvis. Parallelising Wavefront Applications on General-Purpose GPU Devices. In *Proceedings of the 26th UK Performance Engineering Workshop*, July 2010. (Cited on page 114.)

[128] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis. Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors. In *Proceedings of the 27th International Symposium on Parallel & Distributed Processing*, pages 1085–1097, May 2013. (Cited on page 66.)

[129] O. Perks, D. A. Beckingsale, A. S. Dawes, J. A. Herdman, C. Mazauric, and S. A. Jarvis. Analysing the influence of InfiniBand choice on OpenMPI memory consumption. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 186–193, 2013. (Cited on page viii.)

[130] O. F. J. Perks. personal communication, Apr. 2014. (Cited on page 161.)

[131] O. F. J. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. WMTrace - A Lightweight Memory Allocation Tracker and Analysis Framework. In *Proceedings of the 27th UK Performance Engineering Workshop*, July 2011. (Cited on page 148.)

[132] O. F. J. Perks, R. F. Bird, D. A. Beckingsale, and S. A. Jarvis. Exploiting Spatiotemporal Locality for Fast Call Stack Traversal. In *2nd Workshop on High-Performance Infrastructure for Scalable Tools*, pages 1–10, June 2012. (Cited on page viii.)

[133] O. F. J. Perks, D. A. Beckingsale, S. D. Hammond, I. Miller, J. A. Herdman, A. Vadgama, A. H. Bhalerao, L. He, and S. A. Jarvis. Towards Automated Memory Model Generation Via Event Tracing. *The Computer Journal*, 56 (2):156–174, Feb. 2013. (Cited on page viii.)

[134] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. URL `http://www.netlib.org/benchmark/hpl`. (Cited on page 31.)

[135] F. Petrini, G. Fossum, J. Fernandez, A. L. Varbanescu, N. Kistler, and M. Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007. (Cited on page 64.)

[136] PGAS. PGAS - Partitioned Global Address Space. URL `http://www.pgas.org/`. (Cited on page 25.)

[137] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995. (Cited on pages 3, 33, and 62.)

[138] D. E. Post. Software Project Management and Quality Engineering Practices for Complex, Coupled Multiphysics, Massively Parallel Computa-

tional Simulations: Lessons Learned From ASCI. *International Journal of High Performance Computing Applications*, 18(4):399–416, Nov. 2004. (Cited on page 3.)

[139] J. Quirk. A parallel adaptive grid algorithm for computational shock hydrodynamics. *Applied Numerical Mathematics*, 20(4):427–453, Apr. 1996. (Cited on pages 47, 79, and 89.)

[140] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A Detailed, Accurate MPI Benchmark. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 52–59, Sept. 1998. (Cited on page 31.)

[141] B. P. Rybakin, L. I. Stamov, and E. V. Egorova. Accelerated Solution of Problems of Combustion Gas Dynamics on GPUs. *Computers & Fluids*, pages 1–18, Nov. 2013. (Cited on page 115.)

[142] M. L. Sætra, A. R. Bordtkorb, and K.-A. Lie. Efficient GPU-Implementation of Adaptive Mesh Refinement for the Shallow-Water Equations. *Journal of Scientific Computing*, July 2014. (Cited on pages 114 and 116.)

[143] H.-Y. Schive, Y.-C. Tsai, and T. Chiueh. GAMER: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186(2):457–484, Feb. 2010. (Cited on pages 81 and 115.)

[144] L. I. Sedov. Propagation of strong shock waves. *Journal of Applied Mathematics and Mechanics*, 10:241–250, 1946. (Cited on pages 44 and 46.)

[145] J. Sitaraman, M. Potsdam, B. Jayaraman, A. Datta, A. Wissink, D. Mavriplis, and H. Saberi. Rotor Loads Prediction Using Helios: A Multi-Solver Framework for Rotorcraft CFD/CSD Analysis. In *Proceedings of the 49th AIAA Aerospace Sciences Meeting*, Jan. 2011. (Cited on page 83.)

[146] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. S. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing Failures in Exascale Computing. Technical Report ANL/MCS-TM-332, Argonne National Laboratory, Mar. 2013. (Cited on page 58.)

[147] G. A. Sod. A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws. *Journal of Computational Physics*, 27(1):1–31, Apr. 1978. (Cited on pages 44, 45, 143, and 173.)

[148] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. (Cited on page 105.)

[149] Stack Overflow. What makes code legacy? URL `http://stackoverflow.com/questions/479596/what-makes-code-legacy`. (Cited on page 3.)

[150] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. *Lecture Notes in Computer Science*, 5335:16–30, Nov. 2008. (Cited on page 171.)

[151] W. G. Sutcliffe. BBC hydrodynamics. Technical Report UCID-17013, Lawrence Livermore Laboratory, Feb. 1974. (Cited on pages 41, 80, and 101.)

[152] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 3, Mar. 2005. (Cited on pages 5 and 22.)

[153] H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, Sept. 2005. (Cited on pages 5 and 22.)

[154] R. Teyssier. Cosmological hydrodynamics with adaptive mesh refinement. *Astronomy & Astrophysics*, 385(1):337–364, Apr. 2002. (Cited on page 81.)

[155] The Khronos Group Inc. OpenCL - The open standard for parallel programming of heterogeneous systems. URL `http://www.khronos.org/opencl/`. (Cited on pages 27, 70, and 118.)

[156] TOP500.org. Home | TOP500 Supercomputer Sites. URL `http://www.top500.org`. (Cited on page 2.)

[157] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna. Scalable Fine-Grained Parallelization of Plane-Wave-Based Ab-Initio Molecular Dynamics for Large Supercomputers. *Journal of Computational Chemistry*, 25(16):2006–2022, Oct. 2004. (Cited on page 16.)

[158] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990. (Cited on page 16.)

[159] B. Van Leer. Towards the ultimate conservative difference scheme III. Upstream-centered finite-difference schemes for ideal compressible flow. *Journal of Computational Physics*, 23(3):263–275, Mar. 1977. (Cited on page 43.)

[160] B. Van Straalen, J. Shalf, T. Ligocki, N. Keen, and W.-S. Yang. Scalability Challenges for Massively Parallel AMR Applications. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009. (Cited on pages 55 and 145.)

[161] J. Vlissides, R. Helm, R. Johnson, and E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (Cited on pages 83, 87, and 119.)

[162] P. Wang, T. Abel, and R. Kaehler. Adaptive mesh fluid simulations on

GPU. *New Astronomy*, 15(7):581–589, Oct. 2010. (Cited on pages 113, 114, and 115.)

[163] H. S. Wijesinghe, R. D. Hornung, A. L. Garcia, and N. G. Hadjiconstantinou. Three-dimensional Hybrid Continuum-Atomistic Simulations For Multiscale Hydrodynamics. *Journal of Fluids Engineering*, 126(5):768–777, Dec. 2004. (Cited on page 121.)

[164] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott. Large Scale Parallel Structured AMR Calculations Using the SAMRAI Framework. In *Proceedings of the 14th ACM/IEEE Conference on Supercomputing*, pages 6–19, Nov. 2001. (Cited on pages 53 and 86.)

[165] P. Woodward and P. Colella. The Numerical Simulation of Two-Dimensional Fluid Flow with Strong Shocks. *Journal of Computational Physics*, 54(1):115–173, Apr. 1984. (Cited on pages 44 and 45.)

[166] A. N. Yzelman, R. H. Bisseling, D. Roose, and K. Meerbergen. MulticoreBSP for C: A High-Performance Library for Shared-Memory Parallel Programming. *International Journal of Parallel Programming*, 42(4):619–642, Aug. 2014. (Cited on page 18.)

[167] U. Ziegler. The NIRVANA code: Parallel computational MHD with adaptive mesh refinement. *Computer Physics Communications*, 179(4):227–244, Aug. 2008. (Cited on page 81.)

APPENDIX A

Experimental Results

This appendix contains extended numerical results for the experiments conducted throughout the course of this dissertation. Each section contains results specific to the corresponding chapter.

A.1 Performance Engineering with Mini-Applications

The results in this section contain timings for the experiments into programming models, scalability and configuration improvements conducted with the CloverLeaf mini-application and discussed in Chapter 4.

Table A.1 contains performance analysis results for the three different GPU-based implementations of CloverLeaf. These results are broken down by kernel, highlighting the difference in timings for the most computationally expensive kernels. Table A.2 contains weak scaling results for CloverLeaf running on HECToR. Finally, Table A.3 details the runtime improvements seen when applying the rank re-ordering operation.

Kernel	Runtime (s) OpenACC	OpenCL	CUDA
advec_cell	0.426	0.382	0.309
advec_mom	0.769	0.773	0.715
calc_dt	0.112	0.189	0.167
pdv	0.241	0.220	0.272
update_halo	0.390	0.320	0.693
viscosity	0.128	0.160	0.155
Other	0.402	0.344	0.277
Total	2.059	2.558	2.780

Table A.1: Runtime for CloverLeaf kernels using three GPU-based programming models.

Nodes	Runtime (s) Titan (OpenACC)	Titan (CUDA)	HECToR (CPU)
1	29.780	15.160	60.296
2	30.070	15.600	60.736
4	30.420	15.890	60.713
8	29.550	16.355	61.101
16	30.870	16.666	60.987
32	29.859	16.781	61.329
64	31.032	16.790	61.077
128	29.320	16.915	61.442
256	30.987	16.838	61.342
512	29.997	16.969	61.780
1,024	31.091	17.0136	62.005
2,048	30.135	17.299	62.826
4,096	32.199	17.596	-
8,192	35.010	17.709	-
16,384	34.767	19.279	-

Table A.2: Weak-scaling runtimes for CloverLeaf on Titan and HECToR.

Nodes	Performance improvement
8	0.37 %
16	0.10 %
32	0.31 %
64	0.35 %
128	0.47 %
256	1.70 %
512	0.96 %
1,024	4.10 %

Table A.3: Performance improvements (% decrease in runtime) for rank-reordering on HECToR.

Resolution	Runtime (s) GPU	CPU
125	14.57	10.62
250	21.05	12.53
500	24.08	11.24
1,000	24.97	13.4
2,000	24.12	16.27
4,000	27.54	23.6
8,000	35.13	37.99
16,000	48.03	72.12
32,000	68.19	129.52
64,000	107.77	242.77
128,000	188.23	474.22
256,000	348.55	940.48

Table A.4: CPU vs. GPU runtime for increasing problem size.

Nodes	Runtime (s) GPU	CPU
1	192.73	10.62
2	122.71	12.53
4	70.64	11.24
8	67.18	13.4

Table A.5: Strong scaling results for the CPU- and GPU-based versions of CleverLeaf.

A.2 Scalable AMR on Graphics Processing Units

This section contains performance timings for the various experiments conducted using the GPU-based version of CleverLeaf as part of Chapter 6.

Table A.4 contains timings comparing the CPU- and GPU-based versions of CleverLeaf as problem size is increased. Table A.5 describes performance results for small-scale runs of CleverLeaf on up to 8 nodes, and Table A.6 contains grind times (in μs/cell) for runs on up to 4,096 GPUs on Titan.

Nodes	Grind Time (μs/cell) Hydrodynamics	Synchronisation	Regridding	Total
1	$6.3 \cdot 10^{-7}$	$1.43 \cdot 10^{-8}$	$6.94 \cdot 10^{-10}$	$7.19 \cdot 10^{-7}$
4	$7.91 \cdot 10^{-7}$	$3.27 \cdot 10^{-8}$	$1.23 \cdot 10^{-9}$	$9.24 \cdot 10^{-7}$
16	$1.14 \cdot 10^{-6}$	$7.67 \cdot 10^{-8}$	$3.43 \cdot 10^{-9}$	$1.34 \cdot 10^{-6}$
64	$1.34 \cdot 10^{-6}$	$1.01 \cdot 10^{-7}$	$7.02 \cdot 10^{-9}$	$1.62 \cdot 10^{-6}$
256	$1.52 \cdot 10^{-6}$	$1.33 \cdot 10^{-7}$	$1.13 \cdot 10^{-8}$	$1.91 \cdot 10^{-6}$
1,024	$1.86 \cdot 10^{-6}$	$1.27 \cdot 10^{-7}$	$8.73 \cdot 10^{-9}$	$2.37 \cdot 10^{-6}$
4,096	$2.69 \cdot 10^{-6}$	$1.25 \cdot 10^{-7}$	$9.06 \cdot 10^{-9}$	$3.67 \cdot 10^{-6}$

Table A.6: Grind times for replication scaled runs on Titan.

A.3 Improving AMR Parameter Selection on Contemporary Compute Platforms

The results in this section contain detailed timings for the various components of the AMR algorithm, obtained during experiments conducted for Chapter 7. Table A.7 contains detailed results of the runtime breakdown for replication scaling. Tables A.8, A.9, and A.10 contain results of the runtime breakdown for varying the regridding frequency, maximum number of levels, and the refinement ratio. The results in Table A.11 show the relative impact of different parameter configurations on runtime and data usage.

	Kernels		Boundaries		Synchronisation		Regridding		Other		Total	
Cores	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform
16	28.053	283.171	21.417	2.960	1.294	-	11.622	-	0.737	0.576	63.123	286.707
32	27.794	282.861	21.790	3.754	1.264	-	12.383	-	1.132	0.576	64.362	287.190
64	26.423	283.048	26.731	4.502	2.441	-	16.019	-	1.395	0.576	73.009	288.127
128	26.468	283.659	26.164	5.436	1.752	-	16.456	-	1.659	0.577	72.498	289.672
256	26.236	284.531	27.409	5.911	3.024	-	21.132	-	1.573	0.574	79.373	291.016
512	26.527	285.680	25.608	6.480	2.250	-	21.460	-	3.072	0.579	78.918	292.739
1,024	25.999	286.752	28.915	5.741	2.860	-	28.487	-	3.287	0.581	89.547	293.074
2,048	26.220	287.431	27.784	7.685	2.790	-	34.889	-	6.450	0.581	98.134	295.696
4,096	25.962	289.805	30.261	7.675	3.004	-	48.467	-	9.335	0.583	117.029	298.063

(a) Runtime breakdown on Cab.

	Kernels		Boundaries		Synchronisation		Regridding		Other		Total	
Cores	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform
16	79.345	612.858	114.105	6.232	9.482	-	171.102	-	1.998	1.133	376.032	620.223
32	77.014	611.857	111.746	6.823	9.441	-	193.309	-	3.116	1.130	394.625	619.810
64	76.614	612.426	129.511	6.933	10.147	-	237.503	-	4.121	1.133	457.895	620.492
128	76.626	612.385	129.418	6.394	9.737	-	252.919	-	5.656	1.157	474.355	619.936
256	76.192	612.955	133.512	6.758	11.268	-	294.047	-	5.117	1.175	520.137	620.888
512	77.149	613.480	126.934	7.067	10.226	-	305.789	-	11.560	1.216	531.658	621.763
1,024	76.163	612.419	138.227	10.665	11.323	-	396.369	-	8.216	1.350	630.298	624.434
2,048	76.877	615.575	128.666	11.018	11.061	-	460.915	-	19.483	1.331	697.002	627.924
4,096	76.374	618.069	140.410	15.245	11.539	-	671.628	-	18.665	1.614	918.616	634.928

(b) Runtime breakdown on Vulcan.

	Kernels		Boundaries		Synchronisation		Regridding		Other		Total	
Cores	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform	AMR	Uniform
24	24.945	197.660	27.551	6.905	1.505	-	15.520	-	0.243	0.422	69.764	204.987
48	25.016	198.398	26.551	6.847	1.393	-	16.168	-	0.401	0.423	69.528	205.668
96	23.884	203.679	32.490	2.865	2.145	-	24.211	-	0.956	0.367	83.687	206.912
192	24.182	203.295	29.825	3.461	1.952	-	23.813	-	0.999	0.376	80.771	207.132
384	23.972	199.039	31.258	8.354	2.549	-	31.765	-	1.305	0.711	90.848	208.104
768	23.885	204.911	30.733	4.165	2.622	-	34.143	-	1.844	0.313	93.226	209.389
1,536	23.853	205.076	31.608	7.321	3.030	-	40.386	-	1.744	0.549	100.621	212.946
3,072	23.771	201.117	31.808	7.850	3.232	-	48.503	-	2.429	0.803	109.743	209.769
6,144	23.888	202.816	32.227	7.842	3.555	-	67.431	-	3.002	0.851	130.103	211.508

(c) Runtime breakdown on ARCHER.

Table A.7: Runtime breakdown on ARCHER.

Frequency	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	31.776	30.886	3.497	106.433	6.558	179.149
4	31.670	32.025	3.595	56.794	7.792	131.875
6	32.031	32.611	3.753	40.025	6.945	115.365
8	32.685	32.356	3.880	27.519	7.505	103.945
10	34.241	32.965	4.229	23.302	7.272	102.010
12	38.861	35.059	4.666	19.635	6.733	104.954
14	43.643	40.008	5.204	18.180	8.483	115.517
16	48.485	40.405	5.689	16.484	6.493	117.556
18	53.164	44.366	6.251	16.567	6.958	127.307
20	57.795	48.184	6.770	15.755	8.965	137.469

(a) 4,096 core run on Cab.

Frequency	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	93.592	143.263	13.925	1461.296	22.516	1734.592
4	93.744	143.262	14.155	732.678	22.585	1006.423
6	94.915	144.945	14.440	494.919	22.695	771.914
8	96.509	146.841	14.423	341.335	22.754	621.863
10	100.290	151.980	15.125	312.832	24.034	604.260
12	111.932	154.971	16.427	265.701	23.874	572.905
14	124.111	159.153	17.905	233.843	23.985	558.997
16	136.168	162.483	18.871	209.087	24.122	550.730
18	147.659	169.469	20.414	194.333	24.537	556.411
20	158.924	175.592	21.343	183.188	25.608	564.656

(b) 4,096 core run on Vulcan.

Frequency	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	28.988	39.410	3.862	176.906	3.758	252.922
4	29.399	35.954	3.933	84.466	3.316	157.069
6	29.736	37.453	4.129	56.938	4.109	132.364
8	30.359	37.154	4.340	43.703	3.429	118.986
10	31.815	39.593	4.626	35.877	3.181	115.092
12	36.146	42.048	5.047	30.411	3.160	116.811
14	40.656	46.455	5.529	28.411	3.517	124.567
16	45.383	50.017	6.011	23.786	3.662	128.858
18	49.894	51.367	6.934	23.597	3.346	135.139
20	54.486	55.094	7.320	21.723	3.649	142.272

(c) 6,144 core run on ARCHER.

Table A.8: Runtime breakdown for varying regridding frequency.

Levels	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	94.690	75.288	2.645	33.445	17.534	223.602
3	42.305	36.783	3.475	49.242	8.185	139.990
4	31.766	30.614	3.517	53.476	7.000	126.373
5	29.847	32.174	3.889	60.165	9.106	135.180
6	29.258	113.917	3.827	81.632	27.699	256.333

(a) 4,096 core run on Cab.

Levels	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	238.693	223.702	6.522	899.780	47.997	1416.694
3	116.528	149.109	11.006	692.314	27.058	996.016
4	93.709	143.400	13.573	731.282	23.719	1005.684
5	88.826	160.685	17.280	910.920	16.202	1193.914
6	88.338	188.825	18.052	1192.732	10.410	1498.357

(b) 4,096 core run on Vulcan.

Levels	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	91.097	96.089	2.963	41.809	13.845	245.803
3	38.715	42.015	3.672	68.558	4.474	157.433
4	29.379	35.835	3.938	83.774	3.322	156.247
5	26.571	40.971	3.857	98.302	3.049	172.749
6	26.424	46.984	4.076	120.417	3.213	201.115

(c) 6,144 core run on ARCHER.

Table A.9: Runtime breakdown for varying maximum level number.

Ratio	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	31.390	41.967	3.944	71.669	8.476	157.445
3	23.302	27.265	1.806	33.891	4.424	90.687
4	23.220	27.116	1.397	40.448	4.474	96.655
5	28.428	30.246	1.304	40.723	4.555	105.256
6	28.173	28.566	1.090	38.088	4.661	100.579

(a) 4,096 core run on Cab.

Ratio	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	188.304	208.618	23.666	763.499	25.448	1209.534
3	130.002	160.502	14.751	446.829	10.622	762.705
4	131.972	165.054	14.650	911.519	13.013	1236.208
5	145.932	160.477	12.907	599.150	11.417	929.883
6	143.731	179.958	15.236	672.710	16.744	1028.379

(b) 4,096 core run on Vulcan.

Ratio	Kernels	Boundaries	Synchronisation	Regridding	Other	Total
2	28.142	51.715	5.587	90.600	5.420	181.464
3	20.269	36.353	2.651	54.603	3.010	116.887
4	20.275	36.558	2.469	83.596	2.935	145.833
5	24.336	40.639	1.981	76.903	12.813	156.672
6	23.509	40.399	1.672	87.458	3.449	156.487

(c) 6,144 core run on ARCHER.

Table A.10: Runtime breakdown for varying refinement ratio.

Frequency	Cab Time	Cab Memory	Archer Time	Archer Memory	Vulcan Time	Vulcan Memory
1	2.1×	0.97×	2.74×	0.97×	2.86×	0.97×
2	1.4×	0.98×	1.52×	0.98×	1.62×	0.98×
3	1.1×	0.99×	1.18×	0.99×	1.21×	0.99×
4	1×	1×	1×	1×	1×	1×
5	0.9×	1.01×	0.91×	1.01×	0.88×	1.01×
6	0.9×	1.02×	0.84×	1.02×	0.8×	1.02×
7	0.8×	1.03×	0.76×	1.03×	0.71×	1.03×
8	0.8×	1.04×	0.75×	1.04×	0.67×	1.04×
9	0.8×	1.06×	0.72×	1.06×	0.64×	1.06×
10	0.8×	1.09×	0.71×	1.09×	0.66×	1.09×
11	0.9×	1.16×	0.72×	1.16×	0.65×	1.16×
12	0.8×	1.23×	0.73×	1.23×	0.65×	1.23×
13	0.8×	1.29×	0.8×	1.29×	0.64×	1.29×
14	0.9×	1.36×	0.69×	1.36×	0.65×	1.36×
15	0.9×	1.42×	0.72×	1.42×	0.65×	1.42×
16	0.9×	1.49×	0.77×	1.49×	0.65×	1.49×
17	0.9×	1.56×	0.76×	1.56×	0.66×	1.56×
18	1×	1.62×	0.79×	1.62×	0.67×	1.62×
19	1×	1.69×	0.83×	1.69×	0.67×	1.69×
20	1×	1.75×	0.85×	1.75×	0.68×	1.75×

(a) Regridding frequency.

Levels	Cab Time	Cab Memory	Archer Time	Archer Memory	Vulcan Time	Vulcan Memory
2	1.74×	2.73×	1.53×	2.73×	1.53×	2.73×
3	1.05×	1.32×	0.92×	1.32×	1.02×	1.32×
4	1×	1×	0.96×	1×	1×	1×
5	1.04×	0.93×	1.04×	0.93×	1.17×	0.93×
6	1.17×	0.93×	1.19×	0.93×	1.44×	0.93×

(b) Maximum level number.

Ratio	Cab Time	Cab Memory	Archer Time	Archer Memory	Vulcan Time	Vulcan Memory
2	1×	1×	0.96×	1×	1.1×	1×
3	0.73×	0.68×	0.64×	0.68×	0.69×	0.68×
4	0.75×	0.7×	0.82×	0.7×	1.11×	0.7×
5	0.81×	0.78×	0.8×	0.78×	0.84×	0.78×
6	0.79×	0.77×	0.85×	0.77×	0.93×	0.77×

(c) Refinement ratio.

Table A.11: Relative impact of parameters on runtime and memory when compared to the default parameter configuraiton.